Facultad de Informática

Grado de Ingeniería Informática

Trabajo Fin de Grado

Ingeniería del Software

EkitApp: conectando alumnos en una red social con Flutter y Firebase

Julen Miner Goñi

Junio 2019

Director: Dr. José Miguel Blanco Arbe

Agradecimientos

Antes de comenzar, me gustaría poder agradecer el apoyo que he recibido a las personas que han seguido este camino conmigo y han hecho que este proyecto se haya llevado a cabo.

Primero, agradecer a Elsa haber comenzado la carrera conmigo y haber estado a mi lado en todo momento durante estos cuatro años, desde pequeños proyectos en clase hasta la junta directiva de Magna SIS. Gracias por el apoyo que he recibido y el que, estoy seguro, seguiré recibiendo. Sin ti este proyecto no habría existido y agradezco el poder haberlo compartido contigo. No se me ocurre mejor manera de terminar la carrera que realizando el último proyecto en conjunto.

Me gustaría también agradecer a Enya todo el apoyo que me ha dado. Me alegro de haberla conocido y por el apoyo en los últimos dos años. También agradezco enormemente el apoyo que me ha dado durante la realización de este proyecto, ya que sin él hubiera sido muchísimo más difícil. Nuestros proyectos llevan una parte del otro.

Quiero dar las gracias también a mi director, José Miguel, por haber creído en nuestra idea desde el principio, por habernos ayudado a darle forma y por haberme guiado durante el desarrollo del proyecto.

Quiero agradecer a mi madre haberme enseñado un artículo que leyó en el periódico cuando yo estaba en cuarto de la ESO. Gracias a ese momento he llegado a este proyecto. También quiero darle las gracias por todo lo que ha tenido que trabajar durante todos estos años para hacer posible que yo pudiera estudiar.

Por último, quiero agradecer todo el apoyo y ayuda que he recibido por parte de mis amigas, que me han ayudado a seguir adelante en todo momento, sobre todo a Alasne y a Chiara por toda la ayuda y apoyo que me han dado durante los últimos años. También a la persona que en poco tiempo cambió mi vida completamente y me animó a realizar muchos cambios positivos.

Índice

1. Introducción 6

2. Antecedentes del proyecto 7
    2.1 Interés por el área 7
    2.2 Vocación de desarrollar una iniciativa empresarial 8
    2.3 Vocación de colaborar 8
    2.4 Antecedentes tecnológicos 8

3. Objetivos 10
    3.1 Objetivos generales 10
    3.2 Objetivos específicos 10
    3.3 Descripción del alcance 11
    3.4 Requisitos mínimos 13
    3.5 Exclusiones 13

4. Análisis y diseño 14
    4.1 Arquitectura de la aplicación 14
        4.1.1 Arquitectura general de la aplicación 14
        4.1.2 Arquitectura específica de Ekitaldi 15
    4.2 Interacción con el usuario 16
    4.3 Modelo de datos 25
        4.3.1 Base de datos 25
        4.3.2 Sistema de ficheros de la aplicación 29
    4.4 Casos de uso 29
        4.4.1 Usuario anónimo 29
        4.4.2 Usuario registrado 31

5. Tecnologías 36
    5.1 Flutter 36
    5.2 Firebase 41
    5.4 Algolia 43

6. Implementación 44
    6.1 Aplicación Flutter 44
        6.1.1 Implementación de la GUI 50
        6.1.2 Implementación de la lógica de negocio 55
        6.1.3 Implementación de la persistencia 56
    6.2 Funciones Cloud 56

7. Pruebas 58
    7.1 Pruebas con usuarios 58
    7.2 Pruebas de la aplicación 59
    7.3 Pruebas de integración Smart-Ekitaldi 65

8. Gestión del proyecto 67
8.1 Gestión del alcance 67
8.2 Gestión de la colaboración 67
8.3 Dedicaciones 71

9. Conclusiones 72
9.1 Tecnologías y metodologías utilizadas 72
9.2 Problema abordado y solución dada 73
9.3 Respecto al grado 73
9.4 Futuro de la aplicación 74

10. Fuentes 75

11. Anexos 76

Índice de figuras

1.	Sello de recepción de la beca	7
2.	Estructura de Descomposición de Trabajo	12
3.	Representación gráfica de la arquitectura general	14
4.	Representación gráfica de la arquitectura específica	15
5.	Diseño de tarjeta sin expandir	18
6.	Diseño de tarjeta expandida	18
7.	Menú contextual, izquierda iOS, derecha Android	18
8.	Sección de comentarios	19
9.	Pantalla de inicio de sesión	19
10.	Barras superior e inferior	20
11.	Menú lateral de opciones	20
12.	Modo día a la izquierda, modo noche a la derecha	21
13.	Filtros que aparecen sobre los eventos	21
14.	Pantalla de inicio	22
15.	Pantalla de búsqueda	22
16.	Pantalla de guardados	22
17.	Pantalla de actividad	22
18.	Proceso de actualización *pull-to-refresh*	23
19.	Perfil de usuario	23
20.	Crear evento	24
21.	A la izquierda barras superiores de iOS, a la derecha las de Android	24
22.	Casos de uso del usuario	29
23.	Diagrama de flujo de iniciar sesión	30
24.	Casos de uso de un usuario registrado	32
25.	Logo de Flutter	36
26.	Esquema de aplicaciones nativas	37
27.	Esquema de aplicaciones híbridas	37
28.	Esquema de las aplicaciones web	38
29.	Esquema de las aplicaciones Flutter	38
30.	Actualización en la programación web	39
31.	Actualización en la programación híbrida	39
32.	Actualización en Flutter	40
33.	Logo de Firebase	41
34.	Logo de Algolia	43
35.	Resultado de ejecutar flutter doctor	44
36.	Directorios de la aplicación	45
37.	Estructura de ficheros de un proyecto de Cloud Functions	56
38.	Esquema del sistema de información conjunto	70
39.	Gráfico de *commits* por semanas	71
40.	Diagrama de la base de datos	76
41.	Diagrama de flujo de crear evento	77
42.	Pantalla de creación de proyecto en Firebase	92
43.	Banner en la pantalla de inicio de Firebase	93
44.	Estructura de ficheros en XCode	94

1. Introducción

En este capítulo se describe brevemente la idea general del proyecto, así como el desarrollo que ha sido llevado a cabo y cómo se han utilizado las novedosas tecnologías Flutter y Firebase. También se explican los apartados que contiene esta memoria y se introducen brevemente.

La idea del proyecto nace de un problema detectado en las universidades respecto a los anuncios sobre eventos (charlas, cursos...) que se realizan en el entorno universitario. El problema reside en que la mayoría de anuncios enviados por correo son eliminados al instante nada más llegar al buzón de entrada de los alumnos. Por ello, se ha querido dar una plataforma tanto al personal de la universidad como a los propios alumnos para que puedan hacer publicidad de sus eventos.

Adicionalmente, se colabora con Elsa Scola (TFG-Elsa) para desarrollar el proyecto. El producto de este trabajo de fin de grado es la aplicación llamada EkitApp que permite el acceso a la red social, mientras que TFG-Elsa provee funciones para detectar SPAM analizando texto y los datos de los usuarios. Dentro de los capítulos **Antecedentes del proyecto** y **Objetivos** se puede leer más sobre cómo fue concebida la idea y cuáles han sido los objetivos y requisitos de este proyecto.

Para desarrollar esta aplicación se han utilizado dos tecnologías clave: Flutter y Firebase. Flutter es una herramienta presentada por Google en 2015, pero su primera versión estable no fue lanzada hasta el año 2018. Permite utilizar un mismo código base para generar la misma aplicación para diferentes plataformas: Android, iOS y Fuchsia, aunque prometen la compatibilidad con navegadores web y aplicaciones de escritorio en un futuro cercano (sin fecha confirmada). Su novedosa forma de mostrar los elementos en los dispositivos hace que el rendimiento ofrecido sea mayor, lo que diferencia a Flutter del resto de *frameworks* de desarrollo de aplicaciones híbridas. Además, la inclusión de la programación declarativa hace que sea menos propensa a errores inesperados, obligando a establecer los estados por los que va a pasar la aplicación a la hora de desarrollar.

Por su parte, Firebase ofrece una gran cantidad de herramientas y servicios que permiten implementar las tecnologías más novedosas del mercado: desde bases de datos noSQL con gran capacidad de escalar hasta herramientas de aprendizaje automático. Todas estas herramientas son sencillas de implementar gracias a su documentación y, al estar todas dentro de una misma plataforma, crear relaciones entre ellas no es un proceso costoso. Además, como tanto Firebase como Flutter son propiedad de Google, la integración de los servicios de Firebase dentro de la aplicación desarrollada con Flutter requiere de poca implementación extra. Sobre las tecnologías y el proceso de desarrollo de la aplicación se puede conocer más en los capítulos de **Análisis y diseño**, **Tecnologías**, **Implementación** y **Pruebas**.

Para finalizar, se presenta un capítulo en el que se detalla la gestión que se ha llevado a cabo durante el proceso de desarrollo de este proyecto, **Gestión del proyecto**, además de las conclusiones del mismo en el capítulo **Conclusiones**.

2. Antecedentes del proyecto

Durante este capítulo se presentan los antecedentes al proyecto. Se cubren temas como el interés por el área en el que se ha trabajado, la vocación de desarrollar una iniciativa empresarial y de colaborar, así como los antecedentes tecnológicos.

2.1 Interés por el área

El desarrollo de aplicaciones móviles es un área que, durante el grado de Ingeniería Informática, se introduce muy brevemente. No obstante, ha captado mi interés. Comencé a indagar más en el tema tras acabar el primer año del grado, durante el verano. Mi compañera Elsa Scola (en lo sucesivo, Elsa) y yo completamos el curso de Desarrollo Android para Principiantes[1], disponible gratuitamente en Udacity. Esto nos llevó a realizar una pequeña adaptación de un juego que habíamos implementado en clase durante el curso anterior, y la publicamos en la tienda de aplicaciones Google Play Store[2].

En otoño de 2017, al comienzo del primer cuatrimestre del tercer curso del Grado en Ingeniería Informática solicité la beca "Google Developer Scholarship"[3] de Udacity (figura 1) para realizar un curso avanzado de producción de aplicaciones Android, y me fue concedida. Comencé el curso el 6 de noviembre del mismo año y tuvo una duración de aproximadamente un mes y medio. Al realizar estos dos cursos supe que éste área me resultaba de gran interés.

Figura 1: Sello de recepción de la beca

[1] Enlace al curso: https://eu.udacity.com/course/android-development-for-beginners--ud837

[2] La aplicación no se encuentra disponible, pero se puede leer el blog en el que publicamos sobre "The Word Game": https://ellenco.tumblr.com/

[3] Enlace a las becas: https://www.udacity.com/google-scholarships

2.2 Vocación de desarrollar una iniciativa empresarial

Durante el proceso de publicación de la aplicación, mencionada en el punto anterior, en la tienda de aplicaciones Google Play Store, tuvimos que pasar por algunas tareas de gestión empresarial, como la decisión de un nombre de empresa que apareciera en la tienda junto a dicha aplicación. Adicionalmente, la participación en la junta directiva, en la que he ocupado el cargo de tesorero y socio de la junior empresa de la facultad, Magna SIS[4], me ha hecho aprender mucho sobre la gestión empresarial durante estos dos últimos años. Esto ha derivado en un gran interés por desarrollar una iniciativa empresarial basada en un proyecto. Esta idea de proyecto ha sido el fundamento sobre el que se ha basado mi trabajo de fin de grado, ya que mi proyecto ha servido como prueba de concepto de lo que en el futuro pudiera ser una aplicación más completa y con un enfoque más amplio de usuarios.

2.3 Vocación de colaborar

Tanto la iniciativa empresarial como la idea del proyecto no podrían haber nacido sin la colaboración, pilar fundamental de este proyecto. En los últimos años he podido trabajar junto con Elsa y gracias a esto han nacido tanto mi proyecto como el suyo, que han sido realizados en colaboración.

Durante el transcurso del grado de Ingeniería Informática, he colaborado con Elsa en numerosos proyectos, tanto de clase como de los que llegaban a Magna SIS, así como los que hemos realizado en nuestro tiempo libre. El elemento común en todos los proyectos ha sido la fluidez en la comunicación y la facilidad del trabajo en equipo. Por lo tanto, la mejor manera para ambos de terminar este camino ha sido colaborar en este Trabajo de Fin de Grado.

Además, al realizar una colaboración entre dos ramas de la Ingeniería Informática diferentes (Ingeniería del Software y Computación), se aportan diferentes puntos de vista al proyecto.

2.4 Antecedentes tecnológicos

Durante la concepción del proyecto, cuando estaba pensando en qué herramientas utilizar, decidí que quería emplear una que no hubiera usado hasta el momento. De esa manera podría aprender a utilizarla, además de hacer un estudio de usabilidad y de posible implantación en los estudios. Al principio iba a hacer uso de Google App Maker[5] para desarrollar el proyecto, una herramienta novedosa que lanzó Google para realizar desarrollos ágiles para soluciones empresariales. Esta herramienta fue probada por un alumno en prácticas en Magna SIS y, aunque ésta tenía muchas ventajas y era muy potente, decidí utilizar otra diferente por sus desventajas. Las mayores

[4] Enlace a la página web de Magna SIS: https://magnasis.com/

[5] Información sobre App Maker: https://gsuite.google.es/intl/es/products/app-maker/

desventajas que tenía eran la necesidad de una suscripción de pago para trabajar en dicha herramienta y la imposibilidad (en el momento de la decisión) de poder hacer pública la aplicación.

Por lo tanto, decidí utilizar una herramienta que me permitiera construir aplicaciones instalables en los dispositivos móviles Android e iOS, por lo que quedaban descartados los desarrollos nativos de cada plataforma. Entre todas las opciones disponibles, decidí utilizar Flutter porque era una herramienta que había sido lanzada recientemente (presentada en 2015, la versión de prueba fue lanzada en mayo de 2017 y la primera versión estable en diciembre de 2018) y el hecho de que se crearan aplicaciones nativas para las dos plataformas a partir del mismo código me resultó muy interesante. Además, permitía una gran integración con Firebase, herramienta que tanto mi compañera Elsa como yo hemos utilizado como proveedor de servicios, como el de la base de datos. Gracias a esto, la integración entre nuestros dos proyectos ha sido más sencilla, ya que los dos trabajábamos con la misma herramienta. Flutter, además, cuenta con otras ventajas, de las que hablo en el capítulo 5.

3. Objetivos

En este capítulo se presentan los objetivos generales y específicos del proyecto, así como de la descripción del alcance, incluyendo la Estructura de Descomposición del Trabajo, los requisitos mínimos y las exclusiones del mismo.

3.1 Objetivos generales

Actualmente, la comunicación de eventos en el entorno universitario se realiza a través de correos electrónicos que son ignorados la mayoría de las veces por gran parte de los estudiantes. El objetivo de este proyecto es crear una red social completa y segura, que sirva como comunidad en la que estudiantes y profesores puedan hacer públicas dichas actividades. Se crea así un núcleo que propicia la iniciativa del alumnado y facilita la comunicación de eventos oficiales a las instituciones universitarias.

En el mercado toda red social ofrece de base una aplicación y un sistema inteligente que lo respalda, por lo que la colaboración de miembros de distintas áreas en este tipo de proyectos es fundamental. Para ello, es imprescindible asentar unas bases sólidas de colaboración entre las dos partes de los Trabajos de Fin de Grado para que la cohesión entre ellos tenga sentido.

Consecuentemente, para realizar este proyecto es muy importante la alta integración de todos los servicios que ofrece Google, desde el desarrollo del software para implementar una aplicación hasta los servicios de Machine Learning para dotar al producto de Artificial Intelligence.

Adicionalmente, se busca comenzar un producto que en el futuro pueda servir como base para comenzar un emprendimiento, buscando así un proyecto que sea útil a la vez que enriquecedor y formativo.

3.2 Objetivos específicos

Específicamente, y para dar acceso a los servicios de la red social que va a dar respuesta al problema propuesto, se va a desarrollar una aplicación móvil. Ésta tiene que ser compatible, por lo menos, con los sistemas operativos móviles iOS y Android, cubriendo así el 97,3% del mercado móvil [6] (hay que tener en cuenta que las versiones obsoletas de estos sistemas operativos se incluyen). Por lo tanto, se ha optado por utilizar Flutter, tecnología novedosa presentada por Google, de código abierto, que ofrece la posibilidad de desarrollar una aplicación nativa para iOS y Android. Respecto a Flutter, se definen los siguientes objetivos:

[6] Cuota de mercado de los SO móviles: https://www.macworld.co.uk/feature/iphone/iphone-vs-android-market-share-3691861/

- Aprendizaje de uso de la herramienta.
- Desarrollo de la aplicación que dará acceso a la red social.
- Estudio sobre el uso que se le puede dar a la herramienta.

Por otro lado, se acuerda con TFG-Elsa la utilización de Firebase como *back-end* para dar soporte a la red social. Los objetivos respecto a esta herramienta son los siguientes:

- Diseñar y desarrollar una base de datos NoSQL.
- Crear y utilizar funciones cloud alojadas en Firebase.

3.3 Descripción del alcance

Uno de los primeros pasos en la realización del TFG es la definición del alcance. Es decir, indicar qué funcionalidades deberá tener el proyecto para cumplir los objetivos y los paquetes de trabajo en los que se va a realizar.

Los paquetes de trabajo, soportados por la Estructura de Descomposición de Trabajo (EDT) de la figura 2, y sus descripciones son los siguientes:

1. **Gestión**: esta rama se descomprime en tres paquetes de trabajo: *Planificación*, *Seguimiento y Control* e *Integración Smart-Ekitaldi*. Dentro de *Planificación* se define el tiempo en el que se va a desarrollar el proyecto, los requisitos que va a tener, etc. En *Seguimiento y Control* se lleva la cuenta de las horas dedicadas al proyecto, así como a qué actividad y paquete han sido dedicadas. Por último, la gestión del mayor riesgo de este proyecto es la Integración entre los dos proyectos Smart y Ekitaldi. Esto conlleva continua comunicación con la otra parte interesada, TFG-Elsa, para no realizar trabajo en vano.
2. **Académica**: incluye el desarrollo tanto de la memoria del proyecto como de su defensa ante el tribunal, con los respectivos entregables.
3. **Producto**: agrupa todo el trabajo necesario para poder realizar el trabajo que será consecuencia de este TFG. Se divide en cuatro apartados:
   3.1. **Tecnologías**: aprendizaje y estudio de las herramientas base para este proyecto: Flutter para desarrollar la aplicación multiplataforma y Firebase como *back-end*.
   3.2. **Análisis y diseño**: estudio previo de lo que posteriormente va a ser implementado. Se divide en cuatro apartados: *Arquitectura de la Solución*, *Base de Datos*, *Interacción con el Usuario* y *Casos de Uso*. En *Arquitectura de la Solución* se dará respuesta a cómo van a dividirse las implementaciones. *Base de Datos* tendrá como resultado la estructura que va a seguir la base de datos conjunta con Smart. *Interacción con el Usuario* definirá la interfaz gráfica que va a utilizarse en la app para que el usuario pueda hacer uso de manera intuitiva de la misma. Por último, *Casos de Uso* recogerá los estudios previos a la implementación de cada funcionalidad del producto.
   3.3. **Implementación**: proceso de desarrollo necesario para que un caso de uso funcione ulterior a su análisis y diseño. Esta implementación también puede generar documentación respecto a los pasos seguidos para llegar a terminar el caso de uso.
   3.4. **Pruebas**: para asegurar el correcto funcionamiento del producto, se realizará una serie de pruebas. Éstas son de dos tipos: por un lado pruebas con usuarios que, bajo

una serie de instrucciones, expresarán sus opiniones y experiencias de uso y por otro pruebas de integración de Smart-Ekitaldi que se harán para comprobar el correcto funcionamiento de las partes conjuntas de los proyectos.

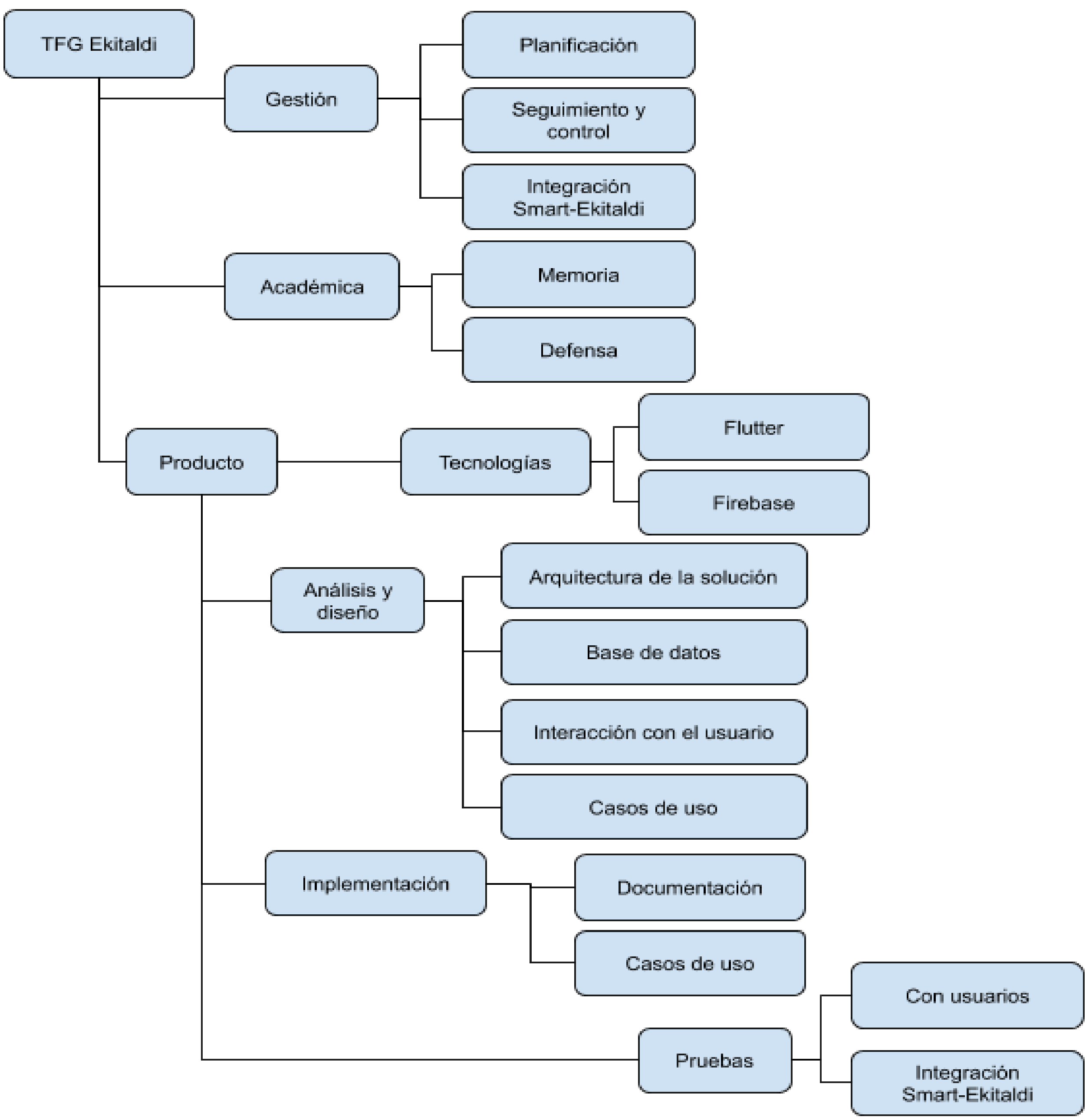

Figura 2: Estructura de Descomposición de Trabajo

3.4 Requisitos mínimos

Los requisitos de la aplicación serán los siguientes:

- Funcionar tanto en iOS como en Android.
- El diseño de la base de datos centralizada y realizada en conjunto.
- Incluir el inicio de sesión y registro por medio de un servicio de autenticación ya existente.
- El código y sus comentarios se realizarán en inglés para facilitar su posterior edición.
- Sistema de información común con TFG-Elsa.
- Para el control de versiones usar Git.
- Ser multi-idioma. Para este TFG, funcionar en castellano e inglés.
- Los usuarios podrán, al menos:
    - Crear publicaciones (eventos).
    - Comentar.
    - Puntuar.
    - Seguir a otros usuarios.

3.5 Exclusiones

Dada la limitación de tiempo y recursos, queda excluido del TFG lo siguiente:

- La publicación de la aplicación en las tiendas de aplicaciones.
- Localización de la aplicación en más idiomas.
- Realización de un análisis e implementación de una política de protección de datos de carácter personal y aviso legal.
- Desarrollo de métodos de analítica de datos.
- El perfil del administrador y sus casos de uso.

4. Análisis y diseño

El análisis y el diseño son dos aspectos muy importantes durante el proyecto. Como resultado de estos procesos, se crea una perspectiva de lo que se va a implementar en el producto y se documenta cómo debe comportarse cada funcionalidad. Además, también se identifica y describe cómo se va a interaccionar con éstas. En este punto, se representan los aspectos y comportamientos del sistema, así como el modelo de datos.

4.1 Arquitectura de la aplicación

La arquitectura general de la aplicación está articulada en torno a una serie de módulos principales que mantienen un conjunto de interacciones. La arquitectura general engloba el software objeto de los dos proyectos (Smart-Ekitaldi). Por cada uno de los dos principales componentes, a su vez, existe una arquitectura propia. En este TFG se presentará la correspondiente a Ekitaldi.

4.1.1 Arquitectura general de la aplicación

La arquitectura, en rasgos generales, de la aplicación conjunta Smart-Ekitaldi queda reflejada en la figura 3:

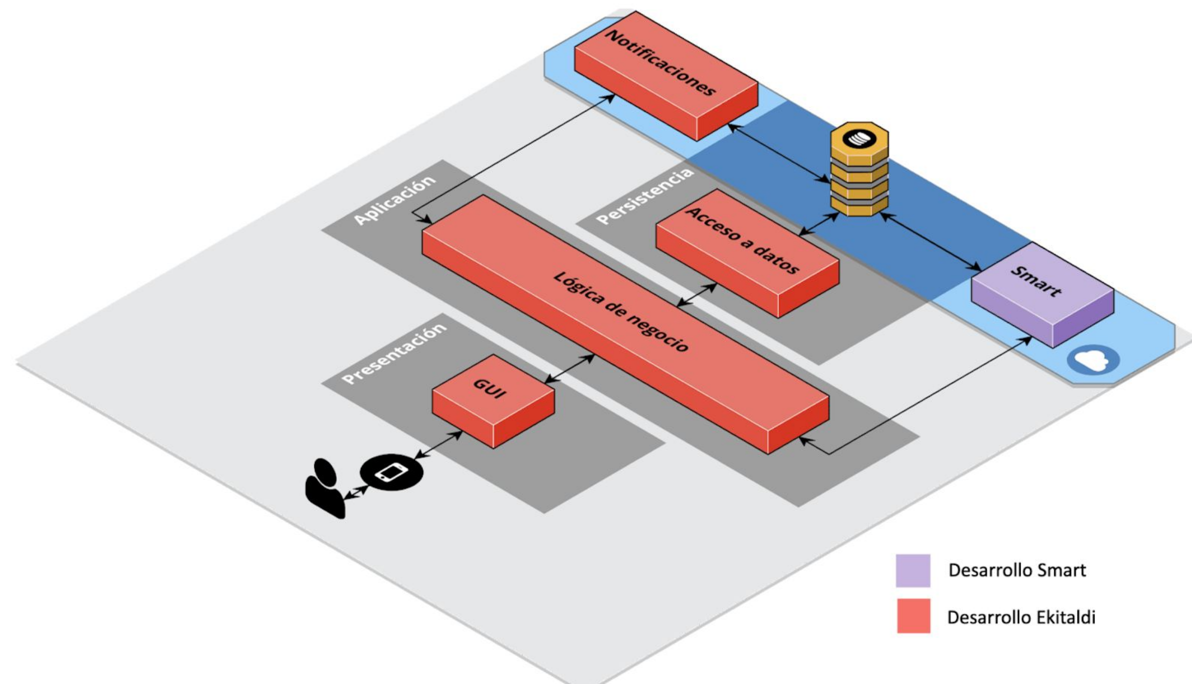

Figura 3: Representación gráfica de la arquitectura general

La primera capa a la que accede el usuario es a la de *Presentación*, que es la que muestra la interfaz gráfica de usuario (*GUI*). Ésta interactúa con la siguiente capa, la de *Aplicación*, que incluye la *Lógica de Negocio*. Desde la *Lógica de Negocio* se invocan los servicios en la nube de las funciones cloud, *Notificaciones* y *Smart*, las cuales acceden a la base de datos cuando lo necesiten. La misma *Lógica de Negocio* interactúa también con la capa de la *Persistencia* mediante el módulo de *Acceso a Datos*. Finalmente, la base de datos, a la que accede también el módulo de *Acceso a Datos*, está almacenada en la nube y es NoSQL. Su estructura, como se puede ver en el punto 4.3.1, forma parte del diseño y desarrollo conjunto de Smart-Ekitaldi.

