

**PROYECTO HIDROPONICO**  
**ELECTIVA CIENCIAS DE LA COMPUTACIÓN I**  
**DATA SCIENCE**

**Integrantes**

**Cristhian Eduardo Sanchez Mosquera**

**Juan Manuel Diaz Valenzuela**

**Linda Valentina López Rubiano**

**WEB SERVICE (Flask y acceso a la base de datos)**

**30 de mayo de 2025**

**NEIVA HUILA**

# Contenido

Introducción .....3

Objetivo General.....3

    Objetivos Específicos .....3

Arquitectura.....3

# Introducción

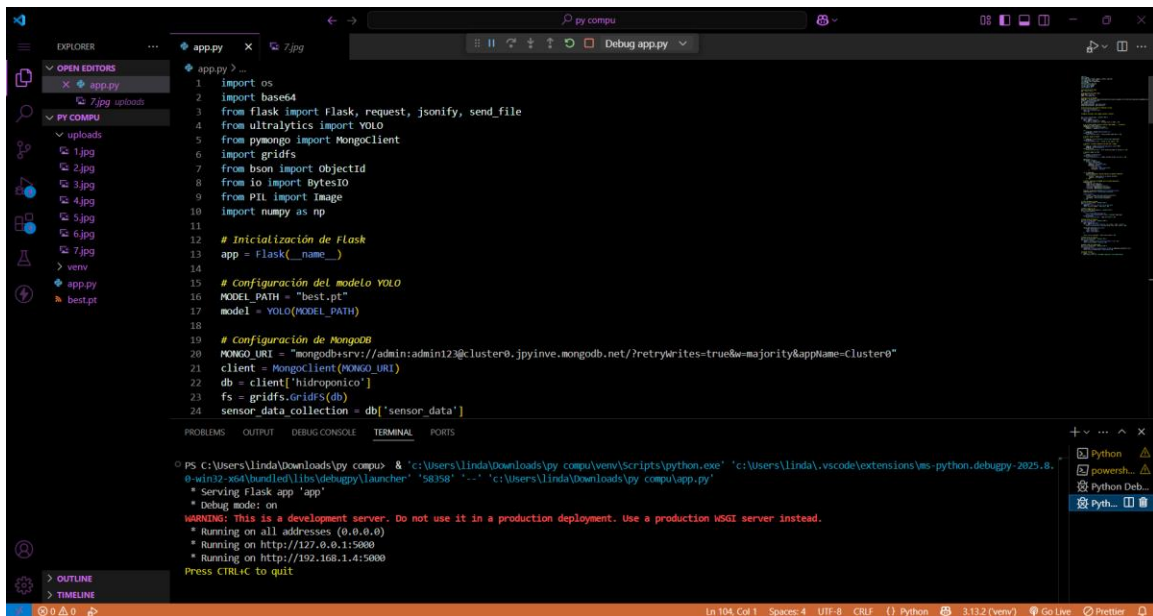
Este proyecto implementa un backend para un sistema de monitoreo y análisis de imágenes en un entorno hidropónico. Utiliza inteligencia artificial para la detección de objetos en imágenes, almacenamiento de datos de sensores y gestión de archivos, facilitando la automatización y supervisión remota del sistema.

## Objetivo General

Desarrollar un backend capaz de recibir, almacenar y analizar imágenes mediante un modelo de inteligencia artificial, así como gestionar y consultar datos de sensores, para optimizar el monitoreo de un sistema hidropónico

## Objetivos Específicos

1. Permitir la carga y almacenamiento seguro de imágenes en una base de datos.
2. Analizar imágenes usando un modelo YOLO para la detección automática de objetos relevantes.
3. Registrar y consultar datos provenientes de sensores ambientales.



