UNIVERSIDAD AUTONOMA
DE MADRID

Escuela Politécnica Superior

20
21

Trabajo fin de grado

Implementación de una aplicación de mensajería para situaciones de crisis para personas con TEA

Andrea Navacerrada Terol

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

Universidad Autónoma de Madrid

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Implementación de una aplicación de mensajería para situaciones de crisis para personas con TEA

Autor: Andrea Navacerrada Terol
Tutor: Javier Gómez Escribano

junio 2021

Todos los derechos reservados.
Queda prohibida, salvo excepción prevista en la Ley, cualquier
forma de reproducción, distribución comunicación pública
y transformación de esta obra sin contar con la autorización
de los titulares de la propiedad intelectual.
La infracción de los derechos mencionados puede ser constitutiva de
delito contra la propiedad intelectual *(arts. 270 y sgts. del Código Penal)*.

DERECHOS RESERVADOS
© 1 de Mayo de 2021 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, nº 1
Madrid, 28049
Spain

Andrea Navacerrada Terol
Implementación de una aplicación de mensajería para situaciones de crisis para personas con TEA

Andrea Navacerrada Terol
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A todas aquellas personas que se superan cada día

Todos somos muy ignorantes,
lo que ocurre es que no todos ignoramos las mismas cosas.

Albert Einstein

Agradecimientos

Quiero dar las gracias a mi pareja y a mi familia por confiar siempre en mí y apoyarme en todas mis decisiones, en especial a mi padre por haberme impulsado a estudiar esta carrera que tanto me ha gustado. A mis amigas con las que he celebrado y celebraré todos mis logros, así como me han apoyado en mis aprendizajes, pues así me gusta llamar a los fracasos. A mis compañeros, a los que ahora puedo llamar amigos, y especialmente a mis compañeras de fatigas Adri y Cande, porque sin ellos nada hubiera sido igual. Y por último, a todos esos profesores que disfrutan enseñando y viendo como sus alumnos prosperan, especialmente a mi tutor, Javier Gómez Escribano, que me ha acompañado en este trabajo, que supone el cierre de un ciclo, para dar paso a una nueva etapa de mi carrera profesional.

Resumen

Muchas de las personas con trastorno de espectro autista, además de sufrir ansiedad de manera más frecuente que los neurotípicos, personas sin trastorno de espectro autista, presentan también dificultades para comunicarse con normalidad. No obstante, el hecho de que no puedan hacerlo no significa que no quieran o que no lo necesiten.

Con este problema encima de la mesa, el objetivo de este trabajo es realizar la implementación de una aplicación, que ha sido analizada previamente, con el propósito de cubrir las necesidades de este colectivo.

Para poder llevar a cabo dicha implementación, se han realizado revisiones del análisis previo, así como se han analizado diferentes alternativas de tecnologías destinadas al desarrollo de aplicaciones móviles con el fin de elegir la más adecuada a las circunstancias.

Finalmente, se ha desarrollado una aplicación Flutter para Android, llamada Conecta Con TEA, que cumple con los requisitos mínimos para ser probada con usuarios finales. Dado que se ha seleccionado un framework multiplataforma, en futuras versiones se podrá escalar a diferentes plataformas.

Palabras clave

TEA, autismo, mensajería instantánea, situaciones de estrés, aplicación

Abstract

Many people with autism spectrum disorder, in addition to suffering from anxiety more frequently than neurotypicals, people without autism spectrum disorder, also have difficulty communicating normally. However, the fact that they cannot do so does not mean that they do not want or need to.

With this problem on the table, the aim of this work is to implement an application, which has been previously analyzed, in order to meet the needs of this group.

In order to carry out this implementation, revisions of the previous analysis have been made, as well as different alternatives of technologies for the development of mobile applications have been analyzed in order to choose the most suitable for the circumstances.

Finally, a Flutter application for Android, called Conecta Con TEA, has been developed, which meets the minimum requirements to be tested with end users. Since a cross-platform framework has been selected, in future versions it can be scaled to different platforms.

Keywords

ASD, autism, instant messaging, stressful situations, app

Índice

1 Introducción **1**

1.1 Motivación . . . 1

1.2 Objetivos . . . 2

1.3 Estructura del documento . . . 2

2 Estado de la Tecnología **3**

2.1 Aplicaciones Nativas y Multiplataforma . . . 3

2.2 Frameworks para el desarrollo de aplicaciones Multiplataforma . . . 4

2.2.1 Comparativa Flutter vs. React Native . . . 4

3 Diseño **7**

3.1 Revisión del análisis de requisitos . . . 7

3.1.1 Requisitos Funcionales . . . 7

3.1.2 Requisitos No Funcionales . . . 9

3.2 Revisión del prototipo . . . 10

4 Implementación **11**

4.1 Tecnologías utilizadas . . . 11

4.1.1 Flutter. Comunicación entre widgets. . . . 11

4.1.2 Almacenamiento de datos . . . 13

4.2 Estructura del proyecto . . . 16

4.3 Descripción de la implementación . . . 18

4.3.1 Ejecución de la aplicación . . . 18

4.3.2 Autenticación . . . 21

4.3.3 Vistas para usuarios autenticados . . . 26

5 Pruebas **37**

6 Conclusiones y Trabajo Futuro **39**

6.1 Conclusiones . . . 39

6.2 Trabajo futuro . . . 40

Bibliografía **42**

Acrónimos **43**

Apéndices **45**

A Estadísticas Stack Overflow **47**

Listas

Lista de figuras

2.1 Estadística frameworks multiplataforma 4
2.2 Arquitectura Flutter 6
2.3 Arquitectura React Native 6

4.1 Comunicación con setState 12
4.2 Comunicación con Provider 13
4.3 Árbol de widgets 14
4.4 Esquema Firestore 15
4.5 Esquema base de datos local 16
4.6 Directorio raíz 16
4.7 Directorio lib 17
4.8 Función main 18
4.9 Widget MyApp 19
4.10 Método build 20
4.11 Interfaz AuthenticationProvider 21
4.12 Interfaz FirestoreProvider 22
4.13 Interfaz StorageProvider 22
4.14 Interfaz MessagingProvider 22
4.15 Pantallas iniciales 23
4.16 LandingPage 24
4.17 SignInPage 25
4.18 SignIn y SignUp 26
4.19 Boton llamar 28
4.20 Contacto de emergencia 29
4.21 Ajustes 31
4.22 Contactos 32
4.23 Navegación 33
4.24 Inputs 35

5.1 Rendimiento 38

A.1 Empezar con Flutter y React Native 47
A.2 Continuar con Flutter y React Native 48

A.3 No continuar con Flutter y React Native 49

Lista de tablas

3.1 Requisitos funcionales 8
3.2 Requisitos funcionales 8
3.3 Requisitos funcionales 9
3.4 Requisitos no funcionales 10
3.5 Requisitos no funcionales 10

1

INTRODUCCIÓN

Toda aplicación debe pasar por las fases de análisis y diseño antes de su implementación, y este caso no será diferente. En el trabajo de Álvaro Martínez [1] se ha realizado el análisis de viabilidad, así como la definición de requisitos y el diseño del prototipo de una aplicación optimizada para situaciones de estrés y para la comunicación basada en pictogramas destinada a personas con Trastorno de Espectro Autista (TEA) . Tras los resultados obtenidos, en este trabajo se continuará el ciclo pasando a la fase de implementación, para lo que será necesario revisar el análisis y diseño definidos en las etapas anteriores.

Antes de empezar con la parte más técnica del proyecto, es interesante entender las necesidades de las personas con TEA para determinar si realmente es necesaria la herramienta que se va a desarrollar, y en su caso hacerla lo más accesible posible. Además, es preciso marcar el objetivo de este trabajo y los puntos de acción necesarios para llegar a él.

1.1. Motivación

Las personas con TEA tienen dificultades para desarrollar con normalidad sus habilidades sociocomunicativas, sin embargo, eso no significa que su necesidad de comunicar sea menor que la de cualquier otra persona. Además de estas dificultades, también presentan un mayor riesgo de padecer ansiedad [2].

Naoki Higashida es una persona con autismo que tenía grandes dificultades para comunicarse, sin embargo, trabajando con diferentes métodos de comunicación, distintos de la comunicación oral, empezó a desarrollar sus habilidades comunicativas. En su libro plasma una gran variedad de preguntas y respuestas, verdaderamente ilustrativas, que permiten entender, en cierta medida, por lo que pasan estas personas en su día a día. Una de las tantas citas que se podrían destacar es “Pero no es que no es que no hablemos: es que no podemos hablar y eso nos hace sufrir” [3]. Es cierto que se podrían mencionar muchas más, no obstante, con esta única frase se ilustran los problemas que se mencionaban en el párrafo anterior.

Dadas las dificultades de este colectivo surge la idea de crear una herramienta de comunicación

adaptada a las necesidades especiales de las personas con TEA. Hoy en día, la tecnología está al alcance de muchos, por lo que obtener cualquier aplicación no supone un gran esfuerzo, sin embargo, una vez hecho el estudio se observa que en el mercado actual no existe un repertorio de aplicaciones de esta línea [1], por ello, se ha decido llevar a cabo la implementación la herramienta analizada y diseñada en las etapas previas.

1.2. Objetivos

El objetivo de este trabajo es implementar un prototipo de una aplicación de mensajería instantánea adaptada para personas con TEA, donde sea posible la comunicación a través de pictogramas así como con texto. Además, el usuario debe tener acceso en todo momento a un botón de marcación rápida a un contacto previamente configurado para posibles situaciones de estrés.

Para alcanzar el objetivo general se han fijado los siguientes objetivos específicos:

- Revisar el análisis y diseño definidos en las fases anteriores.
- Analizar las diferentes opciones en tecnologías dedicadas al desarrollo de aplicaciones.
- Decidir qué tecnologías se utilizarán para llevar a cabo la implementación.
- Implementar un primer Minimum Viable Product (MVP) , en español producto viable mínimo, de la aplicación diseñada con las tecnologías definidas.
- Hacer pruebas de la implementación.

1.3. Estructura del documento

El cuerpo de este documento se compone de seis capítulos. En primer lugar, en el capítulo 1 se encuentra una breve introducción acerca de este trabajo, se describe cual es la motivación para llevar a cabo este proyecto y se detallan los objetivos necesarios para conseguirlo. Más adelante, se encuentra el capítulo 2, de estado de la tecnología, en el que se analizan diferentes alternativas en tecnologías destinadas a cubrir el objetivo final de este proyecto, que es implementar una aplicación. Para continuar, en el capitulo 3, se realiza una revisión de los resultados obtenidos en el trabajo anterior acerca del diseño y la definición de requisitos. Tras ver estos puntos, en el capítulo 4 se expone la implementación de la aplicación a construir. En el se exponen las tecnologías utilizadas, la estructura que sigue el proyecto y las diferentes partes de la implementación. Posteriormente, existe un capítulo de pruebas (capítulo 5), donde se comentan las pruebas realizadas para confirmar el correcto funcionamiento de la implementación, y el cumplimiento de los requisitos definidos. Y por último, en el capítulo 6, se exponen las conclusiones obtenidas así como posibles lineas de trabajo futuro.

2

Estado de la Tecnología

Para emprender la implementación, es necesario analizar las principales tecnologías actualmente destinadas al desarrollo de aplicaciones. Por ello, se dedicará este capítulo a realizar dicho análisis.

2.1. Aplicaciones Nativas y Multiplataforma

Actualmente existen múltiples alternativas para desarrollar una aplicación, la primera cuestión que se ha planteado para empezar a desarrollar la aplicación es si se va a implementar con código nativo o con un framework multiplataforma.

Las aplicaciones nativas están implementadas en el lenguaje oficial del sistema operativo de cada dispositivo. Por ejemplo, las aplicaciones de Android se desarrollan en java o Kotlin, y las del sistema operativo iOS, en objetive-c o Swift. La aplicación nativa optimizada para Android solo se podrá instalar en un dispositivo Android y de la misma manera para iOS. Por otro lado las aplicaciones multiplataforma se podrán ejecutar en distintas plataformas, con un único código, con mínimas adaptaciones para cada una de las plataformas.

Ambas opciones son válidas y tienen diferentes ventajas [4]. Posteriormente, para tomar una decisión, habrá que centrarse en las características que aportan valor para este desarrollo en concreto.

Algunas ventajas de las aplicaciones nativas frente a las aplicaciones multiplataforma:

- Tienen mejor rendimiento. Dado que están optimizadas para cada sistema operativo, la comunicación con las diferentes librerías de este así como con los recursos hardware es directa.
- La experiencia de usuario puede ser más satisfactoria que en aplicaciones multiplataforma, puesto que, en estas últimas no siempre se podrán adaptar las aplicaciones a la apariencia definida por cada sistema operativo, sino que en muchos caso, tendrán un diseño general en todos los dispositivos, sin embargo, las aplicaciones nativas siempre seguirán las guías de diseño de la plataforma.

Algunas ventajas de las a las aplicaciones multiplataforma frente a las aplicaciones nativas:

- Multiplataforma. El código es reutilizable, se desarrolla un único código para los distintos sistemas operativos. Por tanto, aporta mayor flexibilidad para escalar la aplicación a nuevos sistemas operativos.
- Puesto solo existe un único código, el coste y esfuerzo del desarrollo se reduce significativamente.

Dadas estas características, teniendo en cuenta que en el trabajo anterior se ha definido como requisito que la aplicación se pueda ejecutar tanto en dispositivos Android como iOS, puesto que no se dispone del tiempo necesario para formarse en las distintas variantes que requiere implementar aplicaciones nativas para distintas plataformas, se ha decidido llevar a cabo la implementación con un framework multiplataforma.

2.2. Frameworks para el desarrollo de aplicaciones Multiplataforma

Como se ha mencionado en la sección 2.1, en este trabajo se va a utilizar un framework multiplataforma, por lo que se van a analizar diferentes opciones existentes en el mercado actual.

En la figura 5.1 los frameworks más utilizados durante el 2019 y 2020 han sido React Native, Flutter, Cordova, Ionic y Xamarin. Sin embargo, el uso de los tres últimos en el último año ha sufrido una caída importante, mientras que React Native se ha mantenido y Flutter ha aumentado de manera significativa, vistos los datos, se ha decidido hacer una comparativa de React Native y Flutter.

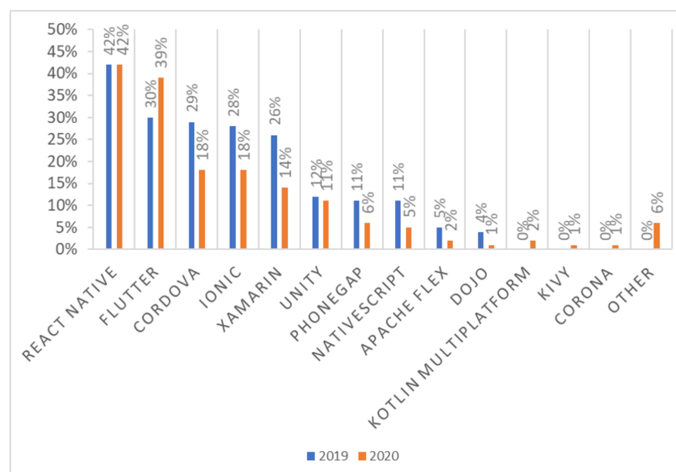


Figura 2.1: Frameworks multiplataforma usados por los desarrolladores de software en el mundo en 2019 y 2020. Fuente: “Statista, 2021”

2.2.1. Comparativa Flutter vs. React Native

Flutter lanzó su primera release estable en febrero de 2018 [5], casi tres años después que React Native que la lanzó en junio de 2015 [6], por lo que, probablemente, la comunidad de React Native es mayor que la de Flutter. Sin embargo, como se puede observar el la Figura 5.1, es un hecho que el uso Flutter está creciendo rápidamente y por consiguiente su comunidad.

