ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

DESARROLLO DE UN SISTEMA WEB RECOLECTOR DE FACTORES AMBIENTALES PARA UNA FINCA FLORÍCOLA UTILIZANDO TECNOLOGÍA IOT

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y REDES DE INFORMACIÓN

JUAN FRANCISCO TORRES MERA

DIRECTOR: ING. PABLO WILIAN HIDALGO LASCANO, MSc.

Quito, febrero 2023

AVAL

Certifico que el presente trabajo fue desarrollado por Juan Francisco Torres Mera, bajo mi supervisión.

ING. PABLO WILIAN HIDALGO LASCANO, MSc.
DIRECTOR DEL TRABAJO DE TITULACIÓN

DECLARACIÓN DE AUTORÍA

Yo, Juan Francisco Torres Mera, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración dejo constancia de que la Escuela Politécnica Nacional podrá hacer uso del presente trabajo según los términos estipulados en la Ley, Reglamentos y Normas vigentes.

_____

JUAN FRANCISCO TORRES MERA

DEDICATORIA

Para Oswaldo y Marcelo, con mucho amor.

AGRADECIMIENTO

A toda mi familia, por su inconmensurable apoyo y paciencia.

A mi madre y hermana, por brindarme la motivación, el ejemplo y la confianza que necesitaba para finalizar mis estudios universitarios.

A mis tías, cada una apoyándome a su manera única.

A mis abuelos, por haber luchado para que yo pueda tener las oportunidades que tengo.

A todos los profesores, tutores, y personal de la EPN que me vieron en mis momentos más difíciles y decidieron ayudarme de una manera u otra. Especialmente a Ricardo Mena, Soraya Sinche y Pablo Hidalgo.

ÍNDICE DE CONTENIDO

RESUMEN

En este Trabajo de Titulación se desarrolla un sistema capaz de recolectar y almacenar mediciones de algunos factores ambientales de un cultivo, por ejemplo, temperatura y humedad para el suelo y el aire. Con este propósito se utilizan dos tecnologías integradoras sumamente presentes en la Industria 4.0, la Web y el Internet de las Cosas (IoT).

Esta recolección de datos ambientales no solo permite automatizar procesos de monitoreo ambiental sino que también consolida y centraliza la información; de esta manera el usuario tiene a disposición registros anteriores y actuales en cualquier momento y lugar, siempre y cuando posea acceso a Internet y un navegador Web.

El almacenamiento de registros ambientales implica una generación de datos, los cuales hoy en día son uno de los recursos más valiosos que puede tener una empresa; para este caso específico los datos ambientales pueden ser utilizados en conjunto con otros tipos de datos generados por fincas para la predicción y control de plagas, aumentando la rentabilidad de los cultivos.

El proyecto se encuentra organizado de la siguiente forma: en el Capítulo 1 se presenta el marco teórico necesario para desarrollar el sistema, en el Capítulo 2 se encuentra tanto el diseño como la implementación del sistema siguiendo la metodología ágil Kanban. El Capítulo 3 agrupa las pruebas de funcionamiento tanto para los componentes del sistema como para la totalidad del sistema, así como el análisis de la solución resultante. Finalmente, las conclusiones y recomendaciones obtenidas del proyecto se detallan en el Capítulo 4.

También se anexan algunos documentos cuyo tamaño no es adecuado para el documento como los tableros Kanban y las fotos de los escenarios de prueba, así como los repositorios de código de las diferentes partes del sistema.

PALABRAS CLAVE: Sistema Web Recolector, Factores Ambientales, MQTT, IoT, Floricultura.

ABSTRACT

This Undergrad Project presents, a system capable of collecting and storing measurements of environmental factors (like temperature, humidity or soil moisture) on agriculture crops. For this purpose, two comprehensive technologies that are highly present in the industry 4.0 are used: the Web and the Internet of Things or IoT.

This environmental data gathering not only allows to automate environmental monitoring processes but also consolidates and centralizes the generated information, therefore the user is able to check past and current records at any time and place as long as they have access to the Internet and a Web browser.

The storage of environmental records involves a generation of data, which nowadays is one of the most valuable resources that a company can have; For this specific case, environmental data can be used in conjunction with other types of data generated by farms for pests predicting and controlling, increasing the profitability of crops.

The project is organized as follows: Chapter 1 presents the theoretical background necessary for developing the system, Chapter 2 contains both the design and implementation of the system following the agile Kanban methodology. Chapter 3 groups the functional tests for both the system components and the system as a whole solution, it also groups the analysis of the final product. Finally, conclusions and recommendations obtained from the project are detailed in Chapter 4.

In addition, some documents like Kanban boards and photos of the testing environments, as well as the system code repositories, are included as Annexes.

KEYWORDS: Data Gathering System, Environmental Variables, MQTT, IoT, Floriculture.

1 INTRODUCCIÓN

En este capítulo se presentan los objetivos y el alcance del proyecto, así como una recopilación de algunos fundamentos teóricos necesarios para el desarrollo de éste. La información consultada se divide en cuatro subsecciones, en las que se agrupan conceptos de: Agricultura, Software, Web e IoT.

1.1 OBJETIVOS

Este Proyecto Técnico considera los siguientes objetivos:

Objetivo General: Desarrollar un sistema Web recolector de factores ambientales para una finca florícola utilizando tecnología IoT.

Objetivos Específicos:

- Analizar la teoría necesaria para el desarrollo del sistema.
- Diseñar los diferentes componentes del sistema.
- Implementar los diferentes componentes diseñados.
- Analizar los resultados de las pruebas de funcionamiento del sistema

1.2 ALCANCE

El sistema es dividido en componentes los cuales son diseñados y posteriormente implementados. Hay un componente de dispositivos embebidos, cuya función es adquirir datos de mediciones de factores ambientales (humedad y temperatura) utilizando sensores comerciales; el dispositivo embebido (cliente MQTT) también será capaz de conectarse a Internet y utilizar el protocolo MQTT (*Message Queuing Telemetry Transport*) para publicar los datos adquiridos. Las publicaciones serán dirigidas hacia un *broker* MQTT, el cual es montado y configurado; éste tiene como función la redirección de los mensajes hacia los subscriptores.

Existe un componente recolector de datos que también es un cliente MQTT que se subscribe a los embebidos publicadores y recolecta los factores ambientales, para posteriormente insertarlos en una base de datos. Adicionalmente se implementa una base

de datos relacional para la persistencia de éstos en el sistema, la cual utiliza un motor SQL Server.

A continuación, se tiene un componente de *Back-End* de la app Web, que brinda un servicio Web de tipo API REST, el cual es codificado utilizando el *framework* de Microsoft .Net Core. Éste provee una interfaz programable para que el *Front-End* pueda consultar los datos en el servidor.

El último componente es la interfaz gráfica para el usuario, la cual es una aplicación Web de *Front-End* elaborada con el *framework* Angular, con HTML (*Hyper Text Markup Language*), TypeScript y CSS (*Cascading Style Sheets*). Así, el usuario podrá visualizar los datos consumidos de *Back-End* utilizando tablas y gráficos.

Para obtener los requerimientos funcionales y no funcionales del sistema, se realizan tres entrevistas al personal técnico de una florícola. Una vez que el sistema se implementa, se realizan pruebas del funcionamiento utilizando al personal técnico de la florícola como usuario final. Por último, el proyecto planteado es desarrollado en base a la metodología ágil Kanban.

En la Figura 1.1 se presenta el diagrama de estructura del sistema desarrollado.

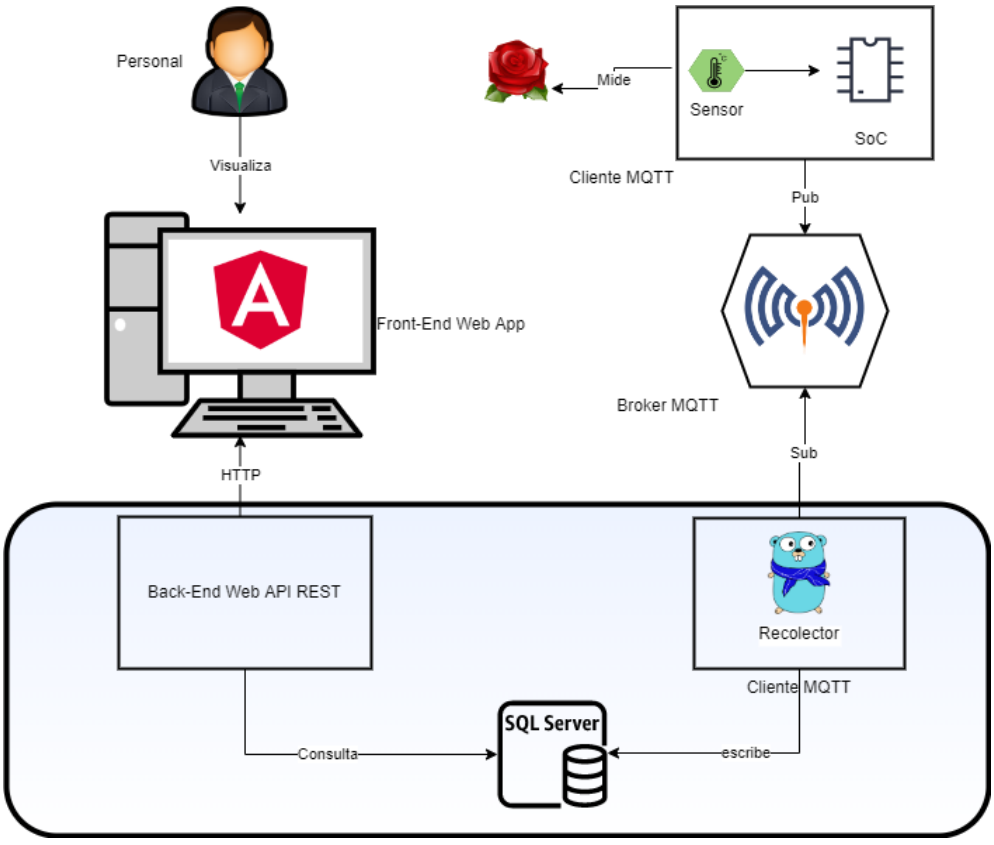


Figura 1.1 Esquema General del Prototipo

1.3 MARCO TEÓRICO

1.3.1 AGRICULTURA Y MEDIO AMBIENTE

Se define a la Agricultura como un conjunto de prácticas enfocadas en el desarrollo y recolección de productos en un suelo, así como la cría y mantenimiento de ganado. [1]

1.3.1.1 Floricultura

Un caso particular de la Agricultura es la Floricultura, la cual se limita al cultivo de flores y otras plantas ornamentales con fines decorativos. Por lo general, estos cultivos son muy susceptibles a las condiciones ambientales por lo que requieren de cuidados especiales; algunos autores sugieren que: “La floricultura es un tipo de producción que conlleva un uso intensivo de la superficie y de la mano de obra. La tecnología de cultivo y el mejoramiento de las especies ornamentales han estado enfocados en una producción de uso eficiente de la superficie.” [2] En otras palabras, los agricultores utilizan estructuras conocidas como invernaderos para optimizar el uso del suelo y controlar las condiciones ambientales dentro de éstos.

1.3.1.2 Clima y Ambiente

Dentro del contexto de la Floricultura, el desarrollo de una planta está relacionado con las condiciones ambientales del lugar en clima, agua y suelo. Estos factores pueden ser fenómenos físicos o químicos, generalmente medibles con instrumentos especializados. Entre los principales están: temperatura, humedad, cantidad de luz, acidez, velocidad del viento, inclinación del terreno, NPK (Nitrógeno, Fósforo y Potasio), electroconductividad, etc. [3]

Cultivar implica tener cierto control sobre estas variables utilizando: los invernaderos previamente mencionados, el riego o abonado del suelo, entre otros. De esta manera, los agricultores pueden influir en el desarrollo del cultivo y la calidad del producto final.

1.3.1.3 Agricultura Inteligente

La Agricultura inteligente es una idea basada en proveer, a las prácticas tradicionales, una infraestructura tecnológica, involucrando así otras prácticas propias de la Industria 4.0, destacando: Computación en la nube, *Big Data*, IA (Inteligencia Artificial) e IoT. [4] Si también se considera el clima y el cambio climático se tiene como resultado el concepto de “*Climate-Smart Agriculture*” o CSA.

Los tres objetivos principales del CSA [5] son:

- Incrementar la productividad de los cultivos priorizando la sostenibilidad.
- Mitigar los efectos del cambio climático en los sembríos.
- Apoyar la preservación del medio ambiente minimizando la emisión de dióxido de carbono y contaminación.

1.3.2 CIENCIAS DE LA COMPUTACIÓN Y SOFTWARE

En esta sección se revisarán algunos conceptos fundamentales para el desarrollo de software. Éste se define como un conjunto de programas y datos utilizado para dar instrucciones a las computadoras y controlar el hardware (su contraparte física) [6], y así proveer soluciones.

Se entiende por Programación a la creación de un grupo de instrucciones que toman una entrada de datos o instrucciones y entrega una salida o realiza acciones; de esta manera un usuario puede utilizar un ordenador para resolver un problema específico.

1.3.2.1 Paradigmas de Programación

Son formas en las que se puede concebir una solución a un determinado problema escribiendo código; cada paradigma ofrecerá un enfoque dependiendo de los requerimientos, niveles de abstracción, herramientas disponibles, limitaciones del hardware, etc. [7]

Por ejemplo, se habla de paradigmas imperativo versus declarativo; el primero dice que el programa le indica a la computadora el qué debe hacer y cómo hacerlo, en cambio el segundo solo declara las propiedades de los resultados abstrayendo la manera de obtener la respuesta.

Se habla también de un paradigma estructurado, donde se codifican las instrucciones línea por línea y consecuentemente se muestra el flujo de control de manera explícita. Por otra parte se tiene una programación funcional, ésta se basa en funciones y composición de funciones para declarar el comportamiento del programa.

Paradigma Orientado a Objetos

También conocido como Programación Orientada a Objetos (POO), es un paradigma que modela a una solución como un conjunto de objetos que interactúan entre sí. Por Objeto se refiere a una abstracción de un elemento real o un concepto, esta abstracción cuenta

con propiedades y métodos para poder definir tanto características como acciones del Objeto. [8]

La POO se basa en cuatro principios:

- La abstracción oculta código complejo dentro de métodos.
- El encapsulamiento aísla el objeto y controla la información que éste puede dar.
- La herencia permite estructurar objetos jerárquicamente, heredando elementos de padres a hijos.
- El polimorfismo permite a objetos hijos tener diferentes “formas” al modificar comportamientos heredados.

En POO también se tienen *clases*, que son plantillas de donde se instancian los objetos, e *interfaces*, siendo abstracciones que extienden a los objetos con una serie de métodos que deberán implementar.

1.3.2.2 Lenguajes de Programación

Resulta complicado dar instrucciones a las computadoras ya que éstas son dispositivos digitales basados en sistemas binarios, consecuentemente solo entienden un lenguaje conocido como “de máquina”, el cual depende de la arquitectura del computador. Para simplificar la tarea se utilizan abstracciones conocidas como lenguajes de programación [9], los cuales tienen sentido para los humanos, pero no son ejecutables por un ordenador hasta que son “compilados” “transpilados” o “interpretados”.

Estos lenguajes implementan uno o varios paradigmas de programación; pueden ser de tipado: estático o dinámico, y fuerte o débil. Existe una gran cantidad de lenguajes, cada cual tiene fortalezas y debilidades al momento de resolver un problema, por lo que un desarrollador debería escoger el más adecuado según la aplicación. La figura 1.2 muestra algunos de lenguajes los más populares según Stack Overflow para el 2022.

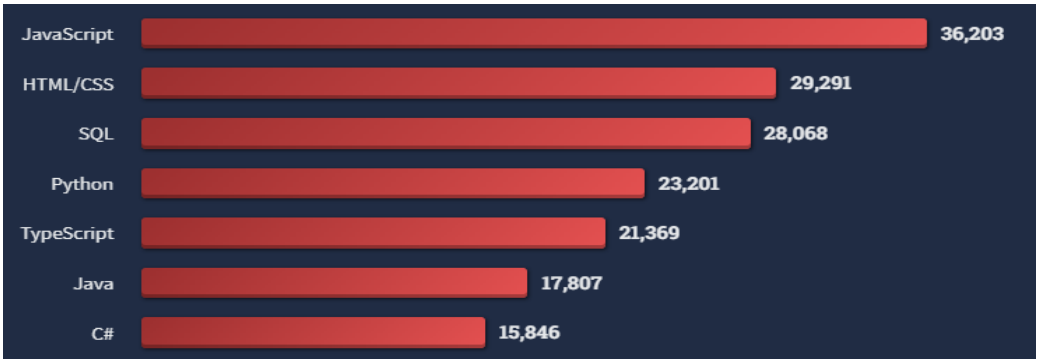


Figura 1.2 Algunos lenguajes entre los más populares según S.O. Survey 2022 [10]

Al desarrollar un programa se presentan problemas comunes; para facilitar soluciones existen recursos como librerías y *frameworks* o marcos de trabajo que aceleran el desarrollo al proveer herramientas útiles para dichos problemas.

1.3.2.3 Estructuras de Datos

Un dato es un “pedazo” de información que representa algún valor del mundo real. Una computadora manipula datos de manera binaria por lo que es necesario llevar un formato o estructura que le dé sentido y permita a la persona modelar información. Las estructuras de datos indican la forma en que se guarda un conjunto de datos en memoria [11]; cada formato presentará ventajas y desventajas cuando estos conjuntos deban ser manipulados.

Arreglo

En esta estructura, los datos son almacenados en memoria uno a continuación de otro. Cada celda de información tiene un índice numérico comenzando desde el cero hasta (n-1) siendo “n” el total de elementos del arreglo.

Pilas y Colas

Son estructuras enfocadas en la entrada y salida de datos, mientras que las pilas siguen el principio LIFO (*Last In First Out*) donde el último dato ingresado es el primero en ser extraído, las colas siguen el principio FIFO (*First In First Out*) donde el primer dato que entra será el primero en salir.

Una aplicación de la pila es el *stack* de direcciones en un computador, que controla el flujo del programa. Una aplicación común de la cola es un *buffer* que recibe datos a ser procesados; por lo general la tasa de entrada y salida de éstos difiere entre sí.

Árbol

Es una estructura representada por un grafo que tiene un nodo raíz o padre, y uno o varios hijos que a su vez pueden tener otros nodos hijos. Son sumamente útiles para representar relaciones jerárquicas.

Tabla de Hash

En esta estructura los datos son agrupados en asociaciones de clave y valor mediante una función conocida como “de hash”; ésta toma la clave como entrada y la procesa obteniendo como salida la dirección donde se almacena el valor. Cabe mencionar que cada vez que se ingrese la misma entrada se obtendrá la misma salida, y que una función de *hash* efectiva prevendrá colisiones (entradas diferentes generando una sola salida).

1.3.2.4 Bases de Datos

Una base de datos es una colección organizada de datos que se almacena en un computador y puede ser accedida, modificada o eliminada mediante un programa llamado Sistema de Administración de Bases de Datos o “DBMS”. [12]

Existen algunos tipos de bases de datos: clave-valor, de documentos, de grafos, de motores de búsqueda, entre otras. Pero la más utilizada es la base de datos relacional.

Bases de Datos Relacionales

Un modelo relacional indica que los datos serán representados como entradas o filas en una tabla cuyas columnas representan un atributo de la entidad modelada. Cada tabla tiene una columna que identifica cada registro y es conocida como clave primaria (*Primary Key* o PK). Las tablas pueden relacionarse entre sí enlazando una PK con un atributo llamado clave foránea (*Foreign Key* o FK) mediante referenciación. [13]

Este tipo de base de datos requiere de un modelamiento previo a su implementación, pues posteriormente a este paso resulta muy difícil modificar la estructura y relaciones de tablas creadas. Por esta situación, se establece el concepto de normalización, una técnica que optimiza el modelo relacional al reducir su redundancia y aumentar la integridad de los datos.

ORM

La Asignación Objeto-relacional también conocida como ORM (*Object-Relational Mapping*) es una técnica utilizada por varios lenguajes de POO para adaptar tablas y relaciones en un formato que el lenguaje pueda entender, generalmente mediante Objetos. [14]

Comúnmente el término ORM es utilizado para referirse a la librería o *framework* que implementa la técnica dentro del programa que requiera comunicarse con una base de datos. Un ejemplo es GORM, una librería del lenguaje Go que provee funcionalidades de ORM a desarrolladores de manera amigable. [15]

1.3.2.5 Sistemas Operativos

Un Sistema Operativo (OS) es un programa informático que interactúa con el hardware de una computadora y provee una plataforma uniforme al resto del software, aplicaciones y servicios. [16]

Básicamente un OS consta de un núcleo llamado Kernel, Aplicaciones y servicios dedicados y una capa externa conocida como “Shell”. El Kernel es la parte más importante del sistema, se encarga de la comunicación con el *firmware* de cada componente del ordenador, abstrayendo el control de éstos al resto de partes del OS.

Las aplicaciones y servicios son programas que se encargan de tareas propias del OS como manejo de la red, interfaz gráfica, sistema de archivos, entre otras. El Shell es un programa que brinda una interfaz de consola al usuario para el control del sistema operativo.

Existen varios tipos de sistemas operativos, cada uno de ellos cuenta con características específicas para una aplicación, entre ellos se tienen:

- De propósito general, enfocados en la interacción con el usuario.
- Para Servidores, que controlan hardware dedicado a proveer servicios en la red.
- Móviles, utilizados en dispositivos como *smartphones* con batería limitada.
- De tiempo real, conocidos como RTOS siendo caracterizados por su capacidad de asegurar el cumplimiento de una tarea (crítica) en un momento específico. [17]

1.3.2.6 Patrones de Diseño de Software

Son formas de solucionar problemas comunes en el desarrollo de software [18], éstos fueron definidos entre los años 70 y 90 a raíz del crecimiento de proyectos de software y de los problemas de mantenibilidad que éstos representaban.

Cada patrón presenta beneficios y desventajas, no entrega una solución específica sino la idea de cómo implementarla en el caso de uso, pudiendo añadir complejidad innecesaria al código. Se observan tres tipos principales:

- Creacionales, utilizados para la instanciación de objetos en POO.
- Estructurales, que describen la estructura de entidades y sus relaciones.
- Comportamentales, que definen cómo pueden comunicarse las entidades entre sí.

Inyección de Dependencia

Es un patrón creacional que indica que en una relación de dependencia entre clases, un objeto no debe ser instanciado sino pasado como parámetro, obteniendo de esta manera parejas más separadas y explícitas [19], haciendo posible el principio de inversión de control donde la dependencia se realiza a través de abstracciones como interfaces.

Patrón Decorador

Es un patrón estructural que añade atributos y funcionalidades a un objeto sin la necesidad de utilizar herencia; de esta manera se evitan estructuras jerárquicas extensas e insostenibles, ya que se añaden elementos al objeto como capas de código. Esto se hace mediante una abstracción que es implementada por un objeto concreto y un objeto decorador, el cual de manera recursiva puede recibir un objeto concreto y devolver un nuevo objeto extendido.

Patrón Observador

Es un patrón comportamental que permite propagar cambios de estado desde un objeto llamado observable hacia diferentes objetos dependientes llamados observadores. Cada vez que un objeto necesita observar a un observable se suscribe, lo que significa que este es añadido a una lista que da seguimiento a los observadores para poder notificar el cambio de estado y consecuentemente, que todo observador consulte el nuevo estado del observable.

1.3.2.7 Arquitecturas de Software

La arquitectura de software representa la manera de cómo se estructura un programa de manera abstracta [20], es decir, se observa a la solución de software como un sistema dividido en componentes y relaciones; cada componente es un pedazo de código que soluciona un problema específico e interactúa con el resto de los componentes para resolver el problema general.

Cabe mencionar que una solución de código puede seguir varias arquitecturas que brinden diferentes abstracciones a diferentes aspectos del sistema, también que cada componente

puede ser visto como un subsistema formado por sus propios componentes y relaciones según los requisitos funcionales y no funcionales del software en desarrollo.

Arquitectura de Capas

Esta arquitectura separa las responsabilidades de la solución de software en un número de capas apiladas; la idea es que cada capa se encargue únicamente de las funcionalidades relacionadas con dicha capa, y que cada capa solo conozca a sus adyacentes, generalmente dando un flujo de control vertical.

La más común es la arquitectura de tres capas; ésta separa responsabilidades en:

- Datos, la capa inferior encargada del modelamiento y almacenamiento de los datos.
- Negocio, la capa intermedia que contiene toda la lógica del dominio.
- Presentación, la capa más cercana al usuario que cumple funciones de interfaz.

Modelo Vista Controlador

También conocida como MVC (*Model View Controller*) [21] es una arquitectura frecuentemente utilizada en web que desacopla un servicio remoto en tres componentes:

- El Modelo agrupa la lógica de manipulación de datos, generalmente se comunica con la base de datos para funciones de CRUD (Crear, Leer, Modificar y Borrar).
- La Vista se encarga de colocar los datos del Modelo en “vistas” o plantillas que dan contexto y formato interpretable por el cliente.
- El Controlador actúa como punto de entrada recibiendo peticiones para seguido interactuar con el Modelo y la Vista y consecuentemente generar y devolver la información de retorno.

Arquitectura Cliente Servidor

Es una arquitectura distribuida que separa responsabilidades en dos componentes, un Cliente y un Servidor. El Cliente se caracteriza por iniciar la comunicación del sistema distribuido mediante peticiones, una vez recibida una respuesta el Cliente puede procesar el mensaje independientemente del Servidor.

El Servidor se encuentra siempre escuchando peticiones, cuando llega una la procesa y genera un mensaje de respuesta que comúnmente contiene la información solicitada o información del por qué no se pudo completar la solicitud.

1.3.2.8 Metodologías Ágiles

Una metodología ágil es un grupo de prácticas utilizadas al desarrollar software [22], éstas contrastan con las prácticas de metodologías tradicionales enfocadas en manejar una documentación comprensiva y extensa, y con una estructura rígida.

Las metodologías ágiles recomiendan [23]:

- Individuos e interacciones sobre procesos y herramientas.
- Software en funcionamiento sobre documentación completa.
- Colaboración con el cliente sobre negociación de contratos.
- Reacciona al cambio sobre cumplimiento de planes.

Kanban

Es una metodología ágil que fundamenta el flujo de trabajo en una pizarra conocida como *Kanban board*. [24] Esta pizarra representará el estado de un proyecto y estará disponible a todos sus miembros colaboradores.

La tabla es estructurada en columnas que modelan fases de realización, desde el inicio (columna extrema izquierda) hasta el fin (columna extrema derecha). Cada columna es llenada por tarjetas que representan pedazos del proyecto conocidas como *"features"* o funcionalidades. Kanban es un sistema de *"pull"* que "jala" funcionalidades de izquierda a derecha según la disponibilidad de los desarrolladores (también conocida como WIP o *Work In Progress*). La idea es que cada tarjeta vaya recorriendo el tablero de izquierda a derecha hasta que todo el proyecto sea completado.

Jira Software

Jira es una plataforma en línea que provee herramientas para el manejo, administración y flujo de trabajo en proyectos de desarrollo de software que utilicen metodologías ágiles, especialmente Scrum y Kanban [25].

Un proyecto en Jira cuenta con la característica flexibilidad de las metodologías ágiles; sin embargo, la plataforma realiza algunas sugerencias como utilizar "historias de usuario" como base para las tarjetas Kanban (llamadas "*issues*" por Jira) [26].

1.3.3 TECNOLOGÍAS WEB

En la actualidad, las tecnologías Web se encuentran presentes en todas partes: Redes sociales, *E-Commerce*, *Retail*, *Blogs*, entretenimiento, *Streaming*, *E-Learning*, etc. Se atribuye esta fuerte adopción a su naturaleza abierta y gratuita, e indiscutiblemente, esta tecnología ha revolucionado el mundo y continúa evolucionando sin dar señal de detenerse. [27]

1.3.3.1 World Wide Web

La WWW también conocida simplemente como “Web” fue inventada por Sir Tim Berners-Lee en 1989 [28] y su definición ha cambiado a través de los años. Si bien comenzó como un conjunto de recursos de “hipertexto” con la capacidad de redirigirse entre sí, ahora se define más adecuadamente como una gran plataforma de software que “corre” sobre Internet.

Varios autores reconocen a la primera fase como Web1, caracterizada principalmente por su contenido estático (generalmente solo lectura) y utilizada en su mayoría por *Blogs* y *Wikis* para compartir información.

La evolución de la Web1 es la Web2 [29], fuertemente marcada por el auge de las redes sociales y su necesidad de contenido dinámico e interacción con el usuario, así como el desarrollo de dispositivos móviles capaces de acceder a la Web y consecuentemente el concepto de “*Responsive Design*”, cuyo objetivo es la responsividad del sitio a diferentes tamaños de pantalla.

Actualmente se habla de una Web3 [30] y Web5 [31] donde la idea principal es la descentralización y el enfoque hacia el individuo (cartera individual con dinero electrónico); sin embargo, ninguno de estos conceptos ha sido adoptado todavía pues las tecnologías descentralizadas como: *blockchain*, crypto o NFT (*Non-fungible token*) aún presentan problemas propios. [32]

1.3.3.2 Internet

Es importante diferenciar entre Web e Internet, la primera es la plataforma de software que llega al usuario final, generalmente a través de un programa llamado “navegador”, en tanto que el segundo es la infraestructura tecnológica que permite una conexión de escala global de computadores (contando servidores, teléfonos inteligentes y todo dispositivo con la capacidad de conectarse a Internet).

Debido a la alta complejidad que presenta enviar y recibir información entre computadores, se divide la tarea en funciones más pequeñas que cumplen un objetivo específico y solo en conjunto resuelven el problema por completo. El modelo OSI (*Open Systems Interconnection*) [33] plantea una división de funciones por medio de siete capas: Física, de Enlace de Datos, de Red, de Transporte, de Sesión, de Presentación, y de Aplicación. Mientras que las primeras capas se encargan de funciones más cercanas al Hardware y canal de comunicación, las últimas se encargan de funciones cercanas al usuario y a la aplicación. Cada capa puede comunicarse únicamente con su adyacente y solamente describe la forma de manejar una función mas no el cómo debe ser implementada.

TCP/IP implementa este modelo por capas (aunque combina algunas de éstas) y ofrece un conjunto de protocolos para construir el Internet. [34] Su nombre es una combinación de los protocolos más importantes de toda la pila: *Transmission Control Protocol* e *Internet Protocol*.

