TRABAJO FIN DE GRADO

Touch&List

edix
Digital Workers

Lucia López Rodríguez

Alejandro Ocaña Bravo

Jesús Rodríguez Herranz

edix

INDICE

1. INTRODUCCIÓN

5 de Octubre de 2020, primer día del grado de Desarrollo de Aplicaciones Multiplataforma. Comienzan los nervios, ya no hay vuelta atrás. Empieza una nueva etapa en la que un terapeuta ocupacional, un arquitecto técnico y una administrativa, trabajarán para cambiar de rumbo, y de profesión, iniciando así una nueva carrera profesional con la que intentarán progresar y mejorar sus vidas. Todo esto, mientras compaginan los estudios, con sus trabajos actuales.

A la vez que empiezan los nervios y la ilusión de empezar algo nuevo, llegan los agobios. Un mal día en el trabajo, tareas personales que atender, a la vez que entregas con fecha que se tienen que terminar. No queda tiempo libre para descansar y empiezan el estrés y las carreras.

Resulta inconcebible atender todo sin organizarse un mínimo previamente, para ello, y gracias a las bases asentadas en la asignatura de Desarrollo de Aplicaciones Móviles, la dirección que tomó este proyecto desde el inicio fue la creación de una aplicación de gestión de tareas, para poder anotar cada una de ellas, organizar eventos, y tal y como indica el grado superior que hemos estudiado, desarrollarlo en un formato Multiplataforma, pudiendo utilizarla en un entorno web, en iOS y en Android.

Este proyecto intenta ofrecer una aplicación potente, sencilla y estable, que permitirá al usuario poder utilizarla desde la plataforma con la que esté más cómodo, manteniendo sus datos independientemente del entorno elegido.

A lo largo de este documento, se presenta la aplicación realizada, una descripción de características y requisitos principales, todas las tecnologías utilizadas durante el desarrollo de la herramienta, así como un análisis que recorrerá todas las fases del proyecto, desde el diseño, hasta las pruebas.

El código fuente de la aplicación se encuentra alojada en el siguiente repositorio: https://github.com/jrodriguezh/Touch-List.git

2. PALABRAS CLAVE

- **Flutter:** Es un framework que usa Dart como único código para crear aplicaciones multiplataforma. Compila a código nativo consiguiendo de esta manera un mayor rendimiento.
- **Dart:** Es un lenguaje de código abierto, estructurado y flexible, orientado a objetos, basado en clases, con herencia simple y soporte de interfaces, clases abstractas y tipado opcional de datos.
- **FireBase:** Es una plataforma en la nube que sirve para el desarrollo de aplicaciones web y móvil. Está disponible para distintas plataformas (iOS, Android y web).
- **FireStore:** Es la base de datos más reciente de Firebase para el desarrollo de apps para dispositivos móviles. Aprovecha lo mejor de Realtime Database con un modelo de datos nuevo y más intuitivo.
- **Widget:** Es una clase que nos permite construir los elementos de los que se compone la UI (interfaz de usuario). En flutter todo es un widget.
- **Tareas/Notas:** Se trata de una entrada de texto que incluirá una acción o información importante que quiera recordar el usuario. Una vez completada se marcará como realizada.
- **Eventos:** Al igual que las tareas, se trata de una entrada de texto a la que se le podrá asignar una franja de tiempo y ser visualizada a través de un calendario.
- **Aplicación:** Es un tipo de aplicación diseñada para ejecutarse en un dispositivo móvil, que puede ser un teléfono inteligente o una tableta.
- **Android:** Es un sistema operativo móvil basado en el núcleo Linux y otros softwares de código abierto.
- **iOS:** Es el sistema operativo diseñado por Apple para sus productos, iPhone, iPad, iPod Touch.
- **Web:** Es un documento accesible desde cualquier navegador con acceso a internet, y que puede incluir audio, vídeo, texto y sus diferentes combinaciones.

3. RESUMEN

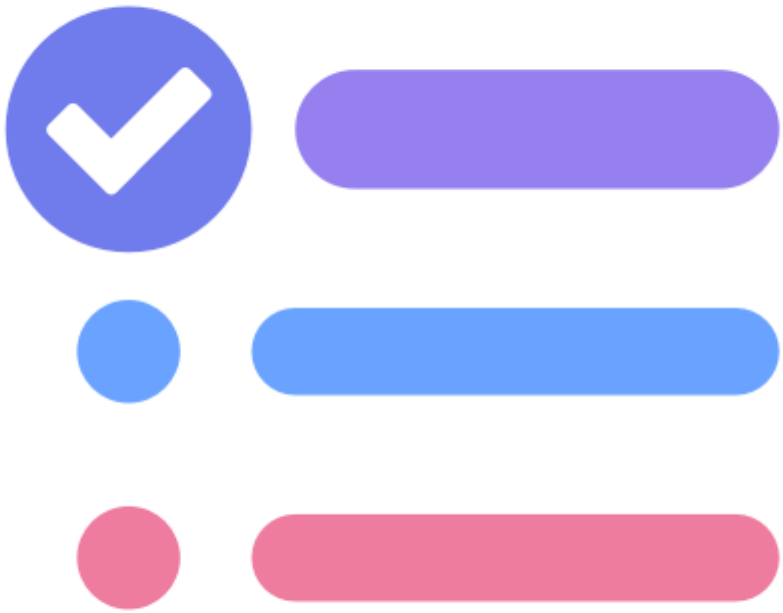
En una sociedad en la que el tiempo es el recurso más importante, ser capaz de organizarse cobra vital importancia. Todavía quedan nostálgicos que prefieren los calendarios y agendas de papel, pero no se puede negar que **los móviles se han convertido en herramientas de trabajo muy completas y versátiles** a través de las que organizar y gestionar nuestro tiempo.

Aunque, según el "Informe Digital 2022" publicado por la plataforma Hootsuite, "el uso medio diario de redes sociales a través de teléfonos móviles en España es de 1 hora y 55 minutos, siendo esta su principal función", nuestra finalidad con este proyecto es devolver a estos dispositivos su utilidad como herramientas y transformarlas en gestores de tiempo, en lugar de sustractores de este. Con la aplicación desarrollada, buscamos otorgar la funcionalidad al usuario de crear distintas tareas, modificarlas y eliminarlas una vez se hayan realizado, y, a su vez, poder añadir eventos y visualizarlos en un calendario, facilitando realizar un seguimiento de sus tareas diarias, independientemente del dispositivo al que tenga acceso en ese momento, ya sea Android, iOS o un entorno Web de escritorio. Devolviendo al usuario el control sobre su tiempo.

Debido a la gran variedad de plataformas que utilizan actualmente dichos usuarios, las empresas deben dedicar un gran esfuerzo en tiempo y dinero para el desarrollo específico de sus aplicaciones en cada una de estas plataformas (Android, iOS, web, Windows, mac y Linux), estableciendo para ello, en muchas ocasiones, diferentes equipos de trabajo que se especialicen en cada una de estas tecnologías. Adoptando la necesidad de implementar dos desarrollos simultáneos o, en su defecto, la migración de un código desarrollado para una plataforma específica sin la posibilidad de reutilizar el trabajo previamente realizado.

Teniendo todo esto en cuenta, durante el proceso de diseño, se decidió realizar el desarrollo a través de la implementación de una nueva tecnología desarrollada por Google en 2017, Flutter, la cual permite, a través del mismo código, y tras la actualización 3.0, desplegar un desarrollo en las seis plataformas previamente mencionadas, Android, iOS, web, Windows, Mac y Linux. Utilizando como herramienta el lenguaje Dart, desarrollado en 2011 y fuertemente influenciado por C#, JavaScript, Java y CoffeScript.

Finalmente, y, a pesar de que este framework permite realizar un desarrollo simultáneo en todas estas plataformas, nuestro proyecto únicamente se presenta desplegado en plataformas móviles y entorno web, debido a que la herramienta utilizada para el servicio de backend, Firestore, todavía no cuenta con una versión estable para su integración completa en Windows, Mac y Linux. Aunque su desarrollo ha sido estructurado para poder implementar estas funcionalidades en un futuro.



4. MÓDULOS FORMATIVOS APLICADOS

- **Programación**: Esta asignatura adquiere el papel más importante de todo el proyecto, ya que se ve reflejado en todos los pasos de desarrollo del mismo. El lenguaje de programación que se ha utilizado ha sido Dart principalmente.
- **Bases de Datos:** Para el funcionamiento de la aplicación creada se ha diseñado una BBDD no relacional. En este caso, se basa en colecciones que se ejecutan a través de instrucciones no SQL.
- **Entornos de Desarrollo:** Este módulo se ha aplicado en general por el uso constante que se ha hecho de git y GitHub.
- **Lenguaje de Marcas:** Dado que la aplicación creada es multiplataforma, este módulo se ha utilizado para la creación de la página web generada a través del propio código de Flutter.
- **Desarrollo de interfaces:** La implementación de este módulo ha sido fundamental para el desarrollo del diseño de interfaz de usuario, tanto para la pantalla de Login y de Registro, como para las pantallas principales de tareas y eventos.
- **Inglés:** El código, así como su documentación estará desarrollado totalmente en inglés.
- **Programación Multimedia y Dispositivos Móviles:** Este módulo se ha utilizado para la creación de la aplicación en Android, Ios, Web y Windows. Así como para la utilización del SDK de Android, Android Studio y Visual Studio.
- **Acceso a Datos:** Ha sido utilizado para el acceso desde la aplicación a la base de datos.
- **Programación de Servicios y Procesos:** Por último, este módulo se ha aplicado para el desarrollo de la programación multihilo.

5. HERRAMIENTAS/LENGUAJES UTILIZADOS

- **Visual Studio Code:** Se ha utilizado como IDE de desarrollo, en el que se ha integrado Flutter con gran facilidad y se han instalado diferentes extensiones para realizar el trabajo.

- **Android Studio:** es el entorno de desarrollo integrado oficial para la plataforma Android, es necesario para configurar el SDK dentro del proyecto.

- **Scrcpy (Emulador):** Proyecto independiente que permite poner en comunicación un dispositivo físico real con el ide de desarrollo a través de una conexión usb utilizando TCP/IP.

- **Firebase:** Se ha utilizado como sistema de autenticación de datos.

- **Flutter:** Framework de desarrollo para crear aplicaciones multiplataforma.

- **Dart:** Es el lenguaje con el que se ha desarrollado gran parte de la aplicación. Se trata de un lenguaje orientado a objetos y con análisis estático de tipo.

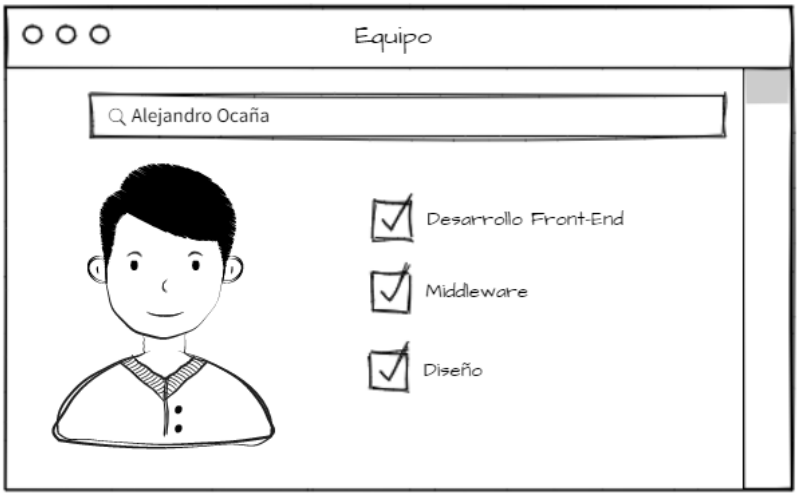
- **Node.js:** Es un entorno controlado por eventos diseñado para crear aplicaciones escalables, permitiéndote establecer y gestionar múltiples conexiones al mismo tiempo.

