Universidad San Carlos de Guatemala
Facultad de Ingeniería -FIUSAC-
Escuela de Ciencias y Sistemas
Curso Arquitectura de Computadoras
y Ensambladores 2 - Sección: “B”

Fase 2 - PROYECTO ÚNICO -

Sistema Inteligente de monitoreo ambiental para Centros de Datos mediante IoT

DOCUMENTACIÓN

Adler Alejandro Pérez Asensio	202200329
Andrés Alejandro Quezada Cabrera	202200174
Angel Samuel González Velásquez	202200263
Irving Fernando Alvarado Asensio	202200349

Catedrático: Ing. Jurgen Ramírez
Auxiliar: Axel Calderón / Danny Cuxum

Guatemala, 22 de febrero de 2025

Introducción

El avance de la tecnología y la creciente dependencia de los centros de datos hacen imprescindible el monitoreo preciso de las condiciones ambientales dentro de los cuartos de servidores. Factores como la temperatura, la humedad, la calidad del aire y la corriente eléctrica pueden afectar directamente el rendimiento y la vida útil del hardware, lo que resalta la necesidad de un sistema de monitoreo eficiente.

Este proyecto propone el desarrollo de un Sistema Inteligente de Monitoreo Ambiental basado en tecnología IoT, capaz de medir y registrar variables ambientales en tiempo real. Utilizando sensores especializados y un microcontrolador Arduino, el sistema recopilará información clave y la visualizará en una pantalla LCD y un dashboard en Processing, permitiendo la toma de decisiones basada en datos.

A través del enfoque del Smart Connected Design Framework, se abordará la implementación del sistema desde sus diferentes capas: percepción, red, procesamiento y aplicación. Cada una de estas capas será documentada detalladamente para explicar cómo los sensores capturan los datos, cómo se procesan y almacenan, y cómo se presentan a los usuarios de manera efectiva.

El sistema se compone de una serie de sensores conectados a un microcontrolador Arduino, que envía los datos recolectados a una base de datos en la nube, desde donde se pueden consultar y visualizar mediante una aplicación web. Además, el sistema incluye funcionalidades como el control de acceso mediante RFID, la activación de ventiladores y alarmas en caso de condiciones críticas, y la visualización de alertas a través de pantallas LCD y LEDs.

Este documento describe la estructura y el funcionamiento del sistema, los componentes utilizados, el flujo de información, los beneficios del monitoreo ambiental en centros de datos y el impacto de la solución propuesta. Además, se incluyen bocetos, diagramas y explicaciones técnicas que permitirán comprender la implementación y facilitar futuras mejoras.

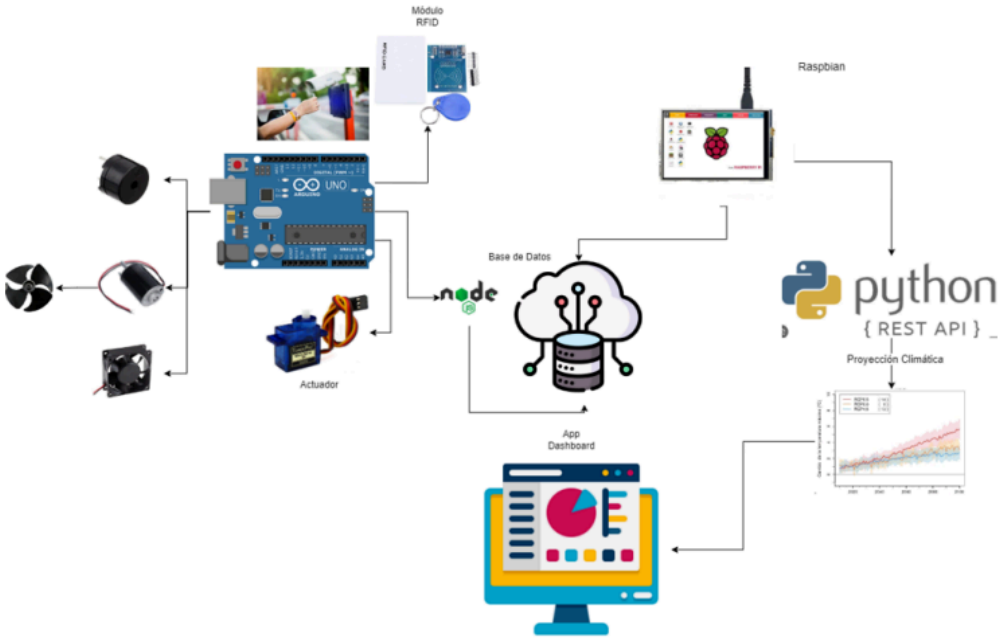
Descripción del Sistema

Fase 1: Consiste en la recopilación de datos mediante sensores, y la activación de alertas por medio de sistemas como la LCD y LEDs. En esta fase se cubren las capas más básicas del sistema, como el hardware de los sensores y el microcontrolador Arduino, el que hace que todo el hardware funcione de forma correcta.

Fase 2: Expande el sistema con la integración de nuevas tecnologías como APIs, una que envía los datos recopilados a una base de datos en la nube, y el análisis de datos históricos, proyecciones climáticas y visualización en una aplicación web interactiva. Esta fase también introduce la capacidad de comunicación entre el microcontrolador Raspberry Pi 4 y la base de datos a través de APIs, y un sistema de análisis de datos a través de Python y herramientas en la nube.

- **Componentes utilizados**

  - ☐ Arduino
  - ☐ DHT11
  - ☐ HC-SR04
  - ☐ Fotorresistencia
  - ☐ ACS712 (5A)
  - ☐ Raspberry Pi 4
  - ☐ Buzzer/Zumbador
  - ☐ Motor Servo
  - ☐ Módulo RFID
  - ☐ Base de Datos (Mongo)
  - ☐ Pantalla LCD
  - ☐ LEDs
  - ☐ Jumpers
  - ☐ Protoboard



Capas del Smart Connected Design Framework

1. Capa de Infraestructura (Infrastructure):

Esta capa representa la infraestructura física que soporta todo el sistema IoT. Está compuesta por los siguientes componentes esenciales:

- **Microcontrolador Arduino:** Es el corazón del sistema que gestiona los sensores y actuadores en el cuarto de servidores, recolectando los datos y ejecutando las acciones correspondientes, como el control de la puerta o barrera con el **servo motor** y los **ventiladores** cuando se activan alertas críticas.

- **Raspberry Pi:** Actúa como el servidor central donde se ejecutan las APIs, se procesa la información y se conecta a la base de datos en la nube. La Raspberry Pi también está encargada de ejecutar los scripts necesarios para la visualización de datos históricos en la web.

- **Sensores y Actuadores:** Estos dispositivos recolectan información y responden a las condiciones del ambiente:

    - **Sensor RFID (RC522):** Controla el acceso al cuarto de servidores mediante tarjetas RFID. Este sensor se encarga de verificar la autenticidad de los usuarios.
    - **Sensor de distancia (HC-SR04):** Detecta la proximidad de personas y permite la interacción con la entrada del sistema.
    - **Servo Motor:** Controla la apertura y cierre de la puerta del cuarto de servidores, permitiendo el acceso de los usuarios.
    - **Motor DC:** Activado cuando se detectan alertas de temperatura alta o de mala calidad del aire, simula la activación de ventiladores para regular la temperatura.
    - **LEDs de advertencia (Verde, Amarillo, Rojo):** Indican el estado de los parámetros de los sensores, proporcionando alertas visuales de condiciones anormales.

2. Capa de Sensores (Sensors):

Los sensores desempeñan un papel crucial en la recolección de datos sobre el entorno del cuarto de servidores:

- **Sensor de temperatura y humedad (DHT11):** Mide la temperatura y la humedad relativa. Es fundamental para garantizar que las condiciones sean adecuadas para el funcionamiento de los servidores.

- **Sensor de movimiento y proximidad (HC-SR04):** Detecta la presencia de personas en el cuarto de servidores, facilitando el control de acceso y la seguridad.

- **Sensor de calidad del aire (MQ135):** Mide la concentración de CO_2 y otros gases, ayudando a evaluar la ventilación y la calidad del aire en el cuarto de servidores.

- **Sensor de corriente (ACS712):** Mide el consumo de energía en el sistema, permitiendo detectar problemas de energía, como caídas o sobrecargas.

- **Sensor de luz (Fotorresistencia):** Detecta la cantidad de luz en el entorno y puede activar o desactivar la iluminación del cuarto de servidores según sea necesario.

