UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Diseño de un sistema IoT de monitorización de humedad del suelo vía aplicación móvil

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Ramos Galindo, Víctor

Tutor/a: Martínez Zaldívar, Francisco José

CURSO ACADÉMICO: 2022/2023

Resumen

En esta memoria se expone el desarrollo de una aplicación móvil en Flutter para el monitoreo de humedad, complementada con un sistema de sensores controlados mediante un microcontrolador ESP32 y una base de datos en tiempo real de Firebase. El objetivo del proyecto es proporcionar una solución eficiente, accesible y económica para la adquisición y visualización de datos de humedad en diversos entornos.

La aplicación móvil se ha diseñado priorizando una interfaz intuitiva que facilite su utilización a los usuarios. Esta permite a los usuarios acceder a los datos de los sensores en tiempo real para facilitar la detección de problemas en áreas específicas y tomar decisiones rápidamente ante estos. En la aplicación se presentan gráficos interactivos e información para mejorar la comprensión en las variaciones en los niveles de los sensores a los largo del tiempo.

El sistema de sensores desarrollado con un ESP32, recopila datos de humedad en tiempo real y los transmite a la base de datos de Firebase, garantizando así su disponibilidad instantánea desde la aplicación móvil. El uso de Real Time Database de Firebase nos permite una sincronización aproblematica entre el dispositivo y la aplicación así como una gestión eficiente de los datos.

Resum

En esta memòria s'exposa el desenvolupament d'una aplicació mòbil en Flutter per al monitoratge d'humitat, complementada amb un sistema de sensors controlats mitjançant un microcontrolador ESP32 i una base de dades en temps real de Firebase. L'objectiu del projecte és proporcionar una solució eficient, accessible i econòmica per a l'adquisició i visualització de dades d'humitat en diversos entorns.

L'aplicació mòbil s'ha dissenyat prioritzant una interfície intuitiva que facilite la seua utilització pels usuaris. Aquesta permet als usuaris accedir a les dades dels sensors en temps real per a facilitar la detecció de problemes en àrees específiques i prendre decisions ràpidament davant d'aquests. En l'aplicació es presenten gràfics interactius i informació per a millorar la comprensió en les variacions en els nivells dels sensors al llarg del temps.

El sistema de sensors desenvolupat amb un ESP32 recopila dades d'humitat en temps real i les transmet a la base de dades de Firebase, garantint així la seua disponibilitat instantània des de l'aplicació mòbil. L'ús de Real Time Database de Firebase ens permet una sincronització sense problemes entre el dispositiu i l'aplicació, així com una gestió eficient de les dades.

Abstract

This report presents the development of a mobile application in Flutter for monitoring humidity, complemented with a sensor system controlled by an ESP32 microcontroller and a real-time Firebase database. The objective of the project is to provide an efficient, accessible, and cost-effective solution for acquiring and visualizing humidity data in various environments.

The mobile application has been designed prioritizing an intuitive interface that facilitates its usability for users. It allows users to access real-time sensor data to facilitate the detection of problems

in specific areas and make quick decisions in response. The application presents interactive graphs and information to improve understanding of variations in sensor levels over time.

The sensor system developed with an ESP32 collects real-time humidity data and transmits it to the Firebase database, ensuring its instant availability from the mobile application. The use of Firebase's Real-Time Database allows for seamless synchronization between the device and the application, as well as efficient data management.

A mi madre, a mi padre y a mi pareja por confiar en mi y apoyarme a lo largo de este camino.

Índice general

I Memoria

1. Introducción y objetivos **1**
1.1. Motivación . . . 1
1.2. Objetivos . . . 2
1.2.1. Objetivos del proyecto . . . 2
1.2.2. Objetivos a desarrollar . . . 2

2. Metodología de trabajo **3**
2.1. Proceso a nivel personal del proyecto . . . 3
2.2. Distribución en tareas . . . 4
2.3. Diagrama temporal . . . 4

3. Desarrollo del Proyecto **5**
3.1. IoT . . . 5
3.1.1. Sensores . . . 5
3.1.1.1. Calibración de los sensores . . . 6
3.1.2. ESP32 . . . 7
3.1.2.1. Código de interés . . . 8
3.1.3. Integración . . . 11
3.2. Base de datos y Autentificación . . . 13
3.2.1. Firebase Real Time Database . . . 14
3.2.2. Firebase Authentication . . . 15
3.3. Diseño de la aplicación . . . 16
3.3.1. Entorno de programación . . . 16
3.3.2. Framework . . . 17
3.3.3. Diseño y progreso . . . 18
3.3.3.1. Diseño y estructura de pantallas . . . 19
3.3.3.2. Login Page . . . 21
3.3.3.2.1. Interfaz . . . 21
3.3.3.2.2. Código de interés . . . 22
3.3.3.3. Home Page . . . 26
3.3.3.3.1. Conexión con Firebase . . . 26
3.3.3.3.2. Interfaz . . . 28
3.3.3.3.3. Código de interés . . . 29
3.3.3.3.4. Gráficas de datos . . . 33
3.3.3.4. Consultor IA . . . 33

3.3.3.4.1. Interfaz e interacción con el Consultor IA 33
3.3.3.4.2. API de OpenAI 34
3.3.3.4.3. Dart openai package 35
3.4. Materiales y presupuesto 36

4. Conclusiones y líneas futuras **37**
4.1. Conclusiones 37
4.2. Líneas futuras 38

Bibliografía **41**

II Anexos

A. Manual de usuario **45**
A.1. Introducción 47
A.1.1. Acerca de “Goty” 47
A.1.2. Requisitos del Sistema 47
A.2. Registro e Inicio de Sesión 47
A.2.1. Crear una cuenta 47
A.2.2. Iniciar sesión en la aplicación 47
A.2.3. Cerrar sesión en la aplicación 47
A.3. Pantalla Principal 48
A.3.1. Visualización de datos de los sensores 48
A.3.2. Estado de los sensores 48
A.3.3. Interacción con las gráficas 48
A.4. Consultas a la Inteligencia Artificial 48
A.5. Cómo Navegar 48

B. Código del Widget *humGraph()* **49**
B.1. **Integración del Widget:** 49
B.2. **Widget *humGraph()*:** 49
B.3. **Funciones y Widgets contenidos en *humGraph()*:** 51
B.3.1. Función *checkState()* 51
B.3.2. Función *selViewId()* 51
B.3.3. Función *selInterval()* 52
B.3.4. Widget *viewButton()* 52
B.3.5. Función *changeView()* 53

Índice de figuras

3.1. Tipos de sensores de humedad 5
3.2. Microcontrolador ESP32 CH9102X 8
3.3. Función *setup() del ESP32* 9
3.4. Condición de intervalo de tiempo 9
3.5. Reconexión WiFi en caso de desconexión 10
3.6. Lectura y mapeo de los valores de los sensores 10
3.7. Creación de los JSON dentro de la función *loop()* 10
3.8. Envío de datos a la base de datos de Firebase 11
3.9. Esquema eléctrico 12
3.10. Montaje del sistema integrado con el ESP32 12
3.11. Controlador ESP32 acabado y en su caja 13
3.12. Sensor de humedad acabado y con caja 13
3.13. Estructura de la Base de datos en Firebase 14
3.14. Reglas de acceso a la base de datos de Firebase 15
3.15. Usuarios registrados en la App 16
3.16. Creacción del proyecto FLutter 18
3.17. Plantillas de proyecto Flutter 18
3.18. Estructura del proyecto tras su generación 19
3.19. Esquema que sigue la estructura de la aplicación 20
3.20. Contenido de la carpeta “lib” del proyecto 20
3.21. Logo creado para la app Goty 21
3.22. Paleta de colores de la interfaz 21
3.23. Ambos estados de la “Login Page” 22
3.24. Clase *Auth* dentro de *auth.dart* 23
3.25. Función *handleSubmit()* dentro de *login_page.dart* 24
3.26. Funciones de inicio de sesión y registro respectivamente 24
3.27. Widget *TextFormField()* y sus propiedades 25
3.28. Clase auxiliar para comprobar autentificación 26
3.29. Guias de Firebase para añadir SDKs 26
3.30. Inicialización de firebase en *main.dart* 27
3.31. Interfaz desarrollada en *home_page.dart* 28
3.32. Mensaje de confirmación del cierre de sesión 29
3.33. Función *fetchDataFromFirebase()* 30
3.34. Función *processData()* 30
3.35. Función *filterData()* 31
3.36. Primer segmento de la función *changeView()* 32
3.37. Segundo segmento de la función *changeView()* 32
3.38. Mensaje de información al pulsar un dato 33

3.39. Interacción con el Consultor IA 34
3.40. Función *completionFun()* 35

Índice de tablas

2.1. Distribución temporal de las tareas 4

3.1. Valores de calibración de los sensores 7
3.2. Listado de materiales y presupuesto 36

Glosario

24/7 24 horas al día, 7 días a la semana.

3V3 Tensión de 3.3 voltios.

Android Sistema operativo móvil de Google.

appbar Barra de navegación superior de una aplicación.

Dart Lenguaje de programación orientado a objetos desarrollado por Google.

drawbar Componente gráfico interactivo que se puede arrastrar en la interfaz de una aplicación.

EPOCH Punto de referencia en tiempo, comúnmente en segundos desde 1970.

ESP32 Microcontrolador Wi-Fi y Bluetooth para IoT.

Firebase Plataforma de desarrollo de aplicaciones en la nube de Google.

Flutter Framework de desarrollo de aplicaciones multiplataforma.

Flux de soldadura Líquido o pasta que limpia y facilita la soldadura.

GND Conexión a tierra eléctrica.

iOS Sistema operativo móvil de Apple.

IoT Internet of Things.

JSON Formato de intercambio de datos web.

nube Infraestructura en línea para almacenamiento y procesamiento remoto.

prompt Texto proporcionado para obtener respuestas del modelo de chat de IA.

pull-up Resistencia que mantiene la señal en estado alto.

SDK Conjunto de herramientas para desarrollo de aplicaciones.

timestamp Marca de tiempo que indica cuándo ocurrió un evento.

tokens Unidades individuales de texto que el modelo de OpenAI procesa en su entrada y salida.

Widgets En Flutter, elementos que componen la aplicación movil.

zoom Aumento en la escala de visualización.

Parte I

Memoria

Capítulo 1

Introducción y objetivos

En el contexto actual de la agricultura moderna, la gestión de recursos eficiente y la obtención de datos precisos son fundamentales a la hora de optimizar la productividad y garantizar la sostenibilidad. La monitorización de los niveles de humedad del sustrato es un aspecto fundamental en el cultivo de diversas especies, ya que este parámetro influye directamente en el crecimiento y desarrollo de las plantas y en la toma de decisiones agronómicas.

En este contexto que se plantea, el presente proyecto se centra en el desarrollo de una aplicación móvil híbrida en Flutter, complementada con un sistema de sensores de humedad que se implementa mediante un microcontrolador ESP32 y una base de datos en tiempo real de Firebase, con la finalidad de hacer mas fácil, económico y accesible el monitoreo y análisis de datos de humedad en el sustrato agrícola.

El motivo que impulsa esta iniciativa nace de la necesidad de proporcionar a agricultores, horticultores y entusiastas del cultivo herramientas accesibles y fiables para recopilar información en tiempo real del entorno de cultivo, en este caso, de los niveles de humedad del suelo. La aplicación móvil brinda una interfaz amigable de fácil comprensión y uso, permitiendo que los usuarios visualicen información relevante, reciban notificaciones y tomen así decisiones en función de las necesidades específicas de cada cultivo.

Poniendo como objetivo la contribución al avance de la agricultura de precisión, con el presente proyecto se busca proporcionar una solución integral que incluye tecnología de vanguardia y sensores de humedad económicos y accesibles para todo el mundo, simplificando el proceso de adquisición de datos y aumentando la eficiencia a la hora de tomar decisiones en el campo agrícola. En los apartados siguientes se detallarán la motivación que impulsa este proyecto, los objetivos específicos que se pretenden alcanzar y la estructura que guiará el desarrollo del mismo.

1.1. Motivación

La inspiración detrás de este proyecto surge de una necesidad personal: el control del entorno de cultivos de interior en el hogar. En un mundo que cada vez se enfoca mas en la autosuficiencia alimentaria, surge la inquietud de querer gestionar las condiciones óptimas para un crecimiento saludable de las plantas, incluso en espacios interiores.

La ausencia de herramientas adecuadas, accesibles y sobretodo económicas, para monitorizar los

niveles de humedad del sustrato en tiempo real se convirtió en un desafío evidente. Se presentaron principalmente dos limitaciones claras, la necesidad de intervenir manualmente en el entorno de cultivo, perturbando así las condiciones de este, y las conjeturas que debía realizar sobre las condiciones de este entorno. Esta situación fomentó el deseo de querer desarrollar una solución tecnológica que me permitiera el control preciso y automatizado.

Gracias al cultivo de interior no solo podemos obtener alimentos mas frescos y saludables, sino que también podremos experimentar y aprender de manera práctica sobre horticultura y cultivo sostenible. Esta motivación personal se ha convertido pues en el catalizador para crear este sistema de monitorización en tiempo real que permita en cierto modo empoderar a los aficionados al cultivo en interiores con herramientas tecnológicas para fomentar el éxito y la prosperidad de sus plantas.

No obstante, siendo el cultivo de interior el aliciente que impulsa el proyecto, este no tiene la finalidad de quedarse solo ahí, la idea es que el sistema que se ha creado sea apto tanto para interiores como exteriores, tanto para aficionados como profesionales.

1.2. Objetivos

1.2.1. Objetivos del proyecto

El propósito final es ofrecer un acceso sencillo y asequible a datos cruciales en el sector de la agricultura para facilitar la toma de decisiones en entornos de cultivo. La finalidad es otorgar a los usuarios, independientemente de su experiencia o recursos económicos, la capacidad de controlar y gestionar los niveles de humedad del suelo en sus cultivos. De esta forma se busca facilitar una poderosa herramienta a los aficionados y profesionales de la jardinería y agricultura para que logren un desarrollo mas saludable y productivo de sus plantas.

1.2.2. Objetivos a desarrollar

El principal objetivo de este proyecto es desarrollar un sistema completo de control de humedad tanto la parte del Internet de las Cosas (IoT) como la parte de desarrollo de la aplicación móvil correspondiente. La parte de IoT constará de un microcontrolador ESP32 en el que se integran cuatro sensores de humedad capacitivos de los cuales se detectará su conexión y desconexión. El ESP32 asumirá la responsabilidad de recopilar datos provenientes de los sensores, procesarlos y posteriormente los transferirá a una base de datos en tiempo real en Firebase. Por otro lado, la aplicación será tanto para Android como para iOS. Aprovechando la potencia de la tecnología de Flutter, esta aplicación presentará una interfaz de usuario amigable y uniforme para ambas plataformas. Gracias a este framework no es necesario realizar un desarrollo independiente para cada sistema operativo ya que con un único desarrollo obtendremos ambos productos optimizando recursos y reduciendo el tiempo de desarrollo. La aplicación deberá contar con un sistema de gestión de usuarios con una interfaz de inicio de sesión y registro. A su vez, tras iniciar sesión, presentará una gráfica por cada sensor en la cual se representarán los valores del sensor correspondiente a lo largo del tiempo. También se incluirán indicadores de conexión y desconexión de sensores. Por último se integrará una pantalla de consultas que se realizarán a una inteligencia artificial.

