Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

DISEÑO E IMPLEMENTACIÓN DE UNA APLICACIÓN DE RECOGIDA DE DATOS EN INTERNET DE LAS COSAS

Alumno: Pierre-Simon Callist Yannick Tondreau

Tutor: Joaquín Cañada Bago
Dpto: Ingeniería de Telecomunicación

Julio, 2019

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Joaquín Cañada Bago, tutor del Proyecto Fin de Carrera titulado: Diseño e Implementación de una Aplicación de Recogida de Datos en Internet de las Cosas, que presenta Pierre-Simon Callist Yannick Tondreau, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Julio de 2019

El alumno: Los tutores:

CAÑADA BAGO JOAQUIN - 25969508R

Firmado digitalmente por CAÑADA BAGO JOAQUIN - 25969508R
Nombre de reconocimiento (DN): c=ES, serialNumber=25969508R, sn=CAÑADA BAGO, givenName=JOAQUIN, cn=CAÑADA BAGO JOAQUIN - 25969508R
Fecha: 2019.07.18 15:18:46 +02'00'

Pierre-Simon Callist Yannick Tondreau Joaquín Cañada Bago

Índice general

1. Introducción **8**

1.1. Introducción 8
1.2. Motivación 8
1.3. Objetivos 9

2. Estado del Arte **10**

2.1. Introducción 10
2.2. Dispositivos 10
2.2.1. Microcontroller Development Boards 10
2.2.2. Single Board Computers 11
2.3. Arquitectura 12
2.3.1. Cloud Computing 12
2.3.2. Fog Computing 13
2.4. Protocolos 13
2.5. Lenguajes de Programación 15
2.6. Bases de Datos 16
2.6.1. Bases de Datos Relacionales 16
2.6.2. NoSQL 17

3. Desarrollo del Proyecto **18**

3.1. Planificación 18
3.1.1. Estimación Temporal 18
3.1.2. Estimación de Costes 18
3.2. Arquitectura 20
3.3. Análisis 20
3.3.1. Identificación 20
3.3.2. Especificación de Requerimientos 21
3.4. Tecnologías a Usar 22
3.4.1. ArduinoJson 22
3.4.2. JsonCpp 23
3.4.3. Raspbian 23
3.4.4. Boost 24
3.4.5. MySQL 24
3.4.6. Apache 25

3.4.7. CodeIgniter 25
3.4.8. Bootstrap 27
3.4.9. Chart.js 27
3.5. Diseño 28
3.5.1. Comunicación Sensor y Servidor 29
3.5.2. Software Sensor 30
3.5.3. Software Servidor 31
3.6. Implementación 39
3.6.1. Software Sensor 39
3.6.2. Base de Datos 42
3.6.3. Servidor Principal 44
3.6.4. Página Web 48

4. Resultados **55**
4.1. Funcionamiento de la Aplicación 55
4.1.1. Ejecución código sensor 55
4.1.2. Recogida de Datos 56
4.2. Conclusiones 57

5. Líneas de Futuro **59**

Bibliografía **60**

Anexos **61**

A. Documentación Código **61**

B. Manual de Usuario **99**

Índice de figuras

2.1. Diagrama de los Pines de un Arduino UNO v3 11
2.2. Arquitectura Cloud 12
2.3. Arquitectura fog 13

3.1. Diagrama de Gantt 18
3.2. Serialización. 22
3.3. Deserialización. 22
3.4. Serialización. 23
3.5. Deserialización. 23
3.6. Patrón MVC. 26
3.7. Comunicación entre componentes MVC. 26
3.8. Ejemplo de gráfica con Chart.js. 28
3.9. Paso de mensajes TCP. 29
3.10. Estructura datos JSON. 29
3.11. Diagrama de clases UML 31
3.12. Diagrama de secuencia UML 33
3.13. ERD de la base de datos. 34
3.14. Tablas de la base de datos. 35
3.15. Storyboard de la página web. 36
3.16. Inicio. 37
3.17. Dispositivos. 37
3.18. Ubicacion. 38
3.19. Añadir nuevo elemento 38
3.20. Sketch vacío. 39
3.21. Diagrama del circuito. 40
3.22. Capacidad de nuestro documento JSON. 41

4.1. Resultado compilación. 55
4.2. Resultado de la subida del código. 55
4.3. Resultado ejecución código sensor. 55
4.4. Ejecución servidor principal recibiendo datos. 56
4.5. Resultado de consulta SELECT. 56
4.6. Datos insertados representados en la gráfica 57

B.1. Creación de la base de datos y usuario. 100

B.2. Página de prueba de Apache2. 101
B.3. Ejecución de make. 102
B.4. Comprobación de ejecución BBDD. 103
B.5. Comprobación de ejecución de apache. 104
B.6. Página Inicial. 105
B.7. Página Dispostivos. 105
B.8. Añadir elemento 106
B.9. Pestañas de medidas. 106
B.10.Seleccionador de fechas. 106
B.11.Intervalo de visualización 107

Índice de cuadros

2.1. Tabla de capas del modelo OSI, superior a inferior 14
2.2. Ejemplo de una tabla 16

3.1. Estimación costes materiales 19
3.2. Estimación costes humanos 19
3.3. Estimación costes materiales 19
3.4. Software por dispositivo 28
3.5. Estructura de los pasos. 42
3.6. Entorno de desarrollo C++. 44
3.7. Entorno de desarrollo Página web. 48

Índice de Códigos

1. Crear usuario y base de datos. 43
2. Creación de tablas. 43
3. Clase Acceptor. 45
4. Clase Handler. 46
5. Función handleRead 47
6. Clase DataManager. 48
7. Clase PHP del controlador. 49
8. Ejemplo modelo de medidas. 50
9. Carga de modelos en constructor del controlador. 51
10. Ejemplo llamada función de un modelo 51
11. Llamada AJAX para datos de gráfica. 54
12. Cabeceras del software del sensor. 61
13. Función setup. 63
14. Inicio función loop. 64
15. Contenido if principal. 65
16. Conexión servidor NTP. 66
17. Cabecera clase Acceptor 68
18. Constructor clase Acceptor 69
19. Función startAccept 69
20. Función handleAccept 70
21. Cabecera clase Handler 70
22. Función create 71
23. Función start 71
24. Función handleRead 72
25. Clase DataManager 73
26. Obtener instancia DataManager 74
27. Constructor DataManager 74
28. Función insertDato 75
29. Función insertFromJSON 76
30. Función getDispositivo 77
31. Función getDispositivo 77
32. Función getUbicacion 78
33. Función actualizaUbicacionDispositivo 79
34. Función getMedida 79

35. Función index 82
36. Función dispositivos 83
37. Función paginacionDatos 84
38. Función removeTabData 85
39. Función addTabData 86
40. Función graphData 87
41. Función addMedida 88
42. Función addDispositivo 89
43. Función getAllData 90
44. Función getDatosMedida 90
45. Función getDatosMedidaDispositivo 91
46. Función getDataList 91
47. Función getNumRows 92
48. Función getNumRowsMedida 92
49. Función getAllMedidas 93
50. Función getMedida 93
51. Función addMedida 93
52. Función getAllDispositivos 94
53. Función addDispositivo 94
54. Función getAllUbicacion 95
55. Funciones de visualización 95
56. Funciones mostrarUbicación 96
57. Funciones loadChart 97
58. Funciones getTree 98
59. Makefile 102

1 Introducción

1.1. Introducción

En esta sección se realizará un breve resumen sobre el contenido del documento, que muestra el diseño e implementación de una aplicación de recogida de datos en internet de las cosas.

Se comenzará explicando el estado actual del Internet de las Cosas, mostrando los dispositivos y las tecnologías que podemos encontrar en este tipo de proyectos. Los dispositivos generalmente se separan en dos grupos: sensores/actuadores y servidores.

Los sensores/actuadores son dispositivos con recursos muy limitados y cualquier operación que requiera de un capacidad de cómputo mayor se realizan normalmente en los servidores, que reciben la información de los sensores/actuadores. En este proyecto no nos encontraremos con ningún actuador, la finalidad del proyecto es recoger datos de sensores.

Otro aspecto importante es la comunicación que se realiza entre el sensor y el servidor, para ello existe una cantidad de protocolos distintos que han sido diseñados o que se han adaptado para su uso en proyectos de este tipo.

Los datos son almacenados en bases de datos, que pueden encontrarse dentro del alcance de nuestra red o fuera (Internet), de esta forma, nos encontramos con dos arquitecturas que pueden trabajar conjuntamente: fog computing y cloud computing.

La implementación del proyecto se ha realizado sobre un sistema Arduino, el sensor, y sobre un sistema Raspberry Pi, el servidor. Para conseguirlo, se han estudiado diferentes librerías y tecnologías que nos permiten cumplir los objetivos definidos en la propuesta del trabajo.

Finalmente, comentaremos los resultados obtenidos de la implementación realizada y las dificultades que se han presentado a la hora de desarrollar este proyecto.

1.2. Motivación

El término Internet de las Cosas hoy en día es cada vez más importante, dado que actualmente todo dispositivo que tengamos a mano requiere de una conexión a Internet. Mediante el Internet de las Cosas, todos los dispositivos estarían conectados a Internet, permitiéndonos interactuar con ellos o que recojan información para nosotros.

Que unos dispositivos recojan información del entorno para posteriormente analizarla o visualizarla es uno de los usos más extendidos de los proyectos en IoT.

1.3. Objetivos

El objetivo del proyecto es desarrollar una aplicación que permita la recogida de datos del entorno, el alcance de la aplicación será el de una red de área local, por lo tanto los datos que se recogerán no saldrán a Internet.

Los datos serán recogidos por un sensor y los enviará a un servidor central, que se encargará de verificar la autenticidad del sensor y de almacenar los datos recibidos en una base de datos.

Adicionalmente, se permitirá la visualización de los datos recogidos en una página web.

2 Estado del Arte

2.1. Introducción

En el año 1991 Mark Weiser presentó un artículo que iniciaba el paradigma de computación ubicua [1]. La computación ubicua trata de extender la capacidad computacional al entorno del usuario y se reconoce como el objetivo a seguir del IoT *Internet of Things*). Pero el término IoT no aparece hasta el año 1999, introducido por Kevin Ashton [2], que define un sistema donde el mundo físico está conectado a Internet por medio de sensores ubicuos.

2.2. Dispositivos

Los dispositivos que se usan para realizar proyectos IoT, generalmente son dispositivos pequeños, con poca memoria, poca capacidad de cómputo, pero eso es lo que los hace tan buenos para dichos proyectos. Debido a que el hardware es tan limitado, los dispositivos terminan siendo pequeños, con un consumo de energía bajo y baratos. Además, tienen la posibilidad de conectar sensores para obtener datos sobre su entorno y más adelante realizar una acción con dichos datos (mediante actuadores), o sólo almacenarla (en un servidor).

Actualmente, los dispositivos más usados en proyectos IoT son los Microcontroller Development Boards y los Single Board Computers, también se pueden usar conjuntamente, debido a que ambos dispositivos o placas tienen objetivos distintos.

2.2.1. Microcontroller Development Boards

Los microcontroladores son unos SoC (*System-on-a-Chip*) que proporcionan memoria y capacidad de cómputo. Contienen un procesador, memoria (RAM), y memoria reprogramable (EPROM) para almacenar los programas que va a ejecutar el microcontrolador. Los *Microcontroller development boards* son PCBs (*Printed Circuit Board*) con circuitería adicional para incorporar el microcontrolador y poder realizar la programación y prototipado de una forma más conveniente. Los sensores y actuadores se conectan al microcontrolador mediante Pines de Entrada/Salida de Propósito General (GPIO, *General Purpose Input/Output*) [3].

Algunas placas de desarrollo disponibles actualmente son: **Arduino UNO, Node MCU (ESP 8266), Adafruit Feather HUZZAH ESP8266**.

A continuación hablaremos más en detalle sobre la placa de desarrollo **Arduino UNO**.

La placa Arduino UNO es una placa de microcontrolador de código abierto basado en el microcontrolador Microchip ATmega328P y desarrollada por Arduino [4]. El procesador de dicho microcontrolador mantiene una arquitectura de 8 bits y su conjunto de instrucciones se basa en RISC (*Reduced instruction set computer*). La memoria que incluye es de tres tipos:

- **Memoria Flash**: 32KB de memoria no volátil. Esta memoria es usada para almacenar las aplicaciones que desarrollamos.
- **Memoria SRAM**: 2KB de memoria volátil. Esta memoria es usada para almacenar las variables de la aplicación en tiempo de ejecución.
- **Memoria EEPROM**: 1KB de memoria no volátil. Esta memoria es usada para almacenar datos que deben de estar disponibles incluso después de apagar la placa.

Además, la placa tiene 14 pines de E/S digitales y 6 pines de entrada analógica, mediante los cuales podemos conectar sensores y actuadores a la placa.

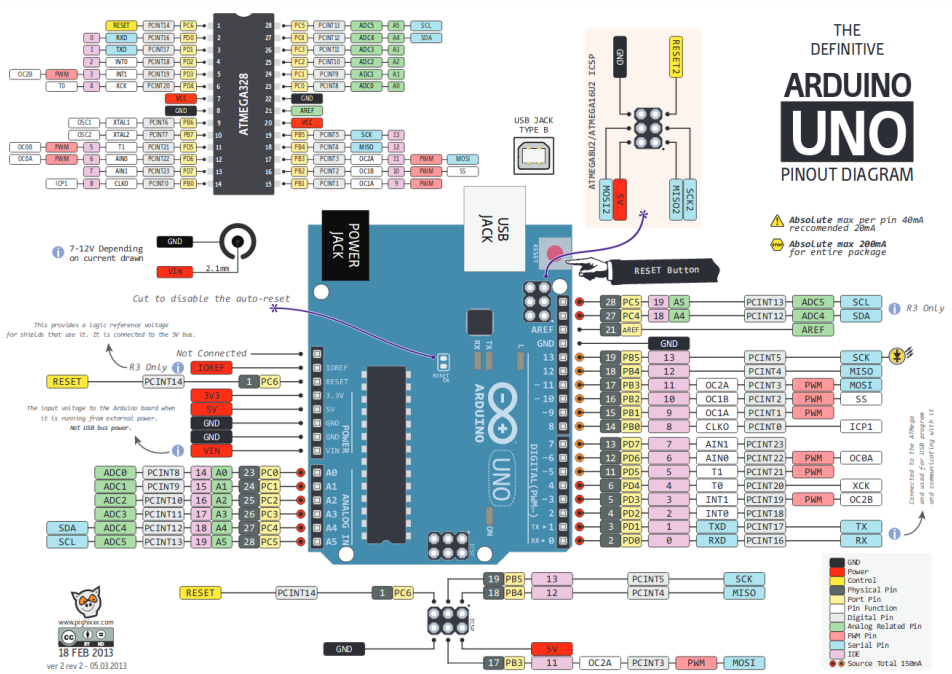


Figura 2.1: Diagrama de los Pines de un Arduino UNO v3

2.2.2. Single Board Computers

Los SBCs son ordenadores completos montados en una sola placa, incluyen microprocesador, memoria, puertos de entrada/salida para conectar periféricos, sensores o

actuadores. Ofrecen mayor capacidad de cómputo que las placas de microcontroladores, por ejemplo, un microprocesador a 1.4GHz con arquitectura ARM de 64-bits de una Raspberry Pi 3B+ [5] frente a un microprocesador a 20Mhz con arquitectura AVR de 8-bits [6] de un Arduino UNO R3.

La Raspberry Pi, es una serie de SBCs desarrollada por la Raspberry Pi Foundation. El primer modelo lanzado por la compañía es la Raspberry Pi 1 Model B en febrero de 2012, y actualmente el modelo más reciente es la Raspberry Pi 3 Model B+, lanzada en marzo 2018. Todos los modelos de Raspberry Pi incluyen un procesador con arquitectura ARM.

ARM (*Advanced RISC Machine*) es una familia de arquitecturas RISC para procesadores desarrollada por ARM Holdings, dicha compañía realiza el diseño de la arquitectura del procesador para más adelante vender una licencia de la arquitectura a otras empresas. Al hacer uso de un conjunto de instrucciones RISC, permite que el diseño de los procesadores requieran una cantidad menor de transistores, reduciendo de esta forma los costes, el calor y la energía que pueda producir el procesador, esto los hace ideales para su uso en aplicaciones de baja potencia. Actualmente, los procesadores ARM se usan en la mayoría de los dispositivos móviles que podemos encontrar.

2.3. Arquitectura

2.3.1. Cloud Computing

La computación en la nube permite ofrecer servicios como almacenamiento de datos y capacidad de procesamiento a través de Internet. En IoT es un paradigma que se usa mucho dados los recursos limitados que tienen los dispositivos.

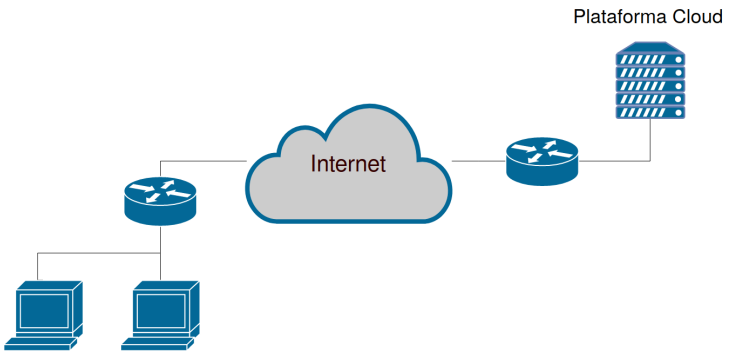


Figura 2.2: Arquitectura Cloud

2.3.2. Fog Computing

La computación en la niebla permite ofrecer los mismos servicios que la computación en la nube, pero a nivel de área local. En este paradigma podemos encontrar un dispositivo con mayores capacidades de cómputo y memoria en una red IoT, que procesa o almacena la información recibida.

Ambas arquitecturas pueden ser usadas conjuntamente, permitiendo mandar información al servidor de computación fog, para que este posteriormente la envíe a una plataforma en la nube.

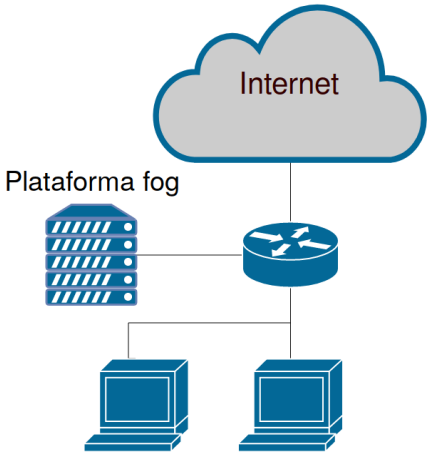


Figura 2.3: Arquitectura fog

2.4. Protocolos

La comunicación entre dispositivos IoT se realiza mediante el uso de diversos protocolos de comunicación que podemos estructurar en base al modelo OSI (*Open System Interconnection*).

El modelo OSI es un lineamiento funcional para tareas de comunicaciones y, por consiguiente, no especifica de un estándar de comunicación para dichas tareas. Sin embargo, muchos estándares y protocolos cumplen con los lineamientos del modelo OSI. Mantiene una estructura multinivel con la idea que cada nivel se dedica a resolver una parte del problema de comunicación, por lo tanto cada nivel ejecuta funciones distintas, pero a su vez cada nivel es dependiente del nivel inferior y del nivel superior.

En la Tabla 2.1 tenemos información sobre las capas que componen el modelo OSI, además de una breve descripción de su función e incluyendo ejemplos de protocolos que operan a dicho nivel.

A continuación explicaremos brevemente los protocolos que son usados en muchos de los proyectos IoT que se están desarrollando actualmente, ordenados de capa de menor nivel a mayor nivel.

Capa	Descripción	Protocolos
Aplicación	Proporciona servicios a las aplicaciones terminales. Proporciona y obtiene datos de la capa de presentación.	HTTP, HTTPS, DHCP, etc
Presentación	Responsable de formatear y entregar datos a la capa de aplicaci ón para su procesamiento. También se encarga de encriptar y desencriptar en caso que sea necesario.	
Sesión	Proporciona el mecanismo necesario para abrir, cerrar y administrar una sesión entre procesos de una aplicación terminal.	H.245, RTCP, SDP, etc
Transporte	Encargado de entregar información a través de las redes, proporciona servicios de comunicación de punto a punto para las aplicaciones terminales.	TCP, UDP, SCTP, etc
Red	Proporciona los medios para transferir datos de longitud variable, llamados paquetes, de un punto a otro pasando por una o más redes.	IPv4, IPv6, ICMP, etc
Enlace de Datos	Transfiere datos entre puntos de redes adyacentes o entre puntos de la misma red. Proporciona los medios para transferir datos entre redes y detectar errores que pueden ocurrir entre la capa física.	Ethernet, IEEE 802.11 (Wi-Fi), ATM, etc
Física	Responsable de la transmisión de datos en bits a través de una enlace físico que conecta los nodos de la red.	Bluetooth, LoRa, USB, etc

Cuadro 2.1: Tabla de capas del modelo OSI, superior a inferior

1. **802.15.4** Más que un protocolo es un estándar que define el nivel físico y control de acceso al medio en redes LR-WPANs (*Low Rate Wireless Personal Area Network*). Es la base de Zigbee y 6LoWPAN, las cuales extienden el estándar desarrollando las capas superiores [7].

2. **TCP** Es uno de los protocolos principales de la familia de protocolos de internet.

Proporciona una conexión de host a host a nivel de transporte, las conexiones pueden dar comportamiento impredecibles, pero en caso de errores TCP lo detecta y pide el reenvío de dichos paquetes que se pueden haber perdido.

3. **IP** Es uno de los protocolos principales de la familia de protocolos de internet. Este protocolo es el responsable de encapsular los datos en datagramas y enrutarlos desde una dirección origen a una dirección destino. Actualmente existen dos versiones del protocolo IPv4 e IPv6.

4. **MQTT** Protocolo de paso de mensajes basado en suscripciones que trabaja sobre TCP. Actualmente, se ha convertido en el estándar de las comunicaciones IoT dado a que es un protocolo ligero y flexible. En el protocolo se definen dos entidades: el broker y los clientes, los clientes se conectan al broker suscribiéndose a un topic (tema), los clientes pueden enviar mensajes bajo un topic hacia el broker y luego los reenvía a los clientes que estén suscritos a dicho topic.

