Documentación Técnica – Módulo Backend de GridFS para Flask

DOCENTE:

Juan Antonio Castro Silva

PRESENTADO POR:

María José García Arias

Joseph Gutierrez Martinez

Electiva Ciencias de la Computación I

Neiva – Huila

2025

1. Descripción General

Este sistema backend está diseñado como un microservicio RESTful para almacenar, recuperar y consultar datos de sensores e imágenes en el contexto de cultivos hidropónicos inteligentes. Utiliza MongoDB como motor de base de datos, junto con GridFS para almacenamiento de archivos binarios (imágenes) y una API web desarrollada en Flask para probar la posterior implementación. La lógica de persistencia se encuentra desacoplada en el módulo gridfs_utils.py.

2. Objetivos

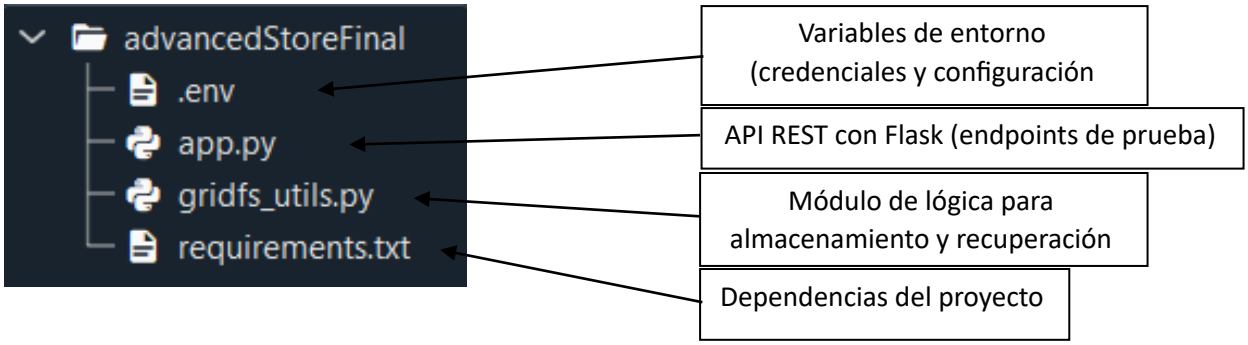
2.1. Objetivo General

Desarrollar una solución de almacenamiento eficiente para datos ambientales y visuales, proveniente de sensores IoT desplegados en invernaderos, utilizando MongoDB y GridFS.

2.2. Objetivos Específicos

- Permitir la carga de datos de sensores (temperatura, humedad, etc.) de forma segura y organizada.
- Gestionar el almacenamiento y recuperación de imágenes asociadas a eventos o monitoreo visual.
- Evitar el almacenamiento redundante mediante intervalos mínimos de guardado.
- Proporcionar endpoints de prueba para la integración con clientes IoT o aplicaciones móviles.

3. Estructura del Proyecto



4. Tecnologías y Librerías

- Flask: Framework para la API REST
- MongoDB + GridFS : Base de datos y almacenamiento binario
- Pymongo: Cliente oficial de MongoDB
- Dotenv: Carga de configuraciones sensibles (.env)
- Dnspython: Soporte para nombres DNS en URI de MongoDB

5. Flujo de Datos

5.1. Backend REST

- Los sensores o cámaras realizan peticiones POST hacia los endpoints /upload-data o /upload-image.
- Las funciones del módulo gridfs_utils.py procesan y almacenan los datos:
  - Los datos de sensores son insertados como documentos JSON en la colección sensor_data.
  - Las imágenes se almacenan en GridFS, con metadatos como nombre y tipo MIME.
- Se evita la sobrecarga verificando el tiempo entre lecturas similares (≥10s).
- La API permite recuperar datos por tipo de sensor, listar imágenes o descargarlas por ID.

5.2. Backend MQTT

- Sensores ESP32 publican lecturas por MQTT.
- mqtt.py recibe los mensajes, los interpreta y los guarda localmente.
- Si hay conexión, se replica a MongoDB en la nube.
- Se usan hilos secundarios para reintentar sincronización.

6. Detalle del Módulo gridfs_utils.py

6.1. Conexión a la Base de Datos

```
load_dotenv()

MONGO_URI = f"mongodb+srv://{os.getenv('MONGO_USER')}:{os.getenv('MONGO_PASSWORD')}@{os.getenv('MONGO_CLUSTER')}/{os.getenv('MONGO_PARAMS')}"
client = MongoClient(MONGO_URI)
db = client[os.getenv("MONGO_DB")]
fs = gridfs.GridFS(db)
```

Carga variables desde .env y conecta tanto a la base de datos lógica como al sistema de archivos GridFS.