Según los datos recogidos por Stack Overflow [7], lugar de encuentro de millones de desarrolla-

dores, en 2020, el porcentaje de desarrolladores que aún no habían utilizado el framework de React Native, pero que mostraban interés por hacerlo, era 14,00 %, en un rango de 1,50 % el framework que menos obtenía y un 18,10 % el que más, posicionándose por encima de Flutter, que obtenía un 10,70 % de los desarrolladores que no han trabajado con este.

También se recogieron los datos de desarrolladores que estaban usando un lenguaje en específico, y que además, mostraban interés en continuar desarrollando con dicho lenguaje. En este caso, Flutter, con un 68,80 %, se posicionaba por encima de React Native, este con un 57,90 %, teniendo en cuenta que el mayor porcentaje obtenía tenía un 71,50 % y el que menor un 27,60 %. Además de esto, también se obtuvo el porcentaje de desarrolladores que no mostraban interés en continuar utilizando un lenguaje en concreto, en este caso, Flutter obtenía un menor porcentaje, 31,20 %, frente a React Native que alcanzaba un porcentaje de 42,10 %, siendo la cota inferior 28,50 % y la superior 72,40 % para otros frameworks o lenguajes.

Aunque la curva de aprendizaje de ambos no es demasiado alta, y teniendo en cuenta que esto puede variar dependiendo de los conocimientos previos y experiencia de cada desarrollador, analizando los datos recogidos en estas encuestas, se podría decir que existe un menor temor y un mayor interés para empezar a desarrollar con React Native que con Flutter. Sin embargo, en cuanto a desarrolladores que ya están utilizando los distintos frameworks, se podría decir que Flutter es el preferido y que tiene un mejor futuro que React Native. Por lo tanto, si continua esta tendencia, la comunidad de Flutter crecerá considerablemente incluso llegando a superar la de React Native.

En cuanto al rendimiento, a pesar de que ambos frameworks prometen un rendimiento competitivo, tanto el de Flutter como el de React Native será, siempre, peor que el de una aplicación nativa. El factor que marca la diferencia entre el rendimiento de ambos frameworks es la arquitectura y la forma en que se ejecuta cada uno de ellos. En el caso de React Native la comunicación entre React Native y la plataforma no es directa, necesita un puente que traduce el código escrito en código nativo [8]. Esto afecta directamente a la inicialización de la aplicación ya que esta traducción no es inmediata. Sin embargo, el código de Flutter es pre-compilado y se comunica directamente con la plataforma [9] por lo que la inicio y rendimiento será mejor que el de React Native. En las figuras 2.2 y 2.3 se muestra como es la comunicación entre los diferentes componentes de Flutter y React Native respectivamente.

Ambos frameworks son de código abierto por lo que es posible crear una aplicación con cualquiera de ellos con los recursos mínimos. Además ambos tienen la propiedad de Hot Reload [10] [11], la cual permite ver inmediatamente los cambios que se hagan durante la ejecución. Esto hace que la experiencia de desarrollo sea mucho más cómoda evitando reiniciar constantemente la ejecución mientras se depura y se comprueba el funcionamiento. Dart, lenguaje en el que está basado Flutter, cuenta también con un inspector [12] que permite depurar el código, así como visualizar diferentes datos de la aplicación, como el árbol de widgets, el rendimiento de cada acción o el tamaño de esta, entre otros.

Finalmente, se podría decir que, actualmente, React Native es el más utilizado de ambos fram-

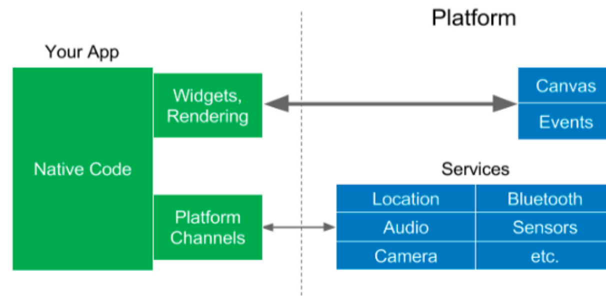


Figura 2.2: Arquitectura de Flutter. Fuente: "Medium"

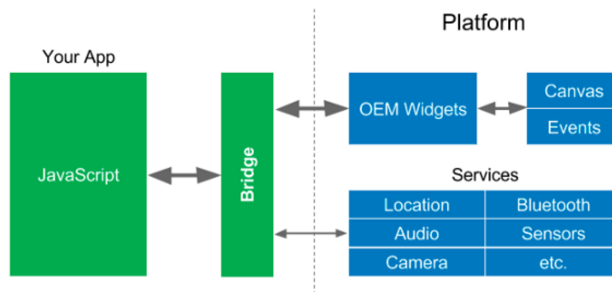


Figura 2.3: Arquitectura de React Native. Fuente: "Medium"

works, pero sin embargo, de acuerdo con las tendencias, esto podría cambiar en un futuro muy cercano. Con lo que respecta al rendimiento, aunque ninguno obtendrá mejores resultados que una aplicación nativa, una misma aplicación hecha en Flutter y en React Native debería de alcanzar mayor rendimiento en Flutter dada la comunicación directa con la plataforma. Y por último cabe destacar que los dos frameworks aportan herramientas para mejorar la experiencia de desarrollo, sin embargo, Flutter contaría con más herramientas dedicadas a la de depuración e inspección que harán el desarrollo mucho más sencillo y eficiente.

3

Diseño

3.1. Revisión del análisis de requisitos

En el trabajo que precede a este, se hizo un análisis en el que se definieron requisitos funcionales y no funcionales para la herramienta [1]. En esta segunda parte del proyecto, en la que vamos a implementar un primer prototipo de la herramienta, es necesario hacer una revisión de dichos requisitos y determinar el alcance que tendrá esta implementación.

3.1.1. Requisitos Funcionales

En la tabla 3.1 se encuentran los requisitos funcionales que se encuentran dentro del alcance de esta implementación, y por lo tanto, con los que el producto final debería contar.

Por otro lado, en la tabla 3.2 están los requisitos funcionales que se van a contemplar en este trabajo, pero que sin embargo, experimentarán alguna modificación. A continuación se aclara cuáles son estas modificaciones.

Es cierto que la aplicación contará con una pantalla de inicio, no obstante, esta primera versión no contará con la opción de “Relajarse”. Por lo tanto, en cuanto al requisito RF04 (tabla 3.2), en la pantalla de inicio se encontrará el botón de “Hablar”, de acuerdo con lo definido, con el que se podrá navegar a la pantalla que muestra la lista de contactos, y además se reemplazará el botón de “Relajarse” por un botón de acceso directo al contacto de emergencia. En el requisito RF08 (tabla 3.2) se específica que se permitirá al usuario editar algunos aspectos del perfil del usuario. Nuevamente, en esta definición se hará una modificación, pues a pesar de que si se permitirá al usuario editar la imagen para su avatar y el nombre, no habrá la opción de escribir una frase de estado ya que no se ha especificado ningún lugar en el que mostrar está frase, y además tampoco se considera que esta funcionalidad aporte valor a una aplicación destinada a personas que se comunican, en mayor medida, mediante pictogramas. Y por último, el requisito RF10 (tabla 3.2), al igual que los dos anteriores, será contemplado con una variante, y es que no se implementará el envío de mensajes de audio, dado que está aplicación está principalmente a destinadas personas con dificultades en la comunicación oral, se ha considerado que

ID Requisito	Definición
RF01	Registro de usuarios. La aplicación contará con una pantalla que permita el registro de usuarios. Se permitirá el registro de usuarios mediante Google.
RF02	Botón de emergencia. La aplicación contará con un botón que actúe de llamada de emergencia a un teléfono configurado previamente.
RF03	Configuración. La aplicación contará con una pantalla de ajustes, donde se podrán personalizar varios aspectos relativos a la herramienta.
RF06	Pantalla de chat online. La aplicación contará con una pantalla que ilustre la lista de contactos.
RF07	Foto de contacto personalizable. La aplicación deberá dar la opción de personalizar la foto con la que los usuarios guardan a sus contactos, de modo que les sea más fácil reconocerlos. Por defecto, la foto será la que tenga el contacto en su perfil.
RF09	Conversación. La aplicación contará con una pantalla en la que se muestre la conversación con un cierto contacto. Se visualizará la foto del contacto así como una barra que habilite el envío de mensajes.

Tabla 3.1: Requisitos funcionales dentro del alcance de esta entrega.

ID Requisito	Definición
RF04	Pantalla de inicio. La aplicación contará con una pantalla de inicio en la que habrá dos botones “Relajarse” y “Hablar”.
RF08	Perfil de usuario. La aplicación contará con una pantalla que permita editar el perfil del usuario la foto y una frase de estado.
RF10	Tipos de mensaje. La aplicación permitirá enviar mensajes de texto, audios, mensajes basados en pictogramas, y la ubicación actual del usuario.

Tabla 3.2: Requisitos funcionales con modificaciones dentro del alcance de esta entrega.

en esta primera entrega no es una funcionalidad esencial, y que, aun así, se podría incluir en siguientes entregas si esto fuera necesario para el usuario final. Y por último, en cuanto a requisitos funcionales se refiere, en la tabla 3.3 se sitúan aquellos requisitos que no se situarán dentro del alcance de esta primera entrega, y que por lo tanto no se implementarán. Se ha decidido dejar fuera el requisito RF05 (tabla 3.3) porque se considera que para que esta funcionalidad tenga valor, es necesario realizar un estudio más profundo acerca de técnicas de relajación y como esas técnicas pueden ayudar a las personas con TEA, e implementar una sección basada en dichos estudios. En cuanto al requisito RF11 (tabla 3.3), aunque en efecto, la aplicación contará con una selección de pictogramas, no se permitirá, por el momento, la introducción de pictogramas propios. También estará fuera del alcance el requisito RF12 (tabla 3.3), no obstante, se podrá incorporar en futuras versiones.

ID Requisito	**Definición**
RF05	Pantalla de relajación. La aplicación contará con una pantalla de relajación, donde una foto, o un GIF personalizable se mostrará. En esta pantalla además, se darán indicaciones genéricas de relajación.
RF11	Pictogramas. La aplicación permitirá la introducción de pictogramas para que los pueda usar la persona con TEA. No obstante, la aplicación contendrá por defecto varios pictogramas comunes.
RF12	Tamaño de fuente. La aplicación permitirá cambiar el tamaño de fuente de todos los textos de la aplicación.

Tabla 3.3: Requisitos funcionales fuera del alcance de esta entrega.

3.1.2. Requisitos No Funcionales

Al igual que con los requisitos funcionales, se ha hecho una revisión de los no funcionales para determinar cuáles estarían dentro o fuera del alcance de este trabajo.

La tabla 3.4 recoge los requisitos no funcionales que si estarán dentro del alcance del proyecto, y del mismo modo que los funcionales, al final del proyecto, en el periodo de pruebas, se deberá comprobar que estos se cumplan. Sin embargo, para confirmar que se cumplen los requisitos RFN03, RNF04 y RNF09 (tabla 3.4), sería necesario recibir feedback de usuarios finales, y esto no se llevará a cabo en este proyecto.

Por último, en la tabla (tabla 3.5) se recopilan los requisitos no funcionales que no se contemplarán en esta entrega. En cuanto al requisito RNF02 (tabla 3.5), se ha descartado porque no se cuentan con los recursos necesarios para desarrollar la aplicación para el sistema operativo iOS. Sin embargo, se ha tenido en cuenta este requisito para futuras versiones, por lo que se ha elegido implementar la aplicación con un framework multiplataforma de manera que cubrir esta necesidad sea lo más factible y simple posible. Y finalmente, por no contar con el tiempo suficiente y al ser esta versión un prototipo

ID Requisito	Definición
RNF01	Configurable. La herramienta debe ser personalizable de acuerdo con lo establecido en la sección 6.2.6.
RNF03	Navegabilidad. La navegación entre las diferentes pantallas de la aplicación debe ser simple e intuitiva.
RNF04	Usabilidad y accesibilidad. La aplicación tendrá los elementos necesarios para ser accesible al mayor número de personas.
RNF06	Tiempo de respuesta. La aplicación deberá llevar a cabo las acciones solicitadas por el usuario en un tiempo menor a 200ms
RNF07	Espacio. La aplicación ocupará un espacio variable dentro de la memoria del dispositivo móvil como consecuencia del RF11.
RFN09	Sencillez. La aplicación mantendrá un diseño minimalista y simple.

Tabla 3.4: Requisitos no funcionales dentro del alcance de esta entrega.

y no una solución publicable, no se tendrán en cuenta los requisitos RNF05 y RNF08 (tabla 3.5).

ID Requisito	Definición
RNF02	Sistema operativo. La aplicación debe ser ejecutable tanto en sistemas iOS como en sistemas Android.
RNF05	Protección de datos. La aplicación seguirá y respetará la política de protección de datos vigente.
RNF08	Seguridad. La aplicación mantendrá privadas y cifradas las conversaciones entre dos usuarios.

Tabla 3.5: Requisitos no funcionales fuera del alcance de esta entrega.

3.2. Revisión del prototipo

En el trabajo anterior se definieron también una serie de maquetas que ilustraban el resultado final de la aplicación. De nuevo, estas maquetas han sido revisadas, y para esta implementación, se ha elaborado un nuevo prototipo, que incluye los requisitos indicados en la sección 3.1.1, modificando la vertiente estética, pero respetando la disposición de los componentes que seguían las anteriores maquetas, e integrando pictogramas como fondo de la mayor parte de los botones, con el fin de obtener una solución más intuitiva. Además, se ha determinado un nombre para dicha aplicación. Para tomar esta decisión se ha hecho una lluvia de ideas de palabras clave, como pueden ser autismo, hablar, comunicar, ayudar, superar, y de posibles composiciones con estas palabras clave. Finalmente, el nombre que se ha formulado, y por consiguiente, con el que se han construido las maquetas, es Conecta Con TEA.

4

Implementación

4.1. Tecnologías utilizadas

Como se ha mencionado en la sección 3.1.2, aunque en este trabajo no sea posible implementar la aplicación para el sistema operativo iOS, en futuras evoluciones si se podrá dar una solución para cubrir esta plataforma, por lo que se utilizará un framework multiplataforma. Además según las características de los dos frameworks comparados en la sección 2.2.1 se ha decidido realizar la implementación con Flutter para lo que será el necesario instalar el SDK (Software Development Kit, traducido al español como kit de desarrollo de software) de Flutter [5], así como el SDK de Dart [13]. Junto con este de este framework se utilizarán los servicios de la plataforma Firebase [14]. Puesto que ambas tecnologías pertenecen a Google están perfectamente integradas, y cuentan con una documentación específica para usar la API de Firebase con Flutter [15].

