



UNIVERSIDAD DE JAÉN
Nombre del Centro

Trabajo Fin de Grado

DESARROLLO DE UN SISTEMA DE RECOMENDACIONES DE PUNTOS DE INTERÉS BASADO EN GEOLOCALIZACIÓN

Alumno: Pedro Luis Úbeda Muela

Tutores: Prof. D. Carlos Porcel Gallego,
Prof. D. Luis Gonzaga Pérez Cordón

Dpto: Departamento de Informática

Índice

1. INTRODUCCIÓN	5
1.1 INTRODUCCIÓN GENERAL	5
1.2 MOTIVACIÓN	7
1.3 OBJETIVO DEL PROYECTO	8
1.4 PLANIFICACIÓN	9
1.5 ESTRUCTURA DE MEMORIA	10
2. ESTADO DEL ARTE	11
2.1 APLICACIONES EXISTENTES	11
2.1.1 <i>FourSquare</i>	11
2.1.2 <i>Facebook</i>	12
2.1.3 <i>Google Places</i>	13
2.2 SISTEMAS DE RECOMENDACIÓN	15
2.2.1 <i>Modelos de sistemas de recomendación</i>	15
2.2.2 <i>Tipos de sistemas de recomendación de POIs</i>	19
3. ANÁLISIS DEL PROBLEMA	21
3.1 REQUISITOS	21
3.1.1 <i>Requisitos Funcionales</i>	21
3.1.2 <i>Requisitos No Funcionales</i>	21
3.2 CASOS DE USO	22
3.2.1 <i>Filtrar por categorías</i>	22
3.2.2 <i>Hacer Check-In</i>	22
3.2.3 <i>Marcar lugar como "No interesante"</i>	23
3.2.4 <i>Valorar un lugar</i>	23
3.2.5 <i>Cambiar algoritmo</i>	24
3.2.7 <i>Iniciar sesión</i>	25
3.2.8 <i>Cerrar sesión</i>	25
3.2.9 <i>Ver Check-Ins</i>	26
3.2.9 <i>Diagrama de casos de uso</i>	27
3.3 ESTIMACIÓN DE COSTES	27
3.4 METODOLOGÍA DE DESARROLLO	31
3.5 ESTIMACIÓN DE TIEMPOS	33
4. DISEÑO DE LA SOLUCIÓN	35
4.1 PROTOTIPO	35
4.3 HERRAMIENTAS UTILIZADAS	44
4.3.1 <i>Android Studio</i>	44
4.3.2 <i>Git</i>	44
4.3.3 <i>Firebase</i>	44
5. IMPLEMENTACIÓN DE LA SOLUCIÓN	46
5.1 PRIMERA ITERACIÓN: INTERFAZ E IMPLEMENTACIÓN DE PANTALLA DE LOGIN	46
5.1.1 <i>Análisis de primera iteración</i>	46
5.1.2 <i>Implementación de Actividad de Login</i>	47
5.1.3 <i>Implementación de Login por Email</i>	48
5.1.4 <i>Implementación de Login por Facebook</i>	49
5.1.5 <i>Implementación de Login por Google</i>	50
5.1.6 <i>Diagrama de clases</i>	51
5.1.7 <i>Diagrama de secuencia</i>	52

5.2 SEGUNDA ITERACIÓN: INTERFAZ E IMPLEMENTACIÓN DE PANTALLA PRINCIPAL SIN ORDENAR LUGARES	53
5.2.1 <i>Análisis de segunda iteración</i>	53
5.2.2 <i>Implementación de Lista de lugares</i>	53
5.2.3 <i>Implementación de Mapa</i>	59
5.2.4 <i>Diagrama de clases</i>	63
5.2.5 <i>Diagrama de secuencia</i>	64
5.3 TERCERA ITERACIÓN: INTERFAZ CON DETALLES DE UN LUGAR	64
5.3.1 <i>Análisis de tercera iteración</i>	64
5.3.2 <i>Implementación de Interfaz y Actividad de Vista Detallada</i>	64
5.3.3 <i>Implementación de Descarga de Imagen de un POI</i>	66
5.3.4 <i>Diagrama de clases</i>	68
5.3.5 <i>Diagrama de secuencia</i>	69
5.4 CUARTA ITERACIÓN: IMPLEMENTACIÓN BASE DE DATOS FIREBASE	70
5.4.1 <i>Análisis de cuarta iteración</i>	70
5.4.2 <i>Diseño de la Base de Datos</i>	70
5.4.3 <i>Conexión de Base de Datos y Appoi</i>	72
5.4.4 <i>Diagrama de clases</i>	75
5.5 QUINTA ITERACIÓN: IMPLEMENTACIÓN DE BARRA LATERAL: CIERRE DE SESIÓN, LISTA DE IGNORADOS, LISTA DE CHECK-INS Y AJUSTES	75
5.5.1 <i>Análisis de quinta iteración</i>	75
5.5.2 <i>Interfaz de barra lateral y cierre de sesión</i>	76
5.5.3 <i>Implementación de Vista de Ignorados y de Check-Ins totales</i>	78
5.5.4 <i>Implementación de Vista de Ajustes</i>	82
5.5.5 <i>Diagrama de clases</i>	84
5.5.6 <i>Diagrama de secuencia</i>	85
5.6 SEXTA ITERACIÓN: FILTRO DE CATEGORÍAS Y BARRA DE BÚSQUEDA	85
5.6.1 <i>Análisis de sexta iteración</i>	85
5.6.2 <i>Implementación de Filtro de Categorías</i>	86
5.6.3 <i>Implementación de Filtro de Búsqueda</i>	87
5.6.4 <i>Diagrama de clases</i>	89
5.6.5 <i>Diagrama de secuencia</i>	90
5.7 SÉPTIMA ITERACIÓN: SISTEMA DE PERFIL Y AMIGOS	91
5.7.1 <i>Análisis de séptima iteración</i>	91
5.7.2 <i>Implementación de Registro de perfil</i>	91
5.7.3 <i>Implementación de Vista de perfil</i>	94
5.7.4 <i>Implementación de Vista de lista de amigos</i>	96
5.7.5 <i>Diagrama de clases</i>	100
5.7.6 <i>Diagrama de secuencia</i>	101
5.8 OCTAVA ITERACIÓN: IMPLEMENTACIÓN DE ALGORITMOS DE RECOMENDACIÓN	101
5.8.1 <i>Análisis de octava iteración</i>	101
5.8.2 <i>Implementación de Algoritmo de Filtrado Colaborativo basado en usuario</i>	102
5.8.3 <i>Implementación de Algoritmo de Filtrado Colaborativo basado en lugar</i>	105
5.8.4 <i>Implementación de Algoritmo de Filtrado Colaborativo híbrido</i>	107
5.8.5 <i>Implementación de Algoritmo de Filtrado basado en contenido: TF-IDF</i>	108
5.8.6 <i>Diagrama de clases</i>	110
5.8.7 <i>Diagrama de secuencia</i>	111
6. PRUEBAS Y VALIDACIÓN	112
7. CONCLUSIONES	115
ANEXO 1: GUÍA DE INSTALACIÓN	117
ANEXO 2: MANUAL DE USUARIO	118

A2.1 INICIO DE SESIÓN Y REGISTRO	118
A2.2 HACER CHECK-IN, PUNTUAR E IGNORAR	119
A2.3 VER Y CAMBIAR DATOS DE PERFIL	122
A2.4 VER LISTA DE AMIGOS Y BUSCAR USUARIOS	123
A2.5 AJUSTES Y CIERRE DE SESIÓN	124

1. Introducción

1.1 Introducción general

Con el desarrollo de las urbes crecen también la cantidad de puntos de interés (*POIs* del inglés *Point of interest*) tales como cines, tiendas, restaurantes... Estos lugares sirven para enriquecer la vida de las personas brindándoles nuevas experiencias culturales y de ocio. Por ello, se vuelve necesario la existencia de herramientas que permitan ayudar a descubrir nuevos POIs que puedan interesar a sus usuarios ahorrándoles tiempo y esfuerzo en la toma de decisión.

Un **sistema de recomendación** se define como una estrategia de toma de decisión para usuarios que están bajo ambientes de información compleja [1]. También ha sido definido, dentro del ámbito del comercio electrónico, como una herramienta que ayuda al usuario a navegar a través de registros de conocimientos relacionados con sus preferencias [2]. Un sistema de recomendación pone solución al problema de **sobrecarga de información** con la que un usuario se puede encontrar a través de la obtención de recomendaciones personalizadas y exclusivas adaptándose a sus preferencias [3]. Por lo tanto, este tipo de herramienta resulta interesante para facilitar la toma de decisión de visitar un POI a las personas.

En la actualidad pueden encontrarse Sistemas de recomendación de POI basados en **localización** que sugieren lugares cercanos posiblemente interesantes, aunque estos no necesariamente pueden basarse en las preferencias personales del usuario, si no en los sitios más famosos y mejor valorados por otros usuarios. A la hora de diseñar un Sistema de recomendación de POIs es importante la recolección de información que pueda ayudar a predecir los mejores lugares para cada usuario. Esta información puede ser un registro de lugares que el usuario ya ha visitado, respuestas a una encuesta realizada sobre sus preferencias, palabras clave usadas en sus redes sociales, etc. Una recomendación errónea puede provocar una experiencia negativa impactando gravemente en la confianza del usuario en estos sistemas.

Un **dataset** consiste en una colección de datos correspondientes a una tabla de una única base de datos o matriz de datos estadística, incluyendo las columnas que representan una variable concreta y cada fila siendo un miembro del conjunto [4]. La existencia de *datasets* de puntos de interés en ciudades junto con el uso

generalizado de dispositivos móviles inteligentes que incorporan tecnología GPS hacen viable el estudio y desarrollo de sistemas de recomendación de POIs. Obteniendo datos geográficos del usuario (altitud y latitud) y comparándolos con los de estas bases de datos podemos saber que POIs visita el usuario y cuáles están cerca de él. Algunos sistemas además permiten al usuario hacer un “Check-In” una vez en el lugar reflejándolo en redes sociales, permitiéndole compartir la experiencia con sus amigos y dándole la oportunidad de conocer nuevos.

A diferencia de otros sistemas de recomendación como, por ejemplo, de películas en servicios de streaming o productos en tiendas online, los de recomendación de POIs presentan como reto la movilidad del usuario. Dadas las correlaciones entre la información de un POI y la información de un sistema basado en localización podemos concluir tres propiedades de la movilidad humana: correlaciones geo-sociales, patrones geo-temporales e indicaciones de geo-contenido [5]; ninguna de estas se encuentra presente en otros sistemas de recomendación. Así, teniendo en cuenta estas características únicas se propondrá un modelo de sistema de recomendación de POIs que cumpla con varias expectativas:

- **Efectividad:** el sistema debe proporcionar predicciones de calidad, asignando una puntuación numérica a cada POI cercano calculada mediante algoritmos de filtrado contrastados.
- **Localización:** las recomendaciones deben basarse en los puntos de interés cercanos a la posición actual del usuario, pues el uso más común de estos sistemas de recomendación es obtener ideas de lugares desconocidos que visitar en ciudades que se están visitando por primera vez. Para ello habrá que usar la función GPS que los dispositivos móviles actuales suelen tener disponibles a través del consentimiento del usuario [11].
- **Preferencias:** la aplicación debe almacenar un historial de lugares visitados por el usuario y las valoraciones de los mismos, para así tener una idea de las preferencias del mismo y mejorar las recomendaciones. Para muchos algoritmos es fundamental poseer de un feedback previo del usuario a la hora de calcular la predicción.
- **Personalización:** el usuario debe poder customizar su experiencia pudiendo elegir en todo el momento el algoritmo de filtrado, filtrar por categorías o

nombres la lista de POIs mostrada, ignorar lugares no interesantes, elegir si quiere que la app muestre notificaciones, etc.

- **Robustez:** la aplicación debe contemplar todas las interacciones posibles con el usuario y estar preparada ante ellas, de forma que no ocurran errores en tiempo de ejecución o funcione de forma inesperada.
- **Social:** la creación de perfiles de usuario y la posibilidad de añadir a una lista de amigos a otros debe formar parte de este sistema. No solo es interesante para el usuario poder seguir a sus amigos en la app y ver sus lugares visitados y puntuaciones, sino que sirve a la vez para mejorar la predicción de los algoritmos de Filtrado Colaborativo, pues según el *principio de homofilia* las personas con relación de amistad suelen tener preferencias similares [6].

1.2 Motivación

La geolocalización es un aspecto que cada vez cobra más y más importancia. No solo una gran cantidad de aplicaciones web y móviles de todo tipo se apoyan en ella para ofrecer mejores servicios personalizados al usuario, sino que está empezando a ser relevante en áreas más específicas.

Por ejemplo, con el gran desarrollo urbanístico (según Naciones Unidas, se estima que el 70% en 2050 de la población mundial vivirá en ciudades [7]) también se expande el fenómeno de las Ciudades Inteligentes o Smart Cities, donde la geolocalización juega un papel fundamental para conectar el entorno inmediato con el ciudadano.

También hay que tener en cuenta el binomio geolocalización y Big Data. Aplicaciones de transporte como Uber o Cabify se apoyan en la localización del usuario para detectar los vehículos cercanos a él; y también en la recolección de datos para recomendar el mejor tipo de coche según las preferencias mostradas anteriormente. Además, permite calcular la mejor ruta para el conductor en cuanto a tarifas y número de pasajeros, predecir para el usuario a la hora que llegará el vehículo... También se está usando esta combinación en la ambiciosa investigación de los vehículos autónomos: se recogen datos del estado del tráfico y carreteras en tiempo real y, al ser geolocalizados los vehículos, pueden reaccionar y evitar obstáculos durante el trayecto.

Debido al incesante crecimiento del desarrollo de aplicaciones para Smartphones [8], dispositivos que un alto porcentaje de la población lleva encima y que disponen de tecnología GPS, y su aumento anual de descargas, como vemos en la figura 1, es de especial interés orientar este proyecto a una aplicación para el Sistema Operativo más usado en dispositivos móviles: Android [9].

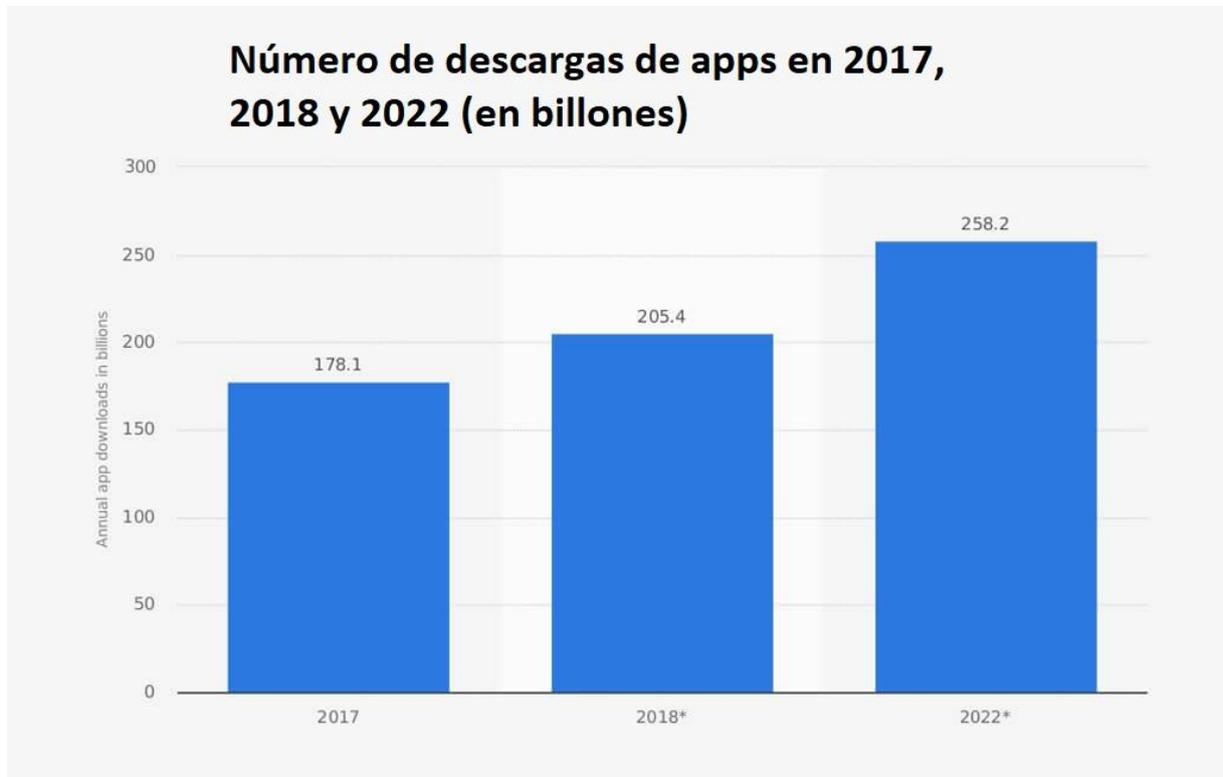


Figura 1: Descargas de apps en diversas tiendas de Smartphones en el periodo 2017-2022 (predicción) según App Annie.

Por lo tanto, he decidido realizar este Trabajo de Fin de Grado para aprender y familiarizarme con las distintas herramientas y librerías de geolocalización y algoritmos de recomendación, adquiriendo a la vez conocimientos sobre desarrollo de aplicaciones en Android con el que ha sido nombrado por Google lenguaje oficial de dicha plataforma: Kotlin [10].

1.3 Objetivo del proyecto

La principal finalidad de este trabajo es el desarrollo de una aplicación para dispositivos móviles Android que ofrezca al usuario un sistema de recomendación de puntos de interés basado en geo-localización. Este objetivo general se compone de varios objetivos más concretos (además de los de efectividad, robustez y personalización descritos en la sección 1.1):

- **Analizar las propiedades y características de los modelos basados en geolocalización:** se debe investigar en la literatura existente sobre los distintos sistemas de recomendación existentes que pueden ser aplicados a puntos de interés, así como sus ventajas e inconvenientes.
- **Implementar un esquema de recomendación basado en localización que sugiera POIs relevantes para los usuarios:** una vez realizado el análisis de los modelos de recomendación, se diseñará uno o varios sistemas a partir de los conocimientos adquiridos que predecirá los mejores puntos de interés cercanos para el usuario.
- **Desarrollar e implementar la aplicación para la recomendación de POIs basándose en la localización:** para aplicar el esquema diseñado se desarrollará una aplicación para dispositivos móviles que lo implementará y aprovechará la localización GPS en la obtención de POIs.
- **Implantar el sistema y probarlo, validándolo con algún conjunto de datos que haya disponible:** tras desarrollar una aplicación funcional, debe ser puesta a prueba en varios escenarios, analizando y valorando los resultados finales obtenidos.

1.4 Planificación

La metodología a realizar para completar el proyecto será la siguiente (en orden):

1. Investigación de la literatura sobre sistemas de recomendación de puntos de interés ya existentes.
2. Recopilar información sobre los sistemas y librerías que permiten obtener localizaciones cercanas: cómo implementarlos, costos y cuotas, etc.
3. Aprendizaje del lenguaje Kotlin y sus diferencias con Java para la codificación en Android Studio.
4. Implementación de una Base de Datos: decidir el tipo (relacional o NoSQL), configuración...
5. Gestión de usuarios de la aplicación permitiendo login en Facebook y Google.
6. Prototipado y planificación de la aplicación: diagramas de casos de uso, especificación de requisitos, estimación de tiempos...
7. Desarrollo de la aplicación en Android Studio.
8. Pruebas y validación final.

1.5 Estructura de memoria

La conformación de esta memoria consistirá en 6 apartados principales:

1. Introducción: se describe de forma general los conceptos más importantes del proyecto, su motivación, metodología...
2. Estado del arte: estudio de aplicaciones similares y documentos relacionados con la temática que servirán de apoyo.
3. Análisis del problema: determinar los requisitos necesarios de la aplicación y sus soluciones correspondientes.
4. Diseño de la solución: planificación de las soluciones finales, tales como herramientas que se van a usar, prototipo en papel, arquitectura del software, diagrama de Gantt...
5. Implementación de la solución: detalles del desarrollo del proyecto como porciones de código interesantes, explicación de las distintas clases...
6. Pruebas y validación: resultados de testear el producto final, determinando la validez de la solución y las conclusiones alcanzadas.
7. Conclusiones: reflexiones sobre la experiencia de realizar este trabajo, así como ideas de posibles mejoras futuras al mismo.

2. Estado del arte

En esta sección se realizará una investigación sobre aplicaciones existentes con funcionalidad similar a la del proyecto, además de la literatura sobre métodos y algoritmos utilizados en este tipo de sistemas de recomendación. El objetivo de ello es detectar carencias o virtudes a tener en cuenta a la hora de diseñar e implementar la solución.

2.1 Aplicaciones existentes

Existen varias aplicaciones web y móviles que permiten la localización de puntos de interés cercanos al usuario, además de redes sociales que permiten hacer “Check-In” en lugares, es decir, confirmaciones de haber estado en un lugar y mostrarlo a los amigos de dicha red. Estudiar acerca de cómo extraer esa información de estas RSBL es importante pues algunos de los algoritmos de filtrado usados en los sistemas de recomendación tienen en cuenta los lugares que visitan y valoran los “amigos” en redes sociales del usuario, así como datos como el grado de amistad entre ellos y la distancia.

2.1.1 FourSquare

Red social creada en 2009 y que se distingue de la mayoría al centrarse en la recomendación de puntos de interés cercanos. Su principal característica es permitir al usuario marcar los lugares (“Check-In”) que está visitando en tiempo real. Para recomendar, se basa en el historial de estas marcas, *likes* y marcas de otros usuarios [12]. La idea es que el usuario visite lugares interesantes a la vez que conoce a otras personas que podrían convertirse en nuevos amigos. Desde su lanzamiento, FourSquare cuenta con una API para que los desarrolladores puedan hacer uso de sus datos almacenados.

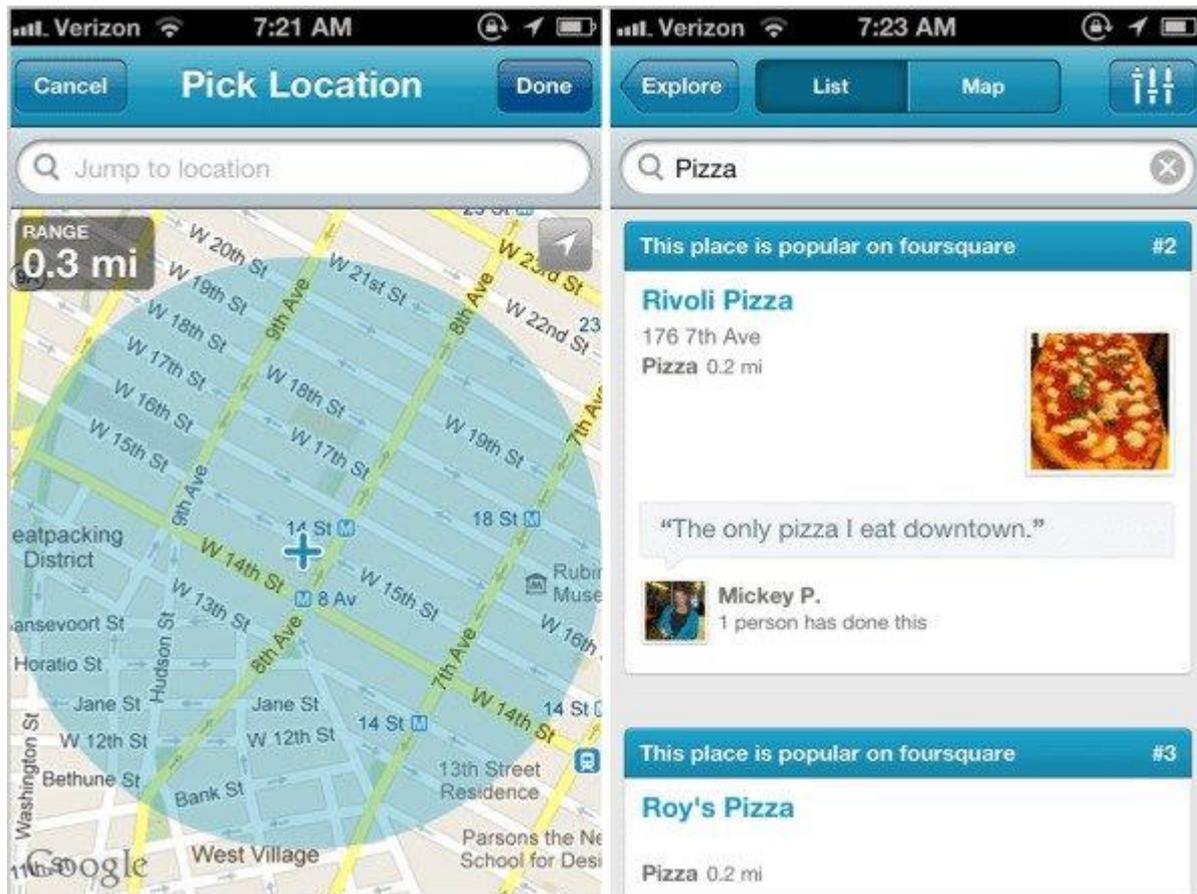


Figura 2: Interfaz de la aplicación FourSquare.

Como se puede apreciar en la interfaz de su app móvil en la figura 2, cuenta con un buscador donde el usuario puede filtrar por palabras clave los lugares dentro de un determinado radio respecto a su posición y presentar los resultados tanto con marcadores en un mapa como en una lista con más detalles del lugar. Esta dualidad de mostrar los puntos de interés será tomada en cuenta en el diseño de interfaz de este proyecto.

2.1.2 Facebook

Fundada en 2004, la que es actualmente la red social con más usuarios activos del mundo [13] cuenta con la opción de hacer “Check-In”, tal y como muestra la figura 3, en cualquier lugar de interés, sin necesidad de estar físicamente en él. Esta marca queda como una publicación que otros usuarios pueden darle “like”, comentarla, ir a la página del lugar donde ver más información del mismo...



Figura 3: Ejemplo de “Check-In” en Facebook.

La API de Facebook permite, una vez hecho “login” en la aplicación, consultar esta información de lugares tanto del usuario como de sus amigos, pero siempre que estos también hayan usado la misma aplicación. Puesto que Facebook permite con facilidad implementar un botón de inicio de sesión en aplicaciones Android, resulta interesante hacerlo en este proyecto para obtener el “feedback” necesario en los algoritmos sociales de filtrado.

2.1.3 Google Places

La aplicación Google Maps que viene instalada por defecto en los dispositivos Android es una de los métodos más famosos de recomendación de puntos de interés. Mediante el módulo Local Guide, una vez el usuario da su consentimiento, este sistema registra su actividad geográfica y reconoce cuando ha permanecido en un lugar de interés, pudiendo solicitarle mediante una notificación su opinión sobre dicho lugar en una reseña pública [14]. Conforme registra un historial de lugares, la aplicación procede a recomendarle nuevos lugares interesantes que pueden no haber sido visitados.

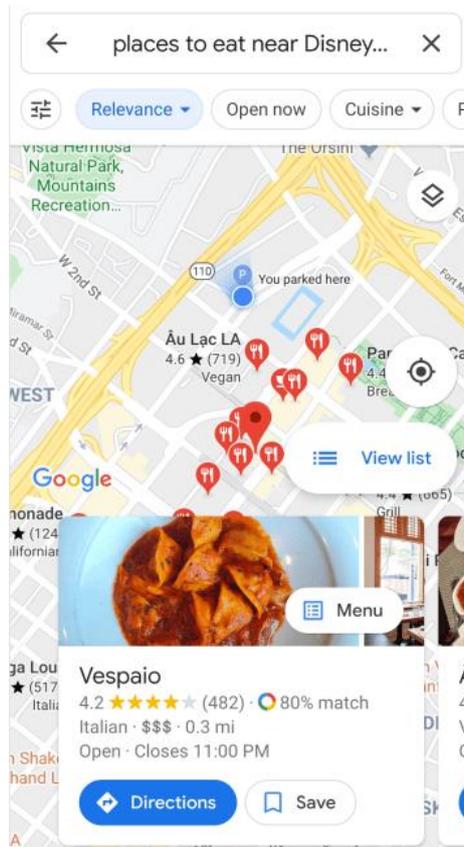


Figura 4: Google Maps mostrando información de un punto de interés registrado en Places.

Dentro de la colección de APIs de Google, Places es la más interesante para el sistema que se quiere construir. Consiste en una gran base de datos con lugares de interés de todo tipo cuyos propietarios inscriben y aportan los datos, permitiéndoles dar a conocer sus negocios con mayor visibilidad al estar esta API integrada en Google Maps. En la figura 4 se observa como Places muestra la información sobre un POI.

De todas las funciones disponibles, Nearby Search es la más útil al permitir obtener una lista de los lugares de interés cercanos al usuario en un radio determinado [15]. Hay que tener en cuenta que para implementar esta funcionalidad en la aplicación será necesario obtener una cuenta de desarrollador con información de pago, al igual que otras APIs de Google como Maps que pueden resultar necesarias para el desarrollo.

2.2 Sistemas de recomendación

En este apartado primero se realiza un repaso de los principales modelos y estrategias de sistemas de recomendación generales y después de cómo se orientan los mismos a los puntos de interés.

Un sistema de recomendación es una herramienta que sugiere ítems de interés a un usuario basándose en sus preferencias explícitas e implícitas, las preferencias de otros usuarios y/o los atributos del ítem y usuario [16].

2.2.1 Modelos de sistemas de recomendación

A continuación, se resumirá los principales modelos o algoritmos de selección en los que se basan los sistemas de recomendación y que serán adaptados en el proyecto. Actualmente se pueden encontrar tres tipos de modelos de sistemas de recomendación: Filtrado Colaborativo (FC), Filtrado Basado en Contenido e Híbridos.

2.2.1.1 Filtrado Colaborativo

El FC está considerado uno de los enfoques más efectivos y es usado por numerosos sistemas como el de Amazon, que fue pionero. Su fundamento principal es la similitud de gustos entre distintos usuarios: si muchos de los productos que dos usuarios han consumido coinciden, es altamente probable que los productos que uno de ellos ha consumido y el otro no resulten interesantes para este último. Este modelo necesita pues apoyarse en una matriz de usuario-objeto, pudiendo ser el objeto en el caso de un Sistema de Recomendación de POIs las visitas de lugares del usuario dado.

Generalmente los modelos FC se clasifican en dos: basado en memoria o vecino más cercano (Nearest Neighbour) y basados en modelo.

2.2.1.1.1 Basado en memoria

Los sistemas de filtrado colaborativo basados en memoria se caracterizan por utilizar la base de datos de usuarios e ítems al completo para calcular predicciones de afinidad de un usuario a un objeto concreto [17]. Se suele utilizar una matriz de usuario y elementos, cuyo valor de celda es la puntuación del usuario al elemento, para representar este tipo de problemas.

El FC basado en memoria se divide en **basado en usuario** y **basado en ítem** según que tome como referencia a la hora de calcular la similitud. En ambos casos, el procedimiento general de este tipo de algoritmos es el siguiente [18]:

- 1.- Se calcula la similitud entre distintos usuarios o ítems.
- 2.- Se selecciona un vecindario mediante dicha similitud.
- 3.- Se calcula la predicción final basándose en las puntuaciones del vecindario seleccionado.

Existen varios métodos para calcular la similitud entre dos usuarios o ítems, siendo los más utilizados el **coeficiente de correlación de Pearson** y la **similitud coseno** [19].

Aunque este sistema cuenta con ventajas como la no necesidad de realizar un análisis de contenido o la independencia del dominio, también se debe tener en cuenta y prevenir en lo posible una serie de problemas asociados [20]:

- **Escalabilidad:** al confiar el modelo en la matriz usuario-objeto al completo se requiera gran cantidad de espacio de almacenamiento, pudiendo llegar a ser computacionalmente ineficiente conforme aumenta la base de datos con el tiempo.
- **Dispersión:** si los usuarios difieren notablemente en las puntuaciones de ítems se dificulta la búsqueda de un vecindario adecuado.
- **Escasez de datos:** al basarse en información de los usuarios, si esta no es suficientemente abundante es complicado emplear el sistema. También hay que considerar el caso del usuario objetivo que acaba de registrarse y por lo tanto no se posee datos de preferencias del mismo (problema de **Arranque en frío**).

2.2.1.1.2 Basado en modelo

Los métodos de filtrado colaborativo basados en modelo adaptan modelos estadísticos que utilizan valoraciones conocidas y realizan recomendaciones sobre predicciones de dichos modelos [21]. Existen gran variedad de algoritmos de FC basado en modelo: redes bayesianas, procesos de decisión de Markov, modelos de clustering y modelos de semántica latente, entre otros [22].

Uno de los más utilizados son los modelos de factores latentes, como los basados en descomposición en valores singulares, y consisten en transformar a los usuarios y objetos en un mismo factor de espacio latente, convirtiéndolos en comparables. Los factores latentes intentan explicar las valoraciones de los usuarios por las características propias de usuarios y elementos. Por ejemplo, en el caso de películas los factores podrían hacer referencia a características obvias como los géneros de comedia y drama; pero también podrían medir las preferencias de los usuarios a través de factores menos evidentes, como la profundidad de desarrollo de personajes [23]. En el caso de recomendación de POIs, en un restaurante estos factores podrían ser la calidad, el sabor, precio, ambiente, vistas, ... Estos factores latentes pueden tener importancia y dominar la ocurrencia de los "Check-In", haciendo que cada uno se produzca como la combinación de los intereses y las propiedades del lugar con estos factores. Por ejemplo, a un usuario que le gusta la comida china y comer al aire libre estará interesado en restaurantes especializados en comida china y con terraza.

Entre las ventajas del enfoque basado en modelo se encuentra una mayor protección frente a ataques de inyección de perfiles [24], consistentes en añadir perfiles falsos en el sistema de recomendación con el objetivo de alterar las predicciones hacia productos concretos. También suelen ser más rápidos en la predicción que los basados en memoria. Como desventaja, existe una inflexibilidad a la hora de añadir más datos a los modelos, pues normalmente construirlos es una tarea que consume mucho tiempo y recursos [25].

2.2.1.2 Filtrado basado en contenido

Los filtrados basados en contenido realizan la selección de elementos a recomendar basándose en la correlación entre las preferencias del usuario objetivo y el contenido de los elementos, en oposición al filtrado colaborativo que se apoya en la correlación entre usuarios con gustos similares [26]. Este modelo se recomienda especialmente cuando se tiene abundante información de los productos y no tanta de los usuarios.