3. Capa de Conectividad (Connectivity):

La conectividad asegura que los datos recolectados por los sensores sean enviados y procesados de forma eficiente. Esta capa está formada por:

- **Comunicación Serial:** Inicialmente, el Arduino transmite los datos mediante comunicación serial a la Raspberry Pi utilizando un cable USB y un baud rate de 9600 bps. Los datos de sensores como la temperatura, humedad, calidad del aire y corriente se envían en un formato estructurado, como: `25.3,60,300,400,2.1` (donde cada número representa un valor de sensor específico).

- **API para subida de datos (API 1):** Una API (en Python, por ejemplo) en la Raspberry Pi se encarga de recibir los datos desde Arduino, estructurarlos y almacenarlos en la base de datos en la nube (como MySQL o GCP). Esta API también registra la fecha y hora de las mediciones.

- **API para consulta de datos (API 2):** Esta segunda API, también en Python, se conecta a la base de datos en la nube para recuperar los datos históricos de los sensores. Los datos recuperados son luego utilizados por la aplicación web para mostrar gráficos y permitir a los usuarios consultar las tendencias pasadas.

4. Capa de Análisis (Analytics):

El análisis de los datos es esencial para asegurar que el sistema actúe de manera preventiva y eficiente:

- **Detección de Anomalías:** El sistema establece umbrales para cada parámetro (temperatura, humedad, calidad del aire, consumo de energía) y genera alertas cuando estos parámetros superan los valores críticos.

- **Análisis Predictivo:** Mediante el uso de algoritmos de predicción (implementados en Python), el sistema puede anticipar eventos como un aumento de temperatura o consumo de energía, lo que permite activar medidas preventivas, como encender ventiladores o alertar al personal de mantenimiento.

- **Almacenamiento y Análisis Histórico:** Los datos recolectados y las alertas generadas se almacenan en la base de datos en la nube, lo que permite hacer un seguimiento de tendencias y optimizar la eficiencia del sistema a largo plazo.

5. Capa de Aplicación Inteligente (Smart App):

Esta capa proporciona la interfaz de usuario, permitiendo a los usuarios interactuar con el sistema y visualizar los datos. Aquí se incluyen:

- **Aplicación Web (Dashboard):** Desarrollada utilizando frameworks como **React** o **Angular**, la aplicación web permite a los usuarios visualizar en tiempo real los datos de los sensores, como la temperatura, la humedad, la calidad del aire y el consumo de corriente. También ofrece gráficos históricos y alertas visuales cuando los valores superan los límites críticos.

- **Pantalla LCD:** A nivel local, la **pantalla LCD** en el sistema de hardware muestra mensajes importantes y alertas en tiempo real sobre las condiciones del ambiente. Los LEDs de advertencia también se activan, dependiendo de la condición de los parámetros monitoreados.

- **Notificaciones de Emergencia:** Si el sistema detecta una condición crítica, se activan alertas visuales (LEDs de colores) y sonoras (buzzer/zumbador). Los ventiladores se encienden si la temperatura es demasiado alta y la calidad del aire o la humedad exceden los límites establecidos.

Implementación del Código

Explicación del código en Arduino

La arquitectura del sistema se basa en una comunicación fluida entre los dispositivos IoT (sensores, actuadores) y las plataformas de almacenamiento y análisis (API, base de datos en la nube). Los sensores conectados al microcontrolador Arduino envían datos a una plataforma intermedia (API) que los almacena en una base de datos en la nube. Estos datos luego son recuperados por una aplicación web que visualiza los resultados en un dashboard interactivo, permitiendo a los usuarios monitorizar las condiciones del cuarto de servidores.

Las librerías a instalar son:

- DHT sensor library: Para usar sensores DHT11 o DHT22.
- LiquidCrystal I2C: Para controlar la pantalla LCD con interfaz I2C.
- Librería MFRC522 y SPI.h: Para controlar el módulo RFID RC522
- Servo.h: Para controlar el servomotor.

Configuración del `setup()`:

El código inicia la comunicación serie, la pantalla LCD y el sensor `DHT`, mostrando "Bienvenidos" en la LCD. Configura los pines de sensores y actuadores como entrada o salida. Se inicializa el módulo RFID y el servo, posicionándolo en 0°.

```
void setup() {
  Serial.begin(9600);
  dht.begin();
  lcd.init();
  delay(100);
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Bienvenidos");

  // Configuración de pines
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  pinMode(TRIG_PIN_ENTRADA, OUTPUT);
  pinMode(ECHO_PIN_ENTRADA, INPUT);
  pinMode(MOTOR_DC, OUTPUT);
  pinMode(LED_PIN, OUTPUT);
  pinMode(BUZZER_PIN, OUTPUT);
  pinMode(LED_ROJO, OUTPUT);
  pinMode(LED_AMARILLO, OUTPUT);
  pinMode(LED_AZUL, OUTPUT);
  pinMode(MODO_NORMAL_PIN, INPUT);
  pinMode(MODO_GUARDAR_PIN, INPUT);
  pinMode(MODO_MOSTRAR_PIN, INPUT);
  pinMode(LDR_PIN, INPUT);

  // Inicialización de RFID
  SPI.begin();
  rfid.PCD_Init();

  // Inicialización de servo
  servo.attach(SERVO_PIN);
  servo.write(0);
```

Función `loop()`

Ejecuta continuamente funciones clave: `manejarBotones()`, `verificarRFID()`, `verificarServo()`. Cada 0.5 segundos, envía datos a través del puerto serie. Dependiendo del `modoActual`, muestra valores en tiempo real, guarda datos en EEPROM o recupera valores almacenados.

```
void loop() {
  manejarBotones();
  verificarRFID();
  verificarServo();

  // Enviar datos por Serial cada 0.5s
  if (millis() - ultimaActualizacionSerial >= 500) {
    ultimaActualizacionSerial = millis();
    enviarDatosSerial();
  }

  switch (modoActual) {
    case TIEMPO_REAL:
      if (alertaActiva) {
        if (millis() - tiempoAlerta >= 3000) {  // Prioridad de 3s a alertas
          alertaActiva = false;
        }
      } else if (millis() - ultimaActualizacionLCD >= 500) {
        ultimaActualizacionLCD = millis();
        mostrarValoresTiempoReal();
      }
      break;

    case GUARDAR:
      guardarValoresEEPROM();
      modoActual = TIEMPO_REAL;  // Volver automáticamente al modo normal
      break;

    case MOSTRAR_GUARDADOS:
      mostrarValoresEEPROM();
      break;
  }
}
```

Verificación de RFID

La función `verificarRFID()` chequea si se detecta una tarjeta nueva y, si es así, la compara con la lista de usuarios autorizados mediante `esUsuarioAutorizado()`. Si el usuario está autorizado, se abre el servo durante 5 segundos y muestra "Acceso permitido", de lo contrario, muestra "Acceso denegado".

```
void verificarRFID() {
  if (servoCerrado) {

    if (rfid.PICC_IsNewCardPresent()) {

      if (rfid.PICC_ReadCardSerial()) {
        byte uid[4];

        for (byte i = 0; i < 4; i++) {
          uid[i] = rfid.uid.uidByte[i];
        }

        if (esUsuarioAutorizado(uid)) {
          servoCerrado = false;
          servoTime = millis();
          servo.write(90);
          mostrarAlerta("Acceso permitido");
        } else {
          mostrarAlerta("Acceso denegado");
        }
      }
    }
  }
}
```

Control del servo

`verificarServo()` cierra el servo si han pasado 5 segundos y no hay un objeto a menos de 10 cm del sensor de entrada. Si hay un obstáculo, se muestra una alerta.

```
void verificarServo() {
  //Si el servo está abierto y han pasado 5 segundos, cerrarlo si la lectura del
  if (!servoCerrado && millis() - servoTime >= 5000) {
    float distancia = obtenerDistanciaEntrada();
    if (distancia > 10) {
      servoCerrado = true;
      servo.write(0);
    } else {
      mostrarAlerta("Despeje la entrada");
    }
  }
}
```

Manejo de botones

`manejarBotones()` detecta si se presionan los botones de cambio de modo con un tiempo de rebote de 50 ms.

```
void manejarBotones() {
  static unsigned long lastDebounceTime = 0;
  static const unsigned long debounceDelay = 50;

  if (millis() - lastDebounceTime > debounceDelay) {
    if (digitalRead(MODO_NORMAL_PIN) == HIGH) {
      modoActual = TIEMPO_REAL;
      alertaActiva = false;
      lastDebounceTime = millis();
    }
    if (digitalRead(MODO_GUARDAR_PIN) == HIGH) {
      modoActual = GUARDAR;
      lastDebounceTime = millis();
    }
    if (digitalRead(MODO_MOSTRAR_PIN) == HIGH) {
      modoActual = MOSTRAR_GUARDADOS;
      lastDebounceTime = millis();
    }
  }
}
```

