



MeetMap

¿Ya tienes plan para hoy?

CICLO FORMATIVO DE GRADO SUPERIOR:

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

AUTORES:

**Almudena Fernández Cárdenas
Daniel García Ayala
Iker Iturrealde Tejido**

Licencia

Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirlGual 4.0 Internacional. Para ver una copia de esta licencia, visite

<https://creativecommons.org/licenses/by-nc-sa/4.0/>



DEDICATORIAS Y AGRADECIMIENTOS

Almudena

A los tres pilares de mi vida, que brillantes, ocultos y en ocasiones derruidos, no dejan de guiarme en el camino.

¡Dicen que nunca se rinde!

Daniel

A mi madre por ver el progreso y querer probarlo, aunque no se entere nada de cómo funcionan las cosas.

Iker

A mi novia y amigos, que fueron capaces de aguantarme y opinar sobre cada pequeño avance que hacía en el mapa.

RESUMEN

MeetMap es una aplicación móvil innovadora que permite a los usuarios explorar una amplia variedad de planes de ocio en la ciudad de Madrid en los próximos 100 días. La plataforma está diseñada para que los usuarios puedan conectarse con otras personas interesadas en el mismo plan de ocio y comunicarse con ellas mediante un chat. A través de una interfaz intuitiva y fácil de usar, los usuarios pueden acceder a una gran cantidad de información sobre eventos y actividades culturales, deportivas, sociales y de ocio en general.

La idea de desarrollar esta aplicación viene asociada a los cambios en los hábitos sociales que se han visto acentuados a raíz de la pandemia de Covid-19 que asoló las ciudades.

Con el confinamiento, los ciudadanos empezaron a buscar otras alternativas a la hora de conocer gente y amenizar su tiempo de ocio en casa con actividades que les evadiesen de la realidad. Dentro de estas alternativas se produjo un gran aumento del uso de los dispositivos móviles (teléfonos, tablets, etc.) y en consecuencia se elevó la descarga y consumo de contenido que ofrecían diferentes aplicaciones que permitían conocer gente sin necesidad de saltarse las restricciones impuestas en aquellos momentos.

Cuando se empezaron a flexibilizar las restricciones, este consumo de aplicaciones lejos de disminuir se mantuvo al alza y se convirtió en un recurso para personas con más problemas a la hora de socializar y aunque el cara a cara parece que vuelve a ser una opción mayoritaria, muchas personas siguen optando como forma complementaria este tipo de aplicaciones para establecer un primer contacto.

Esta aplicación pretende cubrir las necesidades de las personas usuarias de las aplicaciones para conocer gente y de ocio. Se ha podido comprobar que las diferentes aplicaciones que hay en las tiendas de aplicaciones (*Play Store*, *Apple Store*, etc.) relacionadas con los planes de ocio, no están diseñadas para poder interactuar con los diferentes usuarios que también puedan realizar estas actividades. Lo mismo ocurre con las aplicaciones para conocer gente, están únicamente destinadas a interactuar con el resto de los usuarios y de forma individual concretar planes de ocio que en muchas ocasiones son siempre los mismos porque no se tiene conocimiento de planes alternativos o el acceso a los mismo se vuelve complejo.

En definitiva, MeetMap no es solo una aplicación móvil, es una herramienta completa y útil para aquellos que buscan planes de ocio en la ciudad de Madrid. Con un diseño intuitivo, características personalizables y amplia variedad de planes se quiere conseguir que los usuarios tengan una gran experiencia interactiva y social que les permita descubrir nuevas actividades y eventos, hacer nuevos amigos y disfrutar de su tiempo libre de manera más eficiente y con todo ello se espera que se convierta en una aplicación popular entre los usuarios.

ABSTRACT

MeetMap is an innovative mobile application that allows users to explore a wide variety of leisure plans in the city of Madrid in the next 100 days. The platform is designed for users to connect with other people interested in the same leisure plan and communicate with them through a chat. Through an intuitive and user-friendly interface, users can access a wealth of information about cultural, sporting, social, and leisure events and activities in general.

The idea of developing this application is associated with changes in social habits that have been accentuated as a result of the Covid-19 pandemic that devastated cities. With confinement, citizens began to look for other alternatives to meet people and liven up their leisure time at home with activities that would distract them from reality. Within these alternatives, there was a great increase in the use of mobile devices (phones, tablets, etc.) and consequently, the download and consumption of content offered by different applications that allowed people to meet others without violating the restrictions imposed at that time.

When the restrictions began to be relaxed, this consumption of applications did not decrease, but remained on the rise and became a resource for people who had more difficulty socializing. Although face-to-face seems to be a majority option again, many people still opt for this type of application as a complementary way to establish initial contact.

This application aims to meet the needs of users of applications to meet people and leisure. It has been found that the different applications available in app stores (PlayStore, AppStore, etc.) related to leisure plans are not designed to interact with other users who may also be participating in these activities. The same applies to applications for meeting people, which are only intended to interact with other users and individually plan leisure activities that are often the same because there is no knowledge of alternative plans or access to them becomes complex.

In short, MeetMap is not just a mobile application, it is a complete and useful tool for those looking for leisure plans in the city of Madrid. With an intuitive design, customizable features, and a wide variety of plans, the goal is to provide users with a great interactive and social experience that allows them to discover new activities and events, make new friends, and enjoy their free time more efficiently. With all of this, it is hoped that it will become a popular application among users.

Índice de contenido

1. INTRODUCCIÓN	12
2. CONTEXTO FUNCIONAL Y TECNOLÓGICO	15
2.1 Contexto funcional	15
2.2 Contexto tecnológico	16
3. PLANIFICACIÓN DEL PROYECTO	20
4. DESCRIPCIÓN DE LA SOLUCIÓN: ANÁLISIS Y DISEÑO	24
4.1 Especificación de requisitos.....	24
4.2 Selección de plataforma tecnológica	24
4.3 Descripción del diseño de la solución	24
5. DESCRIPCIÓN DE LA SOLUCIÓN: CONSTRUCCIÓN.....	31
5.1 Documentación descriptiva	31
5.2 Problemas encontrados y soluciones	88
6. VALIDACIÓN DE LA SOLUCIÓN	93
6.1 Documentación descriptiva	93
6.2 Problemas encontrados y justificación	96
7. FUENTES	100

Índice de figuras

1. Ejemplo 1 de público objetivo	15
2. Ejemplo 2 de público objetivo	15
3. Esquema diseño centrado en el Usuario	17
4. Diagrama diseño centrado en el Usuario	18
5. Etapas sprint metodologías ágiles	18
6. Tablero Kanban de la herramienta Trello	19
7. Detalle de tarea en herramienta Trello	19
8. Lista de tareas MeetMap	22
9. Diagrama Gantt desarrollo MeetMap	23
10. Prototipo inicial MeetMap en Figma	25
11. Prototipo final MeetMap en Figma	25
12. Flujo de navegación MeetMap en Figma	26
13. Paleta de colores MeetMap	26
14. Análisis accesibilidad colores MeetMap	27
15. Personalización diseño MeetMap	28
16. Diagrama de Casos de Uso	29
17. Diagrama de Colecciones	30
18. Código Splash Activity	31
19. Código animación fade in Splash Activity	32
20. Secuencia vista Splash Activity	32
21. Vista Initial Activity	33
22. Función modo claro	34
23. Mensajes slider	34
24. Función salir de la aplicación	34
25. Función permisos ubicación	35
26. Solicitud usuario permisos ubicación	35
27. Aplicación en inglés	36
28. Archivo strings proyecto	36
29. Opción de AndroidStudio para internacionlizar la aplicación	36
30. Selector de idiomas para la aplicación	37
31. Traducción de strings en los idiomas seleccionados	37
32. Control internet AndroidManifest.xml	38
33. Estados alerta de la conexión a internet	38
34. Funciones control conexión a internet	39
35. Función principal Login Activity	39

36. Login Activity sin internet: dos y tres reintentos	40
37. Función registro del usuario	41
38. Advertencias y errores en el registro de usuario	42
39. Selección cuenta de Gmail para registro MeetMap	42
40. Registro MeetMap con cuenta de Google	43
41. Inicio sesión usuario y contraseña MeetMap	44
42. Función inicio sesión usuario y contraseña MeetMap	44
43. Función inicio sesión Google MeetMap	45
44. Vista para resetear contraseña	45
45. Función para resetear contraseña	46
46. Proceso para cambiar contraseña	46
47. Diseño del campo de contraseña	47
48. Navegación secciones principales	47
49. Botón hamburguesa y despliegue menú lateral	48
50. Lógica de navegación entre secciones del menú lateral	49
51. Proceso carga Map Fragment en la actividad principal	50
52. Obtención de la información de la Api de Madrid	51
53. Función carga de datos de la Api de Madrid	51
54. Función Observe() fragmento mapa	52
55. Creación de la lista de locators filtrada	53
56. Función ApplyFilter() fragmento mapa	53
57. Mapa sin filtros y con filtros	54
58. Función onInfoWindowClick() fragmento mapa	54
59. Vista tarjeta información al seleccionar marcador	55
60. Vista actividades a menos de 1Km de la ubicación del usuario	55
61. Función distance() fragmento mapa	56
62. Cambio de fragmento con información de la actividad seleccionada	56
63. Selección iconos según categoría listado actividades a menos de 1Km	57
64. Indicación para pulsar sobre la ubicación del usuario	58
65. Antes y después de la implementación del clustering en el diseño del mapa	58
66. Vista detalle actividad en diferentes estados	59
67. Transferencia información mapa-detalle actividad	59
68. Validación campos actividades procedentes de la API de Madrid	60
69. Selección imagen personalizada por categoría	61
70. Función: añadir la actividad a favoritos	61
71. Actualización de la base de datos con favoritos	62
72. Función: eliminar actividad de favoritos	62

73. Función: comprobar si la actividad está en favoritos	63
74. Obtención del usuario para la suscripción a la actividad	63
75. Función: verificación suscritos	64
76. Selección imágenes aleatorias para burbujas de suscritos	65
77. Contador de personas suscritas	65
78. Retardo para la muestra de botones de suscripción	66
79. Vista de MeetMappers (personas suscritas)	66
80. Función: actualización de suscriptores	67
81. Obtención de la imagen para la lista de suscriptores	67
82. Función: apertura vista perfil usuario suscrito	68
83. Separación individual de los suscriptores	68
84. Diseño toolbar vista suscriptores	68
85. Vista perfil usuario suscrito	69
86. Función: obtención datos para ser mostrados en la vista	70
87. Personalización título appbar en la vista perfil de usuario suscrito	70
88. Vista favoritos	71
89. Creación objeto con los datos de actividades favoritas	71
90. Vista favoritos cuando no hay ninguna actividad guardada	72
91. Función: carga de actividades favoritas	72
92. Selección ícono en función de categoría de la actividad	73
93. Clase interna para obtener los datos de favoritos	74
94. Escucha del botón del corazón para eliminar la actividad de favoritos	74
95. Escucha ítem de actividad favorita para abrir detalle de actividad	74
96. Función: actualización lista de favoritos y su visibilidad	75
97. Comprobación de la existencia de chats iniciados	75
98. Vista de la lista de chats cuando no se ha iniciado ninguno	76
99. Creación del contenido del ítem chat en el adapter	77
100. Vista lista de chats con último mensaje	77
101. Función para guardar chat en Firebase	78
102. Acceso desde la lista de chats al chat individual	78
103. Inicio vista chat individual con recuperación de mensajes	79
104. Barra de navegación chat individual	80
105. Diseño vista chat individual	80
106. Función enviar mensaje en chat individual	81
107. Comprobación procedencia mensaje	81
108. Obtención de los datos del usuario al iniciar la vista de editar perfil	82
109. Diseño redondeado de la imagen del perfil	82

110. Editar imagen perfil con apertura de galería del dispositivo móvil	83
111. Guardar los cambios de la imagen y campos del perfil	83
112. Función del botón guardar para salvar cambios del perfil	83
113. Actualización de los datos del usuario para guardarlos en Firebase	84
114. Función del botón cancelar para mantener datos anteriores	84
115. Edición del perfil y visualización de los cambios	84
116. Vista preguntas frecuentes para los usuarios	85
117. Estilo enlace desarrollador y conexión GitHub	85
118. Vista de contacto con los desarrolladores	86
119. Cierre de sesión	86
120. Creación de alerta y eliminación de cuenta de Firebase	87
121. Alerta para la eliminación de cuenta	87
122. Diseño menú lateral con botón transparente	88
123. Lógica visibilidad menú lateral y botón transparente	89
124. Tratamiento lista mutable para clustering	90
125. Código para mostrar las actividades cuando están agrupadas	90
126. Selección marker y listado actividades	91
127. Creación original markers	92
128. Código para aplicar filtros	92
129. Autenticación usuario Firebase	93
130. Datos del usuario predefinidos con el registro	94
131. Datos del usuario después de actualizar información del perfil	94
132. Imágenes empleadas por el usuario en su perfil	94
133. Actividad guardada como favorita por el usuario	95
134. Suscripción del usuario a la actividad anterior	95
135. Inicio chat con otro usuario suscrito a la actividad	95
136. Solución código actualizar imagen perfil	97
137. Errores Google play console	97
138. Código del primer error: datos perfil usuario	97
139. Código del segundo error: verificación favoritos	98
140. Error petición API de Madrid	98
141. Código error al seleccionar ítem lista actividades con cluster	99

1. INTRODUCCIÓN

Este documento responde a la realización del módulo de Proyecto del CFGS en el Ciclo de Desarrollo de Aplicaciones Multiplataforma.

El objetivo principal de este proyecto es la implementación de una aplicación móvil (en adelante nos podremos referir a ella como app) con un diseño accesible, que muestre las actividades de ocio en los próximos cien días en la ciudad de Madrid teniendo en cuenta la geolocalización del dispositivo móvil y, de esta forma, poner en contacto a personas con gustos similares a través del ocio local, sostenible y cultural en Madrid.

Otros objetivos más a largo plazo es conseguir que los usuarios tengan esta aplicación como referente de ocio y creación de comunidad social y que se pueda extender su implementación a otras localidades nacionales e internacionales.

La idea de implementar esta aplicación radica en los cambios de ocio y consumo que han surgido a raíz de la pandemia provocada por el virus Covid-19¹. Esta situación provocó el aumento de descargas de aplicaciones móviles para conocer gente² y nuevas formas de socializar empleando la tecnología³. Estos hechos sumados a la cantidad abrumadora de planes de ocio disponibles en la ciudad de Madrid pueden hacer del desarrollo de este proyecto una buena herramienta para los habitantes de Madrid.

¹ "Un estudio de la consultora McKinsey en mayo de 2020 ya revelaba que en ocho semanas de pandemia habíamos avanzado el equivalente a cinco años en adopción digital por parte de consumidores y empresas. Este salto digital es más marcado dependiendo del sector: en envío a domicilio de compras online avanzamos el doble, 10 años en 8 semanas; y en entretenimiento, el equivalente a 7 años se logró en 5 meses. Para darnos una idea, a Disney Plus le llevó dos meses alcanzar los cinco millones de suscriptores que le costaron 7 años a Netflix. (...)

Según el informe global Digital 2021, de We are social, una agencia de publicidad, las redes sociales crecieron un 13,2% más que durante el año anterior. 490 millones de usuarios nuevos se incorporaron a ellas haciendo un total de 4.200 millones en los primeros meses de 2021. Aumenta también el tiempo que los usuarios dedican a las redes sociales en un 0,4%, y es de 2 horas y 25 minutos promedio." Gonzalo, M. (2021, 17 de marzo). La pandemia que nos volcó a las redes. Newtral. <https://www.newtral.es/pandemia-redes-sociales-digitalizacion-covid-19/20210317/>

² "Según la investigación, en los últimos meses se ha apreciado un aumento en la actividad de los usuarios que ya utilizaban estas aplicaciones antes de la crisis sanitaria. Ha incrementado la cantidad de mensajes, las citas a través de videoconferencia y otras funcionalidades implementadas durante la cuarentena, como el 'Pasaporte de Tinder', que permite a las personas conectarse a escala global." Redacción (2020, 4 de junio). Así han evolucionado las aplicaciones de citas online durante el confinamiento. ReasonWhy. <https://www.reasonwhy.es/actualidad/aplicaciones-citas-online-cambios-confinamiento-coronavirus>

³ "Lo que está claro es que las aplicaciones van a seguir usándose, tal como ya lo hacían antes de la pandemia. Y, además -como hemos comentado ya- las redes sociales también se empiezan a usar con el fin de ligar. Según la encuesta anterior, "un 21% de los participantes afirma que ahora más que nunca recurrirá a los entornos virtuales, como las aplicaciones de citas, para encontrar pareja, mientras que para el 79% seguirá siendo una herramienta clave en su búsqueda de pareja". G.Portalatín, B. (2022, 22 de marzo). Amor en Tinder (y otras apps de citas): cómo la pandemia ha condicionado la forma de ligar de los 'millennials'. LaSexta. https://www.lasexta.com/bienestar/sexualidad/amor-tinder-otras-apps-citas-como-pandemia-condicionado-forma-ligar-millennials_20220322623873cbf6355200015c9dc4.html

Para conseguir estos objetivos, MeetMap pretende ser una aplicación móvil que ofrezca, a través de una interfaz accesible y amigable, diferentes actividades y eventos de la ciudad de Madrid dispuestos en un mapa que posibilite la interacción del usuario. Esta aplicación intentará enmarcarse en los objetivos de desarrollo sostenible de la Agenda 2030 ofreciendo actividades sostenibles y fomentando el ocio local y de proximidad. Por otra parte, MeetMap hará posible que los usuarios de esta aplicación puedan ponerse en contacto y conocerse para disfrutar de estos planes de forma conjunta.

La aplicación contará con un mapa que permita visualizar de manera clara y detallada los diferentes planes y eventos en la ciudad, mostrando la ubicación del usuario, lo que facilitará la búsqueda de planes cercanos. Además, se podrá filtrar por categorías y temáticas para que el usuario encuentre rápidamente los planes que más le interesen.

Por otro lado, los usuarios podrán guardar sus planes favoritos, editar su perfil y filtrar los planes en función de la temática y la cercanía. Este proyecto quiere contar con un diseño intuitivo y una interfaz fácil de usar, para que los usuarios puedan navegar rápidamente por los planes disponibles.

Otra característica clave de esta aplicación será la posibilidad de contactar con las personas que se han suscrito a una actividad y poder hablar con ellas a través de un chat integrado. De esta manera, los usuarios podrán conocer a nuevas personas con intereses similares y ampliar su círculo social.

Además, la aplicación incluirá una sección de perfil donde los usuarios tendrán la posibilidad de agregar información sobre sus intereses y preferencias. También se implementará la posibilidad de guardar los planes favoritos de los usuarios para que puedan acceder a ellos fácilmente en cualquier momento.

MeetMap con una imagen juvenil y un desarrollo muy accesible, quiere dirigirse a todo tipo de público de cualquier edad ofreciendo una información detallada de cada plan con la posibilidad de suscribirse y poder ver las personas que están apuntadas a esta misma actividad. De esta forma se quiere conseguir hacer accesible un ocio alternativo con muchas variedades de planes que puedan ser del gusto de personas de diferentes edades y, a partir del filtrado de actividades, también se podrá conectar a personas con intereses similares.

Las actividades y eventos que se van a mostrar en esta aplicación serán obtenidos de una API del Ayuntamiento de Madrid que ofrece Actividades Culturales y de Ocio Municipal en los próximos 100 días. Los planes se van actualizando de forma constante y la aplicación los presentará en un mapa para que los usuarios puedan conocer las actividades más cercanas a su ubicación, así como el resto de los eventos que se pueden realizar en toda la ciudad de Madrid.

En definitiva, el desarrollo de esta aplicación se va a llevar a cabo para ofrecer una experiencia divertida, intuitiva y amigable a los usuarios, que aúna la parte de ocio y la parte social.

A continuación, se va a explicar el desarrollo del proyecto de la aplicación móvil, y para una revisión más detallada del código implementado, se puede consultar el [repositorio de GitHub del proyecto MeetMap](#).

2. CONTEXTO FUNCIONAL Y TECNOLÓGICO

2.1 Contexto funcional

MeetMap es una aplicación móvil que tiene como público objetivo personas que hacen uso habitual del dispositivo móvil y con conocimientos en aplicaciones y herramientas móviles relacionadas con el ámbito social y de ocio. Esta aplicación está enfocada a un público mayoritariamente sin pareja, con dificultades para conocer gente nueva, sin conocimientos amplios sobre los planes sociales y culturales de la ciudad de Madrid, tanto hombres como mujeres entre 25 y 70 años y de cualquier nivel socioeconómico.



Carmen Rodríguez

Sevilla

28 años

- Estudiante de medicina
- Recién llegada a Madrid
- Becada Séneca
- Muy sociable
- Intereses culturales: arte, música y escritura

Uso aplicaciones móviles y redes sociales

- Muy activa en redes sociales
- Tiene un canal de Youtube sobre pop español de los años 90 y 2000
- Ha colaborado en alguna tertulia radiofónica sobre la generación millennial

Ejemplo 1 de público objetivo



Isak Johansson

Suecia

25 años

- Trabaja en el sector hostelería
- Lleva dos meses viviendo en Madrid
- Comparte piso con estudiantes ERASMUS
- Músico callejero
- Intereses culturales: música, arte urbano y fotografía
- Quiere conocer personas españolas para...

Uso aplicaciones móviles y redes sociales

- Tiene una cuenta en Instagram para dar a conocer su música y sus fotografías
- Muy activo en TikTok

Ejemplo 2 de público objetivo

Si el usuario hace uso de esta aplicación podrá realizar las siguientes acciones (casos de uso):

- Registrarse
- Iniciar sesión si ya se ha registrado
- Cambiar la contraseña si no se recuerda
- Verificar el email cuando se registra
- Navegar entre las diferentes secciones principales de la aplicación: mapa, favoritos y chat
- Navegar entre los diferentes apartados relacionados con el propio usuario y la sesión
 - o editar perfil: cambiar imagen, actualizar datos (nombre, apellidos, teléfono, descripción)
 - o preguntas frecuentes
 - o contactar con los creadores y desarrolladores
 - o cerrar sesión
 - o eliminar cuenta
- Informarse sobre los eventos culturales de la ciudad de Madrid según la ubicación y la proximidad
- Filtrar los diferentes planes en función de la temática
- Consultar los detalles de cada evento desde diferentes secciones de la aplicación
- Seleccionar y deseleccionar el plan como favorito
- Suscribirse a la actividad
- Consultar personas que se han inscrito a cada actividad
- Ver el perfil de las personas inscritas
- Chatear con las personas inscritas en una actividad o con las personas registradas en la app de forma individual

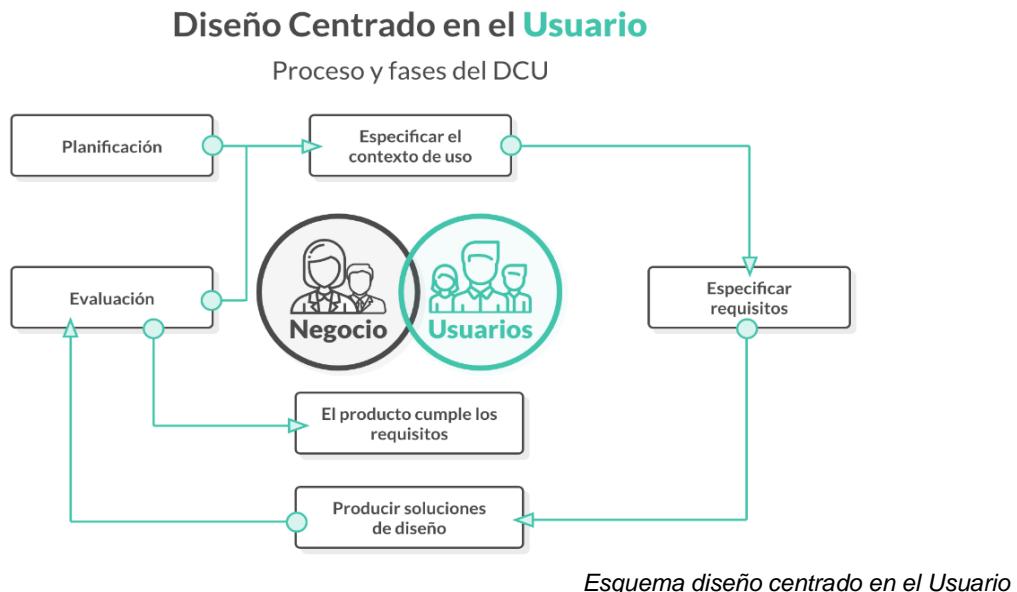
2.2 Contexto tecnológico

Para satisfacer todos los casos de uso disponibles para el usuario de la aplicación, MeetMap debe tener una serie de características (*feature list*):

- Conexión a internet de forma permanente mientras se usa la aplicación
- Acceso a la ubicación en tiempo real del usuario de la aplicación para poder mostrarle los planes más cercanos
- Acceso a la galería de su dispositivo móvil para poder cambiar la imagen del perfil
- Creación y uso de base de datos con las siguientes utilidades:

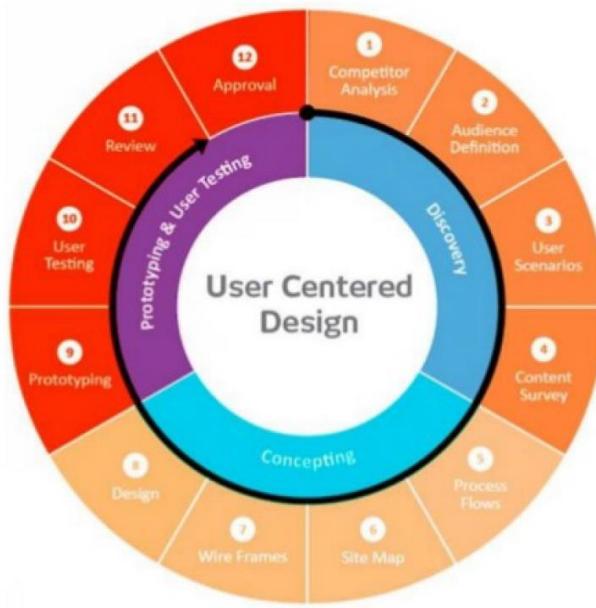
- registrar a todos los usuarios y hacer las comprobaciones necesarias para el buen funcionamiento de la app
- recoger los datos de cada usuario para el perfil
- reunir todas las imágenes de perfil de los usuarios
- almacenar las personas inscritas en cada actividad
- registrar los chats iniciados por cada usuario
- guardar por cada usuario si tiene marcado como favorito o no cada actividad
- Uso y consumo de la API REST datos.madrid.es para obtener la agenda de actividades culturales y de ocio municipal en los próximos 100 días en la ciudad de Madrid.

Para que la aplicación se encuadre dentro de estos contextos: funcional y tecnológico, es necesario que el proceso de desarrollo de esta aplicación cuente con un diseño centrado en el usuario y su experiencia⁴ y para ello se van a emplear las metodologías ágiles que permiten una colaboración y coordinación continua entre los desarrolladores y los clientes, teniendo en cuenta las fases de desarrollo y la retroalimentación de los usuarios.

