RID | UNAJ
Repositorio Institucional Digital UNAJ

Universidad Nacional
ARTURO JAURETCHE

Navarro, Edgardo Leonel

Tecnologías de la información y las comunicaciones (TIC) mediante IoT para la solución de problemas en el medio agroindustrial

2022

Instituto: Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Esta obra está bajo una Licencia Creative Commons Argentina.
Atribución – no comercial – sin obra derivada 4.0
https://creativecommons.org/licenses/by-nc-nd/4.0/

Documento descargado de RID - UNAJ Repositorio Institucional Digital de la Universidad Nacional Arturo Jauretche

Cita recomendada:
Navarro, E. L. (2022) *Tecnologías de la información y las comunicaciones (TIC) mediante IoT para la solución de problemas en el medio agroindustrial* [Informe de la práctica Profesional Supervisada] Universidad Nacional Arturo Jauretche
Disponible en RID - UNAJ Repositorio Institucional Digital UNAJ https://biblioteca.unaj.edu.ar/rid-unaj-repositorio-institucional-digital-unaj

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Carrera de Ingeniería en Informática

PRACTICA PROFESIONAL SUPERVISADA
Informe final

Tecnologías de IoT y aprendizaje automático para la solución de problemas en el medio productivo y el cuidado del medio ambiente

Edgardo Leonel Navarro

Florencio Varela, Julio de 2022

Estudiante

Apellido y Nombres: Navarro, Edgardo Leonel

Correo electrónico: dismoi.leo@gmail.com

Organización donde se realiza la Práctica Profesional Supervisada

Universidad Nacional Arturo Jauretche

Av. Calchaquí 6200, B1888 Florencio Varela, Provincia de Buenos Aires.

+54 9 11 4275-6100

Sector: Instituto de ingeniería y agronomía.

Tutor organizacional

Apellido y Nombres: Mg. Ing. Osio, Jorge

Correo electrónico: josio@unaj.edu.ar

Docente supervisor

Apellido y Nombres: Dr. Ing. Cappelletti, Marcelo

Correo electrónico: mcappelletti@unaj.edu.ar

Docente tutor del Taller de Apoyo para la Producción de Textos Académicos

Apellido y Nombres: Prof. Lic. Kelly, Carolina

Correo electrónico: kellygcarolina@gmail.com

Coordinador de la carrera de Ingeniería en Informática

Dr. Ing. Morales, Martín

martin.morales@unaj.edu.ar

Resumen

Este proyecto se realizó en el marco del Proyecto de Investigación de la Universidad Nacional Arturo Jauretche UNAJ INVESTIGA 2020 (Código del Proyecto 80020200300027UJ y Resolución Rectoral N° 183/21 de fecha 08/08/2021), cuyo título es "Tecnologías de IoT y aprendizaje automático para la solución de problemas en el medio productivo y el cuidado del medio ambiente", y consistió en el diseño y desarrollo de un sistema de control remoto de huertas, capaz de alertar al usuario de algún inconveniente, de manera que este pueda tomar medidas de contingencia tempranas ante cualquier problema. Esta necesidad surgió ante el ineficiente o nulo uso de tecnologías en la zonas aledañas a la UNAJ; la falta de información precisa, actual e histórica, de magnitudes climatológicas, del estado del suelo, del agua y del nivel de radiación solar; la insuficiente innovación tecnológica, dado que existe una carencia importante en cuanto a la implementación de la tecnología informática de avanzada; mano de obra sin adecuada calificación, ya que muchas veces el problema no es la ausencia de información para este sector, sino el desconocimiento de la tecnología existente y de la posibilidad de utilización de la información brindada por ella; la falta de una infraestructura adecuada; entre otras razones.

Por todo lo anterior, el objetivo del proyecto consistió en el diseño de un sistema con una arquitectura escalable, con capacidades de suplir la falta de tecnología en el campo y alertar al usuario ante cualquier inconveniente que se pueda presentar, fuera de los parámetros establecidos para sus cultivos. Además, también consistió en la creación de un sistema capaz de brindar al usuario información acerca de sus cultivos para poder, de este modo, prevenir con antelación cualquier problema que pueda atentar contra sus cultivos.

Las tareas del proyecto abarcaron: la investigación de una arquitectura capaz de crecer de modo fácil, con una base que pueda crecer de manera positiva a futuro; la implementación de un sistema de microcontroladores, en base a la arquitectura investigada; el diseño e implementación de un sistema web seguro y con fácil

integración de nuevos desarrollos en este; la implementación de un sistema de alarmas que avise al usuario cuando la información registrada se desvía de los parámetros establecidos del sistema.
Con este sistema se logró cubrir la falta de implementación de sistemas de tecnología para el área aledaña a la UNAJ, brindar un sistema de alarmas vía email para alertar al usuario ante inconvenientes, ofrecer al usuario la posibilidad de tener un sistema web que le permita analizar información de sus cultivos, y así prevenir y garantizar su cuidado ante problemas climatológicos futuros.

Palabras clave: **MQTT – Microcontroladores – IoT – Api – Alertas**

Abstract

This Project has been accomplished in a research program of Universidad Nacional Arturo Jauretche, UNAJ INVESTIGA 2020 (project code 80020200300027UJ and Rectoral Resolution N° 183/21, date: 08/08/2021), and it has consisted of the design and development of a control system for vegetable plots, where the end users receive alerts about events that take place in the vegetable plot or greenhouse. The motivation arises from the inefficient or null use of technology on the places around of UNAJ; The lack of accurate historical and current information, weather measurements, the ground capabilities, water, and the level of solar radiation; Inefficiency on innovation with cutting edge technology and the existence of a legacy infrastructure. personnel in this field are often not trained in theory and practice in new technologies, among other reasons.
The objective of the project consisted of the design and implementation of a system with scalable architecture, with capabilities of filling the gap on inexistent technologies on the field. The system will send alerts to the user preventing him about events that are out of the normal parameters with his vegetable plot or greenhouse.

The tasks for the project were: investigation about an architecture with the capability of easy scalability with new features, this was because is needed a scalable system in the future; the implementation of a microcontroller system based on the architecture mentioned with the capability of register temperature and humidity of the place; The design And development of a web system with the capacity to recollect all the information registered by the microcontroller system; Implement an alarm that sends an alert to the user when the microcontroller system register an anomaly.
With this system, has been achieved to cover the lack of use of technology infrastructure in places near to UNAJ. Has achieved to provide the user with an alarm system via email to alert him about anomalies, to give him the possibility of having a web system that allows him to analyze data on his vegetable plots, and thus prevent and guarantee quality in the face of future weather problems.

Keywords: **MQTT – Microcontrollers – IoT – Api – Alerts**

Dedicatorias y agradecimientos

Agradezco a mis tutores Jorge Osio y Marcelo Cappelletti por el acompañamiento durante la realización de la PPS, a la profesora Carolina Kelly por el apoyo y enseñanza brindada durante el desarrollo del presente informe.
Todo este trabajo se lo dedico a mi familia, principalmente a mi mamá y mi papá que sin ellos no sería quien soy hoy y parte de este desarrollo corresponde a ellos.
También, va una dedicatoria a mis amigos y compañeros de la universidad, que estuvieron apoyándome durante todo este transcurso y siempre estando presentes para brindarme su ayuda.
Gracias a todos por estar presentes, sin ustedes no hubiera llegado a este momento.

Índice

Resumen 2
Abstract 3
Dedicatorias y agradecimientos 4
Introducción 7
Objetivos 8
Marco teórico y espacio de trabajo 8
Importancia del proyecto 8
Metodología de trabajo 11
Herramientas 11
Sistema de microcontroladores 11
Sistema Web 13
Base de Datos 16
Desarrollo 16
Diseño y Arquitectura 16
Arquitectura sistema web 17
Arquitectura del sistema de microcontroladores 19
Protocolo MQTT 19
Desarrollo del sistema de microcontroladores 21
Partes de la arquitectura MQTT 21
Configuración del entorno 22
Configuración del entorno para desarrollo del *Publisher* 25
Broker (Raspberry PI 3) 35
Suscriber (Raspberry PI 3) 38
Desarrollo web 44
Desarrollo Back-End 45
Explicación de estructura básica del sistema: 49
Controlador app 53
Controlador "Usuarios" 58
Usuarios.controller.ts 59
Usuarios.module.ts 61
Usuarios.service.ts 63
Controlador "Nodos" 65

Nodos.controller.ts 66
Nodos.module.ts 67
Nodos.schema.ts y configNodos.schema.ts 68
Nodos.service.ts 69
Carpeta auth 72
LocalGuard y JwtGuard 74
Front End 76
Configuración del entorno de desarrollo 76
Carpeta servicios 80
informacionDeNodoService.js 80
informacionDeUsuarioService.js 81
Carpeta redux 81
¿Qué es Redux? 82
Carpeta components 85
inicioDeSesion 86
Integración 89
Conexión del Sistema web y la base de datos 91
Conexión del Sistema de microcontroladores y base de datos 92
get_database() 93
update_nodo(informacion) 94
insert_information_nodo(información) 94
insert_configuracion() 96
comprobarAlerta(informacion) 98
enviarMail(informacion) 98
Pruebas de funcionamiento y resultados 99
Pruebas de alertas del sistema de microcontroladores 100
Prueba del sistema web 101
Trabajo futuro 104
Versión actual 104
Versiones futuras 104
Conclusión 107
Apéndices 108
Apéndice 1 Instalación de “*Visual studio Code*”: 108
Índice de Figuras 109
Bibliografía 111

Introducción

En los últimos años han ocurrido avances tecnológicos que han permitido el desarrollo de aplicaciones que generaron un gran crecimiento productivo en diferentes áreas en donde la tecnología no estaba presente o no era primordial.

Con la llegada de la llamada "cuarta revolución industrial", se presentó el internet de las cosas (IoT), el cual es el mayor responsable de estas aplicaciones que generaron el crecimiento productivo mencionado. IoT nos permite conectar cualquier dispositivo a internet, ya sea un teléfono inteligente, un reloj, un equipo médico, automóviles, y cualquier dispositivo electrónico que se nos ocurra.

Haciendo uso de estas nuevas herramientas y observando el ineficiente o nulo uso de tecnología en las zonas aledañas a la UNAJ, en lo que corresponde al desarrollo productivo en el sector hortícola, se propuso el desarrollo de un sistema que sea capaz de mantener, prevenir de inconvenientes y mejorar la productividad en huertas, a campo abierto o invernaderos. Este sistema apunta a solucionar los problemas de falta de infraestructura tecnológica en el área y brindar información para realizar análisis y toma de decisiones para mitigar inconvenientes cotidianos del ámbito.

El sistema está capacitado para recolectar información de los invernaderos, almacenarla en una base de datos y exponérsela al usuario para que haga un análisis y actúe según corresponda. También, la aplicación cuenta con un sistema de alarmas que son enviadas al email del usuario cada vez que se registre un valor fuera de los parámetros de funcionamiento ideal para el tipo de cultivo que se posea en el invernadero.

En el presente documento se explicará el desarrollo del proyecto, las herramientas utilizadas, los inconvenientes surgidos, en qué fase se encuentra actualmente, las pruebas que se realizaron, el futuro de este y las conclusiones obtenidas en el instante actual.

Objetivos

El siguiente informe tiene por objetivo general el diseño y la implementación de una red de sensores que permita obtener información precisa de magnitudes climatológicas, del estado del suelo, del agua y ambiente, y en base a estos datos obtenidos generar información clara para el usuario y, en caso de ser necesario, generar alarmas que alerten a este sobre el estado actual de su invernadero a fin de poder mitigar o evitar cualquier inconveniente que pueda ocasionar la perdida de los cultivos.

Este desarrollo está destinado para el sector agroindustrial del área de influencia de la Universidad Nacional Arturo Jauretche (UNAJ), que cuenta con pequeños, medianos y grandes productores hortícolas, florícolas y frutícolas, fundamentalmente, en los cultivos intensivos, con los que se busca maximizar la producción en espacios reducidos, utilizando un solo tipo de producto.

El propósito es que el sistema, utilizando todos sus recursos, mejore la producción de cultivos en invernaderos de manera económica y con aplicación tecnológica. En este caso, la tecnología son los sensores y microcontroladores correspondientes.

Los sensores recolectarán la información y la enviarán a los microcontroladores, los cuales se encargarán de procesarla y guardarla en la base de datos para su futuro uso por el usuario del sistema.

Marco teórico y espacio de trabajo

Importancia del proyecto

La UNAJ se localiza en el partido de Florencio Varela, uno de los municipios con mayor superficie rural dentro del Conurbano Bonaerense. Actualmente, cuenta con, aproximadamente, unos 500 pequeños, medianos y grandes productores hortícolas, florícolas y frutícolas. En general, sus actividades productivas están basadas,

fundamentalmente, en los cultivos intensivos, con los que se busca maximizar la producción en espacios reducidos, utilizando un solo tipo de producto. En particular, dentro de la producción hortícola, las modalidades de producción son tanto a campo abierto como bajo cubierta (invernadero). Una de las características sobresalientes de esta zona rural es que limita con áreas de similares características, con los partidos vecinos, lo que transforma a la subregión en estratégica para la producción de alimentos frescos de origen vegetal, tanto para el consumo directo como para la industrialización. Tanto para la producción en una huerta a campo abierto como bajo cubierta, se pueden identificar en esta región una serie de problemas, entre ellos:

- Falta de información precisa, actual e histórica, de magnitudes climatológicas, del estado del suelo, del agua y del nivel de radiación solar;
- Insuficiente innovación tecnológica, dado que existe una carencia importante en cuanto a la implementación de la tecnología informática de avanzada;
- Mano de obra sin adecuada calificación, dado que muchas veces el problema no es la ausencia de información para este sector, sino el desconocimiento de la tecnología existente y de la utilización de la información brindada por ella;
- Falta de una infraestructura adecuada;
- Falta de capacitación en planificación y gestión;
- Vulnerabilidad en el proceso de comercialización.

Sin dudas, el extraordinario avance que vienen teniendo en los últimos años las Tecnologías de la Información y la Comunicación (TIC) permite pensar en diferentes estrategias que contrarresten estos inconvenientes. Al respecto, se han comenzado a emplear últimamente desarrollos basados en sistemas del Internet de las Cosas (IoT: *Internet of Things),* en una gran variedad de aplicaciones informáticas de gran impacto social, como en la telemedicina, la robótica, la seguridad del lugar y de las personas, entre otros diferentes tipos de servicios. En todos los casos, los sistemas IoT integran, por medio de redes interiores y exteriores de comunicación, la

tecnología electrónica, microelectrónica e informática, para gestionar las aplicaciones informáticas de manera inteligente, en pos de cumplir con los requerimientos deseados.

El concepto de IoT ha adquirido gran relevancia en los últimos años, debido a la posibilidad que ofrece de interconectar objetos entre sí a través de la red de internet. Si bien las redes de sensores basadas en IoT son conocidas, su implementación y desarrollo no han sido aun completamente explotados en determinados sectores, como, por ejemplo, para la optimización de recursos en el caso de los sectores agroindustriales. Se prevé que el uso de redes de sensores basados en IoT se intensifique en los próximos años, y es por eso por lo que se considera necesario el estudio, el diseño y la implementación de nuevos desarrollos que brinden soluciones a situaciones problemáticas de manera eficiente y, en la medida de lo posible, de bajo costo. En este sentido, el proyecto pretende contribuir a:

- Fortalecer la actividad en el área de las TIC que contribuyen a remediar las problemáticas existentes dentro del territorio de influencia de la UNAJ, en las áreas de medio ambiente, medio productivo y educación;
- Desarrollar un sistema autónomo y escalable que se encargue de minimizar los consumos de energía, juntamente con la implementación de un sistema informático que provea información precisa de magnitudes climatológicas, del estado del suelo y del ambiente;
- Implementar una red de sensores que permitan recabar información y concentrarla en una central/servidor para su procesamiento y la posterior generación de alarmas, acciones, etc.

Metodología de trabajo

Para la realización del sistema de control remoto de invernaderos se aplicó una metodología Kanban, la cual consiste en un tablero con las columnas: "qué hacer", "haciendo", "realizado".
Esto se efectuó con la aplicación en línea Trello, la cual permite la utilización de un tablero con tarjetas para desplegarlas en diferentes columnas. Este tablero se manipuló de manera que quede efectuado como la metodología Kanban, así, de este modo, se aprovecha el control que permite la metodología sobre las tareas que se deben realizar.

Herramientas

Las herramientas utilizadas para el desarrollo del proyecto se dividen, de igual modo que el proyecto, en tres grandes grupos.

Sistema de microcontroladores

En primer lugar, para la creación del sistema de microcontroladores se utilizó:

1. Sensor de temperatura DHT11.
2. Sensor de humedad de suelo HL-69.
3. Librería de DHT11.
4. Librería PubSubClient.
5. Librería ESP8266WiFi.
6. Librería Pymongo.
7. Microcontrolador Wemos D1.
8. Entorno de desarrollo Arduino.
9. Raspberry PI3 + Instalación de paquetes necesarios.

1- Sensor de temperatura DHT11:

Sensor que se encarga de tomar medidas de temperatura y humedad ambiente.

2- Sensor de humedad de suelo HL-69

Sensor que se encarga de captar la humedad contenida en el suelo, donde es colocado.

3- Librería de DHT11.

Para darle un correcto funcionamiento al sensor DHT11, se utilizó la librería DHT11 que contiene una lista de métodos para dar una mayor facilidad en su uso.

4- Librería PubSubClient.

Esta librería es una librería que permite enviar y recibir mensajes con el protocolo MQTT.

5- Librería ESP8266WiFi.

Esta librería contiene funcionalidades para facilitar la conexión de un microcontrolador a una red WiFi.

6- Librería Pymongo.

Esta librería contiene funcionalidades para facilitar la conexión a una base de datos de MongoDB y manipularla.

7- Microcontrolador Wemos D1

Microcontrolador programable (C/Arduino ide) donde se pueden conectar los sensores para su correspondiente programación. Cuenta con un módulo integrado de WiFi ESP8266, que nos permite conectarnos a la red para enviar los datos capturados por los sensores.

8- Entorno de desarrollo Arduino ide.

Entorno de desarrollo que ofrece herramientas para facilitar la programación de microcontroladores y posee una completa compatibilidad con el microcontrolador Wemos D1.

9- Raspberry PI4/Raspberry PI3 + instalación de paquetes necesarios

Microordenador utilizado para la recepción, procesamiento y envío de la información de los sensores recibida, por medio de los microcontroladores, a la base de datos. Este microordenador tiene instalado un sistema operativo Raspberry PI OS (adaptación de Debian).

Además, se instalaron paquetes necesarios para poder dar funcionalidad al protocolo MQTT y, para poder configurar la Raspberry como *Broker* (más adelante se dará la explicación sobre qué es un bróker), se tuvo que instalar el paquete Mosquitto, el cual permite configurar la Raspberry como este tipo indicado.

También se instaló Python 3 para la creación del programa que funciona como *Suscriber* (más adelante se da la explicación de que es un *Suscriber*), el cual se ejecuta sobre la misma Raspberry.

Sistema Web

Para el sistema web se utilizaron diversas herramientas que se adaptan de buena manera a un sistema Api REST: esto se determinó de este modo, ya que se buscó que la aplicación fuera lo más escalable posible.

Como consecuencia de este tipo de aplicación, el sistema web se divide en dos partes: *Front-End* y *Back-End*.

El *Back-End* está compuesto solo por el Framework NestJS, que ya posee diferentes herramientas para la construcción de esta parte del sistema web (que se explicarán más en detalle en la sección “Desarrollo Web”).

Herramientas y lenguajes de programación:

- React.
- Ant-Design.
- Visual Studio Code.
- NestJS.
- Postman.
- Heroku.
- NPM.
- Git con GitHub.
- Draw.IO.

React

Biblioteca JavaScript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página.

Ant-Design

Es un kit de interfaz de usuario que contiene componentes ya creados para la utilización en los desarrollos de páginas web con React.

Visual Studio Code

Es un IDE (entorno de desarrollo integrado) que se ejecuta en Windows, macOS y Linux. Sirve para el desarrollo de aplicaciones y cuenta con herramientas integradas para facilitar el proceso de desarrollo.

NestJS

Es un *Framework* basado en angular que permite la creación de aplicaciones NodeJS y se programa en TypeScript. En este caso, se lo utiliza para la creación de la Api que consumirá el *Front-End.*

Postman

Es una plataforma API para realizar diseños y pruebas sobre las Apis creadas. Esta plataforma la utilizamos para probar el funcionamiento de la Api sin hacer uso del *Front-End.*

Heroku

Heroku es una plataforma de servicios en la nube (conocidos como PaaS o *Platform as a Service*) que permite manejar los servidores y sus configuraciones, el escalamiento y la administración. La utilizamos para publicar la aplicación de Front-End y la Api en la web.

NPM

NPM (*Node Package Manager*) es el sistema de gestión de paquetes utilizado para la importación de módulos y funcionalidades en nuestra aplicación de *Front-End y Back-End.*

Git con GitHub

Git es un sistema de control de versiones pensado en la eficiencia, la confiabilidad y la compatibilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.

Por otra parte, GitHub es un portal creado para alojar proyectos, utilizando el sistema de control de versiones Git.

Fue utilizado para alojar tanto la aplicación Front-End como la aplicación Back-End.

Draw.IO

Draw.IO es un sistema web que permite realizar diferentes tipos de diagramas y diseños de aplicaciones, y fue utilizado para la creación de los diagramas generales, los diagramas de flujo y el diseño del sistema.

Base de Datos

Para la base de datos, se utilizó la herramienta Mongo Atlas, la cual nos permite configurar un clúster con instancias de bases de datos MongoDB en la nube. Para este caso, solo fue necesario la utilización de una sola instancia.

Desarrollo

Para llevar a cabo la creación del sistema de control de invernaderos se tuvieron que hacer grandes divisiones. Estas fueron realizadas para una mejor administración del proyecto ya que este contaba con tres partes bien definidas: el sistema de microcontroladores, el sistema web y el almacenamiento en la nube.
Sin embargo, para poder comprender estas tres divisiones, hay que comprender cómo interactúan entre ellas, para crear el producto final, que es el sistema de control de invernaderos.

Diseño y Arquitectura

El sistema de control de invernaderos remoto es una aplicación web que permite monitorear invernaderos para asegurar la correcta producción de los cultivos. Esta aplicación web, además de brindar un monitoreo de los invernaderos y sus sectores, también debe ofrecer alarmas para que los usuarios implementen mecanismos de acción correctiva ante algún desvío o incidente producido en el mismo invernadero.
Para ello se debió diseñar el sistema de manera que pueda ser capaz de:

- Monitorear
- Incluir sistemas de control
- Ser escalable

A partir de estas consideraciones, se decidió que la arquitectura ideal para la aplicación web estuviera compuesta por una página web que consume una Api

REST, a través de peticiones GET o POST, que le devuelve la información solicitada de la base de datos. Además, la Api tiene que estar capacitada para recibir las peticiones de alguna acción correctiva necesaria, por ejemplo, el activado de riego de un invernadero.

No obstante, implementar este diseño para la aplicación web no define todo el sistema de control de invernaderos. En lo que respecta al sistema de microcontroladores, se definió una arquitectura a partir del protocolo MQTT donde los microcontroladores son los *Publisher* y la Raspberry es el *Broker* y el *Suscriber*. También, se definió que la base de datos tiene que estar contenida en la nube para facilitar el consumo y la publicación de ambas partes del sistema de control de invernaderos.

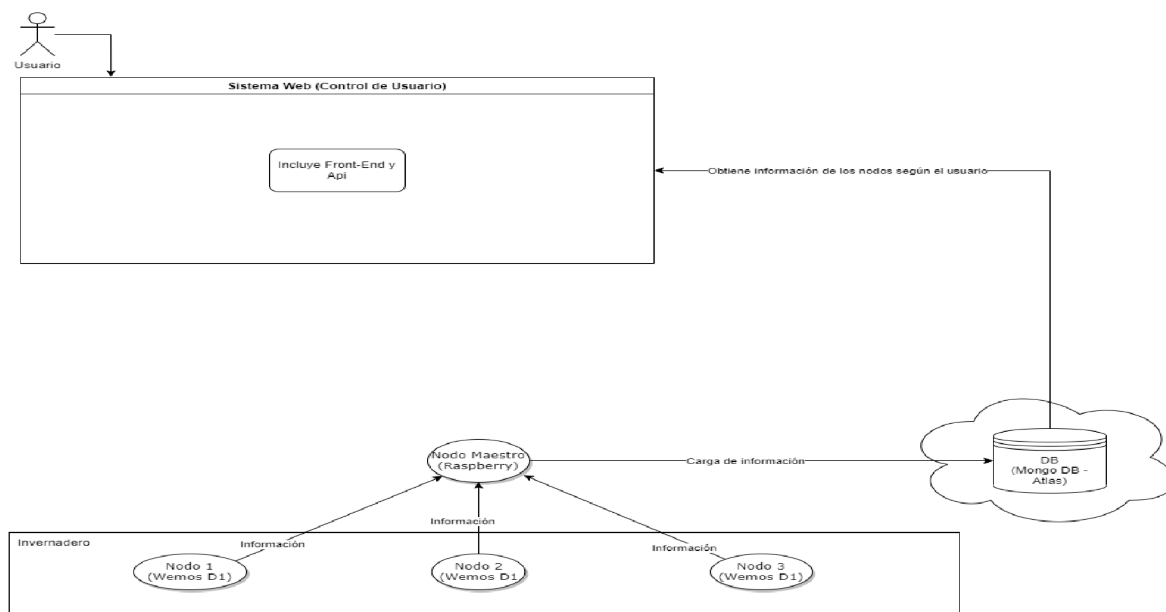

Figura 1 - Arquitectura general del sistema.
Fuente: Elaboración propia.

Arquitectura sistema web

La arquitectura implementada para el sistema web sigue una desconstrucción de las partes donde el *Front-End* (parte que ve el usuario) y *Back-End* (Lógica del sistema) están separados en diferentes aplicaciones. Esto se debe a que, de esta manera, se obtendrá una mayor escalabilidad en la aplicación para el crecimiento a

futuro y una mejor estabilidad. Además, este tipo de sistemas brinda una mejor seguridad de la aplicación, lo que generará que el sistema esté, constantemente, en correcto funcionamiento.

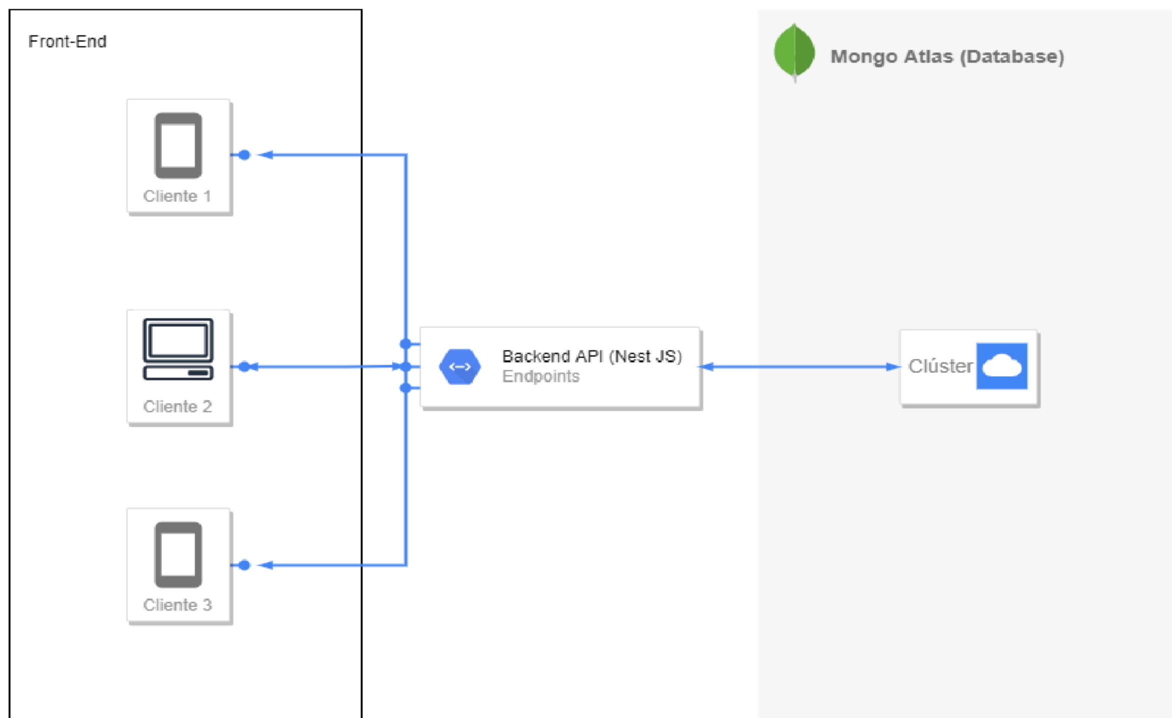

Figura 2 - Arquitectura del sistema web.
Fuente: Elaboración propia.