Envío de datos por serial

`enviarDatosSerial()` obtiene mediciones de sensores y envía valores de temperatura, humedad, distancia, calidad del aire, corriente, luz y estado de la puerta a través del puerto serie.

```
void enviarDatosSerial() {
  float distancia = obtenerDistancia();
  int calidadAire = analogRead(MQ135_PIN);
  calidadAire = calcularCo2(calidadAire);
  int luz = !digitalRead(LDR_PIN);
  float humedad = dht.readHumidity();
  float temperatura = dht.readTemperature();
  float corriente = obtener_corriente();
  float dista = obtenerDistanciaEntrada();
  int puerta = 0;

  //Devolvemos 1 si la puerta esta abierta y 0 si esta cerrada
  if (servoCerrado) {
    puerta = 0;
  } else {
    puerta = 1;
  }


  Serial.println(String(temperatura) + "," + String(humedad) + "," + String(distancia) + "," + String(luz) + ",
}
```

Obtención de valores de sensores

Funciones como `obtener_corriente()`, `calcularCo2()`, `obtenerDistancia()` y `obtenerDistanciaEntrada()` leen los sensores y retornan sus valores.

```
float obtener_corriente() {
  int lectura = analogRead(CORRIENTE_PIN);
  float voltajeSensor = lectura * (5.0 / 1023.0);
  return (voltajeSensor - 2.5) / sensibilidad;
}

float calcularCo2(int valorADC) {
  return valorADC;
}
```

Modo de visualización en LCD

`mostrarValoresTiempoReal()` obtiene valores de sensores y los muestra en la LCD si no hay alertas activas. Si hay errores en el sensor DHT11, se muestra "DHT11 Error".

```
void mostrarValoresTiempoReal() {
  float distancia = obtenerDistancia();
  int calidadAire = analogRead(MQ135_PIN);
  calidadAire = calcularCo2(calidadAire);
  int luz = !digitalRead(LDR_PIN);
  float humedad = dht.readHumidity();
  float temperatura = dht.readTemperature();
  float corriente = obtener_corriente();

  if (isnan(humedad) || isnan(temperatura)) {
    mostrarAlerta("DHT11 Error");
    return;
  }

  manejarAlertas(temperatura, humedad, corriente);

  if (!alertaActiva) {
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("T:" + String(temperatura, 0) + " H:" + String(humedad, 0) + " D:" + String(obtenerDistancia(), 1));
    lcd.setCursor(0, 1);
    lcd.print("C2:" + String(calidadAire) + " C:" + String(corriente) + " L:" + String(luz));
  }
}
```

Manejo de alertas

`manejarAlertas()` enciende el buzzer y LEDs si hay valores críticos en corriente, humedad, temperatura, calidad del aire o iluminación, activando el motor DC si es necesario. Se da prioridad a mostrar alertas en la LCD.

```
void manejarAlertas(float temperatura, float humedad, float corriente) {

  // Nivel de co2
  int calidadAire = analogRead(MQ135_PIN);
  calidadAire = calcularCo2(calidadAire);

  // Nivel de luz
  int luz = digitalRead(LDR_PIN);  // Lee 1 si no hay luz y 0 si hay luz POSIBLE CAMBIO A !digitalRead(LDR_PIN)

  /*
    - Si no se detecta consumo de corriente el buzzar debe de pitar

    - Si humedad, co2 o temperatura son críticos, se enciende el ventilador (motor DC)


    - Si no hay luz (fotoresistencia) se deben de encender los leds (iluminación)
    */

  if (corriente < 0.5 || corriente > 15) {
    mostrarAlerta("Corriente ALERTA");
    digitalWrite(BUZZER_PIN, HIGH);
    digitalWrite(LED_AMARILLO, HIGH);
  } else {
    digitalWrite(BUZZER_PIN, LOW);
    digitalWrite(LED_AMARILLO, LOW);
  }

  if (humedad < 30 || humedad > 60) {
    mostrarAlerta("Humedad ALERTA");
    digitalWrite(MOTOR_DC, HIGH);
    digitalWrite(LED_AZUL, HIGH);
  } else {
    digitalWrite(MOTOR_DC, LOW);
    digitalWrite(LED_AZUL, LOW);
  }

  if (temperatura < 18 || temperatura > 30) {
    mostrarAlerta("Temperatura ALERTA");
    digitalWrite(MOTOR_DC, HIGH);
    digitalWrite(LED_ROJO, HIGH);
  } else {
    digitalWrite(MOTOR_DC, LOW);
    digitalWrite(LED_ROJO, LOW);
  }

  if (calidadAire > 500) {
    mostrarAlerta("CO2 ALERTA");
    digitalWrite(MOTOR_DC, HIGH);
  } else {
    digitalWrite(MOTOR_DC, LOW);
  }
}
```

Explicación de la API para subir datos

Este programa en Processing recibe datos de sensores de una maqueta a través de una conexión serial con un Arduino. Representa visualmente estos datos mediante indicadores y notificaciones, proporcionando una interfaz gráfica para el monitoreo de un data center.

Este código está diseñado para leer datos de sensores conectados a un dispositivo Arduino y almacenarlos en una base de datos en la nube utilizando MongoDB. A continuación te explico cada parte del código y su propósito:

Importación de módulos

```
import serial
import time
from pymongo import MongoClient
from datetime import datetime
```

- **serial (y pyserial)**: Se utiliza para establecer una comunicación con el puerto serial de Arduino.
- **time**: Proporciona funciones para manejar el tiempo (como `sleep` para pausas).
- **pymongo**: Permite interactuar con la base de datos MongoDB desde Python.
- **datetime**: Se usa para obtener la fecha y hora actual, que se añadirá a los registros de los sensores.

Conexión a MongoDB

```
# Conexión con MongoDB en la nube
client = MongoClient("mongodb+srv://grupo16:arqui2@arqui2g16.vc63g.m
db = client["arqui2G16"]
```

```
ongodb.net/?retryWrites=true&w=majority&appName=arqui2G16")
```

- Establece una conexión a la base de datos MongoDB alojada en la nube (a través de MongoDB Atlas).
- `client` es el objeto que maneja la conexión.
- `db` accede a la base de datos llamada "arqui2G16".

Definición de colecciones

```
# Definir las colecciones para cada sensor
coleccion_temperatura = db["temperatura"]
coleccion_humedad = db["humedad"]
coleccion_distancia = db["distancia"]
coleccion_luz = db["luz"]
coleccion_calidadAire = db["calidadAire"]
coleccion_corriente = db["corriente"]
coleccion_distancia_puerta = db["distancia_puerta"]
coleccion_puerta = db["puerta"]
```

- Cada sensor tiene su propia colección en MongoDB. Las colecciones se nombran según el tipo de dato que miden (por ejemplo, `temperatura`, `humedad`, `distancia`, etc.).

Apertura del puerto serial

```
# Abrir el puerto serial donde Arduino está enviando los datos
ser = serial.Serial('COM6', 9600)  # Cambiar 'COM6' por el puerto correcto
```

- Se establece una conexión con el puerto serial en el que Arduino está enviando los datos. El puerto puede variar según el sistema, y la velocidad de transmisión de los datos es de 9600 baudios.