Un ejemplo de esta variante de filtrado es el **TF-IDF**, que consiste en una métrica diseñada para encontrar los términos más relevantes dentro de una colección de documentos [27]. Podría usarse en la recomendación de POIs, por ejemplo, para calcular los términos más importantes que aparecen en el nombre y subcategorías

de los lugares preferidos de un usuario, y compararlos con el de los lugares cercanos para predecir la similitud.

2.2.1.3 Filtrado basado en demografía

Los sistemas de recomendación demográficos se basan en las puntuaciones de los usuarios y sus atributos demográficos tales como la edad, el género o la raza para determinar un grado de similitud entre ellos y predecir una recomendación [28].

Por ejemplo, una cadena de comida rápida que es frecuentada y puntuada positivamente por personas jóvenes tiene más probabilidad de ser recomendada a un usuario en ese rango de edad.

2.2.1.4 Filtrado basado en utilidad

Estos sistemas crean funciones que predicen cuanta “utilidad” tiene un ítem para el usuario a través de varios atributos. Una vez determinado que resulta más útil para el usuario se basa la recomendación de otros ítems en el cálculo de su utilidad bajo los mismos parámetros. La dificultad de estos sistemas radica en tener que ingeniar estas funciones de forma específica según la industria a la que se vaya aplicar, pudiendo ser complejo.

2.2.1.5 Modelo híbrido

Finalmente, el modelo híbrido utiliza la información de la matriz usuario-objeto en conjunto con otra información, y combina cualquiera de los anteriores modelos descritos (u otros que no han tenido cabida) para obtener dos resultados. A los resultados de cada sistema de recomendación se les puede aplicar diversas técnicas, siendo la más usada la ponderación: se combinan y ponderan en un solo resultado final [29], y a continuación se elige el mejor de los resultados o se presentan ambos resultados como recomendación.

Existen otros esquemas de hibridación además de la ponderación, como la combinación de características que mezcla los mejor del FC y el basado en contenido en un algoritmo, considerando la información que proporciona el FC como datos extra a utilizar en el filtrado basado en contenido; o el esquema en cascada, que emplea una técnica en el conjunto inicial y, sobre el resultado, aplica una segunda técnica para refinarlo [30].

2.2.2 Tipos de sistemas de recomendación de POIs

Dentro de los sistemas de recomendación de POIs se encuentran dos métodos: el basado en GPS y el basado en Redes Sociales Basadas en Localización (RSBL).

2.2.2.1 Basados en GPS

La mayoría de personas llevan consigo teléfonos móviles constantemente, los cuales suelen llevar un sensor GPS que puede ser aprovechado para registrar la movilidad de los usuarios. Normalmente estos teléfonos recogen registros temporales de los sitios en los que ha estado el usuario con cierta frecuencia, creando la llamada trayectoria GPS. La trayectoria GPS son una secuencia de puntos geográficos (longitud y latitud) tomados en cortos periodos de tiempo, normalmente pocos segundos. Dentro de esta secuencia se encuentran los puntos de estancia: lugares en los que el usuario ha permanecido sin moverse suficiente cantidad de tiempo para ser considerados POIs. Entre las desventajas de este sistema se encuentran:

- Privacidad: muchos usuarios son reacios a permitir que su teléfono móvil recopile información geográfica, pues lo consideran una intromisión en su intimidad.
- Escasez de muestras: frecuentemente no se permite hacer público la información GPS de los usuarios de teléfonos móviles, lo que sumado al anterior punto resulta en poca cantidad de datos para el sistema de recomendación.
- Ausencia de información semántica: los datos GPS se reducen a coordenadas geográficas de latitud y longitud que no pueden asociarse directamente a lugares de interés como restaurantes, hoteles, tiendas... Esto se suele solucionar con librerías de terceros que relacionan las coordenadas con POIs, pero es difícil cuando hay mucha densidad de POIs en una misma área.
- Escasez de información social: no es fácil obtener conexiones sociales a partir de datos de GPS. Suele depender de que el usuario acepte dar permisos o use ciertas conexiones de comunicación para captar otros usuarios amigos [31].

2.2.2.2 Basados en redes sociales

Gracias al desarrollo de las RSBL las personas pueden hacer “Check-In” del lugar en el que se encuentran y compartirlos con sus contactos sociales, pudiéndose obtener información social, espacial, temporal y de contenido. Debido a las fuertes correlaciones entre la distancia geográfica y los vínculos sociales que se han descubierto en distintos estudios, los sistemas de recomendación de POIs en RSBL potencian las propiedades geográficas y sociales para hacer más eficaz la recomendación. Según La primera ley de la geografía de Waldo Tobler [32], todas las cosas están relacionadas entre sí, aunque las cosas más cercanas lo están más que las lejanas. En el ámbito de los POIs en RSBL, esto significa que los usuarios preferirán visitar lugares cercanos a ellos antes que los lejanos; además de poder preferir de esos lugares lo que otros usuarios prefieren. Las técnicas de recomendación de POIs se suelen centrar en calcular la influencia tanto de la geografía como del apartado social, y combinarlas en un modelo fusionado. La influencia social se suele modelar a través de FC basado en amigos con enfoques tanto basados en memoria como basados en modelo.

3. Análisis del problema

Una vez estudiado el estado del arte, en esta sección se realizará un análisis de la propuesta de este trabajo y plantearán soluciones desde la perspectiva de la ingeniería del software: análisis de requisitos, casos de uso, etc.

3.1 Requisitos

A la hora de especificar el funcionamiento de una aplicación es importante determinar los requisitos funcionales y no funcionales. Los requisitos funcionales consisten en aquellas funciones en las que el propio usuario realiza la acción, como por ejemplo una barra de búsqueda en la que se tiene que escribir y pulsar el botón de “Buscar”. Por otra parte, los requisitos no funcionales se refieren a restricciones o cualidades del propio producto, por ejemplo, unos mínimos de rendimiento, usabilidad, medidas de seguridad...

3.1.1 Requisitos Funcionales

- Registro e inicio de sesión a través de varios métodos: email, Facebook y Google.
- Personalización de recomendación para cada usuario mediante algoritmos de filtrado.
- Personalización de perfil.
- Buscar a otros usuarios y poder agregar a la lista de amigos.
- Filtrar los puntos de interés recomendados por categorías y nombre.
- Definir los intereses sobre tipos de lugares interesantes.
- Hacer “Check-In” en un punto de interés cercano, ignorarlo y valorarlo, además de poder deshacer cada acción.
- Ver los detalles de un punto de interés determinado.
- Abrir un navegador GPS hacia el punto de interés seleccionado.

3.1.2 Requisitos No Funcionales

- Interfaz usable e intuitiva.
- Seguridad en el manejo de información como datos de sesión o geolocalización.

- Eficiencia en las funciones más costosas como la búsqueda de lugares cercanos, con multitarea.
- Prevención ante la posibilidad de un aumento significativo de los usuarios, adaptando los planes de pago necesarios (escalabilidad).

3.2 Casos de uso

Los casos de uso se definen como la descripción de las interacciones que un actor realiza con un sistema. A continuación, se mostrarán algunos ejemplos de las acciones más relevantes, junto con un diagrama final.

3.2.1 Filtrar por categorías

Título:	Filtrar por categorías.
Descripción:	Mostrar en la lista de puntos de interés solo los pertenecientes a las categorías seleccionadas.
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado el botón de “Refrescar”.
Escenario exitoso:	<ol style="list-style-type: none"> 1.- U: Pulsa el botón de categorías. 2.- A: Muestra la lista de categorías 3.- U: Marcar y/o desmarcar las casillas con las categorías según se quiera filtrar. 4.- U: Pulsar el botón “Refrescar”. 5.- A: Actualiza la lista con los nuevos resultados filtrados.
Alternativa:	El usuario pulsa el botón “Cancelar”, dejando la lista en su estado inicial.

3.2.2 Hacer Check-In

Título:	Hacer Check-In.
Descripción:	Registrar una visita en un punto de interés cercano.
Actores:	Usuario (U) y aplicación (A).

Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado el botón de “Check-In”.
Escenario exitoso:	<p>1.- U: Pulsa dos veces en un mismo lugar en la lista superior o pulsa una marca de lugar en el mapa inferior.</p> <p>2.- A: Muestra la pantalla con detalles del lugar.</p> <p>3.- U: Pulsa el botón de “Check-In”.</p> <p>4.- A: Registra el lugar en la base de datos remota para el usuario dado.</p>
Alternativa:	El usuario pulsa el botón “Volver”, regresando a la pantalla principal.

3.2.3 Marcar lugar como “No interesante”

Título:	Marcar lugar como “No interesante”.
Descripción:	Elimina un lugar determinado de la lista de lugares cercanos.
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado el botón de “No interesante”.
Escenario exitoso:	<p>1.- U: Pulsa dos veces en un mismo lugar en la lista superior o pulsa una marca de lugar en el mapa inferior.</p> <p>2.- A: Muestra la pantalla con detalles del lugar.</p> <p>3.- U: Pulsa el botón de “No interesante”.</p> <p>4.- A: Elimina el lugar del mapa y de la lista.</p> <p>5.- A: Registra en la base de datos remota el lugar no deseado.</p>
Alternativa:	El usuario pulsa el botón “Volver”, regresando a la pantalla principal.

3.2.4 Valorar un lugar

Título:	Valorar un lugar.
Descripción:	Registrar una valoración (de 0 a 5 estrellas) en un punto de interés

	cercano.
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado en las estrellas de valoración.
Escenario exitoso:	<p>1.- U: Pulsa dos veces en un mismo lugar en la lista superior o pulsa una marca de lugar en el mapa inferior.</p> <p>2.- A: Muestra la pantalla con detalles del lugar.</p> <p>3.- U: Pulsa el botón de “Check-In”.</p> <p>4.- A: Registra el lugar en la base de datos remota para el usuario dado.</p> <p>5.- A: Muestra las 5 estrellas de valoración bajo el botón de “Check-In”.</p> <p>6.- U: Pulsa las estrellas según la valoración deseada.</p> <p>7.- U: Registrar la valoración para el lugar dado en la base de datos remota.</p>
Alternativa:	El usuario ya ha realizado previamente “Check-In, saltándose los pasos 3 y 4.

3.2.5 Cambiar algoritmo

Título:	Cambiar algoritmo.
Descripción:	Alterna el algoritmo que selecciona los lugares recomendados.
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado el botón radial con el algoritmo deseado.
Escenario exitoso:	<p>1.- U: Pulsa en el botón superior izquierdo de navegación.</p> <p>2.- A: Despliega la barra lateral con diversas opciones.</p> <p>3.- U: Pulsa la opción “Ajustes”.</p> <p>4.- A: Abre la ventana de ajustes.</p> <p>5.- U: Selecciona el botón radial correspondiente al algoritmo deseado.</p>

	6.- A: Almacena la opción elegida y actualiza la lista de lugares según el nuevo algoritmo.
Alternativa:	El usuario pulsa el botón de “Volver” pues considera correcto el algoritmo actual en el paso 5.

3.2.7 Iniciar sesión

Título:	Iniciar sesión.
Descripción:	Inicia sesión con cualquier método para utilizar la aplicación
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Haber abierto la aplicación sin un login guardado.
Postcondiciones:	Se ha introducido datos correctos de cualquier tipo de login.
Escenario exitoso:	<p>1.- U: Pulsa el botón de “Iniciar sesión con Facebook”.</p> <p>2.- A: Llama a la API de Facebook, que se encarga de gestionar los datos de sesión.</p> <p>3.- U: Concede autorización a la aplicación y sus permisos.</p> <p>4.- A: Muestra la ventana principal.</p>
Alternativa:	El usuario selecciona la opción de Google o correo en el paso 1.

3.2.8 Cerrar sesión

Título:	Cerrar sesión.
Descripción:	Cierra la sesión del usuario actual, volviendo a la pantalla de login.
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado la opción “Sí” en el diálogo de confirmación.
Escenario exitoso:	<p>1.- U: Pulsa en el botón superior izquierdo de navegación.</p> <p>2.- A: Despliega la barra lateral con diversas opciones.</p> <p>3.- U: Pulsa la opción “Cerrar sesión”.</p> <p>4.- A: Muestra una ventana de confirmación.</p> <p>5.- U: Selecciona la opción “Sí”.</p>

	6.- A: Cierra sesión eliminado el token. 7.- A: Muestra la pantalla inicial de "Inicio de Sesión".
Alternativa:	El usuario pulsa la opción "No", quedándose abierta la barra lateral sin cambios.

3.2.9 Ver Check-Ins

Título:	Ver Check-Ins.
Descripción:	Muestra una lista con los lugares que el usuario ha marcado como "visitados".
Actores:	Usuario (U) y aplicación (A).
Precondiciones:	Se ha iniciado sesión y se está situado en la pantalla principal con la lista de lugares.
Postcondiciones:	Se ha pulsado el objeto de menú "Ver Check-Ins".
Escenario exitoso:	1.- U: Pulsa en el botón superior izquierdo de navegación. 2.- A: Despliega la barra lateral con diversas opciones. 3.- U: Pulsa la opción "Ver Check-Ins". 4.- A: Pantalla de carga, obtiene lugares de BD. 5.- A: Muestra lista de lugares.
Alternativa:	El usuario pulsa el botón "Volver", regresando a la pantalla principal.

3.2.9 Diagrama de casos de uso

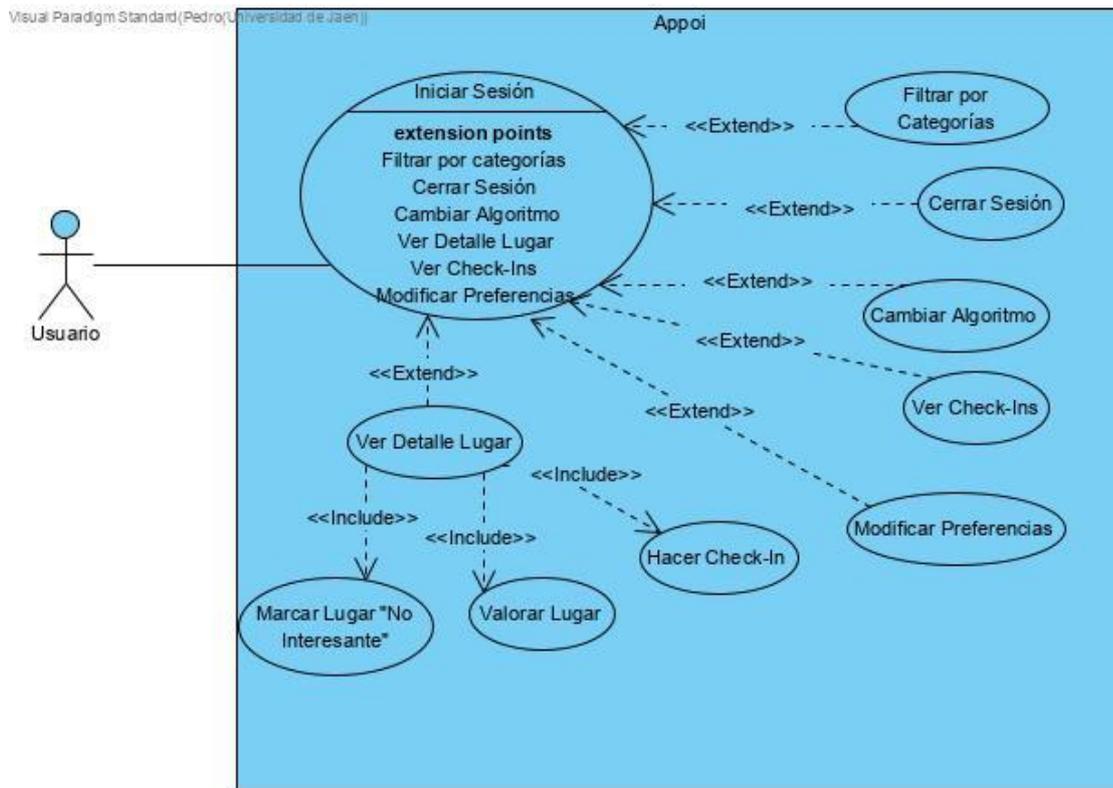


Figura 5: Diagrama de casos de uso.

3.3 Estimación de costes

Es importante realizar un presupuesto aproximado de los recursos humanos, hardware y software necesarios para la implementación del proyecto.

El software utilizado tal como el entorno de desarrollo o la herramienta para el control de versiones es gratuito, salvo el usado para diseñar el prototipo (Balsamiq) que cuenta con una cuota de 9€/mes. Sin embargo, el uso de algunas APIs y servicios en la nube tienen un coste asociado según el uso a tener en cuenta.

SKU	\$200 de crédito mensual Uso gratuito equivalente	Rango de volumen mensual (Precio por miles)		
		De 0 a 100,000	De 100,001 a 500,000	Más de 500,001
Mobile Native Static Maps	Cargas ilimitadas	\$0.00	\$0.00	
Mobile Native Dynamic Maps	Cargas ilimitadas	\$0.00	\$0.00	
Embed	Cargas ilimitadas	\$0.00	\$0.00	
Embed Advanced	Hasta 14,000 cargas	\$14.00	\$11.20	<div style="border: 1px solid black; padding: 5px; text-align: center;"> COMUNICATE CON EL EQUIPO DE VENTAS </div> <p>para obtener descuentos por volumen.</p>
Static Maps	Hasta 100,000 cargas	\$2.00	\$1.60	
Dynamic Maps	Hasta 28,000 cargas	\$7.00	\$5.60	
Static Street View	Hasta 28,000 panorámicas	\$7.00	\$5.60	
Dynamic Street View	Hasta 14,000 panorámicas	\$14.00	\$11.20	

Figura 6: Tabla de precios de la API de Google Maps.

En el caso de la API de Google, se debe tener en cuenta que ofrece un crédito mensual de 200 dólares estadounidenses sólo por tener una cuenta de Google Developer con información de facturación asociada. Al agotarse ese crédito, si así está configurado, se comenzaría a cobrar al propietario de la clave asociada a la API el coste estipulado para que siga funcionando.

Así, tal y como se aprecia en la figura 6, se podría llamar a la función Dynamic Maps hasta 28,000 veces en un periodo mensual sin llegar a gastar más que el crédito gratuito; y una vez rebasado, se aplicaría una tarifa de 7 dólares por cada mil llamadas hasta el total de 100,000 llamadas, pasando la tarifa a 5.6 dólares hasta las 500,000. La aplicación solo usará en “Embed”, por lo que no hay cargo asociado.

RANGO DE VOLUMEN MENSUAL (Precio por LLAMADA)		
0–100,000	100,001–500,000	500,000+
0.032 USD por cada uno (32.00 USD por 1000)	0.0256 USD por cada uno (25.60 USD por 1000)	Póngase en contacto con ventas para conocer el precio por volumen

Figura 7: Tabla de precios de la API de Google Places en la función Nearby Search.

Por otra parte, es imprescindible la API de Google Places, más concretamente la función de Nearby Search para obtener puntos de interés cercanos a la posición del usuario. Como describe la tabla de la Figura 7, se pueden solicitar hasta 6,250 llamadas con el crédito mensual gratuito. Se puede concluir que el uso de esta API

es significativamente más costoso que la de Google Maps, lo que empeora la situación al ser propensa de ser llamada varias veces en una sesión normal de la aplicación.

Debido a ello, es importante considerar como parte de una futura escalabilidad una fuente de ingresos (por ejemplo, publicidad) para compensar el gasto de estos servicios por numerosos usuarios futuros.

Producto	Plan Spark (Gratuito)	Plan Blaze (pago por uso)
Realtime Database: Base de datos no relacional	-5 GB almacenamiento -1 GB de descarga al día -20,000 operaciones de carga por día -50,000 operaciones de descarga por día	-0.026 USD por GB de almacenamiento -0.12 USD por GB descargado -0.05 USD por cada 10000 operaciones de carga -0.0004 USD por cada 10,000 operaciones de descarga
Authentication: Gestión de usuarios	-Autenticación mediante correo, Facebook, Google y otros servicios -Autenticación telefónica: 10,000 al mes	-Autenticación mediante correo, Facebook, Google y otros servicios -Autenticación telefónica: 0.06 USD por verificación
Storage: Almacenamiento de imágenes	-5 GB almacenados -1 GB/día descargados -20,000 operaciones de carga por día -50,000 operaciones de descarga por día -Único depósito por proyecto	-0.0026 USD por GB almacenado -0.12 USD por GB descargado -0.05 USD por cada 10,000 operaciones de carga -0.0004 USD por cada 10,000 operaciones de descarga -Varios depósitos por proyecto

Figura 8: Tabla de planes de Firebase.

Como punto final al análisis de coste de APIs, Firebase es otro servicio que se utilizará en el proyecto y que cuenta con varios planes de pago. Como refleja la figura 8, el módulo de Authentication, que se implementará en la aplicación para el registro de usuarios, entra dentro de los productos gratuitos. Sin embargo, la base de datos en tiempo real NoSql y el almacenamiento de imágenes cuenta con unos límites mensuales en el plan gratuito Spark.

En conclusión, para las pruebas y validaciones de este trabajo no se requerirá de ningún coste adicional por el uso de APIs, pues cuentan con cuotas gratuitas más que suficientes. Por ello, se contarán con coste 0 en la tabla final de gastos.

En cuanto a hardware, se utilizará un ordenador de sobremesa y un smartphone Android Huawei P10, donde se desarrollará y probará respectivamente la aplicación.

Finalmente, para calcular el gasto humano se tendrá en cuenta que en este proyecto solo habrá un único programador. Basándose en el estudio de la consultora Michael Page [34], el salario medio en España de un programador Junior ronda los 20,000€ brutos, lo que significa una cantidad de 1,666€ mensuales aproximadamente. Al ser la estimación de tiempo de mes y medio (45 días), se deduce un coste salarial de 2,500€

Recurso	Coste
1 x Huawei P10	300€
1 x Ordenador sobremesa: -Intel i5 3570 -Disco duro HDD 1 TB -Nvidia GTX 960 -8 GB Ram DDR3	600€
1 x Programador	2,500€
Android Studio	0€
Git	0€
Visual Paradigm	0€
Balsamiq MockUps	18€
Google Places/Maps	0€
Firebase	0€

En conclusión, se estiman un coste de 3400€ en total para el desarrollo del proyecto. Si bien estos costes de Firebase y la API de Google pueden ser variables una vez desplegada la app, no suponen ningún costo para el proceso de realización.

3.4 Metodología de desarrollo

Una parte fundamental al plantear un proyecto de desarrollo es elegir la metodología de ingeniería del software más adecuada. La metodología marca las pautas de trabajo a seguir a lo largo del ciclo de vida, el tipo de relación entre el equipo y el cliente, en qué momentos se realizan análisis o revisiones, etc. Entre las más famosas en la actualidad se encuentran la **metodología tradicional o cascada** y las metodologías ágiles como **SCRUM** [34].

El desarrollo en cascada se caracteriza por la linealidad: cada fase del proyecto se realiza en una secuencia que depende de haber terminado la anterior. Como una cascada que va siempre en una dirección, se pasan por las fases de análisis, diseño, implementación, verificación y mantenimiento sin volver a ninguna de ellas una vez terminadas. Se caracterizan por una definición precisa y específica de todos los requerimientos del proyecto desde el principio del mismo, de modo que no se empieza con el trabajo de implementación hasta que se obtiene una rigurosa planificación. Este tipo de metodología no se adapta a proyectos donde en cualquier momento los requisitos pueden variar.

Entre las ventajas de esta metodología tradicional se encuentran el ser un modelo usado con frecuencia y conocido debido a su antigüedad; estar orientado a resultados; y promover un proceso efectivo de trabajo: planificar y diseñar de forma exhaustiva antes de codificar. Entre las desventajas, está la poca tolerancia a errores detectados a posteriori, pues requieren volver a rediseñar y programar todo el sistema aumentando costes y tiempos. También requiere mucho tiempo hasta tener el producto terminado debido a la linealidad de fases [35].

Por otra parte, la metodología ágil SCRUM es un enfoque incremental: el desarrollo se divide en ciclos temporales llamados **iteraciones** o **sprints**, que suelen durar de 2 a 4 semanas, y que representan un incremento sobre el proyecto final [36]. Cada iteración pasa por su proceso de análisis, diseño e implementación independiente, y el objetivo principal es poder entregarle al cliente varias versiones del proyecto durante el ciclo de vida total, siendo más fácil de adaptar a posibles cambios sobre

la marcha. Para cada iteración se establece con el cliente una lista de **historias de usuario**, que consisten en requisitos explicados de forma concisa en lenguaje no técnico. Estas tareas se suelen asignar con una puntuación numérica, normalmente perteneciente a la sucesión de Fibonacci, que sirve para definir el tamaño de la misma, siendo solo relevante para el equipo.

Existen varios roles a asignar dentro de un equipo SCRUM:

- **Product Owner:** supervisa que el resto del equipo trabaje de forma correcta desde el punto de vista del negocio. También se encarga de escribir las historias de usuario y darles prioridad.
- **Scrum Master:** sirve de guía para todo el equipo, verificando que se cumplan las reglas y procesos de SCRUM. No es un líder como tal, pues cada equipo se autogestiona, pero se encarga de que se cumplan las relaciones entre equipos y elimina obstáculos.
- **Team:** son el grupo de desarrollo con los conocimientos técnicos que se encargan de realizar las historias de usuario de cada iteración.

Existen varios tipos de reuniones recurrentes entre los participantes de esta metodología, como el Sprint Planning donde el Product Owner presenta al resto las historias de usuario a realizar, dándoles prioridad y estableciendo que equipos se hacen cargo de ellas. También existe el Daily sprint meeting, una reunión diaria de 15 minutos de duración en la que cada equipo comenta que resultados y avances obtuvo el día anterior, con el objetivo de sincronizarse entre ellos para el trabajo del día[37].

Como ventajas SCRUM ofrece una rápida respuesta a cambios, la participación cercana del cliente en el proceso de desarrollo y la entrega por ciclos del producto. Como inconvenientes, puede ser un problema una falta de líderes competentes al depender de ellos cada grupo. También suele faltar documentación [38].

Para este proyecto se puede determinar que una metodología ágil no es lo ideal al no existir las figuras de un cliente y un equipo de desarrollo pues, al final, el principal atractivo de este modelo es el organizar de forma efectiva el trabajo, en proyectos grandes, entre los distintos componentes de un equipo numeroso y mantener un continuo contacto con el cliente. Puesto que se tiene una idea relativamente clara de lo que se va a realizar, tiene más sentido optar por una metodología en cascada.

Sin embargo, debido a una cuestión personal al estar interesado en familiarizarme con una metodología tan usada y requerida actualmente como **SCRUM**, se ha decidido usar la misma en el desarrollo de este trabajo. Pero para ello, debido a las circunstancias, se tiene que adaptar mediante algunos cambios la metodología SCRUM típica. Por ejemplo, el asumir todos los roles a la vez, no existir un cliente que mantener al día, o la ausencia de reuniones de equipo que limitarán a revisiones periódicas de lo hecho hasta el momento. Se mantendrá la división del trabajo en iteraciones e historias de usuario, analizando y diseñando mediante diagramas de clases y de secuencias cada incremento.

3.5 Estimación de tiempos

Como se ha decidido llevar a cabo una metodología ágil, el desarrollo se dividirá en entregas incrementales (iteraciones). Se estiman un total de ocho iteraciones que se alargarán en un total de 45 días aproximadamente.

Iteración	Tiempo (días)
Primera iteración: interfaz e implementación de pantalla de login	13
Segunda iteración: interfaz e implementación de pantalla principal sin ordenar lugares	6
Tercera iteración: interfaz con detalles de un lugar	4
Cuarta iteración: Implementación base de datos Firebase	7
Quinta iteración: implementación de barra lateral; cierre de sesión, ignorados, Check-Ins y ajustes	4
Sexta iteración: filtro de categorías y barra de búsqueda	5
Séptima iteración: sistema de perfil y amigos	6
Octava iteración: implementación de algoritmos de recomendación	10

Cada iteración está compuesta por una o varias historias de usuario con una puntuación acorde a la dificultad y tamaño estimados, dando en total el proyecto una cantidad de 223 puntos de historia.

4. Diseño de la solución

4.1 Prototipo



Figura 9: Prototipo de pantalla de inicio de sesión.

En la figura 9 se muestran las distintas opciones de inicio de sesión que estarán disponibles: correo y contraseña propios o a través de proveedores externos como Facebook y Google.



Figura 10: Prototipo de pantalla principal.

En la figura 10 vemos que la pantalla principal estará dividida en dos partes: una lista con los POIs cercanos ordenados según su puntuación de afinidad al usuario y un mapa con los mismos puntos marcados. Pulsar una fila centra el mapa en el POI y carga la pantalla de detalles del lugar.



Figura 11: Prototipo de barra de menú lateral y sus opciones.

La barra de menú lateral de la figura 11 se despliega desde la pantalla principal y permite acceder a distintas pantallas como de Perfil o Ajustes, así como cerrar la sesión actual.

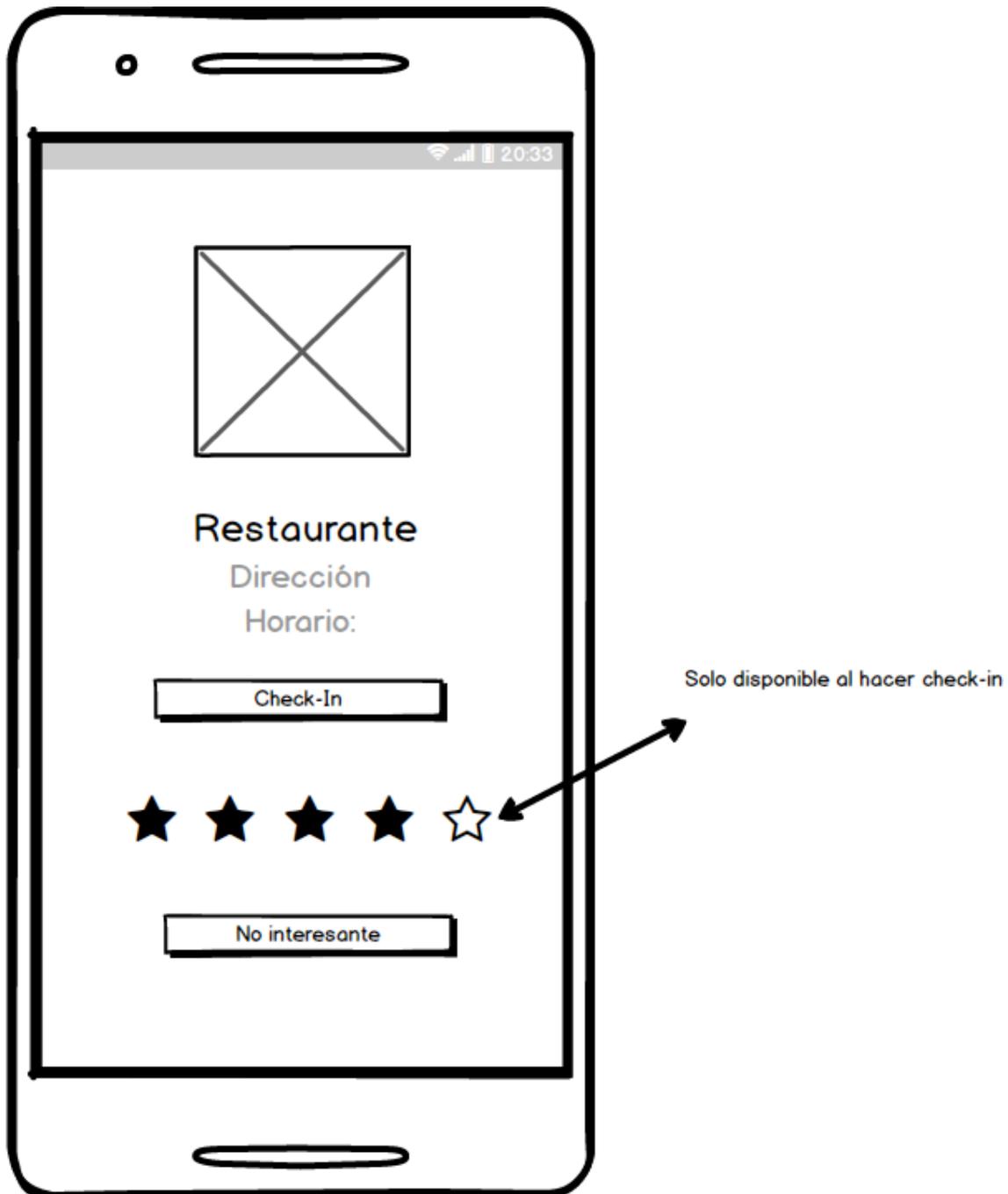


Figura 12: Prototipo de detalles de un punto de interés.

Esta vista detallada de un POI, en la figura 12, se accede al pulsar sobre uno en una lista (ya sea la principal o la lista de "Check-Ins" o Ignorados). Muestra información como una foto del lugar, el nombre, dirección y su horario. Permite mediante un botón hacer "Check-In" al POI, puntuarlo mediante una barra de estrellas del 0 al 5 y marcarlo como ignorado.

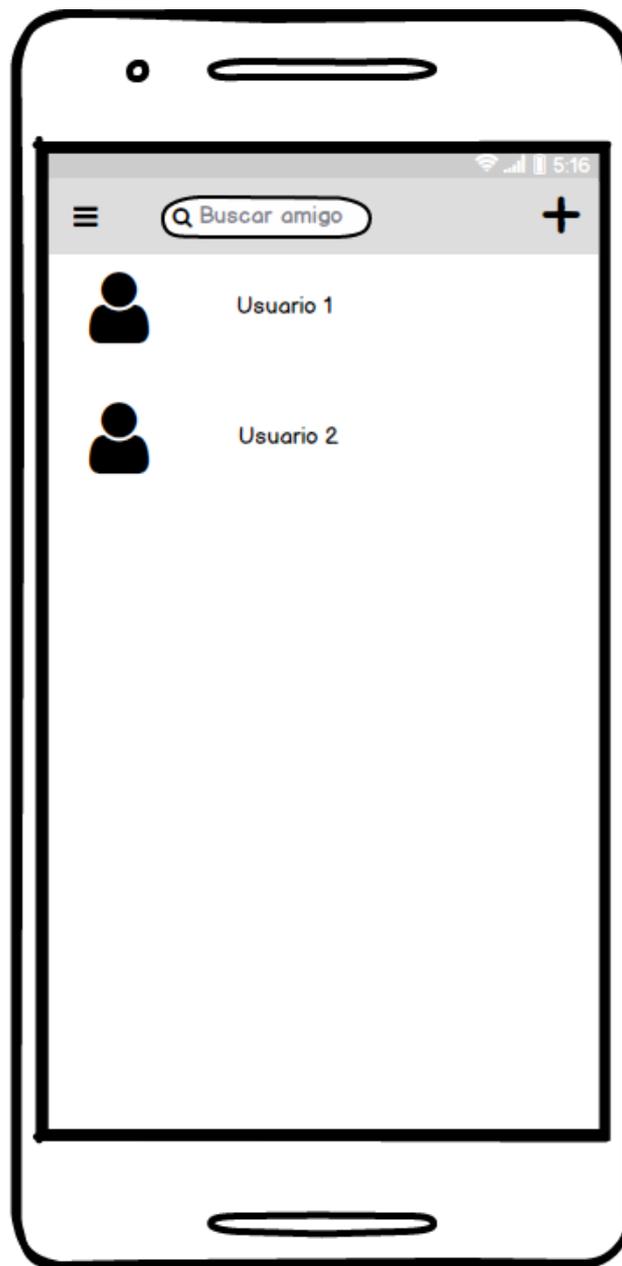


Figura 13: Prototipo de lista de amigos.