5. **HTTP** HyperText Transfer Protocolo (HTTP) es un protocolo de nivel de aplicación para sistemas distribuidos, colaborativos de información hypermedia [8]. Es una de las bases de la World Wide Web, usa un modelo cliente - servidor, donde nuestro navegador web es el cliente y comunica con un servidor que aloja una página web. En 2015 se publicó HTTP/2.

2.5. Lenguajes de Programación

Un lenguaje de programación proporciona una serie de instrucciones que permiten escribir secuencias de órdenes y algoritmos a modo de controlar el comportamiento físico y lógico. Los lenguajes de progrmación más usados actualmente en proyectos IoT son los siguientes: C/C++, Java y Python.

C/C++

El lenguaje de programación C/C++ en proyectos IoT, es sin duda el más usado actualmente, esto se debe principalmente a la flexibilidad que proporciona el lenguaje de programación. Dado que se trabaja con dispositivos con recursos restringidos, si se implementa una función compleja, C/C++ nos da mayor control sobre los recursos del dispositivo. Además, el lenguaje de programación viene por defecto con el IDE de Arduino que es compatible con diversas placas de desarrollo con las librerías adecuadas.

2.6. Bases de Datos

Una base de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso, en otras palabras, una base de datos se usa como una forma electrónica de almacenar, administrar y obtener información.

Para una aplicación de recogida de datos una base de datos es recomendable, incluso obligatorio. Es necesario que una infraestructura almacene los datos que se recogen para mantener un historial de los datos, o para realizar un procesamiento posterior de la información, por lo tanto, dicha infraestructura no es la misma para cualquier aplicación.

Existen diferentes tipos de bases de datos, pero actualmente las más usadas son las bases de datos **relacionales** y **NoSQL**.

2.6.1. Bases de Datos Relacionales

Las bases de datos relacionales siguen el modelo relacional postulado por Edgar Frank Codd, quién publicó sus bases en el año 1970 [9]. El sistema encargado de manejar una base de datos relacional se denomina RDBMS (*Relational Database Management System*).

En una base de datos relacional los datos se almacenan en tablas, en las cuales cada columna de la tabla representa un atributo de los datos, y cada fila (también llamado tupla) representa una entrada de datos. Cada tabla tiene un atributo o par/es de atributos que representan la clave, que no se puede repetir. La clave usada en el cuadro 2.2 es el ID, en caso de haber usado el apellido, la segunda entrada de la tabla sobreescribiría el primer elemento dado que tienen el mismo apellido.

ID	Nombre	Apellidos
1	John	Doe
2	Jane	Doe

Cuadro 2.2: Ejemplo de una tabla

En este proyecto usaremos MySQL, que junto a Oracle y Microsoft SQL Server son los RDBMS más usados [10].

2.6.2. NoSQL

NoSQL (*Not only SQL*) nace como respuesta al aumento en el volumen de datos almacenados, la frecuencia con la que se accede y las necesidades de potencia y almacenamiento. Agrupa una gran variedad de diferentes tecnologías que hacen frente a los problemas de escalado y agilidad que se presentan en las bases de datos relacionales. Uno de los sistemas de gestión de bases de datos NoSQL más usados actualmente es MongoDB.

3 Desarrollo del Proyecto

El trabajo persigue el diseño e implementación de una aplicación de almacenamiento de datos que se ejecutará en dispositivos con recursos restringidos (cliente y servidor). En esta sección se mencionan todos los elementos que han sido necesarios para obtener una implementación final del proyecto.

3.1. Planificación

En esta sección realizaremos una planificación sobre los recursos externos adicionales para realizar el proyecto y mostraremos en un diagrama de Gantt el tiempo que ha sido necesario para realizar el proyecto.

3.1.1. Estimación Temporal

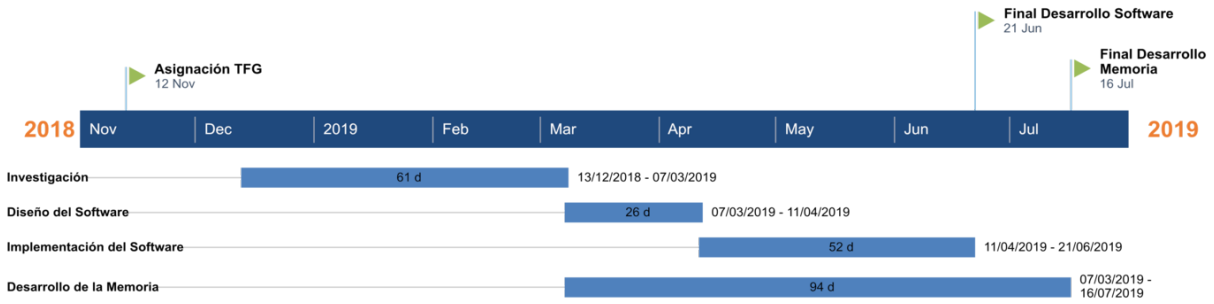


Figura 3.1: Diagrama de Gantt

3.1.2. Estimación de Costes

A continuación se presentarán en tablas los costes materiales y los costes humanos que serán requeridos para el desarrollo del proyecto.

Costes Materiales

Los costes materiales incorporan todos aquellos elementos que son necesarios para el desarrollo del proyecto, como elementos hardware, software, conexión a Internet, etc.

Recurso Hardware	Precio
Raspberry Pi 3B+	36 €
Arduino Uno Rev. 3	20 €
Arduino Ethernet Shield 2	22 €
Sensor Digital DHT22	8 €
Cables de conexión	10 €

Cuadro 3.1: Estimación costes materiales

Adicionalmente, además de los costes mostrados en la Tabla 3.3, se requerirá de una conexión activa a Internet para poder realizar la investigación necesaria, dicho coste asciende a 30€/mes.

Costes Humanos

Los costes humanos incorporan el coste del personal que se dedica a desarrollar el proyecto. En este proyecto todas las tareas involucradas en el proceso de desarrollo serán conducidas por una misma persona, Pierre Simon Callist Yannick Tondreau.

Los salarios mostrados a continuación son recogidos desde la página web Indeed [11].

Puesto	Coste Anual	Coste Mensual	Coste por Hora
Desarrollador de Software	25.707€/año	1.786€/mes	11,61€/hora

Cuadro 3.2: Estimación costes humanos

Teniendo en cuenta los costes de la Tabla 3.2 y de la Tabla 3.3, y estimando que el proyecto tendrá una duración de 5 meses, podemos establecer el coste total del proyecto como se muestra a continuación.

Recurso	Precio
Hardware	96 €
Conexión Internet	30€ x 5 meses = 150€
Recursos Humanos	11,61€ x 4 horas x 100 días = 4.644€
Total	4.890€

Cuadro 3.3: Estimación costes materiales

3.2. Arquitectura

Como hemos mencionado anteriormente en la estimación de costes, además de desarrollar el software necesario, también haremos uso de una placa Arduino y de una Raspberry Pi. La placa Arduino recogerá datos a través de un sensor y enviará dichos datos mediante una conexión TCP a la Raspberry Pi. Posteriormente, la Raspberry Pi almacenará los datos y también hospedará el servidor web para visualizar los datos desde nuestra red de área local.

La arquitectura con la que trabajaremos es una arquitectura cliente - servidor, en el caso de este proyecto el cliente (Arduino) envía datos a un servidor (Raspberry Pi) para que los almacene en la base de datos. Además de mantener una arquitectura cliente - servidor como modelo de diseño de software, también mantiene una arquitectura fog como modelo red.

3.3. Análisis

Para que el desarrollo de nuestro proyecto se realice con éxito, es importante tener una plena comprensión de los requisitos de este. En este apartado identificaremos los elementos a tener en cuenta a la hora de implementar, diseñar o usar nuestro software, para ello realizaremos un proceso de identificación y más adelante hablaremos sobre los requerimientos de nuestro software.

3.3.1. Identificación

A continuación se identificaran los elementos clave de nuestro proyecto, y mencionaremos los problemas que pueden surgir en el diseño o la implementación de dicho elemento.

El objetivo del proyecto es obtener una aplicación mediante la cual puedes conectar un sensor y obtener una representación gráfica de los datos que se están midiendo, para ello, hace falta lo siguiente:

- **Servidor Principal**: Este elemento del proyecto maneja la comunicación a través de la red con el sensor.
- **Base de Datos**: Este elemento del proyecto almacenará los datos recibidos por parte del sensor.
- **Servidor Web**: Este elemento del proyecto presentará la aplicación web.

Cada uno de estos servicios se ejecutará de forma idependiente, por lo tanto será necesario mantener una configuración correcta en todo momento para no obtener resultados indeseados.

Para la implementación de cada uno de los elementos mencionados se hará uso de librerías y software adicionales (Sección 3.4) que serán necesarios para cumplir con el objetivo del proyecto.

3.3.2. Especificación de Requerimientos

Los requerimientos se refieren al comportamiento que va a tener el software que vamos a desarrollar. Estos se pueden dividir en requisitos funcionales y no funcionales, los cuáles mecionaremos a continuación.

Requisitos Funcionales

Los requisitos funcionales describen cada una de las funciones que el software debe realizar. En el caso de este proyecto, los requisitos funcionales principales son los siguientes:

1. El sensor medirá datos, y los enviará al servidor.
2. El servidor almacenará datos que envíe un sensor.
3. El servidor presentará una aplicación web para visualizar los datos.
4. La aplicación web aceptará entradas de usuario para filtrar los datos de visualización.
5. La aplicación web permitirá añadir nuevos dispositivos y medidas.

Requisitos No Funcionales

Los requisitos no funcionales describen una serie de propiedad que añaden restricciones tanto en el diseño como en la implementación del software. En el caso de este proyecto, los requisitos no funcionales principales son los siguientes:

1. El servidor que comunica con el sensor, se realizará mediante conexiones TCP e implementado en C/C++.
2. Para la visualización de los datos se podrá seleccionar el rango de fechas y los dispositivos que se deseen.
3. La ubicación de los dispositivos se podrán visualizar en un mapa

3.4. Tecnologías a Usar

A continuación mostaremos los recursos adicionales que usaremos en nuestro proyecto para así cumplir con los requisitos que hemos definido anteriormente.

3.4.1. ArduinoJson

ArduinoJson es una librería escrita en C++ para Arduino y el Internet de las Cosas. La librería es compatible con una gran variedad de dispositivos que son usados para proyectos IoT, como el ESP8266, Teensy board, etc.

ArduinoJson nos ofrece realizar la serialización (Figura 3.2) y deserialización (Figura 3.3) en formato JSON mediante una API elegante y fácil de usar. La librería en si ocupa poco dado que solo contiene cabeceras y no mantiene ninguna dependencia externa con otra librería.

```
DynamicJsonDocument doc(1024);

doc["sensor"] = "gps";
doc["time"]   = 1351824120;

JsonArray data = doc.createNestedArray("data");
data.add(48.756080);
data.add(2.302038);

serializeJson(doc, Serial);
```

Figura 3.2: Serialización.

```
DynamicJsonDocument doc(1024);
deserializeJson(doc, json);

const char* sensor = doc["sensor"];
long time          = doc["time"];
double latitude    = doc["data"][0];
double longitude   = doc["data"][1];
```

Figura 3.3: Deserialización.

Hemos decidido usar esta librería dado que el formato JSON está muy extendido en el ámbito del Internet de las Cosas, también nos ofrece más libertad a la hora de componer los mensajes de datos que enviará el sensor. En caso de no usar JSON, tendríamos que diseñar e implementar una función para que parsee los mensajes que recibimos del sensor y poder extraer los datos de él.

3.4.2. JsonCpp

Jsoncpp es una librería escrita en C++ que permite la manipulación, la serialización (Figura 3.4) y deserialización (Figura 3.5) de objetos o valores JSON. Esta librería la usaremos para deserializar los mensajes en formato JSON que recibimos del sensor.

Es importante saber que no podemos usar la misma librería para los dos dispositivos en los que estará montados nuestro software. Para la placa Arduino usamos ArduinoJson porque es específica para su arquitectura y las restricciones que presenta el hardware, pero en la Raspberry Pi no estamos tan limitados como en la placa Arduino, permitiéndonos realizar operaciones adicionales con los datos.

```
Json::Value obj;
obj["sensor"] = "gps";

Json::Value array;
array.append(48.756080);
array.append(2.302038);

obj["data"] = array;

cout << obj.toStyledString() << endl;
```

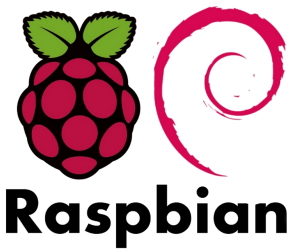
Figura 3.4: Serialización.

```
Json::Reader reader;
Json::Value obj;
reader.parse(datos, obj)

string sensor    = obj["sensor"].asString();
string time      = obj["time"].asString();
double latitude  = obj["data"][0].asDouble();
double longitude = obj["data"][0].asDouble();
```

Figura 3.5: Deserialización.

3.4.3. Raspbian



Raspbian es el sistema operativo basado en Debian desarrollado especialmente para Raspberry Pi. La Raspberry Pi Foundation lo ha declarado como el sistema operativo oficial de la plataforma desde 2015.

Dado que es el sistema operativo oficial es lógico usarlo, pero esto no significa que sea el único sistema operativo disponible. Además de Raspbian también tenemos la opción de usar Ubuntu, FreeBSD, Chromium OS, Windows 10 IoT Core, etc. Pero finalmente decidimos usar Raspbian dado que en comparación con el resto de sistemas operativos es más ligero, dándonos un mayor margen para memoria en la Raspberry Pi.

3.4.4. Boost

Boost es un conjunto de librerías para C++ que puede proporcionar una gran variedad de capacidades a nuestro software, como: algebra lineal, programación con hilos, procesamiento de imágenes, etc. Para nuestro proyecto haremos uso de dos librerías de Boost: **Boost.Asio** y **Boost.Program Options**.

Boost.Asio

Boost.Asio es una librería que nos permite procesar datos de forma asíncrona, esto significa que cuando se inicia un operación, no se espera hasta que se finalice para seguir con la ejecución del programa. Pero Boost.Asio no es la única librería que nos ofrece esta funcionalidad Boost.Thread, también permite ejecución asíncrona.

La diferencia entre Boost.Asio y Boost.Thread, es que la primera nos permite la ejecución asíncrona con recursos externos, como lo son las conexiones de red, y con la segunda librería la ejecución asíncrona se realiza sobre recursos de CPU.

Boost.Program Options

Boost.Program Options es una librería que hace fácil parsear opciones que se introducen en la línea de comandos, como lo es habitual para ejecutar aplicaciones desde la terminal. Dado que nuestra aplicación servidor no tendrá ninguna interfaz de usuario, se ha decidido usar esta librería para parsear las opciones que introducirá el usuario.

3.4.5. MySQL

MySQL es un administrador de bases de datos relacionales de código abierto. Como ya hemos mencionado anteriormente, es una de las bases de datos más usadas a nivel mundial.

Para el número de datos con el que vamos a operar, y para las operaciones que vamos a realizar MySQL nos ofrece las capacidades más que suficientes para nuestro proyecto.

Su puesta en marcha en sistemas Raspberry Pi con Raspbian es muy simple dado que el paquete del software se encuentra en los repositorios oficiales.

MySQL Connector/C++

MySQL Connector/C++ es la librería que nos ofrece MySQL para comunicar con bases de datos MySQL mediante C++. Dado que nuestra base de datos es MySQL, deberemos de usar esta librería para poder comunicarnos con la base de datos y permitirnos almacenar o leer datos.

Además de MySQL Connector/C++ existe otra librería llamada MySQL++ desarrollada por la comunidad que ofrece manejar bases de datos mediante los contenedor STL propios de C++. Hemos decidido usar MySQL Connector/C++ dado que es la solución que nos ofrece oficialmente MySQL.

3.4.6. Apache

Apache es un servidor HTTP de código abierto y multiplataforma desarrollado por la Apache Software Foundation, es uno de los servidores web más usados, en 2018 se ha estimado que el 39 % de las páginas webs activas son mantenidas por este servidor.

Apache nos ofrece una gran variedad de características, de las cuales la mayoría son implementados como módulos que extienden la funcionalidad del servidor, que varían desde esquemas de autenticación hasta el soporte de lenguajes de programación en el servidor como Perl, Python, PHP, etc.

La instalación y configuración en sistemas Raspberry Pi con Raspbian se encuentra en el Anexo B.

3.4.7. CodeIgniter

CodeIgniter es un framework de desarrollo de aplicaciones para crear páginas web usando PHP. Su objetivo es permitirnos desarrollar proyectos mucho más rápidamente frente a si se decide implementar el código desde cero. Para ello, ofrece un amplio conjunto de librerías que son comúnmente usadas, así como una interfaz simple y una estructura lógica para acceder a dichas librerías.

Model-View-Controller

El patrón de arquitectura en la cuál se basa CodeIgniter es **MVC** (*Model-View-Controller*), este patrón divide una aplicación en tres partes interconectadas, esto se realiza para separar la lógica de la aplicación con su representación hacia el usuario final.

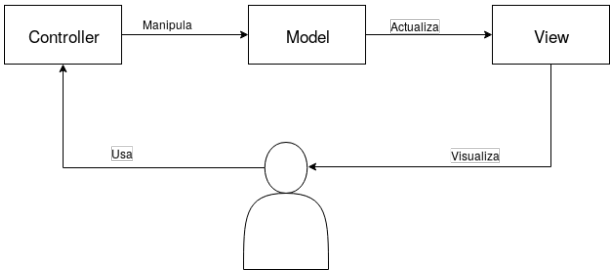


Figura 3.6: Patrón MVC.

- **Model**: Es el componente central del patrón, representa la estructura de datos de la aplicación, totalmente independiente de la interfaz de usuario.
- **View**: Representa la interfaz que se presenta al usuario.
- **Controller**: Conjunto de funciones que se ejecutarán dependiendo a la interacción que tiene el usuario con la página.

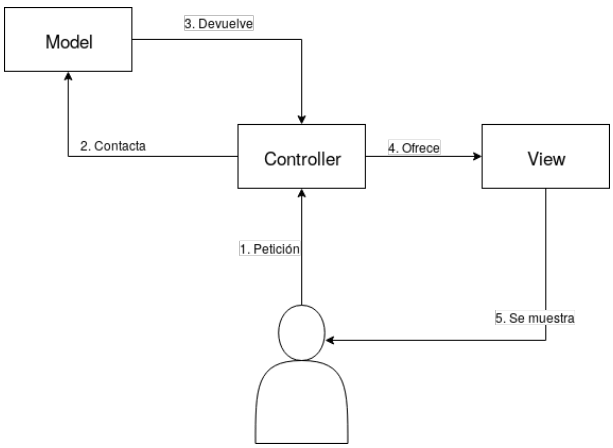


Figura 3.7: Comunicación entre componentes MVC.

3.4.8. Bootstrap

Bootstrap es una librería de código abierto para desarrollar aplicaciones web mediante el uso de código HTML, CSS y JavaScript. Esta librería nos permite establecer un diseño predefinido a nuestra aplicación web, sin nosotros tener que implementar todo el código CSS para darle estilo a la aplicación.

Además de Bootstrap, se usarán dos librerías adicionales que funcionan junto a Bootstrap, se pueden considerar como addons a la librería Bootstrap.

- **Tempus Dominus**: Ofrece un datepicker mediante el cuál obtendremos fechas de selección para definir un rango de representacón en las gráficas.
- **Treeview**: Ofrece un menú que se usará para añadir o eliminar dispositivos de nuestra selección.

3.4.9. Chart.js

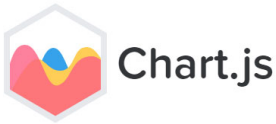


Chart.js es una librería en JavaScript que nos ofrece funciones y objetos mediante los cuáles podemos crear gráficas de diferentes tipos. Otra solución disponible es Google Charts, pero en mi opinión, Chart.js es más simple de usar.

Para nuestra aplicación sólo usaremos gráficas lineales, en las cuáles el eje X representa el tiempo, y el eje Y representa el valor de la medida tomada (Figura 3.8)

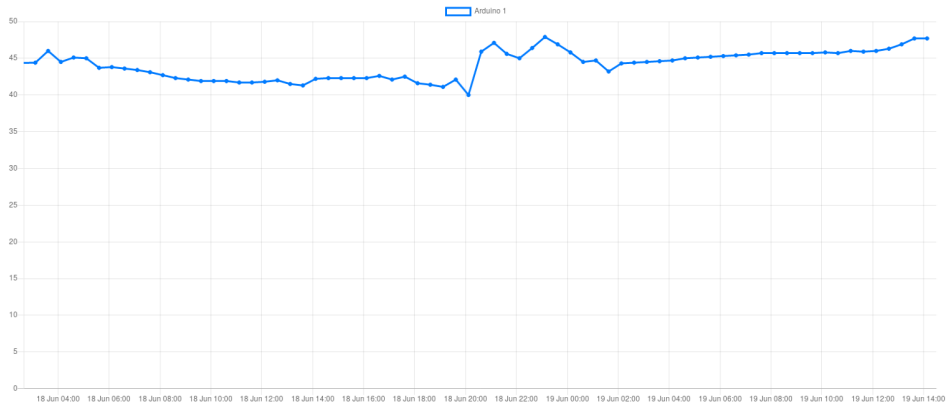


Figura 3.8: Ejemplo de gráfica con Chart.js.

3.5. Diseño

En la etapa de diseño traducimos los requerimientos funcionales y no funcionales en una representación software, este es el primer paso en la fase de desarrollo de nuestro proyecto.

Para nuestro proyecto disponemos de dos elementos hardware que usaremos para realizar la recogida de datos. Por un lado tenemos un dispositivo, que se encargará de obtener datos a través de un sensor y enviarlo mediante Ethernet a otro elemento hardware, el servidor, que se encargará de insertar dichos datos en una base de datos para su posterior visualización desde una página web.

De esta forma, podemos separar los componentes software por dispositivo en el que se ejecuta como se muestra en el cuadro 3.4.