Función para procesar y almacenar datos

```
# Función para procesar la línea recibida
def procesar_datos(linea):
    # Parsear la línea CSV recibida
    datos = linea.strip().split(',')
    if len(datos) == 8:  # Asegurarse de que hay 8 valores en la línea
        temperatura = float(datos[0])
        humedad = float(datos[1])
        distancia = float(datos[2])
        luz = float(datos[3])
        calidadAire = float(datos[4])
        corriente = float(datos[5])
        distancia_puerta = float(datos[6])
        puerta = datos[7]

        # Obtener la fecha y hora actual
        fecha_hora = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

```python
        # Crear un diccionario con los datos y la fecha
        documento_temperatura = {"temperatura": temperatura, "fecha_hora": fecha_hora}
        documento_humedad = {"humedad": humedad, "fecha_hora": fecha_hora}
        documento_distancia = {"distancia": distancia, "fecha_hora": fecha_hora}
        documento_luz = {"luz": luz, "fecha_hora": fecha_hora}
        documento_calidadAire = {"calidadAire": calidadAire, "fecha_hora": fecha_hora}
        documento_corriente = {"corriente": corriente, "fecha_hora": fecha_hora}
        documento_distancia_puerta = {"distancia_puerta": distancia_puerta, "fecha_hora": fecha_hora}
        documento_puerta = {"puerta": puerta, "fecha_hora": fecha_hora}

        # Insertar los datos en las colecciones correspondientes
        coleccion_temperatura.insert_one(documento_temperatura)
        coleccion_humedad.insert_one(documento_humedad)
        coleccion_distancia.insert_one(documento_distancia)
        coleccion_luz.insert_one(documento_luz)
        coleccion_calidadAire.insert_one(documento_calidadAire)
        coleccion_corriente.insert_one(documento_corriente)
        coleccion_distancia_puerta.insert_one(documento_distancia_puerta)
        coleccion_puerta.insert_one(documento_puerta)
```

- **Procesamiento de los datos**: La función `procesar_datos` toma una línea de texto (que representa un conjunto de lecturas de sensores), la divide en campos separados por comas (CSV) y extrae los valores de cada sensor.
- **Conversión de los valores**: Cada valor de sensor (como temperatura, humedad, etc.) se convierte al tipo de dato adecuado (por ejemplo, flotante para números decimales).
- **Fecha y hora**: Se obtiene la fecha y hora actuales y se añade a cada documento que se insertará en la base de datos.
- **Creación de documentos**: Para cada tipo de dato, se crea un diccionario con el valor y la fecha, y luego se inserta en la colección correspondiente de MongoDB.

Bucle principal para leer datos del puerto serial

```python
# Bucle para leer datos del puerto serial
while True:
    if ser.in_waiting > 0:
        # Leer la línea de datos
        linea = ser.readline().decode('utf-8').strip()  # Intentar leer y decodificar como UTF-8
        print(linea)  # Para ver qué datos está leyendo
        procesar_datos(linea)  # Procesar y almacenar en MongoDB

    else:
        print("Esperando nuevos datos...")  # Si no hay nuevos datos, muestra un mensaje

    ser.flushInput()  # Limpiar el buffer de entrada después de cada lectura
```

- **Lectura del puerto serial**: Dentro de un bucle infinito, el programa verifica si hay datos disponibles en el puerto serial. Si hay datos, los lee.

- **Decodificación de la línea**: Se decodifica la línea leída como UTF-8 y se elimina cualquier espacio adicional al principio y final.
- **Procesamiento**: Si hay datos, la función `procesar_datos` se llama para procesarlos y almacenarlos en MongoDB.
- **Espera**: Si no hay datos disponibles, el programa muestra un mensaje "Esperando nuevos datos..." y sigue esperando.
- **Limpieza del buffer**: Después de cada lectura, se limpia el buffer de entrada del puerto serial para evitar datos residuales de lecturas anteriores.

Pausa entre lecturas

```
time.sleep(0.5)  # Esperar 0.5 segundos entre lecturas
```

- El programa espera 0.5 segundos antes de intentar leer datos nuevamente, para evitar lecturas excesivas.

Explicación de la API para subir datos

Este código está implementando una API web utilizando **Flask** para interactuar con los datos de sensores almacenados en una base de datos **MongoDB**. La API ofrece varios endpoints para obtener datos de los sensores, tanto de forma general como con filtros específicos. Aquí te explico cada parte del código en detalle:

Importación de módulos

```
from flask import Flask, jsonify, request
from pymongo import MongoClient
from datetime import datetime
from flask_cors import CORS
```

- **Flask**: Es un micro-framework para crear aplicaciones web en Python. Aquí se utiliza para crear una API RESTful.
- **jsonify**: Convierte los datos en formato JSON para enviarlos como respuesta HTTP.
- **request**: Se usa para obtener los parámetros de la URL o cuerpo de la solicitud (por ejemplo, parámetros de consulta para filtros).
- **MongoClient**: Permite interactuar con MongoDB.
- **datetime**: Se utiliza para trabajar con fechas y horas.

- **CORS**: Se usa para habilitar **Cross-Origin Resource Sharing**, lo que permite que la API sea consumida desde diferentes dominios o puertos, evitando bloqueos por políticas de seguridad del navegador.

Configuración de la aplicación Flask

```
app = Flask(__name__)
CORS(app)
```

- **Flask(name)**: Inicializa la aplicación Flask.
- **CORS(app)**: Permite que la aplicación Flask sea accesible desde otras aplicaciones o dominios diferentes a la API.

Conexión con MongoDB

```
# Configuración de conexión a MongoDB
client = MongoClient("mongodb+srv://grupo16:arqui2@arqui2g16.vc6
db = client["arqui2G16"]
```

```
g.mongodb.net/?retryWrites=true&w=majority&appName=arqui2G16")
```

- **MongoClient**: Establece una conexión con la base de datos MongoDB alojada en la nube usando MongoDB Atlas.
- **db**: Obtiene la base de datos `arqui2G16`.

Definición de las colecciones de sensores

```
# Definir las colecciones
colecciones = {
    "temperatura": db["temperatura"],
    "humedad": db["humedad"],
    "distancia": db["distancia"],
    "luz": db["luz"],
```

```
    "calidadAire": db["calidadAire"],
    "corriente": db["corriente"],
    "distancia_puerta": db["distancia_puerta"],
    "puerta": db["puerta"]
}
```

- Cada tipo de sensor tiene su propia colección en MongoDB (por ejemplo, `temperatura`, `humedad`, `distancia`, etc.). La variable `colecciones` es un diccionario que mapea el nombre del sensor a su correspondiente colección en la base de datos.

Rutas y endpoints de la API

1. Obtener los últimos 30 registros de un sensor específico

```
@app.route('/datos/<sensor>', methods=['GET'])
def obtener_datos_sensor(sensor):
    if sensor in colecciones:
        datos = list(colecciones[sensor].find({}, {"_id": 0}).sort("fecha_hora", -1).limit(30))  # Últimos 30 registros
        return jsonify(datos), 200
    return jsonify({"error": "Sensor no encontrado"}), 404
```

- **Ruta `/datos/<sensor>`**: Devuelve los últimos 30 registros del sensor especificado en la URL (por ejemplo, `/datos/temperatura`).
- Si el sensor existe en el diccionario `colecciones`, se obtiene una lista de los últimos 30 documentos de la colección correspondiente, ordenados por `fecha_hora` en orden descendente.
- Si el sensor no es válido, se devuelve un error 404.

2. Obtener los últimos 30 registros de todos los sensores

```
@app.route('/datos', methods=['GET'])
def obtener_todos_los_datos():
    datos_completos = {}
    for sensor, coleccion in colecciones.items():
        datos_completos[sensor] = list(coleccion.find({}, {"_id": 0}).sort("fecha_hora", -1).limit(30)) # Últimos 30 registros
    return jsonify(datos_completos), 200
```

- **Ruta `/datos`**: Devuelve los últimos 30 registros de todos los sensores.
- La respuesta es un diccionario con cada sensor como clave y una lista de los últimos 30 registros como valor.

3. Obtener datos filtrados por fecha

```python
@app.route('/datos/<sensor>/filtrado', methods=['GET'])
def obtener_datos_filtrados(sensor):
    if sensor not in colecciones:
        return jsonify({"error": "Sensor no encontrado"}), 404

    # Parámetros opcionales: fecha inicio, fecha fin, límite
    fecha_inicio = request.args.get("fecha_inicio")  # Formato esperado: YYYY-MM-DD HH:MM:SS
    fecha_fin = request.args.get("fecha_fin")  # Formato esperado: YYYY-MM-DD HH:MM:SS
    limite = request.args.get("limite", default=500, type=int)  # Máximo de registros a devolver

    filtro = {}

    # Si el usuario proporciona un rango de fechas, filtramos por eso
    if fecha_inicio and fecha_fin:
        try:
            fecha_inicio = datetime.strptime(fecha_inicio, "%Y-%m-%d %H:%M:%S")
            fecha_fin = datetime.strptime(fecha_fin, "%Y-%m-%d %H:%M:%S")
            filtro["fecha_hora"] = {"$gte": fecha_inicio.strftime('%Y-%m-%d %H:%M:%S'),
                                    "$lte": fecha_fin.strftime('%Y-%m-%d %H:%M:%S')}
        except ValueError:
            return jsonify({"error": "Formato de fecha inválido. Usa YYYY-MM-DD HH:MM:SS"}), 400

    # Consultar MongoDB con los filtros
    datos = list(colecciones[sensor].find(filtro, {"_id": 0}).sort("fecha_hora", -1).limit(limite))