Capítulo 2

Metodología de trabajo

2.1. Proceso a nivel personal del proyecto

Como se ha comentado en la sección 1.1, el proyecto surgió a partir de una necesidad personal y, a partir de eso, comenzó de lleno la investigación de soluciones ante esta necesidad. Puesto que en el mercado no encontramos ninguna opción asequible que cumpliera con los requisitos, se decidió crearla por nosotros mismos.

Lo primero fue investigar qué partes tendría este proyecto ya que abarca varios campos y a la hora de comenzar con el desarrollo se requiere de formación previa en dichos campos. A continuación se comenzó con la formación en IoT y a investigar qué componentes se debían escoger que cumplieran con los requisitos del proyecto y que fueran lo mas económicos posible. Una vez se tuvieron todos los componentes se pasó a realizar pruebas con ellos, entender su funcionamiento y averiguar cual sería la forma óptima de integrar todos los componentes.

En la siguiente fase se realizó un modelo de pruebas con el que ir ajustando los fallos y necesidades del conjunto IoT e incluso se llevó a cabo una integración con una aplicación móvil para IoT llamada "Blynk" para comprobar que el sistema de pruebas funcionaba correctamente de cara a la integración final del proyecto con la aplicación móvil. Obviamente, esta aplicación no sirve para el proyecto puesto que muchas de sus características son de pago, por lo que debíamos desarrollar una aplicación propia.

En este punto se comenzó a investigar sobre Firebase y las herramientas que este ofrece y se decidió utilizar *Real Time Database* para ir almacenando la información de los sensores. Tras un poco de trabajo se consiguió que el ESP se conectara a Firebase y comenzase a mandar todos los datos con un formato que posteriormente facilitaría el uso de estos en el desarrollo de la aplicación. Una vez llegado hasta aquí, se decidió que era momento de comenzar con la aplicación móvil.

Como se estaba iniciando una nueva fase del proyecto, tocó volver a la tarea de formación. Se empezó a investigar sobre el desarrollo de aplicaciones multiplataforma y nos topamos con Flutter. Previamente se había trabajado con otros frameworks como *ionic* pero era la primera vez que se trabajaba con Flutter, por lo que hubo que aprender desde cero su funcionamiento así como el lenguaje de programación. Tras realizar varios cursos online, se procedió a desarrollar la aplica-

ción y esta fue la etapa mas larga, ya que ha sido donde mas problemas han surgido durante el desarrollo. A la vez que se desarrollaba la aplicación se continuaron haciendo retoques en la parte de IoT y se hizo el montaje final del producto físico. Una vez se tuvo todo solo hubo que realizar pequeños ajustes y el proyecto estaba finalizado.

2.2. Distribución en tareas

El proyecto se ha dividido en las siguientes fases o tareas:

I - **Investigación.** Fase de estudio y formación en la que se adquieren las habilidades y conocimientos necesarios para el buen desarrollo del proyecto.

II - **Desarrollo IoT.** Fase dedicada a la parte IoT del proyecto la cual engloba desde la elección de componentes, programación e integración de estos hasta el montaje final del producto físico.

III - **Implementación de Firebase.** Fase en la cual se crea y configura el proyecto de Firebase así como en la que se implementa tanto en la parte de IoT como en la aplicación móvil.

IV - **Desarrollo de la aplicación móvil.** Última fase del proyecto en la que discurre todo el desarrollo de la aplicación móvil.

2.3. Diagrama temporal

Aquí se presenta la Tabla 2.1 en la que podemos apreciar cómo se han distribuido las distintas tareas a lo largo del tiempo. Como se puede apreciar hay tareas que se han realizado de forma simultánea a otras.

	MAYO	JUNIO	JULIO	AGOSTO
Investigación	X		X	
Desarrollo IoT		X		X
Implementación de Firebase		X	X	
Desarrollo de la aplicación móvil			X	X

Tabla 2.1: Distribución temporal de las tareas

Capítulo 3

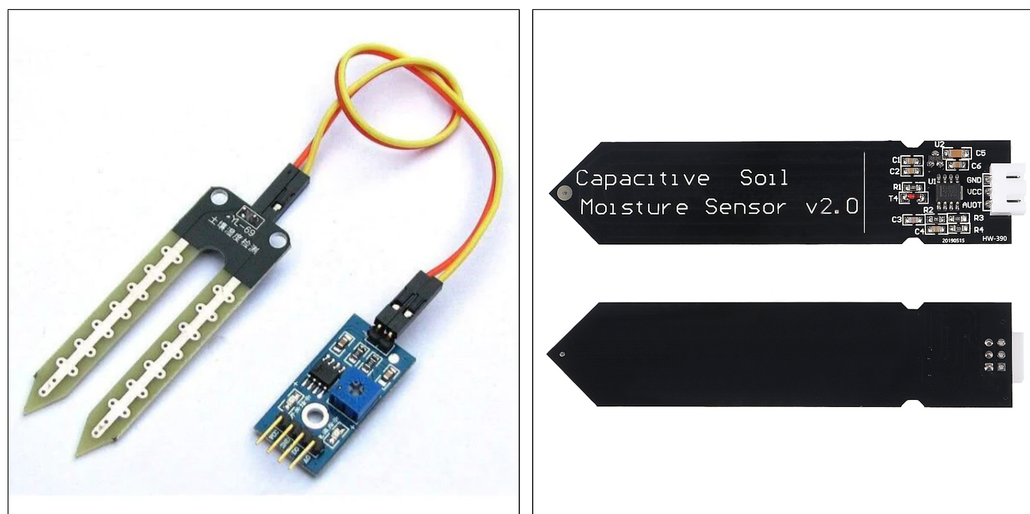
Desarrollo del Proyecto

3.1. IoT

En este apartado vamos a tratar todo lo relacionado a la parte de IoT. Trataremos por separado los componentes que lo forman así como la integración que se ha llevado acabo con ellos.

3.1.1. Sensores

A la hora de obtener información del entorno para posteriormente poderla procesar, es necesario el uso de sensores. En nuestro caso, queremos obtener información sobre el nivel de humedad del suelo y para esto existen principalmente dos tipos de sensores, resistivos y capacitivos (Figura 3.1).



(a) Sensor de humedad resistivo FC-28 **(b) Sensor de humedad capacitivo HW-390**

Figura 3.1: Tipos de sensores de humedad

Un sensor resistivo, como su nombre indica, se basa en la propiedad de resistencia de los materia-

les y en nuestro caso, del suelo. Está compuesto de dos sondas con un recubrimiento de un metal conductor llamadas electrodos (apreciables en la Figura 3.1a) los cuales miden la resistencia que ofrece el suelo. Este factor viene dado por la cantidad de sales disueltas, a mayor cantidad de agua, mayor será la cantidad de sales disueltas por lo que se generará una resistencia mayor. Y a menor cantidad de agua sucederá lo contrario y tendremos una resistencia menor. Este tipo de sensores pueden variar su comportamiento con factores como la temperatura, resistividad o el área del material resistivo.

Por otro lado tenemos los sensores capacitivos, que de nuevo, como su nombre indica, se basan en una propiedad llamada capacitancia. Esta propiedad se puede definir coloquialmente como la reacción que se produce en dos placas de materiales conductores que se encuentran separados entre sí a muy poca distancia al aplicarles una carga eléctrica. Al aplicar un voltaje alterno, los electrones se mueven de una placa a otra creando una carga positiva en una placa y una carga negativa en otra, generando así una corriente eléctrica y es esta corriente la que mide el sensor.
La capacitancia se ve influida principalmente por tres factores, el área y distancia de las placas y el dieléctrico, que es el material que se encuentra entre las placas conductoras. El dieléctrico por lo tanto será el suelo y el agua que este contenga.

Ahora bien, a la hora de elegir un sensor u otro, los sensores resistivos presentan una clara desventaja ya que al tener los componentes metálicos de sus electrodos expuestos y, debido al paso del tiempo, estos acaban oxidándose modificando así la conductividad de los mismos y haciendo que las medidas que toman se vuelvan totalmente inexactas.
Es debido a esto por lo que me decanté por utilizar los sensores capacitivos HW-390 (Figura 3.1b) para el proyecto al no tener ningún material expuesto que pudiera oxidarse. Otro factor determinante a la hora de elegir este tipo de sensores ha sido que, a diferencia de los resistivos, estos no hacen circular una carga eléctrica a través del suelo la cual podría afectar negativamente a las raíces de nuestras plantas y cultivos. Además, debido a que la tierra al secarse puede que se contraiga, esto haría que los electrodos de los sensores resistivos no hicieran buen contacto y de nuevo nos proporcionasen medidas inexactas.

Cabe recordar en este punto que el objetivo del proyecto es mantener el presupuesto lo mas bajo posible para que sea accesible para todo el mundo y es debido a esto que solo se presentan estos dos sensores al ser los más económicos del mercado. Si es verdad que existen sensores de mas alta gama, incluso resistivos con recubrimiento (como por ejemplo el sensor SM150T [1]), que se consideran de uso industrial y que resultan mas precisos pero de nuevo estos se escapan del propósito de este proyecto.

3.1.1.1. Calibración de los sensores

Como bien sabemos, todos los sensores requieren de una calibración para poder funcionar correctamente en el ámbito al que van a ser destinados.
El proceso de calibración ha consistido en tomar medidas con el sensor en el sustrato seco y húmedo y mapear esos valores que nos ofrece el sensor en un rango del 0 al 100 para transformarlo en un porcentaje de humedad.

SENSOR	VALORES	
	SECO	HUMEDO
0	2720	1880
1	2730	1870
2	2720	1870
3	2700	1870

Tabla 3.1: Valores de calibración de los sensores

Es de esperar, que la calibración de los sensores dependa del tipo de suelo, ya que cada suelo tiene unas propiedades distintas y hará que nuestro sensor arroje valores distintos. Debido a esto, se debe realizar una calibración del sensor sobre el suelo en el que vaya a ser utilizado previamente a la implementación del sistema de control de humedad.

3.1.2. ESP32

A la hora de escoger un controlador de entre todas los posibilidades del mercado se tomó la decisión de utilizar un ESP32 como núcleo central de la solución IoT. El ESP32 destacó por su capacidad intrgral para abordar los requisitos del proyecto ya que ofrece un conjunto de características esenciales.

Este dispositivo es una solución completa ya que integra una variedad de funcionalidades en un solo dispositivo. Uno de sus aspectos clave es su capacidad para aceptar múltiples entradas analógicas, que son fundamentales para la adquisición de datos de nuestros sensores de humedad. Esta característica nos va a permitir adquirir datos de forma precisa para realizar un control eficaz del cultivo. El hecho de que el dispositivo porte ya de por sí un puerto de alimentación integrado en forma de puerto micro USB, constituye un elemento conveniente que facilita la conectividad y la administración de energía así como la gestión del mismo. Esto facilita la interacción con el dispositivo permitiendo su conexión a fuentes de alimentación convencionales. Asimismo, el ESP32 integra una antena WiFi brindándonos conexión inalámbrica, necesaria para realizar la transmisión de datos a nuestra base de datos en tiempo real en Firebase.

En la Figura 3.2 podemos apreciar visualmente las características que acabamos de mencionar. En la parte izquierda se encuentra el puerto de alimentación y a la derecha del todo en forma de rectángulo negro, la antena WiFi.

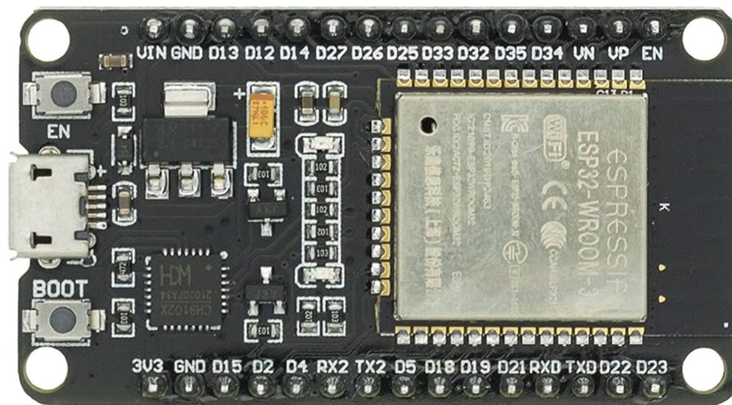


Figura 3.2: Microcontrolador ESP32 CH9102X

Concretamente, el que se erige como el cerebro de todo el sistema de IoTes el microcontrolador ESP32 CH9102X. La elección de este modelo entre todas las variantes del ESP32 se ha basado en una consideración equilibrada entre la capacidad de procesamiento que presenta, su costo y su perfecta adecuación a los requisitos del proyecto. Es cierto que no es el chip mas potente que puede integrar el ESP32 pero posee potencia suficiente para realizar las tareas encomendadas y es aquel con el costo mas atractivo. Esto nos asegura que sea una opción económica siguiendo así con los propósitos del proyecto sin comprometer la capacidad de procesamiento necesaria para gestionar los datos de humedad y llevar a cabo la conexión con Firebase.

3.1.2.1. Código de interés

En esta sección vamos a comentar la programación que se ha llevado a cabo sobre el ESP32 la cual se realiza en C#. Para programarlo hemos hecho uso de un cable de datos micro-USB/USB para conectar el dispositivo a un PC y del software de desarrollo *Arduino IDE*.

A continuación pasaremos a comentar aquellas partes del código que resultan mas relevantes para comprender el funcionamiento que se ha establecido para este dispositivo.
A la hora de programar estos dispositivos, existen dos funciones fundamentales, *setup()* donde se realizan las configuraciones iniciales, y *loop()* que es donde se programan las acciones que se repiten en bucle que llevará a cabo el dispositivo. En la Figura 3.3 podemos observar la primera función que acabamos de mencionar

```
void setup() {
  // Start the serial communication
  Serial.begin(115200);
  Serial.println();

  // Connect to WiFi
  WiFi.begin(SSID, WIFI_PASSWORD);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  configTime(0, 0, ntpServer);
```

```
  // Firebase init
  Firebase.begin(FIREBASE_HOST, FIEREBASE_AUTH);
  Firebase.reconnectWiFi(true);

  // Check Firebase connection
  if (!Firebase.ready()) {
    Serial.println("Error connecting to Firebase");
    Serial.println(firebaseData.errorReason());
    while (!Firebase.ready()){
      Serial.println("Retrying to connect to Firebase...");
      Firebase.begin(FIREBASE_HOST, FIEREBASE_AUTH);
      Firebase.reconnectWiFi(true);
    };
  }
  Serial.println("Connection to Firebase established.");
}
```

Figura 3.3: Función *setup() del ESP32*

Lo primero que se hace en esta función es iniciar la comunicación serial para que nosotros desde el sofware de *Arduino IDE* podamos recibir información desde el dispositivo en la consola serial. Después se realiza la conexión a la red WiFi y hasta que esta no sea efectiva no se procederá al siguiente paso. A continuación inicializamos la configuración de Firebase para posteriormente poder comunicarnos con la base de datos. Por último se comprueba que la conexión con Firebase funcione correctamente, de no ser así se procederá a la reconexión con el servicio de Firebase hasta que funcione correctamente.

