**Documentación**

**Técnica - AirHealthy**

Modelo Fijo

**Sistema de Monitoreo de Niveles de Dióxido de Carbono (CO2) presentes en el aire usando ESP32 y un Raspberry PI**



Índice

[**Introducción 4**](#_lpk63sis8a0f)

[**Dependencias del Servidor 5**](#_8bfhrmqmze0r)

[NodeJS 5](#_bvhg4u6rtg0v)

[NPM 5](#_34d7bl2pppnf)

[Bibliotecas 5](#_hoxidchkqac0)

[**Variables Globales 7**](#_2dwh07g5p436)

[**Modelo de Red WLAN 8**](#_vw8iw9iwg4xo)

[Datos de Conexión para Interfaz de red ‘Interna’ del Raspberry Pi 9](#_sczeq4z2cz0c)

[**Detalles del Servidor 10**](#_6sm8ylh5u3l)

[1. Propósito del Servidor 10](#_3v06l1hiamc4)

[2. Configuración y Conexión 10](#_tinb31e8l6jd)

[3. Puntos de Conexión (endpoints) 11](#_gbpxnv2oe3vv)

[4. Métodos de Autenticación 12](#_cfbaq9i9lf9n)

[**Diccionario de Datos - SQLite 13**](#_qas0zbm0mkib)

[**Componentes del Sistema 14**](#_bfto8bsg8t71)

[**Conexión al Servidor por medio de SSH 16**](#_c510djbgw1cy)

[**Requerimientos H&W 17**](#_t3c5o7s0z6i3)

[**Funciones Esenciales del Servidor 18**](#_h0st9ttysexf)

[1. Función: serverEnd() 18](#_p6wu4twsri4q)

[2. Función: serverListen() 18](#_u4q6eme55czj)

[3. Función: exception() 19](#_z1rj2mqivu9r)

[**Funciones del Código Fuente 20**](#_o51jkblz4gcj)

[1. \_\_POST\_\_REGISTRAR\_MEDICION\_\_() 20](#_fiamohrtjrzu)

[Descripción 20](#_kwl6fj2zg8wg)

[Parámetros 20](#_gq8dq9ys6a1l)

[Proceso 20](#_5ch38vib1zxh)

[Errores y Excepciones 23](#_ugnksckccx1s)

[2. \_\_GET\_\_MEDICIONES\_PROMEDIADO\_\_() 24](#_za5g0zhc929n)

[Descripción 24](#_mt1zks9p5szm)

[Parámetros 24](#_ldbldtqlup6l)

[Proceso 24](#_55anwa5lceg)

[Errores y Excepciones 26](#_j637is1es5jy)

[3. \_\_GET\_\_LISTA\_MEDICIONES\_\_() 27](#_e054yjch0k8p)

[Descripción 27](#_e6ahpeptevrw)

[Parámetros 27](#_1w7pnsud5ghg)

[Proceso 27](#_27tt41vuh7fz)

[Errores y Excepciones 28](#_ej7y7vwkzzcq)

[4. \_\_GET\_\_DASHBOARD\_\_() 29](#_jtr5gqmofdab)

[Descripción 29](#_22fi94tl4w0z)

[Parámetros 29](#_3wjj4y749wy9)

[Proceso 29](#_ea1sfgwdsb4n)

[Notas Adicionales 30](#_o24efh3q233x)

[Errores y Excepciones 30](#_4r1k20hq6elb)

[5. \_\_GET\_\_DESCARGAR\_\_() 31](#_lidqr59xwlrq)

[Descripción 31](#_qgis51n464wk)

[Parámetros 31](#_nffndzfg6gz2)

[Proceso 31](#_7xsoyxvq8blr)

[Errores y Excepciones 32](#_7vkyxwbf7gce)

[**Funciones Complementarias 33**](#_yyxedgttjl7j)

[1. Función: obtenerTablaArp(dev) 33](#_o4ga317zaii)

[Argumentos 33](#_ttt8yfdrclta)

[Salida 34](#_tikvumj3f1pf)

[Errores 34](#_t3gr90n7rxs8)

[2. Función: obtenerNombreHost(ip) 35](#_737h7bhxk4q1)

[Argumentos 35](#_6kwf6xp034yy)

[Salida 35](#_pvuulf5ju1og)

[Errores 35](#_8scgydpxpps5)

[3. Función: lista\_arp(dev) 36](#_xi0sty13dc9z)

[Argumentos 36](#_k6rvrzsva6hq)

[Salida 36](#_kzf1a9k3ln08)

[Errores 36](#_svrr5syznd99)

[4. Función: lista\_medidores() 37](#_rejymnf1nius)

[Argumentos 37](#_5cxxi3bbn0ux)