Según la arquitectura descripta en la "*Figura 2 - Arquitectura del sistema web*", el proceso de una consulta se da de la siguiente manera: un cliente ingresa a la página web con "Usuario" y "Contraseña"; este paso ya ejecuta la primera consulta a la Api, donde el sistema del *Front-End* crea una *Request Post*, en la cual estará empaquetado el usuario y la contraseña. Esta *Request Post* es procesada por la Api y se encarga de comparar esta información con la que está en la Base de Datos en la Nube, es decir, si este "Usuario" y "Contraseña" son válidos, la base de datos confirmará dicha pregunta que le hace la Api y devolverá la información que

corresponda al momento a la Api, que procesará esta información y se la enviará al *Front-End*.
El proceso descripto anteriormente es el modo en que interactúan las dos partes del sistema web y la base de datos.

Arquitectura del sistema de microcontroladores

En el sistema de microcontroladores se implementó una arquitectura con el protocolo MQTT. Para comprenderla, primero hay que explicar qué es el protocolo MQTT.

Protocolo MQTT

MQTT son las siglas de *MQ Telemetry Transport*, aunque en primer lugar fue conocido como *Message Queing Telemetry Transport*. Es un protocolo de comunicación M2M (*machine-to-machine*) de tipo *message queue* y está basado en la pila TCP/IP, como base para la comunicación. En el caso del MQTT, cada conexión se mantiene abierta y se "reutiliza" en cada comunicación. Es una diferencia, por ejemplo, con una petición HTTP 1.0 donde cada transmisión se realiza a través de cada conexión.
El funcionamiento del MQTT es un servicio de mensajería *push* con patrón publicador/suscriptor (*publisher-subscriber*). En este tipo de infraestructuras, los clientes se conectan con un servidor central denominado *Broker*.
Para filtrar los mensajes que son enviados a cada cliente, estos se disponen en *topics* organizados jerárquicamente. Un cliente puede publicar un mensaje en un determinado *topic*, otros clientes pueden suscribirse a este *topic*, y el *Broker* le hará llegar los mensajes a quienes están suscritos allí. De este modo, los clientes inician una conexión TCP/IP con el *Broker*, obtienen la información, y el servidor se encarga

de mantener un registro de la conexión del cliente, que se mantiene abierta hasta que aquel la finaliza.

Explicado el funcionamiento general del protocolo MQTT, la arquitectura empleada queda de la siguiente manera:

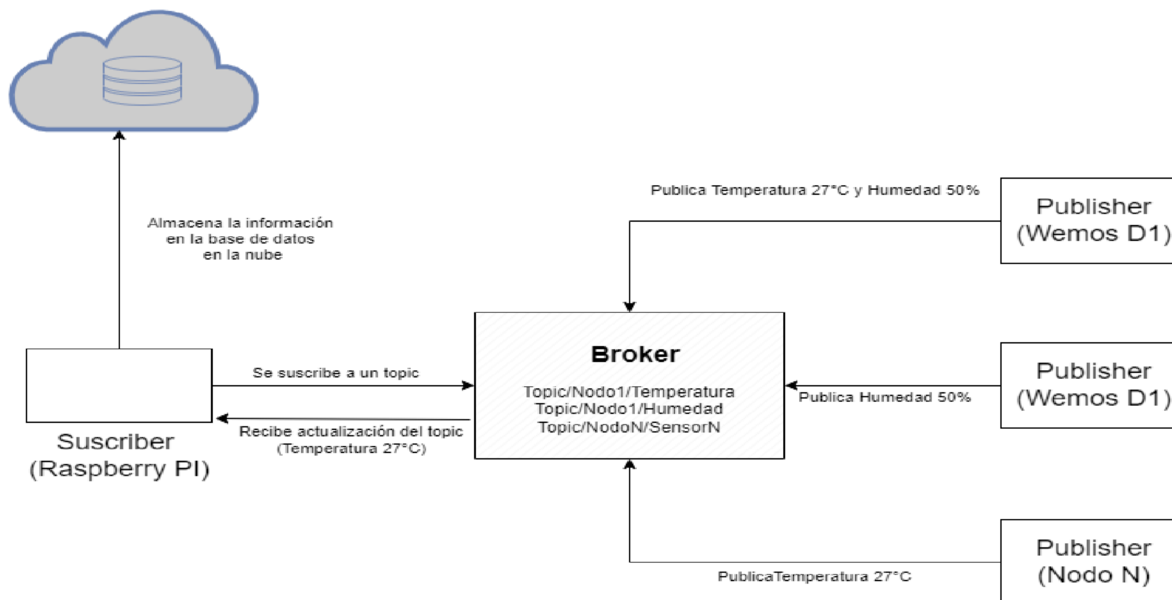

Figura 3 - Arquitectura MQTT del sistema de microcontroladores.

Fuente: Elaboración propia.

Como ya se explicó, esta arquitectura sigue el siguiente proceso: cada nodo, que es un microcontrolador (*Publisher*), tiene conectado diferentes tipos de sensores, en este caso un DHT11 (sensor de temperatura y humedad ambiente) y un HL-69 (sensor de humedad del suelo), y, a su vez, se conecta a través de WiFi al servidor (*Broker*). Se registra como *Publisher* y en qué *topics* va a publicar.

El *Broker* (*Raspberry PI*) es el servidor donde están creados los *topics* y a qué *Publisher* pertenecen, esto quiere decir que el cliente *Publisher* va a publicar en ese *topic*. Cada vez que un cliente *Publisher* actualiza su *topic*, el *Broker* da alerta al *Suscriber* que está suscrito a ese *topic*.

El *Suscriber* (*Raspberry PI*) se encarga de obtener las actualizaciones de los *topics* a los cuales está suscrito y de almacenarla en la base de datos.

Para que este proceso se dé correctamente, previamente, tanto los *Suscribers* como los *Publishers* se deben conectar en el *Broker* con “usuario” y “contraseña” configurados por el *Broker*. En base a esa configuración, se garantiza que el sistema funcionará de manera segura, ya que no se conectarán al *Broker* nodos que no pertenezcan al sistema de microcontroladores.
Por otra parte, tanto el *Broker* como el *Suscriber* pueden convivir perfectamente en la misma *Raspberry PI*, por lo que se empleó incluirlos juntos, ya que son dos aplicaciones que funcionan con programas diferentes: el *Suscriber* con la aplicación creada en Python y el *Broker*, con *Software* Mosquitto.

Desarrollo del sistema de microcontroladores

La implementación del sistema de microcontroladores se llevó a cabo en base al protocolo MQTT, como se explicó en la sección de “Diseño y Arquitectura”. Pero, para poder poner en práctica este protocolo, es necesario crear las configuraciones generales previas a la configuración y la programación de cada microcontrolador y microordenador.

Partes de la arquitectura MQTT

Como ya se explicó en la sección de “Diseño y Arquitectura”, el sistema de comunicación MQTT contiene tres componentes, los cuales, cada uno, cumplen un rol especifico.

Publisher: Es quien genera y envía la información al *Broker*, es decir, los *Publishers* son cada uno de los microcontroladores que contienen sensores dentro de un invernadero. En este caso, la información que enviarán los *Publishers* será cada cinco minuto.

Broker: Es como un servidor que colecta la información enviada de cada uno de los *Publisher* en el sistema. Además, guarda la información y la distribuye a cada uno de los suscriptores. Cabe destacar que toda la información enviada por un *Publisher* no implica que un suscriptor esté suscripto completamente a ella, ya que, en nuestro caso, cada sensor programado sobre un microcontrolador (*Publisher*) estará diferenciado de otro y, por ende, un suscriptor podría estar suscripto solo a un sensor y no a todos los sensores que posee el microcontrolador.

Suscriber: Como ya se indicó en el *Broker*, un *Suscriber* es un componente que se suscribe a un tipo de mensaje que es enviado de un *Publisher*. Esta suscripción la realiza a través del *Broker* y puede ser parcial o completa, dependiendo de qué es lo que se quiere obtener.

Explicadas las partes básicas que componen a este tipo de comunicación, podemos explicar cuáles son los que componen a nuestro sistema y cuáles de estas partes representan a cada uno.
Para el *Suscriber* y el *Broker*, se utilizó la misma *Raspberry PI 3*, y para el *Publisher* se utilizó un Wemos D1 (microcontrolador basado en Arduino que tiene un módulo ESP8266 integrado).

Configuración del entorno

Para configurar el entorno, hay que definir una serie de variables necesarias para cada uno de los componentes que están implicados en un sistema con el protocolo MQTT, *Publisher, Broker y Suscriber.*

Para el *Publisher*:

- IP del MQTT *Broker*.
- Usuario MQTT (para que el *Broker* lo identifique).
- Contraseña MQTT (para que el *Broker* lo identifique).

- Id de cliente MQTT (identificador que lo diferencia de otros *Publishers* en el *Broker*).
- *Topics* que posee.
- SSID del WiFi al que se va a conectar.
- Contraseña del WiFi al que se va a conectar.

Para el *Broker*:

- IP del MQTT *Broker*
- Mosquitto.conf (archivo de configuración)

Para el *Suscriber*:

- IP del MQTT *Broker*.
- *Topics* del *Publisher* a los cuales está suscripto.
- Usuario MQTT.
- Contraseña MQTT.

Ya definidas qué variables son necesarias, se procede primero a configurar el primer *Publisher*, el Wemos D1.

Publisher (Wemos D1)

Antes de programar el *Publisher* (Wemos D1), debemos definir cómo conectaremos los sensores para programarlos, por lo tanto, debemos conocer cómo está compuesto el Wemos D1, qué conexiones posee, y qué necesitamos para poder programarlo. Como el Wemos D1 ya posee integrado un módulo ESP8266 (módulo WiFi), solo debemos preocuparnos por las conexiones de los sensores.

DHT11:

Apreciando las conexiones del Wemos D1 (ver "*Figura 4 – Conexiones de Wemos D1*"), y las conexiones del DHT11 (ver "*Figura 5 – Conexiones de sensor DHT11*"), debemos conectar el pin de datos del DHT11 al pin digital D4 del Wemos; el pin Vcc

del DHT11 a la salida de 3,3V del Wemos y el pin de GND del DHT11 a algunas de las salidas GND que posee el Wemos.

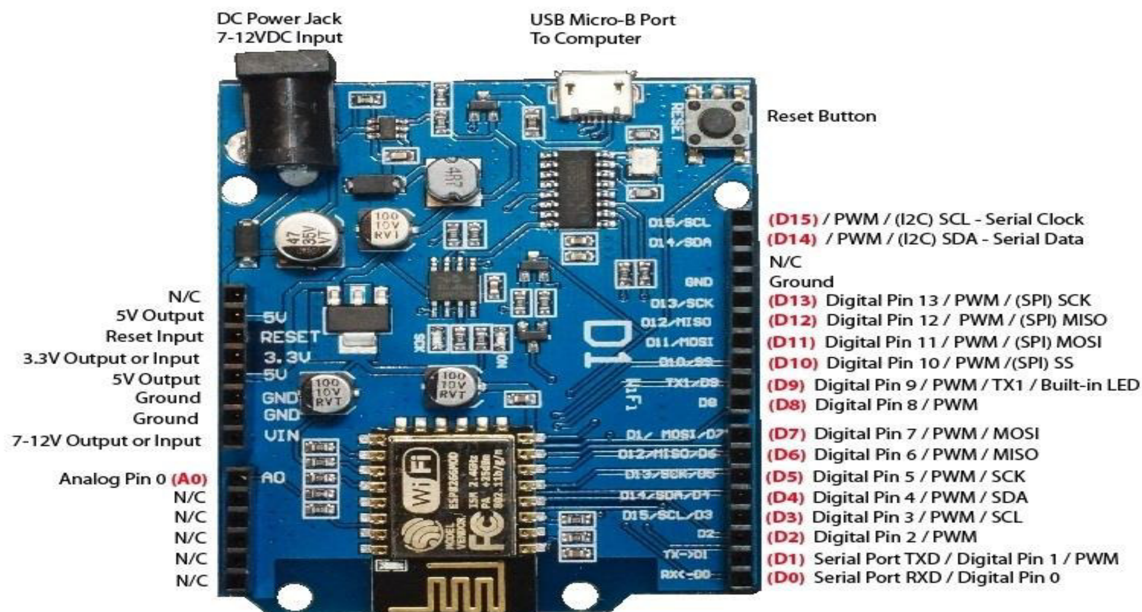

Figura 4 - Conexiones de Wemos D1.

Fuente: Disponible en https://www3.gobiernodecanarias.org/medusa/ecoblog/rsuagued/files/2020/01/esp8266-d1-r1.jpg.

Fecha de consulta: 15/05/2022

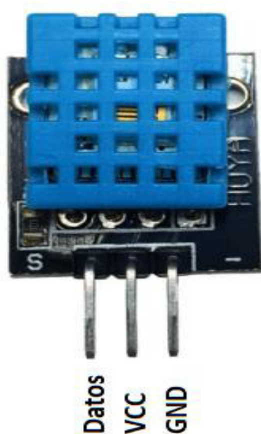

Figura 5 - Conexiones de sensor DHT11.

Fuente: Disponible en: https://www.profetolocka.com.ar/2020/12/29/micropython-midiendo-temperatura-y-humedad-con-el-dht11/.

Fecha de consulta: 15/05/2022

HL-69:

Para conectar los pines del HL-69 (ver "*Figura 6: Conexiones del sensor HL-69*"), se definió, para el pin Vcc del HL-69, el pin de salida de 5V del Wemos; para el pin de GND, se seleccionó una de las otras salidas de GND que posee el Wemos y, para el pin A0 del HL-69, se seleccionó el pin analógico 0 del Wemos (A0).

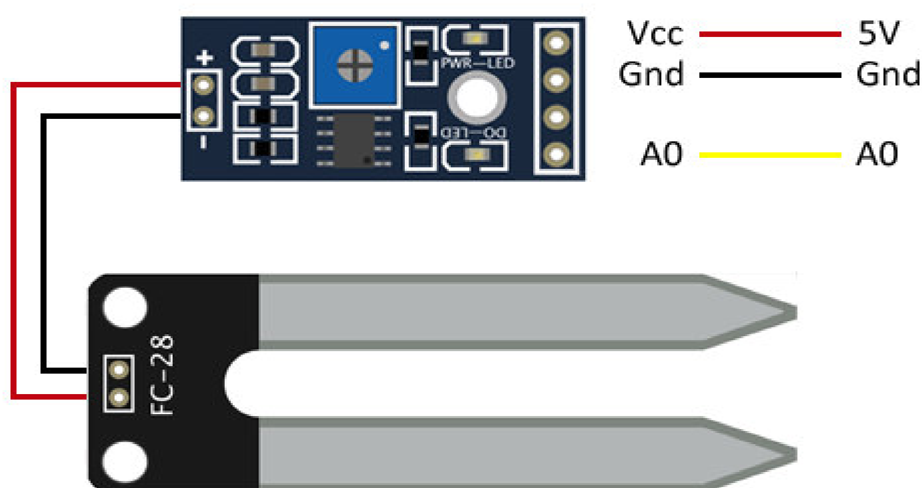

Figura 6 - Conexiones del sensor HL-69.

Disponible en: https://grupoelectrostore.com/shop/sensores/temperatura/modulo-sensor-de-humedad-y-temperatura-de-suelo-fc-32-higrometro/.

Fecha de consulta: 15/05/2022.

Configuración del entorno para desarrollo del *Publisher*

Ya definidas las conexiones de los sensores sobre el Wemos D1, procedemos a la explicación del código del *Publisher*.

En primer lugar, es necesario tener instalado el entorno de desarrollo para Arduino (https://www.arduino.cc/en/software), que hay que descargar e instalar.

Segundo, debemos ingresar al gestor de placas del entorno de desarrollo de Arduino y agregar la placa Wemos D1 (ver "*Figura 7 – Cómo agregar una nueva placa en entorno de desarrollo Arduino*"). Para esto, ingresamos en "*Herramientas*

-> Gestor de Tarjetas" y en el filtro de búsqueda buscamos ESP8266 e instalamos la distribución que aparece de *ESP8266 Community*. Una vez realizado esto, la placa Wemos D1 figuraría para poder ser seleccionada desde "*Herramientas -> Placa -> ESP8266 -> LOLIN (WeMos D1 R1)".* Seleccionada esta placa y conectada a la computadora, ya se va a poder cargar códigos para su programación.

Ya agregada la placa necesaria para que se puedan cargar los códigos de programación, es necesario agregar también las librerías de los sensores y los módulos para utilizar sus funcionalidades (ver "*Figura 8 - Cómo instalar una nueva librería en el entorno de desarrollo Arduino*").

Para agregar una librería se necesita ingresar a "*Programa -> Incluir librería -> Administrar bibliotecas…"*. En el buscador que aparece en esa sección, se tienen que buscar las librerías de DHT11, PubSubClient y ESP8266WiFi e instalarlas. Una vez realizado esto, ya se podrán utilizar las funcionalidades que incluyen estas librerías para los sensores y módulos.

Ya configurado todo el entorno para su programación, procedemos a la explicación del respectivo código del *Publisher* el cual podemos ver en la "*Figura 9 – Primera parte del código de programación del Publisher*" y en la *"Figura 10 – Segunda parte del código de programación del Publisher*".

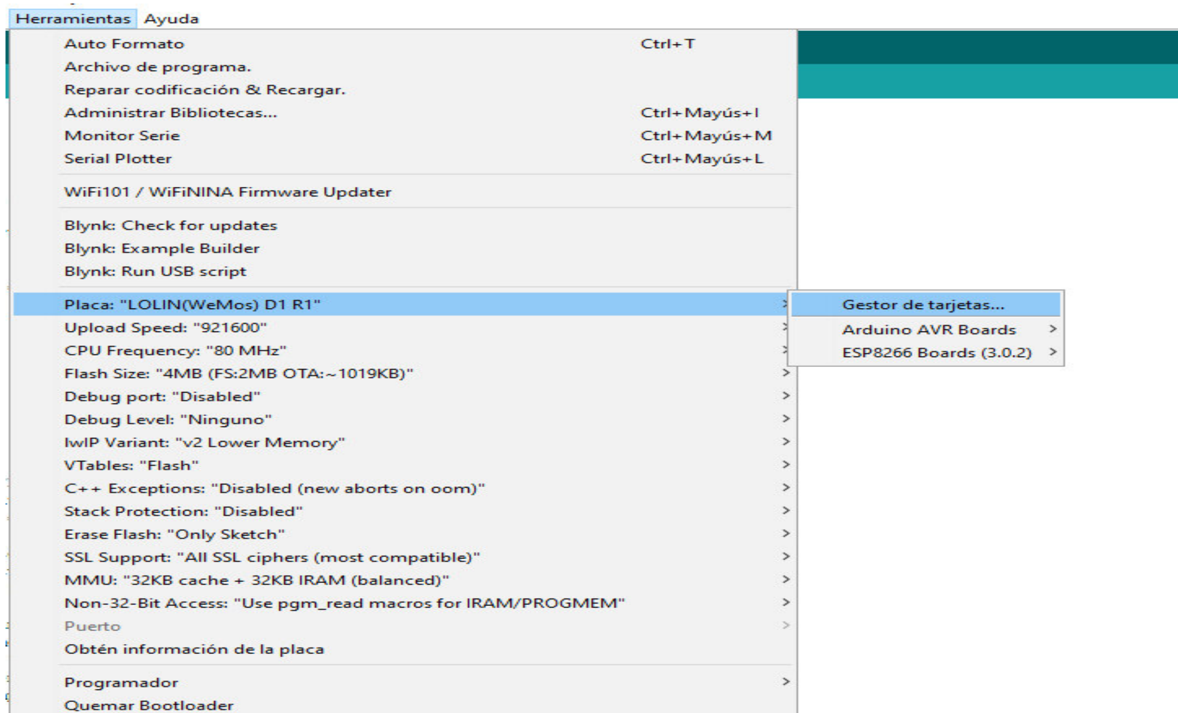

Figura 7 - Cómo agregar una nueva placa en entorno de desarrollo Arduino.

Fuente: Elaboración propia.

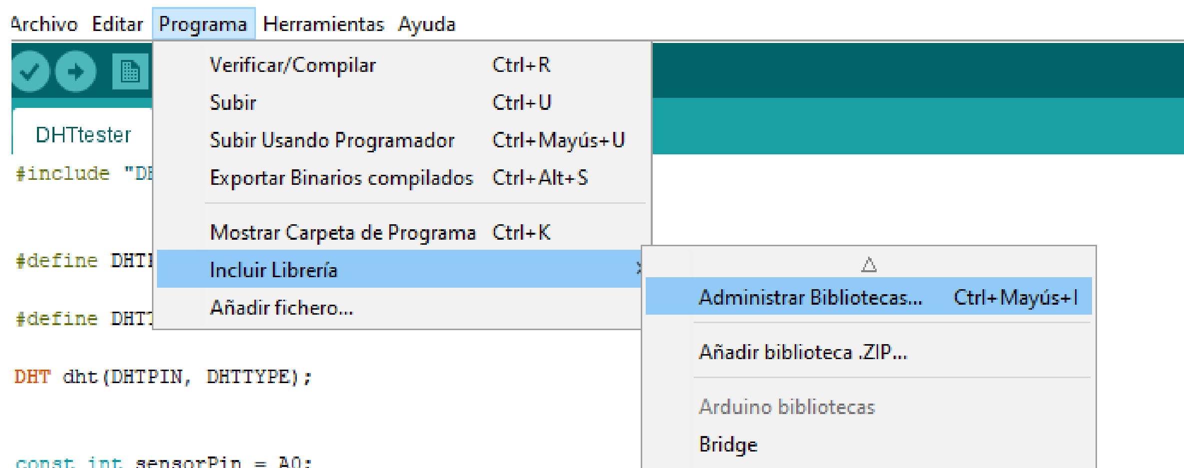

Figura 8 - Cómo instalar una nueva librería en el entorno de desarrollo Arduino.

Fuente: Elaboración propia.

```
1
2  //****************************************************************************************//
3  //**** CODIGO DE PROGRAMACION DEL PUBLISHER (NODO1) EL CUAL ESTA HECHO PARA EL WEMOS D1 ****//
4  //**** AUTOR: Leonel Navarro... LeNaJosh                                               ****//
5  //**** Año: 2022                                                                       ****//
6  //****************************************************************************************//
7
8
9  #include "DHT.h"          //libreria del sensor DHT11
10 #include "PubSubClient.h" // libreria para conectar y publicar en el Broker
11 #include "ESP8266WiFi.h"  // Libreria del ESP8266
12
13
14 #define DHTPIN 4          // Pin digital (D4) donde se encuentra conectado el DHT11
15 #define DHTTYPE DHT11     // Definicion del tipo de dht que se va a utilizar, en este caso DHT11
16 DHT dht(DHTPIN, DHTTYPE); //Inicializacion del sensor dht, se le pasan por parametro el pin donde se encuentra conectado y que tipo de sensor es
17
18 const int humedadSueloPin = A0;                                          //Pin analogico a donde va a estar conectado el sensor de humedad de suelo HC-69
19
20 // WiFi
21 const char* ssid = "RiverPlate";                // SSID de red wifi a conectar
22 const char* wifi_password = "River2020+-";      // Password de red WiFi a conectar
23
24 // MQTT
25 const char* mqtt_server = "192.168.1.63";                                // IP del MQTT broker (ip de la raspberry)
26 const char* humedadAmbiente_topic = "invernadero1/nodo1/humedadAmbiente";     // Topic de humedad ambiente del Nodo1
27 const char* temperaturaAmbiente_topic = "invernadero1/nodo1/tempertura";     // Topic de temperatura ambiente del Nodo1
28 const char* humedadSuelo_topic = "invernadero1/nodo1/humedadSuelo";          // Topic de humedad suelo del Nodo1
29 const char* mqtt_username = "huertabroker";                              // MQTT usuario (huertabroker)
30 const char* mqtt_password = "huerta1";                                   // MQTT password (huerta1(uno))
31 const char* clientID = "client_invernadero1";                            // MQTT client ID
32
33
34 //Conexion al broker
35
36 WiFiClient wifiClient;                                                   // Inicializa el WiFi
37 PubSubClient client(mqtt_server, 1883, wifiClient);                      // Inicializa el cliente MQTT, 1883 es el puerto definido como listener en la configuracion del Broker
38
39
40 // Funcion para conectar al MQTT Broquer a traves de WiFi
41 void connect_MQTT() {
42   Serial.print("Conectando a");
43   Serial.println(ssid);
44
45   // Inicializacion de la conexion al WiFi
46   WiFi.begin(ssid, wifi_password);
47
48   // Espera hasta que la conexion sea exitosa antes de continuar
49   while (WiFi.status() != WL_CONNECTED) {
50     delay(100);
51     Serial.print(".");
52   }
53
54   // Debugging - Salida de la direccion IP a la que nos conectamos
55   Serial.println("WiFi connected");
56   Serial.print("IP address: ");
57   Serial.println(WiFi.localIP());
58
59   // CONEXION AL MQTT BROKER
60   // client.connect, nos devuelve un valor booleano que nos dice si la conexion con el MQTT Broker fue hecha correctamente
61   // Si la conexion falla, te indica si estas usando el usuario y contraseña correcto al Broker que te queres conectar
62   if (client.connect(clientID, mqtt_username, mqtt_password)) {
63     Serial.println("Conectado al Broker MQTT del Invernadero");
64   }
65   else {
66     Serial.println("La conexion al Broker MQTT falló...");
67   }
68 }
69
70
71 void setup() {
72   Serial.begin(9600);       //Inicializacion del monitor serie (para debugging)
73   dht.begin();              //Inicializacion del sensor dht11
74 }
75
```

Figura 9 - Primera parte del código de programación del Publisher.