Ahora pasaremos a comentar las secciones de interés de la función *loop()*. Al comienzo de esta encontramos que todo su contenido está envuelto por una condición "if" como se aprecia en la Figura 3.4 la cual sirve para que la ejecución del código se realice cada cierto intervalo de tiempo en función de la variable global "interval", en nuestro caso decidimos que la información se envíe cada minuto. Gracias a esto evitamos saturar de datos tanto el servidor de Firebase como la aplicación móvil y reducimos la necesidad de cómputo del ESP32. A fin de cuestas esto actúa como un temporizador que podemos ajustar fácilmente cambiando la variable "interval".

```
void loop() {

  // This block executes every "interval"
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
```

Figura 3.4: Condición de intervalo de tiempo

Justo después de dicha condición, lo primero que ejecuta el bucle *loop()* es una comprobación de la conexión WiFi. De esta forma nos aseguramos que siempre esté conectado y de no ser así se reconecte como aparece en la última línea de código de la Figura 3.5.

```
// WiFi reconnection if lost
if ((WiFi.status() != WL_CONNECTED)) {
  Serial.print(millis());
  Serial.println("Reconnecting to WiFi...");
  WiFi.disconnect();
  WiFi.reconnect();
}
```

Figura 3.5: Reconexión WiFi en caso de desconexión

Tras esta comprobación se procede a leer los datos de los sensores de humedad y, gracias a la calibración realizada, se mapean los valores obtenidos en un rango de 0 a 100 con el valor correspondiente en porcentaje de humedad (Figura 3.6).

```
// Read and map the sensor values:          (A, B, 100%, 0%)
sensorValue[0] = map(analogRead(sensorPin1), 1870,2808,100,0);
sensorValue[1] = map(analogRead(sensorPin2), 1870,2808,100,0);
sensorValue[2] = map(analogRead(sensorPin3), 1870,2808,100,0);
sensorValue[3] = map(analogRead(sensorPin4), 1870,2808,100,0);
```

Figura 3.6: Lectura y mapeo de los valores de los sensores

A continuación se genera el "timestamp" en EPOCH mediante la función *getTime()* de la Figura 3.7b para posteriormente asociar esta marca temporal a cada valor de humedad y añadirlos a un JSON (Figura 3.7a). Este JSON, el cual se genera uno por sensor, será lo que posteriormente se envíe a la *Real Time Database*.

```
//-----------------------------------------------------
// Create sensor value + timestamp JSON

timestamp = getTime();
//Serial.println(timestamp);
json0.set(humPath.c_str(), String(sensorValueJson[0]));
json0.set(timePath, String(timestamp));

json1.set(humPath.c_str(), String(sensorValueJson[1]));
json1.set(timePath, String(timestamp));

json2.set(humPath.c_str(), String(sensorValueJson[2]));
json2.set(timePath, String(timestamp));

json3.set(humPath.c_str(), String(sensorValueJson[3]));
json3.set(timePath, String(timestamp));

//-----------------------------------------------------
```

(a) Función *loop()*

```
// Function that gets current epoch time
unsigned long getTime() {
  time_t now;
  struct tm timeinfo;
  if (!getLocalTime(&timeinfo)) {
    Serial.println("Failed to obtain time");
    return(0);
  }
  time(&now);
  Serial.println("EPOCH time:");
  Serial.println(now);
  return now;
}
```

(b) Función *getTime()*

Figura 3.7: Creación de los JSON dentro de la función *loop()*

Por último se realiza en envío de la información a Firebase, pero no sin antes comprobar que los sensores estén conectados, ya que dependiendo de esto, se enviará una información u otra. En el caso de no estarlo, únicamente se envía el estado del sensor a la base de datos para que aparezca como desconectado. En el caso de si estar conectado, además de enviar el estado del sensor, se envía el JSON para que se genere un nuevo nodo de información en la base de datos con el valor de humedad y el timestamp. De la estructura de la base de datos se hablará mas adelante en el apartado 3.2.1. Todo este proceso lo podemos apreciar en la Figura 3.8, en la que aparece el proceso

de enviar los datos a Firebase para un solo sensor, esto se repetiría para cada uno de los sensores conectados al ESP32.

```
// Send Data to Firebase
// Sensor_0:
if(sensorValue[0]<-60){ // Disconnected
  if(desc[0] == true){
    desc[0] = false;
    connect[0] = true;
    Serial.println("sensor_0_disconnected");

    if (Firebase.setBool(firebaseData, nodoRaiz + "/sensor0/status", false)) {
    Serial.println("Data of sensor_0 sent correctly.");
  } else {
    Serial.println("Error while sending sensor_0 data.");
    Serial.println(firebaseData.errorReason());
  }
  }
}else{  // Connected
  if(connect[0]==true){
    Serial.println("sensor_0_connected");
    connect[0] = false;
  }
  desc[0] = true;

  if (Firebase.RTDB.pushJSON(&firebaseData, nodo0.c_str(), &json0)\
   && Firebase.setBool(firebaseData, nodoRaiz + "/sensor0/status", true)) {
    Serial.println("Data of sensor_0 sent correctly.");
  } else {
    Serial.println("Error while sending sensor_0 data.");
    Serial.println(firebaseData.errorReason());
  }
}
```

Figura 3.8: Envío de datos a la base de datos de Firebase

3.1.3. Integración

La integración del conjunto del sistema resulta sencilla, ya que el único añadido a los componentes ya mencionados en los apartados anteriores es un conjunto de resistencias. Se trata de cuatro resistencias de pull-up de 1 MΩ cada una. Estas van situadas entre la alimentación de 3V3 y los pines analógicos de cada sensor siguiendo el esquema de conexión de la Figura 3.9. La alimentación es de 3V3 en vez de de 5V (el ESP32 es capaz de proporcionar ambas) debido a que los sensores operan mejor para este valor de tensión. La función que desempeñan estas resistencias es la de arrojar valores de lectura al ESP32 incluso cuando no están conectados los sensores. Esto nos sirve para, mediante software, poder detectar la conexión y desconexión de nuestros sensores a partir de detectar los valores generados en los pines analógicos por las resistencias.

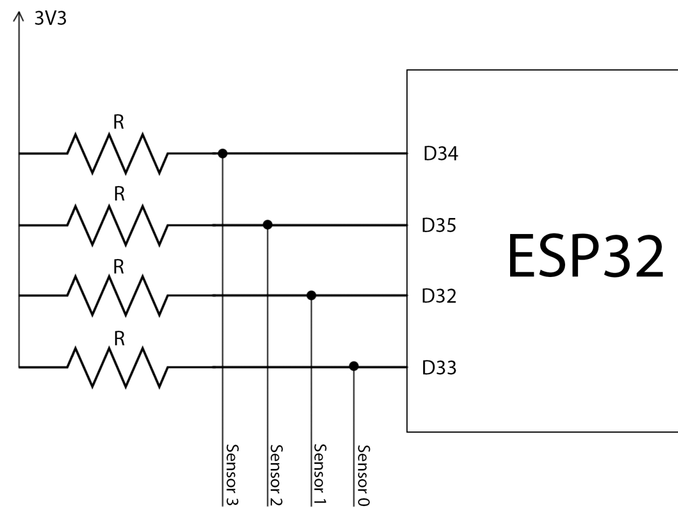


Figura 3.9: Esquema eléctrico

El montaje se comenzó soldando al ESP32 las resistencias con soldaduras de estaño, ayudandonos de un estañador y de Flux de soldadura, siguiendo el esquema de conexiones ya mencionado y posteriormente añadiendo el conjunto de cables necesarios para conectar los sensores, es decir, se añaden cuatro cables a GND y cuatro cables a 3V3 además de los cuatro cables de las entradas analógicas correspondientes a cada uno de los sensores como se aprecia en la Figura 3.10.

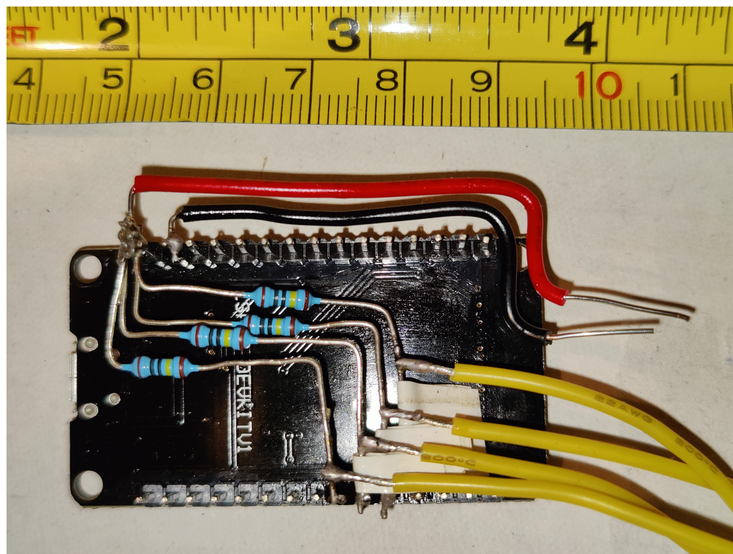
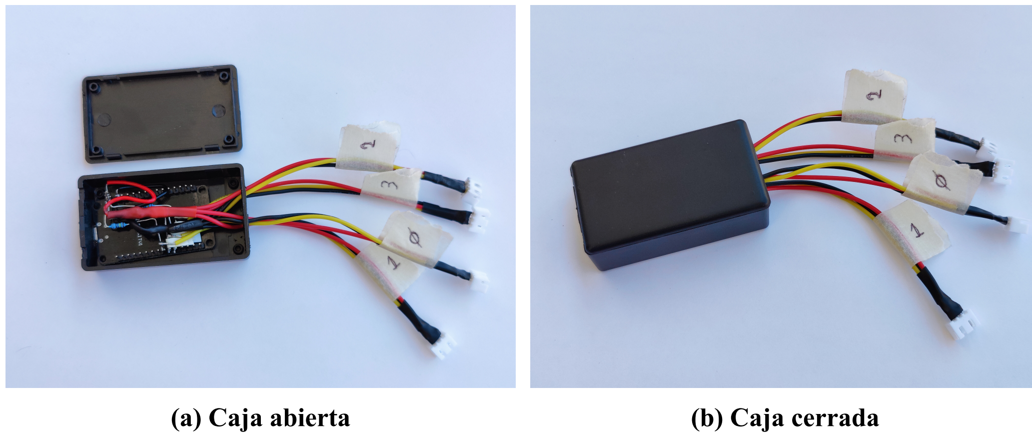


Figura 3.10: Montaje del sistema integrado con el ESP32

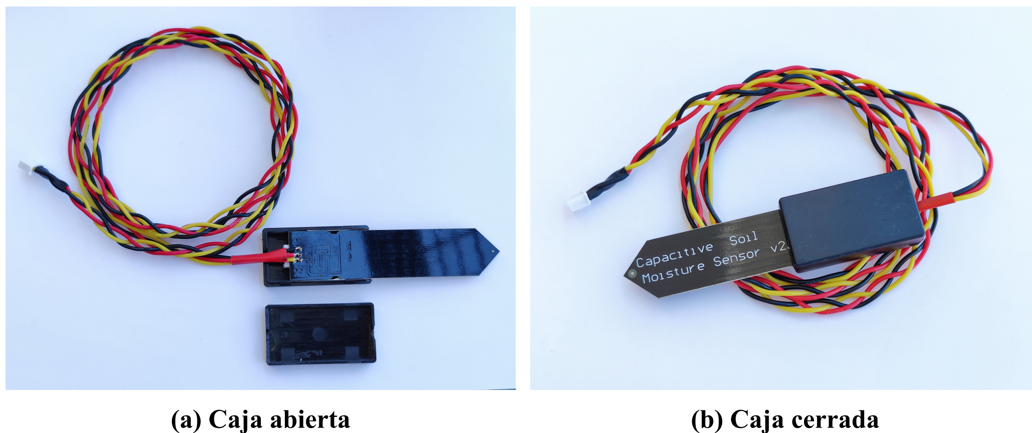
Tras esto se procedió a introducir el ESP32 en una caja de su medida, lo mas ajustada posible para mantener el tamaño que ocupa al mínimo posible y seguidamente se instalaron los conectores hembra de tres pines (Figura 3.11).



(a) Caja abierta **(b) Caja cerrada**

Figura 3.11: Controlador ESP32 acabado y en su caja

En cuanto a la parte de los sensores, a estos se les ha fabricado una carcasa para proteger la parte del circuito y se les han soldado tres cables a cada uno de sus pines que posteriormente se han trenzado para mantener la consistencia de los mismos (Figura 3.12).



(a) Caja abierta **(b) Caja cerrada**

Figura 3.12: Sensor de humedad acabado y con caja

Por último se han añadido los conectores macho de tres pines al final de estos cables trenzados y se ha comprobado el funcionamiento del sistema de IoT completo.
Nótese que todos los cables se han protegido con tubos termoretráctiles para evitar cortocircuitos indeseados y la exposición de los mismos al exterior.

3.2. Base de datos y Autentificación

A la hora de tener que determinar la forma de almacenar y gestionar los datos desde la nube, la elección de la plataforma adecuada se reveló como un paso crucial. Entre las distintas opciones disponibles en el panorama tecnológico, Firebase emergió como la opción principal, en gran medida debido al conjunto que características que presenta ya que se alinean perfectamente con las necesidades del proyecto que son, el poder mantener operativa 24/7 una base de datos en tiempo

real y la facilidad de implementación de un sistema de autentificación de usuarios todo desde la misma plataforma.

3.2.1. Firebase Real Time Database

Nuestra base de datos en tiempo real se ha implementado haciendo uso de *Real Time Database* de Firebase. Aquí la información se presenta en nodos como se aprecia en la Figura 3.13, que en realidad es un fichero JSON que alberga toda la información.

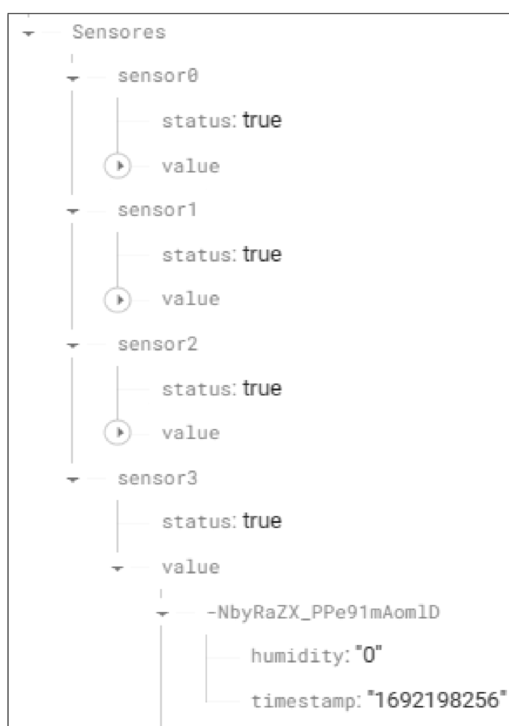


Figura 3.13: Estructura de la Base de datos en Firebase

La estructura del fichero se decidió que fuera de la siguiente forma. El nodo principal llamado "Sensores" alberga un nodo hijo por cada sensor que incluya nuestro sistema y a su vez cada uno de estos nodos hijo contiene otros dos nodos hijo, uno llamado "status" que hace referencia al estado de dicho sensor (conectado o desconectado) y otro llamado "value" donde se van almacenando cada una de las instancias de lectura de ese sensor. Dentro de cada uno de estas instancias, existen de nuevo otros dos subnodos, uno es "humidity" que es donde se almacena el valor de la lectura de humedad que ha realizado el sensor y otro es "timestamp" que es el instante temporal en EPOCH en el que la medida fue tomada.