[Salida 37](#_8slzxs411u7j)

[Errores 37](#_s227y9xmezeo)

[5. Exportación de las Funciones 38](#_kast0o1etppw)

[Se exportan todas las funciones que se harán uso en el archivo socket.js del proyecto (Imagen 9.8). 38](#_ymsjq7qpzx42)

# 

# Introducción

En la era actual, donde la preocupación por la calidad del aire y el medio ambiente está en constante aumento, se vuelve crucial desarrollar tecnologías que nos permitan monitorear y comprender mejor nuestra atmósfera. En este contexto, surge nuestro proyecto de un sensor de calidad de aire utilizando componentes como el sensor de gas MQ-135 y el sensor de temperatura y humedad DHT-11, integrados con el poderoso microcontrolador ESP32. El objetivo principal de este proyecto es proporcionar una solución innovadora y accesible para la medición y monitoreo en tiempo real de la calidad del aire.

Para lograr este propósito, hemos diseñado un sistema completo que no solo recopila datos de manera precisa y confiable, sino que también los transmite de forma eficiente a un servidor centralizado. Este servidor, alojado en una Raspberry Pi, utiliza tecnologías como Node.js y SQLite para gestionar la recepción, almacenamiento y procesamiento de los datos ambientales proporcionados por los sensores.

Una de las características destacadas de nuestro sistema es la interfaz web que proporciona una visualización clara y detallada de los datos recopilados. Esta interfaz permite a los usuarios monitorear en tiempo real los niveles de dióxido de carbono, temperatura y humedad del ambiente, además de ofrecer estadísticas útiles como el promedio de datos de la última hora, del día actual y de las últimas 24 horas. Además, hemos implementado una funcionalidad que permite la descarga de los datos almacenados en un formato conveniente, facilitando su análisis y uso posterior.

# Dependencias del Servidor

## NodeJS

| Node.js es un entorno de ejecución de JavaScript en el lado del servidor. Utiliza el motor V8 de Google, es conocido por su manejo asíncrono eficiente, y viene con NPM para gestionar dependencias. Ampliamente utilizado en el desarrollo web, es versátil, de código abierto y permite a los desarrolladores utilizar JavaScript para construir aplicaciones del lado del servidor (Imagen 1.1). | Imagen 1.1. Entorno de Ejecución NodeJS |
| --- | --- |

## NPM

| NPM, o Node Package Manager, es un componente fundamental del ecosistema de Node.js que facilita la gestión de dependencias en proyectos de software. Permite a los desarrolladores instalar, compartir y gestionar bibliotecas y herramientas utilizando comandos simples como npm install. Además, sirve como un repositorio global que alberga una amplia variedad de paquetes públicos que pueden ser incorporados en proyectos. Con la capacidad de automatizar tareas mediante scripts y seguir un sistema de versionado semántico, NPM mejora la eficiencia en el desarrollo de software en Node.js (Imagen 1.2). | Imagen 1.2. Gestor de Paquetes NPM |
| --- | --- |

## Bibliotecas

La interfaz de programación (API) requiere de algunas dependencias para su correcto funcionamiento. No olvidemos que usamos NodeJS para darle forma al servidor.

| Imagen 1.3. Bibliotecas del archivo “constantes.js” perteneciente al proyecto del servidor. |
| --- |

En el caso del archivo **constantes.js**, se hace uso de las bibliotecas correspondientes (Imagen 1.3).

Por el otro lado, tenemos al archivo con nombre **socket.js** el cual dispone también de las bibliotecas necesarias para hacer posible el servidor (Imagen 1.4).

| Imagen 1.4. Bibliotecas del archivo “socket.js” perteneciente al proyecto del servidor. |
| --- |

Y por último **constantes** y **server**, los cuales no son bibliotecas de Node.js, más bien son módulos (archivos del proyecto) que existen en el directorio de nuestro sistema, es decir, del servidor y estos contienen las funciones complementarias y esenciales para hacer funcionar el servicio (Imagen 1.5).

| Imagen 1.5. Importación de archivos “constantes.js” y “server.js” pertenecientes al directorio del proyecto. |
| --- |

# Variables Globales

Las variables globales en el código cumplen funciones clave en la configuración y funcionamiento del servidor, así como en la gestión de datos. Establecen la configuración del servidor, como el puerto en el que escucha las solicitudes y la ruta a la base de datos. También facilitan el acceso y la manipulación de datos, permitiendo la conexión con la base de datos y proporcionando un nombre para la tabla donde se almacenan las mediciones. Estas variables son esenciales para garantizar que el servidor funcione correctamente y que los datos se gestionen de manera eficaz (Imagen 2.1 e Imagen 2.2).