La primera capa conocida como “de acceso al medio” se implementa a través de protocolos como Ethernet o Wi-Fi (802.11) y como su nombre lo indica, éstos son definidos por el medio por lo que considera~~n~~ aspectos como características eléctricas, modulación, velocidad de transmisión, codificación, CRC (*Cyclic Redundancy Checking*), entre otras. Estos protocolos son manejados generalmente a nivel de hardware, consecuentemente, utilizan una dirección física o MAC (*Media Access Control*) para identificar un equipo en la red.

En la segunda capa o “de Internet” se pueden diferenciar dos tipos de protocolos según su uso: aquellos utilizados para definir, mantener y modificar rutas; y aquellos utilizados para enviar información a través de dichas rutas. El protocolo de Internet o IP es por un gran margen el más utilizado para enlazar equipos. Esto lo hace empleando rutas definidas y direcciones lógicas (*IP address*) para enviar información en forma de “paquetes IP”. En la actualidad se disponen de dos versiones que “conviven” de este protocolo: IPv4 e IPv6.

Transmission Control Protocol (TCP) y *User Datagram Protocol* (UDP) representan dos de los protocolos de la tercera capa también conocida como “de Transporte”. Ambos tienen como tarea principal establecer canales “extremo a extremo” para la comunicación entre aplicaciones por medio de “puertos de red”. Nótese que UDP es un protocolo no orientado a conexión, mientras que TCP es orientado a conexión, por lo cual utiliza mecanismos como acuse de recibo (ACK), saludo de tres vías, reenvío de paquetes, control de flujo, entre otros, los cuales son necesarios para asegurar la conexión.

La cuarta y última capa (de Aplicación) dispone de una gran variedad de protocolos, cada uno destinado a cumplir los requerimientos específicos de una determinada aplicación. Algunos ejemplos son: *Domain Name Service* (DNS), *Dynamic Host Configuration Protocol* (DHCP), *Simple Mail Transfer Protocol* (SMTP). Dentro de esta capa se encuentran los protocolos HTTP (*HyperText Transfer Protocol*) y MQTT (*Message Queuing Telemetry Transport*), que serán descritos más a profundidad posteriormente.

1.3.3.3 Hypertext Transfer Protocol

También conocido por sus siglas HTTP, es un protocolo de aplicación concebido para la transferencia de texto enriquecido con enlaces externos o “hipertexto”. Actualmente HTTP es utilizado para enviar de todo: hipertexto, datos, código ejecutable, imágenes, audio, video, etc. [35] Por esta razón se lo considera el protocolo por excelencia de la Web.

También hay que considerar HTTPS (de *Hypertext Transfer Protocol Secure*) que representa una extensión de HTTP para conexiones seguras utilizando *Transport Layer Security* o TLS. En estos tiempos, la mayoría de sitios en la Web cuentan con un certificado digital y utilizan HTTPS.

Este protocolo utiliza una arquitectura Cliente - Servidor e implementa una comunicación de tipo “petición/respuesta” donde el cliente siempre inicia la conexión mediante una solicitud, y el servidor (en permanente escucha) atiende y responde el pedido.

HTTP define paquetes divididos en dos: encabezado y cuerpo. El primero contiene información como el tipo de mensaje, el método HTTP, código de respuesta, localización del recurso, tipo de contenido, entre otros. El cuerpo encapsula el mensaje a ser transmitido, ya sea hipertexto, datos, imágenes, etc.

Identificador de Recursos Uniforme (URI)

Es una cadena de caracteres que permite identificar un recurso en la red. En la Web, un sitio es accedido utilizando su URI. Tanto localizadores (URLs) como nombres (URNs) son por definición URIs, pero comúnmente se suele llamar URL a las direcciones Web.

Un identificador se compone por: esquema, autoridad, ruta, consulta y fragmento. Es jerárquica y su sintaxis luce de la siguiente manera:

```
URI = scheme ":" ["//" authority] path ["?" query] ["#" fragment]
```

Métodos de Petición HTTP

Una solicitud HTTP cuenta con un método que representa una acción para un determinado recurso. Existen algunos métodos, pero los cuatro más importantes son: GET, POST, PUT y DELETE, los mismos que reflejan las acciones de un CRUD (*Create, Read, Update, Delete*), un concepto fundamental al administrar datos.

GET permite al cliente obtener el contenido de un determinado recurso sin realizar cambios en el servidor.

POST es utilizado si el cliente desea insertar contenido dentro de un recurso o crear un nuevo recurso, por lo que el cliente debe adjuntar la información a ser "posteada" dentro del cuerpo del mensaje siguiendo las reglas semánticas establecidas por el servidor.

PUT da la posibilidad de modificar o actualizar el contenido del recurso objetivo sin tener que instanciar uno nuevo.

DELETE se utiliza para borrar un recurso, aunque muchas veces, este último no es del todo borrado sino archivado y esto dependerá completamente de la implementación del servicio Web.

Códigos de estado HTTP

Tras recibir una petición, el servidor devolverá un mensaje informando el estado de la solicitud. Para estandarizar las posibles respuestas se utilizan códigos de estado definidos por un número de tres dígitos; cada código tiene un significado único, pero son agrupados en cinco clases:

1XX para respuestas de información.

2XX si el mensaje fue recibido o procesado de manera exitosa.

3XX indicando una redirección y futuras acciones.

4XX para errores generados por parte del cliente.

5XX si existen errores del lado del servidor.

1.3.3.4 Web Stack

Es un conjunto de tecnologías necesarias para entregar una página Web completa; generalmente está compuesta por: un sistema operativo, un programa dedicado a escuchar peticiones conocido como "servidor web", una base de datos para la persistencia de información, lenguajes de programación y *frameworks*. [36]

Este conjunto es nombrado utilizando las iniciales de las tecnologías que lo componen, por ejemplo:

- MEAN: MongoDB, Express.js, Angular, Node.js.
- LAMP: Linux, Apache, MySQL o MariaDB, PHP y Perl.
- MERN: MongoDB, Express.js, React.js, Node.js

Los componentes del *stack* suelen ser intercambiables y no se limitan a las tecnologías mencionadas; sin embargo, es imperativo considerar que mientras más crece la pila, mayor será la complejidad de la solución completa, por lo que es recomendable ajustar el *stack* a los requisitos del sitio web.

1.3.3.5 Servidor Web

Es un programa informático dedicado que ofrece un servicio Web, es decir, escucha y responde peticiones HTTP en uno o varios puertos de red enlazando la solicitud con el recurso o fragmento de código objetivo. [37] En adición se encarga de redirecciones, certificados TLS, autenticación, entre otros. Algunos ejemplos son: Apache, Nginx, Internet Information Services (IIS), Node.js, etc.

Es importante diferenciar entre un servidor físico (hardware) y el software servidor; también se debe mencionar que este servicio Web puede proveer tanto *Front-End* como *Back-End*. Por lo general se reserva el puerto de red 80 para HTTP y 443 para HTTPS.

1.3.3.6 Navegador Web

Como el servidor, éste también es un programa informático, pero actúa como cliente. Básicamente cumple la función de Interfaz Gráfica, dándole la posibilidad al usuario final de interactuar con la Web sin tener que utilizar una consola o escribir código. Este “browser” [38] tiene algunos componentes como: caché, extensiones, herramientas, *cookies*, historial; pero destacan dos, una sección de cuadro de texto para ingresar y acceder a un URI y el DOM o *Document Object Model*.

El DOM es una interfaz que recibe documentos XML o HTML y los modela en una estructura de datos tipo árbol, donde cada nodo es un Objeto. De esta manera el navegador puede renderizar en pantalla el contenido estructurado en el documento de entrada. Actualmente, este modelo se encuentra estandarizado para asegurar la portabilidad de la Web y todo navegador debería seguir dicho estándar.

Algunos ejemplos de *browser* son: Chrome de Google, Edge de Microsoft, Firefox (*open-source*), Brave (con bloqueador de anuncios), entre otros. Pero la gran mayoría de usuarios prefiere Chrome, por lo que es común que el software sea desarrollado enfocándose en éste.

1.3.3.7 Web App

También conocida como Aplicación Web, es un programa informático que "corre" sobre la Web [39]. Es la evolución de los sitios web caracterizados por un contenido estático, maximizando la interacción con el usuario, aumentando la complejidad del software y consecuentemente, utilizando más recursos de hardware en cliente y servidor.

Esta tecnología nace con la Web2 como respuesta a la necesidad de interactividad en sitios como Facebook o Youtube, y es rápidamente adoptada por otras entidades por su capacidad de atraer nuevos usuarios.

Dentro del desarrollo web se utiliza comúnmente los términos: "*Frontend*" y "*Backend*" [40] para dividir una aplicación web. Cabe mencionar que esta situación genera tres roles en la industria: desarrollador *Frontend*, desarrollador *Backend*, y desarrollador *FullStack*; [41] siendo este último un programador con habilidades en los dos lados en vez de especializarse en uno solo.

Front-End

El *Front-End* o *Frontend* es una parte de la aplicación web; se define como el software que "corre" en el cliente, generalmente en el navegador. Este componente se encarga de la lógica de presentación de la app, es decir, contiene el código a ser renderizado por el navegador en forma de GUI o (*Graphical User Interface*).

Las tres tecnologías más importantes del *Frontend* son: HTML, CSS y JS; sin embargo, renderizar contenido dinámico puede ser una tarea muy compleja, especialmente si la información se encuentra almacenada en el servidor. Por esta razón se utilizan marcos de trabajo o librerías que contienen soluciones a problemas comunes en el desarrollo *Frontend*.

Back-End

También conocido como *Backend*, es el software que "corre" en el servidor. Este se encarga de la lógica de dominio o negocio y de la persistencia de los datos en caso de requerirla. Comúnmente, el *Backend* es una interfaz de programación de aplicaciones que

utiliza HTTP para comunicarse con el cliente respondiendo sus peticiones y un ORM para interactuar con la base de datos.

Al aumentar la complejidad del dominio, otras tecnologías y arquitecturas deberán ser añadidas al *Back-End* según la necesidad, entre éstas destacan: bases de datos en caché, autenticación, integraciones con otras APIs en la web, contenerización, microservicios, mensajes y colas, etc.

1.3.3.8 REST API

Para entender este concepto primero se debe definir el significado de API. Una Interfaz de programación de aplicaciones (API) hace referencia a una abstracción que permite a un componente de software interactuar con otro sin tener que entender cómo funciona internamente. Haciendo una analogía, una API es como el volante de un automóvil, que hace posible direccionar el vehículo sin que el conductor deba conocer el sistema mecánico dentro del carro.

Dentro del contexto Web, una API es un servicio que escucha peticiones de clientes y abstrae su funcionamiento interno, proporcionando una interfaz para que dichos clientes interactúen con el servidor. Ahora, cuando una API cumple las condiciones dadas por REST [42], ésta pasa a ser RESTful.

Representational State Transfer

En español “Transferencia de Estado Representacional”, es una arquitectura de sistemas distribuidos creada para la Web. Esta se basa en recursos y no en acciones como otras tecnologías distribuidas como SOAP (*Simple Object Access Protocol*) o RPC (*Remote Procedure Call*).

REST fue creada bajo cinco principios planteados por Roy Fielding, uno de los principales autores de la especificación de HTTP y éstos son: Arquitectura cliente-servidor, ausencia de estado, posibilidad de caché, sistema por capas, e interfaz uniforme. [43]

Actualmente las condiciones impuestas por REST no son tan estrictas como en su concepción, pero de cierta manera se mantienen los principios. Una API REST actual:

- Implementa únicamente lógica del servidor dentro del contexto de la aplicación.
- No tiene estado por lo que cada petición se procesa de manera única, a menos que un cliente que mantiene sesión indique lo contrario y el servidor tenga caché.
- Dispondrá de memoria caché si ésta ayuda a optimizar el servicio.

- Recibe URIs estructuradas de manera explícita, por ejemplo: “https://pokeapi.co/api/v2/pokemon/ditto” donde se apunta el recurso “ditto” en la colección “pokemon”.
- Mantiene un formato estandarizado dentro del cuerpo del mensaje, es muy común el uso de JSON por su facilidad de lectura para desarrolladores.

1.3.4 TECNOLOGÍAS IOT

Dentro de la Industria 4.0, una de las tecnologías con mayor auge es el Internet de las Cosas. Su importancia radica en la necesidad de generar datos para su análisis, así como automatizar procesos que requieran de una interacción física con el entorno.

En años pasados, esta tecnología atravesaba un crecimiento sumamente acelerado debido a la gran oferta de dispositivos inteligentes capaces de conectarse a Internet; éstos a su vez, se caracterizaban por un costo relativamente bajo que permitía escalar los sistemas y generar servicios comerciales muy rentables. Luego, la pandemia por el COVID-19 obstaculizó las cadenas de suministro de microcontroladores en todo el mundo, disminuyendo la oferta al mercado creciente del IoT y afectando directamente su desarrollo. [44] [45]

Actualmente, con la caída del mercado de cryptomonedas, la demanda de microcontroladores ha disminuido, dando un respiro a otros sectores como el IoT. También es importante mencionar la mitigación de la pandemia que restablece parcialmente la oferta de chips, así como la invasión de Rusia a Ucrania [46], cuya producción de gases nobles es vital para crear dispositivos electrónicos [47]. Estas y otras situaciones pintan un panorama mixto pero alentador para el IoT [48], que ya muestra una tendencia al crecimiento otra vez.

1.3.4.1 Internet of Things

El Internet de las Cosas es todavía un concepto en evolución que incluye conocimientos de diferentes áreas: ciencias computacionales, sistemas embebidos, ingeniería de hardware y software, telecomunicaciones, sistemas de información, ciberseguridad, inteligencia artificial, análisis de datos, etc.

Se define como una red de dispositivos (limitados en cuanto a recursos de hardware) interconectados que provee una plataforma caracterizada por cierto nivel de inteligencia y por la capacidad de automatizar procesos y recolectar datos [49]. El objetivo de esta

plataforma es brindar servicios a usuarios, mejorando su calidad de vida ó a empresas, optimizando recursos y aumentando su producción.

1.3.4.2 IoT vs Web

En el apartado anterior se discutieron las diferencias entre Web e Internet, ahora se añade IoT a la comparación para entender mejor este concepto, así como su filosofía, enfoque y aplicación. Primero hay que mencionar que las dos tecnologías funcionan con Internet, sin embargo, la Web funciona con HTTP estrictamente en la capa de Aplicación de TCP/IP, en cambio IoT tiene varios protocolos que pueden ser implementados según el caso de uso. También es necesario considerar la naturaleza de la información transmitida, mientras que en Web es variada y dirigida al usuario, en IoT son datos restringidos en tamaño y compartidos entre dispositivos limitados.

En la tabla 1.1 se realiza un resumen de las diferencias clave entre las tecnologías mencionadas en esta sección.

Tabla 1.1 Comparación Web vs IoT

	Web	**IoT**
Definición	Software que “corre” sobre Internet	Plataforma creada por una red de dispositivos limitados
Uso de Internet	Estrictamente HTTP	Protocolos en diferentes capas y arquitecturas
Tipo de información	Hypertexto, datos, multimedia	Datos de tamaño limitado
Orientación	Hacia los usuarios	Hacia dispositivos limitados
Implementación	Mayormente a través de software	A través de Hardware y Software

1.3.4.3 Arquitecturas IoT

IoT es una tecnología joven y aún no dispone de un estándar aceptado por todos, pese a esto, la necesidad de escalabilidad implícita en el concepto, hace imperativo el usar una abstracción en el sistema que permita el crecimiento sin añadir demasiada complejidad. La respuesta más frecuente es la arquitectura por capas [50]. A continuación, se describen las arquitecturas más importantes.

Arquitectura IoT de tres capas

Indica que los componentes del IoT se separan en tres capas adyacentes, según las funciones de dichos componentes:

- Capa de Percepción: Incluye los elementos que interactúan con el medio físico y sus controles (sensores, actuadores, embebidos, microcontroladores, entre otros).
- Capa de Red: Agrupa elementos cuyas funciones se basan en la interconexión y el transporte de la información.
- Capa de Aplicación: Engloba los componentes encargados de estructurar la plataforma de servicios, generalmente con tareas de: almacenamiento y procesamiento de datos, control y administración de dispositivos, abstracción de funciones e interfaz para usuarios, etc.

Arquitectura IoT de cinco capas

Extiende al modelo de tres capas dividiendo la última en otras tres para dar un total de cinco. Esta división separa el bloque superior con el fin de especificar con mayor detalle las responsabilidades de cada nueva capa, y éstas son: [50]

- Capa de Procesamiento: Para el almacenamiento y procesamiento de datos.
- Capa de Aplicación: Para definir y proveer los servicios a los usuarios.
- Capa de Negocio: Para la administración no solo de dispositivos, sino también de los servicios, funcionalidades, aplicaciones, modelos de negocio, etc.

Arquitectura IoT Fog Computing

Se basa en el modelo de tres capas añadiendo una extra entre Percepción y Red. La idea principal se basa en “Fog Computing” [51] o computación en la niebla, la cual indica que parte de la inteligencia del sistema es implementada cerca de los dispositivos con el objetivo de reducir la latencia generada al procesar datos y tomar decisiones de manera centralizada. Algunas tareas descritas en la capa de Niebla son:

- Monitoreo de recursos de hardware, acceso y disponibilidad de elementos físicos.
- Preprocesamiento de datos para reducir la carga de mensajes.
- Almacenamiento temporal descentralizado de datos.
- Seguridad, encriptación de mensajes y autenticación de dispositivos.

1.3.4.4 Infraestructura de comunicaciones IoT

Considerando que se definió al IoT como una red, resulta necesario revisar algunos conceptos requeridos para entender la forma en que los diferentes dispositivos se interconectan para formar el sistema.

Redes y su clasificación

Una red es una conexión entre dos o más dispositivos que les da la posibilidad de intercomunicarse [52]. Generalmente se la representa como un grafo donde cada nodo es un dispositivo o computador y cada arista muestra una conexión. Las redes tienen algunas clasificaciones donde se las agrupa destacando un aspecto en particular.

Un ejemplo es el medio físico, consecuentemente se tienen redes cableadas e inalámbricas dependiendo si el medio es guiado o no.

También se considera la topología lógica de la red, dando como resultado configuraciones: de estrella, malla, anillo, árbol jerárquico, entre otras.

Otro aspecto es la función de la red, aquí se distingue entre redes de acceso que interconecta los dispositivos finales entre sí y con el resto de la red, y el "core" o red de núcleo que es la parte centralizada donde converge y se distribuye la información hacia las de acceso.

Una red también es clasificada según su tamaño geográfico, definiendo así redes de área: personal, local, de campus, metropolitana y extendida, también conocidas como: PAN, LAN, CAN, MAN y WAN respectivamente.

Generalmente una red para soluciones IoT consta de una o varias redes de acceso con dispositivos limitados interconectados en malla o estrella a un *Gateway* que les brinda salida hacia un *core*, compuesto por puntos de acceso inalámbricos, *switches*, *routers*, que a su vez habilitan la conexión con Internet y consecuentemente a la parte centralizada del sistema (comúnmente en Nube ya sea como IaaS "*Infrastructure as a Service*" o plataformas dedicadas al IoT).

Protocolos de comunicaciones IoT

Con el fin de estandarizar las comunicaciones entre dispositivos IoT, se establecen varios protocolos que pueden ser implementados según el caso de uso de un sistema IoT. Éstos trabajan en diferentes capas y utilizan diferentes arquitecturas, pero también procuran mantener principios de interoperabilidad y separación de responsabilidades. La naturaleza de un sistema IoT implica algunas características que definen las reglas que un protocolo

implementará. Por ejemplo: los recursos de hardware limitados y la disponibilidad energética de los dispositivos IoT [53], la accesibilidad del entorno físico, la frecuencia y el modo de la comunicación, entre otras.

Consecuentemente los protocolos generalmente manejan: conexiones inalámbricas, bajo consumo energético, optimización de cabeceras y formato de mensajes, arquitecturas distribuidas, etc.

1.3.4.5 MQTT

Ideado inicialmente como “MQ Telemetry Transport” o “Message Queue Telemetry Transport” ahora solamente MQTT, es un protocolo de capa aplicación (de TCP/IP) diseñado para la comunicación M2M o “de máquina a máquina” y basado en la arquitectura distribuida pub/sub o publicador-suscriptor [54].

Características de MQTT

Es un protocolo sumamente ligero, razón por la cual se lo utiliza para intercomunicar dispositivos IoT. Requiere de un protocolo de transporte, generalmente TCP o UDP. Es un estándar abierto de OASIS y una recomendación ISO. Extiende el modelo cliente-servidor desacoplándolos en cliente-broker-cliente para la publicación y subscripción de mensajes. Cuenta con una variante destinada a redes de sensores llamada MQTT-SN. Utiliza por defecto los puertos 1883 y 8883 si se usa sobre TLS. Actualmente se mantienen dos versiones del protocolo: v3 y v5, siendo esta última una extensión que soluciona algunos problemas como códigos de razón en ACKs o subscripciones compartidas.

Funcionamiento de MQTT

MQTT separa las responsabilidades de los integrantes de la comunicación en clientes y *broker*. La idea es que los clientes puedan “empujar” mensajes a través de una publicación en vez de “jalarlos”, como en el típico modelo cliente-servidor, también un cliente puede suscribirse a un tópico para recibir mensajes empujados hacia dicho tópico.

Un *broker* es un servidor intermediario que se encarga principalmente de reenviar los mensajes publicados a sus respectivos subscriptores. También debe mantener la conexión con los clientes mediante sesiones, conocer y asegurar la estructura de tópicos, llevar un registro del estado de clientes y mensajes.

La información es transmitida mediante mensajes, éstos se caracterizan por ser pequeños y pueden representar tanto datos como acciones dentro del contexto IoT. Se organizan en

tópicos estructurados en un árbol conocido como *Topic Tree* [55]. Un mensaje contiene un parámetro llamado calidad de servicio, representado por un número:

- Cero si el mensaje debe ser enviado máximo una vez ya sea que llegue o no.
- Uno si el mensaje debe llegar, aunque tenga que ser reenviado.
- Dos si el mensaje debe llegar y no puede ser reenviado, por lo que requiere un saludo previo.

Otros conceptos importantes de MQTT involucran:

- Sesiones persistentes vs “limpias” definidas por el cliente al conectarse, éstas determinan si el *broker* llevará memoria de dicho cliente o no.
- Mensajes retenidos en el *broker* para clientes que recién se suscriben a un tópico que posee información inicial.
- Una “última voluntad y testamento” que representa un mensaje enviado por el *broker* a los suscriptores de un tópico cuyo publicador acaba de perder conexión.

1.3.4.6 Dispositivos IoT

Un dispositivo IoT es un equipo electrónico que interactúa de alguna manera con el entorno físico, por lo que también se los conoce como dispositivos de borde (*the edge*) [56]. Se caracterizan por ser pequeños y disponer de recursos limitados tanto en hardware como en suministro de energía. Los dispositivos más comunes son: sensores, actuadores y sus respectivos controladores.

Sensor

Es un componente electrónico que transduce una magnitud física en señales eléctricas, de manera que se pueda correlacionar entrada y salida y, consecuentemente, representar una medida del entorno físico mediante electricidad (generalmente voltaje). Si la salida es directamente proporcional a la entrada se trata de un sensor lineal, caso contrario se habla de un sensor no lineal.

Los dispositivos sensores también se clasifican en:

- Analógicos: Si la salida eléctrica es una señal continua tanto en tiempo como en amplitud.

- Digitales: Si el dispositivo incluye un bloque de conversión analógica-digital o ADC, que cumple funciones de muestreo, cuantización y codificación de la señal continua para obtener una salida discreta y codificada en bits.

Otros conceptos importantes de los sensores en el IoT [57] son:

- Ajuste de magnitud y señal en intervalos definidos.
- Amplificación, atenuación y filtrado de señales.
- Acoplamiento de impedancias
- Calibración e interpretación de señales
- Número de bits de información y resolución.

Actuador

Es un componente electrónico capaz de generar una acción o "actuar" en el entorno físico utilizando propiedades electromagnéticas [58]. Algunos ejemplos son: servomotores, electroválvulas, electroimanes, bombas de líquidos, luces, parlantes, relés, etc.

Hay que considerar que muchos actuadores requieren de un mayor suministro energético, por lo que comúnmente son separados del sistema de control e incluidos dentro de sistemas de potencia. Esta situación puede generar problemas de ruido e interferencia en el resto de dispositivos IoT, consecuentemente se presenta la necesidad de implementar una capa de percepción con componentes robustos en ambientes ruidosos.

Controlador

Es un dispositivo electrónico que interactúa con sensores y actuadores para darles funciones de control, procesamiento y comunicación. Este conjunto representa el concepto de "dispositivo inteligente" o "*Thing*" dentro del contexto IoT. Algunos ejemplos son: microcontroladores, PLC (*Programmable Logic Controller*), PID (*Proportional–Integral–Derivative controller*), FPGA (*Field Programmable Gate Array*), hardware dedicado y propietario; los más utilizados son los sistemas embebidos.

Sistema Embebido

Este concepto hace referencia a un conjunto de componentes electrónicos ensamblados en una placa de tamaño reducido. En IoT, se caracteriza por incluir hardware que cumple funciones comunes como: procesamiento, regulación energética, comunicaciones guiadas

y no guiadas, ADC (*Analog-to-Digital Converter*) y DAC (*Digital-to-Analog Converter*), multiplexación de señales, puertos y pines de propósito general entre otras.

Dos ejemplos muy comunes son Arduino y Raspberry Pi [59]. Éstas son placas de desarrollo diseñadas para el aprendizaje y el prototipado de proyectos, pero también son vistas en ambientes de producción por su facilidad de uso y su bajo costo.

Un Arduino se basa en un microcontrolador de 8 bits montado en una placa con regulación de voltaje, entrada USB (*Universal Serial Bus*) y su conversión a UART (*Universal Asynchronous Receiver-Transmitter*), y puertos I/O (*Input/Output*) analógicos y digitales.

La organización de Arduino provee también un IDE (*Integrated Development Environment*) para el desarrollo simplificado de código basado en C++ para ser "quemado" como *firmware* del microcontrolador.

Un Raspberry Pi se basa en un microprocesador ARM (*Advanced RISC Machines*) y es básicamente una computadora de propósito general embebida en un chip, por lo que cuenta con puertos para periféricos, HDMI (*High-Definition Multi-media Interface*), USB, Ethernet, tarjeta SD (*Secure Digital*), así como GPIO (*General Purpose Input/Output*) de tipo digital. Este requiere un sistema operativo similar a los de propósito general, destacando Raspbian, NOOBS y Windows IoT.

ESP32

Es un grupo de sistemas embebidos creados por ESPRESSIF para soluciones que requieran de conexión inalámbrica. Éstos se basan en módulos que combinan un microprocesador y un sistema de radiofrecuencia [60].

No es tan popular como Arduino y Raspberry Pi porque presenta una curva de aprendizaje más marcada, sin embargo, cuenta con las mejores características de ambos:

- Recursos de Hardware más cercanos al Raspberry Pi.
- Muy bajo consumo energético.
- Sistema operativo basado en FreeRTOS.
- Consta con módulos de Wi-Fi y Bluetooth.
- Dispone de ADC y DAC (según el modelo).
- Listo para ambientes de producción.
- Cuenta Esp-Idf, un *framework* de desarrollo que integra varios SDK utilizables según el caso de uso.

2 METODOLOGÍA

En este capítulo se detalla el diseño y la implementación del Sistema utilizando: Kanban como metodología ágil, Jira para administrar el desarrollo del proyecto, Diagramas de arquitecturas y UML para describir el software, así como las herramientas para implementarlo.

2.1 PLANIFICACIÓN DEL FLUJO DE TRABAJO

Debido a la naturaleza flexible de Kanban, un proyecto puede ser administrado de muchas maneras, razón por la cual resulta pertinente definir el flujo de trabajo que el presente proyecto aplicará.

2.1.1 ESTABLECIMIENTO DE TARJETAS KANBAN

Jira sugiere utilizar historias de usuario para definir problemas horizontales siguiendo la fórmula "*As a [persona], I [want to], [so that].*" [26]; éstas a su vez pueden ser agrupadas en "*Epics*" o épicos y divididas en "*Tasks*" o tareas para completar el proyecto. Este concepto es ilustrado en la figura 2.1.

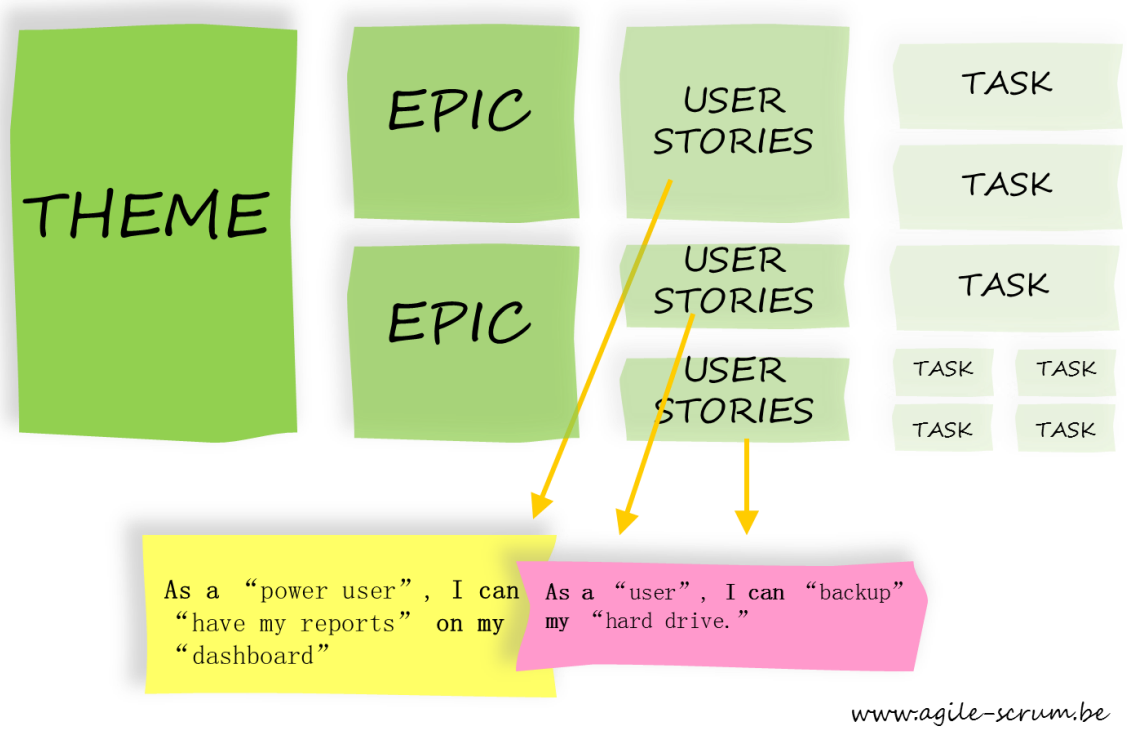


Figura 2.1 Ejemplo de User Stories [61]

En el presente proyecto:

- *Theme* es el Sistema a desarrollar.
- Un *Epic* representa un conjunto de historias que están relacionadas entre sí.
- Un *Task* divide el problema que presenta una historia de usuario para plantear una solución mediante el cumplimiento de varias tareas.