Sensor	**Servidor**
Captura Datos	Servidor TCP
	Base de Datos
	Servidor Web

Cuadro 3.4: Software por dispositivo

3.5.1. Comunicación Sensor y Servidor

Para empezar es necesario tener claro la forma y la codificación en la que enviaremos y recibiremos los datos del sensor. Como ya se ha mencionado varias veces, para la comunicación entre el sensor y el servidor haremos uso del protocolo TCP, que nos ofrece las características que necesitamos para nuestra conexión. El protocolo TCP nos permite establecer conexiones, detectar errores y terminar conexiones, que es lo que necesitamos principalmente para nuestro caso.

Dado que el protocolo TCP se encargará de iniciar la conexión, el paso de mensajes y la desconexión, debemos de establecer la estructura del mensaje que va a enviar el sensor para que el servidor pueda extraer los datos, como podemos ver en la Figura 3.9.

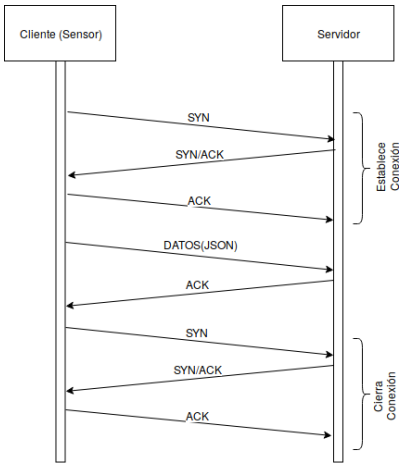


Figura 3.9: Paso de mensajes TCP.

```
{
  "id": 1,
  "nombre": "Arduino",
  "dato": [{
    "medida": "Temp",
    "valor": 22.4,
    "unidad": "celsius",
    "timestamp": 1559752158
  }],
  "ubicacion": {
    "lat": 0.00000,
    "lon": 0.00000
  }
}
```

Figura 3.10: Estructura datos JSON.

El paquete de datos que llevará el protocolo TCP será un mensaje en formato JSON (*JavaScript Object Notation*) con la estructura que se puede apreciar en la Figura 3.10. Se ha decidido usar este formato (JSON) dado que su soporte se encuentra en casi todos los lenguajes de programación permitiendo una serialización y deserialización simple y eficiente. Los atributos que componen la estructura del objeto JSON son los siguientes:

- **id**: ID del sensor, por ejemplo: 1.
- **nombre**: Nombre que se le ha asignado al sensor, por ejemplo: “Sensor Humedad Exterior”
- **modelo**: Modelo del dispositivo que recoge datos, por ejemplo: “Arduino Uno Rev. 3”.
- **dato**: Array de objetos JSON que contiene los valores que ha medido el sensor, dependiendo del número de medidas que realice el sensor existirá uno o más elementos en el array.

  - **medida**: Nombre de la medida, por ejemplo: “Temp”.
  - **valor**: Valor de la medida, por ejemplo: 22.4.
  - **unidad**: Unidad en la que se ha medido el valor, por ejemplo: “celsius”.
  - **timestamp**: Marca de tiempo desde el epoch, número de segundos que han pasado desde el 1 de enero de 1970.

- **ubicacion**: Contiene la latitud y la longitud, indicando la ubicación del sensor.

3.5.2. Software Sensor

El software del sensor no requiere de un gran proceso de diseño. Su función será la de recoger datos del entorno para enviarlas mediante una conexión TCP a un servidor. Su software puede ser diseñado mediante el siguiente proceso:

1. Obtener la medida del entorno que se le ha asignado al sensor.
2. Conectará con el servidor TCP de la Raspberry Pi para enviarle los datos que ha medido, si el envío se realiza correctamente entonces se espera un tiempo para volver a enviar datos, en caso contrario vuelve a intentar actualizando el dato. Los datos serán enviados con la estructura que se define en *Comunicacion Sensor y Servidor*.
3. El paso dos se repite hasta que se apague el dispositivo de forma manual.

Como se puede observar, su función es bastante simple, esto se debe en gran parte a lo restrictivo que es la placa Arduino en cuanto a capacidad de procesamiento y memoria, por ello el resto del procesamiento se realiza en el servidor.

3.5.3. Software Servidor

A continuación mostraremos el proceso de diseño que hemos seguido para los diferentes componentes software que mantienen/componen el software del servidor.

Servidor Principal

Empezaremos con el servidor principal, con esto nos referimos al componente software que se encargará de recibir los datos del sensor mediante el protocolo TCP y de insertar los datos en la base de datos.

Para diseñar dicho componente software haremos uso de dos tipos de diagramas UML (*Unified Modelling Language*), que es un lenguaje de modelado que persigue el objetivo de visualizar el diseño de un sistema. Principalmente se mostrará el diagrama de clases que describe la estructura de nuestro software, y a continuación se presentará el diagrama de secuencia para describir el paso de mensajes.

En la Figura 3.11 se presenta el diagrama de clases en UML del componente software. Como podemos observar tenemos tres clases y cuatro estructuras, cada clase tiene una función a completar en el proceso de la comunicación.

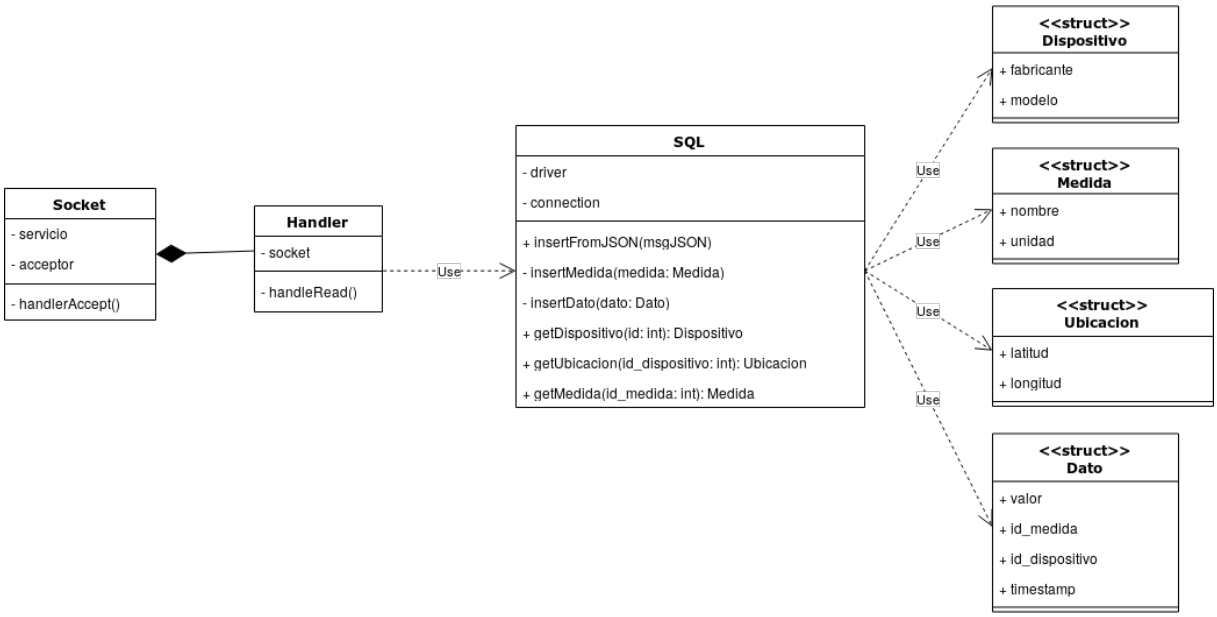


Figura 3.11: Diagrama de clases UML

La clase **Acceptor** se encargará de recibir conexiones TCP y de establecerlas, se ejecutará como un demonio o servicio, esto quiere decir que permanecerá a la escucha de conexiones hasta que se detenga el proceso. Una vez establecida una conexión le consederá el control de esta a una instancia de la clase **Handler** y seguirá a la escucha de nuevas conexiones. Los elementos que podemos destacar de esta clase son:

- **servicio**: Este atributo indica que está clase realizará operaciones con dispositivos de E/S.
- **acceptor**: Este atributo nos permitirá estar a la escucha de conexiones TCP para aceptarlas y asignarle una función a ejecutar en caso afirmativo.
- **handleAccept()**: Esta función se ejecutará cada vez que se acepte una nueva conexión, en ella le dará el control de la conexión a una instancia de la clase Handler.

La clase **Handler** se encargará de leer los datos que se reciban por parte de la conexión que se ha establecido anteriormente. Se creará una instancia de esta clase cada vez que se acepte una nueva conexión. Los elementos que podemos destacar de esta clase son:

- **socket**: Este atributo representa la conexión establecida y nos permite asingarle funciones a ejecutar para la lectura y escritura de datos (lectura $\rightarrow$ recibir, escritura $\rightarrow$ enviar).
- **handleRead()**: Esta función se ejecutará cada vez que recibamos datos en el formato que hemos indicado para la comunicación, en ella pasaremos los datos al singleton DataManager para que realice la inserción de los datos.

La clase **DataManager** se encargará de manejar los datos recibidos en la conexión, en este caso su utilidad será simplemente la de insertar los datos en una base de datos MySQL. Existirá una única instancia de esta clase, representando la conexión que se ha establecido con la base de datos. Los elementos que podemos de esta clase son:

- **driver**: Este atributo representa el controlador que se está usando para establecer la conexión con la base de datos.
- **conexion**: Este atributo representa la conexión establecida con la base de datos, se requiere del host que ejecuta el servicio del base de datos, un usuario, y una contraseña.
- **insertMedida(medida: Medida)**: Esta función inserta un tipo de medida en la base de datos.
- **insertDato(dato: Dato)**: Esta función inserta un nuevo dato medido por un sensor en la base de datos.
- **insertFromJSON(datos)**: Esta función parseará los datos en formato en JSON para así insertarlos.

- **getDispositivo(id: int)**: Esta función nos devolverá el dispositivo/sensor asociado a dicha id del dispositivo.
- **getUbicacion(id_dispositivo: int)**: Esta función nos devolverá la ubicación asociada con el dispositivo que tenga dicha id.
- **getMedida(id_medida: int)**: Esta función nos devolverá la medida asociada con dicha id de la medida.

Los elementos vistos anteriormente, clases, atributos y funciones, se han diseñado de la siguiente manera debido a las librerías Boost.Asio y MySQL Connector/C++ que hemos mencionado en la Sección 3.4 *"Tecnologías a Usar"*.

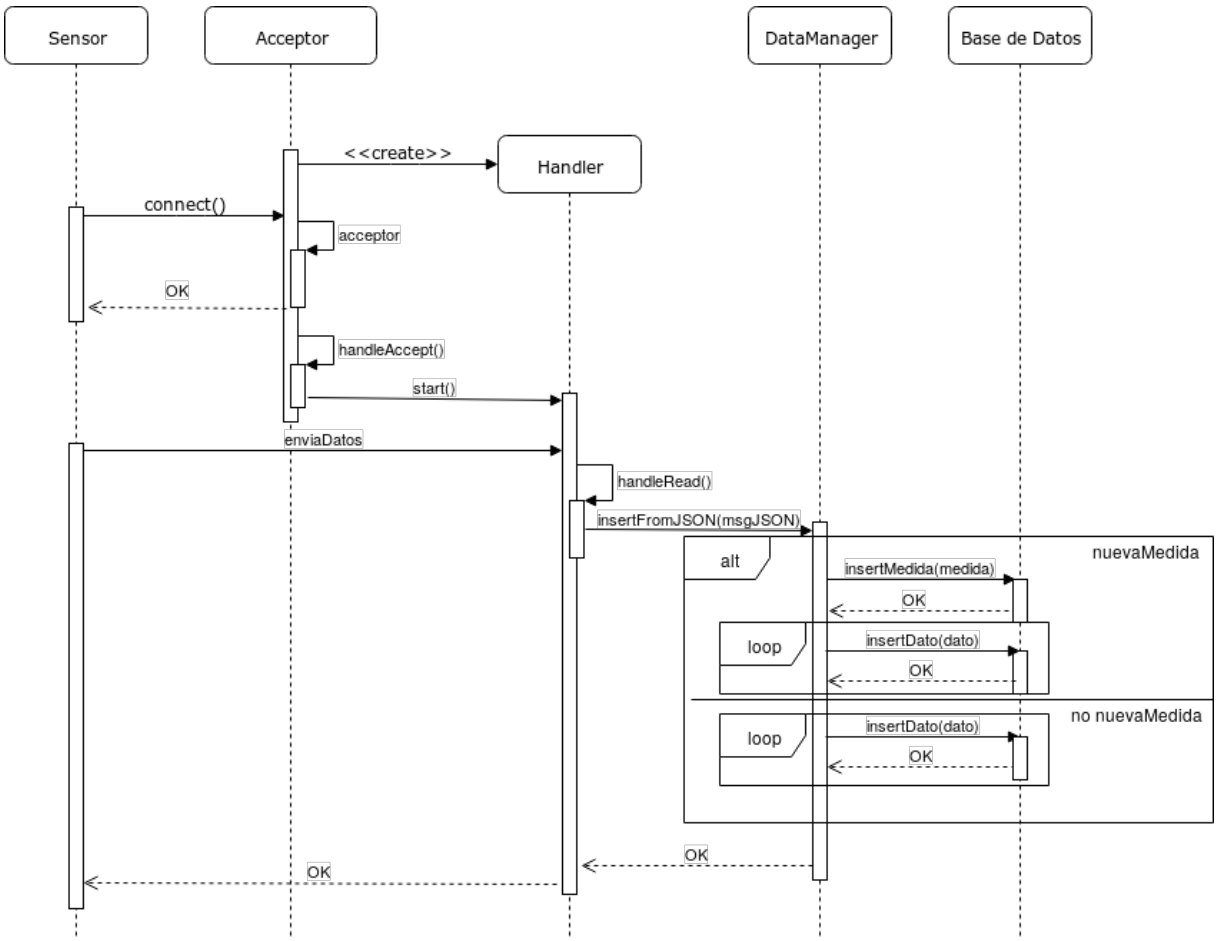


Figura 3.12: Diagrama de secuencia UML

En la Figura 3.12 se presenta el diagrama de secuencia en UML, este diagrama nos permite representar el intercambio de mensajes que se realiza entre los distintos objetos que actúan en nuestro proyecto. De esta forma, podemos representar como se realiza la inserción de los datos recogidos desde el comienzo de la comunicación entre el sensor y el servidor.

Base de Datos

La base de datos es un componente bastante importante de nuestro proyecto, en ella se almacenarán los datos que irán recogiendo los sensores que estén conectados con el servidor. Para diseñar el modelo de la base de datos debemos de tener en cuenta la estructura del objeto JSON que usamos para enviar los datos, de esta forma evitaremos confusiones a la hora de la implementación.

Para diseñar la base de datos haremos uso del **modelo ER** o **ERM** (*Entity Relationship Model*), el modelo ER define una representación conceptual de la base de datos permitiéndonos identificar las entidades que actúan y las relaciones que existen entre ellas. Los elementos que podemos apreciar en el modelo ER son los siguientes:

- **Entidad**: Representa algún aspecto del mundo real que es fácil de identificar del resto.
- **Relaciones**: Representa la relación que mantiene una entidad con otra.
- **Atributos**: Propiedades que contiene una entidad.

En la Figura 3.13 se presenta el **diagrama ER** o **ERD** (*Entity Relationship Diagram*) que hemos obtenido tras identificar las entidades, sus atributos, y las relaciones que se establecen.

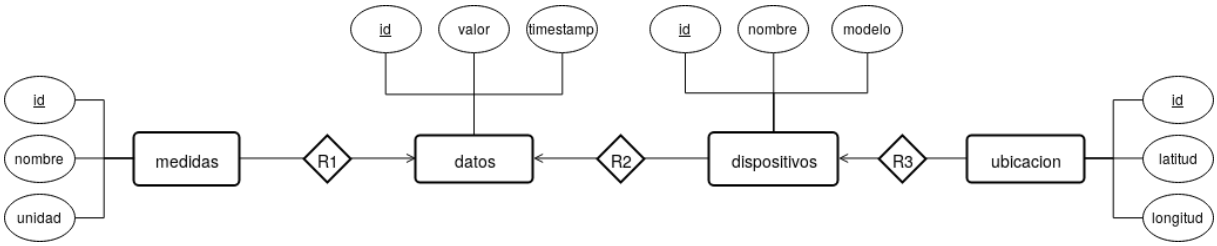


Figura 3.13: ERD de la base de datos.

Las entidades que hemos identificado son las siguientes:

- **medidas**: Entidad que representa un tipo de medida que se recoge en la aplicación. Está compuesta por tres atributos, un identificador, el nombre de la medida y la unidad de la medida.
- **datos**: Entidad que representa un dato que se ha recogido en la aplicación. Está compuesta por tres atributos, un identificador, el valor del dato y la marca de tiempo de recogida del dato.
- **dispositivos**: Entidad que representa un dispositivo que recoge datos en la aplicación. Está compuesta por tres atributos, un identificador, el nombre del dispositivo y el modelo de este.

- **ubicacion**: Entidad que representa una ubicación. Está compuesta por tres atributos, un identificador, la latitud y la longitud.

Además de las entidades mencionadas anteriormente, también identificamos las siguientes relaciones entre entidades:

- **R1**: Relación entre la entidad medidas y la entidad datos, uno a muchos, por cada tipo de medida podemos obtener varios datos.
- **R2**: Relación entre la entidad dispositivos y la entidad datos, uno a muchos, un dispositivo puede recoger varios datos.
- **R3**: Relación entre la entidad dispositivos y la entidad ubicación, uno a muchos, una ubicación puede ser compartida por varios dispositivos.

Finalmente, con el diagrama ER que realizamos podemos obtener las tablas resultantes que se encontrarán en nuestra base de datos (Figura 3.14). Como podemos observar, las relaciones entre las entidades en la base de datos se realizan mediante las claves externas o foreign keys, estas claves representan un atributo (generalmente el identificador) de una tabla externa.

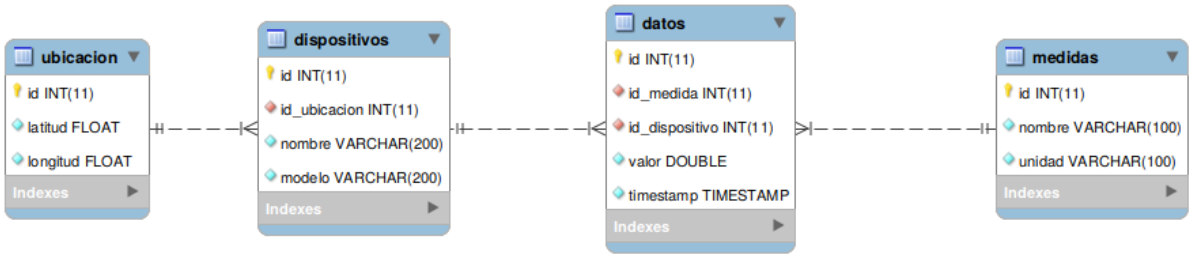


Figura 3.14: Tablas de la base de datos.

Página Web

La página web nos ofrecerá una representación en forma de gráficas sobre los datos que se están recogiendo también nos permitirá añadir nuevos dispositivos a nuestra red de recogida de datos si así lo deseamos.

Para el diseño de este componente software haremos uso de los storyboard. Los storyboard representan una narración gráfica sobre el uso del software que vamos a desarrollar, nos permite a la vez diseñar una interfaz para las ventanas que se van a mostrar y describir su ejecución.

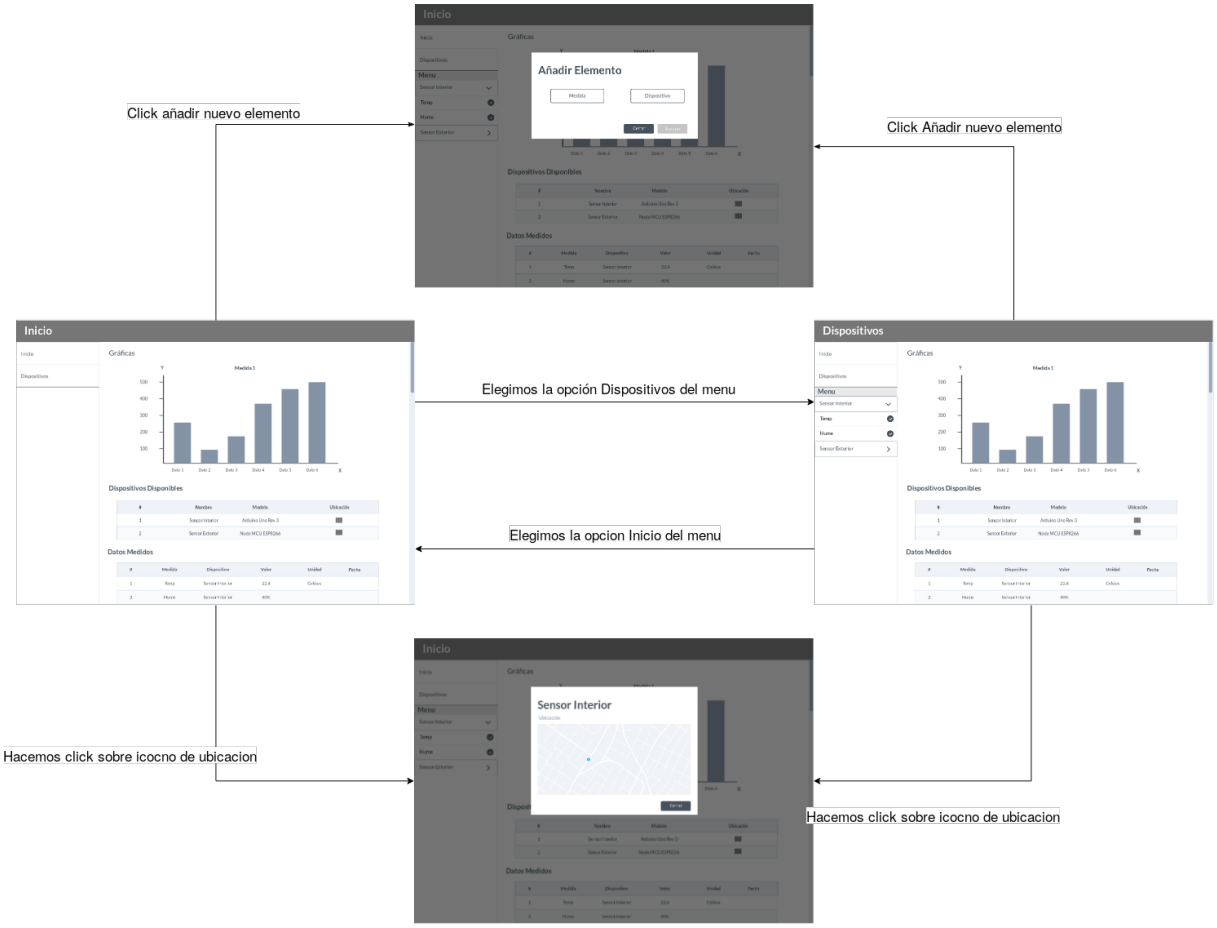


Figura 3.15: Storyboard de la página web.