En la nube, el servicio que da soporte al acceso a las funcionalidades de *Notificaciones* y *Smart* es FIrebase Functions y, a la base de datos, Firestore.

4.1.2 Arquitectura específica de Ekitaldi

Como el proyecto general se divide en dos, uno el objeto de este TFG (Ekitaldi) y otro el correspondiente a la componente Smart (TFG-Elsa), la arquitectura general de la aplicación también se estructura en dos grandes bloques. La arquitectura correspondiente al desarrollo Smart se puede encontrar en el punto 6.1.2 de la memoria del TFG-Elsa. La arquitectura correspondiente al desarrollo Ekitaldi se muestra en la figura 4 (el esquema de la arquitectura de la derecha muestra las tecnologías que se han utilizado para dar respuesta a los problemas del esquema de la izquierda. Estas tecnologías se dividen entre la nube (parte azul) y los dispositivos).

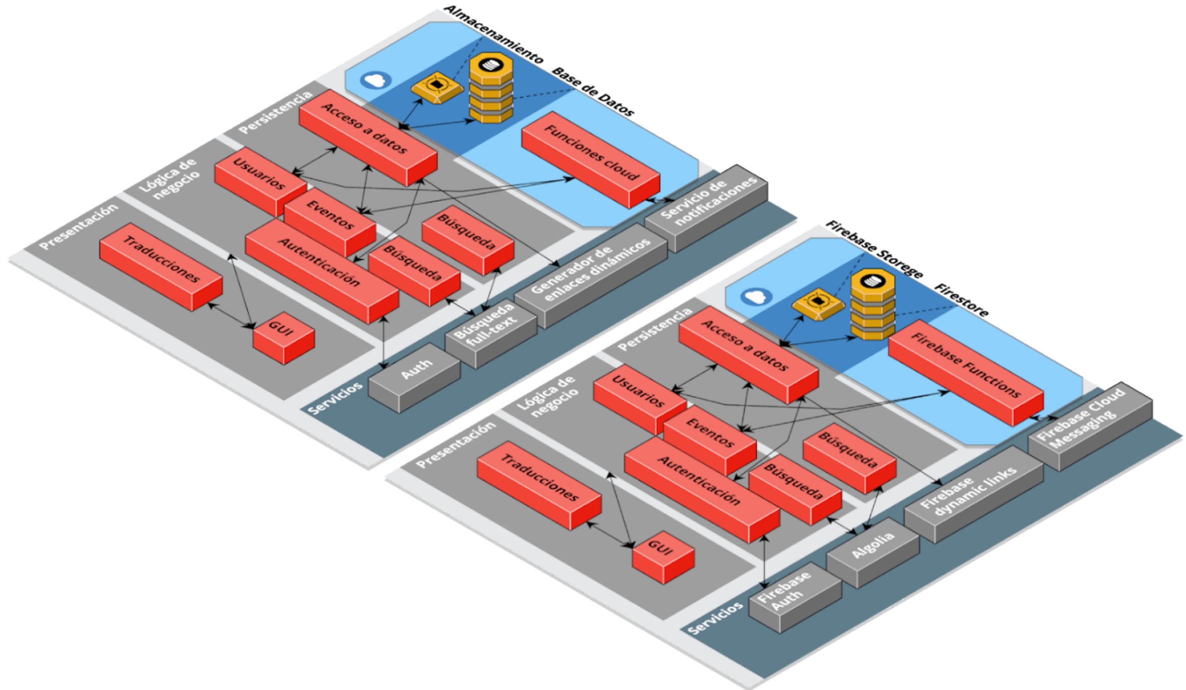

Figura 4: Representación gráfica de la arquitectura específica

Encontramos las tres mismas capas principales que en la arquitectura general: *Presentación*, *Lógica de Negocio* y *Persistencia*. En *Presentación* encontramos el paquete de *Traducciones* y el de la interfaz de usuario, la cual hace peticiones a la primera y también a todos los paquetes de la *Lógica de Negocio*.

En la *Lógica de Negocio* encontramos varios paquetes. El de la *Autenticación* se encarga de realizar el inicio de sesión con los tres métodos posibles y hacer las llamadas al servicio de autenticación ofrecido por terceros. Además, también es el responsable de mantener la sesión iniciada y mantener el identificador del usuario. El paquete *Búsqueda* se encarga de realizar las peticiones al servicio de búsqueda *full-text* para obtener los eventos que concuerden. Los paquetes *Usuarios* y *Eventos* se encargan de todas las peticiones que haya que hacer para actualizar u obtener datos de los mismos y *Usuarios*, además, se encarga de almacenar los datos del usuario que está utilizando la aplicación en ese momento (el que haya iniciado sesión).

En la capa *Persistencia* se encuentran por un lado el paquete *Búsqueda* y *Acceso a Datos* y, por otro lado, en la nube, el servicio de almacenamiento y el de la base de datos. El paquete *Búsqueda* se encarga de mantener actualizados los datos del servicio de búsqueda *full-text*, ya que este servicio mantiene una copia de la base de datos original, aunque guarda menos datos. El paquete *Acceso a Datos* se encarga de hacer las llamadas pertinentes a la base de datos para mantenerla actualizada en todo momento, así como las necesarias al servicio que genera los enlaces dinámicos cada vez que se crea o edita un evento (estos enlaces se utilizan para poder compartir los eventos mediante un enlace). Por último, en la nube se encuentran la base de datos NoSQL y el servicio de almacenamiento, que se utiliza para almacenar las imágenes tanto de usuarios como de eventos.

Finalmente, también en la nube (cuadro azul en la figura 4), se encuentran las funciones alojadas a las que se puede llamar mediante un enlace. En este paquete se alojan las funciones que sirven para enviar las notificaciones a los usuarios, así como todas las funciones de la parte Smart. Asimismo, esta función llama al servicio de notificaciones para enviarlas a cada dispositivo.

4.2 Interacción con el usuario

La interacción con el usuario es una parte fundamental a tener en cuenta, por lo que la interfaz gráfica de usuario debe ser atractiva e intuitiva. Para ello hay que utilizar un método de visualización que separe las publicaciones de manera clara y visual.

Las tarjetas son habitualmente utilizadas para diferenciar el contenido dentro de una aplicación. Google las introdujo y popularizó al presentar en 2014 sus patrones de diseño de interfaces de usuario Material Design, en los que se busca la organización de la información. En las nuevas actualizaciones presentadas por Google (Material Design 2.0, mayo 2018) podemos ver cómo siguen actualizando y utilizando las tarjetas en la mayoría de sus aplicaciones[7], además de la información de diseño de tarjetas que podemos encontrar en la documentación oficial[8]. Un artículo escrito por Nick Babich, desarrollador de software y especialista en interfaces gráficas de usuario, explica en su blog online[9] cómo se pueden utilizar las tarjetas para mejorar la experiencia de usuario dentro de nuestra aplicación.

A la hora de organizar todas las funcionalidades dentro de la aplicación se utilizan dos métodos: barra de navegación inferior y menú lateral. La barra de navegación inferior alberga botones para

[7] Presentación de Material Design 2.0: https://andro4all.com/2018/07/diseno-google-material-design-video

[8] Tarjetas en Material Design: https://material.io/design/components/cards.html

[9] Enlace al artículo: http://babich.biz/using-card-based-design-to-enhance-ux/

alternar entre las diferentes secciones de la aplicación. Estas secciones son a las que los usuarios más van a acceder, por lo que, para decidir cuáles incluir, hay que pensar qué queremos hacer más accesible. Por otro lado, el apartado del menú lateral contiene habitualmente accesos a otras secciones fuera del inicio y configuraciones, así como la opción de cerrar sesión. Además de estos dos métodos, es habitual colocar botones que añadirán o cambiarán las publicaciones que se muestran. Estos botones suelen ubicarse en forma de botón flotante en la parte inferior derecha sobrepuesto al resto de contenidos (diseño propuesto por Material Design) o en la parte superior derecha, junto al título de la sección. La segunda opción es más recomendable ya que coincide con ambos patrones de diseño (Android e iOS).

Por último, se tiene en cuenta que la aplicación debe seguir los patrones de diseño definidos en cada uno de los sistemas operativos móviles (Android e iOS). Android los define con Material Design [10] e iOS con las guías de diseño de Apple[11].

La piedra angular de esta aplicación son los eventos. Éstos pueden incluir mucha información que en ocasiones puede llegar a resultar abrumadora para el usuario, por lo que se decide que debe poder mostrarse un pequeño extracto de cada publicación con la información necesaria. Además, tiene que hacer saber al usuario que puede ser pulsado para visualizar el resto de la información, que está oculta. Teniendo esto en cuenta y tras el previo análisis, se decide mostrar los eventos publicados en tarjetas (figura 5) que contienen la siguiente información:

- Un recorte de la imagen, que al pulsarla se mostrará en su totalidad.
- La foto del usuario que ha creado el evento.
- Título del evento.
- Primeros 150 caracteres de la descripción del evento.
- Fecha en la que se celebra.
- Ubicación.
- Campus (pulsable para mostrar una pantalla de eventos de dicho Campus).
- Etiqueta/Categoría (pulsable para mostrar una pantalla de eventos de dicha categoría).
- Dos botones: uno para guardar el evento y otro para compartirlo.
- La puntuación que tiene el evento y tres estrellas para evaluarlo.
- Un botón para cambiar la asistencia a dicho evento.

El diseño de las tarjetas hace que al usuario se le haga “natural” pulsarlas para ver más información, tal y como se explica en el artículo de Nick Babich (bajo el título “Design With Thumbs in Mind”), por lo que el resto de los datos del evento son mostrados en dicho momento (figura 6). Además de la información que se veía en la tarjeta sin expandir, se ve:

- La descripción completa del evento.
- El nombre del usuario que ha creado el evento.
- Sección de comentarios.

[10] Material Design: https://material.io/

[11] Guías de diseño de Apple: https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/

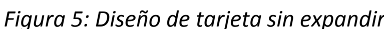
Figura 5: Diseño de tarjeta sin expandir

Figura 6: Diseño de tarjeta expandida

Además de la pulsación habitual, si el usuario mantiene la tarjeta pulsada un segundo, se muestra un menú contextual con dos opciones, siempre y cuando el evento haya sido creado por dicho usuario (figura 7): Editar evento y Borrar evento.

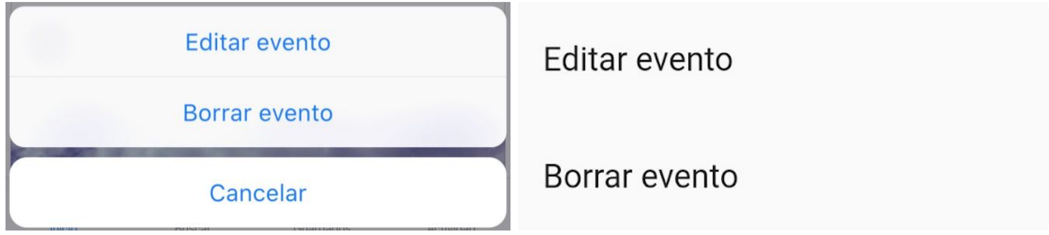

Figura 7: Menú contextual, izquierda iOS, derecha Android

La sección de comentarios se muestra como en muchas aplicaciones (figura 8). En la parte superior hay una barra que permite la escritura, y debajo están los comentarios separados por una línea. Cada comentario muestra al final la puntuación sobre una flecha que indica al usuario que puede deslizar el comentario. Si lo desliza, se muestran las opciones de puntuación tal y como se hace en cada evento.

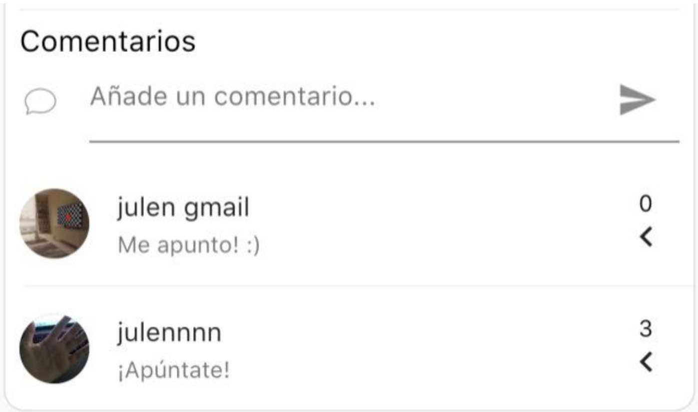

Figura 8: Sección de comentarios

En la pantalla de inicio de sesión se muestran los tres botones de inicio de sesión bajo el logo de la aplicación y un pequeño texto. Además, se muestra una previsualización de algunos eventos para que el usuario anónimo pueda conocer qué tipo de publicaciones contiene la red social. Esto se puede ver en la figura 9.

Figura 9: Pantalla de inicio de sesión

En la pantalla de Inicio, aparte de los eventos, se muestran tres botones en la parte superior y cuatro en la inferior, como se puede ver en la figura 10. En la parte superior izquierda encontramos un botón para abrir el menú de ajustes de la aplicación, como se ve en la figura 11, mediante el que podemos entrar a nuestro perfil, cambiar al modo noche (la aplicación pasa de tener un fondo blanco a un fondo gris oscuro, figura 12), cambiar el idioma y cerrar la sesión. En la parte superior derecha encontramos el botón de añadir un nuevo evento y el botón de mostrar filtros (desaparece si cambiamos de pestaña en la parte inferior). Si pulsamos este último, se muestran los filtros en elementos llamados *Chips* encima de la lista de eventos (figura 13) que si son pulsados cambian los

eventos que se muestran debajo para ajustarse a los filtros seleccionados. Por otro lado, los botones de la parte inferior cambian lo que se muestra en la pantalla para mostrar los siguientes apartados: *Inicio*, que muestra los eventos (figura 14), *Buscar*, que muestra una barra de entrada de texto y la posibilidad de escribir para buscar eventos o usuarios (figura 15), *Guardados*, que muestra los eventos que el usuario haya guardado (esta lista no es pública, figura 16) y *Actividad*, que muestra las notificaciones que haya recibido el usuario de otros usuarios (figura 17).

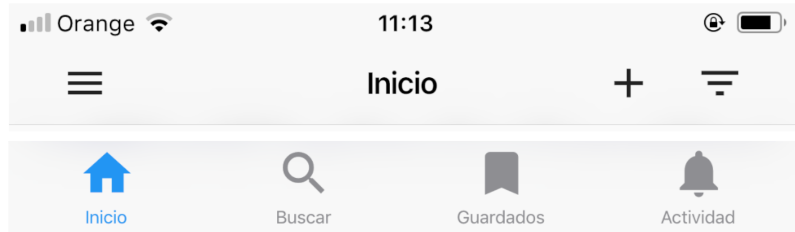

Figura 10: Barras superior e inferior

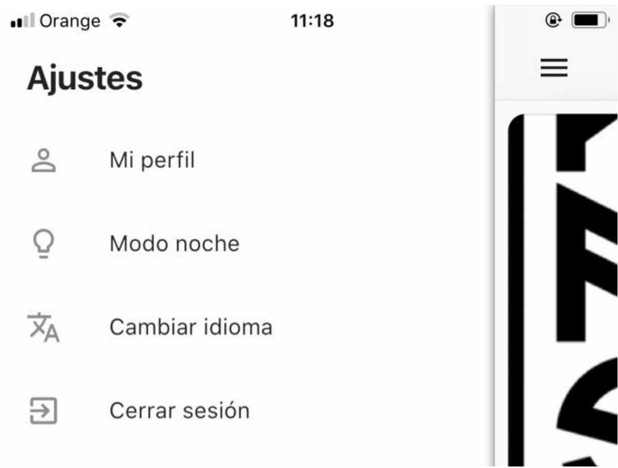

Figura 11: Menú lateral de opciones

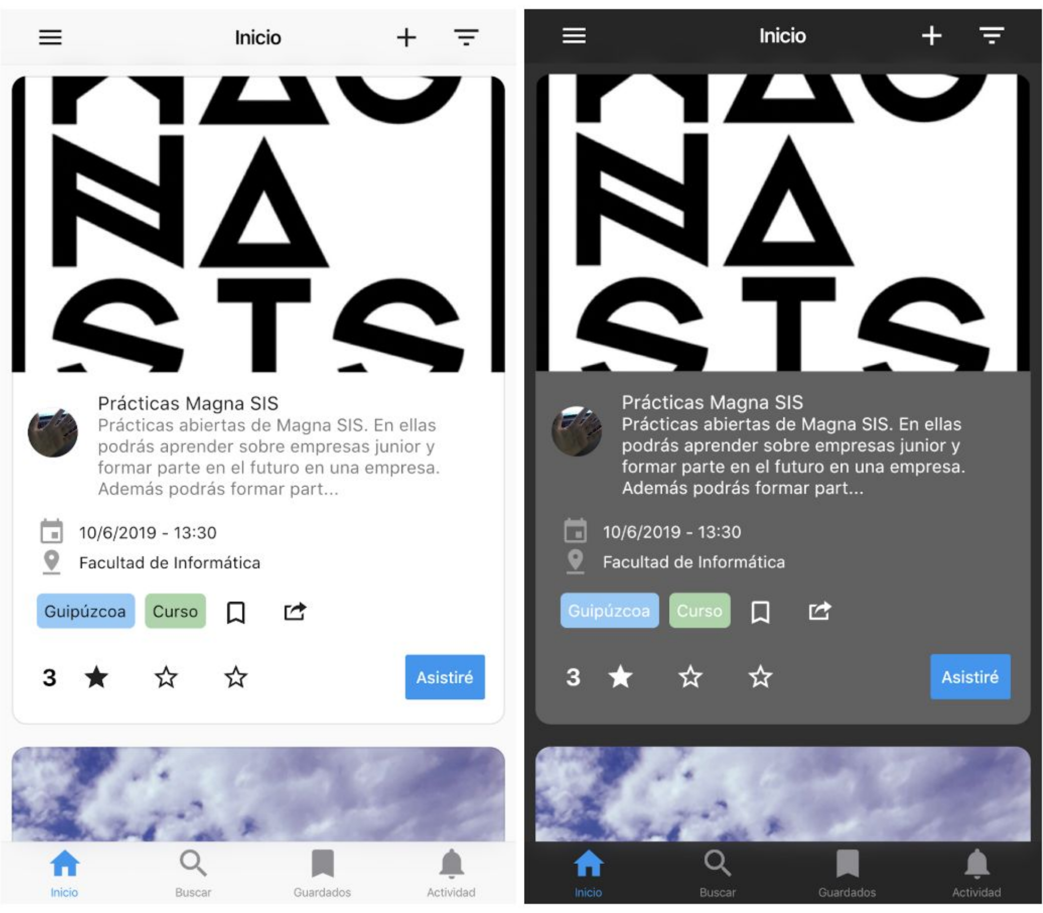

Figura 12: Modo día a la izquierda, modo noche a la derecha

Figura 13: Filtros que aparecen sobre los eventos

Figura 14: Pantalla de inicio

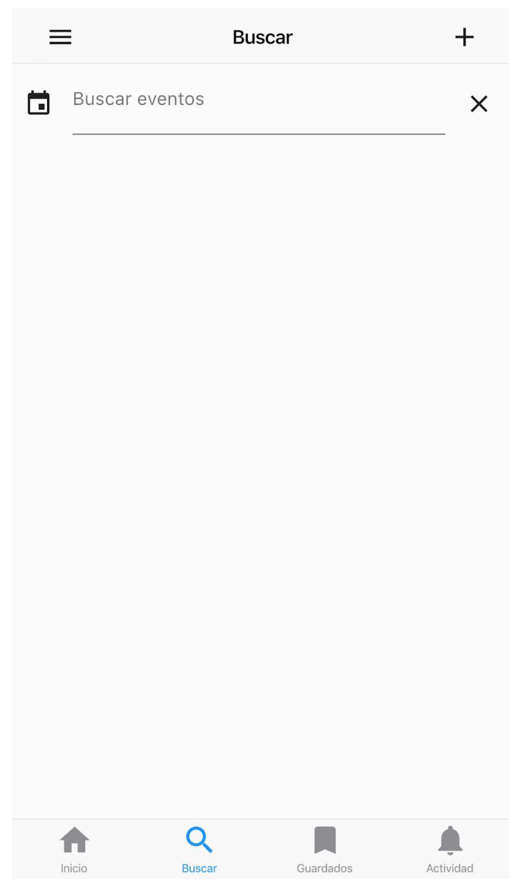

Figura 15: Pantalla de búsqueda

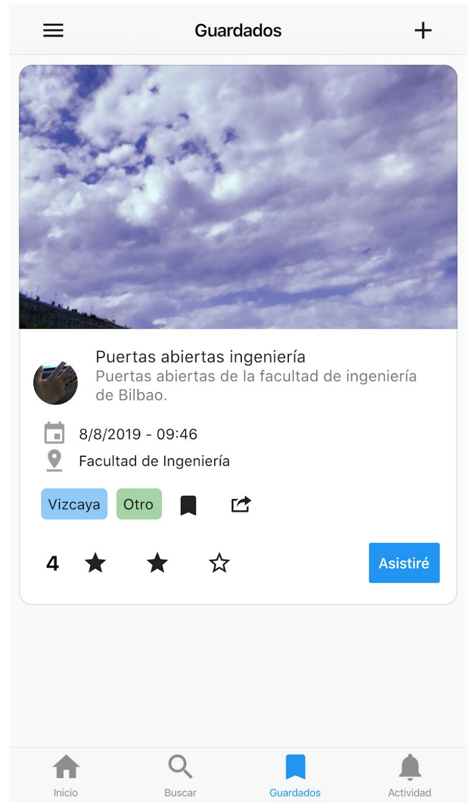

Figura 16: Pantalla de guardados

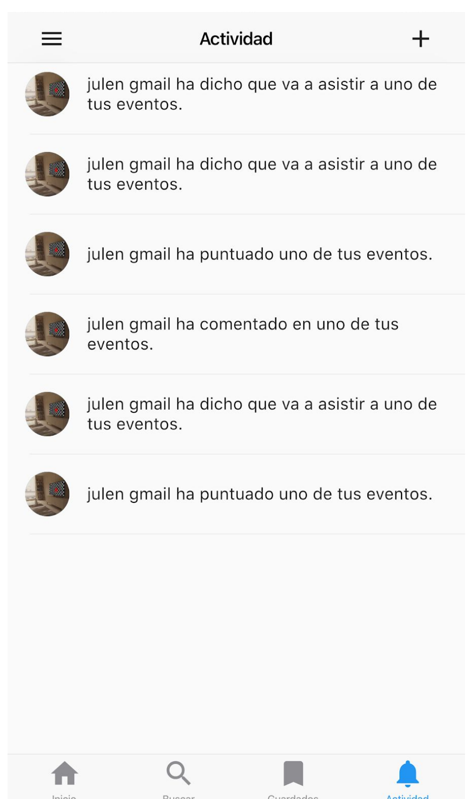

Figura 17: Pantalla de actividad

Todas las pantallas que muestran contenido que puede cambiar precisan de una forma de actualización. Para ello se implementa un método de actualización arrastrando el contenido hacia abajo, tal y como siguen muchas aplicaciones, llamado *pull-to-refresh*. Si se arrastra el contenido hacia abajo, se muestra una sombra que indica que se está realizando una acción, curvándose al llegar al final y mostrando un círculo que indica la actualización (figura 18).

Figura 18: Proceso de actualización pull-to-refresh

Tanto si se pulsa la imagen de algún usuario como el botón de "Mi perfil" o si se busca un usuario, se muestra el perfil de dicho usuario (figura 19). Enseña un espacio superior con la imagen de perfil desenfocada como fondo y la imagen de perfil normal junto al nombre y un aviso de "Te sigue" si el usuario cuyo perfil se está viendo sigue al usuario que utiliza la aplicación. Seguido, podemos ver algunos datos del usuario, como el número de *seguidores*, el número de *seguidos* y el número de eventos a los que va a asistir o ha asistido, un botón de seguir o dejar de seguir (este botón cambia a "Editar perfil" si el perfil visualizado es el del usuario que maneja la aplicación) y las puntuaciones que tiene el usuario. Estos dos apartados desaparecen si arrastramos el contenido hacia arriba, dejando visibles sólo los eventos a los que ha marcado la asistencia y los eventos creados.

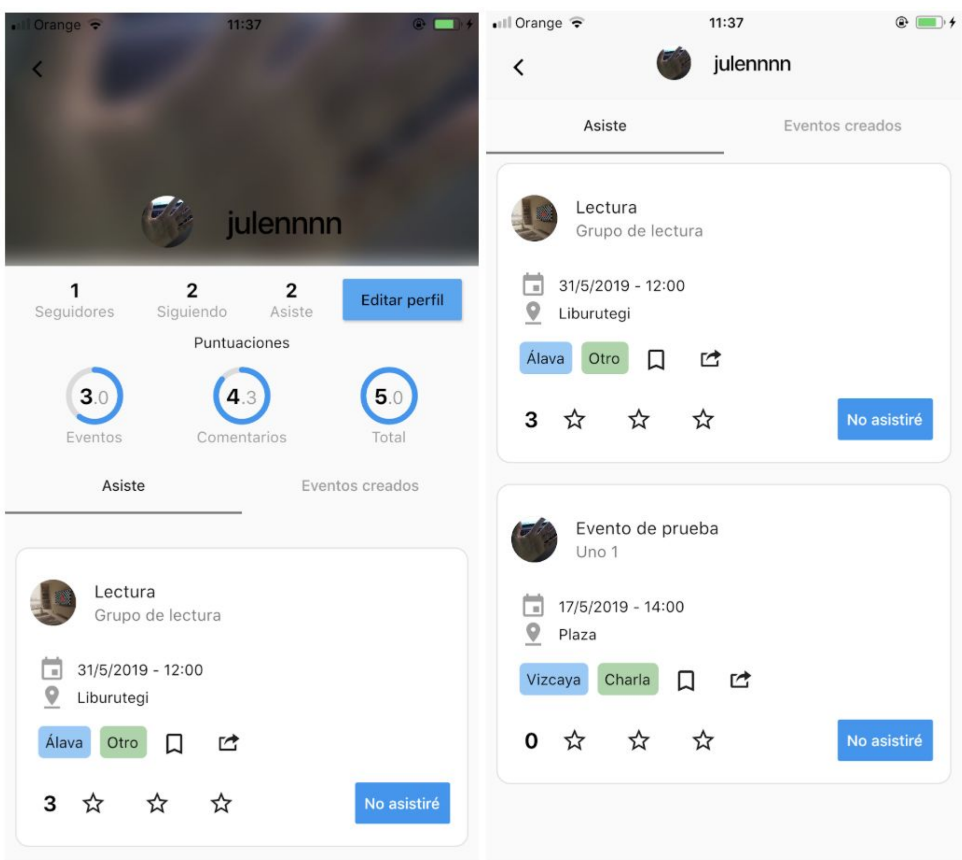

Figura 19: Perfil de usuario

Por último, el apartado de crear un evento (editar un evento sigue la misma interfaz) se muestra en dos secciones (figura 20). En la primera se puede escribir el título y la descripción del evento, seleccionar la etiqueta y añadir una imagen (opcional). Una vez rellenados los campos obligatorios, se puede pulsar "Siguiente", mostrando un mensaje de error si alguno de los campos queda vacío. En la segunda se puede seleccionar la fecha del evento, el campus en el que se va a celebrar dicho evento y la ubicación. Debajo se muestra un botón para terminar con el proceso.