En la captura de pantalla de la figura 2.2 se muestra la estructura de problemas en Jira; cabe mencionar que la plataforma asigna automáticamente íconos para diferenciar los diferentes elementos: proyecto, *Epic*, *User Story* y *Task.*

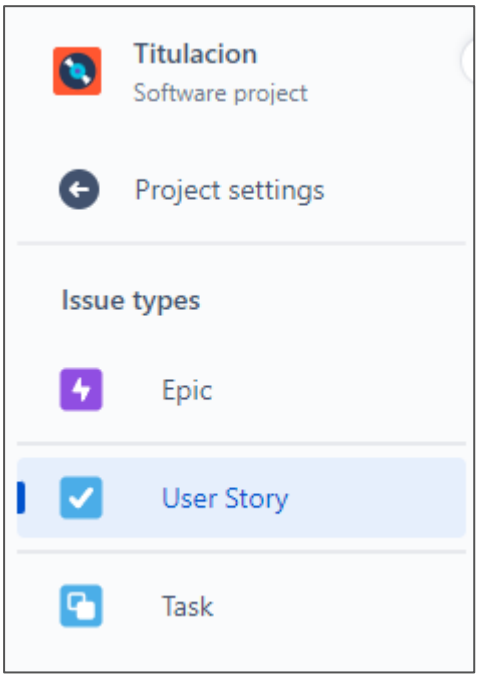


Figura 2.2 Estructura de *Issues* en Jira

La figura 2.3 muestra un ejemplo de tarjeta Kanban para *User Story* indicando los elementos característicos de una historia de usuario.

Nótese que el ID o identificador de historia de usuario es autogenerado por Jira y por lo general los dos primeros identificadores son reservados para el proyecto como tal y el *Epic* por defecto.

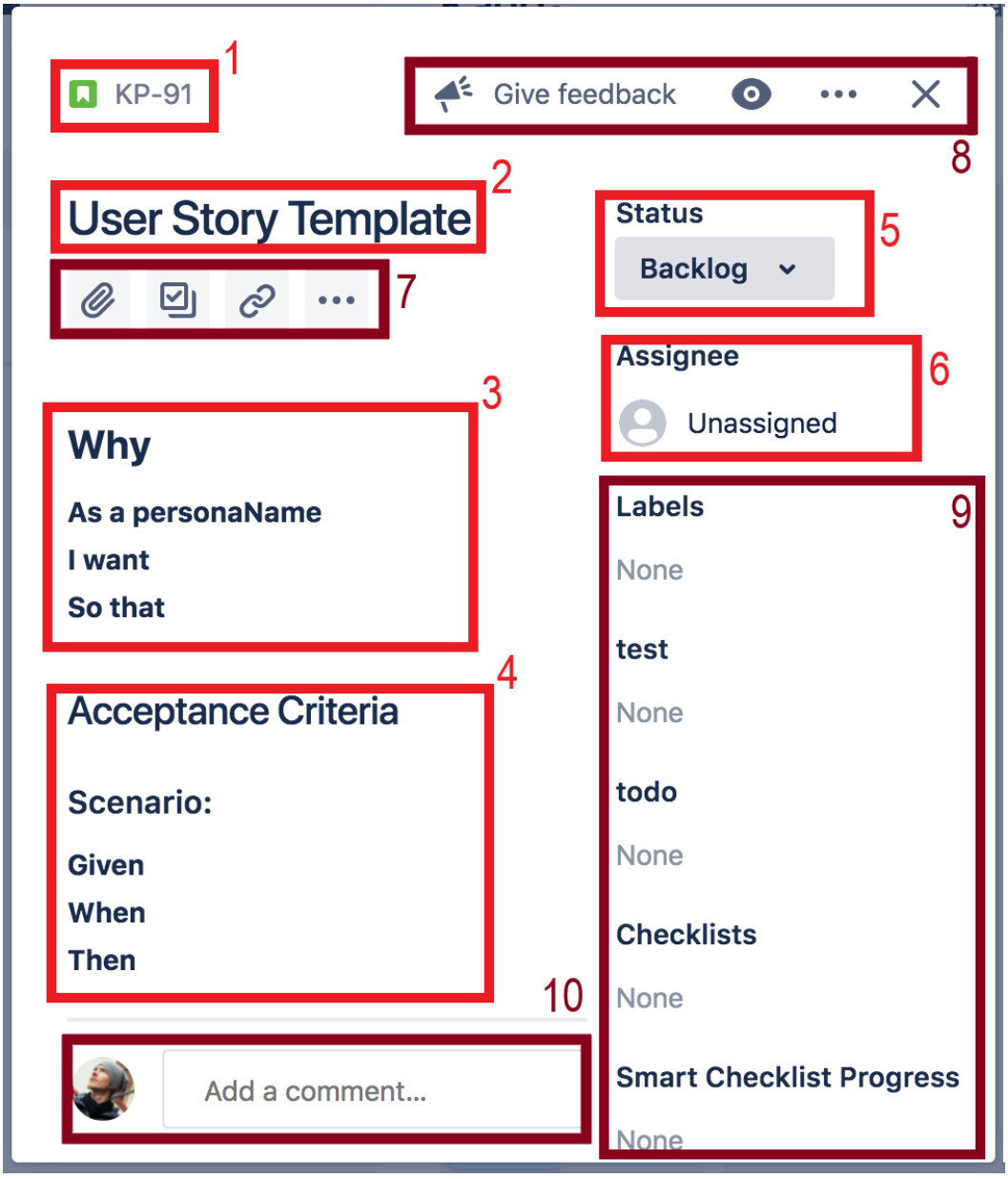


Figura 2.3 Ejemplo de *User Story* [62]

Para facilidad del lector, se resalta y enumera las secciones más importantes en rojo y las secundarias en un matiz más oscuro. A continuación, se enlistan las secciones:

1. ID de historia de usuario.
2. Título de la historia de usuario
3. Descripción de la historia de usuario.
4. Criterio de aceptación de historia de usuario.
5. Estado (columna del tablero Kanban) de la tarjeta de usuario.

6. Desarrollador encargado de la tarjeta.
7. Acciones de la tarjeta.
8. Más acciones de la tarjeta.
9. Más información de la tarjeta.
10. Añadir un comentario a la tarjeta.

2.1.2 ESTRUCTURA DEL TABLERO KANBAN

En este proyecto se establecen cuatro columnas (ver figura 2.4):

- ***TO DO***: Define la fase inicial de un "*Issue*" o problema.
- ***IN PROGRESS***: Agrupa los problemas en desarrollo.
- ***TESTING***: Esta fase agrupa problemas listos para pruebas y análisis de resultados.
- ***DONE***: Es la fase final del proyecto con problemas terminados.

No se establece un WIP ya que la cantidad de trabajo por desarrollador y la disponibilidad del equipo de desarrollo dependen de una sola persona. También se prioriza el movimiento de problemas hacia la fase "TESTING" requerido en capítulos posteriores del presente trabajo.

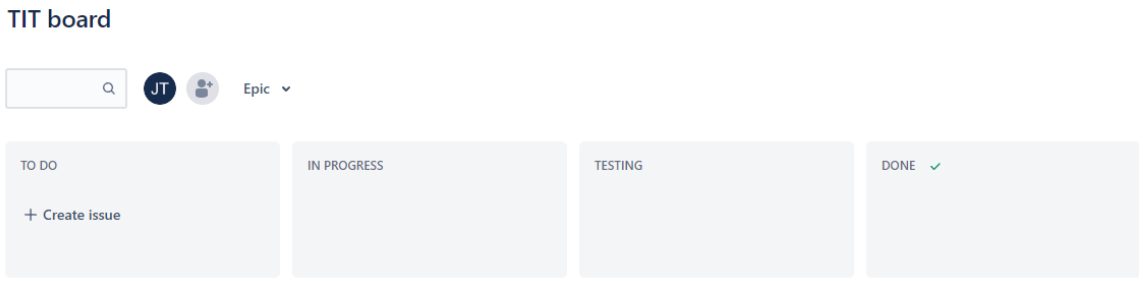


Figura 2.4 Dashboard Kanban en Jira Software

2.1.3 ESTABLECIMIENTO DEL FLUJO DE TRABAJO

Para completar el presente proyecto será necesario seguir los siguientes pasos:

- Extraer los requerimientos del sistema mediante entrevistas.
- Definir los requerimientos funcionales como historias de usuario en Jira.

- Definir los requerimientos no funcionales como atributos de calidad.
- Utilizar los requerimientos definidos para diseñar el sistema.
- Utilizar el diseño establecido para definir las tareas o *Tasks* de Jira necesarias para completar cada *User Story*.
- Implementar el sistema a través del cumplimiento de las diferentes tareas, actualizando el tablero Kanban en cada iteración.
- Probar el sistema asegurándose que el criterio de aceptación de cada historia de usuario sea cumplido antes de dar dicha historia por completada.

La figura 2.5 resume el flujo de trabajo descrito anteriormente.

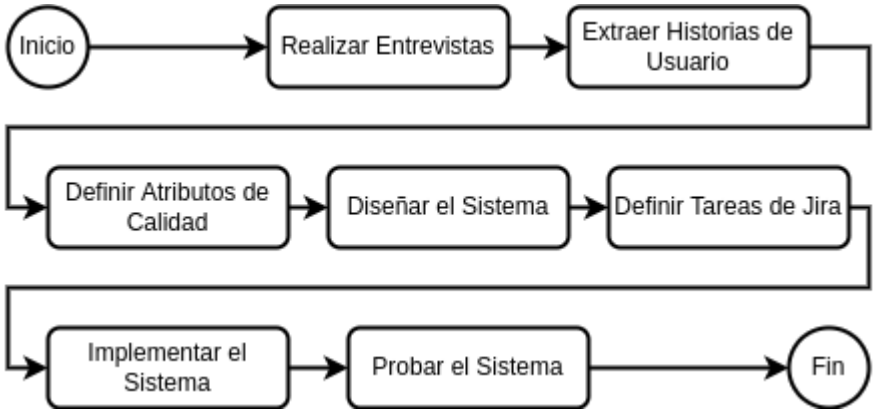


Figura 2.5 Diagrama de Flujo de Trabajo Establecido

2.2 REQUERIMIENTOS DEL SISTEMA

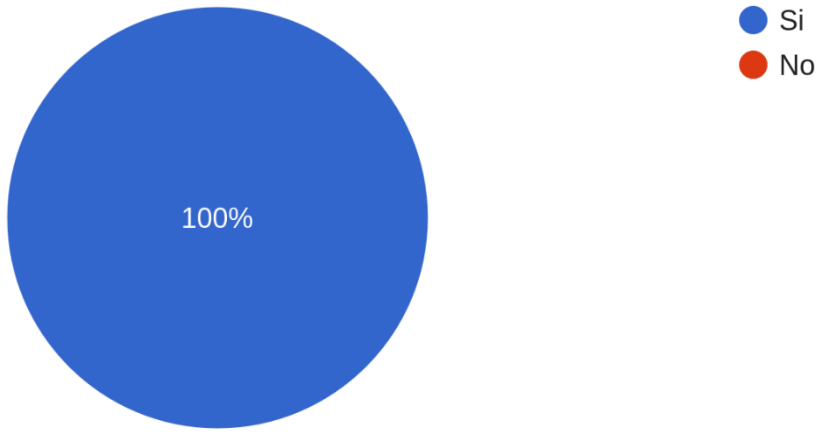
Para definir los requerimientos, se realizan tres entrevistas al personal de una florícola. A continuación, se extraen los requisitos funcionales del proyecto como historias de usuario y se plantean los requisitos no funcionales como los atributos de calidad del sistema.

2.2.1 ENTREVISTAS

Con el fin de entender las necesidades de los usuarios se realizan tres entrevistas estructuradas utilizando la plataforma Google Forms. Los resultados se muestran a continuación.

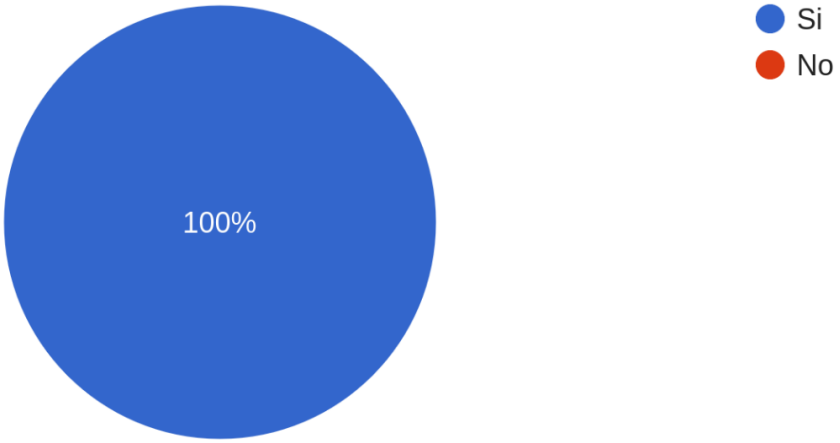
2.2.1.1 Pregunta 1

¿Considera usted importante realizar mediciones de las condiciones ambientales en un cultivo?



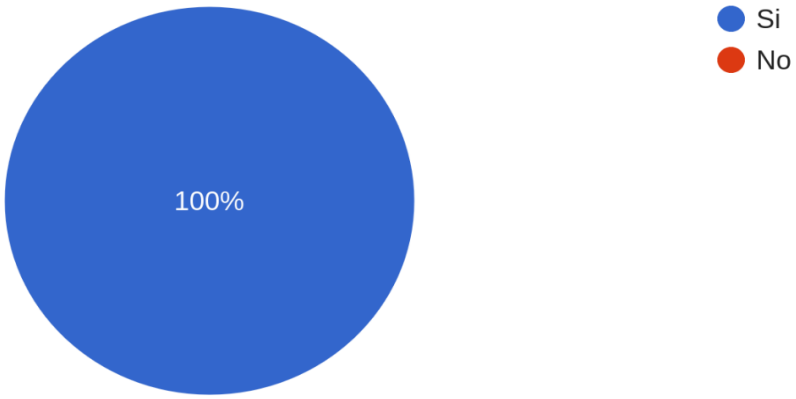
2.2.1.2 Pregunta 2

¿Le gustaría disponer de un sistema capaz de generar dichas mediciones de manera automática?



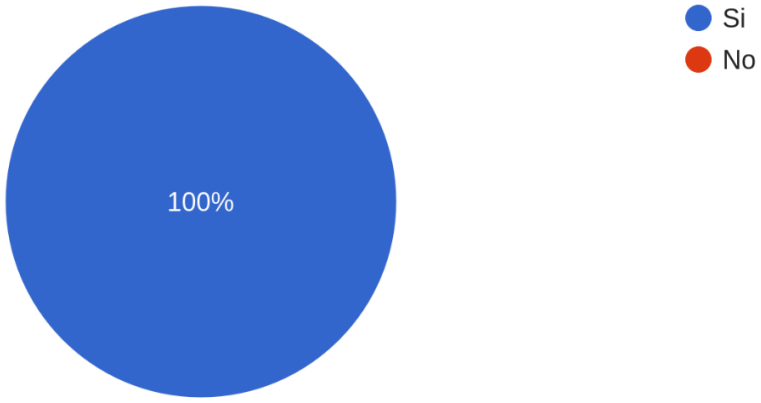
2.2.1.3 Pregunta 3

¿Considera usted importante guardar de alguna manera las mediciones para poder consultarlas después?



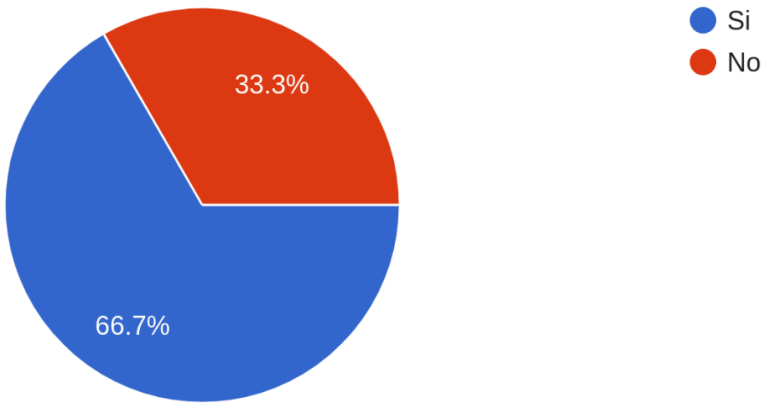
2.2.1.4 Pregunta 4

¿Le gustaría que el mismo sistema que recolecta mediciones ambientales se encargue de guardarlas?



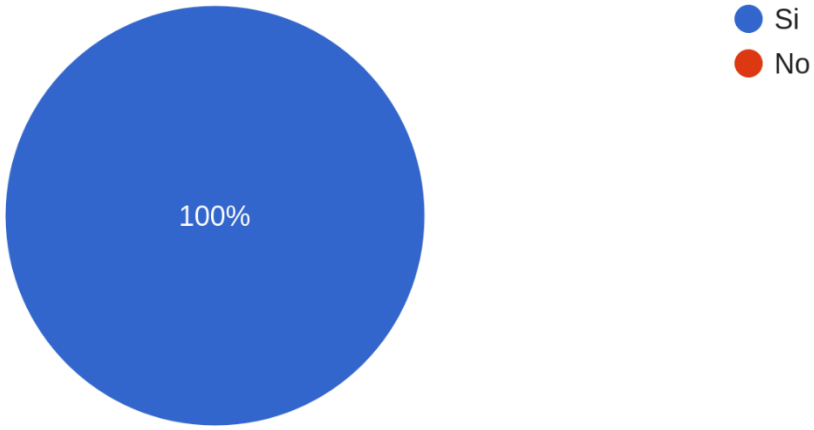
2.2.1.5 Pregunta 5

¿Considera usted importante poder asociar dichas mediciones a la infraestructura de la finca?



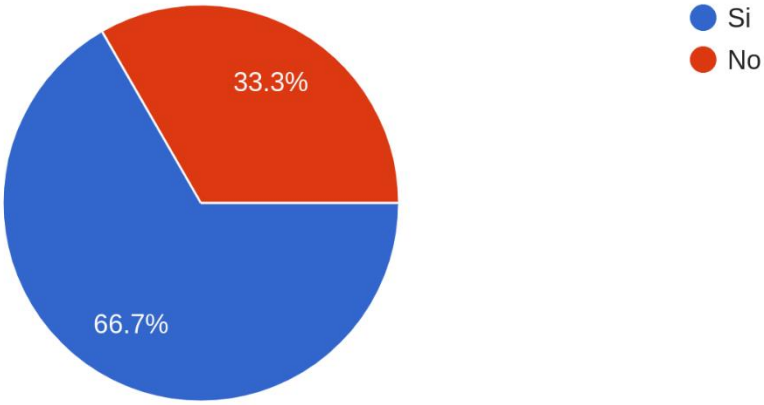
2.2.1.6 Pregunta 6

¿Considera usted importante que el sistema se ajuste al tamaño de los cultivos, permitiendo añadir nuevos medidores de condiciones ambientales?



2.2.1.7 Pregunta 7

¿Considera usted importante poder visualizar los cambios en los cambios ambientales utilizando gráficos?



2.2.2 HISTORIAS DE USUARIO

De las entrevistas se obtienen las historias de usuario descritas en Jira, que se indican a continuación.

2.2.2.1 Historia de usuario TIT-3

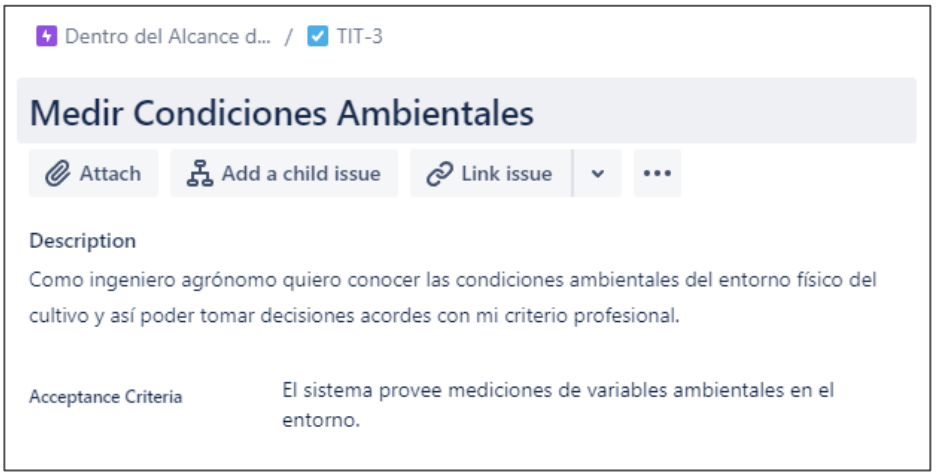


Figura 2.6 Historia de Usuario TIT-3

2.2.2.2 Historia de usuario TIT-4

Figura 2.7 Historia de Usuario TIT-4

2.2.2.3 Historia de usuario TIT-5

Figura 2.8 Historia de Usuario TIT-5

2.2.2.4 Historia de usuario TIT-6

Figura 2.9 Historia de Usuario TIT-6

2.2.2.5 Historia de usuario TIT-7

Figura 2.10 Historia de Usuario TIT-7

2.2.2.6 Historia de usuario TIT-8

Figura 2.11 Historia de Usuario TIT-8

2.2.2.7 Epic con ID: TIT-2

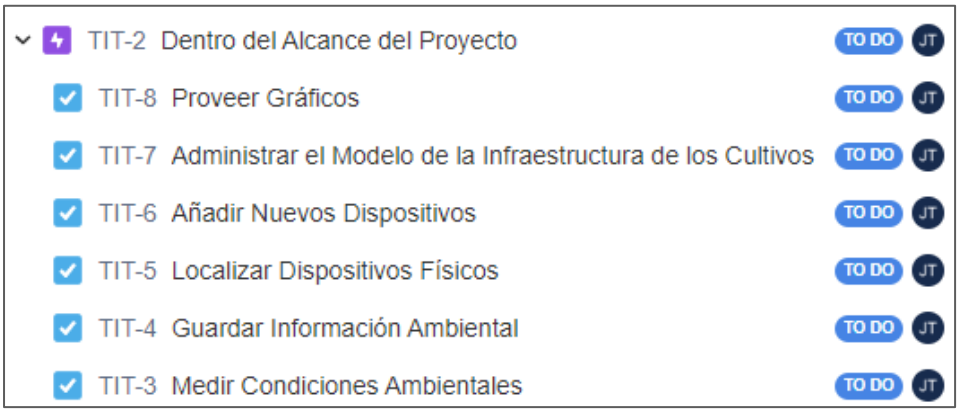


Figura 2.12 Epic con las Historias de Usuario creadas

2.2.3 ATRIBUTOS DE CALIDAD

Una vez definidos los requerimientos funcionales, se establecen los requerimientos no funcionales en forma de atributos de calidad. Existen muchos atributos que impactan en la calidad del software, idealmente un proyecto cuidaría de cada uno de dichos atributos, pero en la práctica un sistema resulta exitoso si prioriza los requisitos no funcionales indispensables para su correcto funcionamiento y a su vez mantiene cierta flexibilidad para responder a los cambios en las necesidades del software. Este proyecto requiere:

2.2.3.1 Escalabilidad del Sistema

Si bien no se espera que el número de usuarios del proyecto aumente exponencialmente, se presentan preguntas como: ¿Qué pasa si aumenta la infraestructura de la finca o se adquiere otra?, ¿Y si es necesario medir nuevos parámetros o añadir nuevos controles? La respuesta es disponer de la escalabilidad que le permita al sistema crecer con el número de dispositivos físicos y funcionalidades del sistema; esto a su vez implica cierto nivel de modularidad en la solución para que éste pueda responder a la evolución de necesidades sin tener que afectar todo el sistema.

2.2.3.2 Interoperabilidad entre Componentes

Como consecuencia del atributo anterior, se conciben diferentes componentes de un sistema modular. Esto genera preguntas como: ¿De qué manera se intercomunican estos componentes?, ¿Si se añaden o modifican componentes será necesario modificar otros para adaptar la comunicación?. Entonces se debe asegurar un nivel de interoperabilidad en el sistema mediante protocolos, estándares y tecnologías comunes en la industria, que estandaricen la intercomunicación entre las partes del sistema.

2.3 DISEÑO DEL SISTEMA

Siguiendo los principios de las metodologías ágiles, Kanban sugiere código útil sobre una documentación fuerte y estática; sin embargo, resulta imperativo describir la solución en términos generales ya sea para seleccionar las tecnologías correctas, así como para definir las tareas de Jira que, a su vez, detallarán el cómo se pretende implementar el código y cumplir los requisitos del proyecto.

2.3.1 CONSIDERACIONES GENERALES

- Las aplicaciones web actualmente dominan las soluciones empresariales por su capacidad de centralizar la información y proveerle al usuario una plataforma uniforme que solo requiere de un navegador web, una herramienta sumamente familiar para dicho usuario. El proyecto en desarrollo requiere de estas características para cumplir con los requerimientos extraídos.
- Al requerir varios dispositivos físicos interactuando con el entorno y con el resto del sistema, la web puede no ser la mejor opción. En principio se podrían considerar pequeños servidores web dentro de los dispositivos para intercomunicarlos; esto limitaría la escalabilidad del sistema, además delegaría demasiada responsabilidad a los dispositivos. Una mejor alternativa es utilizar tecnologías IoT que se ajustan con la naturaleza de carácter limitado y de tráfico frecuente pero pequeño.

2.3.2 ESTRUCTURA DEL SISTEMA

Siguiendo las consideraciones anteriores, así como los atributos de calidad, se divide la solución en componentes. Algunos deben ser desarrollados, otros implementados; y, para asegurar la escalabilidad del sistema, se podrán agregar nuevos componentes siempre y cuando exista la interoperabilidad necesaria (ver Tabla 2.1).

Tabla 2.1 División del Sistema y sus funciones

Aplicación de Front-End	Encargada de funciones de GUI, además consulta, estructura y grafica la información necesaria.
Servicio Web RESTful	Provee una API al FE para utilizar los servicios centralizados del BE.
Base de Datos	Estructura y almacena los datos en un modelo relacional.
Recolector de Datos	Suscribe y recolecta la información ambiental de los dispositivos.
Broker MQTT	Coordina y redirige el tráfico MQTT entre dispositivos.
Dispositivos Físicos	Proveen el control de sensores y actuadores y se comunican con el resto del sistema.

Cabe mencionar que para cumplir con una historia de usuario pueden intervenir uno o varios de dichos componentes.

2.3.2.1 Arquitecturas Distribuidas

La figura 2.13 muestra el sistema en una arquitectura cliente servidor en comparación a una arquitectura publicador-suscriptor.

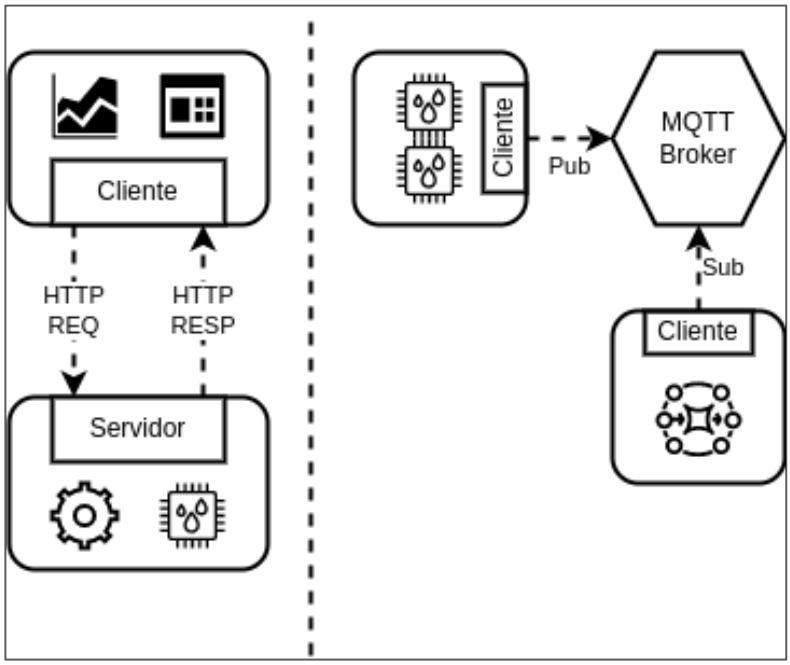


Figura 2.13 Arquitectura Cliente/Servidor vs Pub/Sub

A la izquierda se observa la separación de responsabilidades de los componentes en cliente y servidor intercomunicados por HTTP. El cliente consta de la FE app y “corre” sobre el navegador del usuario; el servidor provee una interfaz de comunicación para abstraer el resto de los componentes. A la derecha se ve la separación de los componentes IoT intercomunicados a través de MQTT.

2.3.2.2 Arquitecturas de Capas

En la figura 2.14 se observa el sistema en una arquitectura de tres capas tradicional en comparación a una arquitectura IoT.