A continuación se presenta la manera en que se declaran las variables en el servidor.

| Imagen 2.1. | Imagen 2.2 |
| --- | --- |

# Modelo de Red WLAN

1. Cliente (sensor) - Servidor

Referencia de Imagen: [Imagen 3.1]

| Imagen 3.1. Modelo de red WLAN del servidor en función de los sensores. |
| --- |

1. Cliente (persona) - Servidor

Referencia de Imagen: [Imagen 3.2]

| Imagen 3.2. Modelo de red WLAN del servidor en función de los clientes usando dispositivos personales. |
| --- |

## Datos de Conexión para Interfaz de red ‘Interna’ del Raspberry Pi

* **SSID**: airasphealthy
* **WPA-PSK**: raspados1108
* **TIPO:** WPA2

# Detalles del Servidor

### **Propósito del Servidor**

Partiendo del concepto de Iot, este servidor es una Interfaz de Programación de Aplicaciones (API) que permite la comunicación entre los ESP32 y la Raspberry Pi. El propósito de este servicio es que los Módulos de Sensores Fijos (ESP32) envíen los datos correspondientes al servidor que se encargará de almacenar y procesar los datos para devolver la información solicitada por el lado del cliente (persona) visualizandolo finalmente desde una página web.

### **Configuración y Conexión**

* **Dirección del Hos**t: 192.168.10.1
* **Puerto**: 4040
* **Protocolo**: HTTP
* **URL Base**: http://192.168.10.1:4040/

### **Puntos de Conexión (endpoints)**

| **Endpoints** | **Método HTTP** | **Invoca la función** | **Parámetros de la Función** |
| --- | --- | --- | --- |
| **/mediciones** | **GET** | [**\_\_GET\_\_LISTA\_DE\_MEDICIONES\_\_()**](#_e054yjch0k8p) | 1. **request** 2. **response** |
| **/mediciones/prom** | **GET** | [**\_\_GET\_\_MEDICIONES\_PROMEDIADO\_\_()**](#_za5g0zhc929n) | 1. **request** 2. **response** 3. **date** (*string*)    1. “xdia”    2. “hoy”    3. “xhora” 4. **value** (*date* | *time*) |
| **/mediciones/prom/hoy** | **GET** |
| **/medicion** | **POST** | [**\_\_POST\_\_REGISTRAR\_MEDICION()**](#_fiamohrtjrzu) | 1. **request** 2. **response** 3. **json** (*object*)    1. co2    2. temp    3. hum |
| **/dashboard** | **GET** | [**\_\_GET\_\_DASHBOARD\_()**](#_jtr5gqmofdab) | 1. **request** 2. **response** |
| **/descargar** | **GET** | [**\_\_GET\_\_DESCARGAR\_()**](#_lidqr59xwlrq) | 1. **request** 2. **response** |

## 

### **Métodos de Autenticación**

El mecanismo de autenticación se basa en un método básico que funciona mediante validación de credenciales (Imagen 4.1).

En este caso, se cuenta con un objeto (JSON) simple que almacena la credencial correspondiente para cada punto de conexión. El objeto cuenta con la siguiente estructura:

| Imagen 4.1. Variable Global que permite almacenar los hash (claves encriptadas) que requiere el cliente (sensor) para enviar datos al servidor y visualizar la información de los promedios si se requiere. |
| --- |

De esta manera, nos aseguramos que tras cada cliente (sea sensor o una persona) que intente lanzar una solicitud al servidor, disponga del hash o credencial correspondiente que le permita identificarse y consumir del recurso que sea necesario (Imagen 4.2).

| Imagen 4.2. Proceso (algoritmo) del servidor que permite validar las claves encriptadas enviadas por el cliente al momento de solicitar una petición al mismo. |
| --- |

## 

# Diccionario de Datos - SQLite

| **Nombre de la Tabla** | mediciones |
| --- | --- |
| **Descripción de la Tabla** | Esta tabla permite almacenar 4 datos esenciales + índice primario (primary key) de la tabla.  Cuyos datos son los siguientes:   1. Co2 2. Temperatura 3. Humedad 4. FechaHora |
| **Columnas de la Tabla** | * Id * Co2 * Temperatura * Humedad * FechaHora |
| **Relaciones** |  |
| **Índices** | * Id (Primary Key) |
| **Extras** | * Columna **Id** → Auto Incrementable |
| **Procedimientos Almacenados** |  |
| **Triggers** |  |