Fuente: Elaboración propia.

```
void loop() {
  connect_MQTT();                                         //llama a la funcion que realiza la conexion del publisher al broker mqtt
  Serial.setTimeout(2000);

  float humedadAmb = dht.readHumidity();                  // leo la humedad del sensor dht11
  float tempreaturaAmb = dht.readTemperature();           // leo la temperatura del sensor dht11
  int humedadSuel = analogRead(humedadSueloPin);          // leo la humedad del suelo del sensor HC-69

  Serial.print("Humedad ambiente: ");
  Serial.print(humedadAmb);
  Serial.println(" %");
  Serial.print("Temperatura Ambiente: ");
  Serial.print(tempreaturaAmb);
  Serial.println(" *C");
  Serial.print("Humedad de suelo: ");
  Serial.print(humedadSuel);
  Serial.println(" %");


  // MQTT solo transmite datos de tipo string
  String hs = "Humedad ambiente: " + String((float)humedadAmb) + " % ";
  String ts = "Temperatura ambiente: " + String((float)tempreaturaAmb) + " C ";
  String humedadSensorSueloString = "Humedad Suelo: " + String((int)humedadSuel) + "% ";



//PUBLICACION DE INFORMACION TEMPERATURA AMBIENTE

  // PUBLICA en el MQTT Broker (topic = "invernadero1/nodo1/tempertura" definido al principio)
  if (client.publish(temperaturaAmbiente_topic, String(tempreaturaAmb).c_str())) {
    Serial.println("¡Temperatura enviada!");
  }

  // de nuevo, client.publish devuelve un booleano si la informacion se publica en el broker o no
  // si el mensaje falla de ser publicado, intentara de nuevo hasta que la conexion sea terminada
  else {
    Serial.println("Fallo al enviar temperatura. Reconectando al MQTT Broker e intentando nuevamente.");
    client.connect(clientID, mqtt_username, mqtt_password);
    delay(10);                                            //Este delay asegura que el cliente.publish no choque con la llamada de client.connect
    client.publish(temperaturaAmbiente_topic, String(tempreaturaAmb).c_str());
  }




//PUBLICACION DE INFORMACION HUMEDAD AMBIENTE

  // PUBLICA en el MQTT Broker (topic = "invernadero1/nodo1/humedadAmbiente" definido al principio)
  if (client.publish(humedadAmbiente_topic, String(humedadAmb).c_str())) {
    Serial.println("¡Humedad ambiente enviada!");
  }

  // de nuevo, client.publish devuelve un booleano si la informacion se publica en el broker o no
  // si el mensaje falla de ser publicado, intentara de nuevo hasta que la conexion sea terminada
  else {
    Serial.println("Fallo al enviar humedad ambiente. Reconectando al MQTT Broker e intentando nuevamente.");
    client.connect(clientID, mqtt_username, mqtt_password);
    delay(10);                                            //Este delay asegura que el cliente.publish no choque con la llamada de client.connect
    client.publish(humedadAmbiente_topic, String(humedadAmb).c_str());
  }




//PUBLICACION DE INFORMACION HUMEDAD SUELO


    // PUBLICA en el MQTT Broker (topic = "invernadero1/nodo1/humedadSuelo" definido al principio)
  if (client.publish(humedadSuelo_topic, String(humedadSuel).c_str())) {
    Serial.println("¡Humedad del suelo enviada!");
  }

  // de nuevo, client.publish devuelve un booleano si la informacion se publica en el broker o no
  // si el mensaje falla de ser publicado, intentara de nuevo hasta que la conexion sea terminada
  else {
    Serial.println("Fallo al enviar humedad del suelo. Reconectando al MQTT Broker e intentando nuevamente.");
    client.connect(clientID, mqtt_username, mqtt_password);
    delay(10);                                            //Este delay asegura que el cliente.publish no choque con la llamada de client.connect
    client.publish(humedadSuelo_topic, String(humedadSuel).c_str());
  }

  client.disconnect();                                    // Desconexion del MQTT Broker
  delay(3000 * 100);                                       // imprime nuevos valores cada a 5 minutos
}
```

Figura 10 - Segunda parte del código de programación del Publisher.

Fuente: Elaboración propia.

Explicación del código:

Las primeras tres líneas que arrancan con *"#include"* son las librerías que estamos importando para poder utilizar las funcionalidades que nos van a facilitar el uso de los módulos y sensores.
Luego las siguientes tres líneas son:

- *#define DHTPIN 4*
- *#define DHTTYPE DHT11*
- *DHT dht(DHTPIN, DHTTYPE);*

Las primeras dos líneas son para definir el pin de conexión y el tipo de sensor de temperatura ambiente que estamos utilizando. En la tercera línea (*DHT dht(DHTPIN,DHTTYPE);*) se ejecuta una de las funciones importadas de la librería (DHT.h), la cual inicializa una instancia del sensor en que el pin y tipo de sensor definido en las dos líneas anteriores se pasan por parámetro. Seguido a esto viene la línea:

- *const int humedadSueloPin = A0*

que muestra en qué pin analógico se encuentra conectado el higrómetro (sensor de humedad de suelo HL-69) y guarda el valor en la variable "*humedadSueloPin"*.
Luego se definen dos variables que contienen el SSID y la contraseña del WiFi al que se va a conectar el *Publisher:*

- *const char* ssid = "RiverPlate"*
- *const char* wifi_password = "River2020+-"*

Luego de definir eso, se definen entonces todas las variables referidas a MQTT, las cuales están indicadas en las siguientes líneas:

- *const char* mqtt_server = "192.168.1.34";*
- *const char* humedadAmbiente_topic = "invernadero1/nodo1/humedadAmbiente";*
- *const char* temperaturaAmbiente_topic = "invernadero1/nodo1/tempertura";*
- *const char* humedadSuelo_topic = "invernadero1/nodo1/humedadSuelo";*
- *const char* mqtt_username = "huertabroker";*
- *const char* mqtt_password = "huerta1";*
- *const char* clientID = "client_invernadero1";*

siendo que:

- mqtt_server es la IP del *Broker* al que nos vamos a conectar.
- humedadAmbiente_topic es el *topic* para la humedad ambiente.
- temperaturaAmbiente_topic es el *topic* para la temperatura ambiente.
- humedadSuelo_topic es el *topic* para la humedad del suelo.
- mqtt_username es el nombre de usuario para conectarte al *Broker*.
- mqtt_password es la contraseña para conectarte al *Broker*.
- clientID es el identificador del cliente que se conecta al *Broker*.

Los valores de mqtt_username y mqtt_password son definidos y configurados en el *Broker*, y el *Publisher* debe tener conocimiento de estos valores para poder conectarse.
Ya definidas todas las variables referidas al MQTT *Broker*, nos conectamos con las siguientes dos líneas:

- *WiFiClient wifiClient;*
- *PubSubClient client(mqtt_server, 1883, wifiClient);*

Utilizando la librería “PubSubClient.h” para facilitarnos la conexión al *Broker*, hacemos uso de la función *client* y le pasamos, por parámetro, la IP del *Broker* al que nos queremos conectar, el puerto y el cliente WiFi.

Todas estas líneas explicadas hasta el momento son definiciones de variables, e iniciación e importación de librerías. A partir de ahora, todo el código está definido en tres funciones: la primera es *void connect_MQTT()*, que será utilizada dentro de la función *loop*, y las otras dos son *void setup()* y *void loop(),* que son las funciones clásicas de programación en microcontroladores de tipo Arduino.
La función *void setup()* es para definir, dentro de ella, alguna configuración e inicialización extra y, en la función *void loop()*, está definida toda la lógica de lo que hará el microcontrolador. Dentro de esta función, se encuentra el llamado a la función *void connect_MQTT().* De este modo, esta función se llamará cada ciclo de ejecución de la función *void loop()* y pondrá en marcha su funcionalidad.

Función *void connect_MQTT()*
Para la explicación de las funciones, se obviarán las líneas que comienzan con Serial, ya que son para pruebas locales.
Dentro de la función *void connect_MQTT()*, lo primero que se hace es conectarse al WiFi con la función:

- *WiFi.begin(ssid, wifi_password);*

Y se pasa, por parámetro, las variables *ssid* y *wifi_password*, que incluyen los valores definidos de la red WiFi local a la que se quiere conectar.

El siguiente bloque de código hace que el sistema aguarde hasta que la conexión sea exitosa; si no, no continúa con su ejecución e informa del error.
Ya hecha la conexión a la red WiFi local, se intenta hacer la conexión al *Broker* MQTT con la función:

- *Client.connect(ClientID, mqtt_username, mqtt_password)*

Según esta conexión sea exitosa (o no), continuamos la ejecución.

Función *void setup():*
La función *void setup()*, en esta primera instancia, solo inicializa el sensor DHT11 con la línea:

- *dht.begin()*

En una futura optimización del código, acá se realizará la conexión al *Broker* por una única vez.

Función *void loop():*
En esta función se encuentra toda la lógica programada del *Publisher*, es decir, aquí se lee la información que captan los sensores, se la procesa y se la publica en el *Broker MQTT*.
Lo primero y principal que realiza esta función es llamar a la otra función que definimos "*connect_MQTT()*", que se va a conectar al *Broker* y va a permitir publicar información en el *topic* que corresponda.
Luego se inicializan tres variables:

- *float humedadAmb = dht.readHumidity();*
- *float tempreaturaAmb = dht.readTemperature();*
- *int humedadSuel = analogRead(humedadSueloPin);*

A cada una de estas variables se le asigna la lectura de su correspondiente sensor, para su uso posterior. Para la lectura del sensor DHT11 se utiliza la librería que

importamos y para el sensor HL-69, como está conectado a un pin analógico, se lo puede leer con la función analogRead(pin conectado).

Luego de obtenida la lectura de los sensores, se procede a publicarlos en el *Broker.* Para ello es necesario convertir cada uno de los resultados a tipo "*string*[1]", ya que MQTT solo comparte datos de este tipo. Pero esta conversión la realizamos directamente al momento de publicar el dato al *topic* en el *Broker*; de esta manera, la publicación de la información se hace del siguiente modo:

- *client.publish(temperaturaAmbiente_topic, String(tempreaturaAmb).c_str())*

En esta línea se realiza la publicación en el *topic* de "*temperaturaAmbiente*", pasando por parámetro y convirtiendo el valor contenido en la variable que guardó el dato, obtenido del sensor DHT11 anteriormente.

Para publicar la información de los tres sensores se hace el mismo proceso: solo se cambia el *topic* al cual se publica y la variable con el valor del sensor que se quiere publicar. El proceso de cómo publica la información está indicado en las siguientes líneas:

```
if (client.publish(temperaturaAmbiente_topic, String(tempreaturaAmb).c_str())) {
    Serial.println("¡Temperatura enviada!");
  }
else {
  client.connect(clientID, mqtt_username, mqtt_password);
  delay(10)
  client.publish(temperaturaAmbiente_topic, String(tempreaturaAmb).c_str());
}
```

Figura 11 - Porción de código de un Publisher.
Fuente: Elaboración propia.

Como se indica en la imagen, primero se ejecuta *client.publish(topic, valor a publicar)* dentro de un condicional, por ende, si la publicación de la información es

[1] Un objeto *string* es una secuencia o cadena de caracteres.

correcta, se puede continuar normalmente para publicar en el *Broker* la información del siguiente sensor; si, en cambio, se obtiene error, se vuelve a conectar al *Broker* y se intenta publicar la información nuevamente.
Como se indicó antes, esta porción de código se ejecuta para cada uno de los sensores que haya conectados y con los que se quiera publicar datos en un *topic*.
Luego de publicada la información se realiza un *client.disconnect()*, que es para desconectarse del *Broker* generando una pausa de cinco minutos hasta poder volver a ejecutar todo el código de la función *loop()* nuevamente.

Broker (Raspberry PI 3)

Para crear el *Broker*, vamos a utilizar la *Raspberry PI 3*. A tal efecto, es necesario la instalación de un software adicional en la Raspberry, que, en este caso, es “Mosquitto**”**. Para instalar Mosquitto, introducimos en una terminal de la Raspberry:

- *sudo apt-get install mosquitto (instala el Broker/Servidor MQTT)*
- *sudo apt-get install mosquitto-clients -y (para poder realizar debugging[2])*

Después de la instalación del *MQTT Broker Mosquitto***,** se realizaron las siguientes modificaciones de configuración de la aplicación. Para ello, se abrió el archivo ubicado en “*/etc/mosquitto/mosquitto.conf*” (ver “*Figura 12 – Archivo de configuración de Mosquitto*”) y se debe corroborar que se cumplan los siguientes cuatro puntos:

1. Nos garantizamos que el *Broker* solo incluya la configuración por defecto. Esto se realiza comentando la línea “*include_dir /etc/mosquitto/confi.d*”, utilizando el símbolo “#”.

[2] *Debugging* es el proceso de análisis de códigos de programación, paso a paso, en su funcionamiento.

2. No debemos permitir que usuarios anónimos se conecten al MQTT *Broker*, por eso agregamos la línea "*allow_anonymous false*".
3. Queremos guardar las contraseñas en un archivo separado, por eso generamos un archivo nuevo con la siguiente línea: *"password_file /etc/mosquitto/pwfile".*
4. Queremos que la comunicación entre *Broker* y Clientes esté cifrada. Generamos los certificados SSL en la ubicación "*/etc/ssl/mqtt"* con la línea "*sudo openssl req -new -x509 -days 365 -nodes -out huerta.local.pem -keyout huerta.local.key*". Y los indicamos en el archivo de configuración.
5. Necesitamos que el *Broker* sea accesible por el puerto 8883. Agregamos la línea *"listener 8883".*

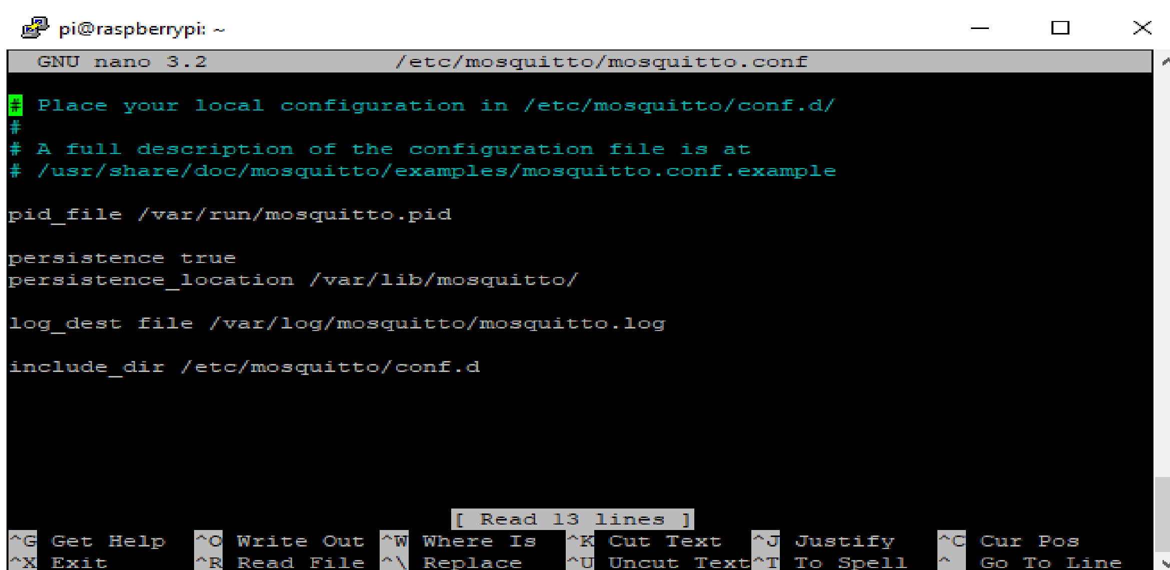

Figura 12 - Archivo de configuración de Mosquitto.
Fuente: Elaboración propia.

Luego de aplicar estos cuatro puntos sobre el archivo de configuración, el archivo queda de la manera descripta en la imagen de la "*Figura 13 – Archivo de configuración Mosquitto con modificaciones realizadas*":

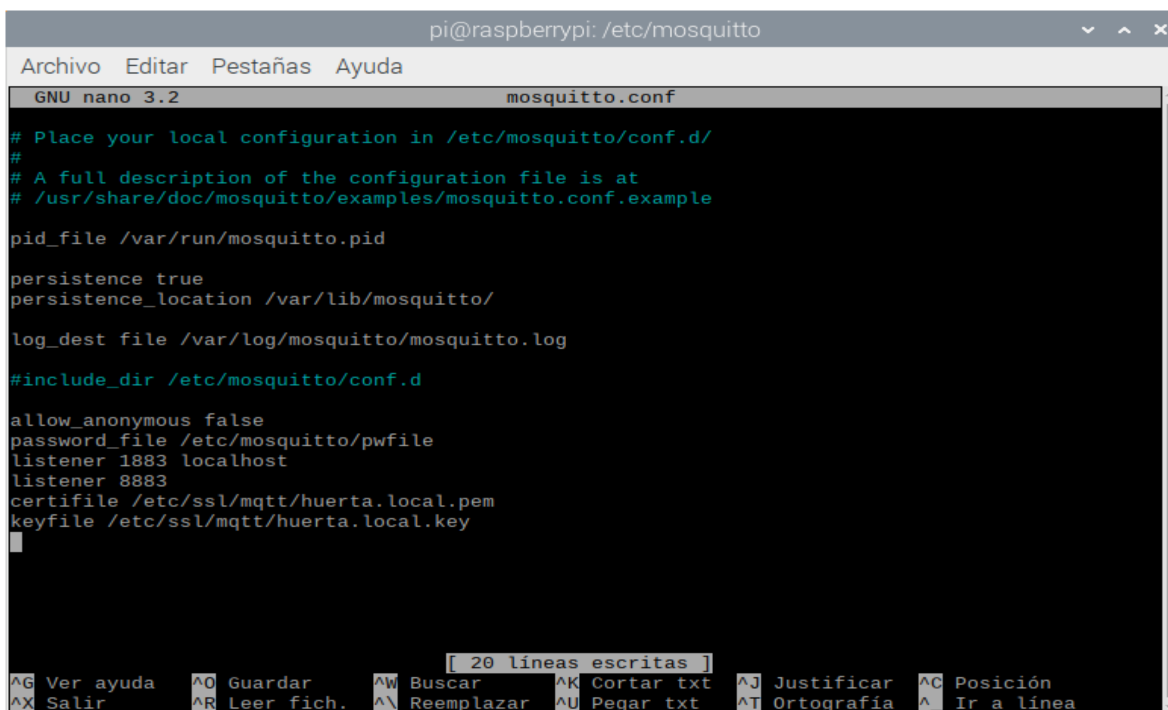

Figura 13 - Archivo de configuración Mosquitto con modificaciones realizadas.
Fuente: Elaboración propia.

Y, luego de la modificación del archivo de configuración, se creó un nuevo usuario y contraseña a partir del siguiente comando:

- *sudo mosquitto_passwd -c /etc/mosquitto/pwfile username*

Donde "*Username*" debe reemplazarse por el nombre definido para el usuario que poseerá el *Broker*. Al ingresar este comando en la terminal, se le solicita al usuario que ingrese una contraseña y que la vuelva a reingresar.

Ahora, ejecutando "*sudo systemctl start mosquitto*", ya queda funcionando el *Broker*, con la configuración empleada, para recibir datos enviados por *Publishers*.

Suscriber (Raspberry PI 3)

El *Suscriber*, que en este caso es la misma *Raspberry* que contiene el *Broker*, fue creado en el lenguaje de programación Python versión 3.7.3, con la instalación de algunas librerías necesarias para poder darle uso al código creado como *Suscriber* del protocolo MQTT.

Este *Suscriber* está compuesto por tres archivos (2 módulos y un archivo principal) con extensión "*.py*", y cada uno de ellos está orientado a solventar una parte necesaria de un *Suscriber* como tal. Asimismo, esta estructura implementada para el *Suscriber* es necesaria para dar una dinamicidad a la integración de nuevos *Publishers*, sin tener que realizar modificaciones de código del archivo principal, ya que uno de los tres archivos corresponde a un archivo de configuración donde se indican cada uno de los *Publishers* que contiene el sistema. En términos más generales, en este archivo se define cómo está compuesto un invernadero.

usuario_config.py

En este archivo se define la estructura que tendrá el invernadero y, en caso de que un invernadero esté compuesto por un nodo con sensores (*Publisher*) y se desee agregar un nuevo nodo, en este archivo de configuración se indicará el nuevo nodo que se insertará en la base de datos de manera que el programa principal del *Suscriber* lo identificará sin ningún inconveniente.

El archivo de configuración tiene una estructura de diccionario, donde la información está compuesta por pares de clave valor.

La raíz general del módulo (ya que este archivo se importa en el programa principal como un módulo) está compuesta por la variable "*config*" y, dentro de esta variable, está contenida toda la información necesaria del invernadero.

Los valores que posee son:

- email: email asociado al invernadero.
- nombreUser: nombre de usuario asociado al invernadero.

- invernadero: nombre asociado al invernadero.
- nodos: arreglo que contiene, en cada posición, la información de los nodos que posee el invernadero. La información que posee de cada nodo es el nombre, tipo, función para que tipo de plantación está configurado, si se han ingresado nuevos parámetros, y un arreglo, con sus *topics* asociados.

La estructura más clara se ve en la "*Figura 14 – Estructura del archivo "usuario_config.py"*", donde se muestra cómo es la estructura del archivo *usuario_config.py.*

```
##Archivo de configuracion del invernadero, aca va toda la informacion necesaria para el suscriber del invernadero

config = {
    "email":"leednavarro@gmail.com",
    "nombreUser": "LeNaUnaj",
    "invernadero": "invernadero1",
    "nodos": [{
            "nombre":"nodo1",
            "topics":["temperatura","humedadAmbiente", "humedadSuelo"],
            "funcion": "tomate",
            "tipo":"Wemos D1",
            "nuevosParametros": False
        },
        {
            "nombre":"nodo2",
            "topics":["temperatura","humedadAmbiente", "humedadSuelo"],
            "funcion": "tomate",
            "tipo": "Wemos D1",
            "nuevosParametros": False
        }],

}
```

Figura 14 - Estructura del archivo "usuario_config.py".

Fuente: Elaboración propia.

database_connection.py

Este es el archivo que se encarga de la conexión con la base de datos en la nube y de brindarle al archivo principal una serie de funciones para facilitar el procesamiento de datos, que debe de hacer éste para comportarse como *Suscriber* según el protocolo MQTT.

Lo que contiene este archivo se explicará más detalladamente en la sección "Integración – Conexión del sistema de microcontroladores y base de datos".

get_MQTT_data.py

El archivo "*get_MQTT_data.py*" es el programa principal del *Suscriber* y en él está contenida toda la lógica del *Suscriber*. Acá es donde se van a utilizar las librerías y los módulos necesarios para poder conectar el *Suscriber* al *Broker* y poder suscribirse a los *topics* que corresponda.

Sin más motivos de introducción al archivo, procedemos a la explicación de la primera versión de su código, el cual está disponible en la "*Figura 15 – Código de implementación del Suscriber Versión 1, primera parte*" y en la "*Figura 16 – Código de implementación del Suscriber Versión 1, segunda parte*".

```
C: > Users > lnavarro > Desktop > pps > get_MQTT_data.py
1   import paho.mqtt.client as mqtt
2   import database_connection as db
3   import usuario_config as cfg
4   import datetime
5
6   MQTT_ADDRESS = '192.168.1.63'
7   MQTT_USER = 'huertabroker'
8   MQTT_PASSWORD = 'huerta1'
9   #Estos topics deberian ser dinamicos
10  MQTT_TOPIC_TEMPERATURA = 'invernadero1/nodo1/temperatura'
11  MQTT_TOPIC_HUMEDAD_AMBIENTE = 'invernadero1/nodo1/humedadAmbiente'
12  MQTT_TOPIC_HUMEDAD_SUELO = 'invernadero1/nodo1/humedadSuelo'
13  MQTT_TOPIC_TEMPERATURA2 = 'invernadero1/nodo2/temperatura'
14  MQTT_TOPIC_HUMEDAD_AMBIENTE2 = 'invernadero1/nodo2/humedadAmbiente'
15  MQTT_TOPIC_HUMEDAD_SUELO2 = 'invernadero1/nodo2/humedadSuelo'
16
17
18  def on_connect(client, userdata, flags, rc):
19      """callback para cuando el cliente recive ack de conexion del servidor"""
20      print('Conectado con codigo de resultado: ' + str(rc)) ##rc me da el codigo correspondiente de conexion al broker (sirve para analisis)
21
22
23      #ACA DEBO TOMAR LOS TOPICS DEL ARCHIVO DE CONFIGURACION (TRABAJO A FUTURO)
24      #TOPIC_EJEMPLO = cfg.config["invernadero"]+"/"+(cfg.config["nodos"][0])["nombre"]+"/"+(cfg.config["nodos"][0])["topics"][0]
25      #print(TOPIC_EJEMPLO)
26
27      #Suscripcion a los topics que me interesan
28      client.subscribe(MQTT_TOPIC_TEMPERATURA)
29      client.subscribe(MQTT_TOPIC_HUMEDAD_AMBIENTE)
30      client.subscribe(MQTT_TOPIC_HUMEDAD_SUELO)
31
32      client.subscribe(MQTT_TOPIC_TEMPERATURA2)
33      client.subscribe(MQTT_TOPIC_HUMEDAD_AMBIENTE2)
34      client.subscribe(MQTT_TOPIC_HUMEDAD_SUELO2)
35
36      #Inserta  actualiza la configuracion de los nodos en la base de datos
37      db.insert_configuracion()
38
```

Figura 15 - Código de implementación del Suscriber Versión 1, primera parte.

Fuente: Elaboración propia.

```
38
39  #on_message se llama cada vez que hay una publicacion nueva en el topic suscripto
40  def on_message(client, userdata, msg):
41      """Callback para cuando es recibido una publicacion de un publicador del servidor"""
42
43      fecha = datetime.datetime.now()
44
45      ##Objeto base para donde se va a cargar la informacion que luego se debe guardar en la base de datos
46      informacionUpdate = {
47          "nombreNodo":"",
48          "topicUpdate":"",
49          "valorUpdate":"",
50          "email": cfg.config["email"],
51          "fechaAlta": fecha,
52          "tipo": "Wemos D1",
53          "descripcion": "nodo a partir de Wemos D1"
54      }
55
56      topicSplit = msg.topic.split("/")                 #divido el topic en (invernadero/nodo/topic)
57      informacionUpdate["nombreNodo"] = topicSplit[1]    #seteo en la informacion a realizar el update, el nodo al que pertenece el mensaje que lleg
58      informacionUpdate["topicUpdate"] = topicSplit[2]
59      informacionUpdate["valorUpdate"] = (msg.payload).decode('UTF-8')
60      esAlerta = db.comprobarAlerta(informacionUpdate)
61      if(esAlerta == True):
62          #primero envio mail y luego inserto
63          db.enviarMail(informacionUpdate)
64
65      db.insert_information_nodo(informacionUpdate)
66
67
68  def main():
69      mqtt_client = mqtt.Client()
70      mqtt_client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
71      mqtt_client.on_connect = on_connect
72      mqtt_client.on_message = on_message
73
74      mqtt_client.connect(MQTT_ADDRESS,1883)
75      mqtt_client.loop_forever()
76
77
78  if __name__ == '__main__':
79      print('Suscriber MQTT - Invernaderos UNAJ')
80      main()
```

Figura 16 - Código de implementación del Suscriber Versión 1, segunda parte.
Fuente: Elaboración propia.

El código del "*get_MQTT_data.py*" comienza con la importación de tres módulos. El primero es librería "*paho*", que permite utilizar funcionalidades de MQTT de manera que el programa en *Python* pueda funcionar como *Suscriber*, como lo indica el protocolo MQTT. Esta librería se importa con la línea:

- *import paho.mqtt.client as mqtt.*

Los siguientes dos módulos importados son los otros dos archivos *Python*, explicados anteriormente: uno es el que se encarga de la conexión con la base de datos y da funcionalidades para su uso, y el otro es el archivo de configuración del invernadero.
En esta primera versión del programa *Suscriber*, se definen las variables necesarias para la conexión al *Broker* y la suscripción a los *topics*. En una versión futura del programa, estas variables se deberían de consumir también del archivo de configuración. Las variables que contienen la información para conectarse al *Broker*

serían más seguras, y las que tienen los *topics*, más dinámicas y permitirían agregar nuevos *topics* diferentes a los ya establecidos.