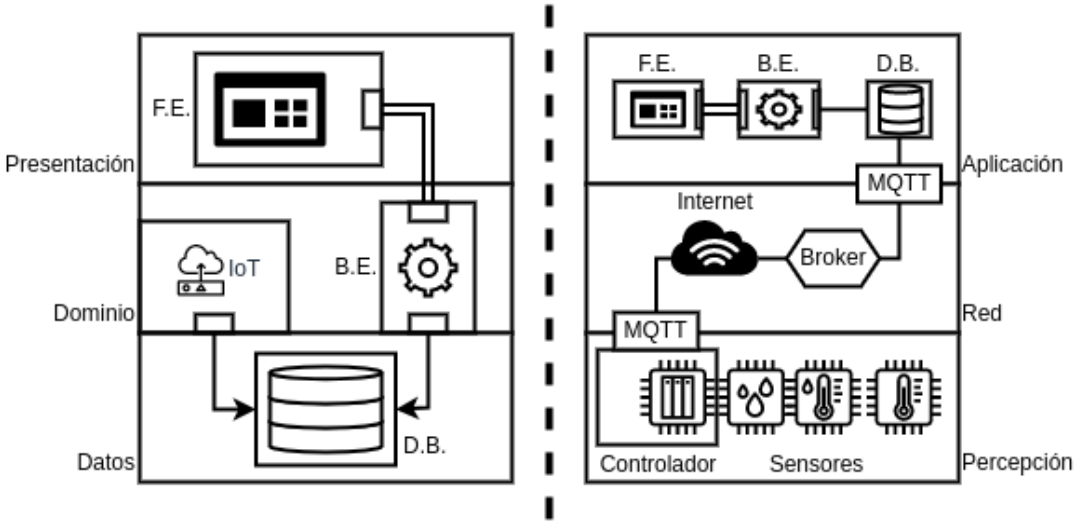


Figura 2.14 Arquitectura de Tres capas Tradicional vs IoT

En la parte izquierda de la figura se observa cómo se dividen los componentes en las tres capas tradicionales del software: Presentación, Dominio y Datos. Cabe mencionar que la lógica correspondiente al IoT es abstraída en su propio bloque dentro del dominio y se comunica únicamente con la base de datos. En la parte derecha de la figura se presenta la arquitectura de tres capas en IoT agrupando los componentes en capas de: Aplicación, Red y Percepción. En esta última, se incluyen no solo conceptos de software sino también los dispositivos y cierta infraestructura de telecomunicaciones que habilita el sistema.

2.3.3 COMPONENTES DEL SISTEMA

En esta sección se establecen las tecnologías que se utilizan para cada componente, así como su arquitectura, especificaciones y algunos detalles de su diseño en caso de ser necesario.

2.3.3.1 Aplicación Web de Front End

Para proveer una interfaz gráfica al usuario se propone crear una aplicación utilizando Angular, un marco de trabajo creado y mantenido por Google que proporciona un gran número de soluciones comunes a problemas de FE.

La figura 2.15 muestra el diagrama de casos de uso, donde se presentan las funcionalidades que la interfaz gráfica debe ofrecer al usuario.

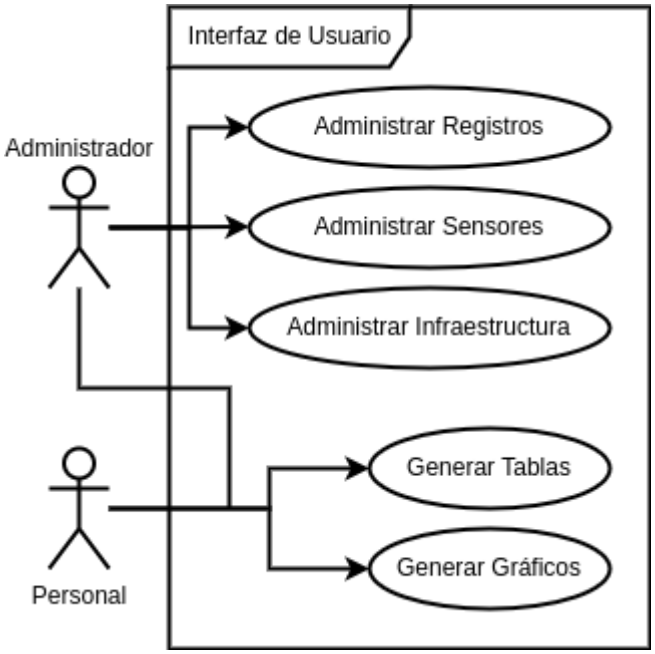


Figura 2.15 Diagrama de Casos de Uso

Estructura de Angular

En el desarrollo FE es importante entender el concepto de componentes FE (que no deben ser confundidos con los componentes del sistema); éstos son bloques que agrupan código HTML, CSS y JS para armar aplicaciones mediante el ensamblado de dichos bloques. Entre las herramientas más populares que utilizan este concepto están: React, Vue y Angular.

En la figura 2.16 se presenta la Arquitectura definida por Angular.

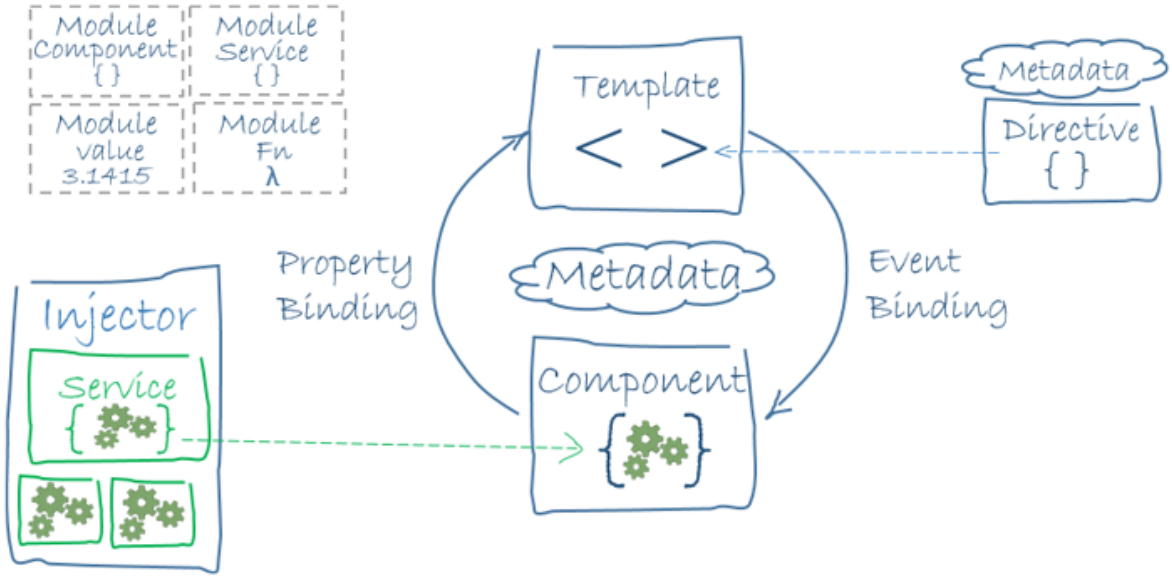


Figura 2.16 Diagrama de elementos Angular [63]

La idea es definir componentes ligados a “*Templates*” que utilizan HTML para renderizarse en el DOM; estas entidades se comunican dinámicamente a través de “*Bindings*” con la finalidad de cargar datos y escuchar eventos. Si un pedazo de código implementa una lógica reutilizable o extensa se utiliza un “*Service*”, un objeto que no debe ser renderizado, sino que “inyecta” lógica mediante inyección de dependencia. Finalmente, un módulo representa una colección de objetos con un objetivo específico, permitiendo escalar el proyecto.

Componentes FE

Los componentes son estructurados jerárquicamente y comparten datos y funcionalidades a través de decoradores de entrada y salida; también se utilizan enrutadores o “Angular Routing” para mostrar diferentes pantallas en la misma SPA (*Single Page Application*).

La figura 2.17 ilustra la jerarquía de componentes propuestos para la interfaz gráfica. Nótese que un primer enrutador solo da acceso a la pantalla de ingreso y a la principal; en cambio, el segundo enrutador provee acceso a las pantallas de administración, reportes y gráficos, pero “enmarcado” en un componente con controles y embebido dentro de la vista principal.

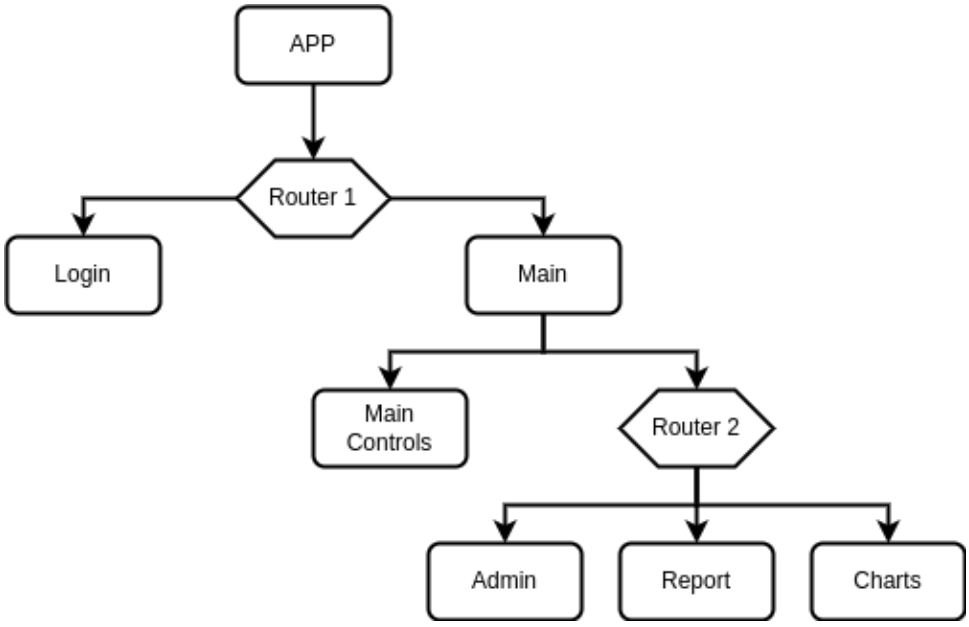


Figura 2.17 Estructura de componentes de FE app

Ngx-Charts

Es un *framework* declarativo para Angular [64] que le dará al proyecto la capacidad de generar gráficas dinámicas y responsivas a partir de los datos consultados por la *app*. Se lo inserta dentro del componente que lo requiera, declarando los atributos necesarios de entrada (datos, colores, ejes, etc.). Hay que considerar que los datos de entrada deben ser arreglos de objetos JS con un formato específico de cada gráfico, por lo que resulta una buena idea utilizar un servicio que ajuste el formato según la necesidad.

Servicios y RxJS

RxJS es una librería de JavaScript que se utiliza para manejar lógica asíncrona [65] empleando un patrón observador. Se usa en el proyecto para que los diferentes componentes puedan “reaccionar” a cambios de estado, especialmente de datos consultados del BE.

Se propone un servicio cuya función sea consultar datos del *Backend* y manejar toda la lógica referente a HTTP, devolviendo un observable de RxJS para que los componentes dependientes se puedan suscribir.

También se utilizará un servicio destinado a la restructuración de datos para que puedan ser recibidos y graficados por Ngx-Charts.

Angular Material

Es una herramienta de Angular que provee componentes que siguen la especificación *Material Design* [66]. Estos componentes serán insertados dentro de la aplicación para proveer una UI de manera declarativa.

Se utilizarán entre otras:

- **Barra de Cajón Lateral**: para navegar en la app.
- **Tarjetas**: para envolver componentes.
- **Botones**: para realizar acciones como consultas.
- **Tablas**: para organizar y mostrar los datos consultados.

La figura 2.18 muestra un ejemplo tomado de Angular:

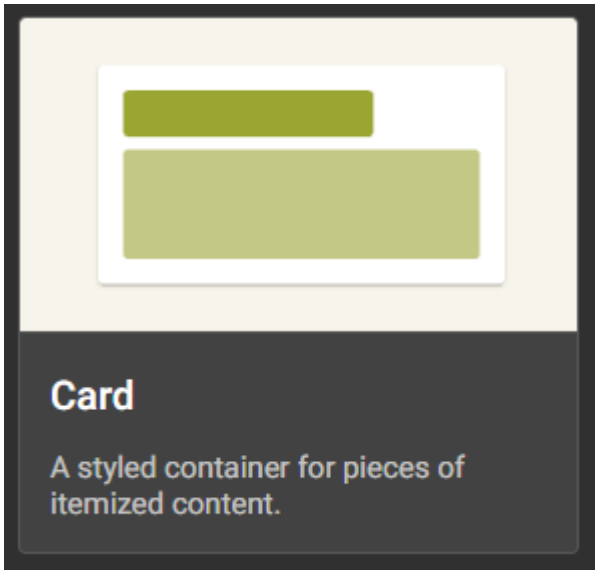


Figura 2.18 Componente Tarjeta mostrado como una tarjeta por angular [67]

2.3.3.2 API REST

Para este componente se propone la creación de una Web API que escuche en un servidor a diferentes clientes que utilizan la interfaz para realizar peticiones. “Disparado” por una petición, el servicio se encargará de consultar la base de datos para completar la acción en la capa de Persistencia y devolver el resultado al cliente. La figura 2.19 muestra el diagrama de comunicaciones que ilustra este proceso.

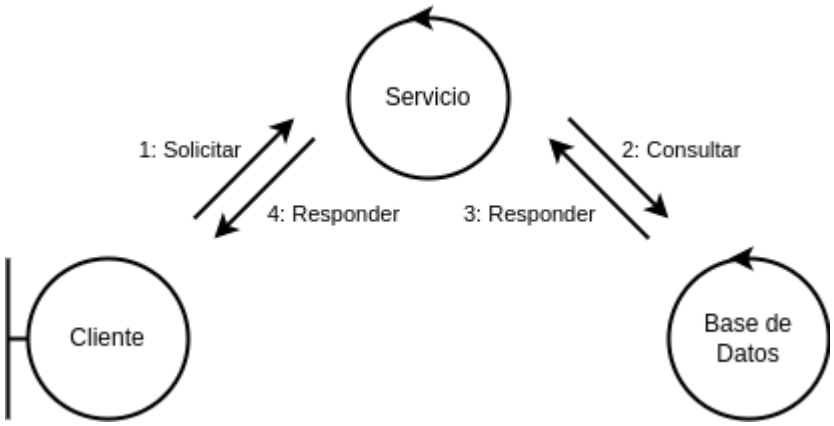


Figura 2.19 Diagrama de comunicaciones del servicio

El servicio web será implementado utilizando un API de .NET Core, un *framework* de Microsoft multiplataforma gratuito y de software abierto que utiliza C# como lenguaje de programación. Se seguirá la arquitectura recomendada por .Net que se presenta en la figura 2.20.

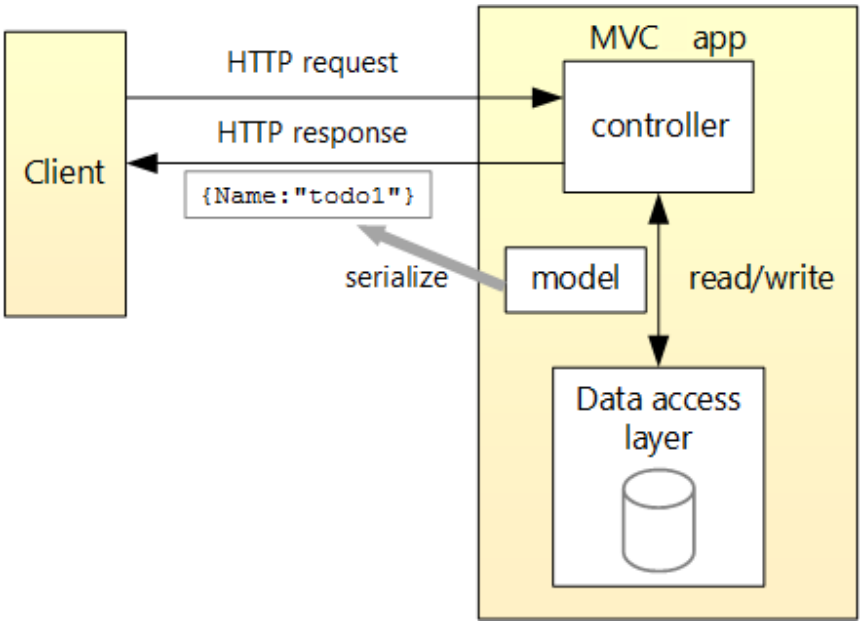


Figura 2.20 Arquitectura sugerida por API .NET [68]

El diagrama muestra un cliente consultando información desde la app BE, la cual se deriva de una arquitectura MVC sin vista, adecuada para el servicio web. Otras funcionalidades deberán ser inyectadas como servicios al controlador.

Posibles Respuestas

El interfaz será capaz de devolver respuestas 200 para operaciones correctas, 201 si algún elemento fue creado, 404 si no se encuentra el recurso solicitado y 500 para indicar errores internos del servidor en caso de ser necesario.

Siguiendo los principios de REST, no se mantendrán sesiones en el servidor y como cuerpo de mensaje únicamente se aceptarán objetos JSON correspondientes a los modelos establecidos según la base de datos.

Entity Framework

Para el "mapeo" y acceso a los datos se utiliza un ORM de C# conocido como *Entity Framework* (EF), éste se encargará de abstraer la comunicación con la capa de Persistencia del sistema.

Considerando que la base de datos se conectará con dos o más programas diferentes, se utiliza el principio de "*Database First*" donde primero se crea la base y luego se establece el mapeo y la abstracción del código *Backend* en vez de "*Code First*" donde EF toma control de la creación, migración y relaciones de la base.

2.3.3.3 Base de Datos

Se implementa un modelo relacional mediante el motor SQL Server y SSMS (*SQL Server Management Studio*). Siguiendo la especificación de requerimientos se define:

- Una entidad para representar los diferentes sensores en campo, cuya tarea es la medición de condiciones ambientales.
- Una entidad que represente la infraestructura física donde los sensores serán desplegados.

Estas entidades son claves para la persistencia de datos, pero aún no representan un modelo relacional capaz de cumplir con los atributos de calidad del sistema, por lo que será necesario cierto nivel de normalización donde se establece que:

- Un sensor puede ser de varios tipos, por lo que se define una entidad “tipo de sensor” que establece las características de un sensor comercial.
- Un sensor puede realizar varias mediciones, éstas deben ser almacenadas en una tabla de registros con su respectiva fecha de ingreso y unidad de medida.
- Una finca florícola se compone de varios bloques, éstos son los invernaderos que contienen las flores, por lo que se requiere de una entidad finca y una entidad bloque.
- Un bloque puede tener varios sensores dependiendo de la necesidad, relacionando finalmente la infraestructura y los dispositivos.

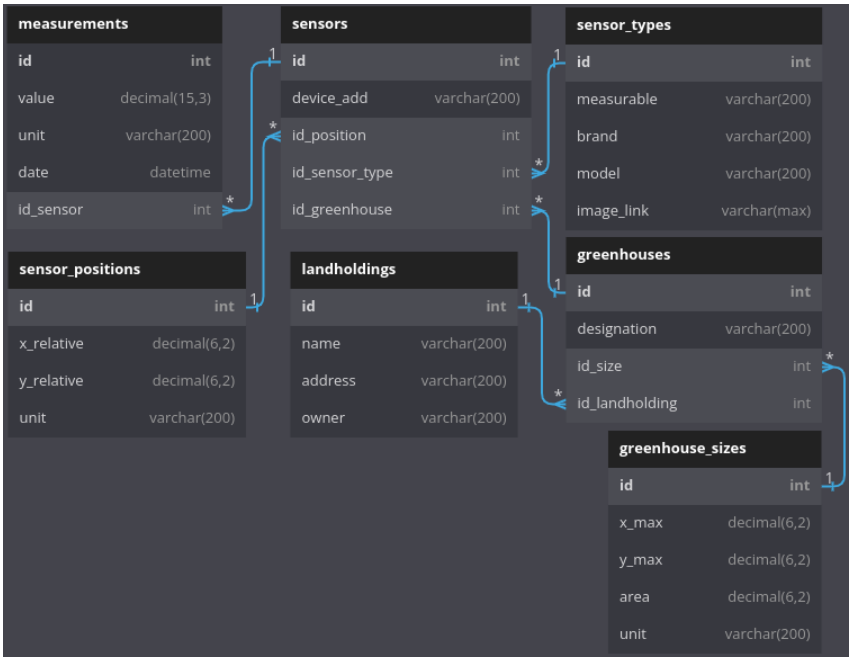


Figura 2.21 Diagrama Relacional

El diagrama relacional de la base de datos se aprecia en la figura 2.21.

2.3.3.4 Recolector de Datos

Este componente se implementará a través de un programa a desarrollar en Go. Este lenguaje provee un balance entre herramientas de alto nivel y rendimiento necesario para las necesidades del componente. El programa deberá ser capaz de comunicarse mediante MQTT con el *broker* y conectarse a la base de datos del proyecto, así como la de soportar el tráfico generado por los diferentes dispositivos IoT.

Arquitectura del Recolector

Go no propone ninguna arquitectura específica, pero en la industria es común observar el desarrollo de aplicaciones de servidor en Go utilizando una arquitectura hexagonal. Esta se basa en separar la lógica del dominio en un "*core*" o núcleo del hexágono, y una capa exterior que contiene lógica dependiente de otras API. Estas capas son "conectadas" a través de conceptos llamados puertos y adaptadores, que utilizan inyección de dependencias e inversión de control para que la capa exterior pueda cambiar con las herramientas externas sin modificar la lógica interior. Generalmente la capa exterior se divide en *Frameworks* o tecnologías conducidas, si el programa es el que inicia la interacción, y conductoras, si la interacción es iniciada por el exterior.

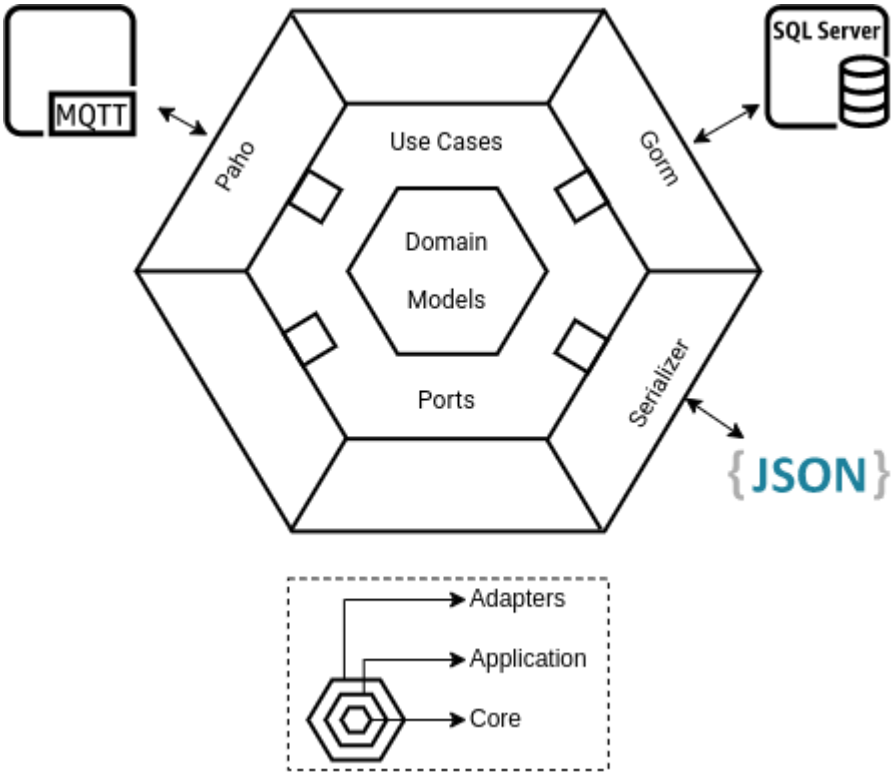


Figura 2.22 Arquitectura Hexagonal del Recolector

La figura 2.22 ilustra la arquitectura del programa, a continuación se describen sus partes.

En la capa de Dominio se tiene la lógica de modelamiento de datos, así como las funciones que adecúan la información que llega desde el exterior sin depender de otras capas.

En la capa de Aplicación se implementará la utilización de las funciones internas y de las tecnologías conducidas según el caso de uso, generando una API que las tecnologías conductoras puedan usar.

En la capa externa se define un cliente MQTT como conductor encargado de la comunicación con el *broker* mediante la librería Paho; también se tiene un serializador que toma la información binaria provista desde afuera y la transforma en un formato compatible con el modelo implementado en el núcleo; y, finalmente una ORM como tecnología conducida para la comunicación con la base de datos.

De esta manera se prepara para la escalabilidad del programa ya que nuevos elementos pueden ser agregados a la arquitectura sin modificar código de otros.

Cliente MQTT

Paho proporciona una API para el manejo de mensajes y conexiones MQTT mediante un cliente. Dado que pueden existir varios clientes en una misma aplicación, se define una estructura “recolector” que incluye un cliente y las configuraciones necesarias para la conexión y recolección de publicaciones; entre ellas:

- ID de Cliente que representa el nombre del cliente MQTT.
- QoS o calidad de servicio, puede ser “0”, “1”, “2”.
- Suscripciones y sus funciones “*handlers*”.

Tras instanciar un colector, el adaptador llama a la función de recolectar que recibirá el puerto inyectado para utilizar la API de la capa Aplicación correspondiente al caso de uso que este adaptador conduce. Dentro de esta función también se suscribe al cliente del recolector emparejando el tópico MQTT y su respectivo “*handler*” como lo indica Paho.

GORM

GORM es un ORM que abstrae la comunicación con la base de datos para Go. Para el recolector se lo implementa como un adaptador que recibirá modelos a ser buscados o creados en la base; estos modelos estarán nombrados según el estándar de GORM y así no se requerirá especificar nombres de tablas y propiedades.

La función de buscar recibirá un modelo genérico por referencia, un parámetro y un valor, siendo estos últimos la propiedad que determinará qué entrada es devuelta y el valor a empatar. Finalmente se devuelve un error de haber existido uno.

La función de crear recibe únicamente la referencia de un modelo genérico, utiliza el mapeo de GORM para transformar el modelo genérico en uno específico según el valor recibido e insertarlo en la base de datos. Al final se devuelve un error si se generó alguno.

Cabe mencionar que este adaptador, al ser conducido, es inyectado en la capa de Aplicación para ser utilizado según el caso de uso.

Serializador JSON

Es el adaptador responsable de convertir los cuerpos de los mensajes binarios en un formato para Go, especialmente un modelo de la capa Núcleo. Dado que el tráfico planteado para MQTT serán binarios generados a partir de formatos JSON, el adaptador implementará un “Marshal” de JSON proporcionado por Go. Se basará en dos funciones:

- Un codificador que recibe referencias de modelo o mapas y devuelve binarios y errores si son generados.
- Un decodificador que recibe tanto binarios como referencias de objeto para devolver un error si se da el caso.

Entre las razones por la cuales se implementa un adaptador para la serialización y no se la realiza directamente en capa de Aplicación destacan:

- Separa mejor la lógica y provee reutilización de código.
- Permite un mejor escalamiento pues se pueden aumentar otros adaptadores que comparten el puerto de serialización y así soportar otros formatos en un futuro como BSON o propietarios de fabricantes de dispositivos IoT.
- Mejora la modularidad del programa aislando el uso de la librería de JSON.

Contenerización

Mediante este concepto, una aplicación se empaqueta dentro de un contenedor, que básicamente es un conjunto de programas que aislan y proveen un entorno de ejecución para dicha *app*. Es similar a una máquina virtual pero mucho más ligera, pues el motor de contenedores abstrae y comparte el Kernel del computador anfitrión en vez de virtualizar otro. Dentro del proyecto, contenerizar el recolector es importante porque:

- Facilita el despliegue de nuevas instancias del programa, de esta manera se puede distribuir la carga de la recolección evitando “cuellos de botella”.
- Mejora la portabilidad, consecuentemente la recolección puede ser implementada tanto en nube como en “niebla” como en premisas de la finca o empresa.
- Aumenta la escalabilidad del sistema, pues el componente se independiza y abre paso a arquitecturas más complejas como la de microservicios.

Para este propósito se utilizará Docker, una herramienta desarrollada en Go muy popular en la actualidad que provee no solo la capacidad de “correr” contenedores para el desarrollo, sino también construir nuevos contenedores mediante un archivo “Dockerfile”.

2.3.3.5 Broker MQTT

Este componente será implementado utilizando Mosquitto MQTT, un servidor *broker* desarrollado y mantenido por Eclipse y Cedalo [69].

Se lo utiliza en el presente proyecto por sus características:

- Es de código abierto y tiene una gran comunidad.
- Es desarrollado utilizando C y compilado directamente a binarios lo que implica un alto rendimiento necesario para servir mensajes MQTT.
- Es multiplataforma y puede ser contenerizado; Mosquitto provee instaladores para diferentes OS, así como el código fuente mediante un repositorio de Github.
- Sigue las especificaciones de MQTT en diferentes versiones para asegurar la interoperabilidad del sistema.

Árbol de Tópicos MQTT

La figura 2.23 muestra la estructura jerárquica de los tópicos del proyecto, donde se agrupan aquellos que generan datos ambientales dentro de la rama de sensores para poder escalar el sistema creando otras ramas.

De esta manera se define que cualquier dato ambiental será publicado en un tópico:

root / *sensor* / <*variable*> / <*device*>

Donde *root* y *sensor* son nodos fijos mientras que *variable* dependerá del fenómeno físico que el sensor mida y *device* dará una identificación única a cada dispositivo sensor.

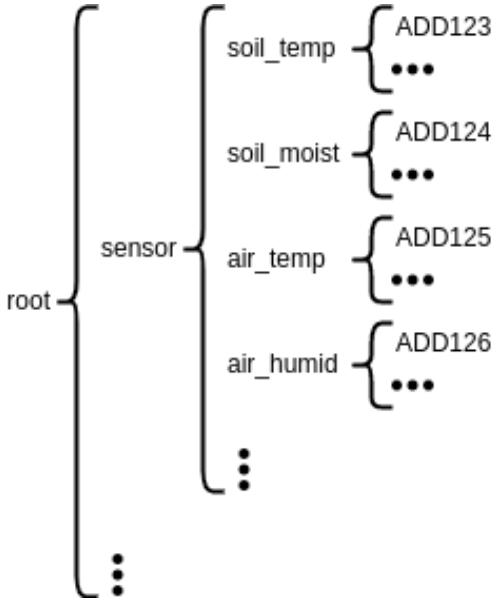


Figura 2.23 Árbol de Tópicos MQTT

El recolector se suscribirá a todos y cada uno de los nodos finales utilizando "*wildcards*". El diagrama de actividades de la figura 2.24 detalla la comunicación entre los clientes y el *broker*.

