

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Desarrollo de
aplicaciones móviles
multiplataforma con
Flutter

Curso 2018/2019

Alumno/a:

Víctor Vázquez Rodríguez

Director/es:

Antonio Corral Liria

A mi familia, por brindarme la oportunidad de estudiar y ser quien quiero ser.

A mis compañeros, por todas los trabajos y proyectos que hemos sacado adelante juntos.

A mis amigos, por ser la vía de escape a los estudios y recordarme quién soy.

A Francis, por ser la luz que me ha alumbrado con su razón y serenidad en estos 4 años que habrían sido muy diferentes sin ti.

A Rosa, por acompañarme en los peores momentos y levantarme cuando no me veía capaz de seguir.

Al tribunal, por prestarse a la evaluación de este trabajo.

A mi tutor, Antonio Corral, por mantenerme firme en el camino cuando yo no confiaba en el proyecto.

A mis profesores, por mostrarme una cercanía y amabilidad con las que aprender es más sencillo.



Índice

1.	Introducción	1
1.1.	Motivación	2
1.2.	Objetivos	2
1.3.	Planificación temporal	3
1.4.	Estructura de la memoria	4
2.	Estudio del dominio	5
2.1.	El tratamiento de la cirrosis hepática	5
2.2.	Tecnologías multiplataforma alternativas.....	6
2.2.1.	React Native	6
2.2.2.	Ionic.....	7
2.2.3.	Xamarin.....	7
2.3.	Tecnología multiplataforma elegida.....	8
3.	Tecnologías utilizadas.....	9
3.1.	Dart.....	9
3.2.	Flutter.....	10
3.3.	Android Studio	11
3.4.	Visual Studio Code	12
3.5.	JSON.....	13
3.6.	Codemagic.....	14
3.7.	Material Design.....	15
4.	Aplicación B-BlockMe.....	17
4.1.	Análisis.....	17
4.1.1.	Objetivos	17
4.1.2.	Actores	18
4.1.3.	Requisitos funcionales.....	19
4.1.4.	Requisitos no funcionales.....	24
4.1.5.	Requisitos de información.....	25
4.2.	Diseño.....	25
4.2.1.	Arquitectura cliente servidor	25
4.2.2.	Gestión del estado.....	27
4.2.3.	Prototipado de la interfaz.....	29
4.3.	Implementación.....	35

4.3.1.	Configuración del entorno de trabajo	35
4.3.2.	Autenticación y registro	37
4.3.3.	Lista de pacientes.....	42
4.3.4.	Detalles de un paciente	46
4.3.5.	Añadir un paciente	49
4.3.6.	Estado del tratamiento.....	51
4.3.7.	Añadir una medición	53
4.4.	Pruebas.....	55
4.4.1.	Pruebas unitarias.....	55
4.4.2.	Pruebas de interfaz.....	56
4.5.	Despliegue.....	58
5.	Conclusiones y trabajo futuro	61
5.1.	Conclusiones.....	61
5.2.	Trabajo futuro.....	61
	Bibliografía	63

Índice de figuras

Figura 1: Cuota de mercado de los sistemas operativos móviles	1
Figura 2: Diagrama de Gantt del proyecto	3
Figura 3: Logo de React Native	6
Figura 4: Logo de Ionic.....	7
Figura 5: Logo de Xamarin.....	7
Figura 6: Logo de Dart.....	9
Figura 7: Logo de Flutter.....	10
Figura 8: Logo de Android Studio.....	11
Figura 9: Logo de Visual Studio Code	12
Figura 10: Logo de JSON.....	13
Figura 11: Definición de un objeto JSON.....	13
Figura 12: Definición de una colección JSON.....	14
Figura 13: Logo de Codemagic	14
Figura 14: Logo de Material Design	15
Figura 15: Diagrama de casos de uso.....	20
Figura 16: Diagrama de secuencias de la interacción con el servicio web	26
Figura 17: Diagrama de secuencias de la gestión del estado.....	28
Figura 18: Boceto de pantalla de inicio de sesión del paciente.....	29
Figura 19: Boceto de pantalla de inicio de sesión del doctor.....	30
Figura 20: Boceto de pantalla de registro de doctor.....	30
Figura 21: Boceto de pantalla principal del paciente.....	31
Figura 22: Boceto de pantalla de añadir medición (frecuencia)	31
Figura 23: Boceto de pantalla de añadir medición (presión).....	32
Figura 24: Boceto de pantalla de añadir medición (síntomas).....	32
Figura 25: Boceto de pantalla de añadir medición (resumen).....	33
Figura 26: Boceto de pantalla principal del doctor.....	33
Figura 27: Boceto de pantalla de añadir paciente.....	34
Figura 28: Boceto de pantalla de código del nuevo paciente.....	34
Figura 29: Boceto de pantalla de detalles del paciente	35
Figura 30: Salida del comando flutter doctor	36
Figura 31: Pantalla de inicio de Android Studio	36
Figura 32: Pantallas de inicio de sesión de paciente y doctor.....	37
Figura 33: Construcción de los formularios dependiendo del estado	38
Figura 34: Validación y envío de datos de acceso del paciente	39
Figura 35: Función loginPatient de AuthModel	40
Figura 36: Función loginPatient de HttpAuth.....	40
Figura 37: Pantalla de registro de doctor.....	41
Figura 38: Implementación del listener de navegación	42
Figura 39: Pantalla principal del doctor	43
Figura 40: Constructor de la lista de pacientes.....	44
Figura 41: Función loadPatientsPage de DoctorModel	45
Figura 42: Función fetchPatientsPage de HttpDoctor	46

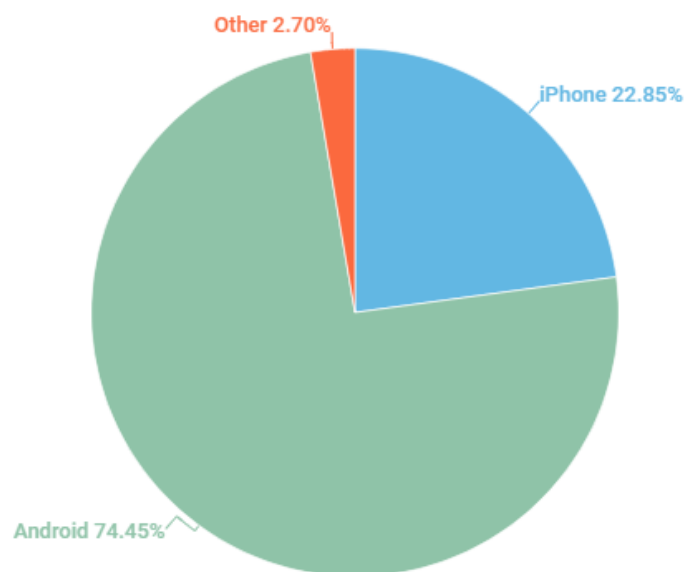
Figura 43: Pantalla de detalles del paciente.....	47
Figura 44: Función loadPatient de DoctorModel	48
Figura 45: Diálogo de confirmación al eliminar un paciente	48
Figura 46: Función deletePatient de DoctorModel.....	49
Figura 47: Pantalla de creación de paciente.....	50
Figura 48: Función addPatient de DoctorModel	50
Figura 49: Pantalla de código del nuevo paciente.....	51
Figura 50: Pantalla de términos y condiciones del paciente.....	52
Figura 51: Pantalla principal del paciente.....	52
Figura 52: Función loadPatientData de PatientModel.....	53
Figura 53: Pantallas de añadir medición.....	54
Figura 54: Función addMeasurement de PatientModel	54
Figura 55: Prueba de ejecución correcta de addPatient.....	55
Figura 56: Prueba de ejecución errónea de addPatient.....	56
Figura 57: Prueba del widget DoseDisplay cuando no hay dosis	57
Figura 58: Prueba del widget DoseDisplay para múltiplos de 40	57
Figura 59: Ejecución correcta de un proceso de build	58

1. Introducción

Desde la aparición de lo que conocemos hoy en día como *smartphones* con la presentación del primer iPhone el 29 de junio de 2007, estos dispositivos no han parado de crecer tanto en prestaciones como en número de usuarios. En España, el 92% de los ciudadanos era propietario de uno de estos dispositivos en 2016 [1].

Estos dispositivos se han convertido en elementos indispensables de nuestras vidas y, por tanto, también en una gran oportunidad de negocio para las empresas. Debido a que los usuarios siempre llevan su *smartphone* encima, permiten ofrecer un mayor servicio y, por consiguiente, obtener un mayor consumo por parte de estos. No solo abre la puerta a conseguir una mayor disponibilidad de servicios ya existentes, sino que también se abren nuevas posibilidades que ofrecer a los usuarios.

Es por ello que, para muchas organizaciones, sus aplicaciones móviles suponen una gran parte de sus ingresos y su mantenimiento y desarrollo es crucial. Sin embargo, el desarrollo de aplicaciones móviles presenta un reto: la inexistencia de una plataforma única. Actualmente, iOS y Android son los dos principales sistemas operativos, repartiéndose entre ellos más del 93% de la cuota de mercado [2], tal y como se puede apreciar en la Figura 1.



Figures covering Jan 2018 - Jan 2019 supplied by Statcounter

Figura 1: Cuota de mercado de los sistemas operativos móviles

El desarrollo de aplicaciones para cada uno de estos sistemas operativos es diferente tanto en metodología como en herramientas y tecnologías usadas. Por tanto, desarrollar y mantener una aplicación para cada plataforma implica mantener dos bases de código, con el coste en tiempo y dinero que esto supone, además de los problemas derivados de las inconsistencias que se pueden producir entre ambas versiones de la aplicación.

Por esta razón, durante los últimos años se han ido sacando al mercado tecnologías cuyo objetivo es sencillo: programar una vez y tener una aplicación funcional en las dos plataformas. Estas tecnologías son el objeto de estudio de este trabajo, concretamente una de reciente publicación llamada Flutter. Así pues, se van a estudiar sus ventajas e inconvenientes mediante el desarrollo de una aplicación móvil propia.

La aplicación que se va a desarrollar tendrá como objetivo ayudar al tratamiento de pacientes de cirrosis hepática. En estos tratamientos, el paciente comienza con una dosis asignada por su facultativo de un medicamento concreto y, tras un tiempo con esa dosis, el paciente vuelve a consulta para que se le realice una revisión y se reajuste la dosis. Lo que se busca es ir aumentando esta dosis hasta que se dé con la cantidad que produce un efecto betabloqueante. La aplicación permitirá acortar los tiempos de estos tratamientos al realizar el seguimiento de la dosis y su aumento a través de esta, sin necesidad de ir a consulta y siempre bajo el control del facultativo.

1.1. Motivación

A la hora de decidir sobre el tema de este trabajo, el desarrollo de aplicaciones móviles fue una de las principales opciones desde un primer momento, debido a que es un campo muy importante actualmente en el que apenas se profundiza durante los estudios del grado.

Cuando estaba pensando en qué trabajo realizar, fui contactado por un doctor del Hospital del Poniente, el cual estaba interesado en desarrollar una aplicación móvil para iOS y Android como parte de un estudio que pretendía realizar. La aplicación permitiría acortar los tiempos de tratamiento en pacientes con cirrosis hepática. Estos pacientes necesitan de un seguimiento sobre sus constantes vitales para controlar la dosis que puede ser realizado mediante sus *smartphones*.

Esta oportunidad me llevó finalmente a decantarme por la opción de enfocar el trabajo al desarrollo de esta aplicación usando herramientas multiplataforma. En concreto, elegí Flutter debido a ser una tecnología muy reciente y estar respaldada por Google, la cual lo promocionaba como un *framework* con ciclos de trabajo rápidos y fácil de aprender, por lo que se ajustaba perfectamente a mis necesidades.

1.2. Objetivos

- Estudiar y comparar las distintas soluciones presentes en el mercado para el desarrollo de aplicaciones móviles multiplataforma.
- Estudiar y comprender los principios básicos de Dart y Flutter.

- Estudiar y comparar los patrones de gestión del estado para aplicaciones creadas con Flutter.
- Desarrollar una aplicación móvil multiplataforma para la supervisión del tratamiento de pacientes con cirrosis hepática. La aplicación dará soporte tanto a pacientes como a doctores y funcionará tanto en Android como en iOS.
- Lograr que la interfaz de la aplicación desarrollada sea lo más intuitiva y clara posible, dado que esta será utilizada por usuarios de un gran rango de edades.
- Escribir una suite de pruebas que permita asegurar el correcto funcionamiento de la interfaz y de la lógica de la aplicación desarrollada.
- Configurar un sistema de integración y despliegue continuo para la aplicación que facilite la prueba y el despliegue de nuevas versiones de esta.

1.3. Planificación temporal

Se ha utilizado Scrum para realizar la planificación y el desarrollo del trabajo, dividiendo las tareas de este en *sprints*. Cabe destacar que, debido a los requisitos del anteproyecto, se presentó en el mismo una planificación por *sprints* en la que se explicaba el trabajo a realizar en cada *sprint*. Esto no es una práctica recomendable en Scrum, ya que la planificación de un *sprint* se produce al comienzo de este, eligiendo los requisitos del *Product Backlog* que se van a cumplir en el *sprint*.

En la Figura 2 se puede ver el desarrollo final de los seis *sprints* que han sido necesarios para llevar a cabo el trabajo.

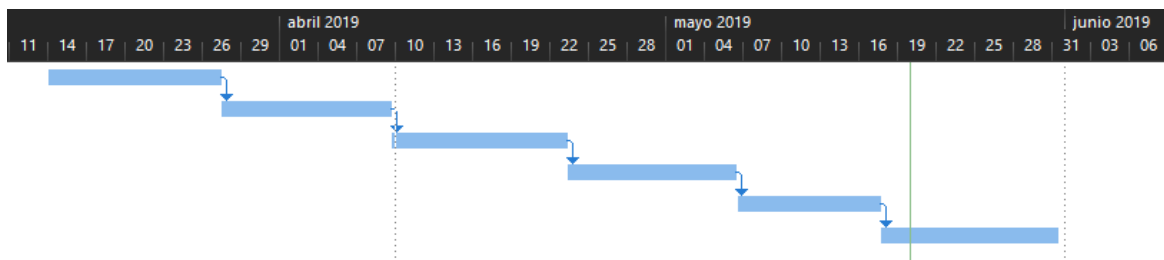


Figura 2: Diagrama de Gantt del proyecto

En el primer *sprint*, que tuvo lugar del 14 al 27 de marzo, se realizaron las primeras reuniones tanto con el doctor como con el director del TFG para garantizar la viabilidad del proyecto. Se identificaron los requisitos de la aplicación y se definieron las tecnologías y herramientas a usar.

Luego, hasta el 9 de abril, se procedió al modelado del sistema. Esto es, la refinación de los requisitos obtenidos del doctor, creación del diagrama de casos de uso y diseño de prototipos de la interfaz.

Del 9 al 23 de abril, todas las funcionalidades de la aplicación, recogidas en los requisitos, fueron implementadas siguiendo la arquitectura y patrones definidos en el *sprint* anterior.

A continuación, del 23 de abril al 6 de mayo, se definió el plan de pruebas de la aplicación y se escribieron las pruebas unitarias y de interfaz correspondientes.

Hasta el 17 de mayo, se configuró el entorno de integración y despliegue continuo de la aplicación que permite empujar fácilmente nuevas versiones a las tiendas de aplicaciones móviles.

Por último, del 17 de mayo al 31 del mismo mes, se redactó este documento detallando lo aprendido durante el proyecto.

1.4. Estructura de la memoria

En este primer capítulo se ha realizado una pequeña introducción al tema del trabajo desarrollado y los objetivos que se persiguen en el mismo, justificando su elección. A continuación, se indican cuáles son los siguientes capítulos y su contenido.

- Capítulo 2 – Estudio del dominio.

En este capítulo se explican aspectos relativos al contexto del proyecto que se desarrolla tales como la enfermedad a cuyo tratamiento intenta ayudar, sus beneficios y una breve introducción a otras herramientas de desarrollo de aplicaciones multiplataforma.

- Capítulo 3 – Tecnologías utilizadas.

En este capítulo se realizará una pequeña introducción a las tecnologías y herramientas utilizadas en el desarrollo de la aplicación y la consecución de los objetivos de este trabajo expuestos anteriormente.

- Capítulo 4 – Aplicación B-BlockMe.

En este capítulo se explican cada una de las fases de la creación de la aplicación móvil multiplataforma. Esto es, el análisis de requisitos, actores y casos de uso; el diseño de la arquitectura e interfaz; la implementación de las funcionalidades; las pruebas de la lógica y de la interfaz y el despliegue de la aplicación.

- Capítulo 5 – Conclusiones y trabajo futuro.

En este capítulo se van a abordar las conclusiones a las que se ha llegado durante la realización de este trabajo, así como presentar algunas líneas de trabajo que podrían extender lo realizado en este.

Por último, al final de este documento se incluye la bibliografía usada para la realización de este trabajo.

2. Estudio del dominio

Antes de comenzar a desarrollar el proyecto, se van a explicar detalles importantes del contexto de este. Por una parte, se va a explicar en qué consiste la enfermedad y el tratamiento al que intenta ayudar nuestra aplicación, además de presentar múltiples tecnologías alternativas que se podrían haber usado para compararlas con Flutter.

2.1. El tratamiento de la cirrosis hepática

La cirrosis hepática es la fase final de la enfermedad hepática crónica con fibrosis. El hígado se endurece y deja de funcionar correctamente al ser más difícil que pase la sangre por este. Esto puede provocar hemorragias y otras complicaciones como ascitis o intoxicación.