# Componentes del Sistema

| **Raspberry PI** | **ESP32** |
| --- | --- |
| * **Responsabilidad**: Servidor central. * **Sistema Operativo**: Raspbian OS * **Tecnologías Implementadas**: Node.js y SQLite. * **Conexiones**: Recibe datos de los ESP32 por medio de HTTP. * **Interfaz de Red**: * Nombre → “Interna” * SSID → airasphealthy * WPA-PSK → raspados1108 * Dirección IPv4 → 192.168.10.1 * Rango de IP’s → 192.168.10.# * Máscara de Red → 255.255.255.0 (192.168.10.0 - 192.168.10.255) | * **Responsabilidad**: Recoger y enviar datos de monitoreo. * **Sensores**: CO2 (MQ-135), Temperatura y Humedad (DHT-11). * **Protocolo de Comunicación**: HTTP. * **Conexiones**: Se comunica con la Raspberry Pi mediante WLAN. |
| **Servidor (Node.js)** | |
| * Responsabilidad: Procesar y almacenar datos recibidos. * Tecnologías: Express.js, SQLite3 driver. * API Endpoints: * */mediciones* * */mediciones/prom* * */mediciones/prom/hoy* * */medicion* * */dashboard* [Consultar más detalles.](#_gbpxnv2oe3vv) | |

### 

### 

| **Base de Datos** |
| --- |
| * **Nombre o tipo:** SQLite * **Responsabilidad**: Almacenamiento persistente de los datos recolectados por los sensores. * **Tablas**:  1. datos\_sensores  * **Relaciones**: Ninguna. * **Índices**:  | **Tipo** | **Campo** | **Tabla** | | --- | --- | --- | | Primary Key | Id | datos\_sensores | |

## 

# Conexión al Servidor por medio de SSH

Para realizar una conexión al servidor usando ssh debemos realizar los siguientes pasos:

1. Abrir la ventana de línea de comandos de su sistema operativo.
2. Escribir lo siguiente: \_ ssh <nombreusuario>@direccion\_ip \_
3. Reemplazar nombreusuario por “**cynan**” (sin comillas) y en direccion\_ip por “**192.168.10.1**” (sin comillas).

| Nota: asegurarse de estar conectado a la red airasphealthy antes de proceder con la conexión, de lo contrario no se podrá establecer comunicación. |
| --- |

1. Al presionar enter, nos pedirá la contraseña de acceso al servidor. Para este caso ingresamos la siguiente: “**pi**” (sin comillas).

Con estos pasos, habremos podido ingresar al servidor.

# Requerimientos H&W

* **Requerimientos de Software**
  + Sistema Operativo: GNU/Linux
  + Distribución: Debian o Raspberry Pi OS (Raspbian)
  + Paquetes: *hostapd*, *dnsmasq*, *iptables*.
  + NodeJS (>= v18.x)
* **Requerimientos Mínimos de Hardware**
  + CPU: Quad-Core, Single-Thread a 1.2GHz
  + RAM: 1GB LPDDR2 a 900MHz
  + Conectividad: Ethernet, WiFi 802.11 b/g/n (2.4 GHz)
  + Almacenamiento: MicroSD 32GB (Clase 10)

# Funciones Esenciales del Servidor

**server.js**

Esté módulo consiste de tres funciones principales, que nos permiten manejar eventos y respuestas del servidor.

## Función: serverEnd()

* **Objetivo**: Esta función maneja los eventos cuando un cliente se desconecta del servidor.
* **Parámetros**:
  + **req**: Objeto de solicitud HTTP entrante.
  + **res**: Objeto de respuesta HTTP saliente.
* **Proceso**:
  + Genera y registra una marca de tiempo que indica cuándo un cliente se ha desconectado (Imagen 7.1).
* **Código Relevante:**

| Imagen 7.1. Proceso de la función serverEnd() |
| --- |

## Función: serverListen()

* **Objetivo**: Esta función notifica que el servidor está escuchando en un puerto específico.
* **Parámetros**:
  + **p**: es un número (entero) que especifica en donde está escuchando el servidor.
* **Proceso**:
  + Imprime un mensaje en la consola indicando el puerto en el que el servidor está escuchando y a su vez sirve para informar que el servidor se encuentra activo y funcionando correctamente (Imagen 7.2).
* **Código Relevante:**

| Imagen 7.2. Proceso adicional de la función que imprime mensaje cuando el servidor se encuentra en línea. |
| --- |

## Función: exception()

* **Objetivo**: Esta función maneja las excepciones, generando una respuesta JSON con un estado 404 cuando no se puede generar ninguna otra respuesta.
* **Parámetros**:
  + **req**: Objeto de solicitud HTTP entrante.
  + **res**: Objeto de respuesta HTTP saliente.
* **Proceso**:
  + Crea un objeto con un estado de 404 y un mensaje de error.
  + Escribe el código de estado 404 en la respuesta HTTP.
  + Envía el objeto de error como una cadena JSON en la respuesta HTTP.