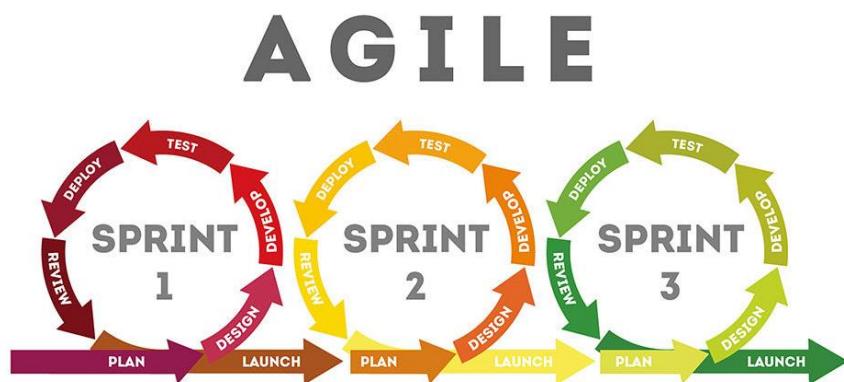


⁴ "El Diseño Centrado en el Usuario es una filosofía de diseño que tiene como objetivo la creación de productos que satisfagan las necesidades y resuelvan los problemas de los usuarios a los cuales está destinado dicho producto.

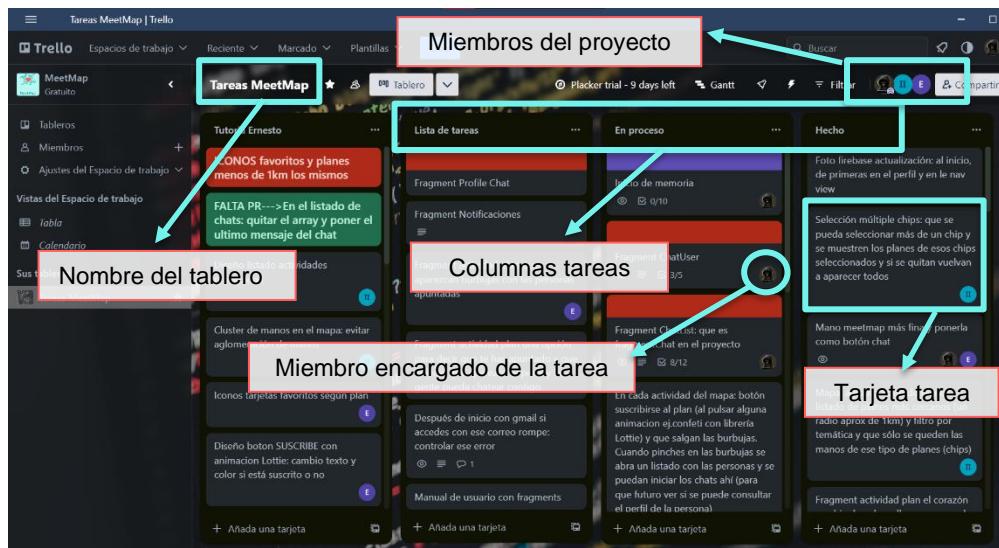
El Diseño Centrado en el Usuario tiene su origen en la Ingeniería del Software, el diseño industrial y militar. Don Norman, cofundador de Nielsen Norman Group, fue la primera persona que comenzó a utilizar el término Diseño Centrado en el Usuario. El término DCU es utilizado como método para el diseño de interfaces de usuario. Cuando se diseña siguiendo la filosofía DCU el usuario debe de situarse en el centro de todo. No es posible concebir el producto desligado de su uso, el contexto o las necesidades del usuario final." Arias del Prado, J. (2020, 4 de marzo). Diseño Centrado en el Usuario (DCU). Todas las claves del proceso. Blog.UXABLES. <http://www.uxables.com/diseno-ux-ui/diseno-centrado-en-el-usuario-dcu-todas-las-claves-del-proceso/>

*Diagrama diseño centrado en el Usuario*

Para la elaboración de este proyecto se han seguido los doce principios del manifiesto Agile⁵ y se ha empleado como metodología ágil de trabajo el tablero Kanban para llevar un control exhaustivo de las tareas a realizar. La coordinación entre los diferentes miembros del equipo de desarrollo se ha llevado a cabo mediante reuniones semanales (*sprints*) donde se han ido definiendo los siguientes pasos que se tenían que realizar y se han efectuado las pruebas oportunas para dar por concluidas las tareas que estaban asignadas para cada *sprint*. Estas reuniones se han tenido de formas telemática y, en determinadas ocasiones, de forma presencial.

*Etapas sprint metodologías ágiles*

⁵ Cunningham, W. (2001) Principios del Manifiesto Ágil. agilemanifesto.org/iso/es/principles.html



Tablero Kanban de la herramienta Trello

EditProfile activity

en la lista **Hecho**

Miembros Notificaciones Siguendo ✓

Fechas 20 de feb. - 6 de mar. a las 13:21 cumplida ✓

Start time: 8:00 AM Duration: 2w 4h 21m Status: Open Effort (e): 2w 4h 21m

NOTA: el email, ¿no se tendría que poder modificar?

Checklist Ocultar los elementos marcados Eliminar

- Imagen redonda perfil
- Campos obligatorios: email y contraseña
- Campos opcionales: descripción, name, numTel, foto, surname
- Recoger datos
- Guardar datos en la BBDD
- Diseño con scroll
- Text-area para la descripción
- Si se da al botón eliminar cuenta que se elimine de la base de datos

Añada un elemento

Power-Ups

- Add dependency
- Add plan attribute
- Añadir Power-Ups

Automatización

- Añadir botón

Acciones

- Mover
- Copiar
- Crear plantilla
- Archivar

Detalle de tarea en herramienta Trello

3. PLANIFICACIÓN DEL PROYECTO

Antes de comenzar con la explicación de la planificación del proyecto es necesario hacer un breve comentario sobre el proceso que ha seguido el desarrollo de esta aplicación móvil.

MeetMap surgió como idea de desarrollo de aplicación móvil para el proyecto integrador llevado a cabo de forma coordinada entre los módulos de Desarrollo de Interfaces y de Programación Multimedia y Dispositivos Móviles. Durante esta etapa del proyecto se comenzó con las primeras tareas de desarrollo de la aplicación que fueron realizadas por Almudena Fernández Cárdenas, Iker Iturrealde Tejido y Nerea Ramos Escobar.

Posteriormente a la entrega de esta primera fase del proyecto, el equipo de desarrollo sufrió un cambio de miembros y Nerea Ramos Escobar dejó de pertenecer al equipo y se sumó a este proyecto Daniel García Ayala.

Esta aclaración se debe tener en cuenta a la hora de analizar la planificación del proyecto, porque, desde el inicio de la primera fase hasta la entrega final de la aplicación, se ha realizado sin solución de continuidad.

En la primera fase (16 de enero al 1 de marzo de 2023) se desarrolló el esqueleto de la aplicación:

- Creación de *Firebase*.
- Vistas de *splash*, inicio, registro, inicio de sesión.
- Navegación del menú principal.
- Navegación del menú lateral y algunas de las secciones (editar perfil y preguntas frecuentes).
- Cierre de sesión y eliminación de la cuenta.
- Vista principal del mapa de actividades con un desarrollo básico.
- Vista de la actividad detallada sin completar.

En la segunda fase del proyecto (2 de marzo al 7 de junio de 2023) se ha ido ampliando la aplicación y mejorando el desarrollo de lo anteriormente descrito, para conseguir una aplicación completa y con una interfaz mejorada.

Para la planificación del proyecto hemos utilizado la aplicación *Trello* basada en la herramienta ágil de tablero Kanban mencionada anteriormente (Ver figura [Tablero Kanban de la herramienta Trello](#), pág.19). Como se puede observar las tareas se han dividido en: pendientes, en proceso y finalizadas y se han ido asignando según la lógica de creación de las vistas y teniendo en cuenta si alguna tarea era dependiente de otra para que se asignase de forma consecutiva. Las primeras tareas que se han tenido que realizar para poder avanzar en el desarrollo y diseño de la aplicación, así como para ir probando el funcionamiento de esta, han sido:

- Búsqueda y selección de imágenes sin derechos e iconos
- Creación de la base de datos en *Firebase*
- Elaboración del menú principal de navegación

En el tablero de *Trello* se han ido fijando las tareas que se iban a realizar asignando cada una de ellas a un miembro del equipo de desarrollo estableciendo un tiempo estimado para su consecución, así como una breve descripción de la misma y, si era necesario, una subdivisión de la tarea en hitos más específicos (Ver figura [Detalle de tarea en herramienta Trello](#), pág. 19).

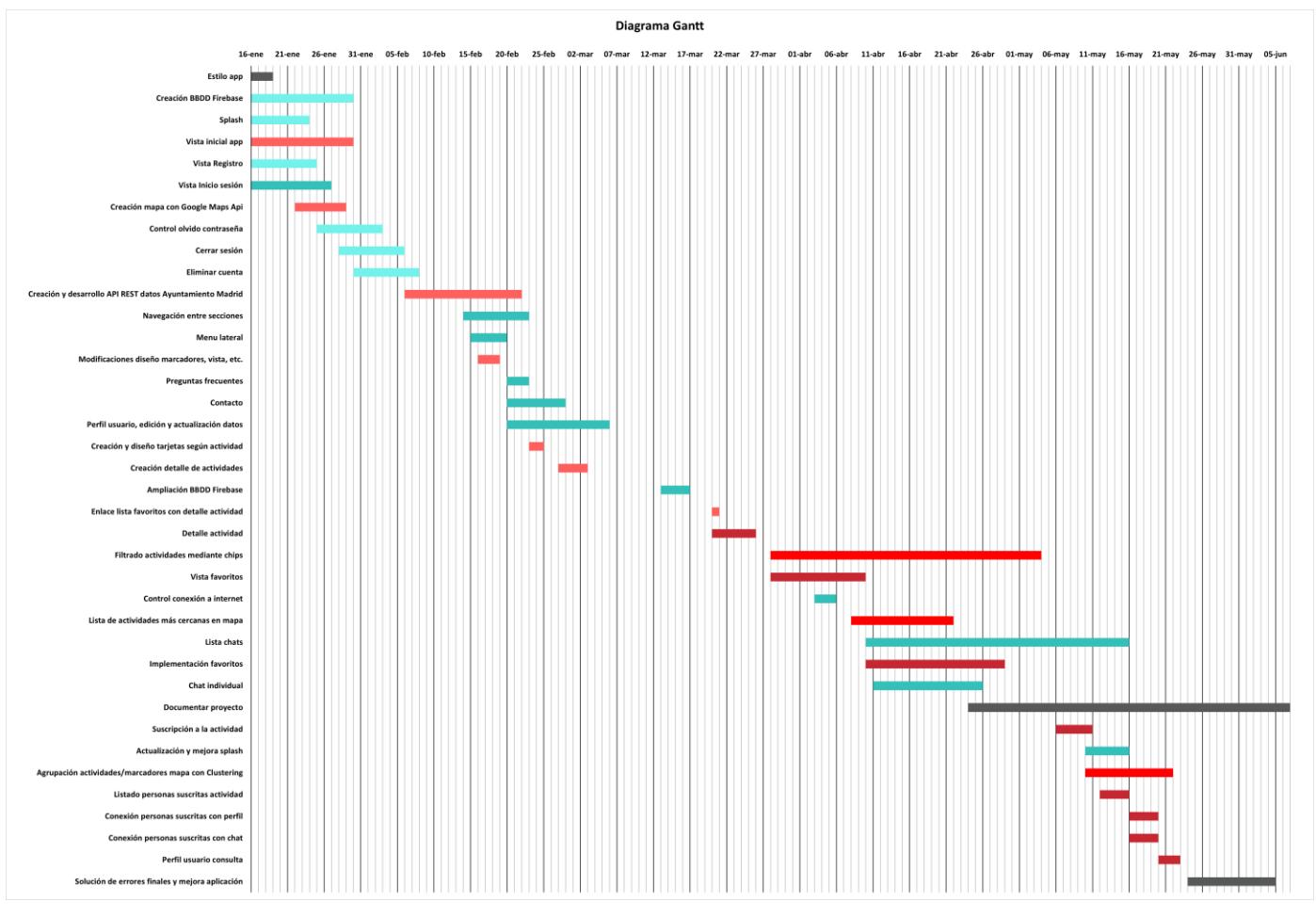
Los recursos generales necesarios para poder realizar las tareas han sido:

- Prototipo de la aplicación en [Figma](#)
- Paleta de colores para unificar el estilo y las vistas
- Imágenes e iconos
- Conexión con la plataforma *Firebase*
- Datos de las actividades de la Api del Ayuntamiento de Madrid

A continuación, se muestra una tabla con la lista de las tareas que se han realizado para implementar la aplicación de MeetMap con el tiempo empleado en cada una y con el desarrollador encargado de cada tarea. De forma consecutiva se muestra el Diagrama de Gantt elaborado a partir de esta lista.

TAREA	INICIO	DURACIÓN	FIN	ENCARGADO
Estilo app	16-ene	3	19-ene	Todos
Creación BBDD Firebase	16-ene	14	30-ene	Nerea
Splash	16-ene	8	24-ene	Nerea
Vista inicial app	16-ene	14	30-ene	Iker
Vista Registro	16-ene	9	25-ene	Nerea
Vista Inicio sesión	16-ene	11	27-ene	Almudena
Creación mapa con Google Maps Api	22-ene	7	29-ene	Iker
Control olvido contraseña	25-ene	9	03-feb	Nerea
Cerrar sesión	28-ene	9	06-feb	Nerea
Eliminar cuenta	30-ene	9	08-feb	Nerea
Creación y desarrollo API REST datos Ayuntamiento Madrid	06-feb	16	22-feb	Iker
Navegación entre secciones	14-feb	9	23-feb	Almudena
Menu lateral	15-feb	5	20-feb	Almudena
Modificaciones diseño marcadores, vista, etc.	16-feb	3	19-feb	Iker
Preguntas frecuentes	20-feb	3	23-feb	Almudena
Contacto	20-feb	8	28-feb	Almudena
Perfil usuario, edición y actualización datos	20-feb	14	06-mar	Almudena
Creación y diseño tarjetas según actividad	23-feb	2	25-feb	Iker
Creación detalle de actividades	27-feb	4	03-mar	Iker
Ampliación BBDD Firebase	13-mar	4	17-mar	Almudena
Enlace lista favoritos con detalle actividad	20-mar	1	21-mar	Iker
Detalle actividad	20-mar	6	26-mar	Daniel
Filtrado actividades mediante chips	28-mar	37	04-may	Iker
Vista favoritos	28-mar	13	10-abr	Daniel
Control conexión a internet	03-abr	3	06-abr	Almudena
Lista de actividades más cercanas en mapa	08-abr	14	22-abr	Iker
Lista chats	10-abr	36	16-may	Almudena
Implementación favoritos	10-abr	19	29-abr	Daniel
Chat individual	11-abr	15	26-abr	Almudena
Documentar proyecto	24-abr	44	07-jun	Todos
Suscripción a la actividad	06-may	5	11-may	Daniel
Actualización y mejora splash	10-may	6	16-may	Almudena
Agrupación actividades/marcadores mapa con Clustering	10-may	12	22-may	Iker
Listado personas suscritas actividad	12-may	4	16-may	Daniel
Conexión personas suscritas con perfil	16-may	4	20-may	Daniel
Conexión personas suscritas con chat	16-may	4	20-may	Daniel
Perfil usuario consulta	20-may	3	23-may	Daniel
Solución de errores finales y mejora aplicación	24-may	12	05-jun	Todos

Lista de tareas MeetMap

*Diagrama Gantt desarrollo MeetMap*

4. DESCRIPCIÓN DE LA SOLUCIÓN: ANÁLISIS Y DISEÑO

4.1 Especificación de requisitos

Los requisitos necesarios para poder desarrollar y hacer las pruebas pertinentes de este proyecto son:

- Dispositivo móvil con sistema operativo *Android*
- Versión de API de *Android* 33 o superior
- Conexión a internet

4.2 Selección de plataforma tecnológica

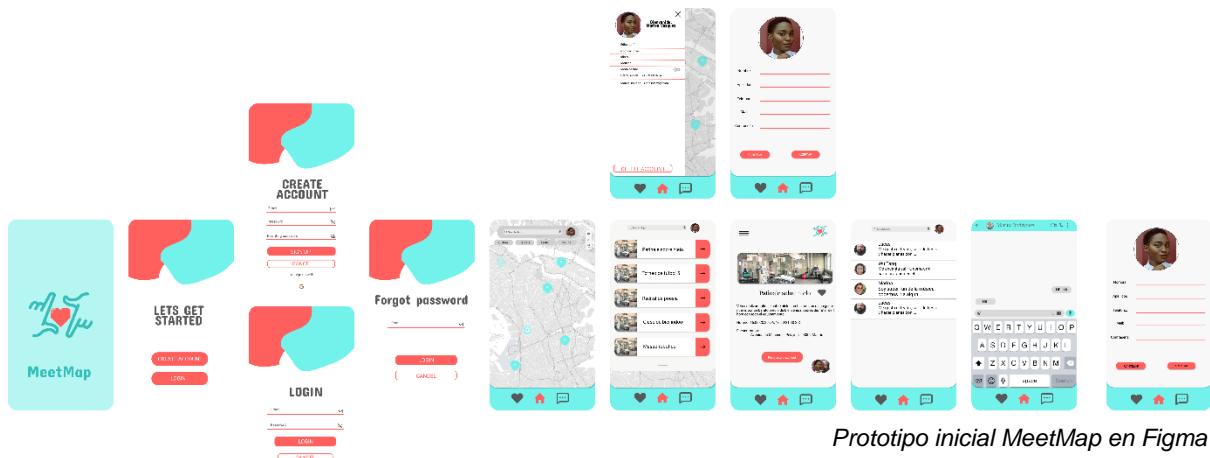
La plataforma tecnológica seleccionada y que da soporte para cumplir con los requisitos anteriores son:

- IDE *Android Studio*
- Dispositivos móviles para poder emular la aplicación
- *Firebase* para comprobar la creación de la base de datos empleada
- API REST del Ayuntamiento de Madrid

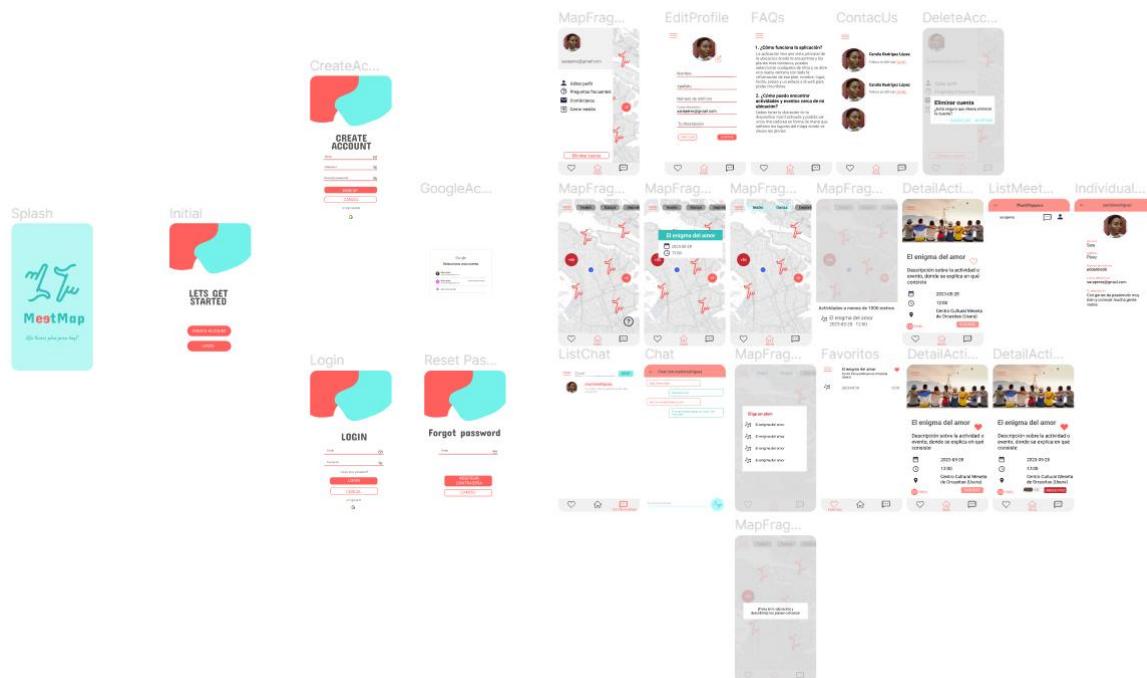
4.3 Descripción del diseño de la solución

A la hora de desarrollar esta aplicación se ha estudiado, analizado y descrito el diseño de esta para tener un punto de partida a la hora de elaborar el proyecto. Para este cometido, se ha empleado la herramienta de diseño [Figma](#) a través de la cual se elaboró un prototipo inicial de aplicación con la implementación de colores, iconos, tipologías, componentes y el contenido de cada vista de la app.

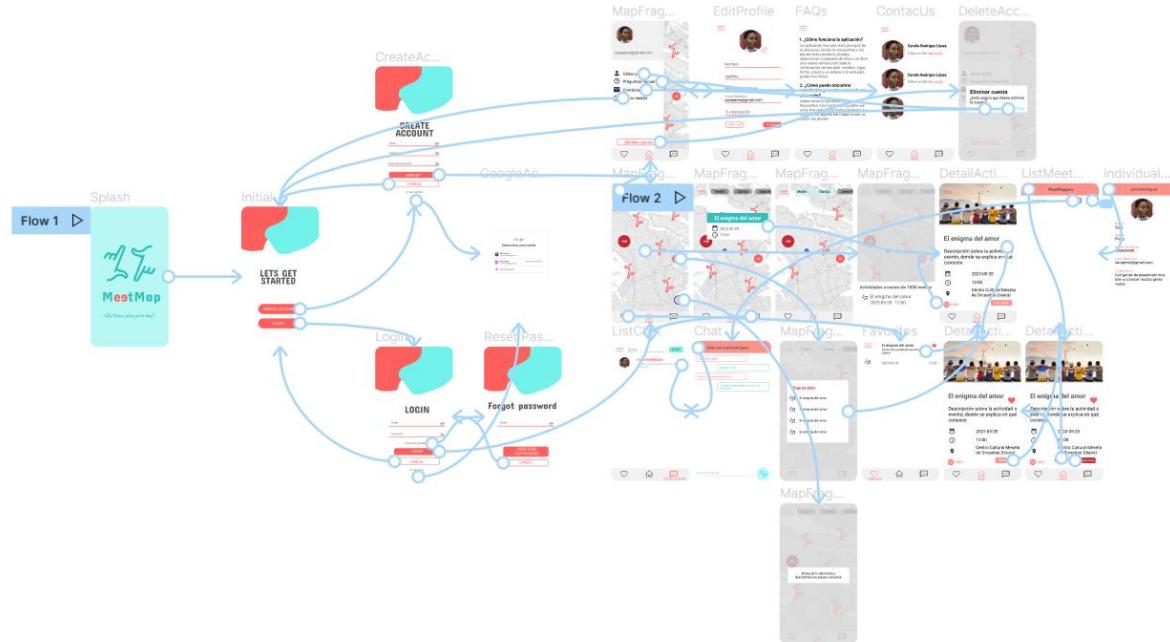
Con este prototipo se empezó a trabajar en las primeras etapas de desarrollo del proyecto y dio lugar a establecer el esqueleto general de la aplicación sobre el cual, posteriormente se fueron haciendo modificaciones de estilo y funcionalidad.



A medida que se ha ido desarrollando el proyecto, este diseño ha sufrido muchos cambios de estilo y ha sido ampliado con la elaboración de un prototipo final que se asemeja en mayor medida al resultado obtenido después de implementar la aplicación. En esta imagen se puede observar un proyecto más amplio y con un estilo más limpio y definido dando lugar a una visión más fiel de la realidad de la aplicación.

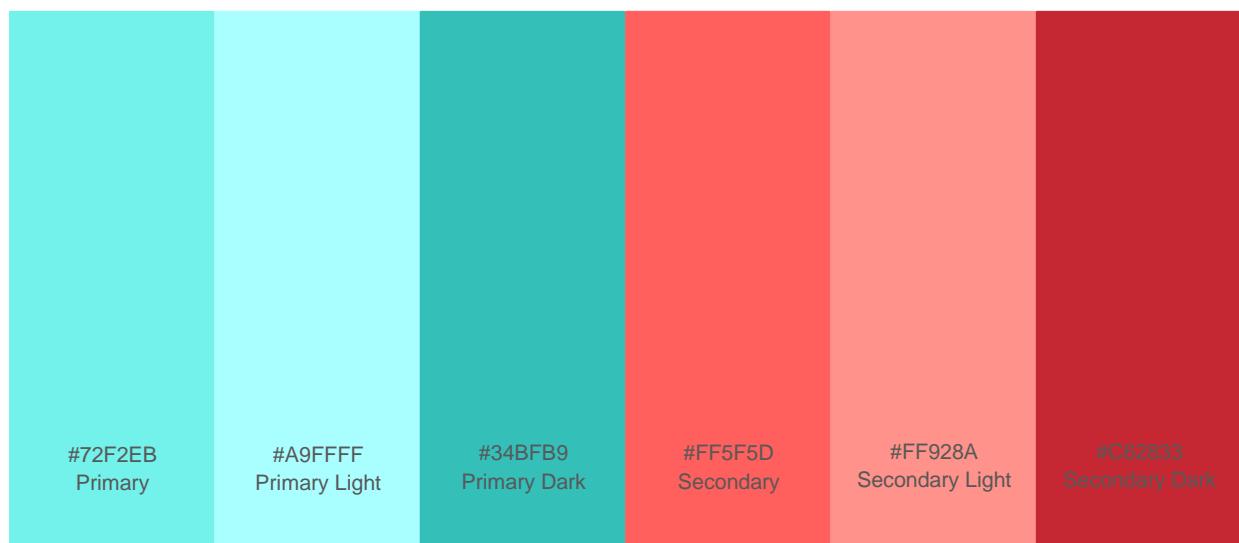


A través de [Figma](#) también se ha establecido el flujo de navegación de la aplicación entre las diferentes actividades y componentes (botones, tarjetas, menús, etc.) y de esta forma todos los desarrolladores han conocido en cada momento la funcionalidad que tenían que implementar para que el proyecto avanzase.



Flujo de navegación MeetMap en Figma

Para el diseño de esta aplicación móvil se ha empleado la siguiente paleta de colores que refleja un estilo claro y alegre. Se ha jugado con la combinación de los colores primario y secundario y sus diferentes tonalidades claras y oscuras.



Paleta de colores MeetMap

Para esta aplicación se han usado dos fuentes en sus diferentes formatos Regular y Bold y sus diferentes tamaños, obtenidas de la herramienta *Google Fonts*:

Concert One

Roboto

La fuente **Concert One** se ha empleado para los títulos de las vistas iniciales: Splash, Initial, Login y SignUp. Para el resto de los textos se ha empleado la fuente Roboto. Y, de forma puntal, para el eslogan se ha empleado la fuente *Satisfy*.