Por lo general, esta enfermedad no da señales ni tiene síntomas hasta que las lesiones hepáticas se hacen grandes. Cuando se da el caso de que sí hay síntomas, estos pueden incluir fatiga, pérdida de apetito, náuseas o hinchazón, entre otros. La causa más común de la cirrosis hepática es el consumo de alcohol en altas cantidades de manera crónica. Otras causas pueden ser las hepatitis B o C, la acumulación de grasa en el hígado por la obesidad o el consumo de ciertos medicamentos.

Aunque los daños producidos por esta enfermedad no se pueden revertir, sí que se puede parar su progreso y evitar que desarrolle complicaciones para el paciente. Una manera de conseguir esto es con el uso de betabloqueantes. Se trata de medicamentos capaces de estimular ciertos receptores del sistema nervioso para disminuir la frecuencia cardíaca, la presión arterial y las necesidades de oxígeno del corazón.

Este efecto betabloqueante se consigue con dosis concretas de medicación que dependen del paciente que se esté tratando. Por tanto, una de las fases del tratamiento consiste en identificar cuál es esta cantidad. Para ello, el paciente comienza a tomar la medicación y acude a consulta cada cierto tiempo para que el doctor revise sus constantes vitales y decida si aumenta la dosis. Esta dosis se va aumentando hasta alcanzar la cantidad que produce el efecto betabloqueante.

La utilidad de nuestra aplicación se centra en este procedimiento. En vez de tener que ir a consulta de forma regular, aprovechamos el *smartphone* del paciente para que introduzca sus constantes vitales (frecuencia cardíaca y presión arterial) diariamente. Es la propia aplicación la que, en base a estas constantes, aumenta la dosis del paciente hasta que se alcance la cantidad betabloqueante. La idea del estudio es que con la aplicación se puedan acortar los tiempos del tratamiento.

Se trata además de un área inexplorada hasta el momento. Se ha realizado un estudio de mercado antes de comenzar el proyecto y no se encontró ninguna aplicación que esté destinada al tratamiento de esta enfermedad. Nuestra aplicación es parte de un estudio médico innovador que trata de mejorar la calidad de este tratamiento.

2.2. Tecnologías multiplataforma alternativas

Nosotros vamos a desarrollar esta aplicación móvil para iOS y Android usando Flutter. Sin embargo, existen en el mercado otras herramientas para el desarrollo de aplicaciones multiplataforma. En los siguientes apartados se van a introducir algunas de ellas, indicando sus ventajas y desventajas.

2.2.1. React Native



Figura 3: Logo de React Native

Creado por Facebook, React Native es un *framework* utilizado principalmente para el desarrollo de aplicaciones móviles, aunque también permite trabajar para la plataforma universal de Windows. La idea principal es permitir el uso de React, una famosa librería para la construcción de interfaces de usuario mantenida por Facebook, en las plataformas nativas [3].

Así, una aplicación construida con esta herramienta usa componentes React para la interfaz y código JavaScript para la lógica. Simplificando, permite utilizar las mismas librerías usadas para el desarrollo de aplicaciones web, pero para plataformas móviles.

Esto hace que esta herramienta resulte especialmente atractiva para desarrolladores u organizaciones que ya tengan aplicaciones web implementadas con estas tecnologías. Sin embargo, la librería React tiene una curva de aprendizaje pronunciada, lo que dificulta su adopción por parte de usuarios nuevos.

Las aplicaciones desarrolladas con este *framework* no se compilan a las plataformas de destino, sino que su código es interpretado durante la ejecución en el dispositivo. Esto hace que sea más difícil detectar errores en el código durante el desarrollo, además de suponer un peor rendimiento frente a alternativas que sí que compilan el código, como es el caso de Flutter.

2.2.2. Ionic



Figura 4: Logo de Ionic

Ionic es un *framework* de código abierto basado en la tecnología de Apache Cordova que permite crear aplicaciones móviles híbridas. A diferencia de las aplicaciones nativas, las aplicaciones híbridas son implementadas usando tecnologías web (HTML, CSS y JavaScript) y ejecutadas en el dispositivo de destino aprovechando las capacidades de su motor de navegador [4].

El HTML usado para definir la interfaz es renderizado durante la ejecución, al igual que ocurre con el código JavaScript, el cual se interpreta sobre la marcha en el dispositivo. Como ocurría con React Native, el principal inconveniente de este acercamiento es la pérdida de rendimiento. Sin embargo, si comparamos estas dos herramientas, React Native sale favorecida por el hecho de apoyarse en un único lenguaje, JavaScript, en vez del conglomerado de tecnologías que usa Ionic.

2.2.3. Xamarin



Figura 5: Logo de Xamarin

En sus comienzos, Xamarin era una empresa que desarrolló implementaciones de la plataforma .NET de Microsoft para plataformas móviles. Microsoft acabó comprando esta empresa y usando su nombre para identificar al conjunto de herramientas que

permiten utilizar una base de código C# para el desarrollo de aplicaciones en diversas plataformas, permitiendo la compartición de código entre estas.

Con Xamarin, se puede usar C# no sólo para la implementación de la lógica de negocio, sino que también provee de controles de interfaz personalizados para cada plataforma [5]. Sin embargo, esto hace que la implementación de estas interfaces con C# no se pueda compartir entre plataformas, haciendo que el porcentaje de código compartido sea mucho menor que en otras soluciones, como Flutter.

Por último, otra de las desventajas de Xamarin es el gran tamaño de sus aplicaciones. Las aplicaciones desarrolladas con esta herramienta se compilan para cada plataforma, pero siguen teniendo un tamaño que, en ocasiones, es notablemente mayor que el de aplicaciones nativas.

2.3. Tecnología multiplataforma elegida

De todas las tecnologías presentadas en el apartado 2.2, la más atractiva era React Native. Frente a Ionic o Xamarin, React Native apuesta por la simplicidad y el uso de una única tecnología. Permite construir aplicaciones mucho menos pesadas que Xamarin y es capaz de aprovechar mejor las capacidades de cada dispositivo que Ionic. Además, la madurez de la herramienta y la gran comunidad que tiene detrás son también un gran incentivo a la hora de decantarse por React Native.

Sin embargo, React Native tiene un problema principal, que es el hecho de que su código es interpretado durante la ejecución, lo que provoca que el acceso a las funcionalidades nativas de la plataforma se realice a través de un puente. Esto puede causar problemas de rendimiento, por lo que muchas veces se programan las aplicaciones con React Native intentando mantener el uso de este puente al mínimo.

Flutter no sufre de estas limitaciones. Este *framework* usa un lenguaje compilado llamado Dart. El código escrito en este lenguaje es compilado antes de tiempo (AOT) a código nativo para cada plataforma. De esta forma, las aplicaciones escritas con Flutter se pueden comunicar con la plataforma sin necesidad de usar un puente. La compilación del código a código nativo de las plataformas supone un mayor rendimiento para las aplicaciones escritas con Flutter del que obtendríamos usando React Native.

Mirando al mercado, Flutter es el único SDK de desarrollo móvil que permite la creación de vistas reactivas sin necesidad de usar un puente JavaScript. Es por esta razón que nos decantamos por esta herramienta sobre todas las demás.

Se profundizará más en el funcionamiento de Flutter en el capítulo 3 junto con las demás tecnologías y herramientas utilizadas.

3. Tecnologías utilizadas

A continuación, se van a introducir una a una las tecnologías y herramientas utilizadas en la realización de este trabajo.

3.1. Dart



Figura 6: Logo de Dart

Dart es un lenguaje de programación de código abierto creado por Google en 2011 con la intención de proporcionar una alternativa más moderna a JavaScript. Se trata de un lenguaje especialmente optimizado para la creación de interfaces de usuario [6].

Es el lenguaje que se utiliza para la creación de aplicaciones con Flutter. Existen varias razones por las que este lenguaje es el ideal para esta herramienta. La razón principal es su versatilidad. Dart permite la compilación AOT (*Ahead Of Time*), con lo cual se obtiene código nativo con un mucho mejor rendimiento que si se usase un lenguaje interpretado como JavaScript. El código Dart también puede ser compilado sobre la marcha (JIT, *Just In Time*), con lo que se da soporte a la funcionalidad de recarga rápida de Flutter para la actualización del código durante la ejecución.

De esta forma, el código Dart puede usar una compilación JIT durante el desarrollo para acelerar los tiempos de implementación. Una vez terminada la implementación, la compilación AOT produce el código nativo optimizado y rápido. Las herramientas de Dart permiten tener ciclos de desarrollo rápidos y tiempos de ejecución y lanzamiento breves.

También se puede compilar el código Dart a JavaScript de manera directa, lo que permite su ejecución en navegadores. Con esto se puede compartir código entre aplicaciones web y móviles. Por último, Dart posee su propia máquina virtual que utiliza el propio lenguaje como lenguaje intermedio, actuando básicamente como un intérprete.

Se trata de un lenguaje declarativo y muy fácil de leer y visualizar, por lo que no necesita de otro lenguaje como XML o JSX para la construcción de interfaces. Todo el código de la aplicación se puede implementar en un mismo lenguaje.

Otro aspecto importante es el hecho de que Dart, al igual que JavaScript, trabaja en un único hilo, lo que evita la ejecución preventiva de código. Esto es especialmente relevante para la implementación de interfaces o comunicación entre cliente y servidor, ya que permite al desarrollador asegurar que las funciones críticas se ejecutarán completamente.

Debido a que es el lenguaje utilizado en Flutter, usaremos Dart para la implementación de la aplicación. Esta decisión se ve incentivada también por la simplicidad del lenguaje y su parecido a otros lenguajes ya conocidos como Java y JavaScript.

3.2. Flutter



Figura 7: Logo de Flutter

Flutter es un *framework* de desarrollo de aplicaciones móviles multiplataforma creado por Google. Es de código abierto y permite construir aplicaciones tanto para Android como para iOS. Su versión 1.0 fue lanzada al mundo el 4 de diciembre de 2018, por lo que es una tecnología muy nueva. A pesar de su corta edad, se trata de una tecnología muy madura debido a que es utilizada en Google para crear sus herramientas internas.

Está construido por capas, estando el motor escrito en C/C++ y las librerías en Dart. También usa Skia para el renderizado 2D. El objetivo de esta herramienta es permitir a los desarrolladores construir aplicaciones multiplataforma a partir de una única base de código, la cual es compilada a código nativo para cada una de las plataformas objetivo. Además, se aprovecha de la flexibilidad de Dart en cuanto a su compilación y ejecución para obtener ciclos de desarrollo más rápidos y tiempos de ejecución más bajos.

A diferencia de otras soluciones, en Flutter se construye toda la aplicación usando Dart, incluida la interfaz de usuario. Para ello, se apoya fuertemente en el paradigma de la programación orientada a objetos, más concretamente en la composición y herencia de los llamados *widgets* [7], los cuales componen la interfaz. Estos *widgets* son fundamentales para cualquier aplicación móvil. La peculiaridad de Flutter respecto a otras soluciones se encuentra en que no utiliza los *widgets* de la plataforma, sino que provee los suyos propios. Flutter construye los *widgets* a nivel de aplicación, lo que hace que el desarrollador pueda personalizarlos y extenderlos de manera sencilla, consiguiendo una mayor libertad en el diseño de las aplicaciones. Esto se consigue gracias a que Flutter solo requiere de la plataforma un *canvas* o lienzo donde "pintar" los *widgets*.

Otro de los puntos fuertes de Flutter es la gran calidad de sus herramientas de desarrollo. Una de estas herramientas, y quizás la más popular, es la recarga rápida. Al usar otras tecnologías, el desarrollador tiene reiniciar la aplicación cada vez que haga un cambio en el código y quiera ver su efecto. La recarga rápida de Flutter hace uso de la compilación JIT de Dart para reflejar los cambios en el código durante la ejecución de manera casi instantánea manteniendo el estado. Esto contribuye a reducir los tiempos de los ciclos de desarrollo.

La decisión de usar Flutter frente a otras herramientas para el desarrollo de aplicaciones multiplataforma como las comentadas en el apartado 2.2 viene dada principalmente por ofrecer un mayor rendimiento, como ya se indicó en el apartado 2.3. Se quiere estudiar en este trabajo la viabilidad de esta nueva tecnología. Además, Flutter tiene una curva de aprendizaje muy inferior a las demás soluciones, principalmente por el uso de un único lenguaje simple y sencillo.

También se tuvo en cuenta la descripción de Flutter como una herramienta que facilita la iteración rápida sobre el producto, ya que los plazos para la realización de este proyecto eran limitados.

3.3. Android Studio



Figura 8: Logo de Android Studio

Android Studio es el entorno de desarrollo oficial para la creación de aplicaciones para Android. Está basado en otro entorno llamado IntelliJ IDEA, con el que comparte el editor de texto y otras herramientas para desarrolladores. A estas herramientas se añaden otras funciones específicas para el desarrollo de aplicaciones Android.

Algunas de estas funciones que aporta Android Studio para facilitar el desarrollo son:

- Un sistema de compilación flexible basado en Gradle.
- Un rápido emulador con varias funciones.

- Un entorno unificado en el que se pueden realizar desarrollos para todos los dispositivos Android (móviles, *wearables*, TV, ...).
- Visualización de los cambios realizados a la aplicación sin la necesidad de compilar una nueva APK.
- Gran cantidad de herramientas y *frameworks* de prueba.
- Herramientas de inspección del código para detectar problemas de rendimiento, usabilidad, compatibilidad, etc.
- Soporte integrado para Google Cloud Platform, la plataforma de computación en la nube de Google.

Además de todo esto, en la instalación de Android Studio se incorpora también el SDK de Android y sus herramientas. Esto es necesario para poder compilar una APK que se pueda ejecutar en un dispositivo Android.

Nosotros instalaremos Android Studio para obtener este SDK y poder compilar nuestras aplicaciones desarrolladas con Flutter. También haremos uso de otras herramientas que incorpora el IDE como el simulador.

3.4. Visual Studio Code



Figura 9: Logo de Visual Studio Code

Visual Studio Code es un editor de texto desarrollado por Microsoft usando tecnologías de código abierto como Electron. Al ser solo un editor, no incorpora tantas funcionalidades como un entorno de desarrollo completo. Viene de serie con Git para el control de versiones del código, IntelliSense para la detección de errores y las sugerencias de código y herramientas de depuración integradas.

El verdadero potencial de este editor se encuentra en la personalización. Más allá de la infinidad de temas y fuentes de entre los que elegir, podemos instalar extensiones de su mercado propio, la mayoría de ellas mantenidas por la comunidad. Estas extensiones añaden soporte para lenguajes específicos o proporcionan herramientas adicionales para la realización de algunas tareas.

En nuestro caso, se van a utilizar las extensiones de Dart y Flutter, las cuales proporcionan soporte tanto para el lenguaje como para las peculiaridades del *framework*. Sobre todo, permiten ejecutar la aplicación que se está desarrollando y depurar el código desde el editor directamente.

Aunque podríamos haber desarrollado la aplicación usando únicamente Android Studio, ya que este IDE tenía que ser instalado sí o sí para obtener las herramientas de Android, la elección de Visual Studio Code viene dada por su velocidad. La menor cantidad de características del editor de texto hacen que sea mucho más liviano y rápido. Además, no necesitamos usar la mayoría de las funcionalidades que aporta Android Studio.

3.5. JSON

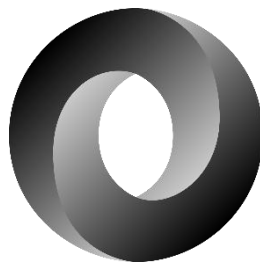


Figura 10: Logo de JSON

Al transmitir datos a través de una red, necesitamos usar un formato con el que representar la información y que conozcan tanto emisor como receptor para poder comprender el contenido. Uno de estos formatos es JSON (*JavaScript Object Notation*) [8]. JSON fue creado con el objetivo de ser sencillo de generar y procesar por máquinas además de ser fácil de leer y escribir para los humanos.

A pesar de estar basado en JavaScript, se trata de un formato de texto completamente independiente del lenguaje que se use. Consta de dos estructuras: objetos y colecciones. Los objetos son conjuntos no ordenados de pares clave-valor. La sintaxis que se usa para escribir un objeto se puede ver en la Figura 11.

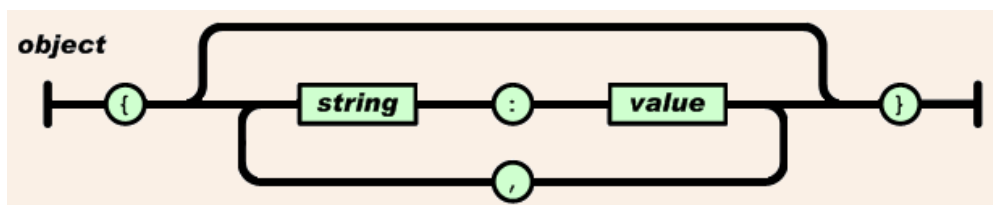


Figura 11: Definición de un objeto JSON

Por otra parte, las colecciones son conjuntos ordenados de valores. La sintaxis utilizada para definir estas colecciones se puede ver en la Figura 12.