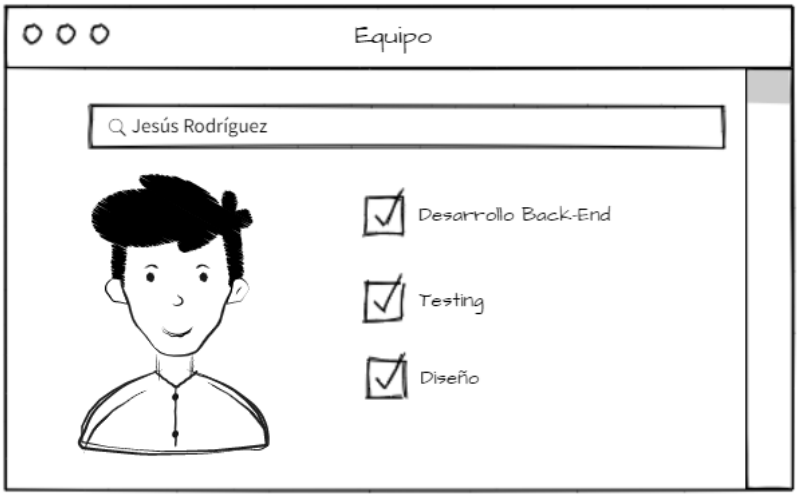
- **Uizard.io:** Herramienta de diseño gratuita que facilita la creación de maquetas gracias a sus plantillas.

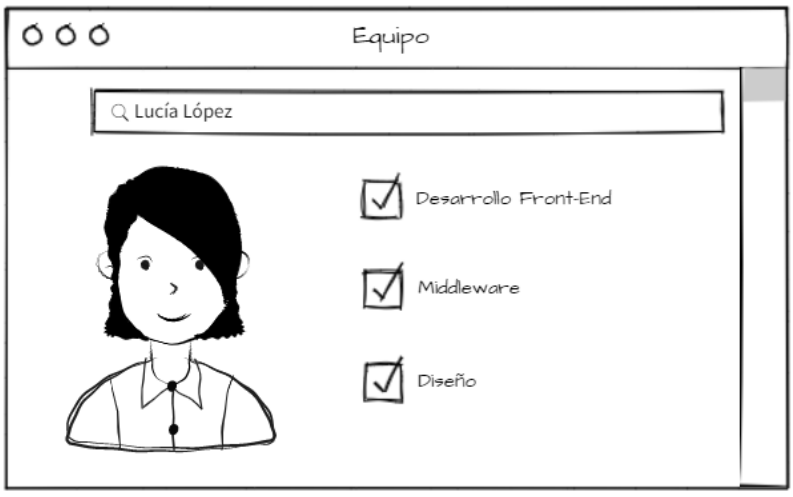
- **GitHub:** Es una plataforma de desarrollo colaborativo, que permite gestionar proyectos y controlar versiones de código.

- **JSON:** acrónimo de JavaScript Object Notation, es un formato de texto sencillo para el intercambio de datos.

6. COMPONENTES DEL EQUIPO







7. FASES DEL PROYECTO

En este apartado se da una visión general de las fases seguidas a lo largo de la realización del proyecto, que son las correspondientes a la mayoría de un desarrollo de software.

- ➔ Análisis: Definición de requisitos con el objetivo de dilucidar exactamente qué es lo que se desea construir.

- ➔ Diseño: Definición de la arquitectura de la aplicación. Esto incluye diversos diagramas que definirán el diseño de pantallas y su relación, interfaz, base de datos y seguridad.

- ➔ Implementación: Programación de la aplicación utilizando el framework Flutter y el lenguaje Dart.

- ➔ Pruebas: Realización de una fase de pruebas generales para comprobar la calidad y efectividad del software desarrollado.

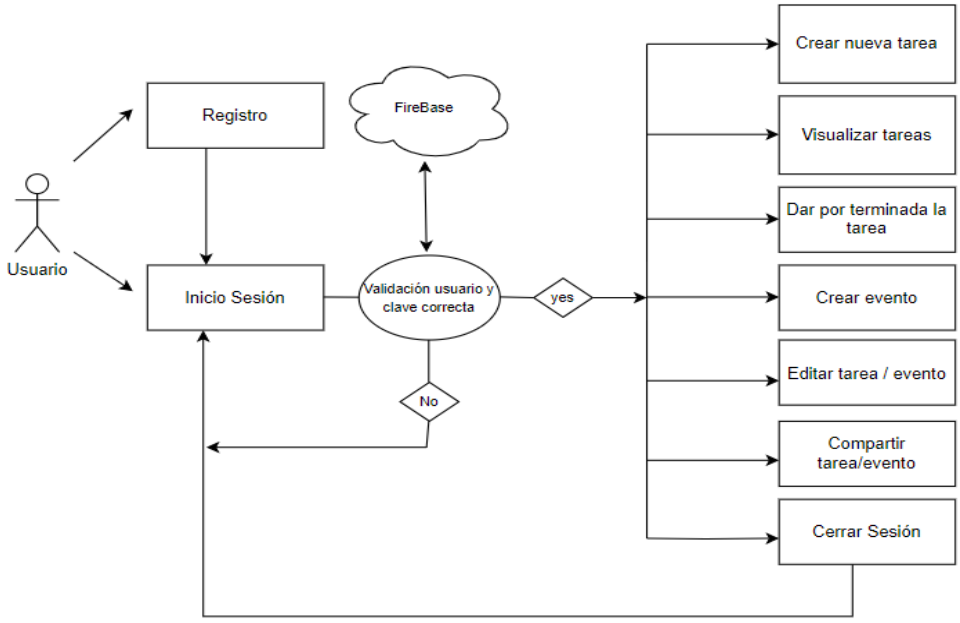
- ➔ Documentación: La documentación es algo indispensable para el correcto mantenimiento de cualquier producto software. Con el fin de dejar constancia por escrito de las fases llevadas a cabo en el ciclo de vida del producto.

- ➔ Mantenimiento: En este proyecto puede no tener mucha cabida, pero se orienta a un trabajo futuro con bastante proyección con el fin de mejorar y ampliar la aplicación.

ANÁLISIS:

Durante esta primera fase se decidió realizar una lluvia de ideas, para definir los requisitos que se utilizarán como base para el desarrollo y diseño de todos los aspectos de la aplicación.

En primer lugar, se realizó un diagrama con los distintos casos de uso posibles para visualizar de forma más clara la comunicación y el comportamiento del sistema interactuando con los usuarios y/u otros sistemas.



A través de este diagrama, se obtuvieron como resultado los siguientes requisitos:

- **Requisitos funcionales:**

Los requisitos funcionales describen todas las interacciones que tendrán los usuarios con el software.

GESTIÓN DE USUARIOS:

1. **Registro**
    a. La aplicación permitirá al usuario introducir los datos para poder crear una nueva cuenta.
    b. La aplicación se encargará de validar los datos.
    c. Se mostrará una alerta de error si alguno de los datos introducidos es incorrecto.

d. Una vez validados los datos, se enviará una petición a Firebase y se guardarán los datos.
e. Se enviará un correo electrónico de confirmación al usuario al email proporcionado.
f. Se mostrará una nueva ventana en el caso de que el registro se haya realizado con éxito.

2. Identificación

a. Para iniciar sesión el usuario deberá identificarse con su correo electrónico y contraseña correspondiente.
b. El sistema se encargará de validar y permitir o denegar el acceso a la aplicación y mostrará un mensaje de error en el caso de que la identificación no sea válida.
c. Si el usuario y contraseña son correctos, la aplicación redirigirá al usuario a la pantalla principal, pudiendo ver las tareas y eventos creados en otro inicio de sesión.

3. Cierre de sesión

a. Cualquier usuario de la aplicación podrá finalizar sesión en la aplicación desplegando el menú de la pantalla principal y pulsando la opción “Log Out”.
b. En el caso de que el usuario pulse el botón de cierre de sesión, el sistema mostrará un mensaje, para asegurarse de que el usuario quiere cerrar sesión, permitiendo aceptar o cancelar.
c. El usuario será redirigido a la pantalla principal, pero en este caso sin estar logado.

4. Modificación/recuerdo de contraseña

a. La pantalla “login” dispondrá de un botón a través del cual se accederá a una segunda pantalla de recuerdo de contraseña.
b. Dicha pantalla contará con una caja de texto donde podrá introducir su email para comenzar el proceso de recuerdo de contraseña.

c. Una vez introducido el email, podrá interactuar con un botón que realizará una petición a Firebase para que este genere un link de reseteo de contraseña y lo envíe a su correo electrónico.
d. Una vez el usuario haga click en el link e introduzca una nueva contraseña su contraseña habrá sido restablecida.

APLICACIÓN:

1. **Pantalla principal**
    a. La pantalla principal constará de dos pestañas reflejadas en la parte inferior. Una pestaña en la que se podrá añadir notas escritas por el usuario y otra en la que se accederá al calendario, en la cual se podrán añadir eventos con diferentes opciones.

2. **Pestaña de notas**
    a. Mostrará una lista de las notas creadas en la sesión actual o anteriores.
    b. Se podrá añadir una nota a través de un botón, o modificar una ya creada clicando en ella. Permitiendo aceptar, borrar o compartir.

3. **Pestaña de calendario**
    a. Mostrará un calendario con el mes actual, pudiendo elegir cualquier día del mes para ver los eventos programados en una vista previa en la parte inferior. Haciendo doble clic o un toque mantenido, se abrirán los eventos programados detallados por hora.
    b. Los eventos se crearán a través de un botón flotante dispuesto en el margen inferior derecho. Se abrirá una nueva pantalla en la que seleccionar en un desplegable el comienzo y la finalización del evento, pudiendo elegir tanto el día como la hora.

    Se podrá personalizar el nombre del evento, así como los detalles del mismo.

- **Requisitos no funcionales:**

Requisitos complementarios o atributos de calidad. Especifican criterios que juzgan las operaciones del sistema en lugar de su comportamiento.

1. **Seguridad**
    a. Para poder utilizar la aplicación se requiere el registro previo, siendo los datos introducidos por el usuario validados por email y cifrados.
2. **Mantenibilidad**
    a. La aplicación estará disponible para todo tipo de dispositivos de diferentes plataformas, Android e IOs, y para web.
    b. Será necesario el acceso a internet para la creación de tareas, así como para compartirlas por otras aplicaciones a otros usuarios.
3. **Interfaz y usabilidad**
    a. La aplicación constará de una interfaz fácil, intuitiva y atractiva de tal forma que el usuario pueda interactuar con sus acciones de la forma más sencilla posible.