Exceptuando la definición de variables y la importación de los módulos/librerías, el sistema se compone de tres funciones: "*on_connect()", "on_message"* y *"main()".* Aparte de esas tres funciones, hay una iniciación de la función "*main()"* para que se inicialice el programa.

main()

Como ya se ha señalado, la función principal del programa es la función "*main()",* que es la que inicia el programa. Aquí es donde se inicializa la aplicación como *Suscriber*, conectándola al *Broker*, con la información definida en las variables descriptas al inicio del archivo y anteriormente mencionadas.

Para ello, crea un cliente MQTT, a partir de la librería "*paho"*, que posee una función que le permite hacer esto. En adición a lo último, al crear el cliente, también se crean dos métodos, que estarán relacionados a las otras dos funciones que hemos creado: "*on_connect"* y "*on_message"*. Estos dos métodos y las funciones que creamos también se llaman "*on_connect()"* y "*on_message()".* La diferencia es que, para seleccionar los métodos que posee el cliente, este debe de indicarlos a través de la notación punto (desde el cliente creado). Por ejemplo, "*mqtt_client.on_connect".* Es decir, al cliente creado, que es quien se conecta al *Broker* con ayuda de la librería "*paho*", se le indica que sus dos métodos ejecutarán las dos funciones creadas por nosotros en el código, de modo que cuando se realice la conexión al *Broker* se procese el código creado en estas dos funciones, las cuales tienen la lógica de negocio del *Suscriber*.

on_connect()

La función "*on_connect()"* es la que retorna un *callback*[3] del *Broker* y en donde se indican los *topics* a los que estará suscrito el *Suscriber*.

[3] *Callback* es la llamada de retorno que produce una función en programación.

Esta función recibe cuatro parámetros, que son de uso para pruebas locales. El más importante es el parámetro "*rc*", porque brinda un código con relación a cómo fue la conexión con el *Broker*. Los posibles códigos de respuesta que puede dar son:

- 0: *Connection successful*
- 1: *Connection refused – incorrect protocol version*
- 2: *Connection refused – invalid client identifier*
- 3: *Connection refused – server unavailable*
- 4: *Connection refused – bad username or password*
- 5: *Connection refused – not authoriced*

Como se puede apreciar, en caso de uso, solo nos interesaría el código 0 que es cuando se conecta correctamente al *Broker*; pero, en nuestro caso, la conexión al *Broker* se debe establecer sin falla, por ende, no realizamos ninguna comprobación a una conexión que damos por hecha como siempre exitosa, ya que el sistema para que se inicialice se tiene que conectar al *Broker*.
Lo único importante que llevamos a cabo en esta función es la suscripción a los *topics*.

on_message()
El método *on_message()* del cliente dispara un evento, cada vez que un *Publisher* publica información al *topic* al que el *Suscriber* está suscripto. De esta forma, cada vez que un *Publisher* publica nueva información, se va a ejecutar nuestra función que tiene tres parámetros, de los cuales uno es el que vamos a utilizar. En este parámetro llega la información del *topic* que tiene nuevos mensajes, esto quiere decir que de este parámetro podremos obtener tanto el *topic* que disparó el evento, como la información publicada en este *topic*.
Ya dentro de la función *"on_message()",* y con la información obtenida del parámetro "*msg*", creamos un diccionario (objeto compuesto por claves y valores), y le asignamos información necesaria para realizar la actualización en la base de datos.

Los datos que ponemos en el diccionario son:

- nombreNodo: nombre del nodo a actualizar.
- topicUpdate: *topic* del nodo a actualizar.
- valorUpdate: valor con el que se va a actualizar el *topic* del nodo.
- email: email correspondiente al usuario del nodo.

Con esta información almacenada, se efectúa el uso del módulo de la base de datos con la función *insert_information_nodo(informacionUpdate);* donde "informacionUpdate" es el diccionario creado con los datos, explicados previamente. Independientemente de la inserción de información en la base de datos, se realiza una comprobación con el método *comprobarAlerta(InformacionUpdate)* del módulo de la base de datos para saber si los valores están dentro de los parámetros configurados. En caso de que los parámetros no estén dentro de los valores configurados, se utiliza otro método del módulo de la base de datos que se encarga de enviar un email al usuario del invernadero.

Desarrollo web

La implementación del sistema web se llevó a cabo con una arquitectura de API donde el *Front-End* está separado del *Back-End* y este último funciona como servicio que provee información al *Front-End*.
Para poder implementar un sistema de este tipo, se diseñó primero la Api *Back-End* ya que es la que contiene toda la lógica de funcionamiento del sistema web y es el *Front-End* el que se ha adaptado a las funcionalidades que le brinda el *Back-End*.

Desarrollo Back-End

Para poder iniciar el desarrollo del *Back-End*, hay que preparar el entorno para poder trabajar: para ello, se utilizó "*Visual Studio Code"* (instalación en apéndice 1) como entorno de desarrollo y NestJS, como *Framework.*

Visual Studio Code permite la utilización de una terminal (ver "*Figura 17 – Visual Studio Code, terminal y entorno donde está alojado el sistema Back-End"*). En esta terminal es en donde utilizaremos comandos para la instalación tanto de NestJS como de otras dependencias necesarias para el funcionamiento de nuestro sistema *Back-End*.

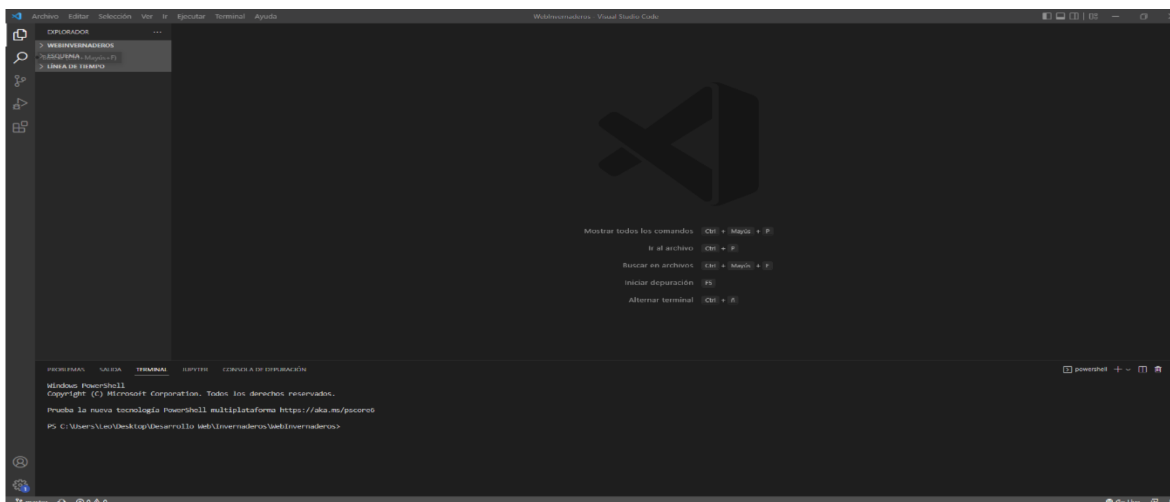

Figura 17 - Visual Studio Code, terminal y entorno donde está alojado el sistema Back-End.
Fuente: Elaboración propia.

Como se ha explicado en un principio, para la instalación de dependencias se iba a precisar la utilización del gestor de paquetes "NPM". Para poder utilizar este gestor, primero es necesario instalar NodeJS, ya que, como también hemos mencionado, NestJS funciona sobre NodeJS, y es necesario este último para su funcionamiento. Para poder instalarlo, solo es necesario ingresar a https://nodejs.org/es/download/, seleccionar el instalador de Windows y solo seguir los pasos indicados por el instalador.

Una vez instalado NodeJS, en la terminar de *Visual Studio Code* procedemos a realizar uso de NPM y comenzamos las instalaciones necesarias para poder iniciar el desarrollo de la aplicación *Back-End*, necesaria para el sistema web.

Instalación de NestJS
Para instalar NestJS utilizamos primero el comando:

- *npm i -g @nestjs/cli*

Este comando instalará el CLI[4] de NestJS, el cual posee funcionalidades para facilitar la inicialización, el desarrollo, y la mantención de aplicaciones creadas con NestJS.
Ya instalado NestJS, se procede a crear un proyecto de NestJS; en este caso, el *Back-End* con:

- *nest new project-name*

donde "project-name" se reemplaza por el nombre de nuestro *Back-End* que, en nuestro caso, se llama "WebInvernaderos". Este comando crea una estructura de aplicación como la indicada en la "*Figura 18 – Estructura básica de una aplicación NestJS*".

[4] CLI es una interfaz de línea de comandos, que permite insertar instrucciones sencillas para facilitar la interacción del usuario.

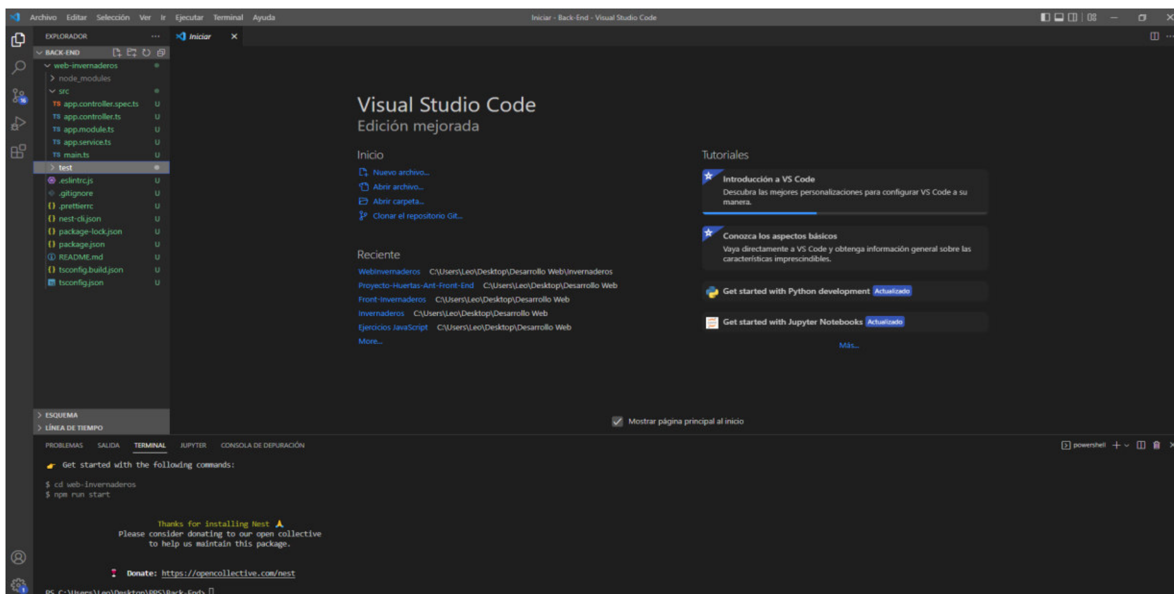

Figura 18 - Estructura básica de una aplicación NestJS.
Fuente: Elaboración propia.

Esta estructura básica, que genera el CLI de NestJS**,** ya contiene Git instalado y, para poder utilizarlo con GitHub, solo es necesario realizar la conexión entre este sistema (repositorio local) y el repositorio remoto (GitHub).

Para realizar esta conexión se utilizó una cuenta creada en GitHub, necesaria para poder almacenar repositorios, en donde se creó un repositorio con el mismo nombre del sistema *Back-End.*

Ingresando a ese repositorio en GitHub**,** se copia el enlace como está indicado en la siguiente imagen:

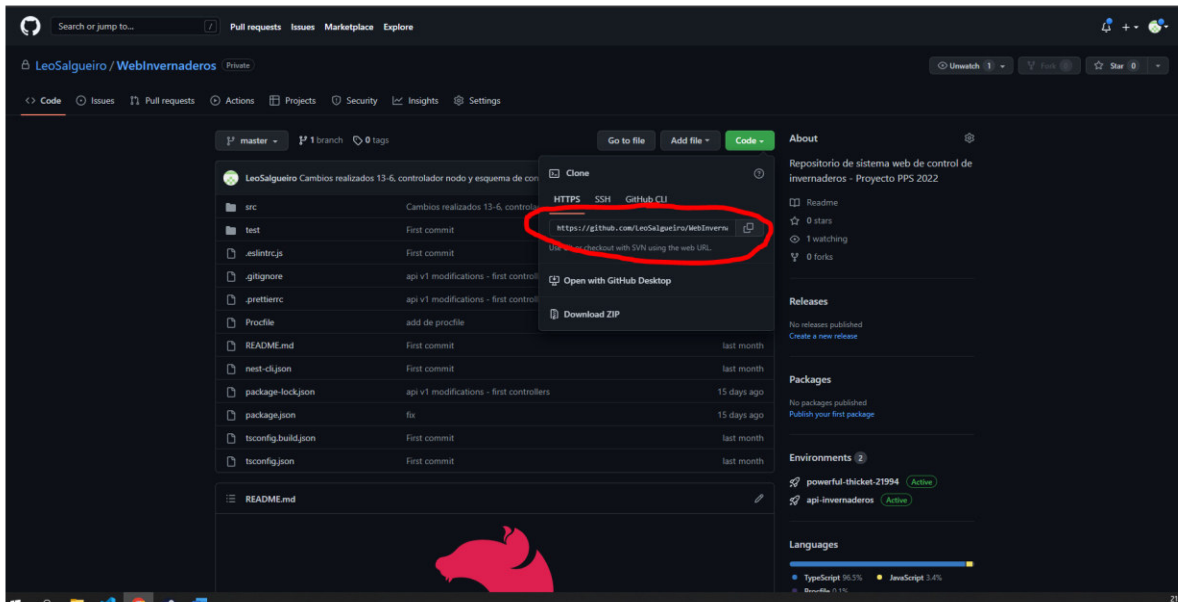

Figura 19 - Enlace de conexión a repositorio remoto.
Fuente: Elaboración propia.

Con ese enlace podemos establecer la conexión entre nuestra aplicación *Back-End* local y el control de versiones remoto. Para ello, ejecutamos el siguiente comando en la terminal de *Visual Studio Code* de nuestro sistema:

- *git remote add origin https://github.com/LeoSalgueiro/WebInvernaderos.git*

Ejecutado este comando, ya tenemos acceso a nuestro repositorio remoto desde nuestro repositorio local. Ya realizada esta conexión, ejecutamos el siguiente comando para subir todo nuestro código creado al repositorio remoto:

- *git push origin master.*

Esto copió todo nuestro código local y lo puso en el repositorio remoto. A partir de acá, cada cambio y código nuevo agregado en nuestro sistema local se fue subiendo con este último comando explicado, para ir teniendo un control de versionado de nuestro sistema. Además, estos cambios actualizan el proyecto que

se encuentra en producción, ya que el servidor que aloja el sistema capta los datos automáticamente de GitHub.

Explicación de estructura básica del sistema:

La estructura básica está compuesta por tres carpetas y una serie de archivos de configuración.

En primer lugar, la carpeta llamada “node_modules” es la que contiene todas las dependencias necesarias que utiliza la aplicación para poder funcionar. En esta carpeta, las dependencias se van agregando automáticamente a medida que vamos instalándolas con el gestor de paquetes; por esta razón, no es necesario entrar en más detalles.

En segundo lugar, la carpeta “src” contiene todos los archivos de la lógica de funcionamiento de nuestra aplicación, es el corazón de nuestro sistema y es donde se realizaron todas las implementaciones que se irán explicando más adelante. Esta carpeta contiene los archivos “*main.ts*”, “*app.service.ts*”, “*app.module.ts*”, “*app.controller.ts*” y “*app.controler.spec.ts*”:

- main.ts: es el archivo principal donde se inicia toda la aplicación; aquí se indican las configuraciones generales de la aplicación para ponerla en producción.
- App.controller.spec.ts: es el archivo de programación para realizar *testing* (no utilizado).
- App.controller.ts: es el archivo donde se programan los ingresos externos a la aplicación. Un controlador posee diferentes tipos de solicitudes permitidas para hacer.
- App.module.ts: es el archivo en donde se configura la importación de dependencias que pueden utilizar “*app.controller.ts*” y “*app.service.ts*”.

- App.service.ts: es el archivo en donde se programa la lógica del controlador y en donde está la funcionalidad básica que puede ofrecer el controlador.

Como se puede observar, el CLI creó cuatro archivos que comienzan con "app". Este conjunto de cuatro archivos creados son un controlador. A partir de ahora, cuando hagamos referencia a la palabra "controlador", nos referimos a un conjunto de cuatro archivos compuestos por un "*controller*", un "*module"*, un "*service"* y un "*controller.spec".*

La tercera carpeta es *test,* la cual contiene archivos de "*testing"*, que no van a ser utilizados para este proyecto.
Luego de estas tres carpetas, hay una serie de archivos que son de configuración, de los cuales solo importa explicar dos: ".gitignore" y "package.json":

- .gitignore: es el archivo en donde se indica qué archivos no deben ser tenidos en cuenta, cada vez que se suben nuevas modificaciones al control de versiones remoto.
- package.json: es el archivo de configuración que indica todo lo necesario para la puesta a producción del sistema e incluye *scripts* de ejecución de la aplicación, dependencias necesarias, el nombre de la aplicación, la versión actual, entre otros.

Ya explicada toda la estructura básica generada por el CLI de NestJS**,** procedemos a la explicación de cómo funciona NestJS.
NestJS es un *Framework* basado en "AngularJS[5]", donde las aplicaciones se programan en base a controladores, los cuales serán consultados por la aplicación *Front-End*. Un controlador tiene la habilidad de recibir solicitudes HTTP y, a través de un servicio, brindar lo solicitado por la solicitud.
Cada controlador está compuesto de tres archivos:

[5] AngularJS, es un *framework* de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.

- Controller.ts: es el archivo donde se programan las entradas de qué solicitudes va a permitir ese controlador; debe ser indicado en el "*module.ts"* de "*app.module.ts"* para que funcione.
- Service.ts: es el archivo donde se programa la lógica de qué va a hacer el controlador; por ejemplo, si el controlador devuelve información de los nodos, acá se va a programar la funcionalidad de consultar la información a la base de datos y de prepararla para el "*controller.ts"*, para que sea enviada a quien realizó la consulta. También debe ser indicada en el *"app.module.ts"* para que funcione.
- Module.ts: es el archivo donde se importan las dependencias que necesite el controlador, donde también se indican cuál es el "*controller.ts"* y el "*service.ts*" que le corresponde, y en donde se exporta al controlador como objeto para poder ser utilizado.

Comprendiendo esta estructura de un controlador, la aplicación *Back-End*, actualmente, cuenta con tres controladores: "Nodos", "Usuarios" y "app". Tanto el controlador "Usuarios" como "Nodos" cuentan con unos archivos extras que tienen la denotación "*schema*", que hace referencia al esquema de un documento de la base de datos. La representación de un documento de la base de datos en un controlador indica que ese controlador hace uso de ese documento de la base de datos y, por ende, se encarga de hacer escrituras y lecturas en este. Esto se verá más adelante.
De este modo toda la aplicación de *Back-End* quedó compuesta de la manera expuesta en la "*Figura 20 – Estructura de sistema Back-End completa v1.0*".

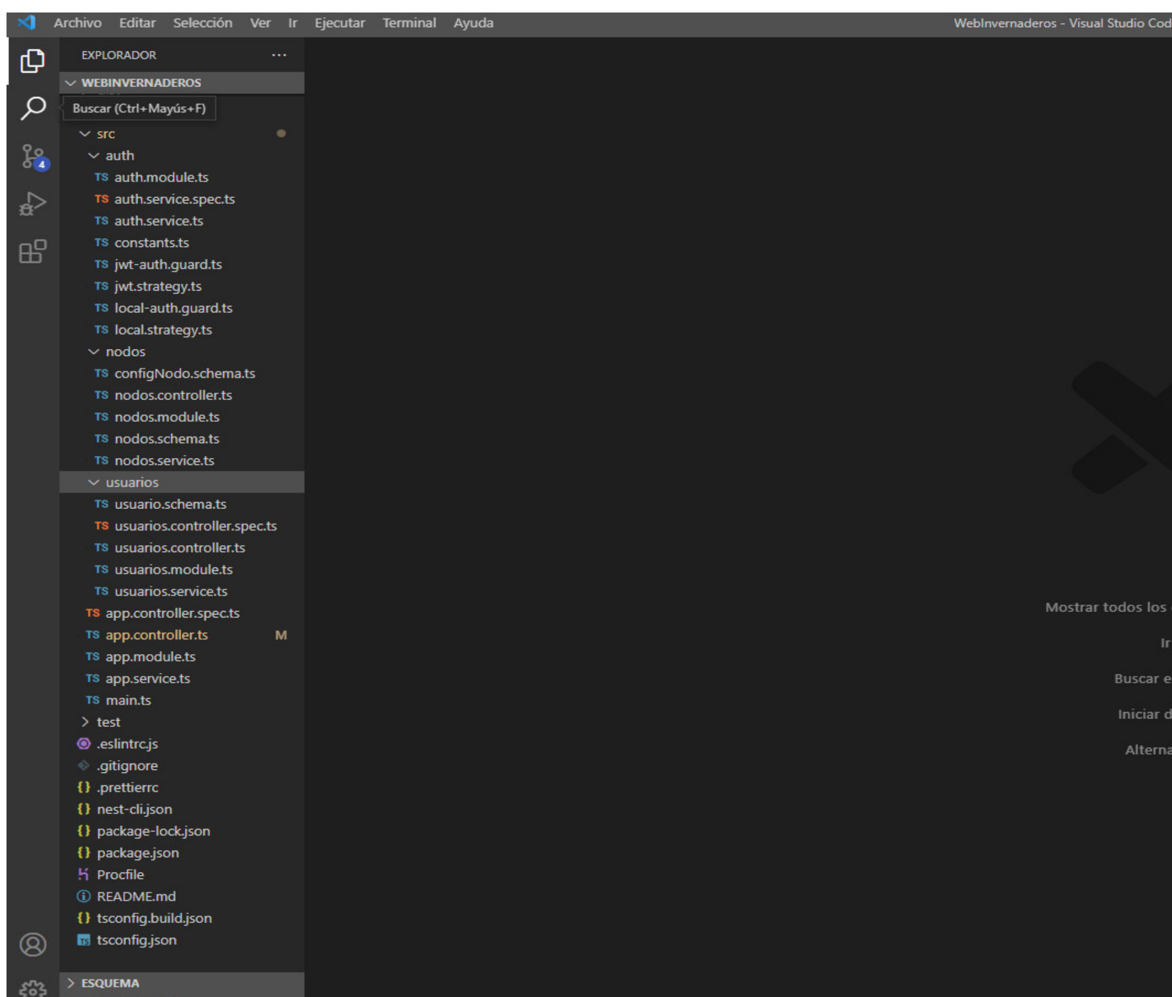

Figura 20 - Estructura de sistema Back-End completa v1.0.

Fuente: Elaboración propia.

Como se puede observar en la estructura final, están los tres controladores (Nodos, Usuarios y app). Excepto el controlador app, que es el principal y está en la raíz del sistema, los demás controladores se deben crear en una carpeta separada. Además, hay una tercera carpeta que es en donde está la lógica de los guardianes, la cual se explicará luego.

Para explicar el funcionamiento del sistema *Back-End* y comprender su flujo de ejecución, es recomendable comenzar por el primer controlador, el controlador app.

Controlador app

El controlador app es la base del sistema de *Back-End,* adonde se ingresa por primera vez si se quiere ingresar a la aplicación. Este controlador, en su archivo "*app.controller.ts"*, tiene expuesto un *endpoint*[6] de autenticación con usuario y contraseña, el cual, de ser la información correcta, devuelve un *token*[7] que es necesario para realizar las consultas a cualquier otro *endpoint* expuesto en la aplicación.

Para la creación de este *token*, se utiliza lo que se denomina "guardián", que es el encargado de crearlo y de asegurarse de que se cumpla la regla de que cada solicitud llegue con el *token* requerido por la aplicación (ver "*Figura 21 – Funcionamiento de un guardián*"). Además del "guardián" del token, hay un "guardián" que se encarga de la comprobación de que el usuario, que está solicitando el *token* para hacer consultas, sea un usuario válido. Para este caso, está el denominado "*LocalGuard*". Y, para usar estos guardianes, se utiliza lo que NestJS declara como "*Decorators*", que se indican al inicio de cada *endpoint* con las siglas *"@UseGuards("nombre de guardián a utilizar")*".

Los "*Decorators*" también se utilizan para definir qué tipo de solicitud HTTP recibe el *endpoint*, ya sea, un *Post*, un *Get*, etc. (entre otros tipos de uso que no serán utilizados en nuestra aplicación).

Figura 21 - Funcionamiento de un guardián.
Fuente: https://docs.nestjs.com/guards.
Fecha de consulta: 02/06/2022.

[6] Un *endpoint* es un punto de ingreso que se puede consultar a través solicitudes HTTP/HTTPS
[7] Un *token* es una clave hexadecimal única y cifrada.

La estructura de “*app.controller.ts*” está descripta de la manera que se detalla en la *“Figura 22 – Estructura de “app.controller.ts””*.

```
app.controller.ts 1, M ×
src > app.controller.ts > ...
1  import { Controller, Request, Post, UseGuards, Get } from '@nestjs/common';
2  import { LocalAuthGuard } from './auth/local-auth.guard';
3  import { AuthService } from './auth/auth.service';
4
5  @Controller()
6  export class AppController {
7    constructor(private authService: AuthService) {}
8
9    @UseGuards(LocalAuthGuard)
10   @Post('auth/login')
11   async login(@Request() req) {
12     return this.authService.login(req.body);
13   }
14 }
15
```

Figura 22 - Estructura de “app.controller.ts”.
Fuente: Elaboración propia.

Como se puede observar en la figura anterior, “*app.controller.ts*” está compuesto por dos secciones: la primera es en la que se importan todas las dependencias necesarias para su funcionamiento y la segunda, donde está definida la clase[8], que define a “*app.controller.ts”.* La gran diferencia que posee con una clase pura de POO (Programación Orientada a Objetos) es que, en vez de estar compuesta por atributos y métodos como esta, está compuesta por atributos y *endpoints.* En este caso, está compuesta por un atributo privado (“*authService*”) y por un *endpoint* de tipo *Post* a la dirección “*auth/login*”; esto quiere decir que para solicitar este *endpoint* desde otra aplicación se debe de ingresar a la dirección base más “*auth/login*”. Por ejemplo, “https://www.ejemplo.com/auth/login”: este *endpoint*, al momento de ser requerido, hace uso del atributo privado de la clase, el cual tiene asignado un “*auth.service*”, que está ubicado en la carpeta “*auth*” del sistema *Back-End*. Esto se

[8] Una clase es un objeto representado en código en programación orientada a objetos (POO).

logra importando, al principio del archivo "*app.controller.ts*", el archivo "*auth.service*" con la siguiente línea de código:

- *import { AuthService } from './auth/auth.service';*

Y, luego en el constructor de la clase,

- *constructor(private authService: AuthService) {}*

se hace asignación de este archivo al atributo privado mencionado previamente. Esto permite que este archivo pueda utilizar toda la funcionalidad desarrollada del archivo importado. Con respecto al "guardián", es necesario importarlo al inicio del archivo y luego solo resta indicarlo con un "*Decorator*" en el *endpoint*.
La única funcionalidad que tiene este archivo es tomar las solicitudes de tipo *Post*, que lleguen a la dirección "*auth/login*", haciendo uso del "*authService*" y su método "*login*", donde recibe por parámetro el cuerpo de la solicitud *Post,* el cual tendrá de contenido "usuario" y "contraseña".
El método "login" está definido en el archivo "*auth.service.ts*", el cual se puede observar en la "*Figura 23 – Composición del archivo "auth.service.ts"*". Este solo utiliza el servicio de JWT[9] y crea un *token* que se lo devuelve a "*app.controller.ts*", para que, a su vez, se lo devuelva al usuario que realizó la solicitud.