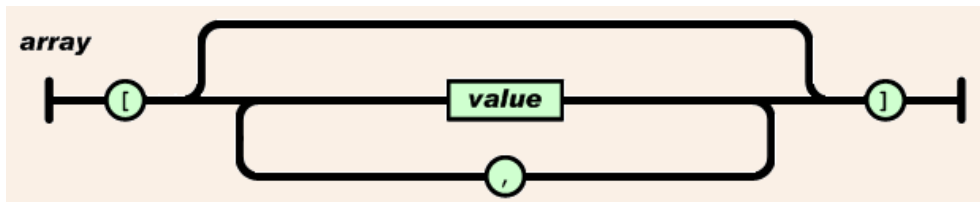


Figura 12: Definición de una colección JSON

En JSON, un valor puede ser una cadena de texto con comillas dobles, un número, un valor booleano (*true* o *false*), un valor nulo o, incluso, objetos y colecciones. Esto permite que las estructuras se puedan anidar, dando lugar a representaciones más complejas.

Usaremos JSON como formato para la transmisión de información entre nuestra aplicación móvil y el servicio web. Se ha elegido este formato frente a otros por su simplicidad y facilidad de uso.

3.6. Codemagic



Figura 13: Logo de Codemagic

Codemagic es una herramienta online de integración y despliegue continuo construida específicamente para aplicaciones creadas con Flutter. Este servicio, que salió junto con la versión 1.0 de Flutter, funciona enteramente online sin tener que añadir ningún tipo de configuración especial al proyecto como ocurre con otras herramientas de integración continua como Travis. Directamente se vincula con la cuenta de GitHub y permite escuchar a los *push* o *pull request* que se producen en el repositorio para lanzar un proceso de *build*.

Estos procesos se componen de varias fases o tareas, en las que se recupera el código del repositorio y se realizan las distintas operaciones deseadas. Con Codemagic, podemos ejecutar las pruebas escritas para el proyecto y, si se ejecutan correctamente, compilar el código a las plataformas deseadas y obtener los artefactos derivados.

Uno de los aspectos más importantes de Codemagic es que nos permite automatizar el proceso de firmado del código y el despliegue de las nuevas versiones de la aplicación a las tiendas de aplicaciones de las distintas plataformas.

Usaremos Codemagic en este proyecto para automatizar la ejecución de las pruebas y el despliegue de la aplicación de manera sencilla y facilitar así el mantenimiento de esta.

3.7. Material Design

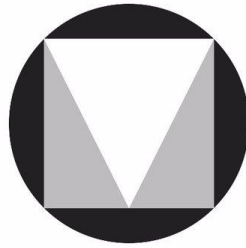


Figura 14: Logo de Material Design

Material Design es un lenguaje visual creado por Google para la creación de interfaces de usuario. Trata de juntar los principios básicos del buen diseño junto con las innovaciones en tecnología y ciencia.

Se trata de una norma sobre los distintos componentes que forman parte de una interfaz [9] como son los botones, los campos de texto o los selectores múltiples. Se define cómo deben ser estos componentes y cuál debe ser su comportamiento con una serie de reglas y recomendaciones.

No solo se habla de componentes en Material Design, sino que también contiene normas para la navegación entre pantallas, las animaciones, los colores utilizados y la accesibilidad para usuarios con necesidades especiales.

Aplicamos las normas de Material Design al diseño de la interfaz de nuestra aplicación, que luego se implementará usando una librería de Flutter que aporta componentes ya definidos con estos criterios.

4. Aplicación B-BlockMe

La aplicación de la que se ha estado hablando en este trabajo y cuyo desarrollo se va a cubrir a continuación recibe el nombre de B-BlockMe. Su principal objetivo es permitir a los doctores añadir pacientes en tratamiento de cirrosis hepática, pudiendo ver su progreso. Los pacientes, por su parte, pueden usar la aplicación para introducir sus constantes vitales y controlar las dosis del tratamiento.

Ahora, se van a cubrir una a una las distintas fases en las que se ha dividido la creación de esta aplicación.

4.1. Análisis

La primera fase del desarrollo de la aplicación es la de análisis, en la que se definen tanto los objetivos de esta como los requisitos que se deben satisfacer para cumplirlos. Tras una serie de reuniones con el doctor, en las que este nos expresa su visión de la aplicación, se realiza una revisión de la información dada para obtener una especificación más refinada.

4.1.1. Objetivos

A continuación, se describen los objetivos principales que dirigen el desarrollo de esta aplicación.

Objetivo 01	Interfaz simple e intuitiva
Versión	1
Autor	Víctor Vázquez Rodríguez
Descripción	La aplicación deberá tener una interfaz lo más clara e intuitiva posible que facilite su uso.
Importancia	Alta
Estabilidad	Alta
Comentarios	Los usuarios de esta aplicación serán muy diversos, pudiendo ser personas de avanzada edad o con escasa habilidad con la tecnología, por lo que se debe simplificar al máximo su uso.

Objetivo 02	Autenticación de usuarios
Versión	1
Autor	Víctor Vázquez Rodríguez
Descripción	Los usuarios podrán registrarse y autenticarse en la aplicación.
Importancia	Muy alta
Estabilidad	Alta
Comentarios	Se cuenta con dos tipos de usuarios: doctores y pacientes. Los doctores podrán crear una nueva cuenta con la que acceder, mientras que los pacientes solo podrán acceder con el código proporcionado por su doctor.

Objetivo 03	Introducción de mediciones
Versión	1
Autor	Víctor Vázquez Rodríguez
Descripción	Los pacientes podrán introducir una nueva medición de sus constantes vitales cada día durante la duración del tratamiento.
Importancia	Muy alta
Estabilidad	Alta
Comentarios	Las constantes vitales que se deben recoger son la frecuencia cardíaca y la presión arterial (tanto sistólica como diastólica), además de un indicador en caso de sufrir algún síntoma.

Objetivo 04	Soporte multiplataforma
Versión	1
Autor	Víctor Vázquez Rodríguez
Descripción	La aplicación deberá funcionar tanto en Android como en iOS.
Importancia	Alta
Estabilidad	Alta
Comentarios	Ninguno

Objetivo 05	Comunicación con servidor web
Versión	1
Autor	Víctor Vázquez Rodríguez
Descripción	La aplicación se comunicará con un servidor web para enviar y recibir información.
Importancia	Alta
Estabilidad	Alta
Comentarios	Debido a la sensibilidad de los datos, estos no se almacenarán en el dispositivo móvil, sino que se guardarán en el servidor web con el que se comunicará la aplicación.

Objetivo 06	Gestión de pacientes
Versión	1
Autor	Víctor Vázquez Rodríguez
Descripción	Los doctores deberán poder ver los pacientes que han añadido, obteniendo información sobre el progreso de sus tratamientos. También podrán añadir nuevos pacientes y borrar los existentes.
Importancia	Muy alta
Estabilidad	Alta
Comentarios	Ninguno

4.1.2. Actores

Los actores son los distintos tipos de usuarios que puede haber en la aplicación y que tendrán acceso a las funcionalidades de esta.

Actor 01	Usuario no autenticado
Descripción	Aquel usuario que todavía no ha introducido credenciales que le acrediten como doctor o paciente.
Requisitos asociados	RF-01 Registrarse RF-02 Iniciar sesión (doctor) RF-03 Iniciar sesión (paciente)

Actor 02	Doctor
Descripción	Usuario que se ha autenticado como doctor y tiene acceso a la lista de pacientes que gestiona.
Requisitos asociados	RF-04 Ver listado de pacientes RF-05 Ver detalles del paciente RF-06 Borrar paciente RF-07 Añadir paciente RF-08 Simular introducción de medición RF-09 Cerrar sesión

Actor 03	Paciente
Descripción	Usuario que se ha autenticado como paciente y puede ver el estado de su tratamiento e introducir una nueva medición cada día.
Requisitos asociados	RF-10 Ver estado del tratamiento RF-11 Introducir medición

4.1.3. Requisitos funcionales

Aquí, nos encontramos con los requisitos que describen las funcionalidades presentes en la aplicación. En el diagrama de casos de uso de la Figura 15, se pueden ver las relaciones entre estas funcionalidades y los actores que tienen acceso a ellas.



Figura 15: Diagrama de casos de uso

Cada uno de estos casos de uso tiene una serie de pasos que describen el flujo de su ejecución por parte del usuario. La descripción detallada de estos requisitos se describe a continuación.

RF-01	Registrarse	
Actor asociado	Actor 01 – Usuario no autenticado	
Descripción	El personal sanitario que no disponga de cuenta podrá crear una introduciendo los datos necesarios.	
Precondición	Ninguna	
Secuencia normal	Paso 1	Pulsar en "Acceso personal sanitario" en la pantalla de inicio de sesión del paciente.
	Paso 2	Pulsar en "Registrarse".
	Paso 3	Introducir los datos requeridos en el formulario.
	Paso 4	Pulsar en "Registrarse"
Postcondición	Se crea una nueva cuenta en el sistema para el usuario	
Excepción	<ul style="list-style-type: none"> • Si el email introducido ya está en uso no se realiza el registro y se notifica al usuario. • Si la contraseña introducida no contiene tanto letras como números no se realiza el registro y se notifica al usuario. • Si la contraseña no supera los 6 caracteres de longitud no se realiza el registro y se notifica al usuario. 	

RF-02	Iniciar sesión (doctor)	
Actor asociado	Actor 01 – Usuario no autenticado	
Descripción	Un usuario podrá acceder a su cuenta de personal sanitario si posee una introduciendo sus credenciales.	
Precondición	Ninguna	
Secuencia normal	Paso 1	Pulsar en "Acceso personal sanitario" en la pantalla de inicio de sesión del paciente.
	Paso 2	Introducir email y contraseña en el formulario.
	Paso 3	Pulsar en "Acceder".
Postcondición	El usuario está autenticado como doctor y es redirigido a su lista de pacientes.	
Excepción	<ul style="list-style-type: none"> • Si las credenciales introducidas no son correctas, no se realiza la autenticación y se notifica al usuario. 	

RF-03	Iniciar sesión (paciente)	
Actor asociado	Actor 01 – Usuario no autenticado	
Descripción	Un usuario se podrá autenticar como paciente al introducir el código proporcionado por su doctor.	
Precondición	Ninguna	
Secuencia normal	Paso 1	Introducir el código de acceso en el formulario de la pantalla de inicio de sesión del paciente.
	Paso 2	Pulsar en "Acceder".
Postcondición	El usuario está autenticado como paciente y es redirigido a la pantalla de términos y condiciones.	
Excepción	<ul style="list-style-type: none"> • Si el código dado no existe, no se realiza la autenticación y se notifica al usuario. 	

RF-04	Ver listado de pacientes	
Actor asociado	Actor 02 – Doctor	
Descripción	El doctor podrá ver el listado de pacientes que ha añadido a la aplicación.	
Precondición	El usuario debe estar autenticado como doctor.	
Secuencia normal	Paso 1	Estar en la pantalla principal del doctor.
Postcondición	Se muestran los pacientes que gestiona el doctor.	
Excepción	<ul style="list-style-type: none"> • Si no hay ningún paciente, la lista estará vacía. 	

RF-05	Ver detalles del paciente	
Actor asociado	Actor 02 – Doctor	
Descripción	El doctor podrá ver datos detallados sobre un paciente y su tratamiento, así como un gráfico que muestre la progresión de las constantes vitales del mismo a lo largo del tiempo.	
Precondición	El usuario debe estar autenticado como doctor y tener algún paciente añadido.	
Secuencia normal	Paso 1	Pulsar sobre el paciente en la lista de pacientes.

Postcondición	Se muestran los datos detallados del paciente seleccionado y su tratamiento.
Excepción	<ul style="list-style-type: none"> Si el paciente todavía no ha introducido ninguna medición, en lugar del gráfico se muestra un mensaje notificando de esta circunstancia al usuario.

RF-06	Borrar paciente	
Actor asociado	Actor 02 – Doctor	
Descripción	El doctor podrá eliminar pacientes bajo su propio criterio tanto durante el tratamiento como al finalizar el mismo.	
Precondición	El usuario debe estar autenticado como doctor y tener algún paciente añadido.	
Secuencia normal	Paso 1	Pulsar sobre el icono de la papelera en la pantalla de detalles del paciente.
	Paso 2	Pulsar en "Aceptar".
Postcondición	Todos los datos del paciente y su tratamiento son eliminados del sistema.	
Excepción	Ninguna.	

RF-07	Añadir paciente	
Actor asociado	Actor 02 – Doctor	
Descripción	El doctor podrá añadir nuevos pacientes indicando los datos apropiados para su tratamiento.	
Precondición	El usuario debe estar autenticado como doctor.	
Secuencia normal	Paso 1	Pulsar sobre el icono de "más" en la pantalla de la lista de pacientes.
	Paso 2	Introducir los datos requeridos en el formulario.
	Paso 3	Pulsar en "Aceptar".
Postcondición	Se ha creado un nuevo paciente y se redirige al doctor a la pantalla que le informa del código generado para este.	
Excepción	<ul style="list-style-type: none"> Si el número de historia introducido ya está en uso, no se realiza la creación del paciente y se notifica al usuario. 	

RF-08	Simular introducción de medición	
Actor asociado	Actor 02 – Doctor	
Descripción	El doctor podrá realizar una simulación de la introducción de constantes vitales en una medición para mostrar el proceso a su paciente.	
Precondición	El usuario debe estar autenticado como doctor y haber añadido un nuevo paciente.	
Secuencia normal	Paso 1	Pulsar en "Ir a simulación" en la pantalla que muestra el código del paciente recién creado.
	Paso 2	Introducir una frecuencia cardíaca.

	Paso 3	Introducir una presión arterial (tanto sistólica como diastólica).
	Paso 4	Indicar si se han notado alguno de los síntomas indicados.
	Paso 5	Pulsar en "Aceptar".
Postcondición	El doctor es redirigido de nuevo a la pantalla del código del nuevo paciente.	
Excepción	Ninguna.	

RF-09	Cerrar sesión	
Actor asociado	Actor 02 – Doctor	
Descripción	El doctor podrá salir de su cuenta siempre que lo desee.	
Precondición	El usuario debe estar autenticado como doctor.	
Secuencia normal	Paso 1	Pulsar en el icono de "salida" en la pantalla de la lista de pacientes.
Postcondición	El usuario ya no está autenticado como doctor y es redirigido a la pantalla de inicio de sesión.	
Excepción	Ninguna.	

RF-10	Ver estado del tratamiento	
Actor asociado	Actor 03 – Paciente	
Descripción	El paciente podrá ver en todo momento la dosis actual de su tratamiento, así como notificaciones que le indiquen si debe ir al médico.	
Precondición	El usuario debe estar autenticado como paciente.	
Secuencia normal	Paso 1	Marcar la casilla indicando que está de acuerdo con los términos y condiciones.
	Paso 2	Pulsar en "Aceptar".
Postcondición	El usuario ve la información relativa a la dosis actual y al estado de su tratamiento.	
Excepción	<ul style="list-style-type: none"> Si el usuario pulsa "Cancelar" en la pantalla de términos y condiciones, se cierra su sesión y se redirigido a la pantalla de inicio de sesión. 	

RF-11	Introducir medición	
Actor asociado	Actor 03 – Paciente	
Descripción	El paciente podrá introducir cada día una nueva medición de sus constantes vitales mientras dure el tratamiento.	
Precondición	El usuario debe estar autenticado como paciente.	
Secuencia normal	Paso 1	Pulsar el icono de "más" en la pantalla principal del paciente.
	Paso 2	Introducir una frecuencia cardíaca.
	Paso 3	Introducir una presión arterial (tanto sistólica como diastólica).

	Paso 4	Indicar si se han notado alguno de los síntomas indicados.
	Paso 5	Pulsar en "Aceptar".
Postcondición	Se registra la nueva medición y el usuario es redirigido a su pantalla principal, no pudiendo añadir otra medición hasta el día siguiente.	
Excepción	Ninguna.	

4.1.4. Requisitos no funcionales

También existen otros requerimientos que debe cumplir la aplicación que, aunque no constituyan funcionalidades por sí mismos, sí que condicionan la manera en la que se desarrollan estas. En este apartado se describen de manera formal estos requisitos no funcionales de la aplicación.

RNF-01	Usabilidad
Descripción	La aplicación va a ser utilizada por personas de todo tipo de edades y con una habilidad con la tecnología variable. Por tanto, se debe garantizar la facilidad de uso, especialmente por parte de los pacientes.
Comentarios	Se aplicará un diseño minimalista, intentado crear una interfaz simple y lo más limpia posible.

RNF-02	Requisitos legales
Descripción	La aplicación y el tratamiento que esta realiza de sus datos deberá cumplir con los requisitos legales en esta materia debido a la sensibilidad de esta información.
Comentarios	Ya que no se utilizan datos que identifiquen a los pacientes ni se almacena información en los dispositivos, no se tiene que aplicar la normativa especial para datos sensibles.

RNF-03	Seguridad
Descripción	El acceso a la información del servicio web a través de la aplicación debe realizarse de la manera más segura posible.
Comentarios	Comunicación a través de HTTPS y uso del estándar OAuth2 para las cuentas de los doctores.

RNF-04	Fluidez
Descripción	La aplicación debe ser lo más fluida posible para no afectar a la experiencia de uso del usuario.
Comentarios	Se presta especial atención a los indicadores de rendimiento y se depura el código para maximizarlos.

4.1.5. Requisitos de información

Los requisitos de información definen los datos que almacena el sistema. Como ya se ha indicado en algunas ocasiones, nuestra aplicación no almacena ninguna información en los dispositivos móviles, delegando esta responsabilidad a un servicio web con el que se comunica a través de la red.