Por otro lado, la base de datos requiere de medidas de seguridad para que no todo el mundo pueda ver y editar nuestros datos y de nuevo, gracias a Firebase, esto se realiza de forma muy sencilla desde el apartado de "Reglas" de nuestra *Real Time Database*. Un ejemplo es el que se puede apreciar en la Figura 3.14 en el que observamos que podemos aplicar una regla de acceso para cada nodo de nuestra base de datos, en este caso, se aplica al nodo raíz "Sensores" directamente. En este

ejemplo, el cual es el utilizado en el desarrollo del proyecto, permite la escritura de la base de datos únicamente a usuarios autenticados y la lectura por otro lado a todo el mundo. Obviamente no son reglas seguras ya que la lectura debería ser posible únicamente para usuarios autenticados pero con esta configuración se facilita el desarrollo pudiendo cambiarse esta regla de cara al lanzamiento del producto al mercado.

```
1  {
2    "rules": {
3      "Sensores": {
4        ".read": true,
5        ".write": "auth != null",
6      }
7    }
8  }
```

Figura 3.14: Reglas de acceso a la base de datos de Firebase

3.2.2. Firebase Authentication

En consonancia con el resto del proyecto, para el proceso de gestión de usuarios de nuestra app se ha optado por utilizar de nuevo las herramientas que nos brinda Firebase, concretamente Firebase Authentication. Esta potente herramienta garantiza una autentificación segura y sin problemas, lo que resulta crucial a la hora de proteger la privacidad y la integridad de los datos de los usuarios.

Firebase Authentication nos ofrece una amplia gama de métodos de registro y autentificación de usuarios brindándonos flexibilidad tanto a los desarrolladores a la hora de elegir qué métodos son mas afines para nuestra app, como a los usuarios al poderles facilitar varias formas de acceder a nuestra aplicación. Una de las ventajas que nos ofrece es la capacidad de realizar un registro de usuarios utilizando solo el correo electrónico o si se prefiere, implementar inicios de sesión a través de cuentas de proveedores de identidad populares como Google, Facebook, Apple, Microsoft, etc. Esta flexibilidad posibilita a los usuarios optar por opciones de inicio de sesión que mejor se adapten a sus preferencias y experiencia, a la vez que simplifica el proceso de acceso y registro.

En la Figura 3.15 podemos ver varios usuarios de pruebas registrados en la aplicación. Este panel de administración que se muestra en la figura lo podemos encontrar en la página de Firebase, lo que nos facilita la supervisión y gestión de los usuarios registrados. En caso de surgir cualquier problema esto nos permite una respuesta ágil y facilita la administración eficiente de la base de usuarios.

Al adoptar Firebase Authentication, garantizamos que los datos de humedad de los sensores de los usuarios que se recopilan y almacenan en el ecosistema de Firebase estén respaldados por un sistema de seguridad robusto y en conformidad con una buena práctica de autentificación.

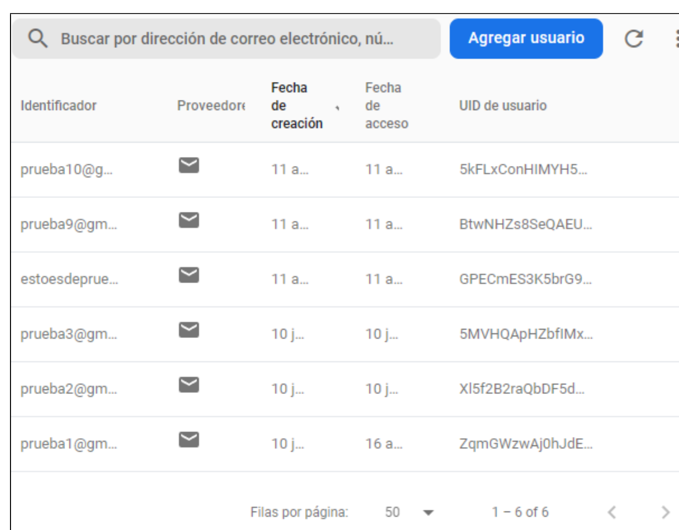


Figura 3.15: Usuarios registrados en la App

3.3. Diseño de la aplicación

3.3.1. Entorno de programación

A la hora de comenzar con el desarrollo de la aplicación, lo primero fue configurar el entorno de programación. Como software para el desarrollo se escogió Visual Studio Code en el cual tuvimos que instalar una serie de extensiones:

- **Flutter:** *Flutter support and debugger for Visual Studio Code.*
- **Flutter Widget Snippest:** *A set of helpful widget snippets for day to day Flutter development.*
- **Awesome Flutter Snippest:** *A collection snippets and shortcuts for commonly used Flutter functions and classes.*
- **Dart:** *Dart language support and debugger for Visual Studio Code.*
- **dart-import:** *Fix Dart/Flutter's imports.*

Tras la instalación de estas extensiones, solo nos falta un dispositivo en el que ir probando nuestra aplicación. Para eso existen dos opciones, o bien conectamos un dispositivo físico, o bien emulamos uno. En nuestro caso hemos empleado ambos métodos a lo largo de todo el desarrollo. Para el primero conectamos un *One Plus 9* al PC, habilitando en este las opciones de desarrollador y la depuración USB así como el desbloqueo OEM para que nos permita ejecutar nuestra aplicación desde Flutter directamente en el dispositivo. Esto nos ha servido para comprobar los gestos y la manejabilidad de la aplicación en un dispositivo final, lo cual ha resultado de gran ayuda, pero no es lo más cómodo, de ahí que se hayan empleado ambos métodos. Para el segundo se requiere de un emulador que, en nuestro caso, hemos utilizado el emulador de *Android Studio* y hemos emulado

un *Google Pixel 3a* entre otros, pero siendo este el mas frecuente.

Una vez hemos preparado nuestro entorno ya podemos comenzar a crear nuestro proyecto de Flutter y empezar con el desarrollo de la aplicación.

3.3.2. Framework

La decisión de emplear Flutter como el framework de preferencia para la creación de la aplicación móvil se fundamenta en una serie de consideraciones cruciales.

Una de las razones principales detrás de esta elección es su enfoque en el desarrollo de aplicaciones móviles híbridas de alta calidad. Es cierto que existen otros frameworks en el mercado, como por ejemplo *ionic framework*, que también permiten la creación de aplicaciones híbridas, pero Flutter destaca por su excepcional rendimiento y su capacidad para generar interfaces de usuario altamente personalizables y atractivas ya que basa su diseño en Widgets, que son elementos con los cuales se va construyendo la aplicación de forma sencilla. Al hacer uso de Flutter, reducimos drásticamente el tiempo y los recursos requeridos para el desarrollo y el mantenimiento de la aplicación móvil [2].

Por otro lado, también es cierto que otra opción podría haber sido, en vez de darle un enfoque híbrido, realizar un desarrollo nativo independiente para cada sistema operativo (una aplicación para iOS y otra para Android) pero se decidió un enfoque híbrido por varios factores. La principal consideración ya mencionada anteriormente es la optimización de resursos y tiempo de desarrollo. El hecho de hacer un desarrollo nativo requiere de equipos y recursos específicos para cada plataforma así como el desarrollo del código en dos lenguajes diferentes, lo que aumenta significativamente la complejidad y duración del proceso de desarrollo. En contraste, gracias al enfoque híbrido con Flutter podemos aprovechar un solo conjunto de habilidades y un único código base para implementar la aplicación en ambas plataformas. Ademas de simplificar el proceso de desarrollo, garantiza una coherencia en la experiencia del usuario a través de distintos sistemas operativos.

Otro factor determinante de la elección fue la estabilidad y robustez de Flutter, ya que es respaldado por Google [3] y se ha ganado una comunidad activa de desarrolladores que lo apoyan. Esto se puede traducir en constantes actualizaciones, mejoras y soporte técnico que asegura que la aplicación se mantenga actualizada y funcionando a lo largo del tiempo.

En cuanto al lenguaje de programación, Flutter emplea Dart, el cual es un lenguaje moderno y versátil que desencadena una serie de beneficios significativos en el proceso de desarrollo de aplicaciones móviles [4].
Se trata de un lenguaje orientado a objetos enfocado en la programación reactiva y asíncrona, lo que resulta fundamental en el desarrollo de aplicaciones altamente responsivas. En nuestro caso requerimos de actualizaciones en tiempo real de los datos por lo que esta característica se vuelve esencial para una experiencia fluida y sin interrupciones para los usuarios .

Otro aspecto de este framework que nos facilita el desarrollo es el uso de Widgets en el que se base Flutter. Los Widgets son elementos de la interfaz de usuario los cuales son reutilizables y personalizables y conforman la estructura visual de la aplicación. Para Flutter **todo** es un Widget, lo

que brinda coherencia y consistencia en la elaboración de la interfaz. Flutter nos permite combinar y anidar estos Widgets para crear elementos visuales complejos, permitiéndonos un control preciso del diseño y apariencia de la aplicación.

3.3.3. Diseño y progreso

Para comenzar con nuestra app, lo primero fue crear el proyecto de flutter. Para ello lo primero que debemos hacer desde Visual Studio Code será acceder al panel *View* →*Command Palette...* y a continuación buscamos la opción *Flutter: New Project* y la seleccionamos (Figura 3.16).

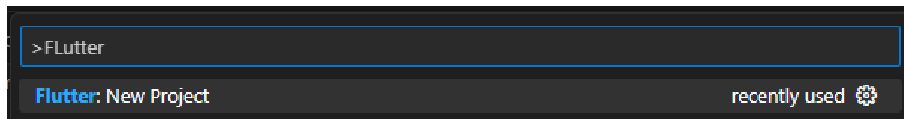


Figura 3.16: Creacción del proyecto FLutter

A continuación nos ofrecerá utilizar alguna plantilla predeterminada (Figura 3.17) , en nuestro caso seleccionamos *Application (empty)* para generar un esquema vacio para una aplicación movil.

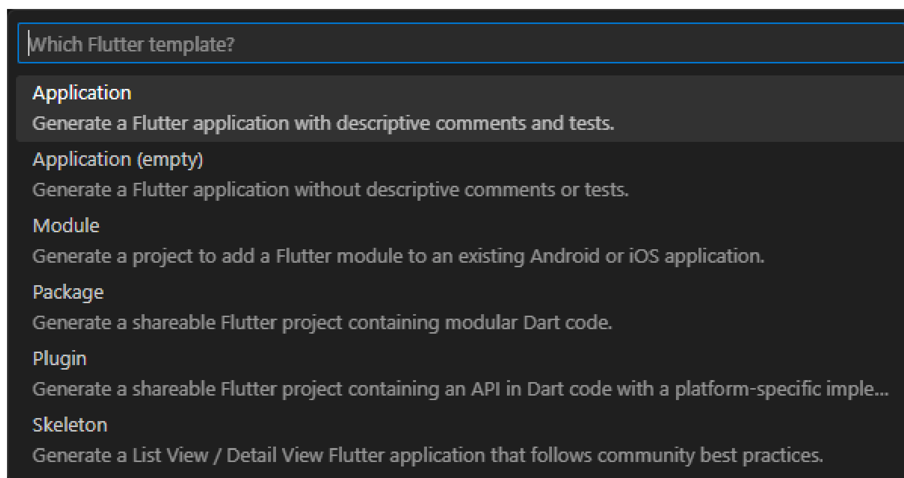


Figura 3.17: Plantillas de proyecto Flutter

Por último nos pedirá un nombre y una ubicación para nuestro proyecto y tras facilitar estos datos comenzará a generarnos una estructura de proyecto como la de la Figura 3.18.

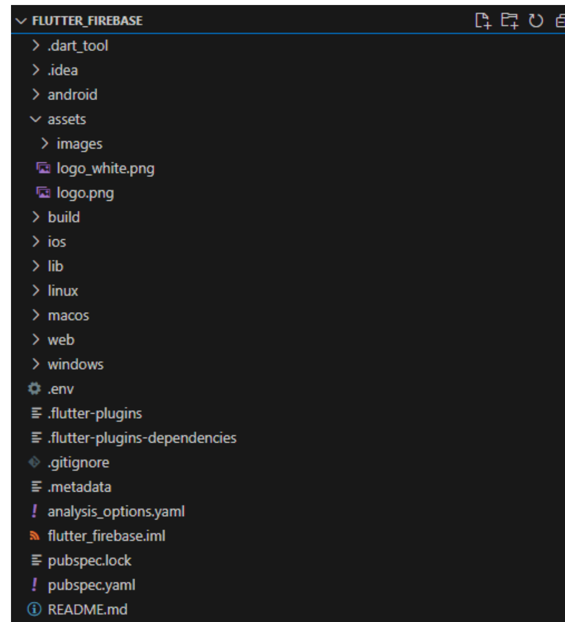


Figura 3.18: Estructura del proyecto tras su generación

3.3.3.1. Diseño y estructura de pantallas

Llegado este punto debemos elegir la estructura así como la apariencia que queremos que tenga nuestra aplicación. Para la estructura se quiso mantener un diseño intuitivo y fácil de manejar así que se decidió seguir un esquema como el de la Figura 3.19 mediante el cual queremos que haya una pantalla inicial que será la de "Login Page" donde los usuarios realizarán o bien el inicio de sesión o bien el registro como nuevo usuario. Después de iniciar sesión, la aplicación nos dirigirá directamente a la "Home Page", que será la pantalla principal donde encontremos todos los datos de nuestros sensores y donde podamos interactuar con ellos. Llegados a este punto, de nuevo fijándonos en la Figura 3.19, podemos observar que el desplazamiento entre pantallas pasa a ser lateral, pudiendo saltar de una a otra sin necesidad de tener que pasar por alguna en concreto. Con esto se quiere conseguir simplificar el proceso de navegación de la aplicación ya que puedes acceder a cualquier pantalla desde cualquier otra, sin necesidad de recordar si se accedía a lo que buscas desde una pantalla o desde otra. Además, gracias a esta estructura la aplicación puede expandirse de forma sencilla.