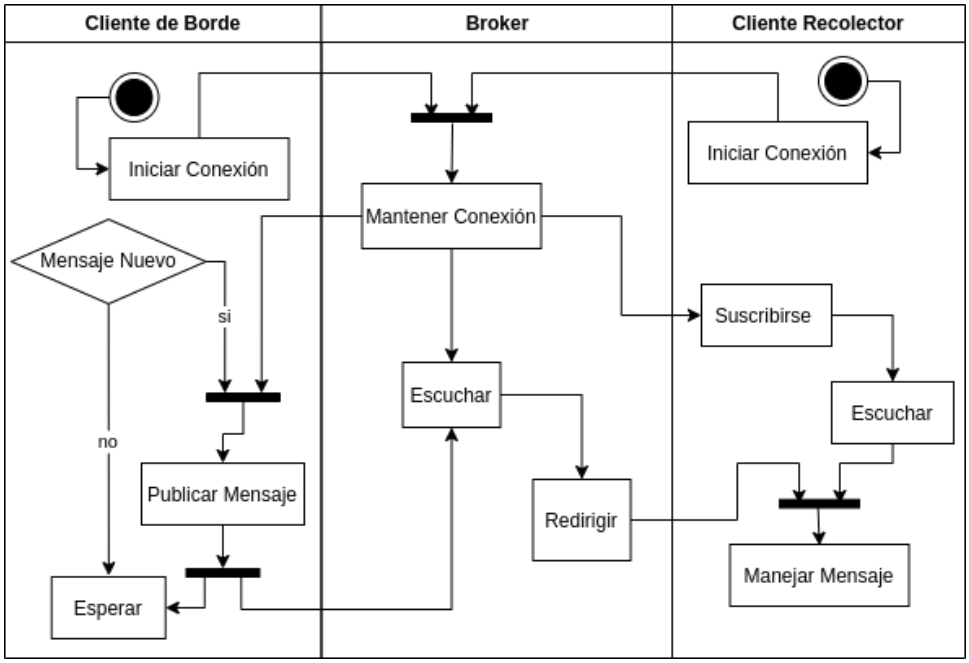


Figura 2.24 Diagrama de Actividades Pub/Sub

2.3.3.6 Dispositivos Físicos

En el borde se tienen los dispositivos capaces de interactuar con el entorno físico, así como comunicarse con el resto del sistema.

Sensores

Siguiendo los requisitos funcionales obtenidos y en base al alcance del proyecto, se propone el uso de los siguientes sensores comerciales:

- **DS18b20:** Es un sensor digital de *Dallas Semiconductor* que mide temperatura; se lo utiliza comúnmente en proyectos IoT por su versatilidad, su amplio rango, linealidad y protocolo de comunicación “1-Wire” (ver figura 2.25).

Figura 2.25 DS18b20 en Sonda [70]

- ***Capacitive Soil Moisture Sensor 1.2*:** Es un sensor genérico analógico diseñado para medir la humedad de un suelo mediante una sonda. A diferencia de su contraparte resistiva, este sensor mide la variación en capacitancia del suelo, por lo que no necesita exponer metales, reduciendo su corrosión y la contaminación del ambiente (ver figura 2.26).

Figura 2.26 Sensor Capacitivo de Humedad [71]

- **DHT21:** Es un sensor digital diseñado para medir tanto la temperatura como la humedad de gases, comúnmente aire. Se comunica a través de *Single-Bus* y requiere entre 3.3 y 5 voltios para su funcionamiento (ver figura 2.26).

Figura 2.27 DHT21 [72]

ESP32

Se utilizará un embebido ESP32 Devkit1c-V4 para el desarrollo (ver figura 2.28); éste no solo provee herramientas necesarias para cubrir el alcance del proyecto, sino que también presenta funcionalidades que permitirán la escalabilidad del sistema:

- Conexión Inalámbrica a través de Wi-Fi y *Bluetooth*, así como librerías para el desarrollo IoT.
- GPIO que incluyen entradas analógicas y proveen funcionalidades de ADC.
- Hardware dedicado a la seguridad de la comunicación.
- Capacidad de red tanto en estrella como en malla para actuar únicamente como *Gateway* o “mota”.
- SDK y una comunidad activa para el desarrollo de proyectos embebidos.

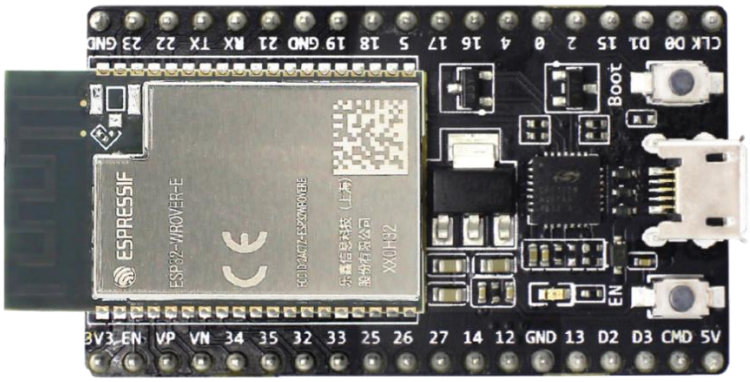


Figura 2.28 ESP32 DevKit1C [73]

Esquema de Conexión

Se conectan los sensores (según la recomendación) a una fuente de voltaje, tierra y a las entradas de propósito general del ESP32. Cabe mencionar que los sensores analógicos deberán conectarse a entradas que soporten la conversión ADC.

La figura 2.29 describe el esquema de conexión indicado.

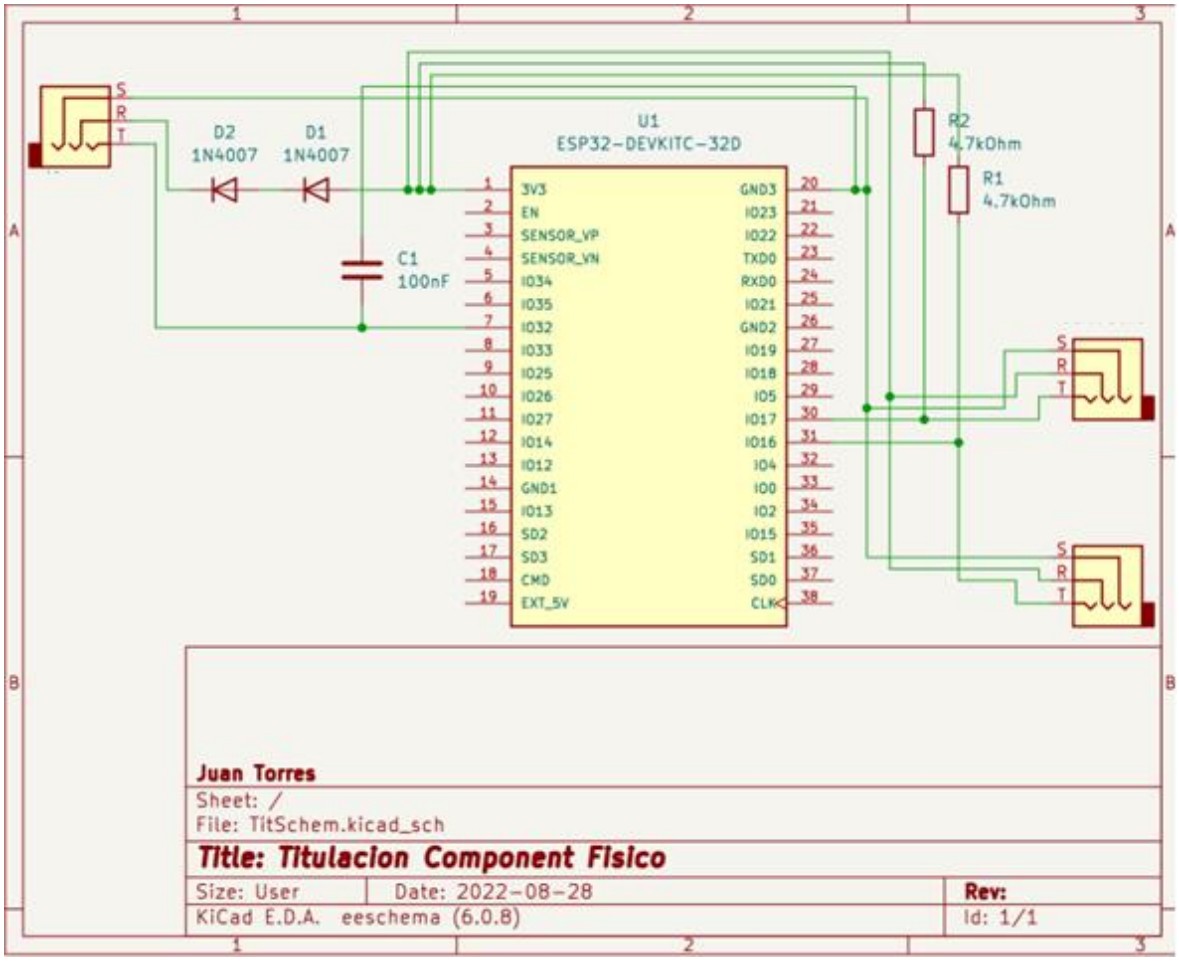


Figura 2.29 Esquema en KiCad

Software Embebido

Para el software de los dispositivos de borde se utiliza Platform.io, una extensión de VSCode que provee una plataforma de desarrollo uniforme para diferentes dispositivos embebidos, entre ellos el ESP32. De esta manera, no será necesario interactuar directamente con el SDK ESP-IDF, aumentando la portabilidad del software.

Debido a la naturaleza restringida del software embebido será necesario utilizar un lenguaje de bajo nivel como C o C++, así como evitar arquitecturas y diseños de software complejos y seguir especificaciones provistas por Espressif para el desarrollo de prototipos.

El diagrama de secuencia de la figura 2.30 muestra el comportamiento del software para la medición y publicación de variables ambientales, abstrayendo el comportamiento interno de la API de sensor y del cliente.

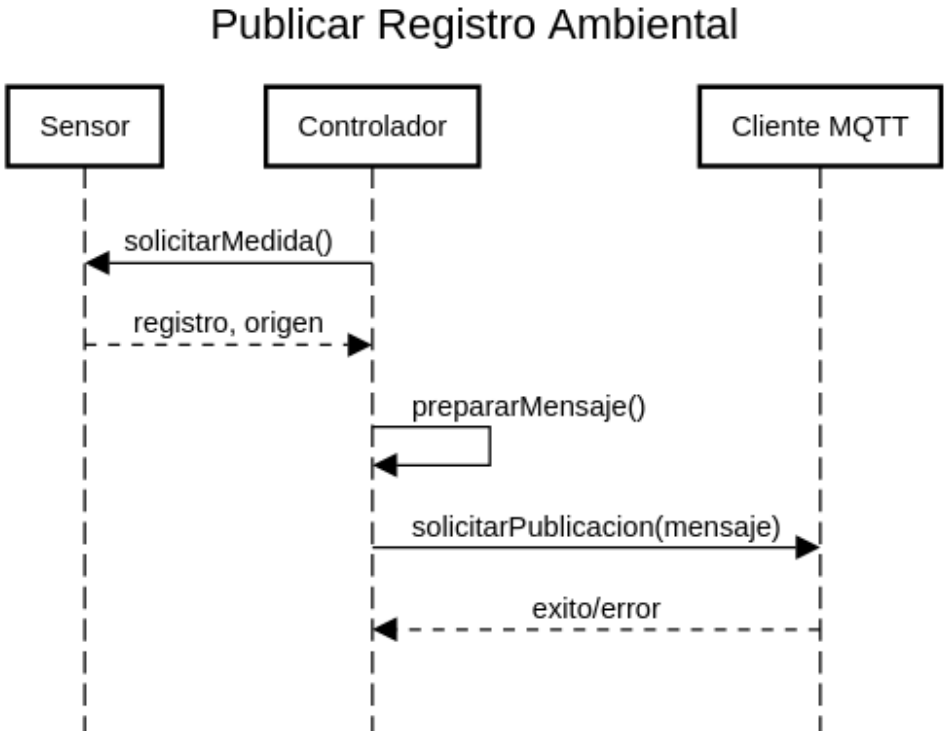


Figura 2.30 Diagrama de Secuencia para Publicación de Registros

Esta secuencia ocurrirá una vez cada cierto tiempo por cada sensor y gracias a RTOS pueden ocurrir de manera concurrente y hasta en “tiempo real” para tareas críticas, en caso de ser necesario.

2.3.4 TAREAS DE JIRA

A continuación, se utiliza el diseño propuesto para crear las tareas que subdividen a las historias de usuario como indica la metodología utilizada.

2.3.4.1 Tareas para TIT-3: Medir Condiciones Ambientales

Para cumplir con este caso de uso (ver figura 2.31) se plantean dos tareas: armar un prototipo con el hardware propuesto e implementar el software embebido utilizando Platform.io.

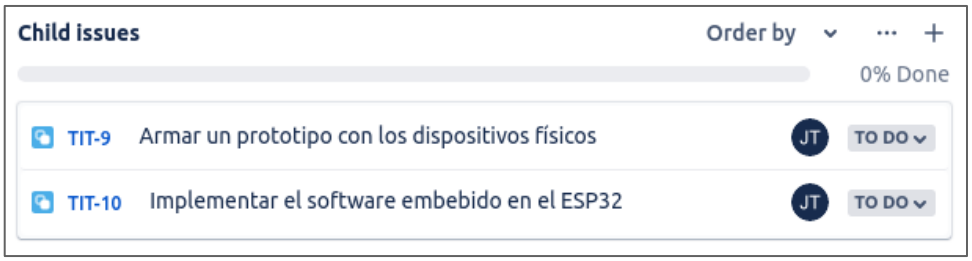


Figura 2.31 Tareas de TIT-3

2.3.4.2 Tareas para TIT-4: Guardar Información Ambiental

Esta historia de usuario es más compleja que la anterior y requiere de algunas tareas, descritas en la figura 2.32, para su cumplimiento.

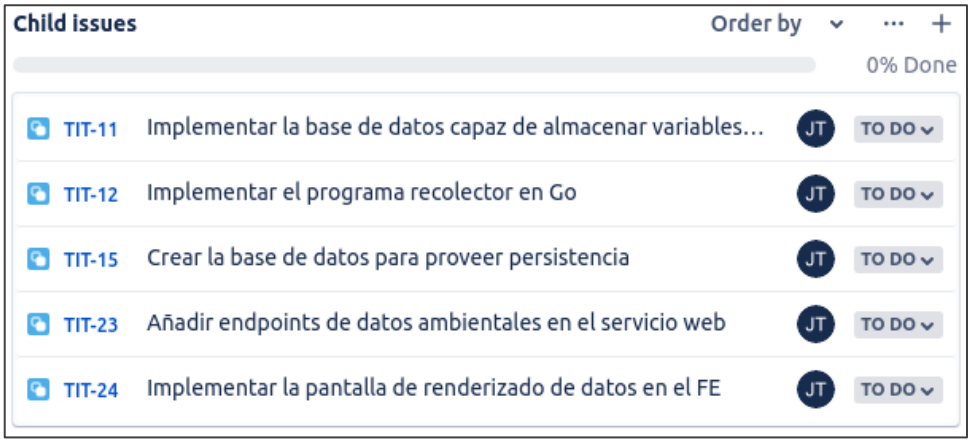


Figura 2.32 Tareas de TIT-4

2.3.4.3 Tareas para TIT-5: Localizar Dispositivos Físicos

En cuanto a la localización, se necesitarán dos tareas: implementar entidades necesarias en la base de datos y crear *endpoints* en el servicio para proveer salida (ver figura 2.33).

Figura 2.33 Tareas de TIT-5

2.3.4.4 Tareas para TIT-6: Añadir Nuevos Dispositivos

En la figura 2.34 se proponen dos tareas para este caso de uso.

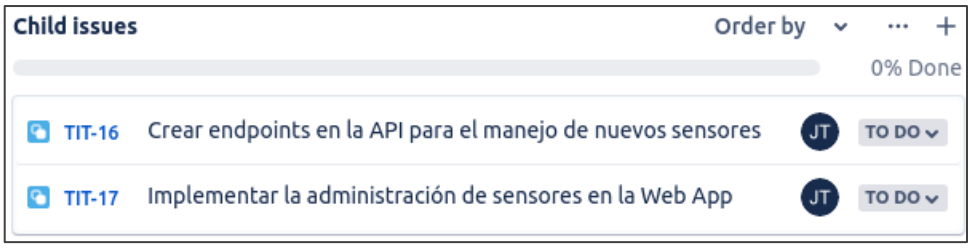


Figura 2.34 Tareas de TIT-6

2.3.4.5 Tareas para TIT-7: Administrar Modelos de Infraestructura

Para relacionar la infraestructura de la finca y los dispositivos se proponen las tareas que se indican en la figura 2.35.

Figura 2.35 Tareas de TIT-7

2.3.4.6 Tareas para TIT-8: Proveer Gráficos

Finalmente, para mostrar gráficos al usuario se implementará el servicio que adapte los datos al formato requerido por NGX-Charts y la pantalla o página correspondiente (ver figura 2.36).

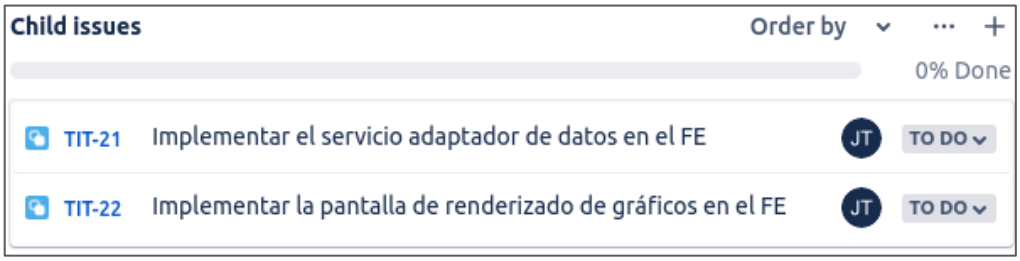


Figura 2.36 Tareas de TIT-8

2.4 IMPLEMENTACIÓN DEL SISTEMA

En esta sección se presenta la información más relevante de la implementación del proyecto mediante el cumplimiento de las tareas propuestas en la sección anterior.

2.4.1 AMBIENTE DE DESARROLLO

Para la implementación del proyecto será necesario preparar el entorno donde se va a desarrollar el software, mediante la instalación y configuración de las herramientas tecnológicas necesarias.

2.4.1.1 Resumen de Herramientas de Desarrollo

En la tabla 2.2 se resumen las herramientas instaladas y configuradas en el equipo del desarrollador.

Tabla 2.2 Herramientas de desarrollo

Hardware	i7-4790, 16GB RAM, WD SSD, GTX-1060	
Sistema Operativo	Arch Linux	Windows 10 (VM)
IDE o Editor de Texto	VS Code	Visual Studio 2022
Virtualización	Docker + Docker Desktop	VirtualBox
Imágenes	SQL Server, Mosquitto, Alpine	W10
Compilación e Interpretación	Node.js, Go gc, Platform.io CMake	.NET CLR
SDK	ESP-IDF, Angular CLI, Go kit	.NET SDK
Pruebas	Postman, Arduino IDE, Chrome	Swagger
Versionamiento y Repositorio	Git, GitHub, DockerHub	

Cabe mencionar que se utiliza una máquina virtual en Virtualbox con imagen de Windows 10 para el desarrollo con herramientas de Microsoft; si bien .NET Core es multiplataforma,

resulta más sencillo utilizar Windows como entorno para aplicaciones que utilicen este *Framework*.

2.4.1.2 Primera Actualización del Tablero Kanban

Una vez configurado el entorno de desarrollo se procede a actualizar el tablero Kanban para indicar los diferentes casos de uso que se encuentran en desarrollo. Debido a las dimensiones del tablero, éste se lo presenta en el ANEXO A.1.

2.4.2 BASE DE DATOS

El primer componente del sistema a implementar es la base de datos; cumpliendo las tareas relacionadas a la persistencia se podrá desarrollar el resto de las tecnologías.

2.4.2.1 Segunda Actualización del Tablero Kanban

En primera instancia, se mueven tareas de cualquier caso de uso, que correspondan a la persistencia de datos, a la columna de "en proceso". El resultado se muestra en el ANEXO A.2.

2.4.2.2 Implementación del Modelo Relacional

Se escribe un *script* de SQL que cumple con las tareas de persistencia, de esta manera la base de datos puede ser levantada automáticamente una y otra vez durante el desarrollo del proyecto.

El *script* consta básicamente de una sección donde se crea y utiliza la base de datos dentro del motor SQL, luego se crean las tablas con sus propiedades y relaciones, y finalmente se añaden algunas entradas de prueba.

En la figura 2.37 se presenta una parte del código utilizado, el resto del *script* se muestra en el ANEXO B.

testdb > db.sql

```
1  create database TitDB;
2  go
3  use TitDB;
4  go
5  create table landholdings(id int primary key identity(1,1),
6   landholding_name varchar(50) not null,
7    landholding_address varchar(50),
8     landholding_owner varchar(50));
9  go
```

Figura 2.37 Fragmento de script SQL

2.4.2.3 Levantamiento del Motor Base de Datos en Docker

Con el objetivo de instanciar un contenedor con SQL Server primero se utiliza DockerHub para encontrar su imagen; seguido se inicia el contenedor con los parámetros de configuración que ésta requiera según la documentación. Finalmente, se crea la base de datos utilizando el *script* previamente descrito.

Docker Desktop permite revisar imágenes volúmenes y contenedores a través de su interfaz gráfica como se muestra en la figura 2.38.

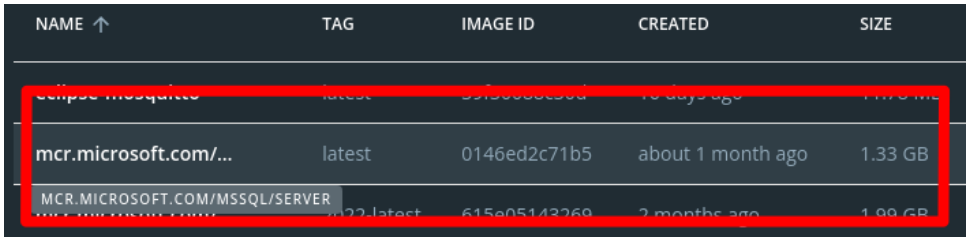


Figura 2.38 Imagen de SQL Server en Docker Desktop

2.4.3 RECOLECTOR IOT Y MOSQUITTO

En esta sección se describe la implementación del programa recolector y el levantamiento del *broker* MQTT, ambas acciones mediante el cumplimiento de las tareas relacionadas a dichos propósitos.

2.4.3.1 Tercera Actualización del Tablero Kanban

Siguiendo con el flujo de trabajo, se mueven las tareas cumplidas en la fase anterior hacia la etapa de pruebas, así como también el nuevo grupo de tareas (correspondientes al componente recolector y *broker* MQTT) hacia la columna de desarrollo. El tablero resultante se muestra en el ANEXO A.3.

2.4.3.2 Levantamiento y Configuración del Broker MQTT

Mosquitto se puede instalar directamente sobre el sistema operativo, pero también se puede desplegar mediante un contenedor aprovechando las herramientas de Docker. Para esto es necesario encontrar su imagen en DockerHub, revisar su documentación y seleccionar la configuración más conveniente para el proyecto.

En este caso se importa como volumen un archivo de configuración “mosquitto.conf”, que se lo crea externamente con los parámetros requeridos por el *broker* del sistema como se muestra en la figura 2.39.

```
1  listener 1883
2  allow_anonymous true
3
4  persistence true
5  persistence_location /var/lib/mosquitto/
```

Figura 2.39 Configuraciones en mosquitto.conf

Al ser un entorno de desarrollo, el servidor es configurado con parámetros básicos; para levantarlo en producción probablemente será necesario añadir más líneas siguiendo la documentación de “mosquitto.conf”.

2.4.3.3 Programa Recolector

Se estructura el código según la arquitectura propuesta en el diseño; el punto de entrada principal del programa está localizado dentro de “cmd”. También se añaden dos directorios que contienen archivos de prueba, “testdb” para la base de datos y “testmqtt” para el *broker* MQTT, tal como se puede visualizar en la figura 2.40.

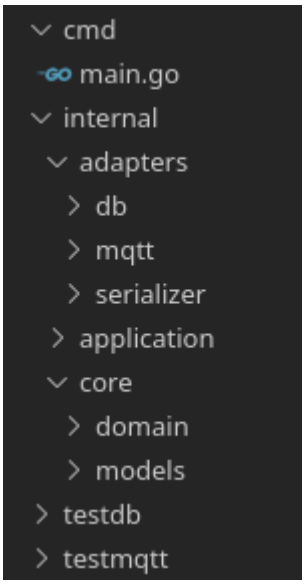


Figura 2.40 Estructura de directorios del recolector

Una vez completado el programa, éste podrá ser encontrado en el ANEXO C.

Núcleo

Dentro del núcleo de la aplicación se escribe la lógica de dominio y los modelos para representar datos.

La lógica de dominio utiliza un "*struct*" para exportar sus funcionalidades a capas exteriores, esto lo hace a través de dos métodos (*receiver function* en Go):

- El primer método permite completar una medición. Ésta recibe modelos de la medida y el sensor de origen, añade las propiedades faltantes y devuelve el modelo completo.
- El segundo método transforma una cadena de tópico en un arreglo o "*slice*", que comprueba que siga la estructura del árbol propuesto en diseño y devuelve un error en caso de haber alguno.

Nuevas funcionalidades de lógica de dominio se deberán añadir como nuevos métodos. La figura 2.41 muestra un ejemplo correspondiente al primer método.

```
func (l Logic) RegComplete(s models.Sensor,
    m models.Measurement) (models.Measurement, error) {
    if s.ID <= 0 {
        err := errors.New("sensor id must be a valid id")
        return models.Measurement{}, err
    }
    m.MeasurementDate = time.Now()
    m.IDSensor = s.ID
    return m, nil
}
```

Figura 2.41 Fragmento de código del núcleo

Los modelos dan estructura a los datos; se los implementa a través de un *struct* correspondiente a las mediciones o "*Measurement*" y otro correspondiente a los sensores o "Sensor". Nuevos modelos se deberán escribir dentro de esta capa y sin depender de librerías externas ya que se encuentra dentro del *core*.

```go
type Measurement struct {
    ID               int
    MeasurementValue float32
    Unit             string
    MeasurementDate  time.Time
    IDSensor         int
}
```

Figura 2.42 Ejemplo de modelo en el núcleo

Capa de Adaptadores

En esta capa se implementa todo el código dependiente de librerías externas. Se tienen tres adaptadores: “db” para la base de datos, “mqtt” para el cliente MQTT y “*serializer*” para el aplanamiento de datos en JSON.

El adaptador de base de datos se implementa mediante un *struct* que a su vez debe implementar el puerto de persistencia a través de una interfaz o “*interface*”. Su función constructora recibe una cadena de conexión correspondiente a la instancia de la base dentro del motor SQL y devuelve un error si no se pudo conectar. El adaptador implementa funciones del puerto para crear y buscar modelos dentro de la base; cabe mencionar que las funciones reciben modelos por su referencia (*pointers*) siguiendo la estructura de GORM (ver figura 2.43).

```go
func NewAdapter(connectionString string) (*Adapter, error) {
    db, err := gorm.Open(sqlserver.Open(dsn), &gorm.Config{})
    if err != nil {
        log.Fatalf("db connection failure: %v", err)
    }

    return &Adapter{db: db}, nil
}
```

Figura 2.43 Fragmento de código adaptador GORM

El adaptador serializador JSON implementa la interfaz correspondiente al puerto de serialización mediante un *struct* con dos métodos: *Decode* recibe una entrada binaria y el puntero al modelo donde van los datos; realiza el “*Unmarshalling*” y devuelve un error si alguno es generado. Por el contrario, *Encode* recibe el modelo y lo transforma en JSON como cadena binaria e igual devuelve un error si falla el proceso (ver figura 2.44).

```go
func (jsona Adapter) Decode(input []byte, model interface{}) error {
    if err := json.Unmarshal(input, model); err != nil {
        return errors.Wrap(err, "serializer.Redirect.Decode")
    }
    return nil
}
```

Figura 2.44 Fragmento de código adaptador Serializador

El adaptador MQTT recibe el puerto de la API para utilizar el caso de uso que necesite ya que es de tipo conductor. Éste abstrae la lógica del recolector mediante un *struct* al cual lo instancia y luego llama al método de recolectar para finalmente esperar hasta que termine la ejecución del programa (ver figura 2.45).

```go
// Run starts the MQTT client collector
func (mqtta Adapter) Run(settings []byte, connStr string) {
    h := newHub(settings, connStr)
    h.Collect(mqtta.api)
    var in string
    for {
        fmt.Scanln(&in)
        if in == "exit" {
            log.Println("finishing client")
            break
        }
        time.Sleep(time.Second)
    }
}
```

Figura 2.45 Fragmento de código adaptador MQTT

Dentro de la abstracción provista por el recolector se implementa el cliente MQTT con las configuraciones recibidas como parámetros. Cabe mencionar que la configuración sensible o privada es insertada como variable de entorno y la común mediante un archivo JSON. El método de recolectar toma el cliente instanciado y lo suscribe al tópico de recolección utilizando una función de *callback* que pasa tópico y mensaje hacia la API del adaptador.

La figura 2.46 muestra un fragmento de este código.

```go
func (h hub) Collect(api application.APIPort) {
    h.handler = func(client paho.Client, msg paho.Message) {
        err := api.CreateMeasurement(msg.Topic(), msg.Payload())
        if err != nil {
            log.Print(err)
        }
    }
    if token := h.client.Subscribe(h.settings.Topic,
        h.settings.QualityOfService,
        h.handler); token.Wait() && token.Error() != nil {
        log.Fatal(token.Error())
    }
}
```

Figura 2.46 Fragmento de código recolector

Capa de Aplicación

En esta capa se implementan tanto los puertos como la API para los casos de uso del programa. Los puertos son simplemente interfaces que deben ser implementados por los adaptadores para desacoplar las capas; de esta manera se pueden instanciar varios adaptadores que compartan puerto. Se implementan tres interfaces: una para la base de datos con métodos para crear y buscar, una para el serializador con métodos de codificar y decodificar, y una para la API de aplicación que provee un punto de entrada a los adaptadores conductores (ver figura 2.47).

```go
// APIPort is the technology neutral
// port for driving adapters
type APIPort interface {
    CreateMeasurement(topic string, payload []byte) error
}

// DbPort is the port for a db adapter
type DbPort interface {
    Create(modelP interface{}) error
    Find(parameter string, value string, modelP interface{}) error
}

// Serializer is the por for a marshaller adapter (json, bson, msgpa
type SerializerPort interface {
    Decode(input []byte, model interface{}) error
    Encode(input interface{}) ([]byte, error)
}
```

Figura 2.47 Ejemplos de puertos en capa aplicación

Por otro lado, se tiene la API implementada mediante un *struct* que a su vez implementa su puerto correspondiente. El caso de uso donde se recibe un dato ambiental para ser registrado se implementa mediante un método que recibe tópico y mensaje, seguido se extrae el dispositivo publicador del tópico y se lo identifica en la base de datos. Luego se decodifica el mensaje, se completa el modelo de la variable ambiental y se lo inserta dentro de la base de datos. Si se produce algún error a través de este flujo, se lo devuelve como salida de función.