Se accede desde el menú lateral y muestra en una lista aquellos usuarios que se han agregado como amigos, como vemos en la figura 13. Se pueden filtrar por su nombre en el campo de texto superior y acceder a sus perfiles al pulsar sobre sus filas correspondiente.

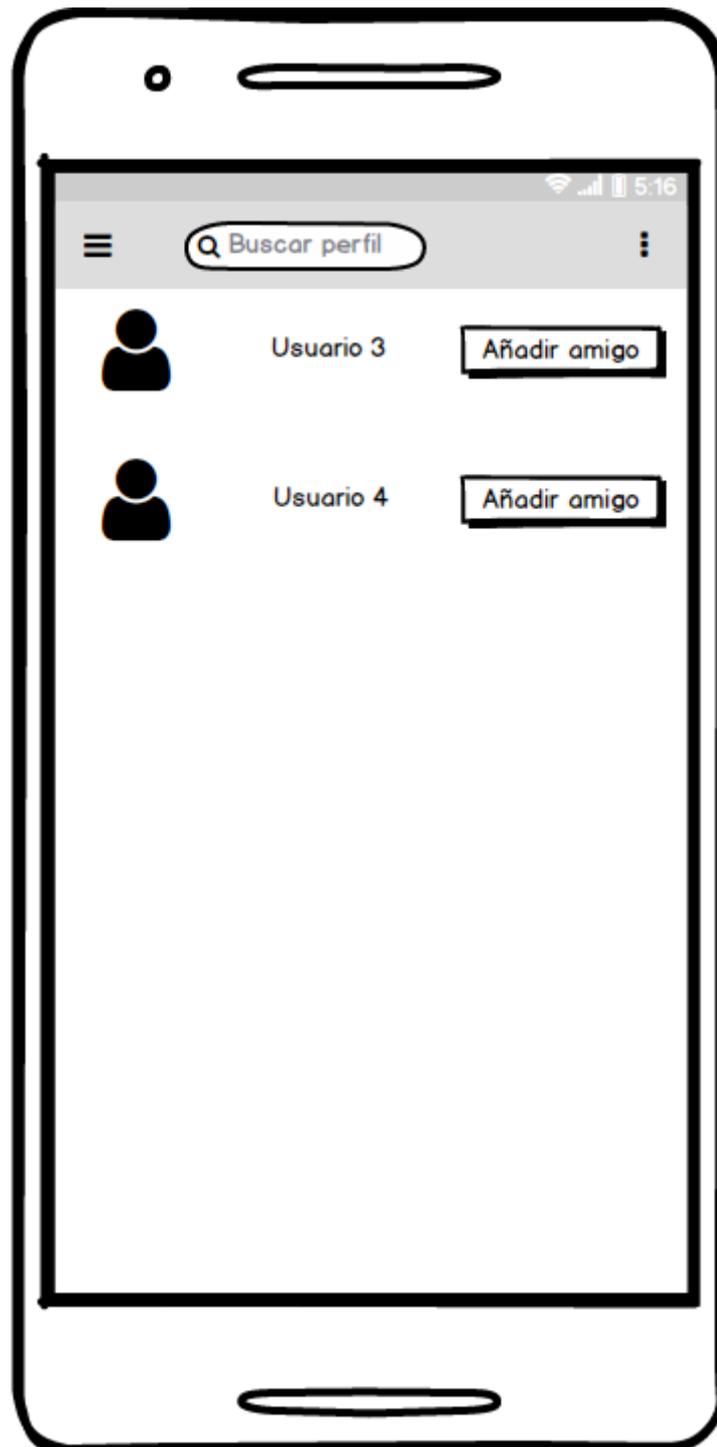


Figura 14: Prototipo de búsqueda de usuarios.

La pantalla de la figura 14 se accede al pulsar el botón "+" de la pantalla de amigos. Funciona de forma similar, pero muestra los usuarios no agregados a amigos de la BD. Existe un acceso directo a "Añadir a Amigos" en la propia fila.

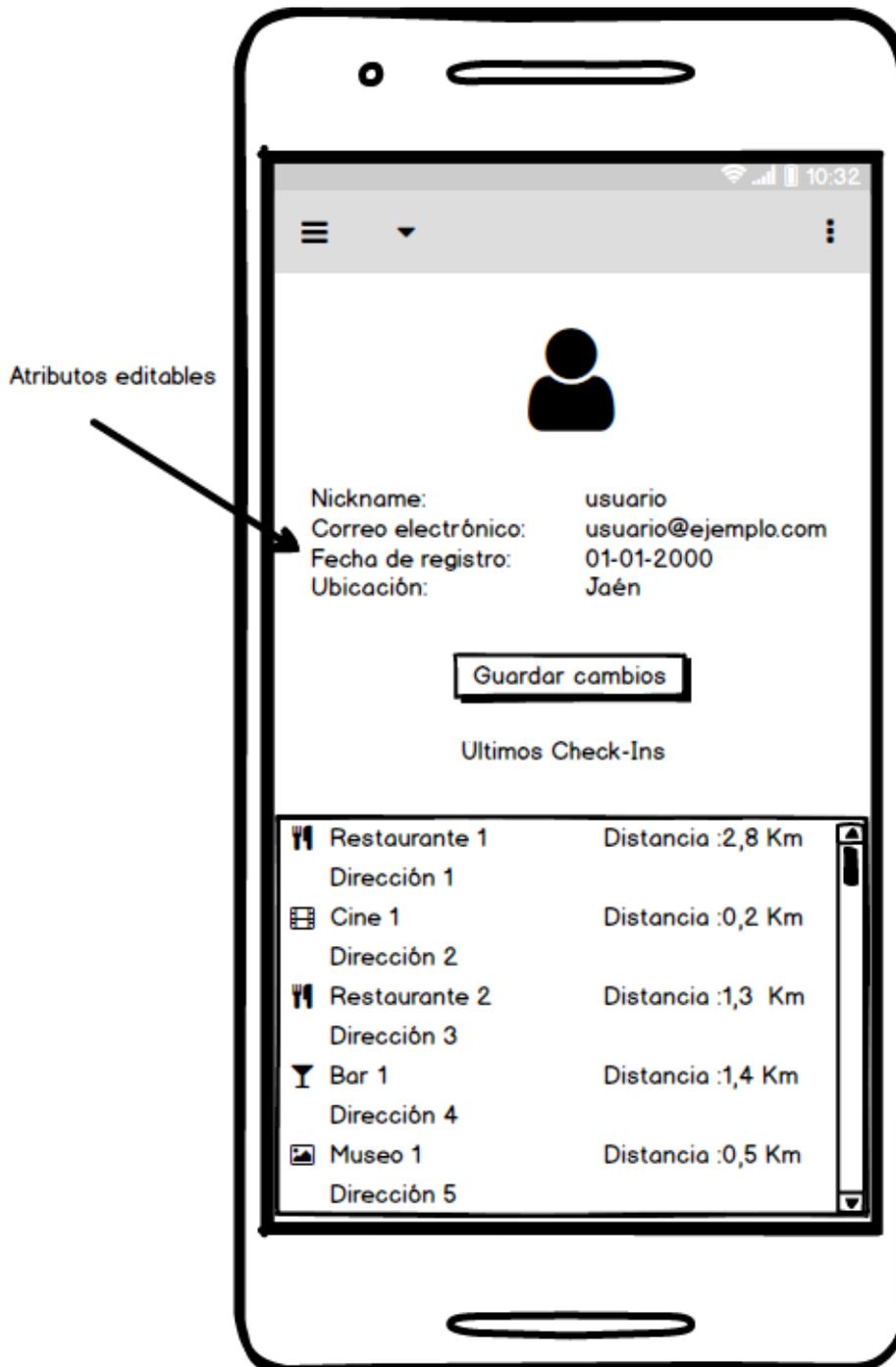


Figura 15: Prototipo de vista de Perfil.

Se accede desde el menú lateral. Se muestran la foto de perfil y los datos del usuario, así como los últimos 5 Check-Ins en una lista en la mitad inferior de la pantalla como muestra la figura 15. Se pueden editar varios datos y la foto de perfil. También es la misma pantalla de la vista de perfil de otro usuario distinto, sin permitir la edición de información y cambiando el botón de "Guardar Cambios" por el de "Agregar a Amigos".

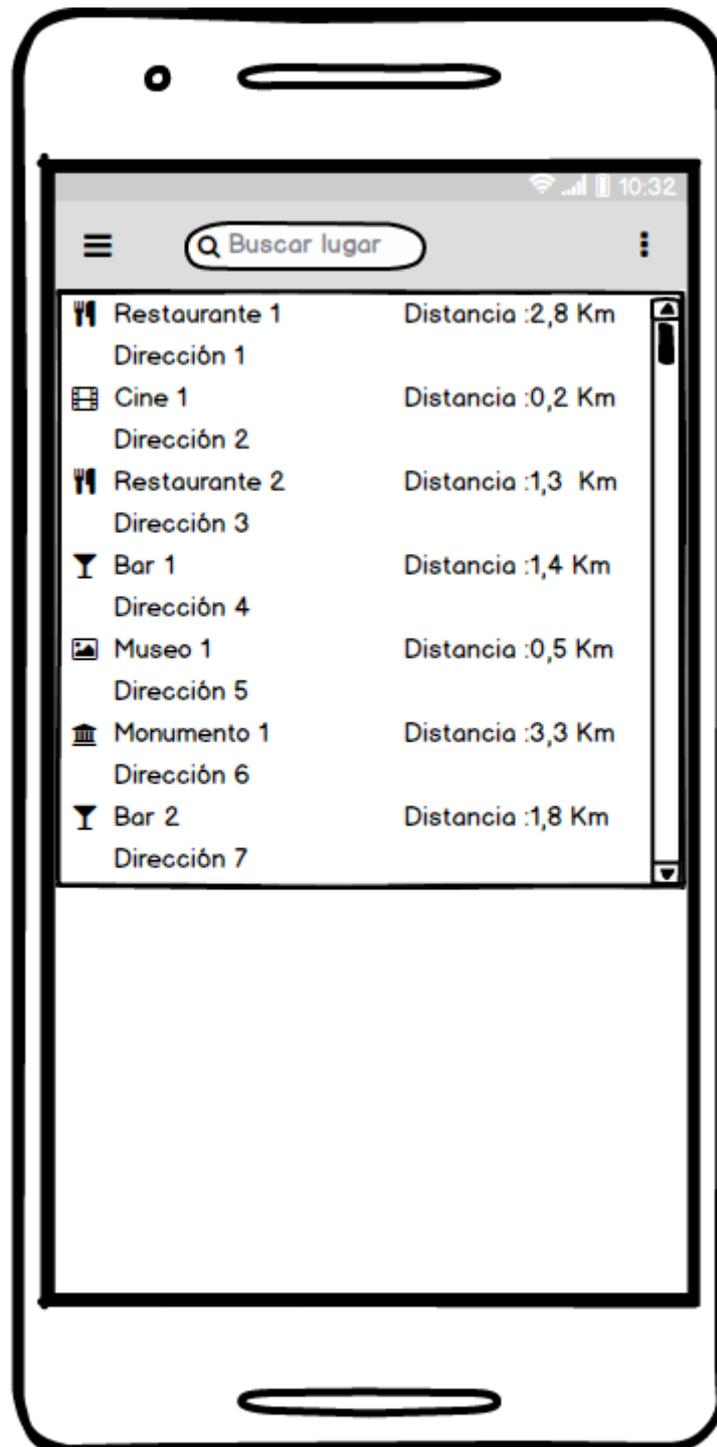


Figura 16: Prototipo de vista de Check-Ins e Ignorados.

La vista de la figura 16 representa las vistas de lugares visitados y de lugares ignorados. Ambas se acceden desde el menú lateral, mostrando los POIs correspondientes en una lista. En la barra superior se pueden filtrar por nombre. Al pulsar sobre una fila se accede a su vista detallada como en la lista principal.

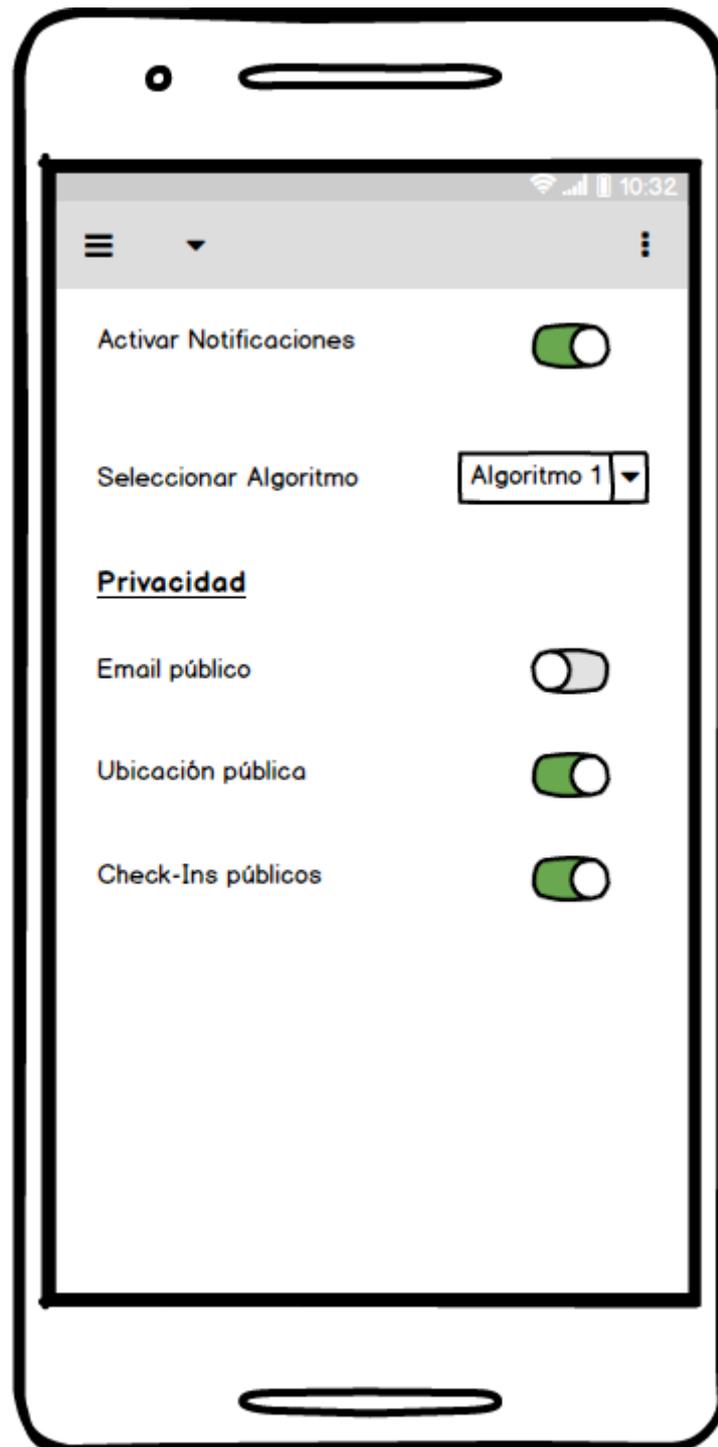


Figura 17: Prototipo de vista de ajustes.

Se accede desde el menú lateral y permite modificar varios ajustes como las notificaciones, la selección de algoritmo de recomendación y la privacidad (datos de perfil que se muestran al público como ubicación o Email), visibles en la figura 17.

4.3 Herramientas utilizadas

A continuación, se realizará un repaso de las distintas herramientas que se utilizarán para el desarrollo del proyecto, pasando por el entorno de desarrollo, el control de versiones y la base de datos.

4.3.1 Android Studio

Android Studio es el IDE oficial de Google para desarrollo en Android. Basado en el IDE IntelliJ IDEA de JetBrains, y licenciado mediante Apache 2.0, permite escribir y compilar aplicaciones en Java y Kotlin, además de contar con herramientas gráficas que facilitan el diseño de las interfaces. También destaca su integración con Git, permitiendo de forma cómoda realizar las operaciones típicas de *pull*, *commit*, *push*, cambios de rama, etc.

4.3.2 Git

Git es una de las herramientas más usadas [39] para llevar el control de versiones en proyectos software. Permite organizar el desarrollo de una aplicación en varias ramas que divergen y convergen, de modo que en un equipo de varios desarrolladores cada uno puede implementar nuevas características y testearlas en el proyecto de forma independiente y sin estorbar al resto.

Aunque solo hay un programador para el desarrollo de Appoi, Git resulta de utilidad para dividir el proceso en las distintas iteraciones de la metodología ágil, además de preservar online una copia del proyecto mediante el gestor de repositorios GitHub.

4.3.3 Firebase

Firebase consiste en una plataforma de desarrollo de aplicaciones web y para dispositivos móviles propiedad de Google [40]. Ofrece una colección de productos y servicios multiplataforma en la nube orientados a facilitar y complementar la creación de aplicaciones sin necesidad de crear un servidor propio para un proyecto. Para la realización de esta aplicación se hará uso de dos de sus servicios:

- **Firestore Authentication:** permite la autenticación de usuarios mediante diversos proveedores de inicio de sesión como Facebook o Google además del clásico registro de correo y contraseña; todo esto utilizando código únicamente en el lado del cliente.

- **Firestore Database:** base de datos NoSQL que puede almacenar y recuperar información en tiempo real y entre distintos usuarios. Es primordial en Appoi para manejar los datos de los usuarios y sus “Check-Ins” y valoraciones.

Todas estas funciones se pueden integrar de forma relativamente sencilla en Android Studio mediante sus SDK correspondientes y apoyadas por la extensa documentación de la API.

5. Implementación de la solución

En esta sección se detalla el proceso de implementación de la aplicación, que es la parte principal de este trabajo al aplicarse todos los conocimientos adquiridos y análisis realizados en las anteriores secciones. Cada iteración se compone de una breve introducción, la explicación de cada historia de usuario implementada y diagramas de clases y de secuencia (si procede).

5.1 Primera iteración: Interfaz e implementación de pantalla de login

Para la primera iteración se implementará la pantalla con los distintos métodos de inicio de sesión disponibles: Email y contraseña, Facebook y Google.

5.1.1 Análisis de primera iteración

El primer paso en la implementación consistirá en diseñar la pantalla de inicio de sesión, necesario para utilizar esta aplicación. Para ello, en Android Studio se deberá crear una **actividad**, esto es, una parte de la aplicación en la que el usuario interactúa mediante interfaz gráfica. Las actividades se componen de un archivo formato KT con el código Kotlin para la lógica y otro archivo en formato XML con la información gráfica asociada. Estos ficheros de interfaz se almacenan en la carpeta */res/layout* dentro de la carpeta principal de la app.

Cada actividad representa una serie de interacciones posibles entre sistema y usuario y pasan por diferentes estados como se aprecia en la figura 18. Mediante la sobrecarga de los métodos asociados a cada etapa del ciclo de vida es posible controlar el comportamiento de la actividad. Por ejemplo, con “onStart()” se puede ejecutar código en el instante exacto en que la actividad se muestra visualmente en pantalla, lo que ocurre después de crearse la actividad por primera vez, momento en el que se ejecutará el código en el método “onCreate()” [41].

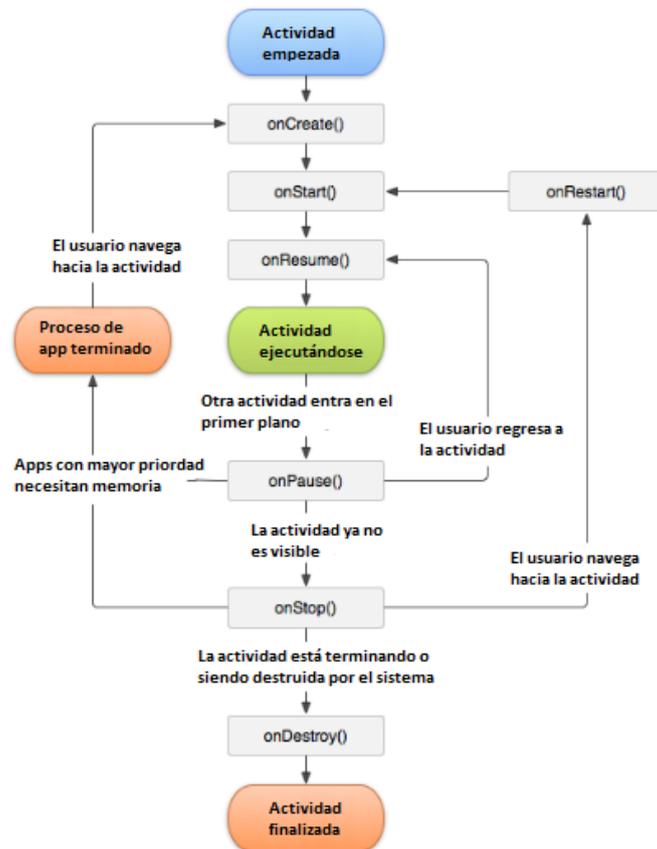


Figura 18: Ciclo de vida de una actividad en Android.

Así, será necesario crear una actividad que contendrá todos los métodos disponibles para iniciar sesión: mediante email y contraseña, Facebook y Google.

5.1.2 Implementación de Actividad de Login

Puntos de historia: 8.

El primer paso será crear un fichero **LoginActivity.kt** y otro **login_activity.xml** asociado. En la actividad principal de la app, **MainActivity.kt**, se llamará a LoginActivity desde el método "OnStart()" en caso de no existir un login recordado previo. La actividad principal es la que se ejecuta en primer lugar al iniciar la app, y se debe indicar su condición en el fichero **AndroidManifest.xml**.

La forma de iniciar una actividad desde otra es mediante un **intent**, que consiste en una descripción abstracta de distintas operaciones como lanzar actividades; iniciar o conectarse a un servicio; o transmitir mensajes a distintos componentes de tipo BroadcastReceiver [42].

Para el diseño de la interfaz es necesario definir un **layout** sobre el que situar los elementos de la actividad. Para este caso se ha optado por un **Constraint Layout**, en el que cada componente se relaciona con otros o con los bordes de la pantalla. Es necesario establecer al menos una relación de distancia vertical y otra horizontal por cada elemento.

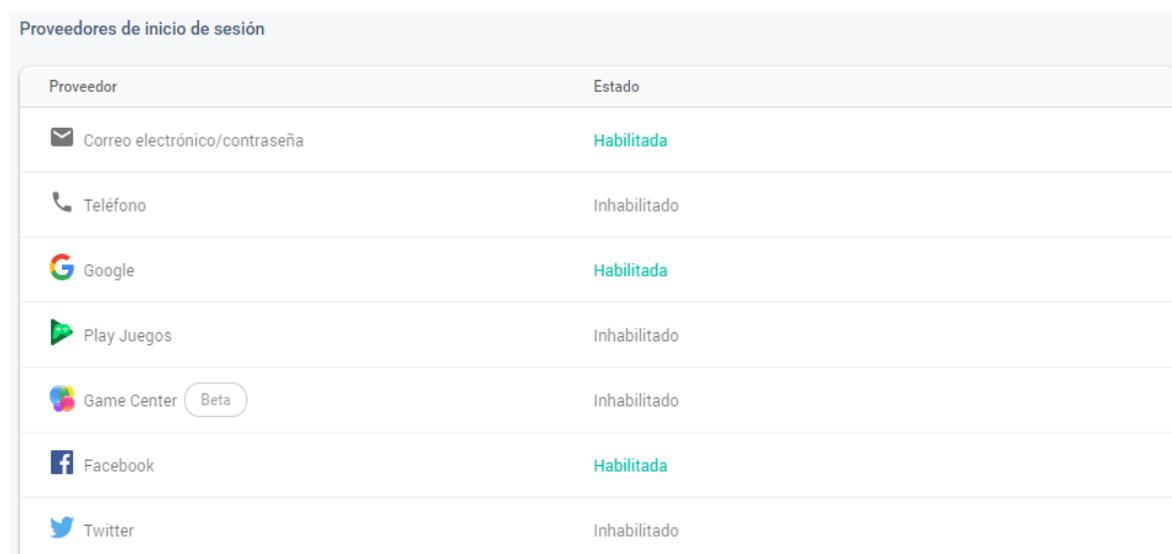
Se necesitará situar en pantalla dos campos de texto para el correo electrónico y para la contraseña mediante la clase `EditText`, configurando el atributo “`autofillHints`” en `username` y `password` respectivamente para que Android sugiera al usuario correos ya usados en el dispositivo en el primer caso y oculte la contraseña en el segundo.

También se deberá crear dos botones debajo de los campos: uno de login que verificará con Firebase que el usuario y contraseña está registrado, dando paso a la actividad principal de la app; y otro de registro con los datos introducidos, que enviará un correo de confirmación a la dirección dada.

5.1.3 Implementación de Login por Email

Puntos de historia: 3.

Para la implementación de funciones de inicio de sesión es necesario preconfigurar Firebase, habilitando el login mediante correo y contraseña en la pestaña de proveedores de inicio de sesión. Aquí se puede hacer lo propio para el resto de logins de Appoi, Google y Facebook, como muestra la figura 19.



Proveedor	Estado
✉ Correo electrónico/contraseña	Habilitada
☎ Teléfono	Inhabilitado
🌐 Google	Habilitada
🎮 Play Juegos	Inhabilitado
🎮 Game Center Beta	Inhabilitado
📘 Facebook	Habilitada
🐦 Twitter	Inhabilitado

Figura 19: Proveedores de inicio de sesión del proyecto de Firebase.

A continuación, una vez asociado al proyecto de Android Studio el SDK de Firebase, se utilizará un *listener* en los dos botones para realizar las acciones de registro y login. Para llevar a cabo ambas acciones primero se deberá capturar una instancia de la clase **FirebaseAuth**, que permitirá interactuar con el servicio de autenticación del proyecto de Firebase, mediante el método estático “`FirebaseAuth.getInstance()`”. A esta instancia, almacenada en la variable “`auth`”, se le pasará los *strings* correspondientes al email y contraseña en sus métodos “`auth.signInWithEmailAndPassword(email, password)`” y “`auth.createUserWithEmailAndPassword(email, password)`” para iniciar sesión y registrarse respectivamente. El correo de confirmación que se le enviará al usuario es configurable en la consola de Firebase.

En caso de un login correcto, solo queda llamar al método “`finishActividad()`” que se encargará de eliminar de la pila de actividades el actual `LoginActivity` e iniciar mediante *intent* la actividad principal con la información de usuario.

Por último, es importante añadir una función que permita recuperar la contraseña en caso de olvido, por ejemplo, a través de un `TextView` bajo ambos botones que llame al método “`auth.sendPasswordResetEmail(email)`”.

5.1.4 Implementación de Login por Facebook

Puntos de historia: 5.

Para añadir un botón de inicio de sesión con Facebook es necesario crear una cuenta de Facebook Developers, crear un proyecto de app Android y descargar e importar el SDK de Facebook [43]. Tras seguir unos pasos que implican añadir la clave hash del proyecto de Android Studio y el nombre del paquete en el proyecto de Facebook, se proporcionará un “App Id” y un “App Secret” que se introducirán en la consola de Firebase para vincular ambos proyectos, como se ve en la figura 20, de modo que un inicio de sesión mediante Facebook en la app se considere un usuario único (identificado por el email) en la base de datos de Firebase.

Facebook

Habilitar

App ID
462182537825312

App Secret
b217aba8550cfc9320883bc7c12be03ez

Para completar la configuración, añade este URI de redirección de OAuth a la configuración de tu aplicación de Facebook. [Más información](#)

https://appoi-262713.firebaseio.com/_/auth/handler

Cancelar Guardar

Figura 20: Vinculación de Facebook con Firebase.

Solo resta añadir a la interfaz el botón de tipo `LoginButton` de la librería de Facebook y gestionar su lógica mediante el método `registerCallback()`. Esta función detectará la presión del botón y, en caso de login exitoso, devolverá un objeto de tipo `LoginResult`. De dicho objeto se puede extraer un `AccessToken` del cual, a través del método `getCredential(token)` de la clase `FacebookAuthProvider`, se obtiene el objeto `AuthCredential` de este login de Facebook. Estas credenciales son imprescindibles para iniciar sesión en Firebase (y obtener así acceso a su base de datos) mediante el método `auth.signInWithCredential(credencial : AuthCredential)`. Toda esta sección una vez obtenido el token se lleva a cabo en el método `handleFacebookToken(token: AccessToken)`.

5.1.5 Implementación de Login por Google

Puntos de historia: 3.

Finalmente, para añadir el login mediante Google se debe añadir a las dependencias del proyecto los Servicios de Google Play. Una vez habilitado el acceso mediante Google en la consola de Firebase y añadido al fichero XML el botón de tipo `SignInButton`, el primer paso es crear y configurar en el método `OnCreate()` un objeto de clase `GoogleSignInOptions` al que es necesario pasar el identificador del servidor de Firebase, que se puede obtener de las credenciales del proyecto.

```

val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
    .requestIdToken(getString(R.string.id_server))
    .requestEmail()
    .build()
googleSignInClient = GoogleSignIn.getClient(this, gso);

```

Al ser presionado el botón de login se inicia un *intent* mediante este objeto, que carga la actividad de inicio de sesión de Google donde el usuario podrá acceder directamente con las cuentas almacenadas en el smartphone. Mediante el método “onActivityResult()”, que es llamado en una actividad de Android cuando otra actividad lanzada desde la misma ha finalizado su ejecución, se puede obtener el objeto de tipo `Task<GoogleSignInAccount>` del que a su vez se puede obtener las credenciales de forma similar al login de Facebook, y así iniciar sesión en Firebase.

5.1.6 Diagrama de clases

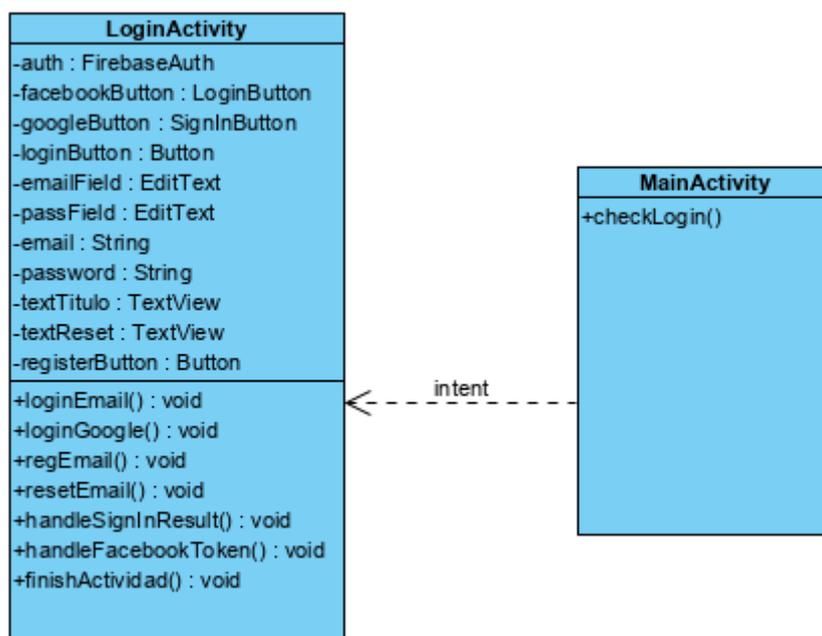


Figura 21: Diagrama de clases de la primera iteración.

En el diagrama de clases de la figura 21 tenemos una clase principal, MainActivity, que va a comprobar al inicio el estado de login mediante su método “checkLogin()”. Si no existe sesión previa, lanza la actividad LoginActivity, que contiene los métodos para cada tipo de inicio de sesión. Además de las distintas variables que

representan los elementos de interfaz, destaca “auth” de la clase FirebaseAuth, que va a encargarse de conectar con Firebase Auth para iniciar sesión o registrarse. Este objeto global también se utilizará en MainActivity para cerrar sesión.

5.1.7 Diagrama de secuencia

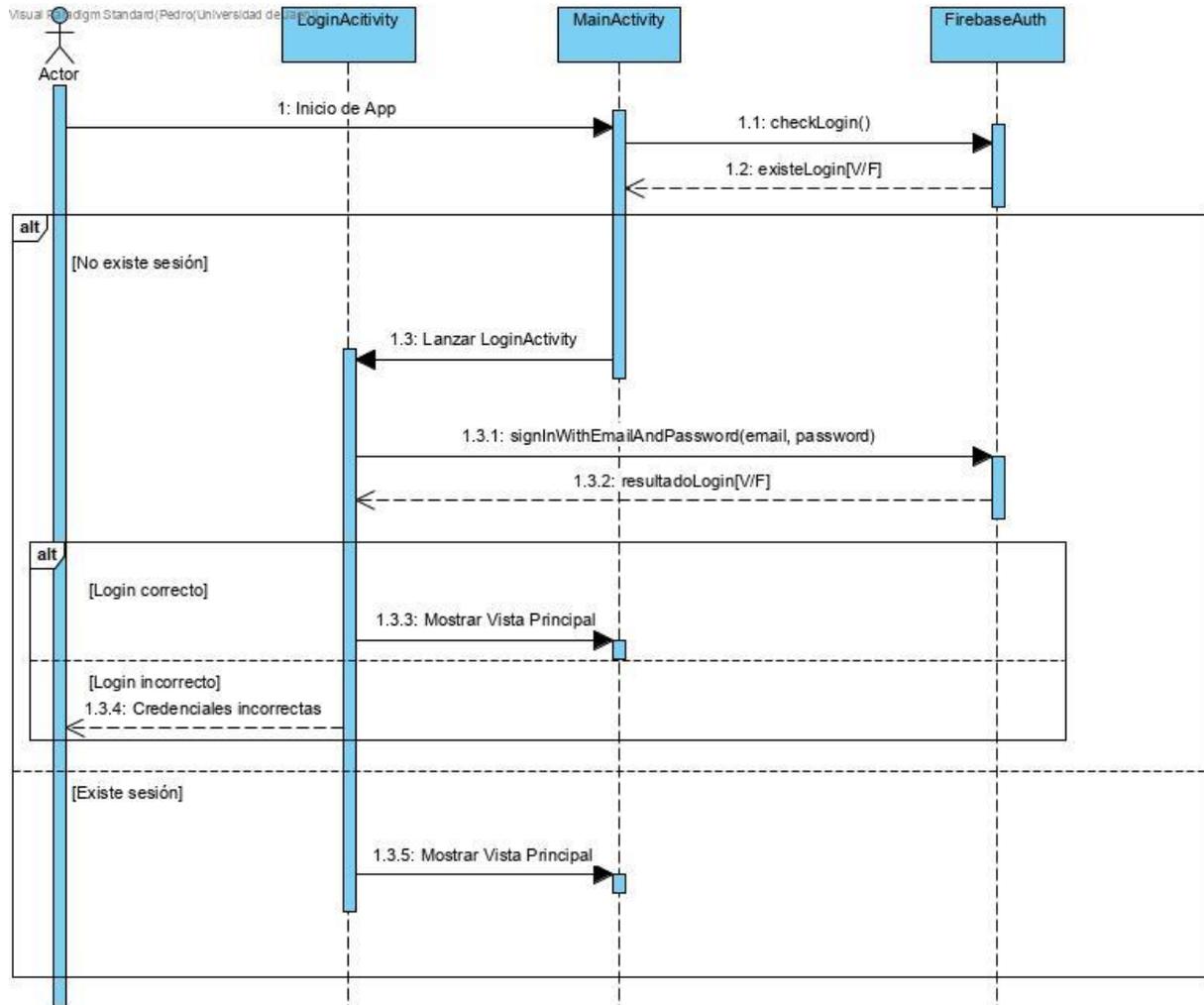


Figura 22: Diagrama de secuencia de la primera iteración.