Para la edición del código se utilizará el editor Visual Studio Code [16], que cuenta con extensiones para Flutter y Dart, que hacen la experiencia de desarrollo mucho más sencilla y práctica.

Será necesario también contar con al menos un emulador [17] para visualizar y probar la aplicación.

Además de lo anterior, también se precisará de un proveedor de pictogramas, que en este caso se utilizará la API de ARASAAC [18] que proporcionará todos los pictogramas que contenga la aplicación.

4.1.1. Flutter. Comunicación entre widgets.

En Flutter todos los elementos de la interfaz son widgets. En ellos se describe como se visualizará la interfaz gráfica. Se pueden distinguir dos grandes conjuntos, por un lado se encuentran los StatelessWidgets, que como su nombre indica son widgets sin estado, y por otro lado están los StatefulWidgets, que al contrario que los anteriores si tendrán estado. Esto es así porque en la interfaz hay elementos dinámicos, que según los datos que se encuentran por debajo de la interfaz, se tendrán que mostrar de una manera u otra, los cuales serán representados con StatefulWidgets, pero también hay elementos estáticos que no cambian nunca independientemente del back-end de la aplicación, por lo que todos estos elementos deberán ser trazados por StatelessWidgets. Haciendo un buen uso de esta

característica, se puede lograr un mejor rendimiento ya que no será necesario renderizar todos los widgets cada vez que se produzca un evento, sino que simplemente se reconstruirán los que tengan estado y se vean afectados por algún evento. [19]

Para manejar los cambios de estado es necesario conocer la metodología que lo permita. Los propios StatefulWidgets cuentan con el método setState que se encarga de asignar el nuevo estado y notificar al widget para que se reconstruya inmediatamente. Este método es el más básico, y con él se podría construir una aplicación, sin embargo, no es la opción más recomendable, ya que no se podrá separar la implementación de la lógica de negocio de la implementación de la interfaz, consiguiendo un código mucho menos mantenible, legible y reutilizable que con otras opciones. Además de esto, la comunicación entre los distintos widgets también será más compleja ya que como se muestra en la figura 4.1, para notificar a los demás widgets, tendrá que recorrerse todo el árbol, hasta llegar al widget que se quiere realmente notificar, pasando por otros a los que no les interesa ese cambio. No obstante, Flutter ofrece un repertorio de opciones para tratar este aspecto. [20]

Figura 4.1: Comunicación entre widgets con el método setState

Entre las opciones disponibles, en esta implementación se utilizará el patrón Provider para abordar este problema. Esta alternativa está recomendada por Flutter, ya que es ideal para aplicaciones sencillas o implementaciones de MVPs y para desarrolladores que no tiene mucha experiencia con el framework [21]. Al contrario que con la variante anterior, con el patrón Provider es posible abstraer en cierta medida la parte lógica de la gráfica. Además la comunicación entre widgets es mucho más sencilla y eficiente, ya que los widgets que consuman del provider escucharan los cambios, evitando así recorrer todo el árbol e impactar a widgets que no deberían de sufrir cambios. En la figura 4.2 se muestra cómo sería esta comunicación. En este caso el nodo raíz, marcado con la letra P y de color naranja, sería el provider. En el primer nodo a la izquierda de color azul dispararía el evento que actualizaría los datos en el provider y, directamente, se notificaría a los nodos consumidores, que en este caso serían los dos nodos de color azul de la derecha. Cabe destacar que para que un widget pueda consumir de un provider se tiene que encontrar debajo del mismo en el árbol de widgets.

Una vez entendida y definida la metodología que se va a emplear, se define como debería de ser

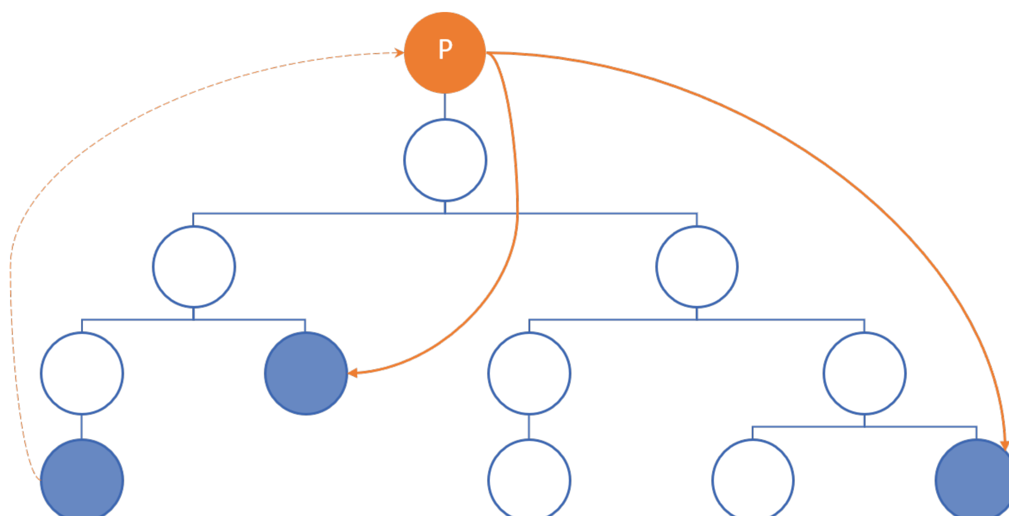


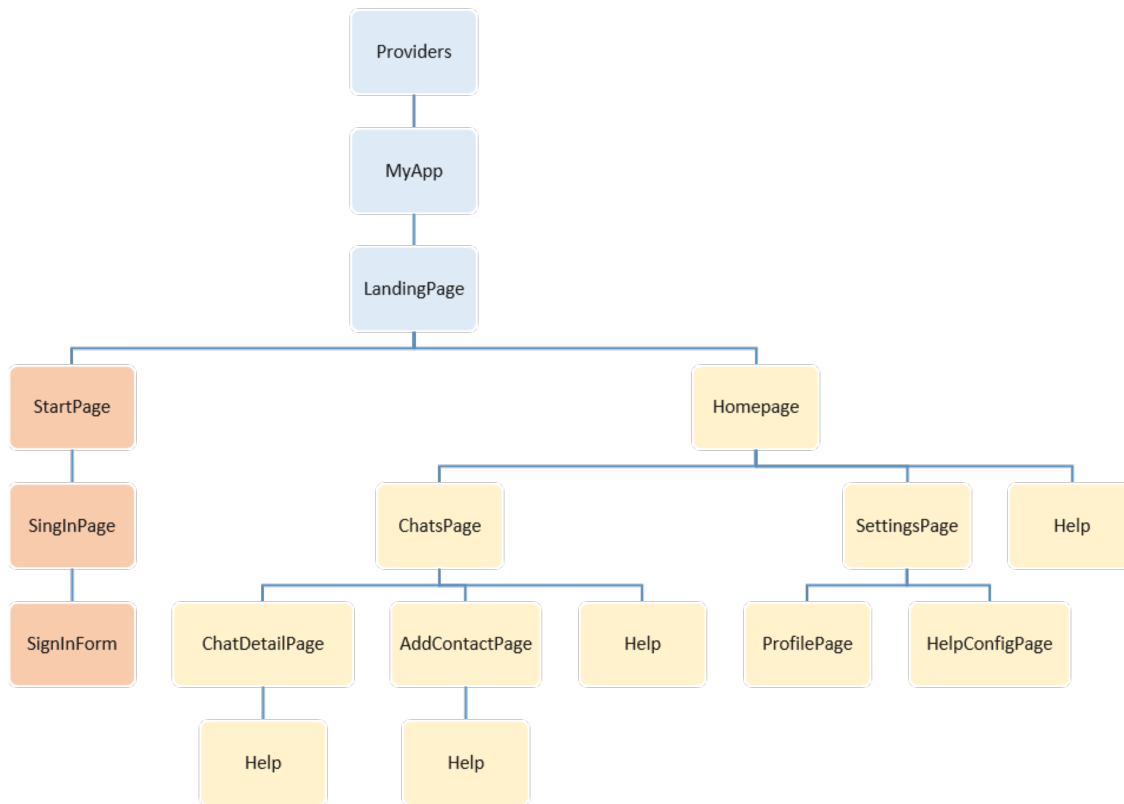
Figura 4.2: Comunicación entre widgets con Provider

el árbol de widgets de una manera simplificada. En la Figura 4.3, se muestra la jerarquía que debería de seguir la aplicación, en ella la figura 4.3(a) representa el árbol de widgets principales, y la figura 4.3(b) constituye el nodo Help del árbol. Los providers se declararán en la raíz de la aplicación para que sean accesibles por todos los widgets. Con el nodo LandingPage se hará la distinción necesaria para mostrar las vistas dedicadas a usuarios no autenticados o autenticados. No habrá dos widgets diferentes para el inicio de sesión y el registro de usuarios, por motivos de reutilización de código. Para la llamada de emergencia también se utilizará un widget, en este caso el LandingCall, que muestre la pantalla de configuración del contacto de emergencia, o la de la llamada en función de si están configurados o no los datos de dicho contacto. El acceso a esta funcionalidad se podrá hacer desde todas las pantallas, para un usuario autenticado, excepto desde las pantallas de ajustes.

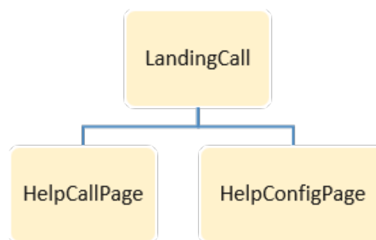
4.1.2. Almacenamiento de datos

Se usarán los servicios de Firebase para la autenticación de usuarios, almacenamiento y envío de notificaciones. La autenticación estará gestionada por Firebase Authentication. Se utilizarán dos diferentes servicios para el almacenamiento, por un lado, Firestore, donde se encontrará la base de datos NoSQL para los datos de los usuarios, en la figura 4.4 se puede ver el esquema que seguirá esta base de datos. Y por otro lado, Storage, donde se almacenarán objetos multimedia, en este caso los únicos archivos multimedia que se almacenarán serán las imágenes de avatar de cada usuario, y las de sus contactos si así se requiriera. También se empleará el servicio Firebase Cloud Messaging (FCM) con el que se administrará el envío de notificaciones.

Como se muestra en la figura 4.4, habrá una colección ,llamada usuarios, que contendrá un documento por cada uno de los usuarios de la aplicación, que estará referenciado con el ID de cada uno de ellos. Este documento se creará en el momento en que el usuario se registre, y contendrá



(a) Árbol de widgets



(b) Nodo Help

Figura 4.3: Esbozo de la jerarquía de los widgets principales.

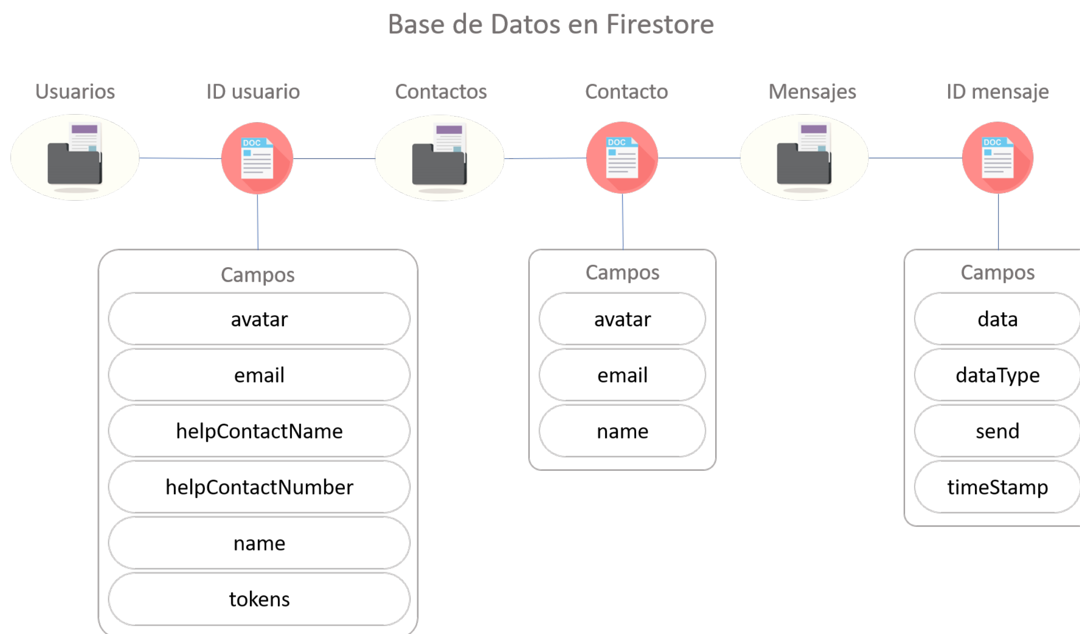


Figura 4.4: Esquema de la base de datos de Firestore

una serie de campos, destinados a la identificación del usuario en la aplicación (name, email, avatar), en el se encontrarán también los campos que identifican al contacto de emergencia del propio usario (helpContactName, helpContactNumber), y por último una lista de tokens donde se guardarán los tokens asociados a cada dispositivo donde el usuario tenga iniciada una sesión, esto se utilizará para el envío de notificaciones. Además, a través de este documento, se accederá a la colección de contactos de cada usuario, la cual contiene un documento por cada contacto con los campos, con los datos de dicho contacto necesarios para su identificación (name, email, avatar), y que, este a su vez, referencia a la colección de mensajes de ese contacto en concreto. La colección de mensajes contendrá un documento por cada mensaje que haya entre el usuario del documento y el contacto en cuestión, este documento contará con cuatro campos. Uno de los campos contendrá el cuerpo del mensaje, que este podrá ser texto, una lista de mapas con la información de pictogramas o un mapa con las coordenadas de la ubicación. Habrá un campo dataType, el cual será un entero con el que se identificará si se recibe un mensaje basado en texto, pictogramas o ubicación. Y por motivos de visualización habrá dos campos, send y timeStamp, con los que se mostrarán los que se identificará si el mensaje debe ser visualizado como emisor o como receptor y el orden en que deben ser mostrados, respectivamente.

Por otro lado, cada dispositivo, en el que instale la aplicación y se ejecute, tendrá una base de datos local que contendrá datos relativos a los pictogramas, y será común para todos los usuarios. El objetivo de esta base de datos en acortar los tiempos de respuesta para visualizar los pictogramas en la pantalla de conversación. Sin embargo, esto no significa que no se requiera la conexión a internet, ya que no se almacenará en ella la imagen de cada pictograma como tal. En la figura 4.5 se muestra el esquema de relaciones que seguirá esta base de datos, que será gestionada por el gestor de base

de datos sqlite.