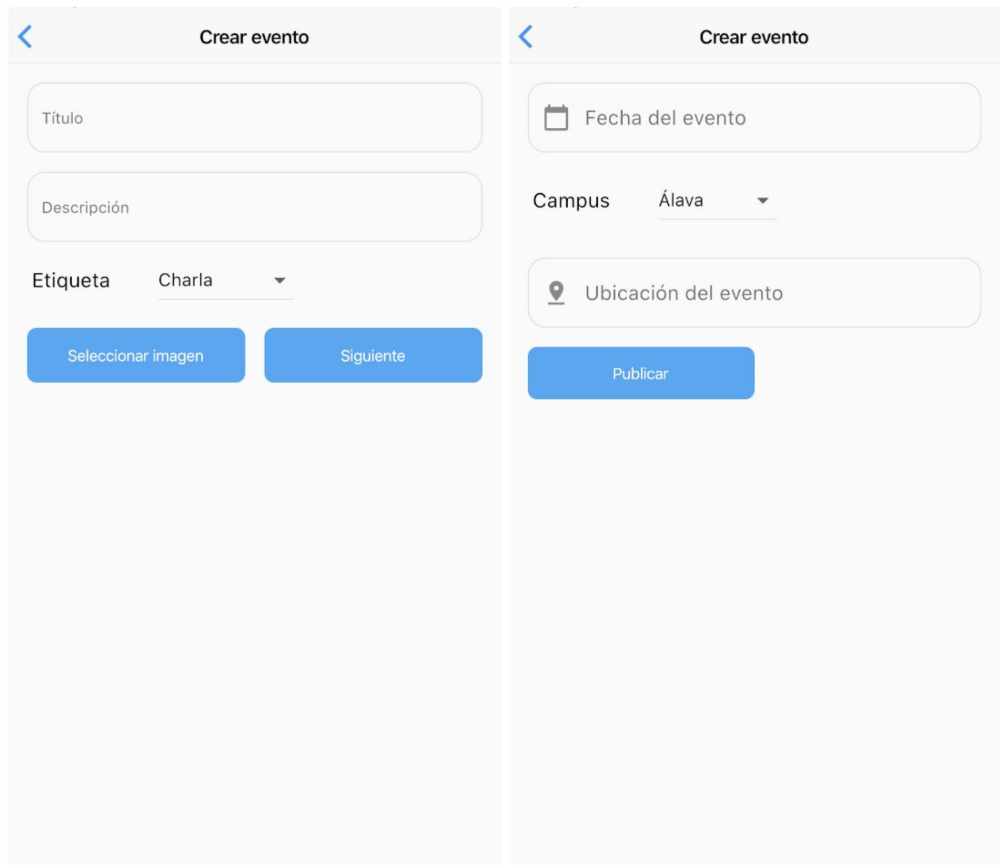

Figura 20: Crear evento

Puesto que la aplicación debe estar disponible tanto para Android como para iOS, la interfaz cambia ligeramente entre los dos sistemas operativos móviles para dar una experiencia de diseño diferente acorde a cada uno. Los cambios más notables se pueden encontrar en la barra superior (figura 21) y en los menús contextuales que aparecen (figura 7).

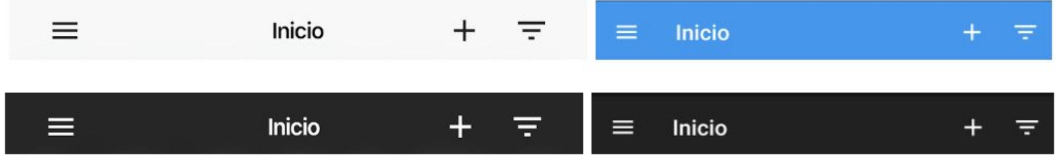

Figura 21: A la izquierda barras superiores de iOS, a la derecha las de Android

4.3 Modelo de datos

El modelo de datos de este proyecto se enfoca en la descripción del sustento de información de los elementos que intervienen en el problema a resolver y en la manera en que éstos se relacionan.

4.3.1 Base de datos

Las bases de datos NoSQL son muy utilizadas en proyectos donde se manejan grandes volúmenes de datos, ya que habitualmente son fácilmente escalables y, además, soportan procesamiento paralelo de datos de forma simultánea[12]. Existen numerosos tipos de bases de datos NoSQL y difieren en el sistema de almacenamiento de los datos. En nuestro caso empleamos el sistema de base de datos documental, debido a que Firestore se basa en éste. Este sistema se caracteriza por la estructuración de los datos, "a partir del modelo de datos NoSQL de Cloud Firestore, almacenas los datos en documentos que contienen campos que se asignan a valores. Estos documentos se almacenan en colecciones, que son contenedores para tus documentos y que puedes usar para organizar tus datos y compilar consultas"[13].

A la hora de hacer consultas en una base de datos NoSQL documental, cada vez que necesitemos un dato, el sistema cargará el documento completo, es decir, se hará la transferencia completa aunque no necesitemos el resto de los datos. Por ello, y con el fin de mejorar la eficiencia de las consultas, datos de diferentes documentos suelen estar duplicados, así se evita tener que transferir varios documentos. A causa de la duplicación de datos, es posible que las características ACID[14] no se mantengan, por lo que el programador tiene que asegurarlo a la hora de modificar la base de datos.

A continuación se muestra la estructura de la base de datos, separada en cuatro colecciones de documentos: colección de usuarios, colección de eventos, colección de comentarios y colección de notificaciones. Además, también se muestra el diseño gráficamente en el anexo 1.

[12] C. Lee and Y. Zheng, "SQL-to-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems," *2015 IEEE International Conference on Systems, Man, and Cybernetics*, Kowloon, 2015, pp. 2022-2026.

[13] Documentación oficial de Firestore: https://firebase.google.com/docs/firestore#how_does_it_work

[14] Acrónimo de **A**tomicity, **C**onsistency, **I**solation and **D**urability: Atomicidad, Consistencia, Aislamiento y Durabilidad en español.

Colección de usuarios (*users*): almacena los datos de cada usuario, y el identificador de cada uno de los documentos es el identificador del usuario (proporcionado por el servicio Firebase Auth). Guardamos en estos documentos sus puntuaciones, seguidores, seguidos, eventos a los que asiste, eventos creados y eventos guardados. Algunos datos, como los seguidores y los seguidos, se duplican para acceder a un solo documento para consultarlos sin tener que recoger los documentos de otros usuarios.

ID del documento: ID del usuario de Firebase Auth. **name**: string - El nombre del usuario. **imageUrl**: string - La dirección de la imágen de perfil del usuario. **commentScore**: number - La puntuación media que obtengan sus comentarios. **eventScore**: number - La puntuación media que obtengan sus publicaciones de eventos. **totalScore**: number - La puntuación media total del usuario. **followers**: array<string> - Los identificadores de los seguidores del usuario actual. **following**: array<string> - Los identificadores de los usuarios a los que sigue el usuario actual. **attending**: array<string> - Los identificadores de los eventos a los que asiste el usuario actual. **createdEvents**: array<string> - Los identificadores de los eventos que ha creado el usuario actual. **saved**: array<string> - Los identificadores de los eventos guardados por el usuario actual.(ID del evento) **creationDate**: timestamp - La fecha de creación del usuario actual.

Tabla 1: Diseño de los documentos users

Colección de eventos (*events*): almacena los datos de los eventos, el identificador de cada uno de los documentos es el identificador del evento el cual se asigna automáticamente al generar el documento. Se duplican los datos del usuario creador para no tener que acceder a su documento cuando se quiera mostrar en el evento. Además, también se duplican algunos comentarios, ya que la mayoría de usuarios solo ven unos pocos comentarios, y de esta forma evitamos tener que acceder a los documentos de los comentarios.

ID del documento: ID del evento (automático)
title: string - El título de la publicación.
description: string - La descripción de la publicación.
imageName: string - El nombre de la imágen en Firebase Storage.
oneStar: array<string> - Los identificadores de los usuarios que han puntuado el evento actual con una estrella.
twoStars: array<string> - Los identificadores de los usuarios que han puntuado el evento actual con dos estrellas.
threeStars: array<string> - Los identificadores de los usuarios que han puntuado el evento actual con tres estrellas.
score: number - La puntuación media del evento actual.
comments: array<map> - Parte de los comentarios a mostrar del evento.

- **uid**: string - El identificador del usuario creador del comentario actual.
- **name**: string - El nombre del usuario creador del comentario actual.
- **imageUrl**: string - La imagen de perfil del usuario creador del comentario actual.
- **comment**: string - El texto del comentario actual.
- **commentID**: string - El identificador del comentario actual.
- **oneStar**: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con una estrella.
- **twoStars**: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con dos estrellas.
- **threeStars**: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con tres estrellas.
- **score**: number - La puntuación media del comentario actual.
- **creationDate**: timestamp - La fecha de creación del comentario actual.

creationDate: timestamp - La fecha de creación del evento actual.
eventDate: timestamp - La fecha de celebración del evento actual.
campus: number - El número del campus (0 - Araba, 1 - Bizkaia, 2 - Gipuzkoa).
eventLocation: string - La ubicación en la que se celebrará el evento actual.
attending: array<string> - Los identificadores de los usuarios que asistirán al evento actual.
tag: number - El número de la etiqueta (0 - Charla, 1 - Taller, 2 - Competición, 3 - Exposición, 4 - Actuación, 5 - Curso, 6 - Fiesta, 7 - Otros).
uid: string - El identificador del usuario creador del evento actual.
imageUrl: string - La imagen de perfil del usuario creador del evento actual.
name: string - El nombre del usuario creador del evento actual.

Tabla 2: Diseño de los documentos events

Colección de comentarios (*comments*): almacena los datos de los comentarios de los eventos, el identificador de cada uno de los documentos es el identificador del comentario y se asigna automáticamente al generar el documento. Se duplican los datos del usuario creador para no tener que acceder a su documento cuando se quiera mostrar en el evento.

ID del documento: ID del comentario (aleatorio)
uid: string - El identificador del usuario creador del comentario actual.
name: string - El nombre del usuario creador del comentario actual.
imageUrl: string - La imagen de perfil del usuario creador del comentario actual.
comment: string - El texto del comentario actual.
oneStar: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con una estrella.
twoStars: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con dos estrellas.
threeStars: array<string> - Los identificadores de los usuarios que han puntuado el comentario actual con tres estrellas.
score: number - La puntuación media del comentario actual.
creationDate: timestamp - La fecha de creación del comentario actual.
polarity: number - La polaridad del texto del comentario actual.
eventID: string - El identificador del evento al que pertenece el comentario actual.

Tabla 3: Diseño de los documentos comments

Colección de notificaciones (*notifications*): almacena los datos de notificaciones de cada usuario, el identificador de cada uno de los documentos es el identificador del usuario (proporcionado por el servicio Firebase Auth). Tiene una lista con los tokens de cada usuario para saber a qué dispositivos enviar las notificaciones y el idioma que utiliza para adaptar los textos. Además, se guardan las notificaciones en una lista para poder mostrarlas en la pantalla de Actividad.

ID del documento: ID del usuario de Firebase Auth.
tokens: array<string> - Los identificadores de los dispositivos del usuario actual a los que mandar notificaciones.
lang: string - El idioma que utiliza el usuario en la aplicación.
countryCode: string - El código del país del idioma que utiliza el usuario en la aplicación.
notifications: array<map> - Las notificaciones del usuario actual.
- **type**: number - El tipo de notificación (0 - seguimiento, 1 - puntuación, 2 - asistencia, 3 - comentario).
- **date**: timestamp - La fecha en la que se produce la notificación.
- **fromUid**: string - El identificador del usuario origen de la notificación.
- **eventID**: string - El identificador del evento al que está asociada la notificación (tendrá valor nulo si el tipo es 0).

Tabla 4: Diseño de los documentos notifications

4.3.2 Sistema de ficheros de la aplicación

Por otro lado, la lógica de los ficheros de la implementación de la aplicación sigue una estructura similar a la de la arquitectura de la aplicación (que puede verse expresada gráficamente en el punto 4.1.2), por lo que se divide en tres directorios principales:

- **Acceso a Datos**: contiene los archivos que conectan con el servicio a la base de datos tanto para almacenar nuevos datos como para actualizarlos y hacer consultas. Hay dos archivos principales: el que conecta con Firestore y el que conecta con Algolia.
- **Lógica de Negocio**: los archivos de la lógica de negocio se encargan de realizar el procesamiento detrás de la interfaz gráfica de usuario.
- **Interfaz Gráfica de Usuario (GUI)**: contiene todos los archivos que se encargan de mostrar interfaces gráficas a los usuarios para que puedan interaccionar con la aplicación y con sus datos. Dentro encontramos una carpeta que contiene los archivos que ofrecen el servicio multi-idioma, una carpeta en la que se guardan los Widgets que se han creado para el proyecto (como las tarjetas de los eventos) y, por último, una carpeta que tiene las cuatro interfaces que corresponden a las cuatro "pestañas" de la pantalla de inicio.

Para la mayor parte de la implementación se ha utilizado el control de versiones Git. Concretamente, el servicio Git en la nube que ofrece BitBucket.

4.4 Casos de uso

En este punto se describen los casos de uso de cada uno de los actores que intervienen en el manejo de la aplicación.

4.4.1 Usuario anónimo

El usuario anónimo es aquel que entra a la aplicación y no ha iniciado sesión, por lo que no se conoce su identidad. Se describe el siguiente diagrama de casos de uso para este tipo de usuario:

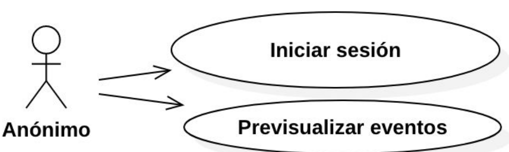

Figura 22: Casos de uso del usuario anónimo

Iniciar sesión: un usuario anónimo puede registrarse en el servicio con los métodos de autenticación de Twitter, Facebook o Google. Una vez inicia sesión con su cuenta en uno de estos servicios, se debe que comprobar si es la primera vez que el usuario entra a la red social. Si es así, se genera un documento nuevo en Firestore con algunos de los datos que proporciona Firebase Auth.

El servicio Firebase Auth almacena un token en el dispositivo para que el usuario no tenga que realizar el inicio de sesión cada vez que abra la aplicación. Como además nos da la información necesaria para construir el perfil del usuario, el proceso de inicio de sesión no se diferencia del de registro, ya que no es necesario pedir información adicional al usuario.

A continuación se detalla el funcionamiento del caso de uso en la aplicación mediante su diagrama de flujo (figura 23):

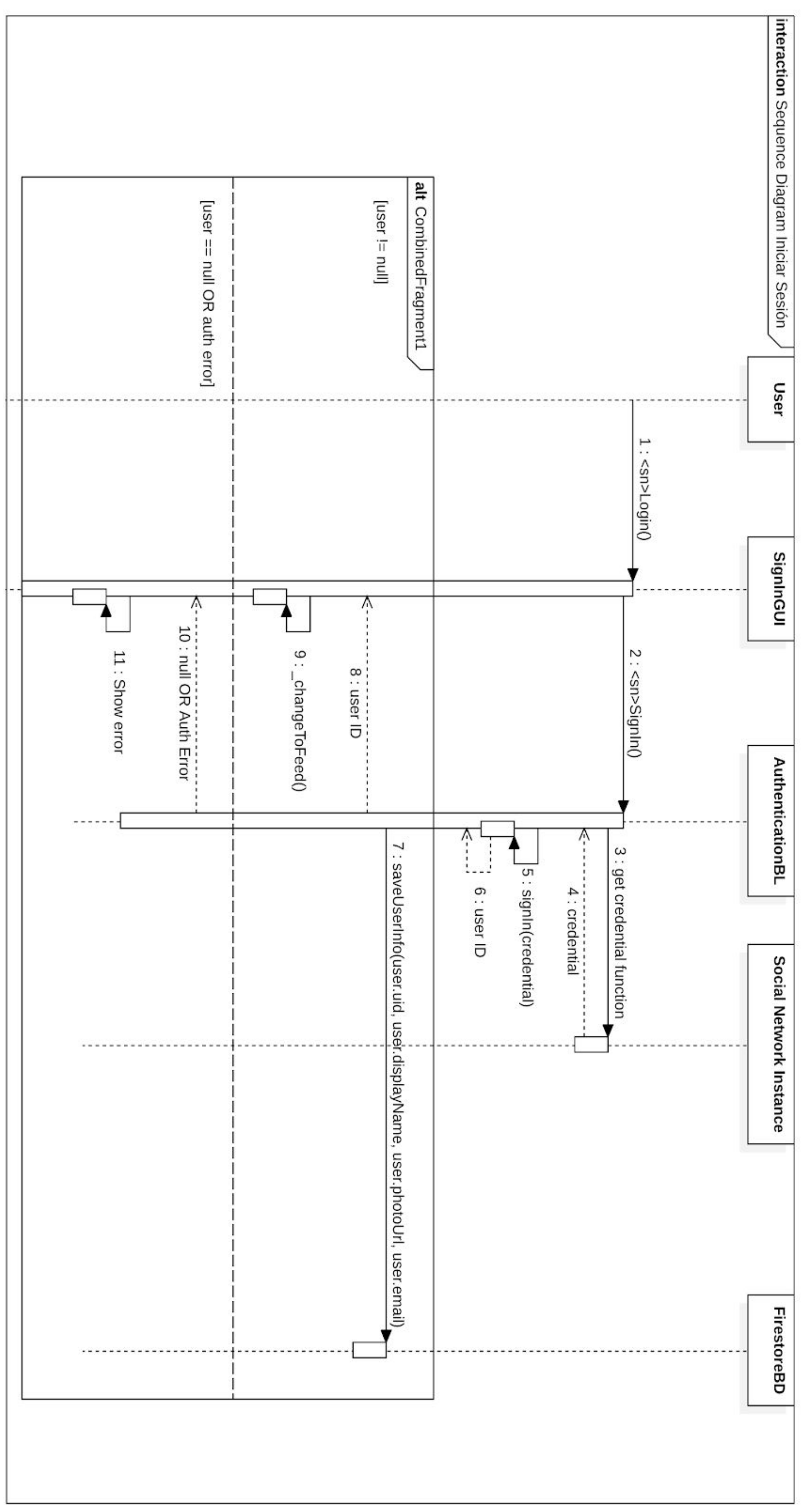

Figura 23: Diagrama de flujo de iniciar sesión

Previsualizar eventos: en la misma pantalla en la que se ven los botones de inicio de sesión, se muestran tarjetas en la parte inferior con la imagen del evento, su título y descripción. Estas tarjetas a modo de previsualización se deslizan de manera automática.

4.4.2 Usuario registrado

El usuario registrado es aquél que ha pasado por el inicio de sesión, por lo que se conoce su identidad. Se describe el siguiente diagrama de casos de uso para este tipo de usuario (figura 24).

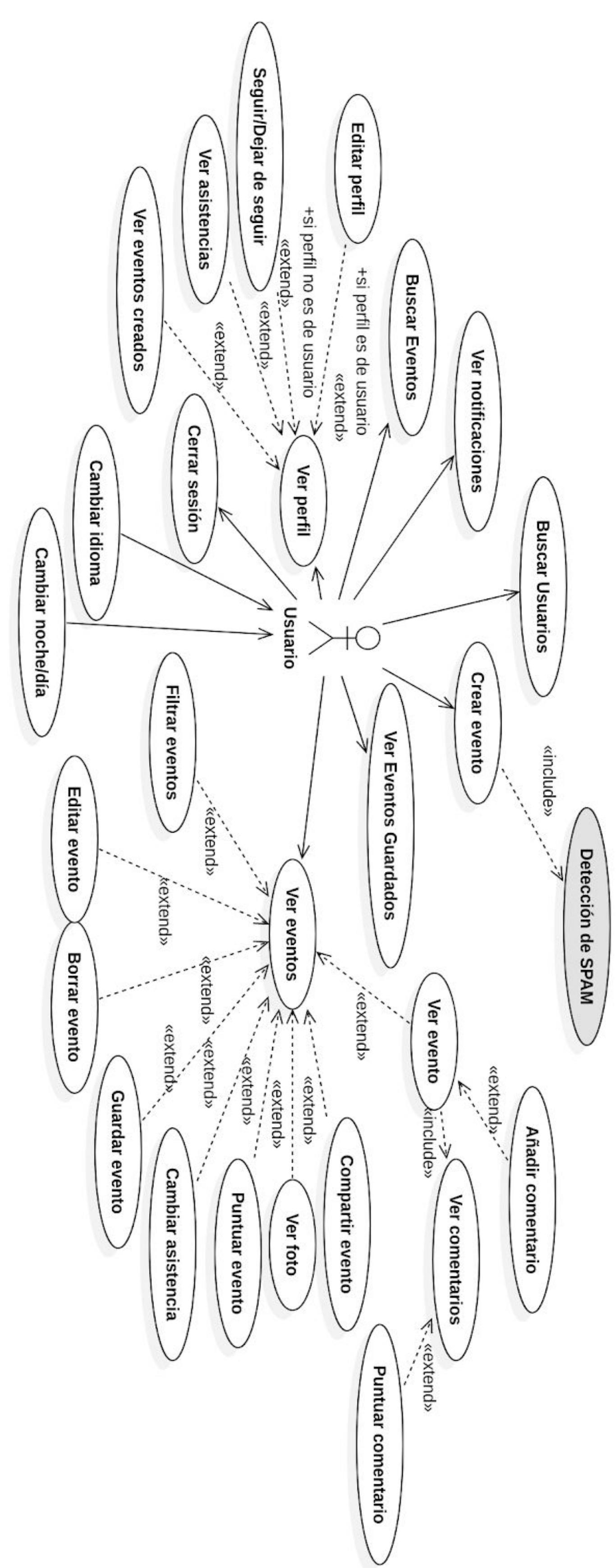

Figura 24: Casos de uso de un usuario registrado

Crear evento: las publicaciones de esta red social son los eventos. Cualquier usuario puede publicar un evento que se muestre al resto de usuarios. Éste tiene los campos establecidos en el diseño de la base de datos. El usuario debe rellenar los siguientes datos a la hora de crearlo: el título, la descripción o texto que acompañe, una imagen (opcional), la fecha del evento, el campus, la localización y una etiqueta que acompañe al evento. La resolución de la imagen se reduce para que los tiempos de carga al subir la imagen al almacenamiento en la nube se vean drásticamente reducidos. Toda esta información se pide en diferentes pantallas por las que navega el usuario. Primero se pide el título, la descripción, la etiqueta y la imagen. Una vez rellenados, se llama al caso de uso *Detección de SPAM*, perteneciente a Smart (TFG-Elsa), para solo permitir continuar en caso de que el contenido de la descripción no sea SPAM. A continuación, la fecha, el campus y la localización. El usuario tiene la opción de cancelar durante todo el proceso y la opción de publicar en la segunda pantalla. El diagrama de flujo de este evento puede verse en el anexo 2.

Ver eventos: los usuarios pueden ver los eventos que publiquen los usuarios. Éstos se almacenan en la base de datos NoSQL Firestore, por lo que se hace una petición a ésta para que devuelva una colección con los diez eventos más recientes (se muestran primero estos diez eventos y se da una opción para cargar más). Los eventos se muestran en tarjetas con información reducida: la imagen en un tamaño fijo, la foto del usuario que haya creado el evento, el título, los primeros 150 caracteres de la descripción, la ubicación, la fecha del evento, el campus, la categoría del evento, la opción de puntuar y el botón para marcar/desmarcar la asistencia al evento. Desde estas tarjetas, el usuario puede acceder a los siguientes casos de uso que completan la experiencia del usuario con la interacción de los eventos: *Filtrar eventos*, *Editar evento*, *Borrar evento*, *Ver foto*, *Puntuar evento*, *Cambiar asistencia*, *Guardar evento*, *Ver evento*.

Filtrar eventos: permite al usuario cambiar los eventos que se muestran. Éstos se pueden filtrar por dos parámetros que se muestran sobre la lista de eventos al pulsar el botón de filtro: Campus y Categoría. Los filtros pueden combinarse.

Editar evento: es accesible al pulsar prolongadamente la tarjeta de un evento, y permite al usuario editar la información de un evento que haya creado. Muestra las mismas interfaces que *Crear evento* pero con los campos rellenos con los datos correspondientes.

Borrar evento: es accesible al pulsar prolongadamente la tarjeta de un evento, y permite al usuario borrar un evento que haya creado.

Ver foto: en la tarjeta en la que se muestra la información del evento se puede pulsar la imagen para poder verla a tamaño completo. Esto se hace porque la imagen que se muestra en la tarjeta siempre tiene un tamaño definido, lo que hace que no se pueda ver la imagen completa en estas tarjetas. En la pantalla de visualización de imagen es posible hacer zoom.

Puntuar evento: en la tarjeta en la que se muestra el evento, se muestran junto a un número tres estrellas que sirven para puntuar el evento. El número que se muestra es la puntuación del evento. Estas estrellas aparecen marcadas con el número de estrellas con el que haya puntuado el usuario dicho evento, siendo una estrella la puntuación más baja y tres la más alta.

Cambiar asistencia: en la tarjeta en la que se muestra la información del evento aparece un botón para cambiar la asistencia a un evento. Gracias a esto podemos decir que asistiremos a un evento o podemos quitar la asistencia si el usuario cambia de opinión. El botón muestra el texto "Asistiré" si aún no se ha marcado la asistencia y, en el caso de haber pulsado el botón anteriormente, se muestra el texto "No asistiré".

Guardar evento: en la tarjeta en la que se muestra la información del evento aparece un botón para poder guardar el evento o borrarlo de los eventos guardados. Estos eventos pueden guardarse para que el usuario pueda verlos más tarde y no se muestran de manera pública. Se pueden ver los eventos guardados en el caso de uso *Ver Eventos Guardados*.

Compartir evento: en la tarjeta en la que se muestra la información del evento aparece un botón para poder compartir el evento mediante un enlace. Al pulsarlo, se abre el menú de compartir por defecto de cada sistema operativo (Android o iOS), lo que permite enviar el enlace por diferentes medios.

Ver evento: si se pulsa la tarjeta de uno de los eventos, esta tarjeta se expande para mostrar la información que no se muestra en el caso de uso *Ver Eventos*. La información adicional que se muestra es la siguiente: la descripción entera, el nombre del usuario que ha creado el evento, la fecha de creación, los comentarios (provienen del caso de uso que incluye, *Ver Comentarios*) y la posibilidad de comentar el evento (caso de uso *Añadir comentario*).

Ver comentarios: cuando se pulsa una tarjeta de un evento, ésta se expande para mostrar toda la información (caso de uso *Ver evento*), y en la parte inferior de la tarjeta se muestran los comentarios que se han hecho en los eventos. Cada comentario muestra la foto de la persona que haya hecho el comentario, su nombre, el comentario y la puntuación que tenga. Si se arrastra el comentario hacia la izquierda se puede puntuar el comentario (caso de uso *Puntuar comentario*).

Puntuar comentario: cada comentario puede deslizarse de derecha a izquierda para mostrar tres estrellas que sirven para puntuar el comentario, de la misma forma que se puede puntuar un evento. Este sistema de puntuación está escondido para ocupar menos espacio en los comentarios. La puntuación de los comentarios se muestra en la parte derecha de cada comentario.

Añadir comentario: antes de los comentarios se muestra un campo de texto en el que se puede escribir un comentario y un botón para enviarlo.

Ver eventos guardados: en la pantalla de inicio uno de los botones de la navegación inferior es el de ver los eventos que haya guardado el usuario. Estos eventos son los que se hayan marcado con el caso de uso *Guardar Evento*. Esta lista de eventos es privada.

Ver notificaciones: en la pantalla de inicio uno de los botones de la navegación inferior es el de ver las notificaciones del usuario, llamado *Actividad*. Recoge las notificaciones que haya recibido el usuario: seguimiento, asistencia a sus eventos, puntuaciones, comentarios...

Buscar eventos: en la pantalla de inicio uno de los botones de la navegación inferior es el de búsqueda, donde se permite buscar entre los eventos mediante un campo de escritura.

Buscar usuarios: en la pantalla de inicio uno de los botones de la navegación inferior es el de búsqueda, donde se permite buscar entre los usuarios mediante un campo de escritura.