Los colores empleados en los textos de la aplicación han sido analizados con una herramienta de accesibilidad de [Adobe Color](#) para conseguir un contraste adecuado entre el color del fondo y el color de la fuente:

The image contains two side-by-side screenshots of the Adobe Color website's accessibility tools.

Top Screenshot (Color Scheme: #575858 Text over #72F2EB Background):

- Color de texto:** #575858 (black square)
- Color de fondo:** #72F2EB (light blue square)
- Proporción de contraste:** 5,31 / 1 (green checkmark)
- Vista previa:**
 - Texto normal:** Con un alto contraste de color todo es más fácil de leer
 - Texto grande:** Con un alto contraste de color todo es más fácil de leer
 - Componentes gráficos:** Icons (square, circle, triangle) are visible.
- Resultados:** Pass (green checkmark) for Text and graphics, Pass (green checkmark) for Text and background, Pass (green checkmark) for Icons and graphics.

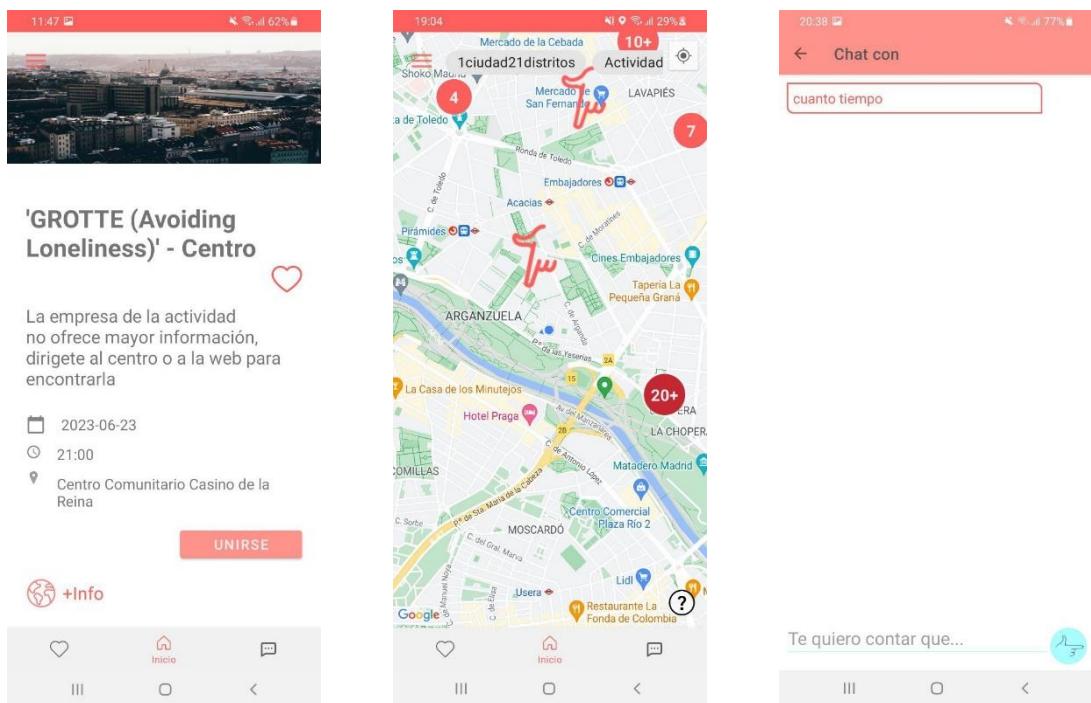
Bottom Screenshot (Color Scheme: #575858 Text over #FF928A Background):

- Color de texto:** #575858 (black square)
- Color de fondo:** #FF928A (red-orange square)
- Proporción de contraste:** 3,31 / 1 (red X)
- Vista previa:**
 - Texto normal:** Con un alto contraste de color todo es más fácil de leer
 - Texto grande:** Con un alto contraste de color todo es más fácil de leer
 - Componentes gráficos:** Icons (square, circle, triangle) are visible.
- Resultados:** Fallo (red X) for Text and background, Pass (green checkmark) for Text and graphics, Pass (green checkmark) for Icons and graphics.

Análisis accesibilidad colores MeetMap

Para la elección de iconos⁶ y de componentes (botones, tarjetas, chips, etc.) de la aplicación móvil de este proyecto se han seguido las recomendaciones de estilo de los estudios material⁷ de Google.

Para concluir con el estilo, a la hora de diseñar la app, cabe mencionar el uso inmersivo de las imágenes empleadas en la vista de detalle de la actividad, así como la personalización de los marcadores del mapa y del botón enviar del chat donde se ha empleado la imagen de las manos tan característica de MeetMap.



Personalización diseño MeetMap

Para poder diseñar de forma correcta esta aplicación móvil se ha necesitado establecer los diferentes estados en los que se puede encontrar el usuario: no registrado y registrado, y las diferentes interacciones del usuario cuando utiliza la aplicación. Para poder visualizar de forma gráfica todo lo anteriormente descrito ello se ha elaborado un diagrama de Casos de Uso.

⁶Obtenidos de la plataforma Flaticon <https://www.flaticon.es/>

⁷Estos estudios de diseño se pueden consultar online: Google. (s.f.). Material Studies. <https://m2.material.io/design/material-studies/about-our-material-studies.html>

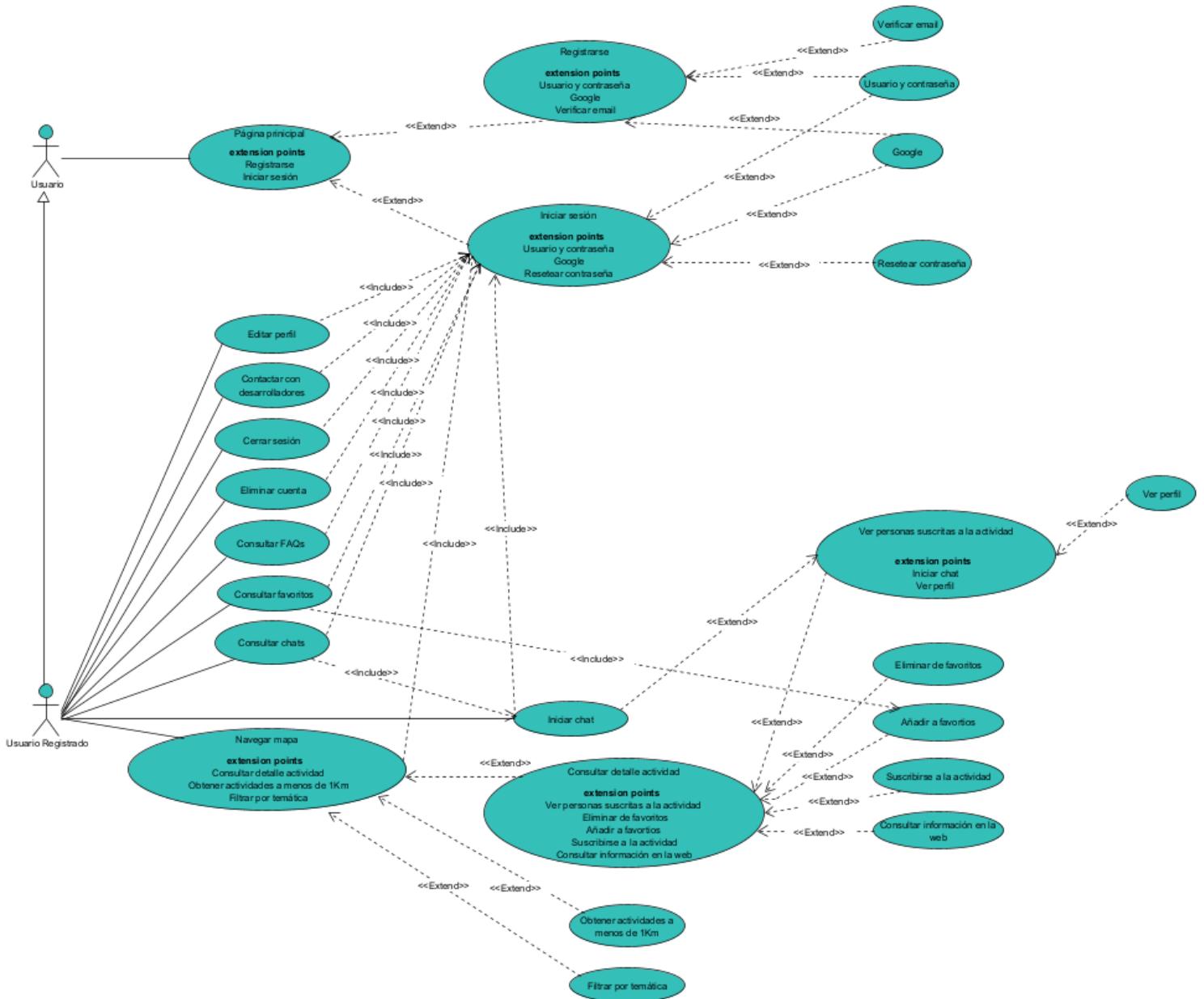


Diagrama de Casos de Uso

Para el diseño de la solución de este proyecto también se ha necesitado elaborar un diagrama de Colecciones con las diferentes tablas que debe tener la base de datos implementada en *Firebase*⁸.

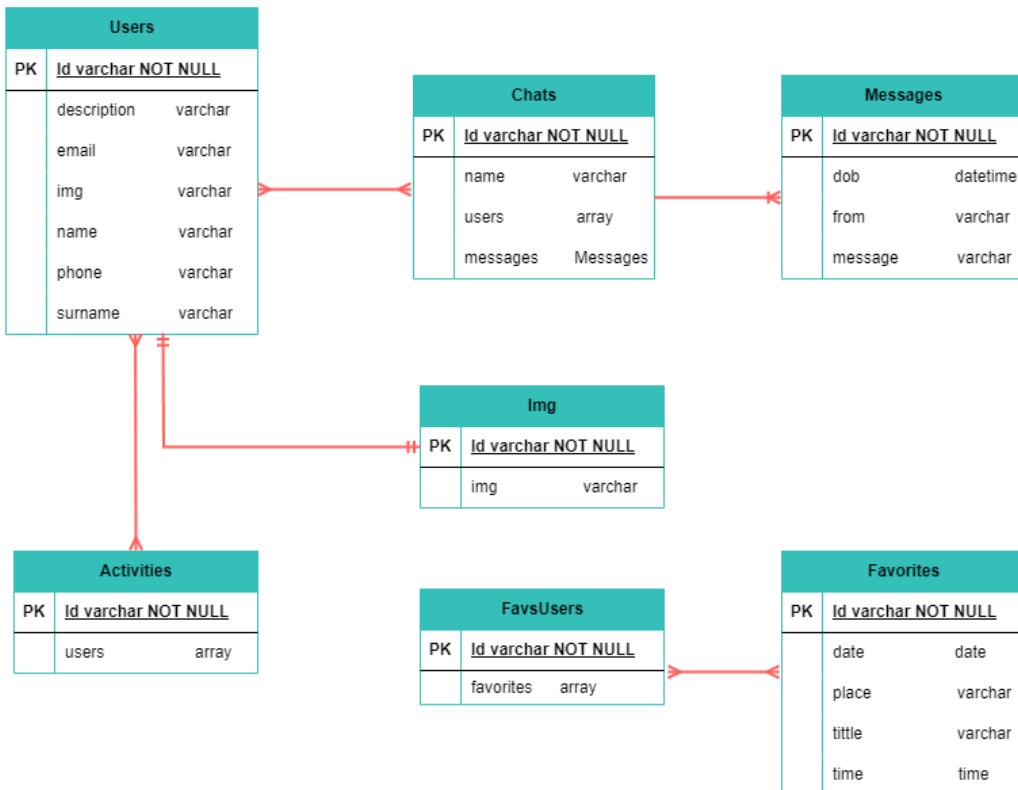


Diagrama de Colecciones

⁸ Firebase es una plataforma de desarrollo de aplicaciones móviles y web creada por Google. Proporciona una amplia gama de servicios y herramientas para ayudar a los desarrolladores a construir, mejorar y escalar aplicaciones de manera eficiente. Firebase ofrece características como autenticación de usuarios, almacenamiento en la nube, bases de datos en tiempo real, alojamiento web, mensajería en la nube, análisis de datos y pruebas de aplicaciones. Su objetivo es simplificar el desarrollo de aplicaciones, permitiendo a los desarrolladores centrarse en la lógica de la aplicación en lugar de preocuparse por la infraestructura subyacente.

5. DESCRIPCIÓN DE LA SOLUCIÓN: CONSTRUCCIÓN

5.1 Documentación descriptiva

A continuación, se va a describir de forma exhaustiva cada vista y funcionalidad de la aplicación mostrando por cada actividad del proyecto las capturas de imagen necesarias del código implementado y del resultado de este.

➤ Splash Activity

Cuando la aplicación se inicia aparece un fondo con el color primario de la aplicación para seguidamente mostrarse una animación en la que las dos manos (isotipo, con el color secundario en formato oscuro) que forman parte del imagotipo se van acercando para posteriormente mostrarse el logotipo de MeetMap con un eslogan, combinando los colores primario y secundario en modo oscuro. Cuando terminan las dos animaciones y se observa el imagotipo completo se inicia de forma secuencial la siguiente vista.

```

    val animLeftHand = AnimationUtils.loadAnimation( context: this, R.anim.move_left_to_right)
    val animRightHand = AnimationUtils.loadAnimation( context: this, R.anim.move_right_to_left)
    val animtext = AnimationUtils.loadAnimation( context: this, R.anim.fadein)

    title.visibility = View.INVISIBLE
    slogan.visibility = View.INVISIBLE

    val handsAnimationListener = object : Animation.AnimationListener {
        override fun onAnimationStart(animation: Animation?) {}

        override fun onAnimationEnd(animation: Animation?) {
            // Se ejecuta cuando la animación de las manos termina
            title.visibility = View.VISIBLE
            slogan.visibility = View.VISIBLE
            title.startAnimation(animtext)
            slogan.startAnimation(animtext)
        }

        override fun onAnimationRepeat(animation: Animation?) {}
    }

    // Establece el AnimationListener en ambas animaciones de las manos
    animLeftHand.setAnimationListener(handsAnimationListener)
    animRightHand.setAnimationListener(handsAnimationListener)
    // Inicia las animaciones de las manos
    leftHand.startAnimation(animLeftHand)
    rightHand.startAnimation(animRightHand)
}

```

Código Splash Activity

Como se puede observar en este fragmento de código existen tres animaciones diferentes para conseguir la composición final de esta vista: hay una animación para cada mano, una sale de la parte izquierda de la pantalla y la otra de la parte derecha y confluyen en el centro.

Cuando esta animación termina, aparece el logotipo de MeetMap con el eslogan con otra animación *fade in* (el logotipo con el eslogan está oculto y va apareciendo).

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true">
    <alpha
        android:duration="2000"
        android:fromAlpha="0.0"
        android:interpolator="@android:anim/accelerate_interpolator"
        android:toAlpha="1.0"/>
</set>
```

Código animación *fade in* Splash Activity

El resultado final se puede observar en las siguientes imágenes:



Secuencia vista Splash Activity

➤ Initial Activity

La segunda actividad que se encontrará el usuario nada más empiece a utilizar la aplicación es la *initial*. Aunque, si el usuario ya ha iniciado sesión con alguna cuenta anteriormente, esta actividad no volverá a aparecer a no ser que el usuario cambie de dispositivo, quiera cambiar de cuenta o cierre la sesión actual.



Vista Initial Activity

En esta actividad, se encuentra la llamada a la función `forceLightMode()`, una función global que nos permite evitar el modo oscuro en toda actividad que se quiera. El modo oscuro no ha sido implementado puesto que desde el equipo se pensó que no estaría relacionado con el diseño ideado ni con el plan de aplicación ante la prácticamente inexistencia de planes nocturnos:

*Función modo claro*

La actividad es bastante simple, un diseño simple con estructura redondeada siguiendo la paleta de colores de la aplicación, con un ligero carrusel o “slider” sobre el cual se profundizará más adelante y dos botones, el de iniciar sesión y el botón de crear cuenta. Cada uno de ellos lleva a su propia vista respectivamente.

El slider es un pequeño “viewPager” añadido en la misma vista. Este viewPager está formado por un total de 4 strings que se deslizarán automáticamente (o a elección del usuario) e irá acompañado de unos pequeños puntos en la parte inferior que indican el número de ventana.

*Mensajes slider*

En esta actividad se encuentran dos funciones adicionales a nivel de backend:

- **OnBackPressed()**: modificado para que, si el usuario quiere salir, al darle al botón de atrás no se regrese al splash y en su lugar salga una toast diciendo que si vuelve a hacer *click*, saldrá de la app (esta función se encuentra también en la MainAppActivity).

```

private var doubleBackToExitPressedOnce = false
override fun onBackPressed() {
    if (doubleBackToExitPressedOnce) {
        finishAffinity()
        return
    }

    doubleBackToExitPressedOnce = true
    Toast.makeText(this, resources.getString(R.string.backAgain), Toast.LENGTH_SHORT).show()

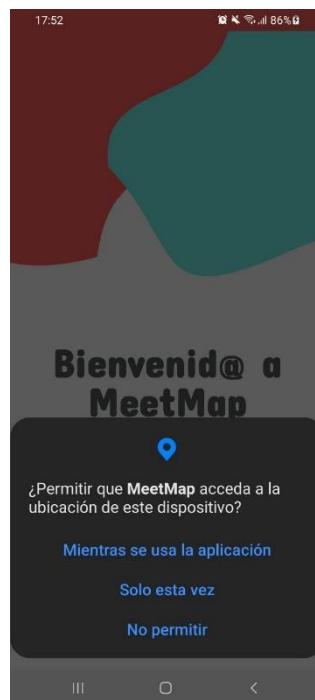
    Handler(Looper.getMainLooper()).postDelayed(Runnable { doubleBackToExitPressedOnce = false }, 2000)
}

```

Función salir de la aplicación

- **RequestLocationPermission()**: una función clave para el desarrollo correcto de la aplicación. Esta función lanza, nada más comienza la actividad, una ventana al usuario en la cual se le pregunta sobre el uso de su ubicación. El equipo decidió situar esta función en esta parte del código para que una vez que el usuario entre al mapa la experiencia sea lo más agradable posible, porque si se ponía en el mapa (aunque hay una reconfirmación por si acaso), la primera vez que el usuario entraba, su ubicación actual no estaría disponible hasta reiniciar la app.

```
private fun requestLocationPermission() {
    if (ActivityCompat.shouldShowRequestPermissionRationale(
        this,
        android.Manifest.permission.ACCESS_FINE_LOCATION
    ) {
        Toast.makeText(
            this,
            resources.getString(R.string.location),
            Toast.LENGTH_SHORT
        ).show()
    } else {
        ActivityCompat.requestPermissions(
            this,
            arrayOf(android.Manifest.permission.ACCESS_FINE_LOCATION),
            MapFragment.REQUEST_CODE_LOCATION
        )
    }
}
```

Función permisos ubicación*Solicitud usuario permisos ubicación*

Cabe destacar que la aplicación está disponible tanto para el idioma español como para el inglés y se actualizará según el idioma del dispositivo. Si el dispositivo móvil está en otra lengua diferente por defecto, la app se presentará en inglés.



Aplicación en inglés

Internacionalización de la aplicación (Inglés/Español)

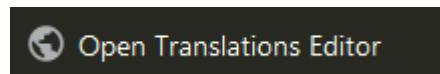
Para lograr que la aplicación esté disponible en diferentes idiomas y se actualice según la lengua del dispositivo móvil se han seguido los siguientes pasos:

- Dentro del proyecto de *Android Studio*: carpeta *res* → *values* → *strings*



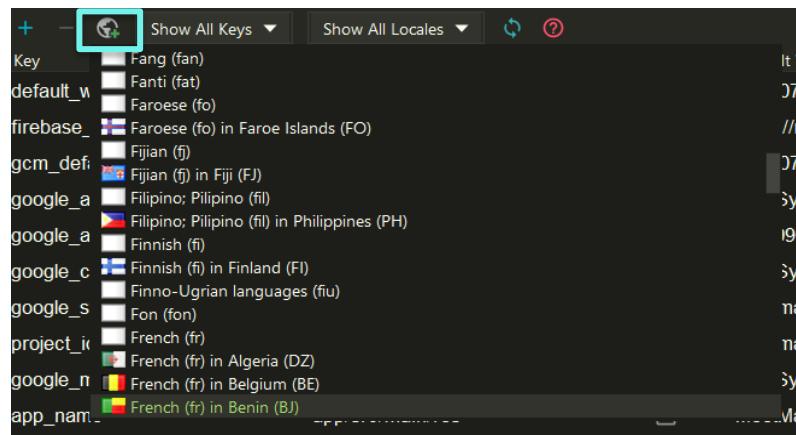
Archivo strings proyecto

- Botón derecho con el ratón y seleccionar



Opción de AndroidStudio para internacionalizar la aplicación

- En este momento se seleccionan los idiomas en los que la aplicación estará disponible si se cambia la lengua del dispositivo. Para ello hay que pulsar en el icono del mundo con el “+”.



Selectores de idiomas para la aplicación

- En el caso de la aplicación MeetMap se ha optado por seleccionar el inglés como idioma por defecto y tener disponible la aplicación en español. Esto quiere decir, que si el móvil tiene como lengua otro idioma que no sea inglés o español, la aplicación se verá en inglés.

Key	Resource Folder	Untranslatable	Default Value	Spanish (es)
updatedData	app/src/main/res	<input type="checkbox"/>	Updated data	Datos actualizados
emailRequired	app/src/main/res	<input type="checkbox"/>	Email is required	Email obligatorio
passRequired	app/src/main/res	<input type="checkbox"/>	Password is required	Contraseña obligatoria
repassRequired	app/src/main/res	<input type="checkbox"/>	Repeat password is required	Repetir contraseña obligatoria
emailValid	app/src/main/res	<input type="checkbox"/>	Email is not valid	El email no es válido
passNeeds	app/src/main/res	<input type="checkbox"/>	Password needs: Capital letters, numbers and symbols	La contraseña necesita: mayúsculas, números y símbolos
passSame	app/src/main/res	<input type="checkbox"/>	Passwords must be the same	Las contraseñas tienen que ser iguales
verifyEmail	app/src/main/res	<input type="checkbox"/>	Please verify your email	Por favor, verifica tu email
emailExists	app/src/main/res	<input type="checkbox"/>	This email already exists	Este email ya existe
accept	app/src/main/res	<input type="checkbox"/>	Accept	Aceptar
authenticationError	app/src/main/res	<input type="checkbox"/>	User authentication failed	Se ha producido un error de autenticación
incorrectEmailPass	app/src/main/res	<input type="checkbox"/>	Incorrect email or password	Email o contraseña incorrectos
accountNotExists	app/src/main/res	<input type="checkbox"/>	The account does not exist. Please check your email	La cuenta no existe. Por favor, verifica tu email
checkMail	app/src/main/res	<input type="checkbox"/>	Check your email to reset your password	Compruebe su email para cambiar la contraseña
tryAgain	app/src/main/res	<input type="checkbox"/>	Try again, something went wrong	Inténtelo otra vez, algo no ha salido bien
date	app/src/main/res	<input type="checkbox"/>	Date:	Fecha:
schedule	app/src/main/res	<input type="checkbox"/>	Schedule:	Horario:
place	app/src/main/res	<input type="checkbox"/>	Place:	Lugar:
link	app/src/main/res	<input type="checkbox"/>	+Info	+Info
formSubmit	app/src/main/res	<input type="checkbox"/>	Form submitted successfully	El formulario se ha enviado correctamente
noHayInfo	app/src/main/res	<input type="checkbox"/>	The company of the activity does not have any information	La empresa de la actividad no tiene información

Traducción de strings en los idiomas seleccionados

Conexión a internet

Para poder utilizar esta aplicación uno de los requisitos es tener conexión a internet es por ello, que si el usuario en la actividad inicial selecciona una de las dos opciones para iniciarse en la aplicación (registrarse o iniciar sesión) y la conexión a internet falla, la aplicación lanza una alerta advirtiendo que no hay conexión a internet y da la opción de reintentar establecer la conexión; si el usuario lo reintenta dos veces, el mensaje de la alerta y el botón cambian de texto y la opción que se da es salir de la aplicación para resolver los problemas de conexión y poder entrar a la app de forma satisfactoria.

La aplicación está diseñada de tal forma que cuando detecta la conexión a internet, independientemente de si se ha pulsado en reintentar o no, la alerta desaparece y carga la vista que estaba pendiente de ejecución cuando hubiese conexión a internet.

La comprobación a internet se hace en todas las actividades de la aplicación que requieren grabar, actualizar, recuperar y eliminar información:

- SignUp Activity
- Login Activity
- Main Activity
- Chat Activity

Para hacer estas comprobaciones es necesario que en el archivo AndroidManifest.xml del proyecto se añada la siguiente línea de código:

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

Control internet AndroidManifest.xml

Se ha creado una función principal que comprueba la conexión a internet a partir de la cual se lleva el control del estado de la alerta: mostrada con el primer texto, mostrada con el segundo texto u oculta.



Estados alerta de la conexión a internet

```

private fun showNoInternetAlert() {
    if (noInternetDialog?.isShowing == true) {
        noInternetDialog?.dismiss()
    }

    val builder = AlertDialog.Builder(context, this)
    // Modificamos el texto del botón de "Reintentar"
    val buttonText = if (retryCount >= 2) {
        builder.setTitle("We want to see you soon!")
        builder.setMessage("MeetMap wants to count on you, when you have internet w...")  

        "Exit"
    } else {
        builder.setTitle("Oops no internet connection!")
        builder.setMessage("MeetMap is looking forward to having you. Do you want t...")  