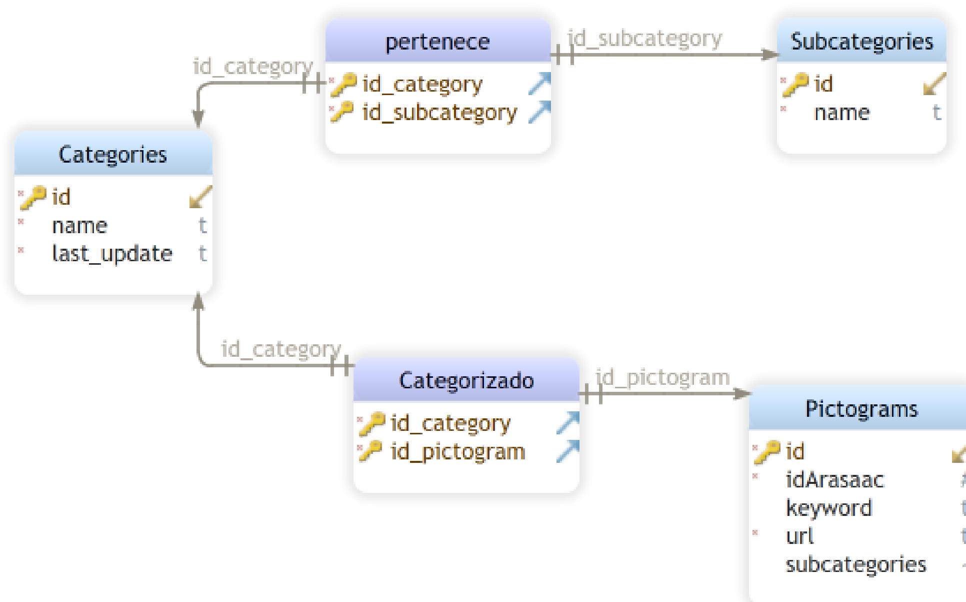


Figura 4.5: Esquema de la base de datos local

4.2. Estructura del proyecto

El proyecto de Flutter se estructura tal y como se describe a continuación. En la figura 4.6 se muestra el contenido del directorio raíz del proyecto.

```
conectacontea
├─ android
├─ assets
├─ ios
├─ lib
├─.packages
├─ pubspec.lock
├─ pubspec.yaml
└─ test
```

Figura 4.6: Contenido del directorio raíz.

- /andriod & /ios: en estos directorios se encuentra el código específico para ejecutar la aplicación en los sistemas operativos de Android y iOS respectivamente. Ambos proyectos son autogenerados en el momento de la creación del proyecto Flutter, y las modificaciones en el código dentro de estos directorios serán mínimas.

- /assets: en este directorio se deben guardar todos los archivos estáticos que se necesiten para la aplicación, como imágenes, videos, iconos o fuentes, que se vayan a utilizar como parte de la interfaz gráfica.
- /lib: este es el directorio principal del proyecto, puesto que aquí se encuentra todo el código Dart con el que se construye la aplicación. Más adelante se detallará el contenido de este.
- /test: este otro contendrá las pruebas del código Dart.
- /.packages: aquí se encuentran los paquetes y librerías utilizadas por Flutter.
- /pubspec.lock: este archivo contiene las versiones de los paquetes y librerías mencionadas anteriormente.
- /pubspec.yaml: este archivo es de configuración específica del proyecto, y en él se configuran las dependencias requeridas.

Como ya se ha mencionado, dentro de la carpeta lib se encuentra el código Dart que se ha programado para crear la aplicación. En la figura 4.7 se ilustra estructura de este directorio:

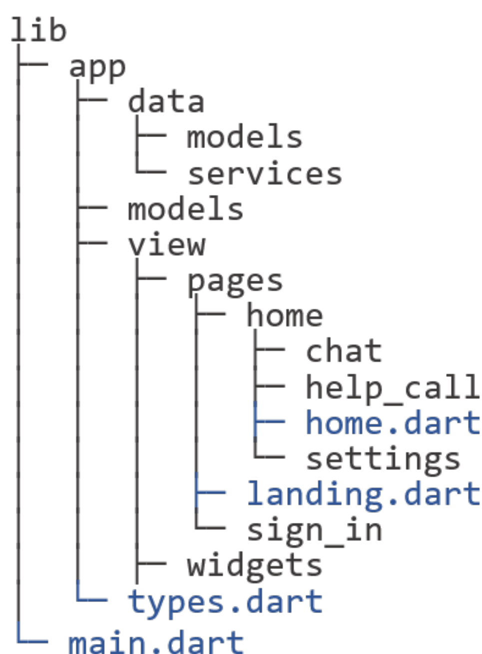


Figura 4.7: Contenido del directorio lib.

En el capítulo 3 se ha mencionado la decisión de usar el patrón Provider. El directorio /lib se ha estructurado tal y como se muestra en la figura 4.7 con el fin de obtener un código más modular y coherente con dicho patrón. Por un lado, en la carpeta /lib/app/view se encuentra todo el código que construye la interfaz, esta se ha organizado a su vez en dos subcarptas, /lib/app/view/pages, y /lib/app/view/widgets. Dentro del directorio ../widgets están los widgets que se pueden reutilizar en diferentes puntos de la aplicación, y dentro de ../pages se encuentran los archivos que conforman las diferentes pantallas, que hacen uso de los widgets anteriormente mencionados. Por otro lado está la carpeta /lib/app/models, en ella se encuentran los modelos que manejarán los datos recibidos por los distintos servicios y notificarán a las diferentes partes de la interfaz, para de esta manera, separar la interfaz de la lógica. Y por último, está la capeta /lib/app/data, donde se encuentran por un lado, dentro

de /lib/app/data/services la implementación de las clases que dan acceso a distintos servicios, como firestore o firebase authentication, entre otros, y por otro lado, dentro de models, se encuentran las clases con las que se convertirá la información proporcionada por los servicios en objetos de Dart, con el fin de tener una mejor accesibilidad a dicha información.

4.3. Descripción de la implementación

A continuación se describirá el proceso de implementación, y como aclaración, solo se mostrarán ciertas partes del código que se crean necesarias para complementar la explicación. Para llevar a cabo toda la implementación ha sido necesario hacer constantes consultas a la documentación oficial de Flutter [22], a la documentación oficial de Dart [23], así como al gestor de paquetes de estos [24]. También se ha consultado la documentación de Firebase, tanto la especifica de Flutter [15], como la general [14].

4.3.1. Ejecución de la aplicación

Es oportuno empezar por describir el archivo main.dart ya que este es imprescindible en cualquier aplicación de Flutter. Este archivo contiene la función main(), la cual da paso a la aplicación. Es recomendable tener el código mínimo indispensable dentro de esta función, con el objetivo de obtener una solución modular, con un código limpio, legible y reutilizable. Por lo tanto, tal y como se muestra en la figura 4.8, esta función se limita a lanzar la aplicación con la función runApp, que recibe como argumento el widget MyApp, que se explicará a continuación, además se especifica la orientación con la que se lanzará la aplicación y se mantendrá en todas las vistas. Dado que se realiza esta acción antes de ejecutar la función runApp, es necesario crear el enlace entre la capa de widgets y el motor de Flutter previamente, esto se hace con el método estático, ensureInitaliazed, de la clase WidgetsFlutterBiding, original de Flutter.

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  SystemChrome.setPreferredOrientations([DeviceOrientation.portraitUp])
      .then((_) {
    runApp(new MyApp());
  });
}
```

Figura 4.8: Función main.

El widget MyApp, figura 4.9, podría extender tanto de la clase StatefulWidget como de la clase StatelessWidget según las necesidades. En este caso, MyApp, será un StatefulWidget ya que se deben precargar datos y configuraciones iniciales para lo que se necesita inicializar el estado. En primer lugar,

para establecer la comunicación entre los servicios de Firebase y Flutter, se deben inicializar, esto se realizará con método initializeFlutterFire. Este método, se ejecutará de manera asíncrona dado que el método estático, initializeApp, de la clase Firebase proporcionada por el paquete firebase_core, retorna la aplicación de Firebase encapsulada en un objeto Future. Además de lo anterior, se precargará la categorización de pictogramas, desde la base de datos local, en una instancia de la clase Categorization con el fin de minimizar los tiempo de carga de estos en la vista del chat. Para esto se ha implementado la clase PictoDatabase, que contiene los métodos necesarios para realizar las operaciones precisas sobre la base de datos. Para realizar estas operaciones se ha instalado el plugin sqflite. Además también se ha agregado la dependencia del plugin path para definir la ubicación para almacenar la base de datos.

```
class MyApp extends StatefulWidget {
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  bool _initialized = false;
  bool _error = false;
  Categorization _categorization = Categorization();

  void initializeFlutterFire() async  { ... }

  void initPictograms() async  { ... }

  @override
  void initState() {
    initializeFlutterFire();
    initPictograms();
    super.initState();
  }

  @override
  Widget build(BuildContext context) { ... }
}
```

Figura 4.9: Implementación del widget MyApp.

Una vez implementadas estas acciones, se implementa el método build, figura 4.10, con el que cuentan todos los widgets ya que se encarga de construir la interfaz. En este método se comprobará la correcta inicialización de la aplicación de Firebase, y en caso afirmativo, retornará el árbol construido por la clase MultiProvider, para lo que es necesario añadir la dependencia del plugin Provider. En la cima del árbol se encontrarán los providers declarados, de manera que todos los widgets sucesores puedan acceder a estos. Por debajo de estos providers se encontrará el widget GestureDetector, con el cual se permite ocultar el teclado pulsando cualquier parte de la pantalla, se ha puesto como padre de la aplicación, MaterialApp, para que esta funcionalidad se extienda a todas las vistas. Con la clase MaterialApp se crea la aplicación con el diseño de Material, y se indica el tema que se usará en la aplicación, así como la vista inicial.

```
@override
  Widget build(BuildContext context) {
    if (_error) {
      return Container();
    }

    if (!_initialized) {
      return Container();
    }

    return MultiProvider(
      providers: [
        Provider<AuthenticationProvider>(
            create: (_) => FirebaseAuthentication()),
        Provider<FirestoreProvider>(
            create: (_) => FirestoreDatabase(uid: null)),
        Provider<StorageProvider>(create: (_) => FireStorage()),
        Provider<MessagingProvider>(create: (_) => FireMessaging()),
        ChangeNotifierProvider<UserModel>(create: (_) => UserModel()),
        ChangeNotifierProvider<AvatarModel>(create: (_) => AvatarModel()),
        ChangeNotifierProvider<SignInModel>(create: (_) => SignInModel()),
        ChangeNotifierProvider<ContactModel>(create: (_) => ContactModel()),
        ChangeNotifierProvider<Categorization>(
          create: (_) => _categorization,
        ),
        ChangeNotifierProvider<ChatDetailModel>(
            create: (_) => ChatDetailModel()),
      ],
      child: GestureDetector(
        onTap: () {
          FocusScopeNode currentFocus = FocusScope.of(context);
          if (!currentFocus.hasPrimaryFocus &&
              currentFocus.focusedChild != null) {
            FocusManager.instance.primaryFocus!.unfocus();
          }
        },
        child: MaterialApp(
          title: "Conecta Con TEA",
          theme: CustomTheme.customTheme(),
          home: LandingPage(),
        ),
      ),
    );
  }
```

Figura 4.10: Implementación del método build del widget MyAPP.

4.3.2. Autenticación

Servicios para la autenticación

La autenticación de usuarios se gestiona con el servicio de Firebase Authentication, para realizar todas las acciones relacionadas con este servicio se ha creado la interfaz AuthenticationProvider con los métodos que se muestran en la figura 4.11, y la clase FirebaseAuthentication que implementa dicha interfaz. Para usar este servicio, es necesario instalar el plugin firebase_auth. Además, para implementar el inicio de sesión a través de Google, también se necesita el plugin de google_sign_in.

```
abstract class AuthenticationProvider {
  User? get currentUser;
  Stream<User?> authStateChanges();
  Future<User?> signInWithGoogle();
  Future<User?> signInWithEmail(String email, String password);
  Future<User?> signUpWithEmail(String email, String password);
  Future<void> signOut();
}
```

Figura 4.11: Interfaz AuthenticationProvider para el servicio Firebase Authentication.

Se requiere también la dependencia del plugin cloud_firestore para utilizar el servicio de Firestore, donde se guardará la información del usuario, para el manejo de este servicio se ha definido la interfaz Firestore Provider, vista en la figura 4.12, que es implementada por la clase FirestoreDatabase.

Se emplea también el servicio de Firebase Storage donde se almacenarán las imágenes de avatar de cada usuario, para el que se necesita la dependencia del plugin firebase_storage. Este servicio se gestiona con la clase FireStorage que implementa la interfaz StorageProvider, figura 4.13.

Y por último, se añade también la dependencia del plugin firebase_messaging, a través del cual se accede al servicio de Firebase Cloud Messaging, con el que se crearan los tokens que identifican cada dispositivo para el envío y recepción de comunicaciones. Los métodos para el uso de este servicio están definidos en la interfaz MessagingProvider, figura 4.14, que es implementada por la clase FireMessaging.

Implementación de la autenticación

Para mostrar la vista correcta, según el usuario este autenticado o no, se ha creado el widget LandingPage, que lanzará la pantalla inicial para iniciar sesión o registrarse, definida en el widget StrartPage, o la home de la aplicación para un usuario autenticado, definida en el widget HomePage. Para lograr este comportamiento se emplea la clase StreamBuilder, original de Flutter, la cual permite trabajar con un flujo de datos asíncrono. En la Figura 4.15 se muestra de una manera más visual el flujo que sigue así como las pantallas que se visualizarían en un caso u otro, y en la figura 4.16 la implementación de dicho widget.

```
abstract class FirestoreProvider {
  get getCurrentUser;
  set currentUser(String uid);
  void unSetcurrentUser();

  bool isUser(String email);
  Future<void> addUser(Map<String, dynamic> userData);
  Future<void> updateUser(Map<String, dynamic> userData);
  Future<QuerySnapshot> userFromEmail(String email);
  Stream<UserModel> userModel();

  bool isContact(String email);
  Future<void> addContact(Map<String, dynamic> localData);
  Future<void> updateContact(Map<String, dynamic> data);
  Stream<List<ContactModel>> contacts();

  Future<void> sendMessage(
    String reciverEmail,
    String senderEmail,
    dynamic data,
    MessageType dataType,
  );
  Stream<List<MessageModel>> messages(String email);

  Future<void> saveTokenToDatabase(String token);
  Future<void> removeTokenToDatabase(String token);
  Future<List<dynamic>> getTokens(String email);
}
```

Figura 4.12: Interfaz FirestoreProvider para el servicio Firestore.

```
abstract class StorageProvider {
  Future<void> uploadAvatar(String avatarPath, String uid);
  Future<String> getUrl(String path);
}
```

Figura 4.13: Interfaz StorageProvider para el servicio Firebase Storage.