Ver perfil: pulsando sobre la imagen de un perfil en la tarjeta del evento o buscando un perfil en el buscador se muestra el perfil de esa persona. Pueden verse la foto, el nombre, si el usuario te sigue o no, número de seguidores, número de seguidos, número de eventos a los que ha marcado la asistencia, las puntuaciones (puntuación de eventos, de comentarios y total) y un botón para poder seguir o dejar de seguir al usuario (caso de uso *Seguir/Dejar de seguir*). Si el perfil que se está mostrando es el del usuario que está usando la aplicación, se muestra un botón para editar el perfil (caso de uso *Editar Perfil*) en vez del botón para seguir o dejar de seguir al usuario. También se ven

los eventos a los que ha marcado que va a asistir (caso de uso *Ver asistencias*), así como los eventos que ha creado (caso de uso *Ver eventos creados*).

Editar perfil: en la información del perfil aparece un botón para poder editar el perfil propio. Este botón se muestra si el perfil que se esté visualizando es el del usuario que está usando la aplicación. Se abre una nueva pantalla en la que se puede modificar la foto de perfil y el nombre que se muestra. Si se modifica cualquiera de estos dos, los datos se modifican en todos los eventos y en todos los comentarios (ya que estos dos datos se guardan por duplicado en los documentos de estas colecciones).

Seguir/Dejar de seguir: en la información del perfil aparece un botón para poder seguir o dejar de seguir al usuario que se esté visualizando. Este botón no se muestra si el perfil que se esté visualizando es el del usuario que está usando la aplicación.

Ver asistencias: debajo de la información del perfil hay dos pestañas. Una de ellas es *Asistencias*, donde se ven los eventos a los que el usuario haya marcado su asistencia. Dentro de cada evento pueden realizarse todos los casos de uso que extienden a *Ver Eventos*.

Ver eventos creados: debajo de la información del perfil hay dos pestañas. Una de ellas es *Creados*, donde se ven los eventos que el usuario haya creado. Dentro de cada evento pueden realizarse todos los casos de uso que extienden a *Ver Eventos*.

Cambiar noche/día: desde el menú lateral puede cambiarse el modo de visualización de la aplicación entre el Modo día (un modo claro con colores blancos y grises claros con textos oscuros) y el Modo noche (un modo oscuro con colores negros y grises oscuros con textos claros).

Cambiar idioma: desde el menú lateral puede cambiarse el idioma en el que se muestran los textos principales de la aplicación (los textos de los eventos no cambian).

Cerrar sesión: desde el menú lateral el usuario puede cerrar su sesión. Una vez pulse la opción, el token de Firebase Auth se elimina y se muestra la pantalla de inicio en el que se le permite volver a iniciar sesión (caso de uso *Iniciar sesión* del *Usuario Anónimo*).

5. Tecnologías

En este apartado se describen y analizan algunos aspectos destacables del proyecto. Concretamente, aquellos relacionados con los servicios y tecnologías que se han usado, y aquellas tecnologías en las que ha resultado necesario pasar por un proceso de formación previa de manera autónoma. Se estructura en tres apartados, uno por cada tecnología utilizada, siendo dos de ellos sobre Flutter y Firebase, las dos tecnologías más novedosas y relevantes de entre las usadas exhaustivamente en el Trabajo de Fin de Grado. Aunque son unas tecnologías con breve recorrido, han demostrado que saben adaptarse a las necesidades del mercado con soluciones eficientes y simples de implementar.

5.1 Flutter

Figura 25: Logo de Flutter

Flutter[15] es un entorno de trabajo de código abierto creado por Google, presentado en 2015 y lanzado en mayo de 2017. Es utilizado para desarrollo de aplicaciones multiplataforma nativas. Se generan versiones para las plataformas Android, iOS y Google Fuchsia (el nuevo sistema operativo que pretende reemplazar a Android en los próximos años). Además, Google anunció durante *Google I/O* (mayo 2019) la posibilidad de generar aplicaciones para escritorio (Windows, macOS y Linux) y versiones web en un futuro cercano utilizando el mismo código implementado para las aplicaciones móviles.

Esta herramienta de desarrollo permite crear nuevas funcionalidades sin comprometer la calidad o el rendimiento de la aplicación. Posibilita conseguir gran rendimiento e integración de aplicaciones nativas con desarrollo ágil, además de disponer de herramientas de diseño de UI multiplataforma/portables.

A la hora de desarrollar aplicaciones móviles existen tres métodos: desarrollo nativo, desarrollo híbrido y desarrollo web app. Las diferencias más notables entre ellos están en la eficiencia de las aplicaciones y en la integración que tienen las mismas con el resto del sistema operativo móvil.

La **programación nativa** es la que se utiliza específicamente para cada sistema operativo (Objective-C o Swift para iOS y Java o Kotlin para Android). Dota a las aplicaciones de la mayor eficiencia posible, además de poder integrarse con el sistema completamente y permitir el acceso

[15] Página oficial de Flutter: https://flutter.dev/

completo a los recursos disponibles. Con este método, el acceso a los recursos o a la plataforma para crear elementos visuales se hace de manera directa (figura 26), por lo que se consigue el mayor rendimiento posible.

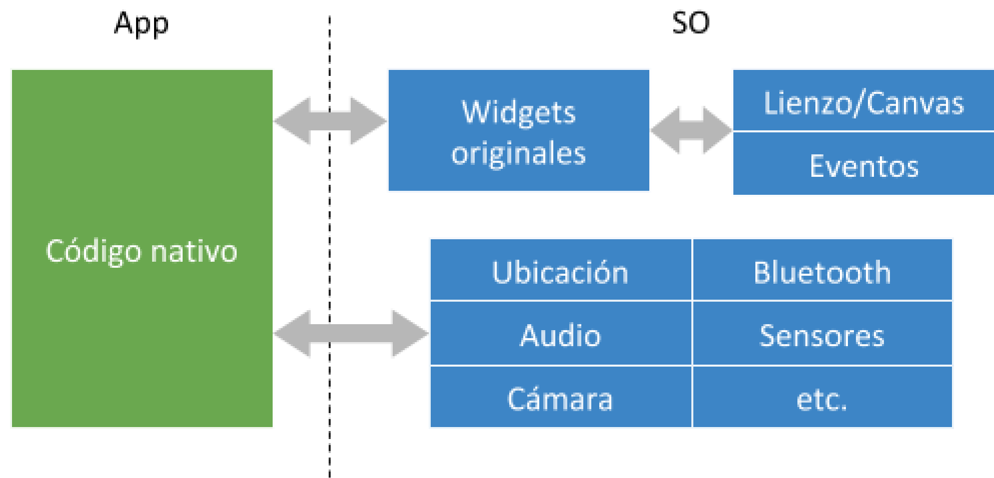

Figura 26: Esquema de aplicaciones nativas

La **programación híbrida** permite utilizar un mismo código para generar aplicaciones para diferentes plataformas. Esto ayuda a que el mantenimiento necesario sea menor y que las actualizaciones se produzcan a la vez. Sin embargo, al no desarrollarse con el lenguaje nativo (normalmente se utiliza JavaScript o similar), cada *framework* debe encargarse de comunicar el código desarrollado con los recursos del sistema. Por este motivo, las aplicaciones híbridas precisan de un "puente" para poder crear los elementos gráficos o acceder a los servicios del sistema (figura 27). Esto hace que el rendimiento se pueda ver afectado drásticamente.

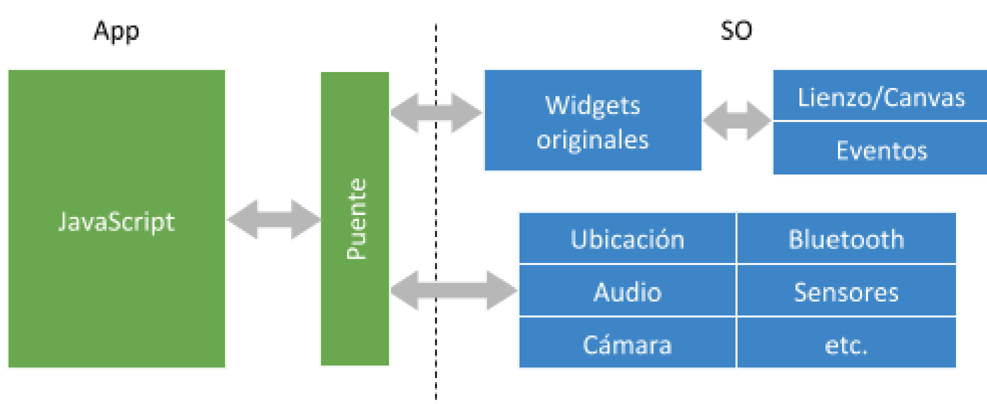

Figura 27: Esquema de aplicaciones híbridas

El último método, la **programación de web apps,** permite al desarrollador crear una página web que se adapte a las pantallas de los móviles para incrustarlas en un elemento llamado *Web View* y así utilizarlas como aplicaciones. La ventaja de esto es que estas aplicaciones ocupan mucho menos espacio dentro del smartphone, pero a cambio se pierde el acceso a muchos de los recursos del sistema operativo. Para poder acceder a estos recursos, este método precisa también de un "puente" (figura 28).

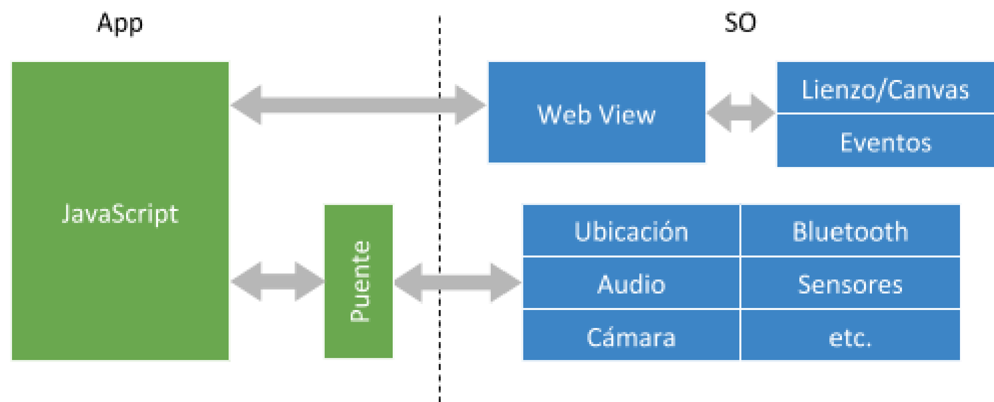

Figura 28: Esquema de las aplicaciones web

Flutter podría considerarse una herramienta de programación híbrida de aplicaciones por su capacidad de generar versiones de una misma aplicación para diferentes plataformas. No obstante, se diferencia del resto de herramientas (Xamarin, React Native...) en que su compilación genera código que se acerca más a la programación nativa, ya que gran parte de lo generado es código nativo (Objective-C, Swift, Java, Kotlin...). Gracias a esto mejora el rendimiento frente al resto de *frameworks* de programación híbrida. Además, gracias a la utilización de los *Widgets* (descritos más adelante) incluidos en Flutter (figura 29), las aplicaciones no tienen que acceder a los elementos visuales de los sistemas operativos, lo que tiene dos ventajas. La primera es que el acceso al lienzo y a los eventos se hace de manera más rápida y solo en los momentos necesarios. La segunda es que los elementos visuales se verán igual independientemente de la versión del sistema operativo que se esté utilizando. Es decir, una aplicación desarrollada con Flutter se verá igual tanto en iOS 8 como en iOS 13.

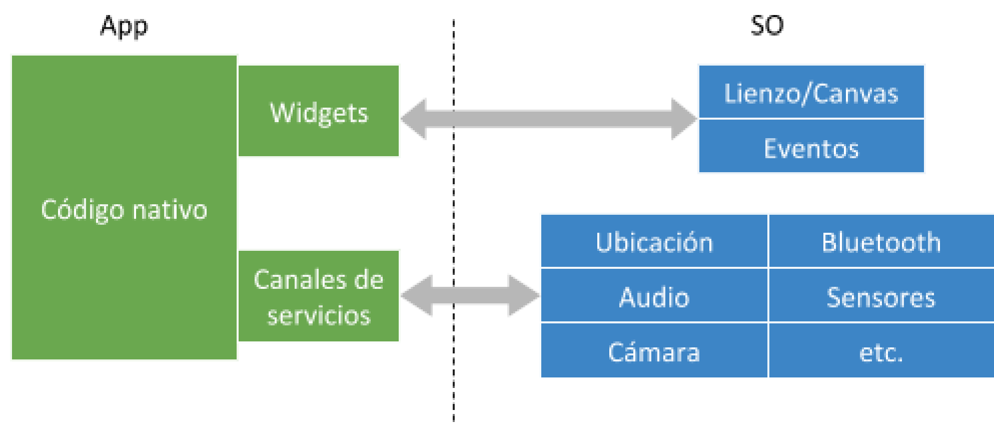

Figura 29: Esquema de las aplicaciones Flutter

Un *Widget* representa una serie de características gráficas que forman una clase para poder utilizarse en la interfaz gráfica de usuario. Son un conjunto de estándares que permiten la definición de elementos personalizados, encapsulados y reutilizables. Por ejemplo, el *Widget* "Text()" muestra en la pantalla el texto que se le pase como argumento. La forma en la que funcionan estos *Widgets* explica el gran rendimiento de las aplicaciones que se desarrollan con esta tecnología, por lo que constituyen un pilar fundamental de Flutter. Para entender el porqué, hay que entender cómo se

construye el *layout* (disposición, diseño) tanto en Flutter como en el resto de *frameworks* de programación híbrida.

Tradicionalmente se definen largas colecciones de normas para construir el diseño de una aplicación. Reglas y restricciones que establecen la posición, tamaño, color y etc. de los elementos gráficos. Por ejemplo CSS, que aplica múltiples estilos a cada elemento desde el inicio. Esto hace que cada elemento tenga hasta docenas de normas aplicadas, normas que pueden interaccionar entre ellas y hasta entrar en conflicto. En consecuencia, el dispositivo tiene que manejar muchos estilos para diferentes elementos, lo que hace que el rendimiento baje cuantos más elementos se incluyan.

Para solucionar esto, Flutter define una forma diferente de aplicar estilos a los *Widgets*. Se basa en aplicar a cada *Widget* las normas de *layout* correspondientes, en vez de atender a las normas generales. Cada *Widget* tendrá sus propias normas de estilo definidas dentro del mismo, o heredadas de otro *Widget* superior (por ejemplo, los *Widget Center* y *Padding* aplican un estilo concreto a los *Widget* hijos). Además, cada colección de pequeñas normas de *layout* es a su vez otro *Widget*, lo que simplifica su escritura y comprensión.

Otro de los motivos que explican la eficiencia de Flutter se encuentra en la forma en la que actualiza cada elemento gráfico cuando cambian (al cambiar un texto o un color cuando la aplicación está siendo ejecutada, por ejemplo). Las aplicaciones creadas con **programación web** hacen uso de un DOM virtual (Document Object Model, utilizado por JavaScript para manipular un documento HTML y representado como un árbol de elementos, figura 30). Este elemento es inmutable, por lo que se debe reconstruir cada vez que algún elemento gráfico de la aplicación cambia. El DOM virtual después es comparado con el DOM real para realizar el menor número de modificaciones posibles. Aunque esto pueda parecer mucho trabajo para el sistema, es más eficiente que modificar el DOM real directamente, ya que se harían cambios innecesarios y manipularlo sería muy costoso.

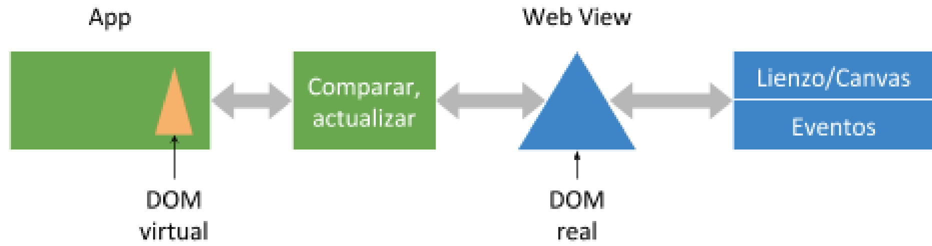

Figura 30: Actualización en programación web

Las aplicaciones creadas con programación híbrida siguen una solución parecida pero, en lugar de utilizar el DOM, manipulan los elementos nativos de la plataforma móvil (figura 31). Para ello, se crea un árbol virtual con la información de los elementos nativos para realizar las modificaciones necesarias. A continuación, haciendo uso del *puente*, comparan y actualizan los elementos nativos reales realizando el menor número de cambios posibles y, finalmente, el sistema actualiza el *lienzo*.

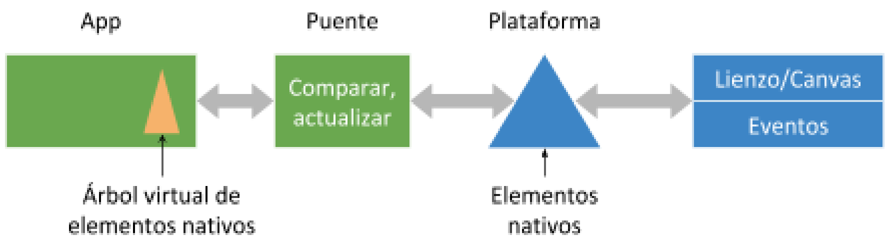

Figura 31: Actualización en programación híbrida

Aunque esta solución es eficiente para la mayoría de los casos, el rendimiento bajará cuantos más elementos sean mostrados y actualizados a la vez. Para solucionarlo, Flutter propone una alternativa diferente a las dos anteriores, que consiste en no utilizar elementos nativos de la plataforma (figura 32). Lo que hace Flutter, gracias a sus propios Widgets, es construir su propio árbol de elementos y "pintarlos" directamente en el *lienzo* sin tener que realizar costosas actualizaciones ni hacer uso del puente.

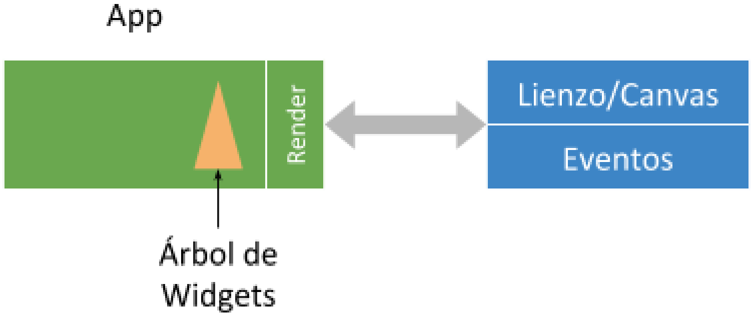

Figura 32: Actualización en Flutter

Otra de las ventajas que ofrece Flutter es que se basa en programación declarativa. Esto conlleva que los estados por los que pueda pasar la aplicación deben estar definidos de antemano. Gracias a esto, y si los estados están bien definidos, la aplicación no debería pasar por situaciones de error provocadas por errores inesperados. En cambio, la programación imperativa permite situaciones en las que diferentes estados pueden llegar a superponerse (por ejemplo, diferentes mensajes de error que aparecen a la vez cuando no deberían hacerlo). Otros *frameworks* que utilizan programación declarativa son, por ejemplo, *React Native* y, a partir de la conferencia *WWDC* de junio de 2019, *SwiftUI* (Apple ha cambiado su framework de desarrollo nativo para pasar a utilizar programación declarativa[16]). A raíz de utilizarse la programación declarativa, la gestión de estados de la aplicación se convierte en una parte fundamental del desarrollo con Flutter. La interfaz gráfica de usuario dependerá en todo momento del valor que se le dé a los estados[17].

Los componentes que incluye Flutter son la *plataforma Dart*, *Flutter Engine* y la *librería Foundation*.

Plataforma Dart: es el lenguaje en el que se escriben las aplicaciones Flutter. Es un lenguaje propiedad de Google y en un principio estaba pensado para programación web. Una de las grandes ventajas de Dart es su compatibilidad con hot reload. Esta función permite ver al momento el resultado de los cambios que se realizan en el código.

Flutter Engine: proporciona procesamiento a bajo nivel y está escrito principalmente en C++. Hace de interfaz entre el desarrollo Flutter y el SDK de los sistemas operativos (Android e iOS en este caso). Implementa también algunas librerías imprescindibles para el desarrollo como pueden ser las interfaces de entrada salida, conexiones de red, animaciones…

[16] Artículo sobre SwiftUI:
https://www.hackingwithswift.com/articles/191/swiftui-lets-us-build-declarative-user-interfaces-in-swift
[17] Más información sobre el manejo de estados:
https://flutter.dev/docs/development/data-and-backend/state-mgmt/intro

Librería Foundation: proporciona clases y funciones básicas que se utilizan para construir las aplicaciones, como las APIs para comunicarse con el dispositivo. Incluye *Widgets*[18] utilizados para construir las interfaces gráficas en Flutter. Existen dos conjuntos de Widgets para construir las interfaces de la aplicación: *Material Widgets* que tienen el diseño de *Material Design* y *Cupertino Widgets* que tienen el diseño de Apple.

Aparte de los Widgets y funciones que vienen integradas por defecto en Flutter, se pueden integrar paquetes extra al desarrollo. Estos paquetes proporcionan nuevos Widgets y funcionalidades que pueden aumentar la funcionalidad de la aplicación. En la página oficial de paquetes[19] pueden encontrarse tanto paquetes creados por el equipo oficial de Flutter como otros creados por terceros.

Además, Flutter puede integrarse en desarrollos ya existentes que hayan sido realizados de forma nativa, por lo que si ya existe una aplicación, no hace falta volver a escribirla en Flutter.

5.2 Firebase

Figura 33: Logo de Firebase

Firebase[20] es una plataforma desarrollada por Google que reúne diferentes servicios para incorporar en aplicaciones web y móviles. Estos servicios se centran en dotar a las aplicaciones de funcionalidades en la nube, como bases de datos, funciones web o almacenamiento online. Al ser una plataforma desarrollada por Google, de la misma forma que Flutter, la integración de los servicios ofrecidos por Firebase es sencilla y precisa de poco tiempo de desarrollo. Dentro del repositorio de paquetes de Dart se pueden encontrar paquetes de integración de Firebase desarrollados por el equipo oficial. Sus servicios se agrupan en cuatro categorías: *Desarrollo*, *Calidad*, *Analíticas* y *Crecimiento*.

Dentro de **Desarrollo**[21] se encuentran los siguientes servicios: *Cloud Firestore*, *Kit de AA* (Aprendizaje Automático), también llamado *ML Kit* y en fase Beta, *Cloud Functions*, *Authentication*, *Hosting*, *Cloud Storage* y *Realtime Database*. Entre éstos, se utilizan en el proyecto *Cloud Firestore*, *Cloud Functions*, *Authentication* y *Cloud Storage*.

Cloud Firestore, "Almacena y sincroniza datos entre usuarios y dispositivos a escala global mediante una base de datos NoSQL alojada en la nube. Cloud Firestore te ofrece sincronización en vivo, soporte sin conexión y consultas de datos eficaces. Su integración con otros productos de Firebase te permite compilar aplicaciones verdaderamente sin servidores."[22]. Este servicio se utiliza

[18] Catálogo de *Widgets*: https://flutter.dev/docs/development/ui/widgets
[19] Página de paquetes Dart: https://pub.dev/
[20] Web de Firebase: https://firebase.google.com
[21] Servicios de Desarrollo: https://firebase.google.com/products/?hl=es-419#develop-products
[22] Cloud Firestore: https://firebase.google.com/products/firestore/?hl=es-419

para dar soporte al modelo de datos. La base de datos de Cloud Firestore es de tipo documental, como se explica en el punto 5.2, y se organiza en colecciones.

Cloud Functions, "Extiende tu app con código de back-end personalizado sin necesidad de administrar ni escalar tus propios servidores. Las funciones pueden activarse con eventos que emiten los productos de Firebase, los servicios de Google Cloud o terceros, por medio de webhooks."[23]. Este servicio se utiliza para dar acceso al servicio de notificaciones mediante un enlace.

Authentication, "Maneja tus usuarios de una forma simple y segura. Firebase Auth ofrece múltiples métodos de autenticación, incluyendo email y contraseña, proveedores de terceros como Google o Facebook, y utilizando tu cuenta existente en el sistema directamente. Construya su propia interfaz o haga uso de nuestra UI personalizable y de código abierto"[24]. Este servicio se utiliza para permitir a los usuarios iniciar sesión en la red social sin tener que implementar un sistema de autenticación desde cero. Se hace uso de los métodos de autenticación de Facebook, Twitter y Google.

Cloud Storage, "Almacena y envía contenido generado por usuarios como imágenes, audio, vídeo con el servicio de almacenamiento de objetos potente, simple y de bajo coste construido por la escala Google. El SDK de Firebase para Cloud Storage añado seguridad Google a las cargas y descargas de archivos de tus aplicaciones Firebase, sin importar la calidad de la red."[25]. Este servicio se utiliza para almacenar tanto las imágenes de perfil de los usuarios como las imágenes que acompañan a los eventos.

Dentro de **Calidad**[26] se encuentran los siguientes servicios: *Crashlytics*, *Performance Monitoring* y *Test Lab*. Entre éstos, se utilizan en el proyecto *Cloud Firestore*, *Cloud Functions*, *Authentication* y *Cloud Storage*. Estos servicios ofrecen estadísticas sobre el rendimiento y la estabilidad de la app. La inclusión de estos servicios no se encuentra dentro del alcance de este Trabajo de Fin de Grado.

Dentro de **Analíticas** se encuentran servicios que ofrecen información del uso que dan los usuarios a la aplicación. Como la publicación de la aplicación producto de este Trabajo de Fin de Grado está dentro de las exclusiones, no se ha incluido ningún servicio de analítica de comportamiento de usuarios.

Dentro de **Crecimiento**[27] se encuentran los siguientes servicios: *In-App Messaging* en fase Beta, *Google Analytics*, *Predictions*, *A/B Testing* en fase Beta, *Cloud Messaging*, *Remote Config* y *Dynamic Links*. Entre éstos, se utilizan en el proyecto *Cloud Messaging*, y *Dynamic Links*.

Cloud Messaging, "Envía mensajes y notificaciones a usuarios en todas las plataformas -Android, iOS y web- de manera gratuita. Los mensajes pueden ser enviados a un dispositivo concreto, a grupos de dispositivos o a segmentos de usuarios concretos. Firebase Cloud Messaging (FCM) escala hasta las aplicaciones más grandes, enviando cientos de miles de millones de mensajes al día."[28]. Este servicio se utiliza para enviar las notificaciones a los usuarios.

[23] Cloud Functions: https://firebase.google.com/products/functions/?hl=es-419
[24] Authentication: https://firebase.google.com/products/auth/?hl=es-419
[25] Cloud Storage: https://firebase.google.com/products/storage/?hl=es-419
[26] Servicios de Calidad: https://firebase.google.com/products/?hl=es-419#quality-products
[27] Servicios de Crecimiento: https://firebase.google.com/products/#grow-products
[28] Cloud Messaging: https://firebase.google.com/products/cloud-messaging/

Dynamic Links, “Utiliza Dynamic Links para ofrecer una experiencia de usuario personalizada en iOS, Android y en la web. Puedes usarlos para llevar de la web a partes concretas de la aplicación, para que tus usuarios puedan compartir, campañas sociales y de márketing y más. Dynamic Links te ofrece atributos para entender mejor el crecimiento móvil.”[29]. Este servicio se utiliza para que los usuarios puedan compartir mediante un enlace los eventos de la aplicación.

Para utilizar estos servicios, además de los dos planes de pago, Firebase ofrece un plan gratuito con límites diarios de uso de recursos. Estos límites son suficientes para la fase de desarrollo inicial.

5.4 Algolia

Figura 34: Logo de Algolia

Algolia es un servicio de búsqueda *full-text*. Las búsquedas *full-text* permiten hacer búsquedas más avanzada en las bases de datos. Además, es tolerante a errores ortográficos. Es decir, el sistema supone lo que el usuario quiere escribir para mostrar los resultados. También funciona introduciendo dos palabras para buscarlas en diferentes posiciones del texto o en diferentes campos de los documentos.

Para buscar los datos, Algolia necesita tenerlos almacenados en su propia base de datos para crear índices. Por lo tanto, parte del contenido de la base de datos original está duplicada. No precisa de la base de datos completa sino de los campos que se quieran incluir en las búsquedas, además del identificador de los documentos, para poder asociarlos con los de la base de datos original.