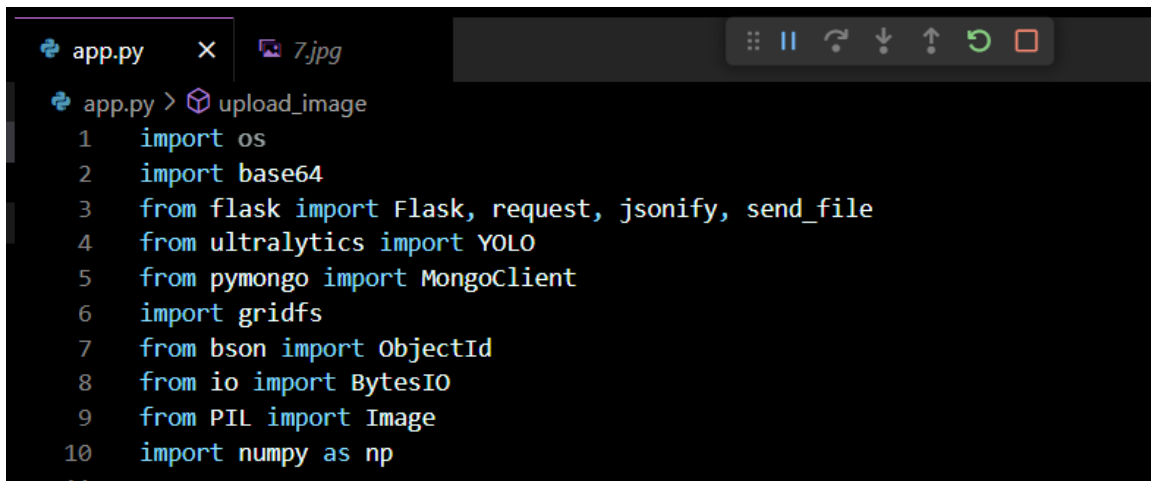
## Arquitectura

El backend está construido con **Flask** (Python) y se conecta a una base de datos **MongoDB** en la nube. El almacenamiento de imágenes se realiza usando **GridFS**. El análisis de imágenes se lleva a cabo con un modelo **YOLO** (You Only Look Once) para la

detección de objetos.

La arquitectura general es la siguiente:

- **Cliente (Frontend o Dispositivo IoT):** Envía imágenes y datos de sensores al backend mediante peticiones HTTP.
- **Backend (Flask):**
  - Recibe imágenes y datos de sensores.
  - Almacena imágenes en MongoDB usando GridFS.
  - Procesa imágenes con YOLO y guarda los resultados.
  - Expone endpoints para consultar imágenes, predicciones y datos de sensores.
- **Base de Datos (MongoDB):**
  - Almacena imágenes, predicciones y datos de sensores.



```
app.py x 7.jpg
app.py > upload_image
1 import os
2 import base64
3 from flask import Flask, request, jsonify, send_file
4 from ultralytics import YOLO
5 from pymongo import MongoClient
6 import gridfs
7 from bson import ObjectId
8 from io import BytesIO
9 from PIL import Image
10 import numpy as np
```

En esta sección se importan todas las librerías necesarias para el funcionamiento del backend:

- os, base64: Utilidades del sistema y manejo de imágenes en base64.
- Flask: Framework web para crear la API.
- ultralytics.YOLO: Para cargar y usar el modelo de detección de objetos YOLO.
- pymongo, gridfs: Para conectarse y guardar archivos en MongoDB.
- bson.ObjectId: Para manejar identificadores únicos de MongoDB.

- io.BytesIO, PIL.Image, numpy: Para procesar imágenes.

```
# Inicialización de Flask
app = Flask(__name__)
```

- Se crea la aplicación Flask, que será el servidor web del backend.