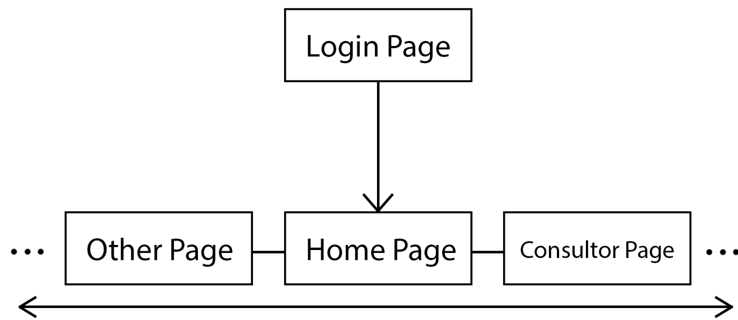


Figura 3.19: Esquema que sigue la estructura de la aplicación

En cuanto a las pantallas que se incluirán a desarrollar en nuestra aplicación para este proyecto son las siguientes.

- **Login Page:** Página donde los usuarios se registran e inician sesión.
- **Home Page:** Página principal de la aplicación donde se encuentra toda la información de los sensores.
- **Consultor Page:** Página que contendrá un consultor de dudas basado en inteligencia artificial.

Para estructurar las pantallas dentro de nuestro proyecto en Visual Studio Code debemos hacerlo desde “lib”, donde incluiremos todos los ficheros de nuestra aplicación como se muestra en la Figura 3.20.

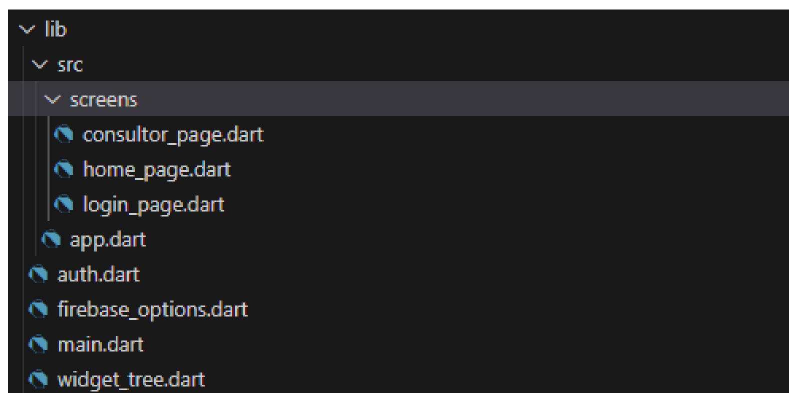


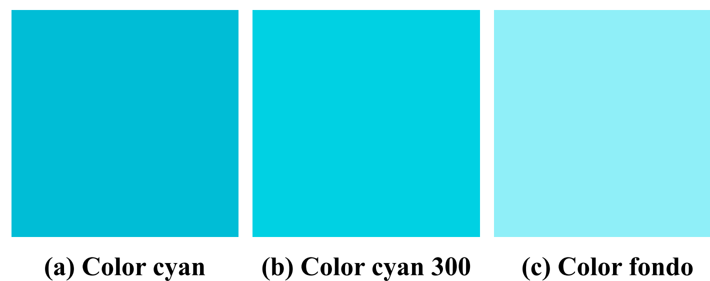
Figura 3.20: Contenido de la carpeta “lib” del proyecto

Hablemos ahora de la imagen que va a presentar nuestra aplicación. A la hora de hacer llegar información a la gente no hay mejor forma que con una imagen, es decir, es mas fácil llegar con formas y colores. Por este motivo decidí que lo mejor era crear un logo único para que se asocie

directamente al nombre de la aplicación, el cual decidí que fuera **Goty**, haciendo alusión a una gota de agua ya que estamos desarrollando un sistema de control de humedad. En cuanto al logo creado lo podemos observar en la Figura 3.21.

Figura 3.21: Logo creado para la app Goty

Siguiendo con los colores y, manteniendonos en la línea de representar con formas y colores lo que significa esta aplicación, se optó por tonos azules, concretamente se utiliza como color primario el *cyan* (Figura 3.22a), como color de acento el *cyan 300* (Figura 3.22b) y como color de fondo un RGB(176, 238, 247) (Figura 3.22c).



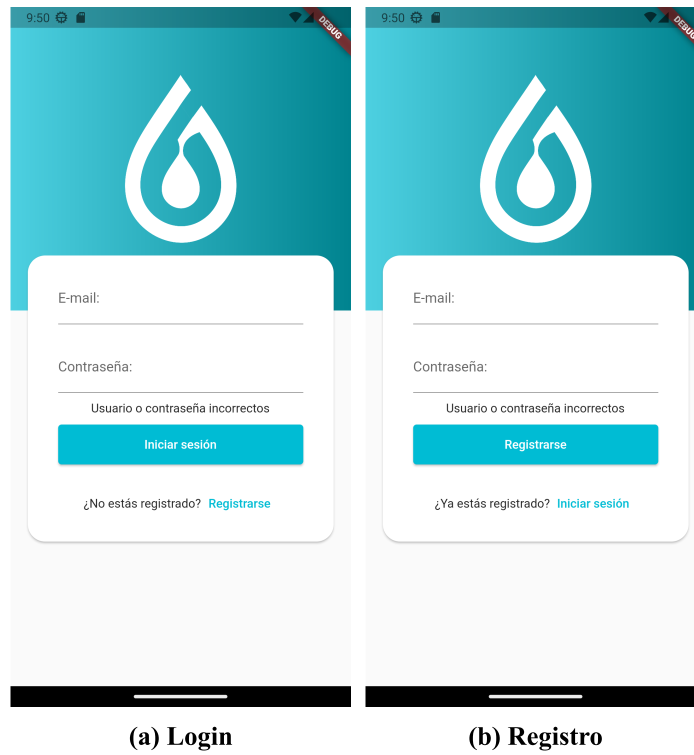
(a) Color cyan **(b) Color cyan 300** **(c) Color fondo**

Figura 3.22: Paleta de colores de la interfaz

3.3.3.2. Login Page

3.3.3.2.1. Interfaz .

Comenzamos ahora si con el desarrollo del código de la aplicación y lo primero fue diseñar nuestra página de inicio de sesión para que los usuarios puedan acceder. Manteniendo la uniformidad del proyecto conseguí una interfaz sencilla y amigable que tuviera lo justo y necesario para realizar su función. Podemos apreciar el trabajo terminado en la Figura 3.23 en la que se aprecian los dos estados de esta pantalla de Login y Registro.



(a) Login **(b) Registro**

Figura 3.23: Ambos estados de la "Login Page"

3.3.3.2.2. Código de interés .

Hablemos un poco de las partes del código que han sido relevantes para esta primera pantalla. Lo primero fue crear un fichero que llamamos *auth.dart* (apreciable en la estructura de ficheros de la ya mencionada Figura 3.19) en el cual creamos la clase *Auth*, que contiene métodos y atributos necesarios para llevar a cabo la autentificación con Firebase (Figura 3.24) tanto el login, como el registro como el logout.

```
1  import 'package:firebase_auth/firebase_auth.dart';
2  
3  class Auth{
4    final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;
5  
6    User? get currentUser => _firebaseAuth.currentUser;
7  
8    Stream<User?> get authStateChanges => _firebaseAuth.authStateChanges();
9  
10   Future<void> signInWithEmailAndPassword({
11     required String email,
12     required String password,
13   }) async {
14     final user = await _firebaseAuth.signInWithEmailAndPassword(
15       email: email,
16       password: password
17     );
18   }
19 
20   Future<void> createUserWithEmailAndPassword({
21     required String email,
22     required String password,
23   }) async {
24     final user = await _firebaseAuth.createUserWithEmailAndPassword(
25       email: email,
26       password: password
27     );
28   }
29 
30   Future<void> signOut() async {
31     await _firebaseAuth.signOut();
32   }
33 }
```

Figura 3.24: Clase *Auth* dentro de *auth.dart*

Dentro del fichero *login_page.dart*, que es donde se encuentra el desarrollo de esta primera pantalla, encontramos ciertas partes del código que pueden resultar de interés, como puede ser la función *handleSubmit()*, la cual es llamada tras pulsar el botón de “Iniciar sesión” / “Registrarse”. Como se aprecia en la Figura 3.25, dentro de esta a su vez se llama a otras dos funciones, *signInWithEmailAndPassword()* y *createUserWithEmailAndPassword()*, cuya llamada depende de si nos encontramos en el menú de iniciar sesión o en el de registrarse. El contenido de ambas es apreciable en la Figura 3.26.

```
handleSubmit() async {

  // Check if user is logged in:
  if (!_formKey.currentState!.validate()) return;

  final email = _controllerEmail.value.text;
  final password = _controllerPassword.value.text;

  setState(() => _loading = true);

  // Check if is login or register
  if(_isLogin) {
    await signInWithEmailAndPassword();
  }else {
    await createUserWithEmailAndPassword();
  }

  setState(() => _loading = false);
}
```

Figura 3.25: Función *handleSubmit()* dentro de *login_page.dart*

Si nos fijamos bien, estas funciones que acabamos de mencionar hacen uso de la ya citada clase *Auth* y de los métodos definidos en ella. En el caso de que surgiese un error a la hora de hacer el inicio de sesión o el registro en Firebase el error sería recogido dentro de la variable *errorMessage* e informado al usuario al respecto.

```
Future<void> signInWithEmailAndPassword() async {
  try {
    await Auth().signInWithEmailAndPassword(
      email: _controllerEmail.text,
      password: _controllerPassword.text,
    );
  } on FirebaseAuthException catch (e) {
    setState(() {
      errorMessage = e.message;
    });
  }
}
```

(a) *signInWithEmailAndPassword()*

```
Future<void> createUserWithEmailAndPassword() async {
  try {
    await Auth().createUserWithEmailAndPassword(
      email: _controllerEmail.text,
      password: _controllerPassword.text,
    );
  } on FirebaseAuthException catch (e) {
    setState(() {
      errorMessage = e.message;
    });
  }
}
```

(b) *createUserWithEmailAndPassword()*

Figura 3.26: Funciones de inicio de sesión y registro respectivamente

Por otro lado, también cabe destacar una de las propiedades del Widget utilizado para las cajas de texto de "E-mail" y "contraseña" (Figura 3.23), llamado *TextFormField()*, la cual se trata de *validator*. Esta propiedad sirve para indicar si el texto que se ha introducido en este campo de texto es valido o no. Podemos apreciar esta propiedad en la Figura 3.27.

```
TextFormField(
  decoration: const InputDecoration(
    labelText: "E-mail:"
  ), // InputDecoration
  controller: _controllerEmail,
  validator: (value) {
    if (value == null || value.isEmpty){
      return 'Por favor introduzca su E-mail';
    }
    String pattern = r"^[a-zA-Z0-9.a-zA-Z0-9.!#$%&'*+-/=?^_`{|}~]+@[a-zA-Z0-9]+\.[a-zA-Z]+";
    if (!RegExp(pattern).hasMatch(value)) {
      return 'Formato de E-mail inválido.';
    }
    return null;
  },
), // TextFormField
```

(a) Campo de texto “E-mail”

```
TextFormField(
  decoration: const InputDecoration(
    labelText: "Contraseña:"
  ), // InputDecoration
  controller: _controllerPassword,
  obscureText: true,
  validator: (value) {
    if (value == null || value.isEmpty){
      return 'Por favor introduzca su Contraseña';
    }

    String pattern = r'^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[!@#\$&*~]).{8,}$';

    if (!_isLogin){
      if (!RegExp(pattern).hasMatch(value)) {
        return ''' La contraseña debe tener al menos 8 caracteres,\n incluyendo mayusculas, números y símbolos.''';
      }
    }

    return null;
  },
), // TextFormField
```

(b) Campo de texto “contraseña”

Figura 3.27: Widget *TextFormField()* y sus propiedades

Con el *validator* se comprueba que la cadena de texto cumpla ciertas condiciones, en el caso del “E-mail” (Figura 3.27a) se comprueba que la cadena contenga letras, números o símbolos previamente a una “@”, seguido de una cadena de números o letras antes de un “.” y por último una cadena de letras, es decir, que sigan el siguiente formato:

- cadena1 @ cadena2 . cadena3

En cuanto al caso de la “contraseña” (Figura 3.27b) se comprueba que la cadena de texto tenga al menos 8 caracteres y que incluya mayúsculas, números y símbolos.
Si las cadenas de texto introducidas en alguno de estos campos no cumplen las condiciones citadas, se procede a informar al usuario mediante un texto en rojo que aparece en la parte inferior del respectivo cajón de texto.

Para finalizar con esta sección, hay que destacar que he empleado una página previa a “Login Page” que no posee interfaz y su función es redirigir a la página de login si el usuario no está autenticado o la “Home Page” si por lo contrario el usuario sí que está autenticado. Esta página previa se trata del “Widget Tree” y podemos observar su desarrollo en la Figura 3.28. Cada vez que arranca la aplicación pasará primero por esta página y aquí se decide, en función de si el usuario inició sesión o no, cual será la primera página a cargar.

```dart
import 'package:flutter_firebase/auth.dart';
import 'package:flutter_firebase/src/screens/home_page.dart';
import 'package:flutter_firebase/src/screens/login_page.dart';
import 'package:flutter/material.dart';

class WidgetTree extends StatefulWidget {
  const WidgetTree({Key? key}) : super(key: key);

  @override
  State<WidgetTree> createState() => _WidgetTreeState();
}
class _WidgetTreeState extends State<WidgetTree> {
  @override
  Widget build(BuildContext context) {
    return StreamBuilder(
      stream: Auth().authStateChanges,
      builder: (context,snapshot) {
        if (snapshot.hasData) {
          return const HomePage();
        } else {
          return const LoginPage();
        }
      },
    ); // StreamBuilder
  }
}
```

Figura 3.28: Clase auxiliar para comprobar autentificación

3.3.3.3. Home Page

3.3.3.3.1. Conexión con Firebase .

Pasamos ahora a hablar de la siguiente pantalla que es la “Home Page”. En esta pantalla se encuentra la mayor parte del trabajo realizado por lo que es la mas extensa de todas y, de nuevo, hablaremos de los aspectos clave que nos ayuden a comprender su funcionamiento.
Para comenzar debemos hablar de cómo se ha hecho la conexión con Firebase. Desde la propia página de Firebase encontraremos un apartado en su página inicial con los iconos de la Figura 3.29 los cuales nos llevan a las respectivas guías de Android e iOS para instalar el SDK de Firebase en nuestra aplicación. Tras finalizar los pasos ya tendremos los SDK de ambas plataformas en nuestro proyecto, ahora el siguiente paso es la conexión con Firebase.