```
abstract class MessagingProvider {
  Future<String?> get token;
  Future<void> sendNotification(List<dynamic> tokens, String senderEmail);
}
```

Figura 4.14: Interfaz MessagingProvider para el servicio Firebase Cloud Messaging.

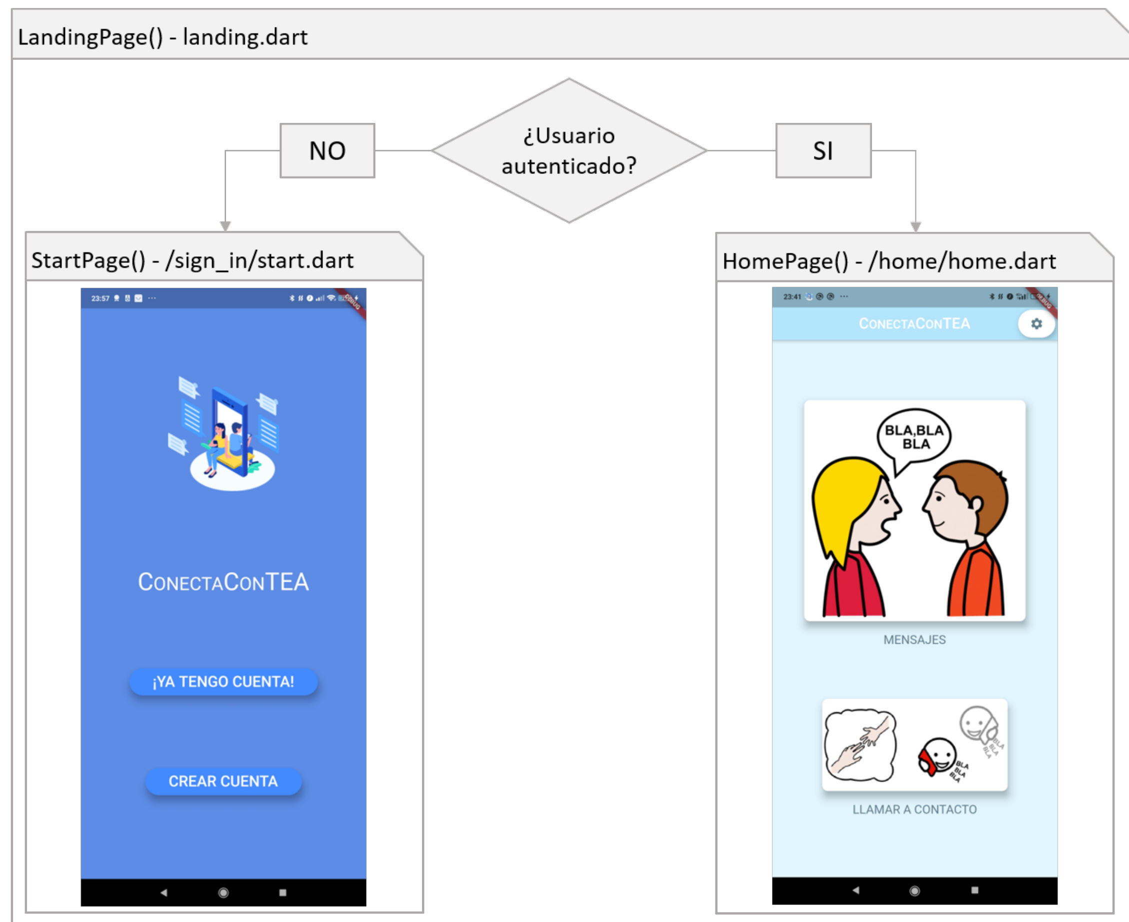


Figura 4.15: Pantallas iniciales.

```
class LandingPage extends StatelessWidget {
  const LandingPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final auth = Provider.of<AuthenticationProvider>(context);
    return StreamBuilder<User?>(
      stream: auth.authStateChanges(),
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.active) {
          final User? user = snapshot.data;
          if (user == null) {
            return StartPage();
          }
          final db = Provider.of<FirestoreProvider>(context);
          if (db.getCurrentUser == null) db.currentUser = auth.currentUser!.uid;
          return StreamBuilder<UserModel>(
              stream: db.userModel(),
              builder: (context, snapshot) {
                if (snapshot.hasData) {
                  return Consumer<UserModel>(
                      builder: (build, localUser, child) {
                    localUser.copyData = snapshot.data!;
                    return HomePage();
                  });
                } else {
                  return Scaffold(
                    body: Center(
                      child: CircularProgressIndicator(),
                    ),
                  );
                }
              });
        } else {
          return Scaffold(
            body: Center(
              child: SpinKitHourGlass(
                color: Colors.blueAccent,
                size: 50.0,
              ),
            ),
          );
        }
      },
    );
  }
}
```

Figura 4.16: Implementación del widget LandingPage

Desde la pantalla definida por el widget StartPage, se pulse el botón "¡Ya tengo cuenta!" o "Crear cuenta" ambos navegarán hacia la vista al widget SignInPage, que es la estructura base del formulario para iniciar sesión o registrarse, modificando el modelo SignInPage con el tipo de formulario que se desea según el botón pulsado.

En el widget SignInPage, el único elemento que sufre cambios será el título del widget AppBar, original de Flutter, donde se mostrará si está en el formulario de inicio de sesión o de registro. Este widget, AppBar, es un StatefulWidget, por lo que no será necesario hacer toda la pantalla dinámica. Para pintar un título u otro, consumirá del modelo SignInModel, cuando se produzcan cambios, este modelo notificará a todos los widgets que consuman de él con el método notifyListeners de la clase ChangeNotifier, original de Flutter, y su función build del widget que consume lo reconstruirá con el nuevo estado. En la figura 4.17 se muestra parte del código de SignInModel donde se implementa lo anterior.

```
class SignInModel with ChangeNotifier {
  FormType? formType;
  bool isLoading = false;
  SignInModel({
    this.formType,
  });

  void updateWith({FormType? formType, bool? isLoading}) {
    this.formType = formType ?? this.formType;
    this.isLoading = isLoading ?? this.isLoading;
    notifyListeners();
  }

  ...
}
```

Figura 4.17: Implementación del widget SignInModel

Esta estructura base que se crea con el widget SignInPage, contendrá el StatefulWidget SignInForm, que es el que construirá el formulario correspondiente, según el valor que tenga la propiedad formType del modelo SignInModel. Además, en este modelo se implementaran los métodos que realizan el inicio de sesión o registro, ya sea con credenciales o con la cuenta de Google, con las llamadas pertinentes a los servicios necesarios, en este caso a Firebase Authentication para el registro y autenticación, a Firestore para, en caso de nuevo usuario guardar la información del usuario en la base de datos, esto es necesario para que los datos del usuario estén disponibles en cualquier dispositivo y también para el resto de contactos, también necesitará los servicios de Firebase Storage, pues sí se registra un nuevo usuario y añade una imagen para su avatar se almacenará en el volumen de Storage con la ruta /avatar/email_usario/avatar.png, cabe mencionar que esta ruta se compone con el correo electrónico y no con el id del usuario con el fin de que todos los usuarios puedan obtener la imagen de avatar de sus contactos. Si no especificará ninguna imagen, se crearía con el avatar por defecto, que se encuentra en la ruta /avatar/default_avatar.png. Y por último el servicio de Firebase Cloud Messa-

ging para obtener el token del dispositivo y registrarlo en la base de datos de Firestore, para que se puedan recibir notificaciones en el dispositivo en el que se ha iniciado sesión.

La decisión de crear un único widget para ambas vistas viene motivada por que ambos formularios tienen elementos en común y de esta manera se reutiliza el código común para las dos vistas. En la figura 4.18 se muestran las pantallas que cumplen con el requisito RF01 (tabla 3.1), con un pequeño esquema de cómo se llegaría a cada una de ellas.

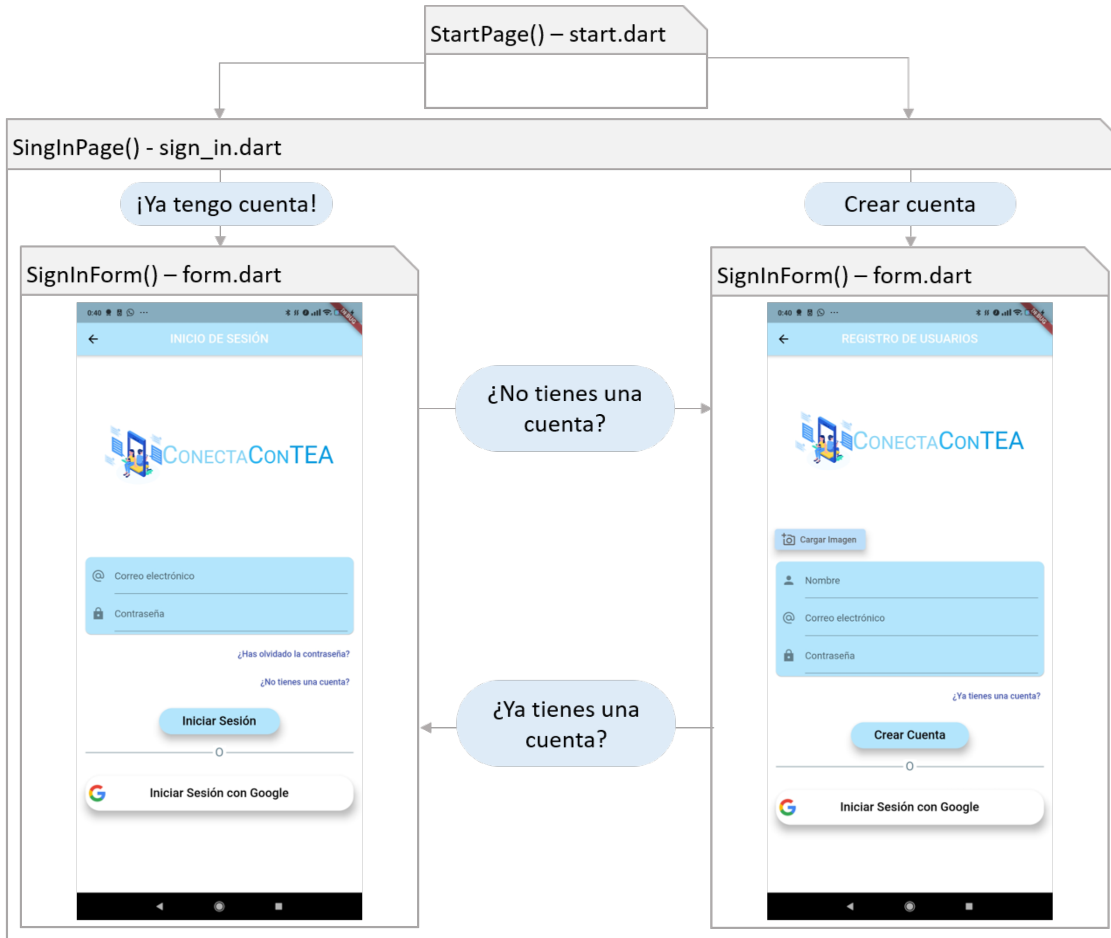


Figura 4.18: Inicio de sesión y registro de usuarios

4.3.3. Vistas para usuarios autenticados

Pantalla de inicio

De acuerdo con el requisito RF04 (tabla 3.2), se ha implementado el StatelesWidget HomePage, que define la pantalla inicial de un usuario autenticado. HomePage es un widget estático ya que una vez renderizado no se modificará su aspecto. La función de esta vista es crear los botones que permiten el acceso directo a las distintas funcionalidades de la aplicación, con el fin de obtener una navegación más intuitiva. Esta vista cuenta con tres botones, uno para lanzar la vista de ajustes, definida en el

widget SettingsPage, otro para navegar a la vista de llamada al contacto de emergencia, implementada con el widget HelpCallPage o bien, si no hubiera ningún contacto de emergencia configurado, a la vista de configuración de contacto de emergencia creada con el widget HelpContactPage, por lo que en lugar de navegar directamente a cualquiera de estos se lanzará el widget LandingCallPage donde se implementa la lógica para este comportamiento. Y por último, el botón de navegación a la vista donde se muestran todos los contactos del usuario definida en el widget ChatsPage. En la figura 4.15 se puede observar la disposición de la pantalla inicial.

Contacto de emergencia

Todas las pantallas de la aplicación, para usuarios autenticados, tendrán un botón de acceso directo a la llamada de emergencia, a excepción de la pantalla de ajustes y la de configuración del propio contacto de emergencia. Como ya se ha mencionado anteriormente, en el caso de no estén configurados los datos de dicho contacto, al pulsar este botón, en cualquiera de las pantallas, se mostrará la vista que permite configurarlo.

Para permitir el comportamiento explicado se ha implementado el widget LandingCallPage, siguiendo el mismo procedimiento que se ha descrito en la sección 4.3.2 para lanzar la vista adecuada.

En este caso, el StatelessWidget LandingCallPage recibirá los datos referentes al usuario autenticado, proporcionados por la base de datos remota, en un stream y tomará el valor del campo helpContactNumber, de modo que si este está vacío lanzará la vista de configuración y en caso contrario la de hacer la llamada.

Para la configuración del contacto de emergencia se ha implementado el StatefulWidget HelpContactPage, el cual consume del modelo UserModel, del que tomará los datos actuales, si los hubiera para mostrarlos en los distintos campos, y en el que se actualizarán los nuevos datos a configurar. Este widget, a parte de un campo de texto para el nombre y otro para el número de teléfono, tendrá dos botones, uno para cancelar la acción, de modo que al pulsar este no se realizará ninguna operación y se retorne a la pantalla anterior, y otro para guardar la configuración en el documento del usuario actual en la base de datos remota con los datos actualizados en el modelo, al guardar se navegará a la pantalla anterior.

Una vez configurado el contacto de emergencia, se mostrará siempre la pantalla definida en el StatelessWidget HelpCallPage, excepto en la pantalla de ajustes que se podrá seguir accediendo al contacto de emergencia para hacer cambios. Este widget, una vez lanzado, no necesita reconstruirse por lo que se ha implementado como estático. En esta pantalla lo que aparece es el nombre del contacto configurado y dos botones, uno para especificar que no se va a llamar, y cuando es pulsado vuelve a la pantalla anterior, y otro botón para realizar la acción. Para lanzar la llamada en el dispositivo, se ha añadido el plugin url_launcher como dependencia, que permitirá lanzar la aplicación externa del dispositivo. En la figura 4.19 se muestra la construcción de dicho botón. En este punto se consumirá

del modelo UserModel para obtener el dato del número de contacto, y con este dato con la función launch, proporcionada por el plugin mencionado, se lanzará la aplicación teléfono del dispositivo para realizar la llamada a dicho número.