Antes de empezar a trabajar en el proyecto, se ha tenido que determinar la metodología que se iba a llevar a cabo a lo largo del proyecto, teniendo en cuenta que una metodología de desarrollo de software es vital para su puesta en marcha. Se trata del conjunto de técnicas y métodos que se utilizan para diseñar una solución de software informático y son imprescindibles por una cuestión de organización, ya que las tareas necesitan estar ordenadas y saber cómo se van a realizar. También sirve para controlar el desarrollo del proyecto y minimizar márgenes de errores anticipándose así a estas situaciones. Además, permiten ahorrar tiempo y gestionar los recursos disponibles.

La metodología con la que se ha decidido trabajar ha sido la metodología Scrum, ya que **permite hacer una división muy específica y organizada del proyecto.**

Además, esta metodología permite la planificación por semanas y, al final de cada Sprint o iteración, se va revisando el trabajo validado de la anterior semana. En función de esto, se priorizan y planifican las actividades en las que se invertirán los recursos en el siguiente Sprint.

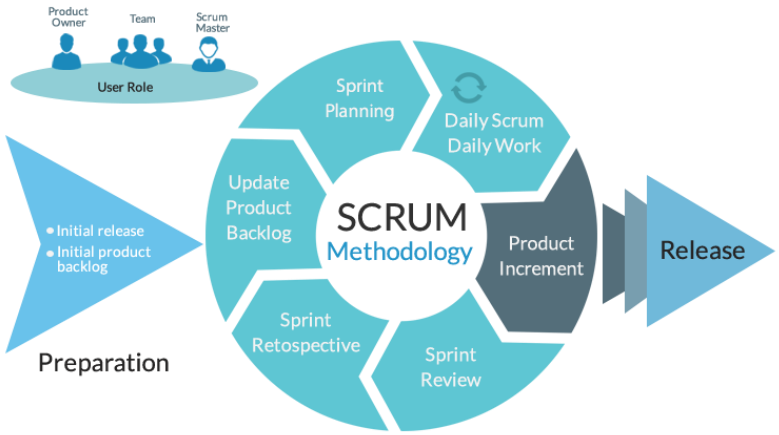
De esta manera, **el equipo se centra en tareas más pequeñas** donde cada miembro tiene clara cuál es su función.

En función de las necesidades y los requerimientos, **se crean listas de tareas y se van etiquetando** según están completadas, asignadas o en proceso. Así se consigue una mayor colaboración y organización en el equipo.

Las **fases de la metodología Scrum** se resumen en **5 pasos o etapas de implementación**:

1. Inicio
2. Planificación y estimación
3. Implementación
4. Revisión y retrospectiva
5. Lanzamiento

A continuación, se ilustra la metodología de desarrollo utilizada en este proyecto mediante un gráfico sencillo.



DISEÑO:

El diseño proporciona una idea completa del software desarrollado en el proyecto y puede ir sufriendo modificaciones a lo largo del desarrollo.

En base a los requisitos especificados en el análisis, y dado a las integraciones que ofrece Flutter, se ha decidido realizar una relación con Firebase para realizar la autenticación y protección de datos del usuario en la nube.

En cuanto al modelo de datos, se planteó que el usuario pudiera acceder a sus notas y eventos desde cualquier plataforma. Por lo que, aun habiendo desarrollado en un principio una base de datos para implementar el almacenamiento localmente, finalmente se decidió realizar la migración a Cloud Firestore, garantizando así también la protección de la información.

Se trata de una base de datos NoSQL orientada a los documentos. A diferencia de una base de datos SQL, no hay tablas ni filas; En su lugar, se almacenan los datos en documentos, que se organizan en colecciones.

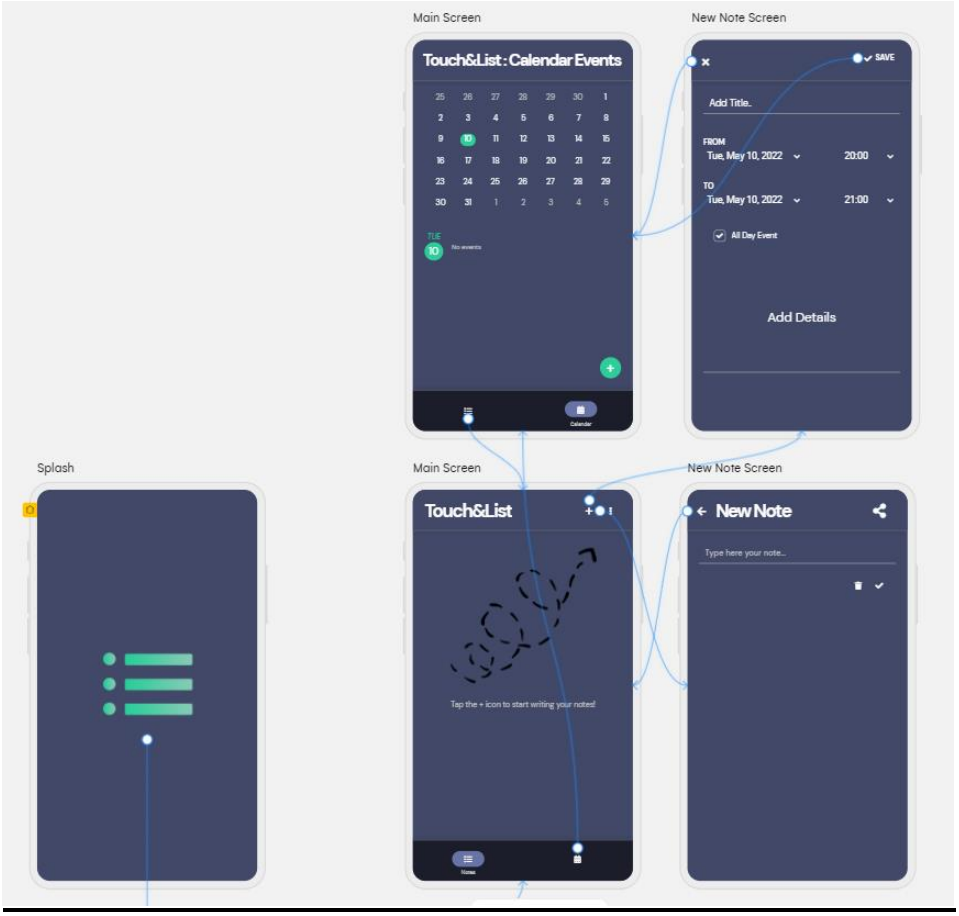
Cada documento contendrá un conjunto de pares clave-valor, dado que Cloud Firestore está optimizado para almacenar grandes colecciones de documentos pequeños.

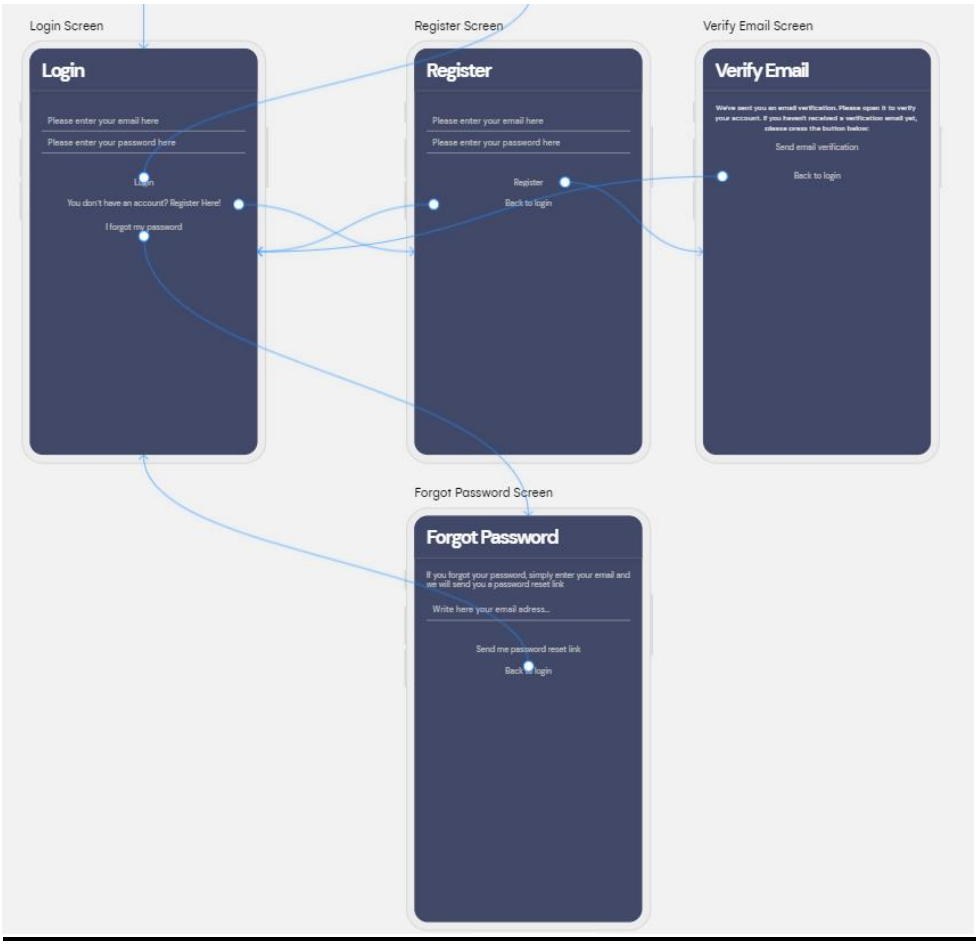
Todos los documentos se deberán almacenar en colecciones, y podrán contener subcolecciones y objetos anidados. Además, ambos podrán incluir campos primitivos, como strings, o tipos de objetos complejos, como listas.

Las colecciones y los documentos se crearán de manera implícita en Cloud Firestore; solo se debe asignar datos a un documento dentro de una colección. Si la colección o el documento no existiesen, Cloud Firestore los creará.

Rescatando el diagrama de casos de usos realizado anteriormente, se procedió a diseñar la interfaz, con el propósito de que sea intuitiva y simple, pero que a la vez sea capaz de cubrir todos los requisitos enumerados en el análisis.