Los únicos datos que son almacenados de manera más o menos persistente son los de autenticación de los usuarios: El token de acceso a la API para los doctores y el código de acceso para los pacientes. Estos datos se almacenan con el propósito de evitar que el usuario tenga que introducir sus credenciales cada vez que accede a la aplicación.

4.2. Diseño

En la fase de diseño, se definieron varios puntos importantes de la aplicación antes de proceder a su creación. Era necesario definir cómo iba a ser la interacción con el servicio web y cómo iba a gestionar la aplicación su estado. Además, también se realizaron prototipos de las distintas pantallas de la aplicación para que fueran la base de la creación de la interfaz final.

4.2.1. Arquitectura cliente servidor

En este trabajo ya se ha comentado que la aplicación no almacena información en los dispositivos móviles, sino que se comunica con un servicio web el cual gestiona todos los datos, además de encargarse de la lógica que controla los tratamientos.

De esta forma, la mayoría de las funcionalidades de la aplicación requieren del servicio web, siendo la aplicación encargada solamente de controlar la comunicación con este y proporcionar al usuario una interfaz donde pueda introducir los datos que se envían y ver los que se reciben.

Para ilustrar esta situación, en la Figura 16 se presenta un diagrama de secuencias. En él, se representa a grandes rasgos el flujo de funcionamiento del inicio de sesión como doctor por parte de un usuario.

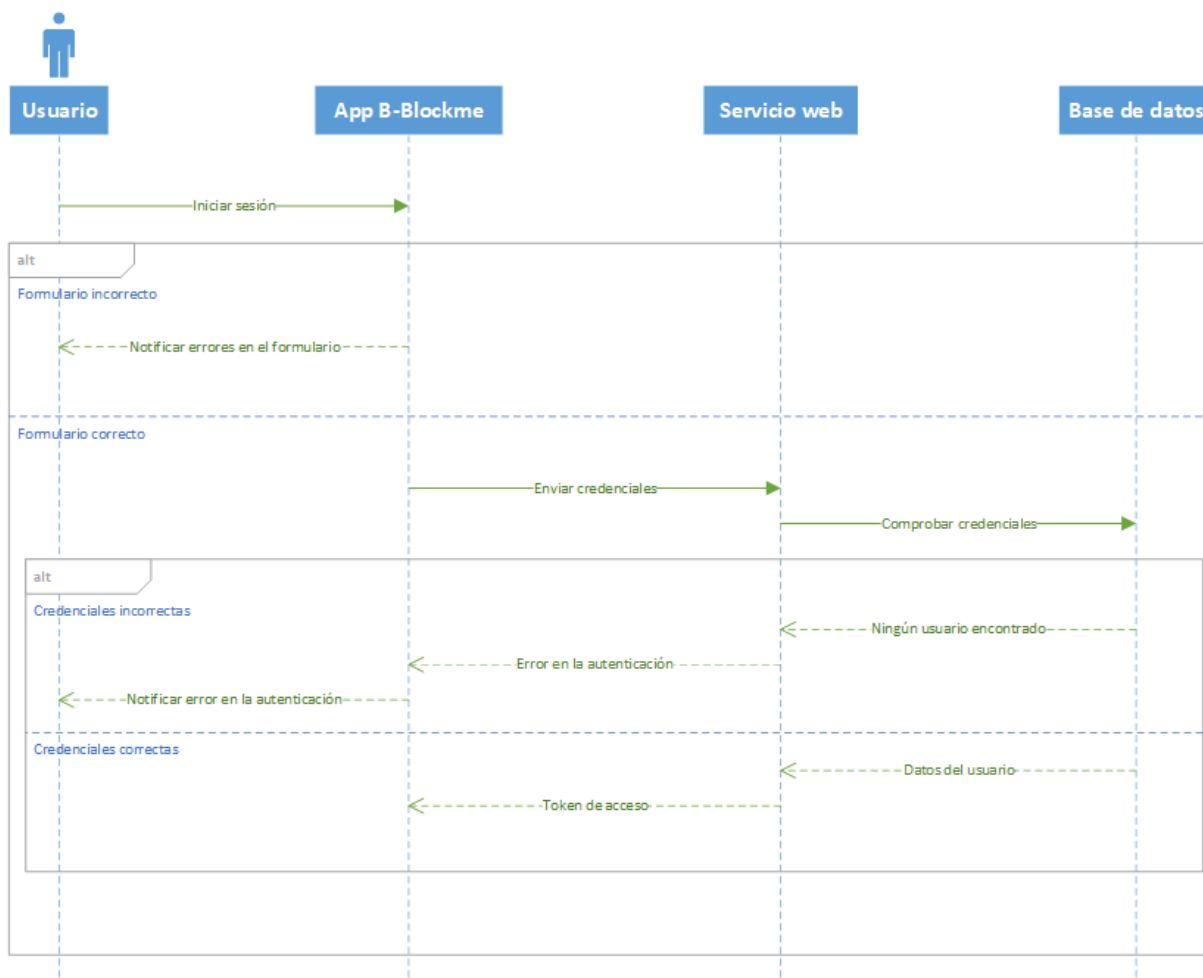


Figura 16: Diagrama de secuencias de la interacción con el servicio web

El proceso comienza con el usuario introduciendo unas credenciales de acceso en un formulario de la aplicación. El formulario comprobará que los datos introducidos son válidos, en cuyo caso, envía las credenciales al servicio web. Si no son válidos, se notifica al usuario en la interfaz para que los corrija y no se llegan a enviar.

Una vez recibidas las credenciales, el servicio web busca en la base de datos algún usuario registrado que coincida con estas. Una vez encontrado, se recuperan los datos de dicho usuario y el servicio envía a la aplicación un token de acceso como respuesta. Este token será necesario para las siguientes peticiones que realizará el usuario como doctor, ya que los recursos del servidor están protegidos.

En caso de que no se encuentre ningún usuario que coincida con las credenciales dadas, el servicio responde a la petición de la aplicación con un error, el cual se notifica al usuario a través de la interfaz.

4.2.2. Gestión del estado

A la hora de construir una aplicación, independientemente de la plataforma, uno de los aspectos a los que hay que prestar más atención es a la gestión del estado. Por regla general, en una aplicación se muestran al usuario datos que pueden verse alterados por las entradas de este. Los datos mostrados por pantalla deben cambiar durante la ejecución de la aplicación.

Podemos considerar entonces que la aplicación tiene un estado, a partir del cual se construye la interfaz que se muestra al usuario. Este estado se puede ver alterado, por ejemplo, por las aportaciones del usuario o los datos que lleguen de un servicio web.

Flutter ofrece una manera de gestionar el estado: *StatelessWidget* y *StatefulWidget*. En las aplicaciones construidas con Flutter, todo son *widgets*, desde los botones y campos de texto hasta el navegador que realiza las transiciones entre pantallas. Estos objetos se componen formando un árbol, que da lugar a la interfaz y a su comportamiento. Un *StatelessWidget* es un objeto que no posee estado y, por tanto, no se reconstruye. Por otra parte, un *StatefulWidget* sí que posee estado, el cual puede alterarse para que se reconstruya el objeto y refleje los cambios.

Sin embargo, el uso de esta solución para manejar el estado implica que tanto este como la lógica que lo altera estén fuertemente acoplados a la interfaz, lo cual no se considera una buena práctica. Las aplicaciones deben construirse por componentes y capas independientes, facilitando su reutilización, desarrollo y mantenimiento.

Existen diversos patrones arquitectónicos para la gestión del estado de una aplicación manteniendo la separación por capas [10]. Nosotros hemos elegido *ScopedModel* por su simplicidad. Esta librería se aprovecha de la existencia de otro tipo de *widget* llamado *InheritedWidget*. Este objeto se sitúa en el árbol y es accesible desde todos los demás objetos ubicados por debajo. Aprovechando esto, *ScopedModel* ubica un objeto de Modelo en el árbol. Este Modelo tiene una serie de atributos de solo lectura, además de funciones públicas. Los *widgets* ubicados en el árbol por debajo del *ScopedModel* pueden acceder a su Modelo asociado y leer los atributos o llamar a las funciones. Estos atributos constituyen el estado de la aplicación, y las funciones son las encargadas de la lógica que altera este estado, incluyendo llamadas a bases de datos, servidores, etc.

Esta librería aporta otro objeto importante llamado *ScopedModelDescendant* o, lo que es lo mismo, un descendiente del Modelo. Ubicamos estos objetos por debajo del *ScopedModel* y nos proveen de un *builder* con el que se pueden construir componentes de interfaz en base a los valores de los atributos del Modelo. Cuando cambian estos atributos (el estado), el Modelo notifica a todos los descendientes usando la función *notifyListeners* para que se reconstruyan. El Modelo también permite añadir *listeners* independientes, es decir, funciones que se ejecutan al producirse cambios en los atributos sin necesidad de usar un *ScopedModelDescendant*. Esto es especialmente útil cuando se necesita realizar navegación entre pantallas de la aplicación en base a los cambios en el estado.

Es fácil apreciar como esta librería aplica principalmente el patrón *Observer* para su funcionamiento. Simplificando, tenemos varios objetos (descendientes y *listeners*) que "observan" a otro objeto (modelo) que hace de sujeto. Los observadores se subscriben al sujeto, el cual les notifica de los cambios que sufre para que actúen en consecuencia. Con este patrón se consigue establecer una dependencia de tipo *uno a muchos* con un bajo acoplamiento entre los componentes.

Podría parecer que no hay mucha diferencia entre el uso de *ScopedModel* o utilizar las herramientas que aporta Flutter directamente. Una diferencia importante, a parte de la clara separación que se consigue entre interfaz y modelo con este acercamiento, es que con *ScopedModel* se reconstruyen solo los objetos *ScopedModelDescendant*, mientras que al usar *StatefulWidget*, se reconstruye todo el árbol por debajo del *widget* que ha visto su estado alterado.

En el diagrama de secuencias de la Figura 17, se puede ver un ejemplo de esta gestión del estado con la funcionalidad de ver detalles de un paciente.

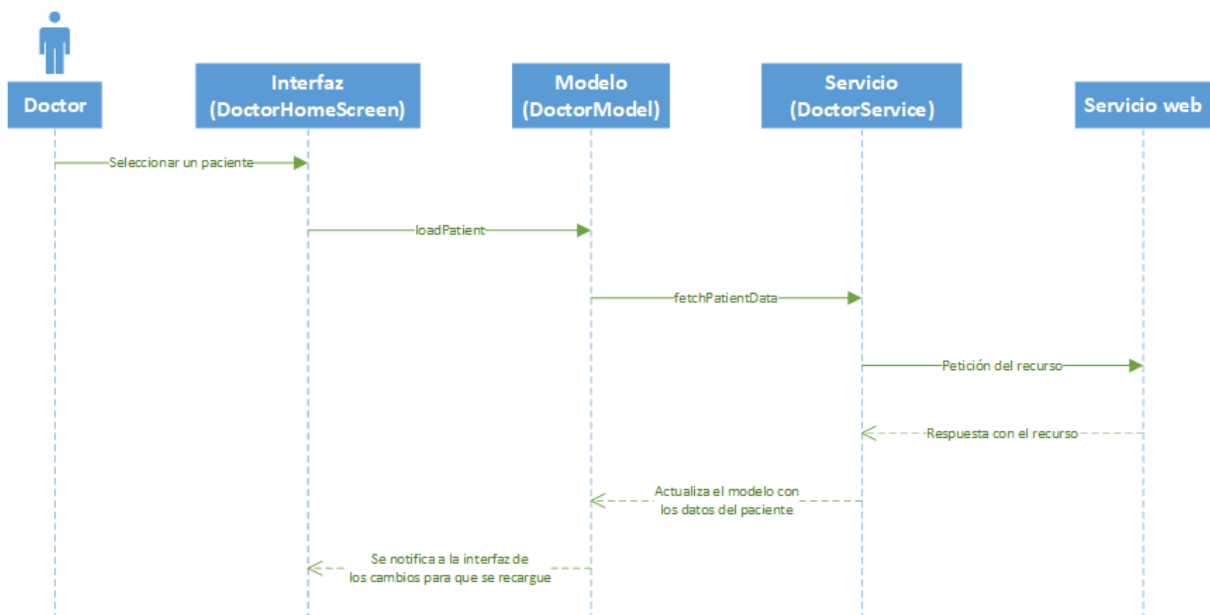


Figura 17: Diagrama de secuencias de la gestión del estado

El usuario de tipo doctor comienza en la pantalla de la lista de pacientes, de los que selecciona uno. La interfaz llama entonces a la función *loadPatient* del objeto de modelo *DoctorModel* que está situado en la raíz del árbol. Esta función comienza poniendo el atributo del modelo *isLoading* a *true* y llamando a *notifyListeners* para que la interfaz se actualice y muestre un símbolo de carga.

A continuación, la función del modelo llama a la función *fetchPatientData* de la interfaz *DoctorService*. Se usa una interfaz para esconder los detalles de implementación de estas funciones y poder cambiar la fuente de estos datos de manera sencilla. En nuestro caso, la implementación de *DoctorService* que se usa, realiza una petición al servicio

web, el cual responde con los datos del paciente requerido. Esos datos se usan para crear un objeto *Patient* que se almacena en el atributo *selectedPatient* del modelo, con el cual se crea en la interfaz la pantalla de detalles del paciente.

Se vuelve a cambiar el valor de *isLoading* a *false* y se llama a *notifyListeners* para que se reconstruya la interfaz, que ahora mostrará los datos detallados del paciente.

4.2.3. Prototipado de la interfaz

Antes de comenzar a construir la interfaz con código, es recomendable realizar bocetos o prototipos. Esto es muy útil para tener una idea del aspecto final que se desea de la interfaz en vez de ir tomando decisiones conforme se crea. Realizar cambios en un boceto es mucho más visual y sencillo que hacerlo en código.

A continuación, se presentan los prototipos de las pantallas de la aplicación que se realizaron al comienzo del proyecto. Estos bocetos fueron hechos a mano con papel y lápiz, y cabe destacar que el aspecto final de dichas pantallas ha variado durante el desarrollo, pero no a grandes rasgos.

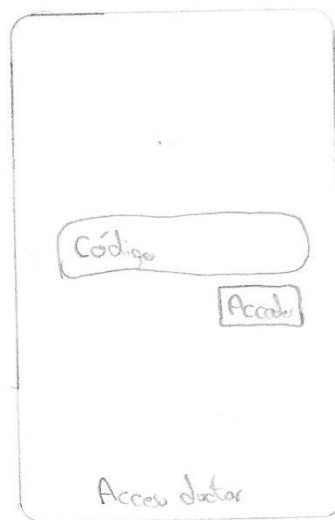


Figura 18: Boceto de pantalla de inicio de sesión del paciente

Las pantallas de inicio de sesión del paciente y del doctor, presentadas en las Figuras 18 y 19 respectivamente, son las primeras que ve el usuario al acceder a la aplicación, pudiendo cambiar el formulario de acceso que se muestra al pulsar en los botones de "Acceso doctor" y "Acceso paciente" ubicados en la parte inferior de las pantallas.

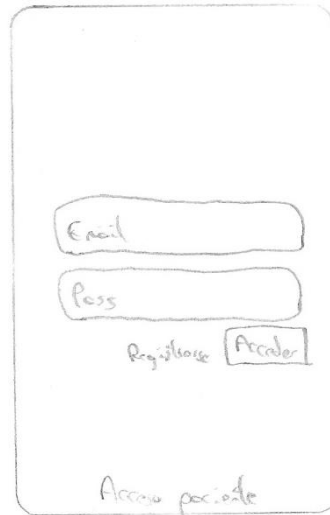


Figura 19: Boceto de pantalla de inicio de sesión del doctor

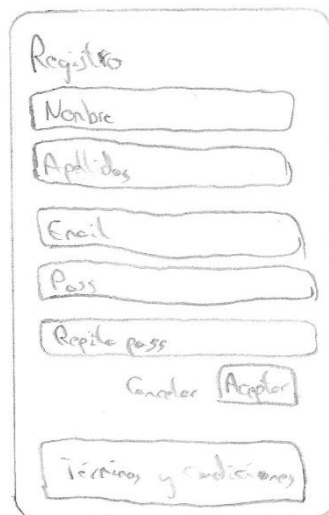


Figura 20: Boceto de pantalla de registro de doctor

A la pantalla de registro de un nuevo doctor, que se puede ver en la Figura 20, se accede desde el inicio de sesión del doctor pulsando en el botón de "Registrarse". El usuario podrá consultar los términos y condiciones antes de realizar el registro pulsando en el botón correspondiente de la parte inferior de la pantalla.

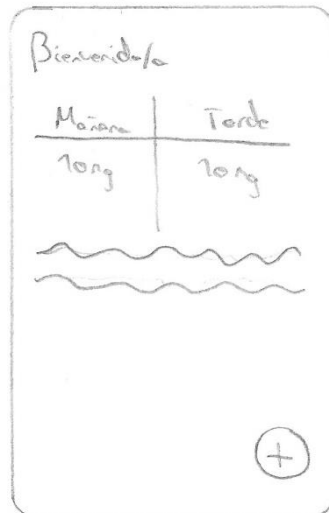


Figura 21: Boceto de pantalla principal del paciente

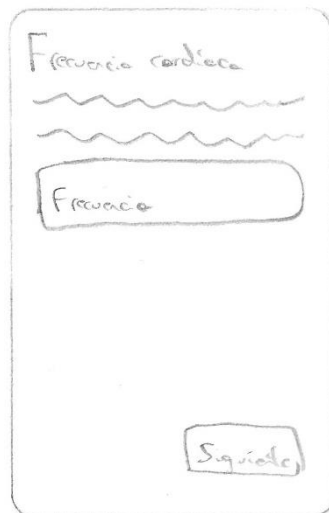


Figura 22: Boceto de pantalla de añadir medición (frecuencia)

Cuando un paciente accede a la aplicación con su código de acceso, se le muestra la pantalla de la Figura 21, donde aparecen los datos de su tratamiento. Concretamente, se muestra en una tabla la dosis de la medicación que debe tomar por la mañana y por la noche. En el texto de debajo de la tabla, se muestran otras indicaciones si puede introducir una medición o si debe acudir a su doctor. Pulsando en el botón con el símbolo "+", puede añadir una nueva medición.