`app = Flask(__name__)`.

```
# Configuración del modelo YOLO
MODEL_PATH = "best.pt"
model = YOLO(MODEL_PATH)
```

- Se define la ruta del modelo entrenado ([best.pt](#)) y se carga en memoria para su uso inmediato en la API.

```
# Configuración de MongoDB
MONGO_URI = "mongodb+srv://admin:admin123@cluster0.jpyinve.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
client = MongoClient(MONGO_URI)
db = client['hidroponico']
fs = gridfs.GridFS(db)
sensor_data_collection = db['sensor_data']
prediction_collection = db['prediction']
```

- Se conecta a una base de datos MongoDB en la nube.
- Se selecciona la base de datos hidroponico.
- Se inicializa GridFS para almacenar imágenes.
- Se definen las colecciones para datos de sensores y predicciones.

```
# Función auxiliar para convertir ObjectId a string
def serialize_document(doc):
    doc['_id'] = str(doc['_id'])
    return doc
```

Convierte el campo `_id` de los documentos de MongoDB a tipo [str](#).

Esto es necesario porque los identificadores de MongoDB ([ObjectId](#)) no son serializables directamente a JSON, y convertirlos a string permite enviarlos correctamente en las respuestas de la API.

```

@app.route('/upload-image', methods=['POST'])
def upload_image():
    data = request.get_json()
    if not data or 'imagen' not in data:
        return jsonify({'error': 'No image field in JSON'}), 400

    # Si la cadena base64 incluye el prefijo "data:image/...", eliminarlo
    base64_str = data['imagen']
    if base64_str.startswith("data:image"):
        base64_str = base64_str.split(",")[1]

    try:
        image_data = base64.b64decode(base64_str)
    except Exception:
        return jsonify({'error': 'Invalid base64 image data'}), 400

    # Guardar imagen en GridFS
    try:
        image_id = fs.put(image_data, content_type='image/jpeg')
    except Exception:
        return jsonify({'error': 'Failed to save image'}), 500

```

### Definición del endpoint

```
@app.route('/upload-image', methods=['POST'])
```

```
def upload_image():
```

Se define el endpoint /upload-image que acepta peticiones POST para recibir imágenes.

### Recepción y validación de datos

```
data = request.get_json()
```

```
if not data or 'imagen' not in data:
```

```
    return jsonify({'error': 'No image field in JSON'}), 400
```

Se obtiene el JSON enviado por el cliente y se verifica que contenga el campo imagen. Si falta, se responde con un error 400.

### **Limpieza del string base64**

```
base64_str = data['imagen']
```

```
if base64_str.startswith("data:image"):
```

```
    base64_str = base64_str.split(",")[1]
```

Si la cadena base64 tiene un prefijo tipo [data:image/...](#), se elimina para quedarse solo con la parte codificada.

### **Decodificación de la imagen**

```
try:
```

```
    image_data = base64.b64decode(base64_str)
```

```
except Exception:
```

```
    return jsonify({'error': 'Invalid base64 image data'}), 400
```

Se intenta decodificar la imagen de base64 a bytes. Si falla, se responde con un error 400.

### **Guardado en GridFS**

```
try:
```

```
    image_id = fs.put(image_data, content_type='image/jpeg')
```

```
except Exception:
```

```
    return jsonify({'error': 'Failed to save image'}), 500
```

Se guarda la imagen en la base de datos MongoDB usando GridFS. Si ocurre un error, se responde con un error 500.

```

# Convertir a formato compatible con YOLO (PIL + numpy)
try:
    image_pil = Image.open(BytesIO(image_data)).convert("RGB")
    image_np = np.array(image_pil)
except Exception:
    return jsonify({'error': 'Error processing image for analysis'}), 500

# Analizar imagen con YOLO
try:
    results = model(image_np)
except Exception as e:
    return jsonify({'error': f'Model inference failed: {str(e)}'}), 500

detections = []
for result in results:
    for box in result.bboxes:
        clase = int(box.cls[0])
        confianza = float(box.conf[0])
        detections.append({
            "class_id": clase,
            "class_name": model.names[clase],
            "confidence": confianza,
        })

```

*(Conversión, análisis con YOLO y extracción de resultados)*

### Conversión de la imagen para YOLO

**try:**

```
image_pil = Image.open(BytesIO(image_data)).convert("RGB")
```

```
image_np = np.array(image_pil)
```

**except Exception:**

```
return jsonify({'error': 'Error processing image for analysis'}), 500
```

Se convierte la imagen a formato RGB usando PIL y luego a un array de numpy, que es el formato requerido por el modelo YOLO.