        "Retry"
    }
    builder.setPositiveButton(buttonText) { dialog, which ->
        if (!isConnectedToInternet()) {
            // Incrementamos el contador de reintentos
            retryCount++
            showNoInternetAlert()
        } else {
            // Reiniciamos el contador de reintentos
            retryCount = 0
            hideNoInternetAlert()
        }
        // Si se ha presionado el botón dos o más veces, cerramos la app
        if (retryCount > 2) {
            hideNoInternetAlert()
            finishAffinity()
        }
    }

    builder.setCancelable(false)
    noInternetDialog = builder.create()
    noInternetDialog?.show()
}

private val networkReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        if (!isConnectedToInternet()) {
            showNoInternetAlert()
        } else {
            retryCount = 0
            hideNoInternetAlert()
        }
    }
}

private fun hideNoInternetAlert() {
    noInternetDialog?.dismiss()
    noInternetDialog = null
}

private fun isConnectedToInternet(): Boolean {
    val connectivityManager =
        getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    val networkInfo = connectivityManager.activeNetworkInfo
    return networkInfo != null && networkInfo.isConnected
}

```

Funciones control conexión a internet

Esta función para controlar el acceso a internet se llama en las funciones iniciales de cada actividad:

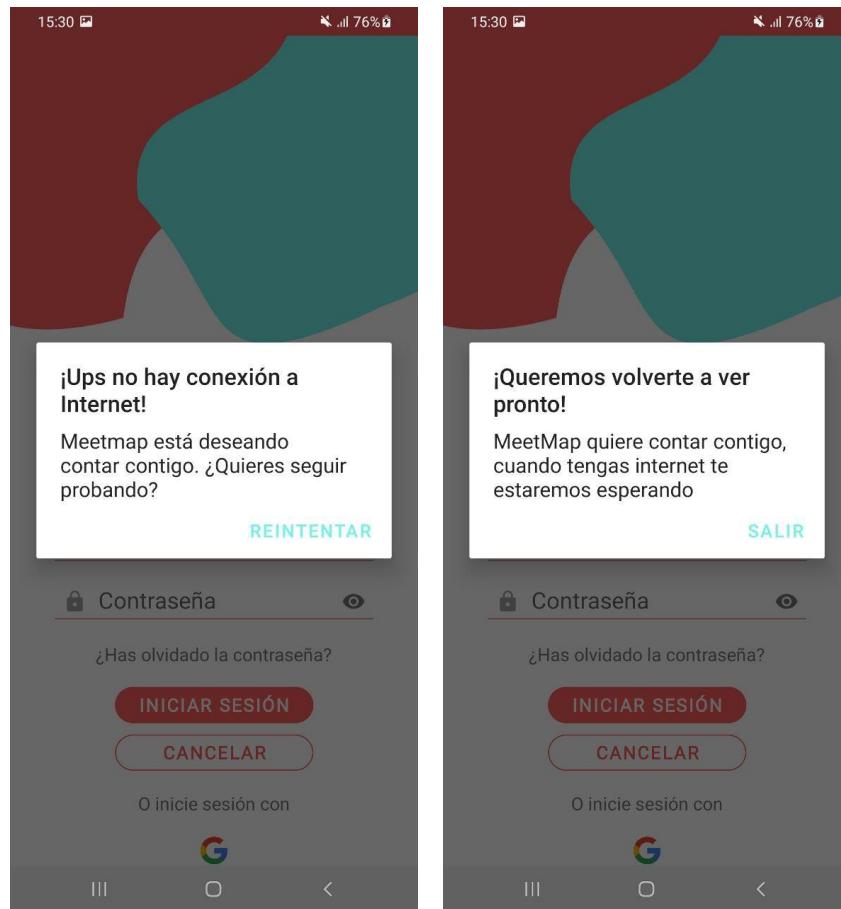
```

fun checkUserValues()
{
    if (isConnectedToInternet()) {
        if (PreferencesManager.getDefaultSharedPreferences(context, this).getEmail().isNotEmpty())
        {
            retryCount = 0
            showMapActivity()
        }
    } else {
        showNoInternetAlert()
    }
}

```

Función principal Login Activity

Los diferentes estados en los que puede estar la aplicación respecto a la conexión a internet son los siguientes:



Login Activity sin internet: dos y tres reintentos

➤ SignUp Activity

Si desde la vista de Initial se ha pulsado el botón “Crear cuenta” se abre la vista en la cual el usuario tiene que registrarse. La persona que se va a registrar tiene dos opciones para ello:

- Introducir correo electrónico y contraseña
- Registrarse a través de Google

Si se opta por la primera opción, es necesario que el usuario introduzca un correo electrónico que obligatoriamente deberá contar entre sus caracteres con “@” y una contraseña que debe ser repetida. Esta contraseña debe cumplir con una serie de requisitos que son los siguientes:

- Mayúsculas
- Minúsculas
- Números

- Más de seis caracteres

La aplicación validará que el correo y la contraseña cumplen con los requisitos que se han comentado. También comprobará que las dos contraseñas que se han escrito coinciden.

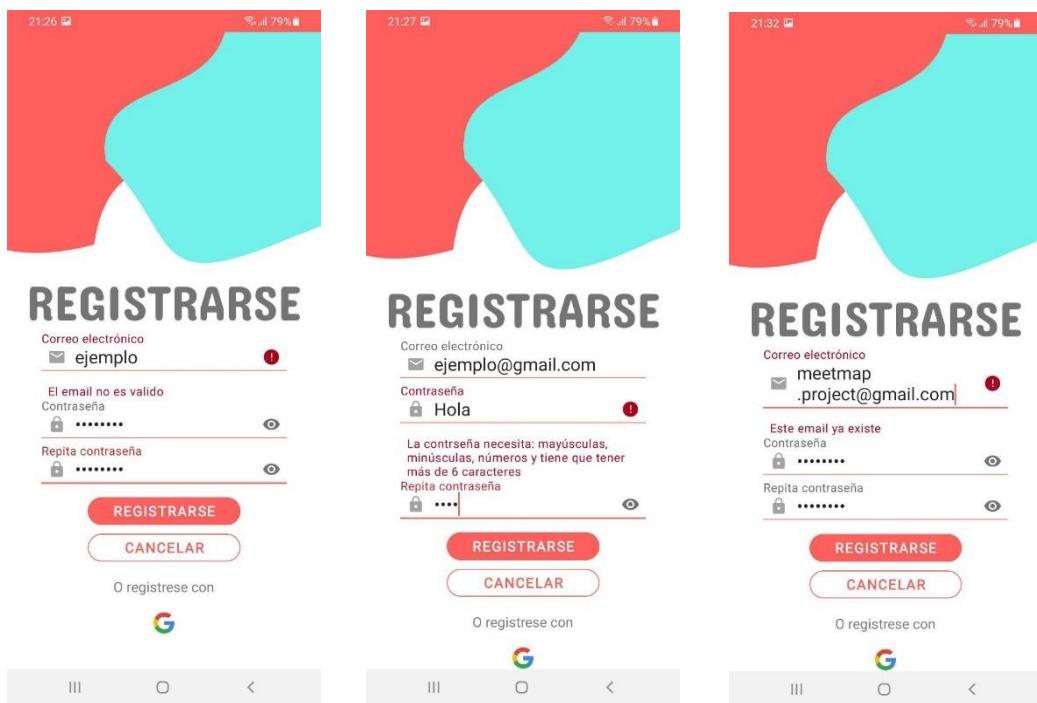
```
fun signUp(){
    val expRegular = Regex(pattern: "^(?=.*[A-Z])(?=.*[0-9])(?=.*[a-z]).{6,15}\$")

    if (email.text.isEmpty() && password.text.isEmpty() && repassword.text.isEmpty()){
        showError(emailTIL, "Email is required")
        showError(passwordTIL, "Password is required")
        showError(repasswordTIL, "Repeat password is required")
    }else if (!email.text.contains(other: "@")){
        showError(emailTIL, "Email is not valid") //This field can't be empty
    }else if(!expRegular.matches(password.text.toString())){
        showError(passwordTIL, "Password needs: Capital letters, small letters, numbers...")
    }else if (!password.text.toString().equals(repassword.text.toString())){
        showError(repasswordTIL, "Passwords must be the same")
    }else{
        fAuth.createUserWithEmailAndPassword(email.text.toString(), password.text.toString()).addOnCompleteListener{
            if (it.isSuccessful){
                fAuth.currentUser?.sendEmailVerification()?.addOnSuccessListener { it: Void? ->
                    Toast.makeText(context: this, "Please verify your email", Toast.LENGTH_LONG).show()
                }
                PreferencesManager.getDefaultSharedPreferences(context: this).saveEmail(email.text.toString())
                val storageRef = Firebase.storage.reference.child(pathString: "img/predeterminado.png")
                storageRef.downloadUrl.addOnSuccessListener { uri ->
                    url = uri.toString()
                    imgNormal()
                }

            }else{
                showError(emailTIL, "This email already exists")
            }
        }
    }
}
```

Función registro del usuario

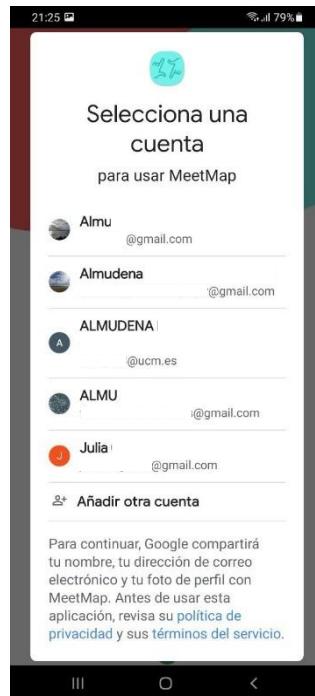
Otra de las comprobaciones que hace el registro de la aplicación es verificar si el correo electrónico que se introduce ya se ha registrado anteriormente. Si es así, se le advierte al usuario para que cambie de correo o inicie sesión.



Advertencias y errores en el registro de usuario

Si el usuario ya ha elegido una cuenta de *Gmail* que esté asociada en su dispositivo, se llevarán a cabo las peticiones que solicita *Google* y una vez satisfechas, se guardarán los datos en *Firebase* y se accederá a la vista principal de la app.

En esta opción, si el usuario intenta registrarse otra vez con el mismo correo de *Google* con el que se ha registrado anteriormente, no se crea un nuevo registro, pero se inicia la aplicación de forma correcta.



Selección cuenta de Gmail para registro MeetMap

```

fun signUpGoogle(){
    val googleConf = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken("serverClientId: "570907010994-lq2g37kb3kop7inhocsoft9gpgcd0ofu.apps.googleusercontent.com")
        .requestEmail()
        .build()
    val googleClient = GoogleSignIn.getClient(this, googleConf)
    googleClient.signOut()
    startActivityForResult(googleClient.signInIntent, GOOGLE_SIGN_IN)
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == GOOGLE_SIGN_IN){
        val task = GoogleSignIn.getSignedInAccountFromIntent(data)
        try {
            val account = task.getResult(ApiException::class.java)
            if (account != null) {
                val credential = GoogleAuthProvider.getCredential(account.idToken, accessToken: null)
                FirebaseAuth.getInstance().signInWithCredential(credential).addOnCompleteListener { it: Task<AuthResult>? {
                    if (it.isSuccessful) {
                        fAuth.currentUser?.sendEmailVerification()?.addOnSuccessListener { it: Void? {
                            Toast.makeText(context, "Please verify your email", Toast.LENGTH_LONG).show()
                        }
                    }
                    fAuth.currentUser?.email?.let { it1 -> PreferencesManager.getDefaultSharedPreferences(context).saveEmail(it1) }
                    val storageRef = Firebase.storage.reference.child(pathString: "img/predeterminado.png")
                    storageRef.downloadUrl.addOnSuccessListener { uri ->
                        url = uri.toString()
                        imgGoogle()
                    }
                } }else{
                    showError(emailTIL, "This email already exists")
                }
            }
        } catch (e: ApiException){
            Log.w(tag: "ERROR", msg: " " + e)
        }
    }
}

```

Registro MeetMap con cuenta de Google

Una vez que la aplicación muestra el mapa con las actividades (vista principal) la sesión se guarda en las preferencias compartidas⁹ para que el usuario, mientras mantenga la sesión iniciada, no tenga que introducir sus credenciales de nuevo y al abrir la aplicación aparezca directamente la vista principal.

⁹ “Las preferencias compartidas Android, o Android Shared Preferences se utilizan como un sistema para almacenar datos en la memoria interna de la aplicación. Estos datos se guardan en la forma clave-valor, es decir como un pequeño diccionario donde la clave siempre es un String y el dato es un tipo simple. Así, vamos a poder almacenar datos que usemos de forma recurrente en varias actividades de nuestra app, sin tener que recurrir constantemente al paso de datos entre actividades, como veíamos en el tema anterior.

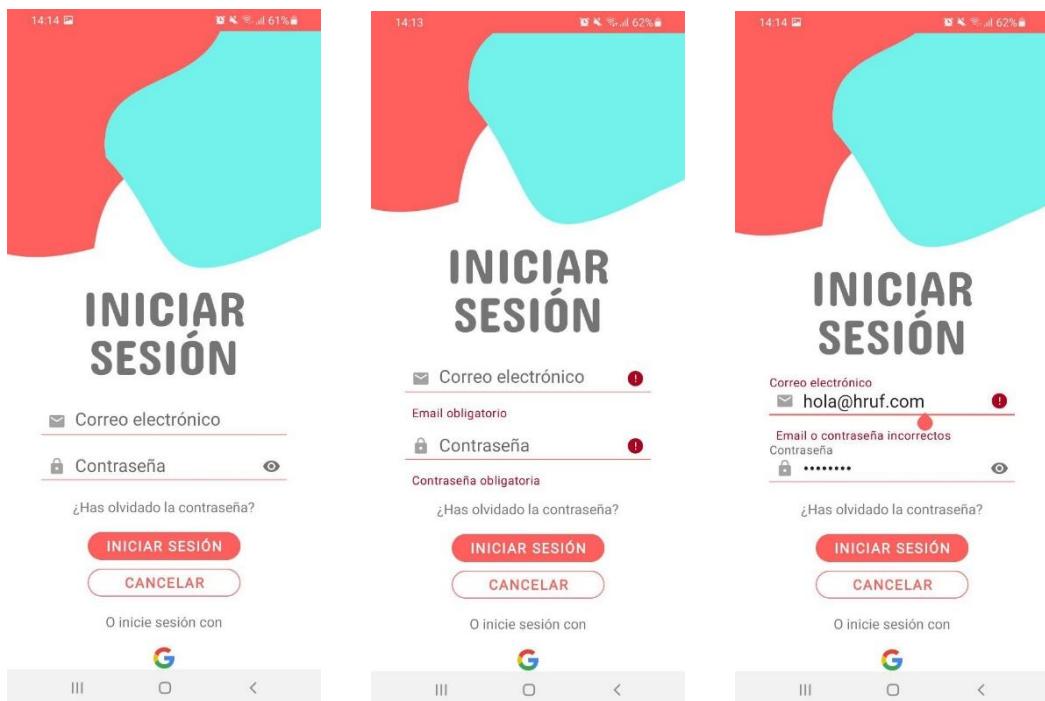
Es también una forma de almacenar datos recurrentes para que el usuario no tenga que introducirlos constantemente, como por ejemplo las credenciales de acceso como nombre de usuario y contraseña. Es un sistema rápido, ligero y que nos va a evitar montar bases de datos complejas para estas tareas tan sencillas.

Por tanto, podemos resumir que las shared preferences son una forma de aligerar el proceso de almacenado de datos simples (aunque se pueden guardar tipos complejos con GSON). ”

Walkiria Apps. (s.f.). Shared Preferences Android Studio. Web.Curso Android Gratis. <https://cursoandroidgratis.com.es/shared-preferences/#:~:text=Las%20preferencias%20compartidas%20Android%2C%20dato%20es%20un%20tipo%20simple.>

➤ Log In Activity

Si desde la vista inicial se ha pulsado el botón de iniciar sesión, se abre la vista de Log In donde el usuario que se ha registrado anteriormente puede iniciar sesión en la app introduciendo el correo y la contraseña que usó para registrarse o eligiendo la opción de *Gmail* si se registró de esta forma.



Inicio sesión usuario y contraseña MeetMap

```
fun login(){
    if (email.text.isEmpty() && password.text.isEmpty()){
        showError(emailTIL, "Email is required")
        showError(passwordTIL, "Password is required")
    }else if(!email.text.contains("@")){
        showError(emailTIL, resources.getString(R.string.emailValid))
    }else{
        fAuth.signInWithEmailAndPassword(email.text.toString(), password.text.toString()).addOnCompleteListener{ it: Task<AuthResult>{
            if (it.isSuccessful){
                PreferencesManager.getDefaultSharedPreferences(context: this).saveEmail(email.text.toString())
                PreferencesManager.getDefaultSharedPreferences(context: this).savePass(password.text.toString())
                showMapActivity()
            }else{
                //showAlert()
                showError(emailTIL, "Incorrect email or password")
            }
        }}
    }
}
```

Función inicio sesión usuario y contraseña MeetMap

```
fun loginGoogle(){
    val googleConf = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken("570907010994-lq2g37kb3kop7inhocsuft9gpgcd0ofu.apps.googleusercontent.com")
        .requestEmail()
        .build()
    val googleClient = GoogleSignIn.getClient(activity, googleConf)
    googleClient.signOut()
    startActivityForResult(googleClient.signInIntent, GOOGLE_SIGN_IN)
}
```

Función inicio sesión Google MeetMap

➤ Reset Password Activity

Puede ser que el usuario no recuerde su contraseña por lo que tiene la opción de resetear la contraseña: se le manda un correo a su cuenta con la que se registró para modificar la contraseña (servicio que proporciona *Firebase*) y de esta forma obtiene su nueva contraseña con la que puede acceder a la aplicación.



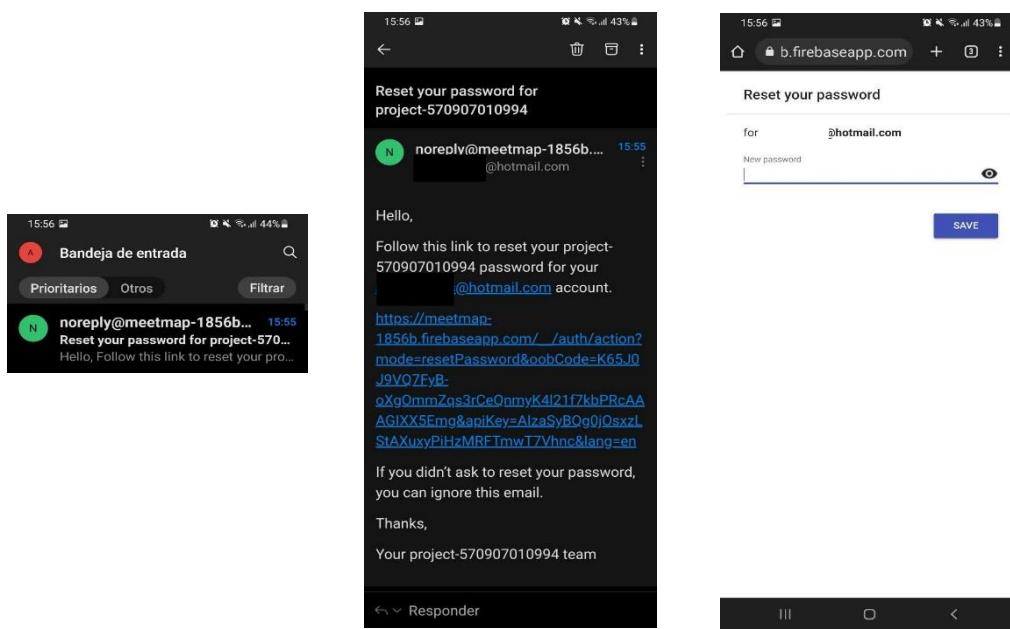
Vista para resetear contraseña

```

fun resetPassword(){
    if (email.text.isEmpty()){
        showError(emailTIL, "Email is required")
    }else if(!email.text.contains("@")){
        showError(emailTIL, "Email is not valid")
    }else{
        fAuth.sendPasswordResetEmail(email.text.toString()).addOnCompleteListener(
            OnCompleteListener<Void> {
                if (it.isSuccessful){
                    Toast.makeText(context, "Check your email to reset your password", Toast.LENGTH_LONG).show()
                }else{
                    Toast.makeText(context, "Try again, something wrong happened", Toast.LENGTH_LONG).show()
                }
            }
        )
    }
}

```

Función para resetear contraseña



Proceso para cambiar contraseña

Se cree conveniente señalar que tanto la vista de registro como la de inicio de sesión son vistas que permiten el desplazamiento (*ScrollActivities*) debido a que los correos pueden tener una longitud variable y, si salta algún error a la hora de registrarse o iniciar sesión mueven los componentes iniciales haciendo la vista con más altura. Si no se implementa la vista con desplazamiento, algunas funcionalidades quedarían ocultas.

Tanto en el *Sign Up* como en el *Log In* se ha implementado la funcionalidad de ocultar la contraseña y se lleva a cabo mediante el icono del ojo abierto (se muestra la contraseña) y el ojo cerrado (contraseña oculta).

```

<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/etpassword"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="@+id/rightguide"
    app:layout_constraintStart_toStartOf="@+id/leftguide"
    app:layout_constraintTop_toBottomOf="@+id/etemail"
    app:passwordToggleEnabled="true"
    app:startIconDrawable="@drawable/ic_lock"
    app:hintTextColor="@color/secondary_dark">

    <EditText
        android:id="@+id/password"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:hint="Password"
        android:inputType="textPassword"
        android:theme="@style/lineET" />
</com.google.android.material.textfield.TextInputLayout>

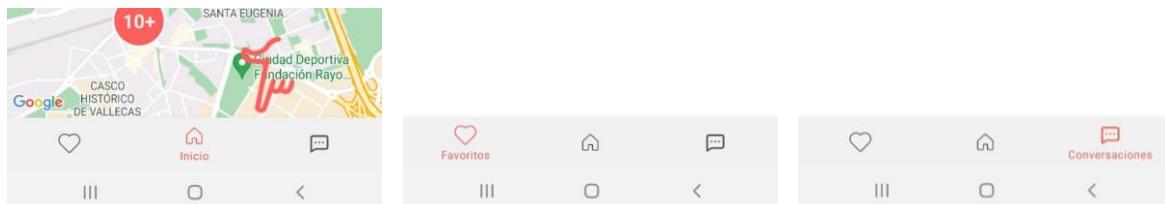
```

Diseño del campo de contraseña

➤ Main Activity

Esta vista es la encargada de gestionar los dos menús principales: el menú que permite navegar entre las tres grandes secciones de la aplicación (mapa, favoritos y chat) y el menú lateral que posibilita el acceso del usuario a su perfil, información de la aplicación y las opciones de cerrar sesión y eliminar cuenta.

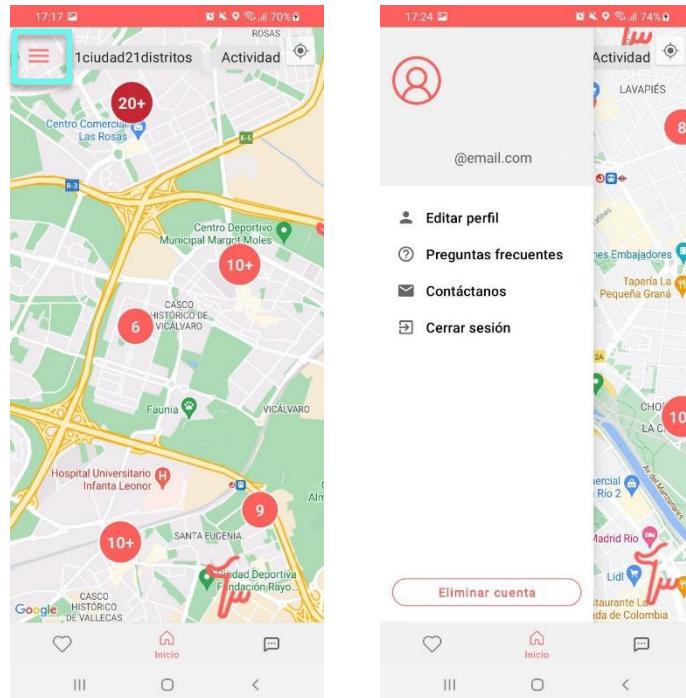
El menú de las tres secciones se muestra de forma permanente mientras se navega y sirve para tener la referencia de la sección en la que se encuentra el usuario ya que este ítem cambia de color.



Navegación secciones principales

Por otro lado, el menú lateral no se muestra por defecto, en la actividad principal se presenta un botón en forma de hamburguesa que al ser pulsado permite desplegarse desde la izquierda hasta el centro de la pantalla del dispositivo móvil y cuando se pulsa sobre la zona que queda sin cubrir por el menú lateral, este se oculta.

La lógica de ambos menús se mantiene constante mientras el usuario está navegando por los fragmentos de la actividad principal.



Botón hamburguesa y despliegue menú lateral

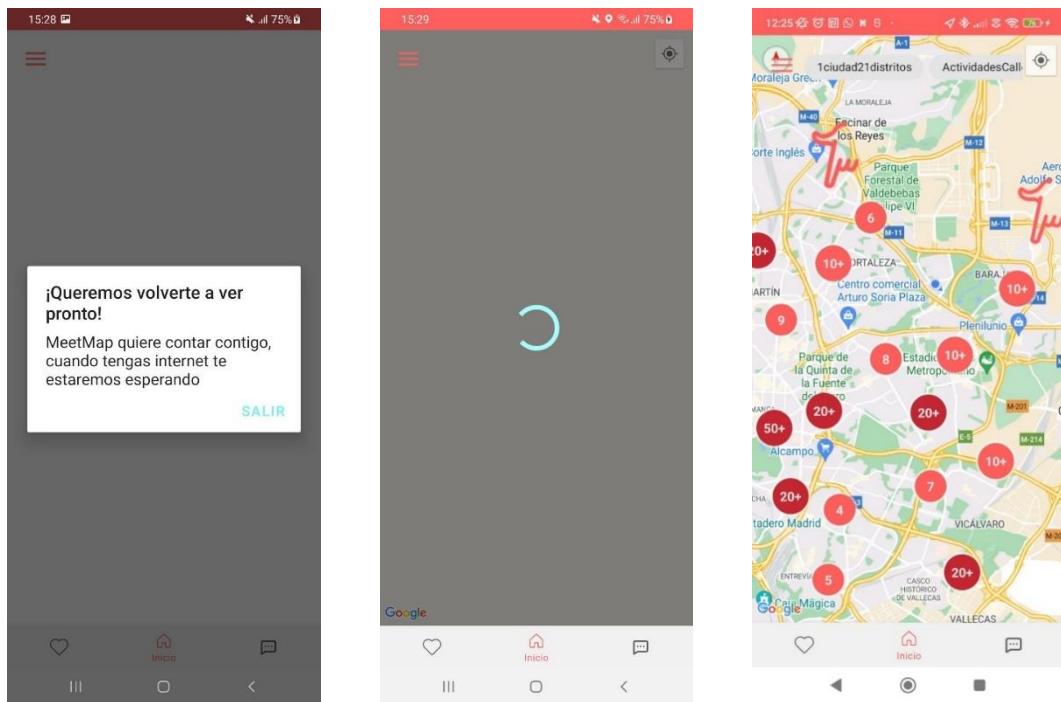
Cuando el usuario pulsa sobre una de las opciones que ofrece la aplicación en el menú lateral, ocurren dos acciones:

- Se produce una navegación al fragmento seleccionado
- Se cierra el menú lateral y se vuelve a mostrar el ícono del menú, que mientras este está abierto no se muestra.