Esta operación consta de una serie de pantallas, donde la primera es la de la Figura 22. Aquí, el paciente introduce el valor de su frecuencia cardíaca. Pulsando en "Siguiente", se traslada a la pantalla de la Figura 23.

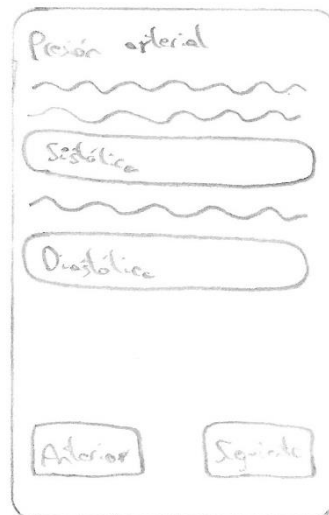


Figura 23: Boceto de pantalla de añadir medición (presión)

En esta pantalla, el paciente introduce su presión arterial tanto sistólica como diastólica. Puede volver a la introducción de la frecuencia cardíaca con "Anterior" o pulsar "Siguiente" para ir a la pantalla de síntomas de la Figura 24. En esta pantalla se muestra una lista de síntomas y el paciente debe indicar si ha sufrido alguno de ellos.

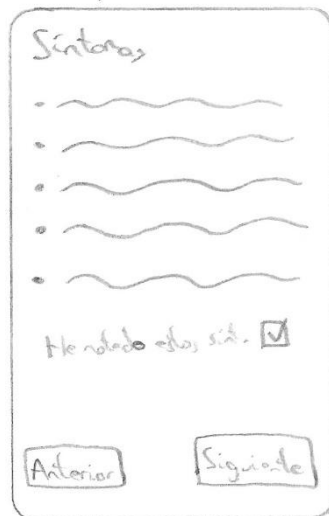


Figura 24: Boceto de pantalla de añadir medición (síntomas)

Cuando se pulsa en "Siguiente" en la pantalla de síntomas, se muestra al paciente una última pantalla con un resumen de los datos que ha introducido para la medición. Esta pantalla se puede ver en la Figura 25. Después de revisarlos, puede pulsar en "Aceptar" para terminar la introducción de la medición.

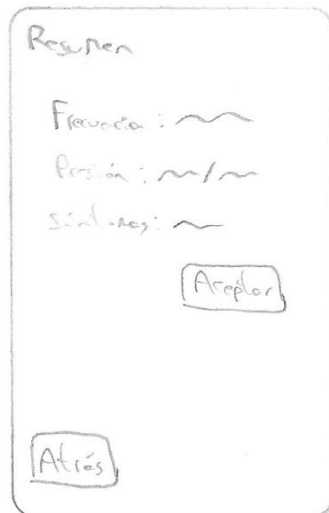


Figura 25: Boceto de pantalla de añadir medición (resumen)



Figura 26: Boceto de pantalla principal del doctor

En la Figura 26 vemos la pantalla principal del doctor, a la cual accede al autenticarse. En esta pantalla se muestra la lista de pacientes que gestiona el doctor, además de un botón en la barra superior para cerrar la sesión y otro en la esquina inferior derecha para añadir un nuevo paciente.

Esta creación de nuevos pacientes se realiza en la pantalla de la Figura 27. Como se puede ver, es un formulario con 3 partes. Primero, la introducción del número de historia y la selección del medicamento que tomará en su tratamiento el paciente. Las dos siguientes partes son para que el doctor defina las dosis inicial y máxima, respectivamente.



Figura 27: Boceto de pantalla de añadir paciente

Una vez que ha rellenado todos los datos del nuevo paciente, el doctor pulsa "Añadir" para terminar su creación, lo que le lleva a la pantalla de la Figura 28, donde se muestra el código de acceso generado para el paciente con la intención de que se lo comunique a este. Desde esta pantalla el doctor podrá iniciar una simulación del proceso de añadir una medición para instruir al paciente en su uso pulsando en "Simulación". También puede volver a la pantalla principal pulsando en "Menú".

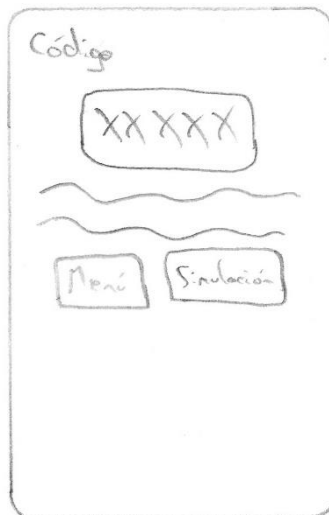


Figura 28: Boceto de pantalla de código del nuevo paciente

Volviendo a la pantalla principal del doctor de la Figura 26, al pulsar sobre uno de los pacientes de la lista, el doctor accede a una pantalla con detalles del paciente seleccionado y su tratamiento.

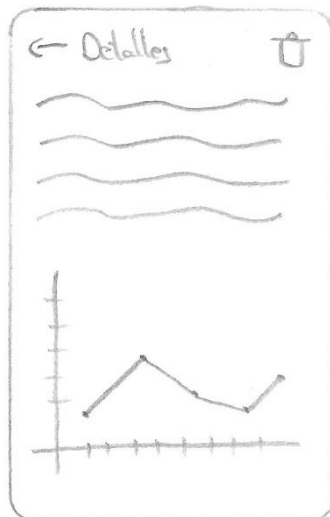


Figura 29: Boceto de pantalla de detalles del paciente

En esta pantalla, que se puede apreciar en la Figura 29, se muestran los datos principales sobre el tratamiento, como son la medicación, fecha de creación o el estado actual de este. Justo debajo, el doctor tendrá acceso a una gráfica de las constantes vitales introducidas por el paciente en las mediciones, con lo que podrá observar su progresión.

Por último, pulsando en el icono de la papelera ubicado en la esquina superior derecha, el doctor podrá borrar los datos del paciente.

4.3. Implementación

Una vez definidos los requisitos y funcionalidades a implementar, la arquitectura a seguir y la interfaz deseada, se procede a la implementación. En los siguientes apartados se van a presentar las partes más relevantes del código y de la interfaz para cada una de las funcionalidades implementadas.

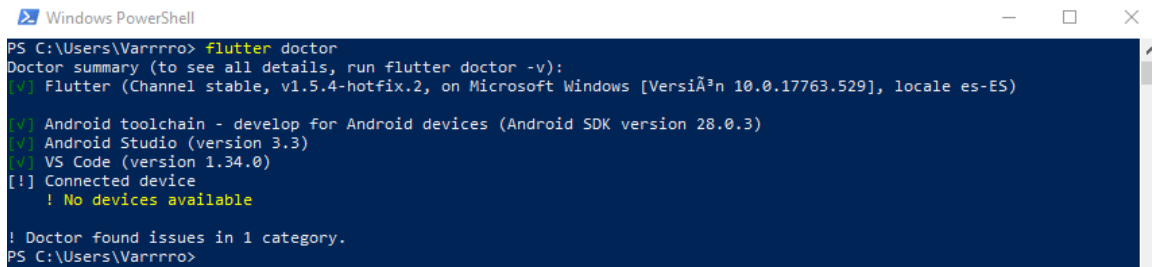
4.3.1. Configuración del entorno de trabajo

Antes de comenzar a comentar la implementación de las distintas funcionalidades, se van a explicar los pasos necesarios para configurar nuestro entorno de trabajo para crear aplicaciones con Flutter.

Lo primero de todo es instalar el propio SDK de Flutter. Para ello, seguimos los pasos indicados en la página oficial para nuestro sistema operativo. En el caso de Windows, solo tenemos que descargar el archivo con el SDK, extraerlo en alguna carpeta accesible por el sistema y añadir la carpeta *bin* a la variable de entorno PATH de nuestro sistema. Con esto, ya podremos usar la herramienta *flutter* desde la línea de comandos.

Desarrollo de aplicaciones móviles multiplataforma con Flutter

El comando `flutter doctor` nos permite conocer el estado de nuestra configuración para desarrollar aplicaciones con Flutter, además de indicarnos lo que nos falte en la misma, tal y como se puede ver en la Figura 30.



```
Windows PowerShell
PS C:\Users\Varrro> flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.5.4-hotfix.2, on Microsoft Windows [Version 10.0.17763.529], locale es-ES)

[✓] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
[✓] Android Studio (version 3.3)
[✓] VS Code (version 1.34.0)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.
PS C:\Users\Varrro>
```

Figura 30: Salida del comando `flutter doctor`

Una vez instalado el SDK de Flutter, teóricamente ya podríamos comenzar a desarrollar la aplicación. Sin embargo, es necesario instalar las herramientas de Android para poder compilar nuestra aplicación a esta plataforma y probar nuestro código con el simulador.

La manera más sencilla de instalar el SDK de Android y las herramientas relacionadas es instalando Android Studio. Este IDE, del que ya hemos hablado en el apartado 2.3, incorpora el SDK, sus herramientas de plataforma y de compilación. A muchas de estas herramientas podemos acceder desde su pantalla de inicio, mostrada en la Figura 31, pulsando en el botón “Configure” de la esquina inferior derecha y sin necesidad de abrir un proyecto.

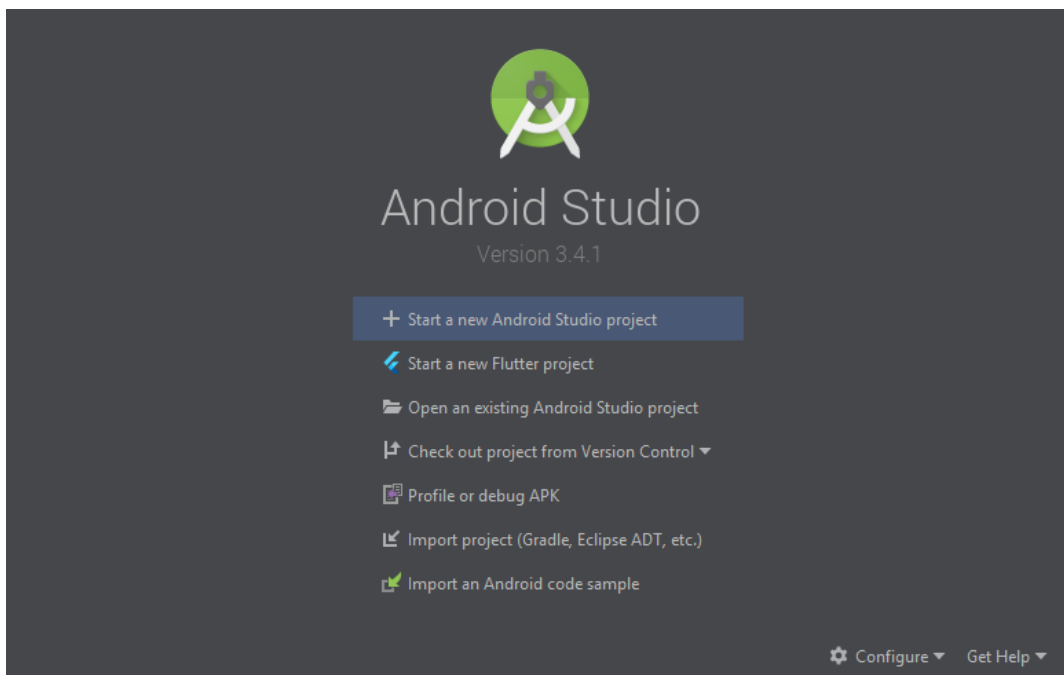


Figura 31: Pantalla de inicio de Android Studio

Una de las herramientas más importantes es el simulador, con el cual podemos ejecutar una simulación de un teléfono Android corriendo en nuestro ordenador, y que usamos para probar el código desarrollado. Podemos crear simuladores para cualquier versión del SDK de Android que queramos desde Android Studio que luego lanzaremos desde el editor de texto.

Este editor va a ser Visual Studio Code, del cual se ha hablado en el apartado 2.4. No contiene de serie la misma cantidad de herramientas y ayudas que tendría un IDE como es Android Studio. Sin embargo, lo preferimos por su versatilidad y rapidez. Instalando en Visual Studio Code las extensiones de Dart y Flutter obtenemos soporte completo para el lenguaje (formateo, sugerencias, detección de errores) y acceso a herramientas de depuración de aplicaciones de Flutter desde el editor. Esto nos permite lanzar los simuladores de Android y refrescar la aplicación en ejecución para que refleje los cambios realizados en el editor.

Todo el código de nuestro proyecto se va a alojar en GitHub, por lo que necesitaríamos crear una cuenta y un nuevo repositorio. Visual Studio Code incluye de serie soporte para Git, con lo que estaríamos listos para comenzar el desarrollo.

4.3.2. Autenticación y registro

La primera funcionalidad de la que vamos a hablar es el inicio de sesión y registro. Ya hemos indicado anteriormente que hay dos tipos de usuarios (doctores y pacientes) en nuestra aplicación, teniendo cada uno una manera de autenticarse diferente. Un doctor debe introducir su email y contraseña, mientras que un paciente introduce el código que le ha dado su doctor. Por tanto, necesitamos dos formularios de inicio de sesión.

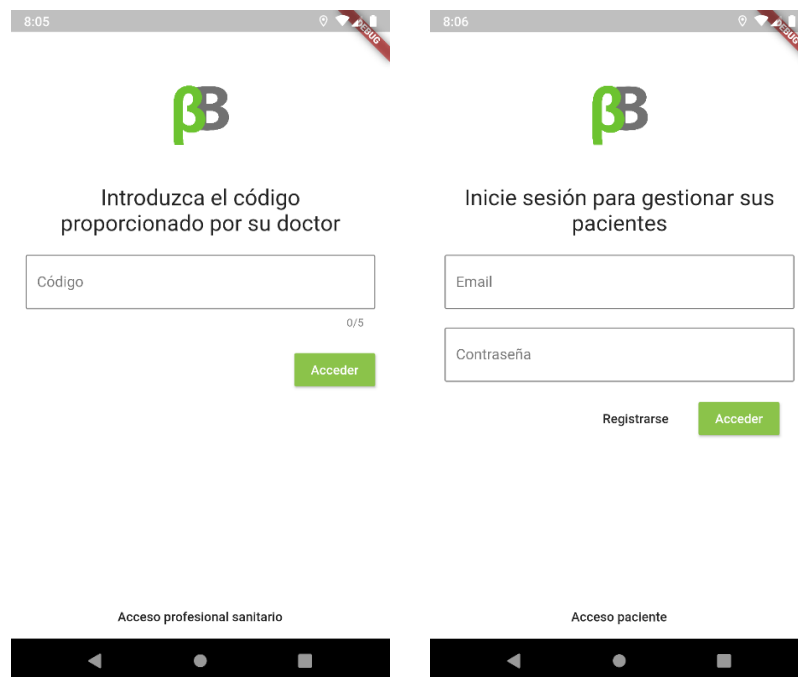


Figura 32: Pantallas de inicio de sesión de paciente y doctor

En la Figura 32 podemos apreciar el aspecto de estas pantallas, aunque técnicamente es una sola. Para tener un diseño uniforme en ambas, se han implementado como una única pantalla en un *StatefulWidget* el cual, al pulsar en el botón inferior de “Acceso profesional sanitario” o “Acceso paciente”, cambia su estado y se reconstruye con el formulario correspondiente.

```
17  @override
18  void initState() {
19    super.initState();
20    _isDoctor = false;
21  }
22
23  @override
24  Widget build(BuildContext context) {
25    return ScopedModelDescendant<AuthModel>(
26      builder: (context, _, model) {
27        return Scaffold(
28          body: SafeArea(
29            child: Column(
30              children: <Widget>[
31                Expanded(
32                  child: ListView(
33                    physics: ClampingScrollPhysics(),
34                    padding: const EdgeInsets.all(16.0),
35                    children: <Widget>[
36                      const SizedBox(height: 40.0),
37                      Image.asset(
38                        'assets/app-logo.png',
39                        width: 70.0,
40                        height: 70.0,
41                      ), // Image.asset
42                      const SizedBox(height: 40.0),
43                      if (_isDoctor) DoctorLogin() else PatientLogin(),
44                    ], // <Widget>[]
45                  ), // ListView
46                ), // Expanded
47                FlatButton(
48                  child: Text(_isDoctor
49                    ? 'Acceso paciente'
50                    : 'Acceso profesional sanitario'), // Text
51                  onPressed: model.isLoading
52                    ? null
53                    : () {
54                      model.clearError();
55                      setState(() => _isDoctor = !_isDoctor);
56                    },
57                ), // FlatButton
58              ], // <Widget>[]
59            ), // Column
60          ), // SafeArea
61        ); // Scaffold
62      },
63    ); // ScopedModelDescendant
64  }
```

Figura 33: Construcción de los formularios dependiendo del estado

En el código de la Figura 33, se puede apreciar cómo está implementada la lógica de este comportamiento. La ventana tiene un atributo *_isDoctor* (la barra baja denota que es privado) cuyo estado inicial se fija a *false* en la función *initState* de *StatefulWidget*. En base al valor de este atributo, se crea el formulario de inicio de sesión del doctor

(*DoctorLogin*) o el del paciente (*PatientLogin*), como se puede ver en la línea 43. Es el botón *FlatButton* ubicado al final de la ventana el encargado de cambiar el valor de este atributo haciendo una llamada a *setState* con los cambios. *setState* es otra función de *StatefulWidget* que se utiliza para cambiar atributos del estado y notificar al árbol de *widgets* que debe reconstruirse para tener en cuenta los nuevos valores.

```

48   String _validateCode(String code) {
49     if (code.isEmpty) {
50       return 'Introduce un valor';
51     } else if (code.length < 5) {
52       return 'El código debe tener 5 caracteres';
53     } else {
54       return null;
55     }
56   }
57
58   void _onLogin() {
59     if (_formKey.currentState.validate()) {
60       AuthModel.of(context).loginPatient(_codeController.text);
61     } else {
62       setState(() => _autovalidate = true);
63     }
64   }

```

Figura 34: Validación y envío de datos de acceso del paciente

La implementación de los dos formularios de acceso es análoga, por lo que vamos a fijarnos en *PatientLogin* para explicar su funcionamiento. El formulario se construye con un *widget* de tipo *Form*, que nos permite aportar funciones de validación a los campos de texto. En el caso del paciente, la única validación que se realiza sobre el código que introduce es que este sea de 5 caracteres. Esto lo realiza la función *_validateCode* de la Figura 34, mientras que el botón de "Acceder", llama a la función *_onLogin*. En esta función, primero se validan los campos del formulario con *validate*, función que llama al *validator* asociado a cada campo de texto del *Form* y devuelve *true* si todos son correctos.