Referencia de Imagen: [Imagen 7.3]

* **Código Relevante:**

| Imagen 7.3. Proceso de la función exception() que permite emitir una respuesta visual al cliente. |
| --- |

## 

# Funciones del Código Fuente

**socket.js**

## \_\_POST\_\_REGISTRAR\_MEDICION\_\_()

**Método HTTP: POST**

| **Descripción** | **Parámetros** |
| --- | --- |
| La función es un controlador del servidor Node.js destinado a procesar y registrar mediciones provenientes de solicitudes HTTP POST. La función valida los datos recibidos, los registra en una base de datos SQLite, y responde al cliente con detalles sobre el resultado del proceso. | * **req** → Objeto que contiene información sobre la solicitud HTTP entrante, tales como los parámetros de la URL y los encabezados. * **res** → Objeto utilizado para enviar la respuesta HTTP al cliente. * **json** → Objeto que contiene el cuerpo de la solicitud HTTP en formato JSON, el cual incluye los datos de medición a registrar en la base de datos. |

### **Proceso**

* **Validación de Datos** → Verifica la presencia y validez del dato *co2*, respondiendo con un error 400 Bad Request si falta o no es un número.
* **Extracción y Conversión de Datos** → Extrae y convierte los datos *co2*, *temp*, y *hum* de json a enteros. Asigna 0 a *temp* y *hum* si no son números válidos.
* **Obtención de Dispositivos** → Obtiene una lista de dispositivos ESP32 conectados a la red interna del Raspberry Pi.
* **Identificación de Dirección IP** → Identifica y normaliza la dirección IP del dispositivo que realiza la solicitud.
* **Identificación de Dispositivo** → Busca el dispositivo en la lista de dispositivos por su dirección IP y valida si es un ESP32 reconocido.
* **Registro en la Base de Datos** → Si el dispositivo es un ESP32 válido, registra los datos recibidos en una base de datos SQLite, maneja los errores de inserción y cierra la conexión con la base de datos.
  + **Inicialización de la Base de Datos**: Se inicia la conexión con la base de datos de ‘SQLite’ usando el valor de la variable ‘**dbroute**’ (Imagen 8.1).

| Imagen 8.1. Inicialización de la base de datos para preparar la lectura o inserción de datos. |
| --- |

* + **Creación de Fecha y Hora**: Se obtiene la fecha y hora actuales en formato ISO 8601 (Imagen 8.2).

| Imagen 8.2. Método de creación de las fechas con el formato ISO (YYYY-MM-DD) |
| --- |

* + **Inserción de Datos**: Se ejecuta una inserción en la tabla ***mediciones*** de la base de datos ***airhealthy.db*** (Imagen 8.3, Imagen 8.4).

| Imagen 8.3. Preparación de la sintaxis SQL para la inserción de valores en una tabla.    Imagen 8.4. Se proporcionan los datos que se almacenarán en la base de datos. |
| --- |

* + **Cierre de la Base de Datos**

| Imagen 8.5. Función que permite cerrar la base de datos. |
| --- |

* **Respuesta al Cliente** (Imagen 8.5)
  + **Éxito**
    - **Status Code** → 200 OK
    - **Cuerpo**:

| Imagen 8.5. Datos que se emiten al cliente desde el servidor. |
| --- |

* + **Dispositivo No Reconocido** (Imagen 8.6)
    - **Status Code** → 200 OK
    - **Cuerpo:**

| **Imagen 8.6. Datos que se emiten al cliente desde el servidor cuando un dispositivo no es un ESP32.** |
| --- |

* + **Error de Validación y Otros Errores** (Imagen 8.7)
    - **Status Code** → 400 Bad Request
    - **Cuerpo**:

| Imagen 8.7. Datos que se emiten al cliente cuando ocurre un error de validación u otros errores afines. |
| --- |

### **Errores y Excepciones**

* Los errores durante la ejecución, incluidos los datos inválidos y los problemas de base de datos, se capturan y se manejan mediante un bloque try/catch, y se envían al cliente en una respuesta HTTP con estado 400 Bad Request y un mensaje de error detallado.

## \_\_GET\_\_MEDICIONES\_PROMEDIADO\_\_()

**Método HTTP: GET**

| **Descripción** | **Parámetros** |
| --- | --- |
| La función está diseñada para manejar solicitudes HTTP GET en un servidor Node.js, y tiene como objetivo recuperar y devolver promedios de mediciones (CO2, temperatura, y humedad) de una base de datos SQLite, pudiendo filtrar dichos promedios por día, hora, o para el día actual. | * **req** → Objeto que contiene información sobre la solicitud HTTP entrante, tales como los parámetros de la URL y los encabezados. * **res** → Objeto utilizado para enviar la respuesta HTTP al cliente. * **date** (opcional) → Cadena de texto que determina el tipo de filtrado por fecha:   + ‘xdia’   + ‘hoy’   + ‘xhora’ * **value** (Opcional) → Objeto que contiene valores específicos que funcionan en conjunto con **date** para llevar a cabo el proceso de filtrado (puede incluir fecha o hora). |