Figura 3.29: Guias de Firebase para añadir SDKs

Ahora debemos introducir una serie de comandos en la consola para realizar la conexión con firebase:

1. Accede a Firebase con tu Cuenta de Google ejecutando el siguiente comando:

   ```
   firebase login
   ```

2. Para instalar la CLI de FlutterFire, ejecuta el siguiente comando desde cualquier directorio:

```
dart pub global activate flutterfire_cli
```

3. Desde el directorio del proyecto de Flutter, se ejecuta el siguiente comando para iniciar el flujo de trabajo de configuración de la app:

```
flutterfire configure
```

4. Desde el directorio de tu proyecto de Flutter, se ejecuta el siguiente comando para instalar el complemento principal:

```
flutter pub add firebase_core
```

5. De nuevo ejecutamos el siguiente comando para actualizar la configuración de Firebase en la app:

```
flutterfire configure
```

6. Ahora debemos agregar los complementos de Firebase que vayamos a utilizar, es decir, *Real Time Database* y *Authentication*:

```
flutter pub add firebase_database
flutter pub add firebase_auth
```

7. Por último, ejecutamos otra vez este comando para volver a actualizar la configuración de Firebase en la app:

```
flutterfire configure
```

Una vez introducida esta serie de comandos solo falta inicializar Firebase en nuestro código, lo cual haremos en el fichero *main.dart* (Figura 3.20) importando los paquetes de la Figura 3.30a y añadiendo las líneas de código de la Figura 3.30b a la función *main()*.

```dart
import 'package:firebase_core/firebase_core.dart';
import 'package:flutter_firebase/firebase_options.dart';
```

(a) Paquetes de Firebase

```dart
await Firebase.initializeApp(
  options: DefaultFirebaseOptions.currentPlatform,
);
```

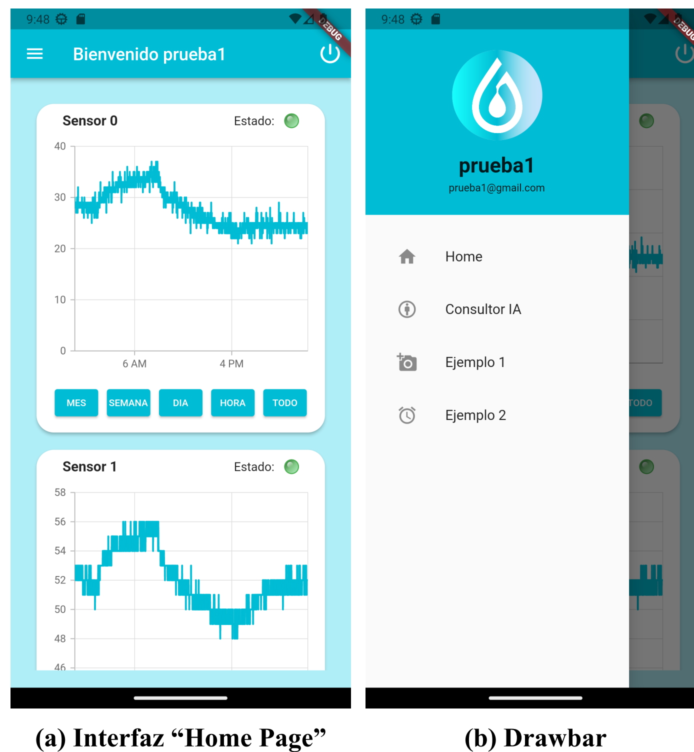
(b) Función de inicialización

Figura 3.30: Inicialización de firebase en *main.dart*

Con esto finalizaría la conexión de Firebase con nuestra app y ya podemos empezar a trabajar con ello, tanto para el sistema de autentificación ya mencionado en la sección Login Page como para la obtención de datos que veremos mas adelante.

3.3.3.3.2. Interfaz .

Comencemos ahora presentando directamente la interfaz de esta pantalla. En la Figura 3.31a podemos apreciar el diseño que se ha decidido para esta.



(a) Interfaz “Home Page” **(b) Drawbar**

Figura 3.31: Interfaz desarrollada en *home_page.dart*

A primera vista se pueden apreciar un conjunto de gráficas, cada una en una tarjeta distinta y con sus botones. Dentro de cada una de estas tarjetas aparece, en la parte superior izquierda el nombre del sensor al que corresponde, y en la derecha un indicador LED del estado del sensor, verde es conectado y rojo desconectado. En la parte central de la tarjeta se encuentra una gráfica interactiva donde se van representando los valores de los sensores en tiempo real. En la zona inferior de la tarjeta se encuentran una serie de botones que sirven para cambiar la vista temporal de la información que aparece en las gráficas.

Si nos fijamos ahora en la parte superior de la interfaz, podemos observar una *appbar* con varios elementos. El primero a mano izquierda se trata de una *drawbar*, que podemos ver desplegada en la Figura 3.31b. En esta *drawbar* encontramos en la parte superior el icono de nuestra aplicación y justo debajo el nombre de usuario y el email del usuario que se encuentra autenticado en ese momento. En la parte inferior se encuentra el menú principal de navegación mediante el cual el usuario podrá cambiar de una página a otra. Es aquí donde vemos el desplazamiento lateral de pantallas que se menciona en la sección Diseño y estructura de pantallas.

Continuando con los elementos de la parte superior de la interfaz, en el centro aparece un mensaje de bienvenida acompañado del nombre de usuario que haya iniciado sesión en ese momento. En la parte derecha de la appbar se encuentra el último elemento a comentar de la interfaz, el botón de cerrar sesión que al presionarlo aparece el mensaje de la Figura 3.32. Dentro de este mensaje aparecen otros dos botones, uno para cancelar el cierre de sesión y otro para confirmarlo.

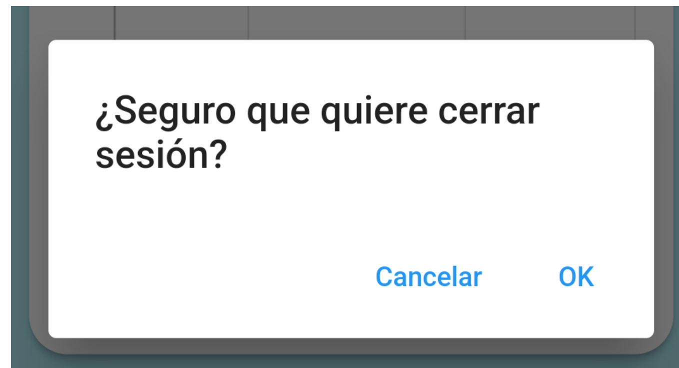


Figura 3.32: Mensaje de confirmación del cierre de sesión

3.3.3.3.3. Código de interés .

Hablemos ahora de las partes del código que se deben destacar. Comenzaremos hablando de una función llamada *fetchDataFromFirebase()*, la cual se encarga de obtener los datos de Firebase, tanto del estado de los sensores como de los valores de humedad y timestamps de estos. En la Figura 3.33 podemos apreciar el inicio y el final de la función, no aparece entera pues los procesos que se ven se repiten para cada uno de los flujos de datos de cada sensor.

```
void fetchDataFromFirebase() {

  databaseReference.child('sensor0/status').onValue.listen((event) {
    final status = event.snapshot.value;
    if(status!=null){
      setState(() {
        status0 = processStatus(status);
      });
    }
  });
```

⋮

```
  databaseReference.child('sensor3/value').onValue.listen((event) {
    final data = event.snapshot.value;
    if (data != null && data is Map<dynamic, dynamic>) {
      setState(() {
        dataPoints_3 = processData(data);

        // Sort datapoint by date:
        dataPoints_3.sort((a, b) {
          return a.timestamp.compareTo(b.timestamp);
        },);

        dataPointsSorted_3 = dataPoints_3;

        dataPointsLast_3 = filterData(dataPointsSorted_3,0);
        //Displays All(0), last Month(1), Week(2), Day(3) data.
        //(option 4 for debug, shows last 5 min)

      });
    }
  });
}
```

Figura 3.33: Función *fetchDataFromFirebase()*

En la parte inicial de esta función se accede al nodo de “status” de cada sensor dentro de la *Real Time Database* de Firebase y se almacenan estos valores en variables las cuales actualizan al instante del cambio el color de los LED de estado en la interfaz.
Por otro lado, en la parte inferior de la función, se realiza la obtención de todos los datos de humedad existentes de cada sensor en la base de datos. Los datos llegan en formato “Map” así que los procesamos mediante otra función llamada *processData()* ilustrada en la Figura 3.34. En esta función se separan los datos por *timestamp* y humedad y se añaden esta vez a una lista, que es la variable que retorna la función. Ademas, el *timestamp* lo pasamos a milisegundos para posteriormente trabajar con estos valores en las gráficas. Con esto conseguimos tener la información separada y de fácil acceso.

```
List<DataPoint> processData(Map<dynamic, dynamic> data) {
  final List<DataPoint> processedData = [];
  data.forEach((key, value) {

    var time = int.parse(value['timestamp'].toString().replaceAll(RegExp(r'[^0-9]'),'')) * 1000; // EPOCH time in miliseconds

    final dateTime = DateTime.fromMillisecondsSinceEpoch(time);
    final humidity = double.parse(value['humidity'].toString());

    processedData.add(DataPoint(dateTime, humidity));
  });
  return processedData;
}
```

Figura 3.34: Función *processData()*

Volviendo a la función *fetchDataFromFirebase()*, el siguiente paso dentro de esta es la ordenación de la lista en función de las marcas temporales, de esta manera nos aseguramos de que siempre aparezcan los datos cronológicamente en las gráficas. Por último aparece una nueva función llamada *filterData()* la cual se ha desarrollado de cara al posible futuro de la aplicación. Como veremos mas adelante, a la hora de representar los datos en las gráficas y hacer uso de los botones de cambio de vista, los datos representados siguen siendo los mismos pero visualizamos solo algunos de ellos. Sin embargo, esto puede llegar a un punto en el cual sature o ralentice la aplicación, y de ahí surge esta función, la cual filtra directamente los datos a representar de forma que no se saturen las gráficas. El código de esta función lo podemos observar en la Figura 3.35.

```
List<DataPoint> filterData(List<DataPoint> dataPoints, option) {
  final DateTime today = DateTime.now();


  switch (option) {
    case 1: // Show last Month
      final DateTime lastMonth = today.subtract(const Duration(days: 30));
      return dataPoints.where((dataPoint) => dataPoint.timestamp.isAfter(lastMonth)).toList();

    case 2: // Show last Week
      final DateTime lastWeek = today.subtract(const Duration(days: 7));
      return dataPoints.where((dataPoint) => dataPoint.timestamp.isAfter(lastWeek)).toList();

    case 3: // Show last Day
      final DateTime lastDay = today.subtract(const Duration(days: 1));
      return dataPoints.where((dataPoint) => dataPoint.timestamp.isAfter(lastDay)).toList();

    case 4: // Show last 5 min  ************** DEBUG ****************
      final DateTime lastMin = today.subtract(const Duration(minutes: 5));
      return dataPoints.where((dataPoint) => dataPoint.timestamp.isAfter(lastMin)).toList();

    default: // Show all
      return dataPoints;
  }
}
```

Figura 3.35: Función *filterData()*

Vamos ahora a observar parte del código de otra función de interés llamada *changeView()*. Esta función se encarga de cambiar la vista de los datos que se ven representados en las gráficas, como comentábamos anteriormente, éstas únicamente cambian los datos visualizados en la ventana de la gráfica, que no el número de datos representados. En la primera parte a comentar de la función (Figura 3.36), en base a la vista seleccionada, es decir, al botón pulsado dentro de la tarjeta de la gráfica (Figura 3.31a), se genera una variable de tipo “DateTime” del instante temporal en el que se pulsó el botón. A este instante temporal se le resta la cantidad de tiempo a visualizar en la gráfica dependiendo de la opción seleccionada.

```
// Calculate the view from now() DateTime:
switch (selectedView) {
  case 1: // Show last Month
    view = DateTime.now().subtract(const Duration(days: 30));
    break;

  case 2: // Show last Week
    view = DateTime.now().subtract(const Duration(days: 7));
    break;

  case 3: // Show last Day
    view = DateTime.now().subtract(const Duration(days: 1));
    break;

  case 4: // Show last hour
    view = DateTime.now().subtract(const Duration(hours: 1));
    break;

  case 5: // Show all data
    view = null;
    break;

  default: // Show all data in the current year
    view = DateTime(DateTime.now().year);
}
```

Figura 3.36: Primer segmento de la función *changeView()*

Tras realizar el cambio de vista, es conveniente realizar también un cambio en las etiquetas del eje temporal para que aparezcan de una forma uniforme. Esto lo conseguimos con esta otra parte de la función *changeView()* que podemos ver en la Figura 3.37.

```
// Update axis labels to display evenly:
int dataPointCount = dataPointsLast_0.length;
if (dataPointCount > 15) {
  interval_0 = (dataPointCount ~/ 15).toDouble();
}
```

Figura 3.37: Segundo segmento de la función *changeView()*

Por último vamos a comentar el código que aparece en el Anexo B. Se trata de la parte principal de la interfaz de esta pantalla. En el apartado B.1, se encuentra la estructura principal de la pantalla, que se construye a partir del Widget *Scaffold()*. Vamos a centrarnos en los Widgets que generan las gráficas de esta pantalla, ya que es su principal característica. Vemos que dentro del *Scaffold()* tenemos cuatro instancias de otro Widget llamado *humGraph()*, que es el encargado de generar la gráfica, cada instancia haciendo referencia a un sensor.

Para generar las gráficas desde este Widget se hace uso de un paquete llamado *syncfusion_flutter_charts* [5] el cual nos permite generar gráficas de todo tipo y, ademas, estas gráficas soportan actualizaciones en tiempo real, lo cual es imprescindible para este proyecto. Dentro de todas las opciones de gráficas que nos proporciona este paquete (línea, área, columna, barra, rango, puntos, tarta, etc), nosotros hacemos uso de un tipo de gráfica llamado *SfCartesianChart()*, que nos genera gráficas de línea en ejes cartesianos. Dentro de este Widget, aparte de toda la configuración de personalización de la que dispone, podemos elegir qué flujo de datos se va a representar en cada eje. Recordar que en la función *processData()* (Figura 3.34) adaptábamos los datos para que estuvieran en una lista, pues ahora las gráficas requieren recibir los datos de esta forma. En nuestro caso asignamos al eje “x” los valores de los *timestamps* y al eje “y” los valores de humedad asociados.

3.3.3.3.4. Gráficas de datos .

En este apartado vamos a tratar el funcionamiento de las gráficas a nivel de usuario. Como ya se ha mencionado, se tratan de gráficas interactivas, es decir, el usuario puede interactuar con ellas de diversas formas. La primera de ellas es mediante los botones que se sitúan en la parte inferior de cada gráfica que, como ya hemos explicado, cambian la vista de los datos representados que se visualizan.

Otra forma de interacción es mediante una doble pulsación, la cual produce que se haga zoom sobre la zona en la que se realizó dicha pulsación. Además, existe otra forma de hacer zoom que es pellizcando con los dedos la pantalla. De esta forma podemos tanto acercar como alejar la visualización. Ademas del zoom, las gráficas se pueden desplazar arrastrando el dedo por ellas.

También disponen de una funcionalidad para mostrar los datos con mas detalle si pulsamos sobre ellos. Al hacer esto surge una ventana con los valores exactos del punto de la gráfica que se ha seleccionado como se puede ver en la Figura 3.38.