Se utilizó la herramienta gratuita **uizard.io,** la cual permite realizar un enmaquetado bastante fiel a la realidad, pudiendo incluso implementar el flujo de trabajo de la aplicación, puede consultarse en el siguiente enlace: https://app.uizard.io/p/72830815

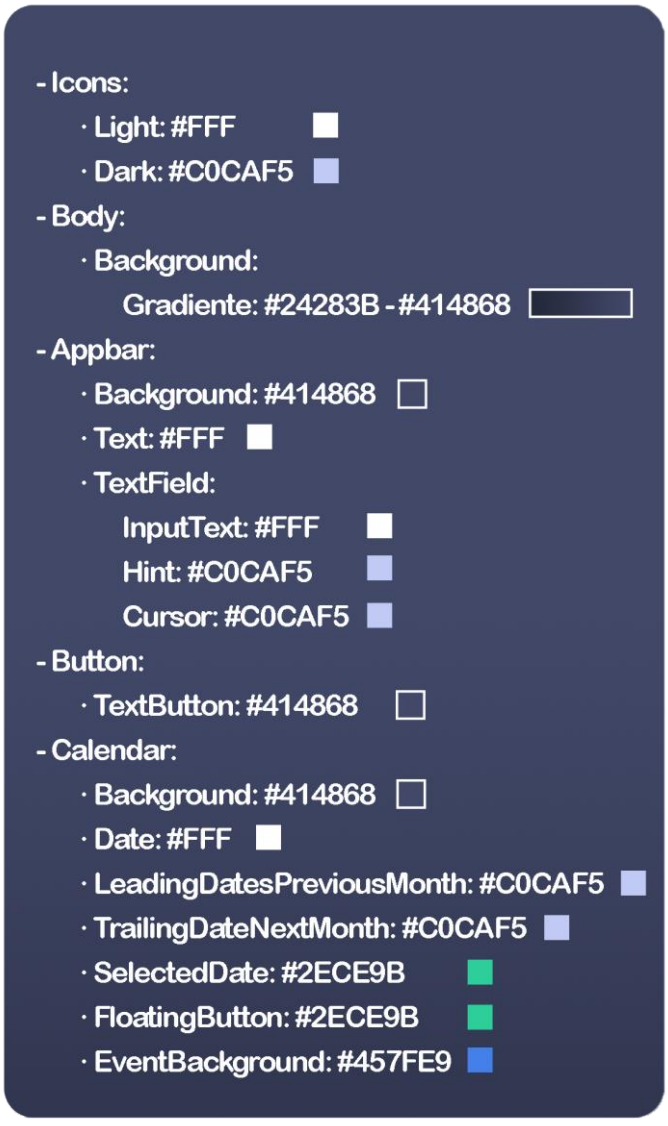

Main Screen
Touch&List : Calendar Events
New Note Screen
SAVE
Add Title..
FROM
Tue, May 10, 2022
20:00
TO
Tue, May 10, 2022
21:00
All Day Event
Add Details
Splash
Main Screen
Touch&List
Tap the + icon to start writing your notes!
New Note Screen
New Note
Type here your note...


Login Screen
Login
Please enter your email here
Please enter your password here
You don't have an account? Register Here!
I forgot my password
Register Screen
Register
Please enter your email here
Please enter your password here
Register
Back to login
Verify Email Screen
Verify Email
Send email verification
Back to login
Forgot Password Screen
Forgot Password
Write here your email adress...
Send me password reset link

Para la realización del icono se escogió un diseño simple, intentando mantener la estética de material design y aplicar la paleta de color escogida:

Obteniendo el siguiente icono:

También se aplicó esta paleta de color en el resto de elementos de la aplicación según el siguiente esquema:



IMPLEMENTACIÓN:

Antes de comenzar con las fases de desarrollo, en este proyecto se ha tenido que estudiar la tecnología con la que se ha decidido trabajar, ya que no es una incluida en el plan de estudio del FPII. Para ello ha sido necesario entender los siguientes puntos:

Flutter:

Flutter es un framework de código abierto, es decir, una herramienta que facilita un esquema a la hora de desarrollar una app. Fue creado por Google y permite desarrollar a partir de un mismo código herramientas compatibles con dispositivos tanto Android como iOS.

Se creó en la compañía multinacional para uso interno, aunque tras ver todo el potencial que podía alcanzar, se decidieron a lanzarlo como un framework de código libre. Hoy en día es uno de los métodos de desarrollo que más está creciendo, y gracias a la importancia de las aplicaciones móviles actualmente, se espera que Flutter siga siendo una herramienta esencial en el desarrollo de software a medida.

El desarrollo de aplicaciones móviles con Flutter permite crear soluciones, tanto para móvil, web como para escritorio, pudiendo adaptarse a cualquier plataforma o entorno final del usuario.

Las aplicaciones se programan con el lenguaje Dart (también desarrollado por Google). Este lenguaje de programación puede incorporar **cambios en tiempo real mientras que se está desarrollando la app**, facilitando el trabajo a los desarrolladores, la funcionalidad denominada, **HotReload,** inyecta ficheros de código fuente actualizados en la Máquina Virtual Dart, sin tener que realizar una nueva compilación, acelerando el proceso de desarrollo.

Para comprender la dinámica de flutter es importante definir su elemento estructural más básico, los **Widgets.**

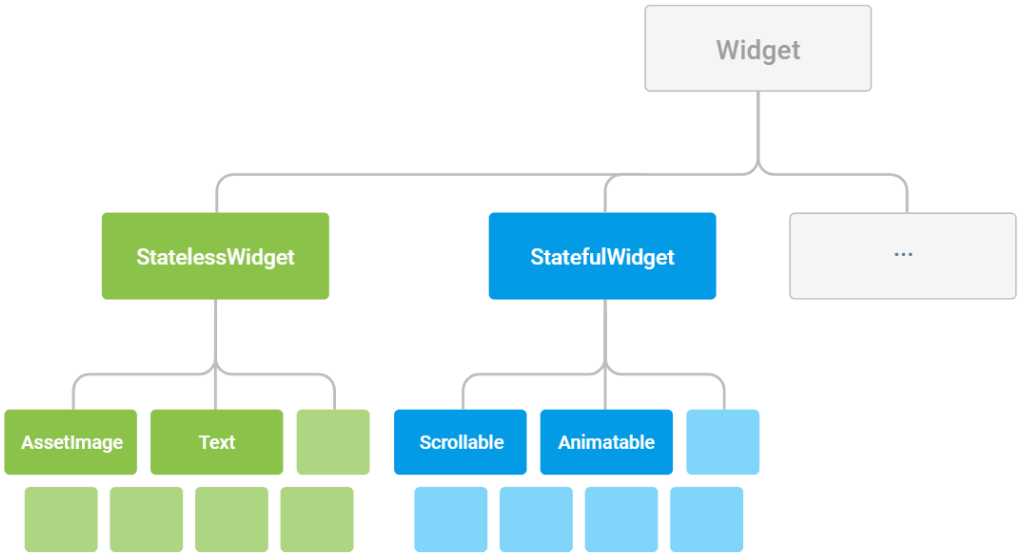
Lo primero que hay que comprender para llegar a desarrollar una aplicación en Flutter/Dart es que **“todo es un widget”.**

Los widgets son componentes básicos de la interfaz de usuario de una aplicación de Flutter. Cada widget es una declaración inmutable de parte de la interfaz de usuario. Esta

característica de componibilidad ayuda a crear una interfaz de usuario sea cual sea su complejidad.

Los widgets forman una jerarquía basada en la composición. Cada widget se integra en el interior y hereda propiedades de su padre. No existe un objeto "application" separado. En su lugar, el widget raíz sirve para esta función.

Esta jerarquía de clases es superficial y amplia para maximizar el número posible de combinaciones:



Como se puede comprobar en el anterior esquema, los widgets en Flutter extienden de dos widgets principales: StatelessWidget y StatefulWidget.

- **StatelessWidget**: Son los denominados Widgets sin estado o estáticos. Se trata de widgets que no guardan ningún tipo de estado, es decir, no tienen valores que pueden cambiar. Por ejemplo, un widget tipo Image que solo muestra una imagen en pantalla.
- **StatefulWidget**: son widgets cuyo estado puede cambiar a través del tiempo. En otras palabras, contienen algún estado mutable que se necesita rastrear. Imagina un widget de texto que muestra la cantidad de veces que se ha hecho clic en un botón.

Documentación oficial

Toda la documentación oficial respecto a flutter puede ser encontrada a través del siguiente link: https://docs.flutter.dev

Firebase:

Es una plataforma en la nube que tiene como función esencial hacer más sencilla la creación tanto de aplicaciones webs como móviles y su desarrollo, procurando que el trabajo sea más rápido, pero sin renunciar a la calidad requerida.

Es especialmente interesante para que los desarrolladores no necesiten dedicarle tanto tiempo al backend, tanto en cuestiones de desarrollo como de mantenimiento.

Podemos englobar la funcionalidad de Firebase en cuatro grupos:

Desarrollo:

- **BBDD en tiempo real:** firebase cuenta con dos bases de datos disponibles realtime database y firestore, la cual utilizamos en nuestro proyecto, las BBDD son de tipo no SQL y almacenan los datos como JSON. Firebase envía automáticamente eventos a las aplicaciones cuando se produce una modificación de sus datos.
- **Autenticación de usuarios:** Permite realizar el registro mediante email y contraseña, así como el acceso utilizando perfiles de otras plataformas como Facebook, Google o Twitter.

- **Almacenamiento en la nube:** Se pueden guardar ficheros de cualquier tipo permitiendo establecer reglas para su acceso a través de la aplicación.

- **Crash reporting:** Permite realizar un seguimiento de errores, pudiendo elaborar informes detallados con datos como el dispositivo y la situación en la que se da la excepción, siendo capaz de clasificarlos por gravedad.

- **Test Lab:** Permite realizar testing de la app en dispositivos Android virtuales pudiendo establecer parámetros de configuración específicos.

- **Remote Configuration:** Permite modificar funciones y aspectos de la aplicación sin la necesidad de publicar una actualización de la versión, simplemente especificando a qué usuarios va dirigido.

- **Cloud Messaging:** Envío de mensajes a los usuarios en tiempo real.

- **Hosting:** Con certificados SSL y HTTP2.

Crecimiento:

- **Notifications:** Proporcionando la utilidad de diseñar e integrar el envío de notificaciones push, siendo capaz de especificar a qué usuario va dirigido.

- **App Indexing:** Permitiendo integrar la aplicación en los resultados del buscador de Google.

- **Dynamic Links:** Pudiendo redirigir al usuario a ciertos contenidos dependiendo de la plataforma en la que se encuentre.

- **Invites:** Aporta la posibilidad de otorgar a los usuarios la capacidad de invitar a otros usuarios a utilizar la app mediante el envío de emails o SMS.

- **Adword:** Campañas publicitarias online.

Monetización:

- Capacidad de integrar publicidad en las aplicaciones a través de AdMob.

Analítica:

- Permite obtener mediciones y análisis de los eventos que tienen lugar en la aplicación, orientadas a la estrategia de marketing.

Esta plataforma puede integrarse en los siguientes ecosistemas:

Android, iOS, Unity, Web y, tras la última actualización, macOS.

Implementación de firebase en la APP

El primer paso para implementar Firebase en Flutter es instalar FlutterFire CLI (command-line interface), que es una interfaz basada en texto que sirve para gestionar la interacción con Firebase desde nuestra terminal.

A través de ella, logueandose con cualquier cuenta de google, se podrá configurar todo el proyecto de Firebase a través del comando *flutterfire configure* y asignando las plataformas con las que se quiere integrar firebase en el proyecto. Una vez realizado esta configuración, se podrá acceder a la consola de firebase a través de https://console.firebase.google.com

Documentación oficial

Para información más detallada sobre el funcionamiento de la plataforma, se puedeencontrar la documentación oficial a través del siguiente enlace: https://firebase.google.com/docs

Finalmente, para la implementación de estas tecnologías en el proyecto, hay que acogerse a un patrón de diseño que sirva como guía en el proceso de desarrollo intentando hacerlo estable y escalable, en este caso, se eligió el patrón BLoC (Business Logic Components).

Bloc:

Patrón de arquitectura de software que permite separar la lógica de negocio de la interfaz gráfica.