    # Si el rango de tiempo es muy grande, aplicamos muestreo simple
    if len(datos) > 500:
        datos = datos[::len(datos) // 500]  # Tomamos solo cada X elementos

    return jsonify(datos), 200
```

- **Ruta `/datos/<sensor>/filtrado`**: Devuelve los registros de un sensor filtrados por un rango de fechas y con un límite de registros.
- El rango de fechas es opcional. Si se proporcionan, se validan y se usa un filtro para buscar los datos entre las fechas indicadas.
- También se permite establecer un límite en la cantidad de registros devueltos, con un valor por defecto de 500.
- Si el rango de datos es muy grande, se realiza un muestreo simple para reducir la cantidad de resultados.

4. Obtener el último dato de un sensor

```python
@app.route('/datos/<sensor>/ultimo', methods=['GET'])
def obtener_ultimo_dato(sensor):
    if sensor not in colecciones:
        return jsonify({"error": "Sensor no encontrado"}), 404

    # Buscar el último dato en la colección del sensor
    dato = colecciones[sensor].find_one({}, {"_id": 0}, sort=[("fecha_hora", -1)])
```

```python
    if not dato:
        return jsonify({"error": "No hay datos disponibles para este sensor"}), 404

    return jsonify(dato), 200
```

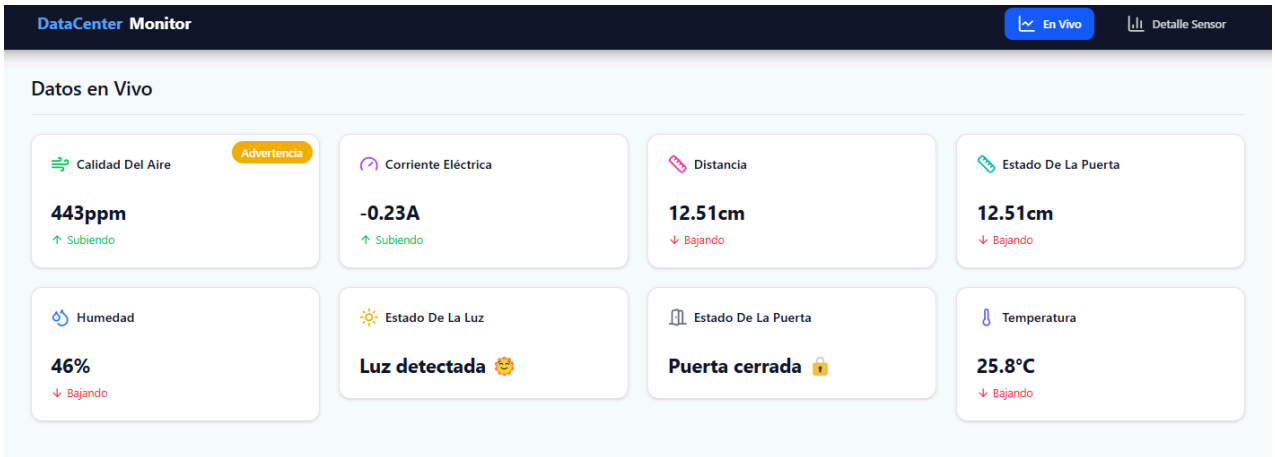
- **Ruta `/datos/<sensor>/ultimo`**: Devuelve el último registro almacenado para el sensor especificado.
- Se busca el último documento en la colección correspondiente, ordenado por `fecha_hora` de forma descendente.
- Si no se encuentran datos para el sensor, se devuelve un error 404.

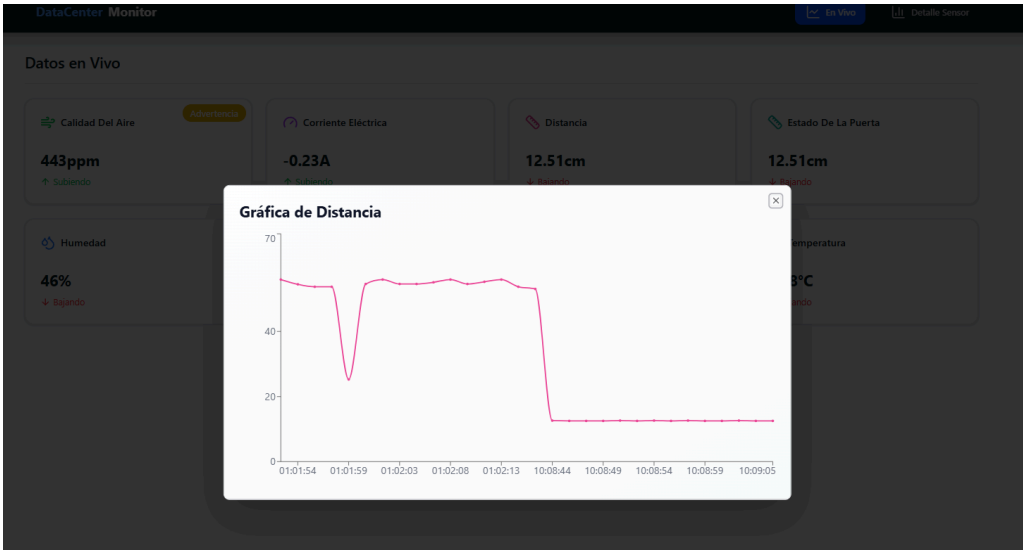
Ejecución de la aplicación

```python
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

- Esto inicia la aplicación Flask en el puerto 5000 y habilita el modo de depuración (`debug=True`), lo que permite detectar y mostrar errores de manera más detallada durante el desarrollo.