En caso de que el código sea correcto, accedemos al *AuthModel* ubicado en el árbol y llamamos a su función *loginPatient* con el código introducido por el usuario. El acceso a *AuthModel* se realiza a través del contexto de construcción del *widget*, el cual guarda una referencia al padre de manera recursiva para cada *widget* del árbol. La función *loginPatient* se encarga de llamar al servicio para que ejecute la lógica de autenticación del paciente y realizar las modificaciones pertinentes al modelo.

Como podemos ver en la Figura 35, comienza fijando el valor de *_isLoading* a *true* y notifica a los descendientes para que se reconstruyan con *notifyListeners*. Este atributo es usado por la interfaz para saber si se debe mostrar un indicador de carga. Si la autenticación termina correctamente, la función del servicio devuelve el identificador del paciente, el cual se almacena en un mapa de claves-valor para que sea accesible desde otras partes de la aplicación. También se cambia la variable de modelo *_loggedUser* para que refleje que hay un usuario de tipo paciente autenticado. Por

último, se vuelve a poner *false* el indicador de carga y se notifica la reconstrucción de la interfaz.

```
101 void loginPatient(String patientCode) async {
102   _isLoading = true;
103   _error = '';
104   notifyListeners();
105
106   try {
107     final patientID = await authService.loginPatient(patientCode);
108
109     final prefs = await SharedPreferences.getInstance();
110
111     prefs.setInt('patient_id', patientID);
112
113     _loggedUser = LoggedUser.patient;
114   } on TimeoutException catch (_) {
115     _error = 'Tiempo de espera expirado';
116   } on NotFoundException catch (_) {
117     _error = 'El código no es válido';
118   } on BadRequestException catch (_) {
119     _error = 'Paciente deshabilitado';
120   } on ServerErrorException catch (_) {
121     _error = 'Error del servidor';
122   } finally {
123     _isLoading = false;
124     notifyListeners();
125   }
126 }
```

Figura 35: Función *loginPatient* de *AuthModel*

Si ocurre algún error durante la ejecución de la función del servicio, este lanza una excepción, la cual es capturada para reflejar el mensaje de error correcto en la interfaz a través de la variable *_error*.

```
103 Future<int> loginPatient(String patientCode) async {
104   final response = await http.get(
105     '${_manager.baseURL}/patients?code=${patientCode}',
106     headers: {
107       HttpHeaders.contentTypeHeader: 'application/json',
108       HttpHeaders.authorizationHeader: 'Basic ${_manager.basicAuth}',
109     },
110   ).timeout(
111     const Duration(seconds: 5),
112   );
113
114   switch (response.statusCode) {
115     case HttpStatus.ok:
116       return _manager.decodeBody(response)['id'];
117     case HttpStatus.notFound:
118       throw NotFoundException();
119     case HttpStatus.badRequest:
120       throw BadRequestException();
121     case HttpStatus.internalServerError:
122       throw ServerErrorException();
123     default:
124       throw UndefinedException();
125   }
126 }
127 }
```

Figura 36: Función *loginPatient* de *HttpAuth*

Nuestra implementación del servicio *AuthService* se comunica mediante HTTP con un servicio web remoto para llevar a cabo las tareas de autenticación y registro. La clase *HttpAuth* contiene esta implementación que, en el caso de *loginPatient* supone realizar una petición a la ruta del servidor encargada de gestionar los códigos de los pacientes. Se interpreta después la respuesta del servidor y, en base al código de estado de esta, se devuelve el identificador del paciente o se lanza una excepción. El código de esta función se puede ver en la Figura 36.

Como ya hemos indicado, el proceso para el inicio de sesión del doctor con *DoctorLogin* es análogo al del paciente, si bien la respuesta de la función del servicio no es solo el identificador del usuario, sino un *token* de acceso necesario para realizar las siguientes llamadas al servicio web.

Para el registro de una nueva cuenta de doctor, pulsamos en el botón "Registrarse" de la pantalla de inicio de sesión del doctor para acceder a la pantalla de registro que se puede ver en la Figura 37. De la misma manera que ya hemos visto, al rellenar el formulario y pulsar en "Registrarse", si los datos introducidos pasan la validación, se llama a la función *registerDoctor* de *AuthModel*, la cual llama al servicio para que se comunique con el servicio web y realice el registro de la nueva cuenta.

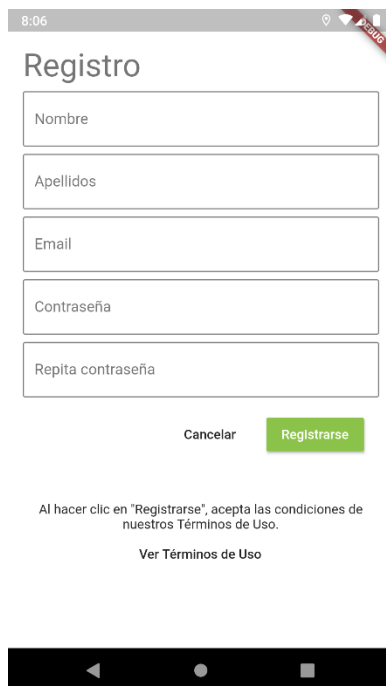


Figura 37: Pantalla de registro de doctor

Cabe destacar que, tanto en el inicio de sesión del paciente como del doctor, si se realiza correctamente, se debe navegar directamente a las pantallas correspondientes para cada usuario. Esto lo conseguimos escuchando a cambios en el modelo desde un

listener del modelo que registramos en la pantalla de acceso, tal y como se ve en la Figura 38.

```
27 @override
28 void didChangeDependencies() {
29   _authModel = AuthModel.of(context);
30   _authModel.addListener(_navigationListener);
31   super.didChangeDependencies();
32 }
33
34 @override
35 void dispose() {
36   _authModel.removeListener(_navigationListener);
37   super.dispose();
38 }
39
40 void _navigationListener() {
41   if (AuthModel.of(context).loggedUser == LoggedUser.patient) {
42     Navigator.of(context).pushReplacementNamed(Routes.patientTerms);
43   } else if (AuthModel.of(context).loggedUser == LoggedUser.doctor) {
44     Navigator.of(context).pushReplacementNamed(Routes.doctorHome);
45   }
46 }
```

Figura 38: Implementación del listener de navegación

De esta forma, cuando *AuthModel* notifique de cambios, la función *_navigationListener* registrada se ejecutará, comprobando si el valor de *_loggedUser* ha cambiado y realizando la navegación correspondiente. Esta manera de realizar la navegación se aplica a lo largo de toda la aplicación, aunque usando indicadores booleanos en vez de un tipo enumerado como en este caso.

4.3.3. Lista de pacientes

Cuando un usuario se identifica como doctor, es redirigido a su pantalla principal, en la que aparece la lista de pacientes que gestiona. Como se puede apreciar en la Figura 39, esta pantalla también tiene botones que le permiten cerrar la sesión o añadir un nuevo paciente.

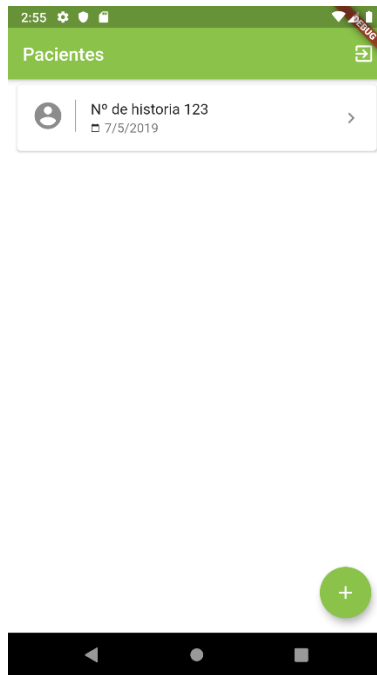


Figura 39: Pantalla principal del doctor

Para crear la lista de pacientes desplazable, usamos un componente *ListView*, el cual permite crear listas de componentes con desplazamiento horizontal o vertical. En nuestro caso, usaremos el constructor *ListView.builder*, el cual recibe el número de elementos de la lista y una función constructora de dichos elementos.

```

63     body: ScopedModelDescendant<DoctorModel>(
64       builder: (context, _, model) {
65         return ListView.builder(
66           itemCount: model.patientCount + 1,
67           itemBuilder: (context, index) {
68             final pageIndex =
69               _pageStartFromIndex(index, model.patientsPerPage);
70
71             if (model.pages.containsKey(pageIndex)) {
72               final patient = model.pages[pageIndex].elementAt(index);
73
74               if (patient == null) {
75                 return Container();
76               }
77
78               return PatientEntry(patient: patient);
79             } else if (model.error.isNotEmpty) {
80               return Center(
81                 child: Padding(
82                   padding: const EdgeInsets.all(20.0),
83                   child: Text(
84                     model.error,
85                     style: TextStyle(color: Colors.red),
86                     textAlign: TextAlign.center,
87                   ), // Text
88                 ), // Padding
89               ); // Center
90             } else {
91               model.loadPatientsPage(pageIndex);
92
93               return Center(
94                 child: Padding(
95                   padding: const EdgeInsets.all(20.0),
96                   child: CircularProgressIndicator(),
97                 ), // Padding
98               ); // Center
99             }
100           },
101         ); // ListView.builder
102       },
103     ), // ScopedModelDescendant

```

Figura 40: Constructor de la lista de pacientes

Los pacientes con los que se construye la lista están almacenados en el modelo *DoctorModel* en páginas. Estas páginas son solo colecciones de elementos de un determinado tamaño identificadas por el índice de inicio. De esta manera, la aplicación realiza peticiones al servicio web para páginas concretas, las cuales mantiene en el modelo. Esto es mucho más eficiente que recuperar los datos de todos los pacientes directamente, ya que la cantidad de información que se pide y recibe es solo la que se está usando en cada momento.

Las páginas se almacenan en *DoctorModel* en un mapa *_pages*, donde cada página es identificada por el índice de su primer elemento. La función constructora de elementos para el *ListView*, que se puede ver en el Figura 40, se llama para cada índice de la lista desde 0 tantas veces como se haya indicado en *itemCount*. En cada llamada, se obtiene

el índice inicial de la página a la que pertenece usando la cantidad de elementos de cada página y se comprueba si dicha página está en el mapa del modelo.

En caso afirmativo, se recupera la información del paciente concreto y se construye el objeto *PatientEntry* que la muestra en la lista. Si la página deseada no está en el modelo, se inicia el proceso para obtenerla llamando a la función *loadPatientsPage* mostrada en la Figura 41 y se muestra un indicador de carga.

```
71 void loadPatientsPage(int startIndex) async {
72   if (_requestedPages.contains(startIndex)) {
73     return;
74   }
75
76   _requestedPages.add(startIndex);
77   _error = '';
78
79   try {
80     final prefs = await SharedPreferences.getInstance();
81     final doctorID = prefs.getInt('doctor_id');
82
83     final newPage = await doctorService.fetchPatientsPage(
84       _patientsPerPage, startIndex, doctorID);
85
86     _requestedPages.remove(startIndex);
87     _pages[startIndex] = newPage;
88   } on TimeoutException catch (_) {
89     _error = 'Tiempo de espera expirado';
90   } on UnauthorizedException catch (_) {
91     _error = 'Acceso restringido';
92   } on ServerErrorException catch (_) {
93     _error = 'Error del servidor';
94   } finally {
95     notifyListeners();
96   }
97 }
98 }
```

Figura 41: Función *loadPatientsPage* de *DoctorModel*

En el modelo se mantiene además una colección de las páginas que están en proceso de ser obtenidas para evitar que se realicen múltiples peticiones de una misma página. Se realiza la llamada a la función *fetchPatientsPage* del servicio encargada de hacer la petición y devolver la página si la hubiera. La implementación de esta función se puede ver en la Figura 42.

```

20 Future<PatientsPage> fetchPatientsPage(
21     int count, int offset, int doctorID) async {
22     final prefs = await SharedPreferences.getInstance();
23     final accessToken = prefs.getString('access_token');
24
25     final response = await http.get(
26         '${_manager.baseUrl}/doctors/${doctorID}/patients?count=$count&offset=$offset',
27         headers: {
28             HttpHeaders.contentTypeHeader: 'application/json',
29             HttpHeaders.authorizationHeader: 'Bearer $accessToken',
30         },
31     ).timeout(
32         const Duration(seconds: 5),
33     );
34
35     switch (response.statusCode) {
36         case HttpStatus.ok:
37             final patients = (_manager.decodeBody(response) as List)
38                 .map((value) => Patient.fromJson(value))
39                 .toList();
40
41             return PatientsPage(patients, offset);
42         case HttpStatus.unauthorized:
43             throw UnauthorizedException();
44         case HttpStatus.internalServerError:
45             throw ServerErrorException();
46         default:
47             throw UndefinedException();
48     }
49 }

```

Figura 42: Función `fetchPatientsPage` de `HttpDoctor`

Si la respuesta del servidor es correcta, contiene en su cuerpo una lista con los datos de los pacientes de la página en formato JSON, los cuales se decodifican y se construye la página que se devuelve al modelo. Una vez que el modelo recibe la nueva página, notifica a los descendientes para que se reconstruyan, mostrando así la lista los nuevos elementos.

4.3.4. Detalles de un paciente

Al seleccionar un paciente de la lista, se navega a una pantalla con datos detallados sobre el tratamiento del paciente seleccionado como son el medicamento, la fecha de inicio o el código de acceso. También se muestran las mediciones introducidas por el paciente en un gráfico.

En concreto, se ha creado un gráfico de líneas usando la librería `charts_flutter`. Este gráfico permite además seleccionar un punto (que representa una medición) y obtener sus datos exactos, los cuales mostramos debajo, tal y como se ve en la Figura 43.



Figura 43: Pantalla de detalles del paciente

Cuando el usuario pulsa sobre un paciente en la lista, se llama a la función *loadPatient* de *DoctorModel* con el identificador del paciente seleccionado como parámetro antes de realizar la navegación. Como ya hemos visto en otros casos, la función hace que se muestre un indicador de carga en la interfaz a través de la variable *_isLoading*. En este caso y como se puede apreciar en la Figura 44, se deben realizar dos llamadas diferentes al servicio *DoctorService*: una para los datos del paciente con *fetchPatientData* y otra para los datos de sus tramos de dosis con *fetchPatientStretches*. Esto se debe a que, siguiente el modelo de una API REST para el servicio web, cada recurso está alojado en una ruta diferente y, por tanto, deben realizarse dos peticiones. Todos los datos recibidos se almacenan en la variable de modelo *_selectedPatient*, a partir de la cual se construye la ventana.

```
100 void loadPatient(int patientID) async {
101   _isLoading = true;
102   _error = '';
103   notifyListeners();
104
105   try {
106     final prefs = await SharedPreferences.getInstance();
107
108     final doctorID = prefs.getInt('doctor_id');
109
110     _selectedPatient =
111       await doctorService.fetchPatientData(patientID, doctorID);
112     _selectedPatient.stretches =
113       await doctorService.fetchPatientStretches(patientID, doctorID);
114   } on TimeoutException catch (_) {
115     _error = 'Tiempo de espera expirado';
116   } on UnauthorizedException catch (_) {
117     _error = 'Acceso restringido';
118   } on NotFoundException catch (_) {
119     _error = 'Recurso no encontrado';
120   } on ServerErrorException catch (_) {
121     _error = 'Error del servidor';
122   } finally {
123     _isLoading = false;
124     notifyListeners();
125   }
126 }
```

Figura 44: Función loadPatient de DoctorModel

En esta pantalla de detalles del paciente, también se puede realizar su eliminación al pulsar sobre el icono de la papelera de la barra superior. Al realizar esto, se muestra el diálogo de confirmación de la Figura 45 que, en caso de ser aceptado, llama a la función *deletePatient* del modelo, la cual se encarga de gestionar la eliminación.