En la figura 22 se muestra el diagrama de secuencia de esta iteración. MainActivity solicita a FirebaseAuth la comprobación de login, y según la respuesta mostrará la pantalla principal o lanzará LoginActivity. Si el usuario debe iniciar sesión, se envían las credenciales a Firebase y dependiendo de su validez se inicia la pantalla principal o se le indica al usuario el error.

5.2 Segunda iteración: Interfaz e implementación de pantalla principal sin ordenar lugares

En esta iteración se diseñará la actividad principal, compuesta por un mapa y una lista de lugares cercanos, pero aún sin aplicar técnicas de filtrado.

5.2.1 Análisis de segunda iteración

El diseño de **MainActivity** debe dividir la pantalla en dos mitades: una superior que mostrará una lista con los puntos de interés cercanos al usuario y otra inferior consistente un mapa de la zona con marcadores en dichos puntos. Aún no se utilizará técnicas de filtrado sobre los POIs, sino que se pretende probar una actividad que recoja información de lugares de interés cercanos cualesquiera (por ejemplo, todos los bares en un radio de 10 kilómetros) y los muestre según el prototipo de la figura 10.

Los principales retos en esta iteración serán el obtener los lugares mediante Google Places para posteriormente convertirlos en objetos legibles por la app y crear y personalizar un ListView para mostrarlos de forma visual al usuario.

5.2.2 Implementación de Lista de lugares

Puntos de historia: 21.

En primer lugar, se crea el archivo **activity_main.xml** que usará una Constraint Layout. Para la lista de lugares se hará uso de un objeto tipo **ListView** que ocupa la mitad superior de la pantalla.

Dentro de MainActivity.kt, es necesario comprobar en “OnCreate()” que el usuario haya concedido el permiso del servicio de ubicación del dispositivo mediante el método “checkLocationPermission() : Boolean”. Dentro se comprueba primero si ya ha concedido previamente este permiso el usuario. En Android, es necesario exponer en el archivo Manifest los permisos que la aplicación necesita y solicitarlo en tiempo de ejecución [44]. El permiso de localización se indica así en el Manifest:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

La clase MainActivity debe implementar **LocationListener** para poder detectar cambios de posición del usuario y actualizar los datos mediante la sobrecarga del método “onLocationChanged(location : Location)”. Para ello, es necesario un objeto

de tipo **LocationManager** y llamar durante la creación de la actividad el siguiente método:

```
// Distancia mínima de 250 metros
val MIN_DISTANCE = 250f
// Tiempo mínimo de 5 minutos
val MIN_TIME = 300L * 1000L

locationManager!!.requestLocationUpdates(LocationManager.GPS_PROVIDER,
    MIN_TIME, MIN_DISTANCE, this)
```

El primer parámetro indica que se utilice el proveedor de GPS, el segundo es el intervalo de tiempo en milisegundos que debe transcurrir para realizar una actualización de la posición (30 segundos) y el tercero es la distancia en metros que el dispositivo debe alejarse desde la última posición capturada para realizar una actualización (250 metros).

Dentro de “OnLocationChanged()” es donde se obtiene la lista de lugares cercanos mediante la API de Google Places apoyándose en dos clases auxiliares: **JSONRequest** y **ParserPlaces**. Primero es necesario construir la dirección URL que se pasará a JSONRequest mediante el método “crearURL(tipo : Srtng) : String” y obteniendo la latitud y longitud actual del objeto Location.

```
private fun crearUrl(tipo : String) : String{
    val radio : Int = 10000

    var sb : StringBuilder = StringBuilder(url)

    // Localización actual ( se actualizan en "OnLocationChanged(Location)")
    sb.append("location=" + latitud + "," + longitud)
    // Radio de búsqueda
    sb.append("&radius=" + radio)
    // Tipo de lugar (bar, cine, tienda...). Solo uno por url
    sb.append("&type=" + tipo)
    // Key
    sb.append("&key=" + API_KEY)

    return sb.toString()
}
```

Una vez almacenada la URL de solicitud de lugares cercanos, se ejecuta de forma **asíncrona** la tarea JSONRequest, que consiste en una clase interna de MainActivity que hereda de **AsyncTask<>**. Esta clase permite ejecutar de forma asíncrona una tarea en un hilo en segundo plano sobrecargando el método “doInBackground()” y

trabajar sobre los resultados una vez terminada la ejecución en “onPostExecute()” [45].

Dentro de “doInBackground()” y mediante la función “readText()” de la clase “java.net.URL” se obtiene el *String* de respuesta de Google Places en formato JSON, que es necesario pasarlo a un objeto *ParserPlaces* para almacenarlo la información de los lugares en una estructura de datos. La clase *ParserPlaces* está compuesta por:

- Un **HashMap** “lugares” cuya clave es un *String* correspondiente al identificador de Google del punto de interés y cuyo valor es un objeto de la clase **Lugar**. Esta clase incorpora como atributos la información relevante asociada a un punto de interés: latitud, longitud, nombre, horario, tipo...
- Un método “getLugares(jsonResponse : String)” que se encarga de extraer la información relevante de la respuesta JSON y almacenarla en el *MashMap*.

```

class ParserPlaces{

    companion object{
        // HashMap para cada Lugar. Clave: id de google, Valor: objeto Lugar
        lateinit var lugares : HashMap<String, Lugar>
    }
    init {
        lugares = HashMap()
    }

    fun getLugares(jsonResponse : String) : HashMap<String, Lugar>{
        // Primero parseamos el resultado en formato JSON
        var obj : JSONObject
        var arr : JSONArray

        obj = JSONObject(jsonResponse)

        // Array de resultados
        arr = obj.getJSONArray("results")

        // Atributos de cada Lugar
        var nombreLugar : String = "nulo"
        var dirLugar : String = "nulo"
        var horario : String = "nulo"
        var idLugar : String = "nulo"
        var lat : String = "nulo"
        var lng : String = "nulo"

        // Iteración por cada Lugar
        for(i in 0 until arr.length()) {
            // Recuperamos objeto JSON
            obj = arr.get(i) as JSONObject

            // Asignamos valores de propiedades a las variables
            if(!obj.isNull("name"))
                nombreLugar = obj.getString("name")

            if(!obj.isNull("vicinity"))
                dirLugar = obj.getString("vicinity")

            if(!obj.isNull("periods[]"))
                horario = obj.getString("periods[]")

            lat =
            obj.getJSONObject("geometry").getJSONObject("location").getString("lat")
            lng =
            obj.getJSONObject("geometry").getJSONObject("location").getString("lng")

            idLugar = obj.getString("place_id")

            // Creamos el objeto Lugar
            var aux : Lugar = Lugar()
            aux.nombre = nombreLugar
            aux.lat = lat
            aux.lng = lng
            aux.fotos = fotos
            aux.tipos = obj.getString("types")
        }
    }
}

```

```
        aux.direccion = dirLugar
        aux.id = idLugar

        // Actualizamos el HashMap
        lugares.put(idLugar, aux)
    }
    return lugares
}
```

En el método “onPostExecute()” se asigna la vista personalizada a la lista de lugares, siendo necesario crear una clase privada que implemente **BaseAdapter** para determinar el comportamiento del ListView. BaseAdapter es una clase abstracta que implementa la interfaz **Adapter**, la cual hace de puente entre un **AdapterView** (como un objeto ListView) y los datos subyacentes de la vista, así como se encarga de crear un **View** por cada dato [46]. A esta clase, llamada **AdapterList**, se le debe crear un archivo XML para la representación gráfica de cada fila, **row_list_main.xml**, que mostrará información del nombre, distancia y un icono representando el tipo de lugar junto con un color de fondo asociado, como muestra la plantilla de la figura 23.



Figura 23: Plantilla de fila de ListView.

En AdapterList se ha de sobrecargar varios métodos, siendo el más relevante “getView()” pues define la implementación de cada fila. Aquí es donde se asocia el archivo row_list_main mediante un **LayoutInflater**. Pasándole el ArrayList de lugares (los valores del HashMap) se obtiene en orden para cada posición (fila) del ListView los datos de cada lugar, tales como el nombre, tipo y coordenadas, necesarias para calcular la distancia mediante la fórmula del semiverseno en el método “getDistancia(lat : String, lon : String)”.

```

inner class ViewHolderMain{
    var textNombre : TextView
    var textDistancia : TextView
    var icono : ImageView

    constructor(v : View){
        textNombre = v.findViewById(R.id.textNombre)
        textDistancia = v.findViewById(R.id.textDist)
        icono = v.findViewById(R.id.iconRow)
    }
}

override fun getView(position: Int, convertView: View?, parent: ViewGroup?):
View {
    var viewHolder : ViewHolderMain
    var vista = convertView

    val lugarAct = listaLugares.get(position)

    if(vista != null){
        viewHolder = vista.tag as ViewHolderMain
    }else{
        // Se obtiene el Layout XML creado en
        res/layout/row_list_main.xml
        val inflater: LayoutInflater =
        LayoutInflater.from(contextMain)
        vista = inflater.inflate(R.layout.row_list_main, parent,
        false)
        viewHolder = ViewHolderMain(vista)

        vista.tag = viewHolder
    }

    // Distancia
    var distance =
        getDistancia(lugarAct.lat, lugarAct.lng)
    viewHolder.textDistancia.text = distance + " Km"

    // Nombre
    val nombre = lugarAct.nombre
    viewHolder.textNombre.text = nombre
    viewHolder.textNombre.setSelected(true)

    // Tipos de Lugar
    val tipos = lugarAct.tipos
    val layParams: ViewGroup.LayoutParams = viewHolder.icono.LayoutParams

    // Configuración de color de fila e iconos
    if (tipos!! == "Cine") {
        vista!!.setBackgroundColor(Color.parseColor("#e9f511"))
    }
    viewHolder.icono.setImageResource(R.drawable.ic_local_movies_black_24dp)
    }else if (tipos!! == "Bar"){
        vista!!.setBackgroundColor(Color.parseColor("#22b0b5"))
        viewHolder.icono.setImageResource(R.drawable.ic_local_bar_black_24dp)
    }
}

```

```

    }
    else if (tipos!! == "Restaurante") {
        vista!!.setBackgroundColor(Color.parseColor("#deb2a0"))

        viewHolder.icono.setImageResource(R.drawable.ic_restaurant_black_24dp)
    }
    else if (tipos!! == "Discoteca") {
        vista!!.setBackgroundColor(Color.parseColor("#da5ae8"))

        viewHolder.icono.setImageResource(R.drawable.ic_music_note_black_24dp)
    }
    else if (tipos!! == "Museo") {
        vista!!.setBackgroundColor(Color.parseColor("#49de69"))
        viewHolder.icono.setImageResource(R.drawable.ic_museum_24)
    }
    else if (tipos!! == "Tienda") {
        vista!!.setBackgroundColor(Color.parseColor("#ed3b3b"))

        viewHolder.icono.setImageResource(R.drawable.ic_local_mall_black_24dp)
    }
    //Reasignamos dimensiones
    viewHolder.icono.layoutParams = layParams

    return vista!!
}

```

Nótese la implementación del patrón de diseño **ViewHolder**, muy recomendable en Android al usar `ListView`. Este patrón crea un objeto dedicado a contener toda la información de una fila (nombre, distancia e icono en este caso) y lo guarda en memoria de forma que no se tiene que volver a renderizar la información en cada frame, mejorando considerablemente el rendimiento.

Tras la ejecución de `JSONRequest` es cuando se realiza la asignación del adapter personalizado al `ListView`:

```

override fun onPostExecute(result: HashMap<String, Lugar>) {

    listLugares?.adapter = AdapterList(this@MainActivity, latitud,
longitud)
}

```

5.2.3 Implementación de Mapa

Puntos de historia: 8.

Para la implementación del mapa se utilizará un objeto **MapView** que ocupará la mitad inferior de la pantalla en el fichero `activity_main.xml`. Dentro de `MainActivity` se utiliza la clave de API de Google Maps en la creación del objeto `MapView`. Cada vez que se llama “`onLocationChanged(location: Location)`” se debe crear un objeto **Marker** que indica la posición actual del usuario y centrar la cámara en él. Esto se

realiza obteniendo las coordenadas del objeto Location y configurando un Marker con **MarkerOptions** (posición, color y texto al presionar sobre él).

```
// Actualizamos Longitud y Latitud
latitud = location.getLatitude()
longitud = location.getLongitude()

val latLng = LatLng(location.getLatitude(), location.getLongitude())

// Ponemos el marcador en la posición actual
markerOptions.position(latLng)
markerOptions.title("Posición actual")
markerOptions.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED))
currentMarker = gmap?.addMarker(markerOptions)

// Actualizamos el mapa con la posición actual
gmap?.moveCamera(CameraUpdateFactory.newLatLngZoom(latLng, 17.0f))
gmap?.animateCamera(CameraUpdateFactory.zoomTo(12f))
```

Dentro del método “onPostExecute()” de JsonRequest, una vez obtenido el HashMap de lugares, se deben añadir una colección de Marker por cada punto de interés mediante un bucle con su información asociada y un color distintivo. Además, se añade un listener al ListView de lugares para que al presionar en una fila el mapa se centre en el Marker correspondiente. En la figura 24 se ve el resultado final.

```
// Marcadores de Los Lugares seleccionados en el MapView
for (i in 0 until lugares.size){
    var lat : String? = lugares.get(i).lat
    var lng : String? = lugares.get(i).lng

    var coord : LatLng = LatLng(java.lang.Double.parseDouble(lat),
    java.lang.Double.parseDouble(lng))

    markerOptions.position(coord)
    markerOptions.title( lugares.get(i).nombre)

    markerOptions.icon(BitmapDescriptorFactory.defaultMarker(
    BitmapDescriptorFactory.HUE_AZURE))

    //Guardamos en el array de datos
    arrMarker.add(gmap?.addMarker(markerOptions)!!)
}

// Obtenemos el ListView de POIs
listLugares?.adapter = AdapterList(this@MainActivity, latitud, longitud)
listLugares?.setOnClickListener(AdapterView.OnItemClickListener { parent,
view, position, id ->

    val marker = arrMarker.get(position)
    val latlng = LatLng(marker.position.latitude,
marker.position.longitude)
    gmap?.moveCamera(CameraUpdateFactory.newLatLngZoom(latlng, 17.0f))
    gmap?.animateCamera(CameraUpdateFactory.zoomTo(12f))

    marker.showInfoWindow()
})
```



Figura 24: Plantilla principal con lista y mapa.

5.2.4 Diagrama de clases

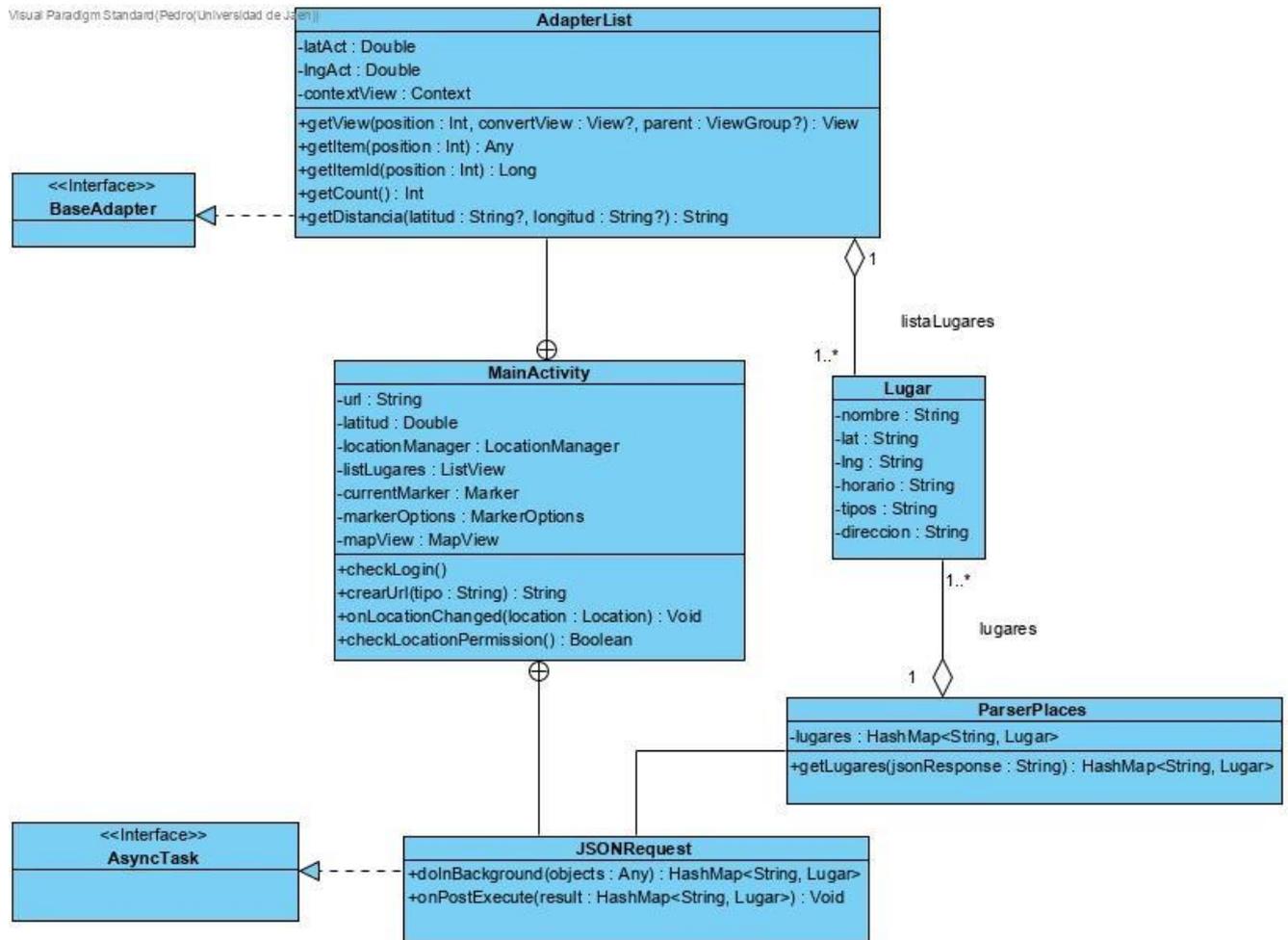


Figura 25: Diagrama de clases de la segunda iteración.

En este diagrama de la figura 25 se han añadido dos clases internas a MainActivity: AdapterList, que implementa la interfaz BaseAdapter para crear un ListView personalizado; y JSONRequest, que implementa AsyncTask para descargar de forma asíncrona el objeto JSON mediante la petición url a la API de Google Places. Al terminar esto último se llama a ParserPlaces que, mediante su método “getLugares()”, convierte el objeto JSON en un ArrayList de lugares, de la clase Lugar, que contiene los atributos necesarios para representar cada lugar en el ListView.

5.2.5 Diagrama de secuencia

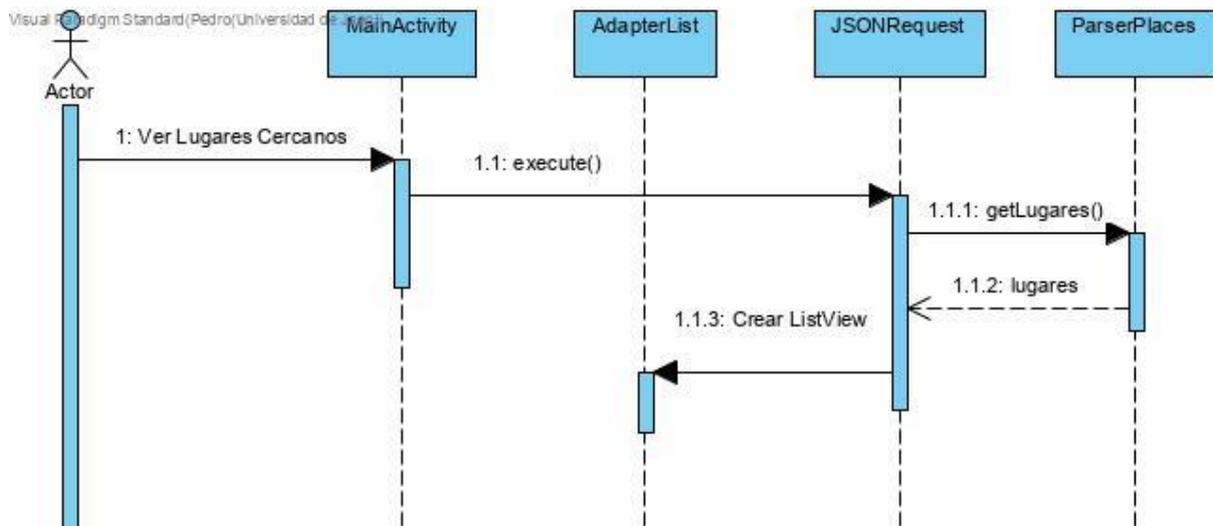


Figura 26: Diagrama de clases de la segunda iteración.

En este diagrama de secuencia, en la figura 26, se aprecia el orden de ejecución de la obtención de lugares cercanos. En primer lugar, MainActivity ejecuta la tarea asíncrona de JsonRequest, que llama a ParserPlaces para recuperar la lista de lugares y finalmente construir con ella el ListView vía AdapterList.

5.3 Tercera iteración: interfaz con detalles de un lugar

En esta iteración se diseñará e implementará la vista detallada de un lugar o punto de interés que se mostrará al usuario al pulsar sobre el mismo en el ListView.

5.3.1 Análisis de tercera iteración

Esta vista debe enseñar datos como el nombre y fotografía del sitio, y también ofrecer funciones al usuario como hacer o deshacer un “Check-In”, calificar el lugar entre una y cinco estrellas y marcar el lugar como “No interesante”, ocultándolo de la lista y mapa.

Estas funciones, sin embargo, necesitan del servicio de la base de datos Firebase que se implementará en la siguiente iteración, por tanto, aquí solo se añadirán los botones a la interfaz XML sin funcionalidad asociada.

5.3.2 Implementación de Interfaz y Actividad de Vista Detallada

Puntos de historia: 8.

Se comienza con la creación de los ficheros **detalle_lugar.xml** y **DetalleLugarActivity.kt** que componen la nueva actividad. En el fichero gráfico se utiliza un *layout* de tipo Constraint Layout. En la parte superior, donde se muestra la fotografía del lugar, se coloca un objeto de tipo **ImageView** que en tiempo de ejecución se sustituye con la imagen correspondiente descargada de Internet. Varios **TextView** justo debajo representan el nombre del lugar, la dirección, y el horario. Este último tiene un color variable según el estado actual del lugar: rojo si está cerrado, verde si está abierto y gris si no existe información. Para los botones de “Check-In” y “No interesante” se utilizan dos objetos **Button** y, entre ellos, un objeto **RatingBar** para la puntuación del lugar. Este es un widget que ofrece la biblioteca de Android que muestra al usuario 5 estrellas en fila, permitiendo registrar una puntuación de 0 a 5 que se almacenará en la BD para que sea utilizada por los algoritmos de filtrado. Esta barra de estrellas tiene por defecto marcado el atributo de *visibility* a 0, volviéndose visible si el usuario realiza o ya ha realizado “Check-In”. En la figura 27 se aprecia como una plantilla de esta pantalla.

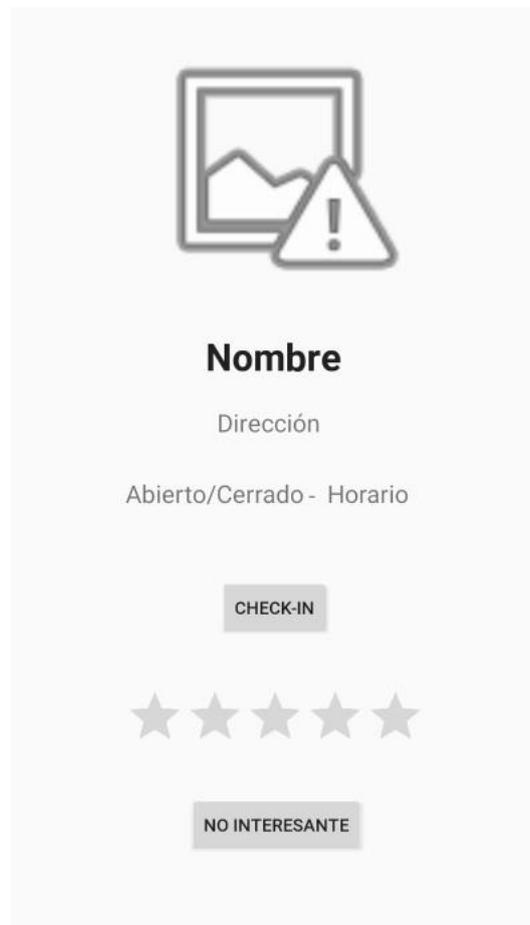


Figura 27: Interfaz de vista detallada.

Solo resta modificar el *listener* del ListView de lugares en MainActivity para que lance vía Intent la nueva actividad. Sin embargo, es necesario pasarle a la actividad el objeto Lugar correspondiente en el Intent, lo que implica implementar en la clase Lugar la interfaz **Parcelable**. Parcelable es una interfaz para clases cuyas instancias pueden ser escritas y restauradas de un objeto **Parcel** [47], que es un contenedor para un mensaje que incorpora referencias de datos y objetos.

5.3.3 Implementación de Descarga de Imagen de un POI

Puntos de historia: 8.

Para mostrar la imagen de un lugar es necesario hacer una llamada extra a la API de Google Places. Una vez más hay apoyarse en una clase interna que implemente AsyncTask<> dentro de DetalleLugarActivity, que se llamará **DescargarImagenTask**.

También hay que crear una clase **Foto** que incluya la referencia (identificador de la API de Google Places), ancho y alto. Al igual que la clase Lugar, debe implementar la interfaz Parcelable para poder pasarse a la actividad DetalleLugarActivity.

En el método “onStart()” de la actividad se realiza la ejecución de esta tarea solo si el array de fotos del objeto Lugar no se encuentra vacío, en cuyo caso se muestra un icono predeterminado. Al método “execute()” se le pasa como parámetros la dirección URL de la API, que se construye previamente mediante “crearUrlFoto()”, y el ImageView. La URL contiene la clave de API, las dimensiones máximas y la referencia de la imagen:

```
var url = "https://maps.googleapis.com/maps/api/place/photo?&" +  
    "key=" + VariablesPrivadas.getApiGoogle() + "&" +  
    "maxwidth=300&" +  
    "maxheight=300&" +  
    "photo_reference=" + lugar.fotos?.get(0)?.ref
```

Dentro de DescargarImagenTask se encuentran dos variables: una referencia al ImageView y un objeto **Bitmap**, que es el resultado de pasarle la dirección URL al método “descargarBitmap(url : String)”. En este método se utiliza la clase **URL** y **URLConnection** para establecer una conexión con Google Places y obtener la información de la imagen en formato **InputStream**, el cual se transforma a Bitmap mediante **BitmapFactory**. Con un mapa de bits podemos mostrar la fotografía en el

ImageView mediante su propio método “setImageBitmap(bitmap : Bitmap)” tras la descarga en “onPostExecute()”.

```
private class DescargarImagenTask : AsyncTask<Any, Int, Bitmap?>() {

    var bitmap : Bitmap? = null
    var imagen : ImageView? = null

    override fun doInBackground(vararg params: Any?): Bitmap? {
        val url = params[0] as String?
        imagen = params[1] as ImageView

        bitmap = descargarBitmap(url)

        return bitmap
    }

    // Actualizamos la imagen al finalizar
    override fun onPostExecute(result: Bitmap?) {

        imagen?.setImageBitmap(bitmap)
    }

    /**
     * Descarga de imagen en formato Bitmap
     */
    fun descargarBitmap(url_ : String?) : Bitmap? {
        var inputStream : InputStream? = null
        var bitmap_ : Bitmap? = null

        try{
            val url : URL = URL(url_)

            val conexion : HttpURLConnection = url.openConnection() as
HttpURLConnection

            conexion.connect()

            inputStream = conexion.inputStream

            bitmap_ = BitmapFactory.decodeStream(inputStream)
        }catch(e : Exception){

        }finally {
            inputStream?.close()
        }

        return bitmap_
    }
}
```

En la figura 28 se aprecia el resultado final de la vista detallada de un POI.

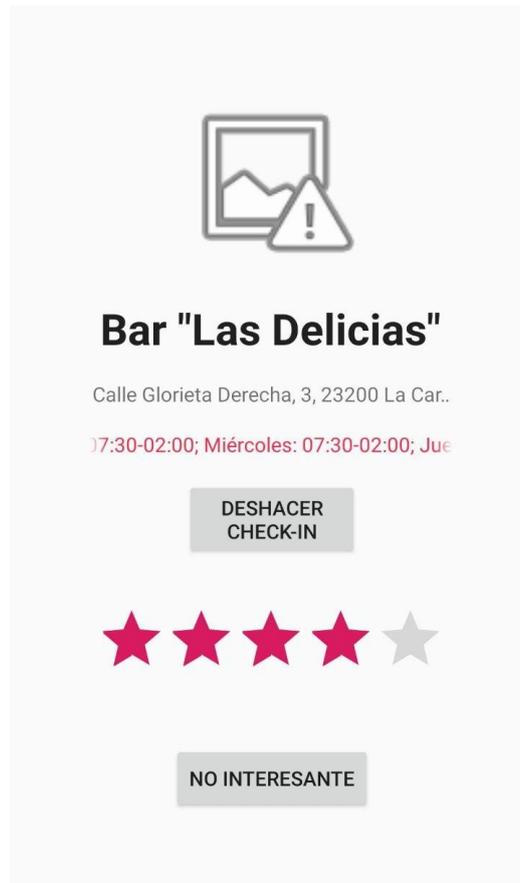


Figura 28: Interfaz de vista detallada final.

5.3.4 Diagrama de clases

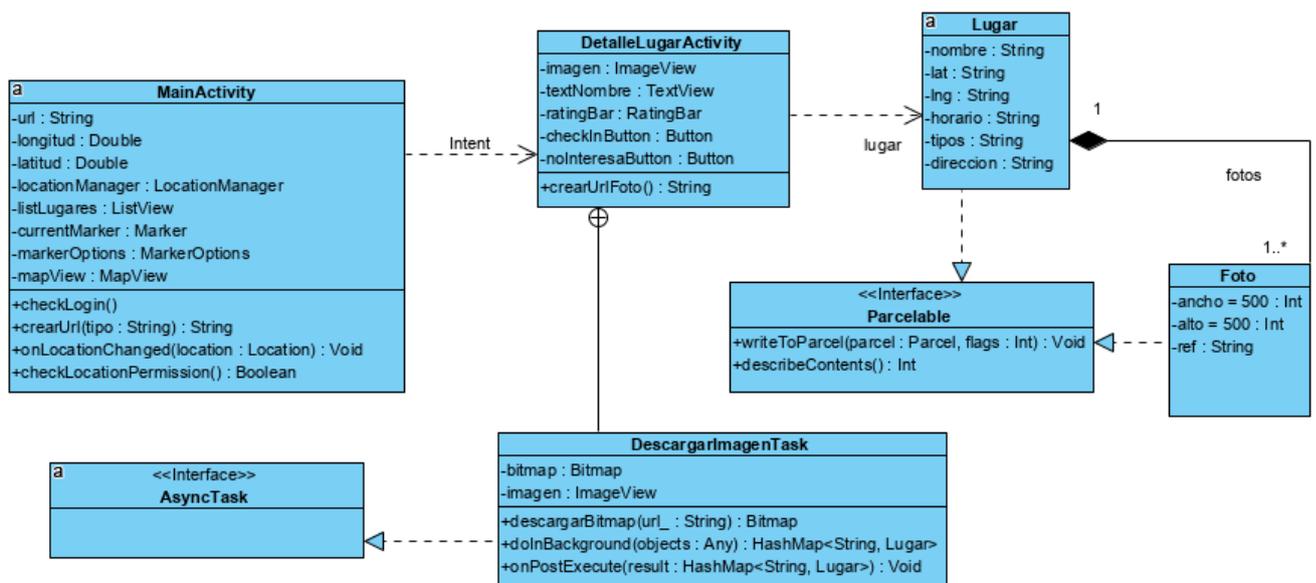


Figura 29: Diagrama de clases de la tercera iteración.

Para este diagrama de clase de la figura 29 destaca la clase `DetalleLugarActivity`, que es iniciada mediante *intent* por `MainActivity`. Está compuesta por un objeto `Lugar` y una tarea asíncrona encargada de descargar imágenes asociadas al POI a través de la información de las mismas. Estos datos que forman una imagen se almacenan en la clase `Foto`, y son las dimensiones y la referencia de Google Places. Nótese que ahora `Lugar` está compuesta por un array de `Fotos`. Además ambas clases implementan la interfaz `Parcelable`, necesaria para pasar objetos de una actividad a otra.