[9] JWT es un estándar, basado en documentos de tipo JSON, para la creación de *tokens* de acceso.

```
TS auth.service.ts 1, M ×
src > auth > TS auth.service.ts > ...
1  import { Injectable } from '@nestjs/common';
2  import { JwtService } from '@nestjs/jwt';
3  import { UsuariosService } from 'src/usuarios/usuarios.service';
4
5  @Injectable()
6  export class AuthService {
7    constructor(
8      private jwtService: JwtService,
9      private usuariosSertvice: UsuariosService,
10   ) {}
11
12   async validateUser(email: string, pass: string): Promise<any> {
13     const user = await this.usuariosSertvice.buscarUsuario(email);
14     if (user && user.password === pass) {
15       const { password, ...result } = user;
16       return result;
17     }
18     return null;
19   }
20   async login(body: any) {
21     const payload = { username: body.email };
22     return {
23       access_token: this.jwtService.sign(payload),
24     };
25   }
26 }
27
```

Figura 23 - Composición del archivo “auth.service.ts”.

Fuente: Elaboración propia.

Como ya se mencionó previamente, se hace uso de “guardianes”; por esta razón, para que se pueda ejecutar el uso del método “login”, primero debe pasar los requerimientos del guardián mencionado con el “*Decorator*”. Este guardián es “*LocalAuthGuard*”, que hace uso del archivo “*local.strategy.ts*” (ver “*Figura 24 – Composición de “local.strategy.ts”*”) y este archivo, a su vez, hace uso del método “*validateUser*”, importando el archivo *“auth.service.ts”*, el cual se muestra en la “*Figura 23 – Composición del archivo “auth.service.ts*”.

Y, por último, este método (el de “*auth.service.ts*”) hace uso de “*usuarios.service.ts*”, que es el servicio que pertenece al controlador “Usuarios” y se encarga de hacer la consulta a la base de datos para chequear que el usuario y contraseña sean válidos; de ser así, el sistema hace todo el retroceso entre los métodos y el guardián le da el “OK” para que pueda realizarse el uso del método “login”, que crea un *token* y lo devuelve al cliente que realizó la solicitud.

Todo este proceso comentado se puede observar con más detalle en la “*Figura 25 – Proceso de login a través con un guardián intermedio*”:

```
TS local.strategy.ts ×
src > auth > TS local.strategy.ts > ...
1   import { Strategy } from 'passport-local';
2   import { PassportStrategy } from '@nestjs/passport';
3   import { Injectable, UnauthorizedException } from '@nestjs/common';
4   import { AuthService } from './auth.service';
5   @Injectable()
6   export class LocalStrategy extends PassportStrategy(Strategy) {
7     constructor(private authService: AuthService) {
8       super();
9     }
10
11    async validate(email: string, password: string): Promise<any> {
12      const user = await this.authService.validateUser(email, password);
13      if (!user) {
14        throw new UnauthorizedException();
15      }
16      return user;
17    }
18  }
19
```

Figura 24 - Composición de “local.strategy.ts”.

Fuente: Elaboración propia.

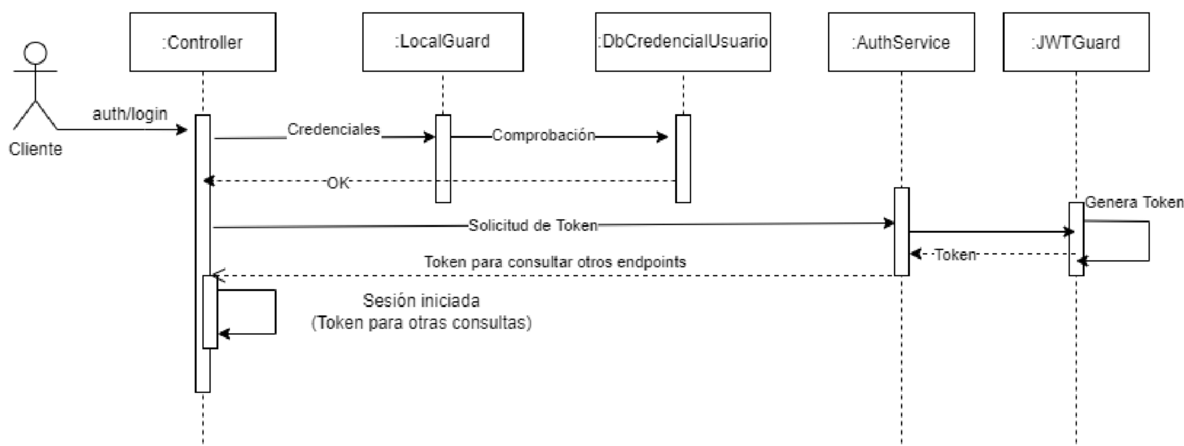

Figura 25 - Proceso de login a través de un guardián intermedio.

Fuente: Elaboración propia.

Luego de que la validación del usuario que realizó la consulta sea exitosa, se va a poder consultar cada uno de los otros *endpoints* con el *token* obtenido.

Controlador "Usuarios"

El controlador llamado "Usuarios" está compuesto por cuatro archivos:

- Usuarios.controller.ts.
- Usuarios.module.ts.
- Usuarios.service.ts.
- Usuarios.schema.ts

El curso de las consultas es el mismo que en cualquier otro controlador. Llega una solicitud HTTP al archivo "*controller.ts*"; este utiliza la funcionalidad que está desarrollada en "*service.ts*" y, de este modo, es posible esa conexión a través del archivo "*module.ts*".
En este caso, hay un archivo extra, el "*schema.ts*" (ver "*Figura 26 – "schema" del controlador Usuarios ("usuarios.schema.ts")*"), que es una representación del documento que se almacena en la base de datos y posee propiedades que son proporcionadas por NestJS, que permiten la manipulación de este sin tener que realizar consultas complejas a la base de datos. Este *schema* es utilizado por el servicio del controlador para realizar las acciones que sean requeridas: consultar un usuario, agregar un usuario, buscar un usuario, etcétera.

```
src > usuarios > TS usuario.schema.ts > ...
1   import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
2   import { Document } from 'mongoose';
3
4   export type UsuarioDocument = Usuario & Document;
5
6   @Schema()
7   export class Usuario {
8     @Prop([String])
9     foto: string[];
10
11    @Prop()
12    nombre: string;
13
14    @Prop()
15    apellido: string;
16
17    @Prop()
18    dni: number;
19
20    @Prop()
21    direccion: string;
22
23    @Prop()
24    fechaNac: string;
25
26    @Prop()
27    genero: string;
28
29    @Prop()
30    email: string;
31
32    @Prop()
33    password: string;
34
35    @Prop()
36    nombreUser: string;
37
38    @Prop()
39    hash: string;
40  }
41
42  export const UsuarioSchema = SchemaFactory.createForClass(Usuario);
43
```

Figura 26 - “schema” del controlador Usuarios (“usuarios.schema.ts”).
Fuente: Elaboración propia.

Usuarios.controller.ts

“Usuarios.controller.ts” está compuesto de la manera que se muestra en la “*Figura 27 – composición de “usuarios.controller.ts”*”.
Dividido en dos secciones (“importación de archivos” y la clase “*UsuariosController*”), el archivo realiza la importación del servicio correspondiente del controlador y algunas utilidades que brinda NestJS para facilitar la programación. En la clase, se define el atributo privado “*usuariosService*”, al que se le asigna toda la funcionalidad que posee el servicio del controlador. Luego, la clase se compone de cuatro *endpoints,* tres *Posts* y un *Get.* Para indicar el tipo de solicitud HTTP, se utiliza el “*Decorator*” con el nombre del tipo de solicitud correspondiente.

(NestJS provee de otros tipos de solicitudes HTTP, pero no serán utilizadas en nuestro sistema *Back-End*).

Cada "*Decorator*" puede recibir, por parámetro, una clave que lo identificará. Si este parámetro no está especificado, el *endpoint* será accesible a través de URL base más el nombre del controlador; por ejemplo, https://www.ejemplo.com/usuarios. En caso de especificar una clave, quedaría como https://www.ejemplo.com/usuarios/clave. Si un *endpoint* de un tipo de solicitud especifico no recibe la clave por parámetro, no puede haber otro *endpoint* del mismo tipo de solicitud HTTP sin la clave por parámetro.

Esta última regla mencionada se puede ver en práctica en los primeros dos *endpoints*, ya que ambos no poseen nombre de clave por parámetro, pero al ser de diferente tipo de solicitud HTTP no hay inconveniente entre ambos.

La funcionalidad de cada uno de los *endpoints* es la siguiente:

- @Get (getUsuarios()): solicita al servicio del controlador que le envíe todos los usuarios de la base de datos.
- @Post (getUsuario(@Request() req)): solicita al servicio del controlador que le envíe el usuario especificado en "req"[10].
- @Post('update'): solicita al servicio del controlador que actualice la información del usuario recibido en "req".
- @Post('insert'): solicita al servicio del controlador que ingrese el nuevo usuario recibido en "req", si es que este no existe.

[10] Req es la abreviatura de *Request* y, a través de ella, podemos acceder al cuerpo de la solicitud HTTP y obtener los parámetros requeridos

```
usuarios.controller.ts ×
src > usuarios > usuarios.controller.ts > ...
1  import { Controller, Request, Get, Post } from '@nestjs/common';
2  import { UsuariosService } from './usuarios.service';
3
4  @Controller('usuarios')
5  export class UsuariosController {
6    constructor(private usuariosService: UsuariosService) {}
7
8    @Get()
9    async getUsuarios() {
10     const result = await this.usuariosService.findAll();
11     return result;
12   }
13
14   @Post()
15   async getUsuario(@Request() req) {
16     const result = await this.usuariosService.buscarUsuario(req.body);
17     return result;
18   }
19
20   @Post('update')
21   async actualizarInformacion(@Request() req) {
22     const result = await this.usuariosService.actualizarInformacionUsuario(
23       req.body,
24     );
25     return result;
26   }
27
28   @Post('insert')
29   async agregarUsuario(@Request() req) {
30     const result = await this.usuariosService.agregarUsuario(req.body);
31     return result;
32   }
33 }
```

Figura 27 - Composición de "usuarios.controller.ts".

Fuente: Elaboración propia.

Usuarios.module.ts

"*Usuarios.module.ts*" está compuesto de la manera que se muestra en la "*Figura 28 – Composición de "usuarios.module.ts"*". Como se puede observar en la imagen, se importan todos los archivos que componen al controlador (*schema*, *controller* y *service*); además, también se importa a *MongooseModule***,** que es una biblioteca de JavaScript que permite definir esquemas con datos fuertemente tipados y modelar una base de datos a través de estos esquemas. En nuestro caso, *MongooseModule* relaciona el esquema que tenemos en nuestro *Back-End* con el documento que está almacenado en la base de datos. Esta relación la realiza en el apartado de *imports*, cuando indica qué esquema se debe relacionar con la base de datos:

- *MongooseModule.forFeature([{name: Usuario.name, schema: UsuarioSchema }])*

Luego está el apartado *providers*, palabra en inglés que significa "proveedores": allí es donde se indican los proveedores de servicios que poseerá el controlador; en este caso, es el archivo "*usuarios.service.ts*".
En tercer lugar, está el apartado *exports*, que permitirá indicar qué partes de nuestro controlador se pueden utilizar fuera de este; en este caso, se indicó el "*usuariosService*", ya que es utilizado en "*authService*", como ya se ha visto.
Por último, está el apartado *controllers*, que es donde se indica el controlador de los *endpoints* del correspondiente controlador.
Todo este archivo se exporta con la línea:

- *export class UsuariosModule {}*

De esta manera, el controlador "app" (controlador principal) puede importar el controlador "Usuarios" y así dar a conocer de su existencia, para que pueda ser consultado desde otra aplicación.

```
src > usuarios > TS usuarios.module.ts > ...
1   import { Module } from '@nestjs/common';
2   import { UsuariosService } from './usuarios.service';
3   import { Usuario, UsuarioSchema } from './usuario.schema';
4   import { MongooseModule } from '@nestjs/mongoose';
5   import { UsuariosController } from './usuarios.controller';
6
7   @Module({
8     imports: [
9       MongooseModule.forFeature([{ name: Usuario.name, schema: UsuarioSchema }]),
10    ],
11    providers: [UsuariosService],
12    exports: [UsuariosService],
13    controllers: [UsuariosController],
14  })
15  export class UsuariosModule {}
16
```

Figura 28 - Composición de "usuarios.module.ts".
Fuente: Elaboración propia.

Usuarios.service.ts

"*Usuarios.service.ts*" está compuesto como se muestra en la "*Figura 29 – Composición de "usuarios.service.ts"*" y, como se puede observar, es un archivo mucho más grande, ya que en él se encuentra toda la lógica del controlador. Allí están programados cada uno de los requerimientos solicitados por el archivo "*usuarios.controller.ts*".

```
src > usuarios > usuarios.service.ts > UsuariosService > agregarUsuario > usuario
1   import { Injectable } from '@nestjs/common';
2   import { Model } from 'mongoose';
3   import { InjectModel } from '@nestjs/mongoose';
4   import { Usuario, UsuarioDocument } from './usuario.schema';
5
6   // This should be a real class/interface representing a user entity
7
8   @Injectable()
9   export class UsuariosService {
10    constructor(
11      @InjectModel(Usuario.name) private usuarioModel: Model<UsuarioDocument>,
12    ) {}
13
14    async findAll(): Promise<Usuario[]> {
15      const result = await this.usuarioModel.find().exec();
16      return result;
17    }
18
19    async buscarUsuario(email): Promise<Usuario> {
20      const result = await this.usuarioModel.findOne({
21        email: email,
22      });
23      return result;
24    }
25
26 >  async actualizarInformacionUsuario(body): Promise<boolean> {…
58    }
59
60    async agregarUsuario(body): Promise<string> {
61      const user = await this.buscarUsuario(body.email);
62
63      const usuario = {
64        foto: body.foto != null ? body.foto : '',
65        nombre: body.nombre != null ? body.nombre : '',
66        apellido: body.apellido != null ? body.apellido : '',
67        dni: body.dni != null ? body.dni : '',
68        direccion: body.direccion != null ? body.direccion : '',
69        fechaNac: body.fechaNac != null ? body.fechaNac : '',
70        genero: body.genero != null ? body.genero : '',
71        email: body.email != null ? body.email : '',
72        password: body.password != null ? body.password : '',
73        nombreUser: body.nombreUser != null ? body.nombreUser : '',
74      };
75
76      if (user != null) {
77        return 'El usuario ya existe';
78      } else {
79        const us = await this.usuarioModel.insertMany(usuario);
80        return 'usuario agregado';
81      }
82    }
83  }
```

Figura 29 - Composición de "usuarios.service.ts".

Fuente: Elaboración propia.

Al igual que los otros archivos, este está compuesto por una sección donde se importan las dependencias y una clase que contiene atributos y métodos.

En la parte de importación, además de importar algunas dependencias necesarias de NestJS, se realiza la importación de **"***usuario.schema***"** y este se lo pasa al constructor de la clase para asignarlo como atributo. Esta asignación del esquema del usuario es necesaria para operar con cada uno de los atributos que componen al documento de la base de datos, ya que este "*schema"* es una representación del documento de la base de datos.

Por su parte, los métodos de la clase que la componen son cuatro:

- findAll(): trae todos los usuarios de la base de datos que coincidan con el "*schema*" indicado.
- buscarUsuario(email): trae el usuario que coincida con el email recibido por parámetro y que coincida con el "*schema*" indicado.
- actualizarInformacionUsuario(body): actualiza la información del usuario que está contenido en el "*body*" recibido por parámetro. Este método actualiza cada campo del documento, solo si trae alguna diferencia en comparación con el que está en la base de datos.
- agregarUsuario(body): agrega un nuevo usuario a la base de datos, chequeando previamente que no exista en ella.

Controlador "Nodos"

El controlador "Nodos", a diferencia del controlador "Usuarios" que está compuesto por cuatro archivos, está compuesto por cinco archivos:

- Nodos.controller.ts.
- Nodos.module.ts.
- Nodos.service.ts.
- Nodos.schema.ts.
- configNodos.schema.ts.

El curso de consultas es igual que en cualquier otro controlador, pero la gran diferencia es que este controlador posee dos "*schemas*"; por ende, el servicio del controlador podrá hacer uso de dos documentos diferentes de la base de datos. Esto está diseñado de esta manera ya que este controlador se encarga no solo de la configuración de los nodos del sistema de microcontroladores, sino que también se encarga de consultar la última información subida a la base de datos de los diferentes nodos que posea el sistema de microcontroladores del usuario. Tanto la

configuración de los nodos como la última información son manejadas por documentos diferentes en la base de datos.

Nodos.controller.ts

El archivo “*nodos.controller.ts*” está compuesto de la manera que se indica en la “Figura 30 – Composición del archivo “*nodos.controller.ts”*”. Y, al igual que el “*usuarios.controller.ts*”, hace importación del servicio del controlador y lo asigna como atributo a la clase “*NodosController*”.
La clase está compuesta por siete *endpoints*:

- @Get('nodos'): solicita al servicio que traiga todos los nodos de un usuario.
- @Get('informacionPorNombre'): solicita al servicio que traiga todos los nodos que coincidan con el nombre recibido por parámetro.
- @Get('nodo'): solicita al servicio que traiga un solo nodo del usuario.
- @Post(''): indica al servicio que ingrese un nuevo nodo a la base de datos para el usuario pasado por parámetro “req”.
- @Get('configNodo'): solicita el servicio que le devuelva las configuraciones que poseen los nodos de un usuario.
- @Get('configUnNodo'): solicita al servicio que traiga la información de configuración del nodo que coincida con el nombre pasado por parámetro y que pertenezca al usuario.
- @Get('ultimaInformacionDeNodos'): solicita al servicio que envíe la última información registrada por un nodo que coincida con el nombre pasado por parámetro y que sea perteneciente al usuario.

```
src > nodos > TS nodos.controller.ts > NodosController > postNodo
import { Controller, Get, Post, Request } from '@nestjs/common';
import { NodosService } from './nodos.service';

@Controller('nodos')
export class NodosController {
  constructor(private nodoService: NodosService) {}

  //@UseGuards(JwtAuthGuard)
  @Get('nodos')
  async getNodos(@Request() req) {
    const result = await this.nodoService.findAll(req.query.email);
    return result;
  }

  @Get('informacionPorNombre')
  async getNodosPorNombre(@Request() req) {
    const result = await this.nodoService.findAllByNombre(
      req.query.email,
      req.query.nombre,
    );
    return result;
  }

  @Get('nodo')
  async getNodo(@Request() req) {
    const result = await this.nodoService.findOne(req.query.email);
    return result;
  }

  @Post('')
  async postNodo(@Request() req) {
    const result = await this.nodoService.insertOne(req.body);
    return result;
  }

  @Get('configNodo')
  async getNodoConfig(@Request() req) {
    const result = await this.nodoService.getConfiguracionNodos(
      req.query.email,
    );
    return result;
  }

  @Get('configUnNodo')
  async getNodoConfigPorNombre(@Request() req) {
    const result = await this.nodoService.getConfiguracionDeNodo(
      req.query.email,
      req.query.nombreNodo,
    );
    return result;
  }

  @Get('ultimaInformacionDeNodos')
  async getUltimaInformacionDeNodo(@Request() req) {
    const result = await this.nodoService
      .getUltimoIngreso(req.query.email, req.query.nombre)
      .then((response) => {
        //console.log(response);
        return response;
      });

    return result;
  }
}
```

Figura 30 - Composición del archivo "nodos.controller.ts".
Fuente: Elaboración propia.

Nodos.module.ts

El archivo "*nodos.module.ts*" está compuesto como se indica en la "*Figura 31 – composición del archivo "nodos.module.ts"*". En este archivo se realizan las

importaciones de todos los componentes del controlador y de los dos esquemas que posee el controlador. Esto lo realiza haciendo uso de MongooseModule**.**
También se hace referencia a cuál es el archivo "*controller*" que posee los *endpoints* y cuál es el proveedor de servicios del controlador. Como se mencionó en la explicación del controlador de "Usuarios", la clase "*NodosModule*" debe ser exportada para que pueda ser indicada en "app.module", ya que, si no es exportada, es como si no existiera y, por ende, no se podría acceder a ningún *endpoint*.

```
src > nodos > TS nodos.module.ts > NodosModule
1  import { Module } from '@nestjs/common';
2  import { MongooseModule } from '@nestjs/mongoose';
3  import { ConfigNodo, ConfigNodoSchema } from './configNodo.schema';
4  import { NodosController } from './nodos.controller';
5  import { Nodo, NodoSchema } from './nodos.schema';
6  import { NodosService } from './nodos.service';
7
8  @Module({
9    imports: [
10     MongooseModule.forFeature([
11       {
12         name: Nodo.name,
13         schema: NodoSchema,
14         collection: 'nodos',
15       },
16       {
17         name: ConfigNodo.name,
18         schema: ConfigNodoSchema,
19         collection: 'configNodo',
20       },
21     ]),
22   ],
23   controllers: [NodosController],
24   providers: [NodosService],
25 })
26 export class NodosModule {}
27
```

Figura 31 - Composición del archivo "nodos.module.ts".
Fuente: Elaboración propia.

Nodos.schema.ts y configNodos.schema.ts

Los archivos "*nodos.schema.ts*" y "*configNodos.schema.ts*" tienen la representación de los esquemas contenidos en la base de datos para los documentos "Nodos" y

“configNodos”. Estos dos esquemas se pueden observar en *la “Figura 32 – Esquemas de “nodos.schema.ts” y “configNodo.schema.ts””.*

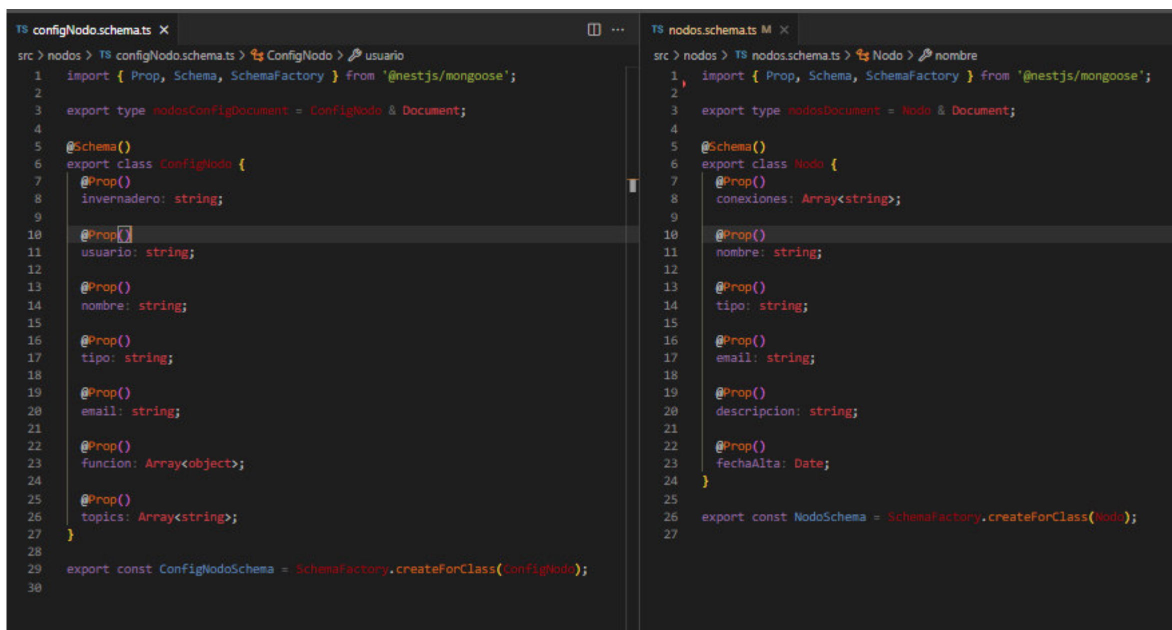

Figura 32 - Esquemas de “nodos.schema.ts” y “configNodo.schema.ts”.
Fuente: Elaboración propia.

Nodos.service.ts

El archivo “*nodos.service.ts*” está compuesto de la manera en que se lo presenta en la “Figura 33 – Composición del archivo “*nodos.service.ts*””.

Como ya es de costumbre, el archivo comienza con la importación de las dependencias necesarias de este, donde lo primordial para la programación son los modelos de los esquemas que utiliza este servicio.

En lo que respecta a la clase, en su constructor crea los atributos para los esquemas de la base de datos y, a diferencia del servicio del controlador “Usuarios”, acá hace uso de dos atributos, ya que este controlador usa dos esquemas diferentes. Cabe destacar que un controlador puede utilizar la cantidad de modelos que precise para sus necesidades.

Esta clase consta de siete métodos, los cuales son:

- findAll(email): busca en la base de datos todos los nodos correspondientes al email recibido por parámetro y los ordena de manera descendente.
- findAllByNombre(email, nombre): busca en la base de datos todos los documentos que corresponden al nombre del nodo y al email recibidos por parámetro.
- findOne(email): busca el último nodo cargado en la base de datos del email recibido por parámetro.
- insertOne(body): inserta un nodo en la base de datos y, por parámetro, recibe un "*body*", el cual debe estar compuesto como el esquema de un nodo.
- getConfiguracionNodos(email): trae la configuración de cada uno de los nodos que correspondan al email recibido por parámetro.
- getConfiguracionDeNodo(email, nombreNodo): trae la configuración de un nodo que coincida con el nombre recibido por parámetro y que pertenezca al email también recibido por parámetro.
- getUltimoIngreso(email, nombre): obtiene de la base de datos la última información ingresada en la base de datos, que corresponda al nombre del nodo y al email recibidos por parámetro.

```
nodos.service.ts 1, M
src > nodos > nodos.service.ts > NodosService
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { ConfigNodo, nodosConfigDocument } from './configNodo.schema';
import { Nodo, nodosDocument } from './nodos.schema';

@Injectable()
export class NodosService {
  constructor(
    @InjectModel(Nodo.name) private nodoModel: Model<nodosDocument>,
    @InjectModel(ConfigNodo.name) private configNodoModel: Model<nodosConfigDocument>,
  ) {}

  async findAll(email): Promise<Nodo[]> {
    return await this.nodoModel.find({ email: email }).sort({ fechaAlta: -1 });
  }

  async findAllByNombre(email, nombre): Promise<Nodo[]> {
    return await this.nodoModel
      .find({ email: email, nombre: nombre })
      .sort({ fechaAlta: -1 });
  }

  async findOne(email): Promise<Nodo> {
    const res = await this.nodoModel
      .find({ email: email })
      .sort({ fechaAlta: -1 })
      .limit(5);
    const result = res[0];
    return result;
  }

  async insertOne(body): Promise<Nodo> {
    return await this.nodoModel.create(body);
  }

  async getConfiguracionNodos(email): Promise<ConfigNodo[]> {
    return await this.configNodoModel.find({ email: email });
  }

  async getConfiguracionDeNodo(email, nombreNodo): Promise<ConfigNodo> {
    return await this.configNodoModel.findOne({
      email: email,
      nombre: nombreNodo,
    });
  }

  async getUltimoIngreso(email, nombre): Promise<Nodo[]> {
    return await this.nodoModel
      .find({ email: email, nombre: nombre })
      .sort({ fechaAlta: -1 })
      .limit(1);
  }
}
```

Figura 33 - Composición del archivo “nodos.service.ts”.

Fuente: Elaboración propia.

Carpeta auth

En esta carpeta se encuentra alojado “*auth.module.ts*” y “*auth.service.ts*”, que son los que se encargan de administrar los “guardianes”.