```
class HelpCallPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('LLAMAR'),
        centerTitle: true,
      ),
      body: Column( ... ),
          Row(
            ...
            children: [
              ...
              Consumer<UserModel>(builder: (context, localUser, child) {
                return CustomIconLabelButton(
                  image: Image.asset('assets/images/yes.png'),
                  onPressed: () =>
                      launch('tel://${localUser.helpContactNumber}'),
                  label: 'SI',
                );
              }),
            ],
          ),
        ],
      ),
    );
  }
}
```

Figura 4.19: Botón “SI” del widget HelpCallPage.

En la figura 4.20 se muestran ambas pantallas, y un pequeño esquema del flujo que se sigue para llegar a ellas desde la LandingCallPage.

De acuerdo con el requisito RF02 (tabla 3.1) y para que todas las vistas puedan llegar a esta funcionalidad se ha customizado un AppBar con un botón, con la misma apariencia que en la pantalla inicial, que navega hasta el widget LandingCallPage.

Ajustes

Conforme al requisito RF03 (tabla 3.1), se ha implementado una pantalla de ajustes donde se podrán modificar datos del usuario autenticado. En está implementación no se ha considerado la posibilidad de personalización de la propia herramienta, como pueden ser el tema o el estilo de la fuente.

Esta vista está definida por el StatelessWidget SettingsPage. Se ha decidido que sea estático ya que siempre tendrá la misma apariencia y ninguno de sus elementos necesita escuchar eventos. El

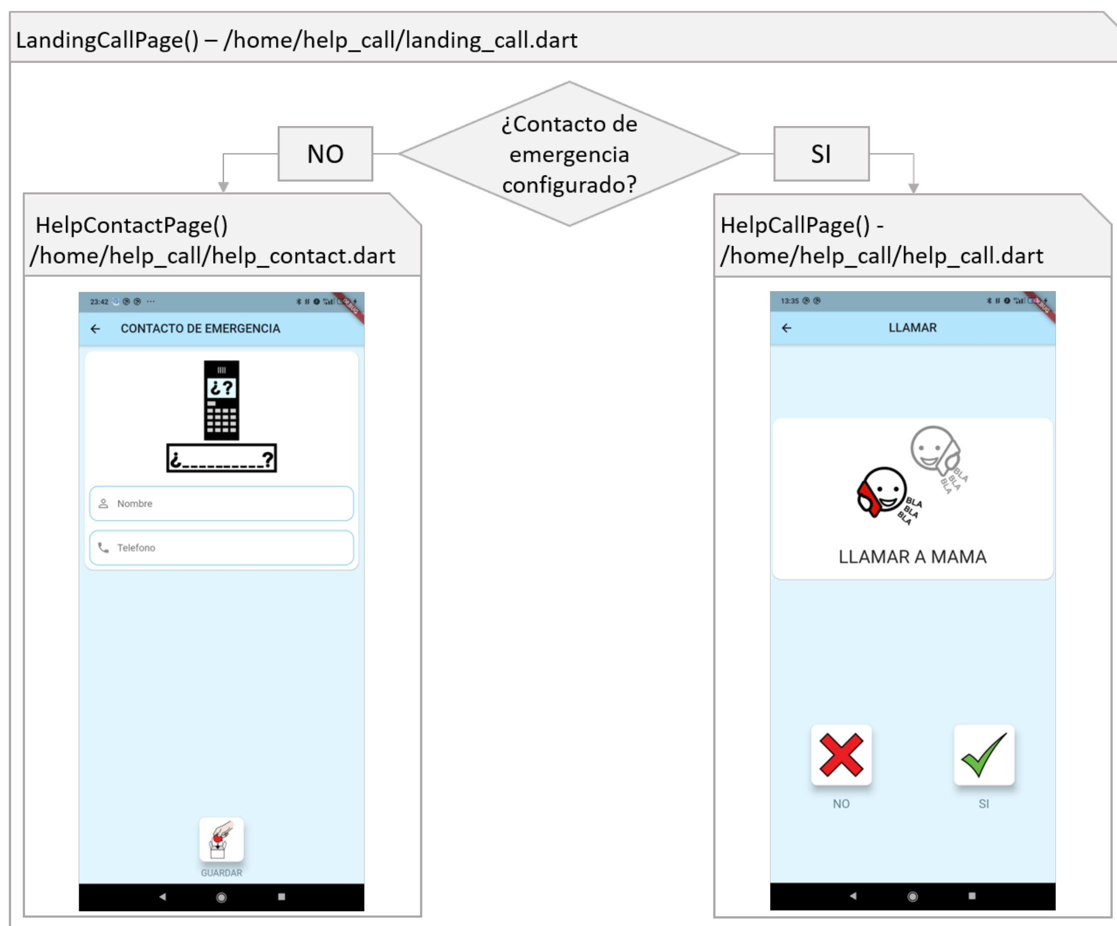


Figura 4.20: Contacto de emergencia.

cuerpo de esta pantalla será implementado por el widget ListView, original de Flutter, el cual muestra las acciones disponibles como elementos independientes.

Desde la sección de ajustes se pueden realizar tres acciones. En primer lugar, se puede modificar el perfil del usuario. Esta pantalla se implementa con el StatefulWidget ProfilePage, cumpliendo con el requisito RF08 (tabla 3.2), el cual muestra los datos actuales del usuario, avatar, nombre y correo electrónico. Cabe aclarar que solo el avatar y el nombre son modificables. Para modificar el avatar se ha implementado el StafulWidget CustomAddImageDialog dentro de app/widgets, ya que se utilizará en todas las pantallas que requieran elegir una imagen, bien sea de la galería, bien sea de la cámara. Para implementar esta funcionalidad se han añadido las dependencias de los plugins image_picker, el cual te permite coger las imágenes de la galería o tomadas por la cámara, e image_cropper, el cual permite recortar las imágenes una vez seleccionadas de manera que todas tengan las mismas dimensiones. Para que los cambios sean efectivos en el momento, se ha creado el modelo AvatarModel. Para simplificar la comunicación entre los distintos widgets, el widget CustomAddImageDialog consumirá de dicho modelo, así como también lo hará el widget ProfilePage, de manera que aunque los cambios se produzcan en uno de ellos, el otro también los reciba y pueda reconstruir la interfaz con el nuevo avatar. También será necesario que ProfilePage consuma del modelo UserModel, ya que en el momento de guardar, se debe actualizar este modelo. Al pulsar el botón de cancelar, se ignoran los cambios y se navega a la pantalla anterior.

En segundo lugar, la siguiente opción navega hasta la pantalla de configuración del contacto de emergencia, directamente lanzando el widget HelpContactPage, esta vez sin pasar por el LandingCallPage, ya que el objetivo no es llamar si no modificar el contacto.

Y por último, se implementa el cierre de sesión. Para implementar esta función, y separar el código, se ha implementado un modelo SettingsModel, el cual realiza las acciones necesarias para cerrar sesión. Para hacer efectivo el cierre de sesión invoca al método signOut de AthenticationProvider para modificar el estado de auteticación, y además también necesita hacer uso del método removeTokenDatabase de FirestoreProvider para borrar el token del dispositivo de la base de datos remota, ya que una vez cerrada la sesión no se deberán recibir notificaciones.

En la figura 4.21 se muestra gráficamente el flujo descrito.

Mensajería

Finalmente conforme con los requisitos RF06, RF07, RF09 (tabla 3.1) y RF10 (tabla 3.2), se han implementado una serie de pantallas que cubren las funcionalidades que estos definen.

En primer lugar, se ha creado una vista que permite visualizar la lista de contactos, navegar hasta las pantallas que permiten añadir un contacto nuevo, editar un contacto y visualizar la conversación de cada uno de ellos, en la figura 4.22 se han marcado, respectivamente, con los números 1, 2 y 3 los

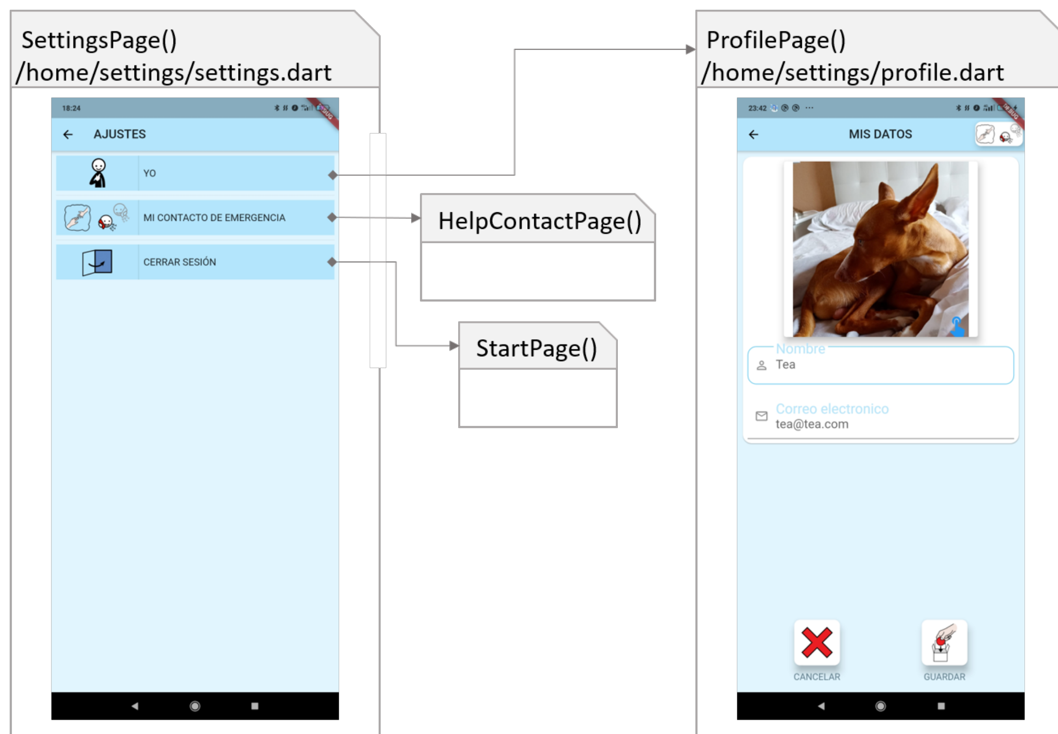


Figura 4.21: Ajustes

botones que realizan la navegación, y en la figura 4.23 se observan las pantallas a las que se dirigen pulsando dichos botones. Además como el resto de las pantallas, esta también cuenta con el botón de acceso directo al contacto de emergencia.

Esta pantalla se ha implementado en el StatefulWidget ChatsPage. Para mostrar la lista de contactos actualizada se ha empleado el uso de StremBuilder, como en ocasiones anteriores, con el cual se recibe de manera asíncrona la lista de contactos del usuario desde la base de datos remota de Firestore, los datos de cada contacto se encapsulan objetos de tipo ContactModel con el fin de facilitar la manipulación de dichos datos.

Al pulsar en el botón 1 (figura 4.22), se lanza la pantalla para añadir un nuevo contacto, en esta pantalla existe la posibilidad de añadir un contacto solo indicando el correo electrónico, de manera que el nombre y avatar que se guardan sean los que tiene configurados el propio contacto, pero también cubre la posibilidad de que el usuario personalice el nombre o el avatar o ambos del contacto que vaya a añadir. Para realizar estas acciones, se emplea el modelo ContactModel, el cual recoge los datos que a guardar. Este mismo modelo proporciona el método con el que se realiza el proceso de añadir el contacto. Este nuevo contacto es registrado en la colección de contactos del usuario en la base de datos remota, invocando al método addContact de FirestoreProvider, así como se registra el propio usuario como contacto en la colección del contacto a añadir. Gracias al StreamBuilder con el que

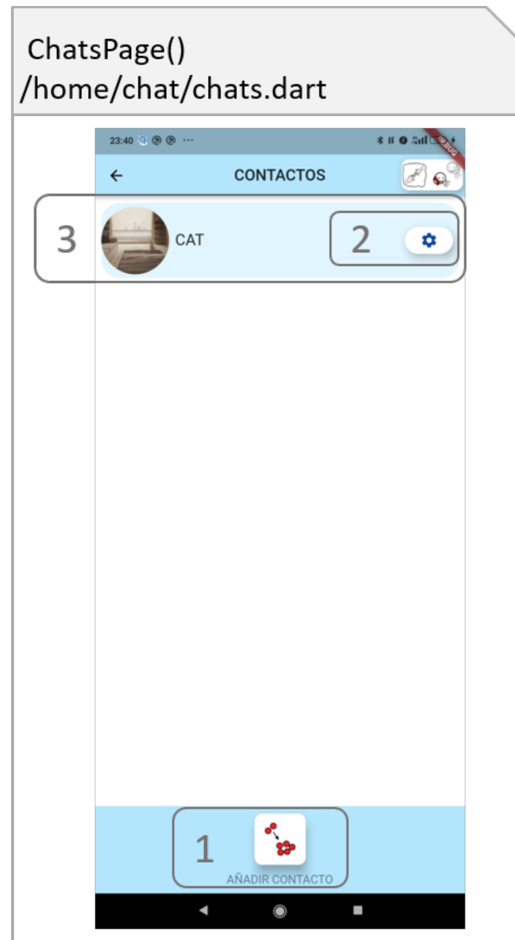


Figura 4.22: Lista de contactos. Widget ChatsPage.

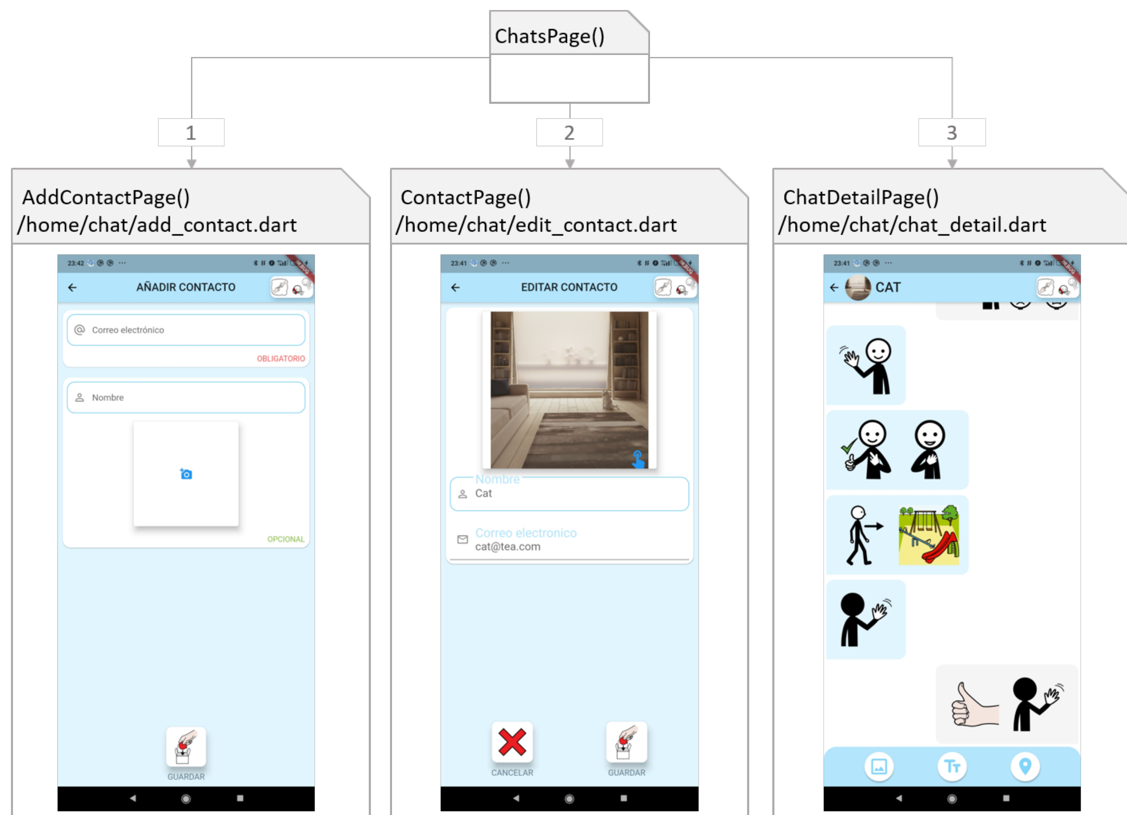


Figura 4.23: Navegación desde la pantalla de contactos.

se implementa la lista de contactos, se actualizarán automáticamente las vistas de ambos usuarios mostrando el nuevo contacto. Para personalizar el avatar, como ocurre en otros widgets, consume del modelo AvatarModel, el cual se actualiza con el widget común CustomAddImageDialog. Al pulsar guardar, el modelo ContactModel ya cuenta con los datos de los campos de texto, ya que a medida que se modifican se guardan también en el modelo, se actualiza el objeto ContactModel con el avatar asignado, si lo hubiera, y se invoca al método que realiza las invocaciones a los servicios pertinentes y que realiza el proceso descrito en la sección 4.3.3 para el widget ProfilePage, con la variante de que este avatar se guarda en la ruta /avatar/email_usuario/email_contacto/avatar.png, con el objetivo de que este cambio solo impacte al usuario que está añadiendo el contacto, y no se sobrescriba el avatar actual del contacto.

También es posible modificar el nombre y avatar de un contacto una vez se ha añadido. Para ello se ha implementado el widget ContactPage, al cual se accede a través del botón 2 (figura 4.22), donde se muestra una vista con los datos actuales del contacto, permitiendo modificar nombre y avatar y no el correo electrónico. Este widget funciona de la misma manera que el widget ProfilePage comentado en la sección 4.3.3, solo que en este caso en lugar de consumir del modelo UserModel, consume del modelo ContactModel. Este objeto, inicialmente, tendrá los datos actuales del contacto puesto que al pulsar el botón 2 (figura 4.22) se actualizan los datos del ContactModel con los del elemento seleccionado. Esto mismo ocurre cuando se pulsa el elemento 3 (Figura 4.22) para abrir la conversación del