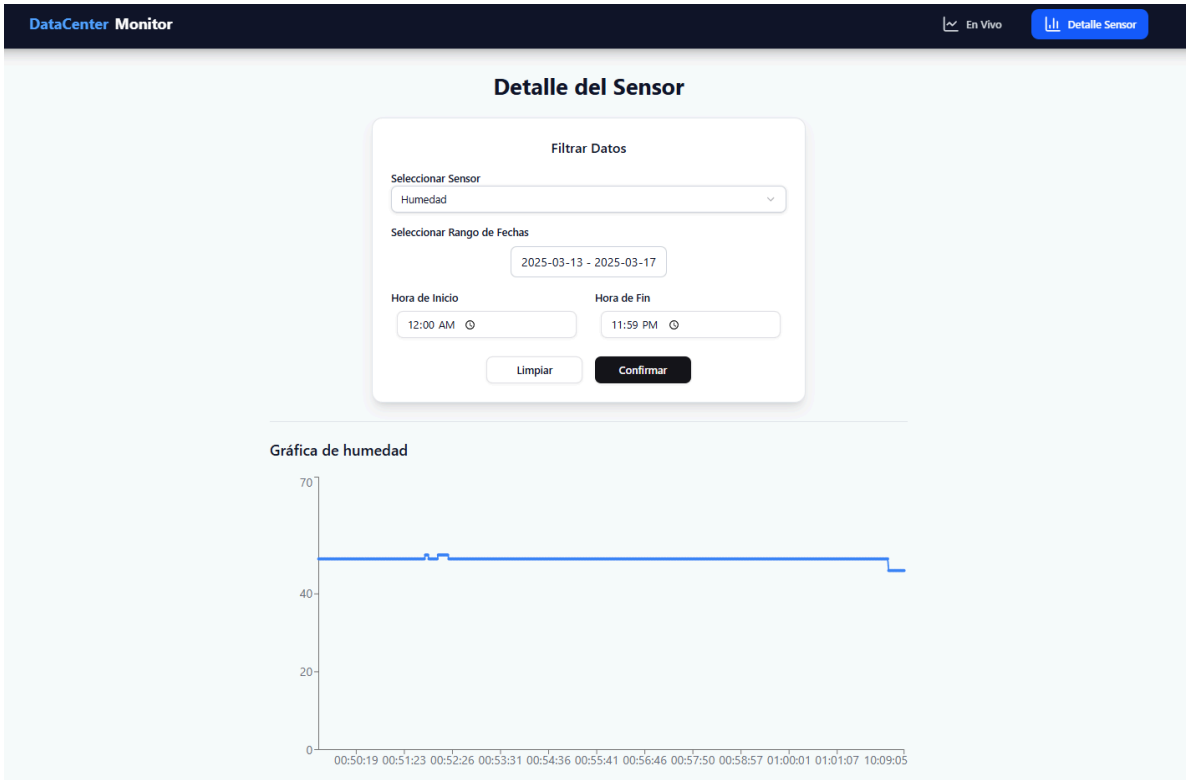
Frontend

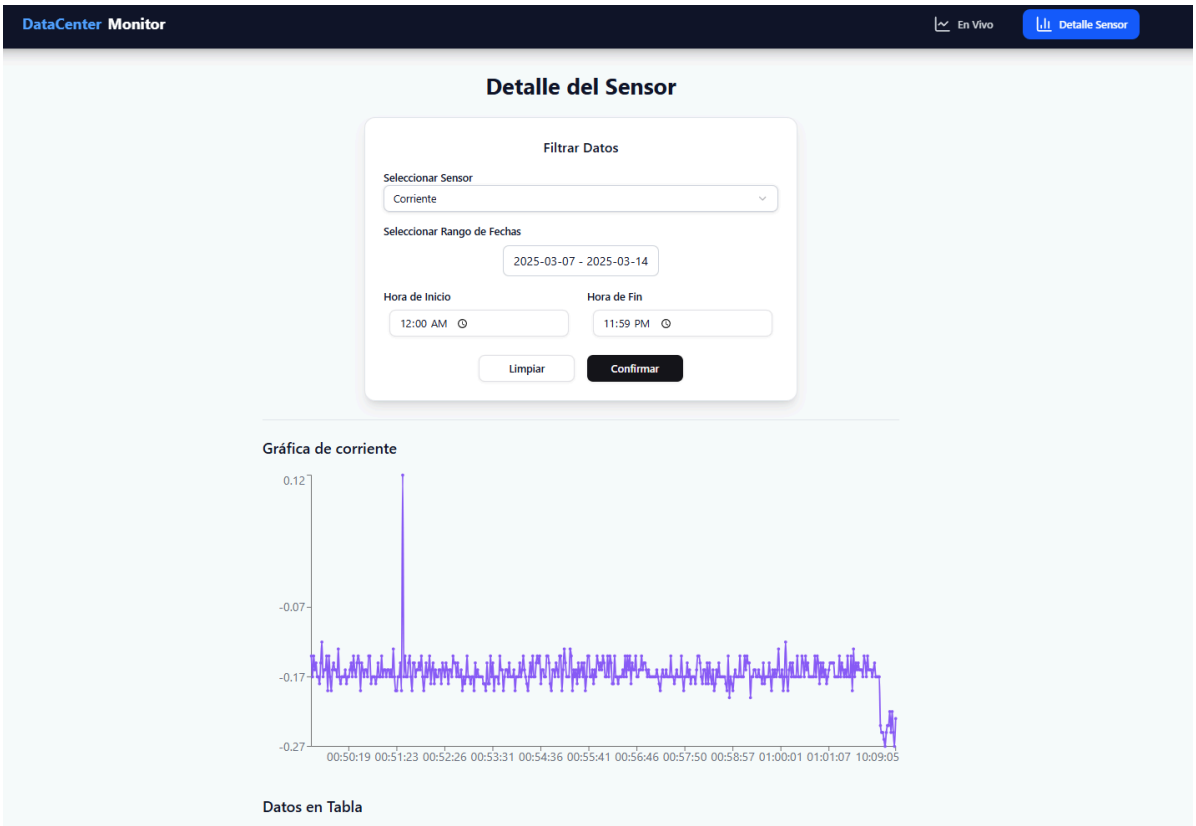



DataCenter Monitor
En Vivo
Detalle Sensor
Datos en Vivo
Calidad Del Aire
Advertencia
443ppm
Subiendo
Corriente Eléctrica
-0.23A
Distancia
12.51cm
Estado De La Puerta
12.51cm
Humedad
46%
Bajando
Gráfica de Distancia
70
40
20
0
01:01:54
01:01:59
01:02:03
01:02:08
01:02:13
10:08:44
10:08:49
10:08:54
10:08:59
10:09:05

Datos en Vivo
Calidad Del Aire
Advertencia
443ppm
Subiendo
Corriente Eléctrica
-0.23A
Distancia
12.51cm
Estado De La Puerta
12.51cm
Humedad
46%
Bajando
Gráfica de Calidad del Aire
580
300
150
0
01:01:54
01:01:59
01:02:03
01:02:08
01:02:13
10:08:44
10:08:49
10:08:54
10:08:59
10:09:05

Datos en Vivo
Calidad Del Aire
Advertencia
443ppm
Subiendo
Corriente Eléctrica
-0.23A
Distancia
12.51cm
Estado De La Puerta
12.51cm
Humedad
46%
Bajando
Gráfica de Distancia de la Puerta
70
40
20
0
01:01:54
01:01:59
01:02:03
01:02:08
01:02:13
10:08:44
10:08:49
10:08:54
10:08:59
10:09:05


DataCenter Monitor
En Vivo
Detalle Sensor
Detalle del Sensor
Filtrar Datos
Seleccionar Sensor
Humedad
Seleccionar Rango de Fechas
2025-03-13 - 2025-03-17
Hora de Inicio
12:00 AM
Hora de Fin
11:59 PM
Limpiar
Confirmar
Gráfica de humedad
70
40
20
0
00:50:19 00:51:23 00:52:26 00:53:31 00:54:36 00:55:41 00:56:46 00:57:50 00:58:57 01:00:01 01:01:07 10:09:05

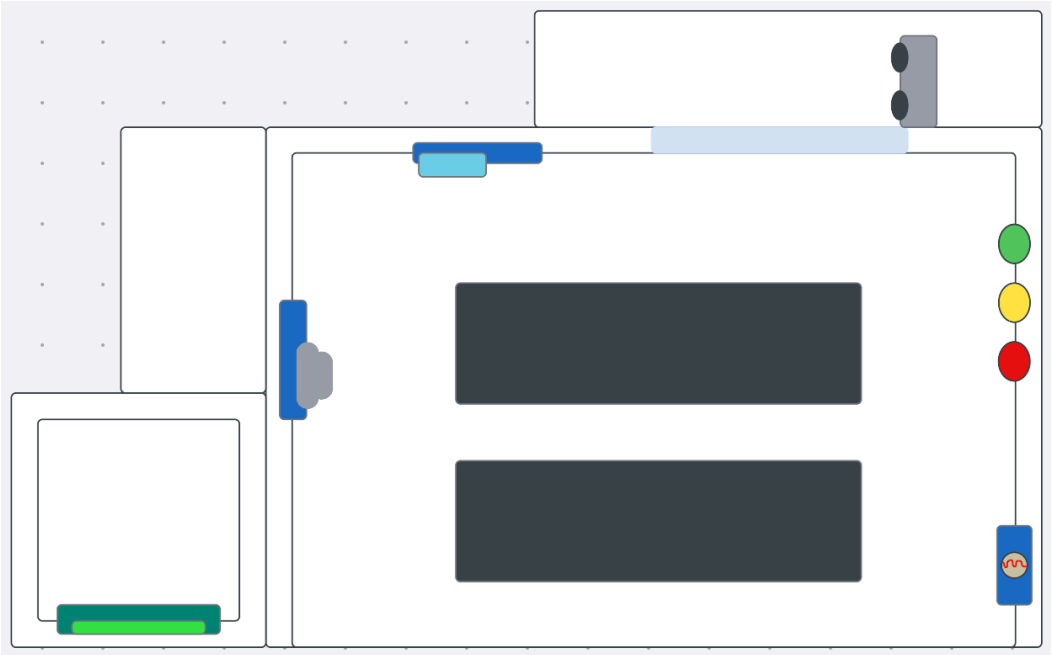

DataCenter Monitor
En Vivo
Detalle Sensor
Detalle del Sensor
Filtrar Datos
Seleccionar Sensor
Corriente
Seleccionar Rango de Fechas
2025-03-07 - 2025-03-14
Hora de Inicio
12:00 AM
Hora de Fin
11:59 PM
Limpiar
Confirmar
Gráfica de corriente
0.12
-0.07
-0.17
-0.27
00:50:19 00:51:23 00:52:26 00:53:31 00:54:36 00:55:41 00:56:46 00:57:50 00:58:57 01:00:01 01:01:07 10:09:05
Datos en Tabla