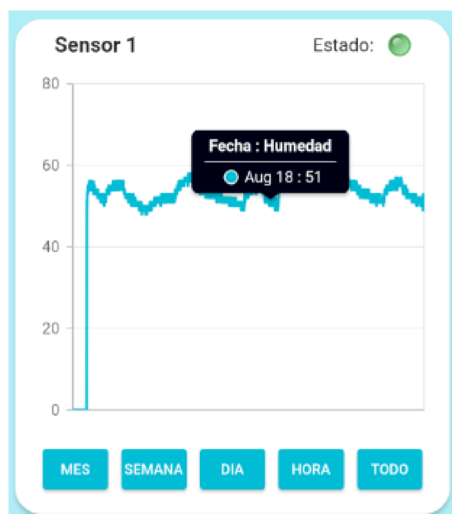


Figura 3.38: Mensaje de información al pulsar un dato

3.3.3.4. Consultor IA

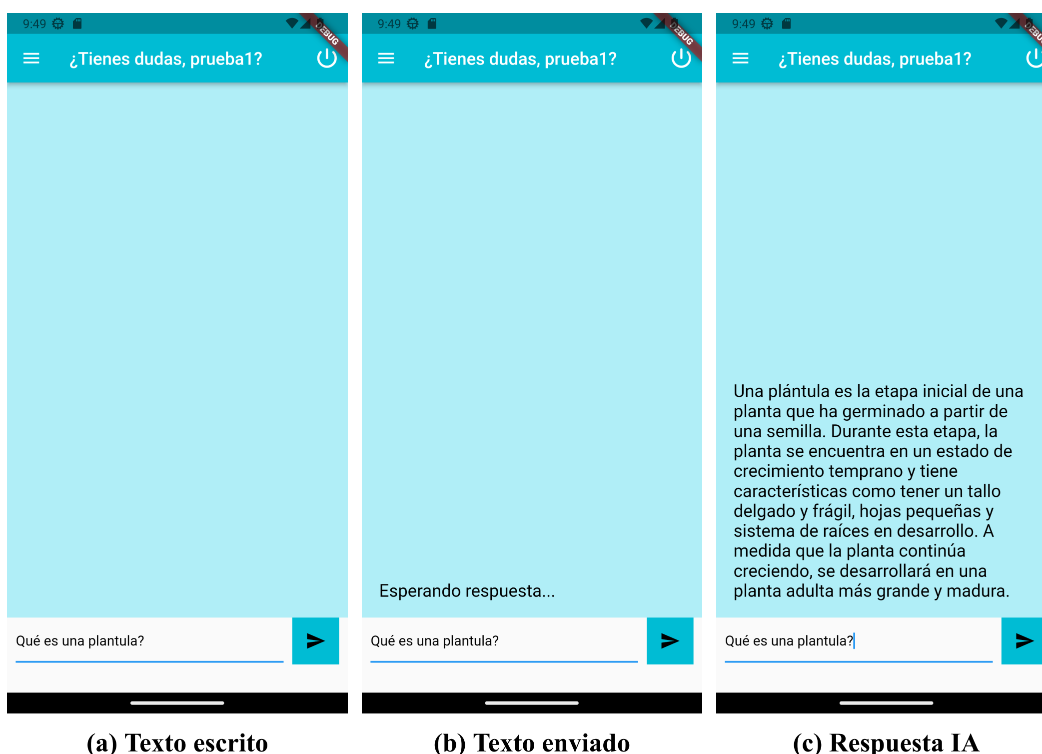
Ahora hablaremos de la última de las pantallas que se han desarrollado, un consultor de inteligencia artificial. La razón de esta pantalla radica en que los datos proporcionados por el sistema de sensores serán utilizados para la toma de decisiones sobre el cultivo, lo cual no siempre resulta sencillo. La idea es que, gracias a la inteligencia artificial, se facilite la toma de decisiones ya que el usuario podrá hacer consultas al momento a través de un chat sobre cualquier duda que le pudiera surgir.

3.3.3.4.1. Interfaz e interacción con el Consultor IA .

En cuanto al apartado gráfico de esta pantalla, decir que mantiene el esquema de colores uniforme de la aplicación así como la estructura de la appbar con la drawbar a mano izquierda y el botón de cerrar sesión a mano derecha. De nuevo nos centramos en la sencillez a la hora de generar la interfaz para facilitar su utilización a los usuarios de forma que, lo único que incluye esta pantalla

es un cajón de texto donde el usuario puede redactar su consulta, un botón para enviarla y un área central donde aparecerá la respuesta.

En la Figura 3.39, además de la interfaz descrita, podemos apreciar una secuencia de interacción con este consultor. Se trata de una secuencia sencilla donde el primer paso es redactar la consulta a realizar en la caja de texto (Figura 3.39a), el segundo es enviar la consulta mediante el botón situado a la derecha de la caja de texto y esperar a la respuesta del chat (Figura 3.39b) y el tercero es la recepción de la respuesta a la consulta la cual se presenta en el medio de la interfaz (Figura 3.39c).



(a) Texto escrito **(b) Texto enviado** **(c) Respuesta IA**

Figura 3.39: Interacción con el Consultor IA

3.3.3.4.2. API de OpenAI .

El desarrollo de este consultor de inteligencia artificial no hubiera sido posible sin hacer uso de la API de OpenAI [6]. Mediante esta API podemos acceder a modelos de inteligencia artificial de OpenAI de todo tipo, desde generadores de imágenes, texto, transcriptores de voz a texto hasta modelos de chat como el que se ha implementado en nuestra aplicación.
Concretamente se hace uso del modelo “gpt-3.5-turbo”, que es un modelo de chat, aunque tambien se ha realizado una implementación de un modelo de texto alternativo que es “text-davinci-003”.

Para interactuar con estos modelos, se les debe mandar un prompt (texto introducido por el usuario) y un conjunto de parámetros que, en nuestro caso, son el modelo que queremos utilizar y el número máximo de tokens. Los tokens son fragmentos individuales en los que se descompone el texto, ya sea una letra, un espacio, un signo de puntuación o incluso una parte de una palabra y son la unidad básica con la que se trabaja en el procesamiento del lenguaje natural. En definitiva, es

una unidad de longitud de texto y con la cual nosotros indicamos cual será la longitud máxima de la respuesta.
Las respuestas que genera el modelo de inteligencia artificial se denominan "completions" que viene de completar y del proceso de generación del texto.

3.3.3.4.3. Dart openai package .

A la hora de realizar la implementación de la API en nuestro proyecto, se podría hacer directamente realizando consultas HTTP a la API pero, en nuestro caso y puesto que estamos programando en Dart, existe un paquete que nos facilita la interacción con la API llamado *dart_openai* [7]. Gracias a este paquete podemos programar de forma sencilla las consultas a la API como se puede comprobar en la Figura 3.40.

```
import 'package:dart_openai/openai.dart';
```

```
completionFun() async {
  setState(() => responseTxt = 'Esperando respuesta...');

  // Text model:
  // final completion = await OpenAI.instance.completion.create(
  //   model: "text-davinci-003", //"gpt-3.5-turbo",
  //   prompt: promptController.text,
  //   maxTokens: 400, // Response length, more tokkens, more $ spending.
  // );

  // Chat model:
  final completion = await OpenAI.instance.chat.create(
    model: "gpt-3.5-turbo",
    messages: [
      OpenAIChatCompletionChoiceMessageModel(content: promptController.text, role: OpenAIChatMessageRole.user)
    ],
    maxTokens: 400, // Response length, more tokkens, more $ spending.
  );



  setState(() {
    responseTxt = completion.choices.first.message.content; // For chat model
    //responseTxt = completion.choices[0].text; // For text model
    debugPrint('RESPUESTA: $responseTxt');
  });
}
```

Figura 3.40: Función *completionFun()*

En esta imagen se aprecia la función *completionFun()*, la cual se encarga de hacer la consulta a la API y almacenar la respuesta de esta. Como comentábamos antes, tenemos implemantados tanto un modelo de texto "text-davinci-003" (parte superior comentada del código de la función) como un modelo de chat "gpt-3.5-turbo" en la parte inferior de la función. Tras recibir el "completion" del modelo de inteligencia artificial, se actualiza en estado de la interfaz para que aparezca el texto recibido mediante la función *setState()*. Cabe destacar que para poder hacer uso de la API de OpenAI debemos generar previamente una "API Key" desde su página web y agregarla a nuestro proyecto para poder iniciar la conexión con la API.

3.4. Materiales y presupuesto

En este apartado vamos a realizar una evaluación del presupuesto total del proyecto hasta el punto que se ha desarrollado. Desde los componentes electrónicos utilizados en el sistema de sensores hasta los costos asociados con el desarrollo de la aplicación móvil y la infraestructura en la nube, este apartado arroja luz sobre los elementos esenciales que respaldan la realización del proyecto. Podemos consultar el listado de materiales en la Tabla 3.2.

Componente	Cantidad	Precio Unitario (€)	Costo Total (€)
IoT			
ESP32 CH9102X	1	3.54	3.54
Sensor Humedad Capacitivo HW-390	4	0.49	1.96
Cables 22 AWG 5m	3	2.61	7.83
Pines para terminales JST XH2.54	12	0.0098	0.1176
Terminales JST XH2.54 (pareja macho y hembra)	4	0.047	0.188
Cubiertas termoretráctiles	1	0.49/m	0.0784
Caja sensor	4	0.58	2.32
Caja ESP32	1	0.935	0.935
Infraestructura en la nube			
Firebase Real Time Database	1	0 (hasta 100 conexiones)	0
Firebse Authentication	1	0	0
Desarrollo de la aplicación			
API OpenAI	1	0.002/1k tokens	0.002/1k tokens
TOTAL			16.969 + uso de la API

Tabla 3.2: Listado de materiales y presupuesto

Capítulo 4

Conclusiones y líneas futuras

4.1. Conclusiones

Las consclusiones finales de este proyecto abordan un proceso de desarrollo enriquecedor que ha logrado integrar tecnologías diversas para afrontar los desafíos planteados. A lo largo de este camino, se ha puesto de manifiesto la capacidad de converger de forma exitosa de diversos campos que son las telecomunicaciones, la informática y la agricultura de precisión, dando lugar a una solución completa.

Desde la elección del microcontrolador ESP32 hasta la implementación de las herramientas de Firebase e incluso el framework de Flutter, cada decisión ha sido tomada realizando un análisis previo exhaustivo de las necesidades del proyecto. La implementación de los sensores de humedad para el suelo y su integración con la nube en el apartado de IoT ha garantizado una recopilación precisa de datos en tiempo real y esto resulta crucial para la toma de decisiones en el ámbito de la agricultura.

La elección de Firebase como herramienta para la gestión de usuarios mediante autentificación como para la gestión de la base de datos en tiempo real ha garantizado la seguridad y la accesibilidad de los datos. Por otro lado la elección de Flutter ha demostrado ser la ideal para la creación de aplicaciones móviles multiplataforma de calidad, con interfaces intuitivas y coherentes que facilitan la experiencia a los usuarios.

En cuanto a la integración de la API de OpenAI ha enriquecido aún mas el proyecto, permitiendo que los usuarios realicen consultas sobre sus cultivos, sobre los datos que reciben de los sensores o incluso de cualquier tema que les pudieran surgir.

En última instancia, este proyecto ha demostrado que la convergencia de las ya citadas áreas al comienzo de esta sección puede generar soluciones significativas y prácticas. El resultado final es un sistema de monitoreo de humedad del suelo que empodera a los usuarios proporcionándoles datos en tiempo real así como una experiencia de usuario fluida. A medida que la tecnología siga evolucionando, el proyecto sienta las bases para futuras innovaciones en el campo de la agricultura de precisión y la aplicación de IoT en entornos agrícolas.

4.2. Líneas futuras

Este apartado abre las puertas hacia las posibilidades de expansión de este proyecto, planteando un vistazo a cómo el proyecto puede adaptarse para abarcar nuevas funcionalidades y desafíos. A continuación expondré algunos de los enfoques potenciales que me gustaría dar a este proyecto.

Nuevos métodos de autentificación:

En la aplicación móvil de este proyecto se ha desarrollado un único método de autentificación creando una cuenta en la base de datos con un correo electrónico. Sin embargo, puesto que se ha implementado con Firebase Authentication, es posible expandir las posibilidades de acceso pudiendo incorporar nuevos métodos de autentificación con cuentas existentes en otras plataformas como puedan ser Google, Facebook o Apple.

Sustitución del cableado:

Actualmente los sensores de humedad se conectan al ESP32 mediante tres cables, lo cual en ciertas situaciones puede dificultar la instalación del sistema. La idea es poder sustituir todo este cableado por una comunicación inalámbrica y una batería por cada sensor. De esta forma, empleando tecnologías como Bluetooth, WiFi, ZigBee o incluso una comunicación por radio frecuencia podríamos mejorar la capacidad de distribución de los sensores a la hora de la instalación así como facilitar la ampliación del número de sensores. Los sensores, debido a su bajo consumo, podrían ser alimentados mediante baterías y que estos se mantuvieran operativos durante largos periodos de tiempo.

Mas tipos de sensores:

Puesto que trabajamos con un ESP32, todavía tenemos múltiples entradas de datos disponibles y que podrían ser aprovechadas. La idea es añadir más tipos de sensores de forma que podamos obtener mas datos acerca de los cultivos como podrían ser sensores de humedad ambiente, para controlar la humedad del aire en el entorno de plantación, sensores de PH para medir el nivel de acided del suelo, muy importante para la absorción de nutrientes en las plantas, sensores de EC para saber la concentración de partículas en el suelo y saber cuando las plantas requieren de fertilizantes, sensores de luz para saber si la cantidad de luz que están recibiendo los cultivos es la adecuada e incluso sensores de CO2 para saber la concentración de éste en el aire ya que es fundamental para la fotosíntesis.

Control de sistemas de riego:

Hasta ahora desde la aplicación móvil solo hemos recibido información, pero también es posible mandarla. Esto se puede aprovechar para vincular un sistema de riego, el cual podría activarse también (o no) desde el ESP32, cuyos parámetros podríamos gestionar desde la aplicación. Incluso se podría automatizar este en base a los valores recibidos de los sensores y en la aplicación recibir

notificaciones de cuando se ha regado.

Control de otros sistemas:

Esta ampliación del proyecto, aunque no es meramente exclusivo, está pensada para entornos de cultivo de interior, ya que los parámetros ambientales en este caso se pueden controlar con precisión. Se plantea, al igual que se ha hecho en el punto anterior, conectar mas sistemas al ESP32 para poder automatizar procesos. Por ejemplo, si lo juntamos a la ampliación de tipos de sensores, podemos conectar sistemas que modifiquen los valores de estos nuevos sensores. Estos sistemas podrían ser un humidificador y un deshumidificador para la humedad ambiente, lamparas de cultivo que se regulan en función de los sensores de luz, un sistema de regulación del PH y fertilizantes en el agua de riego en función de los valores de PH y EC del suelo e incluso generadores de CO2, extractores e intractores de aire, ventiladores y un largo etcétera de posibilidades.

Notificaciones:

Por último y, volviendo a hablar de la aplicación, considero que sería necesario añadir un sistema de notificaciones para el móvil o bien desde la propia aplicación o bien desde Firebase ya que también proporciona este servicio. Las notificaciones pueden resultar cruciales para enterarse del estado de los cultivos o de algún problema con ellos o con alguno de los sistemas. Desde desconexión de sensores hasta valores críticos de humedad son eventos importantes que los usuarios deben saber de inmediato para disponer del mayor tiempo posible de reacción ante estos sucesos.