Algolia ofrece un plan gratuito que incluye un uso limitado de la herramienta, pero que es suficiente para la fase de desarrollo inicial.

[29] Dynamic Links: https://firebase.google.com/products/dynamic-links/

6. Implementación

En este punto se expone el proceso de implementación seguido para realizar este proyecto. Se divide en dos apartados: el primero explica cómo se ha llevado a cabo la implementación de la aplicación utilizando Flutter y el segundo explica cómo se han implementado las funciones que se alojan en el servicio Cloud Firestore.

6.1 Aplicación Flutter

El primer paso a seguir para comenzar a construir una aplicación utilizando Flutter es configurar el entorno de desarrollo. Para ello, debemos elegir el entorno de programación que queramos utilizar. Flutter es compatible con los entornos Android Studio, IntelliJ y Visual Studio Code y el utilizado en este proyecto es Android Studio. A continuación, podemos seguir los pasos indicados en la documentación oficial de Flutter para instalar y configurar las herramientas necesarias. Además, con la instalación inicial de Flutter, se pone a disposición un comando para conocer el estado de la instalación y saber si se ha hecho correctamente:

```
> flutter doctor
```

Este comando nos dará información de todas las instalaciones y versiones que tenemos y avisará si encuentra algún problema o si falta algún componente por instalar. Se puede ver el resultado de ejecutar el comando en la figura 35.

```
[✓] Flutter (Channel stable, v1.5.4-hotfix.2, on Mac OS X 10.14.5 18F132, locale
    es-ES)

[✓] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
[✓] iOS toolchain - develop for iOS devices (Xcode 10.2.1)
[✓] Android Studio (version 3.4)
[✓] IntelliJ IDEA Ultimate Edition (version 2018.3.1)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.
```

Figura 35: Resultado de ejecutar flutter doctor

Una vez terminada la configuración, se puede crear un nuevo proyecto utilizando los editores o mediante un comando.

```
> flutter create myapp
```

Tras este proceso se crea un directorio en el que se inicializan todos los ficheros necesarios para desarrollar el proyecto. La estructura que ha seguido mi proyecto nace del directorio inicial y puede verse en la figura 36.

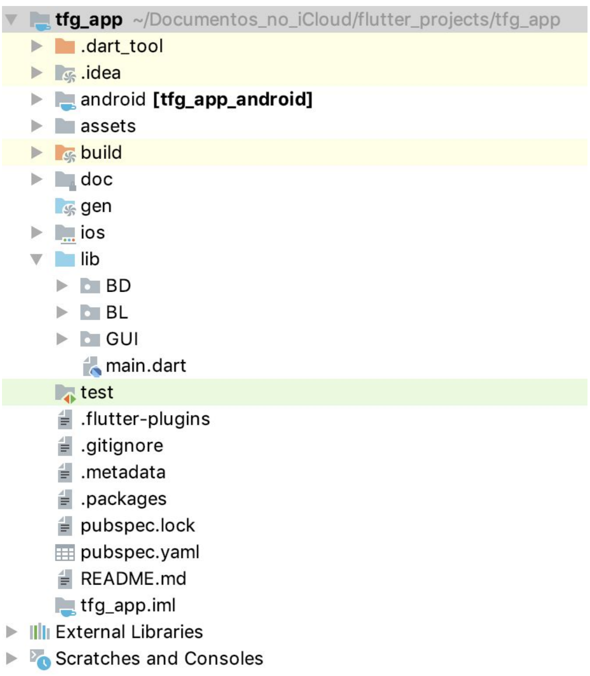

Figura 36: Directorios de la aplicación

El directorio en el que se encuentran todos los ficheros con la implementación es *lib*. Dentro se dividen los ficheros en tres subdirectorios: uno con el acceso a los datos (*BD*), otro con la lógica de negocio (*BL*) y otro con los elementos gráficos (*GUI*). Otros directorios importantes en los proyectos de Flutter son: *android*, donde se guardan los ficheros relacionados con la versión Android, como los ficheros de configuración de la aplicación, ficheros con estilos, el fichero de configuración de Firebase para Android y etc.; *assets*, donde se encuentran los ficheros que se incluyen en el proyecto como imágenes o fuentes de datos (json, txt…) e *ios*, donde se guardan los ficheros relacionados con la versión iOS, como los ficheros de configuración de la aplicación, ficheros de compilación de iOS, ficheros de XCode, el fichero de configuración de Firebase para iOS y etc.

Entre los ficheros fundamentales se encuentra *main.dart*, dentro de *lib*, en el que se incluye el código de inicio de la aplicación. Dentro de este fichero se encuentra el *Widget MaterialApp*[30] que recoge información de configuración de la aplicación como puede verse a continuación:

```
MaterialApp(
   builder: (context, child) => MediaQuery(
       data: MediaQuery.of(context).copyWith(alwaysUse24HourFormat: true),
       child: child),
   title: 'EkitApp',
   locale: Locale(_languageCode, _countryCode),
   localizationsDelegates: [
     AppLocalizationsDelegate(),
     GlobalMaterialLocalizations.delegate,
```

[30] MaterialApp: https://api.flutter.dev/flutter/material/MaterialApp-class.html

```
    GlobalWidgetsLocalizations.delegate,
    const FallbackCupertinoLocalizationsDelegate(),
  ],
  supportedLocales: [Locale('en', 'US'), Locale('es', 'ES')],
  debugShowCheckedModeBanner: false,
  theme: _getTheme(_themeMode),
  home: home);
```

En este caso, aunque *MaterialApp* tiene más argumentos, los importantes son: *title*, *locale*, *localizationsDelegates*, *supportedLocales*, *theme* y *home*. El argumento ***title*** especifica el nombre que se la va a dar a la aplicación. El argumento ***locale*** especifica el idioma que va a ser utilizado. El argumento ***localizationDelegates*** precisa de una lista de clases que proporcionan traducciones. En este ejemplo, el primer elemento de la lista es el proveedor principal de traducciones y el último elemento hace que el menú de copiar y pegar en iOS pueda visualizarse correctamente. El argumento ***supportedLocales*** precisa de una lista de idiomas que soporta la aplicación. El argumento ***theme*** especifica el estilo que va a seguir la aplicación. Se pueden cambiar colores, fuentes, tamaños de texto... Por último, el argumento ***home*** especifica el primer *Widget* que se va a mostrar en la aplicación.

Este fichero *main.dart* es el primero en ejecutarse al iniciar esta aplicación. Éste ejecuta su función *main()* que carga una serie de parámetros de configuración para mostrar la aplicación con los valores que pueden variar, como el idioma o el modo oscuro o claro. Éstos se cargan de *shared preferences* utilizando el paquete *shared_preferences*[31]. También se comprueba si el usuario había iniciado sesión y, en caso afirmativo, carga dicho usuario.

```
void main() async {
 SharedPreferences prefs = await SharedPreferences.getInstance();
 bool themeMode = (prefs.getBool('themeMode') ?? false);
 prefs.setBool('themeMode', themeMode);
 String languageCode = (prefs.getString('languageCode') ?? 'en');
 String countryCode = (prefs.getString('countryCode') ?? 'US');
 prefs.setString('languageCode', languageCode);
 prefs.setString('countryCode', countryCode);
 final Auth auth = new Auth();
 FirebaseUser firebaseUser = await auth.getCurrentUser();
 if (firebaseUser != null) {
   await UserBL.init(firebaseUser.uid);
 }
 runApp(new MyApp(
   themeMode: themeMode,
   languageCode: languageCode,
   countryCode: countryCode,
 ));
}
```

En esta función puede verse cómo se cargan los valores del modo del tema y el idioma y a continuación se vuelven a almacenar. Esto se hace por si los valores obtenidos fueran nulos (es decir,

[31] Paquete *shared_preferences*: https://pub.dev/packages/shared_preferences

si la aplicación se ejecuta por primera vez). Como las funciones de *set* son asíncronas, se ejecutan sin detener el flujo de ejecución, por lo que el resto de la función sigue ejecutándose mientras los valores se almacenan en *shared preferences*.

Finalmente, la función ejecuta *runApp* que tiene como parámetro el *Widget* que va a inicializar la aplicación, en este caso, *MyApp*. Este *Widget* se encuentra en el mismo fichero y extiende a la clase *StatefulWidget*. A la hora de crear *Widgets* propios, se debe extender la clase *StatelessWidget*[32] o *StatefulWidget*[33] dependiendo del tipo de *Widget* que queramos crear. *StatelessWidget* debe ser utilizado para crear *Widgets* que no vayan a ser modificados una vez creados. Los valores contenidos en estos *Widgets* no pueden ser modificados. Sin embargo, esto no implica que no puedan modificarse de otra manera. Por ejemplo, el *Widget Text()* extiende la clase *StatelessWidget* pero el texto que muestra puede modificarse ya que el parámetro de entrada de este *Widget* puede ser modificado si se utiliza dentro de un *Widget* que extienda *StatefulWidget*.

Por otro lado, *StatefulWidget* debe ser utilizado para crear *Widgets* que puedan ser modificados gracias a los cambios de estado. Para crear estos *Widgets* deben crearse dos clases: una que extienda a *StatefulWidget* y otra que extienda a *State<T extends StatefulWidget>*[34]. La primera debe solicitar los parámetros de entrada necesarios y crear el estado llamando a la segunda clase. En la segunda clase, deben crearse variables que son las que sirven para modificar el estado de la aplicación. Pueden crearse tantas como se quieran (si no se añade ninguna variable de cambio de estado, debería utilizarse *StatelessWidget*). Estas variables deben ser inicializadas en el método *initState()*, ejecutado antes del método *build()*. El cambio de estado se realiza cambiando el valor de las variables dentro de la función *setState()*. Estas dos clases proporcionan también otras funciones que se ejecutan durante el ciclo de vida de cada *Widget*, como la función *didUpdateWidget(covariant T oldWidget)* que se ejecuta cuando el *Widget* cambia su configuración y proporciona el estado del *Widget* anterior.

MyApp, por su parte, comienza estableciendo los parámetros de entrada que va a tener y que, en este caso, son requeridos (esa es la razón de que se utilice *@required* y la comprobación mediante el *assert*). Al finalizar la ejecución, crea el estado correspondiente.

```
class MyApp extends StatefulWidget {
 MyApp(
     {@required this.themeMode,
     @required this.languageCode,
     @required this.countryCode})
     : assert(
           themeMode != null && languageCode != null && countryCode != null);
 bool themeMode;
 String languageCode;
 String countryCode;

...
 _MyAppState createState() => _MyAppState();
}
```

Adicionalmente, este *Widget* en concreto define dos métodos estáticos para cambiar su estado desde otras pantallas. Esto se utiliza para poder cambiar tanto entre los modos oscuro y claro como

[32] Documentación sobre StatelessWidget: https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html

[33] Documentación sobre StatefulWidget: https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html

[34] Documentación sobre State: https://api.flutter.dev/flutter/widgets/State-class.html

para cambiar el idioma. A modo de ejemplo se muestra la función que cambia el modo de la aplicación:

```
static void changeTheme(BuildContext context) {
 _MyAppState state = context.ancestorStateOfType(TypeMatcher<_MyAppState>());
 state.setState(() {
   state._themeMode = !state._themeMode;
 });
 SharedPreferences.getInstance().then((prefs) {
   prefs.setBool('themeMode', state._themeMode);
 });
}
```

Esta función busca el estado de *MyApp*, el cual se mantiene durante la ejecución de la aplicación, y modifica su estado desde fuera de la clase, donde habitualmente se modifica el estado. La clase que extiende *State<MyApp>* está implementada de la siguiente manera:

```
class _MyAppState extends State<MyApp> {
 /// false = light, true = dark
 bool _themeMode;
 String _languageCode;
 String _countryCode;

 initState() {
   super.initState();
   _themeMode = widget.themeMode;
   _languageCode = widget.languageCode;
   _countryCode = widget.countryCode;
 }

 Widget build(BuildContext context) {}
}
```

En el método *initState()* se puede ver cómo se accede a los valores de los parámetros de entrada del *Widget* utilizando el objeto *widget*. El método *build()* devuelve el *Widget* que va a establecer la configuración de la aplicación y va a comenzar a mostrar el contenido. Dicho *Widget* es *MaterialApp* que tiene como uno de sus parámetros home, donde se indica qué otro *Widget* va a mostrarse en la interfaz ya que *MaterialApp* no muestra ningún contenido como tal. En el caso en el que el usuario ya se encuentre registrado, la interfaz que se muestra es *MainGUI*, si no, *LogInGUI*.

Se puede observar cómo algunos nombres de variables o métodos tienen un guión bajo delante. Esto se hace para indicar que dicho elemento es privado ya que Flutter no tiene palabras reservadas como, por ejemplo, *private*, *protected* o *public* en Java.

Otro fichero fundamental es *pubspec.yaml*, en la raíz del proyecto, en el que se describen la mayoría de parámetros generales de configuración de la aplicación. Los apartados más importantes de este fichero son *dependencies* y *assets*, aunque también tiene apartados para especificar el nombre, descripción de la aplicación y su versión. Bajo *dependencies* se escriben los paquetes que se

quieran incluir en el proyecto y su versión. Estos paquetes pueden encontrarse en el repositorio de paquetes de Dart[35]. A continuación se pueden ver las dependencias incluidas en este proyecto:

```
dependencies:
  firebase_auth: ^0.11.0
  flutter_twitter_login:
    git: git://github.com/eudangeld/flutter_twitter_login.git
  flutter_facebook_login: ^2.0.0
  google_sign_in: ^4.0.1+3
  cloud_firestore: ^0.11.0+1
  image_picker: ^0.6.0+3
  firebase_storage: ^3.0.0
  photo_view: ^0.3.3
  cached_network_image: ^0.8.0
  flutter_native_image:
    git: https://github.com/btastic/flutter_native_image.git
  flutter_slidable: ^0.4.9
  flutter_inner_drawer: ^0.2.5
  shared_preferences: ^0.5.2
  percent_indicator: ^1.0.16
  algolia: ^0.1.5
  image_cropper: ^1.0.1
  firebase_messaging: ^5.0.1+1
  flushbar: ^1.5.0
  liquid_pull_to_refresh: ^1.1.0
  vibrate: ^0.0.4
  flutter_datetime_picker: ^1.2.0
  flutter_linkify: ^2.1.0
  url_launcher: ^5.0.2
  firebase_dynamic_links: ^0.4.0
  share: ^0.6.1+1
  provider: ^2.0.1+1
  carousel_slider: ^1.3.0
  flutter_localizations:
    sdk: flutter
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2
```

Bajo *assets* se definen las rutas de los ficheros (imágenes por ejemplo) que se quieren incluir en el proyecto. Como ejemplo, este proyecto incluye los siguientes:

```
assets:
  - assets/logo.png
  - assets/facebook-icon.png
  - assets/twitter-icon.png
  - assets/google-icon.png
  - assets/icon-placeholder.png
```

[35] Repositorio de paquetes: https://pub.dev/packages/

A continuación se describe cómo ha sido implementada cada capa de la arquitectura de la aplicación móvil.

6.1.1 Implementación de la GUI

La capa de la interfaz gráfica de usuario está compuesta por *Widgets* que forman las pantallas. Generalmente, cada fichero contiene un *Widget* (o varios relacionados entre sí). También hay dos directorios, uno para *Widgets* personalizados que no completan una pantalla y otro en el que están los ficheros que forman parte de las pantallas a las que se accede mediante la navegación inferior en la pantalla de inicio. Se presenta cada pantalla y cada *Widget* en este punto. Exceptuando *LogInGUI* y *MainGUI*, los fragmentos de implementación de los *Widgets* se pueden encontrar en el Anexo 3. Además, en los anexos 4, 5 y 6 se explica cómo registrar la aplicación en Firebase, cómo realizar las conexiones con la base de datos Firestore y cómo incluir la funcionalidad multi-idioma.

LogInGUI

Dentro del fichero *log_in_gui.dart*. Extiende la clase *StatefulWidget.* El parámetro de entrada requerido en este y otros *Widgets* es *isIos*, utilizado para mostrar diferentes *Widgets* o actuar de manera acorde a cada sistema operativo móvil. Se inicializa la variable *auth* de la clase *Auth* (módulo de autenticación en el punto 6.1.2) para realizar el inicio de sesión con los servicios disponibles: *Twitter*, *Facebook* y *Google*. Esta clase declara valores finales para la definición de elementos gráficos comunes. La variable *_scaffoldKey* se utiliza para mostrar un mensaje de error si ocurre algún problema en el momento del inicio de sesión, además de una variable de estado, *_previews*, donde se guardan los eventos que se van a mostrar en la previsualización. Las funciones adicionales que se definen son: *_twitterLogin()*, *_facebookLogin()* y *_googleLogin()* para realizar el inicio de sesión, las cuales llamarán a *auth* para realizar las acciones necesarias para llevar a cabo la autenticación correspondiente; *_signUpButton()* para generar los botones que se muestran en la interfaz gráfica y *_changeToFeed()* para navegar a la pantalla principal. Además también se incluye la función *_loadPreviews()* que se encarga de realizar una llamada a la lógica de negocio para obtener algunos eventos para la previsualización.

```
class LogInGUI extends StatefulWidget {
 LogInGUI({@required this.isIos}) : assert(isIos != null);
 Auth auth = new Auth();
 bool isIos;
 State<StatefulWidget> createState() => new _LogInGUIState();
}

class _LogInGUIState extends State<LogInGUI> {
 final _scaffoldKey = GlobalKey<ScaffoldState>();
 final EdgeInsets _padding = new EdgeInsets.all(40.0);
 final double _iconWidth = 25.0;
 List<Event> _previews;

 void initState() {
  super.initState();
  _previews = [];
  _loadPreviews();
 }
```

```
  Widget build(BuildContext context) {}
  void _twitterLogin() {}
  void _facebookLogin() {}
  void _googleLogin() {}
  Widget _signUpButton(Image icon, String text, Color buttonColor,
      Color textColor, Function login) {}
  void _changeToFeed() {}
  Future<void> _loadPreviews() async {}
}
```

MainGUI

Dentro del fichero *main_gui.dart*. Extiende la clase *StatefulWidget*. Esta clase incluye *WidgetBindingObserver* que proporciona funciones que son ejecutadas con los cambios de ciclo de vida de la aplicación. Una de estas funciones se llama *didChangeAppLifecycleState()* y se puede ver en el código que se utiliza para el manejo de enlaces dinámicos.

Tiene un valor final, *_innerDrawerKey*, que sirve para abrir y cerrar el menú lateral. Las variables de estado son: *_body*, que almacena el Widget que sirve de cuerpo de la pantalla seleccionada, *_title*, que mantiene el título de dicha pantalla, y *_currentIndex*, que contiene el número de la pantalla seleccionada. Estas variables son necesarias para cambiar entre las pantallas de la barra de navegación inferior que establece este *Widget*. También se crea una variable para manejar las notificaciones, *_firebaseMessaging*.

Las funciones *_retrieveDynamicLink()* y *firebaseCloudMessaging_Listeners()* se utilizan, respectivamente, para manejar la apertura de la aplicación cuando se hace mediante un enlace dinámico y una notificación. La función *_goTo()* sirve para cambiar de pantalla a la del evento correspondiente si se entra por el enlace o notificación. A continuación se encuentran la función *build()* y las funciones *_getNewEventButton()*, *_bottomItems()*, *_getSelectedBody()*, *_getTitle()* y *_onTabSelected()* que construyen la interfaz de usuario y manejan el cambio de pantalla con los botones de la barra de navegación inferior. Para terminar, la función *_open()* se utiliza para abrir el menú lateral y se ejecuta al pulsar el botón izquierdo de la barra superior. Las diferentes pantallas por las que se puede navegar gracias a esta barra se encuentran dentro del directorio *MainGUI* (no confundir con la clase) descritos más adelante.

```
class MainGUI extends StatefulWidget {
  MainGUI({@required this.isIos}) : assert(isIos != null);
  bool isIos;
  State<StatefulWidget> createState() => new _MainGUIState();
}

class _MainGUIState extends State<MainGUI> with WidgetsBindingObserver {
  final GlobalKey<InnerDrawerState> _innerDrawerKey =
      GlobalKey<InnerDrawerState>();

  Widget _body;
  Widget _title;
  int _currentIndex;

  FirebaseMessaging _firebaseMessaging = FirebaseMessaging();
```

```
  void initState() {
    super.initState();
    _body = MainFeedGUI(isIos: widget.isIos);
    _currentIndex = 0;
    firebaseCloudMessaging_Listeners();
    WidgetsBinding.instance.addObserver(this);
  }

  void didChangeAppLifecycleState(AppLifecycleState state) {
    if (state == AppLifecycleState.resumed) {
      _retrieveDynamicLink();
    }
  }

  Future<void> _retrieveDynamicLink() async {}
  void firebaseCloudMessaging_Listeners() {}
  Future<void> _goTo(Map<String, dynamic> message) async {}
  Widget build(BuildContext context) {}
  Widget _getNewEventButton() {}
  List<BottomNavigationBarItem> _bottomItems() {}
  Widget _getSelectedBody(int index) {}
  Widget _getTitle(int index) {}
  ValueChanged<int> _onTabSelected(int index) {}
  void _open() {}
}
```

CreateEventFirstGUI y EditEventFirstGUI

Dentro de los ficheros *create_event_first_gui.dart* y *edit_event_first_gui.dart.* Las pantallas de crear y editar evento muestran la misma información con la diferencia de que en las de editar los campos de texto están completados, por lo tanto sólamente se muestra el *Widget* de creación. Éste extiende la clase *StatefulWidget* y define variables para guardar los contenidos del evento que se está creando (o editando). Se definen variables de estado que declaran casos de error en caso de que alguno de los campos tenga valores incorrectos. Algunas de las funciones implementadas sirven para construir el *Widget* (*_textField()*, *_dropDownMenuItems()* y *_buttons()*), otras para obtener las imágenes y añadirlas al evento (*_getImage()*, *_compressImage()*, *_getImageGallery()*, *_getImageCamera()* y *_deleteImage()*) y finalmente las últimas para validar los datos rellenados y continuar creando el evento.

CreateEventSecondGUI y EditEventSecondGUI

Dentro de los ficheros *create_event_second_gui.dart* y *edit_event_second_gui.dart.* Una vez se termina de rellenar los primeros datos del evento, se pasa a la segunda pantalla. Este *Widget* extiende la clase *StatefulWidget* y requiere de los datos del evento que se han obtenido en la pantalla anterior. Además, define variables para guardar el resto de los contenidos del evento que se está creando (o editando). También se definen variables de estado que declaran casos de error en caso de que alguno de los campos tenga un valor incorrecto. Algunas de las funciones implementadas sirven para construir el *Widget* (*_textField()* y *_dropDownMenuItems()*), otras para seleccionar la fecha y añadirla (*_getDate()*, y *_selectDate()*) y finalmente las últimas para validar los datos rellenados y publicar el evento.

DeleteEventGUI

Dentro del fichero *delete_event_gui.dart*. Extiende la clase *StatefulWidget*. Tiene una variable de estado *_loading* que declara un estado en el que la pantalla carga para borrar el evento. La función *_deleteEvent()* se invoca al pulsar el botón de confirmación de borrado.

ViewEventGUI

Dentro del fichero *view_event_gui.dart*. Extiende la clase *StatefulWidget*. Requiere del objeto del evento con la información a mostrar. Define las siguientes variables de estado: *_attends* para saber el estado de la asistencia al evento, *_comment* para el texto del campo de escritura del comentario, *_controller* para controlar dicho campo de texto, *_comments* con la lista con los comentarios a mostrar, *_saved* para saber si el usuario ha guardado o no el evento y *_heroRandom* para establecer un número aleatorio para hacer más fluida la transición al abrir la pantalla de visualizar (*_ImageGUI*) [36] la imagen.

Por otro lado, las funciones que tiene esta clase sirven para guardar el evento, para cambiar la asistencia, para abrir la imagen, para guardar el comentario, para editar el evento y para borrar el evento.

El *Widget* ***_StarRating*** muestra la puntuación del evento junto a tres estrellas para puntuarlo. Requiere del mismo evento y extiende la clase *StatefulWidget*. Las dos variables de estado almacenan la puntuación y el voto que se le ha dado al evento. Además, contiene dos funciones, una para obtener la votación actual y otra para cambiar el voto.

Por su parte, el *Widget* ***_ImageGUI*** extiende la clase *StatelessWidget*, por lo que no tiene variables de estado. Este *Widget* muestra la imagen del evento en una pantalla por separado (abre una pantalla nueva).

Finalmente, el *Widget* ***_CommentWidget*** extiende la clase *StatefulWidget* y muestra uno de los comentarios (cada comentario del evento crea uno de estos elementos). Sus variables de estado definen la puntuación y el voto que se le ha dado al comentario. Las funciones que contiene son para obtener la votación actual y para cambiar el voto.

ViewEventsByCampusGUI

Dentro del fichero *view_events_by_campus_gui.dart*. Extiende la clase *StatefulWidget* y tiene una variable de estado en la que se almacenan los eventos a mostrar. Requiere del número del campus que se va a buscar en los eventos. Tiene una función que sirve para actualizar la lista de los eventos que se muestren.

ViewEventsByTagGUI

Dentro del fichero *view_events_by_tag_gui.dart*. Extiende la clase *StatefulWidget* y tiene una variable de estado en la que se almacenan los eventos a mostrar. Requiere de el número de la categoría que se va a buscar en los eventos. Tiene una función que sirve para actualizar la lista de los eventos que se muestran.

---

[36] Información sobre animaciones Hero: https://flutter.dev/docs/development/ui/animations/hero-animations

ViewProfileGUI

Dentro del fichero *view_profile_gui.dart*. Extiende la clase *StatefulWidget*. Requiere del identificador del usuario que va a ser mostrado en la pantalla. Tiene variables de estado para guardar la información del usuario que se está mostrando, un booleano para saber si el usuario actual sigue al visualizado y dos listas con los eventos a mostrar: aquellos a los que va a asistir o ha asistido y los que ha creado. Se incluyen también dos variables que guardan la tanda de estas dos listas y una variable más para comprobar si la pantalla está cargando. Las funciones que se incluyen en esta clase sirven para construir la interfaz, para seguir o dejar de seguir al usuario y para editar el perfil en el caso de que sea el perfil del usuario actual. Esta última función dirige a la pantalla *EditProfileGUI*.

EditProfileGUI

Dentro del fichero *edit_profile_gui.dart*. Extiende la clase *StatefulWidget*. Entre sus variables de estado se encuentran las que almacenan tanto la imagen de perfil del usuario como el nombre de usuario, ya que son los dos campos que se pueden actualizar. También hay una variable de estado para declarar el estado de carga de la pantalla.

Carpeta MainGUI

Bajo este directorio, ubicado bajo *GUI*, se encuentran las cuatro pantallas a las que se puede acceder mediante los botones de navegación inferior dentro de la clase *MainGUI: MainFeedGUI, SearchGUI, ViewSavedGUI, ViewNotificationsGUI*.

MainFeedGUI, dentro de *main_feed_gui.dart*, construye la lista para visualizar los eventos en la pantalla inicial. Extiende la clase *StatefulWidget*. Tiene variables de estado para cambiar los eventos a mostrar, los filtros seleccionados y uno para declarar un estado de carga. La inicialización de los eventos se hace cargando los 10 eventos más recientes y no se selecciona ningún filtro. El resto de las funciones sirven para construir la interfaz y obtener los eventos.

SearchGUI, dentro de *search_gui.dart*, construye un buscador para poder encontrar eventos o usuarios. Extiende la clase *StatefulWidget*. Las variables de estado que se utilizan son para almacenar los resultados de la búsqueda, un controlador para poder limpiar el campo de texto y un booleano para cambiar entre la búsqueda de eventos y la de usuarios.

ViewSavedGUI, dentro del fichero *view_saved_gui.dart*, muestra la lista de los eventos guardados por el usuario. Extiende la clase *StatefulWidget*. Las variables de estado albergan la lista de los eventos guardados a mostrar, un booleano para indicar si más eventos están siendo cargados y un número que indica la tanda. Las funciones restantes sirven para construir la interfaz y cargar los eventos guardados.