Fue desarrollado por Paolo Sorares y Cong Hui de Google, de forma específica para Flutter, con la finalidad de poder reutilizar el código entre sus aplicaciones móviles, utilizando flutter con Dart, y aplicaciones web, utilizando Angular.

Un BloC es un componente intermediario entre las vistas y el modelo, como puede ser el presenter cuando se utiliza MVP o el viewmodel al utilizar MVVM. El patrón bloc tiene los siguientes objetivos:

- Centralizar la lógica de negocio.
- Centralizar cambios de estado.
- Mapear al formato que necesita la vista.

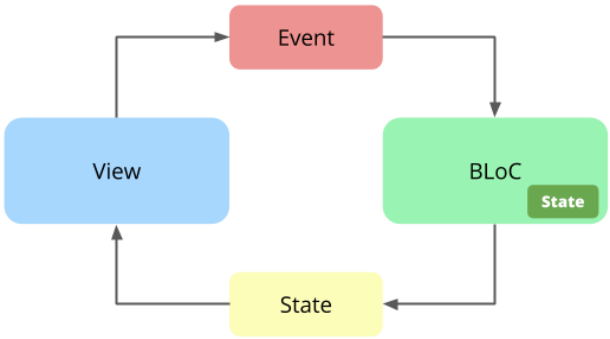
Esto es, que por cada widget lo suficientemente importante, se va a tener su correspondiente BloC. Por ejemplo, la vista (componente o widget) que representa a una página o un estado que es compartido por más de una vista, serían ejemplos donde se debería crear un BLoC.

Centralizar la lógica de negocio, la idea, pretende extraer todo lo que no se considera renderización de la vista, en unas clases llamadas Bloc, ajenas a cualquier tecnología o librería. De esta forma utilizando el principio de inversión de dependencia, al solo contener lógica, se puede escalar mejor a futuros cambios o a utilizarse en diferentes tecnologías. Además, facilita la comprobación del código al separar la funcionalidad de la apariencia.

Centralizar cambios de estado, las clases Bloc, son las encargadas de recibir las acciones o eventos que se producen en la aplicación y que modifican el estado, mantienen el estado en memoria y comunican a las partes implicadas (componentes o widgets) los cambios de estado.

Mapear al formato que necesita la vista, estas clases también van a encargarse de formatear los datos según las necesidades de las vistas, de esta forma, la lógica de presentación de formateo también será reutilizable.

Bloc se basa en un flujo de datos unidireccional.



El binding entre la vista y el estado es solo en una dirección, leyendo el estado desde la vista para renderizar. Para modificar el estado, se manejan los eventos en el componente o widget y después invocando acciones o eventos en el bloc, lo que modifica el estado en el Bloc, funcionando como una caja negra.

Esto es, que el bloc, visto como una caja negra, es un componente que va a recibir eventos o acciones como inputs, teniendo como respuesta uno o varios estados observables como outputs.

Teniendo todos estos conceptos en cuenta, se procedió a desarrollar el proyecto de la siguiente manera:

Desarrollo de la aplicación (Flutter, Firebase y arquitectura Bloc)

Proceso de autenticación

Durante el desarrollo, se implementó la arquitectura Bloc en el proceso de autenticación, para ello, se partió de la clase ***main.dart,*** esta clase será la encargada de gestionar el renderizado de la UI (vistas y widgets) en función de los estados observables en los que se encuentre la clase ***auth_bloc.dart,*** y esta, a su vez, será la encargada de establecer las peticiones pertinentes al provedoor ***(firebase)*** del servicio de autenticación, a través de los métodos de la clase abstracta ***auth_provider.dart.*** Dichos estados, serán emitidos según los eventos, respecto al proceso de autenticación, en los que se encuentre y/o quiera realizar el usuario en un momento dado. Véase la importancia de que, ante un mismo evento, la clase bloc puede emitir diferentes estados observables, adaptándose a cualquier situación posible.

Dentro de la clase ***auth_bloc.dart*** encontraremos los siguientes eventos y estados:

Eventos:

- Debería registrarse
  Posibles estados:
  - Registrándose.

```
//registry
  on<AuthEventShouldRegister>((event, emit) {
    emit(const AuthStateRegistering(
      exception: null,
      isLoading: false,
    ));
  });
```

- Olvido de contraseña
  Posibles estados:
  - Contraseña olvidada y aún no ha introducido el email (El usuario quiere ir a la pantalla de olvido de contraseña)
  - Contraseña olvidada y el usuario ya ha introducido el email (El usuario quiere que se envíe el correo de reseteo de contraseña)
  - Contraseña olvidada y el usuario ya ha introducido el email (El usuario quiere que se envíe el correo de reseteo de contraseña y el proveedor está procesando la petición)
  - Contraseña olvidada y el usuario ya ha introducido el email (El usuario quiere que se envíe el correo de reseteo de contraseña y ha ocurrido una excepción)

```
//forgot password
on<AuthEventForgotPassword>((event, emit) async {
  emit(const AuthStateForgotPassword(
    exception: null,
    hasSentEmail: false,
    isLoading: false,
  ));
  final email = event.email;
  if (email == null) {
    return; //user just want to go to forgot-password screen
  }

  //user wants to send a forgot-password email
  emit(const AuthStateForgotPassword(
    exception: null,
    hasSentEmail: false,
    isLoading: true,
  ));

  bool didSendEmail;
  Exception? exception;
  try {
    await provider.sendPasswordReset(toEmail: email);
    didSendEmail = true;
    exception = null;
  } on Exception catch (e) {
    didSendEmail = true;
    exception = e;
  }
  emit(AuthStateForgotPassword(
    exception: exception,
    hasSentEmail: didSendEmail,
    isLoading: false,
  ));
});
```

- Envío de email de verificación
  Posibles estados:
  - Envío de email de verificación:

```
// send email verification
on<AuthEventSendEmailVerification>((event, emit) async {
  await provider.sendEmailVerification();
  emit(state);
});
```

- Registro
  Posibles estados:
  - Se necesita verificación.
  - Se necesito verificación y ha ocurrido una excepción.

```
on<AuthEventRegister>((event, emit) async {
    final email = event.email;
    final password = event.password;
    try {
      await provider.createUser(
        email: email,
        password: password,
      );
      await provider.sendEmailVerification();
      emit(const AuthStateNeedsVerification(isLoading: false));
    } on Exception catch (e) {
      emit(AuthStateRegistering(
        exception: e,
        isLoading: false,
      ));
    }
  });
```

- Inicialización
  Posibles estados:
  - Logged Out (El usuario aún no está loggeado)
  - Necesita verificación (El usuario requiere verificación)
  - Logged In (El usuario ya está logeado)

```
// initialize
on<AuthEventInitialize>((event, emit) async {
  await provider.initialize();
  final user = provider.currentUser;
```

```
    if (user == null) {
      emit(
        const AuthStateLoggedOut(
          exception: null,
          isLoading: false,
        ),
      );
    } else if (!user.isEmailVerified) {
      emit(const AuthStateNeedsVerification(isLoading: false));
    } else {
      emit(AuthStateLoggedIn(
        user: user,
        isLoading: false,
      ));
    }
  });
```

- LogIn
  Posibles estados:
  - El usuario no está loggeado y el proveedor está cargando la petición.
  - El usuario no está loggeado y el usuario necesita verificación.
  - El usuario no está loggeado y no ha habido ninguna excepción.
  - El usuario no está loggeado pero ha habido una excepción.

```
// log in
  on<AuthEventLogIn>((event, emit) async {
    emit(
      const AuthStateLoggedOut(
          exception: null, isLoading: true, loadingText: "Loading"),
    );
    final email = event.email;
    final password = event.password;
    try {
      final user = await provider.logIn(
        email: email,
        password: password,
      );

      if (!user.isEmailVerified) {
        emit(
          const AuthStateLoggedOut(exception: null, isLoading: false),
        );
        emit(const AuthStateNeedsVerification(isLoading: false));
      } else {
        emit(
          const AuthStateLoggedOut(exception: null, isLoading: false),
        );
        emit(AuthStateLoggedIn(
```

```
          user: user,
          isLoading: false,
        ));
      }
    } on Exception catch (e) {
      emit(
        AuthStateLoggedOut(exception: e, isLoading: false),
      );
    }
  });
```

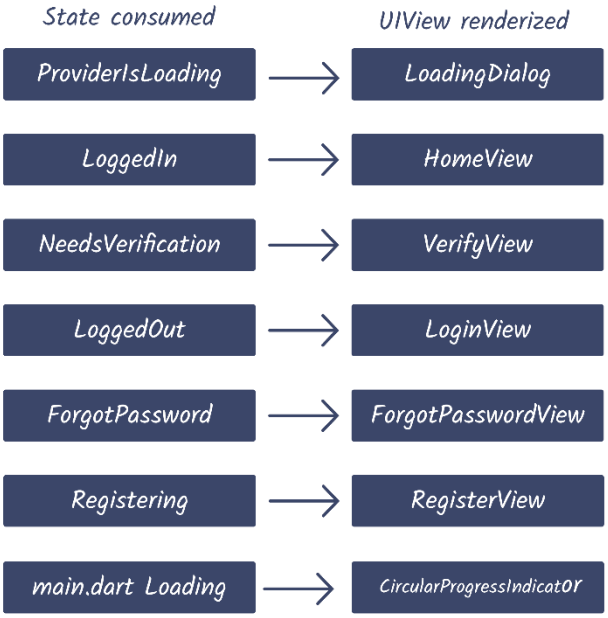
- LogOut
  Posibles estados:
  - El usuario quiere realizar un logout y no ha habido ninguna excepción.
  - El usuario quiere realizar un logout y ha ocurrido una excepción.

```
// log out
  on<AuthEventLogOut>((event, emit) async {
    try {
      await provider.logOut();
      emit(
        const AuthStateLoggedOut(exception: null, isLoading: false),
      );
    } on Exception catch (e) {
      emit(
        AuthStateLoggedOut(exception: e, isLoading: false),
      );
    }
  });
 }
}
```

Dados estos eventos y teniendo en cuenta los diferentes estados posibles, la clase *main.dart* renderizará sobre la UI las vistas adecuadas a la funcionalidad que espera este durante el proceso de autenticación, siguiendo el siguiente esquema:

edix

main.dart




```dart

@override
Widget build(BuildContext context) {
  context.read<AuthBloc>().add(const AuthEventInitialize());
  return BlocConsumer<AuthBloc, AuthState>(
    listener: (context, state) {
      if (state.isLoading) {
        LoadingScreen().show(
            context: context,
            text: state.loadingText ?? "Please wait a moment");
      } else {
        LoadingScreen().hide();
      }
    },
    builder: (context, state) {
      if (state is AuthStateLoggedIn) {
        return const HomeView();
      } else if (state is AuthStateNeedsVerification) {
        return const VerifyEmailView();
      } else if (state is AuthStateLoggedOut) {
        return const LoginView();
      } else if (state is AuthStateForgotPassword) {
        return const ForgotPasswordView();
      } else if (state is AuthStateRegistering) {
        return const RegisterView();
      } else {
        return const Scaffold(
          body: CircularProgressIndicator(),
        ); // Scaffold
      }
    },
  ); // BlocConsumer
}
```

Por último, se estableció una clase abstracta, ***auth_provider.dart,*** que integrase todos los métodos necesarios para realizar el proceso de autenticación, y se hizo que la clase ***firebase_auth_provider.dart*** implementase dichos métodos:

- Initialize();
- Get currentUser;
- LogIn(email, password);
- createUser(email, password);
- LogOut();
- sendEmailVerification();
- sendPasswordReset();

Una vez completado el proceso de autenticación, la aplicación pasará a la vista principal.