```
navview.setNavigationItemSelectedListener { menuItem ->
    when (menuItem.itemId) {
        R.id.nav_profile -> {
            closeNav()
            setThatFragment(profileFragment)
        }
        R.id.nav_manusu -> {
            closeNav()
            setThatFragment(faqsFragment)
        }
        R.id.contactus -> {
            closeNav()
            setThatFragment(contactUsFragment)
        }
        R.id.nav_exit -> {
            PreferencesManager.getDefaultSharedPreferences(binding.root.context).wipe()
            fAuth.signOut()
            startActivity(Intent(packageContext: this, Initial::class.java))
            Intent(packageContext: this, Initial::class.java)
        }
    }
    // Indica que el elemento ha sido seleccionado
    binding.navView.setCheckedItem(-1)
    menuItem.isChecked = true
    true | ^setNavigationItemSelectedListener
}
```

Lógica de navegación entre secciones del menú lateral

Cuando se inicia la actividad principal, se muestra por defecto el fragmento del mapa. Si las actividades no se han podido cargar en el mapa por problemas con la conexión a internet no se mostrará el mapa y en su lugar aparecerá una alerta como en el caso del *Sign Up* o el *Log In* indicando que no hay conexión a internet. Una vez que se vuelva a establecer la conexión a internet el dialogo de alerta desaparecerá y comenzará la animación de carga para, posteriormente, mostrar el mapa con los respectivos marcadores de actividad.



Proceso carga Map Fragment en la actividad principal

❖ Map Fragment

En este fragmento, se encuentra la parte principal de la aplicación. Un mapa interactivo donde el usuario podrá seleccionar multitud de actividades, filtrarlas o ver cuales están cerca de él. Todo el fragmento tiene una gran carga de código, que se ira desglosando poco a poco.

Lo primero que se debe comentar es la necesidad de establecer un mapa gracias a la “Google Maps API”, la cual se obtiene de forma gratuita y te permite establecer un mapa como si de Google Maps se tratase. Este mapa, ofrece multitud de funcionalidades y métodos, que han permitido al equipo desarrollar la aplicación a su antojo.

La parte encargada de ofrecer toda la información y *parsear*¹⁰ el JSON que Madrid ofrece gratuitamente, se encuentra en esta parte del código:

¹⁰ “En programación, parsear es el proceso de analizar una cadena de texto para identificar su estructura sintáctica y extraer información significativa de ella. Por ejemplo, si tienes una cadena de texto que contiene una lista de números separados por comas, para poder trabajar con esos números en tu programa, debes parsear la cadena para extraer los números individuales. Esto implica separar la cadena en elementos separados por comas, eliminar cualquier espacio en blanco o caracteres no numéricos, y luego convertir cada elemento a un valor numérico que pueda ser utilizado por el programa.” Terrón Barroso, A. (2012, 17 de octubre). El parseo de datos en internet: nuevas oportunidades comerciales al filo de la legalidad. INESEM BUSINESS SCHOOL <https://www.inesem.es/revistadigital/gestion-empresarial/el-parseo-de-datos-en-internet-nuevas-oportunidades-comerciales-al-filo-de-la-legalidad/#:-:text=%C2%BFQu%C3%A9%20es%20el%20parseo%20o.extraer%20informaci%C3%B3n%23n%20significativa%20de%20ella.>

```

private val viewModel: MadridViewModel by lazy {
    ViewModelProvider(this)[MadridViewModel::class.java]
}

@SuppressLint("PotentialBehaviorOverride")
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap
    enableLocation()
    viewModel.fetchData()
    map.setOnMarkerClickListener { false }
    map.setOnMyLocationClickListener(this)
}

```

Obtención de la información de la Api de Madrid

Gracias a este código, se crea un *viewModel* que, una vez el mapa se ha creado, ejecuta la función *fetchData*, la cual se encarga de *parsear* manualmente todo el JSON, creando *locators*¹¹ (posteriormente se verá donde se utilizan).

El *parseo* manual resultó ciertamente sencillo gracias a la manera en la que Madrid ofrece el JSON: cada *locator* viene dentro de un árbol grande denominado *graph*, por tanto, se creó una *MadridResponse* que contenía a *graph*, y este *graph* a su vez, contiene cada elemento que puede tener cualquier actividad: hora, coordenadas, título, etc. La parte esencial de este código es la siguiente, ya que es la que recorre todo *graph* y va almacenando todos los *locators*:

```

fun fetchData(){
    val manager = MadridApiRequestManager()
    uiScope.launch {
        val newList = mutableListOf<LocatorView>()
        val madridResponse = withContext(Dispatchers.IO){
            manager.getAll()
        }

        madridResponse.graph.forEach {
            newList.add(LocatorView().fromGraph(it))
        }

        mutableLocators.value = newList
    }
}

```

Función carga de datos de la Api de Madrid

¹¹ Un *locator* es un objeto creado que recoge toda la información de cada plan de la API de Madrid y la almacena para su posterior uso y se puede modificar para añadir más atributos: título, descripción, coordenadas, etc.

Una vez se han creado todos los *locators*, se llamará a la función *Observe()*, la encargada de crear y mostrar cada marcador personalizado. Además, controla un pequeño *loadingSpinner* que se ejecuta mientras se recorren todos los *locators*.

Esta función, recorrerá todo el *viewModel*, y con él, cada uno de los *locators* que existen, recogiendo sus coordenadas. Gracias a estas coordenadas se creará un *marker* que tendrá de título el identificador del *locator* y el título de este. Ese *marker* se añadirá al mapa, a una lista mutable igualando el título del *marker* con todo el *marker* en su conjunto, y a un *hashMap* que igualará el título del *marker* con el id del mismo. Esta lista mutable y el *hashMap* tendrán su relevancia en las diferentes funciones, e inclusive en otros fragmentos que requieran conexión con los *locators*.

```
viewModel.locators.observe(viewLifecycleOwner) { locators ->
    locators.mapNotNull {
        it.location.latitude?.let { lat ->
            it.location.longitude?.let { lng ->
                LatLng(lat, lng)
            }
        }?.let { coordinates ->
            val markerOptions = MarkerOptions().position(coordinates)
                .title("${it.id} ${it.title}")
            val marker = map.addMarker(markerOptions)
            markers[marker.title] = marker
            madridMap[marker.title] = it.id
            marker.setIcon(BitmapDescriptorFactory.fromResource(R.drawable.mano_rosa))
        }
    }
}
```

Función *Observe()* fragmento *mapa*

Adicionalmente, la función *observe()* creará una *locatorList* con todos los *locator*, evitando así la llamada al *viewModel*(mucho más lenta), establecerá unas ventanas personalizadas a cada *marker*, y creará los chips, encargados de filtrar por categoría todos los *markers*.

A continuación, se explicará cada función que se encuentra en este fragmento, dado que el funcionamiento general e importante de la aplicación se acaba de explicar.

- **ChipCreator()**: esta función creará cada uno de los *chips* que se encuentran en la parte superior del fragmento, en un menú deslizable. La creación se realiza de manera automática recorriendo todos los *locators* y revisando el *locator.category*, si no ha sido añadido anteriormente, se añadirá y creará el *chip*. En este método, se encuentra también la funcionalidad de los *chips*, la cual es muy simple, si el usuario hace *click* en cualquier *chip*, su categoría se añadirá a una lista que posteriormente se recorrerá en la función *ApplyFilter()*. Si vuelve a hacer *click* en ese *chip*, se quitará de la lista.

Gracias a esa lista, se creará una lista de *locators* que contengan al menos una de las categorías y ésta se pasará como argumento en *ApplyFilter*, si la lista está vacía, es decir, ningún *chip* ha sido seleccionado, se pasará toda la lista de *locators*. Evitando así el problema de seleccionar un *chip*, deseleccionarlo y encontrar el mapa vacío.

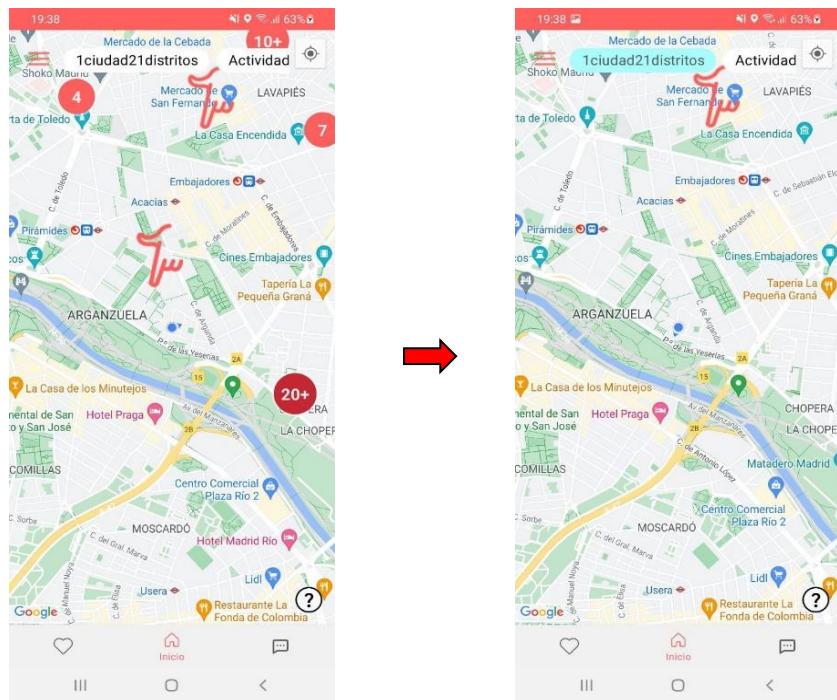
```
if (selectedCategories.isEmpty()) {
    applyFilter(locators)
} else {
    val filteredLocators = locators.filter {
        val cat = it.category.split("/").getOrNull(6)
        selectedCategories.contains(cat)
    }
    applyFilter(filteredLocators)
}
```

Creación de la lista de locators filtrada

- **ApplyFilter()**: esta función es realmente similar al método *observe*, solo que, en vez de recorrer todo el *viewModel*, recorrerá la *locatorsList* que le llegue por argumento. Además de limpiar el mapa y los *markers* previos.

```
map.clear()
markers.clear()
filteredLocators.mapNotNull {
    it.location.latitude?.let { lat ->
        it.location.longitude?.let { lng ->
            LatLng(lat, lng)
        }
    }?.let { coordinates ->
        val markerOptions = MarkerOptions().position(coordinates)
            .title("${it.id} ${it.title}")
        val marker = map.addMarker(markerOptions)
        markers[marker.title] = marker
        madridMap[marker.title] = it.id
        marker.setIcon(BitmapDescriptorFactory.fromResource(R.drawable.mano_rosa))
    }
}
```

Función ApplyFilter() fragmento mapa



Mapa sin filtros y con filtros

- **onInfoWindowClick()**: una vez creada la ventana personalizada de cada *marker* gracias a una función que la propia API ofrece, esta función controlará qué sucede al hacer *click* en dicha ventana. En esencia, es muy sencillo, cambiará de fragmento al detalle de la actividad de la cual se hablará posteriormente. Además, a la hora de crearlo le pasa todo el *marker* por argumento, para poder recoger toda la información necesaria en el detalle de la actividad.

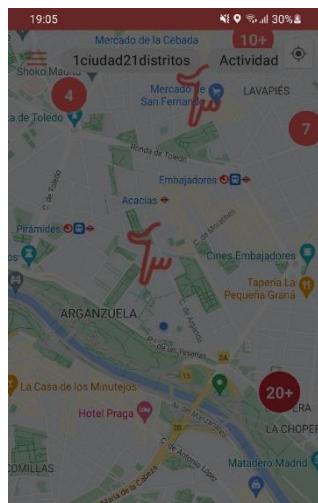
```
override fun onInfoWindowClick(marker: Marker) {
    val infoFragment = InfoActivityFragment()
    infoFragment.setMarker(marker, locatorList)
    requireActivity().supportFragmentManager.beginTransaction()
        .setCustomAnimations(android.R.anim.fade_in, android.R.anim.fade_out)
        .replace(R.id.frame, infoFragment)
        .addToBackStack(null)
        .commit()
}
```

Función onInfoWindowClick() fragmento mapa



Vista tarjeta información al seleccionar marcador

- **onMyLocationClick()**: función que permite al usuario, una vez haga *click* en su propia ubicación, descubrir qué actividades se encuentran a menos de un kilómetro de distancia. Para poder verlas, se despliega un menú desde la parte inferior en el cual se ven todas esas actividades.



Actividades a menos de 1000 metros

Alumnos de los talleres del Centro Cultural Casa del Reloj
2023-06-01 - 2023-06-24
00:00 - 23:59

Actividades a menos de 1000 metros	
Actividades a menos de 1000 metros	
Alumnos de los talleres del Centro Cultural Casa del Reloj	2023-06-01 - 2023-06-24 00:00 - 23:59
Amor, son y boleros	2023-07-02 - 2023-07-02 12:00 - 23:59
AUTORAS. Encuentros en torno a la lectura. Silvia Nanclares	2023-06-30 - 2023-06-30 18:00 - 23:59
Cabaret Magnético	2023-06-09 - 2023-06-09 19:00 - 23:59
Cena para dos	2023-07-01 - 2023-07-01 19:00 - 23:59
Concierto de Primavera	2023-06-25 - 2023-06-25 12:00 - 23:59
De Mozart a La Habana	2023-06-11 - 2023-06-11 12:00 - 23:59
Encuentros de Bhakti Yoga en Madrid	

Vista actividades a menos de 1Km de la ubicación del usuario

Resaltan dos fragmentos de código:

- La función *distance*, que permite hacer ese radio de un kilómetro:

```
private fun distance(lat1: Double, lon1: Double, lat2: Double, lon2: Double): Float {
    val R = 6371
    val dLat = Math.toRadians(lat2 - lat1)
    val dLon = Math.toRadians(lon2 - lon1)
    val a = (sin(dLat / 2) * sin(dLat / 2) +
        cos(Math.toRadians(lat1)) * cos(Math.toRadians(lat2)) *
        sin(dLon / 2) * sin(dLon / 2))
    val c = 2 * atan2(sqrt(a), sqrt(1 - a))
    return (R * c * 1000).toFloat()
}
for (locator in locatorList) {
    val distance = distance(
        location.latitude, location.longitude,
        locator.location.latitude ?: 0.0, locator.location.longitude ?: 0.0
    )
}
```

Función distance() fragmento mapa

- La función que permite cambiar de fragmento en caso de hacer *click* en una actividad. Hay una importante diferencia a la función que había en el *onInfoWindowClick*, puesto que ya no se dispone directamente del marker seleccionado. Sin embargo, gracias a la lista mutable anteriormente creada, *markers*, se puede recuperar el *marker* correspondiente a la actividad sobre la que se hace *click*:

```
val marker = markers["${item.id} ${item.title}"]
if (marker != null) {
    val infoFragment = InfoActivityFragment()
    infoFragment.setMarker(marker, locatorList)
    requireActivity().supportFragmentManager.beginTransaction()
        .setCustomAnimations(android.R.anim.fade_in, android.R.anim.fade_out)
        .replace(R.id.frame, infoFragment)
        .addToBackStack(null)
        .commit()
    bottomSheetDialog.dismiss()
}
```

Cambio de fragmento con información de la actividad seleccionada

Adicionalmente, de cara a *frontend* se ha creado una función *selectionIcon()* que recoge la categoría de cada ítem y devuelve de manera sencilla un ícono personalizado para cada una de ellas. Se puede observar que hay cinco iconos “default” que permiten devolver uno en caso de que la categoría no este implementada en el filtro.

```

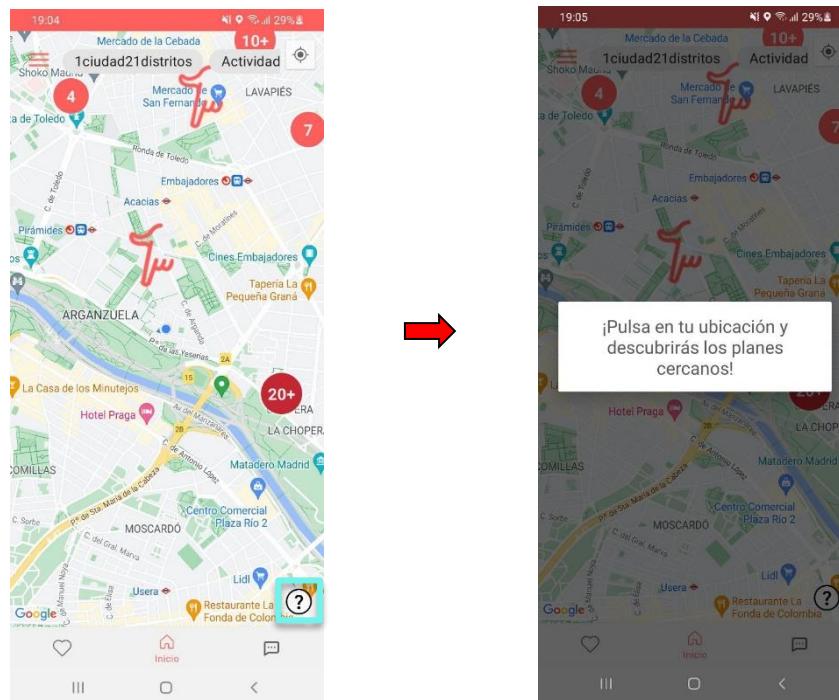
fun selectionIcon(
    category: String,
): Int {
    val options = listOf(
        R.drawable.ico_gen1,
        R.drawable.ico_gen2,
        R.drawable.ico_gen3,
        R.drawable.ico_gen4,
        R.drawable.ico_gen5
    )
    val iconResId = when (category.split("/").getOrNull(6) ?: options.random()) {
        "Musica" -> R.drawable.ico_musica
        "DanzaBaile" -> R.drawable.ico_danzabaile
        "CursosTalleres" -> R.drawable.ico_cursostalleres
        "TeatroPerformance" -> R.drawable.ico_teatro
        "ActividadesCalleArteUrbano" -> R.drawable.ico_arteurbano
        "CuentacuentosTiteresMarionetas" -> R.drawable.ico_cuentacuentos
        "ComemoracionesHomenajes" -> R.drawable.ico_homenaje
        "ConferenciasColoquios" -> R.drawable.ico_conferencias
        "1ciudad21distritos" -> R.drawable.ico_ciudaddistritos
        "ExcusionesItinerariosVisitas" -> R.drawable.ico_visitas
        "ItinerariosOtrasActividadesAmbientales" -> R.drawable.ico_ambientales
        "ClubesLectura" -> R.drawable.ico_lectura
        "RecitalesPresentacionesActosLiterarios" -> R.drawable.ico_recitales
        "Exposiciones" -> R.drawable.ico_exposiciones
        "Campamentos" -> R.drawable.ico_campamentos
        "CineActividadesAudiovisuales" -> R.drawable.ico_cine
    }
}

```

Selección iconos según categoría listado actividades a menos de 1Km

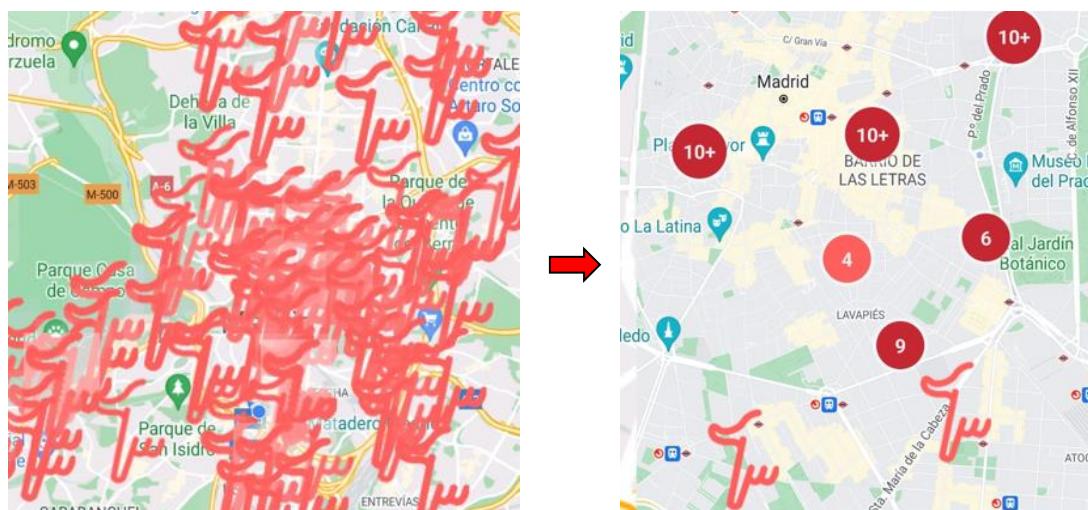
- **onMyLocationButtonClick()**: función obligada por la API, pero sin funcionalidad aparente más allá de regresar al usuario a su ubicación si hace *click* en un botón de la esquina superior derecha.
- **EnableLocation()**: función que permite, una vez comprobado en la función *isLocationPermissionGranted()* que la ubicación esta aceptada, establecer la ubicación actual del usuario.
- **isLocationPermissionGranted()**: función que simplemente permite saber si están aceptados los permisos de ubicación.
- **requestLocationPermission()**: al igual que en la vista inicial de la aplicación, es una función que pide al usuario que acepte los permisos de ubicación, en este caso, por si el usuario se lo saltó anteriormente.

Otra de las funcionalidades que se han implementado en esta aplicación ha sido la creación de un botón que se muestra en el mapa nada más iniciar la app. Dicho botón es una indicación para el usuario y se sitúa en la parte inferior derecha, permite mostrar información de ayuda. De momento, se ha implementado para avisar de la funcionalidad del botón de la ubicación actual, aunque en un futuro se pueden ampliar estas funciones.



Indicación para pulsar sobre la ubicación del usuario

Una vez el desarrollo del mapa estaba concluido, se decidió implementar la tecnología *clustering*, predefinida por la biblioteca *ClusterManager*, que permite al usuario poder visualizar todos los datos sin que visualmente los marcadores se amontonen:

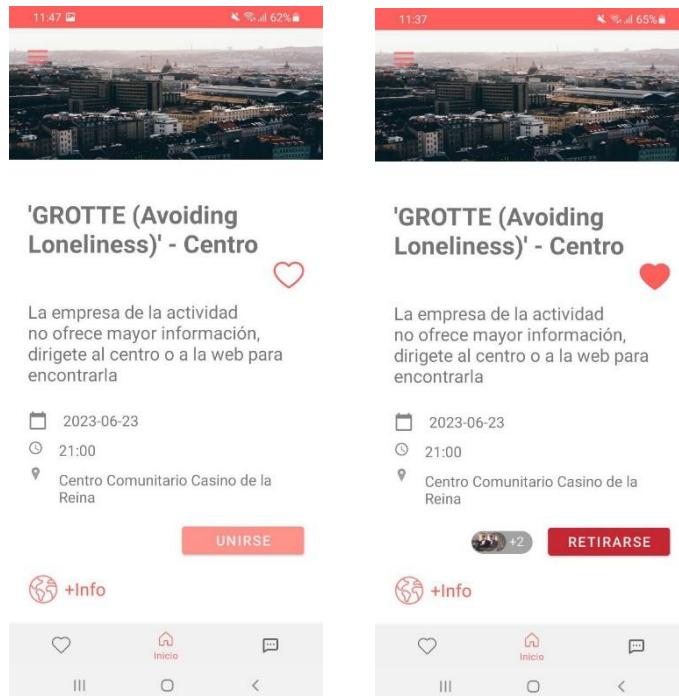


Antes y después de la implementación del clustering en el diseño del mapa

Sin embargo, la implementación de este código ha propiciado diversos fallos que se han tenido que solucionar individualmente, ajustando gran parte de los métodos explicados con anterioridad a este tipo de tecnología y que se van a desarrollar en el siguiente apartado de la memoria donde se tratan los problemas encontrados y las soluciones que se han implementado.

❖ InfoActivity Fragment

Este fragmento mostrará toda la información que cada plan ofrece: título, descripción, fecha, hora, lugar, etc. Adicionalmente, se mostrará una imagen según la categoría (se profundizará más adelante), un botón para acceder a la web de la actividad, un botón para añadir la actividad a favoritos, otro para suscribirse al plan e informar al resto de usuarios y otro donde podrás ver todos los usuarios apuntados a dicha actividad.



Vista detalle actividad en diferentes estados

La primera función que realiza este fragmento es recoger el *marker* del mapa correspondiente y la *locatorList* que se comparte desde el mapa (o desde la lista de favoritos, aunque todo esa información se envía desde el mapa).

```
fun setMarker(marker: Marker, locatorsList: List<LocatorView>) {
    this.marker = marker
    this.locatorsList = locatorsList
}
```

Transferencia información mapa-detalle actividad

Posteriormente, en el `onCreateView()` se llama a la función `fillFields(LocatorView)` pasando directamente el locator correspondiente al *marker*.

- La función `fillFields()` cambiará la vista del usuario según los datos recogidos por el *locator*. Sin embargo, hay que tener en cuenta que, desde la API de Madrid, no todas las actividades disponen de todos los datos, es por ello, que se realizó un control de cada campo para evitar fallos al seleccionar cualquier plan:

```
val desc = locatorView.description.isEmpty {  
    resources.getString(R.string.noHayInfo)  
}  
description.text = desc  
  
val date = if (locatorView.dstart.split(" ")[0]==locatorView.dfinish.split(" ")[0]){  
    locatorView.dstart.split(" ")[0]  
}else if(locatorView.dstart.isEmpty() && locatorView.dfinish.isEmpty()){  
    resources.getString(R.string.dateInfo)  
}  
else{  
    locatorView.dstart.split(" ")[0]+ " - "+ locatorView.dfinish.split(" ")[0]  
}  
fecha.text = date.toString()  
  
val hora = locatorView.time.isEmpty {  
    resources.getString(R.string.hourInfo)  
}  
horario.text = hora  
  
val lug = locatorView.event_location.isEmpty {  
    resources.getString(R.string.placeInfo)  
}
```

Validación campos actividades procedentes de la API de Madrid

En esta función también se crea el objeto `plAct`, realmente útil para guardar cada plan en la base de datos.

También, se establece una imagen personalizada para cada categoría, y como cada día pueden existir nuevas categorías, se han añadido imágenes predeterminadas que la aplicación seleccionará aleatoriamente en caso de existir la categoría. Esta parte es similar a la función `selectionIcon()` que se usa en el *ClusterManager* y en el *MapFragment*, sin embargo, esto devolverá una imagen en vez de un ícono, y por tanto ha sido imposible unirlas para simplificar el código.

```

        val options = listOf(
            resources.getDrawable(R.drawable.amigos, null),
            resources.getDrawable(R.drawable.amigoscielo, null)
        )
        val res = when (locatorView.category.split("/").getOrNull(6) ?: options.random()) {
            "Musica" -> resources.getDrawable(R.drawable.musica, null)
            "DanzaBaile" -> resources.getDrawable(R.drawable.danzabaile, null)
            "CursosTalleres" -> resources.getDrawable(R.drawable.cursotalleres, null)
            "TeatroPerformance" -> resources.getDrawable(R.drawable.teatro, null)
            "ActividadesCalleArteUrbano" -> resources.getDrawable(R.drawable.arteurbano, null)
            "CuentacuentosTiteresMarionetas" -> resources.getDrawable(R.drawable.cuentacuentos, null)
            // "ProgramacionDestacadaAgendaCultura" -> resources.getDrawable(R.drawable.destacada, null)
            "ComemoracionesHomenajes" -> resources.getDrawable(R.drawable.homenajes, null)
            "ConferenciasColoquios" -> resources.getDrawable(R.drawable.conferencias, null)
            "1ciudad2distritos" -> resources.getDrawable(R.drawable.distritos, null)
            "ExcusionesItinerariosVisitas" -> resources.getDrawable(R.drawable.excusiones, null)
            "ItinerariosOtrasActividadesAmbientales" -> resources.getDrawable(R.drawable.ambientales, null)
            "ClubesLectura" -> resources.getDrawable(R.drawable.lectura, null)
        }
    
```

Selección imagen personalizada por categoría

Para agregar la actividad a favoritos, el usuario dispone de un botón el cual tiene la forma de corazón y se modifica entre su trazo y su relleno, dependiendo de si esta actividad se encuentra guardada en la base de datos a tiempo real de *Firebase* o no. Para añadir la actividad, utilizamos la función *addToFavourite()* pasando como parámetro el objeto *plAct* mencionado anteriormente el cual almacena toda la información pertinente a la actividad y, junto con un *HashMap*, permitirá añadir a la base de datos toda esta información. Seguidamente, se realizará la conexión a la base de datos para poder guardar dicho *HashMap*.