Archivos Docker y Directorios de Prueba

Para implementar el programa como contenedor se crea un archivo Dockerfile (mostrado en la figura 2.48) donde se especifica que:

- Se copia una imagen de Go sobre Alpine Linux
- Se actualiza el sistema base
- Se crea un directorio para el programa
- Se descarga las dependencias del programa como Paho o Gorm
- Se copia el código fuente dentro del contenedor
- Se copia un archivo “punto de entrada” en Shell, el cual se detallará a continuación.
- Se compila el programa en binarios para Linux.
- Se determina el comando para ser insertado dentro del punto de entrada en Shell.

```
COPY go.mod .
COPY go.sum .
RUN go mod download
COPY . .
COPY ./entrypoint.sh /usr/local/bin/entrypoint.sh
RUN /bin/chmod +x /usr/local/bin/entrypoint.sh

RUN go build cmd/main.go
RUN mv main /usr/local/bin/

CMD ["main"]
ENTRYPOINT [ "entrypoint.sh" ]
```

Figura 2.48 Fragmento de código Dockerfile

El archivo de punto de entrada es un *script* simple en Shell que retrasa la ejecución del comando insertado unos segundos. Esto con el propósito de asegurarse de que las dependencias del programa como *broker* y base de datos se encuentren “corriendo” normalmente. El script es mostrado en la figura 2.49.

```
#!/bin/sh

set -e
COMMAND=$@

sleep 10

exec $COMMAND
```

Figura 2.49 Código del script shell

Finalmente, se llenan los directorios de prueba con los archivos necesarios para orquestar el programa con sus dependencias. En el caso de *testmqtt* solamente se tiene un archivo de configuración para un *broker* de prueba, y en el directorio *testdb* se construye un contenedor que implementa un *script* para instanciar automáticamente una base de datos de prueba. La figura a continuación muestra la estructura de archivos para pruebas.

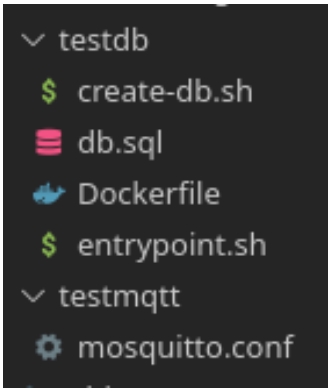


Figura 2.50 Directorios y archivos de prueba

2.4.4 DISPOSITIVOS DE BORDE

Para implementar los dispositivos de borde es necesario codificar el software embebido y crear el prototipo físico con los componentes electrónicos.

2.4.4.1 Cuarta Actualización del Tablero Kanban

Se mueven las tareas cumplidas en la fase anterior hacia la etapa de pruebas, seguido se mueven las tareas relacionadas a los dispositivos de borde hacia la columna de desarrollo. El tablero resultante se muestra en el ANEXO A.4.

2.4.4.2 Codificación del Software Embebido

Siguiendo con el diseño planteado, se utiliza Platform.io para inicializar el software. A continuación se descargan las dependencias y se las ubica siguiendo la estructura del marco de trabajo. En la figura 2.51 se muestra la estructura de archivos del software embebido.

El programa completo puede ser encontrado en el ANEXO D.

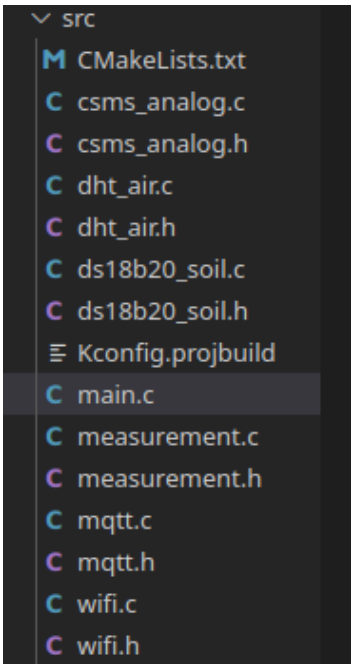


Figura 2.51 Estructura de Archivos del Software Embebido

La lógica correspondiente a la conexión del dispositivo como estación Wifi se encapsula en el archivo “wifi.c”, la cual básicamente se encarga de iniciar y mantener la conexión hasta capa tres de TCP/IP. Un fragmento de este código se muestra en la figura 2.52.

```
esp_event_handler_instance_t instance_any_id;
esp_event_handler_instance_t instance_got_ip;
ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                                                    ESP_EVENT_ANY_ID,
                                                    &wifi_event_handler,
                                                    NULL,
                                                    &instance_any_id));
ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,
                                                    IP_EVENT_STA_GOT_IP,
                                                    &wifi_event_handler,
                                                    NULL,
                                                    &instance_got_ip));
```

Figura 2.52 Fragmento de código correspondiente a WiFi

El archivo “mqtt.c” reúne el código necesario para iniciar la conexión con el *broker*, así como manejar eventos y proveer el cliente que necesita el controlador para enviar mensajes. Una parte de este archivo se muestra en la figura 2.53.

```
esp_mqtt_client_handle_t mqtt_client_start(void)
{
    esp_mqtt_client_config_t mqtt_cfg = {
        .uri = ESP_MQTT_BROKER_URL,
    };
    esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_register_event(client, ESP_EVENT_ANY_ID, mqtt_event_handler, NULL);
    esp_mqtt_client_start(client);

    return client;
}
```

Figura 2.53 Fragmento de código correspondiente a ESP-IDF MQTT

Para homogeneizar una interfaz de registros ambientales se utiliza un modelo codificado en “*measurement*.c”; todo sensor deberá utilizar esta interfaz para proporcionar mediciones al controlador. La figura 2.54 presenta el modelo implementado.

```
typedef struct  {
    const char *unit;
    const float *value;
} measurement_t;
```

Figura 2.54 Modelo de registro para el software embebido

Los archivos de sensores (*dht_air*.c, *csms_analog*.c y *ds18b20_soil*.c) agrupan la lógica necesaria para interactuar con cada sensor y proporcionar datos ambientales al controlador mediante el modelo de registros. En la figura 2.55 se muestra una parte de esta lógica.

```
const char * ds18b20_soil_get_measurement() {
    temperature = ds18b20_get_temp();
    ESP_LOGI(TAG, "ds temperature gotten: %f", temperature);
    ESP_LOGI(TAG, "measurement struct generated!");
    return parse_measurement(temperature, unit);
}
```

Figura 2.55 Fragmento de código correspondiente a lógica de sensor

Finalmente, el archivo principal o "*main*.c" sirve tanto como punto de entrada del programa, así como controlador. Aquí se da inicio a los diferentes subsistemas de RTOS, seguido se llama a las conexiones y a los sensores, se envía la información mediante el cliente MQTT instanciado, para luego entrar en el modo "sueño", ahorrando energía por algunos minutos hasta "despertar" y repetir el proceso. Se presenta una parte de este flujo en la figura 2.56.

```
ESP_ERROR_CHECK(esp_mqtt_client_disconnect(client));

while(client_ready) {
    vTaskDelay(500/portTICK_PERIOD_MS);
    ESP_LOGI(TAG, "gracefully disconnecting...");
}

ESP_LOGI(TAG, "going to sleep, bye!...");

esp_deep_sleep_start();
```

Figura 2.56 Fragmento de código principal del software embebido

2.4.4.3 Prototipo Físico

El hardware diseñado se implementa en una PCB, en vez de soldar directamente el ESP32 resulta una buena idea colocar un zócalo y así poder montar y desmontar el embebido. Se recubre la placa con una caja de plástico cuya tapa es diseñada e impresa en 3D debido a la necesidad de precisión. En la figura 2.57 se presenta el diseño 3D de la pieza; en la figura 2.58 se muestran los componentes desmontados, mientras que en la figura 2.59 se muestra el prototipo físico finalmente ensamblado.

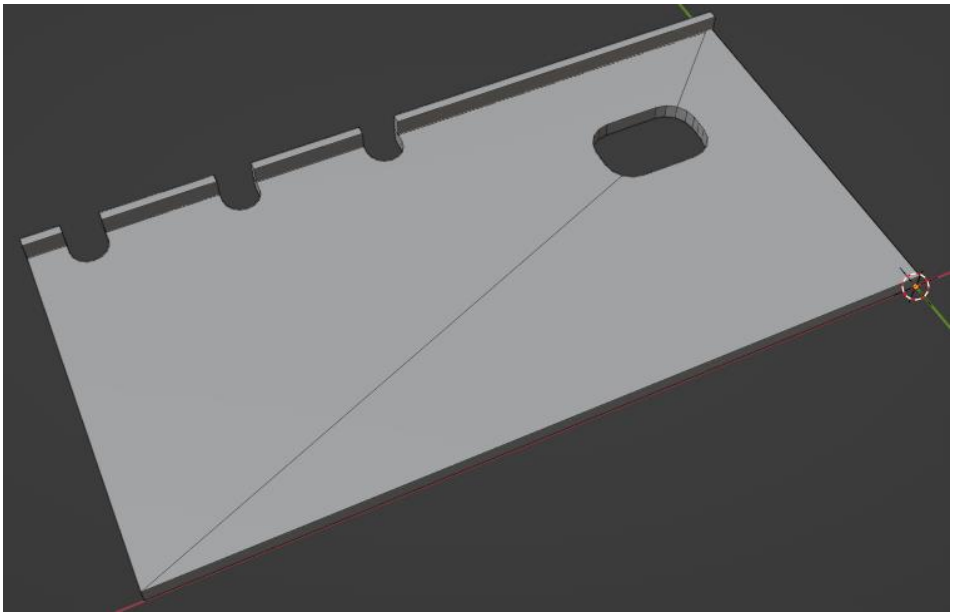


Figura 2.57 Diseño 3D de pieza cobertora

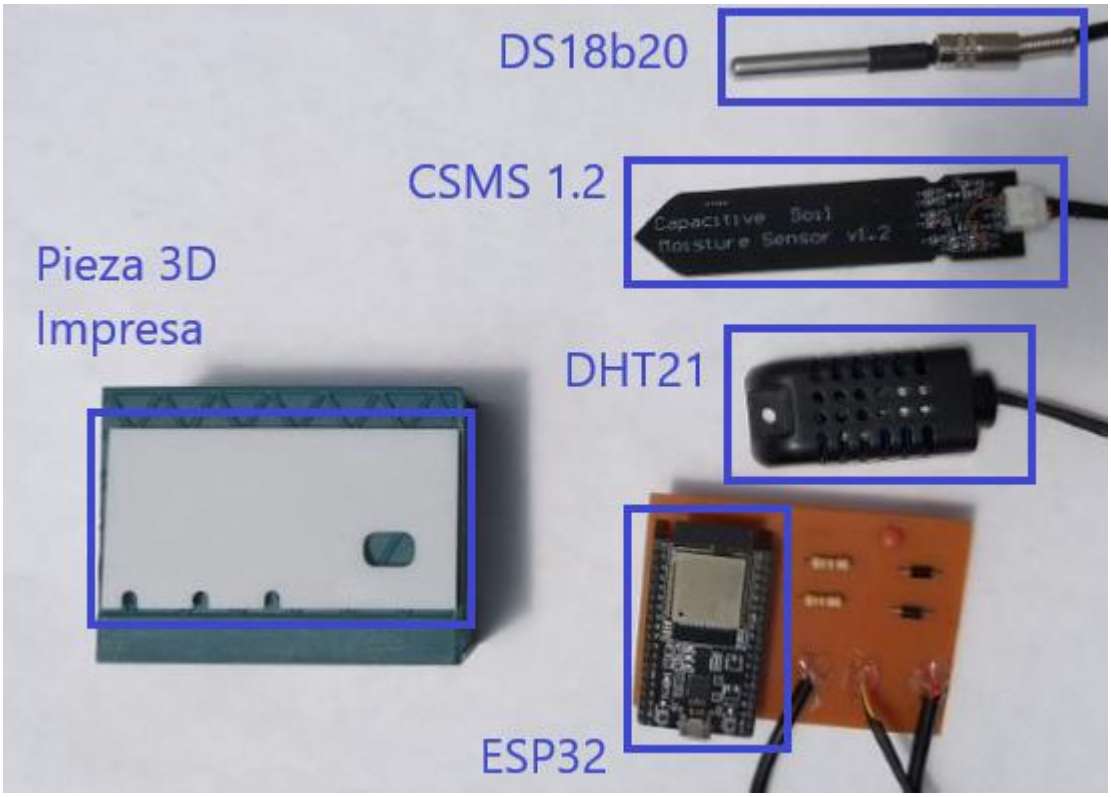


Figura 2.58 Prototipo en partes

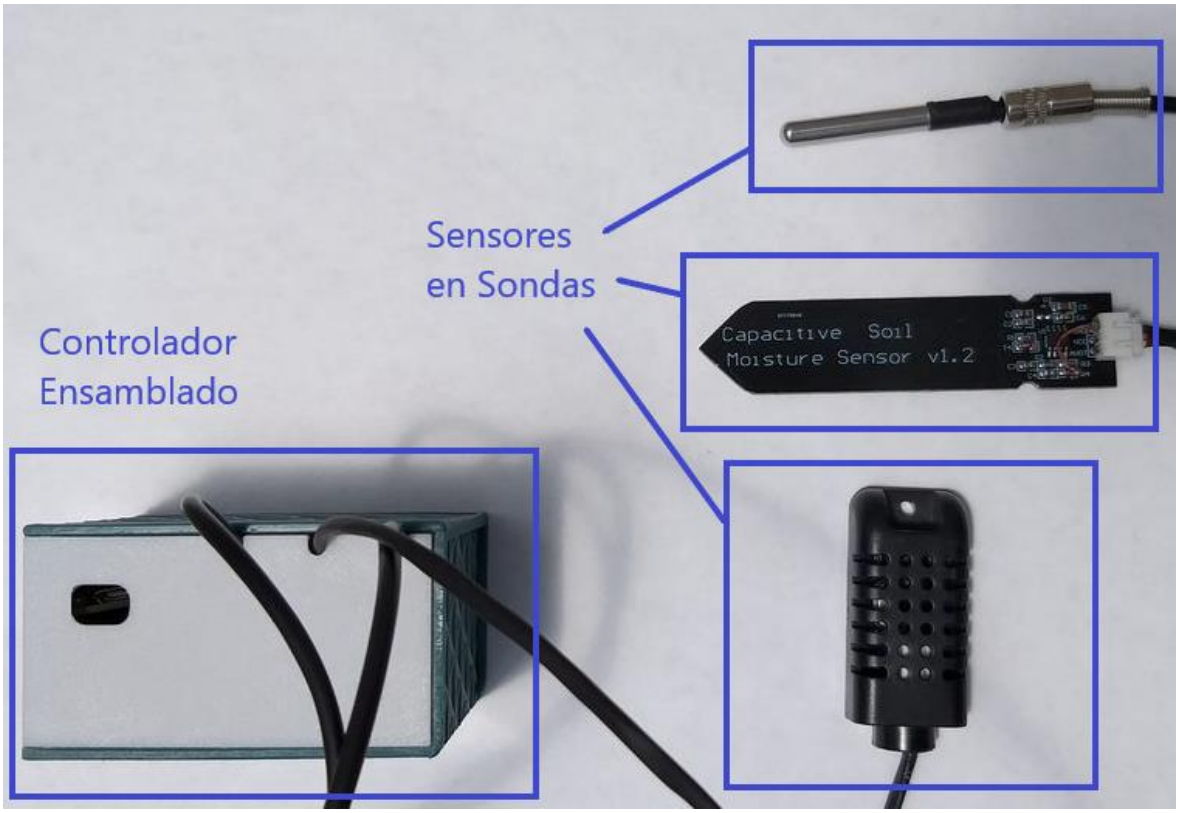


Figura 2.59 Prototipo Ensamblado

2.4.5 SERVICIO REST

Con la base de datos implementada, se prosigue a escribir el código correspondiente a la API REST en un repositorio de .NET Core Web API.

2.4.5.1 Quinta Actualización del Tablero Kanban

Continuando con los pasos establecidos por la metodología, se actualiza el tablero Kanban, moviendo las tareas completadas de la fase anterior a la columna de pruebas y las tareas requeridas para el servicio REST a la columna de desarrollo. El resultado se muestra en el ANEXO A.5.

2.4.5.2 Codificación del Servicio

Siguiendo el diseño se utilizan tres directorios:

- *Controllers*: Donde se ubicarán los controladores MVC
- *Models*: Que contiene los archivos con los diferentes modelos
- *Services*: Para añadir archivos correspondientes a servicios inyectables.

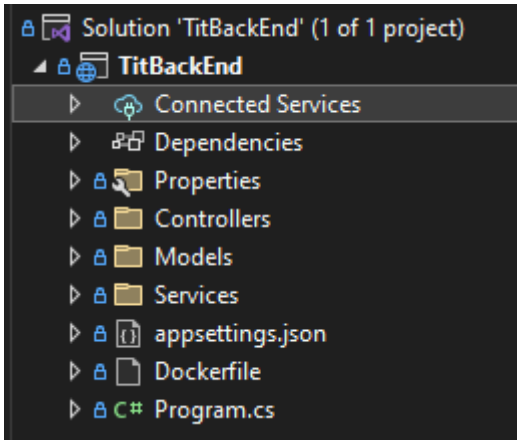


Figura 2.60 Estructura de Archivos del servicio web

La solución de código completa se adjunta en el ANEXO E.

Cabe mencionar que *Entity Framework*, una vez provista una base de datos, se encarga de autogenerar un código base para los modelos. De esta manera, el desarrollador solo deberá preocuparse de establecer las relaciones correctas para la API y los objetos de transferencia de datos o "DTO". Se presenta un ejemplo de modelo a continuación:

```
5 references
public partial class GreenhouseSize
{
    1 reference
    public int Id { get; set; }
    3 references
    public decimal Area { get; set; }
    3 references
    public string Unit { get; set; } = null!;
    3 references
    public decimal XMax { get; set; }
    3 references
    public decimal YMax { get; set; }

    [JsonIgnore]
    1 reference
    public virtual ICollection<Greenhouse>? Greenhouses { get; set; }
}
```

Figura 2.61 Ejemplo de modelo implementado mediante una clase

En cuanto a los controladores, se crea un archivo por cada "*endpoint*" diseñado para la interfaz. Cada archivo contiene el código correspondiente a la lógica de controlador MVC asociando una ruta a un método que a su vez llama a los modelos y servicios necesarios

para completar la petición. Si se produce algún error, será necesario devolver el error al cliente utilizando los códigos HTTP adecuados.

```
// GET api/<GreenhouseSizeController>/5
[HttpGet("{id}")]
[Authorize(Roles = "regular" + "," + "admin")]
1 reference
public async Task<IActionResult> Get(int? id)
{
    var greenhouseSize = await _context.GreenhouseSizes.FindAsync(id);
    if (greenhouseSize == null) return NotFound();

    return Ok(greenhouseSize);
}
```

Figura 2.62 Fragmento de código correspondiente a controlador

En la figura 2.62 es importante notar que se utilizan métodos asíncronos ya que el contexto de la base de datos es inyectado en varios controladores, por lo que el controlador actual deberá esperar hasta que la consulta sea completada para poder emitir una respuesta.

En el archivo principal "*program*.cs" resulta imperativo inyectar los servicios y el contexto de la base de datos provisto por *Entity Framework*, además, se habilita el servicio de Swagger para probar la API y se "abre" el CORS para poder realizar pruebas desde diferentes orígenes (en producción esto debería ser cambiado para permitir solo orígenes conocidos). Las figuras 2.63 y 2.64 presentan la lógica y configuración descrita.

```
builder.Services.AddDbContext<TitDBContext>(options => options.UseSqlServer(
    builder.Configuration.GetConnectionString("DefaultConnection")));
```

Figura 2.63 Fragmento de código principal del servicio web

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=DESKTOP-TI416LH\\SQLEXPRESS;
  },
```

Figura 2.64 Fragmento de configuración correspondiente a base de datos

2.4.6 WEB APP DE FRONT-END

El último componente por implementar es la interfaz de usuario en el navegador mediante la codificación de una *app* en Angular.

2.4.6.1 Sexta Actualización del Tablero Kanban

Se procede a actualizar el tablero Kanban moviendo las tareas completadas en la fase anterior hacia la columna de pruebas y las tareas correspondientes a la app de *FE* a la columna de desarrollo. Se presenta el tablero resultante en el ANEXO A.6.

2.4.6.2 Codificación de la Interfaz de Usuario

Tras iniciar la *app* con Angular CLI, se instala Angular Material para proveer estilos y temas al desarrollador. La terminal también permite instanciar los componentes y servicios propuestos en la fase de diseño.

El código completo podrá ser encontrado en el ANEXO F.

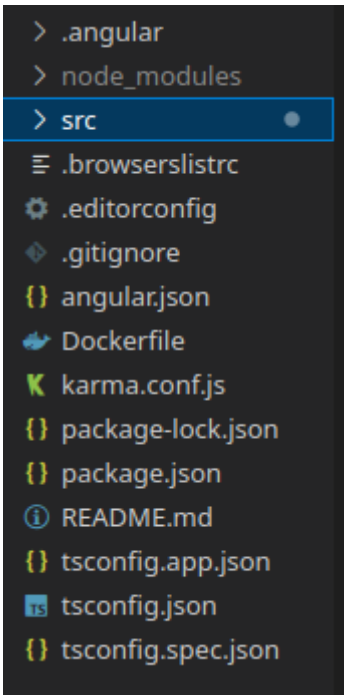


Figura 2.65 Estructura de Archivos del Frontend

Inicialmente, se dividen las pantallas en una de ingreso y en una principal. Dentro de la principal se utiliza un menú de aplicación, una barra lateral de cajón desplegable y un pie de página con el nombre del desarrollador. Parte del código se muestra en la figura 2.66.

```
<mat-sidenav-container class="main-content-container">
    <mat-sidenav #sidenav mode="side" opened class="main-sidenav">
        <app-main-sidenav></app-main-sidenav>
    </mat-sidenav>
    <mat-sidenav-content>
        <router-outlet></router-outlet>
    </mat-sidenav-content>
</mat-sidenav-container>
```

Figura 2.66 Fragmento de código HTML correspondiente a pantalla principal

En la barra lateral se introducen tres grupos desplegables que muestran botones para acceder a las diferentes vistas de la *app*:

Administración es una zona reservada para usuarios con permisos correspondientes, aquí se despliega la información de sensores, registros e infraestructura contenidos en la base de datos de forma visual; también se añaden botones para las diferentes acciones posibles (CRUD).

```
onSensorCreate() {
  const dialogRef = this.dialog.open(ActionDialogComponent, {
    data: {
      "name": "create",
      "type": "sensor",
      "greenhouses": this.greenhouses,
      "sensorTypes": this.sensorTypes,
      "sensorPositions": this.sensorPositions,
      "value": {}
    }
  });
  dialogRef.afterClosed().subscribe(result => {
    if (result != undefined) {
      this.handleData('Sensor', {"name":"create"},result);
      this.loadPageData();
    }
  });
}
```

Figura 2.67 Fragmento de código TypeScript de administración de sensores

Reportes muestra mediante tablas los datos recolectados en campos según la consulta que haga el usuario. Permite también añadir filtros de fecha para una búsqueda más específica. La figura 2.68 muestra un fragmento del código correspondiente.

```
<mat-card-title>Valores Ambientales</mat-card-title>
<mat-card-subtitle>Selecciona y consulta valores de una variable ambiental</mat-card-subtitle>
<mat-card-actions>
    <mat-form-field appearance="fill" style="margin: 4px;">
        <mat-label>Variable Ambiental</mat-label>
        <mat-select [(ngModel)]="selectedValue" name="tipoSensor">
            <mat-option *ngFor="let type of sensorTypes" [value]="type">
                {{type.measurable}}
            </mat-option>
        </mat-select>
    </mat-form-field>
    <button mat-raised-button (click)="generateQuery()" [disabled]="selectedValue==undefined"
        color="accent">Buscar</button>
```

Figura 2.68 Fragmento de código de reportes ambientales

Gráficos contiene la lógica necesaria para graficar los datos mediante las herramientas propuestas en el diseño. Cabe mencionar que los datos deben ser adaptados por lo que se utiliza el servicio correspondiente, el cual es inyectado en el constructor del componente.

```
constructor(private httpHandlerService: HttpHandlerService,
  private dataHandlerService: DataHandlerService) { }

ngOnInit(): void {
  this.httpHandlerService.getData('SensorType')
    .subscribe({
      next: (ts) => this.sensorTypes = ts,
      error: (e) => console.error(e),
      complete: () => console.info('complete')
    });
}
```

Figura 2.69 Fragmento de código correspondiente a gráficos

Dentro del directorio de servicios se tiene agrupado el código reutilizable correspondiente tanto a la consulta HTTP como la adecuación de datos a otras estructuras de objetos de JavaScript. Una parte de esta lógica se muestra en la figura 2.70

```
public deleteData(resource: string): Observable<any> {
  const httpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  };
  console.log(baseURL + resource);
  return this.http.delete<any>(baseURL + resource, httpOptions)
    .pipe(catchError(this.handleError));
}
```

Figura 2.70 Fragmento de código de servicio para manejo de HTTP

Cabe mencionar que los servicios devuelven “observables” que son interfaces que implementan el patrón observador mediante RxJS.

3 RESULTADOS Y DISCUSIÓN

Con el objetivo de validar el Sistema implementado, en esta sección se presentan los resultados de las pruebas de funcionalidad más relevantes obtenidas de la evaluación del producto final. Las pruebas de funcionalidad se han organizado en dos partes:

- Pruebas de funcionamiento individuales de los componentes del Sistema.
- Pruebas de funcionamiento global del Sistema.

3.1 SÉPTIMA ACTUALIZACIÓN DEL TABLERO KANBAN

Siguiendo con la metodología se actualiza el tablero Kanban para tener todas las tareas en la fase de pruebas. El tablero resultante se muestra en el ANEXO A.7.

3.2 PRUEBAS DE FUNCIONAMIENTO INDIVIDUAL

Antes de probar el sistema por completo, es necesario asegurarse que cada componente esté funcionando tal como fue diseñado, por lo que se realizan y se presentan estas pruebas a continuación.

3.2.1 BASE DE DATOS

Para probar la base de datos, se inicia el contenedor del componente escuchando en el puerto 1433 del ambiente de desarrollo (ver figura 3.1).

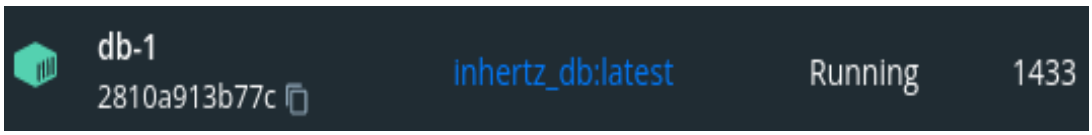


Figura 3.1 Base de datos corriendo en Docker

Se utiliza la herramienta SQL *Server Management Studio*, para conectarse y explorar la base de datos dentro del contenedor (ver figura 3.2).

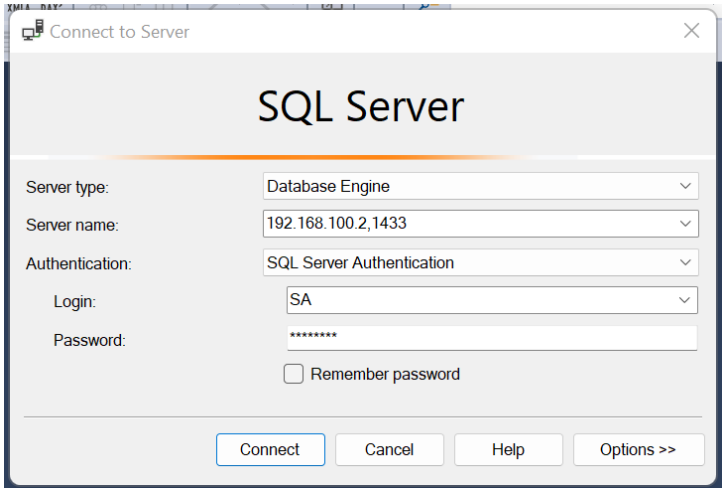


Figura 3.2 Conectando a la base de datos contenerizada

Una vez dentro de la base de datos se pueden observar las tablas creadas de acuerdo con el diseño (ver figura 3.3). También se pueden realizar consultas desde un ordenador externo con relación al contenedor (ver figura 3.4).

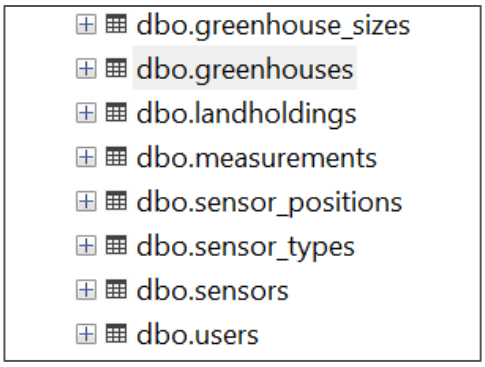
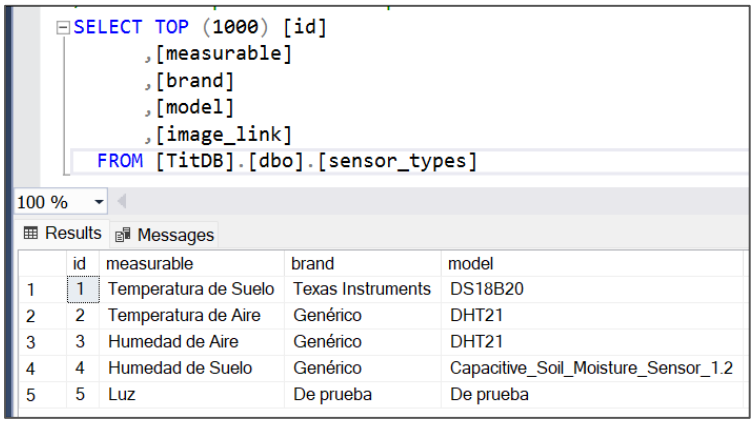


Figura 3.3 Tablas generadas en la base de datos



	id	measurable	brand	model
1	1	Temperatura de Suelo	Texas Instruments	DS18B20
2	2	Temperatura de Aire	Genérico	DHT21
3	3	Humedad de Aire	Genérico	DHT21
4	4	Humedad de Suelo	Genérico	Capacitive_Soil_Moisture_Sensor_1.2
5	5	Luz	De prueba	De prueba

Figura 3.4 Consulta y respuesta en tabla “tipos de sensor”

Se observa que el componente de base de datos está funcionando de acuerdo con el diseño planteado y habilita la persistencia de información en el sistema.

3.2.2 RECOLECTOR IOT

Para probar este componente, se inicia el contenedor implementado en Go con sus dependencias de prueba, tal como fue detallado en secciones anteriores.

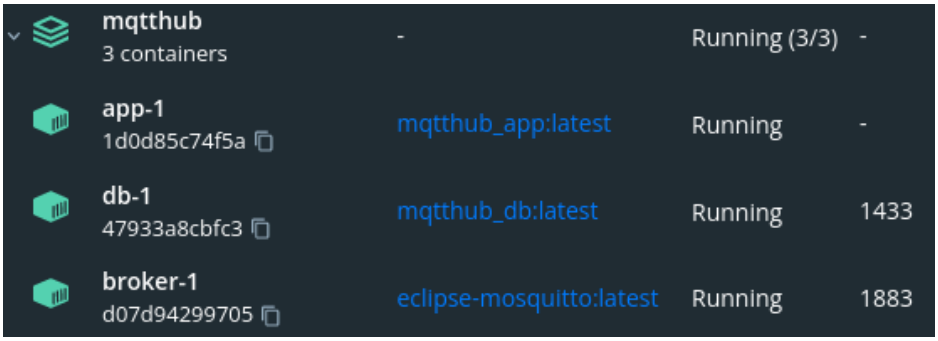


Figura 3.5 Recolector y dependencias corriendo en Docker

El *broker* de prueba muestra que el recolector se ha conectado correctamente (ver figura 3.5); cabe mencionar que la dirección IP desde donde se conecta pertenece a la red virtual generada por los contenedores (ver figura 3.6).