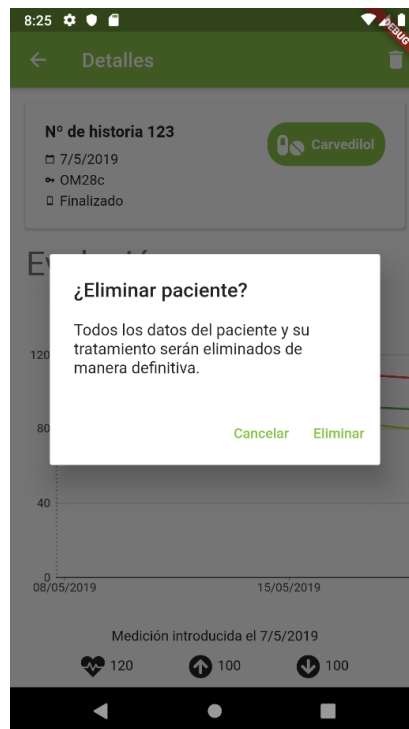


Figura 45: Diálogo de confirmación al eliminar un paciente

Tal y como se ve en la Figura 46, si la eliminación es correcta, se borran todas las páginas de pacientes que había en la variable `_pages` para evitar inconsistencias y se navega de vuelta a la lista de pacientes, que se deberá de cargar de nuevo.

```

158 void deletePatient() async {
159   _isLoading = true;
160   _deletionError = '';
161   notifyListeners();
162
163   try {
164     final prefs = await SharedPreferences.getInstance();
165
166     final doctorID = prefs.getInt('doctor_id');
167
168     await doctorService.deletePatient(_selectedPatient.id, doctorID);
169
170     _pages.clear();
171     _navigateToHome = true;
172   } on TimeoutException catch (_) {
173     _deletionError = 'Tiempo de espera expirado';
174   } on UnauthorizedException catch (_) {
175     _deletionError = 'Acceso restringido';
176   } on NotFoundException catch (_) {
177     _deletionError = 'Recurso no encontrado';
178   } on ServerErrorException catch (_) {
179     _deletionError = 'Error del servidor';
180   } finally {
181     _isLoading = false;
182     notifyListeners();
183   }
184 }
185 }

```

Figura 46: Función `deletePatient` de `DoctorModel`

4.3.5. Añadir un paciente

Pulsando en el botón con el símbolo "+" en la lista de pacientes, el usuario navega a la pantalla de creación de un nuevo paciente. Aquí, lo primero que hará será introducir un número de historia que identifique al paciente y un medicamento para su tratamiento.

Dependiendo del medicamento que se seleccione, se presentan dos componentes distintos debajo. En el caso del tratamiento con carvedilol, tanto la dosis inicial como la final están fijadas y no se pueden cambiar, mientras que con propranolol se puede elegir la dosis inicial que se quiera y se da a elegir entre dos opciones para la dosis final. Este comportamiento lleva a que la pantalla de añadir paciente pueda presentar varios aspectos, tal y como se ve en la Figura 47.

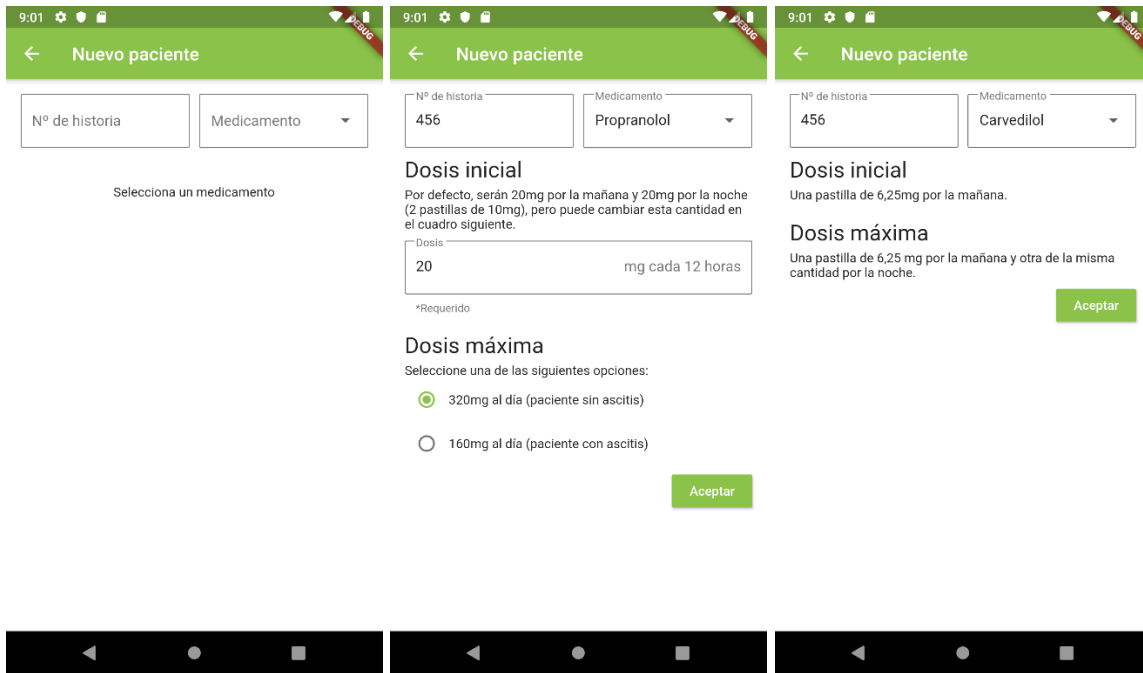


Figura 47: Pantalla de creación de paciente

Independientemente del medicamento elegido, al pulsar el botón “Aceptar” se llama a la función `addPatient` de `DoctorModel` con los datos del nuevo paciente y su dosis inicial. Esta función llama al servicio para que realice la creación del nuevo paciente y genere su código de acceso aleatorio.

```
128 void addPatient(Patient patient, Dose initialDose) async {
129   _isLoading = true;
130   _error = '';
131   notifyListeners();
132
133   try {
134     final prefs = await SharedPreferences.getInstance();
135
136     final doctorID = prefs.getInt('doctor_id');
137
138     final newPatient =
139       await doctorService.addPatient(patient, initialDose, doctorID);
140
141     _newPatientCode = newPatient.code;
142     _pages.clear();
143     _navigateToCode = true;
144   } on TimeoutException catch (_) {
145     _error = 'Tiempo de espera expirado';
146   } on UnauthorizedException catch (_) {
147     _error = 'Acceso restringido';
148   } on ConflictException catch (_) {
149     _error = 'El nº de historia ya está en uso';
150   } on ServerErrorException catch (_) {
151     _error = 'Error del servidor';
152   } finally {
153     _isLoading = false;
154     notifyListeners();
155   }
156 }
```

Figura 48: Función `addPatient` de `DoctorModel`

Como se puede ver en la Figura 48, si el paciente ha sido añadido correctamente, el código devuelto por el servicio se guarda en la variable `_newPatientCode` y, al igual que se hacía al eliminar un paciente, se borran todas las páginas almacenadas en `_pages` para evitar inconsistencias. Por último, se indica a la interfaz con `_navigateToCode` que debe navegar a la pantalla mostrada en la Figura 49, que muestra este código al doctor.



Figura 49: Pantalla de código del nuevo paciente

En esta pantalla, además, el doctor podrá lanzar una simulación del proceso para añadir una nueva medición, con el objetivo de enseñar al paciente como se realiza. Pulsando el botón "Ir a lista", volverá a la lista de pacientes.

4.3.6. Estado del tratamiento

En cuanto al paciente, una vez que accede con el código que le ha proporcionado su doctor, se encuentra con la pantalla de términos y condiciones de la Figura 50. Esta pantalla se muestra en cada acceso, y el usuario debe indicar que acepta estos términos para poder acceder a los datos de su tratamiento e introducir mediciones. Si no los acepta, es devuelto a la pantalla de inicio de sesión.

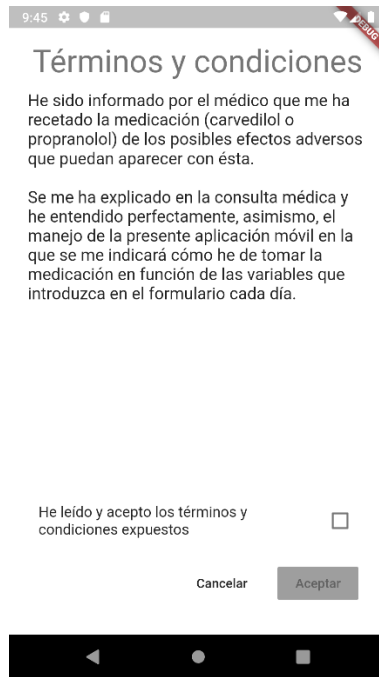


Figura 50: Pantalla de términos y condiciones del paciente

Una vez aceptados estos términos, accede a su pantalla principal, donde se muestra la información sobre el estado de su tratamiento. Concretamente, aparece un *widget* personalizado que hemos llamado *DoseDisplay*, el cual indica al usuario la dosis que debe tomar tanto de día como de noche, tal y como se puede apreciar en la Figura 51.

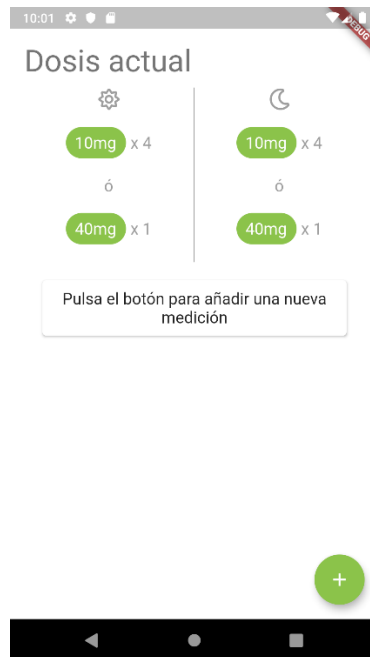


Figura 51: Pantalla principal del paciente

Los datos que se muestran en esta pantalla se obtienen del modelo *PatientModel*. Aquí, la variable *_patient* contiene los datos del paciente y *_currentStretch*, los del tramo de dosis actual.

Estos datos se obtienen al llamar a la función *loadPatientData* del modelo, la cual realiza dos llamadas al servicio para obtener los datos del paciente (*fetchPatient*) y su tramo actual (*fetchCurrentStretch*). Se puede ver la implementación de *loadPatientData* en la Figura 52.

```

63 void loadPatientData() async {
64   _isLoading = true;
65   _error = '';
66   notifyListeners();
67
68   try {
69     final prefs = await SharedPreferences.getInstance();
70
71     final patientID = prefs.getInt('patient_id');
72
73     _patient = await patientService.fetchPatient(patientID);
74     _currentStretch = await patientService.fetchCurrentStretch(patientID);
75   } on TimeoutException catch (_) {
76     _error = 'Tiempo de espera expirado';
77   } on NotFoundException catch (_) {
78     _error = 'No se ha encontrado el usuario';
79   } on ServerErrorException catch (_) {
80     _error = 'Error del servidor';
81   } finally {
82     _isLoading = false;
83     notifyListeners();
84   }
85 }

```

Figura 52: Función *loadPatientData* de *PatientModel*

Cabe destacar que, en esta pantalla principal del paciente, se muestra una indicación para acudir al médico si el paciente ha finalizado su tratamiento, además de inhabilitar el botón de añadir una nueva medición si ya ha añadido una en las últimas 24 horas.

4.3.7. Añadir una medición

Por último, hablaremos de la funcionalidad de añadir medición. Al pulsar el botón con el símbolo "+" en la pantalla principal del paciente, se accede a una pantalla que posee varias "fases": introducción de frecuencia cardíaca, presión arterial, síntomas sufridos y un resumen de los datos introducidos. Podemos ver todas estas fases en la Figura 53.

Se puede navegar libremente entre todas estas partes usando los botones de la parte inferior de la pantalla. Los datos introducidos se van guardando en la variable del modelo *_measurement* haciendo uso de las funciones *setHeartRate*, *setBloodPressure* y *setSymptoms*.

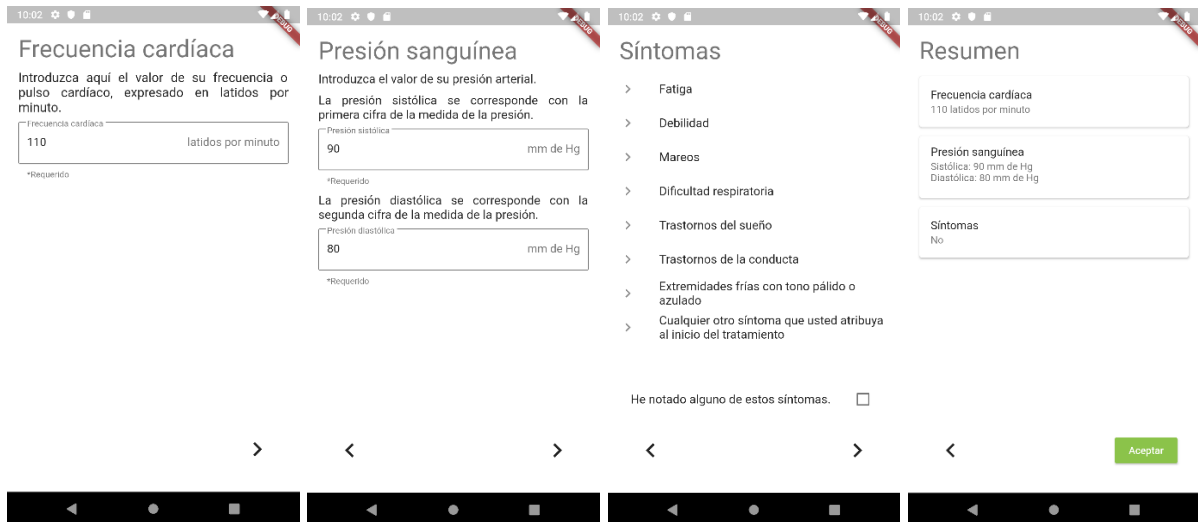


Figura 53: Pantallas de añadir medición

Estas pantallas son las mismas que se utilizan para la simulación que puede realizar el doctor tras añadir un paciente nuevo, mostrándose en ese caso un indicativo en la parte superior de que se trata de una simulación.

```
87 void addMeasurement() async {
88   _isLoading = true;
89   _error = '';
90   notifyListeners();
91
92   try {
93     final prefs = await SharedPreferences.getInstance();
94
95     final patientID = prefs.getInt('patient_id');
96
97     await patientService.addMeasurement(
98       _measurement, patientID, _currentStretch.id);
99
100    _measurement = Measurement.empty();
101    _navigateToHome = true;
102  } on TimeoutException catch (_) {
103    _error = 'Tiempo de espera expirado';
104  } on NotFoundException catch (_) {
105    _error = 'No se ha encontrado el usuario';
106  } on ServerErrorException catch (_) {
107    _error = 'Error del servidor';
108  } finally {
109    _isLoading = false;
110    notifyListeners();
111  }
112 }
```

Figura 54: Función addMeasurement de PatientModel

Al pulsar el botón "Aceptar" de la última pantalla, la de resumen, se vuelve a la pantalla del código del nuevo paciente si se trata de una simulación del doctor. En caso de ser una introducción de una medición real, se llama a *addMeasurement* de *PatientModel*,

que se encarga de llamar al servicio encargado de gestionar la inserción de la nueva medición. La implementación de *addMeasurement* puede verse en la Figura 54.

4.4. Pruebas

Después de ver cómo se han implementado las distintas funcionalidades de la aplicación, en los siguientes apartados se va a exponer como se han escrito y preparado las pruebas de algunos componentes. Tanto Dart como Flutter tienen sus propias librerías para la creación de pruebas (*test* y *flutter_test*, respectivamente), las cuales añadimos al proyecto y usaremos para las pruebas unitarias y de interfaz.

4.4.1. Pruebas unitarias

Para empezar, vamos a ver las pruebas unitarias. Se trata de pruebas pequeñas que se crean para probar el correcto funcionamiento de un componente [11]. En nuestro caso, se van a utilizar para probar las funciones de las clases de Modelo y ver si la actualización de los atributos que componen el estado se realiza correctamente.

Las pruebas unitarias son más fiables y útiles a mayor código cubren. Por ello, se suelen escribir las suficientes para probar todos los caminos y condiciones posibles del código. Además, uno de los objetivos que se deben tener en cuenta al crearlas es que sean automatizables. Esto implica que deben ser independientes unas de otras y poder ser ejecutadas más de una vez. La automatización permite ejecutar estas pruebas con herramientas de integración continua para asegurar la calidad y buen funcionamiento de las nuevas versiones del código.