En “*auth.module.ts*” (ver “*Figura 34 – Estructura del archivo “auth.module.ts”*”) se registra el servicio “*auth.service.ts*” y los dos guardianes a utilizar (“*local-auth.guard.ts*” y “*jwt.guard.ts*”). Estos guardianes tienen estrategias definidas (“*local.strategy.ts*” y “*jwt.strategy.ts*”): en estos archivos están contenidos los métodos del guardián, y, por ende, su funcionalidad. En cambio, en “*local-auth.guard.ts*” y “*jwt.guard.ts*” solo se va a hacer la importación de la raíz de NestJS, ya que los guardianes son estrategias que ofrecen el *framework,* y se las va a exportar para indicar que están disponibles para su uso.

Además, se configura la duración de los *tokens* generados con:

- *JwtModule.register({secret: jwtConstants.secret, signOptions: { expiresIn: '60s' },})*

Por otro lado, “*auth.service.ts*” es quien se encarga de utilizar estos guardianes y asegurarse de que estén activos e intervengan en el flujo de la aplicación. Se compone de una clase (ver “*Figura 35 – Composición del archivo “auth.service.ts”*”) e importa el servicio del controlador “Usuarios”. En la definición de los atributos de clase crea dos atributos, a uno le asigna el servicio del controlador “Usuarios” y al otro el servicio del guardián de *tokens.*

Esta clase cuenta con dos métodos:

- validateUser(email, pass): solicita al servicio del controlador “Usuarios” que realice la validación de la información recibida por parámetro para validar si existe el usuario en la base de datos y la contraseña coincide con este,
- login(body): en base al email que se encuentra en el *body* recibido por parámetro, genera un *token* para poder utilizar para las siguientes consultas, en la aplicación.

```
src > auth > TS auth.module.ts > ...
1   import { Module } from '@nestjs/common';
2   import { AuthService } from './auth.service';
3   import { LocalStrategy } from './local.strategy';
4   import { JwtStrategy } from './jwt.strategy';
5   import { PassportModule } from '@nestjs/passport';
6   import { JwtModule } from '@nestjs/jwt';
7   import { jwtConstants } from './constants';
8   import { UsuariosModule } from 'src/usuarios/usuarios.module';
9
10  @Module({
11    imports: [
12      UsuariosModule,
13      PassportModule,
14      JwtModule.register({
15        secret: jwtConstants.secret,
16        signOptions: { expiresIn: '60s' },
17      }),
18    ],
19    providers: [AuthService, LocalStrategy, JwtStrategy],
20    exports: [AuthService],
21  })
22  export class AuthModule {}
23
```

Figura 34 - Estructura del archivo "auth.module.ts".

Fuente: Elaboración propia.

```
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { UsuariosService } from 'src/usuarios/usuarios.service';

@Injectable()
export class AuthService {
  constructor(
    private jwtService: JwtService,
    private usuariosService: UsuariosService,
  ) {}

  async validateUser(email: string, pass: string): Promise<any> {
    const user = await this.usuariosService.buscarUsuario(email);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }
  async login(body: any) {
    const payload = { username: body.email };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}
```

Figura 35 - Composición del archivo “auth.service.ts”.
Fuente: Elaboración propia.

LocalGuard y JwtGuard

El funcionamiento de los guardianes de la aplicación se define en las estrategias, como ya se ha mencionado.

Por un lado, el archivo “*local.strategy.ts*” está compuesto (ver “*Figura 36 – Composición de “local.strategy.ts”*”) por una clase que hace uso del servicio “*auth.service.ts*” y lo utiliza en un método llamado “validate()”, que, a su vez, solicita al servicio de “*auth.service*” que valide el usuario y la contraseña recibidos por parámetro.

Por otro lado, el archivo "*jwt.strategy.ts*" está compuesto (ver "Figura 37 – Composición de "*jwt.strategy.ts*"") por una clase que hace uso de una librería llamada "*passport-jwt*", que está incluida por el *framework* NestJS, y la asigna a un atributo de esta. Como en "*local.strategy.ts*", la clase también contiene un método "*validate(payload)*", que se encarga de validar que el *token pavload,* recibido por parámetro, sea válido.

La carpeta *auth* también posee un archivo que se llama "*constants.ts*", en el que solo se encuentra alojada una clave que es necesaria para la creación de los *tokens* con el guardián jwt.

```
src > auth > TS local.strategy.ts > ...
1  import { Strategy } from 'passport-local';
2  import { PassportStrategy } from '@nestjs/passport';
3  import { Injectable, UnauthorizedException } from '@nestjs/common';
4  import { AuthService } from './auth.service';
5  @Injectable()
6  export class LocalStrategy extends PassportStrategy(Strategy) {
7    constructor(private authService: AuthService) {
8      super();
9    }
10
11   async validate(email: string, password: string): Promise<any> {
12     const user = await this.authService.validateUser(email, password);
13     if (!user) {
14       throw new UnauthorizedException();
15     }
16     return user;
17   }
18 }
19
```

Figura 36 - Composición de "local.strategy.ts".

Fuente: Elaboración propia.

```
src > auth > TS jwt.strategy.ts > ...
1   import { ExtractJwt, Strategy } from 'passport-jwt';
2   import { PassportStrategy } from '@nestjs/passport';
3   import { Injectable } from '@nestjs/common';
4   import { jwtConstants } from './constants';
5
6   @Injectable()
7   export class JwtStrategy extends PassportStrategy(Strategy) {
8     constructor() {
9       super({
10        jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
11        ignoreExpiration: false,
12        secretOrKey: jwtConstants.secret,
13      });
14    }
15
16    async validate(payload: any) {
17      return { userId: payload.sub, username: payload.username };
18    }
19  }
20
```

Figura 37 - Composición de "jwt.strategy.ts".
Fuente: Elaboración propia.

Front End

Una vez creada la aplicación *Back-End*, resta crear la aplicación de *Front-End*. Esta aplicación va a ser la encargada de consumir los *endpoints* que le ofrece el *Back-End* y va a transformar la información recibida, de manera que para el usuario sea agradable, fácil y sencilla de interpretar. Y va a estar estructurada, de manera visual, en dos partes: la pantalla de inicio de sesión y la pantalla principal, con un menú con diferentes opciones que van modificando la pantalla principal.

Configuración del entorno de desarrollo

Para el desarrollo de la aplicación *Front-End*, se utilizó *Visual Studio Code*, como editor de código, y JavaScript, como lenguaje de programación, con la biblioteca React, la cual sirve para crear interfaces de usuario en una sola página. Además,

también se ha utilizado Ant-Design, que es un kit de interfaz de usuario que contiene componentes (ya se explicará qué es un componente), creados con un diseño más agradable para la visual del usuario.

Para la creación base de la aplicación se debe tener instalado NodeJS. Como ya estaba instalado, se realizaron los siguientes pasos:

- *npx create-react-app proyecto-huertas-ant-front-end.*

Esto crea la estructura básica del proyecto, la cual se puede observar en la "*Figura 38 – Estructura básica de aplicación React*". Allí se puede observar la carpeta "*node_modules*" que es donde se almacenan las funcionalidades de las dependencias externas a nuestra aplicación. Al igual que nuestra aplicación de *Back-End*, esta carpeta cumple la misma función.

La carpeta "*public*" contiene imágenes que utiliza la página y un archivo HTML que contendrá a todo el sistema pero que no debemos modificar.

Luego están los archivos "package.json" y "package-lock.json", que contienen los *scripts* para uso de desarrollo y puesta en producción, y, además, la referencia a todas las dependencias que posee nuestro sistema. Estos archivos también los posee nuestra aplicación *Back-End* ya que todo funciona sobre NodeJS. La diferencia entre las aplicaciones es la finalidad que posee cada una y la manera en que se desarrollan.

Para comprender la aplicación *Front-End*, tendremos en cuenta un concepto fundamental, los "componentes". Como lo fueron en la aplicación *Back-End* los controladores, los componentes son una composición de archivos JavaScript y "*css*". Un componente es la representación de una parte del sistema, desde, por ejemplo, un botón hasta toda la página principal, que es un componente que contiene otros componentes más pequeños.

En la estructura básica que se inicializa en una aplicación React, se crea un componente por defecto, el componente "App", que se encuentra dentro de la carpeta "src", en la que se irán agregando todos los nuevos componentes creados.

En lo que se refiere a “App”, está compuesto por un archivo de extensión “*.css*”, el cual tiene los estilos del componente (color, tamaño, etcétera) y un archivo “*.js*”, en el que está la lógica de qué es lo que hace el componente y cómo se muestra en la pantalla del usuario.

Por su parte, el archivo “*index.js*” es aquel en donde se inicializa el componente “App”, se lo muestra en la pantalla del usuario, no se modifica y siempre se renderiza solo el componente “App”. Todos los nuevos componentes creados se renderizan en “App”, de modo que siempre se renderizarán en la aplicación.

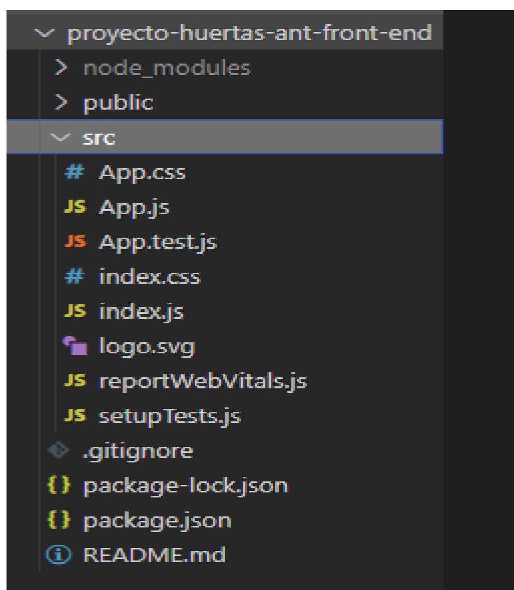

Figura 38 - Estructura básica de aplicación React.
Fuente: Elaboración propia.

Ya comprendida la estructura básica de un proyecto React, podemos observar la estructura del sistema *Front-End* de acuerdo a como se encuentra actualmente (ver “*Figura 39 – Composición del sistema Front-End actualmente*”). En lo que respecta al sistema *Front-End* actual, al ser tan grande, se torna demasiado extenso explicar el funcionamiento paso a paso, de cada uno de los componentes, pero sí resulta necesario saber para que se usan y cuándo son utilizados.

Quitando todo lo que corresponde al funcionamiento básico ya explicado, la aplicación *Front-End* se compone de tres carpetas:

- Servicios.
- Redux.
- Componentes.

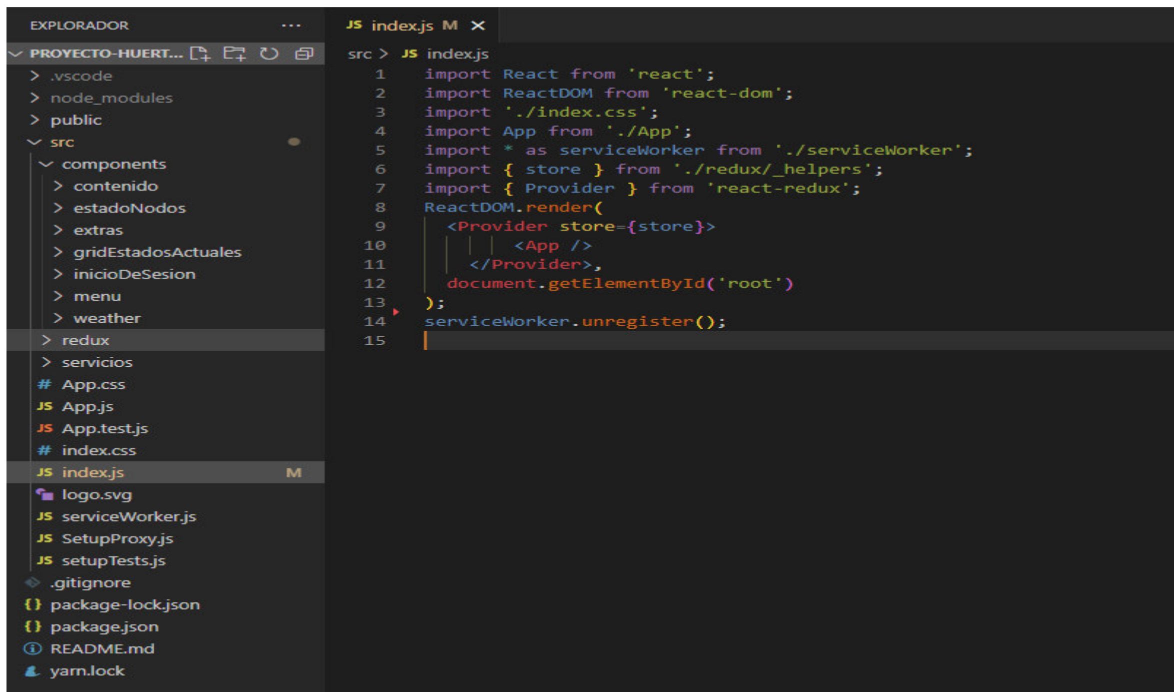

Figura 39 - Composición del sistema Front-End actualmente.
Fuente: Elaboración propia.

Para que nuestra aplicación funcione, también debemos instalar Ant Design, ya que algunos componentes de esta corresponden o requieren componentes propios de este kit de interfaz de usuario.
Para instalar Ant Design, solo debemos ingresar en la terminal del proyecto desde *Visual Studio Code*:

- *npm install antd.*

Esto agregará Ant Design como dependencia al proyecto y solo restará importar los componentes que se utilicen en los archivos que lo necesiten.

Carpeta servicios

La carpeta cuenta con dos archivos JavaScript ("*informacionDeNodoService.js*" e "*informacionDeUsuario.js*"). Estos son servicios que brindan métodos encargados de realizar las consultas a la aplicación *Back-End*.

informacionDeNodoService.js

Este servicio cuenta con siete métodos:

- getAllNodos(parametros): el parámetro "parametros" contiene el email y, a través de este, realiza la consulta a la Api *Back-End* para que le envíe todos los nodos del usuario.
- getNodo(parametros): "parametros" contiene el email del usuario y el nombre del nodo que se solicita a la Api.
- getConfigNodos(parametros): "parametros" contiene el email del usuario y solicita a la aplicación *Back-End* que le envíe la configuración de todos los nodos pertenecientes al usuario.
- registrarNodo(parametros): "parametros" contiene toda la información necesaria para la inserción de un nuevo nodo y la envía a la aplicación *Back-End* para que se encargue de almacenarla en la base de datos.
- getNodoPorNombre(parametros): "parametros" contiene el email del usuario y el nombre del nodo. Este método solicita al *Back-End* que envíe todos los documentos ingresados para el nodo correspondiente al nombre enviado. Esto trae un histórico de información correspondiente al nodo.
- getUltimaInformacionDeNodos(parametros): "parametros" contiene el email del usuario y el nombre del nodo. Este método solicita al *Back-End* que envíe la última información ingresada del nodo en la base de datos.
- getConfigUnNodo(parametros): "parametros" contiene el email del usuario y el nombre del nodo. Este método solicita al *Back-End* que le envíe la configuración correspondiente al nombre del nodo recibido por parámetro.

informacionDeUsuarioService.js

Este servicio cuenta con 6 métodos y dos funciones internas para el manejo de la sesión actual, que serán explicadas cuando nos adentremos en la carpeta "redux".

- getAllUsers(): Este método obtiene todos los usuarios almacenados en la base de datos.
- registrarUsuario(parametros): "parametros" contiene toda la información necesaria para la creación e inserción de un nuevo usuario en la base de datos.
- getUsuario(parametros): "parametros" contiene usuario y trae toda la información correspondiente al usuario recibido por parámetro.
- login(email,contra): este método es el utilizado para iniciar sesión en la Api *Back-End* y, a su vez, en la aplicación *Front-End* ya que, si el usuario y contraseña coinciden con los almacenados en la base de datos y son validados, la aplicación *Back-End* enviará el *token* para poder consultar todos los demás *enpoints*; como consecuencia, habilitará las demás pantallas operativas.
- cambiarContra(parametros): "parametros" contiene email y nueva contraseña a ser modificada por la antigua contraseña. Este método no se encuentra implementado en la aplicación *Back-End*.
- ponerFoto(parametros): "parametros" contiene una foto y el email correspondiente al usuario, al que se le desea asignar una foto al perfil. Este método no está implementado en la aplicación *Back-End.*

Carpeta redux

Ingresar en esta carpeta y explicar, paso a paso, el funcionamiento de los archivos que contiene es un tema para un proyecto en sí mismo. No obstante, explicaremos qué es "redux", cómo funciona y cómo está implementado en el proyecto.

¿Qué es Redux?

Redux es un contenedor predecible del estado de aplicaciones JavaScript y es una excelente herramienta para manejar el estado de una aplicación. Sus ventajas son:

- Posee un estado global e inmutable.
- Tiene un mayor control del estado de la aplicación y el flujo de datos.
- Posibilita una arquitectura escalable de datos.

La principal ventaja de Redux es cómo administra los cambios de estado (ver "*Figura 40 – Cambio de estados en Redux*") y, para entenderlo, es necesario conocer los siguientes tres conceptos:

- El Store como la única fuente de la verdad (donde se almacenan los estados).
- El State es de solo lectura (el estado actual que posee una parte del sistema).
- Los cambios al *State* pueden hacerse únicamente a través de acciones (actions) y funciones puras (reducers).

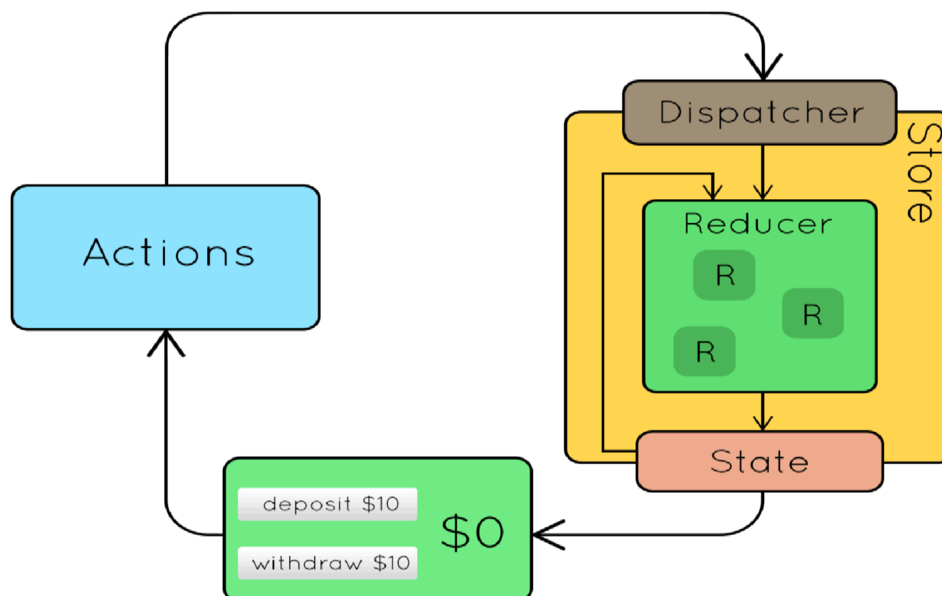

Figura 40 - Cambio de estados en Redux.

Fuente: https://dev.to/arielmirra/que-es-react-y-react-hooks-y-como-se-relacionan-4e1e.

Fecha de consulta: 15/06/2022.

De esta forma, se logra centralizar el estado de la aplicación y, por lo tanto, unificar el lugar para realizar cambios. Esto hace el desarrollo muchísimo más simple (ver "*Figura 41 – Comparación de aplicación React sin Redux y con Redux*").
Todo el estado de la aplicación del usuario está almacenado en un único árbol, dentro de un único *store*. La única forma de cambiar el árbol de estado es emitiendo una acción, la cual es un objeto, que describe qué ocurrió.
Para especificar cómo las acciones transforman el árbol de estado, se utilizan "*reducers*" puros.

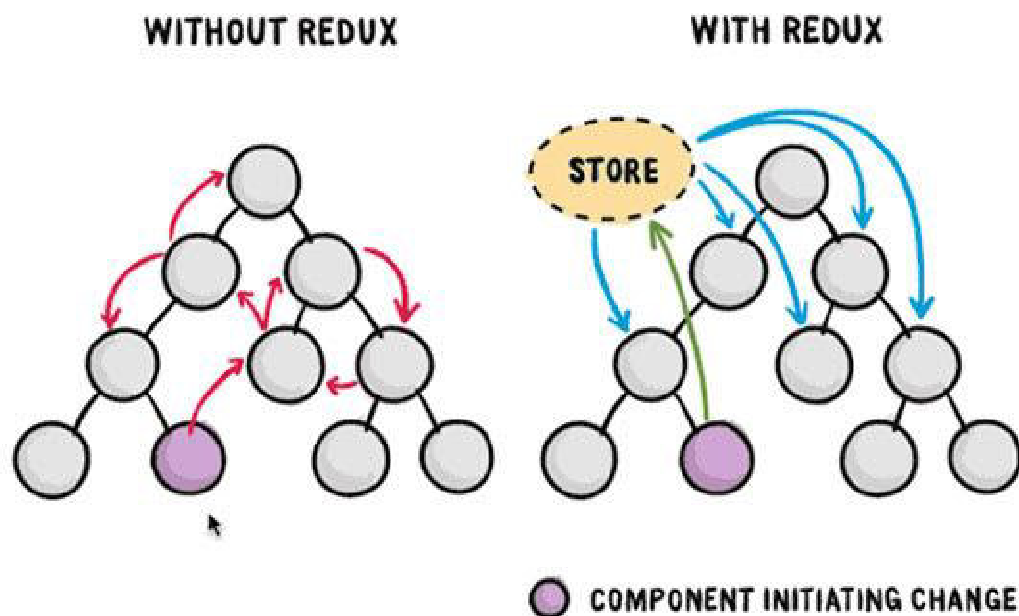

Figura 41 - Comparación de aplicación React sin Redux y con Redux.
Fuente: https://dev.to/arielmirra/que-es-react-y-react-hooks-y-como-se-relacionan-4e1e.
Fecha de consulta: 15/06/2022.

Ya explicado el funcionamiento básico de Redux, ahora podemos comprender el funcionamiento de redux en nuestra aplicación.

Nuestra carpeta redux está compuesta, a su vez, por seis carpetas que se encargan de dar funcionamiento a redux y cada una contiene archivos JavaScript (por razones de extensión, no serán explicados en detalle). Sabiendo esto, las carpetas son:

- _actions: contiene archivos que se encargan de registrar las solicitudes que realiza la aplicación.
- _components: contiene archivos que se encargan de validar que el componente al que se desea ingresar no corresponda a una ruta privada: esto quiere decir que, si el usuario no está validado en la aplicación e intenta ingresar a una dirección URL, correspondiente a una sección de la página que precise estar logueado en el sistema, lo redireccionará a la página de inicio de sesión, la cual no es una ruta privada.
- _constants: contiene una serie de archivos JavaScript que tienen definidas constantes, valores que no se pueden modificar y que van a ser los valores de los tipos de estados posibles que va a mantener la aplicación.
- _helpers: contiene archivos JavaScript que se encargan del manejo de inicio de sesión y la composición de las cabeceras de las solicitudes con el *token*, para utilizarlo en consultas a la Api *Back-End*.
- _reducers: haciendo uso de la carpeta "*_constants*", contiene archivos JavaScript preparados para hacer las modificaciones de los estados de la aplicación.
- _services: contiene los servicios necesarios para intervenir en las consultas que se realicen en la aplicación, para poder controlar todo el estado de la aplicación en un "*store".*

Como se puede observar en nuestro sistema de *Front-End,* redux está implementado para el manejo de sesión. Al realizarse la validación correctamente con la aplicación de *Back-End* y obtenido el *token*, redux se encarga de que el usuario navegue en la página y de modificar los estados, en caso de abandonar la página u ocurra un vencimiento del *token.*

Carpeta components

La carpeta "*components*" contiene carpetas en donde están definidos los componentes que son visibles en la aplicación, desde la pantalla de inicio de sesión hasta cada una de las opciones del menú del sistema, una vez que se ha iniciado sesión.

De esta carpeta se explicará solo cómo funciona la pantalla de "Log In" y cómo está compuesto el menú de navegación del usuario, una vez que inicia sesión, ya que aquel contiene cada uno de los demás componentes de la carpeta "*components*" y, en relación con las finalidades de esta PPS y por las mismas razones de extensión ya referidas, solo es importante explicar el funcionamiento en detalle de algunos de ellos.

La carpeta "*components*" está compuesta por seis carpetas que contienen archivos JavaScript:

- contenido: contiene cada uno de los componentes que representan una opción del menú principal del sistema.
- estadoNodos: contiene cada uno de los gráficos que se muestran en la opción "Estado de Nodos" del menú principal del sistema.
- extras: contiene un componente el cual es utilizado en la pantalla de inicio de sesión.
- inicioDeSesion: contiene los componentes de inicio de sesión y de registro de un nuevo usuario.
- menu: contiene el archivo que muestra la pantalla interactiva del usuario y en esta carpeta se utilizan la mayoría de los componentes del sistema.
- weather: contiene un componente que consume una Api externa para mostrar información climatológica de la semana.

inicioDeSesion

La pantalla de inicio de sesión del sistema (ver "*Figura 42 – Pantalla de inicio de sesión del Front-End*") se compone de un componente *Card*, que contiene un formulario (componente *Form*) y unos botones de "iniciar sesión", "olvidaste tu contraseña" y "registrarse ahora".

En esta pantalla de inicio, ingresando email y contraseña, y presionando el botón de "iniciar sesión", el usuario ejecuta todo el proceso de redux, explicado anteriormente, durante el que se va a realizar la consulta a la Api *Back-End*. Y, en caso de ser usuario y contraseña válidos, se obtendrá un *token* que redux lo insertará en el "Local Storage" del navegador y modificará el estado de la aplicación a logueado, lo que permitirá al usuario navegar por la aplicación libremente. Esto se mantendrá así hasta que el usuario presione "Salir" en el menú de la aplicación (ver "*Figura 43 – Menú de la aplicación Front-End*"), entonces es cuando redux volverá a modificar el estado de la aplicación, eliminando el *token* almacenado y cambiando el valor de las constantes a deslogueado.

El botón de "Regístrate Ahora" lleva a un formulario para registrar un usuario nuevo y el botón "olvidaste tu contraseña" lleva a una pantalla para restablecer tu contraseña. Estos botones, principalmente el de registrar un usuario nuevo, están pensados para que un usuario, sin las competencias o conocimientos en el tema, pueda registrar un sistema de control de invernaderos fácilmente. En la instancia actual del proyecto, esto aún no es posible ya que no están implementados los mecanismos de "dinamicidad" de los archivos de configuración del sistema de microcontroladores; por ende, si registran un nuevo usuario no podría ser ligado, desde la aplicación de *Front-End*, a un sistema de microcontroladores, por esta razón, se ha creado solo un usuario que está relacionado con un sistema de microcontroladores. En caso de que exista la necesidad de agregar un nuevo usuario, el sistema web te permite agregarlo, pero en el sistema de microcontroladores se debe indicarlo por el archivo de configuración del Suscriber manualmente.

Figura 42 - Pantalla de inicio de sesión del Front-End.
Fuente: Elaboración propia.

Al momento de que un usuario inicia sesión, se puede apreciar que la pantalla principal que muestra el sistema (inicio) está compuesta con un menú lateral, un mensaje de alerta y dos componentes. Por una parte, uno que muestra la última información registrada por los nodos del usuario y los parámetros en que los sensores de los nodos deberían estar registrando información. Cabe recordar que, si hay desvío de algún valor de esos parámetros indicados, el sistema de microcontroladores enviará una alerta vía email al usuario, indicando qué nodo y qué sensor está fuera de los parámetros. Por otra, un componente, mencionado con anterioridad, que consume una Api externa, que trae información actualizada del clima del día y de los próximos días.