```
New connection from 172.18.0.4:41444 on port 1883.
New client connected from 172.18.0.4:41444 as goHub (p2, c1, k30).
```

Figura 3.6 Respuestas por consola de conexión del recolector

En la figura 3.7 se puede observar que al enviar un registro ambiental desde un cliente de prueba, el recolector se encarga de inferir el sensor de origen y construir el modelo a ser insertado en la base de datos. Si existe algún error se devuelve un error sin colapsar la aplicación (ver figurar 3.8).

```
New connection from 172.18.0.1:58330 on port 1883.
New client connected from 172.18.0.1:58330 as test_client (p2, c1, k60).
Client test_client disconnected.
```

Figura 3.7 Respuesta por consola de conexión de cliente de prueba

```
00:02:47 record not found

00:02:47 /app/internal/adapters/db/sqlserver.go:41 record not found
[rows:0] SELECT * FROM "sensors" WHERE device_add = 'ADD_123' ORDER BY "sensors
1 ROWS ONLY
```

Figura 3.8 Respuesta de error al ingresar valores incorrectos

De los resultados se puede determinar que el componente recolector se encuentra funcionando según el diseño y provee una interfaz para que los diferentes dispositivos de borde puedan insertar registros ambientales dentro de la capa de Persistencia del sistema.

3.2.3 BROKER MQTT

Para probar el *broker* se inicia el contenedor del componente escuchando en el puerto 1883 del ambiente de desarrollo (ver figura 3.9).

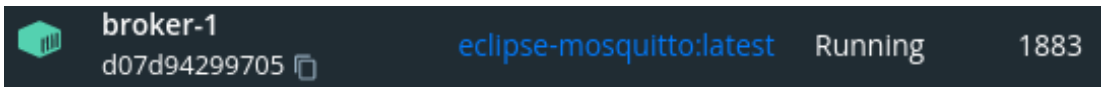


Figura 3.9 Broker MQTT corriendo en Docker

Se utiliza la herramienta *WireShark* para capturar el tráfico que pasa por el puerto del contenedor y el resultado se muestra en la figura 3.10.

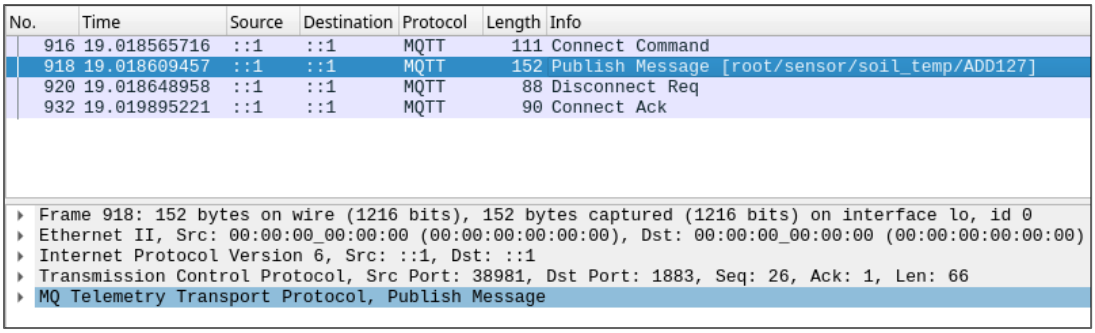


Figura 3.10 Tráfico MQTT capturado

Analizando un paquete enviado por un cliente de prueba se obtiene la captura que se muestra en la figura 3.11.

```
Msg Len: 64
Topic Length: 28
Topic: root/sensor/soil_temp/ADD127
Message: 7b2276616c6f72223a2031372c2022756e69646164223a202243656c63697573227d
```

Figura 3.11 Exploración de paquete MQTT de prueba