Finalmente, ***ViewNotificationsGUI***, dentro del fichero *view_notifications_gui.dart,* muestra la lista de las notificaciones recibidas por el usuario. Extiende la clase *StatefulWidget*. La única variable de estado de esta clase almacena la lista de las notificaciones a mostrar. Las funciones definidas se utilizan para construir la interfaz y para cargar las notificaciones.

Carpeta *Widgets*

Para definir algunos de los *Widgets* de las interfaces de usuario comentadas anteriormente, se ha hecho uso de *Widgets* personalizados ubicados en la carpeta *Widgets* dentro de *GUI*.

El primero, ***EventCard***, dentro del fichero *event_card_widget.dart*, contiene el *Widget* con forma de tarjeta que se utiliza durante toda la aplicación para mostrar los eventos. Éste utiliza a su vez otros *Widgets* personalizados que están definidos en el mismo fichero: *_StarRating*, *_ImageGUI* y *_CommentWidget*, tal y como se pueden encontrar en *ViewEventGUI*. Además de el booleano *isIos*, requiere de la información del evento a mostrar. Las variables de estado de esta clase tienen la misma funcionalidad que las de *ViewEventGUI*, excepto *_showAll*, un booleano que indica si debe mostrarse la información completa en la tarjeta o solo un pequeño extracto. Para ello, se implementa también una función, *_expand()*, que cambia el valor de dicha variable.

NotificationWidget, dentro del fichero *notification_widget.dart*, contiene el *Widgets* de las notificaciones. Extiende la clase *StatelessWidget* y requiere de la notificación a mostrar. Define dos funciones, una para navegar al perfil del usuario de la notificación y otra al evento de la notificación.

SearchResultEvent y ***SearchResultUser***, dentro del fichero *search_result_widget.dart*, contiene los *Widgets* para mostrar los resultados de las búsquedas. Ambos extienden la clase *StatelessWidget* y requieren del resultado de la búsqueda a mostrar, sea del evento o del usuario. Generan pequeñas tarjetas con poca información que al pulsar dirigen a la pantalla del evento o al perfil del usuario.

6.1.2 Implementación de la lógica de negocio

La lógica de negocio está compuesta por cuatro módulos que ofrecen funciones para interactuar con la autenticación, los eventos, los usuario y las búsqueda. En este punto se presenta cada uno de estos módulos. Los fragmentos del código pueden encontrarse en el anexo 1.

Autenticación

Este módulo, dentro del fichero *authentication_bl.dart*, provee funciones para autenticar al usuario, para comprobar si el inicio de sesión se ha realizado anteriormente, para obtener el usuario actual y para cerrar sesión. Las tres funciones de inicio de sesión con redes sociales llaman a la función privada *_signIn()* para realizar la autenticación.

Eventos

Este módulo, dentro del fichero *event_bl.dart*, provee funciones para gestionar los eventos de la aplicación. Sirven para crear, modificar, eliminar, puntuar o comentar, entre otras cosas, eventos, además de para obtener listas de eventos. Algunas de estas funciones, como la de puntuar o comentar, llaman al servicio en la nube de las notificaciones para enviar avisos a los usuarios correspondientes. Por último, la función *predictSpam()* hace una llamada a la función en la nube producto de TFG-Elsa para obtener la predicción de si la descripción del evento puede ser SPAM o no. En el caso de serlo, se impide al usuario crear el evento.

Usuarios

Este módulo, dentro del fichero *user_bl.dart*, provee funciones para gestionar los usuarios de la aplicación. Por un lado, este módulo mantiene guardada la información del usuario actual y, por otro lado, ofrece funciones para gestionar otros aspectos de los usuarios como los seguimientos o la actualización de información del perfil. Algunas de estas funciones, como la de seguir a usuarios o la de cambiar la asistencia a los eventos, llaman al servicio en la nube de las notificaciones para enviar avisos a los usuarios correspondientes.

Búsqueda

Este módulo, dentro del fichero *search_bl.dart*, provee dos funciones para realizar las búsquedas de eventos y las de usuarios. Dado un texto, devuelve la lista con los resultados de las búsquedas llamando al servicio de búsqueda *full-text* de Algolia.

6.1.3 Implementación de la persistencia

La capa de la persistencia contiene dos módulos que se encargan de actualizar y obtener los datos que genera la aplicación. Uno de los módulos realiza las conexiones de acceso a datos con los servicios de Firebase (Firestore y Storage) y el otro realiza las búsquedas con los servicios de Algolia. A continuación se detallan las funciones de cada uno de éstos módulos. Los fragmentos del código pueden encontrarse en el anexo 1.

Acceso a datos de Firebase

Este módulo, dentro del fichero *firestore_db.dart*, realiza las conexiones con la base de datos de la aplicación alojada en Firestore. A la hora de insertar imágenes, realiza conexiones con el servicio Storage, donde son almacenadas. Ofrece funciones para gestionar todos los aspectos necesarios de la base de datos: creación, actualización y borrado de eventos y usuarios, obtención de datos…

Búsqueda de Algolia

Este módulo, dentro del fichero *algolia_db.dart*, realiza las conexiones con la base de datos que tiene el servicio Algolia y que permite realizar búsquedas *full-text*. Esta base de datos contiene datos reducidos de los eventos y los usuarios. Este módulo ofrece funciones para crear, actualizar y borrar los datos.

6.2 Funciones Cloud

En este proyecto se han desarrollado dos funciones en la nube utilizando el servicio Cloud Functions de Firebase. Estas funciones deben ser implementadas en local para luego subirlas y así tenerlas disponibles mediante un enlace.

Al crear un proyecto de Cloud Functions se genera la estructura de ficheros que se puede ver en la figura 37.

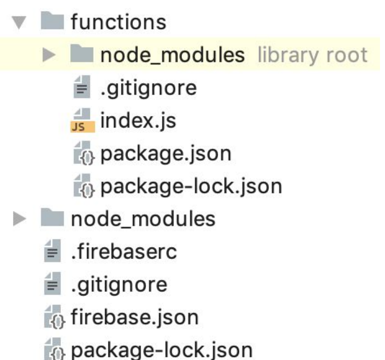

Figura 37: Estructura de ficheros de un proyecto de Cloud Functions

Dentro de la carpeta *functions* se encuentra el fichero *index.js*, que es en el que se implementan las funciones que se quieran subir a Cloud Functions. Desde este fichero podría llamarse a funciones que se encuentren en otro ficheros, pero no ha sido necesario para este proyecto.

Las dos funciones incluidas en este proyecto son: una para enviar una única notificación a un usuario (por lo que si se quisiera enviar más de una, habría que llamar más de una vez a esta función) y otra que se encarga de revisar los eventos que van a celebrarse el día actual (en el que se llame a la función) y enviar notificaciones a cada usuario que ha confirmado la asistencia a dichos eventos. Una vez subidas, estas funciones quedan disponibles mediante enlaces a los que se puede realizar peticiones HTTP.

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
const algoliasearch = require('algoliasearch');

admin.initializeApp(functions.config().firebase);

exports.sendNotification = functions.https.onRequest((req, res) => {});

exports.notifyEvents = functions.https.onRequest((req, res) => {});
```

7. Pruebas

En este capítulo se describen las pruebas que se han llevado a cabo durante el desarrollo del proyecto. Se divide en tres apartados que separan los tres tipos de pruebas que se han realizado. Por un lado, las pruebas con usuarios reales, donde se establecen acciones a realizar. Por otro lado, las pruebas de la aplicación… Finalmente, las pruebas de la integración Smart-Ekitaldi, donde se muestra una batería de pruebas con el resultado esperado, el resultado obtenido y la conclusión de cada prueba.

7.1 Pruebas con usuarios

Se realizan un conjunto de pruebas con usuarios reales para conocer y valorar la experiencia de usuario y recoger posibles ideas para casos de uso o cambios en la interfaz gráfica de usuario. Las personas que realizan las pruebas coinciden con el siguiente perfil: alumnos universitarios de edades comprendidas entre 18 y 30 años.

Las pruebas realizadas son las siguientes:

- Iniciar sesión en la aplicación con uno de los tres métodos posibles.
- Crear un evento.
- Ver los eventos y ver la información completa de uno de ellos.
- Ver la foto de un evento.
- Comentar un evento.
- Puntuar un evento.
- Marcar la asistencia a un evento.
- Cerrar la sesión.

Tras la prueba realizada la primera semana de abril se recogen los siguientes comentarios (se mencionan los más significativos):

1. Les gustaría recibir notificaciones sobre la celebración de los eventos.
2. Les gustaría poder recargar la pantalla del evento (se refiere a poder arrastrar hacia abajo para poder recargar la información del evento y así actualizar los comentarios del mismo).
3. Les gustaría poder tener la posibilidad de puntuar un evento y cambiar la asistencia dentro del mismo.
4. Les gustaría que el botón de asistencia muestre un fondo para que se asemeje más a un botón tradicional (hasta ahora era solo texto sin forma de botón).
5. Les gustaría poder darle a las etiquetas, ya sea del campus o de la categoría, para que se abra una pantalla en la que se muestren eventos que contienen dicha etiqueta.
6. Al crear un evento:

a. Comentan que el selector de fecha debería abrirse al pulsar dentro del campo y no en un botón separado.
b. Les gustaría que el campo de la ubicación mostrara sugerencias al escribir o que se autorellene a medida que el usuario escribe.
c. Comentan que el selector de fecha no es intuitivo, que hay problemas para definir los minutos.

7. Les gustaría disponer de botones para poder borrar, editar y compartir los eventos.
8. Les gustaría poder tener la opción de buscar un evento.
9. Les gustaría poder aplicar filtros, tanto de campus como de categoría, para mostrar eventos.
10. Les gustaría tener un botón para marcar un evento como favorito y así guardar eventos sin confirmar la asistencia.

Teniendo en cuenta los comentarios, se realizaron los siguientes ajustes en la aplicación:

1. Se añaden las notificaciones a la aplicación. Estas notificaciones se reciben cuando un usuario empieza a seguir a otro y cuando se puntúa, comenta o se marca la asistencia a un evento. También se añade una notificación que se envía a los usuarios que marcan la asistencia al evento para recordar que va a ser celebrado ese mismo día.
2. Se cambia el formato de ver toda la información de un evento. Pasa de ser una pantalla separada a expandirse la tarjeta, por lo que la recarga se hace en la misma pantalla en la que se muestran todos los eventos. Sin embargo, para algunos casos se mantiene la pantalla del evento, por lo que se añade la actualización de información.
3. Al igual que en el punto 3, al cambiar el formato de un evento a una tarjeta extendida estas opciones se mantienen. En los casos en los que la pantalla separada se mantiene, se añaden estas opciones.
4. Se sustituye el *Widget* del botón de *FlatButton* a *RaisedButton*.
5. Al pulsar las etiquetas, se muestra una pantalla con los eventos que comparten dicha etiqueta.
6. Se cambian los siguientes aspectos al crear un evento:
    a. Se quita el botón de seleccionar la fecha y el selector de fecha aparece al pulsar el campo de texto de fecha.
    b. Se ha intentado utilizar la API de Google Places, pero sus sugerencias no cumplen con las expectativas.
    c. Se cambia el selector de fecha por uno que muestra los minutos desde el principio.
7. Los botones de borrar y editar evento se añaden. Se muestran al pulsar prolongadamente la tarjeta que contiene el evento. El botón de compartir el evento mediante un enlace se añade a la tarjeta del mismo.
8. Se añade una pantalla de búsqueda, tanto de eventos como de usuarios.
9. Se añade un botón para mostrar y aplicar filtros en la pantalla de inicio.
10. Se añade un botón para guardar eventos y una pantalla para poder verlos.

7.2 Pruebas de la aplicación

Durante el desarrollo del proyecto se han establecido una serie de comportamientos deseados en base a los casos de uso existentes. Estas pruebas se han dividido según las diferentes pantallas que contiene la aplicación:

Pantalla de inicio (figura 13):

Prueba	Resultado esperado
Pulsar el botón de inicio de la barra inferior.	Se muestra la pantalla de inicio con la lista de eventos.
Pulsar el botón de buscar de la barra inferior.	Se muestra la pantalla de búsqueda.
Pulsar el botón de guardados de la barra inferior.	Se muestra la pantalla de guardados con la lista de eventos guardados.
Pulsar el botón de actividad de la barra inferior.	Se muestra la pantalla de actividad con la lista de las notificaciones.
Pulsar el botón de menú de la barra superior.	Se despliega el menú de los ajustes.
Pulsar el botón "+" de añadir evento de la barra superior.	Se muestra la pantalla para comenzar a crear el evento.
Pulsar el botón de mostrar los filtros de la barra superior.	Se muestran los filtros sobre la lista de los eventos.
Pulsar uno o varios filtros.	La lista de los eventos se modifica para mostrar los eventos que contienen la o las etiquetas seleccionadas.
Pulsar el botón de cargar más al final de la lista de eventos.	Se muestran más eventos en la lista. Si no hay más eventos que mostrar, vuelve a aparecer el botón sin modificar la lista.
Arrastrar la lista de eventos de arriba a abajo.	La lista de los eventos se actualiza.

Tabla 5: Pruebas de la pantalla de inicio

Tarjetas de eventos no expandidas (figura 5):

Prueba	Resultado esperado
Pulsar la imagen del evento (si la hubiera).	Se muestra una pantalla nueva con el visualizador de la imagen.
Pulsar la imagen del usuario creador del evento.	Se muestra la pantalla del perfil del usuario creador del evento.
Pulsar la etiqueta del campus.	Se muestra una pantalla con una lista de eventos que compartan el campus.
Pulsar la etiqueta de la categoría.	Se muestra una pantalla con una lista de eventos que compartan la categoría.
Pulsar el botón de guardado.	El botón se oscurece y el evento se guarda en la lista de guardados.
Pulsar el botón de compartir.	Se abre el menú del sistema operativo para poder compartir el evento mediante un enlace.
Pulsar una de las estrellas.	Si se pulsa una estrella no pulsada, se oscurece, cambia el número de la puntuación y se envía una notificación al usuario creador del evento de que se ha puntuado. Si se pulsa una estrella ya pulsada, la puntuación se elimina y la notificación desaparece de actividad.
Pulsar el botón de cambiar la asistencia.	Si antes no ha sido pulsado, cambia el texto a "No asistiré", se guarda el evento en la lista de asistencia y se envía una notificación al usuario creador del evento. Si se pulsa cuando está el texto "No asistiré", cambia el texto a "Asistiré", se elimina el evento de la lista de asistencias y la notificación desaparece de actividad.
Pulsar la tarjeta fuera de los botones.	La tarjeta se expande para mostrar toda la información.
Pulsar la tarjeta prolongadamente.	Si el usuario que pulsa la tarjeta es el creador del evento, se muestra un menú con las opciones de editar y borrar el evento. Si el usuario no es el creador, no ocurre nada.
Pulsar editar evento.	Se muestra la pantalla para editar el evento.
Pulsar borrar evento.	Se muestra una pantalla para confirmar el borrado del evento.

Tabla 6: Pruebas de las tarjetas

Tarjetas de eventos expandidas (figura 6):

Prueba	Resultado esperado
Escribir en la barra de “Añade un comentario…” y pulsar el botón de enviar.	El comentario se añade a la lista de los comentarios y se envía una notificación al creador del evento. Si no se escribe ningún texto, el botón no puede pulsarse.
Arrastrar un comentario de derecha a izquierda.	Se muestran tres estrellas para puntuar el evento.
Pulsar una de las estrellas.	Si se pulsa una estrella no pulsada, se oscurece, cambia el número de la puntuación. Si se pulsa una estrella ya pulsada, la puntuación se elimina.

Tabla 7: Pruebas de las tarjetas expandidas

Menú de ajustes (figura 10):

Prueba	Resultado esperado
Pulsar el botón del perfil.	Se muestra una pantalla con el perfil del usuario.
Pulsar el botón de cambio de modo.	La interfaz cambia a colores claros o colores oscuros y el menú se cierra.
Pulsar el botón de cambio de idioma.	Los textos de la interfaz cambian de idioma y el menú se cierra.
Pulsar el botón de cerrar sesión.	Se cierra la sesión del usuario y se muestra la pantalla de inicio de sesión.

Tabla 8: Pruebas del menú de ajustes

Pantalla de búsqueda (figura 14):

Prueba	Resultado esperado
Escribir en la barra de búsqueda.	El resultado de la búsqueda se actualiza a medida que el usuario escribe.
Pulsar el botón de la cruz al final de la barra de búsqueda.	Si hay algo escrito en la barra de búsqueda, esto se elimina.
Pulsar el botón situado antes de la barra de búsqueda.	Los resultados de la búsqueda cambian entre usuarios y eventos.
Pulsar un resultado de la búsqueda.	Se muestra una pantalla con la información del evento o usuario.

Tabla 9: Pruebas de la pantalla de búsqueda

Pantalla de actividad (figura 16):

Prueba	Resultado esperado
Pulsar una notificación.	Si la notificación es sobre un nuevo seguimiento, se muestra la pantalla del perfil del usuario. SI la notificación es sobre un evento, se muestra la pantalla del evento.
Pulsar la imagen del usuario.	Se muestra la pantalla del perfil del usuario.
Arrastrar la lista de eventos de arriba a abajo.	La lista de los eventos se actualiza.

Tabla 10: Pruebas de la pantalla de actividad

Pantallas de crear un evento (figura 19) y **editar un evento**:

Prueba	Resultado esperado
Seleccionar una imagen.	Se abre un menú en el que se pregunta si se quiere abrir la cámara o la galería del sistema. Al seleccionar la foto, se muestra bajo los botones.
Eliminar la imagen.	Se elimina la imagen seleccionada anteriormente.
Pulsar el selector de etiqueta.	Se muestra una lista con las opciones.
Pulsar siguiente.	Se pasará a la siguiente pantalla si se cumplen con los siguientes requisitos: los campos de título y descripción no deben estar vacíos. Si no se cumple con alguno de los requisitos, se muestra un mensaje de error bajo el campo.
Pulsar el campo de la fecha del evento.	Se muestra un selector de fecha y hora. Si se pulsa confirmar, la fecha se actualiza si la fecha es posterior al día actual.
Pulsar el selector de campus.	Se muestra una lista con las opciones.
Pulsar el botón de crear o editar.	El evento se crea o se modifica y se muestra la pantalla de inicio.

Tabla 11: Pruebas de la pantalla de crear y de editar un evento

Pantallas del perfil de usuario (figura 18):

Prueba	Resultado esperado
Pulsar el botón de seguimiento (visible si el perfil que se está mostrando no pertenece al usuario actual).	Si el usuario actual no seguía al usuario del perfil, se añadirá el usuario actual a los seguidores y se enviará una notificación al usuario del perfil. Si el usuario actual seguía al usuario del perfil, se eliminará al usuario actual de los seguidores y la notificación desaparecerá de actividades en el usuario del perfil.
Pulsar el botón de editar el perfil (visible si el perfil que se está mostrando pertenece al usuario actual).	Se muestra una pantalla para editar la imagen y el nombre del perfil.
Pulsar los botones de “Asiste” o “Eventos creados”.	Se cambia la lista para mostrar los eventos correspondientes.

Tabla 12: Pruebas de la pantalla del perfil de usuario

Estas pruebas habrá que realizarlas a lo largo del ciclo del proyecto para asegurar que el desarrollo realizado cumple con los estándares establecidos.

7.3 Pruebas de integración Smart-Ekitaldi

Las pruebas de integración entre TFG-Elsa y este TFG se han hecho llamando a las funciones desarrolladas por TFG-Elsa desde la aplicación EkitApp. Las funciones son invocadas desde la primera pantalla de crear evento (figura 19) al pulsar sobre el botón “Siguiente”, enviando el texto de la descripción. Sin embargo, para realizar las pruebas se invoca a las funciones de manera manual al pulsar un botón y con unos parámetros concretos.

Los resultados pueden verse en la siguiente tabla:

Datos de entrada	Datos de respuesta esperados	Datos de respuesta obtenidos	Explicación
Datos de un ejemplo que no es SPAM.	200 False	200 False	Al enviar datos correctos, el formato de respuesta es el esperado.
Datos de un ejemplo que es SPAM.	200 True	200 True	
Enviando un ejemplo eliminando un dato de entrada.	Error genérico	500 Internal Server Error	En el alcance del proyecto no se incluye la captura de excepciones.
Enviando un ejemplo en el que un dato numérico es negativo.	- (ver explicación)	200 True/False	La aplicación no puede generar números negativos.
En vez de enviar valores numéricos, enviar texto.	Error genérico	500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos en vez de números a enviar.
El texto es vacío.	Error genérico	La función capaz de analizar en los dos idiomas: 500 Internal Server Error La función que solo analiza en inglés: 200 True/False	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos vacíos a enviar.
El texto es un carácter blanco.	Error genérico	500 Internal Server Error	No entra en el alcance la captura de excepciones y además la aplicación no puede generar textos con un solo carácter blanco a enviar.
Los campos numéricos de envío son 0.	200 True/False	200 True/False	El resultado es el esperado.

Tabla 13: Resultados de las pruebas de integración Smart-Ekitaldi

Estas pruebas se realizan para las dos funciones de la misma manera y se esperan y obtienen los mismos resultados, exceptuando la prueba con texto vacío (puede verse el resultado en la tabla 13). La diferencia se encuentra en los datos de entrada, donde el texto cambia de castellano a inglés. Sin embargo, desde la pantalla de Crear evento solo se invoca a la función que es capaz de analizar el texto sea en inglés o en castellano.

8. Gestión del proyecto

En este capítulo se revisa el proceso seguido durante el proyecto y cómo éste ha sido gestionado. Se analiza la gestión del alcance y los cambios principales que ha sufrido durante el desarrollo respecto a la versión original; la gestión de la colaboración, parte fundamental de este proyecto a tener en cuenta, y el número de horas dedicadas y las tareas a las que han sido asociadas.

8.1 Gestión del alcance

En el inicio del proyecto fue establecido un alcance inicial que ha ido variando durante el desarrollo. Esto se debe a que no conocía las herramientas utilizadas, Flutter y Firebase, por lo que los tiempos estimados al inicio no fueron una base suficientemente firme.

El objetivo principal y resultado de este proyecto era crear una aplicación que sirviera de acceso para una red social. Debido a que las redes sociales incluyen numerosos casos de uso y a que la idea del proyecto nace de otra que abarca más contenido, el alcance inicial fue acotado al tiempo disponible. Algunos casos de uso realizados en este proyecto han extendido el alcance inicial, por lo que se han desarrollado más de los estimados. Entre estos casos de uso se encuentran el poder compartir los eventos mediante un enlace y la pantalla de búsqueda, además del envío y recibo de notificaciones.

Por otro lado, uno de los riesgos del proyecto a tener en cuenta era la coordinación entre TFG-Elsa y este Trabajo de Fin de Grado. Al comienzo del proyecto se establecieron el alcance y los objetivos de forma que, si la integración entre las dos partes no pudiera llegar a realizarse, no afectara a ninguno de los proyectos. Finalmente, la integración ha sido realizada, como puede verse en el punto 5.1 (implementación de la lógica de negocio), y se han realizado pruebas (punto 7.3).

8.2 Gestión de la colaboración

La colaboración entre diferentes áreas del conocimiento es muy importante en grandes proyectos con el fin de integrar diferentes enfoques a un producto. Este proyecto ha sido realizado en colaboración entre dos Trabajos de Fin de Grado: el de Elsa Scola (TFG-Elsa) y el presente (TFG-Julen). La comunicación entre los dos interesados colaboradores del proyecto ha sido una parte de gran importancia a tener en cuenta durante todo el desarrollo. Para gestionar la comunicación de manera adecuada el sistema de información compartido ha sido fundamental.

A la hora de trabajar en las partes del proyecto que se tenían que realizar en conjunto, se ha procurado colaborar de manera simultánea y de forma presencial o mediante videoconferencia para discutir los aspectos de estas partes de la manera más ágil posible. En cambio, si una modificación en

la parte conjunta no ha podido hacerse entre las dos partes, se ha comunicado a la otra persona interesada los cambios realizados para obtener *feedback*.

Por otra parte, se ha realizado un seguimiento de los dos proyectos con el fin de conocer el estado del proyecto en conjunto y así evitar centrarse únicamente en el proyecto propio. Gracias a esto, los dos interesados tienen una visión completa y profunda del proyecto y han podido aportar al trabajo del otro.

Sistema de información compartido

A la hora de almacenar los datos, documentaciones y otros activos generados durante el desarrollo del Trabajo de Fin de Grado se ha utilizado un Sistema de Información conjunto con TFG-Elsa. Se busca así la facilidad de colaboración, creación de contenido conjunto y, además, permitir el acceso a las dos partes de los trabajos.

Para la elaboración y organización del Sistema de Información nos hemos apoyado en la Estructura de Descomposición del Proyecto que se puede ver en la figura 2. Lo que se busca es poder tanto ubicar fácilmente los contenidos que se generan en cada paquete del EDT como poder localizarlos rápidamente. Como servicio que da respaldo a nuestro Sistema de Información compartido hemos utilizado Google Drive por su gran integración de múltiples servicios, como el de edición de documentos o de libretas de cálculo, además de por la facilidad de compartir documentos. El resultado final de directorios se muestra en la figura 38. Se describe cada directorio a continuación:

- **Raíz**: contiene los siguientes directorios: *Desarrollo Producto*, *Gestión*, *Memoria* y *Otros Activos*.
- **Desarrollo Producto**: contiene toda la información relativa al producto y a su desarrollo. Este nodo se subdivide a su vez en cuatro secciones principales: *Análisis y Diseño*, *Implementación*, *Pruebas* y *Tecnologías*.
- **Análisis y Diseño**: contiene toda la información relativa al análisis previo al desarrollo del producto y a su diseño.
- **Implementación**: contiene exclusivamente los activos entregables del producto.
- **Tecnologías**: contiene los documentos relacionados con las tecnologías utilizadas en el producto, así como los pasos seguidos para sus implementaciones.
- **Gestión**: contiene toda la información relacionada con la gestión del proyecto. Este nodo se subdivide a su vez en nueve secciones principales: *Comunicaciones*, *Económica*, *Imagen Corporativa*, *Planificación*, *Plantillas*, *Reuniones*, *Seguimiento y Control*, *Smart-Ekitaldi* y *Tareas*.
- **Comunicaciones**: contiene toda la información relativa a las comunicaciones realizadas.
- **Imagen Corporativa**: contiene elementos de la imagen corporativa como, por ejemplo, el logo, los encabezados…
- **Planificación**: contiene toda la información relacionada con la planificación del proyecto.
- **Plantillas**: contiene los documentos que sirven como plantillas, de manera que se intenta estandarizar su creación entre los dos Trabajos de Fin de Grado (Smart-Ekitaldi). Se encuentran las plantillas de documentos normales y las de las actas de las reuniones.

- **Reuniones**: contiene toda la información relativa a las reuniones realizadas a nivel interno.
- **Seguimiento y Control**: contiene toda la información relativa al monitoreo y control del proyecto (tiempos de trabajo, complimiento de plazos y objetivos...).
- **Smart-Ekitaldi**: contiene los documentos de integración de los dos proyectos, así como los acuerdos que se han tomado durante el desarrollo.
- **Tareas**: contiene toda la información relacionada con las tareas que se llevan a cabo en el proyecto, así como el estado de las mismas.
- **Memoria**: contiene los documentos que forman parte de la memoria.
- **Otros Activos**: contiene todos los activos que se han obtenido o generado durante el proyecto, que por sí mismos no forman parte del producto pero que tienen valor o pueden tenerlo en un futuro. Este nodo se subdivide a su vez en dos secciones principales: *Generados* y *Recursos Descargados*.
- **Generados**: contiene todo activo que se haya generado durante el proyecto pero no pertenezca al producto ni a las tecnologías utilizadas.
- **Recursos Descargados**: contiene todo activo de utilidad descargado que no forme parte del producto.

Para diferenciar los documentos que pertenecen a este trabajo, TFG-Julen, de los que pertenecen al trabajo de TFG-Elsa dentro del Sistema de Información compartido se sigue uno de los siguientes dos métodos, dependiendo del volumen de documentos que hay dentro de cada directorio:

- Si hay cuatro documentos o menos, se diferencian escribiendo en el nombre de cada archivo ".X", siendo X la letra J si el archivo forma parte de TFG-Julen o la letra E si el archivo forma parte de TFG-Elsa. Por ejemplo, el documento de Antecedentes de TFG-Julen se llamaría *TFG.Antecedentes.J*.
- Si hay más de cuatro documentos, el directorio se subdivide en dos, uno con el nombre Smart y otro con el nombre Ekitaldi, en el que van los archivos de cada una de las partes.

Además, el encabezado de cada página dentro de los documentos contiene el nombre del autor.

- TFG
  - Desarrollo Producto
    - Análisis y Diseño
    - Implementación
    - Pruebas
    - Tecnologías
  - Gestión
    - Comunicaciones
    - Imagen Corporativa
    - Planificación
    - Plantillas
    - Reuniones
    - Seguimiento y Control
    - Smart-Ekitaldi
    - Tareas
  - Memoria
  - Otros Activos
    - Generados
    - Recursos Descargados

Figura 38: Esquema del sistema de información conjunto

8.3 Dedicaciones

Durante el desarrollo del proyecto se ha llevado la contabilidad de las horas dedicadas a cada paquete definido en la Estructura de Descomposición del Proyecto (figura 2). El resultado de este seguimiento puede verse reflejado en la tabla 14.

Paquete de trabajo	Horas dedicadas
P.Planificación	4
P.SeguimientoYControl	2
P.Gestión.IntegraciónSmart-Ekitaldi	14
P.Memoria	101
P.Defensa	20
P.Flutter	15
P.Firebase	4
P.ArquitecturaDeLaSolución	2
P.BaseDeDatos	3
P.InteracciónConElUsuario	19
P.AnálisisYDiseño.CasosDeUso	14
P.Documentación	12
P.Implementación.CasosDeUso	141
P.Pruebas.ConUsuarios	2
P.Pruebas.IntegraciónSmart-Ekitaldi	3
Dedicación total:	**355**

Tabla 14: Dedicaciones a cada paquete de la EDT

Estas dedicaciones han sido contabilizadas desde enero de 2019 hasta junio de 2019. El paquete de trabajo del que más dedicación ha precisado ha sido el de la implementación de los casos de uso. Gracias a Bitbucket, el servicio en la nube con el que se ha integrado Git, se ha generado la figura 39 donde puede verse un gráfico con el seguimiento del número de los *commit* realizados durante la implementación. Esta gráfica permite ver en qué punto cronológico se centra la mayoría de la dedicación al paquete de implementación de los casos de uso por cada semana.

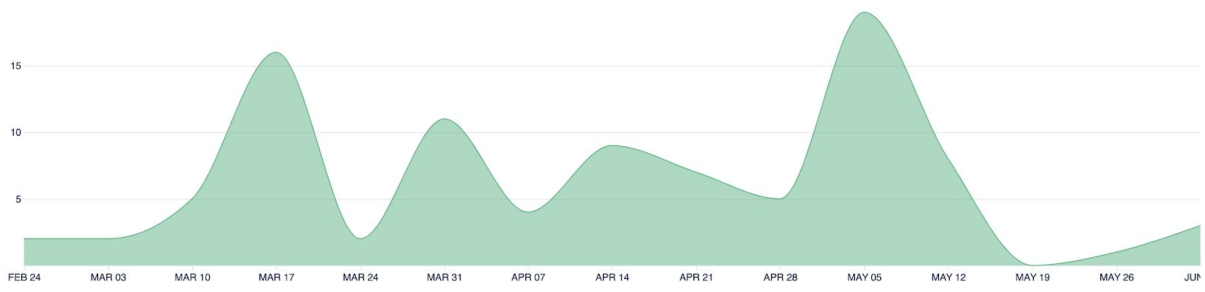

Figura 39: Gráfico de commits por semanas

9. Conclusiones

Tras haber realizado el trabajo y haber podido utilizar las novedosas herramientas, he llegado a ciertas conclusiones sobre lo que éstas (Flutter, Firebase…) pueden aportar a todo tipo de proyectos, ya sean pequeños trabajos de clase como grandes proyectos.

9.1 Tecnologías y metodologías utilizadas

Las novedosas tecnologías utilizadas durante este proyecto han sido: Flutter, Dart (como lenguaje de programación de Flutter), Firebase (y JavaScript para definir las funciones) y Algolia (para integrar las búsquedas en la aplicación).

Tanto Flutter como Dart son razonablemente sencillas de utilizar si tienes conocimientos sobre programación, tras un periodo de aprendizaje de aproximadamente 8 horas gracias a un curso de Udacity[37]. Crear una aplicación simple en Flutter es rápido, y el hecho de poder generar la aplicación nativa para los dos sistemas operativos para *smartphones* con mayor expansión en el mercado hace que sea una herramienta muy atractiva. Flutter puede ser utilizado para crear prototipos rápidos ya que, gracias a sus funcionalidades, como Hot Reload, puedes ver el resultado de lo que estás desarrollando en tiempo real. Además, al tener una variedad de Widgets tan amplia, cualquier cambio, por pequeño que sea, provoca grandes avances en el desarrollo. El modo de utilización de los Widgets es un proceso que necesita de una curva de aprendizaje pequeña, sobre todo si se cuenta con experiencia en el diseño de interfaces de usuario html o xml, ya que se basa en la misma idea de construir los elementos de forma jerárquica. Sin embargo, la creación de una aplicación compleja utilizando Flutter requiere de una curva de aprendizaje mayor, ya que la gestión eficiente de los estados no es trivial y en algunos casos puede ser muy costosa y frustrante.

También añadir que me ha sorprendido gratamente cómo una herramienta con tan poco trayecto (lanzada en mayo de 2017) tiene tanto soporte. Tiene una gran comunidad y se puede ver tanto en el repositorio público[38] en el que se encuentran disponibles muchos paquetes integrables como en el blog Medium[39], en el que se publican muchas soluciones a distintos problemas comunes. Además, Flutter está siendo la herramienta con mayor crecimiento en la plataforma LinkedIn entre los ingenieros del software[40].

Firebase por su parte, es una herramienta que reúne diferentes servicios[41] integrables en proyectos. Al estar todos reunidos en la misma plataforma, implantar uno de ellos es rápido y

[37] Enlace al curso: https://eu.udacity.com/course/build-native-mobile-apps-with-flutter--ud905
[38] Enlace al repositorio: https://pub.dev/
[39] Publicaciones en Medium sobre Flutter: https://medium.com/flutter-community
[40] Enlace al artículo: https://www.mobileappdaily.com/best-skills-for-software-engineers
[41] Servicios que ofrece Firebase: https://firebase.google.com/products?hl=es

requiere de poca configuración, ya que la mayor configuración se hace al conectar la aplicación con Firebase.

9.2 Problema abordado y solución dada

Los eventos que se realizan alrededor de la universidad, tanto los organizados por la misma como los organizados por los estudiantes, tienden a tener una complicada difusión. El método empleado con mayor frecuencia es el envío de mensajes promocionales, que son borrados sin leer (si es que se accede al buzón) en la mayoría de los casos. Esto hace que la participación sea menor de la que podría llegar a ser con una publicidad más eficiente.

Para ofrecer otra alternativa se presenta la aplicación objeto del TFG, en la que se pueden publicar y dar a conocer los eventos en forma de red social. Gracias a las opciones sociales, como los comentarios o las puntuaciones, es más fácil atraer a los potenciales asistentes. Además, los estudiantes tienen una plataforma para difundir los eventos que organicen. Adicionalmente, el hecho de publicar los eventos en una red social promueven interacción entre los diferentes alumnos de los campus. Los alumnos podrán conocer todos los eventos que hasta ahora se perdían y podían resultarles de interés, ver a qué eventos van a asistir sus compañeros, compartir los eventos mediante un enlace… Además, con las opciones de filtrado y búsqueda podrán ver los eventos que más se ajusten a sus preferencias y así conocer a gente con intereses similares. Adicionalmente, con la integración de la parte Smart, se incluye un filtro de SPAM, con lo que se reduce el número de publicaciones que no sean útiles para los usuarios.

En resumen, la aplicación cuenta con todos los elementos necesarios para que los usuarios puedan sacarle el máximo provecho a su experiencia académica y disfruten de los eventos universitarios como no habían podido hacer antes.

9.3 Respecto al grado

Durante la realización del proyecto, he sido capaz de hacer uso de ciertas habilidades que he aprendido durante el grado de Ingeniería Informática. Gracias a la asignatura de Gestión de Proyectos, he podido organizar el proyecto y el sistema de información, planificar el proyecto, hacer el seguimiento… Las asignaturas de Ingeniería del Software y otras de programación han hecho que mis implementaciones estén bien organizadas y documentadas, además de haber aprendido a hacer el análisis y el diseño de los casos de uso. Con Sistemas Web y Herramientas Avanzadas del Desarrollo del Software he aprendido a hacer uso de servicios web, lo que me ha sido útil al crear las funciones de Firebase y al utilizarlas, así como a utilizar Git para llevar el seguimiento del proyecto.

Aunque en las asignaturas de diseño de bases de datos no se expongan las bases de datos NoSQL, los fundamentos que he aprendido me han permitido hacer el diseño de la base de datos NoSQL de mi proyecto, tal y como se muestra en el punto 4.3.1. Además, aunque no dio tiempo a verlo en clase, los apuntes de NoSQL de la asignatura de Gestión Avanzada de la Información han ayudado en el proceso. Adicionalmente, la asignatura de Machine Learning and Neural Networks, que escogí

como optativa, me ha ayudado a entender el proyecto de Elsa y a darle mi opinión en algunos aspectos, además de haber facilitado la integración entre nuestros proyectos.

Por otro lado, como en el grado de Ingeniería Informática el desarrollo de aplicaciones móviles se aborda de manera superficial, la integración de Flutter en los estudios no resultaría en un gran impacto. En la asignatura de Servicios de Aplicaciones en Red se introduce el desarrollo en Android, lo que podría sustituirse por Flutter. Por otro lado, la forma de implementar en Flutter permite crear prototipos rápidamente. Esto podría ser útil en asignaturas como Interacción Persona Computador, impartida en la rama de Ingeniería del Software, en la que se pide realizar un prototipo de aplicación.

Aparte del uso de Flutter, pienso que la creación de aplicaciones móviles es un área de gran interés. Me hubiera gustado cursar una asignatura dedicada a ello durante mis estudios.

9.4 Futuro de la aplicación

Debido al tiempo del que he dispuesto para la realización del proyecto y la enorme amplitud del mismo, algunos aspectos se han excluido de su realización.

Por un lado, si se quisiera llegar a utilizar la aplicación, sería necesario desarrollar los aspectos legales y de protección de datos necesarios para poder lanzar la aplicación al mercado y para poder utilizar el inicio de sesión. También sería recomendable crear un perfil de administrador con casos de uso específicos para la gestión avanzada del contenido que se genera en la red social. Actualmente se pueden editar datos únicamente accediendo a la base de datos, lo que hace que peligre su consistencia. Por otro lado, habría que implementar funciones para el cálculo de las puntuaciones de tanto los perfiles como los eventos y sus comentarios. Estos cálculos podrían realizarse utilizando aprendizaje automático.

También hay otros aspectos de la aplicación que podrían mejorarse. El primero de ellos sería crear una versión web de la red social. El equipo de Flutter ha anunciado la compatibilidad de la herramienta con la web en su versión 1.5. Lamentablemente, coincidiendo con la finalización del proyecto, en junio de 2019, esta funcionalidad está siendo probada y no hay fecha de lanzamiento definida para su versión estable. Hasta entonces no debería incluirse en el proyecto por incompatibilidad con los plugins que se utilizan. También sería recomendable mejorar la gestión de los comentarios dentro de los eventos, ya que por el momento se muestran todos. Además, sería de interés incluir apartados que tengan que ver con el análisis de datos, como un apartado de recomendaciones personalizadas. Para mejorar la experiencia de los usuarios de la aplicación, sería recomendable utilizar el servicio ML Kit (disponible dentro de Firebase) para traducir el texto de los eventos. Además, la aplicación podría traducirse a más idiomas aparte de los ya disponibles.

Mirando los aspectos del desarrollo, deberían desarrollarse pruebas automáticas de las funciones. También sería interesante incluir la utilización de *providers*[42], clases que se encargan de crear estados globales entre varios *Widgets*. Esto podría ser de utilidad por ejemplo para la pantalla de búsqueda, donde un *Widget* puede encargarse de manejar el campo de texto de la búsqueda y otro los resultados de la misma.

[42] Paquete *provider*: https://pub.dev/packages/provider

10. Fuentes

---

1. What's Revolutionary About Flutter?
   https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514
2. Post de Medium sobre la autenticación con email utilizando Firebase Auth en Flutter:
   https://medium.com/flutterpub/flutter-how-to-do-user-login-with-firebase-a6af760b14d5
3. Post de Medium sobre la autenticación con Twitter utilizando Firebase Auth en Flutter:
   https://medium.com/@euedofia/flutter-firebase-authenticate-with-twitter-d73c0602bb0b
4. Post de Medium sobre la autenticación con Facebook utilizando Firebase Auth en Flutter:
   https://flatteredwithflutter.com/firebase-facebook-sign-in-and-flutter/
5. Post de Medium sobre la autenticación con Google utilizando Firebase Auth en Flutter:
   https://medium.com/flutterpub/firebase-google-sign-in-and-flutter-6f7abde9ae5a
6. Post de Medium sobre la creación de enlaces dinámicos enlaces dinámicos:
   https://medium.com/flutter-community/if-you-want-to-make-your-product-popular-the-strategy-of-mouth-publicity-is-unavoidable-part-of-9da78ea32e79
7. Post de Medium sobre la gestión de enlaces dinámicos:
   https://medium.com/flutter-community/handling-firebase-dynamic-links-in-flutter-7c1de6a4e2e
8. Post de Medium sobre la utilización de Firebase Cloud Messaging en Flutter:
   https://medium.com/flutterpub/enabling-firebase-cloud-messaging-push-notifications-with-flutter-39b08f2ed723

11. Anexos

Anexo 1: Diagrama de la base de datos

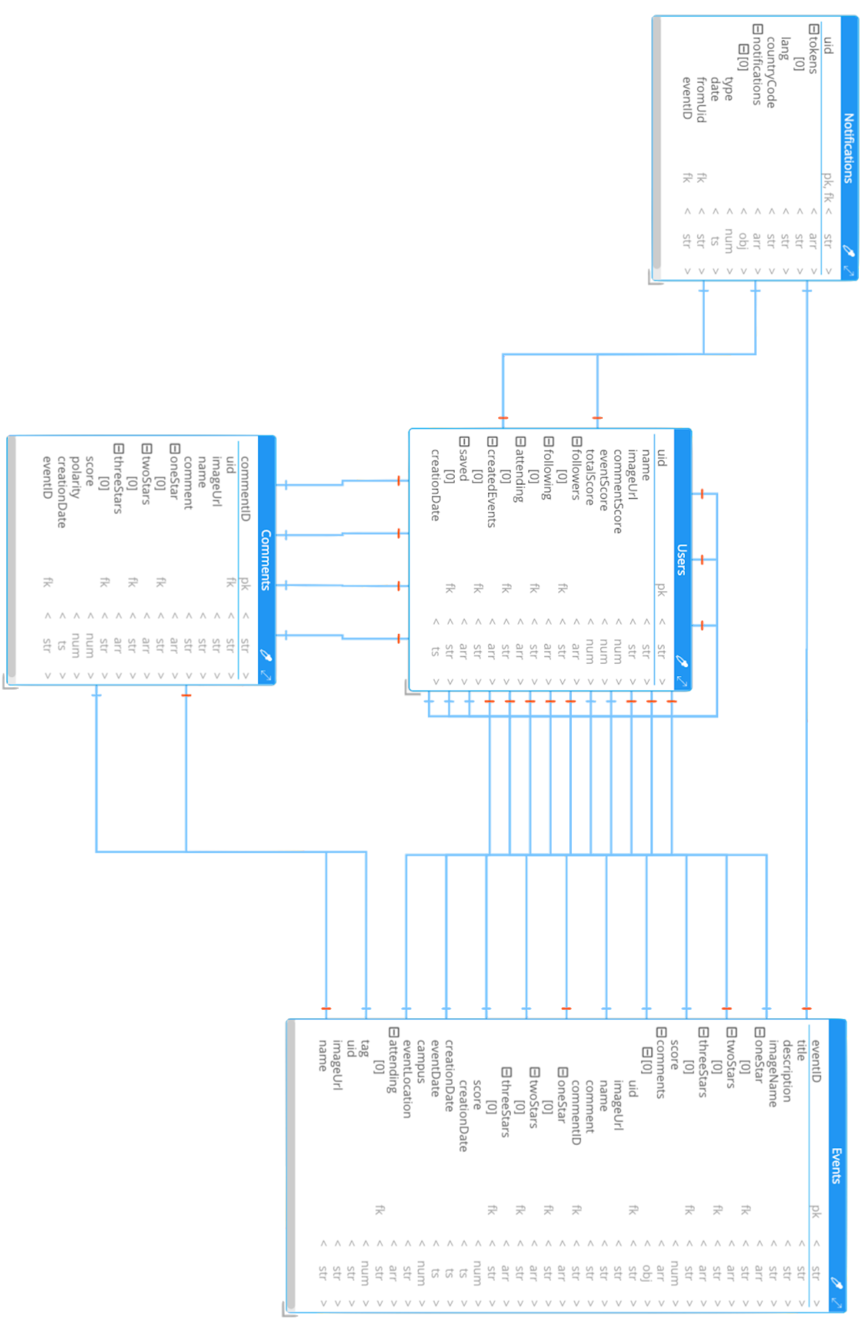

Figura 40: Diagrama de la base de datos

Anexo 2: Diagrama de flujo de crear evento

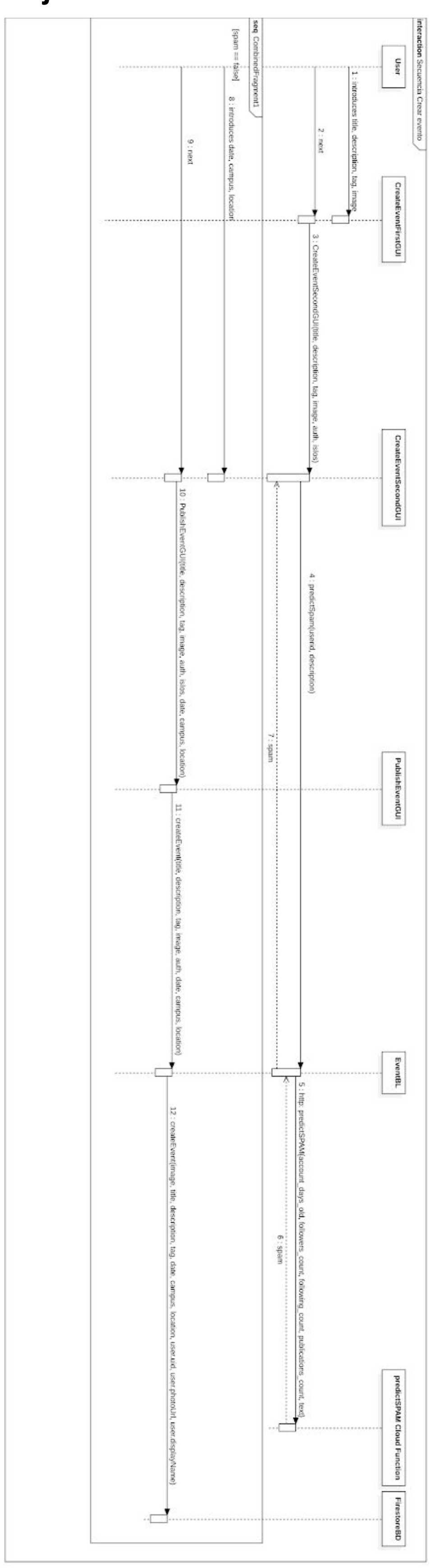

Figura 41: Diagrama de flujo de crear evento

Anexo 3: Fragmentos de código de Flutter

Contenido del fichero *create_event_first.dart*:

```
class CreateEventFirstGUI extends StatefulWidget {
 CreateEventFirstGUI({@required this.isIos}) : assert(isIos != null);
 bool isIos;
 _CreateEventFirstGUIState createState() => _CreateEventFirstGUIState();
}

class _CreateEventFirstGUIState extends State<CreateEventFirstGUI> {
 final _padding = EdgeInsets.fromLTRB(16, 8, 16, 8);
 File _image;
 String _title;
 String _description;
 String _currentTag;
 bool _titleError;
 bool _descriptionError;
 bool _loading;
 bool _spamError;
 bool _spamCallingError;

 void initState() {
   super.initState();
   _currentTag = 0.toString();
   _descriptionError = false;
   _titleError = false;
   _loading = false;
   _spamError = false;
   _spamCallingError = false;
 }

 Widget build(BuildContext context) {}
 Widget _textField(
     String helper, int maxLines, bool errorFlag, String errorMsg, int value) {}
 List<DropdownMenuItem<String>> _dropDownMenuItems() {}
 Widget _buttons() {}
 void _getImage() {}
 Future<File> _compressImage(File file) async {}
 Future<void> _getImageGallery() async {}
 Future<void> _getImageCamera() async {}
 void _deleteImage() {}
 void _next() async {}
 void _validateTitle() {}
 void _validateDescription() {}
}
```

Contenido del fichero *create_event_second_gui.dart*:

```
class CreateEventSecondGUI extends StatefulWidget {
 CreateEventSecondGUI(
     {@required this.isIos,
```

```
      this.image,
      @required this.title,
      @required this.description,
      @required this.tag})
      : assert(isIos != null &&
            title != null &&
            description != null &&
            tag != null);

  bool isIos;
  File image;
  String title;
  String description;
  int tag;
  _CreateEventSecondGUIState createState() => _CreateEventSecondGUIState();
}

class _CreateEventSecondGUIState extends State<CreateEventSecondGUI> {
  final _scaffoldKey = GlobalKey<ScaffoldState>();
  DateTime _eventDate;
  String _currentCampus;
  bool _dateError;
  bool _locationError;
  String _location;
  TextEditingController _dateController = new TextEditingController();

  List<DropdownMenuItem<String>> _dropDownMenuItems() {}

  void initState() {
    _currentCampus = 0.toString();
    _dateError = false;
    _locationError = false;
    super.initState();
  }

  Widget build(BuildContext context) {}
  Future<void> _selectDate() async {}
  String _getDate() {}
  void _validateDate() {}
  void _validateLocation() {}
  void _next() {}
  void _publish() {}
}
```

Contenido del fichero *delete_event_gui.dart*:

```
class DeleteEventGUI extends StatefulWidget {
  DeleteEventGUI({@required this.isIos, @required this.eventID})
      : assert(isIos != null && eventID != null);
  bool isIos;
  String eventID;
  _DeleteEventGUIState createState() => _DeleteEventGUIState();
}
```

```
class _DeleteEventGUIState extends State<DeleteEventGUI> {
 bool _loading;

 TextEditingController _controller;

 void initState() {
   super.initState();
   _loading = false;
 }

 Widget build(BuildContext context) {}
 void _deleteEvent() {}
}
```

Contenido del fichero *view_event_gui.dart*:

```
class ViewEventGUI extends StatefulWidget {
 ViewEventGUI({@required this.event, @required this.isIos})
     : assert(isIos != null && event != null);
 Event event;
 bool isIos;
 _ViewEventGUIState createState() => _ViewEventGUIState();
}

class _ViewEventGUIState extends State<ViewEventGUI> {
 bool _attends;
 Event _viewingEvent;
 String _comment;
 final TextEditingController _controller = new TextEditingController();
 List<Comment> _comments;
 bool _saved;
 void initState() {
   super.initState();
   _viewingEvent = widget.event;
   _attends = UserBL.attends(_viewingEvent.eventID);
   _saved = UserBL.saved(_viewingEvent.eventID);
   _comments = _viewingEvent.comments;
 }

 Widget build(BuildContext context) {}
 Future<void> _updateEvent() async {}
 void _changeSaved() {}
 void _storeComment(String comment) {}
 void _openImage() {}
 void _attend() {}
}

class _StarRating extends StatefulWidget {
 _StarRating(this.event, this.uid);

 Event event;

 String uid;
 _StarRatingState createState() => _StarRatingState();
```

```
}

class _StarRatingState extends State<_StarRating> {
 int _voted;
 int _score;
 void initState() {
   super.initState();
   _voted = _getVoted();
   _score = widget.event.score;
 }

 Widget build(BuildContext context) {}
 int _getVoted() {}
 void _vote(String eventID, String uid, int voted) {}
}

class _ImageGUI extends StatefulWidget {
 _ImageGUI({this.imageUrl, this.isIos})
     : assert(imageUrl != null && isIos != null);

 String imageUrl;
 bool isIos;
 _ImageGUIState createState() => _ImageGUIState();
}

class _ImageGUIState extends State<_ImageGUI> {
 Widget build(BuildContext context) {}
}

class _CommentWidget extends StatefulWidget {
 _CommentWidget(this.comment, this.uid, this.eventID);
 Comment comment;
 String uid;
 String eventID;
 _CommentWidgetState createState() => _CommentWidgetState();
}

class _CommentWidgetState extends State<_CommentWidget> {
 int _voted;
 int _score;
 void initState() {
   super.initState();
   _voted = _getVoted();
   _score = widget.comment.score;
 }

 Widget build(BuildContext context) {}
 List<Widget> _getCommentStars() {}
 int _getVoted() {}
 void _vote(String eventID, String commentID, String uid, int voted) {}
}
```

Contenido del fichero *view_events_by_campus_gui.dart*:

```
class ViewEventsByCampusGUI extends StatefulWidget {
 ViewEventsByCampusGUI({@required this.isIos, @required this.campus})
     : assert(isIos != null && campus != null);
 bool isIos;
 int campus;
 _ViewEventsByCampusGUIState createState() => _ViewEventsByCampusGUIState();
}

class _ViewEventsByCampusGUIState extends State<ViewEventsByCampusGUI> {
 List<Event> _events;
 void initState() {
   super.initState();
   _updateEvents();
 }

 Widget build(BuildContext context) {}
 Future<void> _updateEvents() async {}
}
```

Contenido del fichero *view_events_by_tag_gui.dart*:

```
class ViewEventsByTagGUI extends StatefulWidget {
 ViewEventsByTagGUI({@required this.isIos, @required this.tag})
     : assert(isIos != null && tag != null);

 bool isIos;
 int tag;
 _ViewEventsByTagGUIState createState() => _ViewEventsByTagGUIState();
}

class _ViewEventsByTagGUIState extends State<ViewEventsByTagGUI> {
 List<Event> _events;
 void initState() {
   super.initState();
   _updateEvents();
 }

 Widget build(BuildContext context) {}

 Future<void> _updateEvents() async {}
}
```

Contenido del fichero *view_profile_gui.dart*:

```
class ViewProfileGUI extends StatefulWidget {
 ViewProfileGUI({@required this.isIos, @required this.viewingUserUid})
     : assert(isIos != null && viewingUserUid != null);
```

```
 bool isIos;
 String viewingUserUid;
 _ViewProfileGUIState createState() => _ViewProfileGUIState();
}

class _ViewProfileGUIState extends State<ViewProfileGUI> {
 User _viewingUser;
 bool _following;
 List<Event> _attending;
 List<Event> _created;
 int _batchAttending;
 int _batchCreated;
 bool _loadingMore;
 void initState() {
   super.initState();
   _loadingMore = false;
   _updateUser();
 }

 Future<void> _getUserInfo() async {}
 Widget build(BuildContext context) {}
 Future<void> _updateUser() async {}
 Future<List<Event>> _getEventsFromIDs(
     List<String> eventsIDs, int batch) async {}
 Widget _getUserInfoBar() {}
 Widget _getNumberWithText(int number, String text) {}
 Widget _getScoreWidget(double number, String text) {}
 void _changeFollowing() {}
 void _editProfile() {}
}
```

Contenido del fichero *edit_profile_gui.dart*:

```
class EditProfileGUI extends StatefulWidget {
 EditProfileGUI({@required this.isIos}) : assert(isIos != null);
 bool isIos;
 _EditProfileGUIState createState() => _EditProfileGUIState();
}

class _EditProfileGUIState extends State<EditProfileGUI> {
 final _scaffoldKey = GlobalKey<ScaffoldState>();
 bool _localImage;
 File _newImage;
 String _imageUrl;
 String _name;
 bool _loading;
 TextEditingController _controller;

 void initState() {
   super.initState();
   _imageUrl = UserBL.getImageUrl();
   _localImage = false;
   _name = UserBL.getName();
   _controller = new TextEditingController(text: _name);
```