HomeView (*home_view.dart)*

Para la vista principal, utilizamos un Widget Scaffold (el cual renderiza un layout básico vacío), presentando dos hijos principales:

- **Body:** El cual se utiliza para renderizar los widgets propios de la vista de notas y del calendario.
- **NavigationBar:** El cual dispondrá en la parte inferior de una barra de navegación para poder seleccionar que vista cargaremos en el body.

Funcionalidad Tareas/Notas

La funcionalidad de las notas se presentará a través de tres archivos de vista principales:

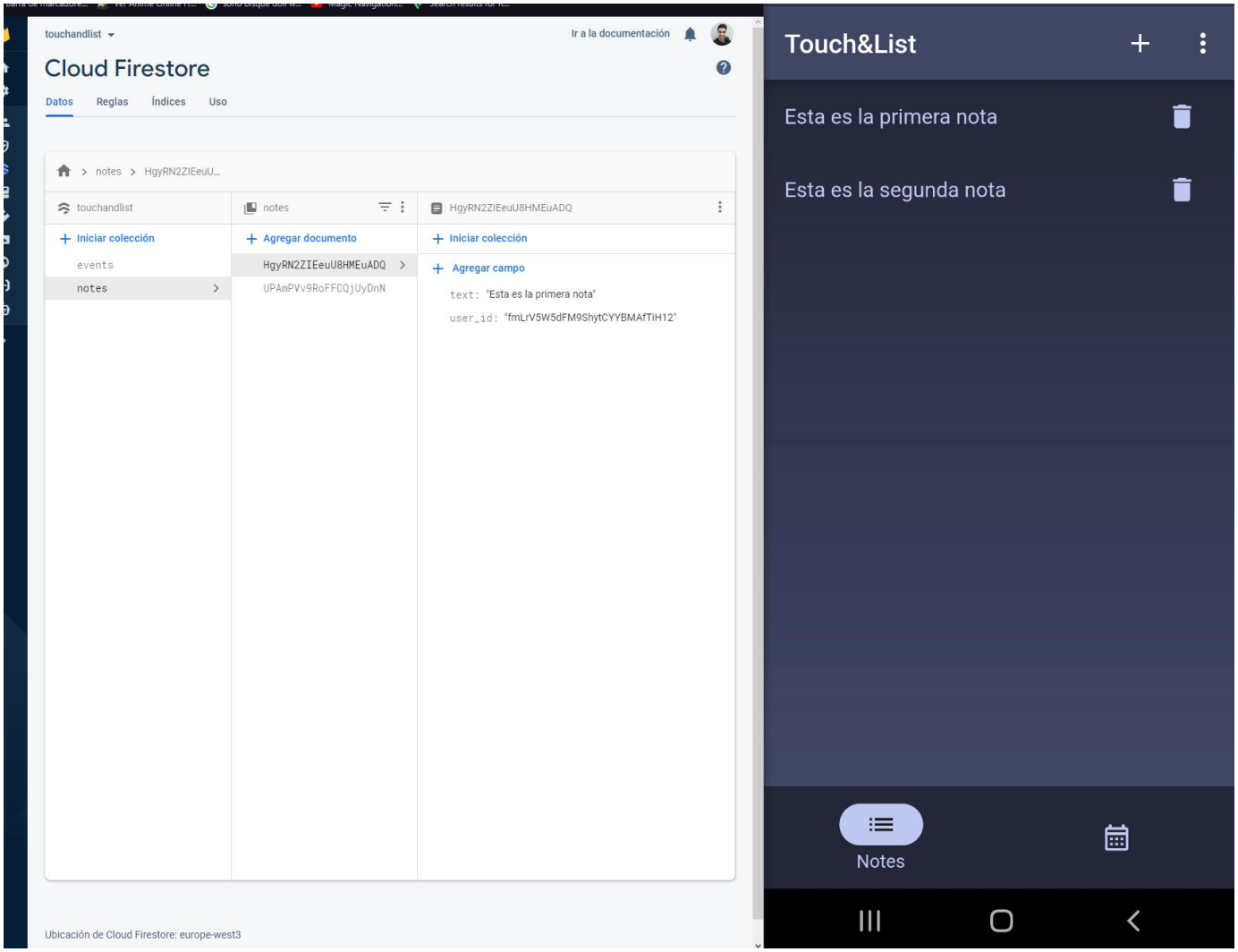
- **notes_view.dart:** Realizará una llamada a firestore donde se encuentran alojadas las notas, mientras el servidor se inicializa y genera una respuesta, presentará una barra circular de carga, una vez la carga haya concluido, renderizará a través de un StreamBuilder, el contenido de *notes_list_view.dart*

- **notes_list_view.dart:** Constará de un Widget ListView, el cual tendrá la longitud del total de las notas asociadas por id que el usuario tenga, por cada documento dentro de la colección notas, se renderizará un elemento. Y, en la parte superior, dispondrá un Widget appbar, que presentará un icono de ”+” y un menú desplegable. En caso de que la longitud sea 0, presentará un icono de una flecha, indicando como generar una nueva tarea/nota. En caso de presionar el icono “+”, pasará a la vista *create_update_note_view.dart y,* en el de presionar el menú desplegable, la funcionalidad de log out.

- **create_update_note_view.dart:** Presentará un Texfield sobre el que rellenar el contenido de la nota, en caso de presionar el botón de añadir, recogerá el texto a través de un textController y lo introducirá como un String dentro de la colección “**Notes”** de firestore.

Los métodos del CRUD relativo a la colección de notas, se encuentran dentro de la clase ***firebase_cloud_storage.dart*** y el modelo de datos de las notas en la clase ***cloud_note.dart,*** ambos en la carpeta ***cloud***.

A continuación, se adjunta una captura de pantalla de la consola de Firestore y los eventos asociados a un usuario:



Funcionalidad Calendario/Eventos

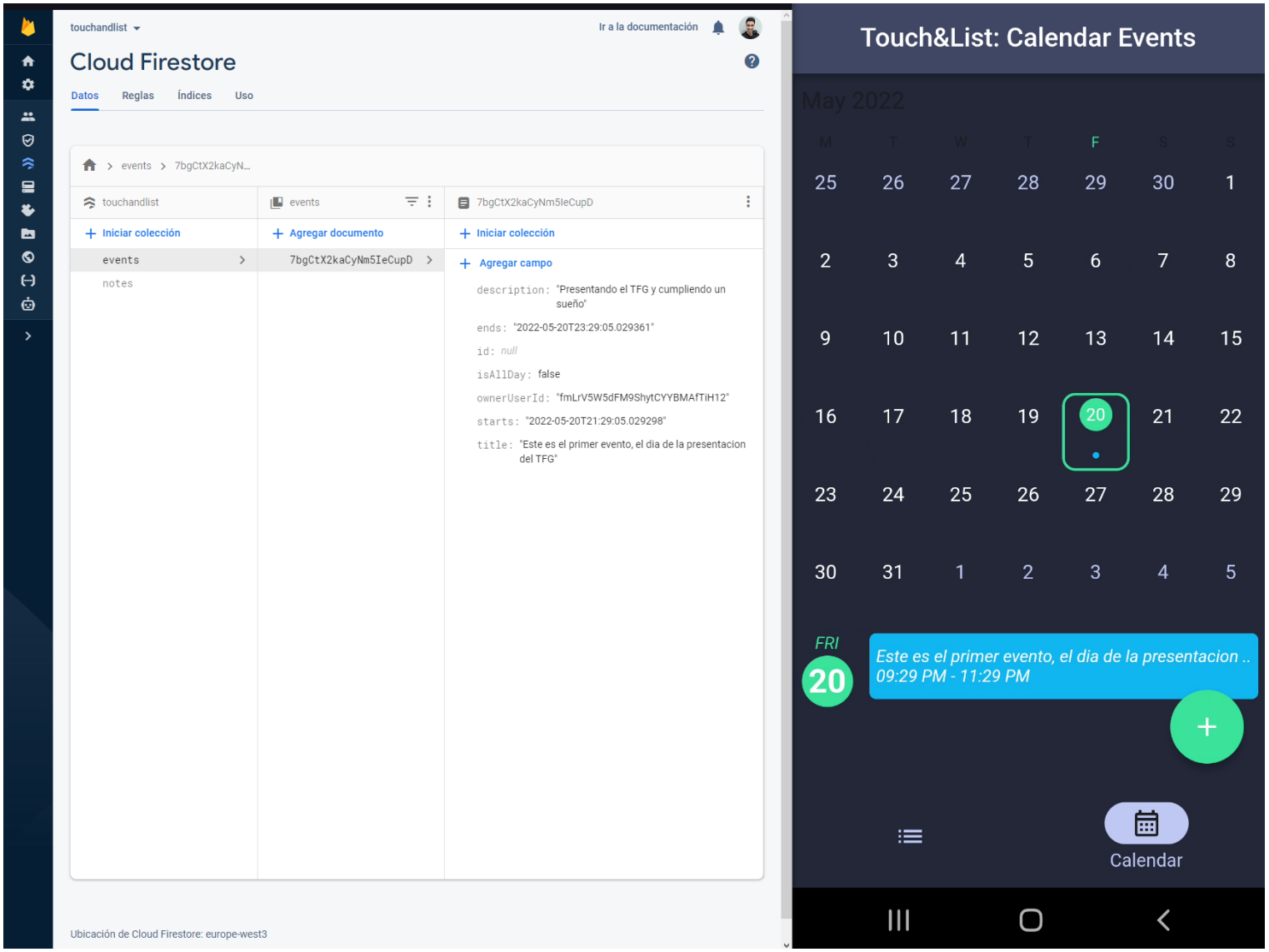
La funcionalidad de la gestión de eventos se presentará a través de tres vistas:

- **calendar_view.dart**: Esta vista constará de un AppBar, un body, que renderizará el CalendarWidget y un floating ActionButton, el cual, en caso de ser pulsado, cargará la vista de edición de eventos a través de la clase ***evento_editing_page.dart.***

- **calendar_editing_page.dart:** Esta vista presentará un formulario a través del cual añadir o editar un evento, para ello constará de dos TextFields donde establecer el título y descripción del evento, un checkbox para marcar la duración de un día completo y cuatro listas desplegables que renderizarán el DatePicker y TimePicker propios de la plataforma que se esté utilizando con la aplicación. En caso de validar el formulario, los datos de este rellenarán una lista de eventos dentro de la clase ***event_provider.dart*** y lo enviarán a la colección ***"Events"*** en firestore.