Sobre el storyboard de nuestra página web (Figura 3.15) podemos destacar dos ventanas o páginas que van a ser las principales:

- **Inicio**
- **Dispositivos**

La página de **Inicio** (Figura 3.16) nos presentará las gráficas con los datos obtenidos de todos los dispositivos sobre una medida, es decir, si nuestros dispositivos están

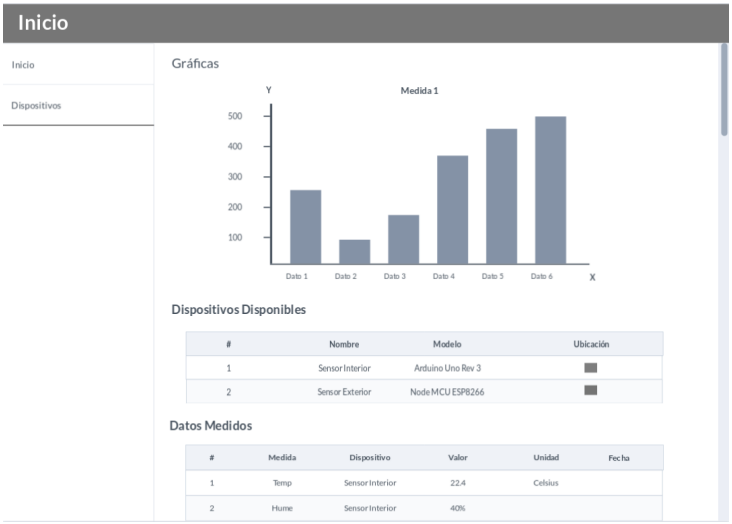


Figura 3.16: Inicio.

midiendo temperatura y humedad, entonces tendríamos dos gráficas con los datos correspondientes. También nos mostrará dos tablas, una con los datos correspondientes sobre los dispositivos que aparecen en las gráficas, y otra sobre todos los datos que se están visualizando en las gráficas.

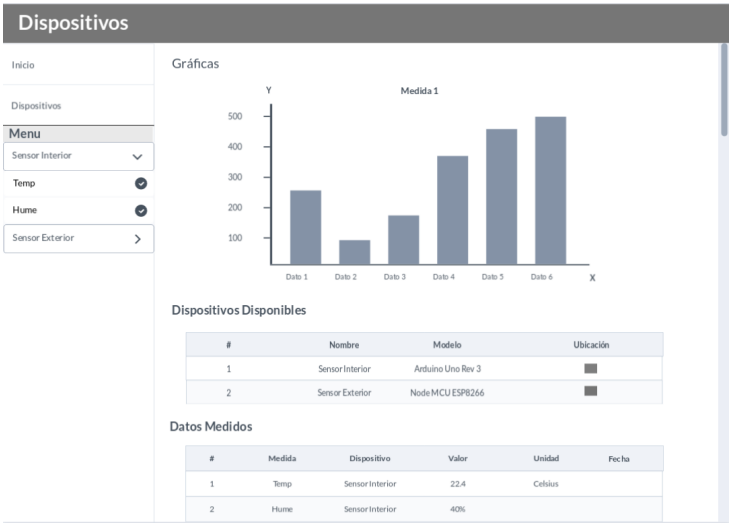


Figura 3.17: Dispositivos.

La página de **Dispositivos** (Figura 3.17) nos presentará las gráficas con los datos obtenidos de todos los dispositivos seleccionados en el seleccionador que está disponible a la izquierda. De esta forma si así se desea, se pueden visualizar los datos únicamente del dispositivo que se seleccione y aparecerán los datos de las medidas que mide dicho dispositivo. Además, al igual que en la página de Inicio, se mostrará dos tablas, una con los datos correspondientes sobre los dispositivos que aparecen en las gráficas, y otra sobre todos los datos que se están visualizando en las gráficas.

Como hemos mencionado antes, ambas páginas contienen las tablas que corres-

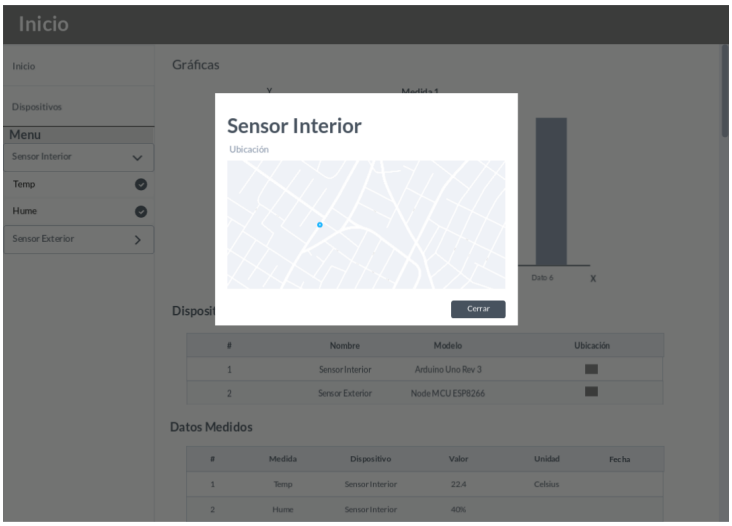


Figura 3.18: Ubicacion.

ponden con la información de las gráficas, la mayoría de esta información podemos representarla en forma de texto o de forma numérica (nombre, fecha, medida, valor medido, etc), pero para la ubicación no nos satisface con ver sus coordenadas, por lo tanto si seleccionamos el icono de ubicación del dispositivo, se nos mostrará una ventana modal con la ubicación del dispositivo en un mapa como se presenta en la Figura 3.18.

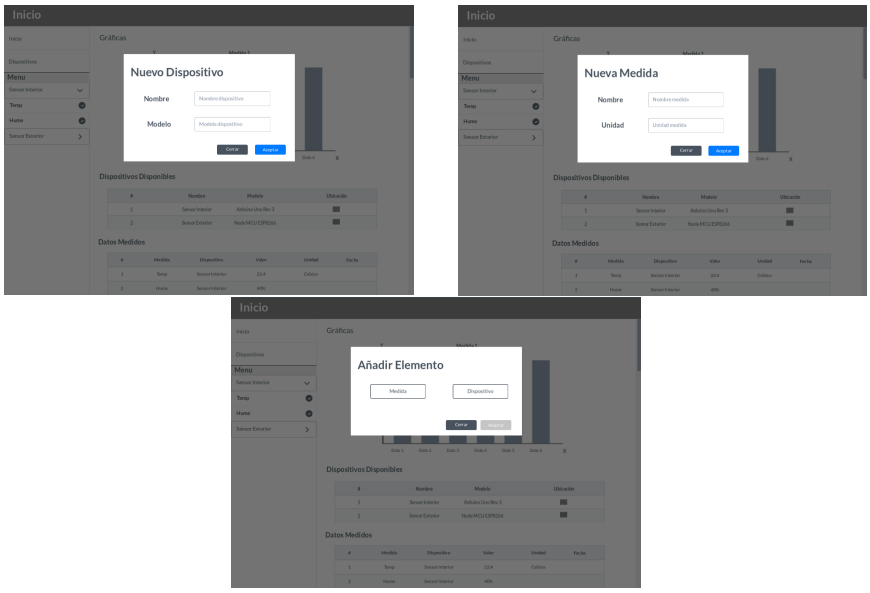


Figura 3.19: Añadir nuevo elemento

También tendremos la opción de añadir nuevas medidas y dispositivos a la base de datos. Para ello, tanto en la página de Inicio como en la página de Dispositivos tendremos acceso a dicha opción, el primer paso es seleccionar el tipo de elemento que se desea añadir e introduciremos los datos necesarios para dicho elemento (Figura 3.19).

3.6. Implementación

En esta sección del documento explicaremos el proceso que se ha seguido para la implementación del software que contiene nuestro proyecto. Estructuraremos esta sección en dos subsecciones, una subsección por dispositivo en el que se ejecuta el software (software sensor y software servidor).

3.6.1. Software Sensor

El software del sensor puede implementarse de diferentes formas dado que las placas de desarrollo usadas para este propósito pueden variar en el lenguaje de programación que usa, o por las tecnologías que integra. Para la solución que implementaremos se usará el software de desarrollo usado para placas Arduino.

Arduino IDE

En la placa Arduino UNO rev.3 implementaremos el software en C/C++, para ello haremos uso del entorno de desarrollo que nos ofrece, **Arduino IDE**. Este entorno de desarrollo nos proporciona todas las herramientas necesarias para desarrollar el código que será implementado. A continuación detallaremos el proceso que se ha seguido para la implementación.

Aunque Arduino IDE sea compatible con Windows, Mac y Linux, se debe de realizar alguna configuración dependiendo del sistema, en nuestro caso realizaremos el desarrollo en un equipo con Linux (**??**).

Figura 3.20: Sketch vacío.

Los proyectos que realizamos en el IDE se denominan Sketch y su estructura, como se observa en la Figura 3.20, se compone de dos funciones: *setup()* y *loop()*.

- **setup()**: Esta función se ejecuta una sola vez al encenderse la placa. Generalmente, en esta función se realiza un proceso de inicialización de variables donde sea necesario.
- **loop()**: Esta función se ejecuta constantemente, como un while sin condición de finalización. Este es básicamente el programa.

Diagrama del Circuito

El diagrama del circuito que se presenta en la Figura 3.21 es el que implementaremos para medir la temperatura y la humedad a través del sensor DHT22. No es una solución específica y dependerá del sensor y elementos adicionales que se puedan conectar a la placa, pero para nuestro objetivo (medir temperatura y humedad), es una solución válida.

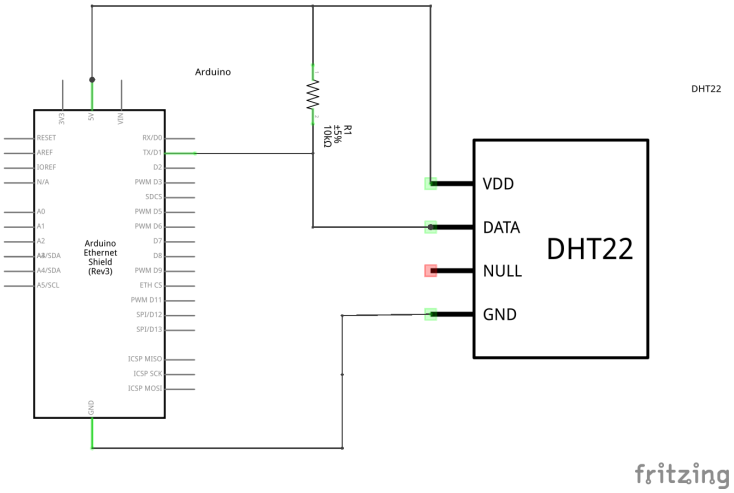


Figura 3.21: Diagrama del circuito.

Conexión con Red

La placa Arduino UNO Rev.3 por defecto no tiene ningún medio para conectarse a una red, por lo tanto debemos de añadir una placa adicional para proporcionarle dicha capacidad. Para ello se ha decidido usar un Ethernet Shield.

El Ethernet Shield nos ofrece un puerto Ethernet para conectar nuestro Arduino a nuestra red de área local, además su programación es relativamente sencilla gracias a las librerías “Ethernet” y “EthernetClient” que nos ofrece Arduino.

Librerías Adicionales

Para poder usar el sensor DHT22 que especificamos en la estimación de costes, haremos uso de dos librerías, y para la serialización de datos en formato JSON una librería adicional.

- **DHT sensor library**: Librería del sensor DHT22, depende de “Adafruit Unified Sensor”.
- **Adafruit Unified Sensor**: Librería abstracta que actua como estándar para interactuar con los sensores.
- **ArduinoJSON**: Librería que nos permite serializar y deserializar datos en formato JSON.

Más información sobre como instalar librerías adiconales en Anexo B.

Transmisión de Datos

El mensaje que vamos a enviar desde el sensor al servidor debe de tener una estructura definida para que el servidor pueda decodificarlo, para ello, como se ha mencionado anteriormente usaremos JSON, que nos permite especificar un conjunto de claves y valores con una estrucutura predefinida.

También debemos de conocer el tamaño en memoria que ocupará dicha estructura en nuestra placa, esto es necesario por las restricciones de memoria de Arduino. Gracias a ArduinoJSON podemos calcular el tamaño del documento JSON para nuestra implementación (Figura 3.22), y se obtiene a partir de la estructura que se ha mencionado en “Comunicacón Sensor y Servidor” (Subsección 3.5.1).

```
const int capacity = JSON_ARRAY_SIZE(3) + JSON_OBJECT_SIZE(2) + 4*JSON_OBJECT_SIZE(4);
```

Figura 3.22: Capacidad de nuestro documento JSON.

La capacidad es variable dependiendo del número distinto de datos que se vayan a transmitir, en este caso transmitiremos tres tipos de datos diferentes: temperatura en celsius, temperatura en farenheit y humedad. Dicho esto, describiremos el significado de cada uno de los elementos que calcula la capacidad de nuestro documento.

- **JSON_ARRAY_SIZE**: Representa arrays en el documento JSON, en nuestra estructura un elemento del array corresponde con un tipo de dato medido.
- **JSON_OBJECT_SIZE**: Representa un objeto en JSON, cualquier conjunto clave - valor, todo documento JSON tiene un objeto como mínimo.

Funciones

De acuerdo con lo que se ha mencionado en la Subsección 3.3.2, el software implementado realizará lo siguiente:

1. Inicializar la placa Ethernet para Arduino.

2. Conectar con un servidor NTP para obtener la fecha y hora actual. Necesario dado que nuestra placa Arduino no incluye un RTC (*Real Time Clock*).
3. Conectar con el servidor mediante un socket TCP.
4. Medir los datos del entorno, en este caso, humedad y temperatura.
5. Enviar los datos medidos en el formato especificado.
6. Cerrar conexión y esperar 30 minutos hasta volver a medir datos.

setup	**loop**
Inicializar Ethernet	Conectar con socket
Conectar con servidor NTP	Medir datos
	Conectar con servidor NTP
	Cerrar conexión y esperar

Cuadro 3.5: Estructura de los pasos.

Estos pasos debemos de ejecutarlos en las funciones que tenemos disponibles en el Sketch (Tabla 3.5). Se ha decidido realizar la conexión con el servidor NTP en la función setup para obtener la fecha y hora inicial, y mediante la función *millis()* ir añadiendo el tiempo que va pasando en cada ejecución del loop. Pero nos encontramos con una limitación, la función se sale de su rango después de 50 días, por lo que necesitaremos solicitar la fecha y hora al servidor NTP transcurridos los 50 días.

Código Fuente

Toda la documentación referente al código fuente del software se puede encontrar en el Anexo A.

3.6.2. Base de Datos

Como hemos mencionado anteriormente, el sistema que usaremos para manejar la base de datos es MySQL, para ello, deberemos de realizar la configuración necesaria, Anexo B.

Crear Base de Datos

Con el software instalado, lo primero que realizaremos será crear un usuario en nuestro servidor MySQL para nuestro proyecto, y la base de datos con sus respectivas tablas que almacenará los datos medidos. La creación de dichos elementos se realiza

desde la consola de MySQL (también se puede realizar desde una interfaz de usuario usando PHPMyAdmin, MySQL Workbench, etc.).

El usuario que crearemos se llamará “tfg_aplicacion” con contraseña “aplicacion” y sólo podrá acceder a la base de datos que vamos a crear. La base de datos se llamará “tfg_recogida_datos”.

```
mysql> CREATE USER 'tfg_aplicacion'@'%' IDENTIFIED BY 'password';
mysql> CREATE DATABASE tfg_recogida_datos;
mysql> GRANT ALL PRIVILEGES on tfg_recogida_datos TO
    'tfg_aplicacion'@'%';
```

Código 1: Crear usuario y base de datos.

En el Fragmento de Código 1 se muestra la creación desde la línea de comandos de MySQL del usuario que usará la aplicación para acceder a la base de datos y también se crea la base de datos. A continuación se mostrará la creación de las tablas que incorpora la base de datos.

```
mysql> CREATE TABLE `tfg_recogida_datos`.`dispositivos` ( `id` INT NOT
    NULL AUTO_INCREMENT , `nombre` VARCHAR(200) NOT NULL , `modelo`
    VARCHAR(200) NOT NULL , PRIMARY KEY (`id`));
mysql> CREATE TABLE `tfg_recogida_datos`.`medidas` ( `id` INT NOT NULL
    AUTO_INCREMENT , `nombre` VARCHAR(100) NOT NULL , `unidad`
    VARCHAR(100) NOT NULL , PRIMARY KEY (`id`));
mysql> CREATE TABLE `tfg_recogida_datos`.`datos` ( `id` INT NOT NULL
    AUTO_INCREMENT , `id_medida` INT NOT NULL , `id_dispositivo` INT
    NOT NULL , `valor` DOUBLE NOT NULL , `timestamp` TIMESTAMP NOT
    NULL , PRIMARY KEY (`id`));
mysql> ALTER TABLE `tfg_recogida_datos`.`datos` ADD CONSTRAINT
    `fk_medida` FOREIGN KEY (`id_medida`) REFERENCES `medidas`(`id`)
    ON DELETE NO ACTION ON UPDATE NO ACTION;
mysql> ALTER TABLE `tfg_recogida_datos`.`datos` ADD CONSTRAINT
    `fk_dispositivo` FOREIGN KEY (`id_dispositivo`) REFERENCES
    `dispositivos`(`id`) ON DELETE NO ACTION ON UPDATE NO ACTION;
mysql> CREATE TABLE `tfg_recogida_datos`.`ubicacion` ( `id` INT NOT
    NULL AUTO_INCREMENT , `id_dispositivo` INT NOT NULL , `latitud`
    FLOAT NOT NULL , `longitud` FLOAT NOT NULL , PRIMARY KEY (`id`));
mysql> ALTER TABLE `tfg_recogida_datos`.`ubicacion` ADD CONSTRAINT
    `fk_ub` FOREIGN KEY (`id_dispositivo`) REFERENCES
    `dispositivos`(`id`) ON DELETE NO ACTION ON UPDATE NO ACTION;
```

Código 2: Creación de tablas.

En el Fragmento de Código 2 se muestra la creación de las tablas, los atributos creados son los mismos que se han mostrado cuando diseñamos la base de datos

(Subsección 3.5.3). Además de crear las tablas, también hay que añadir “Constraints” para crear las relaciones que existen entre la tablas.

3.6.3. Servidor Principal

El servidor principal es el componente software que ejecutará el servidor TCP para establecer la conexión usando sockets entre el sensor y el servidor, en este caso la placa Arduino y la Raspberry Pi respectivamente.

Entorno de Desarrollo

El entorno de desarrollo en el cuál desarrollaremos el software es el que se encuentra presente en la Tabla 3.6.

Sistema Operativo	Ubuntu 18.04 LTS
Editor de Texto	Visual Studio Code
Extensiones	C++, C++ Intellisense
Compilador	g++

Cuadro 3.6: Entorno de desarrollo C++.

Desarrollaremos el software en un sistema diferente al que será usado, en este caso estamos desarrollando el software en un sistema con arquitectura de 64 bits y su uso final será en un sistema con arquitectura ARM. En este caso tenemos todas las librerías disponibles en ambas arquitecturas, pero es algo a tener en cuenta.

Como editor de texto se usará Visual Studio Code de Microsoft, es un editor de texto de código abierto que permite el uso de extensiones para desarrollar en el lenguaje de programación que se decida. En nuestro caso usaremos las extensiones de C++ y C++ Intellisense.

Librerías

La implementación del software se realizará en C/C++ y se hará uso de las librerías que se mencionaron anteriormente en “Tecnologías a Usar” (Sección 3.4). A continuación mencionaremos y describiremos brevemente que nos proporcionan cada una de este librerías a nuestro software.

- **JsonCpp**: Nos ofrece clases y funciones mediante las cuales podemos interactuar con datos que se encuentran en formato JSON, permite serializar y deserializar datos.
- **Boost.Asio**: Nos ofrece clases y funciones mediante las cuales podemos trabajar con recursos de red y elementos de E/S.

- **Boost.Program Options**: Nos ofrece clases y funciones mediante las cuales podemos obtener opciones definidas por el usuario para ejecutar el software.
- **MySQL Connector/C++**: Nos ofrece clases y funciones mediante las cuales podemos interactuar con una base de datos MySQL.

Conexión con Sensor

La primera función que se implementará es la comunicación entre el sensor y el servidor. Para ello, mediante las clases y funciones de la librería Boost.Asio hemos implementado las dos clases "Acceptor" y "Handler" que mencionamos en el diseño del servidor (Subsección 3.5.3).

A continuación mostraremos el constructor de ambas clases en pseudocódigo y describiremos brevemente qué realiza y cuales serán los pasos siguientes.

```
1  Acceptor(io_service& io_service) {
2    this->acceptor = tcp::acceptor(io_service, endpoint(v4(),
     ↪  PUERTO_TCP));
3    startAccept();
4  }
5
6  void startAccept() {
7      connection = new Handler(acceptor.get_io_service());
8      acceptor.async_accept(connection->getSocket(),
       ↪  bind(&Acceptor::handleAccept, this, connection,
       ↪  boost::asio::placeholders::error));
9  }
10
11 void handleAccept(Handler* connection, error_code& err) {
12     if (!err) {
13         connection->start();
14     }
15     startAccept();
16 }
```

Código 3: Clase Acceptor.