Debido a que las funciones de nuestras clases de Modelo son muy parecidas en sus implementaciones, vamos a centrarnos en la función *addPatient* de *DoctorModel* para ver unos ejemplos de cómo se crean estas pruebas. En la Figura 55 se puede apreciar un caso de prueba para la ejecución correcta de la función.

```

123 test('adds new code and does navigation', () async {
124     final testPatient = Patient(id: 456);
125     final testDose = Dose(id: 789);
126
127     when(service.addPatient(testPatient, testDose, 123))
128       .thenAnswer((_) => Future(() => Patient(code: 'AAAAA')));
129
130     await model.addPatient(testPatient, testDose);
131
132     expect(model.newPatientCode, 'AAAAA');
133     expect(model.pages, isEmpty);
134     expect(model.navigateToCode, true);
135 });

```

Figura 55: Prueba de ejecución correcta de *addPatient*

Un detalle importante es que las funciones de las clases de Modelo se apoyan en un servicio para realizar las operaciones con los datos. Como nuestros servicios realizan llamadas a un servidor, es necesario hacer un *mock* de los mismos [12]. Usamos la

librería de *mockito* para los *mocks* de los servicios. Al crear un *mock* de *DoctorService*, en el caso del ejemplo, podemos definir el comportamiento deseado para las llamadas a funciones de este, tal como se puede ver en la línea 127 de la Figura 55.

Así, le estamos indicando al servicio *mock*, que devuelva un objeto paciente con código "AAAAA" cuando se llame a su función *addPatient* con *testPatient*, *testDose* y 123 (que sería el identificador del doctor) como parámetros. El test luego llama a *addPatient* del modelo, y verifica con *expect* que los atributos se han actualizado correctamente tras su ejecución.

```
165 test('throws ConflictException', () async {
166   final testPatient = Patient(id: 456);
167   final testDose = Dose(id: 789);
168
169   when(service.addPatient(testPatient, testDose, 123))
170     .thenThrow(ConflictException());
171
172   await model.addPatient(testPatient, testDose);
173
174   expect(model.error, isEmpty);
175   expect(model.newPatientCode, isNull);
176   expect(model.navigateToCode, false);
177 });
```

Figura 56: Prueba de ejecución errónea de *addPatient*

Escribimos pruebas también para las ejecuciones erróneas, las cuales, en las funciones de nuestras clases de Modelo ocurren cuando el servicio lanza una excepción. Podemos hacer que la implementación *mock* de este lance también la excepción para probar ese flujo de ejecución de la función, tal como se puede ver en la línea 169 de la Figura 56.

4.4.2. Pruebas de interfaz

También podemos probar los *widgets* que componen la interfaz haciendo uso de la librería *flutter_test*. Las pruebas de estos componentes se realizan con la intención de comprobar que muestran la información correcta al usuario y su comportamiento ante las interacciones de este es también correcto [13].

Estas pruebas comparten con las unitarias algunos de sus requisitos, ya que también deben ser automatizables y se centran en un componente (*widget*) concreto.

Para ejemplificar el funcionamiento de estas pruebas, se van a mostrar algunos de los casos de prueba creados para el *widget* personalizado que muestra al paciente la dosis actual de su tratamiento: *DoseDisplay*. Al tratarse de un *widget* estático que no posee estado, lo que vamos a probar es que dependiendo de los valores del objeto dosis que se envía en su creación, se muestra la información correcta en pantalla.

Las pruebas de interfaz se componen de tres partes. Primero, se crea una instancia del *widget* que se quiere probar haciendo uso del objeto *WidgetTester* que recibe el caso de prueba como parámetro. Después, se interactúa con el componente en caso de ser

necesario. Para comprobar los cambios visuales en la interfaz, se crea un *matcher* con la clase *CommonFinders* de la librería de pruebas. Podemos crear *matchers* para texto, iconos o *widgets* de un tipo concreto. Los usaremos luego en las aserciones para ver si hay presente algún *widget* que encaje con el *matcher* o, incluso, un número concreto de ellos. Todo este procedimiento se puede ver en el caso de prueba mostrado en la Figura 57.

```

9   testWidgets('displays nothing label', (WidgetTester tester) async {
10      final testDose = Dose(dayQuantity: 0, nightQuantity: 0);
11
12      await tester.pumpWidget(MaterialApp(home: DoseDisplay(dose: testDose)));
13
14      final quantityFinder = find.text('Nada');
15
16      expect(quantityFinder, findsNWidgets(2));
17   });

```

Figura 57: Prueba del widget *DoseDisplay* cuando no hay dosis

En el caso concreto de nuestro *widget*, muestra dos columnas con los valores de la dosis que debe tomar el paciente por la mañana y por la noche, respectivamente. En el caso de prueba de la Figura 57, se comprueba que, al tener una dosis con cantidades de cero, se muestra correctamente el mensaje de que no debe tomar ninguna medicación en las dos columnas.

Si la dosis fuera superior a los 40 miligramos, el *widget* debe mostrar tanto la cantidad de pastillas de 10mg, como la combinación de pastillas de 40 y de 10 mg que puede tomar el paciente. Para comprobar que esto se muestra correctamente, escribimos el caso de prueba de la Figura 58.

```

92  testWidgets('displays both multiples of ten and forty',
93    (WidgetTester tester) async {
94      final testDose = Dose(dayQuantity: 40, nightQuantity: 40);
95
96      await tester.pumpWidget(MaterialApp(home: DoseDisplay(dose: testDose)));
97
98      final tenQuantityFinder = find.text('10mg');
99      final tenNumberFinder = find.text('x 4');
100     final fortyQuantityFinder = find.text('40mg');
101     final fortyNumberFinder = find.text('x 1');
102
103     expect(tenQuantityFinder, findsNWidgets(2));
104     expect(tenNumberFinder, findsNWidgets(2));
105     expect(fortyQuantityFinder, findsNWidgets(2));
106     expect(fortyNumberFinder, findsNWidgets(2));
107   });

```

Figura 58: Prueba del widget *DoseDisplay* para múltiplos de 40

Para la prueba de *widgets* que posean estado (*StatefulWidget*), la librería de pruebas de Flutter incorpora funciones que permiten realizar pulsaciones sobre botones u otros componentes interactivos, pudiendo comprobar el correcto cambio del estado. Se puede también realizar navegación entre *widgets* que representen ventanas.

4.5. Despliegue

Queremos que nuestra aplicación móvil multiplataforma esté disponible en las tiendas de aplicaciones tanto de Android (*Play Store*) como de iOS (*App Store*). Esto supone que, cuando se termina de implementar una nueva versión de la aplicación, debemos pasar las pruebas, compilar el código para las plataformas objetivo y actualizar la versión ofrecida en ambas tiendas.

Estas tareas son muy pesadas para realizarlas de forma manual cada vez que se quiera empujar una nueva versión de la aplicación a las tiendas. Para solucionar este problema, existen las herramientas de integración y despliegue continuo [14]. Con estas herramientas se puede observar el repositorio de un proyecto para que, cuando se actualice con cambios, se proceda a realizar una serie de pasos. Estos pasos se pueden personalizar según las necesidades de cada proyecto, pero suelen incluir la ejecución de pruebas, construcción de los artefactos necesarios y despliegue a las plataformas deseadas.

Codemagic es una herramienta online de integración y despliegue continuo que ha sido creada específicamente para aplicaciones hechas con Flutter, como ya indicamos en el apartado 2.6. Permite ejecutar las pruebas necesarias [15], compilar para las plataformas que se desee y empujar estas nuevas versiones a las tiendas de aplicaciones móviles de forma automática. En la Figura 59 tenemos un ejemplo de un proceso de *build* finalizado correctamente.

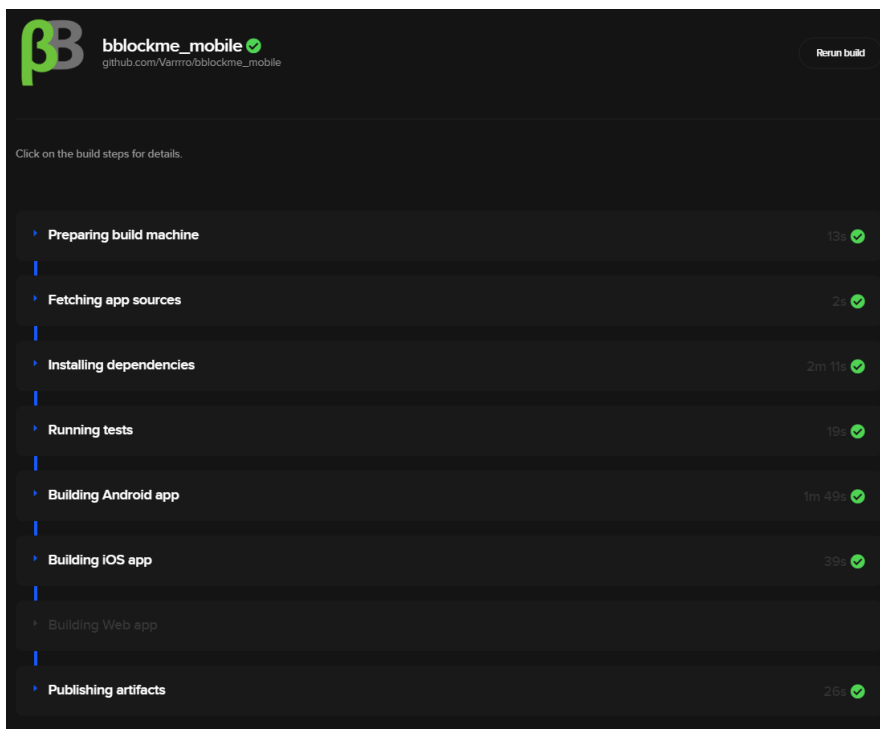


Figura 59: Ejecución correcta de un proceso de build

Una de las principales razones por las que usamos herramientas como Codemagic es la posibilidad de desplegar las nuevas versiones de nuestra aplicación en las tiendas de las distintas plataformas de forma automática. Este proceso de despliegue se puede dividir en dos partes: la firma del código y la publicación de la aplicación.

Tanto Google como Apple requieren que las aplicaciones publicadas en sus tiendas estén firmadas para identificar al autor de la aplicación y verificar que todas las actualizaciones provienen de una fuente fiable. Para poder firmar el código, necesitamos un certificado. La manera de obtener un certificado válido depende de la plataforma. En Android, solo tenemos que generar una *keystore* usando, por ejemplo, la herramienta *Java Keytool* de línea de comandos. Para iOS, necesitamos obtener el certificado directamente de Apple inscribiéndonos en su programa de desarrolladores de iOS y registrando un identificador para la aplicación.

Una vez que tenemos estos certificados, podemos configurar la firma del código para ambas plataformas en Codemagic [16][17]. Una vez está firmado el código, este puede ser publicado en las tiendas. Para ello, primero debemos configurar nuestras credenciales de acceso en la herramienta para que se pueda conectar a las tiendas por nosotros. En el caso de la Play Store de Android, podemos configurar en la consola de Google Play una nueva cuenta de servicio, que es la que usará Codemagic para conectarse. Al crear esta cuenta, obtenemos un archivo JSON con las credenciales, que debemos introducir en Codemagic. Para la App Store, es necesario el identificador de Apple y el de la aplicación, además de una contraseña específica para esta que debemos generar.

Al terminar toda esta configuración, nuestro proceso de *build* de Codemagic podrá firmar aplicaciones y publicarlas tanto a la Play Store como a la App Store [18][19].

5. Conclusiones y trabajo futuro

5.1. Conclusiones

Afronté este proyecto desde el primer momento entrando en un campo desconocido para mí por completo como es el desarrollo de aplicaciones móviles. La oportunidad de trabajar en un proyecto real y ayudar en un estudio médico me motivó para hacer frente a este reto. Haber cumplido los objetivos supone para mí una gran satisfacción personal.

He adquirido muchos conocimientos en lo que respecta a la gestión de proyectos y la ingeniería del software. He realizado reuniones con el cliente, formalizado requisitos, creado e implementado diseños, lo que me ha permitido tener una mejor comprensión de la importancia de todos estos pasos y consolidar las técnicas y procesos que permiten llevarlos a cabo.

Por otra parte, decidí también aprovechar la oportunidad de trabajar en un campo nuevo para aprender nuevas tecnologías. Desde el lenguaje hasta el editor de texto, todas las herramientas usadas han sido una novedad para mí, siendo además Flutter muy reciente. Gracias a esto he aprendido las bases de todas estas tecnologías y ampliado mi currículum como desarrollador.

En cuanto a la aplicación que he desarrollado, hay mucho lugar para la mejora, sobre todo en el aspecto de la seguridad, la interfaz y la experiencia general del usuario. Es normal, tratándose de mi primera aplicación móvil y, además, con tecnologías completamente nuevas para mí, pero me motiva a continuar trabajando en el campo de las aplicaciones móviles para mejorar.

Por último, he podido comprobar que Flutter es una tecnología bastante madura para su corta edad y, sobre todo, muy fácil de aprender y de usar. El desarrollo y la corrección de errores son sencillos gracias a la capacidad de actualizar el código durante la ejecución y ver los cambios. Sin embargo, donde gana en simplicidad y versatilidad, lo pierde en la escasa existencia de herramientas y baja madurez de estas, aunque esta es una situación que está cambiando a pasos agigantados y podemos esperar que este *framework* crezca en popularidad en los próximos meses.

5.2. Trabajo futuro

Como ya se ha indicado a lo largo de este trabajo, la aplicación móvil desarrollada está diseñada para trabajar con un servicio web, aunque se puede cambiar la implementación de los servicios de manera sencilla. Uno de los trabajos de extensión que se pueden realizar es la implementación de dicho servicio web. Esta implementación se va a abordar en el complemento de este trabajo.

También se podría considerar la creación de una aplicación web análoga usando Dart, ya que este lenguaje está orientado a la creación de clientes en general. Actualmente, existe soporte de Angular para el lenguaje y dispone de un compilador directo a JavaScript. La implementación de esta aplicación web podría compartir un gran porcentaje de su código con la aplicación móvil debido al uso de Dart por parte de ambas.

Otra idea interesante sería la utilización de dispositivos *wearables* para la toma de las mediciones. Este tipo de dispositivos está cada vez más de moda e incluyen sensores para controlar algunas constantes vitales, lo que se podría aprovechar para que el paciente no tenga que introducir todas las constantes en la aplicación.

Bibliografía

[1] *España, el país con más "smartphones" por habitante del mundo*. ABC [Online]. Recuperado en mayo de 2019 de:

https://www.abc.es/tecnologia/moviles/telefonía/abci-espana-pais-mas-smartphones-habitante-mundo-201611081019_noticia.html

[2] *iPhone vs Android: cuota de mercado*. PCWorld [Online]. Recuperado en mayo de 2019 de:

<https://www.pcworld.es/articulos/smartphones/iphone-vs-android-cuota-de-mercado-3692825/>

[3] Bonnie Eisenman, *Learning React Native: Building Native Mobile Apps with JavaScript*. 2nd ed. O'Reilly, 2017.

[4] Arvind Ravulavaru, *Learning Ionic 2*. 2nd ed. Packt Publishing, 2017.

[5] Jim Bennett, *Xamarin in Action*. Manning Publications, 2018.

[6] Dzenan Ridjanovic and Ivo Balbaert, *Learning Dart*. 2nd ed. Packt Publishing, 2015.

[7] Flutter documentation, *Introduction to widgets* [Online]. Recuperado en marzo de 2019 de:

<https://flutter.dev/docs/development/ui/widgets-intro>

[8] JSON, *Introducing JSON* [Online]. Recuperado en marzo de 2019 de:

<https://www.json.org/>

[9] Kyle Mew, *Learning Material Design*. Packt Publishing, 2015.

[10] Vadims Savjolovs, *Flutter app architecture 101: Vanilla, Scoped Model, BLoC* [Online]. Recuperado en abril de 2019 de:

<https://medium.com/flutter-community/flutter-app-architecture-101-vanilla-scoped-model-bloc-7eff7b2baf7e>

[11] Flutter documentation, *An introduction to unit testing* [Online]. Recuperado en abril de 2019 de:

<https://flutter.dev/docs/cookbook/testing/unit/introduction>

[12] Flutter documentation, *Mock dependencies using Mockito* [Online]. Recuperado en abril de 2019 de:

<https://flutter.dev/docs/cookbook/testing/unit/mocking>

[13] Flutter documentation, *An introduction to widget testing* [Online]. Recuperado en abril de 2019 de:

<https://flutter.dev/docs/cookbook/testing/widget/introduction>

[14] John F. Dooley, *Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring*. 2nd ed. Apress, 2017.

[15] Codemagic documentation, *Running automated tests* [Online]. Recuperado en mayo de 2019 de:

<https://docs.codemagic.io/testing/running-automated-tests/>

[16] Codemagic documentation, *Android code signing* [Online]. Recuperado en mayo de 2019 de:

<https://docs.codemagic.io/code-signing/android-code-signing/>

[17] Codemagic documentation, *iOS code signing* [Online]. Recuperado en mayo de 2019 de:

<https://docs.codemagic.io/code-signing/ios-code-signing/>

[18] Codemagic documentation, *Publishing to Google Play* [Online]. Recuperado en mayo de 2019 de:

<https://docs.codemagic.io/publishing/publishing-to-google-play/>

[19] Codemagic documentation, *Publishing to App Store* [Online]. Recuperado en mayo de 2019 de:

<https://docs.codemagic.io/publishing/publishing-to-app-store/>

En este Trabajo de Fin de Grado se realiza un estudio de la herramienta Flutter usando como ejemplo el desarrollo de una aplicación móvil para iOS y Android. El objetivo de la aplicación es mejorar la calidad del tratamiento que reciben los enfermos de cirrosis hepática.

Siguiendo los pasos de la Ingeniería del Software, se explicará todo el proceso de creación de la aplicación desde la planificación y el análisis de requisitos hasta la prueba y el despliegue del código implementado.

Se expondrá también el patrón usado para la gestión del estado de la aplicación, además de explicar todas las tecnologías utilizadas en la realización de este proyecto.