5.3.5 Diagrama de secuencia

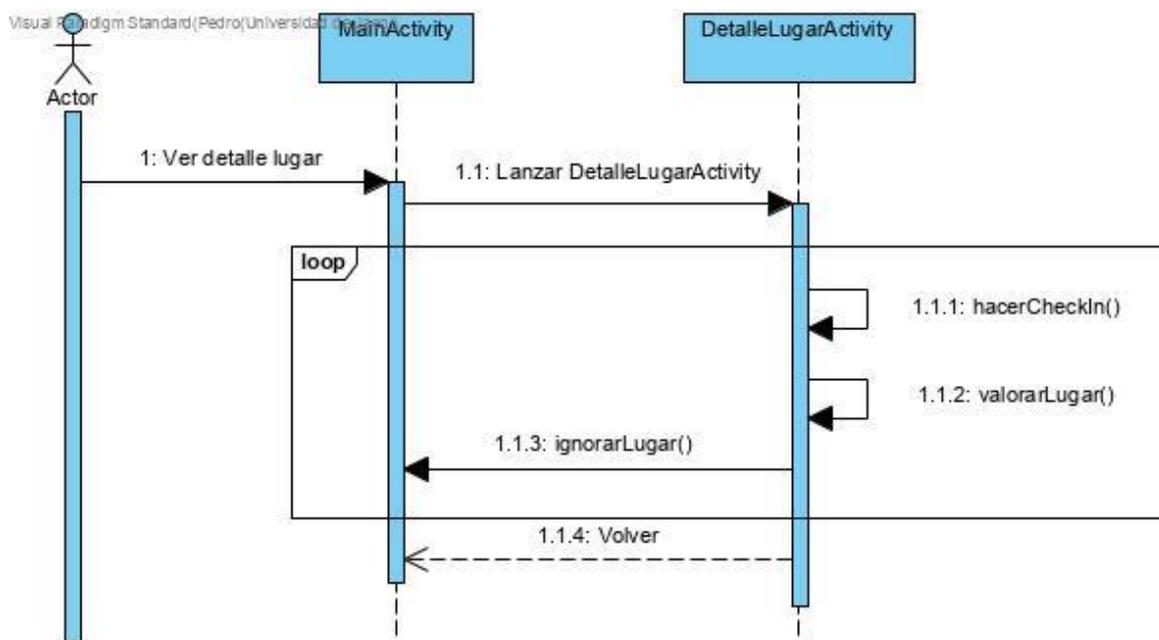


Figura 30: Diagrama de clases de la tercera iteración.

Como vemos en la figura 30, la secuencia de acciones de esta iteración es simple. Cuando el usuario selecciona un POI en la lista de `MainActivity`, esta lanza `DetalleLugarActivity`, pudiendo el usuario realizar acciones de Check-In, valorar el lugar o ignorarlo, lo que termina la actividad devolviéndolo a `MainActivity`.

5.4 Cuarta iteración: Implementación base de datos Firebase

Para la cuarta iteración se debe configurar la base de datos NoSQL de Firebase: Realtime Database.

5.4.1 Análisis de cuarta iteración

La creación de la base de datos implica diseñar un modelo para los datos que serán necesarios almacenar para el funcionamiento de la app, establecer unas reglas desde la consola de Firebase y crear una clase intermediaria en el proyecto de Android cuya instancia permita acceder a métodos de escritura y lectura. Es importante tener en cuenta que Google no permite almacenar o hacer “caching” a la información obtenida de los POIs mediante su API a excepción del atributo “id” que sirve para identificarlos dentro de su propia plataforma [48].

5.4.2 Diseño de la Base de Datos

Puntos de historia: 13.

El primer paso para la preparación de la Base de Datos de Firebase es establecer en la pestaña “Reglas”, mostradas en la figura 31, que solo los usuarios autenticados tienen permiso para leer y escribir, esto es, el identificador de usuario “uid” pertenece al módulo “Auth” del proyecto de Firebase.

```
1
2 // Solo usuarios autenticados
3 {
4   "rules": {
5     "users": {
6       "$uid": {
7         ".read": "$uid === auth.uid",
8         ".write": "$uid === auth.uid"
9       }
10    }
11  }
12 }
```

Figura 31: Reglas de Realtime Database.

A continuación, hay que pensar en cómo estructurar los **nodos** principales de la base. Realtime Database almacena los datos en un árbol de objetos JSON,

llamados comúnmente nodos, con su propia clave asociada [49]. Para adaptarse a las necesidades de la app se ha optado por la siguiente estructura de nodos:

- **Check-Ins:** almacena los “Check-Ins” de los usuarios. Cada entrada tiene una clave única que contiene tres subnodos: el id del lugar, el id del usuario (uid) de Firebase Auth y, si existe, la valoración del lugar de 0 a 5.
- **Usuarios:** Firebase no guarda automáticamente un registro de los usuarios autenticados mediante Auth, por lo que es necesario crear este nodo y agregarlos manualmente en el momento de iniciar sesión junto con el email, comprobando previamente que no estén registrados.
- **Ignorados:** este nodo guarda como entradas principales los uid de usuarios y, dentro de estos, una lista de los id de lugares que este ha marcado como “No interesante”. Se utiliza para ocultarlos de la lista de POIs.
- **Valoraciones:** registro de lugares y sus valoraciones numéricas totales, estando estas asociadas al usuario que las hizo para usarlas en algoritmos de filtrado colaborativo.

En la figura 32 aparece el resultado de la BD con algunos ejemplos.



Figura 32: Estructura de nodos de Realtime Database.

5.4.3 Conexión de Base de Datos y Appoi

Puntos de historia: 5.

Para poder escribir y recuperar datos de la base de datos creada se crea la clase **FirestoreConnector**, que contendrá los métodos necesarios. En primer lugar, se ha de importar la librería específica de Realtime Database en el archivo **Gradle** a nivel de App:

```
implementation 'com.google.firebase:firebase-database:19.2.1'
```

Debido a su naturaleza, esta clase `FirestoreConnector` implementa el patrón de diseño **Singleton**, que en lenguaje Kotlin se realiza de la siguiente manera:

```
companion object{  
    val instance = FirestoreConnector()  
}
```

Se crea un método por cada función necesaria: hacer “Check-In”, deshacer “Check-In”, añadir usuario, etcétera. Para añadir, borrar o modificar un nodo a la base de datos, primero se obtiene una referencia del nodo raíz mediante el método “`getReference(path : String)`” de la instancia `FirestoreDatabase`. A continuación se crea un **ValueEventListener** que de forma asíncrona llama al método “`onDataChange(snapshot : DataSnapshot)`” al asignarlo a la referencia mediante “`addListenerForSingleValueEvent(listener : ValueEventListener)`”. La metodología general para explorar y modificar el árbol de nodos es iterar sobre los “hijos” del objeto **DataSnapshot** mediante un bucle `for`, identificando las claves y valores de los subnodos según sea necesario. Para añadir un nuevo nodo sin valor con una clave aleatoria se utiliza el método “`push()`”. Para añadir finalmente un nodo hijo con uno o más datos asociados, se debe crear una estructura de datos, llamada en Kotlin “`data class`”, personalizada con los atributos necesarios para cada caso: “Check-In”, Ignorado, Usuario y Valoración.

```
data class Usuario(  
    var email : String  
)  
  
data class CheckIn(  
    var id_lugar : String = "",  
    var id_user : String = "",  
    var rating : Float? = null  
)  
  
data class Valoracion(  
    var id_lugar : String = "",  
    var rating : Float? = null  
)  
  
data class Ignorado(  
    var id_lugar : String = ""  
)
```

```

fun addCheckIn(id_lugar_ : String, rating : Float?, button: Button){
    val ref = database.getReference("Check-Ins")

    var key : String? = "null"

    val uid = FirebaseAuth.getInstance().currentUser?.uid

    val listener = object : ValueEventListener{
        override fun onDataChange(snapshot: DataSnapshot) {
            if(snapshot != null) {
                for (ds in snapshot.children) {
                    val valor = ds.getValue(CheckIn::class.java)
                    if (valor != null) {
                        if(uid.equals(valor.id_user) &&
id_lugar_.equals(valor.id_lugar)){
                            key = ds.key
                            crono.countDown()
                            break
                        }
                    }
                }
            }
            if(key == "null"){
                key = ref.push().key
            }

            val checkIn = CheckIn(id_lugar_, uid!!, rating)
            ref.child(key!!).setValue(checkIn).addOnCompleteListener{ task->
                if(task.isSuccessful){
                    button.isEnabled = true
                    button.text = DetalleLugarActivity.CHECKIN_HECHO

                    // Se añade la valoración si existe
                    if(rating != null){
                        writeValoracion(id_lugar_, rating)
                    }
                }
            }

            override fun onCancelled(databaseError: DatabaseError) {
                databaseError.toException()
            }
        }

        ref.addListenerForSingleValueEvent(listener)
    }
}

```

Para almacenar los identificadores de los lugares ignorados se utiliza la estructura de datos **HashSet<T>** cuyo método de búsqueda “contains(dato : T)” devuelve el resultado en complejidad algorítmica $O(1)$. Ahora se recupera en primer lugar la lista

de ignorados y después los POIs cercanos, omitiendo en el HashMap de lugares aquellos cuyo “ids” están contenidos en el HashSet.

5.4.4 Diagrama de clases

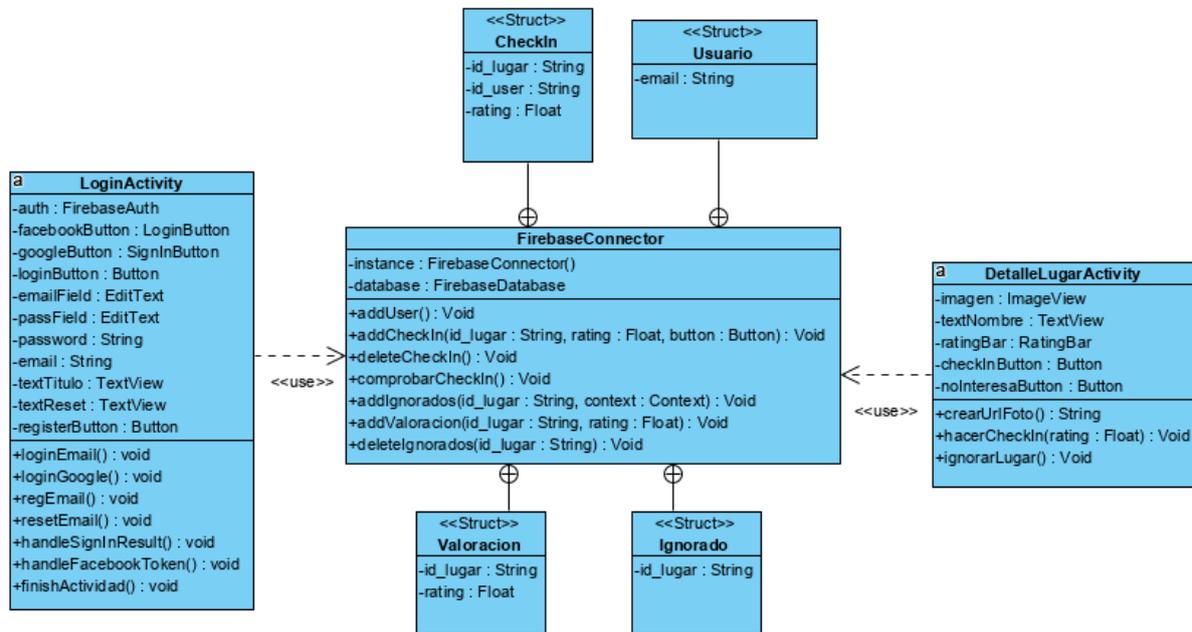


Figura 33: Diagrama de clases de la cuarta iteración.

Como muestra la figura 33, este diagrama de clases se centra en FirebaseConnector y sus métodos que sirven para obtener datos de Realtime Database. Está compuesta de cuatro clases de datos: CheckIn, Usuario, Valoracion e Ignorado, cada una con los atributos necesarios a almacenar. Esta clase es utilizada, mediante llamada a su instancia, por LoginActivity y DetalleLugarActivity.

5.5 Quinta iteración: implementación de barra lateral: cierre de sesión, lista de ignorados, lista de Check-Ins y ajustes

En esta iteración se ha de diseñar e implementar un menú lateral y sus opciones, que implican crear nuevas actividades funcionales.

5.5.1 Análisis de quinta iteración

El llamado “panel lateral de navegación” consiste en un elemento de UI típico de aplicaciones Android que se muestra al pulsar en el icono de tres barras horizontales en la esquina superior izquierda de la pantalla o al hacer el gesto de deslizar el dedo desde el borde izquierdo hacia la derecha. Este panel está formado por una

cabecera o “header” que muestra un fondo, icono e información como el usuario que ha iniciado sesión; y por un conjunto de *ítems* que forman un menú de distintas funciones. De momento se añadirán al menú las opciones de cierre de sesión, vista de lugares a los que se ha hecho Check-In, vista de lugares ignorados y pantalla de ajustes, siendo necesario diseñar nuevas actividades.

5.5.2 Interfaz de barra lateral y cierre de sesión

Puntos de historia: 8.

Como primer paso para la implementación de la interfaz es necesario redefinir la *layout* raíz del archivo XML correspondiente a la actividad principal, que en este momento es una `ConstraintLayout`, y que se convierta en una del tipo **`DrawerLayout`**. La `DrawerLayout` es un tipo de *layout* de alto nivel pensado específicamente para contener vistas de tipo “drawer” como es el panel lateral navegación [50]. La opción más sencilla es crear este tipo de *layout* a nivel de raíz y contener dentro de ella la anterior `ConstraintLayout`.

Al `DrawerLayout` se le añade un elemento de tipo **`NavigationView`** que representa la barra lateral izquierda. Este objeto necesita ser completado con dos archivos de interfaz XML adicionales: uno para la cabecera y otro para los elementos del menú.

- **`nav_header.xml`**: la cabecera está compuesta por una **`LinearLayout`** con orientación vertical, que es un *layout* cuya disposición introduce cada elemento en filas uno encima de otro. Sus objetos son un pequeño `ImageView` para el logo de la app, un `TextView` estático con el nombre de la aplicación y otro `TextView` dinámico que mostrará el correo electrónico del usuario, mostrados en la figura 34.

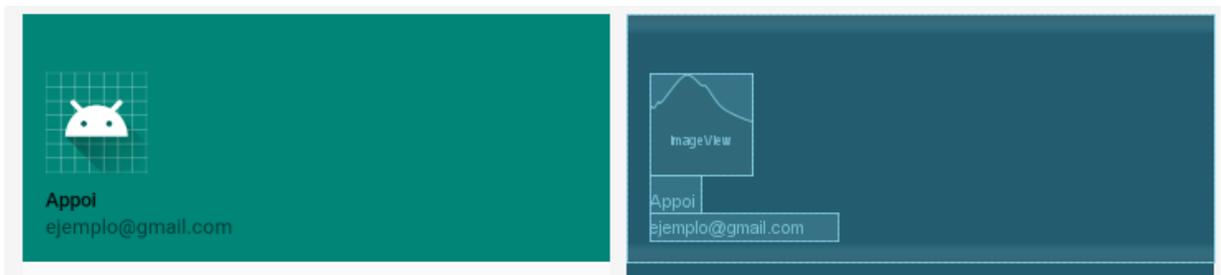


Figura 34: Cabecera de barra lateral.

- **nav_menu.xml**: para el menú se crea el fichero XML en la ruta *res/menu*. Este archivo no utiliza un *layout* típico, sino un contenedor de tipo **Menu** que está compuesto por **grupos** (Group) de **Items**. Cada Item representa una opción del menú, y tiene un identificador con el que hacer referencia en una actividad y asociarle un comportamiento mediante *listeners*.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <group android:checkableBehavior="none"
        android:id="@+id/group1">
    <item
        android:id="@+id/vista_ignorados"
        android:title="Ver ignorados"
        />
    <item
        android:id="@+id/vista_checkIns"
        android:title="Ver Check-Ins"
        />
    <item
        android:id="@+id/ajustes"
        android:title="Ajustes"
        />
  </group>

  <group android:checkableBehavior="none"
        android:id="@+id/group2">
    <item
        android:id="@+id/log_out"
        android:title="Cerrar sesión"
        android:icon="@drawable/ic_exit_to_app_black_24dp"
        />
  </group>
</menu>
```

Finalmente, en el método "OnCreate()" de MainActivity se obtiene la referencia del NavigationView y del DrawerLayout. Para que aparezca el botón de tres barras con el que abrir el panel lateral se le asocia al DrawerLayout un objeto de tipo ActionBarDrawerToggle:

```
// Barra de menú de navegación
val toggleBar : ActionBarDrawerToggle = ActionBarDrawerToggle(this,
drawerLayout, toolbar,
    R.string.navigation_drawer_open, R.string.navigation_drawer_close)
// Se añade
drawerLayout?.addDrawerListener(toggleBar)
toggleBar.syncState()
```

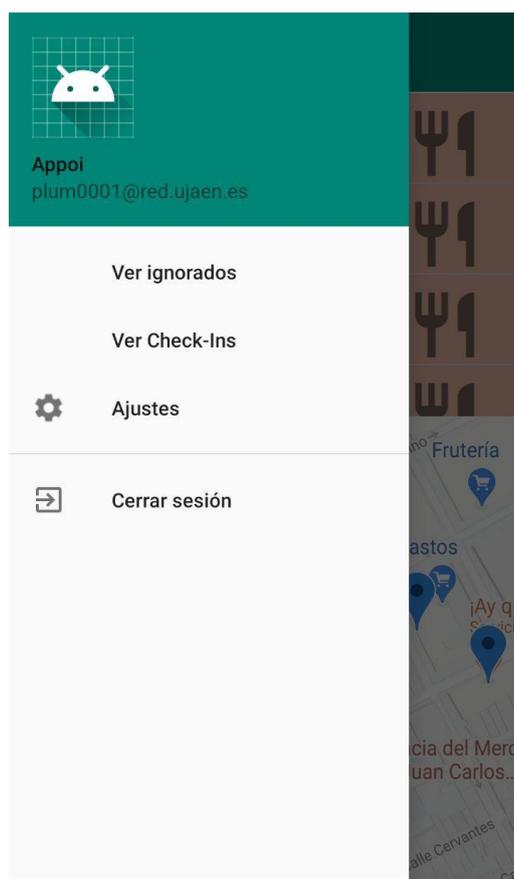


Figura 35: Barra lateral.

Finalmente, se implementa mediante un *switch* el listener para cada ítem del menú, asociando a “Cerrar sesión” los métodos “signOut()” de Firebase, Facebook y Google, y lanzando mediante *intent* la actividad LoginAcitivity. En la figura 35 vemos como queda en la app el menú de barra lateral, al que se la añadirán más ítems en próximas iteraciones.

5.5.3 Implementación de Vista de Ignorados y de Check-Ins totales

Puntos de historia: 13.

Para terminar de implementar la funcionalidad de lugares ignorados restaba dar la posibilidad al usuario de revertir su decisión en caso de error o cambio de opinión. También resulta interesante permitir al usuario revisar todos sus “Check-Ins”, siendo necesario crear en MainActivity otro HashSet<String> para guardar los ids de los “Check-Ins”, creando un método en FirebaseConnector para ello y llamándolo en cada actualización de posición. Para ello, se crean dos actividades que se lanzan desde las opciones de la barra lateral “Ver ignorados” y “Ver Check-Ins”.

Ambas actividades consisten únicamente en un ListView, personalizado mediante la interfaz BaseAdapter de forma similar al ListView principal de POIs, ocupando toda la pantalla. Debido a esta similitud entre ambas se hace conveniente crear una **clase abstracta, ListViewActivity**, de la que heredarán las dos actividades: **IgnoradosActivity** y **VerCheckInsActivity**. También se crea una clase aparte para el *adapter* personalizado del ListView llamada **AdapterListCustom**, que difiere ligeramente del que utiliza MainActivity.

Dentro de ListViewActivity el método más importante es “cargarLugares()” llamado durante la creación. Esta función se encarga de obtener los datos de todos los lugares que se van a mostrar en el ListView mediante la librería de Google Places y los ids de los lugares. Esta librería permite recuperar a través de un ids de Google del POI (guardados en el HashSet correspondiente) la información detallada de los mismos en un objeto **Place**. Así, se itera el HashSet haciendo una llamada al método “fetchPlace(request : FetchPlaceRequest)” del objeto **PlacesClient** por cada id. Además, hay que controlar que el usuario no sea capaz de acceder a esta actividad hasta que no se haya recuperado de la Base Datos los ids de lugares ignorados y de “Check-Ins”, habilitando de forma asíncrona los ítems del menú cuando se termine la lectura de Firebase.

```

fun cargarLugares() {
    var contador = 0

    while (iterator.hasNext()) {
        contador++

        // Obtenemos el nombre y tipos del Lugar
        val placeFields: List<Place.Field> =
            Arrays.asList(Place.Field.TYPES, Place.Field.NAME,
                Place.Field.ADDRESS)

        // Solicitud a la API
        val idIgnorado = iterator.next()
        val request: FetchPlaceRequest =
            FetchPlaceRequest.newInstance(idIgnorado, placeFields)

        placesClient.fetchPlace(request).addOnSuccessListener({ response ->
            val place: Place = response.getPlace()

            val lugar = Lugar()
            lugar.nombre = place.name!!

            var tiposString = ""

            for (type in place.types!!) {
                tiposString = tiposString + type + "|"
            }

            lugar.tipos = tiposString
            lugar.direccion = place.address!!
            lugar.id = idIgnorado

            lugares.add(lugar)
            if (contador == tam) {
                listView.adapter = AdapterListCustom(this@ListViewActivity,
                    lugares)
                listView.setOnClickListener()
            }
        })
    }
}

```

Como atributos abstractos que implementará de forma única cada actividad, esta clase contiene los siguientes:

- **tam : Int**: número de lugares, que es igual al tamaño del HashSet de ids.
- **titulo : String**: título que mostrará la barra superior de la actividad.
- **Iterator : MutableIterator<String>**: el iterador del HashSet correspondiente del que se obtendrán los lugares.

Como método abstracto solo se encuentra “setOnClickListener()”, que define el comportamiento al pulsar una fila del ListView. En el caso de IgnoradosActivity se muestra una ventana emergente que preguntará al usuario si desea quitar de

ignorados ese lugar. En caso afirmativo, se actualizan las estructuras de datos adecuadas y se redibujan los dos ListView de la app. Para VerCheckInsActivity ocurre lo mismo que en ListView de MainActivity: se lanza mediante *intent* la actividad DetallesLugar para el lugar pulsado.

Para la parte gráfica, se crea el fichero **row_list_activity.xml** para representar una fila, que es idéntico a row_list_main.xml salvo por el TextView de distancia, que es sustituido por otro que muestra la **dirección** completa del lugar. También se crea **listview_activity.xml** que contendrá como único elemento dentro de una ConstraintLayout el ListView ocupando toda la pantalla. En la figura 36 vemos como queda la lista de lugares, junto con la ventana emergente al seleccionar un POI.

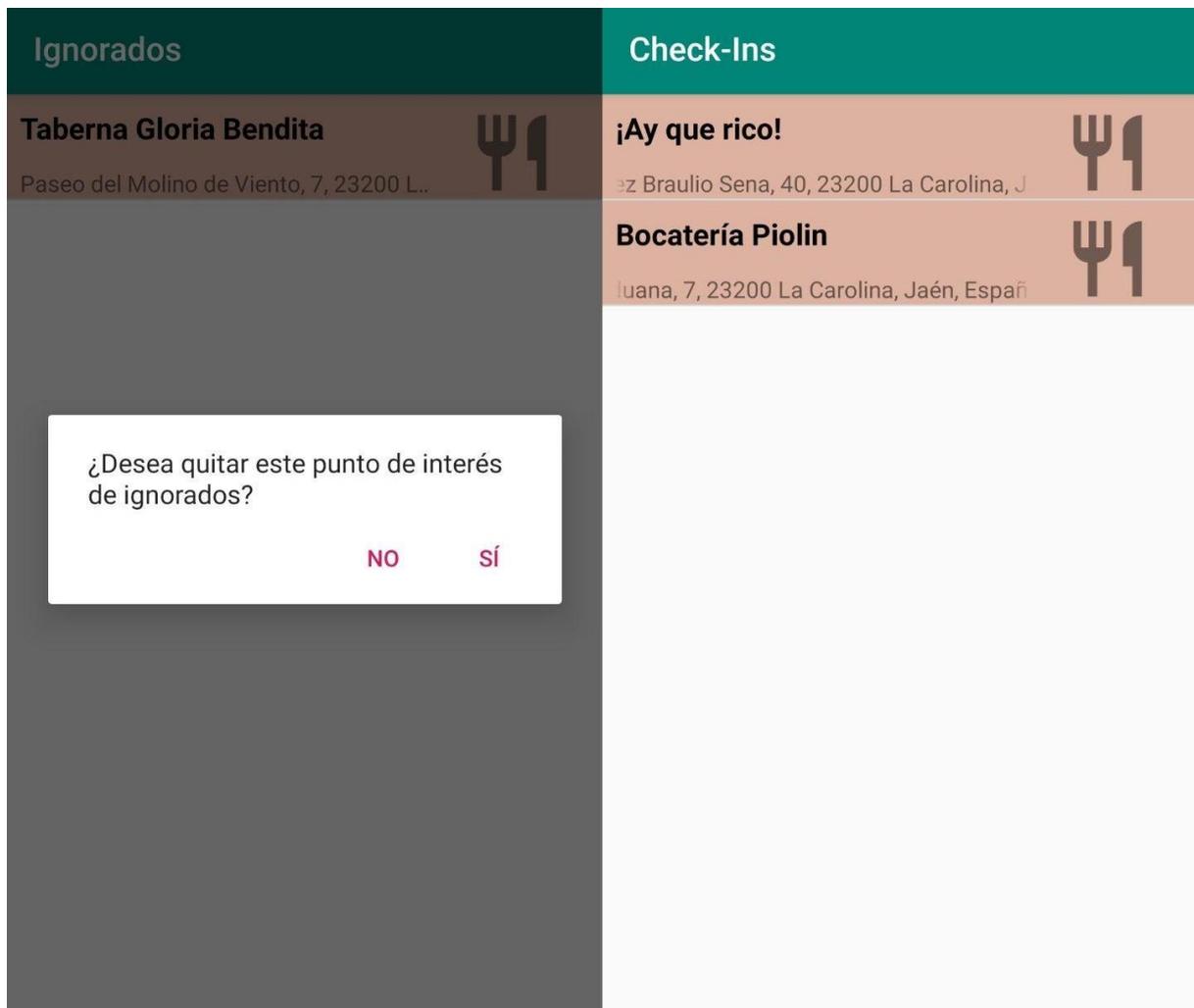


Figura 36: Vistas de Ignorados y "Check-Ins".

5.5.4 Implementación de Vista de Ajustes

Puntos de historia: 8.

Una de las formas más recomendadas para crear una vista de ajustes en Android donde el usuario pueda definir distintas preferencias con respecto a la app es haciendo uso de **PreferenceScreen** y **PreferenceFragment**. PreferenceScreen es un tipo especial de Layout pensado para representar una pantalla de ajustes típica [51]. Contiene componentes específicos como alternativa a otros: PreferenceSwitch en vez de un Switch, por ejemplo. Estos ítems se caracterizan por escribir automáticamente en el espacio **SharedPreferences** de la aplicación. Un objeto SharedPreferences apunta a un archivo que contiene pares de claves-valor y proporciona métodos para leerlos y escribirlos [52]. Cada componente de una PreferenceScreen tiene un valor *key* que lo identifica y un valor *value* que define su estado. Mediante SharedPreferences es posible almacenar de forma persistente distintas preferencias de la app hasta que esta es desinstalada del dispositivo.

PreferenceFragment es una subclase de **Fragment**. Los *fragments* en Android pueden ser vistos como “subactividades” dentro de una actividad que ocupan cierta parte de la interfaz de la misma y son reutilizables en otras. Una actividad puede contener varios de ellos a la vez, registran sus propios eventos de entrada y tienen su propio ciclo de vida [53]. En el caso de un PreferenceFragment, está pensado para ser llenado con una interfaz XML de tipo PreferenceScreen. La principal ventaja es poder sobrecargar el método “onSharedPreferenceChanged()” que detecta un cambio en las SharedPreferences, esto es, un cambio de un tipo de preferencia de la pantalla. Es posible detectar que preferencia ha cambiado el usuario y en qué estado se encuentra cambiando variables globales de otras actividades a las que afecten las preferencias.

Para Appoi se ha decidido agregar dos ajustes: el **envío de notificaciones** y la **selección de algoritmo de recomendación**.

- **Notificaciones:** se controlan mediante un objeto SwitchPreference para indicar si se desea o no que la app muestre notificaciones en la barra de estado de Android. Las notificaciones se envían cuando, estando la app en segundo plano, se ha actualizado la posición y se han encontrado una o más

nuevas ubicaciones con una probabilidad de gustar al usuario alta. Se controlan mediante la variable global Boolean “notificaciones”.

- **Selección de algoritmo de recomendación:** se utiliza un objeto **ListPreference** que consiste en una serie de opciones que simulan el comportamiento de los botones de radio: solo una está activa a la vez. Se llenan de opciones mediante un array de *strings* en el fichero strings.xml. Al seleccionar un algoritmo, se modifica la variable global “algoritmo” con el valor correspondiente de la clase “enum AlgoritmoActivo” que contiene a los 3 que utiliza la app. También se actualizan los textos de *title* y *summary* de la preferencia, mostrando *summary* información sobre el algoritmo activo.

Vemos el resultado final en la figura 37.

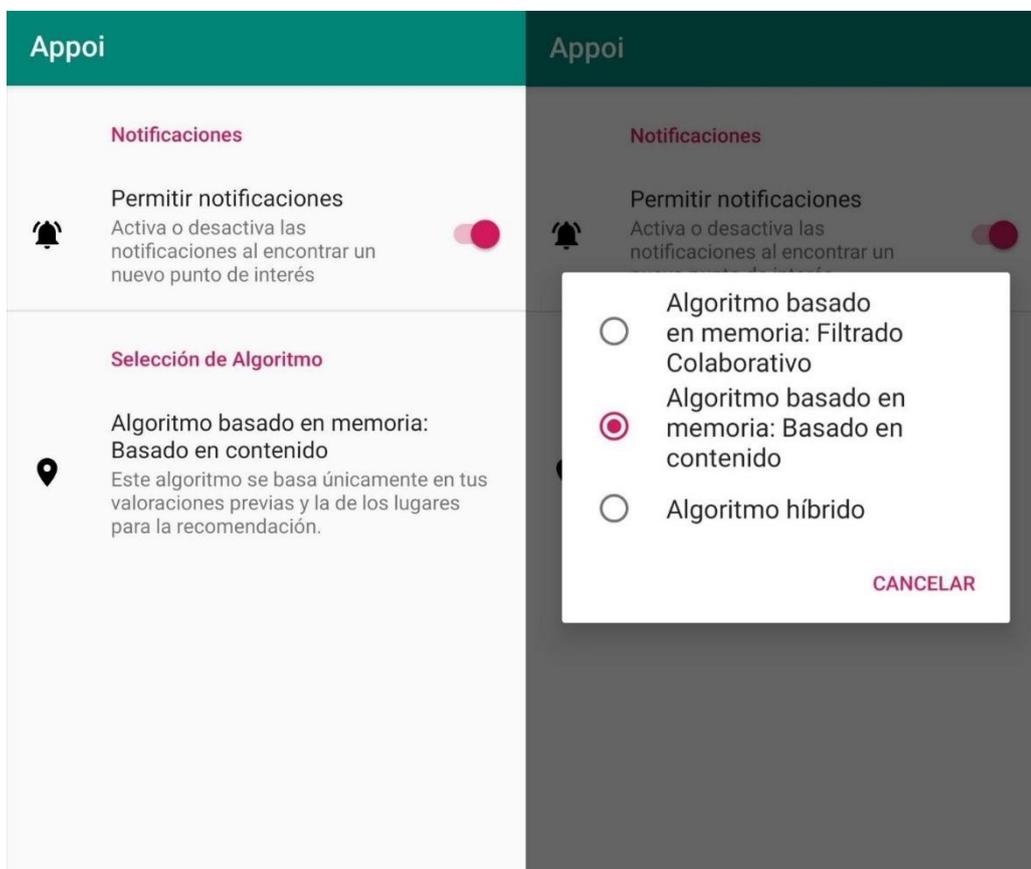


Figura 37: Pantalla de ajustes.

5.5.5 Diagrama de clases

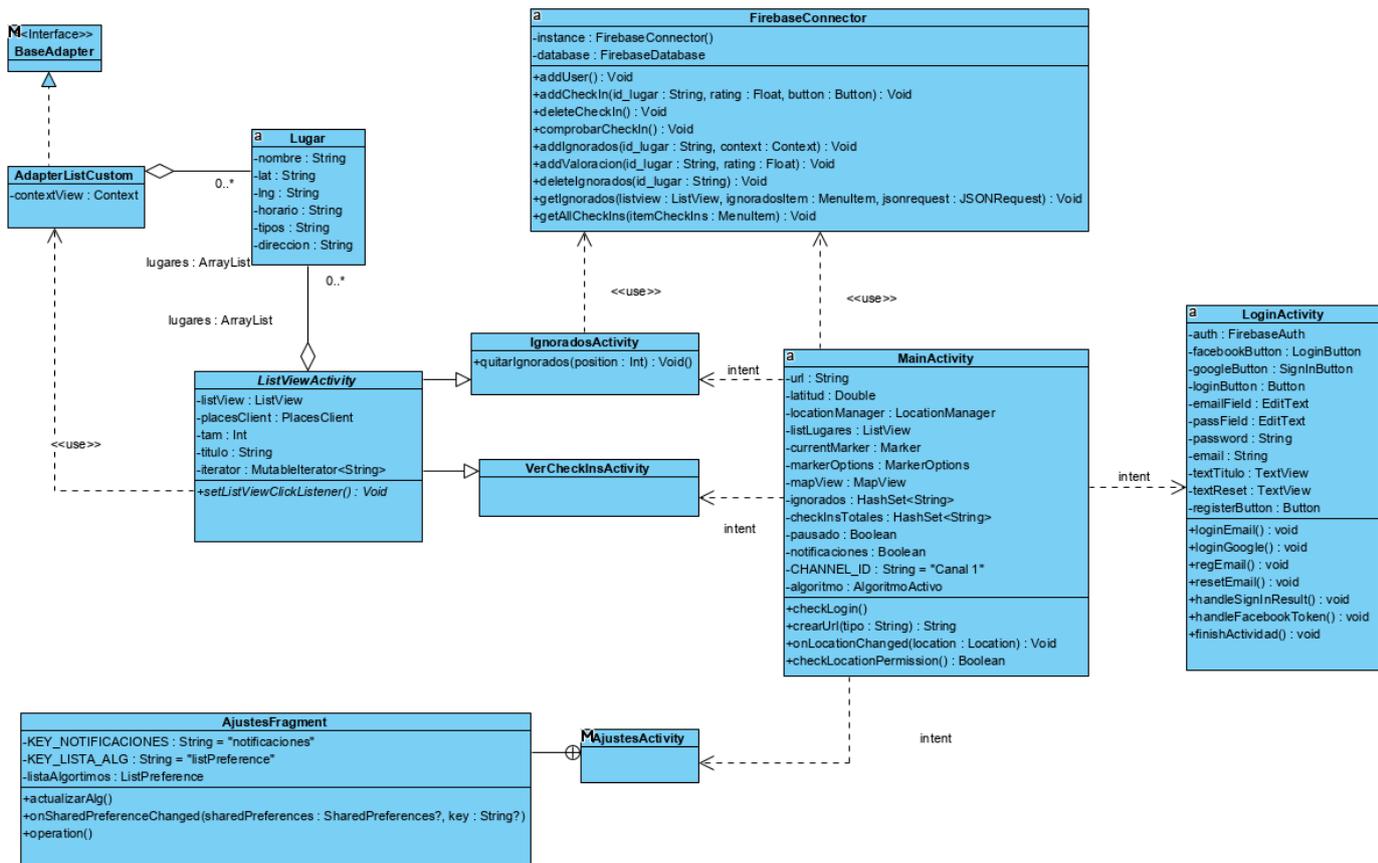


Figura 38: Diagrama de clases de la quinta iteración.

En este diagrama de clases de la figura 38 aparecen representadas las nuevas actividades que se lanzan vía *intent* desde el menú lateral. IgnoradosActivity y VerCheckInsActivity heredan ambas de la clase abstracta ListViewActivity. Esta clase utiliza AdapterListCustom, que implementa BaseAdapter, de forma similar a MainActivity. Tiene también una agregación de lugares necesaria para ambas actividades.

Por otra parte, la actividad de ajustes es AjustesActivity, compuesta por AjustesFragment que es la clase que contiene los métodos y variables para manejar las preferencias del usuario.

5.5.6 Diagrama de secuencia

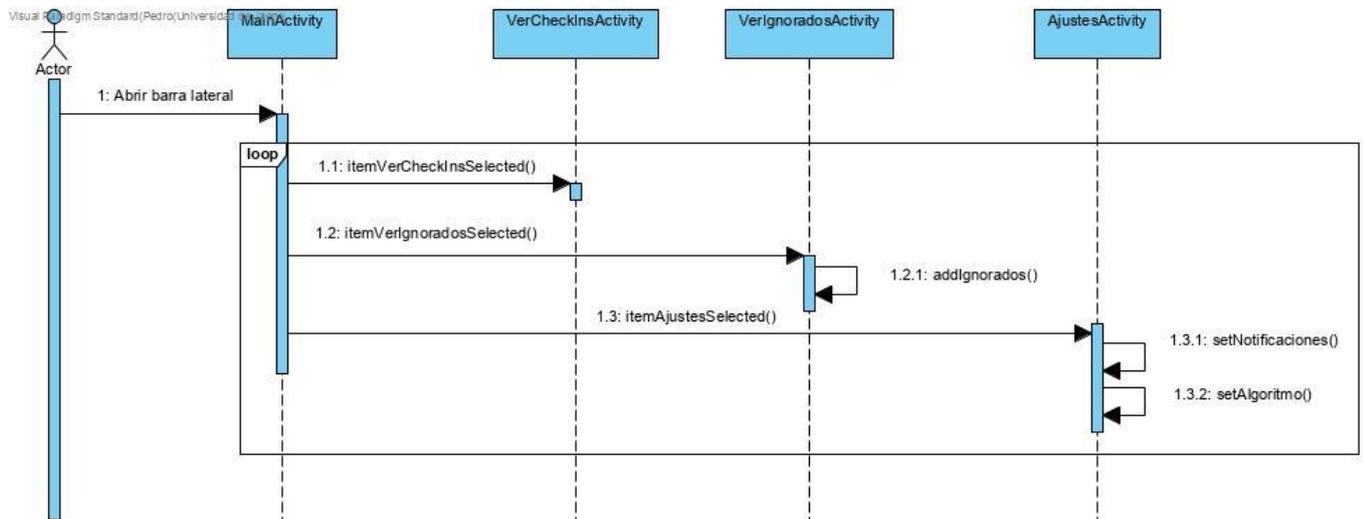


Figura 39: Diagrama de clases de la quinta iteración.

En este diagrama de secuencia, representado en la figura 39, se representa el acceso a las actividades del menú lateral y sus interacciones dentro del bloque *loop*.

5.6 Sexta iteración: Filtro de categorías y barra de búsqueda

Para la sexta iteración se implementarán filtros para personalizar los POIs mostrados en la lista según el usuario.

5.6.1 Análisis de sexta iteración

Es útil para el usuario el poder elegir qué tipos de lugares de interés se le muestran si sabe de antemano que actividad quiere realizar (filtrar por categorías) o, por ejemplo, desea visitar un restaurante de una franquicia conocida y quiere ver directamente cuales son los cercanos para comparar su valoración (filtrar por búsqueda de texto).

Para cada caso se deberá implementar distintos tipos de componentes de interfaz de Android: una lista desplegable o **Spinner** para elegir entre varias categorías de lugar y un campo de texto **EditText** para la búsqueda escrita. Ambas tienen en común el llamar a la función “performFiltering(Constraint : CharSequence)” del Adapter del ListView de lugares de la actividad principal, debiendo este implementar la interfaz **Filterable**.

5.6.2 Implementación de Filtro de Categorías

Puntos de historia: 8.

Para crear una lista desplegable que muestre las distintas categorías de un POI la mejor opción es utilizar el objeto Spinner. Sin embargo, su comportamiento por defecto no es suficiente, pues lo ideal es que el usuario pueda marcar y desmarcar casillas (CheckBox) por cada tipo de lugar que quiera mostrar u ocultar del ListView. Por ello, la mejor opción es crear un *adapter* personalizado mediante una clase que implemente **ArrayAdapter<>** y una clase con los atributos necesarios para representar cada objeto del Spinner: **ItemSpinner**. Esta clase contiene un String con el título del ítem y un Boolean que indica si su CheckBox está marcado o no. Para la parte gráfica de cada ítem se crea el fichero **spinner_item.xml** que consiste en un TextView con una CheckBox a la derecha.

En `main_activity.xml` se añade dentro del componente Toolbar un Spinner al lado derecho. Se crea la clase **AdapterSpinner** que implementa `ArrayAdapter<ItemSpinner>`. Se deben sobrecargar los métodos que detectan cuando se despliega la lista o se presiona sobre un ítem pasándoles la función `getCustomView()`, que implementa la lógica para actualizar el ListView de lugares según el parámetro "tipos" pasando al método `performFiltering()` un *string* con los tipos marcados en los distintos CheckBox separados con ";". Para ello, se debe crear una segunda estructura de datos que contendrá los lugares filtrados por las categorías seleccionadas por el usuario, que se crea a partir de la estructura total de lugares. Se modifica el método `getView()` del *adapter* del ListView para que se base en esta segunda estructura de datos llamada **"listaLugaresFiltradosCat"**. La lista de categorías a filtrar se define en el fichero **strings.xml** en la carpeta `res/values`. Este archivo es inherente a cada proyecto de Android Studio y su objetivo es colocar ahí todas las cadenas de texto estáticas que se usarán en la app (por ejemplo, para el texto de cada TextView) evitando así el cada *string* codificado ("hard-coded") y facilitando la traducción de la app a otros idiomas.