La clase **Acceptor** (Fragmento de Código 3) aceptará en su único constructor por referencia una instancia de io_service (de Boost.Asio) para proporcionar a nuestra clase funcionalidad de E/S. Creamos un objeto de la clase acceptor (de Boost.Asio) con la misma instancia de io_service y especificando qué protocolo IP se usará (IPv4 en nuestro caso) y qué puerto se usará.

Una vez creado el objeto acceptor, se lanza la función que quedará a la escucha de conexiones. Como podemos ver en la función *startAccept*, se crea una nueva conexión

asociándolo a un nuevo Handler y se realiza una escucha de forma asíncrona, de esta forma el programa puede seguir realizando otras funciones hasta que no se acepte una conexión, en este proceso también asociamos la función handleAccept como proceso a ejecutarse al aceptar una conexión.

Cuando se realice la aceptación de una conexión, se inician las funciones que realizará el Handler que se asignó a dicha conexión y seguirá a la escucha de nuevas conexiones.

```
1  Handler(io_service& io_service){
2    this->socket = socket(io_service);
3  }
4
5  void start() {
6
7    socket.async_read_until(buffer, '\r', bind(&Handler::handleRead,
     ↪  shared_from_this(), boost::asio::placeholders::error,
     ↪  boost::asio::placeholders::bytes_transferred));
8
9  }
10
11 void handleRead(error_code &err, size_t bytes_transferred) { ... };
```

Código 4: Clase Handler.

La clase **Handler** (Fragmento de Código 4) aceptará en su único constructor por referencia una instancia de io_service (de Boost.Asio) para proporcionar a nuestra clase funcionalidad de E/S. Creamos un objeto de la clase socket asociándolo al io_service en ejecución para poder manejar la conexión que se le ha asignado al Handler.

Cuando la clase Acceptor acepte una conexión iniciará la función *start* del Handler, en esta función iniciaremos la escucha del socket para la recepción de datos y cuando se reciban datos, se inicia la función *handleRead* que preparará los datos para su posterior inserción en la base de datos.

Decodificación de los datos

En esta parte explicaremos el proceso o pasos que sigue la función *handlRead* de la clase Handler. En esta función se hará uso de la librería JsonCpp, mediante la cual podremos serializar el mensaje recibido como un objeto JSON y manejarlo a través de funciones y atributos.

Como se puede ver en el Fragmento de Código 5, cuando se reciba un mensaje con los datos en formato JSON y no se produzca ningún error, procederemos a crear un objeto JSON a partir del mensaje recibido y comprobar si existe en la base de datos el dispositivo asociado. En caso afirmativo envía un mensaje con el código 200

```
void handleRead(error_code& err, size_t bytes_transferred) {
  if (!err) {
    datos = str(buffer);
    Json::Reader reader;
    Json::Value obj;
    reader.parse(datos, obj);
    if (dispositivo en base de datos) {
      socket.async_write(buffer(ACK_MSG, sizeof(ACK_MSG)));
      try {
          DataManager::getInstance()->insertFromJSON(obj);
      } catch (sql::SQLException &e) {
          Se ha producido una excepcion
      }
    }
    else {
      socket.async_write(buffer(ERROR_MSG_UNAUTHORIZED,
       ↪  sizeof(ERROR_MSG_UNAUTHORIZED)));
    }
  }
  else {
    Se ha producido un error
  }
  socket.close();
}
```

Código 5: Función handleRead

al cliente para notificar que se ha realizado correctamente el envío, y se inserta el objeto JSON en la base de datos. En caso contrario envía un mensaje con el código 201 para indicar que no está autorizado.

Una vez que se hayan realizado todas las operaciones, se cierra el socket puesto que se ha llegado al final de la comunicación.

Conexión con Base de Datos

Para manejar los datos que recibimos por parte del sensor haremos uso de la clase "DataManager", esta clase hará uso de las clases y funciones de la librería MySQL Connector/C++ para conseguirlo.

La clase DataManager se ha decidido diseñarla como un singleton, puesto que solo tendremos una instancia de la clase en ejecución para realizar la inserción de datos. Se ha decidido realizarlo de esta manera para no crear una nueva conexión con la base de datos para cada dato recibido.

En el Fragmento de Código 6 se presenta el constructor de la clase DataManager, desde la configuración que insertará el usuario se obtendrá la conexión con la base

```
DataManager() {
    string host = config.host;
    string usuario = config.usuario;
    string password = config.password;

    this->driver = get_driver_instance();
    this->connection = driver->connect(host, usuario, password);
    this->connection->setSchema(database);

}
```

Código 6: Clase DataManager.

de datos especificando la dirección a la que se tiene que conectar, el usuario con el que accede a la base de datos y la contraseña del usuario.

El resto de funciones de esta clase realiza operaciones de inserción y obtención de datos, que se traduce a instrucciones INSERT y SELECT en SQL que se encuentran documentadas en el Anexo A.

3.6.4. Página Web

Una vez implementados los componentes software que manejan la inserción de los datos necesitaremos una forma de visualizarlos. Inicialmente, para comprobar la correcta inserción de los datos usamos la instrucción SELECT desde la terminal de MySQL sobre la tabla “datos” y al comprobar que funciona, podemos seguir con la implementación de la página web.

Entorno de Desarrollo

El entorno de desarrollo en el cuál desarrollaremos el software es el que se encuentra presente en la Tabla 3.7.

Sistema Operativo	Ubuntu 18.04 LTS
Editor de Texto	Visual Studio Code
Extensiones	PHP Intelephense
Compilador	php

Cuadro 3.7: Entorno de desarrollo Página web.

Como editor de texto se usará Visual Studio Code de Microsoft, es un editor de texto de código abierto que permite el uso de extensiones para desarrollar en el lenguaje de programación que se decida. En nuestro caso usaremos la extensión PHP Intelep-

hense, que nos ofrece varias utilidades para programar en PHP junto al compilador del mismo lenguaje de programación.

Configuración

El primer paso a realizar es instalar y configurar las herramientas que usaremos para desarrollar este elemento software, como ya se ha mencionado en "Tecnologías a Usar" (Sección 3.4) haremos uso de:

- **Apache**: Servidor web escrito en C++.
- **CodeIgniter**: Framework basado en el modelo MVC (*Model-View-Controller*) para desarrollar proyectos en PHP.

La configuración de un servidor Apache y la instalación de CodeIgniter en Raspberry Pi se encuentra documentado en el Anexo B.

Creación del Controlador

El controlador que implementaremos manejará las vistas que se presentarán en nuestra página, crearemos un único controlador en la carpeta "controllers" de nuestra aplicación, que llamaremos "Home.php".

Sólo crearemos un controlador dado que en nuestro proyecto solo tenemos dos vistas: una vista para el inicio y otra vista para la selección de dispositivos. Para manejar ambas de estas vistas nuestro controlador contiene dos funciones, una por cada vista.

```
1 class Home extends CI_Controller {
2   public function __construct() {
3     parent::__construct();
4   }
5   public function home() { ... }
6   public function dispositivos() { ... }
7 }
```

Código 7: Clase PHP del controlador.

Mediante las funciones del controlador **Home**, podremos acceder desde nuestro navegador a los resultados que devuelven, en este caso ambas funciones devolverán una vista, la vista de Inicio y Dispositivos respectivamente. A lo largo de la implementación del proyecto se irán añadiendo funciones al controlador para ampliar la funcionalidad de las vistas, como por ejemplo, realizar llamadas asíncronas mediante AJAX.

El código fuente se puede encontrar en la documentación del mismo componente software Anexo A.

Creación de Modelos

Los modelos representan los datos que pueden encontrarse en la base de datos, se deberá de crear un modelo por cada tabla que tenemos. Por lo tanto, crearemos cuatro modelos de datos en la carpeta “models” de nuestra aplicación.

- **Datos_model.php**: Modelo de datos de la tabla “datos”.
- **Dispositivos_model.php**: Modelo de datos de la tabla “dispositivos”.
- **Medidas_model.php**: Modelo de datos de la tabla “medidas”.
- **Ubicacion_model.php**: Modelo de datos de la tabla “ubicacion”.

En los modelos, se implementan las funciones necesarias para realizar operaciones con la base de datos, por lo tanto para cada modelo que hemos definido debemos de implementar las funciones que sean necesarias. En el Fragmento de Código 8 se muestra como ejemplo el modelo de de la tabla “medidas”.

```
1  class Medidas_model extends CI_Model {
2    var $id;
3    var $nombre;
4    var $unidad;
5    public function __construct() {
6      parent::__construct()
7      $this->load->database();
8    }
9
10   public function getAllMedidas() {
11     $query = $this->db->get('medidas');
12     return $query->result_array();
13   }
14
15   public function getMedida($id) {
16     $query = $this->db->get_where('medidas', array('id' => $id));
17     return $query->result_array();
18   }
19 }
```

Código 8: Ejemplo modelo de medidas.

Como se muestra en el ejemplo (Fragmento de Código 8), tenemos dos funciones definidas, mediante las cuáles podremos obtener todas las medidas, o una medida en concreto gracias a su id.

Una vez implementado el modelo, debemos de cargarlo al construir el controlador para poder usarlo (Fragmento de Código 9), una vez hecho esto podremos usar las funciones del modelo para acceder a la base de datos desde cualquier función del controlador (Fragmento de Código 10).

```
1 public function __construct() {
2   parent::__construct();
3   $this->load->model('datos_model');
4   $this->load->model('dispositivos_model');
5   $this->load->model('medidas_model');
6   $this->load->model('ubicacion_model');
7 }
```

Código 9: Carga de modelos en constructor del controlador.

```
1 $data['medidas'] = $this->medidas_model->getAllMedidas();
```

Código 10: Ejemplo llamada función de un modelo

Implementación de Plantillas

Las vistas que incluye nuestra página web serán solo dos y se realizan en base al diseño previamente definido (Figura 3.15). Nuestras vistas incluirán código HTML, PHP y JavaScript, por ello, usaremos la librería **Bootstrap**, que nos permite escribir el código de nuestras vistas sin tener que preocuparnos tanto por el diseño.

Las vistas de nuestra aplicación se encuentran en la carpeta “views”, en esa misma carpeta, implementaremos dos vistas: home.php y dispositivos.php. Además, se ha decidido incluir todos aquellos elementos que se repitan de las vistas en plantillas, por lo tanto, tendremos dos plantillas: templates/header.php y templates/footer.php.

- **header**: Esta plantilla incluirá la cabecera del la página HTML y también el menú lateral superior para navegar entre vistas.
- **footer**: Esta plantilla incluirá las funciones en JavaScript a ejecutar por la vista.

La cabecera incluirá las siguientes hojas de estilo (archivos CSS) que se usarán en ambas de las vistas. Estos archivos, como su nombre indica, dan el estilo a la página mediante el uso de clases, atributos, por elemento HTML, etc.

- **bootstrap.min.css**: Archivo CSS que incluye el estilo de los elementos que podemos usar con la librería bootstrap.

- **Chart.css**: Archivo CSS que incluye el estilo de los elementos que podemos usar con la librería Chart.js.
- **dashboard.css**: Archivo CSS que incluye el estilo de algunos elementos especiales que requiere nuestra aplicación, se aplica sobre la hoja de estilo de bootstrap.
- **font-awesome.css**: Archivo CSS que nos permite usar una gran variedad de iconos, se requiere de la carpeta "webfonts" que incluye la librería.
- **tempus-dominus.min.css**: Archivo CSS que incluye el estilo de los elementos que nos permiten introducir fechas y horas.

El pie de página incluirá los siguientes archivos de scripts de JavaScript que se usarán en ambas de las vistas. Estos archivos, nos ofrecen funciones mediante los cuales podremos interactuar con los elementos que se encuentran en la página.

- **jquery.min.js**: Archivo JavaScript de la librería jQuery, simplifica el uso de funciones en el lado del cliente.
- **popper.min.js**: Archivo JavaScript de la librería Popper, se encuentra en nuestro proyecto debido a que es una dependencia de Bootstrap.
- **bootstrap.min.js**: Archivo JavaScript de la librería Bootstrap, contiene funciones que son necesarias para los elementos que nos ofrece la librería.
- **moment-with-locales.js**: Archivo JavaScript de la librería Moment.js, nos ofrece funciones para trabajar con fechas, horas, marcas de tiempo, etc.
- **Chart.js**: Archivo JavaScript de la librería Chart.js, es el corazón de la librería y nos ofrece funciones para manipular y crear gráficas.
- **tempus-dominus.min.js**: Archivo JavaScript que nos ofrece funciones para trabajar con las fechas que introducirá el usuario para manipular la gráficas.
- **Google Maps**: API de Google para incluir mapas en nuestra página web, se debe de especificar una clave de desarrollador para poder usarla.

Implementación de Vistas

Las vistas necesitarán obtener los datos para poder presentarlos, para ello se ha decidido realizar llamadas asíncronas mediante JavaScript. De esta forma no es necesario recargar la página siempre que se produce un cambio y la transmisión de datos se realiza de forma asíncrona permitiendo cambiar el contenido de la página de forma dinámica.

Para poder realizar esto, debemos de implementar varias funciones en nuesto controlador que nos permitirán obtener los datos que necesitemos. Las funciones que implementaremos serán las siguientes:

- **home()**: Esta función del controlador nos devolverá la vista de Inicio.
- **dispositivos()**: Esta función del controlador nos devolverá la vista de Dispositivos.
- **paginacionDatos()**: Esta función del controlador devuelve en formato JSON los datos que contiene una página de la tabla.
- **removeTabData(id)**: Esta función del controlador elimina de la caché una medida seleccionada y devuelve los datos en formato JSON.
- **addTabData(id)**: Esta función del controlador añade a la caché una medida seleccionada y devueve los datos en formato JSON.
- **graphData(id)**: Esta función del controlador devuelve en formato JSON los datos que contendrá la gráfica.
- **getDispositivosDisp()**: Esta función del controlador devuelve en formato JSON los datos de dispositivos disponibles dependiendo de la medida.
- **addMedida()**: Esta función del controlador obtiene los datos introducidos por el usuario y añade una medida a la base de datos.
- **addDispositivos()**: Esta función del controlador obtiene los datos introducidos por el usuario y añade un dispositivo a la base de datos.
- **addMedidaStep()**: Esta función del controlador es el paso intermedio que se realiza para introducir una medida a la base de datos.
- **addDispositivoStep()**: Esta función del controlador es el paso intermedio que se realiza para introducir un dispositivo a la base de datos.

Con estas funciones podemos obtener los datos de forma asíncrona y ofrece una mejor experiencia para el usuario al usar la interfaz. La comunicación asíncrona se realizará mediante la función `$.ajax()` de la librería jQuery. Como ejemplo se muestra en el Fragmento de Código 11 cómo se ha realizado para obtener los datos para las gráficas.

Como se puede observar, definimos la url con el dato de entrada de la función del controlador, definimos el método de la comunicación, GET o POST, definimos el formato de los datos que recibiremos, y finalmente definimos la función a ejecutar en

```
$.ajax({
  url: "<?= base_url() ?>index.php/home/graphData/" + idMedida,
  method: "GET",
  dataType: "json",
  success: loadChart
});
```

Código 11: Llamada AJAX para datos de gráfica.

caso de éxito, en este caso, la función loadChart de nuestro conjunto de funciones de JavaScript (archivo myjs.js).

Una vez implementadas las funciones JavaScript necesarias para la página web, concluye el proceso de implementación de las vistas y se puede proceder a cambiar el estado del proyecto de “desarrollo” a “producción”.

4 Resultados

4.1. Funcionamiento de la Aplicación

Una vez finalizada la implementación de los componentes software, podemos proceder a ejecutarlos para que comience a recoger datos.

4.1.1. Ejecución código sensor

En este apartado mostraremos los resultados obtenidos de la compilación, la subida del código a la placa Arduino y su ejecución.

El primer paso que se debe de realizar es compilar el código para comprobar la existencia de errores o falta de memoria.

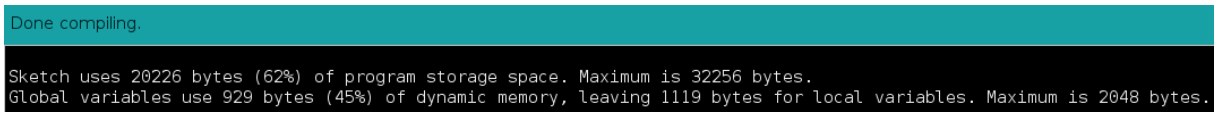


Figura 4.1: Resultado compilación.

Con el código compilado satisfactoriamente, podemos proceder a subirlo a la placa Arduino. Con “subirlo” nos referimos al proceso de flashear el código que se ha desarrollado a la memoria no volátil de la placa Arduino.

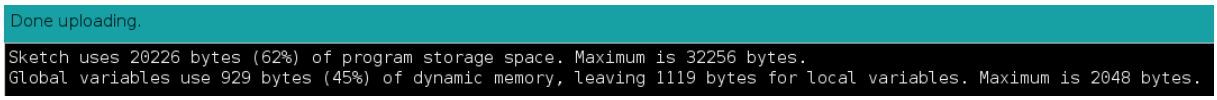


Figura 4.2: Resultado de la subida del código.

Una vez subido el código, procederá a ejecutarse cada vez que la placa Arduino se encienda. En el código tenemos llamadas que escriben al puerto Serie de la placa, permitiéndonos comprobar desde el Arduino IDE los mensajes que van apareciendo.

Figura 4.3: Resultado ejecución código sensor.

4.1.2. Recogida de Datos

Con el sensor conectado a nuestra red de área local, y los servidores en ejecución, podremos ver como a lo largo que transcurre el tiempo, se van insertando datos en nuestra base de datos. Podremos verlo tanto desde los mensajes informativos del servidor principal, como desde la aplicación web.

```
pi@raspberrypi:~/Desktop/cpp_socket_s $ ./server -u tfg -p password -s tfg_recogida_datos
Conexion con BBDD realizada
Nueva conexion recibida: 192.168.1.200
Datos recibidos sin errores
Mensaje ACK enviado
Conexion finalizada
```

Figura 4.4: Ejecución servidor principal recibiendo datos.

Dejando la aplicación ejecutar durante varios días, podemos ver que los datos se insertan sin problemas. A continuación se mostrarán los datos insertados en la base de datos mediante la ejecución de una consulta “SELECT” sobre la tabla de datos.

id	id_medida	id_dispositivo	valor	timestamp
1	2	1	46	2019-06-18 01:36:36
2	1	1	25.8	2019-06-18 02:06:37
3	3	1	78.44	2019-06-18 02:06:37
4	2	1	44.5	2019-06-18 02:06:37
5	1	1	25.8	2019-06-18 02:36:38
6	3	1	78.44	2019-06-18 02:36:38
7	2	1	45.1	2019-06-18 02:36:38
8	1	1	25.8	2019-06-18 03:06:39
9	3	1	78.44	2019-06-18 03:06:39
10	2	1	45	2019-06-18 03:06:39
11	1	1	25.7	2019-06-18 03:36:40
12	3	1	78.25999	2019-06-18 03:36:40
13	2	1	43.7	2019-06-18 03:36:40
14	1	1	25.7	2019-06-18 04:06:41

Figura 4.5: Resultado de consulta SELECT.

También podemos comprobar los datos insertados navegando a la dirección IP de nuestro equipo Raspberry Pi y consultando la gráfica.

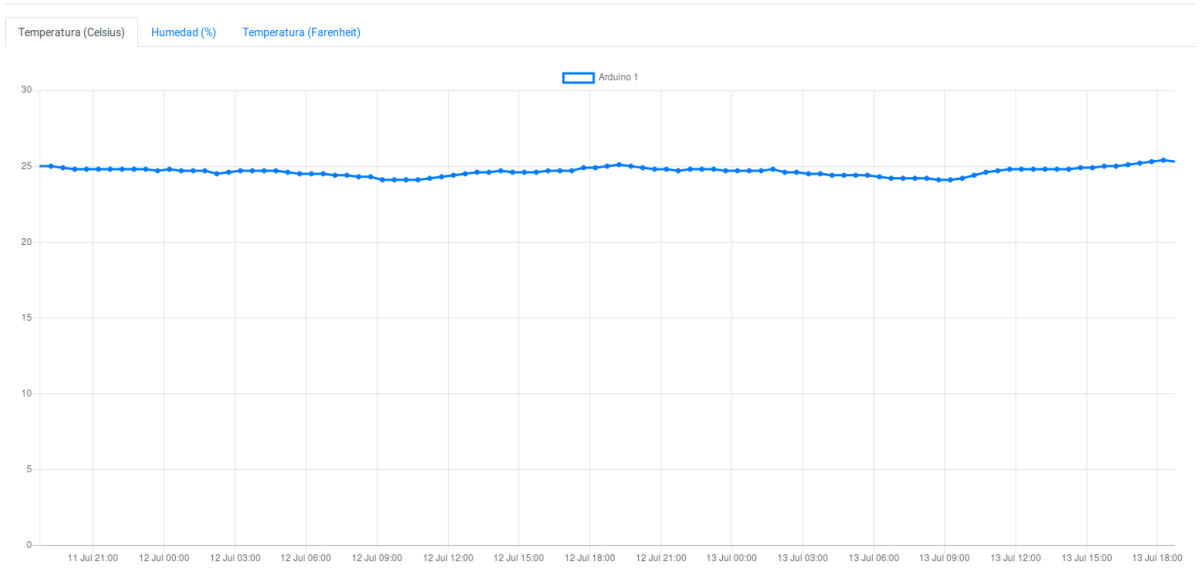


Figura 4.6: Datos insertados representados en la gráfica

4.2. Conclusiones

En esta sección comentaré mi opinión sobre el desarrollo de proyectos IoT y la programación de los diferentes componentes software que he realizado.

El desarrollo de proyectos en IoT se puede realizar de varias formas distintas, debido al número de tecnologías que existen y que pueden trabajar conjuntamente. En nuestro caso, se ha decidido realizar la conectividad entre los dispositivos mediante Ethernet, pero existen tecnologías como Wi-Fi, ZigBee, LTE, 5G, etc. que permiten realizar la conectividad sin cables.