Bibliografía

[1] Delta-T Devices. *SM150T Soil Moisture Sensor*. 2023. URL: https://delta-t.co.uk/product/sm150t/.

[2] Google. *Flutter documentation*. URL: https://docs.flutter.dev/.

[3] Antonio Jesús Ocaña. *Google Flutter, nace el rival de React Native y Xamarin*. URL: https://www2.deloitte.com/es/es/pages/technology/articles/google-flutter-framework-rival-react-native-xamarin.html.

[4] Google. *Dart documentation*. URL: https://dart.dev/guides.

[5] syncfusion.com. *Syncfusion flutter charts*. URL: https://pub.dev/packages/syncfusion_flutter_charts.

[6] OpenAI. *API de OpenAI*. URL: https://openai.com/blog/openai-api.

[7] Mohamed Anas. *Dart OpenAI*. URL: https://pub.dev/packages/dart_openai.

Parte II

Anexos

Apéndice A

Manual de usuario

MANUAL DE USUARIO

DE

GOTY

Autor:
Víctor Ramos Galindo

Agosto 2023

A.1. Introducción

A.1.1. Acerca de “Goty”

“Goty” es una aplicación móvil que ha sido diseñada principalmente para la visualización de datos de sensores de humedad en gráficas en tiempo real. También podemos comprobar el estado de nuestros sensores y realizar consultas a una inteligencia artificial. Te brinda información valiosa sobre el entorno y te permite interactuar de manera inteligente con tus datos.

A.1.2. Requisitos del Sistema

- Dispositivo móvil con sistema operativo Android o iOS.
- Conexión a Internet para acceder a las funciones en línea.
- Espacio de almacenamiento suficiente para la aplicación.

A.2. Registro e Inicio de Sesión

A.2.1. Crear una cuenta

Para poder acceder a todas las funciones de “Goty”, primero debes crear una cuenta de usuario. En la pantalla inicial de la aplicación, en la parte inferior debes seleccionar “Registrarse”. Una vez en la pantalla de registro ingresa tu información personal, como la dirección de correo y una contraseña. La contraseña deba tener una longitud de al menos 8 caracteres y debe incluir mayúsculas, minúsculas, números y caracteres especiales.

A.2.2. Iniciar sesión en la aplicación

Después de crear una cuenta, puedes iniciar sesión en la aplicación con tu correo electrónico y contraseña. Esto te dará acceso a todas las características de ”Goty”.

A.2.3. Cerrar sesión en la aplicación

Una vez se ha iniciado sesión, en la parte superior derecha aparece un icono de “Apagar”. Al pulsarle surgirá una ventana emergente que nos preguntará si queremos cerrar sesión. Debe pulsar en “OK”.

A.3. Pantalla Principal

A.3.1. Visualización de datos de los sensores

En la pantalla principal, podrás ver la información que proporcionan los distintos sensores. La información de cada sensor aparece en una tarjeta con el nombre de este. Los datos se presentan en forma de gráfica para que puedas comprender mejor los cambios en el entorno.

A.3.2. Estado de los sensores

En la tarjeta de cada sensor, en la parte superior derecha aparece un indicador LED. Si la luz está verde significa que el sensor se encuentra conectado. Si por el contrario el indicador LED se encuentra rojo, significa que el sensor está desconectado.

A.3.3. Interacción con las gráficas

Puedes interactuar con las gráficas de las siguientes maneras:

- Doble toque: Hace zoom sobre la región que se pulsa.
- Deslizar el dedo: Mueve la gráfica en la dirección que se desliza.
- Pinzar hacia fuera: Hace zoom de forma precisa sobre la zona en la que se realiza el gesto.
- Pinzar hacia dentro: Quita zoom de forma precisa sobre la zona en la que se realiza el gesto.
- Pulsar sobre un punto de datos: Muestra un cuadro emergente con la información del punto de datos seleccionado.

A.4. Consultas a la Inteligencia Artificial

Para realizar consultas con la IA solo hay que escribir lo que queramos consultar en el cuadro de texto de la parte inferior de la interfaz. Para enviar la consulta hay que pulsar en el botón que se encuentra a la derecha del cuadro de texto, al hacerlo aparecerá un mensaje de espera. Tras unos segundos aparecerá en el centro de la pantalla la respuesta a la consulta. Tenga en cuenta que las respuestas tienen una longitud máxima. Si realizas consultas que requieran de una respuesta larga esta puede que no aparezca por completo.

A.5. Cómo Navegar

Para navegar por las distintas pantallas de la aplicación debe pulsar en el icono de tres barras horizontales de la parte superior izquierda. Al hacerlo se abrirá un menú desplegable en el cual se encuentran las distintos apartados de la aplicación. Para cambiar de uno a otro solo hay que seleccionarlo dentro de este menú desplegable.

Apéndice B

Código del Widget *humGraph()*

B.1. Integración del Widget:

```
//-----------------------------------------------------------------------------------------------------
//                                          SCAFFOLD:
@override
Widget build(BuildContext context) {
    return Scaffold(
      drawer: NavigationDrawer(),
      appBar: AppBar(
        title: _title(),
        actions: [_signOutButton()],
        backgroundColor: Theme.of(context).primaryColor,
      ),
      body: Container(
        color: const Color.fromARGB(255, 176, 238, 247),
        height: double.infinity,
        width: double.infinity,
        padding: const EdgeInsets.all(20),
        child: ListView(
          children: <Widget>[
            humGraph(dataPointsLast_0, 'Nivel Humedad Sensor 0', 'Sensor 0', 0),// _sensorCard('sensor0'),
            humGraph(dataPointsLast_1, 'Nivel Humedad Sensor 1', 'Sensor 1', 1),// _sensorCard('sensor1'),
            humGraph(dataPointsLast_2, 'Nivel Humedad Sensor 2', 'Sensor 2', 2),// _sensorCard('sensor2'),
            humGraph(dataPointsLast_3, 'Nivel Humedad Sensor 3', 'Sensor 3', 3),// _sensorCard('sensor3'),
          ],
        ),
      ),
    );
  }
}
```

B.2. Widget *humGraph()*:

```
Widget humGraph(source, graphName, sensorName, graphId) {
    return Card(
      elevation: 3,
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(20),
      ),
      margin: const EdgeInsets.symmetric(vertical: 10, horizontal: 10),//only(left: 10, right: 10, top: 10, bottom: 10),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        mainAxisSize: MainAxisSize.max,
        children: [
          const SizedBox(height: 10,),
          Padding(
            padding: const EdgeInsets.symmetric(horizontal: 30),
            child: Row(
              mainAxisAlignment: MainAxisAlignment.spaceBetween,
              children: [
                Text(sensorName,
                  style: const TextStyle(
                    fontSize: 16,
                    fontWeight: FontWeight.bold,
```

```
        ),
      ),
      Row(
        children: [
          const Text('Estado:    '),
          LedBulbIndicator(
            initialState: checkState(graphId),
            glow: false,
            size: 17,
          ),
        ],
      ),
    ],
  ),
),
SfCartesianChart(
  margin: const EdgeInsets.all(20),
  palette: const [
    Colors.cyan,
  ],
  primaryXAxis: DateTimeAxis(
    isVisible: true,
    //intervalType: DateTimeIntervalType.auto,
    //rangePadding: ChartRangePadding.auto,
    visibleMinimum: selViewId(graphId),
    interval: selInterval(graphId),
    edgeLabelPlacement: EdgeLabelPlacement.shift,

  ),
  //title: ChartTitle(text: graphName),
  legend: Legend(
    isVisible: false,
    //title: LegendTitle(text: 'Leyenda'),
  ),
  tooltipBehavior: TooltipBehavior( // Show data when tapping a datapoint
    enable: true,
    header: 'Fecha : Humedad',
  ),
  zoomPanBehavior: ZoomPanBehavior(
    zoomMode: ZoomMode.x,              // Only zoom in X axis
    enablePanning: true,               // Allow move the zoomed area
    enableDoubleTapZooming: true,      // Enable zoom with double tap
    enablePinching: true,              // Enable zoom by pinching with 2 fingers
  ),
  series: <ChartSeries>[
    LineSeries<DataPoint, DateTime>(
      name: sensorName,
      dataSource: source,
      xValueMapper: (DataPoint data, _) => data.timestamp,
      yValueMapper: (DataPoint data, _) => data.humidity,
    ),
  ],
),
Padding(
  padding: const EdgeInsets.symmetric(horizontal: 16.0),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    children: [
      Flexible(
        child: Container(
          alignment: Alignment.topCenter,
          height: 50,
          margin: const EdgeInsets.symmetric(horizontal: 5),
          child: FittedBox(
            fit: BoxFit.scaleDown,
            child: viewButton(1, 'MES', graphId)
          ),
        ),
      ),
      Flexible(
        child: Container(
          alignment: Alignment.topCenter,
          height: 50,
          margin: const EdgeInsets.symmetric(horizontal: 5),
          child: FittedBox(
            fit: BoxFit.scaleDown,
            child: viewButton(2, 'SEMANA', graphId)
          ),
        ),
      ),
      Flexible(
        child: Container(
          alignment: Alignment.topCenter,
          height: 50,
          margin: const EdgeInsets.symmetric(horizontal: 5),
          child: FittedBox(
            fit: BoxFit.scaleDown,
            child: viewButton(3, 'DIA', graphId)
          ),
        ),
      ),
```

```
                Flexible(
                  child: Container(
                    alignment: Alignment.topCenter,
                    height: 50,
                    margin: const EdgeInsets.symmetric(horizontal: 5),
                    child: FittedBox(
                      fit: BoxFit.scaleDown,
                      child: viewButton(4, 'HORA', graphId)
                    ),
                  ),
                ),
                Flexible(
                  child: Container(
                    alignment: Alignment.topCenter,
                    height: 50,
                    margin: const EdgeInsets.symmetric(horizontal: 5),
                    child: FittedBox(
                      fit: BoxFit.scaleDown,
                      child: viewButton(5, 'TODO', graphId)
                    ),
                  ),
                ),
              ],
            ),
          ),
        ],
      ),
    );
  }
```

B.3. Funciones y Widgets contenidos en *humGraph()*:

B.3.1. Función *checkState()*

```
checkState(graphId){
   switch (graphId) {
     case 0:
       if(status0) {
         return LedBulbColors.green;
       } else {
         return LedBulbColors.red;
       }

     case 1:
       if(status1) {
         return LedBulbColors.green;
       } else {
         return LedBulbColors.red;
       }

     case 2:
       if(status2) {
         return LedBulbColors.green;
       } else {
         return LedBulbColors.red;
       }

     case 3:
       if(status3) {
         return LedBulbColors.green;
       } else {
         return LedBulbColors.red;
       }

     default:
   }
}
```

B.3.2. Función *selViewId()*

```
selViewId(graphId){
  switch (graphId) {
    case 0:
      return view_0;

    case 1:
      return view_1;
```

```
    case 2:
      return view_2;

    case 3:
      return view_3;

    default:
  }
}
```

B.3.3. Función *selInterval()*

```
selInterval(graphId){
  switch (graphId) {
    case 0:
      return interval_0;

    case 1:
      return interval_1;

    case 2:
      return interval_2;

    case 3:
      return interval_3;

    default:
  }
}
```

B.3.4. Widget *viewButton()*

```
Widget viewButton(view, name, graphId){
  return SizedBox(
    height: 40,
    child: ElevatedButton(
      onPressed: () {
        setState(() {
          switch (graphId) {
            case 0:
              selectedView_0 = view;
              changeView(graphId);
              debugPrint("selectedView: $selectedView_0");
              break;
            case 1:
              selectedView_1 = view;
              changeView(graphId);
              debugPrint("selectedView: $selectedView_1");
              break;
            case 2:
              selectedView_2 = view;
              changeView(graphId);
              debugPrint("selectedView: $selectedView_2");
              break;
            case 3:
              selectedView_3 = view;
              changeView(graphId);
              debugPrint("selectedView: $selectedView_3");
              break;
            default:
          }

        });
      },
      style: ElevatedButton.styleFrom(
        padding: const EdgeInsets.symmetric(vertical: 10),
        backgroundColor: Theme.of(context).primaryColor,
        //primary: Colors.cyan,
        foregroundColor: Colors.white,
        //onPrimary: Colors.white
      ),
      child: Text(name),
    ),
  );
}
```

B.3.5. Función *changeView()*

```
Future changeView(graphId) async {

  DateTime? view;
  int selectedView = 6;

  switch (graphId) {
    case 0:
      selectedView =selectedView_0;
      break;

    case 1:
      selectedView =selectedView_1;
      break;

    case 2:
      selectedView =selectedView_2;
      break;

    case 3:
      selectedView =selectedView_3;
      break;

    default:
      debugPrint('Not a correct graphId');
      selectedView = 6;
  }

  // Calculate the view from now() DateTime adding 2h (GMT+2):
  switch (selectedView) {
    case 1: // Show last Month
      //view = DateTime.now().subtract(const Duration(days: 30));
      view = DateTime.now().add(const Duration(hours: 2)).subtract(const Duration(days: 30));
      break;

    case 2: // Show last Week
      //view = DateTime.now().subtract(const Duration(days: 7));
      view = DateTime.now().add(const Duration(hours: 2)).subtract(const Duration(days: 7));
      break;

    case 3: // Show last Day
      //view = DateTime.now().subtract(const Duration(days: 1));
      view = DateTime.now().add(const Duration(hours: 2)).subtract(const Duration(days: 1));
      break;

    case 4: // Show last hour
      //view = DateTime.now().subtract(const Duration(minutes: 5));
      view = DateTime.now().add(const Duration(hours: 2)).subtract(const Duration(hours: 1));
      break;

    case 5: // Show last 5 min
      //view = DateTime.now().subtract(const Duration(minutes: 5));
      view = null;
      break;

    default: // Show all data in the current year
      view = DateTime(DateTime.now().year);
  }

  switch (graphId) {
    case 0:
      view_0 =view;

      // Update axis labels to display evenly:
      int dataPointCount = dataPointsLast_0.length;
      if (dataPointCount > 15) {
        interval_0 = (dataPointCount ~/ 15).toDouble();
      }
      break;

    case 1:
      view_1 =view;

      // Update axis labels to display evenly:
      int dataPointCount = dataPointsLast_1.length;
      if (dataPointCount > 15) {
        interval_1 = (dataPointCount ~/ 15).toDouble();
      }
      break;

    case 2:
      view_2 =view;

      // Update axis labels to display evenly:
      int dataPointCount = dataPointsLast_2.length;
      if (dataPointCount > 15) {
        interval_2 = (dataPointCount ~/ 15).toDouble();
      }
      break;
```

```
    case 3:
      view_3 =view;

      // Update axis labels to display evenly:
      int dataPointCount = dataPointsLast_3.length;
      if (dataPointCount > 15) {
        interval_3 = (dataPointCount ~/ 15).toDouble();
      }
      break;

    default:
      debugPrint('Not a correct graphId');
  }

}
```