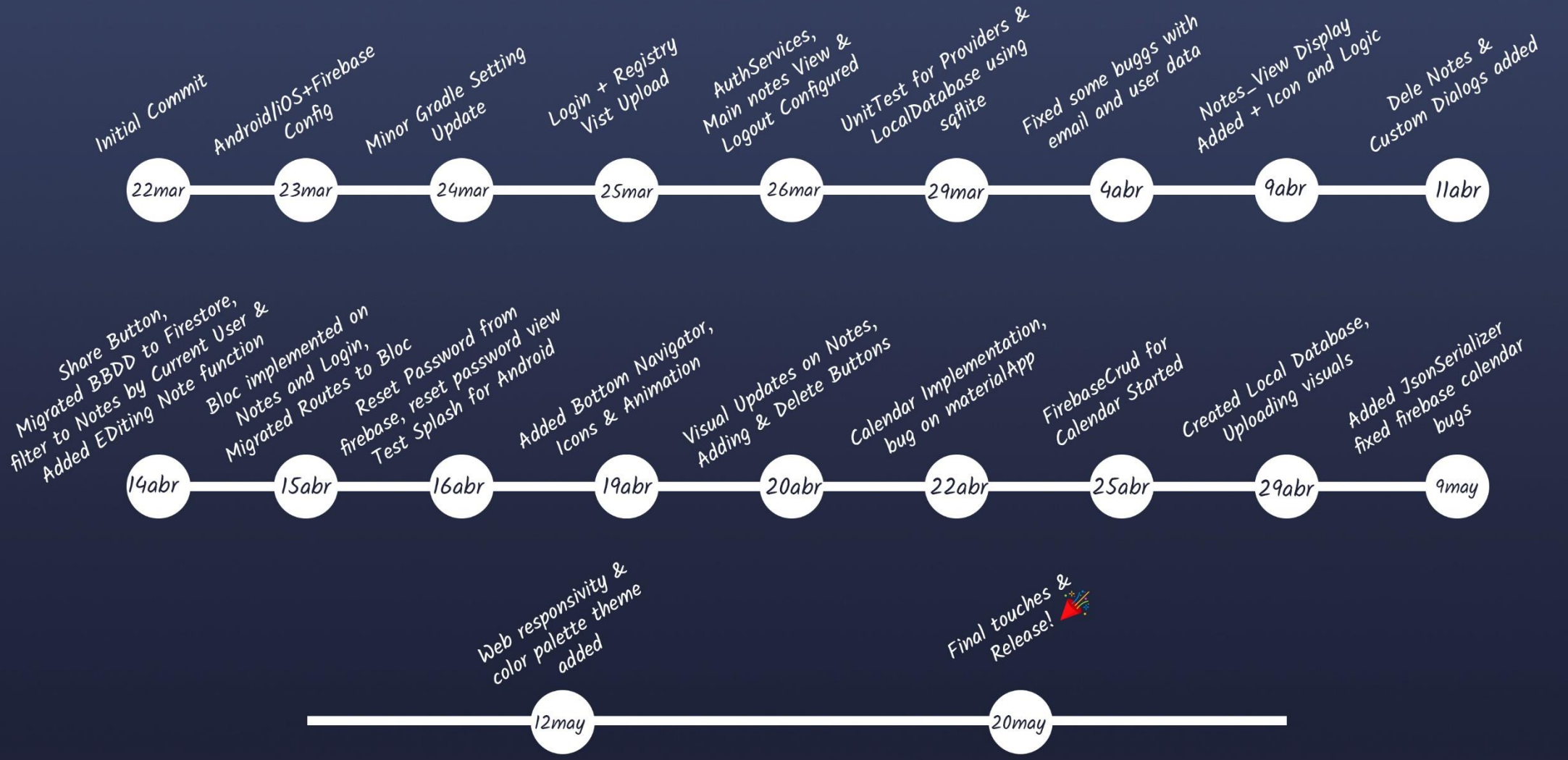
- **event_viewing_page.dart:** A través de esta vista renderizaremos los detalles de un evento previamente generado pudiendo eliminarlo o editarlo.

Los métodos del CRUD relativo a la colección "**events"** de firestore, así como los métodos de modificación de estado que informan al widget ***calendar_widget*** de la adicción o supresión de un evento, se encuentran en las clases ***event_provider.dart*** y ***evento_editing_page.dart*** y el modelo de datos de los eventos en la clase ***event.dart***



A continuación, se adjunta un TimeLine del desarrollo de nuestro proyecto:

Touch&List: TimeLine



Visual Studio Code

En cuanto al IDE utilizado para el desarrollo de la aplicación, se decidió que Visual Studio Code era el que más encajaba con las necesidades del proyecto, ya que ofrece una fácil integración con Flutter. Además de ser un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web, incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código. Sobre este IDE y para un desarrollo más eficiente integramos las siguientes extensiones:

- Bloc
- Dart
- Error Lens
- Flutter
- GitLens

Librerías

A continuación, se realiza una breve descripción de las librerías oficiales utilizadas durante nuestro desarrollo, todas ellas se encuentran alojadas en **https://pub.dev ,** pueden ser integradas en el proyecto realizando *un flutter pub add* seguido de la denominación de la librería, así como el comando *flutter pub get*, una vez introducidas en el archivo. yaml de configuración de dependencias:

- **Flutter**

- **cupertino_icons: ^1.0.2:** Repositorio que contiene el set por defecto de los iconos con el estilo de iOS.
- **firebase_core: ^1.13.1:** Plugin que permite el uso de Firebase Core API, lo que permite conectar la app con el servicio de Firebase.
- **firebase_auth: ^3.3.11:** Plugin que permite utilizar el servicio de autenticación de Firebase.
- **cloud_firestore: ^3.1.10:** Plugin que permite utilizar el servicio cloud de Firestore, una BBDD no relacional con actualización en tiempo real.
- **firebase_analytics: ^9.1.2:** Plugin que permite poner en contacto la aplicación con el servicio de google analytics a través de la herramienta de Firebase.
- **sqflite: ^2.0.2:** Plugin para la integración de SQLite en firebase.

- **path_provider: ^2.0.9:** Plugin para poder utilizar la localización de ficheros propios del sistema en Android, iOS, Linux, macOS y Windows 10+.
- **path: ^1.8.0:** Librería para implementar operaciones de manipulación de rutas, joining, splitting, normalizing…
- **share_plus: ^4.0.4:** plugin para mandar contenido desde la aplicación a través del diálogo de compartir propio del sistema utilizado. Utiliza ACTION_SEND Intent en Android y UIActivityViewController en iOS.
- **bloc: ^8.0.3:** Librería para la implementación del patrón de diseño Business Logic Component para la gestión de estados en flutter.
- **flutter_bloc: ^8.0.1:** Widget para la integración del patrón de diseño Business Logic Component para la gestión de estados en flutter.
- **equatable: ^2.0.3:** Librería que realiza un overrides =0 y hasCode para permitir la comparación de instancias de objetos de una forma más simple.
- **flutter_launcher_icons: ^0.9.2:** Herramienta de comando de línea que permite automatizar la actualización de los launcher icon, dentro de las plataformas de Android e iOS.
- **fluttericon: ^2.0.0:** Librería que permite integrar la utilización de los repositorios de iconos de fluttericon.com a través de código.
- **intl: ^0.17.0:** Librería que permite internacionalización y localización, incluida la traducción de mensajes, plurales, géneros y formateo de fechas y números de forma bidireccional. Pensada para integrar multilenguaje en la aplicación.
- **provider: ^6.0.2:** Librería que permite la comunicación entre Widgets en diferentes posiciones de jerarquía simplificando la localización, y dispose de recursos.
- **syncfusion_flutter_calendar: ^20.1.50:** Widget para la integración de gestión de eventos en nuestro calendario.
- **universal_html: ^2.0.8:** Librería que permite el desarrollo multiplataforma a través de html.
- **json_annotation: ^4.5.0:** Librería para la definición de anotaciones necesarias para implementar la automatización en la serialización y deserialización de JSON.

● **Dev Dependences:**

- **flutter_test:** Librería para la integración de Test dentro de Flutter.

  sdk: flutter

- **build_runner: ^2.0.0:** Librería para implementar la automatización en la serialización y deserialización de JSON.
- **json_serializable: ^6.2.0:** Librería para implementar la automatización en la serialización y deserialización de JSON.

PRUEBAS:

Para la realización de pruebas automatizadas, comprobación de errores y supervisión de casos de uso, el ecosistema de Flutter nos permite realizar tres tipos de Test de forma nativa:

- **Unit Test:** prueba de una única función, método o clase. El objetivo de una prueba unitaria es verificar la exactitud de una unidad lógica bajo una variedad de condiciones. Las dependencias externas de la unidad bajo prueba son generalmente mocked out. Dichas pruebas generalmente no leen ni escriben en disco, ni se renderizan en pantalla, ni reciben acciones del usuario desde fuera del proceso que ejecuta la prueba.
- **Widget Test:** (conocido en otros frameworks de UI como component test) prueba un solo widget. El objetivo de una prueba de widgets es verificar que la UI del widget se vea e interactúe como se espera. Implica múltiples clases y requiere un entorno de prueba que proporcione el contexto apropiado del ciclo de vida del widget.
- **Integration Test:** Consiste en una prueba de una app completa o una gran parte de ella. El objetivo de una prueba de integración es verificar que todos los widgets y servicios que se están probando funcionan juntos de la forma esperada. Estas pruebas son además utilizadas para verificar el rendimiento de tu aplicación.

Para poder lanzar dichos Test, en primer lugar, se debe añadir en el archivo .yaml la dependencia ***test***, dentro del epígrafe de *dev_dependencies,* además, dichos test deben estar alojados en la carpeta *test* del directorio raíz de la aplicación y denominar al archivo *nombrearchivotest_test.dart* para que pueda ser ejecutado de forma automática a través del ide, en nuestro caso, visual Studio. Existen dos formas de inicializar los test, pueden ser ejecutados a través de la propia interfaz del IDE, haciendo click en los símbolos de play que aparecen a la izquierda de cada uno de los test, o, a través del comando *flutter test*.

Con esto en cuenta, y debido a la naturaleza de nuestra aplicación, decidimos elaborar Test Unitarios para las funciones que podrían desencadenar en un fallo crítico dentro de nuestra aplicación, esto es, todas las funciones relacionadas con el proceso de autenticación y persistencia del usuario una vez realizado el Log-in, así como para comprobar, de forma automatizada, su funcionamiento si se decide migrar a otro proveedor para el servicio de autenticación. A través de nuestros test unitarios comprobamos lo siguiente:

- El proveedor del servicio de autenticación se inicia con el lanzamiento de la aplicación.
- No es posible realizar un LogOut si el servicio de autenticación no ha sido inicializado.
- El proveedor puede ser inicializado.
- El usuario debe ser nulo nada más realizar la inicialización del proveedor.
- El servicio de autenticación debe ser capaz de inicializarse en menos de dos segundos.
- Una vez introducidos los datos de registro el usuario debe ser capaz de recibir un correo de verificación.
- El proveedor debe ser capaz de crear un nuevo usuario.