```

private fun addToFavourite(plAct: plAct){
    IsInMyFavourite = true
    checkIsFavourite(plAct)
    val hashMap = HashMap<String, Any>()

    hashMap ["ID"] = plAct.id
    hashMap ["Title"] = plAct.titulo

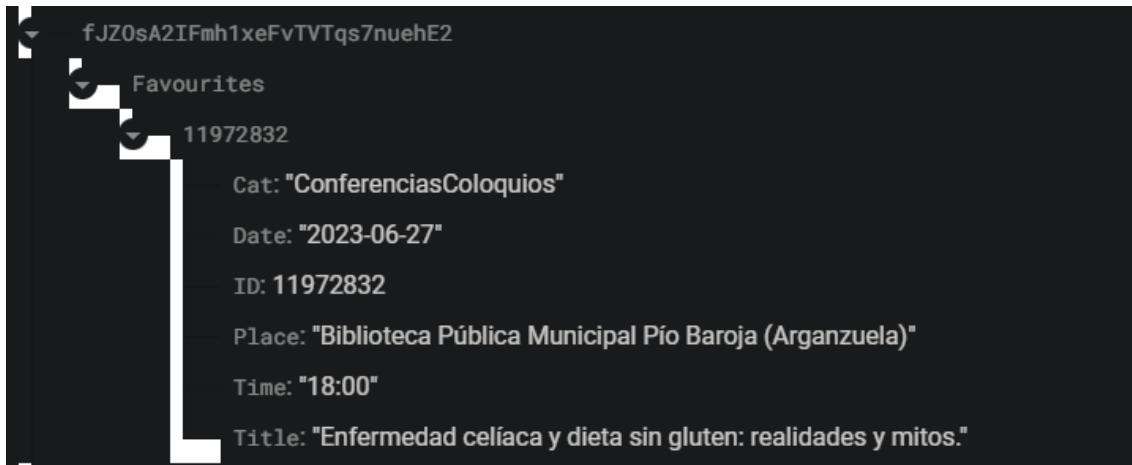
    hashMap ["Date"] = plAct.fecha
    hashMap ["Time"] = plAct.horario
    hashMap ["Place"] = plAct.lugar
    hashMap ["Cat"] = plAct.cat

    val ref = FirebaseDatabase.getInstance().getReference( path: "users");
    ref.child(firebaseAuth.uid!!).child( pathString: "Favourites").child(plAct.id.toString()) DatabaseReference
        .setValue(hashMap) Task<Void>
        .addOnSuccessListener { it: Void!
            Log.d(TAG, msg: "addToFavourite: Added to fav")
        }
        .addOnFailureListener{e ->
            Log.d(TAG, msg: "addToFavourite: Failed to add to fav due to ${e.message}")
        }
}

```

Función: añadir la actividad a favoritos

En la base de datos de *Firebase* a tiempo real se puede observar cómo se guarda toda la información contenida en el *HashMap*.



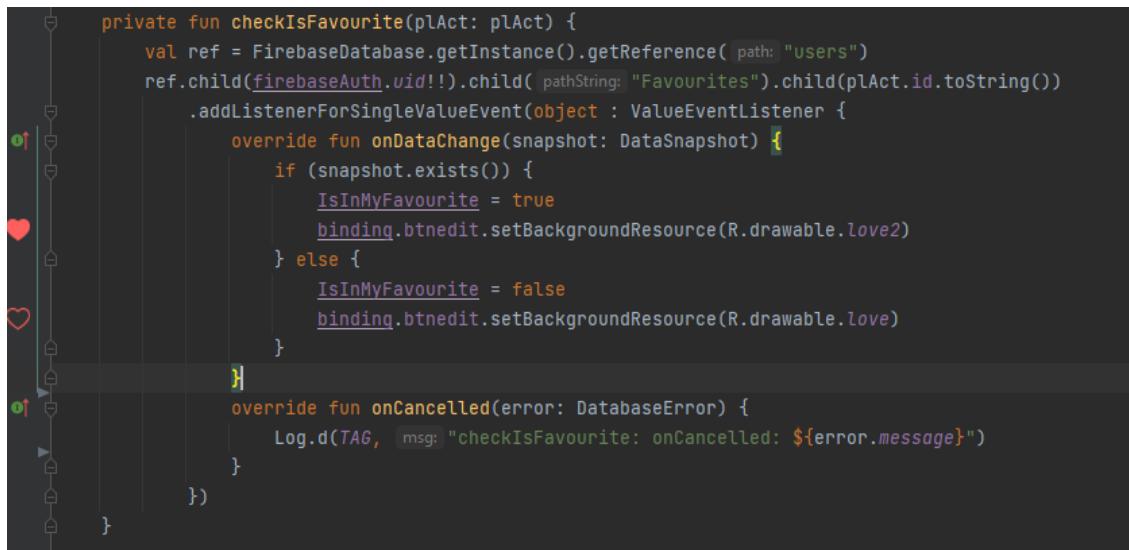
Actualización de la base de datos con favoritos

A su vez, para poder eliminarlo se realiza un procedimiento similar, pero en este caso solo es necesario el objeto *plAct* para poder actuar y en vez de utilizar la función *setValue()* en el directorio de la base de datos, se utiliza la función *removeValue()* para eliminarla.

```
private fun removeFromFavourite(plAct: plAct){
    Log.d(TAG, msg: "removeFromFavourite: Removing from fav")
    IsInMyFavourite = false
    checkIsFavourite(plAct)
    val ref=FirebaseDatabase.getInstance().getReference( path: "users")
    ref.child(firebaseAuth.uid!!).child( pathString: "Favourites").child(plAct.id.toString()) DatabaseReference
        .removeValue() Task<Void!
        .addOnSuccessListener { it: Void!
            Log.d(TAG, msg: "removeFromFavourite: removed from fav")
        }
        .addOnFailureListener{e ->
            Log.d(TAG, msg: "removeFromFavourite: Failed to remove from fav due to ${e.message}")
        }
}
```

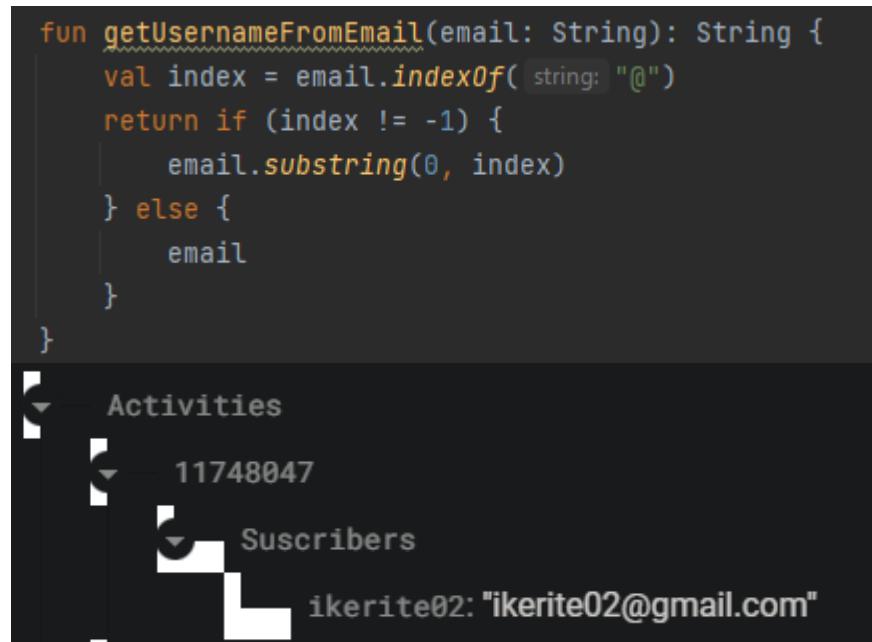
Función: eliminar actividad de favoritos

Para poder verificar si una actividad se encuentra guardada por el usuario se empleará la función *checkIsFavourite()* para que, de la misma manera que las anteriores funciones, acceda al directorio, y a partir de ahí, obteniendo los datos guardados, verifique si se encuentra en dicho directorio y, mediante una variable booleana, modifique el botón de favoritos con la finalidad de que, en caso de no estar en dicha base, solo se muestre el trazo del corazón o por el contrario, el corazón con relleno. Cada vez que se acciona dicho botón (agrega o elimina al usuario), se vuelve a ejecutar la función *checkIsFavourite()* para actualizarlo.



Función: comprobar si la actividad está en favoritos

En este mismo fragmento también se encuentra el botón de suscripción al plan. El procedimiento de este es similar a los anteriores, pues se realiza la conexión a la base de datos con un directorio distinto y almacena el nombre del correo del usuario (sin el carácter “@” ni el dominio) en una variable que contendrá el correo como valor. Todo esto es gracias a la función `getUsernameFromEmail()` que tiene como parámetro un cadena de caracteres para pasar el email y al detectar el carácter “@” este elimina todo lo posterior a él. Por lo tanto, en `Firebase` se vería de la siguiente manera:



Obtención del usuario para la suscripción a la actividad

Las funciones de eliminación `removeFromSubscribe()` y verificación `checkIsSubscribe()` actúan del mismo modo que en el caso anterior, con los siguientes cambios: en la verificación de la suscripción, el botón cambiará de color y se modificará el texto del mismo, tomando sus valores de los archivos “strings.xml” y “colors.xml”. A su vez, dependiendo del estado de la suscripción, se mostrará un *FrameLayout* el cual contiene tres *ImageViews* y un *TextView*.

```
private fun checkIsSubscribe(plAct: plAct, userEmail: String) {
    val ref = FirebaseDatabase.getInstance().getReference("Activities")
    ref.child(plAct.id.toString()).child(pathString: "Subscribers").addSingleValueEvent(object : ValueEventListener {
        @SuppressLint("ResourceAsColor")
        override fun onDataChange(snapshot: DataSnapshot) {
            val usernameFromEmail = getUsernameFromEmail(userEmail)
            var isInSubscribe = false
            snapshot.children.forEach { childSnapshot ->
                val username = childSnapshot.key
                if (username == usernameFromEmail) {
                    isInSubscribe = true
                    return@forEach
                }
            }
            isInSubscribe = isInSubscribe
            if (isInSubscribe) {
                binding.unsubscribe.apply { this: Button
                    backgroundTintList = ContextCompat.getColorStateList(requireContext(), R.color.secondary_dark)
                    setTextColor(ContextCompat.getColor(requireContext(), R.color.light_gray))
                    text = "Unsubscribe"
                }
                binding.bubbles.visibility = View.VISIBLE
            } else {
                binding.unsubscribe.apply { this: Button
                    backgroundTintList = ContextCompat.getColorStateList(requireContext(), R.color.secondary_light)
                    setTextColor(ContextCompat.getColor(requireContext(), R.color.light_gray))
                    text = "Subscribe"
                }
                binding.bubbles.visibility = View.GONE
            }
        }
    })
}
```

Función: verificación suscritos

Los *ImageView* actúan a modo de burbujas mostrando imágenes (sin derechos) de personas como si fuesen los usuarios suscritos. Estas imágenes, se cargan en el método `cargarImagenesAleatorias()` mediante un array con ocho imágenes extraídas de [pixabay](#) y, de forma aleatoria, selecciona tres de ellas para ser mostradas. Antes de que el usuario las pueda ver, sufren un proceso de diseño y para ello, se utilizan los métodos `squareCrop()` y `circleCrop()`. Estos reciben un bitmap como parámetro (imagen seleccionada) y la transforman en un cuadrado y en un círculo.

```

fun cargarImagenesAleatorias() {
    val imageViews = arrayOf(
        binding.bubbleImage1,
        binding.bubbleImage2,
        binding.bubbleImage3
    )

    val arrayFotos = arrayOf(
        R.drawable.imagen1,
        R.drawable.imagen2,
        R.drawable.imagen3,
        R.drawable.imagen4,
        R.drawable.imagen5,
        R.drawable.imagen6,
        R.drawable.imagen7,
        R.drawable.imagen8
    )

    for (imageView in imageViews) {
        val fotoAleatoria = arrayFotos.random()
        val bitmap = BitmapFactory.decodeResource(resources, fotoAleatoria)
        val squareBitmap = squareCrop(bitmap)
        val circularBitmapDrawable = circleCrop(squareBitmap)
        imageView.setImageDrawable(circularBitmapDrawable)
    }
}

```

Selección imágenes aleatorias para burbujas de suscritos

El *TextView* contenido también en el *FrameLayout*, únicamente tiene la función de contador de usuarios suscritos a la actividad, con la misma conexión a la base de datos mencionada y mostrada anteriormente.

```

suscribersRef.addValueEventListener(object : ValueEventListener {
    override fun onDataChange(snapshot: DataSnapshot) {
        numberSubs = 0
        val subscribersList = mutableListOf<Subscriber>()
        for (subscriberSnapshot in snapshot.children) {
            val subscriberName = subscriberSnapshot.key
            if (subscriberName != null) {
                System.out.println("cuenta")
                numberSubs += 1
            }
        }
    }
})

```

Contador de personas suscritas

Al presionar el botón “unirse” dependiendo de su estado, el usuario se unirá al plan o se retirará, sin embargo, al unirse al plan se activará la función *openSuscribersActivity()* que, únicamente, abre la actividad de la lista de suscriptores. Para mejorar la experiencia del usuario con la aplicación, se ha añadido un retardo de un segundo a la función

`checkIsSubscribe()` debido a que la apertura de la actividad que muestra los suscriptores conlleva un tiempo mayor de ejecución que la actualización del botón y el usuario puede observar ese cambio brusco de botón y no es estéticamente atractivo.

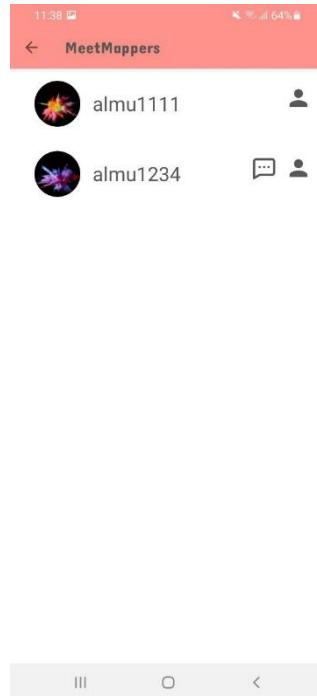
```
addToSubscribe(plAct, email)

Handler().postDelayed({
    checkIsSubscribe(plAct, email)
}, delayMillis: 1000)
openSuscribersActivity(plAct.id, email)
```

Retardo para la muestra de botones de suscripción

➤ Suscribers activity

Esta actividad, muestra toda la lista de “MeetMappers”, es decir, personas suscritas al plan, y contiene la foto de perfil del usuario, su nombre identificativo procedente de su dirección de correo electrónico y dos botones que permiten al usuario iniciar un chat con esa persona o mostrar su perfil. Para ello se han empleado las clases *Suscribers* y *SuscribersAdapter*.



Vista de MeetMappers (personas suscritas)

En esta clase, únicamente se puede observar una función denominada `updateRecyclerView()` que permite el acceso a la base de datos y por cada registro que se encuentre, crea un objeto llamado *Suscriber*, que contendrá el nombre identificativo y el email del mismo, y será añadido

a la lista de suscriptores. Acto seguido, en el adaptador se actualiza la lista pasándola como parámetro.

```
private fun updateRecyclerView() {
    val subscribersRef = FirebaseDatabase.getInstance().getReference(path: "Activities")
        .child(pActivityId.toString())
        .child(pathString: "Subscribers")

    subscribersRef.addValueEventListener(object : ValueEventListener {
        override fun onDataChange(snapshot: DataSnapshot) {
            val subscribersList = mutableListOf<Subscriber>()
            for (subscriberSnapshot in snapshot.children) {
                val subscriberName = subscriberSnapshot.key
                val subscriberMail = subscriberSnapshot.value.toString()
                if (subscriberName != null) {
                    val subscriber = Subscriber(subscriberName, subscriberMail)
                    subscribersList.add(subscriber)
                }
            }
            adapter.updateList(subscribersList)
        }

        override fun onCancelled(error: DatabaseError) {
            Log.w(TAG, msg: "Failed to read value.", error.toException())
        }
    })
}
```

Función: actualización de suscriptores

En la clase del adaptador se crea la función *bind()* que tiene como parámetro el suscriptor. Esta función, accede a la base de datos que tiene el registro de usuarios, es decir, la base de datos *Firebase*, y con esta conexión se accede mediante el correo de los usuarios a toda su información de perfil, lo que permite obtener su imagen de perfil para que pueda ser impresa en la vista. El usuario que está realizando la consulta de perfiles no tendrá habilitado el botón de chat consigo mismo, por lo tanto, su visibilidad y función se deshabilita.

```
usersCollection.document(userEmail) DocumentReference
    .get() Task<DocumentSnapshot>
        .addOnSuccessListener { documentSnapshot ->
            if (documentSnapshot.exists()) {
                val img = documentSnapshot.getString(field: "img")
                Glide.with(itemView.context) RequestManager
                    .load(img) RequestBuilder<Drawable>
                    .circleCrop()
                    .into(pfImage)
            } else {
                Log.d(tag: "SubscribersAdapter", msg: "El documento no existe")
            }
        }
        .addOnFailureListener { exception ->
            // Manejar la falla de la consulta Firestore
            Log.e(tag: "SubscribersAdapter", msg: "Error al obtener la imagen: $exception")
        }
    }
```

Obtención de la imagen para la lista de suscriptores

Cada uno de estos botones, al ser pulsados, ejecutarán una función distinta.

Si el botón pulsado es el de vista de perfil, accionará el método *openPfpActivity()* que realiza la apertura de la actividad “ProfileViewActivity”, pasando como parámetro el email del suscriptor.

```
private fun openPfpActivity(userEmail: String) {
    val intent = Intent(itemView.context, ProfileViewActivity::class.java)
    intent.putExtra("name", userEmail)
    itemView.context.startActivity(intent)
}
```

Función: apertura vista perfil usuario suscrito

Si se pulsa el botón del chat, el usuario podrá iniciar un chat, si no lo había hecho con anterioridad, o continuar con el chat que ya tenía iniciado con el suscriptor seleccionado. Esto es posible por la existencia de la función *newChat()* que se explicará posteriormente.

La función *onCreateViewHolder()* procede a inflar la actividad, es decir, el ítem que será listado por el RecyclerView, denominado “list_item_suscriber.xml”.

La función *onBindViewHolder()* separa los suscriptores individualmente para poder tratarlos utilizando la función *bind()* mencionada anteriormente.

```
override fun onBindViewHolder(holder: SuscriberViewHolder, position: Int) {
    val subscriber = subscribersList[position]
    holder.bind(subscriber)
}
```

Separación individual de los suscriptores

La clase *SuscribersActivity*, gracias a la *toolbar*, permite implementar la función *finish()* en el botón identificativo de flecha, con un título personalizado obtenido del archivo “strings.xml”, con un tipo de letra específico.

```
val customFont = ResourcesCompat.getFont(context, R.font.concert_one)
binding.toolbar.title = "MeetMappers"
binding.toolbar.setTitleTextAppearance(context, R.style.ToolbarTitleStyleMeetMappers)
binding.toolbar.getChildAt(0)?.let { toolbarTitle ->
    if (toolbarTitle is TextView) {
        toolbarTitle.typeface = customFont
    }
}
// Obtener la lista de suscriptores de Firebase
```

Diseño toolbar vista suscriptores

➤ ProfileView Activity

Esta actividad nos muestra la información del usuario seleccionado de la vista anterior, es decir, de un usuario suscrito al plan. En esta vista se muestra toda la información guardada en la base de datos acerca de dicho usuario.



Vista perfil usuario suscrito

Esta actividad tiene una apariencia similar al fragmento *EditProfile*, la intención del equipo de desarrolladores con esta decisión consistía en no romper la coherencia estética de la aplicación. La diferencia principal de esta vista con la de editar perfil, es que los campos de texto no son editables y no se puede realizar ninguna modificación de los datos.

La obtención de los datos se realiza gracias a la función *getUserData()* que recibe como parámetro el correo del usuario, que proviene de la actividad anterior. En esta función, se realiza la conexión con la base de datos de Firestore y se accede a los datos del usuario mediante el parámetro de correo mencionado anteriormente. Debido a ello, los *TextViews* e *ImageView* se completan con la información obtenida y se muestra en dicha actividad.

```

private fun getUserData(useremail: String) {
    Log.d(TAG, msg: "getUserData: $useremail")
    val documentRef = Firebase.firestore.collection( collectionPath: "users").document(useremail)

    documentRef.get()
        .addOnSuccessListener { documentSnapshot ->
            Log.d(TAG, msg: "getUserData: DocumentSnapshot exists")

            val name = documentSnapshot.getString( field: "name")
            val surname = documentSnapshot.getString( field: "surname")
            val numTelf = documentSnapshot.getString( field: "phone")
            val desc = documentSnapshot.getString( field: "description")
            val photo = documentSnapshot.getString( field: "img")

            name?.let { etName.text = it }
            surname?.let { etSurname.text = it }
            numTelf?.let { etPhone.text = it }
            desc?.let { etDescription.text = it }
            etEmail.text = useremail
            binding.toolbar.title = "Profile of" + " " + getUsernameFromEmail(useremail)

            photo?.let { it: String
                val options = RequestOptions().placeholder(R.drawable.predeterminado)
                    .error(R.drawable.predeterminado)
                Glide.with( activity: this) RequestManager
                    .load(it) RequestBuilder<Drawable>
                    .apply(options)
                    .circleCrop()
                    .into(etPfpic)
            }
        }
        .addOnFailureListener { exception ->
            Log.e(TAG, msg: "Error reading user data: ${exception.message}")
        }
}

```

Función: obtención datos para ser mostrados en la vista del perfil

Esta actividad, también cuenta con una *appbar* similar a la mencionada anteriormente tanto en diseño como en funcionalidad. La única diferencia es el título que se muestra puesto que va cambiando en función del usuario que se haya seleccionado.

```

binding.toolbar.title = "Profile of" + " " + getUsernameFromEmail(useremail)

```

Personalización título appbar en la vista perfil de usuario suscrito

❖ Favourites Fragment

Este fragmento presenta un *RecyclerView* con un listado de actividades que el usuario ha seleccionado como favoritas. En cada actividad se muestra la información más relevante de forma clara para mejorar la experiencia del usuario. Para conseguir todo ello, se han empleado las clases *FLI* y *AdapterFLI*.



Vista favoritos

En este fragmento se ejecuta, como función principal, `loadFavouriteActs()` que permite el acceso la base de datos para crear un objeto llamado `FLI` que almacena todos los datos de las actividad guardadas como favoritas por parte del usuario.

```
class FLI {
    var id: String=""
    var titulo: String=""
    var fecha: String=""
    var horario: String=""
    var lugar: String=""
    var isFav: Boolean=false

    ▲ SoySandyYT
    constructor()

    ▲ SoySandyYT
    constructor(
        id:String,
        titulo: String,
        fecha: String,
        horario: String,
        lugar: String,
        isFav: Boolean
    ){
        this.id = id
        this.titulo = titulo
        this.fecha = fecha
        this.horario = horario
        this.lugar = lugar
        this.isFav = isFav
    }
}
```

Creación objeto con los datos de actividades favoritas

Si es la primera vez que el usuario accede a esta sección y no ha seleccionado ninguna actividad como favorita, se mostrará una imagen y un texto para incentivar al usuario a seguir navegando por la aplicación.



Qué tranquilo está todo por aquí...



Vista favoritos cuando no hay ninguna actividad guardada

```

private fun loadFavouriteActs() {
    fliArrayList = ArrayList()
    val ref = FirebaseDatabase.getInstance().getReference(path: "users")
    ref.child(firebaseAuth.uid!).child(pathString: "Favourites")
        .addValueEventListener(object : ValueEventListener {
            override fun onDataChange(snapshot: DataSnapshot) {
                var boolData = false
                var cnt = 0
                val data = ArrayList<FLI>()
                for (ds in snapshot.children) {
                    val aid = "${ds.child(path: "ID").value}"
                    val FLI = FLI()
                    FLI.id = aid
                    data.add(FLI)
                    if(cnt == 0){
                        boolData = true;
                        cnt++;
                    }
                    System.out.println("boolean "+boolData)
                    if(boolData){
                        binding.emptyRecyclerViewImageview.visibility = View.GONE
                        binding.emptyRecyclerViewTextview.visibility = View.GONE
                    }else{
                        System.out.println(" entra en else")
                        binding.emptyRecyclerViewImageview.visibility = View.VISIBLE
                        binding.emptyRecyclerViewTextview.visibility = View.VISIBLE
                    }
                    updateRecyclerViewData(data)
                    updateRecyclerViewData(data)

                    //updateEmptyRecyclerViewVisibility(adapterFLI)
                }
            }

            override fun onCancelled(error: DatabaseError) {
            }
        })

        // Mover la inicialización del adaptador aquí
        adapterFLI = AdapterFLI(requireContext(), fliArrayList)
        adapterFLI.clickListener = this
        recyclerView.adapter = adapterFLI
    }
}

```

Función: carga de actividades favoritas

La clase *AdapterFLI* tiene como función principal *loadActDetails()* que recibe como parámetros los objetos de la lista que se ha cargado previamente y el *holder* del adaptador. En dicha función, se accede a la base de datos a tiempo real de *Firebase* y se extrae toda la información relacionada con las actividades favoritas del usuario para que se pueda imprimir en los distintos *TextView* de cada actividad. En función de la categoría del plan, se mostrará un ícono representativo del evento. Esto se consigue a través de una serie de sentencias condicionales encadenadas.

```

if (cat == "Musica") {
    holder.fli_cat.setBackgroundResource(R.drawable.ico_musica)
} else if (cat == "DanzaBaile"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_danzabaile)
} else if (cat == "CursosTalleres"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_cursostalleres)
} else if (cat == "TeatroPerformance"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_teatro)
} else if (cat == "ActividadesCalleArteUrbano"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_arteurbano)
} else if (cat == "CuentacuentosTiteresMarionetas"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_cuentacuentos)
} else if (cat == "ConferenciasColoquios"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_conferencias)
} else if (cat == "1ciudad21distritos"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_ciudaddistritos)
} else if (cat == "ExcusionesItinerariosVisitas"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_campamentos)
} else if (cat == "ItinerariosOtrasActividadesAmbientales"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_ambientales)
} else if (cat == "ClubesLectura"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_lectura)
} else if (cat == "RecitalesPresentacionesActosLiterarios"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_recitales)
} else if (cat == "Exposiciones"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_exposiciones)
} else if (cat == "Campamentos"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_campamentos)
} else if (cat == "CineActividadesAudiovisuales"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_cine)
} else if (cat == "CircoMagia"){
    holder.fli_cat.setBackgroundResource(R.drawable.ico_circo)
}

```

Selección ícono en función de categoría de la actividad

El *holder* que se ha pasado como parámetro para esta función es una clase interna que se ha creado en esta misma actividad, es decir, dentro de *AdapterFLI* está creada la clase *HolderFLI* para mayor comodidad a la hora de realizar la carga de datos, debido a que esta clase únicamente crea variables que se asignan a los distintos *TextViews* e *ImageViews* que existen en el ítem.