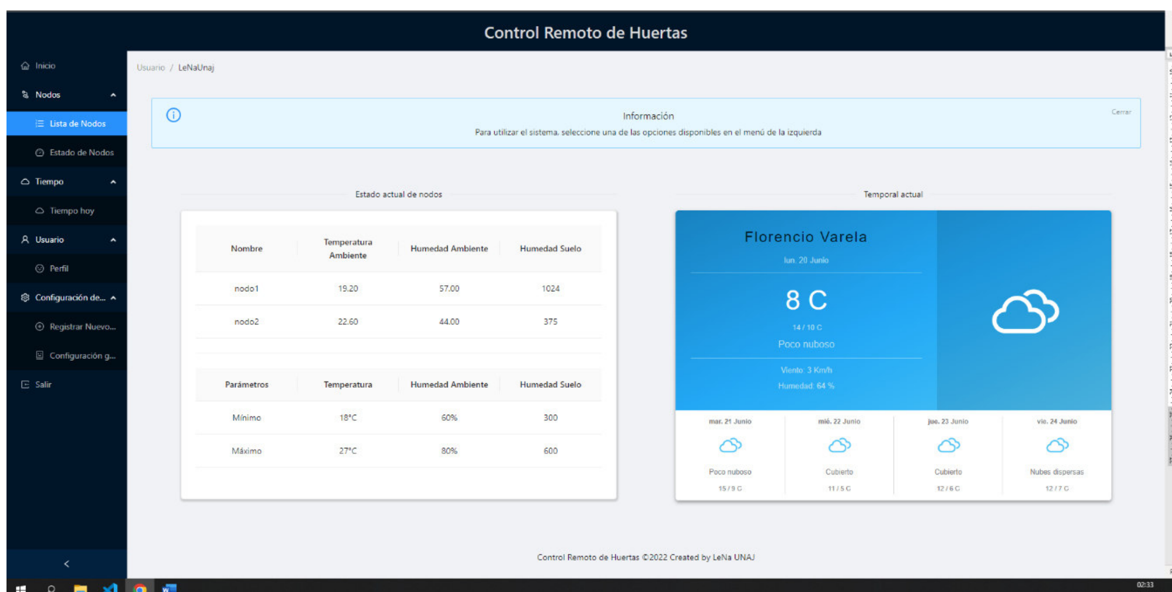

Figura 43 - Menú de la aplicación Front-End.
Fuente: Elaboración propia.

Cada una de las opciones del menú ejecutan componentes, que están indicados en el archivo "*menuOcultable.js*" (según la opción que el usuario elija del menú, solo se reemplazará el componente central de la página). Esto quiere decir que, al iniciar sesión, la página comienza con dos componentes precargados (estado actual de los nodos y temporal). Al momento de seleccionar otra opción, se carga otro componente y se quitan los que se mostraban en ese momento. Para volver a cargar los primeros dos componentes, se selecciona la opción inicio del menú.
Entendido el funcionamiento de las opciones del menú, solo queda indicar qué hacen cada una de las opciones:

- Inicio: carga los componentes de la pantalla principal (los mismos componentes que aparecen cuando se inicia sesión).
- Nodos-Lista de Nodos: despliega la lista de nodos perteneciente al usuario, con información de configuración de cada nodo.
- Nodos-Estado de Nodos: es la opción más importante hasta el momento del sistema. Permite seleccionar el nodo y luego, uno de sus sensores, y despliega un gráfico, con todas las mediciones en el tiempo del nodo, lo que

permite analizar la información de las mediciones previas de los nodos y prevenir o mitigar accidentes en los invernaderos del usuario.

- Tiempo-Tiempo hoy: en el momento actual, solo muestra el mismo componente de la página de "inicio"
- Usuario-Perfil: permite modificar los datos del usuario actual, excepto el email, ya que este es el que liga al usuario con su sistema de microcontroladores.
- Configuración del sistema-Registrar nuevo nodo: permite registrar un nuevo nodo en la base de datos. Esta opción del menú está operativa ya que se puede registrar un nuevo nodo de manera correcta, pero el sistema aún no está capacitado para ligarlo con el sistema de microcontroladores de manera automática.
- Configuración del sistema-Configuración general: opción aún no desarrollada pero lista para ser utilizada, para crear alguna pantalla de configuración.

Integración

Ya creados el sistema de microcontroladores y el sistema web, con sus partes que lo integran, es necesaria la integración de estos dos para que funcionen como un solo sistema que opera en conjunto. Esta integración se obtiene a través de la base de datos que se encuentra alojada en la nube. Establecida una conexión por parte del sistema web para obtener la información almacenada en la base de datos, y establecida una conexión del sistema de microcontroladores para almacenar información en la base de datos en la nube, se completa el ciclo de funcionamiento por separado y se inicia un funcionamiento en conjunto del sistema como uno solo. Para uso de la base de datos se utilizará MongoDB. Esta es una base de datos no relacional, la cual no funciona con tablas como una base de datos relacional, sino que funciona con documentos o colecciones.

La base de datos se encuentra contenida en Mongo Atlas (https://www.mongodb.com/atlas/database), el cual es un servicio gratuito (hasta 512MB de almacenamiento) que ofrece la compañía MongoDB, que permite hacer *hosting*[11] de una base de datos de MongoDB en la nube. Para ello, es necesario registrarse en la aplicación y con una serie de clics se pone en funcionamiento un clúster de bases de datos de MongoDB en la nube.

El clúster en la nube, que es donde se crea la base de datos, nos brindará una cadena de conexión que estará compuesta por el usuario, el nombre de la base de datos y una contraseña que se genera al momento de la creación de esta. La cadena de conexión que se genera (ver "*Figura 44 – Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación Python versión 3.6 o superior*") permitirá al usuario conectarse de manera remota, desde su aplicación a la base de datos, introduciendo solamente, en esta cadena de conexión, sus credenciales.

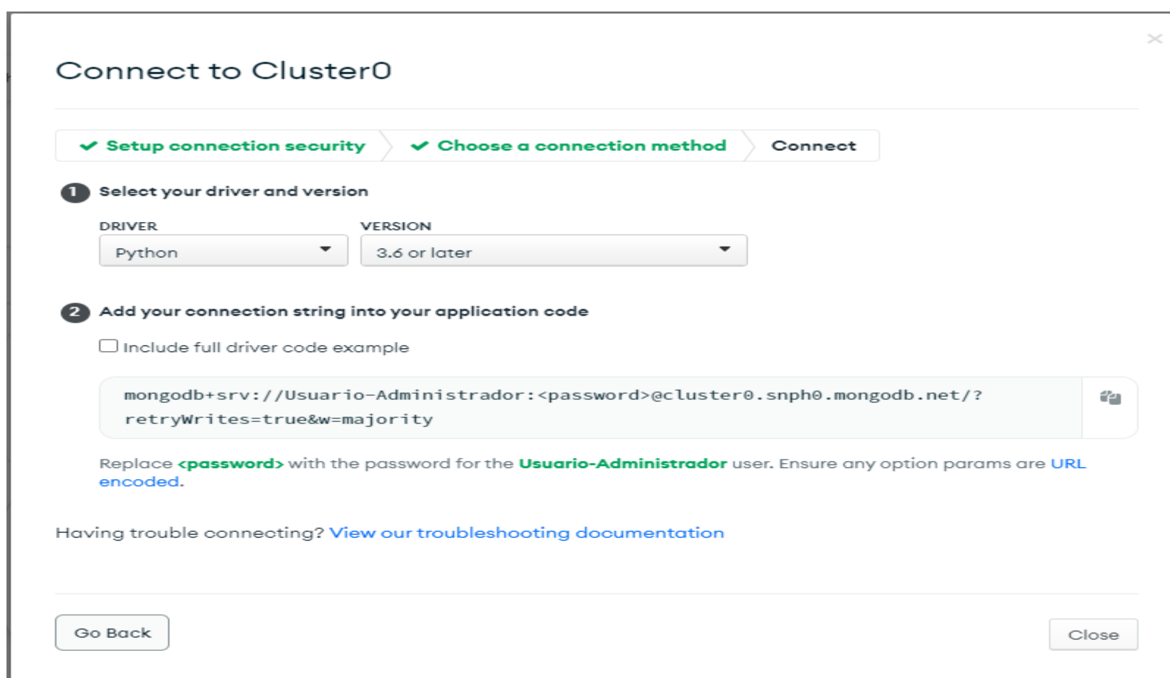

Figura 44 - Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación Python versión 3.6 o superior.
Fuente: Elaboración propia.

[11] *Hosting*: proceso de publicación de una aplicación en un servidor que pertenece a una empresa.

Conexión del Sistema web y la base de datos

Para referirnos a la conexión del sistema web con la base de datos, solo nos debemos centrar en la aplicación de *Back-End*, ya que esta es la encargada de realizar las consultas a la base de datos para obtener e ingresar información.

Para realizar la conexión a la base de datos, precisamos de la cadena de conexión que nos brinda el clúster. En este caso, la conexión es para una aplicación NodeJS[12], ya que el *Back-End* está creado en NestJS, el cual es un *framework* que funciona sobre NodeJS. De este modo, la cadena de conexión al solicitarla en la plataforma web queda de la siguiente manera, expresada en la “*Figura 45 – Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación NodeJS 4.1 o superior*”.

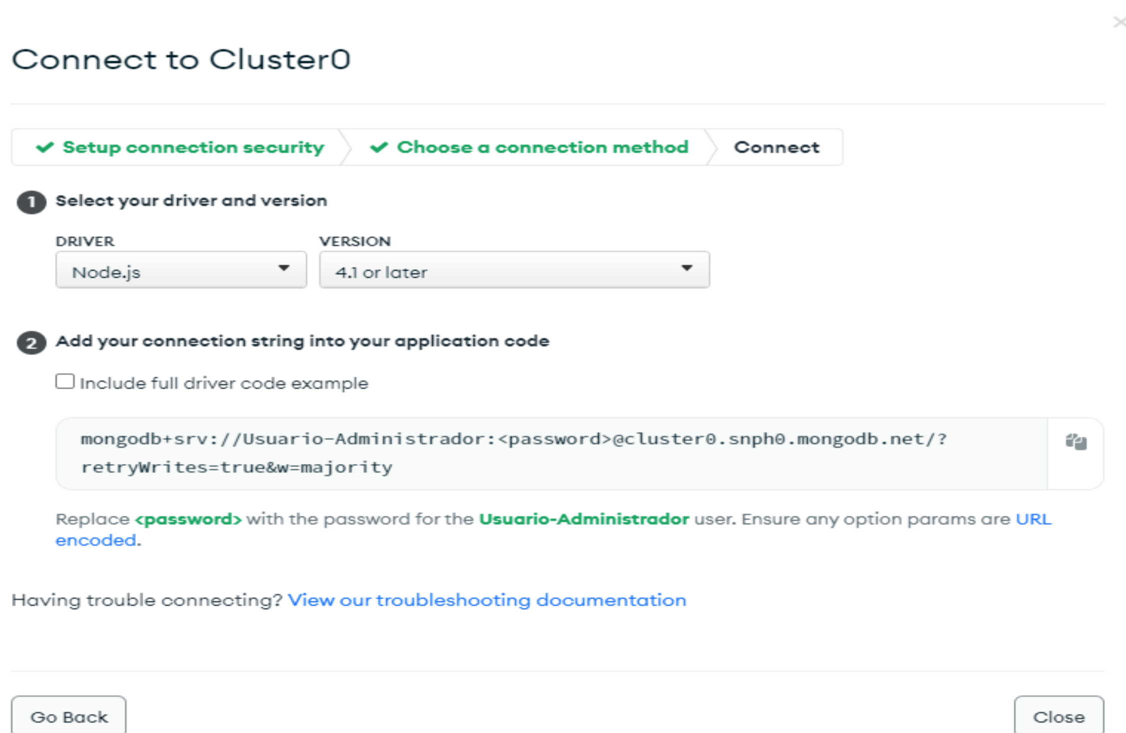

Figura 45 - Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación NodeJS 4.1 o superior.

Fuente: Elaboración propia.

[12] NodeJS es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript.

Utilizando esta cadena de conexión podemos conectar nuestra aplicación de *Back-End* desde el archivo "*app.module.ts*", ya que, en este archivo, es donde se realizan las importaciones de las dependencias necesarias que precisen otras partes de la aplicación, como, en este caso, la conexión a la base de datos para que pueda ser utilizada por los demás archivos del programa.

Conexión del Sistema de microcontroladores y base de datos

En esta instancia del proyecto, cuando ya está creado todo el sistema de microcontroladores, es necesario almacenar la información en la base de datos. Como ya se ha determinado, la base de datos está contenida en la nube. Para realizar la conexión a la base de datos es necesario, como ya se ha explicado, la cadena de conexión cedida por el clúster contenido en la plataforma web.

En lo que respecta al sistema de microcontroladores, la conexión se realiza para una aplicación creada en Python. Por ende, es necesario solicitarle al sistema web (donde está alojado el clúster) que nos dé una cadena de conexión para una aplicación escrita en este lenguaje de programación (ver "*Figura 44 – Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación Python versión 3.6 o superior*").

Dada la arquitectura implementada en nuestro sistema de microcontroladores, solo necesitamos que nuestro *Suscriber* se conecte a la base de datos. A tal efecto, en nuestra aplicación del *Suscriber*, creamos un módulo que contiene funcionalidades para que el usuario pueda conectarse con la base de datos, incluida la cadena de conexión, y realizar una serie de consultas básicas y necesarias para el funcionamiento de la aplicación.

El módulo "*database_connection.py*", en esta primera versión de la aplicación *Suscriber*, posee seis métodos, que son las funciones:

1. *get_database()*.
2. *update_nodo(informacion).*

3. *insert_information_nodo(informacion).*
4. *insert_configuracion().*
5. *comprobarAlerta(informacion).*
6. *enviarMail(informacion).*

get_database()

La función *get_database()* (ver “*Figura 46 – Función get_database() correspondiente al código Python contenido en database_connection.py*”) comienza con la importación de “*MongoClient*”, que pertenece a la librería Pymongo. *MongoClient* nos va a permitir conectarnos a una base de datos, ya que por parámetro se le debe ingresar una cadena de conexión de una base de datos de tipo mongoDB.

Para realizar la conexión a nuestra base de datos en la nube, solo basta con utilizar el MongoClient que importamos de Pymongo y pasarle por parámetro la cadena de conexión que devuelve Mongo Atlas al momento de la creación de clúster, que contiene las bases de datos del usuario.

Esta acción la asignamos a una variable llamada “cliente”. En dicha variable cliente, accedemos a través de corchetes al nombre de la base de datos a la que queremos acceder. Con esa acción, la función ***get_database()*** retorna la base de datos indicada en los corchetes. En otras palabras, devolvemos la base de datos que indicamos entre corchetes, luego de hacer la conexión a Mongo Atlas, para que sea usada como se necesite, en los siguientes pasos del programa.

```
def get_database():
    # Importa MongoClient de la lilbreria de pymongo
    from pymongo import MongoClient

    #cadena de conexion para conectar al cluster que contiene la base de datos en  Mongo Atlas
    CONNECTION_STRING = "mongodb+srv://[illegible]:[illegible]@Cluster0.snph0.mongodb.net/?retryWrites=true&w=majority"

    #asignacion del cluster a un cliente
    client = MongoClient(CONNECTION_STRING)

    #obtiene la base de datos especificada entre corchetes del cluster
    return client['Huertas-Ant']
```

Figura 46 - Función get_database() correspondiente al código Python contenido en database_connection.py. Fuente: Elaboración propia.

update_nodo(informacion)

La función *update_nodo(información)* (ver "*Figura 47 – Función update_nodo(informacion) correspondiente al código Python contenido en database_connection.py.*") recibe por parámetro información que el *Suscriber* quiere actualizar en la base de datos. Al inicio de la función, se asigna a una variable la base de datos, utilizando la función *get_database()*, y se la utiliza para acceder a la colección "nodos".

De acuerdo con la información que recibe por parámetro, el usuario realiza una consulta a la base de datos para buscar el nodo especifico que quiere actualizar y, una vez recibida la respuesta con los datos del nodo, se realiza un recorrido de todas las conexiones que posee ese nodo y se actualiza el indicado en la información que recibimos previamente por parámetro.

Esta función solo realiza la actualización correspondiente, sin dar ninguna respuesta al *Suscriber*, ya que se va a estar ejecutando cada cinco minutos; en caso de que hubiera un fallo, en los próximos cinco minutos se corregiría.

```
46  def update_nodo(informacion):
47      dbname = get_database()
48      collection = dbname["nodos"]
49
50      #primero hago un find para ver que existe el nodo y obtener los topicos que posee
51      result = collection.find({"$and":[{"email":informacion["email"]},{"nombre":informacion["nombreNodo"]}]})
52
53      #para poder acceder a la colleccion resultado de la bd se hace un for donde valor es la colleccion y es solo una posicion de recorrido
54      for valor in result:
55
56          if(informacion["topicUpdate"] == "temperatura"):
57              conexVieja = valor["conexiones"]  #conexion actual a ser recorrida para obtener la posicion del valor a realizar update
58              for sensor in conexVieja:
59
60                  if sensor["nombre"] == "Temperatura Ambiente":
61                      collection.update_one({"email":informacion["email"], "nombre":informacion["nombreNodo"], "conexiones.nombre": "Temperatura Ambiente"}, {"$set":{"conexiones.$.valor":informacion["valorUpdate"]}})
62
63
64          if(informacion["topicUpdate"] == "humedadSuelo"):
65              conexVieja = valor["conexiones"]
66              for sensor in conexVieja:
67                   if sensor["nombre"] == "Humedad de Suelo":
68                      collection.update_one({"email":informacion["email"], "nombre":informacion["nombreNodo"], "conexiones.nombre": "Humedad de Suelo"}, {"$set":{"conexiones.$.valor":informacion["valorUpdate"]}})
69
70
71          if(informacion["topicUpdate"] == "humedadAmbiente"):
72              conexVieja = valor["conexiones"]
73              for sensor in conexVieja:
74                  if sensor["nombre"] == "Humedad Ambiente":
75                      collection.update_one({"email":informacion["email"], "nombre":informacion["nombreNodo"], "conexiones.nombre": "Humedad Ambiente"}, {"$set":{"conexiones.$.valor":informacion["valorUpdate"]}})
76
77
```

Figura 47 - Función update_nodo(informacion) correspondiente al código Python contenido en database_connection.py.
Fuente: Elaboración propia.

insert_information_nodo(información)

La función *insert_information_nodo(información)* (ver "*Figura 48 – Función insert_information_nodo(informacion) parte uno*" y ver "*Figura 49 – Función*

insert_information_nodo(information) parte dos”) recibe por parámetro información que el *Suscriber* quiere ingresar en la base de datos. Al inicio de la función, la base de datos le asigna a una variable, utilizando a su vez la función *get_database()* para acceder a la colección “nodos”.

Luego se realiza una consulta con la información recibida para chequear que exista el nodo y así obtener los *topics* que posee. En base a estos *topics* y a la información recibida por parámetro, se realiza una comparación de los *topics* recibidos de la base y los *topics* contenidos en el parámetro para identificar a cuál *topic* corresponde la información a ser actualizada en la base de datos. En función de estas comparaciones, se crea un objeto, como el modelo de la colección “nodos” de la base de datos, y se realiza un *setting* de cada uno de los campos, con los valores actualizados, y se los ingresa en la base de datos como un nuevo documento.

```
18  def insert_information_nodo(informacion):
19      import pymongo as pai
20      dbname = get_database()
21      collection = dbname["nodos"]
22
23      #primero hago un find para ver que existe y obtener los topicos que posee
24      result = collection.find({"$and":[{"email":informacion["email"]},{"nombre":informacion["nombreNodo"]}]}).sort("fechaAlta", pai.DESCENDING)
25      print('entre')
26      item = {
27
28          "conexiones" : ["","",""],
29          "nombre" : informacion["nombreNodo"],
30          "tipo" : informacion["tipo"],
31          "email" : informacion["email"],
32          "descripcion" : informacion["descripcion"],
33          "fechaAlta": informacion["fechaAlta"]
34      }
35      #para poder acceder a la colleccion resultado de la bd se hace un for donde valor es la colleccion y es solo una posicion de recorrido
36      for valor in result:
37          conexVieja = valor["conexiones"]
38          if(informacion["topicUpdate"] == "temperatura"):
39
40              conexVieja = valor["conexiones"]  #conexion actual a ser recorrida para obtener la posicion del valor a realizar update
41              for sensor in conexVieja:
42
43                  if sensor["nombre"] == "Temperatura Ambiente":
44                      #print("este es el index: " + str(conexVieja.index(sensor)))
45                      item["conexiones"][0] = {"nombre":"Temperatura Ambiente","valor":informacion["valorUpdate"]}
46                  if sensor["nombre"] == "Humedad de Suelo":
47                      item["conexiones"][1] = sensor
48                  if sensor["nombre"] == "Humedad Ambiente":
49                      item["conexiones"][2] = sensor
50
51          if(informacion["topicUpdate"] == "humedadAmbiente"):
```

Figura 48 - Función insert_information_nodo(informacion) parte uno.

Fuente: Elaboración propia.

```
51          if(informacion["topicUpdate"] == "humedadAmbiente"):
52
53              conexVieja = valor["conexiones"]  #conexion actual a ser recorrida para obtener la posicion del valor a realizar update
54              for sensor in conexVieja:
55
56                  if sensor["nombre"] == "Temperatura Ambiente":
57                      #print("este es el index: " + str(conexVieja.index(sensor)))
58                      item["conexiones"][0] = sensor
59                  if sensor["nombre"] == "Humedad de Suelo":
60                      item["conexiones"][1] = sensor
61                  if sensor["nombre"] == "Humedad Ambiente":
62                      item["conexiones"][2] = {"nombre":"Humedad Ambiente","valor":informacion["valorUpdate"]}
63
64          if(informacion["topicUpdate"] == "humedadSuelo"):
65
66              conexVieja = valor["conexiones"]  #conexion actual a ser recorrida para obtener la posicion del valor a realizar update
67              for sensor in conexVieja:
68
69                  if sensor["nombre"] == "Temperatura Ambiente":
70                      #print("este es el index: " + str(conexVieja.index(sensor)))
71                      item["conexiones"][0] = sensor
72                  if sensor["nombre"] == "Humedad de Suelo":
73                      item["conexiones"][1] = {"nombre":"Humedad de Suelo","valor":informacion["valorUpdate"]}
74                  if sensor["nombre"] == "Humedad Ambiente":
75                      item["conexiones"][2] = sensor
76          print("division")
77          print(valor)
78          print("division")
79          print(item)
80
81          break
82      collection.insert_many([item])
```

Figura 49 - Función insert_information_nodo(informacion) parte dos.

Fuente: Elaboración propia

insert_configuracion()

El método *insert_configuracion()* (ver *"Figura 50 – Método insert_configuracion() parte 1" y "Figura 51 – Método insert_configuracion() parte 2")* crea un objeto como el modelo de la configuración de un nodo de la base de datos y, en base al archivo de configuración del *Suscriber,* ingresa toda la configuración correspondiente al nodo. Si la configuración del nodo tiene el valor "nuevosParametros" en *True,* esta función se encarga de actualizar los datos contenidos en la base de datos; si no, mantiene los parámetros mínimos y máximos configurados en la base actualmente.

```python
122 def insert_configuracion():
123     import usuario_config as config
124     dbname = get_database()
125     collection = dbname["configNodo"]
126     #TOPIC_EJEMPLO = cfg.config["invernadero"]+"/"+(cfg.config["nodos"][0])["nombre"]+"/"+(cfg.config["nodos"][0])["topics"][0]
127     listaConfigInsert = []
128 
129     configNodo = {
130             "invernadero": config.config["invernadero"],
131             "usuario": config.config["nombreUser"],
132             "email": config.config["email"],
133             "nombre":"",
134             "tipo":"",
135             "funcion": [{ "nombre":"",
136                     "minHumedadAmbiente":"",
137                     "minHumedadSuelo":"",
138                     "minTemperatura":"",
139                     "maxHumedadSuelo":"",
140                     "maxTemperatura":"",
141                     "maxHumedadAbiente":"",
142                 }],
143             "topics":[]
144         }
145 
146     for indice in range(len(config.config["nodos"])):
147 
148         configNodo["nombre"] = (config.config["nodos"][indice])["nombre"]
149         (configNodo["funcion"][0])["nombre"] = (config.config["nodos"][indice])["funcion"]
150         configNodo["topics"] = (config.config["nodos"][indice])["topics"]
151         configNodo["tipo"] = (config.config["nodos"][indice])["tipo"]
152 
153         query = { "nombre": configNodo["nombre"] };
154         upsert = True
```

Figura 50 - Método insert_configuracion() parte 1.

Fuente: Elaboración propia.

```python
155 
156         #el valor nuevosParametros indicado en el archivos de configuracion indica si hay que modificar los parametros de
157         #alertas de los sensores, el seteo de las variables se da por Api, aca se indica solo el tipo de plantacion
158         if ((config.config["nodos"][indice])["nuevosParametros"] == True):
159             update = { "$set": {
160                     "invernadero": configNodo["invernadero"],
161                     "usuario": configNodo["usuario"],
162                     "email": configNodo["email"],
163                     "nombre": configNodo["nombre"],
164                     "tipo": configNodo["tipo"],
165                     "funcion": configNodo["funcion"],
166                     "topics": configNodo["topics"]
167                     }
168                     };
169         else:
170             update = { "$set":
171             { "invernadero": configNodo["invernadero"],
172               "usuario": configNodo["usuario"],
173               "email": configNodo["email"],
174               "nombre": configNodo["nombre"],
175               "tipo": configNodo["tipo"],
176               "topics": configNodo["topics"]
177             }};
178         collection.update_one(query, update, upsert);
179 
```

Figura 51 - Método insert_configuracion() parte 2.

Fuente: Elaboración propia.

comprobarAlerta(informacion)

El método *comprobarAlerta(informacion)* (ver "*Figura 52 – Método comprobarAlerta(informacion)")* consulta la información de configuración que tiene el nodo y chequea los parámetros que tiene configurados con los que ingresan por parámetro, y los compara. Este método devuelve "*True*" si hay algún desvío y "*False*", en caso contrario. Ante una devolución de "*True*", el programa principal del *Suscriber* llama al método *enviarMail().*

```
def comprobarAlerta(informacion):
    import usuario_config as config
    dbname = get_database()
    collection = dbname["configNodo"]
    email = config.config["email"]
    result = collection.find({"$and":[{"email": email}, {"nombre":informacion["nombreNodo"]}]})
    topic = informacion["topicUpdate"]
    parametros ={
        "minimo":"",
        "maximo":""
        }

    for res in result:
        for topics in res["topics"]:
            if(topic == "temperatura" and topic == topics):
                parametros["minimo"] = res["funcion"][0]["minTemperatura"]
                parametros["maximo"] = res["funcion"][0]["maxTemperatura"]
            if(topic == "humedadAmbiente" and topic == topics):
                parametros["minimo"] = res["funcion"][0]["minHumedadAmbiente"]
                parametros["maximo"] = res["funcion"][0]["maxHumedadAmbiente"]
            if(topic == "humedadSuelo" and topic == topics):
                parametros["minimo"] = res["funcion"][0]["minHumedadSuelo"]
                parametros["maximo"] = res["funcion"][0]["maxHumedadSuelo"]
    print(parametros)
    if((parametros["minimo"] < informacion["valorUpdate"]) and (parametros["maximo"] > informacion["valorUpdate"])):
        #no hay alerta

        return False
    else:
        #hay alerta

        return True
```

Figura 52 - Método comprobarAlerta(informacion).
Fuente: Elaboración propia.

enviarMail(informacion)

El método *enviarMail(informacion)* (ver "*Figura 53 – Método enviarMail(informacion)*"*)* importa la librería "smtplib" para poder enviar un email, a

través de una cuenta de Google. En el método, se utiliza una clave brindada por Google y una cuenta que fue creada para realizar este envío de emails (por ejemplo, sistemahuertasunaj@gmail.com).