contacto adecuado.

Por último, para implementar la pantalla de conversación se ha creado el StatefulWidget ChatDetailPage. Como esta vista es más compleja y tiene varios elementos, se han implementado de forma independiente los widget MessagesChat, el cual muestra la lista de mensajes, e InputChat donde se definen los distintos métodos de entrada para cada tipo de mensaje.

El widget ChatDetailPage únicamente es la estructura base que contendrá a los dos widgets mencionados. Además contiene también un AppBar customizado que permite mostrar el avatar y nombre del contacto de la conversación actual para lo que consume del modelo ContactModel.

El StatefulWidget InputChat consume de los modelos ContactModel, puesto que necesita el email del contacto para enviar los mensajes, Categorization, el cual mantiene la información relativa a los pictogramas, y ChatDetailModel en el cual se almacena toda la información acerca de los mensajes. Este widget incorpora una barra con tres botones, en los que selecciona el tipo de entrada para el mensaje, al pulsar cualquiera de los botones el modelo ChatDetailModel actualiza el dato de tipo de entrada para mostrar la entrada adecuada. En la Figura 4.24 se puede ver la disposición de los diferentes tipos de entrada, así como los diferentes tipos de mensajes. Este widget contiene además el widget WillPopScop, original de Flutter, con el que se permite manejar el comportamiento de la navegación hacía atrás, de manera que cuando se pulse el botón .ªtrás", tanto del AppBar como de la barra de navegación del propio dispositivo, si se está mostrando alguno de los métodos de entrada lo oculte, y sea con una segunda pulsación en dicho botón que navegue a la pantalla anterior.

El primer botón muestra los pictogramas. Estos pictogramas se obtienen a través de la API de ARASAAC, para la cual se ha implementado la clase Arasaac que contiene el método para obtener la colección de pictogramas, esto se hace a través de una petición POST, por lo que ha sido necesario agregar la dependencia del plugin http, el cual permite realizar estas peticiones. La información obtenida con este método se almacena en una base de datos local con el fin de minimizar los tiempos de carga, evitando hacer dicha petición cada vez que se muestre esta pantalla. En esta base de datos local únicamente se almacenan datos necesarios para mostrar la categorización y los propios pictogramas, no se almacenan las imágenes, sino que se guarda la dirección URL que permite obtener el recurso alojado en ARASAAC. Para tener un conjunto más reducido y práctico de pictogramas se han seleccionado únicamente los que en ARASAAC se categorizan como vocabulario nuclear, que recogen un conjunto palabras simples y de uso frecuente [25]. De cara a la implementación se han definido las siguientes categorías: Seres vivos, Objetos, Comunicación, Tiempo, Alimentos, Lugares, Transporte, Ocio, Conocimientos, Educación, Empleos. Por otro lado, con el fin de que el usuario pueda hacer scroll fácilmente, para añadir un pictograma al mensaje se deberá hacer un doble toque sobre el pictograma en cuestión. Además pulsando de manera continuada, sobre cualquiera de los pictogramas, se muestra la palabra clave a la que refiere dicho pictograma, se puede ver esta funcionalidad en el pictograma rodeado por un círculo blanco en la figura 4.24. Esto se ha implementado tanto en la entrada como en los mensajes de la conversación. Cuando se seleccione un pictograma se actualiza el modelo ChatDe-

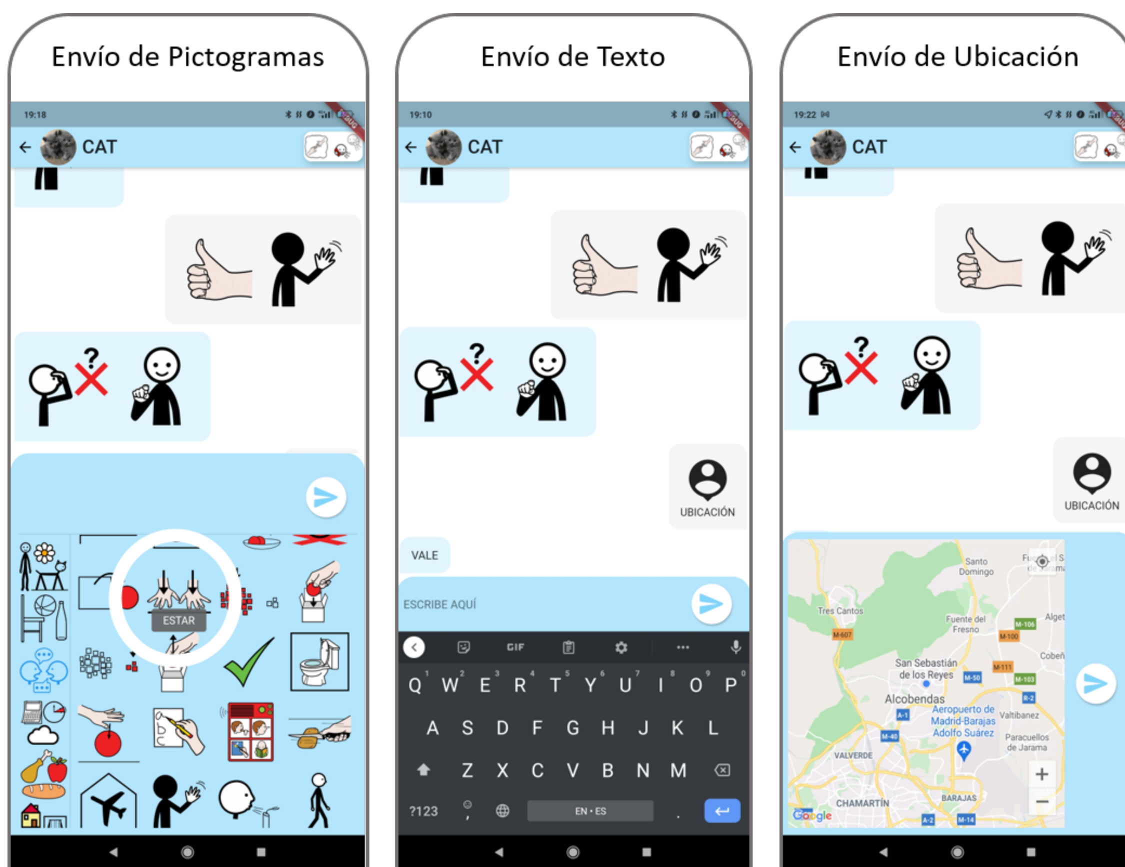


Figura 4.24: Vistas según tipo de entrada.

tailModel y se muestra en la barra superior para ser enviado, para borrar un pictograma del mensaje a enviar, al igual que para seleccionarlo, se hará doble toque sobre él. No se ha establecido límite para el número de pictogramas en un mismo mensaje. Al pulsar enviar, el modelo ChatDetailModel implementa el método que realiza el envío del mensaje y de la notificación al contacto. Para esto, en la interfaz de FirestoreProvider se han definidos los sendMessage, con el cual se escribe el mensaje en la colección de ambos usuarios, y getTokens para obtener los tokens de los dispositivos en los que tenga la sesión abierta el receptor y enviar la notificación con el método sendNotification definido en la interfaz MessagingProvider.

El tipo de entrada de texto difiere gráficamente, sin embargo, el proceso de envío y recepción es el mismo que el de pictogramas.

Y en cuanto a la ubicación, igualmente sigue le mismo proceso de envío y recepción que los métodos anteriores, sin embargo, requiere algunas acciones para tomar dicha ubicación. Para mostrar el mapa se ha añadido la dependencia del plugin google_maps_flutter y para obtener la ubicación actual, se ha añadido también la dependencia del plugin location. Con este último plugin se obtiene las coordenada, compuesta por latitud y longitud, de la posición actual, vista en el mapa con un punto azul. Para el envío de la ubicación se compone un enlace a Google maps con dichas coordenadas de manera que cuando le llegue el mensaje al receptor pueda abrir dicho enlace pulsando sobre el mensaje y se le dirija directamente a la página de google maps que contiene la opción de cómo llegar con destino en la ubicación del contacto. Para utilizar el servicio de google maps ha sido necesario modificar el archivo AndroidManifest.xml del proyecto Android, añadiendo el token de autenticación a la API de google y especificando los permisos que debe conceder el usuario para acceder a la ubicación.

5 Pruebas

En todo desarrollo es necesario realizar pruebas de cada componente implementado para confirmar que el comportamiento de estos es correcto y coherente con los requisitos definidos.

Para comprobar el correcto funcionamiento de las clases que implementan la lógica de negocio se han realizado una serie de pruebas unitarias, de forma paralela a la implementación, con el fin de no arrastrar errores y evitar comportamientos no deseados. Aún que ha incrementado el tiempo de desarrollo, el hecho de realizar las pruebas progresivamente ha facilitado la identificación de los posibles errores, y la corrección de estos.

La práctica para corregir errores encontrados ha sido la depuración de código mediante la herramienta de inspección de DevTools [12], proporcionada por Flutter.

Mediante emuladores y dispositivos que cuentan con el sistema operativo Android, se han comprobado que las funcionalidades implementadas son coherentes con lo que se ha especificado en los requisitos funcionales de la sección 3.1.1. Durante el capítulo 4 de implementación, se ha ido comentando en qué punto se cubre cada uno de los requisitos. Por lo que se puede decir que se han cumplido todos los requisitos funcionales que estaban dentro del alcance de esta entrega.

En cuanto a los requisitos funcionales, ya se ha mencionado en la sección 3.1.2 que no iba a ser posible, para este trabajo, medir los requisitos RNF03, RNF04 y RNF09 (tabla 3.3) ya que esta medición requiere la valoración de los usuarios finales, con los que no se ha probado la aplicación. En cuanto al requisito RNF01, aún que si se ha permitido configurar los datos del propio usuario y de sus contacto, no se da por satisfecho, ya que queda pendiente permitir la configuración del aspecto de la interfaz, como el tamaño de la fuente o el tema. El requisito RNF07 (tabla 3.3) no define un rango con lo que respecta a cuanto debería de ocupar, sin embargo, se ha hecho una medición con la herramienta para medir el uso de la memoria de DevTools [26]. En las pruebas de memoria que se han hecho, se ha determinado que la instalación de la aplicación ocupa 13.3MB mientras que cuando la aplicación este en uso, el valor más alto que se ha tomado de ocupación de memoria ha sido 400MB. Y por último, para medir el rendimiento la herramienta para medir el rendimiento de DevTools [27], para lo que ha sido necesario ejecutar la aplicación en modo profile, con el comando flutter run –profile, puesto que este hace que la aplicación se ejecute en modo pre-realese, y no en modo debug donde los datos no