En general, opino que cuando se decide realizar un proyecto en IoT, se debe de tener un conocimiento amplio sobre las tecnologías que tenemos disponibles para poder usar la adecuada, dado que cada proyecto tendrá unos requisitos diferentes.

Los componentes software que se han desarollado a lo largo del proyecto son muy distintos y cada uno sirve para cumplir una función distinta del proyecto, además que se ha usado de forma extensa tres lenguajes de programación: C++, PHP y JavaScript.

Como la recogida de datos se realiza en una red de área local, no se ha requerido de ninguna plataforma IoT y se ha implementado un servidor que comunique con el sensor. La programación de dicho servidor ha sido bastante más complicada de lo que me esperaba y le he tenido que dedicar más tiempo a estudiar el funcionamiento de las librerías usadas antes de poder ponerme a programar.

En cambio, el resto de componentes software (base de datos, servidor web y software sensor) han sido bastante menos complejos de implementar dado que los elementos con los que trabajaba se han usado a lo largo de la carrera.

Finalmente, el desarrollo del proyecto ha sido un largo proceso de investigación de

tecnologías y aprendizaje de nuevas técnicas de programación en C++. El tema del Internet de las Cosas es un tema que previamente me interesaba bastante dado a las posibilidades que tiene. Realizar un proyecto como este, por simple que parezca, me ha ayudado a entender mejor las tecnologías que participan en este tema.

5 Líneas de Futuro

Una vez realizado el proyecto, se sugieren mejoras que podríamos incluir en versiones futuras de la aplicación.

Se ha decidido por realizar la aplicación web desde cero, pero también se podría haber optado por usar Grafana [12], una plataforma de monitorización y análisis de datos.

Actualmente, la conexión se realiza mediante Ethernet, pero podemos añadir dispositivos adicionales usando tecnologías diferentes para extender el rango de los datos medidos.

Encriptar la información que envía el sensor al servidor principal, para mejorar así la seguridad de nuestra aplicación.

Bibliografía

[1] M. Weiser, “The Computer for the 21ST Century”, *Scientific American*, vol. 265, pág. 94, sep. de 1991. DOI: 10.1038/scientificamerican0991-94. dirección: https://web.archive.org/web/20150311220327/http://web.media.mit.edu/~anjchang/ti01/weiser-sciam91-ubicomp.pdf (visitado 16-06-2019).

[2] K. Ashton. (jun. de 2009). That ’Internet of Things’ Thing, dirección: https://www.rfidjournal.com/articles/view?4986 (visitado 16-06-2019).

[3] A. Gerber. (2017). Choosing the best hardware for your next IoT project, dirección: https://developer.ibm.com/articles/iot-lp101-best-hardware-devices-iot-project/ (visitado 23-04-2019).

[4] Medea. (2013). Arduino FAQ – With David Cuartielles, dirección: http://medea.mah.se/2013/04/arduino-faq/ (visitado 23-04-2019).

[5] Raspberry Pi Foundation. (2018). Raspberry Pi 3 Model B+, dirección: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf.

[6] Microchip Techonology Inc., *ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet*, 2018. dirección: https://www.microchip.com/wwwproducts/en/ATMEGA328P.

[7] IEEE Standars Association. (2015). 802.15.4-2015 - IEEE Standard for Low-Rate Wireless Networks, dirección: https://standards.ieee.org/standard/802_15_4-2015.html.

[8] Network Working Group, “Hypertext Transfer Protocol – HTTP/1.1”, *Request for Comments: 2616*, 1999. dirección: https://www.ietf.org/rfc/rfc2616.txt.

[9] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks”, *Commun. ACM*, vol. 13, n.º 6, págs. 377-387, jun. de 1970, ISSN: 0001-0782. DOI: 10.1145/362384.362685. dirección: http://doi.acm.org/10.1145/362384.362685.

[10] DB-Engines. (2019). DB-Engines Ranking, dirección: https://db-engines.com/en/ranking (visitado 18-04-2019).

[11] Indeed. (jul. de 2019). Salarios para empleos de Desarrolador/a de software en España, dirección: https://www.indeed.es/salaries/Desarrollador/a-de-software-Salaries (visitado 17-07-2019).

[12] Grafana Labs. (2019). Grafana, dirección: https://grafana.com (visitado 18-06-2019).

A Documentación Código

En este anexo se muestra toda la documentación referente al código fuente de cada uno de los componentes software que se han desarrollado.

A.1. Software Sensor

En esta sección se detallará toda la documentación del código fuente que se ha desarrollado para el sensor. La aplicación en ejecución envía la información de los datos medidos al servidor con la dirección IP dada.

A.1.1. Cabecera

```
1 #include <SPI.h>
2 #include <Ethernet.h>
3 #include <EthernetUdp.h>
4 #include <ArduinoJson.h>
5
6 #include "DHT.h"
```

Código 12: Cabeceras del software del sensor.

Como se puede apreciar en el Fragmento de Código 12, las cabeceras que se incluyen son las siguientes:

- **SPI.h**: Esta librería nos permite comunicarnos con dispositivos SPI, usando el Arduino como el dispositivo principal.
- **Ethernet.h**: Esta librería está diseñada para funcionar con el Ethernet Shield, nos permite conectarnos a internet.
- **EthernetUdp.h**: Esta librería nos permite enviar y recibir mensajes UDP, serán necesarios para comunicar con el servidor NTP.
- **ArduinoJson.h**: Esta librería nos permite serializar y deserializar datos en formato JSON.
- **DHT.h**: Esta librería nos permite comunicar con el sensor DHT mediante funciones, como una API.

A.1.2. Variables Globales

Las variables globales que se han definido en el software del sensor, son las siguientes:

- **DHTPIN**: Pin en el que está conectado el sensor digital DHT.
- **DHTTYPE**: Tipo de sensor DHT que usamos, en nuestro caso DHT22.
- **DHT dht(DHTPIN, DHTTYYPE)**: Objeto de la librería DHT mediante el cual comunicaremos con el sensor.
- **byte mac[]**: Dirección MAC a usar para la placa Ethernet.
- **IPAddress server(dirIP)**: Dirección IP del servidor que almacenará los datos.
- **EthernetClient client**: Cliente Ethernet para realizar la comunicación con el servidor.
- **int PUERTO**: Puerto mediante el cual el dispositivo se conectará al servidor para almacenar los datos.
- **unsigned long tiempoAnterior**: Tiempo calculado al final de una medida para calcular el timestamp, evitando re-conexiones con el servidor NTP.
- **unsigned long epoch**: Timestamp que se introducirá junto con la medida, se va recalculando a cada iteración.
- **unsigned long epochInicial**: Timestamp recibido inicalmente por parte del servidor NTP, para calcular si han pasado siete días.
- **const int capacity**: Capacidad que ocupará nuestro objeto JSON en memoria.

A.1.3. Función Setup

```
1  void setup() {
2    // Iniciamos la conexion Ethernet
3    Ethernet.begin(mac, ip);
4
5    // Abrimos comunicacion Serial y esperamos a que se inicie
6    Serial.begin(9600);
7
8    // 1 segundo inicializacion Ethernet
9    delay(1000);
10
11   // Conexion con el servidor NTP para obtener fecha
12   IPAddress timeServer(213, 251, 52, 234);
13   Serial.println("Conectando a servidor NTP");
14   sendNTPpacket(timeServer);
15   epochInicial = epoch
16
17   dht.begin();
18   delay(2000);
19   tiempoAnterior = millis();
20 }
```

Código 13: Función setup.

En el Fragmento de Código 12 se presenta el contenido de la función setup que se encuentra en nuestro software. A continuación, describiremos que realiza cada paso de la función.

1. Se inicia la conexión Ethernet con la dirección MAC e IP que se establecen en la cabecera.
2. Abrimos la comunicación Serial y se espera a que se inicie, adicionalmente, esperamos un segundo más para dar tiempo suficiente a inicializarse la conexión Ethernet.
3. Enviamos un paquete a un servidor NTP para obtener la fecha y hora actual.
4. Inicializamos el sensor y esperamos dos segundos para estar seguros de obtener una medida fiable.
5. Establecemos el tiempo que usaremos para calcular más adelante el tiempo sin conectar con el servidor NTP.

A.1.4. Función Loop

Esta función es más larga que la función loop, por lo tanto para documentarla en este formato, se realizará por partes. Por cada fragmento de código que se presenta se describirán los pasos que realiza.

```
1  void loop() {
2    Serial.println("Conectando al Socket...");
3    bool ackLeido = false;
4
5    // Intentamos conectar
6    if (client.connect(server, PUERTO)) {
7    ...
8    } else {
9          Serial.println("Conexion fallada.");
10   }
11
12   if ( (epoch - epochInicial) => 604800000 ) {
13         sendNTPpacket(timeServer);
14         epochInicial = epoch;
15         tiempoAnterior = millis();
16   }
17   if (ackLeido)
18     delay(60000 * 30);
19   else
20     delay(10000);
21 }
```

Código 14: Inicio función loop.

A continuación describiremos los pasos que se realizan en el Fragmento de Código 14.

1. Establecemos la variables ackLeido a falso, nos permitirá indetificar si se ha recibido el mensaje de confirmación del servidor.

2. Intentamos conectar con el servidor, en caso que se realice una conexión entonces se procederá a medir el dato del sensor y enviarlo al servidor (se mostrará más adelante).

3. En caso que no se realice ninguna conexión, entonces no se realiza nada.

4. Comprobamos si el tiempo que ha pasado es mayor de una semana, en caso afirmativo volvemos a obtener la fecha y hora del servidor NTP.

5. Finalmente, si el servidor ha enviado el mensaje de confirmación, entonces esperamos una media hora para volver a mandar un dato, en caso que no se haya recibido, se esperarán diez segundos antes de volver a conectar.

```
Serial.println("Conectado");
DynamicJsonDocument doc (capacity);
String mensaje;

unsigned long tiempoEjecucion = millis();
epoch += (tiempoEjecucion - tiempoAnterior) / 1000;

doc["id"].set(1);
doc["nombre"].set("Arduino");
JsonObject localizacion = doc.createNestedObject("ubicacion");
localizacion["lat"] = 0;
localizacion["lon"] = 0;
JsonArray dato = doc.createNestedArray("dato");

float humedad = dht.readHumidity();
float tempCelsius = dht.readTemperature();
float tempFarenheit = dht.readTemperature(true);

JsonObject dato_0 = dato.createNestedObject();
dato_0["tipo"] = "temperatura";
dato_0["valor"] = tempCelsius;
dato_0["unidad"] = "celsius";
dato_0["timestamp"].set(epoch);

...
serializeJson(doc, mensaje);
client.println(mensaje);
Serial.println("Esperando por ACK");
delay(200);
if (client.available()) {
  ...
}
tiempoAnterior = tiempoEjecucion;
Serial.println("Desconectando");
client.stop();
```

Código 15: Contenido if principal.

A continuación describiremos los pasos que se realizan en el Fragmento de Código 15. Este fragmento de código es el que se ejecutará en caso que el sensor consiga establecer una conexión con el servidor.

1. Creamos el documento JSON a enviar y lo rellenamos con los datos que mide el sensor.
2. Se envía el el documento JSON al servidor para que este lo almacene en la base de datos.
3. Se escucha para la recepción del mensaje ACK por parte del servidor.
4. Actualizamos los tiempos para evitar pedir el tiempo al servidor NTP.

A.1.5. Obtención tiempo de Servidor NTP

```
void sendNTPpacket(IPAddress address) {
  unsigned int localPort = 8888;
  const int NTP_PACKET_SIZE = 48;
  byte packetBuffer[NTP_PACKET_SIZE];
  EthernetUDP udp;
  udp.begin(localPort);
  memset(packetBuffer, 0, NTP_PACKET_SIZE);   // Bytes del buffer a 0

  // Inicializamos los valores para la peticion NTP
  packetBuffer[0] = 0b11100011;  // LI, Version, Mode
  packetBuffer[1] = 0;           // Stratum, or type of clock
  packetBuffer[2] = 6;           // Polling Interval
  packetBuffer[3] = 0xEC;        // Peer Clock Precision
  // 8 bytes, todos con valor 0 para Root Delay & Root Dispersion
  packetBuffer[12]  = 49;
  packetBuffer[13]  = 0x4E;
  packetBuffer[14]  = 49;
  packetBuffer[15]  = 52;

  // Enviamos el paquete NTP
  udp.beginPacket(address, 123);
  udp.write(packetBuffer, NTP_PACKET_SIZE);
  udp.endPacket();
  bool actualizado = false;
  // Esperamos hasta recibir la respuesta del servidor
  while (!actualizado) {
    ...
  }
  udp.stop();
}
```

Código 16: Conexión servidor NTP.

A continuación describiremos los pasos que se realizan en el Fragmento de Código 16. Este fragmento de código es el que se ejecutará cada vez que se obtiene la fecha y hora actual del servidor NTP.

1. Generamos el paquete que enviaremos al servidor NTP y lo rellenamos con la información necesaria.
2. Enviamos el paquete al servidor NTP que se indica como parámetro de la función.
3. Esperamos la respuesta del servidor hasta que se actualicen la fecha y hora.

A.2. Servidor Principal

En esta sección se detallará toda la documentación del código fuente del servidor principal. Este servidor comunica con el sensor para obtener los datos.

A.2.1. Estructura del Proyecto

La estructura que mantiene el proyecto es bastante simple, tenemos una carpeta "src", en la que se encuentran todos los archivos que contienen el código fuente de la aplicación. En el directorio raíz del proyecto encontraremos el archivo ejecutable y el README del proyecto, que indica todo los elementos que son necesarios para ejecutar la aplicación.

A.2.2. Clase Acceptor

```
1  #ifndef Acceptor_h
2  #define Acceptor_h
3
4  #include <iostream>
5  #include "Handler.h"
6  #include <jsoncpp/json/json.h>
7
8  class Acceptor {
9  private:
10     Handler::pointer connection;
11     tcp::acceptor acceptor;
12     void startAccept();
13 public:
14     Acceptor(boost::asio::io_service& io_service);
15     void handleAccept(Handler::pointer connection, const
       ↪  boost::system::error_code& err);
16 };
17 #endif
```

Código 17: Cabecera clase Acceptor

En el Fragmento de Código 17 se muestra la cabecera de la clase Acceptor, a continuación describiremos los atributos que componen la clase, además de las funciones que guarda.

Atributos

- **connection**: Puntero de un objeto de la clase Handler
- **acceptor**: Representa el servicio de E/S que ejecuta la clase Acceptor, aceptar conexiones.

Funciones

```
1 Acceptor::Acceptor(boost::asio::io_service& io_service):
  ↪ acceptor(io_service, tcp::endpoint(tcp::v4(),
  ↪ Config::getInstance()->getPuertoTCP())) {
2     startAccept();
3 }
```

Código 18: Constructor clase Acceptor

En el Fragmento de Código 18 se encuentra el código fuente del constructor de la clase Acceptor, recibe como parámetro un objeto io_service para indicar que el objeto de esta clase se ejecutara como un servicio, crea el servicio acceptor y ejecuta la función “startAccept()” para empezar a aceptar conexiones.

```
1 void Acceptor::startAccept() {
2     connection = Handler::create(acceptor.get_io_service());
3     acceptor.async_accept(connection->getSocket(),
      ↪ boost::bind(&Acceptor::handleAccept, this, connection,
      ↪ boost::asio::placeholders::error));
4 }
```

Código 19: Función startAccept

En el Fragmento de Código 19 se encuentra el código fuente de la función startAccept, esta función se ejecutará constantemente hasta que se apague manualmente el programa, queda a la espera de una conexion entrante para aceptarla.

```
void Acceptor::handleAccept(Handler::pointer connection, const
↪ boost::system::error_code& err) {
    if (!err) {
        connection->start();
    }
    startAccept();
}
```

Código 20: Función handleAccept

En el Fragmento de Código 20 se encuentra el código fuente de la función handleAccept, esta función se ejecutará cada vez que se acepte una conexión, al realizarlo, le da el control de la conexión a un objeto de la clase Handler.

A.2.3. Clase Handler

```
#ifndef Handler_h
#define Handler_h

#include "DataManager.h"
#include <iostream>
...

class Handler: public boost::enable_shared_from_this<Handler> {
private:
    tcp::socket socket;
    boost::asio::streambuf buffer;
public:
    typedef boost::shared_ptr<Handler> pointer;
    Handler(boost::asio::io_service &io_service): socket(io_service)
    ↪ {};

    static pointer create(boost::asio::io_service& io_service);
    void start();
    void handleRead(const boost::system::error_code& err, size_t
    ↪ bytes_transferred);
    tcp::socket& getSocket();
};
#endif
```

Código 21: Cabecera clase Handler

En el Fragmento de Código 21 se encuentra el código fuente de la cabecera de la clase Handler, a continuación describiremos los atributos que componen la clase, además de las funciones que guarda.

Atributos

- **socket**: Socket TCP mediante el cuál la clase podrá recibir y enviar mensajes.
- **buffer**: Buffer en el que se almacena el mensaje recibido por parte del sensor.

Funciones

```
1 Handler::pointer Handler::create(boost::asio::io_service& io_service)
   ↪ {
2     return pointer(new Handler(io_service));
3 }
```

Código 22: Función create

En el Fragmento de Código 22 se encuentra el código fuente de la función create, esta funcion crea un puntero hacia un nuevo objeto de la clase Handler.

```
1 void Handler::start() {
2     boost::asio::async_read_until(socket, buffer, '\r',
3             boost::bind(&Handler::handleRead, shared_from_this(),
               ↪ boost::asio::placeholders::error,
               ↪ boost::asio::placeholders::bytes_transferred));
4 }
```

Código 23: Función start

En el Fragmento de Código 23 se encuentra el código fuente de la función start, esta función se ejecuta cuando se le pasa el control al Handler, queda a la espera de recibir un mensaje con el formato que hemos especificado.

```
void Handler::handleRead(const boost::system::error_code& err, size_t
    bytes_transferred) {
    buffer.commit(bytes_transferred);
    std::istream is(&buffer);
    std::string datos;
    is >> datos;
    if (!err) {
        Json::Reader reader;
        Json::Value obj;
        reader.parse(datos, obj);
        if
        (DataManager::getInstance()->getDispositivo(obj["id"].asInt(),
        obj["nombre"].asString()) != nullptr) {
            boost::asio::write(socket, boost::asio::buffer(ACK_MSG,
            sizeof(ACK_MSG)));
            DataManager::getInstance()->insertFromJSON(obj);
        }
        else {
            ... // ERROR
        }
    }
    else {
        ... // ERROR
    }
    socket.close();
}
```

Código 24: Función handleRead

En el Fragmento de Código 24 se encuentra el código fuente de la función handleRead, esta funcion que se ejecuta cuando se recibe un mensaje por la parte de un cliente, obtenemos los datos del mensaje y se lo pasamos al DataManager para que realice su insercion en la base de datos. Podemos describir el proceso que realiza la función en una serie de pasos.

1. Obtenemos en texto los datos que ha enviado el sensor.
2. Comprobamos si se ha producido algún error en la comunicación.
3. Parseamos el string obtenido a documento JSON.
4. Enviamos ACK al sensor.
5. Insertamos el dato obtenido en la base de datos haciendo uso de DataManager,

A.2.4. Clase DataManager

```
#ifndef DataManager_h
#define DataManager_h

#include <iostream>
...
using namespace sql;
using namespace std;

class DataManager {
private:
    static DataManager *instance;
    Driver *driver;
    Connection *conn;
    PreparedStatement *statement;
    DataManager();

public:
    static DataManager* getInstance();
    void insertMedida(Medida medida);
    void insertDato(Dato dato);
    void insertFromJSON(Json::Value datos);
    bool actualizaUbicacionDispositivo(int id_dispositivo, Ubicacion *
    ↪  ubicacion_actualizada);
    Ubicacion* getUbicacionDispositivo(int id);
    Dispositivo* getDispositivo(int id);
    Dispositivo* getDispositivo(int id, string nombre);
    Medida* getMedida(string tipo, string unidad);
    Dato* getDato(int id);

    ~DataManager() { delete conn; };
};
#endif
```

Código 25: Clase DataManager

En el Fragmento de Código 25 se muestra la cabecera de la clase DataManager, a continuación describiremos los atributos que componen la clase, además de las funciones que guarda.

Atributos

- **instance**: Atributo que nos permite que sólo exista una instancia de la clase DataManager.
- **driver**: Driver a usar para realizar la conexión con la base de datos MySQL.
- **conn**: Atributo que representa la conexión establecida con una base de datos MySQL.
- **statement**: Atributo que nos permite realizar consultas a la base de datos.