```
inner class HolderFLI(itemView: View): RecyclerView.ViewHolder(itemView){
    var fli_title = binding.fliTitle
    var fli_cat = binding.catIV
    var fli_desc = binding.fliDesc
    var fli_time = binding.fliTime
    var fli_date = binding.fliDate
    var removeFavBtn = binding.removeFavBtn
}
```

Clase interna para obtener los datos de favoritos

Como se puede observar en el *holder* y en el desarrollo de la función, existe una variable independiente de la base de datos: la imagen del corazón relleno. Esta imagen está predeterminada por el equipo de desarrolladores y su función consiste en poder eliminar las actividades de la lista de favoritos directamente desde el propio fragmento, para facilitar la eliminación al usuario y que tenga una mejor experiencia en la aplicación. Para borrar la actividad de la lista, se ejecuta la función *removeFromFavourites()* que, como también se puede observar en la explicación del fragmento *InfoActivity*, elimina el registro de la base de datos.

```
,
```

```
holder.removeFavBtn.setOnClickListener{ it: View!
    removeFromFavourite(context, model.id)
}
```

Escucha del botón del corazón para eliminar la actividad de favoritos

También existe otra escucha en el código que permite pulsar sobre cada ítem de la lista y dirige al usuario a la información detallada de la actividad.

```
holder.itemView.setOnClickListener { it: View!
    clickListener?.onItemClick(position, model, holder.itemView)
}
```

Escucha ítem de actividad favorita para abrir detalle de actividad

Inicialmente, para poder identificar si el *RecyclerView* tenía ítems para mostrar o, por el contrario, estaba vacío, el equipo de desarrolladores pensó en hacer esta comprobación en una función propia, sin embargo, a la hora de implementarlo daba problemas debido a que la detección del conteo del adaptador se realizaba antes de que la conexión a la base de datos se cargara correctamente, por ello hubo que implementarlo dentro de la función

loadFavouriteActs() de manera que consiguiera detectar de una forma distinta si ha existido carga de datos.

```
private fun updateEmptyRecyclerViewVisibility(adapter: AdapterFLI) {
    if (adapter.itemCount == 0) {
        binding.emptyRecyclerViewImageview.visibility = View.VISIBLE
        binding.emptyRecyclerViewTextview.visibility = View.VISIBLE
    } else {
        binding.emptyRecyclerViewImageview.visibility = View.GONE
        binding.emptyRecyclerViewTextview.visibility = View.GONE
    }
}
```

Función: actualización lista de favoritos y su visibilidad

❖ Chat Fragment

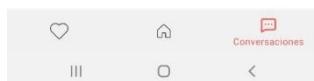
Otro de los ítems del menú principal de navegación es la sección de chats. En este fragmento se presenta los chats que tenemos abiertos con otros usuarios de la app y, si todavía no hay chats activos, aparecerá una imagen invitando a iniciar alguna conversación con otros usuarios.

```
private fun updateEmptyRecyclerViewVisibility(adapter: ChatAdapter){
    if (adapter.itemCount == 0) {
        binding.emptyRecyclerViewImageview.visibility = View.VISIBLE
        binding.emptyRecyclerViewTextview.visibility = View.VISIBLE
    } else {
        binding.emptyRecyclerViewImageview.visibility = View.GONE
        binding.emptyRecyclerViewTextview.visibility = View.GONE
    }
}
```

Comprobación de la existencia de chats iniciados



Qué tranquilo está todo por aquí...



Vista de la lista de chats cuando no se ha iniciado ninguno

La forma de iniciar conversaciones será a partir de la vista del detalle de la actividad o, escribiendo el correo de la persona con la que se quiere hablar. En cualquier de las dos opciones, la aplicación validará que el correo que se ha introducido pertenezca a un usuario dado de alta en la base de datos y si ya se ha iniciado un chat anteriormente con ese usuario no se creará un chat nuevo si no que se mostrará el chat que estaba ya activo.

Los chats que el usuario ha iniciado se muestran en forma de lista donde cada conversación se presenta con el nombre del correo del usuario receptor y con el último mensaje que se ha escrito en ese chat. Si este último mensaje pertenece al usuario actual de la app aparecerán dos *ticks* al lado del mensaje si, por el contrario, el mensaje pertenece al receptor solo se verá el mensaje.

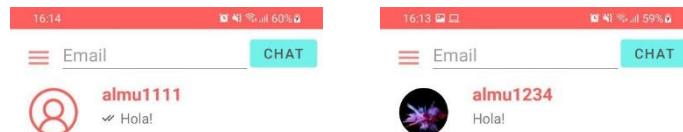
Esta lista está implementada con un *Recycler View* y su respectivo *adapter*. Es en esta clase *adapter* donde se crea el contenido de cada ítem de la lista haciendo las comprobaciones de los usuarios y de los mensajes:

```

val currentUser = PreferencesManager.getDefaultSharedPreferences(holder.itemView.context).getEmail()
val chat = chats[position]
val chatId = chat.id
val messagesRef = db.collection("chats").document(chatId).collection("messages")
val otherUser = if (chat.users[0] == currentUser) chat.users[1] else chat.users[0]
holder.binding.chatNameText.text = otherUser.substringBefore(delimiter: "@")
messagesRef.orderBy(field: "dob", Query.Direction.DESCENDING).limit(1)
    .addSnapshotListener { querySnapshot, exception ->
        if(exception != null){
            holder.binding.usersTextView.text = "Error al obtener mensajes"
            return@addSnapshotListener
        }
        if (querySnapshot != null && !querySnapshot.isEmpty) {
            val layoutParams = holder.binding.chatmessage.layoutParams as ConstraintLayout.LayoutParams
            val textLayoutParams = holder.binding.usersTextView.layoutParams as ConstraintLayout.LayoutParams
            val lastMessage = querySnapshot.documents[0].toObject(Message::class.java)
            val lastMessageText = lastMessage?.message
            val lastMessageSender = lastMessage?.from
            if (currentUser == lastMessageSender){
                holder.binding.chatmessage.visibility = View.VISIBLE
                layoutParams.marginStart = 32.dpToPixels(context)
                textLayoutParams.startToEnd = R.id.chatmessage
            } else {
                holder.binding.chatmessage.visibility = View.GONE
                layoutParams.marginStart = 0.dpToPixels(context)
                textLayoutParams.marginStart = 32.dpToPixels(context)
            }
            holder.binding.chatmessage.layoutParams = layoutParams
            holder.binding.usersTextView.layoutParams = textLayoutParams
            holder.binding.usersTextView.text = lastMessageText
        } else {
            holder.binding.usersTextView.text = "No hay mensajes"
        }
    }
}

```

Creación del contenido del ítem chat en el adapter



Vista lista de chats con último mensaje

La creación de estos chats se guarda en la base de datos de *Firebase* y es a partir de las llamadas a la base de datos donde se comprueba si el chat ya está iniciado y si el usuario está dado de alta en la aplicación.

```
private fun chatUp(otherUser: String, chatId: String)
{
    val users = listOf(user, otherUser)

    val chat = Chat(
        id = chatId,
        name = "Chat con $otherUser",
        users = users
    )

    db.collection( collectionPath: "chats").document(chatId).set(chat)
    db.collection( collectionPath: "users").document(user).collection( collectionPath: "chats").document(chatId).set(chat)
    db.collection( collectionPath: "users").document(otherUser).collection( collectionPath: "chats").document(chatId).set(chat)

    val intent = Intent(requireContext(), ChatActivity::class.java)
    intent.putExtra( name: "chatId", chatId)
    intent.putExtra( name: "user", user)
    intent.putExtra( name: "name", otherUser)
    startActivity(intent)
}
```

Función para guardar chat en Firebase

Una vez que se ha seleccionado un chat o un correo para iniciar la conversación se abrirá una nueva vista en la que los usuarios podrán comenzar a conversar.

```
private fun chatSelected(chat: Chat){
    val intent = Intent(requireContext(), ChatActivity::class.java)
    intent.putExtra( name: "chatId", chat.id)
    intent.putExtra( name: "user", user)
    if(user==chat.users[0])
    {
        intent.putExtra( name: "name", chat.users[1])
    }
    else if(user==chat.users[1])
    {
        intent.putExtra( name: "name", chat.users[0])
    }
    startActivity(intent)
}
```

Acceso desde la lista de chats al chat individual

➤ Chat Activity

Esta vista se encuentra fuera de la actividad principal y, por ese motivo, no se puede acceder al menú de navegación ni al menú lateral. Es una actividad independiente que se conecta con el resto de la aplicación a través de la lista de chats. Por ello, en esta vista se hacen las comprobaciones necesarias de conectividad a internet.

Al iniciarse esta actividad el usuario puede presenciar dos estados de la vista:

- Sin mensajes
- Con mensajes

Si la actividad está sin mensajes, se muestra vacía y a la espera de que el usuario empiece la conversación.

Si el usuario ya ha comenzado la conversación y cierra la aplicación o cambia de actividad dentro de ella, cuando vuelve a un chat ya iniciado, se deben mostrar los mensajes que tenía y para ello, es necesario recuperarlos de la base de datos y mostrarlos mediante una función que se lanza al iniciarse la actividad.

```
private fun initViews(){
    binding.messagesRecylerView.layoutManager = LinearLayoutManager( context: this)
    binding.messagesRecylerView.adapter = MessageAdapter(user)
    val text= name.substringBefore( delimiter: "@")
    binding.toolbar.title = "Chat con $text"
    binding.sendMessageButton.setOnClickListener { sendMessage() }

    val chatRef = db.collection( collectionPath: "chats").document(chatId)

    chatRef.collection( collectionPath: "messages").orderBy( field: "dob", Query.Direction.ASCENDING) .Query
        .get() Task<QuerySnapshot>
        .addOnSuccessListener { messages ->
            val listMessages = messages.toObjects(Message::class.java)
            (binding.messagesRecylerView.adapter as MessageAdapter).setData(listMessages)
        }

    chatRef.collection( collectionPath: "messages").orderBy( field: "dob", Query.Direction.ASCENDING)
        .addSnapshotListener { messages, error ->
            if(error == null){
                messages?.let { it: QuerySnapshot -
                    val listMessages = it.toObjects(Message::class.java)
                    (binding.messagesRecylerView.adapter as MessageAdapter).setData(listMessages)
                }
            }
        }
}
```

Inicio vista chat individual con recuperación de mensajes

Desde esta actividad se puede navegar a la lista mediante un botón de retroceso (flecha) que permite volver hacia atrás y que está acompañado de un texto que indica el usuario con el que se está conversando.



Barra de navegación chat individual

Desde el punto de vista del diseño, en esta vista, al usuario que habla le corresponde el color principal de la app en su tonalidad oscura para mejorar la accesibilidad y al receptor el color secundario que es el mismo color que se emplea para el título del chat. También se puede observar que el botón de enviar está personalizado con el imagotipo de MeetMap.



Diseño vista chat individual

Los mensajes de cada chat individual están implementados mediante un *Recycler View* donde se va comprobando la procedencia de cada mensaje para establecer en qué lugar se tiene que mostrar el mensaje y con qué color.

Cuando el usuario escribe un mensaje en el espacio destinado a ello y pulsa el botón de enviar, se ejecuta la función *sendMessage()* que guarda el mensaje en la base de datos y limpia el campo de texto para el siguiente mensaje.

```
private fun sendMessage(){  
    val message = Message(  
        message = binding.messageTextField.text.toString(),  
        from = user  
    )  
  
    db.collection( collectionPath: "chats").document(chatId).collection( collectionPath: "messages").document().set(message)  
  
    binding.messageTextField.setText("")  
}
```

Función enviar mensaje en chat individual

Una vez el mensaje se ha enviado, el *adapter* del *Recycler View* tiene que establecer la procedencia de dicho mensaje para mostrarlo en su ubicación correspondiente.

```
override fun onBindViewHolder(holder: MessageViewHolder, position: Int) {  
    val message = messages[position]  
  
    if(user == message.from){  
        holder.binding.myMessageLayout.visibility = View.VISIBLE  
        holder.binding.otherMessageLayout.visibility = View.GONE  
  
        holder.binding.myMessageTextView.text = message.message  
    } else {  
        holder.binding.myMessageLayout.visibility = View.GONE  
        holder.binding.otherMessageLayout.visibility = View.VISIBLE  
  
        holder.binding.othersMessageTextView.text = message.message  
    }  
}
```

Comprobación procedencia mensaje

Las siguientes vistas que se van a explicar, son las contenidas en el menú lateral de la actividad principal y por tanto son fragmentos incluidos en dicha actividad.

❖ Edit Profile Fragment

Cuando se inicia esta vista se hace una petición a la base de datos de *Firebase* que es donde se encuentran todos los datos del usuario que se van a mostrar en esta vista.

```
email = PreferencesManager.getDefaultSharedPreferences(binding.root.context).getEmail()
binding.email.setText(email)
fStore.collection("users").document(email).get().addOnSuccessListener { it: DocumentSnapshot? ->
    binding.nombre.setText(it.get("name") as String)
    binding.surnombre.setText(it.get("surname") as String)
    binding.phone.setText(it.get("phone") as String)
    binding.description.setText(it.get("description") as String)
    //binding.mail.setText(email)
    Glide.with(fragment: this) RequestManager
        .load(it.get("img") as String) RequestBuilder<Drawable!>
        .circleCrop()
        .into(img)
}
```

Obtención de los datos del usuario al iniciar la vista de editar perfil

Inicialmente, sin que el usuario haya editado ningún campo, en esta vista se muestra una imagen del sistema cargada por defecto y todos los campos vacíos excepto el del correo electrónico, que se completa con el email con el que el usuario se ha registrado y que no se puede modificar, puesto que es el identificador único del usuario en la aplicación.

La imagen se redondea para mostrarse en el perfil y el usuario dispone de un botón anexo a ella, que permite editar la foto de perfil con cualquier imagen que el usuario tenga en su dispositivo móvil. Este cambio, una vez que se han salvado todas las modificaciones se verá en todas las vistas y secciones de la aplicación donde se muestre la imagen de perfil: menú lateral, lista de chats, diferentes vistas de perfil, burbujas de las personas suscritas a la actividad, etc.

```
Glide.with(fragment: this) RequestManager
    .load(uri) RequestBuilder<Drawable!>
    .circleCrop()
    .into(img)
```

Diseño redondeado de la imagen del perfil

```
binding.btndedit.setOnClickListener{ it: View! }
    val intent = Intent(Intent.ACTION_PICK)
    intent.setType("image/*")
    startActivityForResult(intent, GALLERY_INTENT as Int)
}
```

Editar imagen perfil con apertura de galería del dispositivo móvil

```
fun saveImg(){
    val storageRef = Firebase.storage.reference.child( pathString: "img/" + fAuth.currentUser?.email)
    storageRef.downloadUrl.addOnSuccessListener { uri ->
        url = uri.toString()
        updateData()
    }
    .addOnFailureListener { exception ->
        Log.e( tag: "TAG", msg: "Error al descargar la imagen: ${exception.message}")
        url = PRED_URL
        updateData()
    }
}
```

Guardar los cambios de la imagen y campos del perfil

Los campos que estaban vacíos se pueden editar y de esta forma el usuario pueda personalizar su perfil: nombre, apellido, número de teléfono y descripción. Una vez hechos los cambios pertinentes si el usuario quiere guardar los cambios pulsará en el botón “Guardar”/“Save” y se actualizarán los cambios en la base de datos y donde se esté usando esta información y si no es así, pulsará en el botón de “Cancelar”/“Cancel” y los cambios editados se perderán.

```
binding.btnsave.setOnClickListener { it: View!
    saveImg()
    Handler().postDelayed({
        saveImg()
    }, delayMillis: 2000)

    Toast.makeText(requireActivity(), "Updated data", Toast.LENGTH_LONG).show()
    goBack()
}
```

Función del botón guardar para salvar cambios del perfil

```

fun updateData(){
    val updates = hashMapOf<String, Any>(
        "name" to binding.nombre.text.toString(),
        "surname" to binding.surnombre.text.toString(),
        "phone" to binding.phone.text.toString(),
        "description" to binding.description.text.toString(),
        "img" to url,
    )
    fStore.collection( collectionPath: "users").document(email).update(updates)
}

```

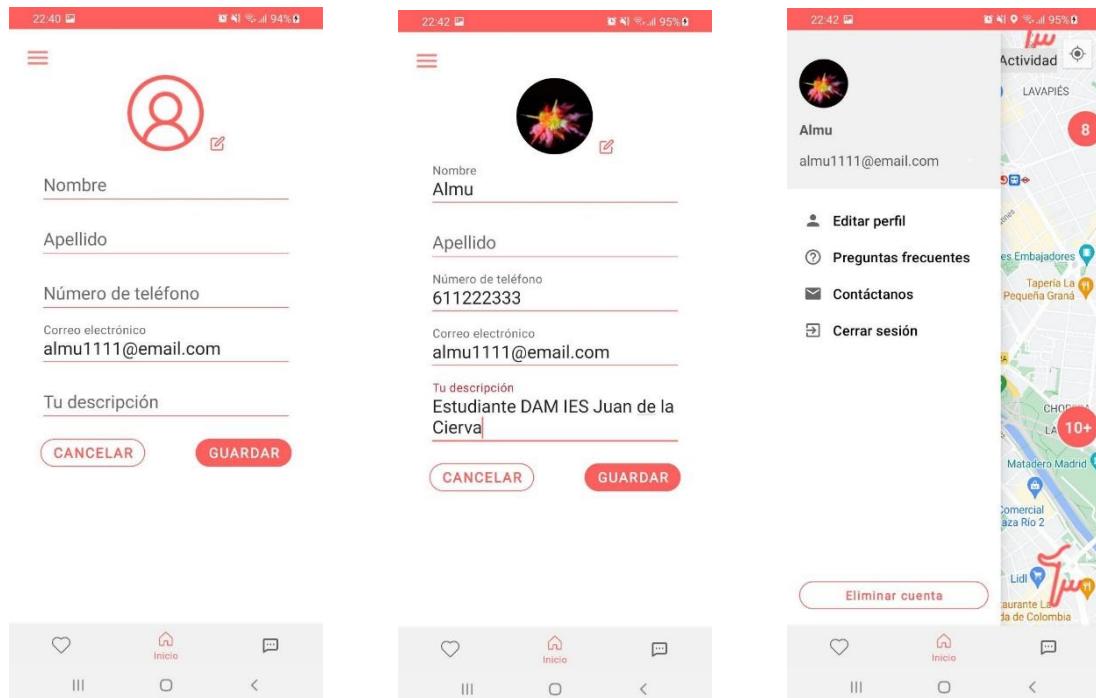
Actualización de los datos del usuario para guardarlos en Firebase

```

binding.btncancel.setOnClickListener { it: View! ->
    binding.email.setText(email)
    fStore.collection( collectionPath: "users").document(email).get().addOnSuccessListener {
        binding.nombre.setText(it.get("name") as String)
        binding.surnombre.setText(it.get("surname") as String)
        binding.phone.setText(it.get("phone") as String)
        binding.description.setText(it.get("description") as String)
        binding.email.setText(email)
    }
    goBack()
}

```

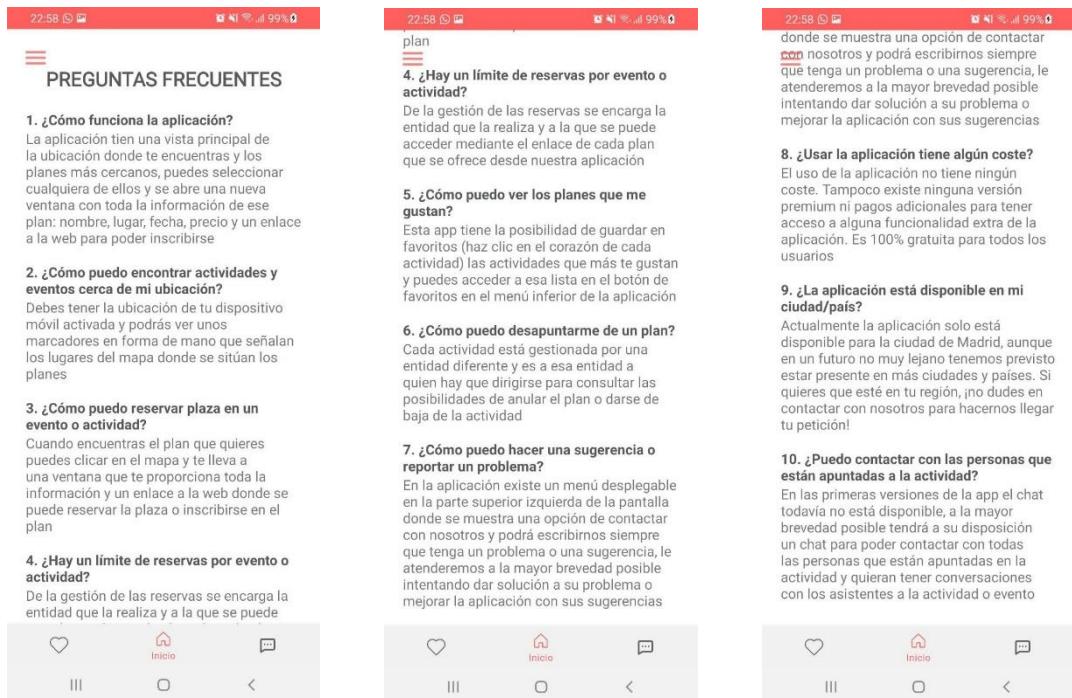
Función del botón cancelar para mantener datos anteriores



Edición del perfil y visualización de los cambios

❖ Fags Fragment

En esta vista se muestran diez preguntas frecuentes que se pueden hacer los usuarios al hacer uso de la aplicación. El equipo de desarrollo ha elaborado las respuestas para una mejor experiencia de usuario y se irán actualizando a medida que lleguen otra serie de dudas sobre la app.



Vista preguntas frecuentes para los usuarios

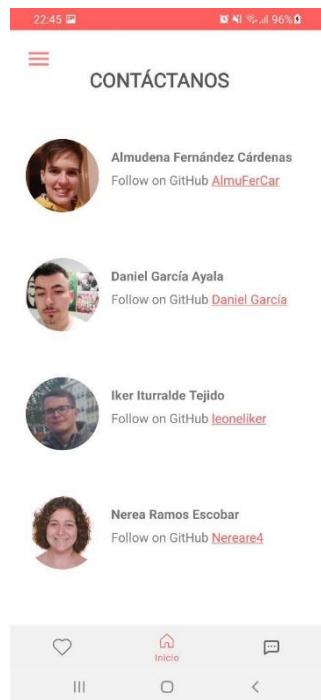
❖ ContactUs Fragment

En este apartado aparecen los nombres, fotos y enlaces al *GitHub* individual de cada desarrollador y administrador de la app para que los usuarios puedan conocerlos e interactuar con ellas y ellos.

```
fun createStyledLinkText(linkText: String, color: Int): SpannableString {
    val spannableString = SpannableString(linkText)
    spannableString.setSpan(ForegroundColorSpan(ContextCompat.getColor(requireContext(), color)), start: 0, spannableString.length, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE)
    //spannableString.setSpan(StyleSpan(Typeface.BOLD), 0, spannableString.length, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE)
    spannableString.setSpan(UnderlineSpan(), start: 0, spannableString.length, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE)
    return spannableString
}

val spannableStringAlmu = createStyledLinkText( linkText: "AlmuFerCar", R.color.secondary)
binding.tvGitAlmuLink.text = spannableStringAlmu
binding.tvGitAlmuLink.movementMethod = LinkMovementMethod.getInstance()
binding.tvGitAlmuLink.setOnClickListener { it: View ->
    val intent = Intent(Intent.ACTION_VIEW, Uri.parse( urlString: "https://github.com/AlmuFerCar"))
    startActivity(intent)
}
```

Estilo enlace desarrollador y conexión GitHub



Vista de contacto con los desarrolladores

Después de los apartados del menú lateral también aparece una opción de Cerrar Sesión que, si se pulsa, la app dirige al usuario a la vista inicial del carrusel para volver a iniciar sesión o registrarse con otra cuenta, de esta forma se borran las preferencias compartidas de la sesión, pero se mantiene la cuenta en la base de datos.

```
R.id.nav_exit -> {
    PreferencesManager.getDefaultSharedPreferences(binding.root.context).wipe()
    fAuth.signOut()
    startActivity(Intent(packageContext: this, Initial::class.java))
    Intent(packageContext: this, Initial::class.java)
}
```

Cierre de sesión

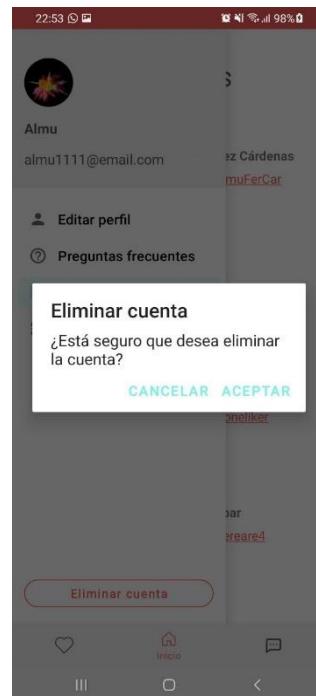
Por el contrario, si se pulsa en Eliminar Cuenta, se borrará la cuenta y toda la información asociada a la misma y, si posteriormente se quiere volver a iniciar sesión con ese correo, será necesario registrarse de nuevo. Como es un proceso irreversible, si el usuario ha pulsado sobre “Eliminar Cuenta” se mostrará una alerta para confirmar esta decisión y que el proceso sea más seguro evitando que con una pulsación accidental se elimine la cuenta.