A continuación, se adjunta el código relativo a los Test Unitarios:

```dart
import "package:test/test.dart";
import 'package:touchandlist/services/auth/auth_exceptions.dart';
import 'package:touchandlist/services/auth/auth_provider.dart';
import 'package:touchandlist/services/auth/auth_user.dart';

void main() {
  group("Mock Authentication", () {
    final provider = MockAuthProvider();
    test("Should not be initialized to begin with", () {
      expect(provider.isInitialized, false);
    });

    test("Can't log out if not initialized", () {
      expect(
        provider.logOut(),
        throwsA(const TypeMatcher<NotInitializedException>()),
      );
    });
```

```
  test("Should be able to be initialized", () async {
    await provider.initialize();
    expect(provider.isInitialized, true);
  });

  test("User should be null after initialization", () async {
    expect(provider.currentUser, null);
  });

  test(
    "Should be able to initialize in less than 2 seconds",
    () async {
      await provider.initialize();
      expect(provider.isInitialized, true);
    },
    timeout: const Timeout(Duration(seconds: 2)),
  );

  test("Create user should delegate to logIn function", () async {
    final badEmailUser = provider.createUser(
      email: "failtestemail@test.com",
      password: "anypassword",
    );

    expect(badEmailUser,
        throwsA(const TypeMatcher<UserNotFoundAuthException>()));

    final badPasswordUser = provider.createUser(
      email: "anyemail@test.com",
      password: "failtestpassword",
    );
    expect(badPasswordUser,
        throwsA(const TypeMatcher<WrongPasswordAuthException>()));

    final user = await provider.createUser(
        email: "passtest@gmail.com", password: "passtestpassword");
    expect(provider.currentUser, user);
    expect(user.isEmailVerified, false);
  });

  test("Login user should be able to get verified", () {
    provider.sendEmailVerification();
    final user = provider.currentUser;
    expect(user, isNotNull);
    expect(user!.isEmailVerified, true);
  });

  test("Should be able to log out and log in again", () async {
    await provider.logOut();
```

```
      await provider.logIn(
        email: "email",
        password: "password",
      );
      final user = provider.currentUser;
      expect(user, isNotNull);
    });
  });
}

class NotInitializedException implements Exception {}

class MockAuthProvider implements AuthProvider {
  AuthUser? _user;
  var _isInitialized = false;
  bool get isInitialized => _isInitialized;

  @override
  Future<AuthUser> createUser({
    required String email,
    required String password,
  }) async {
    if (!isInitialized) throw NotInitializedException();
    await Future.delayed(const Duration(seconds: 1));
    return logIn(
      email: email,
      password: password,
    );
  }

  @override
  AuthUser? get currentUser => _user;

  @override
  Future<void> initialize() async {
    await Future.delayed(const Duration(seconds: 1));
    _isInitialized = true;
  }

  @override
  Future<AuthUser> logIn({
    required String email,
    required String password,
  }) {
    if (!isInitialized) throw NotInitializedException();
    if (email == "failtestemail@test.com") throw
UserNotFoundAuthException();
    if (password == "failtestpassword") throw
WrongPasswordAuthException();
```

```
    const user = AuthUser(
      id: "my_id",
      isEmailVerified: false,
      email: 'testemail@test.com',
    );

    _user = user;
    return Future.value(user);
  }

  @override
  Future<void> logOut() async {
    if (!isInitialized) throw NotInitializedException();
    if (_user == null) throw UserNotFoundAuthException();
    await Future.delayed(const Duration(seconds: 1));
    _user = null;
  }

  @override
  Future<void> sendEmailVerification() async {
    if (!isInitialized) throw NotInitializedException();
    final user = _user;
    if (user == null) throw UserNotFoundAuthException();
    const newUser = AuthUser(
      id: "my_id",
      isEmailVerified: true,
      email: 'testemail@test.com',
    );
    _user = newUser;
  }

  @override
  Future<void> sendPasswordReset({required String toEmail}) {
    throw UnimplementedError();
  }
}
```

Por otro lado, nuestra aplicación recoge todas las excepciones posibles, informando a los usuarios a través de un Dialog de los siguientes casos:

Autenticación:

- Usuario no existe.
- Credenciales incorrectas.
- Contraseña demasiado débil (menos de 6 caracteres).

- El email ya está en uso.
- Formato de email no válido.
- El usuario no se encuentra logueado.

Notas/Tareas:

- No se ha podido crear la nota (fallo en el proceso Create de la BBDD de Firestore).
- No se han podido leer las notas (fallo en el proceso Get de la BBDD de Firestore).
- No se ha podido actualizar la nota (fallo en el proceso Update de la BBDD de Firestore).
- No se ha podido borrar la nota (fallo en el proceso Delete de la BBDD de Firestore).

Eventos:

- No se ha podido crear el evento(fallo en el proceso Create de la BBDD de Firestore).
- No se han podido leer los eventos (fallo en el proceso Get de la BBDD de Firestore).
- No se ha podido actualizar el evento(fallo en el proceso Update de la BBDD de Firestore).Pues
- No se ha podido borrar el evento (fallo en el proceso Delete de la BBDD de Firestore).

DOCUMENTACIÓN:

Durante la implementación del software se fue dejando constancia de todo el procedimiento, a través de comentarios dentro del propio código, para facilitar el mantenimiento y las futuras actualizaciones de la aplicación, ya que puede ser necesario que alguien sin conocimiento previo sobre la estructura de la aplicación tenga que modificar parte de la misma.

MANTENIMIENTO:

Tal y como indica su definición, el mantenimiento de software es la modificación de un producto de software después de la entrega, para corregir errores, mejorar el rendimiento,

u otros atributos. Por lo que en esta fase se ampliarían los puntos pendientes y algunos errores que se han producido en el desarrollo de la aplicación móvil.

Como puede ser:

- Activación de notificaciones y alarmas para los eventos.
- Añadir la opción de compartir los calendarios y listas a diferentes emails.
- Desarrollar la implementación en las plataformas de Windows y MacOs.
- Integrar el sistema de autenticación por gmail, facebook o teléfono móvil de firebase.

Además, al desplegar la aplicación en varias plataformas que se actualizan continuamente, sería necesario programar tareas de mantenimiento para estudiar nuevas versiones y adaptar el proyecto, evitando de este modo comportamientos erróneos de la aplicación en un futuro, desafortunadamente, esto ha ocurrido en la persistencia de eventos, debido a una reciente actualización del ecosistema de firestore y la integración de los datos a través del plugin SFCalendar, aun habiendo funcionado anteriormente, en el momento de entrega de este documento, este bug, aún no ha podido ser subsanado.

9. CONCLUSIONES

Como conclusión del proyecto, en términos generales, se ha conseguido completar los objetivos y requisitos mencionados en los capítulos. Se ha conseguido crear una aplicación multiplataforma, que permite organizar notas y eventos, con una interfaz intuitiva y sencilla.

A nivel personal, el equipo ha adquirido nuevos conocimientos, o ampliado muchos de ellos sobre todas las tecnologías y lenguajes recogidos a lo largo de la memoria. Además de haber aprendido a desarrollar un proyecto de principio a fin, pasando por todas sus fases, aplicando todos los conocimientos adquiridos durante estos dos años de formación profesional.

10. BIBLIOGRAFÍA

About Flutter Event Calendar widget | Syncfusion | Scheduler. (s. f.). Syncfusion. https://help.syncfusion.com/flutter/calendar/overview

Add Firebase to your Flutter app | Firebase Documentation. (s. f.). Firebase. https://firebase.google.com/docs/flutter/setup?platform=ios

Asynchronous programming: futures, async, await. (s. f.). Dart. https://dart.dev/codelabs/async-await

Asynchronous programming: Streams. (s. f.). Dart. https://dart.dev/tutorials/language/streams

bloc | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/bloc

build_runner | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/build_runner

cloud_firestore | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/cloud_firestore

Common Flutter errors. (s. f.). Flutter. https://docs.flutter.dev/testing/common-errors

Concurrency in. (s. f.). Dart. https://dart.dev/guides/language/concurrency

Configure your build |. (s. f.). Android Developers. https://developer.android.com/studio/build/index.html

Core libraries. (s. f.). Dart. https://dart.dev/guides/libraries

cupertino_icons | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/cupertino_icons

Dart SDK overview. (s. f.). Dart. https://dart.dev/tools/sdk

equatable | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/equatable

Firebase CLI reference | Firebase Documentation. (s. f.). Firebase. https://firebase.google.com/docs/cli

Firebase Documentation. (s. f.). Firebase. https://firebase.google.com/docs

firebase_analytics | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/firebase_analytics

firebase_auth | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/firebase_auth

firebase_core | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/firebase_core

Flutter documentation. (s. f.). Flutter. https://docs.flutter.dev

flutter_bloc | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/flutter_bloc

fluttericon | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/fluttericon

flutter_launcher_icons | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/flutter_launcher_icons

flutter_lints | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/flutter_lints

flutter_test library - Dart API. (s. f.). apiFlutter. https://api.flutter.dev/flutter/flutter_test/flutter_test-library.html

intl | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/intl

JavaScript SDK |. (s. f.). Firebase. https://firebase.google.com/docs/reference/node

json_annotation | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/json_annotation

json_serializable | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/json_serializable

path | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/path

path_provider | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/path_provider

Polo, A. (2021, 27 diciembre). *Flutter + Firestore + Clean Architecture + DI - Flutter España*. Medium. https://medium.com/flutter-espa%C3%B1a/flutter-firestore-clean-architecture-di-9b417ce94f1c

Polo, A. (2022a, enero 12). *Integration test with Flutter — Part One — - Flutter Community*. Medium. https://medium.com/flutter-community/test-integration-with-flutter-part-one-401008eab5c7

Polo, A. (2022b, enero 19). *Flutter bloc for beginners - Flutter Community*. Medium. https://medium.com/flutter-community/flutter-bloc-for-beginners-839e22adb9f5

provider | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/provider

share_plus | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/share_plus

sqflite | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/sqflite

Stack Overflow - Where Developers Learn, Share, & Build Careers. (s. f.). Stack Overflow. https://stackoverflow.com

Syncfusion. (s. f.-a). *How to add the appointments to Firestore Database using Flutter Calendar | Syncfusion KB*. https://www.syncfusion.com/kb/12616/how-to-add-the-appointments-to-firestore-database-using-flutter-calendar

Syncfusion. (s. f.-b). *How to customize the blackout dates in the Flutter event calendar (SfCalendar) ? | Syncfusion KB*. https://www.syncfusion.com/kb/12003/how-to-customize-the-blackout-dates-in-the-flutter-event-calendar-sfcalendar

Syncfusion. (s. f.-c). *How to perform the CRUD operations in Flutter Calendar using Firestore database | Syncfusion KB*. https://www.syncfusion.com/kb/12661/how-to-perform-the-crud-operations-in-flutter-calendar-using-firestore-database

Syncfusion. (s. f.-d). *How to work with the Firebase database and the Flutter Calendar for appointments | Syncfusion KB*. https://www.syncfusion.com/kb/12067/how-to-work-with-the-firebase-database-and-the-flutter-calendar-for-appointments

syncfusion_flutter_calendar | Flutter Package. (s. f.). Dart Packages. https://pub.dev/packages/syncfusion_flutter_calendar

Testing Flutter apps. (s. f.). Flutter. https://docs.flutter.dev/testing

uizard. (s. f.). Uizard. https://app.uizard.io/

Understanding null safety. (s. f.). Dart. https://dart.dev/null-safety/understanding-null-safety

universal_html | Dart Package. (s. f.). Dart Packages. https://pub.dev/packages/universal_html

Visual Studio Code. (s. f.). Flutter. https://docs.flutter.dev/development/tools/vs-code