### Análisis de la imagen con YOLO

**try:**

```
results = model(image_np)
```

**except Exception as e:**

```
return jsonify({'error': f'Model inference failed: {str(e)}'}), 500
```

Se ejecuta el modelo YOLO sobre la imagen para detectar objetos. Si ocurre un error, se responde con un error 500.



## Procesamiento de resultados

```
detections = []
```

```
for result in results:
```

```
    for box in result.bboxes:
```

```
        clase = int(box.cls[0])
```

```
        confianza = float(box.conf[0])
```

```
        detections.append({
```

```
            "class_id": clase,
```

```
            "class_name": model.names[clase],
```

```
            "confidence": confianza,
```

```
        })
```

Se recorren los resultados del modelo, extrayendo la clase y la confianza de cada objeto detectado.

Se almacena cada detección en una lista como un diccionario con el ID de la clase, el nombre y la confianza.

```
if not detections:
    # No hay detecciones, devolver mensaje sin guardar predicción
    return jsonify({
        'message': 'Image saved but no objects detected',
        'image_id': str(image_id)
    }), 200

# Guardar predicción en MongoDB (solo la primera detección)
prediction_doc = {
    "image_id": str(image_id),
    "imagen_base64": base64_str,
    "class_id": detections[0]["class_id"],
    "class_name": detections[0]["class_name"],
    "confidence": detections[0]["confidence"]
}
inserted = prediction_collection.insert_one(prediction_doc)
prediction_doc['_id'] = str(inserted.inserted_id)

return jsonify({
    'message': 'Image saved and analyzed successfully',
    "class_name": detections[0]["class_name"],
    "confidence": detections[0]["confidence"]
}), 200
```

*(Manejo de detecciones y guardado de predicción)*

```
if not detections:
```

```
    # No hay detecciones, devolver mensaje sin guardar predicción
```

```
    return jsonify({
```

```
    'message': 'Image saved but no objects detected',  
    'image_id': str(image_id)  
  }, 200
```

Si el modelo YOLO no detecta ningún objeto en la imagen, el sistema responde informando que la imagen fue guardada, pero no se detectaron objetos

### Guardar predicción en MongoDB

```
prediction_doc = {  
    "image_id": str(image_id),  
    "imagen_base64": base64_str,  
    "class_id": detections[0]["class_id"],  
    "class_name": detections[0]["class_name"],  
    "confidence": detections[0]["confidence"]  
}  
  
inserted = prediction_collection.insert_one(prediction_doc)  
prediction_doc['_id'] = str(inserted.inserted_id)
```

Si hay al menos una detección, se guarda la información de la primera detección (clase, nombre y confianza) junto con el ID de la imagen y la imagen en base64 en la colección de predicciones de MongoDB.

### Respuesta exitosa

```
return jsonify({  
    'message': 'Image saved and analyzed successfully',  
    "class_name": detections[0]["class_name"],  
    "confidence": detections[0]["confidence"]  
}), 200
```