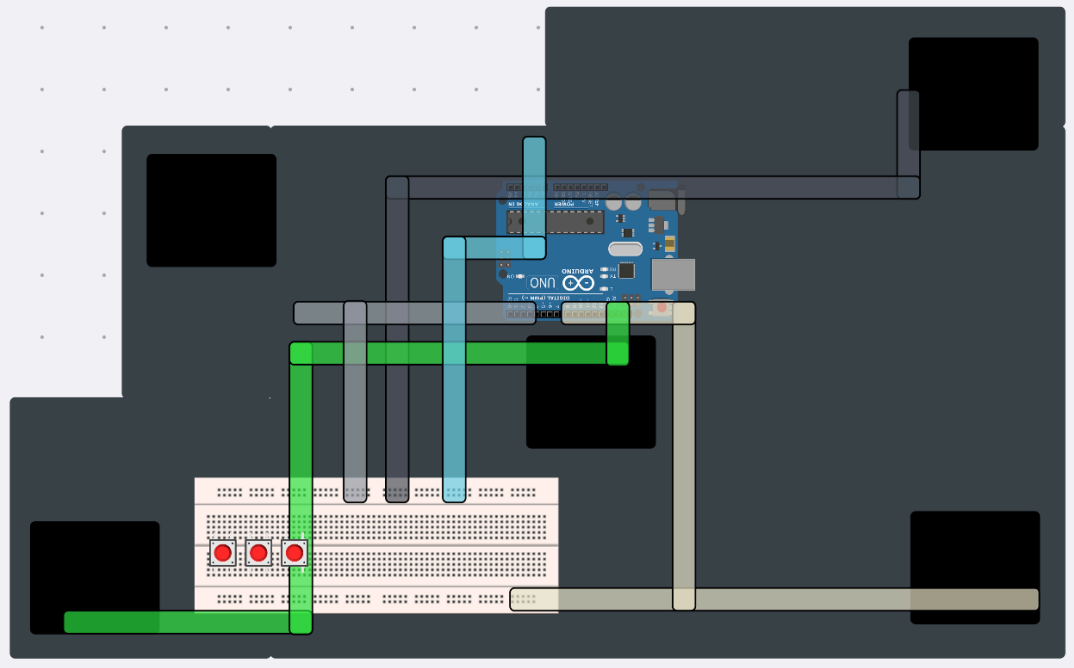
Datos en Tabla

Fecha y Hora	Valor
14/03/2025 10:09:05	-0.23
14/03/2025 10:09:04	-0.27
14/03/2025 10:09:02	-0.25
14/03/2025 10:09:01	-0.22
14/03/2025 10:08:59	-0.25
14/03/2025 10:08:58	-0.22
14/03/2025 10:08:56	-0.24
14/03/2025 10:08:54	-0.24
14/03/2025 10:08:53	-0.25
14/03/2025 10:08:51	-0.27

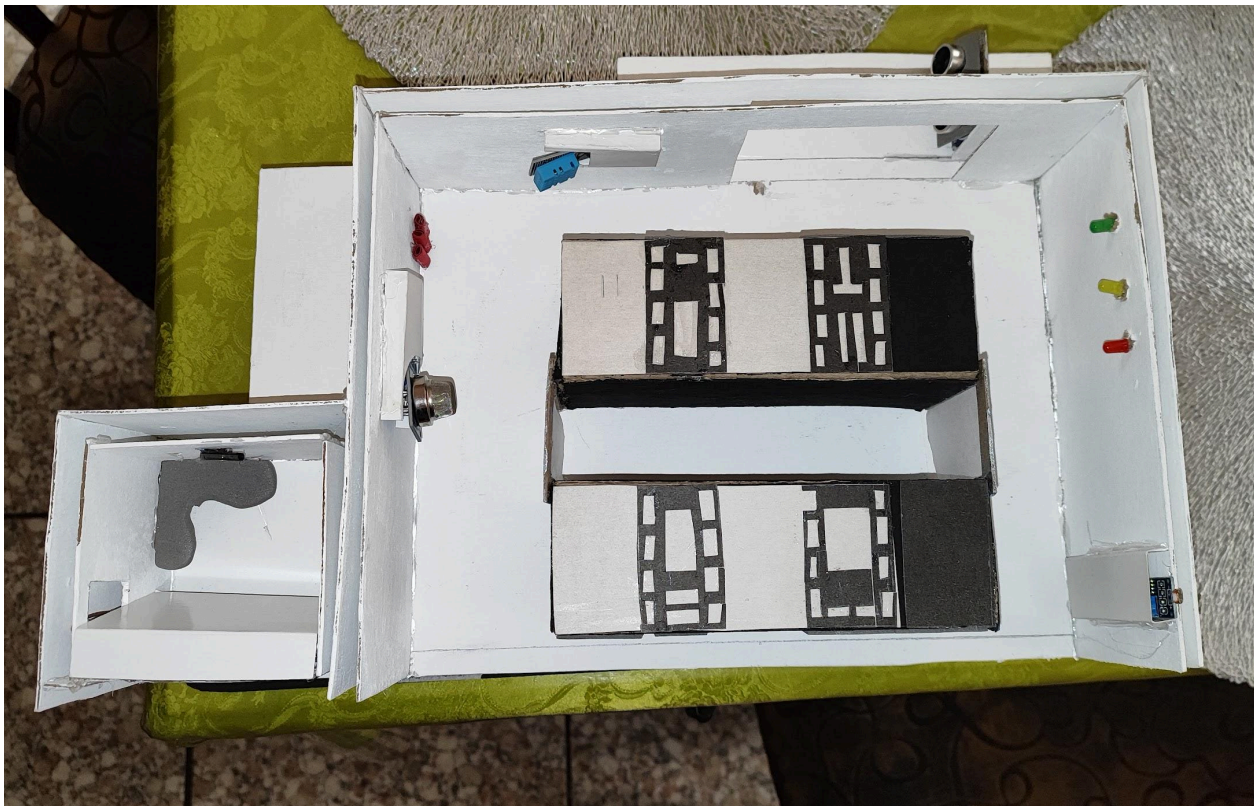
Prototipo Físico

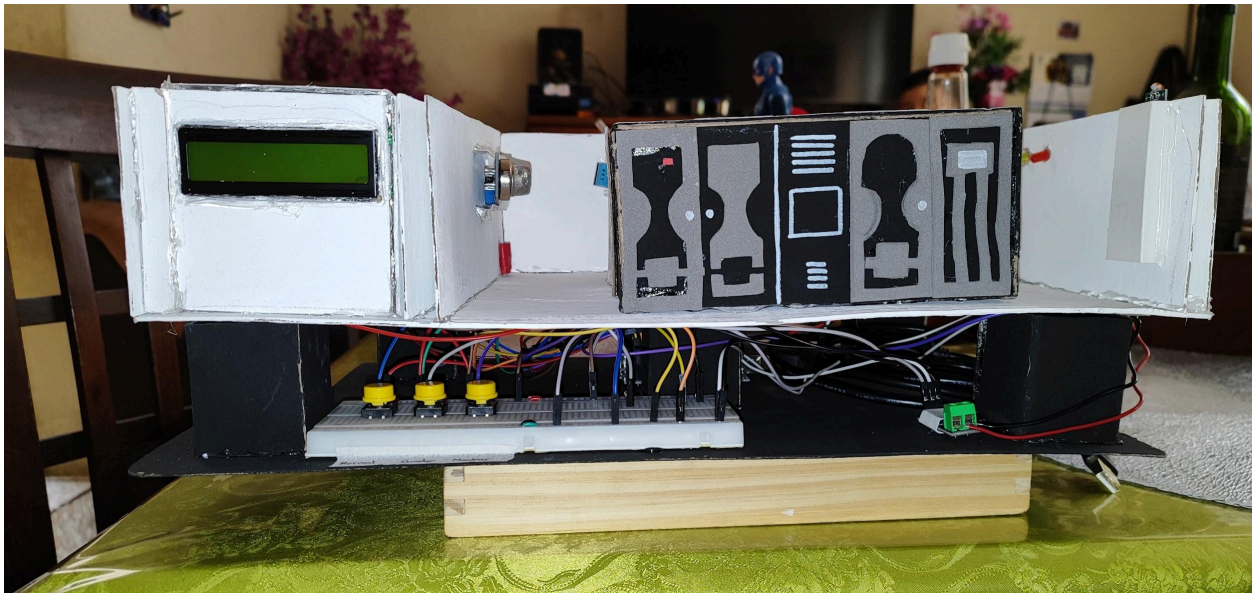
A continuación se presentan los planos y el diseño previo a la realización de la maqueta física, para poder entender en qué parte iría cada sensor y cada parte esencial de la maqueta. También para poder ver cómo sería el cableado físico de cada sensor, y en donde estaría el Arduino Uno y la protoboard en el diseño físico.