Funciones

```
1  DataManager *DataManager::getInstance() {
2      if (!instance) {
3          instance = new DataManager();
4      }
5      return instance;
6  }
```

Código 26: Obtener instancia DataManager

En el Fragmento de Código 26 se encuentra el código fuente de la función getInstance, esta función crea la única instancia que de la clase DataManager en caso que no exista, y en caso que si exista una instancia, devuelve la inicialmente creada.

```
1  DataManager::DataManager() {
2      std::string host = Config::getInstance()->getHostDB();
3      std::string usuario = Config::getInstance()->getUsuario();
4      std::string password = Config::getInstance()->getPassword();
5      std::string schema = Config::getInstance->getSchema();
6
7      driver = get_driver_instance();
8      conn = driver->connect(host, usuario, password);
9      conn->setSchema(schema);
10 }
```

Código 27: Constructor DataManager

En el Fragmento de Código 27 se encuentra el código fuente del único constructor de la clase DataManager, además, dicho constructor sólo podrá ser llamado desde la función getInstance. En el constructor creamos una conexión con una base de datos MySQL que se obtiene a partir de la configuración establecida al ejecutar la aplicación.

```
void DataManager::insertDato(Dato dato) {
    std::time_t result;
    struct std::tm *timeinfo;
    char buffer[30];

    statement =
    ↪  conn->prepareStatement("INSERT INTO datos VALUES (?, ?, ?, ?)");
    statement->setInt(1, dato.id_medida);
    statement->setInt(2, dato.id_dispositivo);
    statement->setDouble(3, dato.valor);

    result = (std::time_t)std::stol(dato.timestamp);
    timeinfo = localtime(&result);
    strftime(buffer, sizeof(buffer), "%F %T", timeinfo);

    statement->setString(4, sql::SQLString(buffer));
    delete statement;
}
```

Código 28: Función insertDato

En el Fragmento de Código 28 se encuentra el código fuente de la función insertDato, esta función inserta el dato que se le pasa como parámetro a la función, en la base de datos.

```
void DataManager::insertFromJSON(Json::Value datos) {
    int idDispositivo = datos["id"].asInt();
    Ubicacion * ubicacion = new Ubicacion;
    ubicacion->lat = datos["ubicacion"]["lat"].asFloat();
    ubicacion->lon = datos["ubicacion"]["lon"].asFloat();

    actualizaUbicacionDispositivo(idDispositivo, ubicacion);
    int numDatos = datos["dato"].size();
    for ( int i = 0; i < numDatos; i++) {
        auto medidaTomada = datos["dato"][i];
        Medida * medida = getMedida(medidaTomada["tipo"].asString(),
        ↪  medidaTomada["unidad"].asString());
        if ( medida != nullptr ) {
            Dato * dato = new Dato;
            dato->id_dispositivo = idDispositivo;
            dato->id_medida = medida->id;
            dato->valor = medidaTomada["valor"].asDouble();
            dato->timestamp = medidaTomada["timestamp"].asString();

            insertDato(*dato);
        }
    }
}
```

Código 29: Función insertFromJSON

En el Fragmento de Código 29 se encuentra el código fuente de la función insertFromJSON, esta función obtiene los datos que se encuentran en el documento JSON para insertarlos en la base de datos. Dependiendo del número de datos diferentes que está midiendo el sensor, el documento JSON tendrá más o menos datos, por lo tanto, se recorre el array de datos del documento JSON, se extraen y se insertan en la base de datos.

```
Dispositivo *DataManager::getDispositivo(int id) {
    Dispositivo *dispositivo = nullptr;
    statement = conn->prepareStatement(...);
    statement->setInt(1, id);
    ResultSet *res = statement->executeQuery();

    if (res->rowsCount() > 0) {
        res->first();
        dispositivo = new Dispositivo;
        dispositivo->id = res->getInt(1);
        dispositivo->nombre = res->getString(2);
        dispositivo->modelo = res->getString(3);
    }
    delete statement;
    delete res;
    return dispositivo;
}
```

Código 30: Función getDispositivo

En el Fragmento de Código 30 se encuentra el código fuente de la primera sobrecarga de la función getDispotivo, esta función nos devuelve el dispositivo que tenga el ID que se pasa como parámetro a la función.

```
Dispositivo *DataManager::getDispositivo(int id, string nombre) {
    Dispositivo *dispositivo = nullptr;
    statement = conn->prepareStatement(...);
    statement->setInt(1, id);
    statement->setString(2, nombre);

    ResultSet *res = statement->executeQuery();
    if (res->rowsCount() > 0) {
        res->first();
        dispositivo = new Dispositivo;
        dispositivo->id = res->getInt(1);
        dispositivo->nombre = res->getString(2);
        dispositivo->modelo = res->getString(3);
    }

    delete statement;
    delete res;
    return dispositivo;
}
```

Código 31: Función getDispositivo

En el Fragmento de Código 31 se encuentra el código fuente de la segunda sobrecarga de la función getDispotivo, esta función nos devuelve el dispositivo que tenga el ID y el nombre que se pasan como parámetros a la función.

Ambas sobrecargas de getDispositivo, serán usadas para conocer si el dispositivo que ha enviado el dato es un dispositivo que se encuentra en la base de datos.

```
1  Ubicacion *DataManager::getUbicacionDispositivo(int id) {
2      ResultSet *res;
3      Ubicacion *ubicacion = nullptr;
4      statement = conn->prepareStatement(...);
5      statement->setInt(1, id);
6      res = statement->executeQuery();
7
8      if (res->rowsCount() > 0) {
9          res->first();
10         ubicacion = new Ubicacion;
11         ubicacion->id = res->getInt(1);
12         ubicacion->lat = res->getDouble(2);
13         ubicacion->lon = res->getDouble(3);
14     }
15     delete statement;
16     delete res;
17     return ubicacion;
18 }
```

Código 32: Función getUbicacion

En el Fragmento de Código 32 se encuentra el código fuente de la función getUbicacion, esta función nos devuelve la ubicación del ID del dispositivo que se le pasa como parámetro a la función.

```
bool DataManager::actualizaUbicacionDispositivo(int id_dispositivo,
    ↪ Ubicacion *ubicacion_actualizada) {
    bool hayCambio = false;
    Ubicacion *ubicacion = getUbicacionDispositivo(id_dispositivo);
    if (ubicacion != nullptr) {
        if (ubicacion->operator!=(*ubicacion_actualizada)) {
            statement = conn->prepareStatement(...);
            statement->setDouble(1, ubicacion_actualizada->lat);
            statement->setDouble(2, ubicacion_actualizada->lon);
            statement->setInt(3, ubicacion->id);
            statement->executeUpdate();

            hayCambio = true;
        }
    }
    delete ubicacion;
    return hayCambio;
}
```

Código 33: Función actualizaUbicacionDispositivo

En el Fragmento de Código 33 se encuentra el código fuente de la función actualizaUbicacionDispositivo, esta función actualiza la ubicación del dispositivo si detecta que sus datos han cambiado. Esto se realiza de esta forma dado que la ubicación estará definida de forma manual en el código del sensor.

```
Medida *DataManager::getMedida(string tipo, string unidad) {
    Medida * medida = nullptr;
    statement = conn->prepareStatement(...);
    statement->setString(1, tipo);
    statement->setString(2, unidad);

    ResultSet *res = statement->executeQuery();
    if (res->rowsCount() > 0) {
        res->first();
        medida = new Medida;
        medida->id = res->getInt(1);
        medida->tipo = tipo;
        medida->unidad = unidad;
    }
    return medida;
}
```

Código 34: Función getMedida

En el Fragmento de Código 50 se encuentra el código fuente de la función getMedida, esta función nos devuelve la medida con el tipo y unidad que se le pasan como parámetros. La función será usada para saber si la medida que nos envía el sensor se encuentra en la base de datos.

A.2.5. Estructuras Adicionales

Adicionalmente para tratar los datos que recibiremos de la base de datos y los datos que introduciremos en la base de datos, tenemos unas estructuras para facilitarnos la tarea.

Ubicación

```
1 struct Ubicacion {
2     int id;
3     float lat;
4     float lon;
5 };
```

Dato

```
1 struct Dato {
2     int id;
3     int id_medida;
4     int id_dispositivo;
5     double valor;
6     std::string timestamp;
7 };
```

Medida

```
1 struct Medida {
2     int id;
3     std::string tipo;
4     std::string unidad;
5 };
```

Dispositivo

```
struct Dispositivo {
    int id;
    std::string nombre;
    std::string modelo;
};
```

A.3. Aplicación Web

En esta sección se detallará la documentación del código fuente de la aplicación web que se ha desarrollado para que presente la información de los datos medidos e insertados en la base de datos.

A.3.1. Estructura del Proyecto

En la carpeta raíz del proyecto nos encontraremos con la siguiente estructura de carpetas para proyectos desarrollados mediante CodeIgniter.

Carpetas

- **application**: La carpeta de la aplicación que hemos desarrollado, aquí se encuentran todos los elementos (controladores, modelos, vistas, etc) que participan en la visualización de la aplicación web.
- **assets**: En esta carpeta se encuentran las hojas de estilo y los scripts de javascript que se usan en la aplicación web.
- **system**: Carpeta de sistema de CodeIgniter, en esta carpeta se encuentran todos los elementos que nos permiten desarrollar aplicaciones web mediante dicho framework.
- **user_guide**: Guía de usuario de CodeIgniter.

A.3.2. Controlador

En un proyecto desarrollado mediante CodeIgniter, los controladores se encuentran en el directorio “application/controllers”.

Home

A continuación mostraremos cada una de las funciones que forman parte del controlador Home, cada una de estas funciones devuelven un resultado, que puede llegar a ser una vista entera, o simplemente parte de la información que se necesite para realizar aplicaciones web más responsivas.

```
1  <?php
2  public function index() {
3          $data['title'] = ucfirst('home');
4          $data['datos'] = $this->datos_model->getAllData();
5          $data['dispositivos'] =
           ↪  $this->dispositivos_model->getAllDispositivos();
6          $data['medidas'] = $this->medidas_model->getAllMedidas();
7          $data['ubicacion'] =
           ↪  $this->ubicacion_model->getAllUbicacion();
8
9          $this->load->view('templates/header', $data);
10         $this->load->view('pages/home', $data);
11         $this->load->view('templates/footer', $data);
12 }
```

Código 35: Función index

En el Fragmento de Código 35 se encuentra el código fuente de la función index del controlador Home, esta función representa la vista inicial de nuestra aplicación web. La función devuelve la cabecera, la vista home y el pie de página, con los datos que se requieran.

```php
<?php
public function dispositivos() {
        $data['title'] = ucfirst('dispositivos');
        $data['datos'] = $this->datos_model->getAllData();
        $data['dispositivos'] =
        ↪ $this->dispositivos_model->getAllDispositivos();
        $data['medidas'] = $this->medidas_model->getAllMedidas();
        $data['ubicacion'] =
        ↪ $this->ubicacion_model->getAllUbicacion();
        $data['medidasDisp'] =
        ↪ $this->dispositivos_model->getMedidasDispositivo();

        $this->load->view('templates/header', $data);
        $this->load->view('pages/dispositivos', $data);
        $this->load->view('templates/footer', $data);
}
```

Código 36: Función dispositivos

En el Fragmento de Código 36 se encuentra el código fuente de la función dispositivos del controlador Home, esta función devuelve la vista segunda vista que se ha diseñado, en la cuál podremos modificar la selección de dispositivos de los cuáles ver los datos.

```php
<?php
public function paginacionDatos() {
        $this->load->library("pagination");
        $id_medida = $this->input->post('medida');
        $config = array();
        // Rellenamos config
        ...
        $this->pagination->initialize($config);
        $page = $this->uri->segment(3);
        $maxPages = $config['total_rows'] / $config['num_links'];
        if ($page > $maxPages) {
                $page = $maxPages;
        }
        $start = ($page - 1) * $config["per_page"];
        $output = array(
                'pagination_link'  =>
                ↪ $this->pagination->create_links(),
                'tabla_datos'   => ''
        );
        // Rellenamos html con la información deseada
        $html = ...;
        if ($config['total_rows']  > 0) {
                $result = $this->datos_model->getDataList($start,
                ↪ $config["per_page"], $id_medida);
                foreach ($result as $row) {
                        $html .= ... ;
                }
        }
        $html .= '</table>';
        $output['tabla_datos'] = $html;
        echo json_encode($output);
}
```

Código 37: Función paginacionDatos

En el Fragmento de Código 37 se encuentra el código fuente de la función paginacionDatos del controlador Home, esta función se encarga de realizar las operaciones necesarias para que en la tabla dividida por páginas que se encuentra en las vistas se muestre la información que se desea.

```php
<?php
public function removeTabData($id) {
        if (!$this->session->userdata('disp_seleccionados')) {
                $datos = array(
                        'disp_seleccionados' => array()
                );
                $this->session->set_userdata($datos);
        }
        $aux = $this->session->userdata('disp_seleccionados');
        for ($i = 0; $i < sizeof($aux); $i++) {
                if ($aux[$i] == $id) {
                        unset($aux[$i]);
                        break;
                }
        }
        $this->session->set_userdata('disp_seleccionados',
        ↪  array_values($aux));
        $result = $this->datos_model->getDataTabs(array_values($aux));
        $output = array(
                'tabs' => $result
        );
        echo json_encode($output);
}
```

Código 38: Función removeTabData

En el Fragmento de Código 38 se encuentra el código fuente de la función removeTabData del controlador Home, esta función elimina de los datos de usuario un dispositivo que ha deseleccionado desde el menú lateral de la vista de dispositivos.

```php
<?php
public function addTabData($id) {
        $idx = intval($id);
        $encontrado = false;
        if (!$this->session->userdata('disp_seleccionados')) {
                $datos = array(
                        'disp_seleccionados' => array()
                );
                $this->session->set_userdata($datos);
        }

        $aux = $this->session->userdata('disp_seleccionados');
        for ($i = 0; $i < sizeof($aux); $i++) {
                if ($aux[$i] == $idx) {
                        $encontrado = true;
                        break;
                }
        }
        if (!$encontrado) {
                array_push($aux, $idx);
        }
        $result = $this->datos_model->getDataTabs($aux);
        $this->session->set_userdata('disp_seleccionados', $aux);
        $output = array(
                'tabs' => $result
        );
        echo json_encode($output);
}
```

Código 39: Función addTabData

En el Fragmento de Código 39 se encuentra el código fuente de la función addTabData del controlador Home, esta función añade a los datos de usuario un dispositivo que se ha seleccionado desde el menú lateral de la vista de dispositivos.

```php
<?php
public function graphData($id) {
        $dispositivos =
        ↪  $this->dispositivos_model->getMedidaDispositivo($id);
        $medida = $this->medidas_model->getMedida($id);
        $datasets = array(
                'datasets' => array()
        );
        foreach ($dispositivos as $dispositivo) {
                $nombre = $dispositivo['nombre'] . ' ' .
                ↪  $dispositivo['id_dispositivo'];
                $arrDisp = array(
                        'label' => $nombre,
                        'lineTension' => 0,
                        'backgroundColor' => 'transparent',
                        'borderColor' => ...,
                        'pointBackgroundColor' => ...,
                        'data' => array()
                );
                $datosDispositivo = $this->datos_model->...;
                foreach ($datosDispositivo as $dato) {
                        $aux = array_merge($arrDisp['data'],
                        ↪  array(array(
                                'x' => $dato['timestamp'],
                                'y' => $dato['valor']
                        )));
                        $arrDisp['data'] = $aux;
                }
                array_push($datasets['datasets'], $arrDisp);
        }
        echo json_encode($datasets);
}
```

Código 40: Función graphData

En el Fragmento de Código 40 se encuentra el código fuente de la función graphData del controlador Home, esta función devuelve los datos que se mostrarán en la gráfica de la pestaña que esta actualmente seleccionada.

```php
<?php
function addMedida() {
        $nombre = trim($this->input->post('nombre'));
        $unidad = trim($this->input->post('unidad'));
        if (strlen($nombre) > 0 && strlen($unidad) > 0) {
                $output = array(
                        'exito' => true,
                );
                $this->medidas_model->addMedida($nombre, $unidad);
        } else {
                $output = array(
                        'exito' => false,
                        'error' =>
                        ↪  'Datos introducidos no v&aacute;lidos.'
                );
        }
        echo json_encode($output);
}
```

Código 41: Función addMedida

En el Fragmento de Código 41 se encuentra el código fuente de la función addMedida del controlador Home, esta función envía los datos introducidos por el usuario al modelo para crear una nueva medida. Añadir una nueva medida a la base de datos habilita a los dispositivos que miden dicha medida mediante un sensor a almacenar los datos en la base de datos.

```php
<?php
function addDispositivo() {
        $nombre = trim($this->input->post('nombre'));
        $modelo = trim($this->input->post('modelo'));
        if (strlen($nombre) > 0 && strlen($modelo) > 0) {
                $output = array(
                        'exito' => true,
                );
                $this->dispositivos_model->addDispositivo($nombre,
                ↪  $modelo);
        } else {
                $output = array(
                        'exito' => false,
                        'error' =>
                        ↪  'Datos introducidos no v&aacute;lidos.'
                );
        }
        echo json_encode($output);
}
```

Código 42: Función addDispositivo

En el Fragmento de Código 42 se encuentra el código fuente de la función addDispositivo del controlador Home, esta función envía los datos introducidos por el usuario al modelo para crear un nuevo dispositivo. Añadir un nuevo dispositivo a la base de datos, habilita a dicho dispositivo a almacenar los datos que envía en la base de datos.

A.3.3. Modelos

En un proyecto desarrollado mediante CodeIgniter, los modelos se encuentran en el directorio “application/models”.

Datos

A continuación se mostrarán las funciones que contiene el modelo “Datos”.

```
<?php
public function getAllData() {
        $query = $this->db->get('datos');
        return $query->result_array();
}
```

Código 43: Función getAllData

En el Fragmento de Código 43 se encuentra el código fuente de la función getAllData del modelo de Datos, esta función devuelve todo el contenido de la tabla “datos”.

```
<?php
public function getDatosMedida($id) {
        $this->db->select('dispositivos.id, dispositivos.nombre' +
        ↪  'datos.valor, datos.timestamp');
        $this->db->from('datos');
        $this->db->join('dispositivos',
        ↪  'dispositivos.id = datos.id_dispositivo');
        $this->db->where(array(
                'datos.id_medida' => $id
        ));
        $query = $this->db->get();
        return $query->result_array();
}
```

Código 44: Función getDatosMedida

En el Fragmento de Código 44 se encuentra el código fuente de la función getDatosMedidos del modelo de Datos, esta función devuelve todos los datos medidos de una medida en específico que se identifica por su id.

```php
<?php
public function getDatosMedidaDispositivo($id_dispositivo, $id_medida)
 ↪  {
        $this->db->select('datos.valor, datos.timestamp');
        $this->db->from('datos');
        $this->db->where(array(
                'datos.id_medida' => $id_medida,
                'datos.id_dispositivo' => $id_dispositivo
        ));
        $query = $this->db->get();
        return $query->result_array();
}
```

Código 45: Función getDatosMedidaDispositivo

En el Fragmento de Código 45 se encuentra el código fuente de la función getDatosMedidaDispositivo del modelo de Datos, esta función devuelve todos los datos medidos de una medida y un dispositivo específico, identificados por el *id* de la medida y el *id* del dispositivo.

```php
<?php
public function getDataList($start, $limit, $id_medida) {
        $this->db->select('datos.id, medidas.nombre' +
         ↪  ' AS "medida", dispositivos.nombre,' +
         ↪  ' dispositivos.modelo, datos.valor,' +
         ↪  ' medidas.unidad, datos.timestamp');
        $this->db->join('medidas', 'medidas.id = datos.id_medida');
        $this->db->join('dispositivos',
         ↪  'dispositivos.id = datos.id_dispositivo');
        $this->db->order_by('timestamp');
        $this->db->limit($limit, $start);
        $query = $this->db->get_where('datos', array('medidas.id' =>
         ↪  $id_medida));

        return $query->result_array();
}
```

Código 46: Función getDataList

En el Fragmento de Código 46 se encuentra el código fuente de la función getDataList del modelo de Datos, esta función devuelve los datos de una medida en específica identificándola mediante su *id*. Dicha función se usa para poder partir la tabla en páginas y de esta forma la mostramos en la aplicación web.

```php
<?php
public function getNumRows() {
        $query = $this->db->get('datos');
        return $query->num_rows();
}
```

Código 47: Función getNumRows

En el Fragmento de Código 47 se encuentra el código fuente de la función getNumRows del modelo de Datos, esta función devuelve el número de filas que contiene la tabla “datos”.

```php
<?php
public function getNumRowsMedida($id_medida) {
        $query = $this->db->get_where('datos', array('datos.id_medida'
        ↪  => $id_medida));
        return $query->num_rows();
}
```

Código 48: Función getNumRowsMedida

En el Fragmento de Código 48 se encuentra el código fuente de la función getNumRowsMedida del modelo de Datos, esta función devuelve el número de datos que existen con una medida en específico.

Medidas

A continuación se mostrarán las funciones que contiene el modelo “Medidas”.

```php
<?php
public function getAllMedidas() {
        $query = $this->db->get('medidas');
        return $query->result_array();
}
```

Código 49: Función getAllMedidas

En el Fragmento de Código 49 se encuentra el código fuente de la función getAllMedidas del modelo de Medidas, esta función devuelve todo el contenido de la tabla “medidas”.

```php
<?php
public function getMedida($id) {
        $query = $this->db->get_where('medidas', array('id' => $id));
        return $query->result_array();
}
```

Código 50: Función getMedida

En el Fragmento de Código 50 se encuentra el código fuente de la función getMedida del modelo de Medidas, esta función devuelve la información sobre una medida con un id en concreto.

```php
<?php
public function addMedida($nombre, $unidad) {
        $data = array(
                'nombre' => $nombre,
                'unidad' => $unidad
        );
        $this->db->insert('medidas', $data);
}
```

Código 51: Función addMedida

En el Fragmento de Código 51 se encuentra el código fuente de la función addMedida del modelo de Medidas, esta función añade un nuevo dato a la tabla medidas, es decir, crea una nueva medida con el nombre y la unidad que se especifican en los parámetros.

Dispositivos

A continuación se mostrarán las funciones que contiene el modelo “Dispositivos”.

```php
<?php
public function getAllDispositivos() {
        $query = $this->db->get('dispositivos');
        return $query->result_array();
}
```

Código 52: Función getAllDispositivos

En el Fragmento de Código 52 se encuentra el código fuente de la función getAllDispositivos del modelo de Dispositivos, esta función devuelve todo el contenido de la tabla “dispositivos”.

```php
<?php
public function addDispositivo($nombre, $modelo) {
        $data = array(
                'id_ubicacion' => 1,
                'nombre' => $nombre,
                'modelo' => $modelo
        );
        $this->db->insert('dispositivos', $data);
}
```

Código 53: Función addDispositivo

En el Fragmento de Código 53 se encuentra el código fuente de la función addDispositivo del modelo de Dispositivos, esta función añade un nuevo dato a la tabla de dispositivos, es decir, crea un nuevo dispositivo con el nombre y el modelo que se especifican en los parámetros.

Ubicación

A continuación se mostrarán las funciones que contiene el modelo “Ubicación”.

```php
<?php
public function getAllUbicacion() {
        $query = $this->db->get('ubicacion');
        return $query->result_array();
}
```

Código 54: Función getAllUbicacion

En el Fragmento de Código 54 se encuentra el código fuente de la función getAllUbicacion del modelo de Ubicacion, esta función devuelve todas las ubicaciones que se encuentran guardadas en la tabla “ubicacion”.