Se crea un cuerpo de email que contiene el *topic*, el nombre del nodo y el valor registrado por el sensor. En el asunto contiene el mensaje "¡Aviso de alerta de invernadero!", y este email, ya armado, se envía al email registrado en el archivo de configuración del *Suscriber*.

```
213
214 def enviarMail(informacion):
215     import smtplib
216     import usuario_config as config
217     from email.mime.multipart import MIMEMultipart
218     from email.mime.text import MIMEText
219     mensaje_format = """   Se ha recibido la alerta del sensor de: {0} del nodo: {1}.
220     Valores Obtenidos: {2}. El cual está fuera de los parámetros ideales.
221                 """
222     content = mensaje_format.format(informacion["topicUpdate"], informacion["nombreNodo"], informacion["valorUpdate"])
223     sender_address = 'sistemahuertasunaj@gmail.com'
224     sender_pass = 'ztbieitrzrgraejx'
225     receiver_address = config.config['email']
226
227     message = MIMEMultipart()
228     message['From'] = sender_address
229     message['To'] = receiver_address
230     message['Subject'] = '¡Aviso de alerta de invernadero!'
231
232     message.attach(MIMEText(content, 'plain'))
233     session = smtplib.SMTP('smtp.gmail.com', 587)
234     session.starttls()
235     session.login(sender_address, sender_pass)
236     text = message.as_string()
237     session.sendmail(sender_address, receiver_address, text)
238     session.quit()
239     print('Mail Sent')
240
```

Figura 53 - Método enviarMail(informacion).

Fuente: Elaboración propia.

Pruebas de funcionamiento y resultados

En lo que respecta a las pruebas de funcionamiento y los resultados obtenidos, solo se han hecho pruebas locales y en entornos de prueba, ya que, por limitaciones de tiempo, no se ha podido realizar la instalación del sistema en un entorno real. Sin embargo, esto no fue un inconveniente en lo que respecta a pruebas del sistema de microcontroladores porque los valores que recolectaban los sensores no se alejaban de los parámetros de una prueba de campo real, ya que los nodos estaban

configurados para una plantación de tomates y los valores obtenidos rondaban valores similares a los que deberían rondar, si estuviese en una plantación real. De todos modos, también se forzaron los parámetros para que se desviaran de sus valores, así el sistema enviaba las alertas de inconvenientes ocurridos al mail.

Por otra parte, se probó el sistema web para comprobar que el inicio de sesión, el menú y cada componente que tiene creado el sistema funcionaran correctamente. Para lograr esto, se montó tanto el sistema Back-End como el Front-End en Heroku (que ofrece 100MB de lectura y una base de datos de 5MB gratuitos). Para realizar estas pruebas, fue necesario que el sistema de microcontroladores ya hubiera insertado información en la base de datos; si no, no se podrían mostrar algunos funcionamientos del sistema web.

Pruebas de alertas del sistema de microcontroladores

Cada nodo tiene almacenado en la base de datos un archivo de configuraciones con todas las especificaciones. Esta configuración del nodo, almacenada en la base de datos, es utilizada por el sistema de microcontroladores para identificar información básica del nodo; por ejemplo, qué tipo de plantación y qué parámetros (mínimos y máximos) tiene configurado. A su vez, estos parámetros son utilizados para enviar las alertas, en caso de que el valor obtenido del sensor no supere el mínimo o se excede del máximo permitido (en tales casos, el sistema envía un email).

En la “*Figura 54 – Email recibido por desvío de parámetros*”, se puede apreciar el email recibido y enviado luego de que en el sistema de microcontroladores hubiera percibido una anomalía en los valores obtenidos.

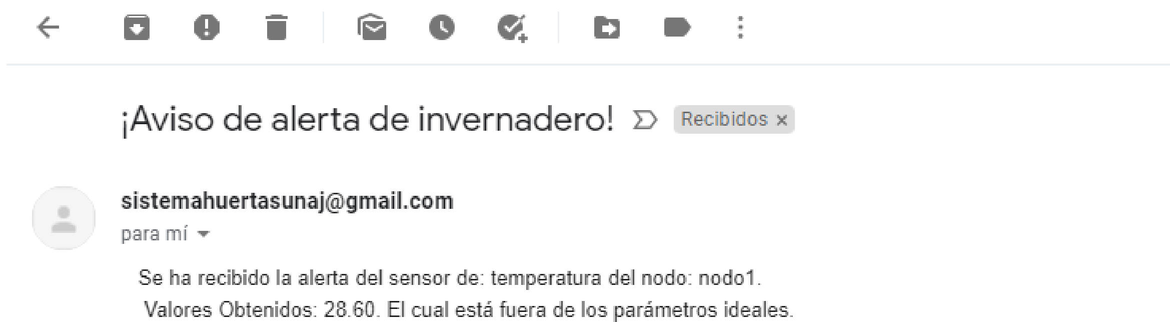

Figura 54 - Email recibido por desvío de parámetros.
Fuente: Elaboración propia.

Prueba del sistema web

Para realizar las pruebas con el sistema web, primero había que ponerlo en producción. Para poner el sistema en producción, se utilizó Heroku y, con la ayuda de GitHub, cada ingreso de modificaciones realizadas en el sistema hacía que el sistema se publicara automáticamente en Heroku, sin tener que hacer el proceso inicial una y otra vez.

Los pasos seguidos fueron los siguientes:

1- Ejecutar "*npm install -g heroku*" en la terminal de Visual Studio Code de cada uno de los proyectos.
2- Crear una cuenta en Heroku para poder subir un proyecto.
3- Ejecutar "*Heroku create*" para crear una aplicación en Heroku, lista para insertar nuestro proyecto.
4- Crear un archivo en nuestro sistema llamado "*Procfile*", el cual contiene el código *"web:npm run start:prod".* Este archivo indica que cada vez que es subido el código de la aplicación a GitHub con Git, tiene que hacer el *deploy* en Heroku.
5- Utilizar "*git push heroku master*", que sube el código de la aplicación a GitHub y construye automáticamente su aplicación en Heroku, permitiéndola usar con un enlace base brindado por la plataforma.

En la "*Figura 55 – Puestas en producción realizados en Heroku para la aplicación Back-End*", se puede observar la fecha y el horario en que fue puesto en producción el sistema y/o también cualquier modificación de este que se haya realizado.

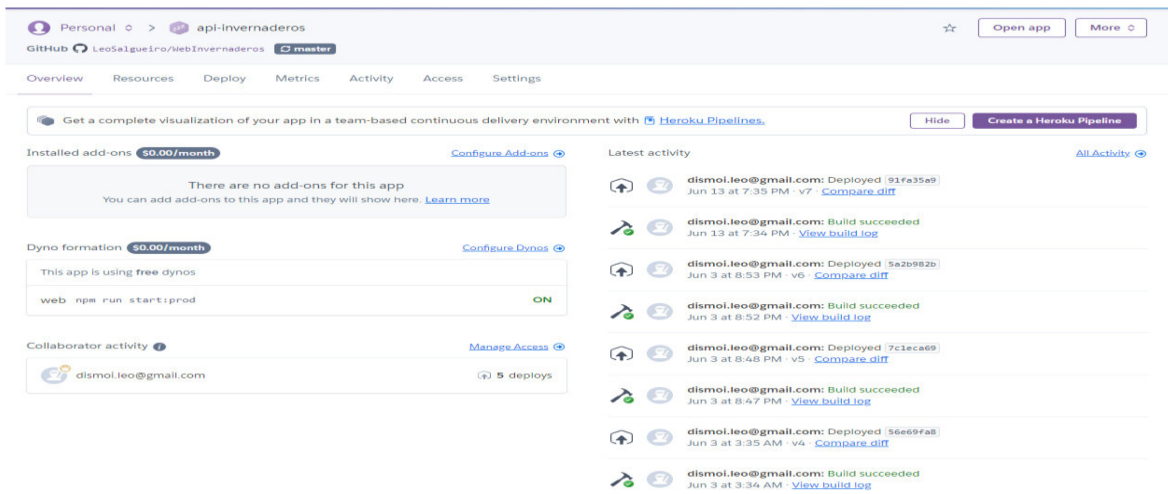

Figura 55 - Puestas en producción realizados en Heroku para la aplicación Back-End.
Fuente: Elaboración propia.

Para la aplicación de Front-End, el proceso de puesta en producción fue exactamente el mismo.

En lo que respecta a las pruebas del sistema web, la parte más importante desarrollada al momento es la sección de "Estado de nodos". Aquí se realizaron las pruebas con la información de la base de datos recolectada por los nodos (ver "*Figura 56 – Sección "Estado de nodos", sensor de temperatura de nodo2*"). Como se puede observar, en la "Figura 56" se muestra un gráfico con la temperatura registrada en una fecha y horario determinados. Además, se indica con línea punteada el parámetro mínimo y el máximo para el tipo de plantación configurada para el nodo. Esto último no se puede apreciar en la imagen ya que el rango máximo son 28°C y, al no registrarse algún valor elevado, no se alcanza a divisar en el gráfico.

Este gráfico permite al usuario realizar un análisis sobre las alarmas detectadas en el sistema en los últimos tiempos y, en base a estos datos, tomar las decisiones

pertinentes en caso de ver irregularidades. La ventaja que posee es que el análisis lo realiza sobre cada sensor que se encuentra en el sistema, de modo que puede analizar las alarmas detectadas por el sistema, de manera individual.

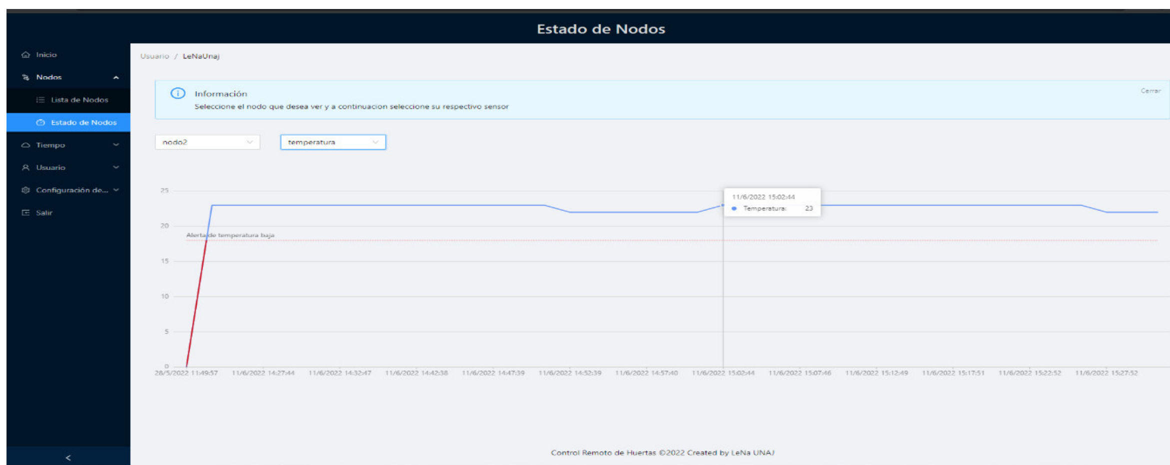

Figura 56 - Sección "Estado de nodos", sensor de temperatura de nodo2.
Fuente: Elaboración propia.

Otras pruebas realizadas sobre el sistema web fueron:

- Prueba de inicio de sesión: el sistema respondió correctamente al inicio de sesión, respetando cada uno de los protocolos de seguridad implementados (JWT). Se comprobó que el sistema *Back-End* devuelve un token para ser utilizado en caso de que el "usuario" y la "contraseña" ingresados sean correctos.
- Prueba de ingreso de un nuevo nodo: se comprobó que la sección de "ingresar un nuevo nodo" funciona correctamente, ya que ingresa información a la base de datos relacionando el nodo con el usuario que lo creó. No obstante, ya se ha aclarado que esta sección funciona para agregar un nuevo nodo, pero no está automatizada con el sistema de microcontroladores para que sea agregada automáticamente al *Suscriber* y este pueda detectar el nuevo nodo ingresado.
- Prueba de listado de nodos en la sección "Lista de Nodos": esta lista correctamente los nodos que el usuario posee en su invernadero.

- Prueba de cambio de la información del usuario desde “Usuarios -> Configuración de usuario”: ingresando en la sección indicada, se puede cambiar la información almacenada del usuario que está logueado en el sistema.

Trabajo futuro

Versión actual

En esta versión del sistema se ha conseguido implementar un sistema de microcontroladores con el protocolo MQTT donde el *Suscriber* está capacitado para recibir toda la configuración necesaria del sistema a través de un archivo de configuración.

Además, el sistema está capacitado para tener almacenados, en la base de datos, los parámetros de configuración de los nodos de manera individual, permitiendo así hacer un control de que la última información leída esté dentro de los parámetros establecidos. Dentro de estos parámetros de configuración almacenados en la base de datos, y, en caso de que se produzcan desvíos, el sistema enviará una alerta al usuario a través de un email. No obstante, para una versión futura, sería ideal que se envíen notificaciones al sistema web cuando ocurren estos acontecimientos, ya que el envío constante de emails puede resultar molesto.

Por otro lado, el sistema web está capacitado para iniciar sesión, realizar análisis de la información histórica, leer la última información ingresada, agregar nuevos nodos (preparados para un desarrollo a futuro en el sistema de microcontroladores), realizar análisis temporales actuales.

Versiones futuras

Como desarrollo a futuro, tanto el sistema web como el sistema de microcontroladores pueden crecer de manera enrome, ya que están creados sobre

arquitecturas altamente escalables y flexibles a cambios. Por ende, los sistemas pueden crecer de manera óptima y con facilidad. Además, muchas partes del sistema fueron pensadas para tener un desarrollo a futuro, ya que el sistema en general no fue solo diseñado para enviar alertas, sino que fue diseñado como un sistema para hacer análisis de datos en el tiempo y que esté preparado para realizar modificaciones automáticas; esto quiere decir que está diseñado para que, por ejemplo, se pueda agregar un nuevo nodo con sus respectivos sensores desde la página web, impactando contra la configuración del sistema de microcontroladores de manera automática, siendo una aplicación completamente escalable y dinámica. Además de poder hacer modificaciones de configuración, también se podrían agregar sistemas de control automáticos o manuales desde el sistema web; por ejemplo, activar el riego de un invernadero de manera manual desde el sistema web, o de manera automática, cuando se detecta un desvío de los parámetros correspondientes a la humedad del suelo. Esto se puede observar mejor en la "*Figura 57 – Diseño de aplicación con middleware intermediario de acciones*" donde se puede ver cómo este diseño no solo funcionaría para la activación de riego, sino para cualquier otra modificación o activación remota que se precisara.

Otro desarrollo a futuro posible e interesante sería la creación de roles en el sistema web, de manera que haya un usuario administrador que se encargue de las configuraciones de los sistemas y un usuario final que solo utilice el sistema para análisis y controles del invernadero.

Otra mejora para el sistema en general sería que el sistema web tuviera configuraciones prestablecidas para configurar sus nodos por tipo de plantación, sin que tenga que ser un usuario administrador quien se encargue de esto. Esto se podría conseguir investigando los parámetros de diferentes tipos de plantación y con ella hacer un repositorio en la base de datos de los tipos de plantación disponible que tiene el sistema.

Como ya se ha mencionado, también sería bueno que las alertas sean enviadas a través de notificaciones al sistema web, y no a través de email, y que estas últimas solo sean implementadas en casos extremos, por parámetros específicos que

indiquen un riesgo elevado del invernadero; por ejemplo, incendios o algún otro acontecimiento excepcional.

Otra mejora sería que el sistema web sea responsivo para poder ser visto en teléfonos celulares. El sistema actualmente es responsivo, pero no a niveles de teléfono celular; por ende, se tendrían que hacer "*mediaqueries*" que modifiquen los componentes, según la resolución de pantalla que se tenga.

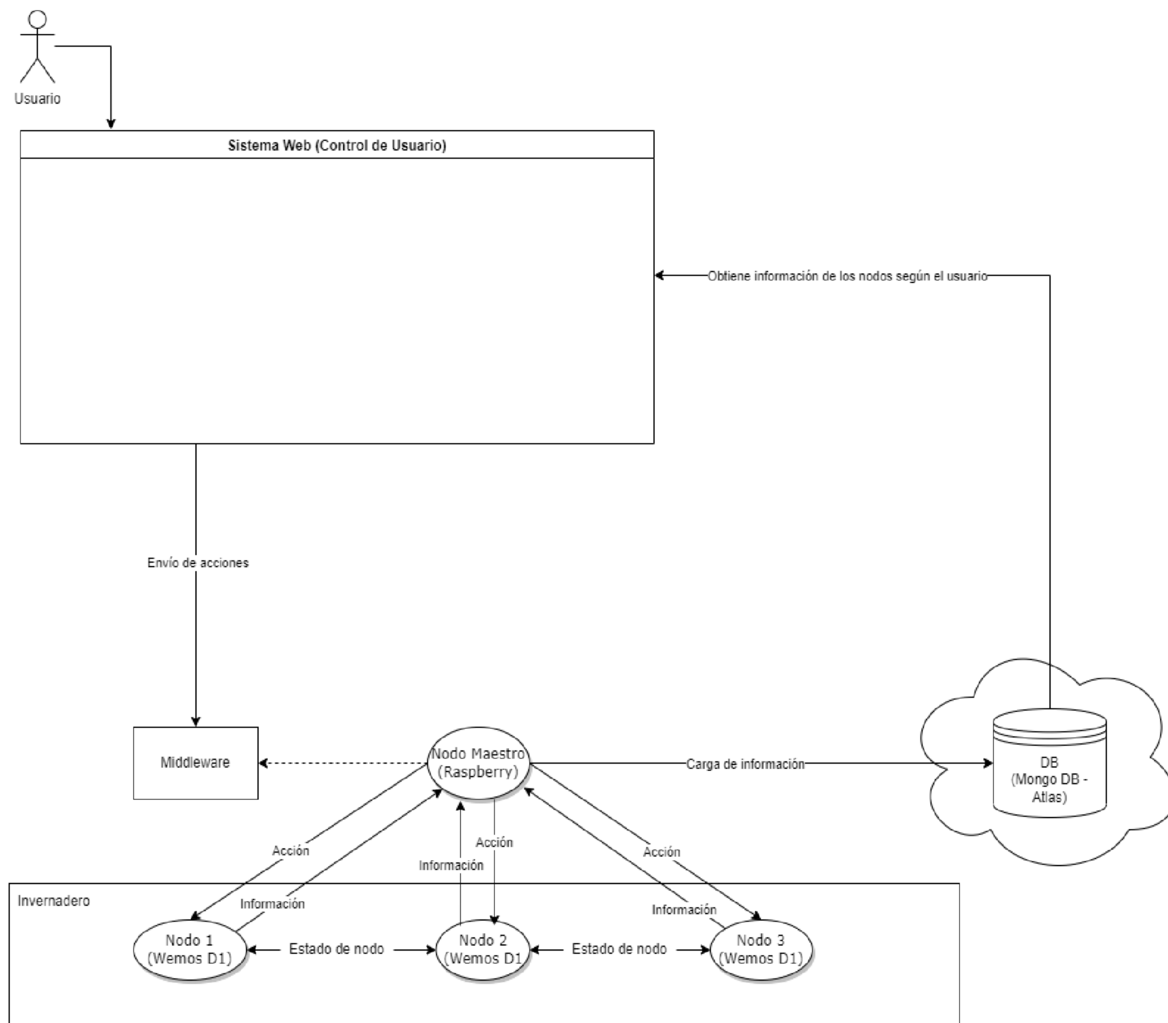

Figura 57 - Diseño de aplicación con middleware intermediario de acciones.
Fuente: Elaboración propia.

Conclusión

En base a todo el desarrollo implementado del sistema en general y las pruebas realizadas se han cumplido los objetivos específicos planteados en la propuesta inicial, excepto el de realización de pruebas en un entorno real ya que los tiempos de entrega han sido limitados para llevar a cabo pruebas de este tipo. Pero, en base a la información recolectada con pruebas locales, el resultado hubiese sido el mismo ya que el proyecto creció de tal manera que la culminación de ese objetivo se resolvía con pruebas pequeñas y ajustes sobre la programación de los sistemas.

Con relación a cada uno de los objetivos se puede decir que:

1. Se consiguió evaluar diferentes tipos de arquitectura y así obtener la más adecuada para el desarrollo del sistema de microcontroladores, llevando a cabo una arquitectura por eventos con el protocolo MQTT.
2. Se analizaron diferentes tipos de microcontroladores y módulos para implementar la arquitectura deseada y se concluyó que el único que pudo satisfacer nuestras necesidades fue el Wemos D1 R1.
3. Se implementaron las conexiones de un sensor de humedad y temperatura ambiente, y otro de humedad de suelo. Además, se los programó para enviar su información y almacenarla en la base de datos.
4. Se evaluaron diferentes tipos de bases de datos y servicios en la nube y se decidió que MongoDB con Mongo Atlas era la mejor elección para el sistema, ya que se manejarían altos volúmenes de información y este servicio era uno de los mejores en cuanto a manejo de altos volúmenes de datos (MongoDB) y de espacios de prueba *free* para desarrollo.
5. Se concluyó que era ideal desestructurar la aplicación web en dos partes y que ambas utilicen como base el lenguaje JavaScript, facilitando la comprensión de ambas aplicaciones del sistema web.

En pocas palabras, se puede decir que se cumplieron todos los objetivos, superando las expectativas del sistema ampliamente y quedando una base muy potente para la creación de un producto comercializable.

Apéndices

Apéndice 1 Instalación de “*Visual studio Code*”:

Para instalar “*Visual Studio Code”* se deben seguir los siguientes pasos:

- Paso 1: ir a la página de “*Microsoft Visual Studio Code” en “Academic Software”* y hacer clic en el botón 'Descargar *Visual Studio Code*‘ para descargar el archivo de instalación.
- Paso 2: abrir el archivo de instalación .exe en la carpeta de descargas para iniciar la instalación.
- Paso 3: en las configuraciones de instalación que aparecen al momento de abrir el archivo de instalación, solo se debe presionar *Next* en todas las opciones que da el instalador para que quede todo configurado por defecto.
- Paso 4: hacer clic en *Install* y listo, ya tenemos instalado “*Visual Studio Code”*.

Índice de Figuras

Figura 1 - Arquitectura general del sistema. 17
Figura 2 - Arquitectura del sistema web. 18
Figura 3 - Arquitectura MQTT del sistema de microcontroladores. 20
Figura 4 - Conexiones de Wemos D1. 24
Figura 5 - Conexiones de sensor DHT11. 24
Figura 6 - Conexiones del sensor HL-69. 25
Figura 7 - Cómo agregar una nueva placa en entorno de desarrollo Arduino. 27
Figura 8 - Cómo instalar una nueva librería en el entorno de desarrollo Arduino. . 27
Figura 9 - Primera parte del código de programación del Publisher. 28
Figura 10 - Segunda parte del código de programación del Publisher. 29
Figura 11 - Porción de código de un Publisher. 34
Figura 12 - Archivo de configuración de Mosquitto. 36
Figura 13 - Archivo de configuración Mosquitto con modificaciones realizadas. ... 37
Figura 14 - Estructura del archivo "usuario_config.py". 39
Figura 15 - Código de implementación del Suscriber Versión 1, primera parte. 40
Figura 16 - Código de implementación del Suscriber Versión 1, segunda parte. .. 41
Figura 17 - Visual Studio Code, terminal y entorno donde está alojado el sistema Back-End. 45
Figura 18 - Estructura básica de una aplicación NestJS. 47
Figura 19 - Enlace de conexión a repositorio remoto. 48
Figura 20 - Estructura de sistema Back-End completa v1.0. 52
Figura 21 - Funcionamiento de un guardián. 53
Figura 22 - Estructura de "app.controller.ts". 54
Figura 23 - Composición del archivo "auth.service.ts". 56
Figura 24 - Composición de "local.strategy.ts". 57
Figura 25 - Proceso de login a través de un guardián intermedio. 57
Figura 26 - "schema" del controlador Usuarios ("usuarios.schema.ts"). 59
Figura 27 - Composición de "usuarios.controller.ts". 61
Figura 28 - Composición de "usuarios.module.ts". 63
Figura 29 - Composición de "usuarios.service.ts". 64
Figura 30 - Composición del archivo "nodos.controller.ts". 67
Figura 31 - Composición del archivo "nodos.module.ts". 68
Figura 32 - Esquemas de "nodos.schema.ts" y "configNodo.schema.ts". 69
Figura 33 - Composición del archivo "nodos.service.ts". 71
Figura 34 - Estructura del archivo "auth.module.ts". 73
Figura 35 - Composición del archivo "auth.service.ts". 74
Figura 36 - Composición de "local.strategy.ts". 75
Figura 37 - Composición de "jwt.strategy.ts". 76
Figura 38 - Estructura básica de aplicación React. 78
Figura 39 - Composición del sistema Front-End actualmente. 79
Figura 40 - Cambio de estados en Redux. 82
Figura 41 - Comparación de aplicación React sin Redux y con Redux. 83
Figura 42 - Pantalla de inicio de sesión del Front-End. 87

Figura 43 - Menú de la aplicación Front-End. 88
Figura 44 - Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación Python versión 3.6 o superior. 90
Figura 45 - Ejemplo de cadena de conexión devuelta por Mongo Atlas para una aplicación NodeJS 4.1 o superior. 91
Figura 46 - Función get_database() correspondiente al código Python contenido en database_connection.py. 93
Figura 47 - Función update_nodo(informacion) correspondiente al código Python contenido en database_connection.py. 94
Figura 48 - Función insert_information_nodo(informacion) parte uno. 95
Figura 49 - Función insert_information_nodo(informacion) parte dos. 96
Figura 50 - Método insert_configuracion() parte 1. 97
Figura 51 - Método insert_configuracion() parte 2. 97
Figura 52 - Método comprobarAlerta(informacion). 98
Figura 53 - Método enviarMail(informacion). 99
Figura 54 - Email recibido por desvío de parámetros. 101
Figura 55 - Puestas en producción realizados en Heroku para la aplicación Back-End. 102
Figura 56 - Sección "Estado de nodos", sensor de temperatura de nodo2. 103
Figura 57 - Diseño de aplicación con middleware intermediario de acciones. 106

Bibliografía

Documentación y recursos NestJS. Recuperado de https://docs.nestjs.com/. Consulta: 07/04/2022.

Documentación y recursos NodeJS. Recuperado de https://nodejs.org/es/. Consulta: 04/03/2022.

Documentación y recursos Heroku. Recuperado de https://www.heroku.com/. Consulta: 05/05/2022.

Documentación y recursos ReactJS. Recuperado de https://es.reactjs.org/. Consulta: 30/04/2022.

Documentación y recursos Ant-Design. Recuperado de https://ant.design/. Consulta: 30/04/2022.

Utilización de recursos Draw.Io. Recuperado de https://app.diagrams.net/. Consulta: 15/03/2022.

Utilización de la aplicación y documentación. Recuperado de https://www.postman.com/. Consulta: 07/04/2022.

Utilización de documentación. Recuperado de https://www.arduino.cc/. Consulta: 05/03/2022.

Investigación. Recuperado de https://sembralia.com/blogs/blog/tomate-en-invernadero. 10/03/2022.