```

<string-array name="categorias_array">
  <item>Categorías</item>
  <item>Bar</item>
  <item>Restaurante</item>
  <item>Tienda</item>
  <item>Discoteca</item>
  <item>Cine</item>
  <item>Museo</item>
</string-array>

```

En la figura 40 vemos como la lista desplegable en la app.

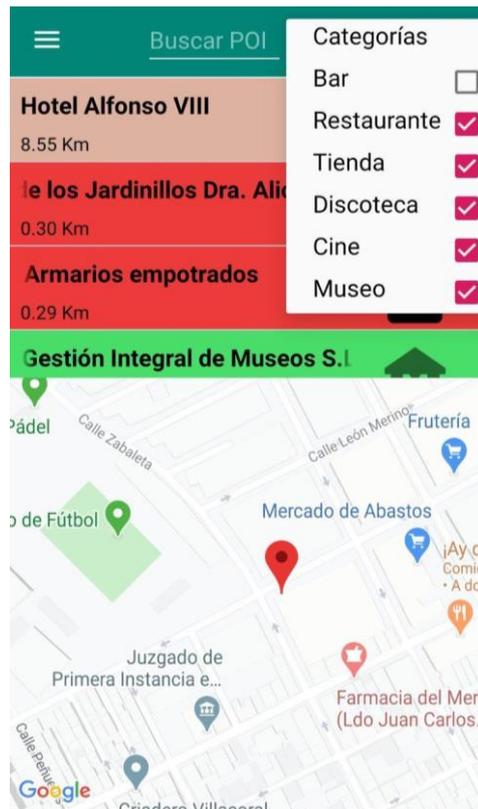


Figura 40: Filtro de categorías.

5.6.3 Implementación de Filtro de Búsqueda

Puntos de historia: 8.

Para simular el comportamiento de una típica barra de búsqueda, se coloca un componente EditText en el centro de la Toolbar de MainActivity. Se le añade un *listener* para que cada vez que el texto cambia se llame a “performFiltering()”, mandando el texto con el prefijo “Búsqueda;”, distinguiéndolo así de una llamada desde el Spinner de categorías. El filtro devuelve solo los lugares que contengan la sucesión de caracteres que se le pasan. Como esta filtración es compatible con la

de categorías, se debe crear una tercera estructura de datos en la que se apoya `AdapterListview` para mostrar las filas de lugares: **“listaLugaresFiltradosFinal”**.

Por lo tanto, cuando al filtrar por categorías se itera sobre **“listaLugares”**, que contiene todos los obtenidos mediante la API, y se almacenan en **“listaLugaresFiltradosCat”** y **“listaLugaresFiltradosFinal”**; y al filtrar por búsqueda de texto se itera sobre **“listaLugaresFiltradosCat”** y almacena en **“listaLugaresFiltradosFinal”**.

Se ve el uso correcto de este filtro en la figura 41

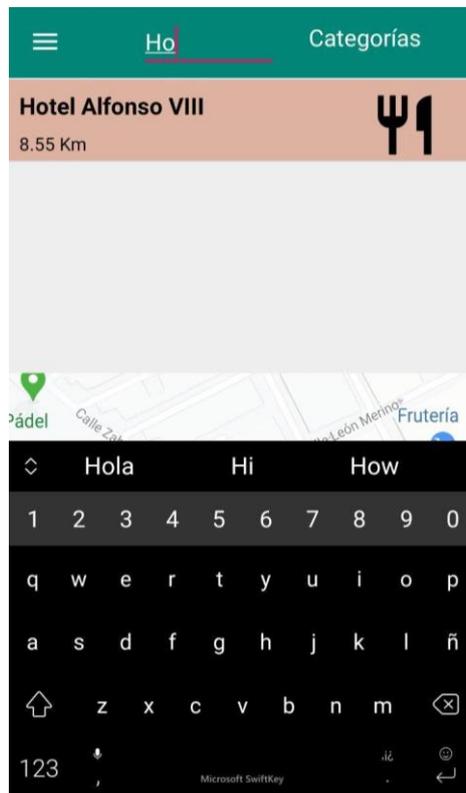


Figura 41: Filtro de búsqueda.

```

override fun performFiltering(constraint: CharSequence): FilterResults? {
    var constraint = constraint
    val results = FilterResults()
    var FilteredArrayNames = ArrayList<Lugar>()

    if(constraint.startsWith("Búsqueda;")){
        val valor = constraint.split(";")[1].toLowerCase()
        for(lugar in listaLugaresFiltradosCat!!){
            val nombre = lugar.nombre
            if(nombre.toLowerCase().contains(valor)){
                FilteredArrayNames.add(lugar)
            }
        }
    }
    }else{
        constraint = constraint.toString()
        val listConstraint = constraint.split(";")
        for (lugarActual in listaLugares) {
            val tipos: String = lugarActual.tipos
            for(constraintInd in listConstraint){
                if (tipos.contains(constraintInd)) {
                    FilteredArrayNames.add(lugarActual)
                    break
                }
            }
        }
        listaLugaresFiltradosCat = ArrayList(FilteredArrayNames)
    }
    results.count = FilteredArrayNames.size
    results.values = FilteredArrayNames
    return results
}
    
```

5.6.4 Diagrama de clases

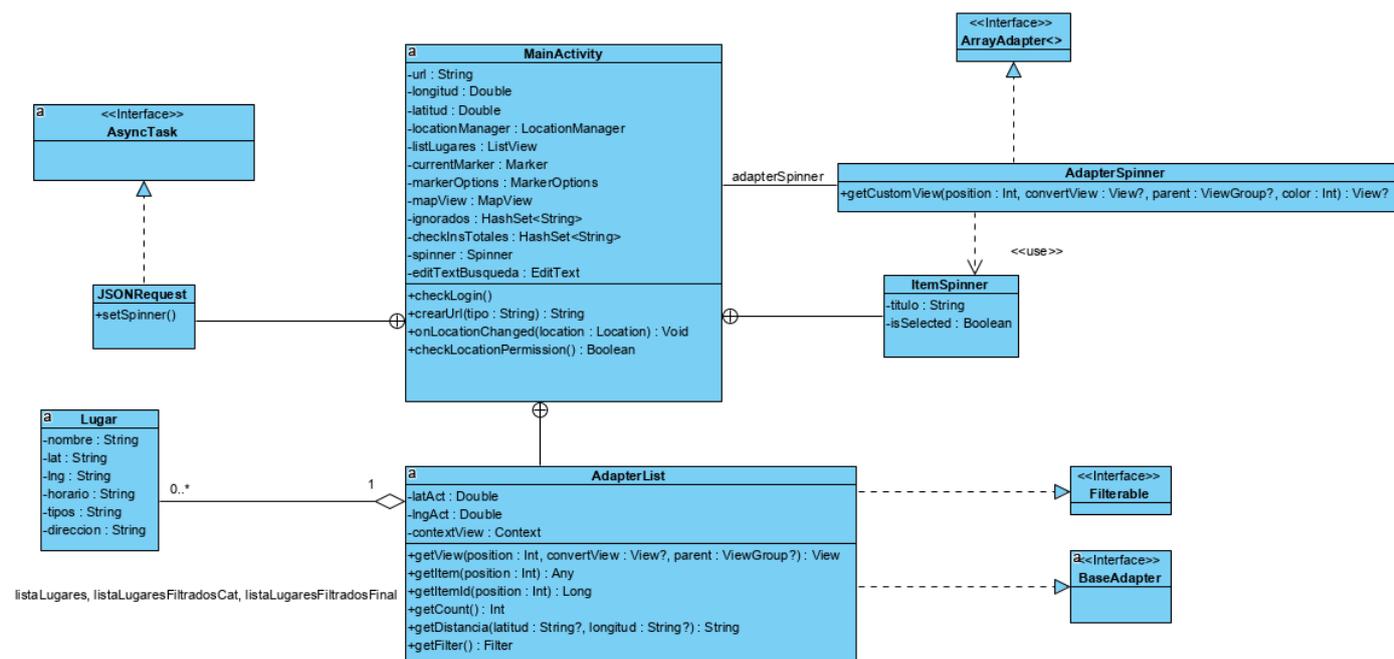


Figura 42: Diagrama de clases de la sexta iteración.

En este diagrama de clases de la figura 42 vemos representado el Spinner mediante la clase AdapterSpinner, que implementa ArrayAdapter, e ItemSpinner, que contiene las variables para representar una fila: el título y el estado de selección. También se ha añadido la implementación de Filterable a AdapterList, que realiza el filtro mediante “getFilter()” sobre el array de lugares tanto de categoría con el Spinner como de texto con el EditText (“editTextBusqueda” en MainActivity).

5.6.5 Diagrama de secuencia

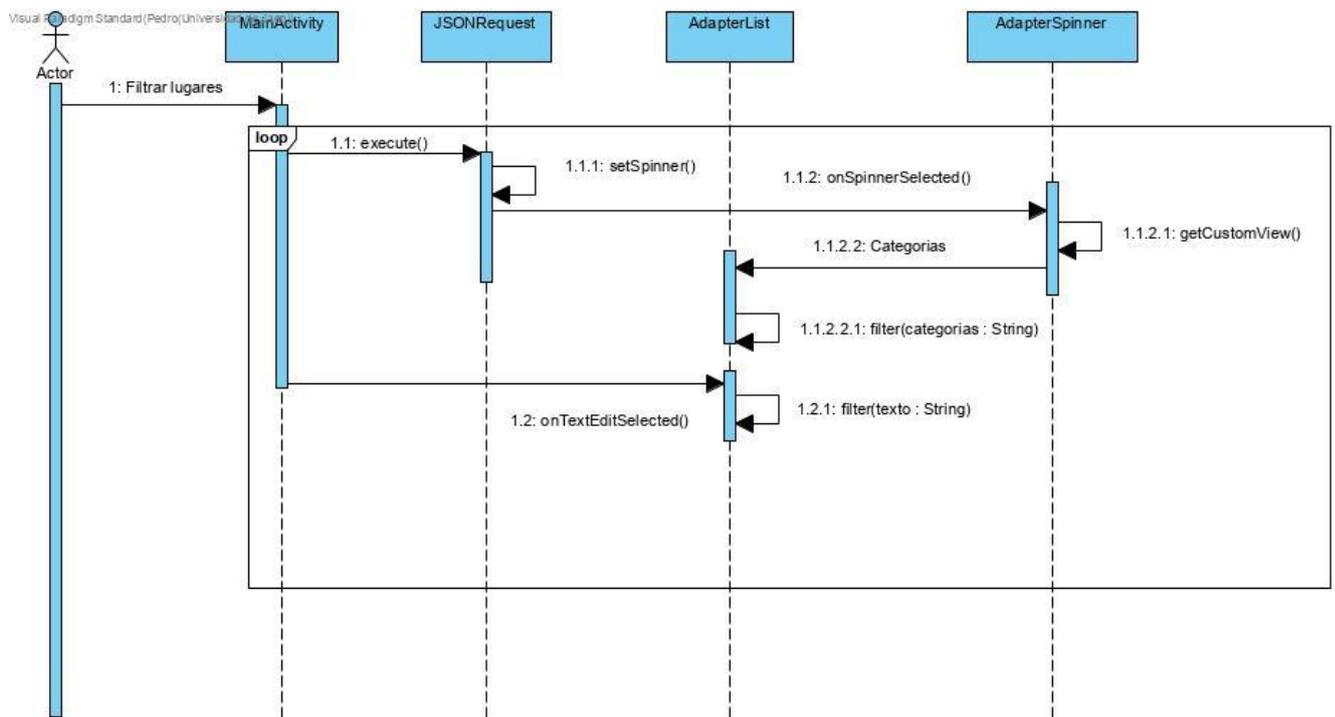


Figura 43: Diagrama de clases de la sexta iteración.

Dentro del bloque *loop* del diagrama de secuencia de la figura 43 se representan las dos operaciones de filtro del ListView de lugares desde MainActivity. Al ejecutar JsonRequest, esta clase se encarga de configurar el Spinner, de modo que al seleccionar un ítem se llama a AdapterSpinner. Esta clase manda a AdapterList la lista de categorías de lugares a filtrar, acción que realiza mediante el método “filter()”.

En el caso del filtrado por nombre, el *listener* del EditText en MainActivity directamente llama al método “filter()” de AdapterList con el texto introducido por el usuario.

5.7 Séptima iteración: Sistema de perfil y amigos

En la séptima iteración se creará un sistema de perfiles de usuario, que permita mostrar datos personales, registro de Check-Ins y foto de perfil. También se permitirá buscar otros usuarios de la app y agregarlos a una lista de amigos.

5.7.1 Análisis de séptima iteración

Los algoritmos de recomendación suelen basarse en la premisa de que dos usuarios con una relación de amistad suelen tener gustos similares, por lo que se puede analizar las preferencias de uno para predecir recomendaciones para el otro. Así, se hace necesario un sistema de amigos dentro de la base de datos de la aplicación con el que alimentar los algoritmos de filtrado colaborativo, así como añadir un aspecto social y de personalización.

Para ello se debe crear primero un perfil por cada usuario de la app que contenga más información que la básica (email y contraseña): nombre de usuario o alias, ubicación, fecha de registro, imagen de perfil y últimos “Check-Ins” realizados. Hay que redefinir la pantalla de registro para tener en cuenta estos nuevos datos, así como ser llamada incluso si el usuario decide iniciar sesión por primera vez usando Google o Facebook para obtener la información adicional. El usuario debe poder ver su perfil y modificar algunos campos, así como ver el de otros usuarios. Para ello accederá desde el menú lateral a una pantalla con el listado de amigos agregados y un botón para buscar de la base de datos total de usuarios otros posibles amigos que agregar. También hay que tener en cuenta las preferencias de privacidad del usuario, que podría no querer que ciertos datos como la ubicación o el correo electrónico sean visibles a otros usuarios.

5.7.2 Implementación de Registro de perfil

Puntos de historia: 13.

Se crea el fichero **activity_registro.xml**, para la nueva actividad **RegistroActivity**, que contiene los campos de texto o EditText necesarios para obtener los datos mediante el usuario, así como un ImageView con una imagen de perfil genérica que al pulsarlo abre mediante **Intent** el gestor de archivos del smartphone para que el usuario pueda elegir su propia imagen. Realtime Database no permite almacenar imágenes, solo cadenas de caracteres. Sin embargo, Firebase ofrece su servicio

Storage para este fin. Se crea en la raíz de Storage la carpeta “avatares” donde se almacenan las imágenes de perfil de todos los usuarios, siguiendo el formato de nombre “uid” + “.jpg”. Antes de subir la imagen se verifica que está en formato “.jpg” y que su tamaño no excede el máximo establecido en la variable “MAX_BYTES” de 5 MB.

```
uri = data.data
val afd = contentResolver.openAssetFileDescriptor(uri, "r");
val sizeFile = afd.length
val cr = this.contentResolver