### **Proceso**

* **Inicialización del Objeto de Respuesta al Cliente** (Imagen 8.8)

| Imagen 8.8. Variable con nombre “info” que funge como objeto con formato JSON que es emitido al cliente cuando el proceso de obtener el promedio de mediciones es exitoso. |
| --- |

* **Consulta a la Base de Datos**
  + **Conexión a la Base de Datos** (Imagen 8.1)
  + **Construcción de la Consulta SQL**
    - Se comienza a elaborar la consulta SQL para recuperar el promedio de los datos de las mediciones.
    - El filtrado de los resultados se realiza de acuerdo con los parámetros **date** y **value** recibidos.
    - Si se proporcionan inválidos o no se cumplen con las condiciones de la función para el filtrado, se retorna como respuesta al cliente **‘400’** y termina la ejecución.
  + **Cierre de la Base de Datos** (Imagen 8.5)
* **Ejecución de la Consulta y Respuesta**
  + La consulta construida se ejecuta y, en caso de éxito, los resultados se asignan a **info.data** y se envía el objeto **info** como respuesta con un status de **‘200’**.
  + En caso de error durante la consulta, se registra el error y se envía el objeto **info** con un status de **’400’**.
* **Respuestas al Cliente** (Imagen 8.9)

1. Éxito:

* **Status Code** → 200 OK.
* **Cuerpo**:

| Imagen 8.9. Objeto con formato JSON emitido al cliente cuando se ha procesado exitosamente el promedio de los datos y a su vez se ha podido validar la clave encriptada. |
| --- |

1. Error de Base de Datos o Parámetros Inválidos (Imagen 8.10):

* **Status Code** → 400 Bad Request.
* **Cuerpo**:

| Imagen 8.10. Objeto con formato JSON que es emitido al cliente cuando ha ocurrido un error al procesar la solicitud, ya sea por clave encriptada errónea o por consulta a la base de datos. |
| --- |

### **Errores y Excepciones**

* En caso de error durante la consulta, se registra el error, se modifica el objeto info con un status de 400, y se envía como respuesta (Imagen 8.10).
* Si la consulta es exitosa, los datos recuperados se asignan a info.data y se envía el objeto info como respuesta con un status de 200 (Imagen 8.9).

## \_\_GET\_\_LISTA\_MEDICIONES\_\_()

**Método HTTP: GET**

| **Descripción** | **Parámetros** |
| --- | --- |
| La función está diseñada para manejar solicitudes HTTP GET en un entorno de servidor Node.js, su principal tarea es recuperar y devolver una lista de mediciones almacenadas en una base de datos SQLite. La función responde con un objeto JSON que contiene el estado del proceso, un mensaje descriptivo, y los datos recuperados de la base de datos en caso de éxito. | * **req** → Objeto que contiene información sobre la solicitud HTTP entrante, tales como los parámetros de la URL y los encabezados. * **res** → Objeto utilizado para enviar la respuesta HTTP al cliente. |

### **Proceso**

* **Inicialización del Objeto de Respuesta al Cliente** (Imagen 8.8)
* **Consulta a la Base de Datos**
  + **Conexión a la Base de Datos** (Imagen 8.1)
  + **Obtener registro de tabla ‘mediciones’** (Imagen 8.11)

En esta parte se realiza una consulta SQL a la base de datos (airhealthy) para recuperar todas las filas de la tabla ‘mediciones’ y así posteriormente entregarlas como respuesta al cliente. Adicionalmente se extrae la fecha y hora (por separado) del registro correspondiente.

| Imagen 8.11. Proceso de consulta a la tabla mediciones de la base de datos.junto con campos adicionales de fecha y la hora. |
| --- |

* **Cierre de la Base de Datos** (Imagen 8.5)
* **Respuesta al Cliente**

1. **Éxito**:

* **Status Code** → 200 OK.
* **Cuerpo**:

Referencia de Imagen: [Imagen 8.9]

1. **Error de Base de Datos**:

* **Status Code** → 400 Bad Request.
* **Cuerpo**:

Referencia de Imagen: [Imagen 8.10]

### **Errores y Excepciones**

* En caso de error durante la consulta, se registra el error, se modifica el objeto info con un status de 400, y se envía como respuesta.
* Si la consulta es exitosa, los datos recuperados se asignan a info.data y se envía el objeto info como respuesta con un status de 200.