Se responde al cliente indicando que la imagen fue guardada y analizada correctamente, junto con la clase detectada y el nivel de confianza.

```

# Listar imágenes en GridFS
@app.route('/list-images', methods=['GET'])
def list_images():
    image_ids = [str(file._id) for file in fs.find()]
    return jsonify({'images': image_ids}), 200

# Obtener imagen por ID
@app.route('/get-image/<image_id>', methods=['GET'])
def get_image(image_id):
    try:
        file = fs.get(ObjectId(image_id))
        return send_file(BytesIO(file.read()), mimetype='image/jpeg')
    except Exception:
        return jsonify({'error': 'Image not found'}), 404

# Guardar datos de sensores
@app.route('/sensor-data', methods=['POST'])
def save_sensor_data():
    data = request.get_json()
    if not data or not all(k in data for k in ("zona", "tipo", "valor")):
        return jsonify({"error": "Missing fields: zona, tipo, valor"}), 400

    sensor_data_collection.insert_one({
        "zona": data["zona"],
        "tipo": data["tipo"],
        "valor": data["valor"]
    })

    return jsonify({"message": "Sensor data saved"}), 200

# Obtener datos de sensores
@app.route('/sensor-data', methods=['GET'])
def get_sensor_data():
    results = list(sensor_data_collection.find({}, {'_id': 0}))
    return jsonify({"data": results}), 200

```

(Listar imágenes, obtener imagen por ID, guardar y obtener datos de sensores)

### Listar imágenes en GridFS

```
@app.route('/list-images', methods=['GET'])
```

```
def list_images():
```

```
    image_ids = [str(file._id) for file in fs.find()]
```

```
    return jsonify({'images': image_ids}), 200
```

Este endpoint devuelve una lista de los IDs de todas las imágenes almacenadas en la base de datos usando GridFS. Es útil para que el cliente pueda consultar qué imágenes hay disponibles.

### Obtener imagen por ID

```
@app.route('/get-image/<image_id>', methods=['GET'])
```

```
def get_image(image_id):
```

```
    try:
```

```

    file = fs.get(ObjectId(image_id))

    return send_file(BytesIO(file.read()), mimetype='image/jpeg')

except Exception:

    return jsonify({'error': 'Image not found'}), 404

```

Permite descargar una imagen específica a partir de su ID. Si la imagen existe, la envía como archivo JPEG; si no, responde con un error 404.

### Guardar datos de sensores

```

@app.route('/sensor-data', methods=['POST'])
def save_sensor_data():

    data = request.get_json()

    if not data or not all(k in data for k in ("zona", "tipo", "valor")):

        return jsonify({'error': "Missing fields: zona, tipo, valor"}), 400

    sensor_data_collection.insert_one({

        "zona": data["zona"],

        "tipo": data["tipo"],

        "valor": data["valor"]

    })

    return jsonify({"message": "Sensor data saved"}), 200

```

Este endpoint recibe datos de sensores (zona, tipo y valor) y los guarda en la base de datos. Si falta algún campo, responde con un error 400.

### Obtener datos de sensores

```

@app.route('/sensor-data', methods=['GET'])
def get_sensor_data():

    results = list(sensor_data_collection.find({}, {'_id': 0}))

    return jsonify({"data": results}), 200

```

Permite consultar todos los datos de sensores almacenados, devolviendo una lista en formato JSON.

```
# Obtener todas las predicciones
@app.route('/predictions', methods=['GET'])
def get_predictions():
    predictions = [serialize_document(doc) for doc in prediction_collection.find()]
    return jsonify({"predictions": predictions}), 200

# Ejecutar servidor
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True, use_reloader=False)
```

*(Obtener todas las predicciones y ejecutar el servidor)*

### Obtener todas las predicciones

```
@app.route('/predictions', methods=['GET'])
```

```
def get_predictions():
```

```
    predictions = [serialize_document(doc) for doc in prediction_collection.find()]
```

```
    return jsonify({"predictions": predictions}), 200
```

Este endpoint devuelve todas las predicciones almacenadas en la base de datos. Cada predicción incluye información sobre la imagen, la clase detectada y la confianza.

### Ejecutar servidor

```
if __name__ == '__main__':
```

```
    app.run(host='0.0.0.0', port=5000, debug=True, use_reloader=False)
```

Inicia el servidor Flask para que la API esté disponible en la red, escuchando en el puerto 5000.