son reales. En la figura se muestran datos obtenidos de esta medición, donde se ha identificado que los tiempo de respuesta más altos ocurren al renderizar los widgets Chats, y ChatsDetail, debido a la lista de contactos y la lista de mensajes que muestran, sin embargo, el valor más alto registrado en esta prueba es de 30ms por lo que nunca se supera la cota de 200ms que se ha definido en el requisito RNF06 (tabla 3.3).

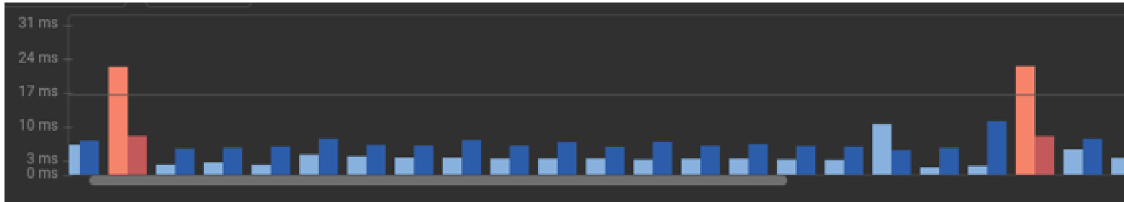


Figura 5.1: Toma de rendimiento

6 Conclusiones y Trabajo Futuro

6.1. Conclusiones

Para concluir, se ha hecho una reflexión del trabajo obtenido con esta implementación, y se puede decir que se ha cumplido con el objetivo propuesto inicialmente, obtener un primer MVP de la aplicación, llamada Conecta Con TEA, que ofrece una alternativa de comunicación respecto a las aplicaciones de mensajería que se encuentran actualmente en el mercado, y que tiene en cuenta las posibles situaciones de crisis y bloqueo que se pueden dar las personas que se encuentran dentro del espectro autista.

Además de las pautas marcadas inicialmente para legar al objetivo, ha sido necesaria una tarea de recopilación de información del Trastorno de Espectro Autista y de las personas con TEA, para entender la motivación de este proyecto y adaptar lo mejor posible, dentro del alcance, la herramienta a las necesidades de las personas con TEA.

Una vez entendida la motivación, realizado el estudio de las diferentes tecnologías y tomada la decisión de implementar esta aplicación con Flutter, ha sido imprescindible dedicar un tiempo a la formación en este framework, empezando con proyectos más pequeños hasta poder entender el funcionamiento de este y poder tomar decisiones acerca de metodologías y arquitecturas empleadas.

Se considera que el framework elegido, Flutter, has sido una buena decisión ya que la curva de aprendizaje has sido baja, ha permitido cumplir con el objetivo, y también contempla los requisitos que no se han realizado en esta implementación. Además, hay decir que la opción recomendada por Flutter para implementar el manejo de estados, Provider, también ha sido una elección correcta dada la inexperiencia en este framework, y que cumple con su función para este MVP, sin embargo, para futuras implementaciones, en las que pueda crecer la aplicación, se podría considerar el uso de otra arquitectura más compleja, como Business Logic Component (BLoC) , que además emplea el uso de Provider.

Para concluir, aunque se ha cumplido con el objetivo, se cree que es una herramienta con potencial y utilidad, en la que se podría seguir trabajando hasta alcanzar un producto que se pueda publicar en

las plataformas correspondientes, como en Google Play en el caso de Android.

6.2. Trabajo futuro

Como trabajo futuro queda pendiente validar el prototipo obtenido con usuarios finales con el objetivo de identificar posibles mejoras, así como implementar los requisitos definidos inicialmente que se han quedado fuera del alcance de este trabajo, en caso de un análisis positivo.

Además, durante la realización de este trabajo se han detectado las siguientes posibles mejoras que no han sido contempladas en la toma de requisitos inicial, ni en este trabajo:

- Limitar el uso de la red , de manera que esta solo sea necesaria para las acciones que no se puedan realizar de otra manera, dando una mejor experiencia para cuando el usuario no tenga conexión.
- Hacer más visible e intuitiva la recepción de mensajes.
- Crear una nueva categoría de pictogramas recientes.

Bibliografía

[1] Álvaro Martínez de Navascués, "Análisis y diseño de una herramienta de comunicación para personas con tea," tesis, Escuela Politécnica Superior, Universidad Autónoma de Madrid, 2020.

[2] Autism-Europe, "People with autism spectrum disorder," julio 2019.

[3] N. Higashida, *La razón por la que salto.* rocaeditorial, 2007.

[4] P. Nawrocki, K. Wrona, M. Marczak, and B. Sniezynski, "A comparison of native and cross-platform frameworks for mobile applications." https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9378923&tag=1, 2021. (Último acceso 06/2021 con la red de la institución).

[5] "Flutter sdk releases." https://flutter.dev/docs/development/tools/sdk/releases. (Último acceso 06/2021).

[6] "React native 0.5.0." https://github.com/facebook/react-native/tree/0.5-stable. (Último acceso 06/2021).

[7] "Stack overflow developer survey." https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-other-frameworks-libraries- (Último acceso 06/2021).

[8] S. Alpert, "State of react native 2018." https://reactnative.dev/blog/2018/06/14/state-of-react-native-2018#architecture, junio 2018. (Último acceso 06/2021).

[9] "Flutter architectural overview." https://flutter.dev/docs/resources/architectural-overview. (Último acceso 06/2021).

[10] "Hot reload." https://flutter.dev/docs/development/tools/hot-reload. (Último acceso 06/2021).

[11] "Fast refresh." https://reactnative.dev/docs/fast-refresh#how-it-works. (Último acceso 06/2021).

[12] "Using the flutter inspector." https://flutter.dev/docs/development/tools/devtools/inspector. (Último acceso 06/2021).

[13] "Get the dart sdk." https://dart.dev/get-dart. (Último acceso 06/2021).

[14] "Firebase." https://firebase.google.com/. (Último acceso 06/2021).

[15] "Flutterfire." https://firebase.flutter.dev/. (Último acceso 06/2021).

[16] "Visual studio code." https://code.visualstudio.com/. (Último acceso 06/2021).

[17] "Cómo crear y administrar dispositivos virtuales." https://developer.android.com/studio/run/managing-avds?hl=es-419. (Último acceso 06/2021).

[18] "Api arasaac." https://arasaac.org/developers/api. (Último acceso 06/2021).

[19] "Introduction to widgets." https://flutter.dev/docs/development/ui/widgets-intro. (Último acceso 06/2021).

[20] "State management." https://flutter.dev/docs/development/data-and-backend/state-mgmt/intro. (Último acceso 06/2021).

[21] "Simple app state management." https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple. (Último acceso 06/2021).

[22] "Flutterfire." https://flutter.dev/. (Último acceso 06/2021).

[23] "pub.dev." https://dart.dev/. (Último acceso 06/2021).

[24] "pub.dev." https://pub.dev/. (Último acceso 06/2021).

[25] "Tableros con vocabulario nuclear - core vocabulary." https://arasaac.org/materials/es/1442? (Último acceso 06/2021).

[26] "Using the memory view." https://flutter.dev/search?q=memory. (Último acceso 06/2021).

[27] "Using the performance view." https://flutter.dev/docs/development/tools/devtools/performance#profile-granularity. (Último acceso 06/2021).

Acrónimos

BLoC Business Logic Component.

MVP Minimum Viable Product.

TEA Trastorno de Espectro Autista.

Apéndices

A

ESTADÍSTICAS STACK OVERFLOW

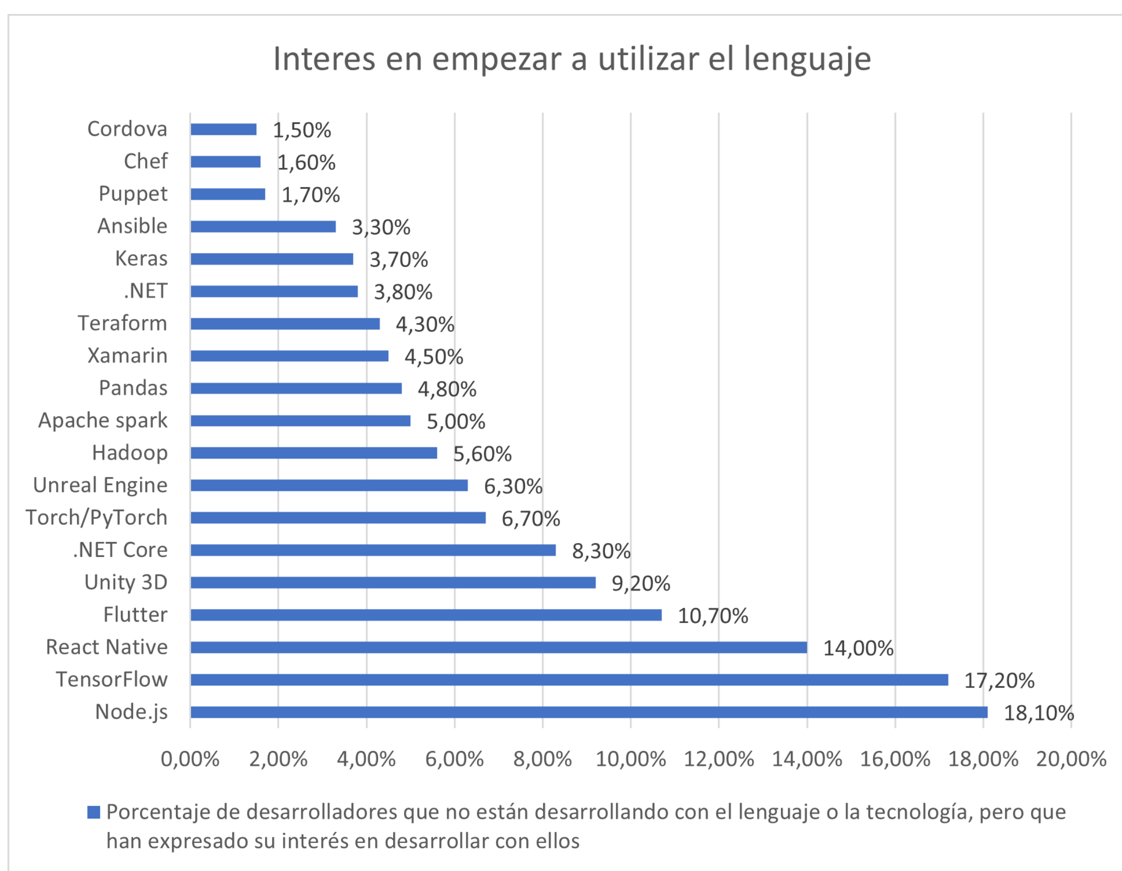


Figura A.1: Desarrolladores con interés en empezar a desarrollar con Flutter y React Native. Fuente: “Stack Overflow”

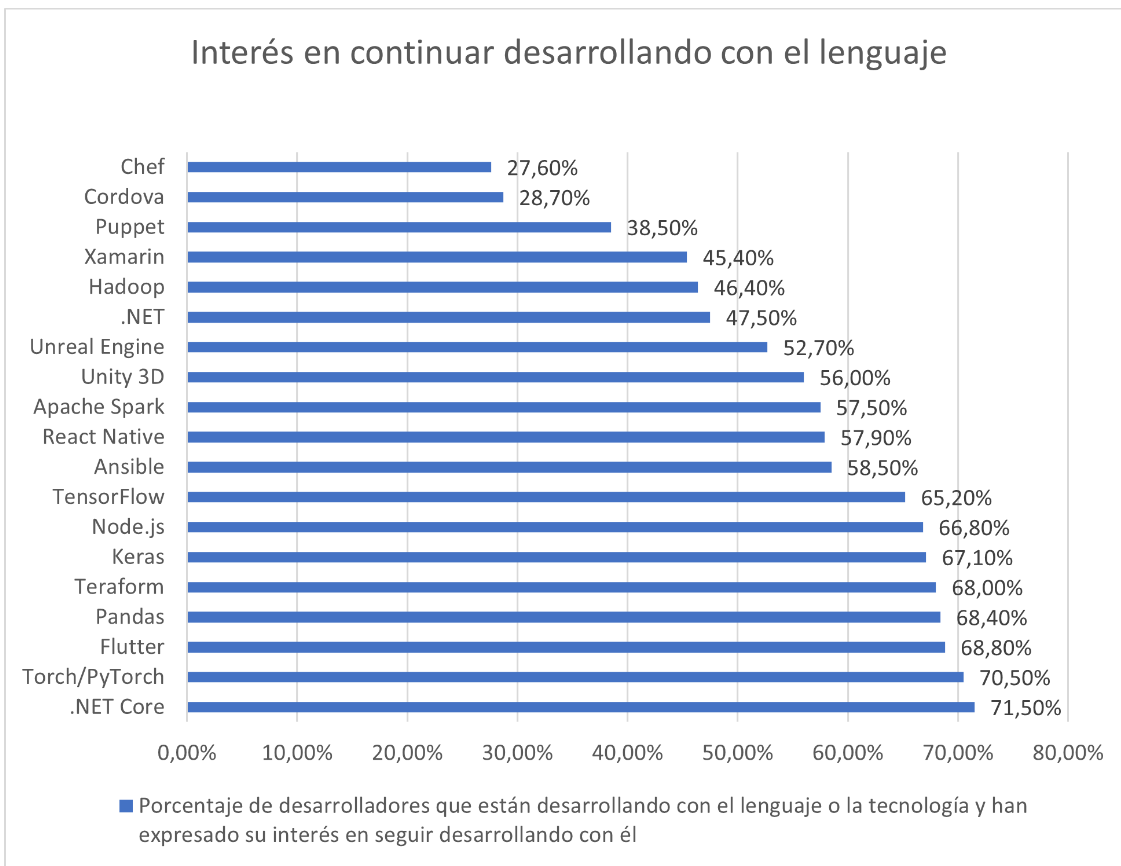


Figura A.2: Desarrolladores con interés en continuar desarrollando con Flutter y React Native. Fuente: “Stack Overflow”

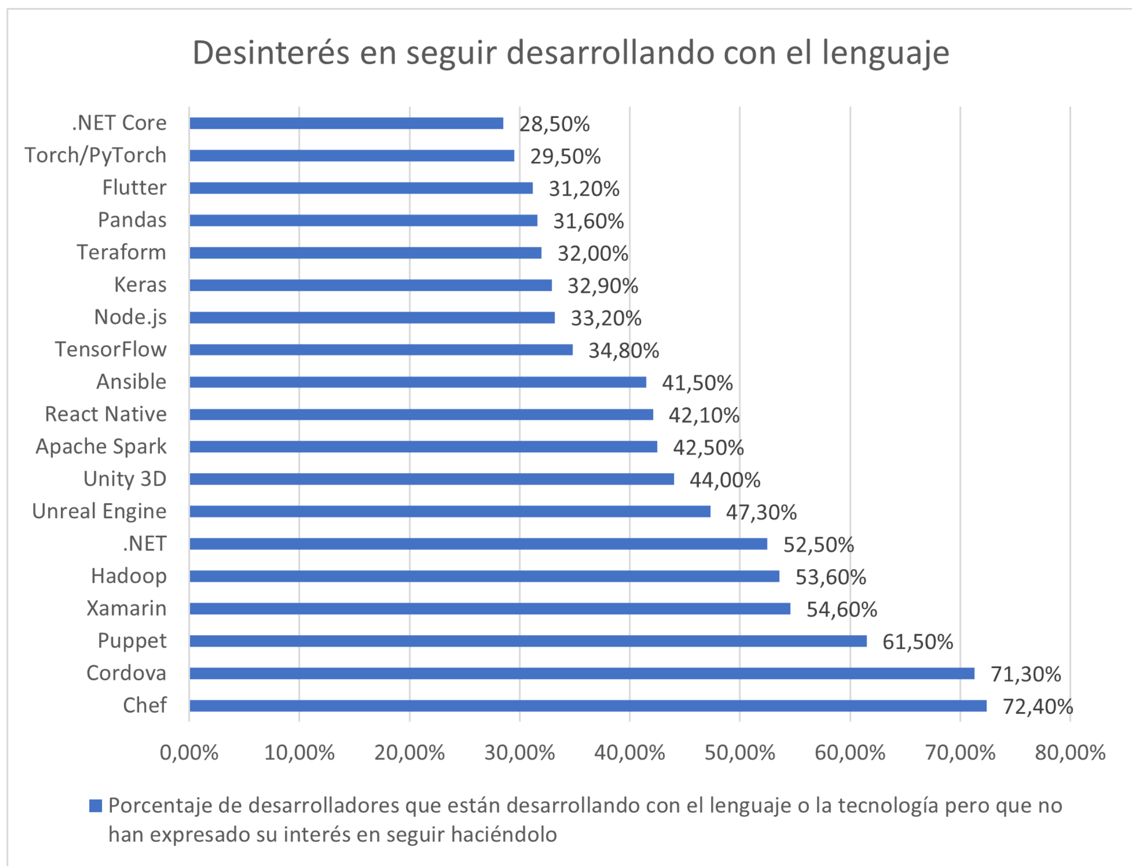


Figura A.3: Desarrolladores sin interés en continuar desarrollando con Flutter y React Native. Fuente: “Stack Overflow”

UAM

UNIVERSIDAD AUTONOMA

DE MADRID