val ext = cr.getType(uri)
val extension = ext.substring(ext.lastIndexOf("/"), ext.length)
if(extension == "/jpg" || extension == "/jpeg") {
    if(sizeFile <= FirebaseConnector.MAX_BYTES){
        avatar.setImageURI(uri)
    }else{
        Toast.makeText(this, "Error: Seleccione una imagen con tamaño menor a 5 MB.", Toast.LENGTH_LONG).show()
        uri = null
    }
}
}else{
    Toast.makeText(this, "Error: Seleccione una imagen en formato JPG.", Toast.LENGTH_LONG).show()
    uri = null
}
```

Existen varios métodos de subir a Storage una imagen, entre los que se ha optado por convertir la imagen a un array de bytes representado en Kotlin mediante la clase **ByteArray**. Se debe crear la clase **Usuario**, que implementa **Parcelable**, para almacenar toda la información del usuario de la app y los posibles amigos.



Figura 44: Pantalla de registro.

La pantalla de registro, en la figura 44 aparecerá al pulsar el botón de “Registro” de la pantalla de login o al iniciar sesión por primera vez con Facebook o Google, apareciendo en cualquiera de los dos últimos casos desactivados los campos de email y contraseña. Tras pulsar el botón “Registrarse” y verificar varias cosas como que el alias no esté ya en uso o coincidan las contraseñas, se guarda en Firebase la información de usuario en el nodo hijo “Datos” de “Usuarios/uid”. Se modifica la estructura de la base de datos para incluir por cada usuario su lista de amigos, sus datos (y los de cada amigo) y su lista de últimos cinco “Check-Ins”, creando los métodos necesarios en FirebaseConnector para ello. La nueva estructura de la BD queda reflejada en la figura 45.



Figura 45: Estructura de usuarios en Realtime Database.

5.7.3 Implementación de Vista de perfil

Puntos de historia: 8.

Se añade al menú de la barra lateral un nuevo ítem “Mi perfil” que carga la actividad **PerfilActivity**, donde el usuario puede ver su información de perfil que otros también pueden ver al buscarle. Mediante la interfaz definida en **activity_perfil.xml** se muestra la información del usuario a través de TextView, Imageview y, ocupando la mitad inferior de la pantalla, un ListView con los últimos 5 “Check-Ins”.

Esta clase se utiliza tanto para que el usuario consulte su perfil como para cargar el perfil de otros usuarios, por lo que durante la creación de la actividad se debe comprobar si el Id del usuario coincide con el de la sesión de FirebaseAuth, adaptando la interfaz y funciones para ello. Por ejemplo, si es el perfil del usuario de

la app se adapta el ImageView de la imagen de perfil aplicándole un filtro de blanco y negro para indicar visualmente al usuario que puede cambiar la imagen pulsando sobre ella. También el TextView correspondiente a la ubicación se sustituye por un EditText que permite escribir una nueva y guardar en la base de datos pulsando el botón “Guardar Cambios”, que en la vista de perfil de otro usuario es cambiado por un botón que permite agregar o quitar a la lista de amigos.

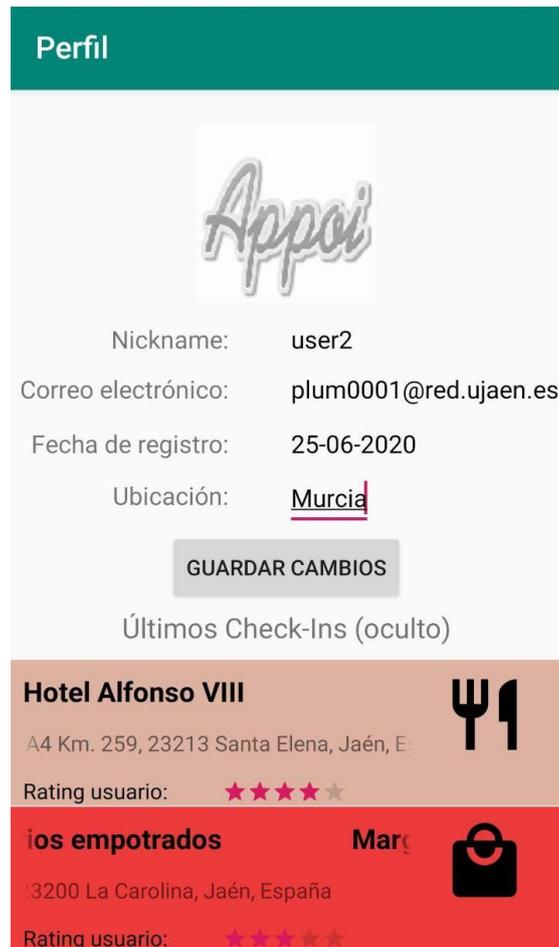


Figura 46: Vista de perfil del usuario de la app.

Como se aprecia en la Figura 46, el ListView de “Check-Ins” está modificado para mostrar directamente la valoración del lugar por el usuario, así como la dirección en lugar de la distancia.

Por último, hay que modificar la pantalla de ajustes para incluir opciones de privacidad que permitan mostrar u ocultar cierta información de perfil al resto: correo electrónico, ubicación y últimos “Check-Ins”. Para ello se añaden nuevos atributos a los nodos de datos de un “Usuario” en Realtime Database: prefEmail, prefUbicacion

y `prefCheckIns`. Tendrán valores binarios que indicarán con un 1 que quieren que se muestre de forma pública la información y 0 que aparezca como oculta.

En `fragment_ajustes.xml` se añaden tres `SwitchPreference` para cada opción y se modifica la clase `Usuario` para incluir estos nuevos tres valores, como se aprecia en la figura 47.

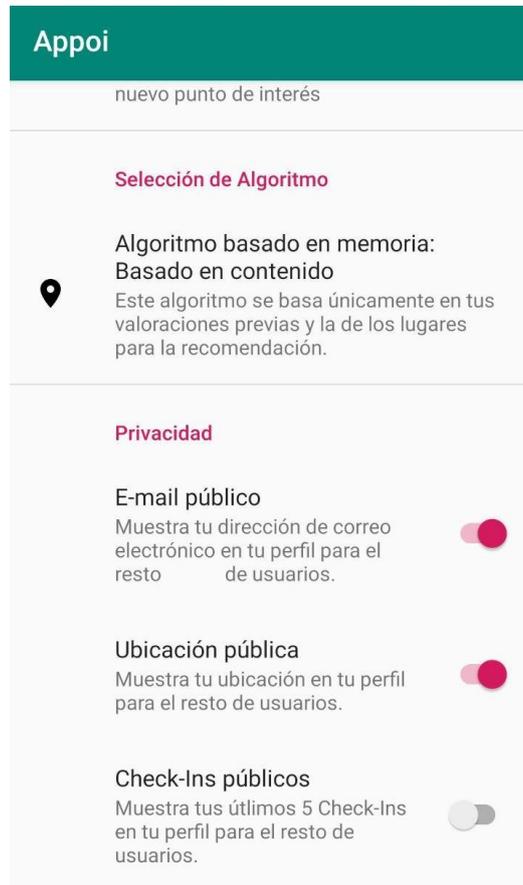


Figura 47: Vista de ajustes de privacidad.

5.7.4 Implementación de Vista de lista de amigos

Puntos de historia: 13.

Se crea el ítem “Ver amigos” en el menú lateral que carga la actividad **AmigosActivity**. Dicha actividad consiste en único `ListView` que muestra cada usuario registrado que ha sido añadido como “amigo”. El `ListView` carga como interfaz de fila el fichero `row_list_amigos.xml` que contiene datos de alias e imagen de perfil.

Además, se modifica la barra superior para incluir una barra de búsqueda que filtra los amigos por el alias y un botón “+” para pasar a la pantalla de buscar y agregar amigos, tal y como muestra la figura 48.

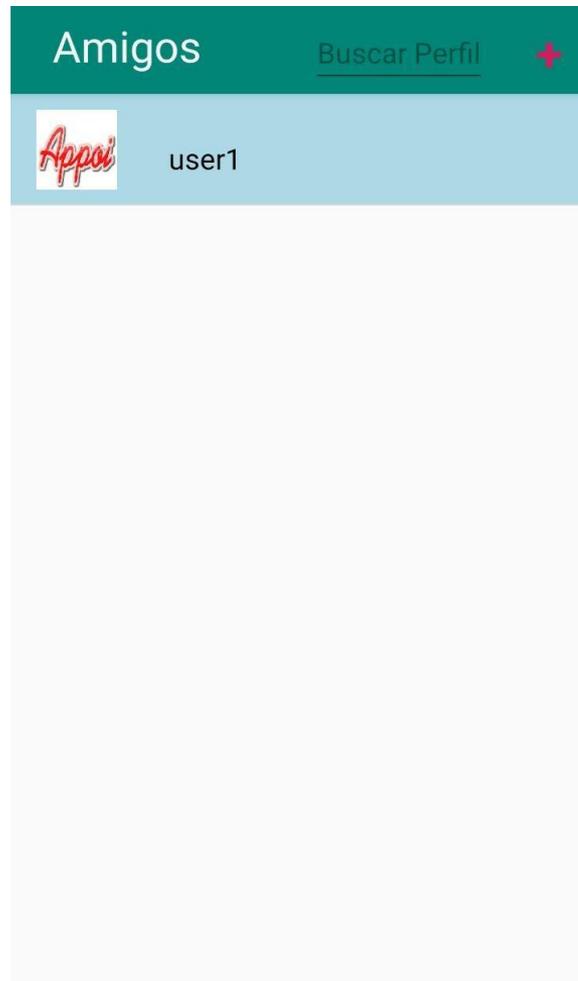


Figura 48: Vista de amigos.

Al pulsar sobre una fila se abre la actividad PerfilActivity explicada en la anterior implementación, pero adaptándose con la información del usuario amigo y sin permitir modificar datos, sustituyendo el botón de confirmar cambios por el de agregar/quitar de la lista de amigos, como vemos en la figura 49.

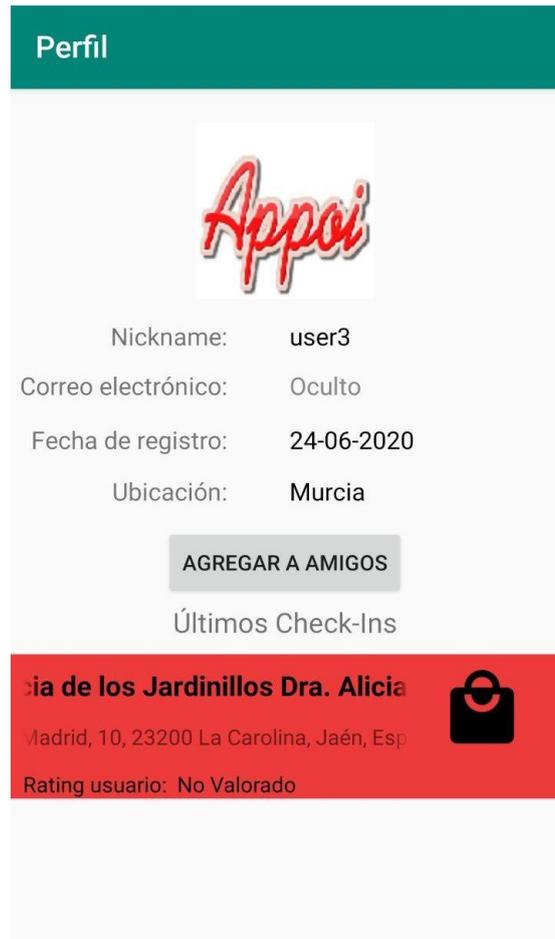


Figura 49: Vista de perfil de otro usuario.

Al pulsar el botón superior izquierdo “+” se abre la actividad **BuscarAmigosActivity**, mostrada en la figura 50. Esta actividad reutiliza la misma interfaz que AmigosActivity, pero esta vez se cargan en el ListView todos los usuarios de la base de datos. Además, la fila muestra ahora en la parte derecha un botón “Agregar a Amigos” que permite directamente agregar al usuario sin necesidad de introducirse en el perfil.

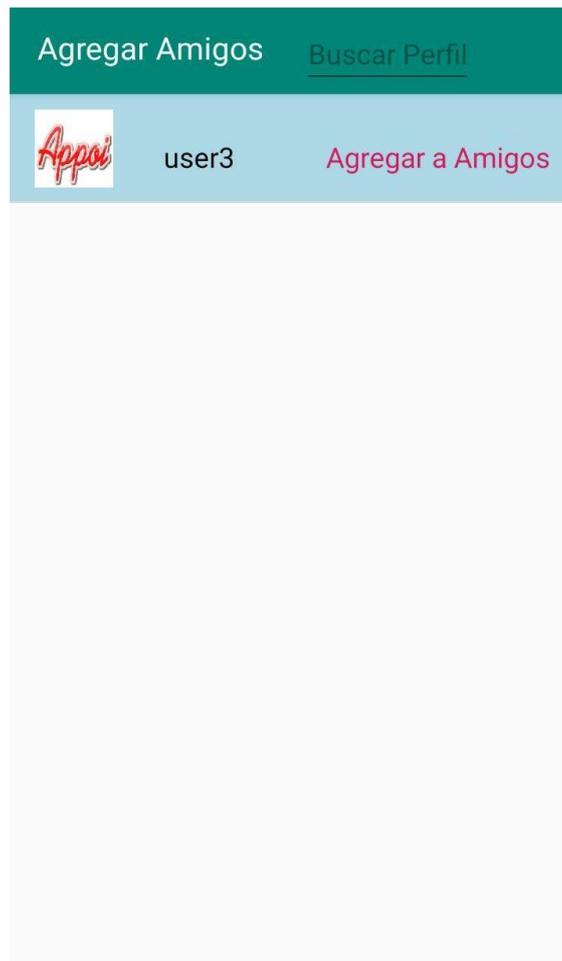


Figura 50: Vista de búsqueda de nuevos amigos.

Cada vez que se inicia la app se descarga la lista de usuarios amigos de Firebase y se almacenan en un ArrayList de Usuarios global en MainActivity llamado “**amigos**”. Este se utiliza para cargar la lista de amigos y para no mostrarlos en la lista de usuarios total de BuscarAmigosActivity.

5.7.5 Diagrama de clases

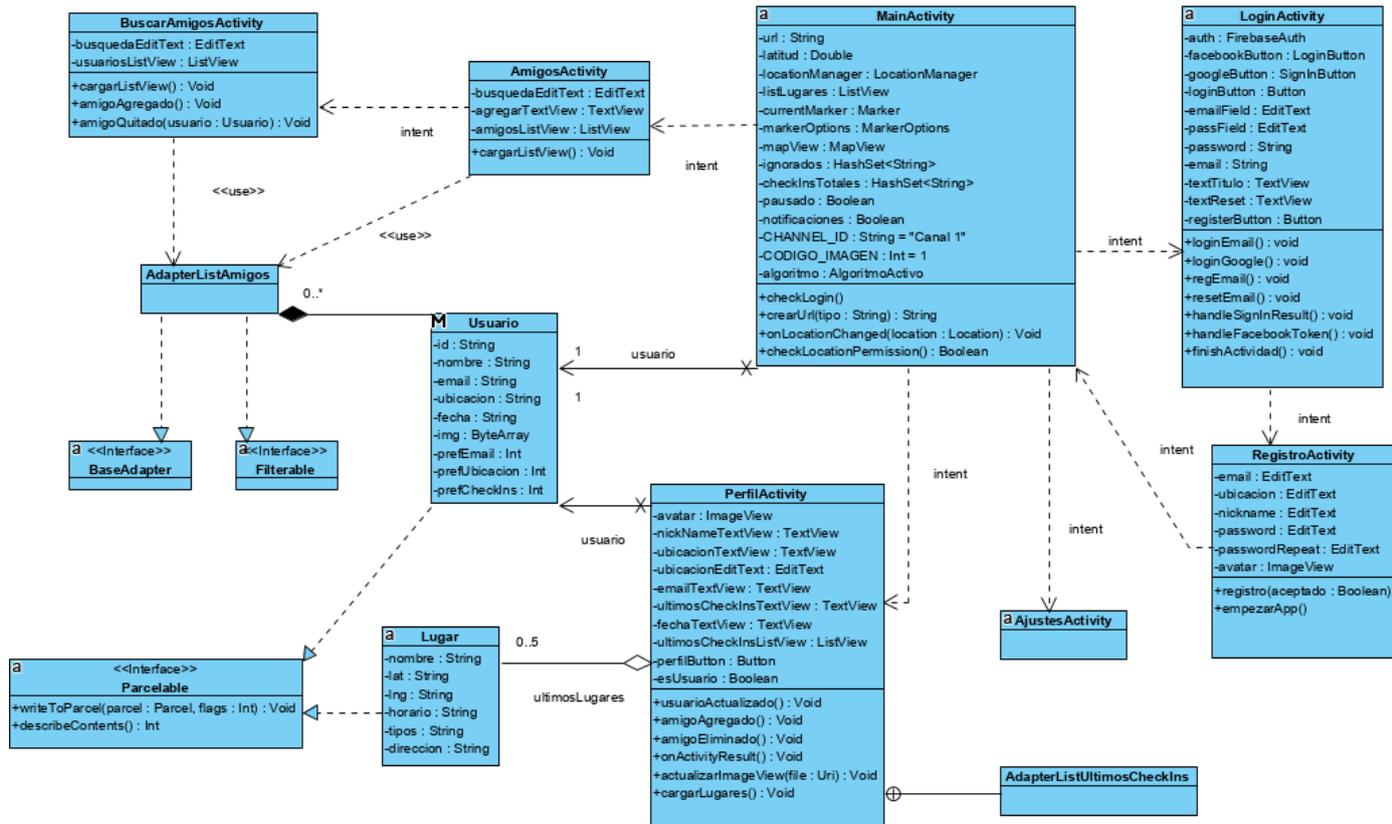


Figura 51: diagrama de clases de la séptima iteración.

En el diagrama de clases de esta iteración, en la figura 51, se añade la clase RegistroActivity que ahora se lanzará desde LoginActivity al registrarse en la app por primera vez mediante cualquiera de los tres métodos. La nueva clase PerfilActivity contiene las variables para los datos de perfil tanto del usuario usando la app como de otros, así como los metodos para actualizar el perfil, añadir/quitar un amigo, etc. Utiliza su propio ListView con un *adapter* personalizado, AdapterListUltimosCheckIns.

Por otra parte, tenemos las clases AmigosActivity y BuscarAmigosActivity que, utilizan la la clase AdapterListAmigos para mostrar la lista de perfiles. Esta última clase implementa, además de BaseAdapter, Filterable para filtrar por nombre de usuario. También se ha creado la clase Usuario para representar los datos de un usuario de la app, de modo que AdapterListAmigos está compuesta por un array de estos y MainActivity tiene uno que es el usuario de la aplicación.

5.7.6 Diagrama de secuencia

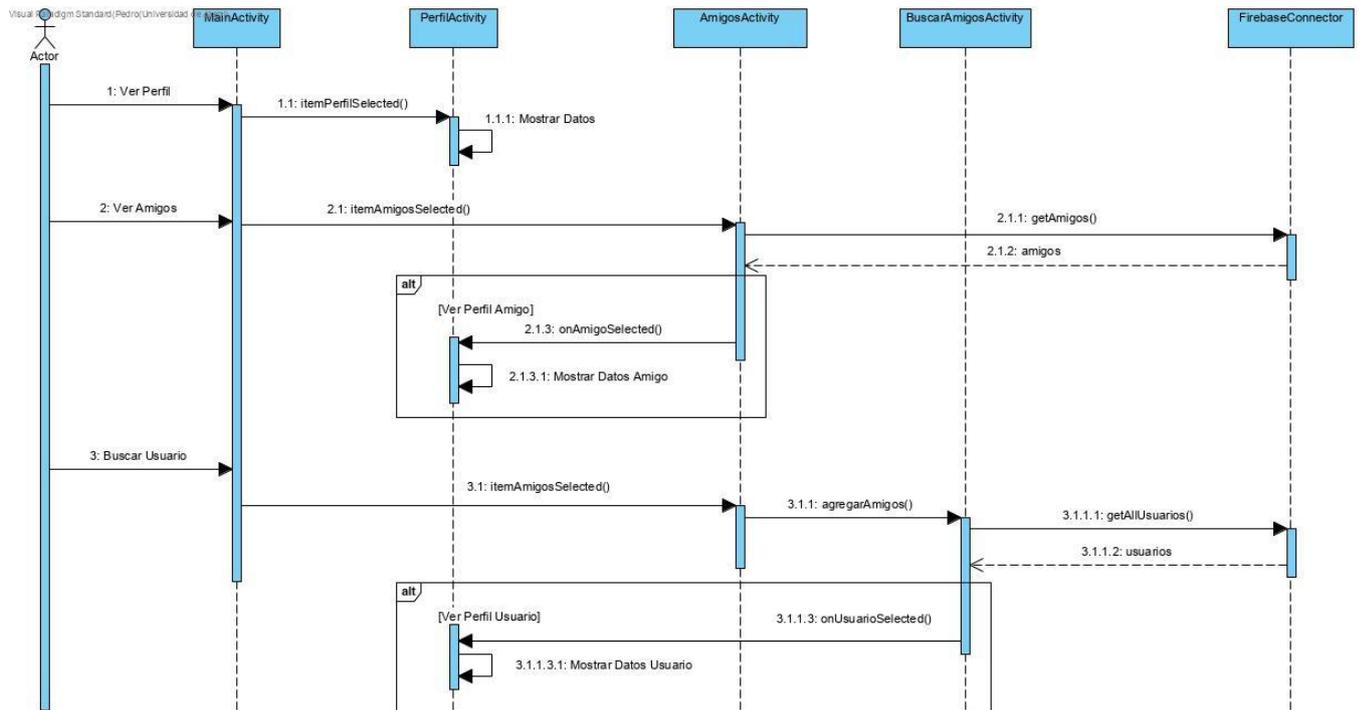


Figura 52: diagrama de secuencia de la séptima iteración.

En la figura 52 vemos el diagrama de secuencia, mostrando las 3 acciones principales de esta iteración. Para que el usuario vea su perfil simplemente se llama a PerfilActivity. Para ver la lista de amigos, AmigosActivity pide primero a FirebaseConnector la lista de amigos, y una vez obtenido el array se muestran en el ListView. Finalmente, para buscar otros usuarios en la BD para añadirlos a Amigos, se llama a BuscarAmigosActivity desde AmigosActivity, que pide a FirebaseConnector la lista de usuarios no agregados a amigos.

5.8 Octava iteración: Implementación de algoritmos de recomendación

Para la última iteración se deben codificar los algoritmos de filtrado, que son el eje principal de este proyecto.

5.8.1 Análisis de octava iteración

Finalmente llega el momento de implementar los distintos algoritmos de recomendación que ordenarán la lista de puntos de interés según una predicción del “rating” del usuario para los lugares no visitados. La forma de hacerlo para cada uno es, una vez obtenida la lista de lugares cercanos, aplicarle el algoritmo

correspondiente al elegido en la pantalla de ajustes, asignarle a cada uno la puntuación calculada y ordenar el ListView según dicho parámetro.

Se implementan tres algoritmos de filtrado colaborativo: basado en usuario, basado en lugar e híbrido, que es una combinación de los dos anteriores. Como alternativa al filtrado colaborativo, se dará la opción de usar un algoritmo basado en contenido: TF-IDF.

5.8.2 Implementación de Algoritmo de Filtrado Colaborativo basado en usuario

Puntos de historia: 18.

Para indicar el nivel de posible afinidad al usuario de un POI cercano se añaden un TextView a la vista de la fila del ListView principal `row_list_main.xml`. Precedido del título "Puntuación: ", este texto indica una cifra porcentual de 0 a 100 que cambia de color según el número: rojo de 0 a 25; naranja de 25 a 50; amarillo de 50 a 75; y verde de 75 a 100. Vemos la nueva fila en la figura 53.



Figura 53: Fila con la información de afinidad calculada mediante algoritmo.

Para implementar el algoritmo de Filtrado Colaborativo es necesario apoyarse en la el **coeficiente de correlación de Pearson** [54]. Esta fórmula, mostrada en la figura 54, sirve para calcular el grado de similitud, entre -1 y 1, entre dos usuarios en base a las puntuaciones que han dado de lugares comunes. El conjunto n indica el total de lugares comunes evaluados, x_i e y_i la puntuación al lugar i del usuario x e y respectivamente, \bar{x} e \bar{y} es la media total de puntuaciones dadas por el usuario x e y respectivamente.

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Figura 54: coeficiente de correlación de Pearson.

Se crean las funciones y estructuras de datos necesarias para obtener los usuarios que, para un lugar dado que no ha sido puntuado por el usuario de la app, hayan

puntuado dicho lugar y al menos un lugar que también lo haya hecho el usuario. Se dan preferencia a los usuarios registrados en la base de datos como **amigos** si cumplen las condiciones. Se calcula el sumatorio de la correlación de Pearson total para todos los usuarios obtenidos, así como el mismo sumatorio, pero multiplicándolo por el rating del usuario dado al lugar objetivo.

```

fun PearsonCorrelationUserBased(lugaresU : ArrayList<FirebaseConnector.Valoracion>,
                                lugaresV : ArrayList<FirebaseConnector.Valoracion>)
{
    var sim = 0.0
    var sumatorio = 0f
    // Cálculo de media de usuario U
    for(lugar in lugaresU){
        sumatorio += lugar.rating!!
    }
    val uMedia = sumatorio/lugaresU.size

    sumatorio = 0f
    // Cálculo de media de usuario V
    for(lugar in lugaresV){
        sumatorio += lugar.rating!!
    }
    val vMedia = sumatorio/lugaresV.size

    var pos = 0
    var nominador : Double = 0.0

    // Cálculo del nominador
    for(lugaru in lugaresU){
        val lugarv = lugaresV.get(pos)
        val ru = lugaru.rating!!
        val rv = lugarv.rating!!

        nominador += ((ru-uMedia)*(rv-vMedia))

        pos++
    }

    pos = 0

    // Cálculo del primer operador del denominador
    var denominador1 : Double = 0.0
    for(lugaru in lugaresU){
        val lugarv = lugaresV.get(pos)
        val ru = lugaru.rating!!

        denominador1 += Math.pow((ru-uMedia).toDouble(),2.0)

        pos++
    }

    pos = 0

    // Cálculo del segundo operador del denominador
    var denominador2 : Double = 0.0
    for(lugaru in lugaresU){
        val lugarv = lugaresV.get(pos)
        val rv = lugarv.rating!!

        denominador2 += Math.pow((rv-vMedia).toDouble(),2.0)

        pos++
    }
}

```

```

val denominador = Math.sqrt(denominador1*denominador2)

    sim = nominador/denominador

    sumatorioTotalCF += (sim*
mapaUsuariosCF.values.toList()[contadorCFUsuarios])
    sumatorioSimCF += sim

    contadorCFUsuarios++

userBasedCF()
}

```

Una vez obtenido el ambos sumatorios, se utiliza el primero para calcular un factor de normalización k de la siguiente manera: $k = 1 / \sum_u^u |sim(u, u')|$. A continuación, se multiplica este factor por el segundo sumatorio, obteniendo el rating predictivo para el usuario de la app para un lugar. Este proceso se repite para cada lugar, siendo necesario recalculer los usuarios a utilizar para la correlación de Pearson.

Finalmente, se le asigna al nuevo atributo “puntuación : Double” de la clase Lugar el rating calculado y se muestra en el ListView en valor porcentual y ordenando las filas en orden descendente.

5.8.3 Implementación de Algoritmo de Filtrado Colaborativo basado en lugar

Puntos de historia: 18.

A diferencia del basado en usuario, el FC basado en ítem (lugar en este caso) se centra en calcular la similitud entre varios ítems y el ítem objetivo en lugar de entre los usuarios. Se comienza calculando una lista de hasta 20 mejores lugares valorados por el usuario objetivo. A continuación, para cada lugar a predecir, se obtiene para cada lugar de los mejores valorados una lista de otros usuarios que han valorado el lugar objetivo y el lugar valorado. Para cada una de estas iteraciones se rellenan tres mapas con el id del usuario como claven en todos y con distintos valores: la valoración del lugar objetivo actual, la valoración del lugar valorado actual y la media total de valoraciones.

Estas estructuras de datos se utilizan en la función de similitud de este algoritmo, que en este caso no será la correlación de Pearson, sino la **similitud coseno**. Esta función trata a cada ítem como un vector en el espacio, calculando el coseno del ángulo entre dos ítems [55]. Esto da un valor comprendido en el intervalo cerrado $[-1, 1]$, siendo 1 que ambos vectores tienen la misma orientación (ángulo 0); o un valor

menor que 1 si existe ángulo. La razón por la que se ha decidido variar a esta función de similitud es porque algunos estudios [56] sugieren que es más precisa en la recomendación basada en Pearson, además de aportar variedad al proyecto.

La fórmula común de la similitud coseno, no tiene en cuenta el comportamiento optimista de los usuarios, por eso en esta implementación se aplica la siguiente variante, en la figura 55, que resta la media de valoraciones del usuario a cada operando del nominador [57]. Siendo u un usuario del conjunto U , $r_{(u,i)}$ la valoración del usuario u al ítem i , $r_{(u,j)}$ la valoración del usuario u al ítem j y \bar{r}_u la media de valoraciones del usuario u :

$$\text{similarity}(i, j) = \frac{\sum_u (r_{(u,i)} - \bar{r}_u)(r_{(u,j)} - \bar{r}_u)}{\sqrt{\sum_u r_{(u,i)}^2} \sqrt{\sum_u r_{(u,j)}^2}}$$

Figura 55: similitud coseno ajustada.

```

fun similitudCoseno(idLugar : String, mapaUsuariosObjetivo : HashMap<String,
Float>,
                    mapaUsuariosValoracion : HashMap<String, Float>,
                    mapaUsuariosMedia : HashMap<String, Float>) {
    val iterador = mapaUsuariosObjetivo.iterator()
    var sumatorioNom = 0.0
    var sumatorioDen1 = 0.0
    var sumatorioDen2 = 0.0

    while(iterador.hasNext()) {
        val usuarioAct = iterador.next().key

        val mediaUsuarioAct = mapaUsuariosMedia.get(usuarioAct)!!
        val rui = mapaUsuariosObjetivo.get(usuarioAct)!!
        val ruj = mapaUsuariosValoracion.get(usuarioAct)!!

        // Nominador
        val nom = (rui-mediaUsuarioAct)*(ruj-mediaUsuarioAct)
        sumatorioNom += nom

        // Denominador 1
        val den1 = Math.pow((rui-mediaUsuarioAct).toDouble(),2.0)
        sumatorioDen1 += den1

        // Denominador 2
        val den2 = Math.pow((ruj-mediaUsuarioAct).toDouble(),2.0)
        sumatorioDen2 += den2

    }

    val denominador = Math.sqrt(sumatorioDen1*sumatorioDen2)

    val similitud = sumatorioNom/denominador

    sumatorioTotalCF += (similitud * mapaMejoresLugares.get(idLugar)!!)
    sumatorioSimCF += similitud.absoluteValue

    itemBasedCF()
}

```

Como en el caso de basado en usuario, al terminar se asigna al atributo “puntuación” de cada lugar objetivo un valor porcentual.

5.8.4 Implementación de Algoritmo de Filtrado Colaborativo híbrido

Puntos de historia: 13.

En el algoritmo de FC híbrido se aplica primero el basado en usuario y después el basado en ítem, almacenando en estructuras de datos las predicciones de puntuaciones para cada lugar y algoritmo. Al final, se itera sobre cada lugar y como puntuación final se asigna la media de ambas puntuaciones.

```

val it = lugares.iterator()
while(it.hasNext()){
    val lugar = it.next()
    val id = lugar.id
    val puntuacion1 = ParserPlaces.mapaUserBasedHibrido.get(id)
    val puntuacion2 = ParserPlaces.mapaItemBasedHibrido.get(id)
    // Cálculo puntuación híbrida
    val puntuacionFinal = (puntuacion1!! + puntuacion2!!)/2

    lugar.puntuacion = puntuacionFinal
}

```

5.8.5 Implementación de Algoritmo de Filtrado basado en contenido: TF-IDF

Puntos de historia: 13.

Al comienzo de este algoritmo, se obtiene un ArrayList con los 20 lugares mejor puntuados por el usuario. A partir de ahí, se calcula un mapa (HashMap) del valor **IDF** de cada término presente en el nombre y categorías de cada lugar. El IDF o frecuencia inversa de documento es un valor que determina la importancia de un término concreto con respecto una colección de documentos. Dada la cardinalidad de la colección de textos N_D y el número de documentos de dicha colección en los que aparece el término t , f_t , el valor IDF de t se calcula de la manera mostrada en la figura 56.

$$IDF_t = \log \left(1 + \frac{N_D}{f_t} \right)$$

Figura 56: cálculo de IDF.

Es importante no incluir palabras como preposiciones o determinantes que son comunes y no aportan información relevante mediante un array de descartes incluido en el fichero strings.xml.

```

<string-array name="palabras_comunes">
    <item>el</item>
    <item>la</item>
    <item>de</item>
    <item>los</item>
    <item>las</item>
    <item>en</item>
</string-array>

```

```

fun calcularIdf(){
    val tam = mejoresLugares.size
    lateinit var hashDoc : HashSet<String>
    for(lugar in mejoresLugares){
        hashDoc = HashSet<String>()
        // Concatenación de palabras nombre + tipos
        val frase = lugar.nombre + " " + lugar.tiposOriginales
        for(palabra in frase.split(" ")){
            if(!arrayDescartes.contains(palabra) &&
!hashDoc.contains(palabra)) {
                if (mapaFrecDocumentos.containsKey(palabra)) {
                    var veces = mapaFrecDocumentos.get(palabra)
                    mapaFrecDocumentos.put(palabra, veces!! + 1)
                } else {
                    mapaFrecDocumentos.put(palabra, 1)
                }
                hashDoc.add(palabra)
            }
        }
    }
    val it = mapaFrecDocumentos.iterator()