```
    _loading = false;
  }
  Widget build(BuildContext context) {}
  void _getImage() {}
  void _saveChanges() {}
}
```

Contenido del fichero *main_feed_gui.dart*:

```
class MainFeedGUI extends StatefulWidget {
  MainFeedGUI({@required this.isIos});
  bool isIos;
  _MainFeedGUIState createState() => _MainFeedGUIState();
}

class _MainFeedGUIState extends State<MainFeedGUI> {
  List<Event> _events;
  int _tagFilter;
  int _campusFilter;
  bool _loadingMore;

  void initState() {
    super.initState();
    _tagFilter = -1;
    _campusFilter = -1;
    _loadingMore = false;
    _getFeed(_tagFilter, _campusFilter);
  }

  Widget build(BuildContext context) {}
  Future<void> _loadMore() async {}
  List<Widget> _buildTagFilterChips(int tagFilter) {}
  List<Widget> _buildCampusFilterChips(int campusFilter) {}
  Future<void> _getFeed(int tagFilter, int campusFilter) async {}
}
```

Contenido del fichero *search_gui.dart*:

```
class SearchGUI extends StatefulWidget {
  SearchGUI({@required this.isIos});
  bool isIos;
  _SearchGUIState createState() => _SearchGUIState();
}

class _SearchGUIState extends State<SearchGUI> {
  List<dynamic> _result;
  TextEditingController _controller;
  bool _searchEvents;

  void initState() {
    super.initState();
```

```
    _controller = new TextEditingController();
    _result = [];
    _searchEvents = true;
  }

  Widget build(BuildContext context) {}
  Future<void> _search(String searchText) async {}
```

Contenido del fichero *view_saved_gui.dart*:

```
class ViewSavedGUI extends StatefulWidget {
  ViewSavedGUI({@required this.isIos});
  bool isIos;
  _ViewSavedGUIState createState() => _ViewSavedGUIState();
}

class _ViewSavedGUIState extends State<ViewSavedGUI> {
  List<Event> _events;
  bool _loadingMore;
  int _batch;

  void initState() {
    super.initState();
    _loadingMore = false;
    _updateSaved();
  }

  Widget build(BuildContext context) {}
  Future<void> _updateSaved() async {}
  Future<List<Event>> _getEventsFromIDs(List<String> eventsIDs,int batch) async{}
  }
}
```

Contenido del fichero *view_notifications_gui.dart*:

```
class ViewNotificationsGUI extends StatefulWidget {
  ViewNotificationsGUI({@required this.isIos});
  bool isIos;
  _ViewNotificationsGUIState createState() => _ViewNotificationsGUIState();
}

class _ViewNotificationsGUIState extends State<ViewNotificationsGUI> {
  List<CustomNotification> _notifications;

  void initState() {
    super.initState();
    _loadNotifications();
  }

  Widget build(BuildContext context) {}
  List<Widget> _getNotifications(List<CustomNotification> notifications) {}
```

```
 Future<void> _loadNotifications() async {}
}
```

Contenido del fichero *event_card_widget.dart*:

```
class EventCard extends StatefulWidget {
 EventCard({@required this.event, @required this.isIos});
 Event event;
 bool isIos;
 _EventCardState createState() => _EventCardState();
}

class _EventCardState extends State<EventCard> {
 bool _attends;
 bool _showAll;
 String _comment;
 final TextEditingController _controller = new TextEditingController();
 List<Comment> _comments;
 bool _saved;
 int _heroRandom;

 void initState() {
   super.initState();
   _attends = UserBL.attends(widget.event.eventID);
   _saved = UserBL.saved(widget.event.eventID);
   _showAll = false;
   _comments = widget.event.comments;
   _heroRandom = Random().nextInt(100000);
 }

 Widget build(BuildContext context) {}
 void _expand() {}
 void _changeSaved() {}
 void _storeComment(String comment) {}
 void _openImage(int heroRandom) {}
 void _attend() {}
 void _editEvent() {}
 void _deleteEvent() {}
}

class _StarRating extends StatefulWidget {
 _StarRating(this.event, this.uid);
 Event event;
 String uid;
 _StarRatingState createState() => _StarRatingState();
}

class _StarRatingState extends State<_StarRating> {
 int _voted;
 int _score;

 void initState() {
   super.initState();
   _voted = _getVoted();
```

```
    _score = widget.event.score;
  }

  Widget build(BuildContext context) {}
  int _getVoted() {}
  void _vote(String eventID, String uid, int voted) {}
}

class _ImageGUI extends StatelessWidget {
  _ImageGUI({this.imageUrl, this.isIos, this.eventID, this.heroRandom})
      : assert(imageUrl != null && isIos != null);

  String imageUrl;
  bool isIos;
  String eventID;
  int heroRandom;
  Widget build(BuildContext context) {}
}

class _CommentWidget extends StatefulWidget {
  _CommentWidget(this.comment, this.uid, this.eventID);
  Comment comment;
  String uid;
  String eventID;
  _CommentWidgetState createState() => _CommentWidgetState();
}

class _CommentWidgetState extends State<_CommentWidget> {
  int _voted;
  int _score;

  void initState() {
    super.initState();
    _voted = _getVoted();
    score = widget.comment.score;
  }

  Widget build(BuildContext context) {}
  List<Widget> _getCommentStars() {}
  int _getVoted() {}
  void _vote(String eventID, String commentID, String uid, int voted) {}
}
```

Contenido del fichero *notification_widget.dart*:

```
class NotificationWidget extends StatelessWidget {
  NotificationWidget(this.notification, this.isIos);
  CustomNotification notification;
  bool isIos;

  Widget build(BuildContext context) {}
  void _goToProfile(BuildContext context) {}
  void _goToEvent(BuildContext context) async {}
}
```

Contenido del fichero *search_result_widget.dart*:

```
class SearchResultEvent extends StatelessWidget {
 SearchResultEvent({@required this.event, @required this.isIos});
 Event event;
 bool isIos;

 Widget build(BuildContext context) {}
}
class SearchResultUser extends StatelessWidget {
 SearchResultUser({@required this.user, @required this.isIos});
 User user;
 bool isIos;

 Widget build(BuildContext context) {}
}
```

Contenido del fichero *authentication_bl.dart*:

```
class Auth {
 final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;
 Future<String> twitterSignIn(String langCode, String countryCode) async {}
 Future<String> facebookSignIn(String langCode, String countryCode) async {}
 Future<String> googleSignIn(String langCode, String countryCode) async {}
 Future<String> _signIn(AuthCredential credential, String langCode, String
countryCode) async {}
 Future<bool> isLogged() async {}
 Future<FirebaseUser> getCurrentUser() async {}
 Future<void> signOut() async {}
}
```

Contenido del fichero *event_bl.dart*:

```
class EventBL {
 void createEvent(File image,
     String title,
     String description,
     int tag,
     DateTime date,
     int campus,
     String location) {}
 Future<void> updateEvent(File image,
     String title,
     String description,
     int tag,
     DateTime date,
     int campus,
     String location,
     String eventID) async {}
```

```
 Future<List<Event>> getFeed(int tagFilter, int campusFilter,
     String lastEventID) {}
 Future<List<Event>> getLoginPreview() {
 Future<List<int>> vote(String eventID, String uid, int voted,
     String creatorUid) async {}
 Future<Comment> addComment(String comment, String eventID,
     String creatorUid) async {}
 Future<bool> changeAttendance(String uid, String eventID,
     String creatorUid) async {}
 Future<bool> changeSaved(String uid, String eventID) async {}
 Future<List<int>> voteComment(String eventID, String commentID, String uid,
     int voted) async {}
 Future<List<Event>> getEventsFromIDs(List<String> eventsIDs) async {}
 Future<List<Event>> getEventsByTag(int tag) async {}
 Future<List<Event>> getEventsByCampus(int campus) async {}
 Future<void> deleteEvent(String eventID, String uid) async {}
 Future<bool> predictSpam(String description, String uid) async {}
}
```

Contenido del fichero *user_bl.dart*:

```
class UserBL {
 static User _user;
 static Future<void> init(String uid) async {}
 static bool isLogged() {}
 static void unfollow(String to) {}
 static Future<void> follow(String to) async {}
 static Future<User> getUser(String uid) async {}
 static Future<User> getUpdatedCurrentUser() async {}
 static String getUid() {}
 static String getImageUrl() {}
 static String getName() {}
 static List<String> getSaved() {}
 static bool attends(String eventID) {}
 static bool saved(String eventID) {}
 static void _setName(String name) {}
 static void _setImageUrl(String imageUrl) {}
 static void changeSaved(bool saved, String eventID) {}
 static Future<void> updateUser(File image, String name) async {}
 static Future<List<CustomNotification>> loadNotifications() async {}
 static Future<void> changeLocale(
     String languageCode, String countryCode) async {}
}
```

Contenido del fichero *search_bl.dart*:

```
class SearchBL {
 Future<List<Event>> searchEvents(String searchText) async {}
 Future<List<User>> searchUsers(String searchText) async {}
}
```

Contenido del fichero *firestore_db.dart*:

```
class FirestoreDB {
 final _firestore = Firestore.instance;
 void saveUserInfo(String userID, String name, String photoURL, String token,
     String langCode, String countryCode) {}
 Future<void> logOut(String userID, String token) async {}
 Future<void> createEvent(
     File image,
     String title,
     String description,
     int tag,
     DateTime date,
     int campus,
     String location,
     String userID,
     String imageUrl,
     String name) async {}
 Future<String> _generateDynamicLink(
     String eventID, String title, String description, String imageUrl) async {}
 Future<void> updateEvent(
     File image,
     String title,
     String description,
     int tag,
     DateTime date,
     int campus,
     String location,
     String eventID) async {}
 Future<List<Event>> getFeed(int tagFilter, int campusFilter, String
lastEventID) async {}
 Future<List<Event>> getLoginPreview() {
 Future<List<int>> voteEvent(String eventID, String uid, int voted) async {}
 Future<Comment> addComment(String comment, String uid, String eventID,
     String imageUrl, String name) async {}
 Future<bool> changeAttendance(
     String uid, String eventID) async {}
 Future<bool> changeSaved(String uid, String eventID) async {}
 Future<List<int>> voteComment(
     String eventID, String commentID, String uid, int voted) async {}
 Future<User> getUser(String uid) async {}
 Future<List<Event>> getEventsFromIDs(List<String> eventsIDs) async {}
 void follow(String from, String to) {}
 void unfollow(String from, String to) {}
 Future<String> updateUser(String uid, File image, String name) async {}
 Future<Map<String, dynamic>> addNotification(
     String toUid, String fromUid, int type, String eventID) async {}
 void removeNotification(
     String toUid, String fromUid, int type, String eventID) async {}
 Future<List<CustomNotification>> loadNotifications(String uid) async {}
 Future<List<Event>> getEventsByTag(int tag) async {}
 Future<List<Event>> getEventsByCampus(int campus) async {}
 Future<void> deleteEvent(String eventID, String uid) async {}
 Future<void> changeLocale(String uid, String languageCode, String countryCode)
```

```
async {}
}
```

Contenido del fichero *algolia_db.dart*:

```
class AlgoliaDB {
 static Future<void> createEvent(
     String eventID,
     String imageName,
     String title,
     String description,
     DateTime date,
     String location) async {}
 static Future<void> deleteEvent(String eventID) async {}
 static Future<void> updateEvent(String eventID,
     String imageName,
     String title,
     String description,
     DateTime date,
     String location) async {}
 static Future<void> createUser(String userID, String name, String photoURL)
async {}
 static Future<void> updateUser(String userID, String name, String photoURL)
async {}
}
```

Anexo 4: Registrar la aplicación Flutter en Firebase

Para poder integrar los servicios de Firebase dentro de la aplicación desarrollada con Flutter, se debe crear un proyecto en Firebase, descargar los archivos de configuración y añadirlos al proyecto con Flutter. Para ello, deben seguirse los siguientes pasos:

- Abrir la consola de Firebase en: http://console.firebase.google.com
- Al añadir un proyecto, hay que darle un nombre, cambiar la ubicación del campo Analytics (opcional) y en la que se va a almacenar la base de datos (opcional). Por defecto, el campo Analytics muestra Estados Unidos y, para que se ajuste a los precios españoles, deberíamos seleccionar a España.

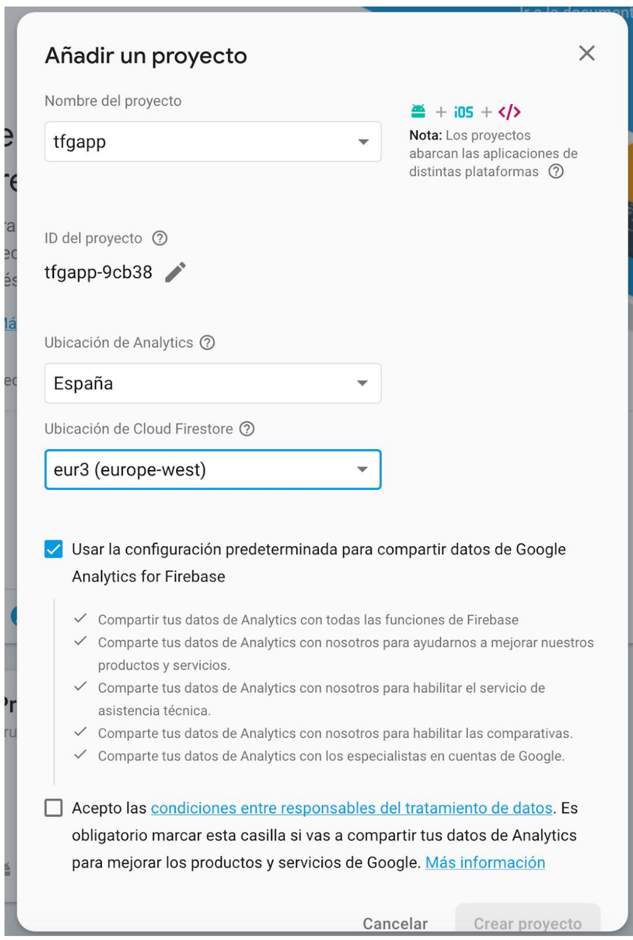

Figura 42: Pantalla de creación de proyecto en Firebase

- Una vez creado el proyecto, se deben descargar dos archivos que serán los que permitan realizar las conexiones a la base de datos desde la aplicación. En el encabezado de la página inicial se pueden ver tres iconos, dos de ellos correspondientes a iOS y Android. Si no se ve la página inicial, se puede ver abriendo las herramientas y pulsando sobre Project Overview.

Figura 43: Banner en la pantalla de inicio de Firebase

- Se deben generar los archivos para iOS y Android, ya que Flutter permite ejecutar en ambos sistemas operativos. Tras sobre los dos iconos, hay que seguir los siguientes pasos (si no se encuentra alguno de los iconos, pulsar sobre Añadir aplicación, ya que la segunda vez que añadas una aplicación no aparecerán los botones como en la imagen superior):
- Para Android:
    - Introducir el ID del paquete de la aplicación Flutter. Este ID se define al crear el proyecto, pero, si no se recuerda, se puede encontrar en la ruta *android/app/src/main/AndroidManifest.xml* (comenzando desde la raíz del proyecto).
    - Descargar el archivo google-services.json y guardarlo en la ruta *android/app/*
    - Abrir el archivo *android/app/build.gradle* y añadir al final "`apply plugin: 'com.google.gms.google-services'`". A continuación, abrir *android/build.gradle*. Dentro de las dependencias de este archivo, añadir "`classpath 'com.google.gms:google-services:3.2.1'`" (no hay que copiar las comas dobles pero sí las simples).
- Para iOS:
    - Introducir el mismo ID del paquete que en el proceso en Android.
    - Descargar el archivo GoogleService-Info.plist. Abrir el archivo *ios/Runner.xcworkspace* con Xcode, desplegar *Runner/Runner* dentro de las opciones y arrastrar el archivo a esa carpeta.

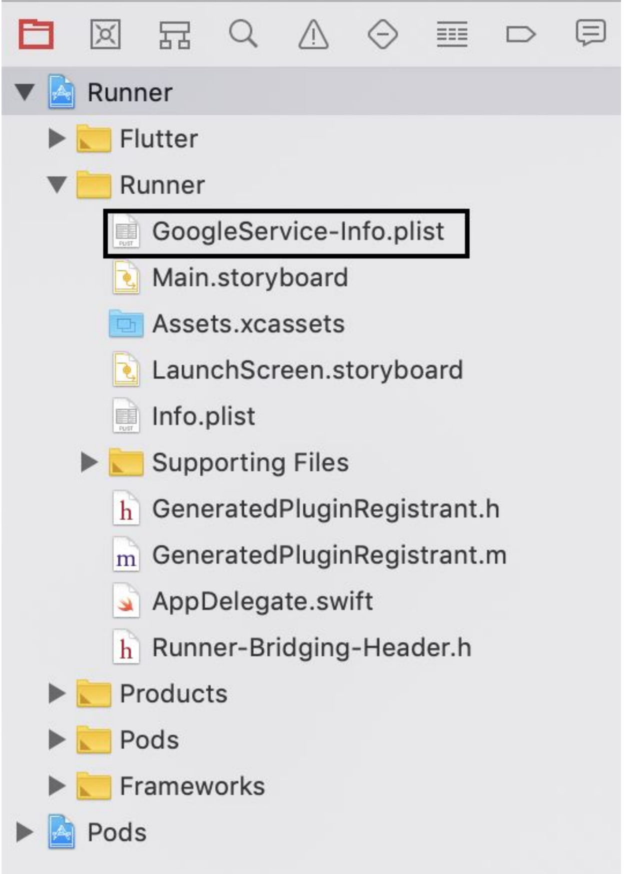

Figura 44: Estructura de ficheros en XCode

- Pulsar *continuar* y saltar la comprobación si no se tiene implementada alguna funcionalidad que haga uso de Firebase.

Tras haber seguido estos pasos, los servicios de Flutter serán integrables en el proyecto con Flutter, servicios como los de la conexión a la base de datos o la autenticación.

Anexo 5: Realizar la conexión con Firestore en Flutter

Antes de comenzar a realizar conexiones con la base de datos Firestore, se debe registrar la aplicación en Firebase tal y como se detalla en el anexo 3.

El primer paso a seguir es añadir el paquete *cloud_firestore*[43] a las dependencias en *pubspec.yaml*:

```
dependencies:
  cloud_firestore: ^0.9.5+2
```

Tras incluirlo, se pueden realizar las conexiones con la base de datos importando el siguiente paquete:

```
import 'package:cloud_firestore/cloud_firestore.dart';
```

El siguiente paso es ir a la consola de Firebase y pulsar sobre Database en la barra de la izquierda. En el panel de Cloud Firestore, pulsar sobre Create database y comenzar en modo de prueba.

Cuando se quiera llamar a la base de datos de firestore, basta con poner Firestore.instance para comenzar a hacer uso de ella. Un ejemplo de uso:

```
Widget _buildBody(BuildContext context) {
 return StreamBuilder<QuerySnapshot>(
   stream: Firestore.instance.collection('baby').snapshots(),
   builder: (context, snapshot) {
     if (!snapshot.hasData) return LinearProgressIndicator();

     return _buildList(context, snapshot.data.documents);
   },
 );
}
```

Para crear un documento nuevo en la base de datos, se debe hacer de la siguiente manera:

```
Firestore.instance.collection('books').document()
  .setData({ 'title': 'title', 'author': 'author' });
```

Si se quisiera actualizar un dato de una base de datos, teniendo la referencia de un documento, se podría hacer de la siguiente manera:

```
reference.updateData({'votes': record.votes + 1})
```

Por si se quisiera profundizar y aprender más, Google propone un pequeño curso sobre cómo conectar las aplicaciones Flutter con Firestore:

https://codelabs.developers.google.com/codelabs/flutter-firebase/#0

[43] Paquete *cloud_firestore*: https://pub.dev/packages/cloud_firestore

Anexo 6: Desarrollar una aplicación multi-idioma en Flutter

Para que una aplicación sea traducible, se debe añadir el paquete llamado *Localization*, además de un paquete que servirá para generar los archivos de traducciones. El paquete *Localozations* ya se encuentra en el sdk de flutter, por lo que no se tiene que utilizar el repositorio de paquetes. Dentro del archivo pubspec.yaml hay que añadir lo siguiente:

```
dependencies:
    flutter:
        sdk: flutter
    flutter_localizations: # add this
        sdk: flutter # add this
# other lines
dev_dependencies:
    intl_translation: ^0.17.1 # add this
```

El paquete *flutter_localizations* es el que ya se encuentra dentro del sdk de flutter y el paquete *intl_translation*, que se debe añadir bajo *dev_dependencies*, es el que ayuda a generar los ficheros de traducciones. Los paquetes bajo *dev_dependencies*, son paquetes que se utilizan durante el proceso de desarrollo, por lo tanto, no se incluye en la aplicación al instalarla en dispositivos.

Clase *AppLocalization*

Se debe crear una clase que sea la que se encargue de manejar las traducciones:

```
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'l10n/messages_all.dart';

class AppLocalizations {
  static Future<AppLocalizations> load(Locale locale) {
    final String name =
        locale.countryCode == null ? locale.languageCode : locale.toString();
    final String localeName = Intl.canonicalizedLocale(name);
    return initializeMessages(localeName).then((bool _) {
      Intl.defaultLocale = localeName;
      return new AppLocalizations();
    });
  }
  static AppLocalizations of(BuildContext context) {
    return Localizations.of<AppLocalizations>(context, AppLocalizations);
  }
  String get title {
    return Intl.message('Hello world App',
        name: 'title', desc: 'The application title');
  }

  String get hello {
    return Intl.message('Hello', name: 'hello');
  }
}
```

```
class AppLocalizationsDelegate extends LocalizationsDelegate<AppLocalizations> {
  const AppLocalizationsDelegate();
  bool isSupported(Locale locale) {
    return ['en', 'es', 'pt'].contains(locale.languageCode);
  }
  Future<AppLocalizations> load(Locale locale) {
    return AppLocalizations.load(locale);
  }
  bool shouldReload(LocalizationsDelegate<AppLocalizations> old) {
    return false;
  }
}
```

Esta clase tiene cuatro partes principales:

- la función *load* que carga el archivo en el que se incluyen las traducciones de los textos en el idioma que se especifique.
- la función *of* ayuda a acceder a los textos desde cualquier parte de la aplicación en la que se haya generado el contexto (esto quiere decir que se puede usar a partir de *build*, pero no dentro de *initState*, ya que el contexto no ha sido creado en ese momento).
- las funciones *get* que listan todos los textos traducibles en la app. Dentro de cada función *get*, hay una sola línea en la que se devuelve el mensaje que se quiera traducir.
- *initializeMessages* es el encargado de inicializar los mensajes.

En el mismo fichero hay otra clase llamada *AppLocalizationsDelegate*, que se encarga de definir qué idiomas están soportados y avisa a la aplicación si debe recargar los widgets.

De este archivo, a lo que más atención se debe prestar son las funciones *get* de los textos traducibles y la función *isSupported* para añadir los idiomas que la aplicación soporta. No hay que preocuparse por el error que aparece en este archivo, ya que se soluciona generando otros archivos más adelante.

Archivos de traducción

Se puede ver que en la clase *AppLocalizations* solo se incluyen los textos en un solo idioma, pero para que éstos se puedan traducir, se deben tener los textos en el resto de idiomas. Para hacer esto, hay que generar primero un archivo *.arb* ejecutando el comando a continuación. Los archivos *.arb* almacenan en formato *JSON* un mapa con los IDs de los textos y su traducción. Para generar este fichero, basta con ejecutar el siguiente comando en la consola:

```
flutter pub pub run intl_translation:extract_to_arb --output-dir=lib/l10n
lib/localizations.dart
```

En este comando se escriben dos directorios relativos, por lo que primero se tiene que acceder con la terminal a la ruta en la que se tenga el proyecto. Una vez ahí, se debe cambiar las rutas relativas por, primero, en vez de *lib/l10n*, la ruta en la que se quiera guardar los archivos *.arb* y, segundo, en vez de *lib/localizations.dart*, la ruta en la que se encuentre el archivo que contiene las clases *AppLocalizations* y *AppLocalizationsDelegate* (el nombre del archivo puede ser diferente).

Una vez ejecutado el comando, se genera un archivo llamado *intl_messages.arb* con todos los textos que se hayan incluido en *AppLocalizations* en la versión del idioma que se haya puesto. Este archivo puede ser copiado, cambiando en el nombre del archivo *"message"* por el código del idioma en el que vayan a estar los textos en ese archivo. Si por ejemplo se quieren tener los textos en español, euskera e inglés, deberían crearse tres archivos: *intl_es.arb*, *intl_eu.arb* e *intl_en.arb*. A continuación se pueden ver ejemplos de cómo serían los archivos de español e inglés con dos textos:

intl_es.arb

```
{
  "@@last_modified": "2018-05-25T11:50:40.743319",
  "title": "App Hola Mundo",
  "@title": {
    "description": "The application title",
    "type": "text",
    "placeholders": {}
  },
  "hello": "Hola",
  "@hello": {
    "type": "text",
    "placeholders": {}
  }
}
```

intl_en.arb

```
{
  "@@last_modified": "2018-05-25T11:50:40.743319",
  "title": "Hello world App",
  "@title": {
    "description": "The application title",
    "type": "text",
    "placeholders": {}
  },
  "hello": "Hello",
  "@hello": {
    "type": "text",
    "placeholders": {}
  }
}
```

El fichero intl_messages.arb se puede borrar de la carpeta, ya que si no se nos generará un ejecutable con él y solo servirá para ocupa espacio. Cuando se añadan más textos para traducir en *AppLocalizations*, habrá que volver a ejecutar el mismo comando para volver a generar *intl_messages.arb*. Esta vez, en vez de copiar todo el archivo, habría que copiar solo la parte correspondiente a los nuevos textos y pegarlos en los archivos que ya existen para después traducirlos.

Para finalizar, se debe ejecutar un comando que genera los ejecutables necesarios para las traducciones. Si en el paso anterior no se borra *intl_messages.arb*, también se generará un archivo ejecutable para éste. El comando a ejecutar es el siguiente:

```
flutter pub pub run intl_translation:generate_from_arb --output-dir=lib/l10n \
  --no-use-deferred-loading lib/localization.dart lib/l10n/intl_*.arb
```

De nuevo, hay que cambiar las rutas relativas por las rutas en las que se hayan guardado los archivos. Tras ejecutar este comando, se genera un fichero llamado *messages_all.dart* que dará solución al error en *AppLocalizations*. También se genera un fichero por cada *.arb* de traducción. En el caso del ejemplo habría cuatro ficheros nuevos: *messages_all.dart, messages_en.dart*, *messages_es.dart* y *messages_eu.dart*. Dentro de estos ficheros, no hay que modificat nada. Cuando cambien los textos traducidos, se debe volver a ejecutar este comando (tras haber cambiado los ficheros .arb) y automáticamente cambiarán los ficheros ejecutables.

Utilización de las traducciones

Una vez se han generado todos los ficheros para que funcionen las traducciones, se pueden realizar los cambios necesarios en el código de la aplicación para que funcionen. Los primeros cambios se deben realizar en el fichero *main.dart.* Se debe modificar de la siguiente manera:

```
import 'package:flutter/material.dart';
import 'package:flutter_localizations/flutter_localizations.dart';

import 'localizations.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      locale: Locale('es'),
      localizationsDelegates: [
        AppLocalizationsDelegate(),
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate
      ],
      supportedLocales: [Locale("en"), Locale("es"), Locale("pt")],
      onGenerateTitle: (BuildContext context) =>
          AppLocalizations.of(context).title,
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new MyHomePage(title: AppLocalizations.of(context).title),
    );
  }
}
```

Los cambios realizados son los siguientes:

- Se importa el fichero localizations.dart que es el que incluye AppLocalizations.
- Se añade dentro del Widget MaterialApp:
    - locale, donde se especifica el idioma que se va a mostrar.

- localizationsDelegates, con la clase nueva.
- supportedLocales, donde se especifica qué idiomas son soportados.
- una traducción que sirve de ejemplo para ver cómo se obtienen los textos (en el título.

Se puede encontrar más información en el siguiente enlace:

https://proandroiddev.com/flutter-localization-step-by-step-30f95d06018d