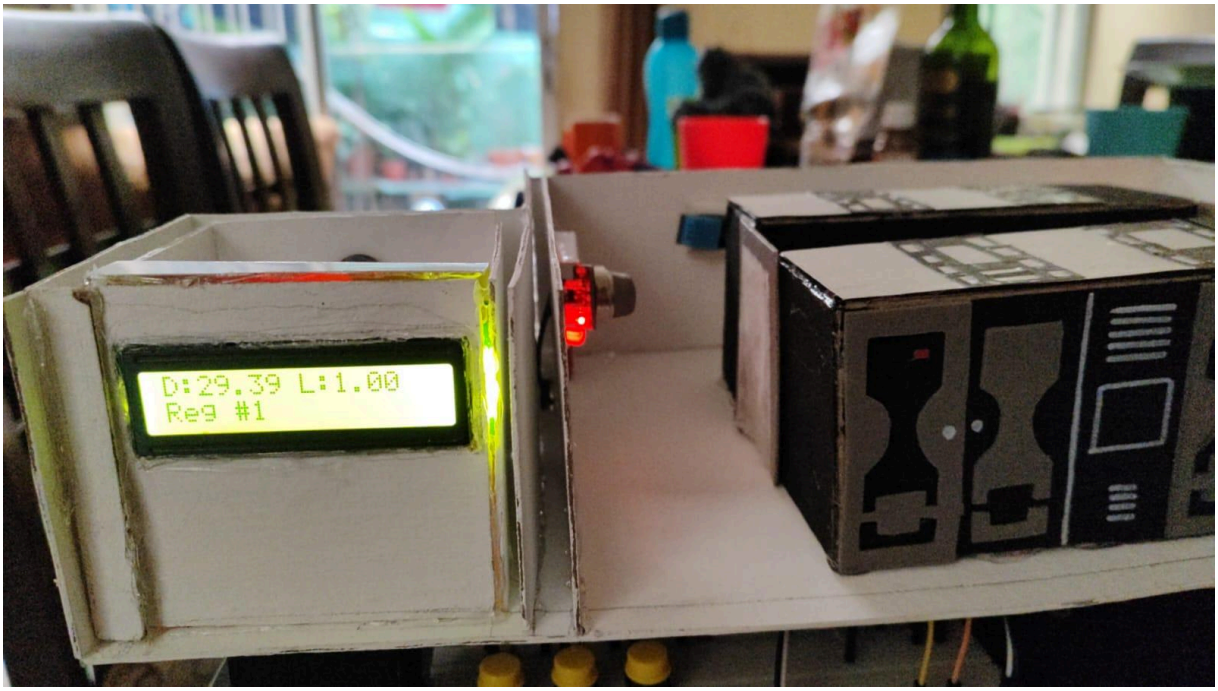


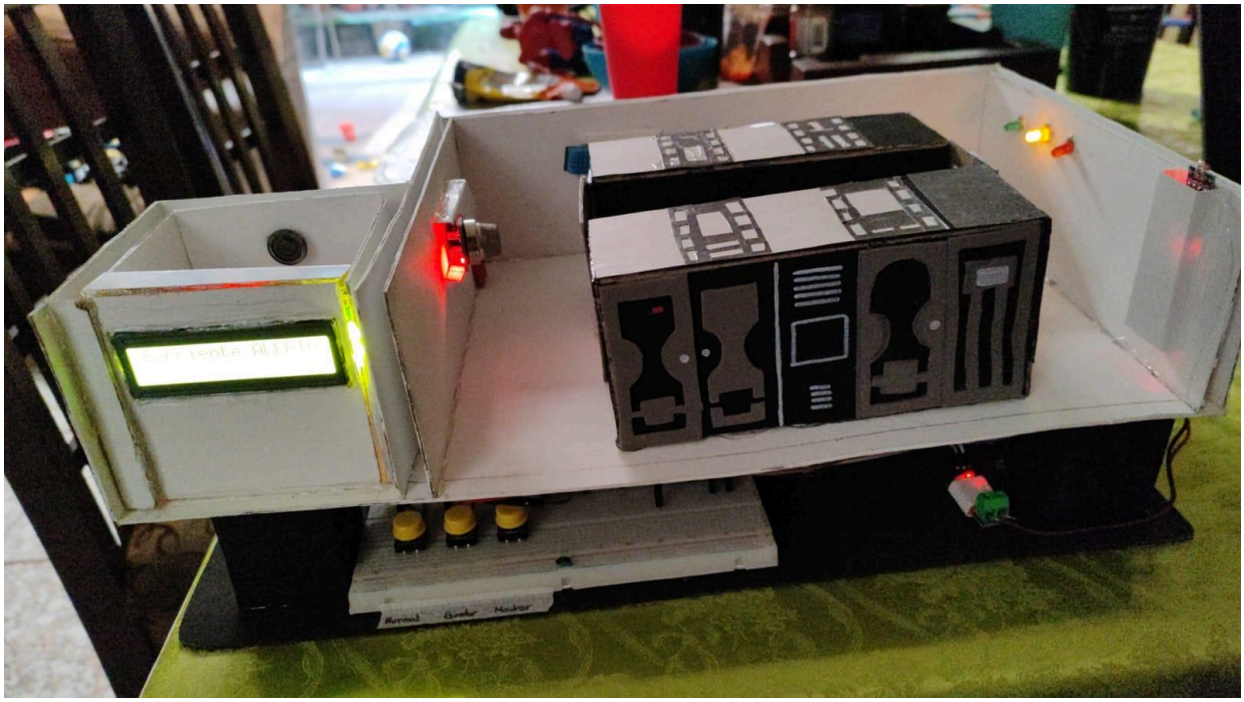


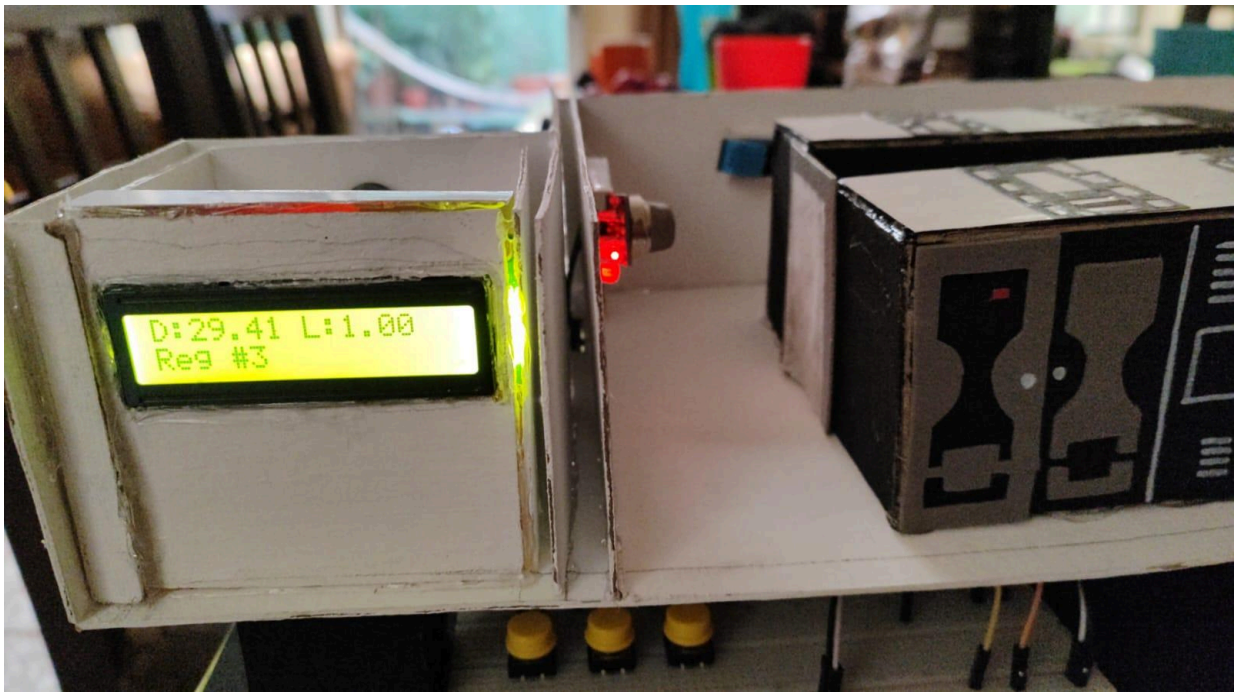
En las siguientes fotografías se muestra la maqueta física finalizada y también se muestra cada parte importante de la maqueta, así como el correcto funcionamiento de esta.






D:29.39 L:1.00
Reg #1





Link del Repositorio

https://github.com/AngelSGonza2107/ARQUI2B_1S2025_G16.git