```
65 6e 73 6f 72 2f 73 6f  69 6c 5f 74 65 6d 70 2f   ensor/so il_temp/
41 44 44 31 32 37 7b 22  76 61 6c 6f 72 22 3a 20  ADD127{" valor":
31 37 2c 20 22 75 6e 69  64 61 64 22 3a 20 22 43  17, "uni dad": "C
65 6c 63 69 75 73 22 7d                           elcius"}
```

Figura 3.12 Carga del mensaje como texto plano en JSON

En la figura 3.12 se puede apreciar tanto el tópico para un sensor de prueba “ADD127” como el mensaje en hexadecimal y en texto plano, por lo que se determina que el componente está funcionando tal como fue planeado.

3.2.4 PROTOTIPO DE DISPOSITIVO DE BORDE

Antes de evaluar el componente entero se utiliza Arduino (por su facilidad) para probar los sensores. Los resultados se muestran en las figuras 3.13 y 3.14.

Figura 3.13 Pruebas a sensor de humedad de suelo con Arduino

Como se puede ver en el graficador, el valor devuelto por el sensor cambia directamente proporcional a la cantidad de agua con la que dicho sensor se encuentre en contacto. Es importante mencionar que la precisión y exactitud de la medición dependerá de la calidad del dispositivo y de cómo sea tratada la señal analógica.

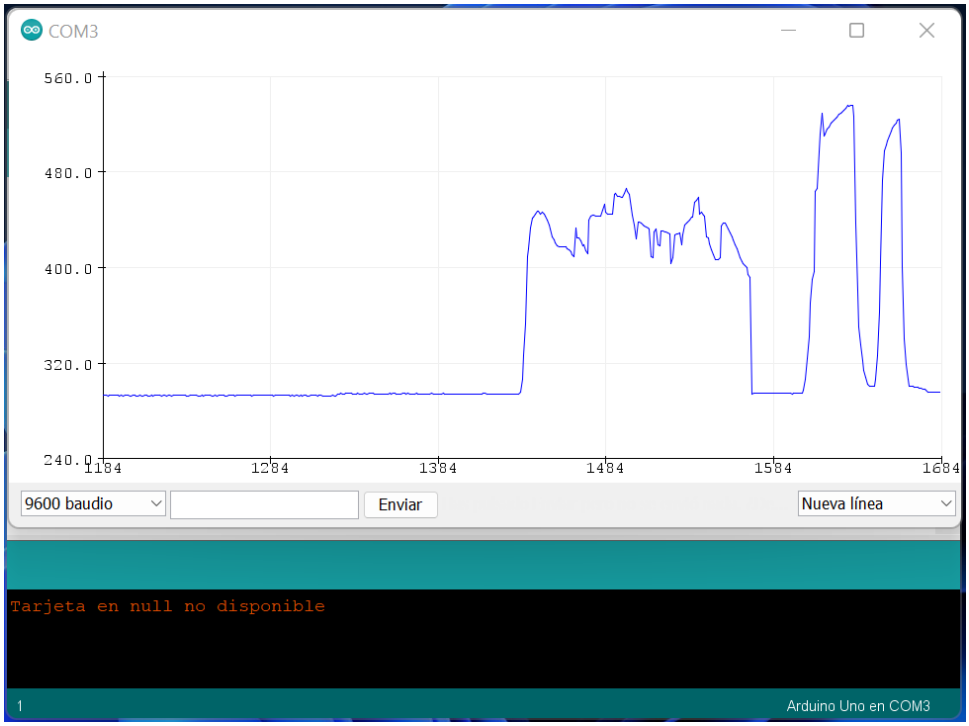


Figura 3.14 Respuesta por puerto serial de la prueba anterior

3.2.5 API REST

Para probar el servicio web se inicia el contenedor correspondiente con sus dependencias de prueba.

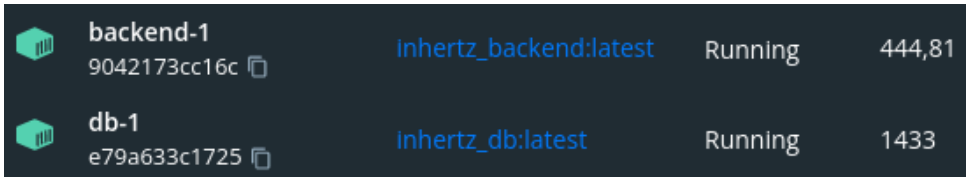


Figura 3.15 Servicio Backend y dependencia corriendo en Docker

Si se inicia dentro de un ambiente de desarrollo, el servicio permite probar la interfaz mediante Swagger (ver figura 3.15). Este último ofrece su propia interfaz, pero de manera visual, mostrando ejemplos de los objetos JSON que el servicio recibe y devuelve.

Pero resulta mejor idea probar toda la API utilizando Postman (ver figuras 3.16 a 3.18), ya que éste permite automatizar las pruebas con código y exportar los resultados, los cuales son adjuntados dentro del ANEXO E.

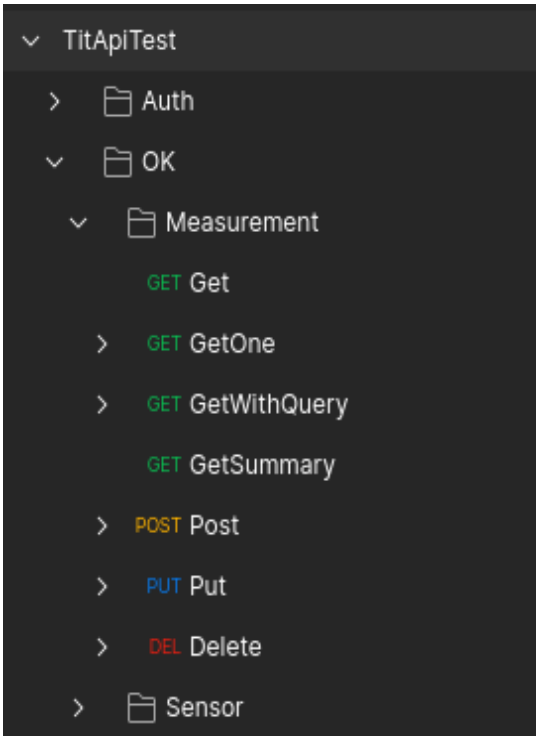


Figura 3.16 Estructura de pruebas automatizadas con Postman

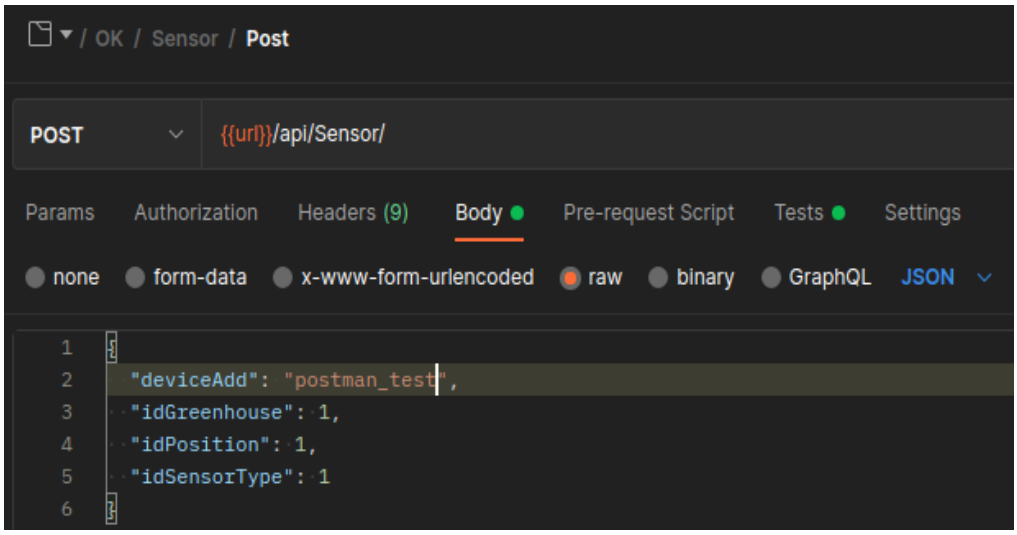


Figura 3.17 Ejemplo de prueba tipo POST

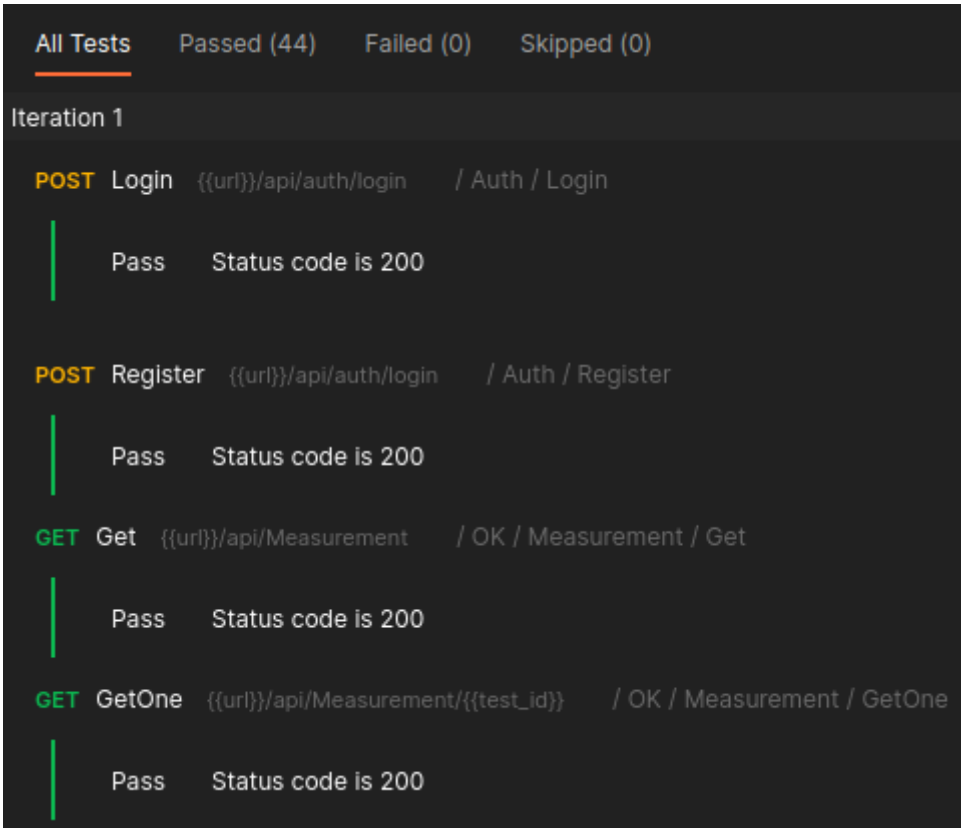


Figura 3.18 Fragmento de respuestas correctas de Postman

Para verificar las evaluaciones automáticas se utiliza *WireShark* y se captura los paquetes con el fin de analizar paquetes de solicitud y respuesta, tal como se muestra en la figura 3.19.

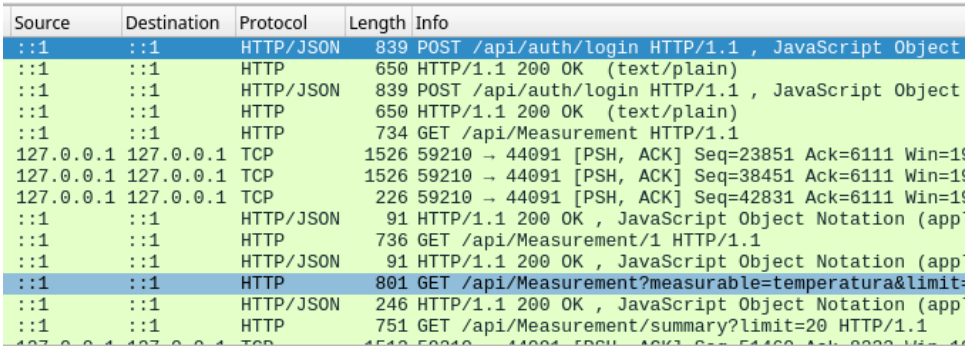


Figura 3.19 Captura de tráfico HTTP

En la figura 3.20 se puede observar un ejemplo de petición HTTP tipo POST en la *endpoint* “/api/SensorType/” con un cuerpo de mensaje en JSON.

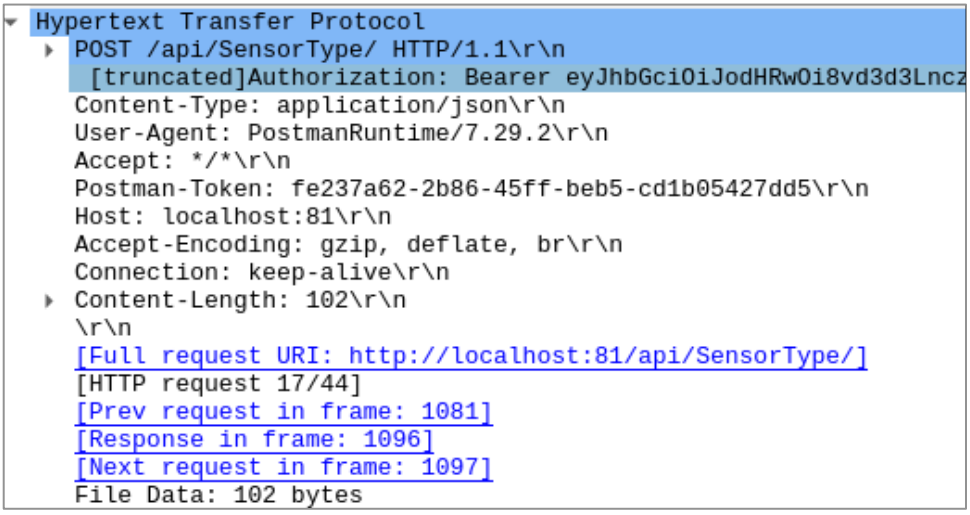


Figura 3.20 Análisis de petición HTTP de prueba

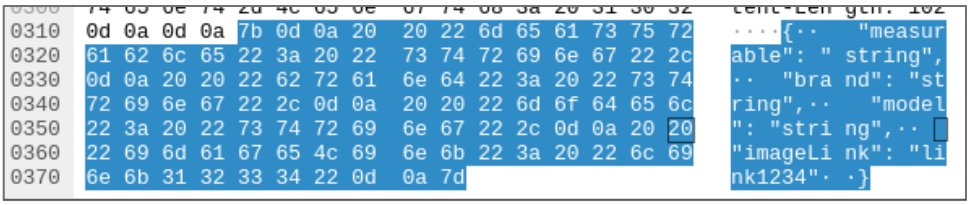


Figura 3.21 Análisis de contenido HTTP de prueba en JSON

La respuesta de la petición anterior se muestra en la figura 3.22.

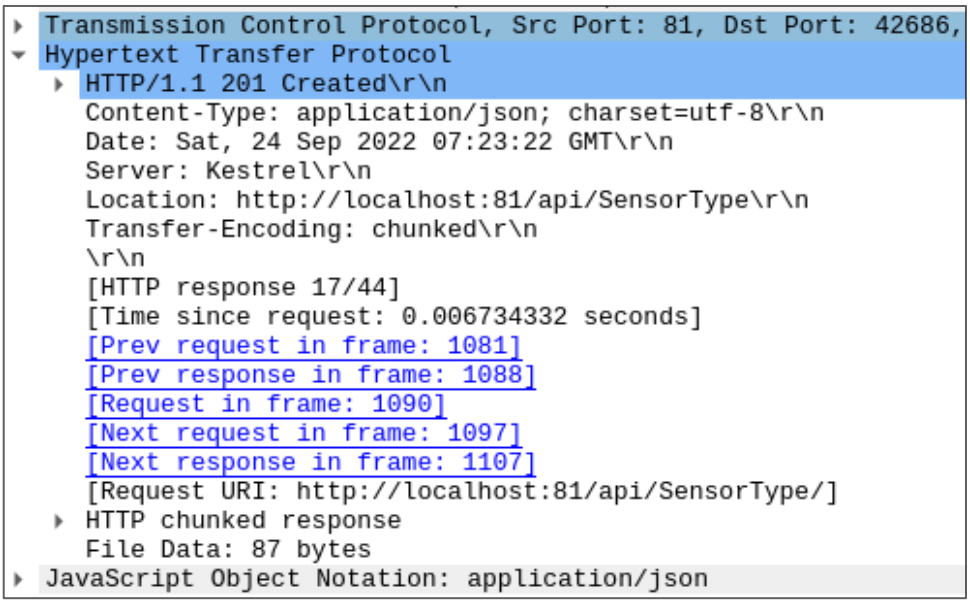


Figura 3.22 Análisis de respuesta HTTP de prueba

El componente *Backend* funciona como su diseño lo indica y provee una Interfaz HTTP para intercomunicar cliente y servidor.

3.2.6 APLICACIÓN DE FRONTEND

Para probar la interfaz gráfica se utiliza el servidor de prueba de Node.js que trae Angular, así como un servicio web de prueba.

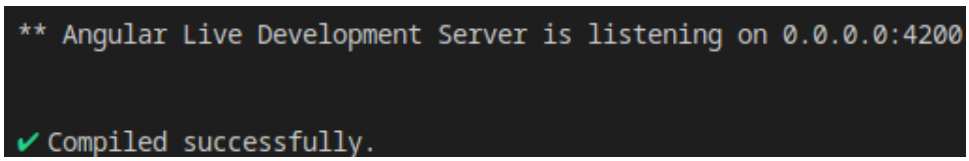


Figura 3.23 Angular corriendo en ambiente de desarrollo

Una vez dentro, se pueden observar los componentes manifestados en el DOM de manera visual. También resulta evidente la URI escrita en la barra de exploración del navegador que indica la vista dentro del *router* (ver figura 3.24).

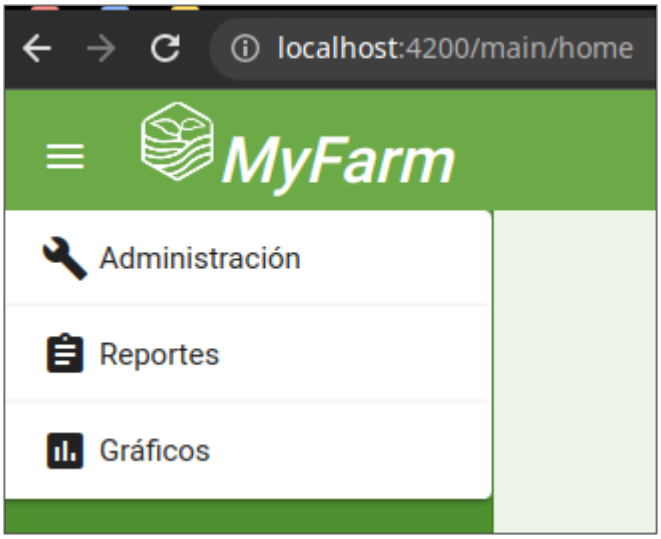


Figura 3.24 Fragmento de pantalla Principal

Al abrir las herramientas del desarrollador en el navegador, se pueden revisar las consultas realizadas por la aplicación al servicio de prueba, así como la respuesta en un arreglo de objetos JSON (ver figura 3.25).

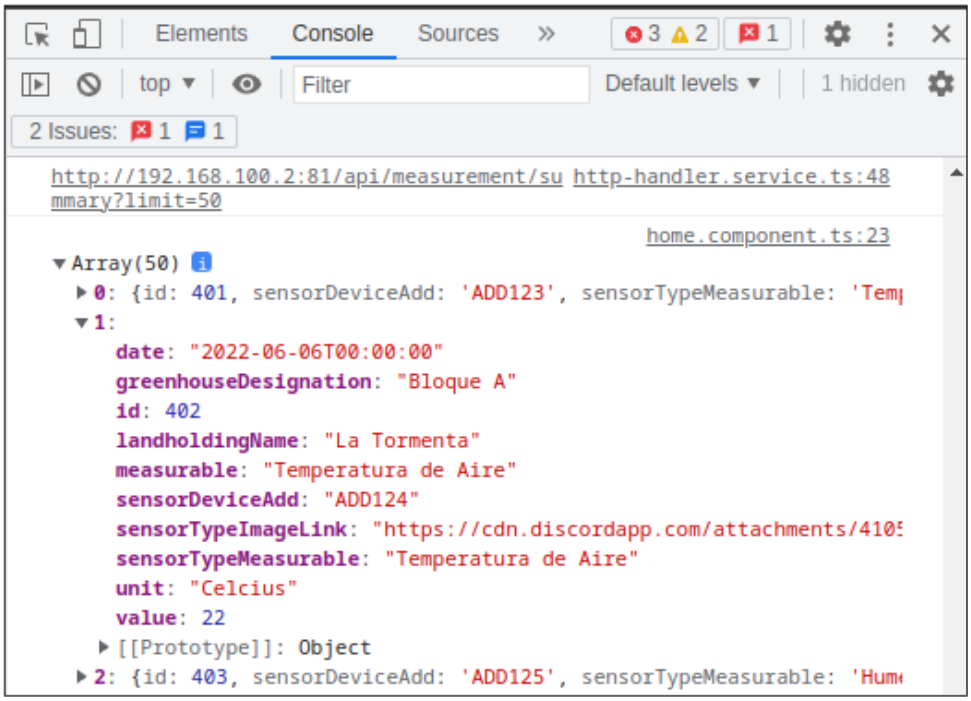


Figura 3.25 Análisis de petición del cliente en Navegador

Los mensajes de error también se muestran si se genera alguno, esto para proveer información de uso al cliente (ver figura 3.26).

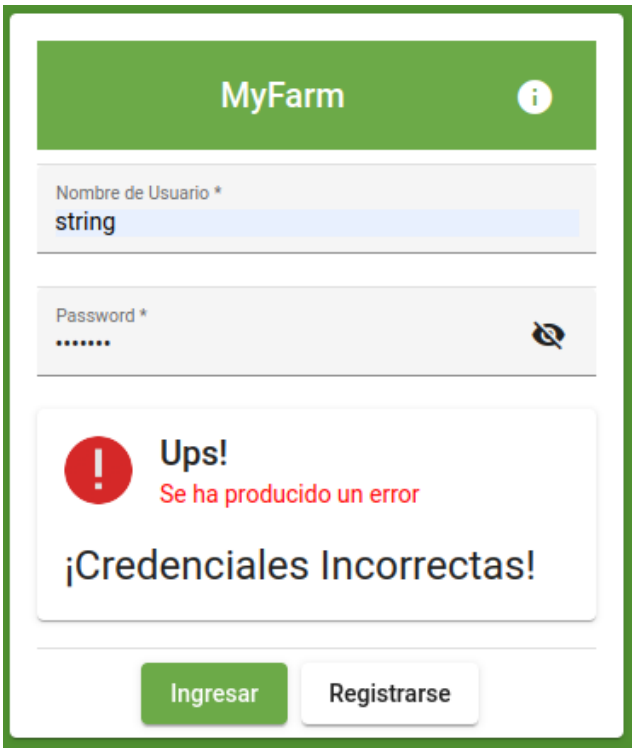


Figura 3.26 Ejemplo de error al momento de ingreso

De las pruebas realizadas se establece que el componente *Frontend* funciona según su diseño y provee una interfaz gráfica a los usuarios.

3.3 OCTAVA ACTUALIZACIÓN DEL TABLERO KANBAN

Una vez finalizadas las pruebas individuales se mueven las tareas de Jira que han sido implementadas mediante los diferentes componentes probados, desde fase de “pruebas” a fase de “completadas”. El tablero resultante se presenta en el ANEXO A.8, nótese que las historias de usuario siguen en la columna de “*testing*” ya que éstas deberán ser probadas en la siguiente sección.

3.4 PRUEBAS DE FUNCIONAMIENTO GLOBAL DEL SISTEMA

En esta sección se prueba todo el sistema, evaluando el cumplimiento de los criterios de aceptación planteados en cada historia de usuario. Primero se “levantan” todos los componentes en sus contenedores. A continuación se ubica el prototipo electrónico en un ambiente de prueba caracterizado por dos escenarios:

- Escenario 1: Una tarde nublada y lluviosa.
- Escenario 2: Una mañana despejada.

Se incluye un grupo de fotos por cada escenario como ANEXO G.1 y G.2 respectivamente.

3.4.1 TIT-3 (MEDIR CONDICIONES AMBIENTALES)

El criterio de aceptación dictamina que el sistema debe poder medir condiciones ambientales, mediante los sensores desplegados en campo generando datos, en tanto que la *app* en el navegador permite observar dichos datos.

Se comprueba que el sistema cumple con este criterio como se muestra en la figura 3.27 para el escenario 1 y en la figura 3.28 para el escenario 2.

Figura 3.27 Datos Generados en el Escenario 1

Figura 3.28 Datos Generados en el Escenario 2

3.4.2 TIT-4 (GUARDAR INFORMACIÓN AMBIENTAL)

El criterio de aceptación de esta historia dice que el sistema debe poder almacenar variables ambientales; es decir, el sistema debe ser capaz de mostrar datos que fueron generados anteriormente mientras continúa generando valores. En las figuras 3.29 y 3.30 se observan dos grupos de mediciones cuya principal diferencia es el tiempo en el que fueron generados. De esta manera se prueba que el proyecto cumple con dicho criterio.

Figura 3.29 Mediciones tomadas en diferentes momentos

Figura 3.30 Tabla con valores anteriores y actuales

3.4.3 TIT-5 (LOCALIZAR DISPOSITIVOS FÍSICOS)

Figura 3.31 Relación entre Fincas, Bloques, Sensores y Registros

Para esta historia de usuario se tiene un criterio que especifica que el Sistema debe mantener una relación entre los diferentes dispositivos físicos y la infraestructura donde se encuentran localizadas. El proyecto cumple con esta condición como se observa en la figura 3.31 en la que se muestra la relación entre los sensores del prototipo y la infraestructura del ambiente de prueba.

3.4.4 TIT-6 (AÑADIR NUEVOS DISPOSITIVOS)

En este caso, el criterio de aceptación se cumple si el Sistema es capaz de añadir nuevos dispositivos físicos sin comprometer el resto de datos. En la figura 3.32 se presenta el proceso para añadir un nuevo sensor a través de la interfaz de usuario.

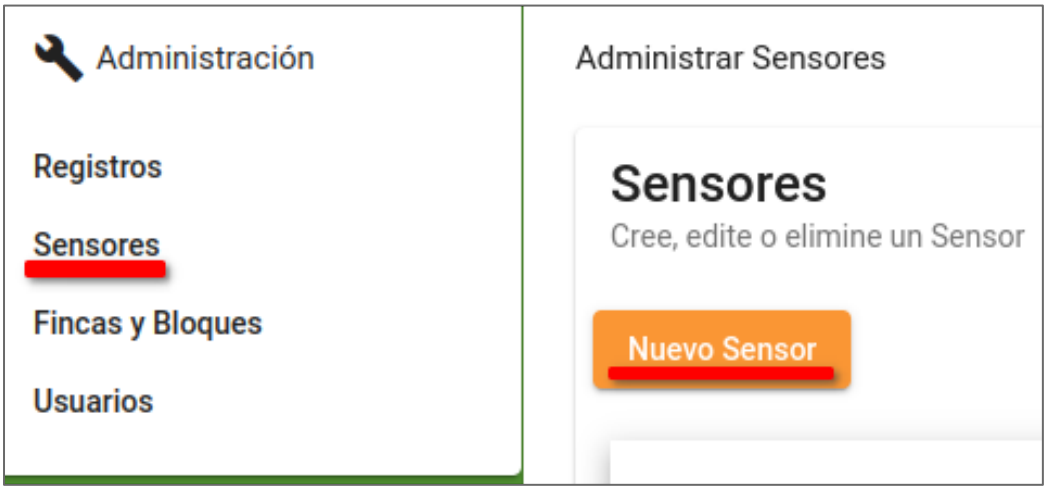


Figura 3.32 Adición de Nuevos Sensores

3.4.5 TIT-7 (ADMINISTRAR LA INFRAESTRUCTURA DE LOS CULTIVOS)

Para cumplir con el criterio de aceptación correspondiente, será necesario que el Sistema pueda gestionar la representación digital de la infraestructura del cultivo, es decir: crear, modificar y borrar modelos de fincas y bloques o invernaderos. Estas opciones se presentan en la figura 3.33.

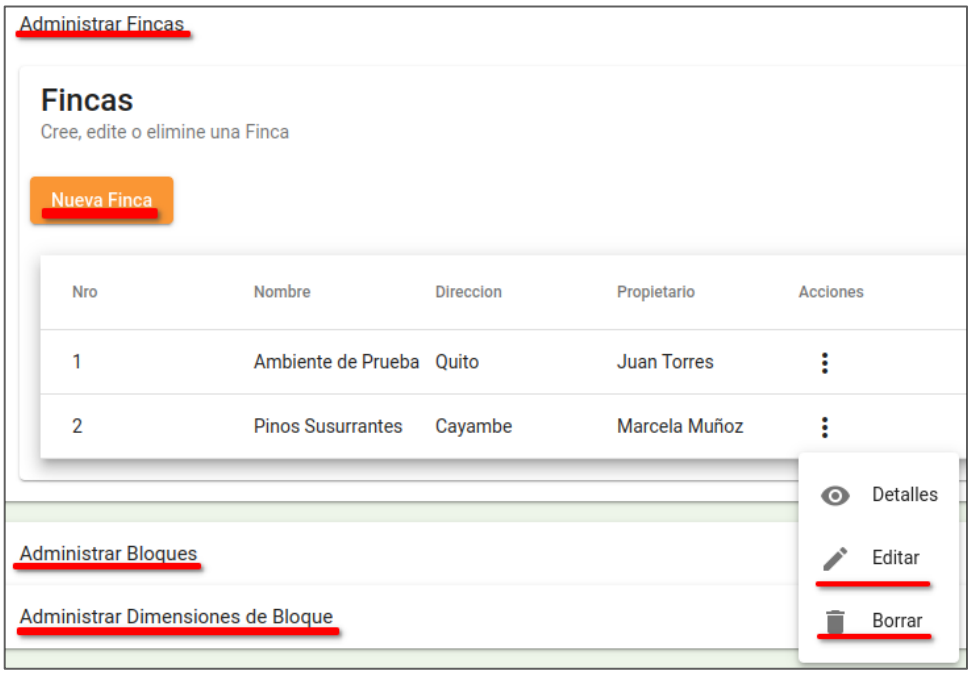


Figura 3.33 Administración de la Infraestructura de Cultivos

3.4.6 TIT-8 (PROVEER GRÁFICOS)

El Sistema cumplirá con el criterio de aceptación correspondiente si es capaz de generar gráficos con los datos generados por los sensores desplegados. A manera de ejemplo, en la figura 3.34 se presenta un gráfico producto de datos de prueba, con lo que se demuestra que se cumple el requerimiento actual.

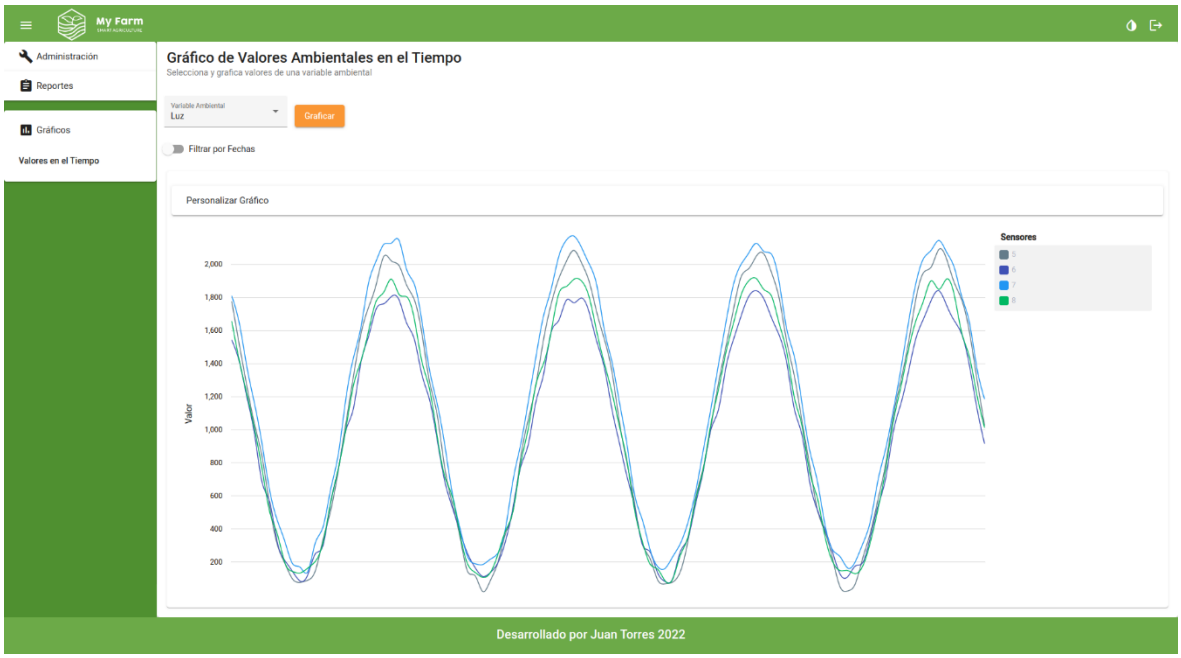


Figura 3.34 Gráfico Generado en la Interfaz de Usuario

3.5 ACTUALIZACIÓN FINAL DEL TABLERO KANBAN

Una vez que se ha probado que el Sistema cumple con todos los criterios de aceptación, se mueven todos los *issues* tipo “historia de usuario” desde la columna de “pruebas” hacia la columna de “realizados”, dando por finalizado el desarrollo del Sistema. El tablero resultante se muestra en el ANEXO A.9.

3.6 ANÁLISIS DEL SISTEMA DESARROLLADO

El Sistema presenta resultados positivos en cada historia de usuario evaluada mediante su correspondiente criterio de aceptación; sin embargo, para poder cumplir con los atributos de calidad se tomaron decisiones de diseño que sobrepasan el alcance del proyecto planteado en el inicio.

En la figura 1.1, que corresponde al esquema general del Sistema planteado en el alcance, se observa la concepción inicial del proyecto, el resultado final es muy similar pero presenta mayor complejidad. Por ejemplo, aprovechando Docker, todos los componentes (exceptuando el prototipo físico) fueron “contenerizados” para mejorar la capacidad de despliegue y portabilidad del sistema, y por ende facilitar la fase de pruebas.

También la necesidad de diferenciar usuarios regulares y administradores, así como la naturaleza distribuida del Sistema representa un desafío de autenticación y consecuentemente de seguridad tanto para la interfaz de usuario como para el servicio Web. Esta situación fue solucionada mediante la implementación de tecnologías estándares de seguridad como JWT (*Json Web Token*), HMAC (*Hash-based Message Authentication Code*) y SHA512 (*Secure Hash Algorithm* 512 bits) para el correcto almacenamiento de credenciales de usuarios.

Otro aspecto importante por analizar es la separación de responsabilidades del código embebido para la conexión con el *broker*. Una finca no necesariamente cuenta con la infraestructura de red capaz de soportar varias conexiones inalámbricas simultáneas, aun cuando MQTT es sumamente ligero. La solución viene dada por dicha separación de código, así como la capacidad del ESP32 de actuar tanto como estación de punto de acceso a la red, y de poder formar mallas con otros ESP32 ampliando el rango de cobertura posible para cubrir todo el cultivo.

Finalmente se hace un recuento de las diferentes arquitecturas que dan estructura al Sistema y se enfatiza la necesidad de éstas. En el presente proyecto se describe la solución desde varias vistas arquitectónicas pues dicha solución presenta una alta complejidad que involucra varias tecnologías y áreas de conocimiento; sin estas herramientas de diseño no sería posible dividir la tarea compleja en pequeñas responsabilidades que puedan ser implementadas con código.

4 CONCLUSIONES Y RECOMENDACIONES

4.1 CONCLUSIONES

- Este trabajo de titulación ha permitido el desarrollo de un sistema Web capaz de recolectar datos ambientales para fincas agrícolas. Con el aporte de la tecnología IoT en este proyecto, se ha logrado automatizar al proceso de monitoreo de condiciones ambientales que afectan el desarrollo de los cultivos.

- Se ha revisado la teoría necesaria para el desarrollo del Sistema haciendo énfasis en la ingeniería de software y las ciencias computacionales, ya que estos temas están presentes a través de todo el proyecto. La tecnología actual está fuertemente ligada a las computadoras, por ende, se ha comprobado la importancia de contar con habilidades de desarrollo de software.

- Se ha diseñado un sistema de recolección de factores ambientales utilizando historias de usuario y atributos de calidad para definir los requerimientos. Se destaca la importancia de las diferentes arquitecturas de software utilizadas para reducir la complejidad del sistema en funcionalidades fáciles de codificar.

- Se ha implementado el Sistema diseñado utilizando tecnologías “de código abierto” y hardware económico. Si bien esta situación eleva la curva de aprendizaje, también reduce en gran medida el presupuesto necesario para completar el proyecto. Si en un caso específico se llegan a necesitar mediciones más precisas o de variables ambientales poco comunes, se puede adquirir dichos sensores sin tener que modificar el resto del sistema.

- Se ha probado el sistema Web implementado y se ha encontrado que el proyecto cumple con los criterios de aceptación propuestos en cada historia de usuario, que en conjunto determinan los requerimientos funcionales establecidos en la fase de diseño siguiendo la metodología Kanban. Para cumplir con los requisitos no funcionales fue necesario añadir algunos elementos fuera del alcance del proyecto.

- Se ha desarrollado una solución escalable gracias al énfasis puesto en dividir el problema en componentes que pueden ser modificados, reemplazados o removidos según la necesidad sin tener que alterar el resto del sistema. También, el uso de IoT permitirá añadir cientos de dispositivos de borde antes de considerar la modificación de otros componentes.

- Durante el desarrollo del proyecto se han utilizado varias tecnologías; para asegurar la interoperabilidad del sistema se escogieron tecnologías estándares y comúnmente usadas en la industria. Esta elección le permitirá al Sistema ser modificado, expandido y conectado a otros servicios según la necesidad.

- Jira ha probado ser una herramienta invaluable para el desarrollo del presente proyecto no solo por la plataforma que trivializa la administración de éste, sino también por las guías provistas por una comunidad con amplia experiencia manejando flujos de trabajo en el desarrollo de software.

- El sistema embebido ESP32 ha resultado ser una excelente opción para el prototipado de dispositivos IoT, pues combina las herramientas de red de un Raspberry Pi con la capacidad de control analógico y digital de un Arduino por un costo muy inferior que la suma de dichas plataformas, utilizadas en conjunto con alta frecuencia en proyectos IoT.

- El desarrollo de este proyecto ha sido un considerable desafío; desde un comienzo se había planteado el uso de varias tecnologías para el desarrollo *Frontend*, *Backend* y de software embebido, sin embargo, durante la realización del Sistema otros conocimientos fueron requeridos para poder brindar una solución robusta y escalable, capaz de explotar las posibilidades del Internet de las Cosas y de la Industria 4.0.

4.2 RECOMENDACIONES

- Se debe tener en consideración que el software desarrollado y contenerizado está configurado para un ambiente de desarrollo y prueba; antes de poner en producción al Sistema será necesario agregar algunas medidas de seguridad como la capa TLS para todas las interfaces abiertas al Internet.

- Se recomienda considerar protocolos de administración IoT soportados por la plataforma ESP32 para gestionar un gran número de dispositivos en el campo, también se podría utilizar "*Over The Air*" para mantener actualizado el software embebido sin tener que manejar manualmente cada embebido.

- Con la "contenerización" del sistema se abren varias posibilidades para explorar, tales como la arquitectura de microservicios, orquestación con Kubernetes, Integración y Despliegue Continuo (CI/CD), prácticas DevOps, entre otras.

Posiblemente en la industria estos temas no generen beneficios justificables, pero pueden ser útiles para futuros trabajos de titulación en la academia.

- Angular Material provee varios componentes para crear una interfaz de usuario visualmente amigable, pero también se podría utilizar conceptos de diseño UI/UX para mejorar la experiencia del usuario y poder brindar un producto con una interfaz mucho más profesional y atractiva.
- Con el fin de complementar el prototipo de borde, se recomienda implementar conceptos de diseño e ingeniería de productos y así optimizar el tiempo de vida y los costos de producción de los dispositivos desplegados en el campo.

5 REFERENCIAS BIBLIOGRÁFICAS

[1] E. Lichtenberg, “Chapter 23 Agriculture and the environment,” *Handbook of Agricultural Economics*, vol. 2, no. PART A, pp. 1249–1313, Jan. 2002, doi: 10.1016/S1574-0072(02)10005-3.

[2] D. E. Morisigue, D. A. Mata, G. Facciuto, and L. Bullrich, *FLORICULTURA Pasado y presente de la Floricultura Argentina*. INTA, 2012. Accessed: Jun. 12, 2022. [Online]. Available: https://inta.gob.ar/sites/default/files/script-tmp-inta-_floricultura___pasado_y_presente_de_la_floricul.pdf

[3] N. T. Baker and P. D. Capel, “National Water-Quality Assessment Program Environmental Factors That Influence the Location of Crop Agriculture in the Conterminous United States Scientific Investigations Report 2011-5108”, Accessed: May 30, 2022. [Online]. Available: http://photogallery.nrcs.usda.gov/.

[4] N. Gondchawar and R. S. Kawitkar, “IJARCCE IoT based Smart Agriculture,” *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 5, 2016, doi: 10.17148/IJARCCE.2016.56188.

[5] M. Taylor, “Climate-smart agriculture: what is it good for?,” *Journal of Peasant Studies*, vol. 45, no. 1, pp. 89–107, Jan. 2018, doi: 10.1080/03066150.2017.1312355.

[6] “ISO/IEC 2382:2015(en), Information technology — Vocabulary.” https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en (accessed Jun. 30, 2022).

[7] “The Programming Paradigm Evolution,” 2012, Accessed: Jun. 30, 2022. [Online]. Available: www.aosd.net/wiki/index.

[8] B. Stroustrup, “What is Object-Oriented Programming?,” *IEEE Softw*, vol. 5, no. 3, pp. 10–20, 1988, doi: 10.1109/52.2020.

[9] C. Strachey, “Fundamental Concepts in Programming Languages,” *Higher-Order and Symbolic Computation*, vol. 13, no. 1/2, pp. 11–49, 2000, doi: 10.1023/A:1010000313106.

[10] Stack Overflow, “Stack Overflow Developer Survey 2022.” https://survey.stackoverflow.co/2022/#section-most-popular-technologies-programming-scripting-and-markup-languages (accessed Jul. 01, 2022).

[11] E. Horowitz and S. Sahni, “Fundamentals of Data Structures”, Accessed: Jun. 30, 2022. [Online]. Available: www.pnu-club.com

[12] K. L. Berg, T. Seymour, and R. Goel, “History Of Databases,” *International Journal of Management & Information Systems (IJMIS)*, vol. 17, no. 1, pp. 29–36, Dec. 2013, doi: 10.19030/IJMIS.V17I1.7587.

[13] P. C. KANELLAKIS, “Elements of Relational Database Theory,” *Formal Models and Semantics*, pp. 1073–1156, Jan. 1990, doi: 10.1016/B978-0-444-88074-1.50022-6.

[14] M. Keith and M. Schnicariol, “Object-Relational Mapping,” *Pro JPA 2*, pp. 69–106, 2009, doi: 10.1007/978-1-4302-1957-6_4.

[15] “GORM Guides | GORM - The fantastic ORM library for Golang, aims to be developer friendly.” https://gorm.io/docs/ (accessed Sep. 27, 2022).

[16] A. S. Tanenbaum and A. S. Woodhull, “Operating systems : design and implementation,” p. 1054, 2006.

[17] P. Hambarde, R. Varma, and S. Jha, “The survey of real time operating system: RTOS,” *Proceedings - International Conference on Electronic Systems, Signal Processing, and Computing Technologies, ICESC 2014*, pp. 34–39, 2014, doi: 10.1109/ICESC.2014.15.

[18] J. O. Coplien, “Software Design Patterns: Common Questions and Answers”, Accessed: Jun. 30, 2022. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.5376&rep=rep1&type=pdf

[19] H. Y. Yang, E. Tempero, and H. Melton, “An empirical study into use of dependency injection in Java,” *Proceedings of the Australian Software Engineering Conference, ASWEC*, pp. 239–247, 2008, doi: 10.1109/ASWEC.2008.4483212.

[20] D. Garlan, “Software Architecture,” *School of Computer Science - Carnegie Mellon University*, Accessed: Jun. 30, 2022. [Online]. Available: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/ftp/encycSE2001/encyclopedia-dist.pdf

[21] A. Ulalah, “Model-View-Controller (MVC) Architecture,” *ACADEMIA*, 2009, Accessed: Jun. 30, 2022. [Online]. Available: http://www.jdl.co.ukhttp://www.johndeacon.net

[22] T. Dingsøyr, S. Nerur, V. Balijepally, and N. B. Moe, “A decade of agile methodologies: Towards explaining agile software development,” *Journal of Systems and Software*, vol. 85, no. 6, pp. 1213–1221, Jun. 2012, doi: 10.1016/J.JSS.2012.02.033.

[23] “Manifesto for Agile Software Development.” https://agilemanifesto.org/ (accessed Sep. 26, 2022).

[24] C. C. Huang and A. Kusiak, “Overview of Kanban systems,” *https://doi.org/10.1080/095119296131643*, vol. 9, no. 3, pp. 169–189, Jan. 2010, doi: 10.1080/095119296131643.

[25] “Jira Software - Features | Atlassian.” https://www.atlassian.com/software/jira/features?tab=kanban (accessed Sep. 27, 2022).

[26] M. Rehkopf, “User Stories | Examples and Template | Atlassian,” *ATLASSIAN Agile Coach*. https://www.atlassian.com/agile/project-management/user-stories (accessed Jul. 06, 2022).

[27] Broadband Search, “Key Internet Statistics to Know in 2022 (Including Mobile) - BroadbandSearch.” https://www.broadbandsearch.net/blog/internet-statistics (accessed Jul. 01, 2022).

[28] “History of the Web – World Wide Web Foundation.” https://webfoundation.org/about/vision/history-of-the-web/ (accessed Jul. 01, 2022).

[29] R. K. Atkinson, K. Sabo, and Q. Conley, “The participatory web,” *Handbook of Technology in Psychology, Psychiatry and Neurology: Theory, Research, and Practice*, pp. 91–120, Aug. 2012, doi: 10.1080/1369118X.2012.665935.

[30] S. Aghaei, M. A. Nematbakhsh, and H. K. Farsani, “EVOLUTION OF THE WORLD WIDE WEB: FROM WEB 1.0 TO WEB 4.0,” *International Journal of Web & Semantic Technology (IJWesT)*, vol. 3, no. 1, 2012, doi: 10.5121/ijwest.2012.3101.

[31] Follows T, “What Even Is Web5?,” *FORBES*. https://www.forbes.com/sites/traceyfollows/2022/06/11/what-even-is-web5/?sh=1901b8db5ad2 (accessed Jul. 01, 2022).

[32] B. Hemmenway Falk and S. Hammer, “Meltdown in the Wild West: The Stablecoin Collapse of 2022 and Consumer Protection Considerations by Brett Hemenway Falk, Sarah Hammer :: SSRN,” *SSRN*, May 24, 2022. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4119627 (accessed Jul. 01, 2022).

[33] M. M. Alani, “OSI model,” *SpringerBriefs in Computer Science*, vol. 0, no. 9783319051512, pp. 5–17, 2014, doi: 10.1007/978-3-319-05152-9_2/COVER/.

[34] T. J. Socolofsky and C. J. Kale, “TCP/IP tutorial,” Jan. 1991, doi: 10.17487/RFC1180.

[35] R. Fielding *et al.*, “Hypertext Transfer Protocol -- HTTP/1.1,” Jun. 1999, doi: 10.17487/RFC2616.

[36] E. Nikulchev, D. Ilin, and A. Gusev, “Technology Stack Selection Model for Software Design of Digital Platforms,” *Mathematics 2021, Vol. 9, Page 308*, vol. 9, no. 4, p. 308, Feb. 2021, doi: 10.3390/MATH9040308.

[37] R. T. Fielding and G. Kaiser, “Collaborative work: The apache http server project,” *IEEE Internet Comput*, vol. 1, no. 4, pp. 88–90, 1997, doi: 10.1109/4236.612229.

[38] A. Grosskurth and M. W. Godfrey, “Architecture and evolution of the modern web browser”.

[39] M. Jazayeri, “Some trends in Web application development,” *FoSE 2007: Future of Software Engineering*, pp. 199–213, 2007, doi: 10.1109/FOSE.2007.26.

[40] H. M. Abdullah and A. M. Zeki, “Frontend and backend web technologies in social networking sites: Facebook as an example,” *Proceedings - 3rd International Conference on Advanced Computer Science Applications and Technologies, ACSAT 2014*, pp. 85–89, Apr. 2014, doi: 10.1109/ACSAT.2014.22.

[41] Stack Overflow, “Developer Survey 2022,” 2022. https://survey.stackoverflow.co/2022/#developer-profile-developer-roles (accessed Jul. 01, 2022).

[42] F. Xinyang, S. Jianjing, and F. Ying, “REST: An alternative to RPC for web services architecture,” *2009 1st International Conference on Future Information Networks, ICFIN 2009*, pp. 7–10, 2009, doi: 10.1109/ICFIN.2009.5339611.

[43] “Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST).” https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (accessed Sep. 27, 2022).

[44] “The impact of Covid-19 on the Internet of Things |.” https://iot-analytics.com/the-impact-of-covid-19-on-the-internet-of-things/ (accessed Sep. 27, 2022).

[45] “How the pandemic disrupts the chip supply chain Insights | Bloomberg Professional Services.” https://www.bloomberg.com/professional/blog/how-the-pandemic-disrupts-the-chip-supply-chain/ (accessed Sep. 27, 2022).

[46] D. A. Leon *et al.*, “The Russian invasion of Ukraine and its public health consequences,” *The Lancet Regional Health – Europe*, vol. 15, Apr. 2022, doi: 10.1016/J.LANEPE.2022.100358.

[47] A. Alper, “Exclusive: Russia’s attack on Ukraine halts half of world’s neon output for chips | Reuters.” https://www.reuters.com/technology/exclusive-ukraine-halts-half-worlds-neon-output-chips-clouding-outlook-2022-03-11/ (accessed Jul. 01, 2022).

[48] H. Mohammad, “Number of connected IoT devices growing 18% to 14.4 billion globally,” *IOT ANALYTICS*, 2022. https://iot-analytics.com/number-connected-iot-devices/ (accessed Jul. 01, 2022).

[49] V. Singhania, “The Internet of Things: An Overview Understanding the Issues and Challenges of a More Connected World,” *Internet Society*.

[50] T. Samizadeh Nikoui, A. M. Rahmani, A. Balador, and H. Haj Seyyed Javadi, “Internet of Things architecture challenges: A systematic review,” *International Journal of Communication Systems*, vol. 34, no. 4, Mar. 2021, doi: 10.1002/DAC.4678.

[51] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC ’12*, 2012, doi: 10.1145/2342509.

[52] O. Bello, S. Zeadally, and M. Badra, “Network layer inter-operation of Device-to-Device communication technologies in Internet of Things (IoT),” *Ad Hoc Networks*, vol. 57, pp. 52–62, Mar. 2017, doi: 10.1016/J.ADHOC.2016.06.010.

[53] L. Tightiz and H. Yang, “A comprehensive review on IoT protocols’ features in smart grid communication,” *Energies*, vol. 13, no. 11. MDPI AG, Jun. 01, 2020. doi: 10.3390/en13112762.

[54] IBM and Eurotech, “MQTT V3.1 Protocol Specification,” *IBM*. https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html (accessed Jul. 01, 2022).

[55] The HiveMQ TEam, “MQTT Topics, Wildcards, & Best Practices - MQTT Essentials: Part 5,” Aug. 20, 2019. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/ (accessed Jul. 01, 2022).

[56] W. Yu *et al.*, “A Survey on the Edge Computing for the Internet of Things,” *IEEE Access*, vol. 6, pp. 6900–6919, Nov. 2017, doi: 10.1109/ACCESS.2017.2778504.

[57] R. Krishnamurthi, A. Kumar, D. Gopinathan, A. Nayyar, and B. Qureshi, “An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques,” *Sensors 2020, Vol. 20, Page 6076*, vol. 20, no. 21, p. 6076, Oct. 2020, doi: 10.3390/S20216076.

[58] A. Rayes and S. Salam, “The Things in IoT: Sensors and Actuators,” *Internet of Things from Hype to Reality*, pp. 63–82, 2022, doi: 10.1007/978-3-030-90158-5_3.

[59] F. Samie, L. Bauer, and J. Henkel, “IoT technologies for embedded computing: A survey,” *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016, Accessed: Jul. 01, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7750968

[60] ESPRESSIF, “ESP32 Wi-Fi & Bluetooth MCU I Espressif Systems.” https://www.espressif.com/en/products/socs/esp32 (accessed Jul. 01, 2022).

[61] “What is user story? | TIGO Software Solutions.” https://tigosoftware.com/what-user-story (accessed Jul. 09, 2022).

[62] “How using Templates for Jira Issues can help your ... - Atlassian Community.” https://community.atlassian.com/t5/Marketplace-Apps-Integrations/How-using-Templates-for-Jira-Issues-can-help-your-Agile-Teams/ba-p/1014179 (accessed Sep. 27, 2022).

[63] “Angular - Introduction to Angular concepts.” https://angular.io/guide/architecture (accessed Jul. 09, 2022).

[64] “Introduction - ngx-charts.” https://swimlane.gitbook.io/ngx-charts/ (accessed Jul. 09, 2022).

[65] “RxJS - Introduction.” https://rxjs.dev/guide/overview (accessed Jul. 09, 2022).

[66] “Angular Material UI component library.” https://material.angular.io/ (accessed Jul. 09, 2022).

[67] “Components | Angular Material.” https://material.angular.io/components/categories (accessed Jul. 09, 2022).

[68] R. Anderson and K. Larkin, “Tutorial: Create a web API with ASP.NET Core | Microsoft Docs,” 2022. https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-6.0&tabs=visual-studio (accessed Jul. 17, 2022).

[69] “Eclipse Mosquitto.” https://mosquitto.org/ (accessed Jul. 18, 2022).

[70] “DS18B20 Waterproof Digital Temperature Sensor Probe – VORDEO.” https://vordeo.com/shop/electronic-components-supplies/module-components/electronic-components-supplies-module-components/ds18b20-waterproof-digital-temperature-sensor-probe/?gclid=CjwKCAjwrNmWBhA4EiwAHbjEQKYcRA4JlpEPLiagDrSheg5GgjfIUvgDN_ue4_lchGPBL4pSNrzO-BoCi0EQAvD_BwE (accessed Jul. 18, 2022).

[71] “Analog Capacitive Soil Moisture Sensor V1.2 Corrosion Resistant: Amazon.com: Industrial & Scientific.” https://www.amazon.com/Analog-Capacitive-Moisture-Corrosion-Resistant/dp/B07N11R8MD (accessed Jul. 18, 2022).

[72] “Amazon.com: HiLetgo DHT21 AM2301 Capacitive Digital Temperature Humidity Sensor : Industrial & Scientific.” https://www.amazon.com/HiLetgo-Capacitive-Digital-Temperature-Humidity/dp/B00M1PMIKW (accessed Jul. 18, 2022).

[73] “ESP32-DevKitC Board I Espressif.” https://www.espressif.com/en/products/devkits/esp32-devkitc (accessed Jul. 18, 2022).

ANEXOS

ANEXO A. Actualizaciones del Tablero Kanban

ANEXO B. Repositorio de Código del Componente Base de Datos

ANEXO C. Repositorio de Código del Componente Recolector IoT

ANEXO D. Repositorio de Código del Software Embebido

ANEXO E. Repositorio de Código del Servicio Web

ANEXO F. Repositorio de Código de la App Front-End

ANEXO G. Fotos del Ambiente de Prueba en Escenarios 1 y 2