```

btnDeleteAccount.setOnClickListener { it: View! -
    val builder = AlertDialog.Builder( context: this)
    builder.setTitle("Delete account")
    builder.setMessage("Are you sure you want to delete the account?")
    builder.setPositiveButton("Accept") { dialog, which ->
        fStore.collection( collectionPath: "users").document(email).delete()
        fAuth.currentUser?.delete()
        fStorage.child( pathString: "img").child(email).delete()
        Toast.makeText(
            context: this,
            "The account has been deleted",
            Toast.LENGTH_LONG
        ).show()
        PreferencesManager.getDefaultSharedPreferences(binding.root.context).wipe()
        startActivity(Intent( packageContext: this, Initial::class.java))
        Intent(binding.root.context, Initial::class.java)
    }
    builder.setNegativeButton("Cancel") { dialog, which -> }
    builder.show()
}

```

Creación de alerta y eliminación de cuenta de Firebase



Alerta para la eliminación de cuenta

5.2 Problemas encontrados y soluciones

A continuación, se van a explicar los problemas encontrados y las soluciones que el equipo de desarrolladores ha implementado.

Cierre del menú lateral

A la hora de implementar el menú lateral se quería mantener en todo momento el menú de navegación y por tanto la actividad predefinida que ofrecía *Android Studio (Navigation Drawer Activity)* no era útil para este objetivo. Por esta razón, se diseñó un menú deslizante de cero con la estructura y funcionalidad que se había acordado.

El problema surgió a la hora de cerrar este menú lateral puesto que la parte que se quedaba sin ser cubierta con el componente permitía la interacción con el usuario e imposibilitaba que el menú se pudiese ocultar.

La primera solución que se propuso fue la creación de un botón transparente que ocupase todo el espacio que quedaba de la actividad en cuestión cuando el menú se desplegase. Al principio fue una solución válida, pero a medida que se fueron haciendo pruebas en diferentes dispositivos con distintos tamaños de pantalla ese botón transparente dejaba zonas de interacción con el usuario.

Por lo tanto, la decisión final y que está implementada y validada fue colocar un botón transparente que ocupase toda la actividad de la aplicación y el menú lateral en una capa superior.



Diseño menú lateral con botón transparente

Este botón transparente ocupa todo el ancho y el alto de la vista y se ajusta al tamaño de la pantalla del dispositivo móvil que se esté empleando sin, cuando se muestra este botón, dejar espacio alguno para la interacción con el usuario.

La lógica de este botón es la siguiente: cuando se pulsa en el ícono del menú lateral (ícono de hamburguesa) se despliega el menú lateral y se muestra el botón (el usuario no lo ve porque el color de fondo es transparente) y, cuando se pulsa sobre este botón (zona no ocultada por el menú lateral) se oculta el menú y el botón, dejando toda la actividad visible para que el usuario interactúe con la aplicación.

También existe la posibilidad que el menú se cierre por la interacción del usuario con alguno de los ítems de este o por el cambio de sección del menú principal. En estos casos, el cierre del menú lateral va asociado a la ocultación del botón transparente.

```
private fun closeNav() {
    binding.navView.isVisible = false
    navview.visibility = View.GONE
    if (isnavview) {
        animateAndHideNavigationView(navview)
        isnavview = false
    }
    buttonsVisibility()
}
```



```
private fun buttonsVisibility() {
    handler.postAtTime(Runnable {
        {
            ImageButton.visibility = View.VISIBLE
            transparentButton.visibility = View.GONE
        }, uptimeMillis: SystemClock.uptimeMillis() + 850
    })
}
```

Lógica visibilidad menú lateral y botón transparente

Clustering en Map Fragment

Otro de los problemas que el equipo de desarrollo se encontró fue, reducir el número de marcadores que se veían en el mapa, puesto que al comienzo se solapaban las manos (marcadores personalizados) y quedaba una masa de color rojo sobre el mapa que no era consecuente con la accesibilidad que se deseaba (Ver figura [Antes y después de la implementación del clustering en el diseño del mapa](#) pág.58).

Por este motivo se tuvieron que hacer diferentes modificaciones de funciones y de diseño para lograr agrupar los marcadores cuando se concentraba en un determinado radio del mapa.

- **Observe()**: como bien se comentó anteriormente, esta es la función principal que permite leer cada *locator* desde la API y transformarlo en un *marker*. Sin embargo, la implementación del *clustering* exigía no crear *markers* propios de la Google Maps API, sino ítems en una lista mutable que se añadiría directamente al *ClusterManager*.

```

    val items = mutableListOf<MyItem>()
    items.clear()
    locators.mapNotNull {
        it.location.latitude?.let { lat ->
            it.location.longitude?.let { lng ->
                LatLng(lat, lng)
            }
        }?.let { coordinates ->
            val item = MyItem(coordinates, "${it.id} ${it.title}", "${it.time} ${it.dstart}")
            items.add(item)
            madridMap[item.getNombre()] = it.id
            val markerOptions = MarkerOptions().position(coordinates).title("${it.id} ${it.title}").visible(false)
            val marker = map.addMarker(markerOptions)
            markers[marker.title] = marker
            madridMap[marker.title] = it.id
        }
    }
}

```

Tratamiento lista mutable para clustering

El *ClusterManager* es una clase que extiende de la predeterminada de la biblioteca de *clustering* en la cual se pueden implementar las modificaciones y métodos necesarios, como color personalizado y la función que permite desplegar una lista en caso de que haya varios elementos agrupados aún con el *zoom* al máximo. Esto es importante resaltarlo porque con el mapa sin *clustering* en la aplicación existían diversas actividades solapadas al encontrarse en el mismo lugar. Por ejemplo, en Matadero (Legazpi), se puede observar cómo hay más de 20 planes en el mismo punto (Ver figura [Selección marker y listado actividades](#) pág.91).

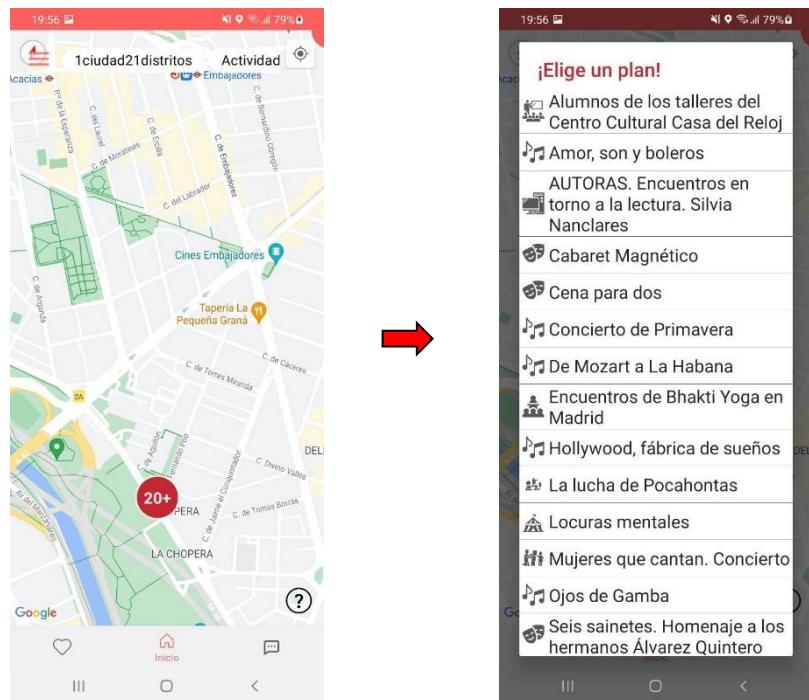
```

dialogBuilder.setAdapter(adapter) { dialog, which ->
    val selectedItem = cluster.items.elementAtOrNull( index: which-1 )
    if (selectedItem != null) {
        val marker = MapFragment.markers[selectedItem.getNombre()]
        Log.i( tag: "pruiea", selectedItem.getNombre() )
        if (marker != null) {
            val infoFragment = InfoActivityFragment()
            infoFragment.setMarker(marker, MapFragment.locatorListFav)
            val fragmentManager = this.fragmentManager
            fragmentManager.beginTransaction()
                .setCustomAnimations(android.R.anim.fade_in, android.R.anim.fade_out)
                .replace(R.id.frame, infoFragment)
                .addToBackStack( name: null )
                .commit()
        }
    }
}

```

Código para mostrar las actividades cuando están agrupadas

De manera adicional, también se implementa la función `selectionIcon`, anteriormente mencionada, la cual permite incluir un ícono personalizado al lado de cada ítem.



Selección marker y listado actividades

Es necesario comentar que la gran mayoría de problemas con el *cluster* vienen de la sobrecarga de datos, es por ello que se implementó al inicio del método *observe* una limpieza completa de todas las listas para evitar que después de cada acceso al fragmento, se dupliquen los elementos.

Por último, se realizaron dos correcciones para permitir, aún con el *clustering*, el enlace con las actividades desde otros fragmentos:

- Se han mantenido la creación de los *markers* de la manera antigua, aunque estos están invisibles. Esto ha permitido crear *markers* como tal, y evitar la modificación del código del *InfoActivityFragment* que recibe como argumento un *marker*. También, de esta forma se mantiene la forma original de crear las listas *markers* y *madridMap*, necesarios en diferentes partes del código. Adicionalmente esto ha permitido no tener que crear otro *windowAdapter* y mantener el que ya existía.

```

let { coordinates ->
    val item = MyItem(coordinates, "${it.id} ${it.title}", "${it.time} ${it.dstart}")
    items.add(item)
    madridMap[item.getNombre()] = it.id
    val markerOptions = MarkerOptions().position(coordinates).title("${it.id} ${it.title}").visible(false)
    val marker = map.addMarker(markerOptions)
    markers[marker.title] = marker
    madridMap[marker.title] = it.id
}

```

Creación original markers

- Se ha creado una *locatorsListFav*, que recoge todos los *locators* que se creen en *observe*; sin embargo, esta nunca se va a modificar. Esto quiere decir que siempre tendrá todos los *locators* inclusive cuando se selecciona un *chip*, de esta forma se solucionó un error que había, en el cual si el usuario dejaba un *chip* seleccionado y quería elegir una actividad que no coincidía con esa categoría desde el fragmento de favoritos, no era posible y rompía.
- ***ApplyFilter()***: de igual manera que en el método *observe()*, es necesario sustituir la creación de *markers* por la creación de los ítems cuya categoría coincide con la seleccionada. Posteriormente, añadir estos ítems a una lista, y la lista por completo en el *ClusterManager*.

```

private fun applyFilter(filteredLocators: List<LocatorView>) {
    map.clear()
    clusterManager.clearItems()
    val items = mutableListOf<MyItem>()
    filteredLocators.mapNotNull { locator ->
        locator.location.latitude?.let { lat ->
            locator.location.longitude?.let { lng ->
                LatIng(lat, lng)
            }
        }?.let { coordinates ->
            val item = MyItem(coordinates, "${locator.id} ${locator.title}", "${locator.time} ${locator.dstart}")
            items.add(item)
        }
    }
    locatorList = filteredLocators
    map.setInfoWindowAdapter(
        CustomInfoWindowAdapter(
            LayoutInflater.from(activity),
            locatorList
        )
    )
    clusterManager.addItems(items)
    clusterManager.cluster()
}

```

Código para aplicar filtros

Almacenamiento actividades favoritas

En un inicio, el equipo de desarrolladores, para poder guardar las actividades que el usuario seleccionaba como favoritas manejó la idea de realizar una base de datos interna en la aplicación, para guardar dichas actividades. Sin embargo, este tipo de implementación generaba problemas en su ejecución que afectaban al correcto funcionamiento de la aplicación y a su rendimiento. Por estos motivos, se optó por hacerlo en una base de datos externa empleando la herramienta *Firebase*, que ha permitido mejorar el rendimiento de la aplicación y dotarla de mayor seguridad ya que, únicamente se necesita una conexión a la base de datos y un objeto temporal para poder guardar la información.

6. VALIDACIÓN DE LA SOLUCIÓN

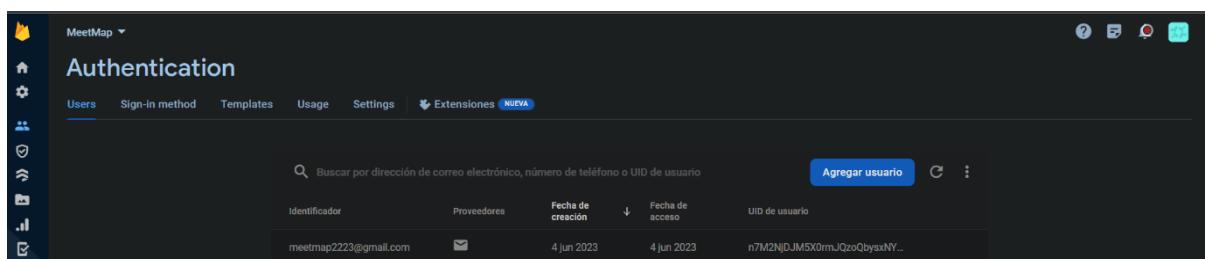
6.1 Documentación descriptiva

Para validar la solución propuesta en el desarrollo de esta aplicación se ha sometido la misma a diferentes pruebas de testeо.

Una de ellas ha sido hacer un seguimiento de las diferentes funcionalidades que hacen uso de la base de datos de *Firebase*.

Para ello se ha creado un correo ficticio que ha permitido al equipo de desarrolladores llevar un control de las actualizaciones de la base de datos a medida que se han ido testeando las diferentes funcionalidades.

En primer lugar, el correo de prueba: meetmap2223@gmail.com se ha registrado en la aplicación como un usuario más:



The screenshot shows the Firebase Authentication interface. On the left is a sidebar with icons for MeetMap, Authentication, and other services. The main area is titled 'Authentication' and has tabs for 'Users', 'Sign-in method', 'Templates', 'Usage', 'Settings', and 'Extensões NUEVA'. A search bar at the top right says 'Buscar por dirección de correo electrónico, número de teléfono o UID de usuario'. Below it is a table with columns: 'Identificador', 'Proveedores', 'Fecha de creación', 'Fecha de acceso', and 'UID de usuario'. A single row is shown for the email 'meetmap2223@gmail.com', with a mail icon in the 'Proveedores' column and the date '4 Jun 2023' in both 'Fecha de creación' and 'Fecha de acceso' columns. The 'UID de usuario' column shows a long string of characters. At the bottom right of the table are buttons for 'Agregar usuario' and a three-dot menu. The entire interface is dark-themed.

Autenticación usuario Firebase

Posteriormente, el usuario accede a su vista de perfil para actualizar sus datos:

Vista del panel Compilador de consultas

meetmap-1856b users Agregar documento

+ Iniciar colección + Agregar campo

users

meetmap2223@gmail.com

description: ""
email: "meetmap2223@gmail.com"
img: "https://firebasestorage.googleapis.com/v0/b/meetmap-1856b.appspot.com/o/img%2Fpredeterminado.png?alt=media&token=3bda85a1-f7d2-4bbb-86f1-c96cad24bebd"
name: ""
phone: ""
surname: ""

Datos del usuario predefinidos con el registro

Vista del panel Compilador de consultas

meetmap-1856b users Agregar documento

+ Iniciar colección + Agregar campo

users

meetmap2223@gmail.com

description: "Prueba para el proyecto de MeetMap con actualización de datos"
email: "meetmap2223@gmail.com"
img: "https://firebasestorage.googleapis.com/v0/b/meetmap-1856b.appspot.com/o/img%2Fpredeterminado.png?alt=media&token=3bda85a1-f7d2-4bbb-86f1-c96cad24bebd"
name: "Proyecto"
phone: "666222333"
surname: "MeetMap"

Datos del usuario después de actualizar información del perfil

Storage

gs://meetmap-1856b.appspot.com > img

Nombre	Tamaño	Tipo	Modificación más reciente
meetmap2223@gmail.com	115.975 bytes	image/jpeg	4 jun 2023, 10:14:47

Nombre: meetmap2223@gmail.com
Tamaño: 115.975 bytes
Tipo: image/jpeg
Creado: 4 jun 2023, 10:14:47
Actualizado: 4 jun 2023, 10:14:47
Ubicación del archivo
Otros metadatos

Imágenes empleadas por el usuario en su perfil

El usuario sigue interactuando con la aplicación y selecciona una actividad para suscribirse y marcarla como favoritos.

The screenshot shows the Firebase Realtime Database interface. A specific node under the 'users' collection is expanded, revealing a 'Favourites' node. Inside 'Favourites', there is a child node with the key '11998343'. This node contains several properties: 'Date: "2023-06-23"', 'ID: 11998343', 'Place: "Centro Comunitario Casino de la Reina"', 'Time: "21:00"', and 'Title: "GROTTÉ (Avoiding Loneliness) - Centro"'. The entire database structure is visible on the left side of the screen.

Actividad guardada como favorita por el usuario

The screenshot shows the Firebase Realtime Database interface. A specific node under the 'activities' collection is expanded, revealing a 'Subscribers' node. Inside 'Subscribers', there are two entries: 'almu1234: "almu1234@email.com"' and 'meetmap2223: "meetmap2223@gmail.com"'. The entire database structure is visible on the left side of the screen.

Suscripción del usuario a la actividad anterior

Una vez suscrito, el usuario comienza un chat con otro usuario de la aplicación.

The screenshot shows the Google Cloud Firestore interface. A new document has been created in the 'chats' collection, with the ID '437f5cd9-94cd-4d28-aeac-a57c1ce0dce5'. The document contains fields: 'delete: false', 'id: "437f5cd9-94cd-4d28-aeac-a57c1ce0dce5"', 'name: "Chat con almu1234@email.com"', and a 'users' array containing two entries: 'meetmap2223@gmail.com' and 'almu1234@email.com'. The left sidebar shows other collections like 'meetmap-1856b', 'chats', and 'users'.

Inicio chat con otro usuario suscrito a la actividad

Por otra parte, cuando la aplicación ha pasado a la fase de producción y se ha subido a la *Play Store* de *Google* también se han obtenido diferentes errores que se comentarán a continuación.

Google permite, a prácticamente cualquier usuario, subir sus aplicaciones móviles a la *Play Store* para ello, se deben llenar ciertos formularios y fichas descriptivas sobre la aplicación. Una vez subida, *Google* analiza la aplicación y, si no incluye ningún aspecto peligroso para el resto de los usuarios, permite añadirla a su tienda.

Una vez añadida, desde la *Google play console* se puede observar multitud de información acerca de la aplicación: estadísticas, reseñas, descargas, errores, etc. Gracias a esto, se pueden solucionar diversos problemas que el equipo de desarrolladores no ha logrado localizar, pero otros usuarios de la aplicación han experimentado en los últimos días al utilizar la app.

6.2 Problemas encontrados y justificación

A la hora de validar las diferentes funcionalidades de la aplicación, cuando se han ido completando, el equipo de desarrolladores y personas de su entorno ha podido testear la app y detectar diversos problemas que imposibilitaban que las pruebas se ajustasen al resultado esperado y que se han ido solucionando a medida que han ido surgiendo.

Después de analizar los resultados de las pruebas de *Firebase* explicadas anteriormente se detectó un error a la hora de actualizar la imagen de perfil en el menú lateral cuando el usuario cambiaba su imagen. Durante toda la parte de desarrollo inicial, una simple imagen que el usuario subía a la base de datos de *Firebase* no se actualizaba de primeras.

Siempre era necesario subir otra foto y, en ese caso, se actualizaba a la foto anterior. Este error estuvo en la parte de desarrollo aproximadamente mes y medio; durante ese tiempo, se descubrió que si la base de datos estaba abierta (la página web de *Firebase*), la imagen si lograba cambiar a tiempo.

Al final, la solución fue obligar de forma interna a guardar la foto por duplicado:

```

binding.btnSave.setOnClickListener {
    saveImg()
    Handler().postDelayed({
        saveImg()
    }, delayMillis: 2000)

    Toast.makeText(requireActivity(), "Updated data", Toast.LENGTH_LONG).show()
    goBack()
}

```

Solución código actualizar imagen perfil

De esta forma, aunque el código no sea lo más optimo, se guardan los cambios dos veces, aunque como se hace de forma interna, el usuario en ningún momento sería consciente de este cambio: ni pausas en la pantalla, ni retrasos, ni errores.

Por otro lado, y haciendo referencia a la herramienta de *Google play console* explicada anteriormente, en la penúltima versión de la aplicación subida a la Play Store se pudieron localizar estos 2 errores:

Error	Tipo
com.ikalne.meetmap.MainActivity\$onCreate\$1.invoke java.lang.NullPointerException	Fallo
com.ikalne.meetmap.fragments.FavouritesFragment.loadFavouriteActs java.lang.NullPointerException	Fallo

Errores Google play console

- El primero de ellos hacía referencia a esta parte del código donde se obtiene el nombre del usuario y la imagen de su perfil:

```

fStore.collection(collectionPath: "users").document(email).get().addOnSuccessListener {
    it: DocumentSnapshot!
    username.setText(it.get("name") as String)
    //binding.mail.setText(email)
    Glide.with(activity: this) RequestManager
        .load(it.get("img") as String) RequestBuilder<Drawable>
        .circleCrop()
        .into(imagenav)
}

```

Código del primer error: datos perfil usuario

Una vez analizado, se llegó a la conclusión que fue por un problema de borrar la base de datos de *Firebase* sobre los usuarios registrados (para reiniciar todo ante el nuevo lanzamiento). Este problema se debió originar al tener algún antiguo usuario su dispositivo con su cuenta abierta, y al borrar la base de datos, no entraba a la ventana *initial* si no que la aplicación intentaba dirigirle directamente al mapa.

Para solucionarlo, el usuario podía simplemente borrar el caché de la aplicación, pero desde el equipo de *Meetmap* se decidió implementar una modificación en el código que consistía en un simple controlador que vigilaría en el mismo *splash* si el usuario estaba registrado o no.

- El segundo error, se refería a esta parte:

```
private fun loadFavouriteActs() {    SoySandyYT, 13/04/2023 19:37 • Changed
    fliArrayList = ArrayList()
    val ref = FirebaseDatabase.getInstance().getReference(path: "users")
    ref.child(firebaseAuth.uid!!).child(pathString: "Favourites")
        .addValueEventListener(object : ValueEventListener {
            override fun onDataChange(snapshot: DataSnapshot) {
                var boolData = false
                var cnt = 0
                val data = ArrayList<FLI>()
                for (ds in snapshot.children) {
                    val aid = "${ds.child(path: "ID").value}"
                    val FLI = FLI()
                    FLI.id = aid
                    data.add(FLI)
                    if(cnt == 0){
                        boolData = true;
                        cnt++;
                    }
                }
            }
        })
}
```

Código del segundo error: verificación favoritos

Este error se pudo generar por el mismo motivo, usuarios registrados anteriormente y una base de datos eliminada. Por tanto, una vez solucionado el anterior problema, este no se ha vuelto a producir. De todas formas, se decidió incluir un controlador en el fragmento para evitar posibles errores futuros.

Si ampliamos la lista de errores, solo se pueden observar hasta 60 días antes, de esta forma, aparece un error que se producía en una versión más antigua:

com.ikalne.meetmap.api.services.BaseRequestFunctionKt.getUrlObject java.net.UnknownHostException	Fallo	9 (1.3)
Error petición API de Madrid		

Tras analizarse el mensaje de error, se descubrió que falló alguna petición a la API de Madrid, que desde el equipo se recogió para su posterior análisis.

Para solucionar un error ajeno a la aplicación, se añadió el control explicado anteriormente que no deja entrar al usuario si no hay internet, adicionalmente, si la API de Madrid da error. De esta forma, se avisa al usuario que algo no ha ido como se esperaba.

Otro de los problemas que se ha encontrado tras una fase de pruebas es que, la función que permitía elegir una actividad cuando estaban agrupadas en un *cluster* funcionaba incorrectamente. Al hacer *click* sobre el primer elemento, accedía al segundo; el segundo te llevaba al tercero, y así sucesivamente.

Tras un análisis de la función se ha descubierto que el *index* de una lista de programación va siempre del elemento 0 hasta el tamaño de la lista menos uno. Sin embargo, la lista que generan los *cluster* (o al menos al seleccionar uno en la clase de menú *dialogBuilder*), empieza en el elemento 1, generando ese fallo.

La solución de este error se realizó de manera sencilla:

```
dialogBuilder.setAdapter(adapter) { dialog, which ->
    val selectedItem = cluster.items.elementAtOrNull(index: which-1)
    if (selectedItem != null) {
        val marker = MapFragment.markers[selectedItem.getNombre()]
        if (marker != null) {
            val infoFragment = InfoActivityFragment()
```

Código error al seleccionar ítem lista actividades con cluster

Como se puede observar, el *selectedItem* recoge el ítem pulsado por el usuario menos uno, logrando reducir el índice de este y así conectar ambas listas.

7. FUENTES

Adobe. (s.f.). Color Contrast Analyzer. <https://color.adobe.com/es/create/color-contrast-analyzer>

Arias del Prado, J. (2020, 4 de marzo). Diseño Centrado en el Usuario (DCU). Todas las claves del proceso. Blog.UXABLES. <http://www.uxables.com/diseno-ux-ui/diseno-centrado-en-el-usuario-dcu-todas-las-claves-del-proceso/>

Ayuntamiento de Madrid. (s.f.). Actividades Culturales y de Ocio Municipal en los próximos 100 días.

<https://datos.madrid.es/portal/site/eqob/menuitem.214413fe61bdd68a53318ba0a8a409a0/?vgnextoid=b07e0f7c5ff9e510VgnVCM1000008a4a900aRCRD&vgnextchannel=b07e0f7c5ff9e510VgnVCM1000008a4a900aRCRD&vgnextfmt=default#/Centros32de32D237a>

Cunningham, W. (2001) Principios del Manifiesto Ágil. agilemanifesto.org
<https://agilemanifesto.org/iso/es/principles.html>

Flaticon. (s.f.). <https://www.flaticon.es/>

G.Portalatín, B. (2022, 22 de marzo). Amor en Tinder (y otras apps de citas): cómo la pandemia ha condicionado la forma de ligar de los 'millenials'. laSexta.
https://www.lasexta.com/bienestar/sexualidad/amor-tinder-otras-apps-citas-como-pandemia-condicionado-forma-ligar-millenials_20220322623873cbf6355200015c9dc4.html

Gonzalo, M. (2021, 17 de marzo). La pandemia que nos volcó a las redes. Newtral.
<https://www.newtral.es/pandemia-redes-sociales-digitalizacion-covid-19/20210317/>

Google. (s.f.). Google Fonts. <https://fonts.google.com/>

Google. (s.f.). Material Studies. <https://m2.material.io/design/material-studies/about-our-material-studies.html>

Pixabay. (s.f.). Persona. <https://pixabay.com/es/images/search/persona/>

Redacción (2020, 4 de junio). Así han evolucionado las aplicaciones de citas online durante el confinamiento. ReasonWhy. <https://www.reasonwhy.es/actualidad/aplicaciones-citas-online-cambios-confinamiento-coronavirus>

Terrón Barroso, A. (2012, 17 de octubre). El parseo de datos en internet: nuevas oportunidades comerciales al filo de la legalidad. INESEM BUSINESS SCHOOL
<https://www.inesem.es/revistadigital/gestion-empresarial/el-parseo-de-datos-en-internet-nuevas-oportunidades-comerciales-al-filo-de-la-legalidad/#:~:text=%C2%BFQu%C3%A9es%20el%20parseo%20o,extraer%20informaci%C3%B3n%20significativa%20de%20ella>

Unsplash. (2018, 15 de octubre). Hombre de pie cerca de la pared blanca.
<https://unsplash.com/es/fotos/d1UPkiFd04A>

Unsplash. (2017, 14 de mayo). Fotografía en primer plano de mujer sonriendo.
https://unsplash.com/es/fotos/mEZ3PoFGs_k

Walkiria Apps. (s.f.). Shared Preferences Android Studio.Curso Android Gratis.
<https://cursoandroidgratis.com.es/shared-preferences/#:~:text=Las%20preferencias%20compartidas%20Android%2C%20o,dato%20es%20un%20tipo%20simple.>