## \_\_GET\_\_DASHBOARD\_\_()

**Método HTTP: GET**

| **Descripción** | **Parámetros** |
| --- | --- |
| La función está diseñada para el cliente y maneja solicitudes HTTP GET en un entorno de servidor Node.js, su principal tarea es mostrar una página html, el cual procesa y describe todos los datos recuperados de los sensores y promedios de los datos bajo ciertas condiciones (todos, ultima hora, hoy y de las últimas 24 horas). | * **req** → Objeto que contiene información sobre la solicitud HTTP entrante, tales como los parámetros de la URL y los encabezados. * **res** → Objeto utilizado para enviar la respuesta HTTP al cliente. |

### **Proceso**

* Se construye un html usando estilos de bootstrap(Imagen 8.12)

| Imagen 8.12. Preparación de sintaxis HTML que permite la importación de una biblioteca de estilos llamada Bootstrap. Se guarda en la variable “bootstrapMinCss”. |
| --- |

* Se llaman a las funciones provenientes del archivo ***constantes.js*** que se encargan de renderizar las tablas y los promedios de los datos correspondientes con cada condición: el promedio de todas las existentes, condición de última hora, condición de hoy partiendo desde las 00:00 horas y la condición de las últimas 24 horas relativas (Imagen 8.13).

| Imagen 8.13. Preparación de las variables que contienen el contenido en formato HTML. Este contenido es proporcionado por las funciones. |
| --- |

* **Respuesta al Cliente** (Imagen 8.14)

1. **En caso de respuesta “Éxito”**:

* **Se muestra el siguiente contenido en pantalla:**

| Imagen 8.14. Respuesta emitida al cliente, se visualiza una página html con datos de las mediciones. |
| --- |

1. **Error**:

En caso de algún error se informa en la consola del sistema y el cliente se desconecta de la sesión.

### **Notas Adicionales**

* **Estilos Adicionales:**
  + Para pantallas con un ancho mínimo de 768px, el ancho de la sección "Medidas Promedio por Día" es del 60%.
  + Para pantallas con un ancho máximo de 768px, el ancho de la misma sección es del 100%.

### **Errores y Excepciones**

* Si ocurre algún error durante la ejecución de la función, se captura y se registra en la consola con un mensaje descriptivo.

## \_\_GET\_\_DESCARGAR\_\_()

**Método HTTP: GET**

| **Descripción** | **Parámetros** |
| --- | --- |
| La función está diseñada para el cliente y permite que el servidor realice una consulta sql a la tabla de *mediciones*, en los cuales todos los registros son recorridos usando la función map() y join() con el propósito de escribir la información cuyos datos están separados por comas y cada registro por saltos de líneas en un archivo de texto. Finalmente se emite al cliente el archivo generando su descarga. | * **req** → Objeto que contiene información sobre la solicitud HTTP entrante, tales como los parámetros de la URL y los encabezados. * **res** → Objeto utilizado para enviar la respuesta HTTP al cliente. |

### **Proceso**

* Se crea una instancia de la base de datos SQLite3 utilizando la ruta especificada en la variable ***dbroute***.
* Se definen las variables nombreArchivo y extensión para el nombre y la extensión del archivo de texto que se generará.
* Se realiza una consulta a la base de datos para seleccionar todos los registros de la tabla ***mediciones***.
* En caso de que ocurra un error durante la consulta, se responde al cliente con un código de estado 500 y un mensaje de "Error interno del servidor".
* Se mapean las filas obtenidas de la base de datos para formatear los datos en un formato de texto que consiste en una línea por registro, separando cada valor por comas.
* Se escribe el contenido formateado en un archivo de texto con el nombre y extensión especificados, utilizando la función ***fs.writeFile***.
* Si ocurre un error durante la escritura del archivo, se responde al cliente con un código de estado *500* y un mensaje de "Error interno del servidor".
* Si la escritura del archivo es exitosa, se configura la respuesta al cliente con un código de estado *200* y se establece el encabezado ***Content-Disposition*** para indicar que se va a descargar un archivo.
* Se crea un flujo de lectura del archivo recién creado y se transmite como respuesta al cliente utilizando ***fs.createReadStream*** y ***pipe***.
* **Respuesta al cliente** (Imagen 8.16) **:**

Al usar el botón ***Descargar Datos*** que se encuentra en la página html (Imagen 8.15) se da como respuesta el archivo de texto formateado (Imagen 8.16).

| Imagen 8.15.Página html que incorpora el botón “Descargar Datos” para descargar un archivo de texto que contiene todos los datos recopilados por los sensores. | Imagen 8.16. Archivo de texto formateado proporcionado por el servidor. |
| --- | --- |

### **Errores y Excepciones**

* Si ocurre algún error durante la ejecución de la función, se captura y se emite al cliente.