    while(it.hasNext()){
        val termino = it.next()
        val idf = Ln(tam/termino.value.toDouble()).absoluteValue
        mapaIdf.put(termino.key, idf)
    }
    FirebaseConnector.instance.getIgnorados(listLugares, itemIgnorados)
}

```

A continuación, para cada lugar cercano no visitado se obtiene cada palabra y se calcula el TF y, finalmente, el TF-IDF de cada una, creando un sumatorio de este valor para cada lugar.

Solo resta ordenar de mejor a peor el ListView de lugares, es decir, de mayor a menor valor TF-IDF.

5.8.6 Diagrama de clases

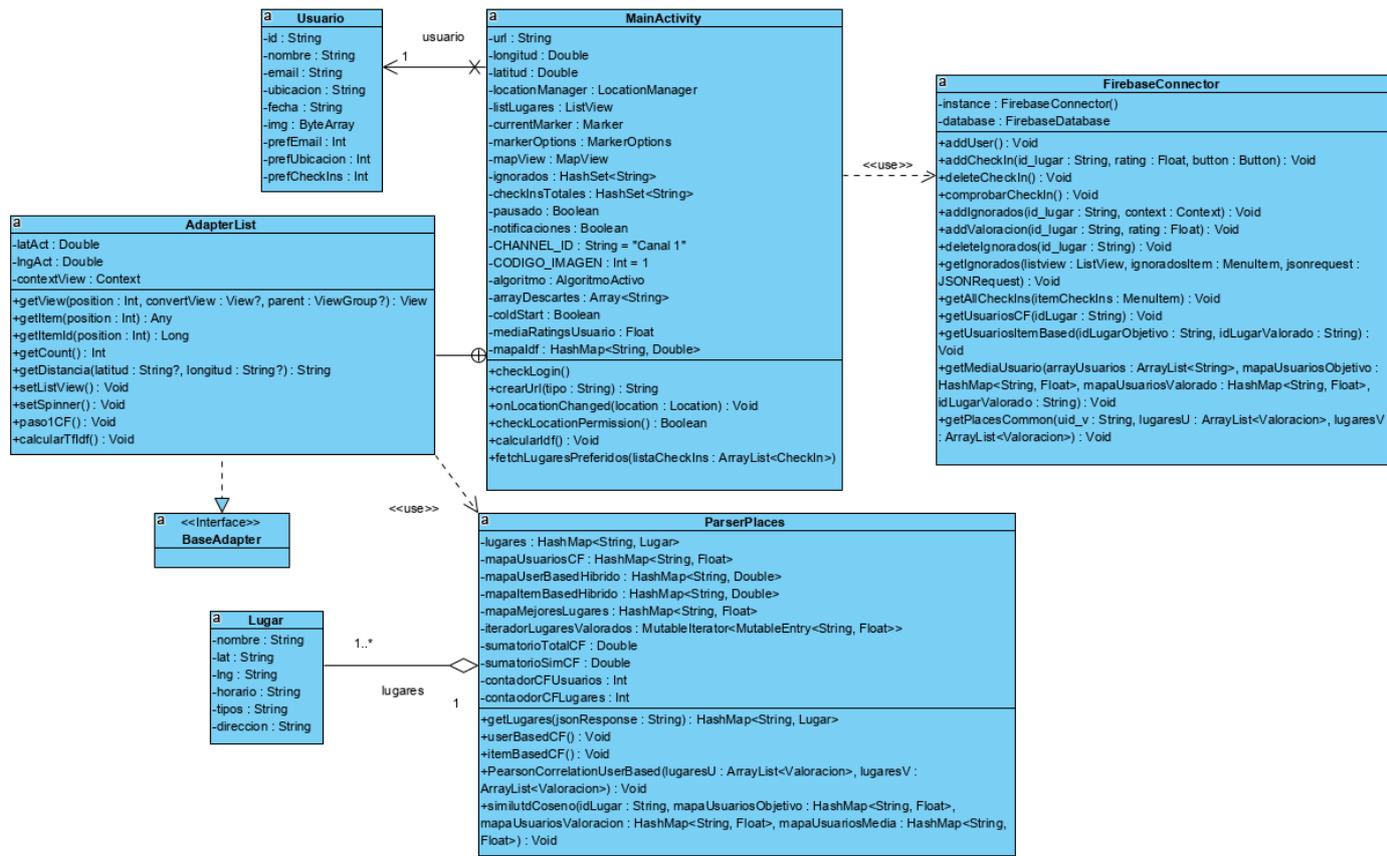


Figura 57: Diagrama de clases de la octava iteración.

En el diagrama de clases de la octava iteración, en la figura 57, se añaden los nuevos métodos relacionados con los algoritmos de filtrado. En ParserPlaces vemos las funciones para calcular el coeficiente de Pearson y la similitud coseno. En FirebaseConnector encontramos métodos para obtener listas de usuarios y sus valoraciones necesarias para los Filtrados Colaborativos. También en AdapterList se crea el método “paso1CF()”, que se llama al principio de los algoritmos FC y varias veces después, por cada POI a calcular. También tenemos en MainActivity “fetchLugaresPreferidos()” que obtiene una lista de los mejores lugares valorados por el usuario para tenerlos en cuenta en TF-IDF. Se les calcula los valores IDF en “calcularIdf()” para luego usarlos en la función de AdapterList “calcularTfIdf()”.

5.8.7 Diagrama de secuencia

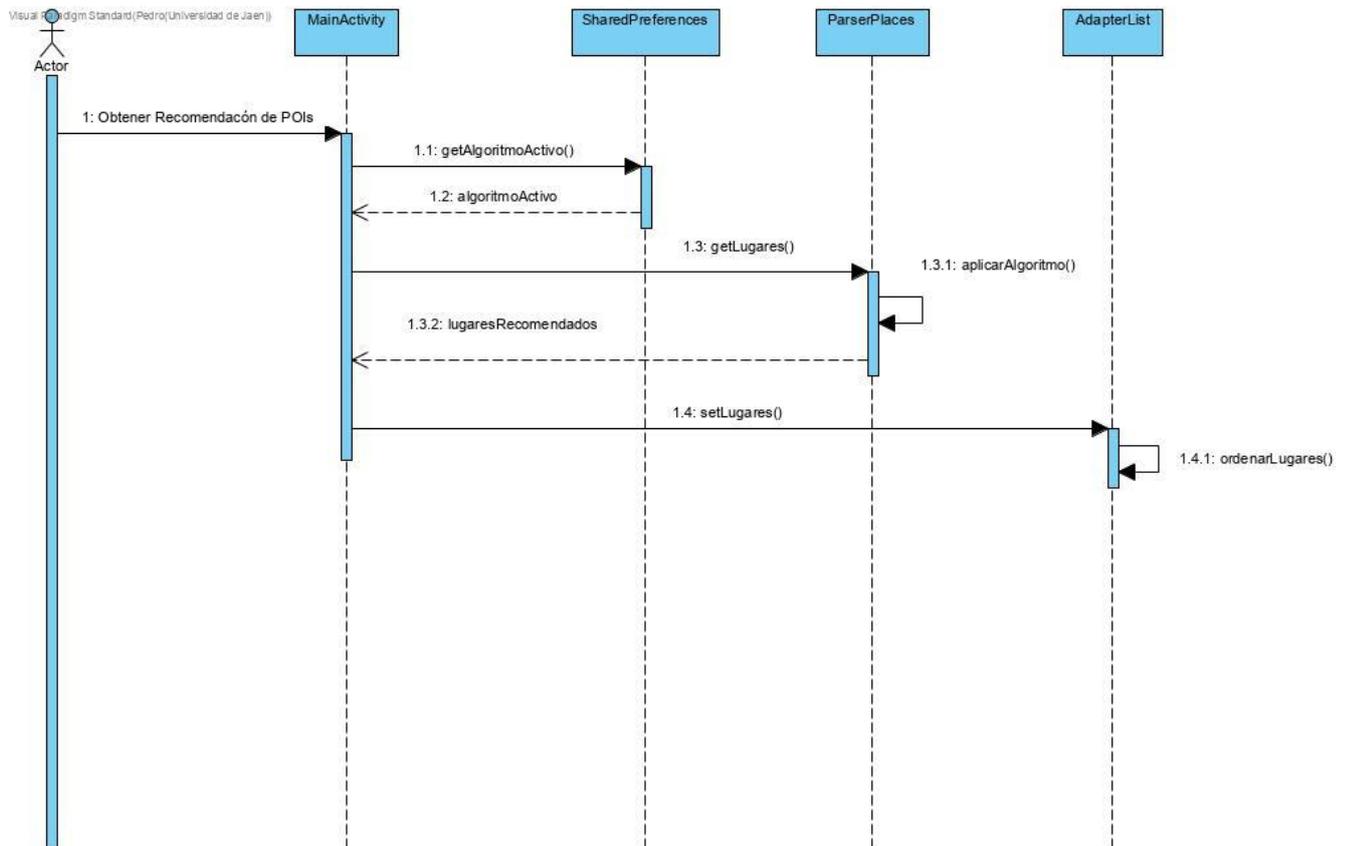


Figura 58: Diagrama de secuencia de la octava iteración.

En el diagrama de secuencia de la figura 58 primero se recupera de las SharedPreferences el algoritmo de recomendación activo. A continuación, se obtienen los lugares cercanos en ParserPlaces, que aplica el algoritmo correspondiente. Al terminar, devuelve a MainActivity los lugares puntuados, que son pasados a AdapterList para ser ordenados y mostrados en el ListView.

6. Pruebas y validación

Durante y tras la implementación de la aplicación esta ha sido instalada en el dispositivo Android Huawei P10 numerosas veces para verificar el correcto funcionamiento de cada función y ayudando al proceso de depuración, siendo más eficiente que probar la app en el emulador de Android Studio.

Se han creado tres cuentas de usuario de prueba, una por cada método distinto de inicio de sesión, con el fin de testar la funcionalidad relacionada con la búsqueda de usuarios/amigos, así como el cálculo de los algoritmos de Filtrado Colaborativo realizando “Check-Ins” a lugares de forma conveniente.

Se ha comprobado de forma exhaustiva su **robustez**, forzando los diversos escenarios de uso posibles para asegurar la ausencia de errores inesperados (comprobación de datos nulos), no habiéndose encontrado ninguno en la versión final. La app comprueba que exista conexión a internet antes de obtener los lugares, mostrando un aviso en pantalla en caso contrario. También se comprueban que el usuario haya concedido todos los permisos necesarios, tales como el de ubicación o internet.

En cuanto a **eficiencia**, este resulta el apartado más perjudicado. Mientras que la llamada a la API de Google Places para obtener los lugares (6 llamadas cada vez, una por cada categoría de lugar) son relativamente rápidas, el problema se encuentra al hacer uso de la base de datos de Firebase. Los algoritmos de Filtrado Colaborativo requieren de una gran cantidad de llamadas a la BD en poco tiempo y, si bien una sola llamada a Realtime Database resulta casi inmediata, el cúmulo de estas provoca una latencia importante a la hora calcular la predicción. Esto es indeseable dado que el conjunto de estas dos operaciones (obtención de lugares y aplicación de algoritmo de preferencia) son las que marcan el tiempo de espera inicial de la app, pues hasta que no finalizan no se muestra la lista principal de lugares y tampoco se permite acceder a la mayor parte de secciones de ajustes.

Ayudándome de la librería de Android **TimingLogger**, he realizado un estudio del tiempo de ejecución de cada algoritmo. Hay que tener en cuenta que he realizado varios “Check-Ins” (entre 3 y 5) con cada cuenta de usuario, de forma que los algoritmos de Filtrado Colaborativo tengan información suficiente para calcular la

puntuación de dos lugares, estando el resto con puntuación “Desconocida”. La lógica de la aplicación tiene en cuenta cuando no puede predecir un lugar, comprobándolo rápidamente en la BD y pasando al siguiente. Aun así, vemos en la tabla y en el diagrama de la figura 59 como los algoritmos FC dan tiempos entre 17 y 34 segundos que el usuario seguramente va a considerar demasiado elevados. Por otra parte, el algoritmo basado en contenido TF-IDF resulta altamente eficiente al ni siquiera requerir de llamadas a Firebase, con tiempos inferiores a 4 segundos.

Pruebas	FC Usuario	FC Ítem	FC Híbrido	TF-IDF
Prueba 1	19594 ms	19794 ms	33366 ms	3899 ms
Prueba 2	17076 ms	20874 ms	34892 ms	3215 ms
Prueba 3	17456 ms	20730 ms	35787 ms	3795 ms
Prueba 4	17117 ms	18923 ms	34296 ms	3870 ms
Prueba 5	18287 ms	22564 ms	32633 ms	3230 ms
Media total:	17906 ms	20577 ms	34194 ms	3601 ms

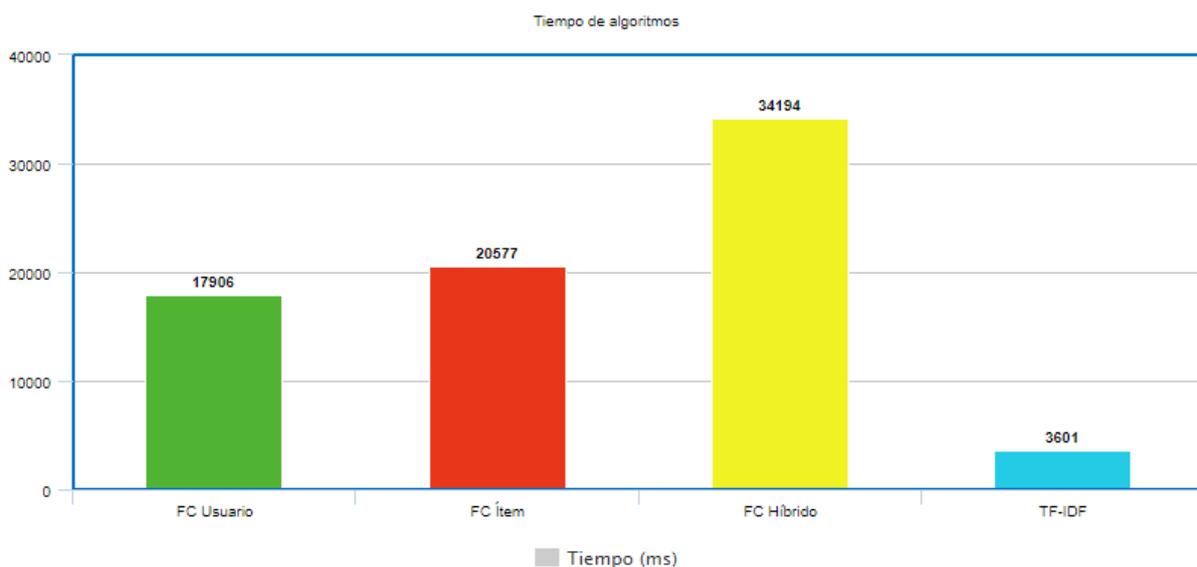


Figura 59: Diagrama de barras de tiempo de algoritmos.

Para la realización de estas pruebas se han registrado 22 solicitudes a Google Places, lo que está lejos del límite gratuito mensual. Sin embargo, está claro que para garantizar la **escalabilidad** a un número elevado de usuarios se hace imprescindible un mayor volumen de solicitudes cuyo coste deberá compensarse con ingresos, ya sea un pago único, modelo de suscripción o publicidad.

No resulta posible realizar un estudio estadístico para determinar la **eficacia** de las recomendaciones de POIs, que suponen el objeto principal de este trabajo, sin una cantidad elevada de usuarios que proporcionen la retroalimentación necesaria,

especialmente en el caso del FC. Sin embargo, el resultado del TF-IDF mostrado en la figura 60 parece coherente en base a lo "Check-Ins" realizados en estas pruebas, dando valor a las categorías y nombres de los lugares.



Figura 60: Resultados de TF-IDF.

En conclusión, se ha logrado un producto funcional que cumple la mayoría de expectativas marcadas para este trabajo práctico.

7. Conclusiones

Para finalizar, expondré a continuación las conclusiones extraídas del desarrollo de este proyecto, reflexionando sobre la experiencia global y pensando ideas de cómo podría mejorarse la aplicación en un futuro.

Mi experiencia con la programación para Android antes de la realización de este trabajo era muy escasa y, en concreto con el lenguaje Kotlin, nula. Gracias al desarrollo de Appoi he tenido que aprender competencias tales como las particularidades de un nuevo lenguaje de programación, el diseño de interfaces adaptativo para dispositivos Android o la creación e integración de servicios Firebase; todas ellas, estoy seguro, resultarán útiles en mi futuro.

También destaco el empleo de una metodología ágil y de una plataforma de gestión de proyectos como GitHub, que me ha servido tanto para aprender más en profundidad como refrescar conceptos. No me arrepiento de haberlas utilizado a pesar de ser un trabajo individual, pues han servido para hacerme una idea de por qué son tan utilizados en proyectos en equipo.

En general, estoy contento con el resultado final y satisfecho de haber trabajado por primera vez en campos tan interesantes e importantes como los sistemas de recomendación, la geolocalización y mezcla de ambos.

Respecto a la aplicación, siento que el aspecto social de ella podría haberse expandido y ofrecer más servicios. La idea original no incluía sistemas de perfiles de usuario como tal, ni relaciones de amistad. Los usuarios registrados en la app serían invisibles entre ellos, pues en principio se cogerían los datos necesarios para los algoritmos de forma aleatoria. Fue cuando estaba avanzado el desarrollo que decidí que sería interesante incluir la variable de relación personal entre usuarios, pues varios estudios como los referenciados a lo largo del trabajo mencionan su importancia en los sistemas de recomendación. Así, incluí el sistema de crear y buscar perfiles de usuario y añadirlos como amigos para que los algoritmos de FC los tuvieran en cuenta. Sin embargo, creo que podría haberse pulido más este aspecto, mostrando por ejemplo una lista pública de amigos de cada usuario.

Aunque sin duda la principal cuestión a mejorar en futuras actualizaciones es la eficiencia. Quizás un replanteamiento del código y el manejo de estructuras de datos implicadas en el cálculo de los algoritmos podría mejorar el tiempo de espera, pero todo indica que Firebase Realtime Database, siendo útil en muchas situaciones, no es ideal para este tipo de problemas.

Por último, me hubiese gustado darle más protagonismo al mapa de la pantalla principal, permitiendo por ejemplo cambiar la posición actual por otra cualquiera que el usuario desee, explorando recomendaciones de POIs en lugares en los que no está en el momento.

Anexo 1: Guía de instalación

En este anexo se reflejará lo necesario para instalar correctamente la aplicación en un dispositivo Android.

Esta app está compilada con SDK 28, lo que significa que está optimizada para Android 9. Sin embargo, su mínima versión de SDK es 23, lo que significa que es **compatible con Android 6.0 Marshmallow** en adelante.

Una vez verificado que el smartphone posee una versión de Android compatible, hay que comprobar que esté permitida la instalación de fuentes desconocidas dentro de los ajustes del sistema, pues Appoi no está firmada para la Play Store de Google. Los pasos para activar esta opción son los siguientes (pueden variar ligeramente según la capa de personalización del sistema):

1. Ingresar en la aplicación de Ajustes.
2. En la lista, seleccionar Seguridad.
3. Ubicar la opción “Permitir fuentes desconocidas” y activar el *switch*.
4. Seleccionar “Aceptar” en la ventana emergente.

Después solo resta introducir el archivo .apk en el dispositivo y abrirlo con un gestor de archivos para iniciar la instalación.

Otra forma alternativa de ejecutar la aplicación es mediante un ordenador a través del IDE Android Studio y abriendo la carpeta del proyecto. Se puede utilizar el emulador integrado o lanzar la app directamente en el dispositivo móvil mediante conexión USB, habiendo instalado los drivers ADB previamente.

Anexo 2: Manual de usuario

En el segundo anexo se detallará un pequeño tutorial de uso de la aplicación una vez instalada correctamente.

A2.1 Inicio de sesión y registro

Lo primero que nos encontramos al iniciar la app es la **pantalla de inicio de sesión** como la de la figura 61.



Appoi

Inicie sesión para continuar

ejemplo@gmail.com

Contraseña (6 car...

REGISTRAR ENTRAR

Recuperar contraseña

Continuar con Facebook

Iniciar sesión

Figura 61: Pantalla de login.

Se debe elegir una de las tres opciones disponibles: Facebook, Google o correo y contraseña. Para las dos primeras se deben elegir los botones correspondientes en la parte inferior y aceptar las pantallas siguientes del proveedor. Para el login mediante correo, se debe pulsar el botón “Registrar”.

Se elija cualquiera de las opciones, lo siguiente que nos encontramos es la **pantalla de registro** como muestra la figura 62.



Figura 62: Pantalla de registro.

Aquí se introducirán los datos: email (si se ha elegido login por correo, sino se autocompleta con el correo del proveedor), nombre de usuario público, contraseña y ubicación (opcional). Si se desea también se puede subir una foto de perfil pública guardada en el dispositivo pulsando sobre la imagen del retrato.

Una vez se pulsa el botón “Registrarse”, se comprueban que los datos introducidos sean válidos (alias o email no repetido, contraseña introducida correctamente y longitud adecuada de datos), se realiza el registro de usuario en la base de datos y aparece la pantalla principal de la aplicación.

A2.2 Hacer Check-In, puntuar e ignorar

Desde la pantalla principal vemos una **lista de POIs cercanos** en la mitad superior y un **mapa** que marca la posición actual y los POIs en la mitad inferior, como muestra la figura 63. Los lugares de interés aparecen ordenados de mayor a menor por el valor “Puntuación” (o “Valor TF-IDF” si se usa dicho algoritmo). La lista de lugares se puede filtrar por nombre escribiendo en la barra de búsqueda superior. También se pueden por varias categorías pulsando sobre “Categorías” a la derecha de la barra de búsqueda, lo que despliega una lista de categorías con casillas al lado.

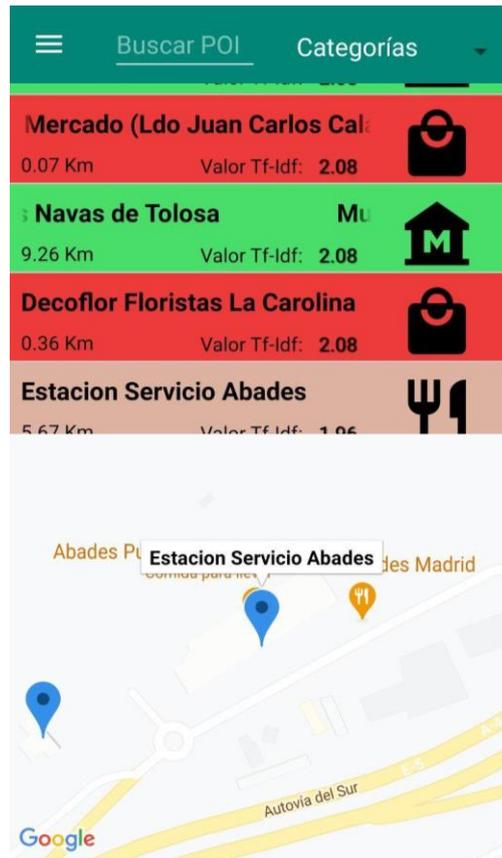


Figura 63: Pantalla principal.

Para realizar un “Check-In”, se debe pulsar sobre el lugar en la lista. Esto abre la vista **detallada del lugar**, en la figura 64. Se muestra una foto del POI y detalles varios como el nombre, dirección y horario. Debajo de este último se encuentra el botón de “**Check-In**”. Al pulsar sobre él se registrará en la base datos, y se mostrará justo debajo una **barra de 5 estrellas** que sirven para valorar el lugar del 0 al 5. Finalmente, debajo se encuentra el botón “No interesante”, que añade a la lista de ignorados el lugar volviendo a la pantalla principal.

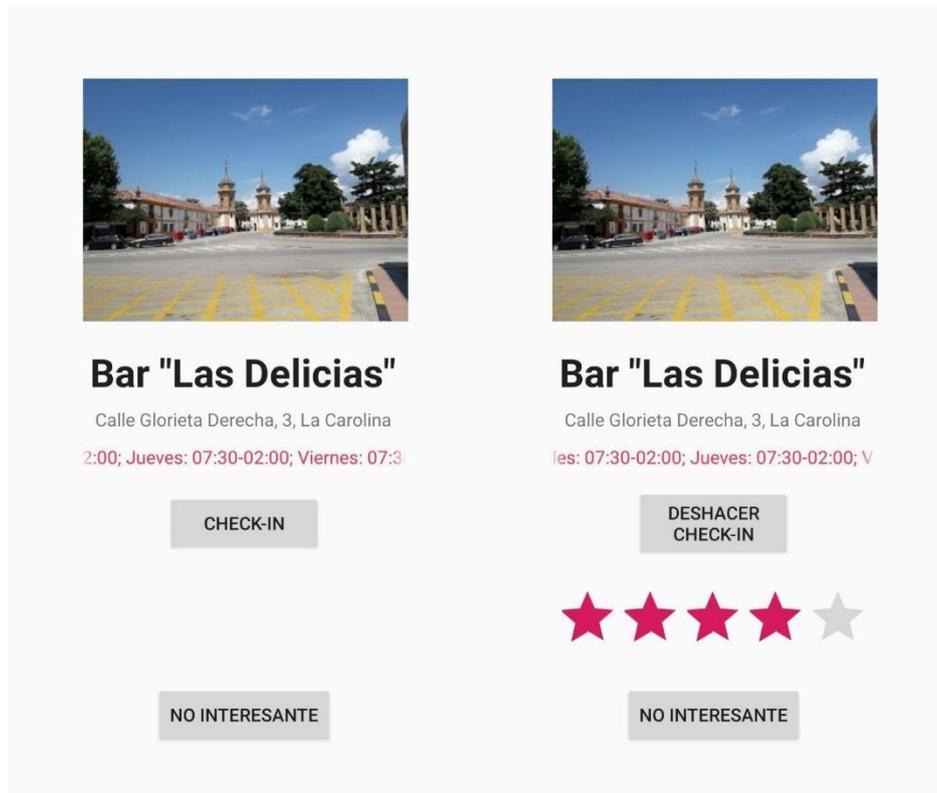


Figura 63: Pantalla principal.

Para comprobar las listas de Check-Ins e ignorados, desde la pantalla principal se abre el **menú lateral izquierdo** pulsando sobre el icono de las tres barras en la esquina superior izquierda, o arrastrando las pantallas hacia la derecha desde el borde izquierdo. En la figura 64 vemos el menú desplegado con las opciones de “**Ver Check-Ins**” y “**Ver ignorados**”. Ambas pantallas mostrarán una lista con los lugares correspondientes. En el caso de los ignorados, al pulsar sobre un lugar se puede eliminar de dicha lista al aceptar la ventana emergente.

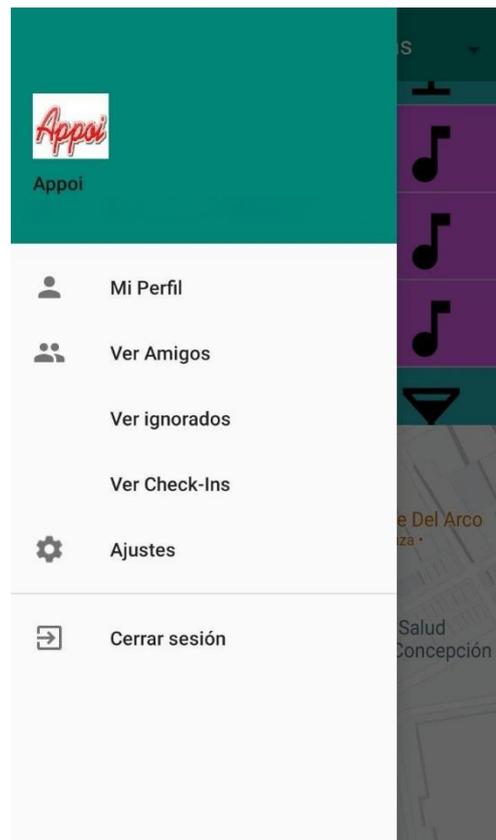


Figura 64: Menú lateral.

A2.3 Ver y cambiar datos de perfil

Desde el menú lateral de la figura 64 se pulsa en la opción **“Mi Perfil”**, cargando la pantalla de perfil de usuario de la figura 65. Aquí se muestra la imagen de perfil y datos de alias (*nickname*), correo electrónico, fecha de registro, ubicación y una lista de los últimos 5 Check-Ins realizados. Pulsando sobre la imagen se puede seleccionar otra para cambiarla. También se puede pulsar sobre el campo de ubicación para modificarlo, solo después de pulsar “Guardar Cambios” justo debajo.



Figura 65: Vista de perfil de usuario.

A2.4 Ver lista de amigos y buscar usuarios

Para ver la lista de usuarios agregados como “amigos” se presiona la opción “**Ver amigos**” del menú lateral. Esta pantalla, en la figura 66, contiene una lista de usuarios con su imagen de perfil y alias y, en la barra superior, una barra de búsqueda por alias y un botón “+”. El botón “+” cargará otra lista de usuarios, esta vez con los que no han sido agregados como “amigos” y con el botón al lado del alias “Agregar a Amigos”. Pulsar un usuario abrirá una vista como la del perfil de la figura 65, pero sin opciones de modificar y cambiando el botón de “Guardar cambios” por “Agregar a Amigos”. El pulsar sobre cualquiera de los botones de agregado añadirá al usuario a la lista de amigos, lo que será tenido en cuenta en la recomendación de lugares de interés.

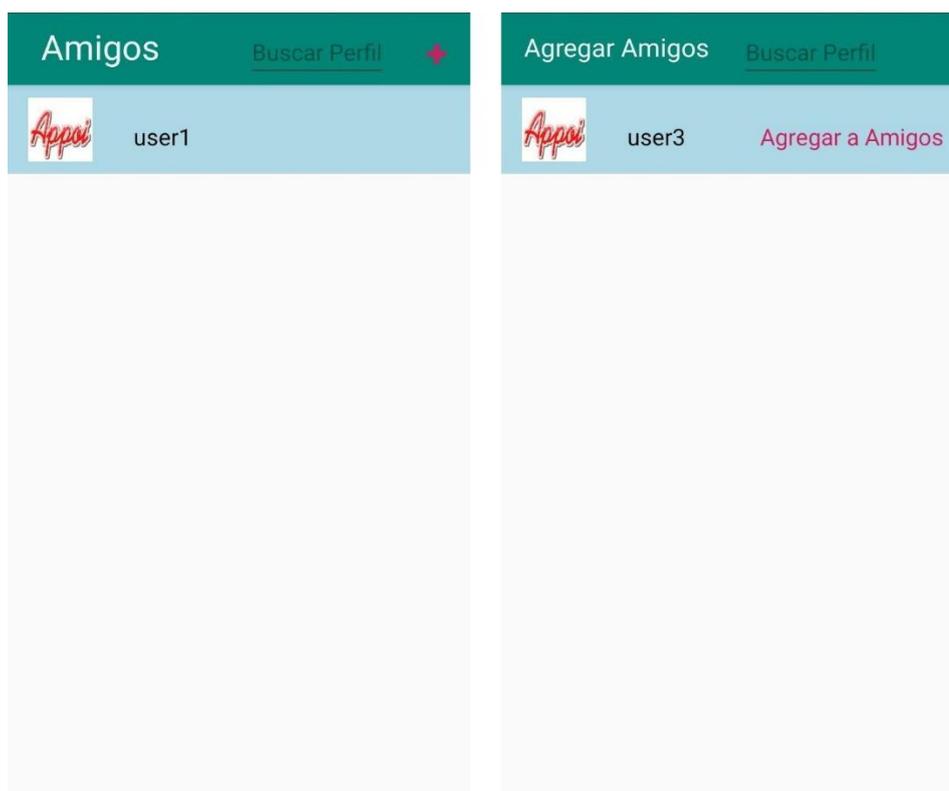


Figura 66: Lista de amigos y búsqueda de usuarios.

A2.5 Ajustes y cierre de sesión

Desde el menú lateral, al seleccionar “Ajustes” se abre la pantalla de la figura 67. Aquí se pueden personalizar varias preferencias: notificaciones (cuando se encuentran nuevos POIs y la app está en suspensión), selección de algoritmo de filtrado (despliega una lista de 4 algoritmos distintos) y opciones de privacidad, que establecen la visibilidad del correo electrónico, la ubicación y los últimos “Check-Ins” cuando otro usuario accede a tu perfil.

Finalmente, la última opción del menú lateral es “Cerrar sesión”, que termina la sesión actual y manda a la pantalla de login.

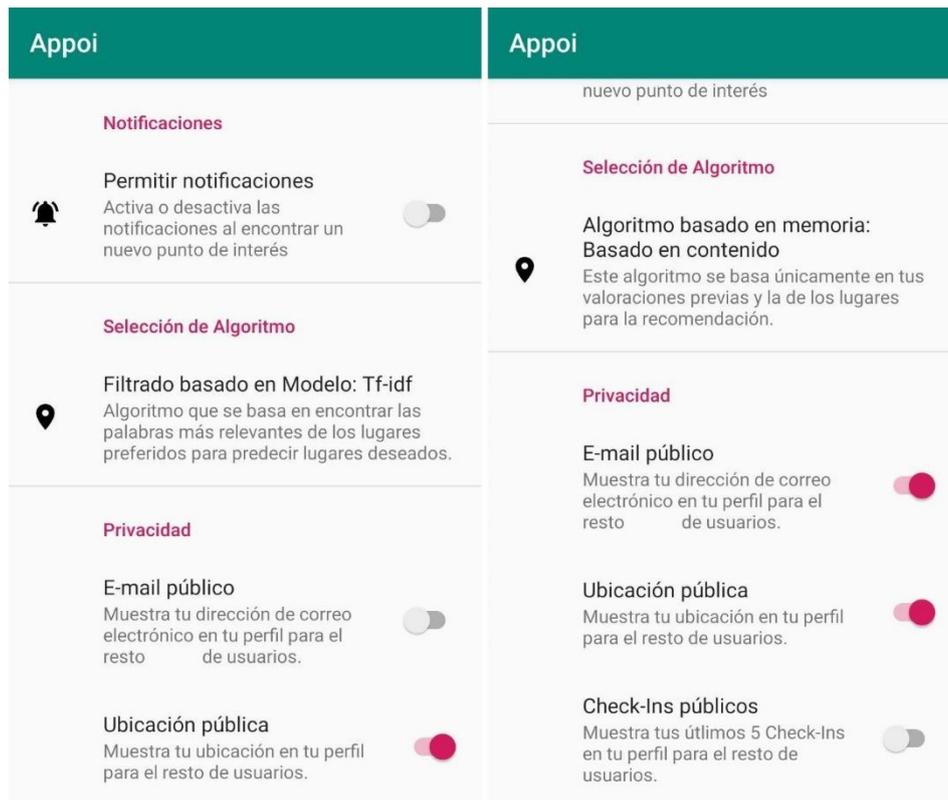


Figura 67: Pantalla de ajustes.

Índice de Figuras

Figura 1: <https://thenextscoop.com/build-profitable-mobile-app-startup/>

Figura 2: <https://www.fastcompany.com/1669015/foursquare-solves-a-basic-ui-problem-that-eludes-google-maps-and-yelp>

Figura 3: <https://www.slashgear.com/facebook-places-deals-hits-uk-and-europe-31129334/>

Figura 4: <https://www.fastcompany.com/40483528/20-incredibly-useful-things-you-didnt-know-google-maps-could-do>

Figura 5: Visual Paradigm

Figura 6: <https://cloud.google.com/maps-platform/pricing/sheet?hl=es-419>

Figura 7: <https://cloud.google.com/maps-platform/pricing/sheet?hl=es-419>

Figura 8: <https://www.ericdecanini.com/2019/03/04/managing-firebase-costs/>

Figura 9: Balsamiq MockUps

Figura 10: Balsamiq MockUps

Figura 11: Balsamiq MockUps

Figura 12: Balsamiq MockUps

Figura 13: Balsamiq MockUps

Figura 14: Balsamiq MockUps

Figura 15: Balsamiq MockUps

Figura 16: Balsamiq MockUps

Figura 17: Balsamiq MockUps

Figura 18: <https://medium.com/@Emmitta/ciclo-de-vida-de-una-actividad-android-f30f8f2d1256>

Figura 19: <https://console.firebase.google.com/>

Figura 20: <https://console.firebase.google.com/>

Figura 21: Visual Paradigm

Figura 22: Visual Paradigm

Figura 23: Android Studio

Figura 24: Captura Appoi

Figura 25: Visual Paradigm

Figura 26: Visual Paradigm

Figura 27: Captura Appoi

Figura 28: Captura Appoi

Figura 29: Visual Paradigm

Figura 30: Visual Paradigm

Figura 31: <https://console.firebase.google.com/>

Figura 32: <https://console.firebase.google.com/>

Figura 33: Visual Paradigm

Figura 34: Android Studio

Figura 35: Captura Appoi

Figura 36: Captura Appoi

Figura 37: Captura Appoi

Figura 38: Visual Paradigm

Figura 39: Visual Paradigm

Figura 40: Captura Appoi

Figura 41: Captura Appoi

Figura 42: Visual Paradigm

Figura 43: Visual Paradigm

Figura 44: Captura Appoi

Figura 45: <https://console.firebase.google.com/>

Figura 46: Captura Appoi

Figura 47: Captura Appoi

Figura 48: Captura Appoi

Figura 49: Captura Appoi

Figura 50: Captura Appoi

Figura 51: Visual Paradigm

Figura 52: Visual Paradigm

Figura 53: Android Studio

Figura 54: <https://towardsdatascience.com/collaborative-filtering-based-recommendation-systems-exemplified-ecbffe1c20b1>

Figura 55: <https://towardsdatascience.com/comprehensive-guide-on-item-based-recommendation-systems-d67e40e2b75d>

Figura 56: https://es.ryte.com/wiki/TF*IDE

Figura 57: Visual Paradigm

Figura 58: Visual Paradigm

Figura 59: <https://www.meta-chart.com/>

Figura 60: Captura Appoi

Figura 61: Captura Appoi

Figura 62: Captura Appoi

Figura 63: Captura Appoi

Figura 64: Captura Appoi

Figura 65: Captura Appoi

Figura 66: Captura Appoi

Figura 67: Captura Appoi

Bibliografía

- [1] Rashid AM, Albert I, Cosley D, Lam SK, McNee SM, Konstan JA et al. *Getting to know you: learning new user preferences in recommender systems*. In: Proceedings of the international conference on intelligent user interfaces; 2002. p. 127–34.
- [2] Schafer JB, Konstan J, Riedl J. *Recommender system in e-commerce*. In: Proceedings of the 1st ACM conference on electronic commerce; 1999. p. 158–66.
- [3] Isinkaye, F. O.; Folajimi, Y. O.; OJOKOH, B. A. Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal*, 2015, vol. 16, no 3, p. 261-273.
- [4] Balagueró, Thaís (13 de noviembre, 2018), *¿Qué son los datasets y los dataframes en el Big Data?*, de <https://www.deustoformacion.com/blog/programacion-diseno-web/que-son-datasets-dataframes-big-data>
- [5] Gao, Huiji; *Personalized POI Recommendation on Location-Based Social Networks, Conclusion and future work*; 2014. pag. 106.
- [6] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 2001
- [7] Noticias ONU (16 de mayo, 2018). *Las ciudades seguirán creciendo, sobre todo en los países en desarrollo*, de <https://news.un.org/es/story/2018/05/1433842>
- [8] Parker, Joe (29 de agosto, 2019). *10 years of growth of Mobile App Market*, de <https://www.knowband.com/blog/mobile-app/growth-of-mobile-app-market/>
- [9] Statcounter GlobalStats (Agosto 2020), *Mobile Operating System Market Share Worldwide - August 2020*, de <https://gs.statcounter.com/os-market-share/mobile/worldwide>
- [10] Sharifov, Maxim (17 de mayo, 2017), *Kotlin on Android. Now official*, de <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>
- [11] Android Developers (4 de junio, 2020), *Datos de ubicación*, de <https://developers.google.com/maps/documentation/android-sdk/location?hl=es-419>
- [12] Gell, Aaron (2 marzo, 2017), *The Not-So-Surprising Survival of Foursquare*, de <https://www.newyorker.com/business/currency/the-not-so-surprising-survival-of-foursquare>
- [13] Clement J. (21 de agosto, 2020), *Most popular social networks worldwide as of July 2020, ranked by number of active users*, de

<https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/>

- [14] Sinha Robin (6 de febrero de 2015), *Google Maps for Android Update Brings New 'Local Guides' Feature*, de <https://gadgets.ndtv.com/apps/news/google-maps-for-android-update-brings-new-local-guides-feature-657816>
- [15] SoftAuthor (julio 2020), *Places API Nearby Search*, de <https://softauthor.com/google-maps-places-api-nearby-search-request/>
- [16] Martínez, Ana Belén Barragans, et al. *What's on TV tonight? An efficient and effective personalized recommender system of TV programs. IEEE Transactions on Consumer Electronics*; 2009, vol. 55, no 1, p. 286-294.
- [17] Nieto, S. M.; *Filtrado colaborativo y sistemas de recomendación. Inteligencia en Redes de Comunicaciones. Madrid, 2007.*
- [18] López, Vreixo Formoso; *Técnicas eficientes para la recomendación de productos basadas en filtrado colaborativo. 2013. Tesis Doctoral. Universidade da Coruña., p.45*
- [19] Breese, John S.; Heckerman, David; KADIE, Carl. *Empirical analysis of predictive algorithms for collaborative filtering. 2013.*
- [20] Castellano, Emilio J.; Martínez, Luis; *Orieb, A Crs For Academic Orientation Using Qualitative Assessments. En e-Learning. 2008. p. 7.*
- [21] Lee, Joonseok; SUN, Mingxuan; Lebanon, Guy; *A comparative study of collaborative filtering algorithms, 2012.*
- [22] Xiaoyuan Su, Taghi M. Khoshgoftaar; *A survey of collaborative filtering techniques, Advances in Artificial Intelligence, 2009.*
- [23] Koren, Yehuda; *Recommender system utilizing collaborative filtering combining explicit and implicit feedback with both neighborhood and latent factor models. U.S. Patent No 8,037,080, 11 Oct. 2011.*
- [24] Mobasher, Bamshad; BURKE, Robin; SANDVIG, Jeff J.; *Model-based collaborative filtering as a defense against profile injection attacks. En AAAI. p. 1388. 2006*
- [25] Carleton College, Northfield, MN, *Model-based recommendation systems*, de http://www.cs.carleton.edu/cs_comps/0607/recommend/recommender/modelbased.html
- [26] Van Meteren, Robin; Van Someren, Maarten. *Using content-based filtering for recommendation. 2000. p. 47-56.*

- [27] González Villa, Juan (13 de febrero, 2019), *TF IDF: herramientas para mejorar la relevancia de tus contenidos*, de https://useo.es/tf-idf-relevancia/#Que_es_el_TF_IDF
- [28] Wang, Yuanyuan; Chan, Stephen Chi-Fai; Ngai, Grace. *Applicability of demographic recommender system to tourist attractions: a case study on trip advisor*. En *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*. IEEE, 2012. p. 97-101.
- [29] Burke, Robin; *Hybrid Web Recommender Systems*, *The Adaptive Web*, 2007.pp 380
- [30] Bluepi (2016), *Demystifying Hybrid Recommender Systems And Their Use Cases*, de <https://www.bluepiit.com/blog/demystifying-hybrid-recommender-systems-and-their-use-cases/>
- [31] N. Eagle, A. Pentland, and D. Lazer. Inferring friendship network structure by using mobile phone data. *Proceedings of the National Academy of Sciences*, 2009.
- [32] Tobler, W.; A computer movie simulating urban growth in the Detroit region. *Economic Geography*, 46(Supplement), 1970, pp 234–240.
- [33] Michael Page (2019), *Tendencias del mercado laboral*, de https://www.michaelpage.es/sites/michaelpage.es/files/PG_ER_IT.pdf
- [34] Maja Majewski (12 de marzo, 2019), *Top 6 Software Development Methodologies*, de <https://blog.planview.com/top-6-software-development-methodologies/>
- [35] Ruby Casallas, Andrés Yie; *Ingeniería de software: Ciclos de Vida y Metodologías*. 2013
- [36] proyectosagiles.org (2018), *Qué es SCRUM*, de <https://proyectosagiles.org/que-es-scrum/>
- [37] SOFTENG (1 de marzo, 2011), *Proceso y Roles de Scrum*, de <https://www.softeng.es/es-es/empresa/metodologias-de-trabajo/metodologia-scrum/proceso-roles-de-scrum.html>
- [38] OBS Business School, *Principales ventajas y limitaciones de las metodologías ágiles*, de <https://obsbusiness.school/int/blog-project-management/metodologia-agile/principales-ventajas-y-limitaciones-de-las-metodologias-agiles>
- [39] Skerrett, Ian (23 de junio, 2014), *Eclipse Community Survey 2014 Results*, de <https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/>

- [40] Firebase, de <https://firebase.google.com/>
- [41] Android Developers (2020), *Cómo interpretar el ciclo de vida de una actividad*, de <https://developer.android.com/guide/components/activities/activity-lifecycle?hl=es>
- [42] Android Developers (2020), *Intent*, de <https://developer.android.com/reference/kotlin/android/content/Intent>
- [43] Firebase (2020), *Autentica mediante el Acceso con Facebook en Android*, de <https://firebase.google.com/docs/auth/android/facebook-login?hl=es>
- [44] Android Developers (2020), *Cómo solicitar permisos de la app*, de <https://developer.android.com/training/permissions/requesting?hl=es-419>
- [45] Android Developers (2020), *AsyncTask*, de <https://developer.android.com/reference/kotlin/android/os/AsyncTask>
- [46] Android Developers (2020), *BaseAdapter*, de <https://developer.android.com/reference/android/widget/BaseAdapter>
- [47] Android Developers (2020), *Parcelable*, de <https://developer.android.com/reference/android/os/Parcelable>
- [48] Android Developers (2020), *Places API Políticas*, de <https://developers.google.com/places/web-service/policies?hl=es>
- [49] Firebase (2020), *Estructura tu base de datos*, de <https://firebase.google.com/docs/database/android/structure-data>
- [50] Android Developers (2020), *DrawerLayout*, de <https://developer.android.com/reference/androidx/drawerlayout/widget/DrawerLayout>
- [51] Android Developers (2020), *PreferenceScreen*, de <https://developer.android.com/reference/androidx/preference/PreferenceScreen>
- [52] Android Developers (2020), *Cómo guardar datos de pares clave-valor*, de <https://developer.android.com/training/data-storage/shared-preferences>
- [53] Android Developers (2020), *Fragments*, de <https://developer.android.com/guide/components/fragments?hl=es>
- [54] Saluja, Chhavi (6 de marzo, 2018), *Collaborative Filtering based Recommendation Systems exemplified...*, de <https://towardsdatascience.com/collaborative-filtering-based-recommendation-systems-exemplified-ecbffe1c20b1>
- [55] Al-Shamri, Mohammad Yahya H.; Al-Ashwal, Nagi H.; *Fuzzy-weighted Pearson Correlation Coefficient for Collaborative Recommender Systems*. 2013. p. 410.

[56] Ekstrand, Michael (24 de octubre, 2013), *Similarity Functions for User-User Collaborative Filtering*, de <https://grouplens.org/blog/similarity-functions-for-user-user-collaborative-filtering/>

[57] Qutbuddin, Muffaddal (7 de marzo, 2020), *Comprehensive Guide on Item Based Collaborative Filtering*, de <https://towardsdatascience.com/comprehensive-guide-on-item-based-recommendation-systems-d67e40e2b75d>