A.3.4. Vistas

En este apartado se describirán las funciones JavaScript que se han implementado para las vistas que se han mencionado anteriormente en el proyecto.

```javascript
function visualizacionSemanal() {
    selectedUnit = "week";
    myChart.options.scales.xAxes[0].time.unit = "week";
    myChart.update();
}
function visualizacionDia() {
    selectedUnit = "day";
    myChart.options.scales.xAxes[0].time.unit = "day";
    myChart.update();
}
function visualizacionHora() {
    selectedUnit = "hour";
    myChart.options.scales.xAxes[0].time.unit = "hour";
    myChart.update();
}
```

Código 55: Funciones de visualización

En el Fragmento de Código 55 se encuentran las funciones JavaScript que se ejecutarán cuando el usuario decida cambiar los datos que se muestran en el eje X de las gráficas. Cambiar estos datos nos permite mantener una mejor visualización de los momentos en los que se ha medido un dato.

```
function mostrarUbicacion(ubicacion) {
    map = new google.maps.Map(document.getElementById('map'), {
        center: {
            lat: $(ubicacion).data('lat'),
            lng: $(ubicacion).data('lon')
        },
        zoom: 8
    });
    var marker = new google.maps.Marker({
        position: {
            lat: $(ubicacion).data('lat'),
            lng: $(ubicacion).data('lon')
        },
        map: map
    });
    $('#modalCenter').modal();
}
```

Código 56: Funciones mostrarUbicación

En el Fragmento de Código 56 se encuentra la función mostrarUbicacion, esta función mostrará una ventana modal con un mapa de la ubicación del dispositivo seleccionado.

```
function loadChart(json) {
    if (myChart)
        myChart.destroy();
    myChart = new Chart(ctx, {
        type: 'line',
        data: json,
        options: {
            scales: {
                xAxes: [{
                    type: 'time',
                    time: {
                        min: minDateSelected,
                        max: maxDateSelected,
                        unit: selectedUnit,
                        displayFormats: {
                            'minute': 'HH:mm',
                            'hour': 'DD MMM HH:mm',
                            'day': 'DD/MM/YYYY',
                            'week': 'DD MMM YYYY'
                        }
                    }
                }],
                yAxes: [{
                    ticks: {
                        beginAtZero: true
                    }
                }]
            },
            legend: {
                display: true
            }
        }
    });
    $('#medida' + idMedidaActual).addClass("active");
    $('#tabContent').tab('show');
}
```

Código 57: Funciones loadChart

En el Fragmento de Código 57 se encuentra la función loadChart, esta función carga una gráfica con los datos que se especifican como parámetro en la función. Los datos se recibirán por mediante llamadas asíncronas (AJAX).

```
function getTree() {
    tree = [
        <?php foreach ($dispositivos as $disp) : ?>
        {
            text: "<?= $disp['nombre'] . ' ' . $disp['modelo']; ?>",
            id_disp: <?= $disp['id'] ?>,
            state: {
                checked: false,
                disabled: false,
                expanded: true,
                selected: false
            }
        },
        <?php endforeach; ?>
    ];
    return tree;
}
```

Código 58: Funciones getTree

En el Fragmento de Código 58 se encuentra la función getTree, esta función carga el menú lateral de dispositivos que se encuentra en la vista de dispositivos. Los datos de los dispositivos se obtienen gracias al controlador “Home”, que nos permite enviar datos hacia las vistas.

El resto de funciones que podemos encontrar en el archivo *myjs.js* y en el pié de página de las vistas, son funciones que realizan llamadas asíncronas para actualizar los datos que se muestran en pantalla. De esta forma, cuando el usuario realiza una interacción con la aplicación web, no es necesario recargar la página entera, y sólo se actualiza el elemento que se desea.

B Manual de Usuario

B.1. Prerrequisitos

En esta sección describiremos los elementos que son necesarios para el correcto funcionamiento del software que se ha desarrollado. Además, se mostrará cómo obtenerlos para un sistema Raspberry Pi con Raspbian.

B.1.1. Servidor Principal

Los prerrequisitos del servidor principal son los siguientes:

- **make**: `sudo apt install make`
- **libboost-all-dev**: `sudo apt install libboost-all-dev`
- **libjsoncpp-dev**: `sudo apt install libjsoncpp-dev`
- **libmysqlcppconn-dev**: `sudo apt install libmysqlcppconn-dev`

Con los elementos mencionados anteriormente instalados, podemos proceder con la compilación del servidor principal.

B.1.2. Base de datos

Los prerrequisitos de la base de datos son los siguientes:

- **mysql-client**: `sudo apt install mysql-client`
- **mysql-server** `sudo apt install mysql-server`

Con los elementos anteriores instalados, podemos proceder a crear una base de datos y un usuario que la pueda manejar, no está recomendado usar el usuario root cuando se quieren realizar operaciones con bases de datos, supone un riesgo de seguridad.

En la nueva versión de Raspbian (Raspbian Buster) los paquetes mysql-client y mysql-server son sustituidos por mariadb-client y mariadb-server. MariaDB es una implementación de código abierto de MySQL.

Creación de base de datos y usuario

Con los paquetes necesarios instalados, conectaremos con el servidor MySQL usando el usuario “root” para crear la base de datos, y un usuario adicional para trabajar con la base de datos creada. Para crear una nueva base de datos, se realizará lo siguiente:

1. Accedemos al servidor MySQL (desde la Raspberry Pi) usando el usuario root: `sudo mysql`
2. Creamos una nueva base de datos con el nombre que se desee, como ejemplo se usará “tfg_recogida_datos”: `CREATE DATABASE tfg_recogida_datos;`
3. Creamos un usuario, definimos el nombre de usuario, los equipos que a los cuáles se les permite acceso a dicho usuario y la contraseña.
   ```
   CREATE USER 'tfg'@'192.168.1.%' IDENTIFIED BY 'contraseña';
   CREATE USER 'tfg'@'localhost' IDENTIFIED BY 'contraseña';
   ```
4. Definimos los permisos del nuevo usuario, tendrá todos los permisos sobre la base de datos previamente creada.
   ```
   GRANT ALL PRIVILEGES ON tfg_recogida_datos.* TO 'tfg'@'192.168.1.%';
   ```
5. Realizamos un flush de los privilegios para que tomen efecto los cambios.
   ```
   FLUSH PRIVILEGES;
   ```

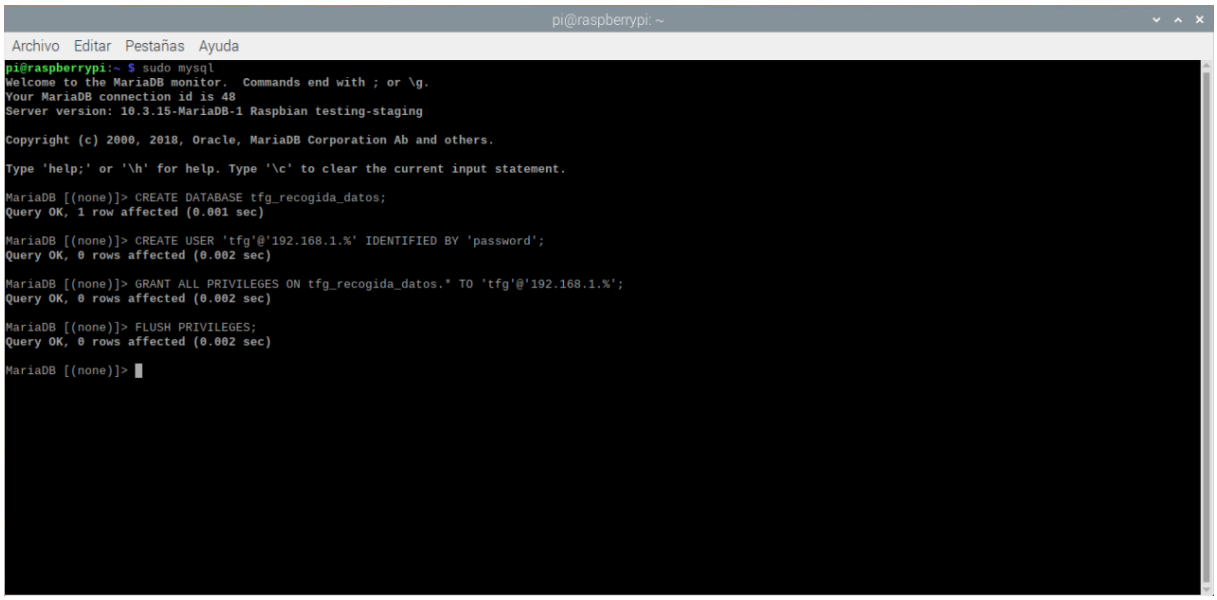


Figura B.1: Creación de la base de datos y usuario.

Una vez realizado los pasos anteriores, procederemos a conectarnos al servidor MySQL con el usuario que hemos creado. La conexión se puede realizar desde un segundo equipo (`mysql -u tfg -p -h IP_EQUIPO_RASPBERRY`) o desde la Raspberry Pi misma (`mysql -u tfg -p`).

Cuando estemos conectados al servidor, podemos empezar a crear las tablas de nuestra base de datos.

B.1.3. Servidor Web

- **apache2**: `sudo apt install apache2`
- **libapache2-mod-php**: `sudo apt install libapache2-mod-php`

Con los paquetes anteriores instalados, desde un navegador web escribiremos la IP de nuestro servidor y se mostrará la página de prueba de Apache2 (Figura B.2).

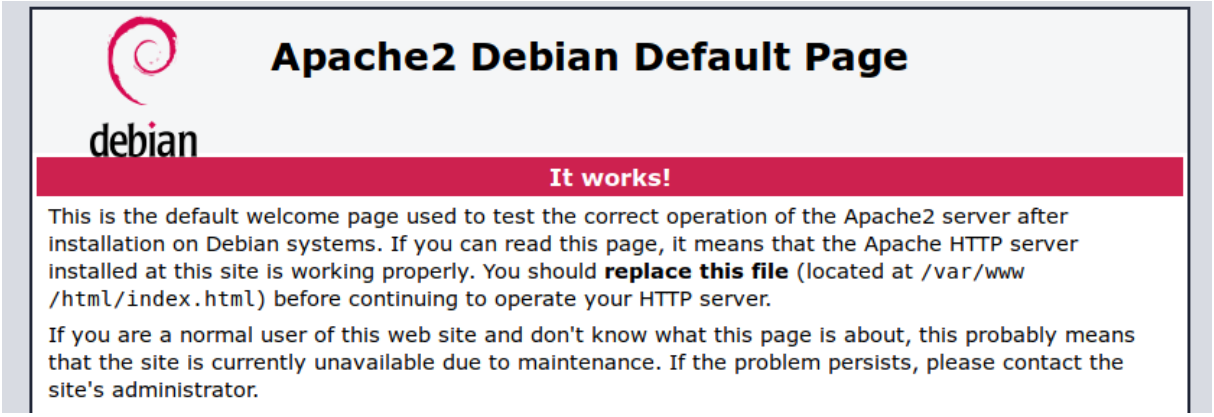


Figura B.2: Página de prueba de Apache2.

Si todo funciona correctamente, podemos proceder a eliminar el archivo *index.html* de apache y copiar el contenido de nuestro proyecto de CodeIgniter a la carpeta /var/www/html.

Configuración CodeIgniter

Para que nuestra aplicación desarrollada mediante CodeIgniter funcione correctamente, debemos de modificar dos archivos: config.php y database.php. Ambos de estos archivos se encuentran en el directorio /var/www/html/application/config/ .

En el archivo config.php deberemos de modificar la línea 26 por la IP del equipo, y en el archivo database.php deberemos de modificar a partir de la línea 76, estableciendo la configuración de la base de datos.

B.2. Compilación

En este apartado se mencionará el método de compilación del servidor principal que ha sido desarrollado en el lenguaje de programación C++.

B.2.1. Servidor Principal

Para la compilación del programa se hace uso de un archivo llamado “Makefile”, que contiene un conjunto de directivas usadas por el software Make. Dicho software, realiza de forma automática la compilación del código fuente para obtener el ejecutable final.

El Makefile del proyecto se encuentra en el directorio base del servidor principal, está diseñado para realizar la compilación en sistemas Linux que tengan los paquetes mencionados anteriormente en los prerrequisitos, ya instalados.

```
1  src = $(wildcard src/*.cpp)
2  obj = $(src:.cpp=.o)
3
4  LDFLAGS = -ljsoncpp -lmysqlcppconn -lboost_system -lboost_regex
   ↪  -lboost_program_options
5
6  server : $(obj)
7          g++ -o $@ $^ $(LDFLAGS)
8
9  clean:
10         rm -f $(obj) server
```

Código 59: Makefile

Para proceder con la compilación es tan simpe cómo encontrarnos en el directorio base del servidor principal y ejecutar `make` desde la línea de comandos.

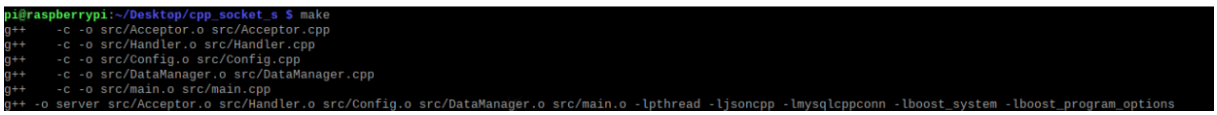


Figura B.3: Ejecución de make.

B.3. Ejecución

En esta sección describiremos cómo ejecutar cada uno de los componentes software que se han desarrollado a lo largo del proyecto.

B.3.1. Servidor Principal

Con el servidor principal ya compilado, se podrá ejecutar desde la línea de comandos. Adicionalmente, se deberán de establecer los parámetros necesarios para que se realice la conexión con la base de datos.

Escribiendo `./server --help` tenemos disponible las opciones que admite el programa. Las opciones obligatorias a rellenar son: la base de datos a conectar, el usuario de la base de datos y la contraseña del usuario.

Un ejemplo sería el siguiente:
`./server -s tfg_recogida_datos -u tfg -p password`.

Si la configuración es correcta, se mostrará un mensaje indicando que la conexión con la base de datos ha sido realizada.

B.3.2. Base de datos

La base de datos estará en ejecución desde que se instalan los paquetes, pero podemos comprobar su ejecución mediante el comando:
`sudo systemctl status mysql`. Dicho comando nos mostrará una salida con la información referente al servicio. Podemos ver su estado en el atributo “Active”.

```
pi@raspberrypi:~/Desktop/cpp_socket_s $ sudo systemctl status mysql
● mariadb.service - MariaDB 10.3.15 database server
   Loaded: loaded (/lib/systemd/system/mariadb.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2019-07-15 15:53:27 BST; 21h ago
     Docs: man:mysqld(8)
           https://mariadb.com/kb/en/library/systemd/
 Main PID: 4206 (mysqld)
   Status: "Taking your SQL requests now..."
    Tasks: 31 (limit: 2200)
   Memory: 54.0M
   CGroup: /system.slice/mariadb.service
           └─4206 /usr/sbin/mysqld
```

Figura B.4: Comprobación de ejecución BBDD.

B.3.3. Servidor Web

El servidor web, al igual que la base de datos se encontrará en ejecución desde la instalación del paquete. Para comprobar el estado del servicio se hará uso del comando: `sudo systemctl status apache2`.

```
pi@raspberrypi:~/Desktop/cpp_socket_s $ sudo systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2019-07-15 16:37:16 BST; 20h ago
     Docs: https://httpd.apache.org/docs/2.4/
  Process: 18724 ExecStart=/usr/sbin/apachectl start (code=exited, status=0/SUCCESS)
 Main PID: 18729 (apache2)
    Tasks: 11 (limit: 2200)
   Memory: 61.7M
   CGroup: /system.slice/apache2.service
           ├─18729 /usr/sbin/apache2 -k start
           ├─18733 /usr/sbin/apache2 -k start
           ├─18736 /usr/sbin/apache2 -k start
           ├─18737 /usr/sbin/apache2 -k start
           ├─18746 /usr/sbin/apache2 -k start
           ├─18752 /usr/sbin/apache2 -k start
           ├─18753 /usr/sbin/apache2 -k start
           ├─18754 /usr/sbin/apache2 -k start
           ├─18990 /usr/sbin/apache2 -k start
           ├─19527 /usr/sbin/apache2 -k start
           └─23343 /usr/sbin/apache2 -k start
```

Figura B.5: Comprobación de ejecución de apache.

B.4. Navegación Página Web

En esta sección se mostrará cómo se realiza la navegación a través de la aplicación web desarrollada para visualizar los datos recogidos.

Página inicial

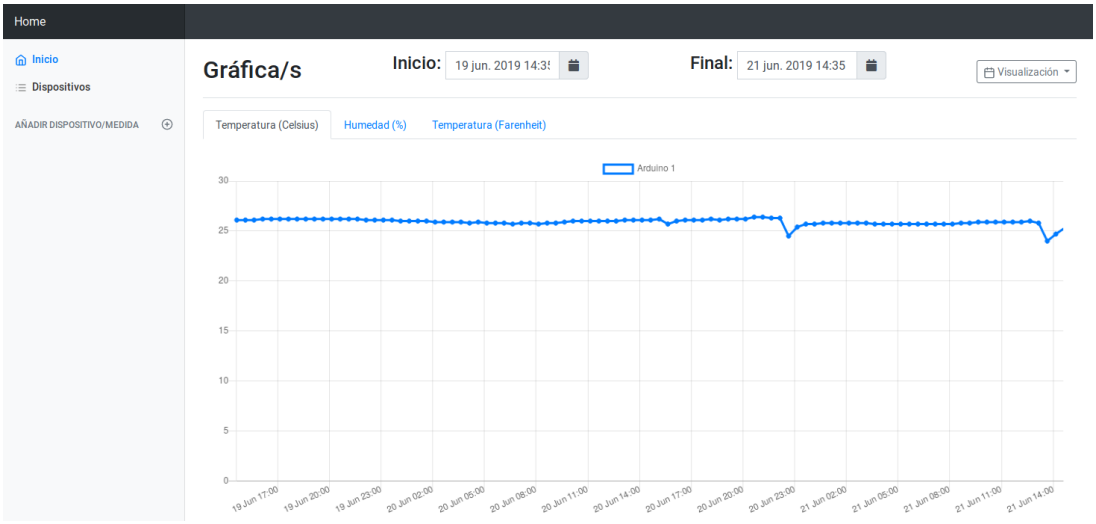


Figura B.6: Página Inicial.

En la Figura B.6 se puede ver la página inicial que aparece cuando navegamos a la dirección IP del equipo Raspberry Pi que está ejecutando los servidores.

Página Dispositivos

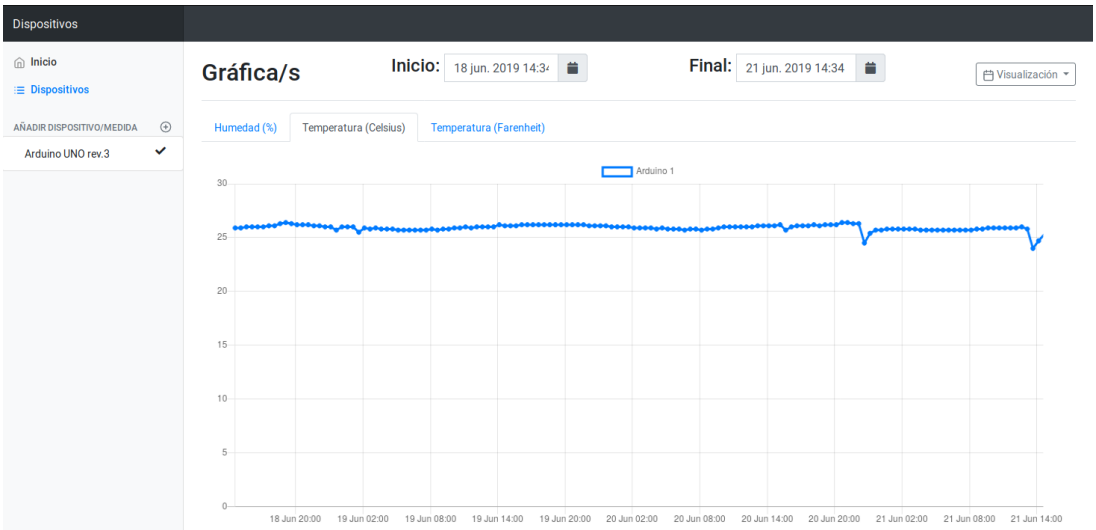


Figura B.7: Página Dispostivos.

En la Figura B.7 se puede ver la página de dispositivos, está página nos permite seleccionar los dispositivos de los cuales deseamos que aparezcan los datos en las gráficas. Adicionalmente, de forma dinámica se activarán y desactivarán las pestañas de medidas, dejando activas las medidas que si están siendo recogidas por los dispositivos seleccionados.

Añadir medida o dispositivo

En el menú lateral izquierdo si se presiona sobre el símbolo +, se abrirá una ventana modal en la cuál podremos seleccionar si añadir una medida nueva o un nuevo dispositivo.

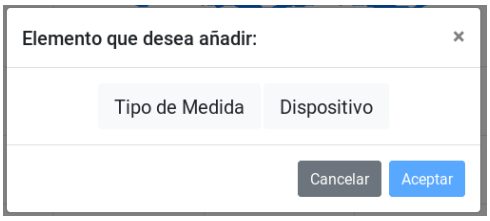


Figura B.8: Añadir elemento

Presionando sobre el elemento que se desea añadir (Figura B.8), la ventana moda cambiará su contenido para introducir los datos necesarios del nuevo elemento.

Interacción con elementos

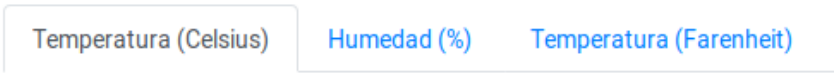


Figura B.9: Pestañas de medidas.

A través de las pestañas de medidas, podemos seleccionar la medida de la cuál queremos que se representen los datos en la gráfica.

Figura B.10: Seleccionador de fechas.

A través de los seleccionadores de fecha y hora, podemos cambiar la fecha inicial y final de representación de datos en la gráfica.

Figura B.11: Intervalo de visualización

A través del menú “Visualización” las entradas mostradas en el eje X cambiarán en función de la representación seleccionada. El salto entre cada entrada será definido automáticamente por la gráfica.