# Funciones Complementarias

**constantes.js**

Las siguientes funciones son dadas ya que sirven como auxiliar para obtener la lista de dispositivos ESP32 conectados a la red ‘interna’ del Raspberry Pi. De tal manera que nos sirven para llevar a cabo el registro de las mediciones en la tabla de ‘mediciones’ de la base de datos (airhealthy).

## Función: obtenerTablaArp(dev)

Esta función ejecuta el comando arp de linux pero con ayuda de la biblioteca de Node.js llamada **arp-a**, el cual retorna la lista de dispositivos en la tabla ARP del sistema para obtener las direcciones ipv4 correspondiente a cada uno de ellos. Luego, procesa esta salida para crear un array de objetos que contienen detalles como la dirección ipv4, dirección mac y la interfaz de red (del servidor) que en este caso pertenece a la red con nombre ***interna*** (Imagen 9.1).

La función posee la siguiente forma:

| Imagen 9.1. Proceso de la función obtenerTablaArp(). |
| --- |

#### **Argumentos**

* **dev** → parámetro de la función que permite decirle a la función **table()** de la biblioteca **arp-a** que obtenga la lista de dispositivos pero de cierta red.. En este caso, la red con nombre ***interna*** del raspberry.

#### **Salida**

Devuelve un array de objetos que contienen detalles como la dirección ipv4, dirección mac y la interfaz de red (*interna*).

#### **Errores**

Si ocurre un error al ejecutar el comando, se devolverá el resultado del error a la consola del sistema..

## Función: obtenerNombreHost(ip)

Esta función permite obtener el nombre del host, es decir, del dispositivo que corresponde a la ip proporcionada a la función. Por ejemplo, si se pasa un valor como “192.168.10.178” la función devolverá “esp32-XXXXX”, cuyo nombre es el hostname del dispositivo que posee esa dirección ipv4 (Imagen 9.2).

| Imagen 9.2. Proceso que realiza la función obtenerNombreHost() para recuperar el hostname dada una dirección ipv4. |
| --- |

**Ejemplo de Uso** (Imagen 9.3)**:**

| Imagen 9.3. Manera en que se hace uso de la función obtenerNombreHost(). |
| --- |

#### **Argumentos**

* **ip →** la dirección ipv4 de un dispositivo.

#### **Salida**

Devuelve el nombre del host del dispositivo correspondiente a la dirección ipv4 proporcionada.

#### **Errores**

Si ocurre un error al intentar recuperar el nombre del host se devolverá el valor *null*.

## 

## Función: lista\_arp(dev)

Esta función tiene la finalidad de obtener una lista de direcciones IP y sus correspondientes nombres de host (si están disponibles) en una red local específica (Imagen 9.4).

| Imagen 9.4. Algoritmo de la función lista\_arp(). |
| --- |

**Ejemplo de Uso** (Imagen 9.5)**:**

| Imagen 9.5. Manera en que se hace uso de la función lista\_arp(). |
| --- |

#### **Argumentos**

* **dev** → parámetro de la función que permite decirle a la función **table()** de la biblioteca **arp-a** que obtenga la lista de dispositivos pero de cierta red.. En este caso, la red con nombre ***interna*** del raspberry.

#### **Salida**

Devuelve un arreglo de objetos (con formato json) que contiene detalles como: dirección ipv4, dirección mac, hostname y el nombre de la interfaz de la red de donde provienen estas listas de dispositivos.

#### **Errores**

Si ocurre un error se imprimirá como mensaje en la consola del sistema.

## Función: lista\_medidores()

Esta función hace uso finalmente de **lista\_arp()** la cual una vez recuperada toda la información junto con sus nombres de host, filtra los datos con la condición de que el nombre del host corresponda al de un ESP32.

| Imagen 9.6. Proceso que realiza la función lista\_medidores() para obtener la lista de microcontroladores ESP32 conectados al servidor raspberry. |
| --- |

**Ejemplo de Uso:**

| Imagen 9.7. Manera en que se hace uso de la función lista\_medidores(). |
| --- |

#### **Argumentos**

Ninguno.

#### **Salida**

Devuelve un array de objetos que corresponde a la lista de dispositivos ESP32 conectados al servidor (en ese momento).

#### **Errores**

Si no existen dispositivos ESP32 conectados al servidor se devolverá el arreglo vacío.

## 

## Exportación de las Funciones

Finalmente, dadas las condiciones sintácticas de trabajar en NodeJS, para cada archivo independiente se debe realizar la exportación de las funciones para que de esta manera se pueda hacer uso de ellas en otros archivos del proyecto.

| Imagen 9.8. Manera en que se exportan las funciones en un archivo en un entorno NodeJS. |
| --- |

## Se exportan todas las funciones que se harán uso en el archivo **socket.js** del proyecto (Imagen 9.8).