6.2. Función: should_store_data(...)

Evita la inserción repetida de datos similares en un corto intervalo:

```
def should_store_data(sensor_type, device_id, min_interval_seconds=10):
    last = db.sensor_data.find_one(
        {"tipo_sensor": sensor_type, "device_id": device_id},
        sort=[("timestamp", -1)]
    )
    if not last:
        return True
    return datetime.utcnow() - last["timestamp"] >= timedelta(seconds=min_interval_seconds)
```

- Revisa el último documento almacenado con ese tipo_sensor y device_id.
- Solo permite insertar si han pasado al menos min_interval_seconds.

6.3. Función: save_sensor_data(...)

```
def save_sensor_data(data, min_interval_seconds=10):
    if not should_store_data(data.get("tipo_sensor"), data.get("device_id"), min_interval_seconds):
        return None
    doc = {
        "tipo_sensor": data.get("tipo_sensor", "unknown"),
        "valor": data.get("valor"),
        "unidad": data.get("unidad"),
        "timestamp": datetime.utcnow(),
        "device_id": data.get("device_id", "esp32"),
        }
    return db.sensor_data.insert_one(doc).inserted_id
```

Almacena documentos con campos:

- tipo_sensor, valor, unidad, device_id
- timestamp en formato UTC (ISO 8601)

6.4. Función: get_sensor_data_by_type(…)

```
def get_sensor_data_by_type(tipo_sensor):
    data = db.sensor_data.find({"tipo_sensor": tipo_sensor}, {"_id": 0})
    return list(data)
```

Filtra datos de sensores por tipo (temperatura, humedad, etc.) sin incluir el campo _id.

6.5. Función: get_all_sensor_data()

```
def get_all_sensor_data():
    data = db.sensor_data.find({}, {"_id": 0})
    return list(data)
```

Devuelve todos los documentos de sensores en la colección sensor_data.

6.6. Función: save_file(...)

```
def save_file(file):
    return fs.put(file.read(), filename=file.filename, content_type=file.content_type)
```

Guarda archivos en GridFS. Extrae nombre y tipo desde file.

6.7. Función: get_file_by_id(...)

```python
def get_file_by_id(file_id):
    file = fs.get(ObjectId(file_id))
    return file.read(), file.content_type
```

Devuelve contenido binario y MIME type para un archivo específico almacenado en GridFS.

6.8. Función: list_all_files()

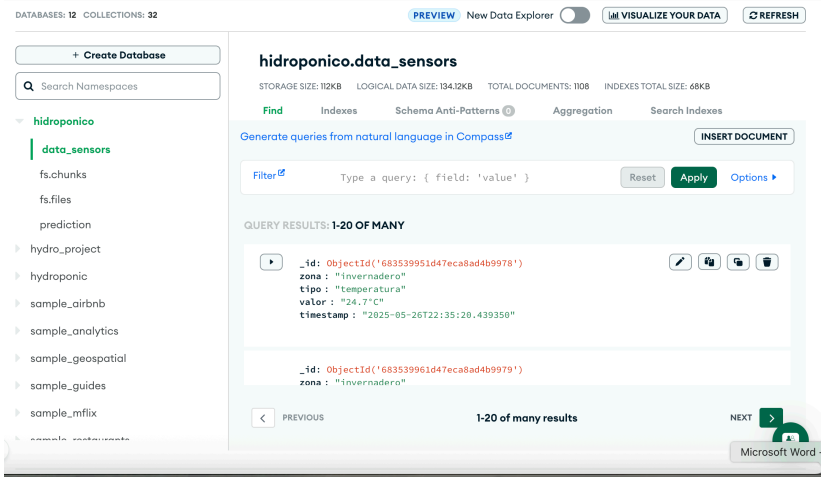
```python
def list_all_files():
    files = fs.find()
    return [{"file_id": str(f._id), "filename": f.filename, "content_type": f.content_type} for f in files]
```

Lista metadatos de todos los archivos almacenados.

7. Creación del Clúster en MongoDB Atlas

Para habilitar el almacenamiento en la nube, fue necesario crear un clúster en MongoDB Atlas. El proceso comenzó con el registro de una cuenta gratuita en la plataforma oficial de MongoDB. Una vez dentro del panel de control, se configuró un clúster compartido (nivel gratuito), seleccionando un proveedor de servicios en la nube (como AWS o Google Cloud) y una región geográfica cercana para optimizar la latencia. Luego, se creó un usuario de base de datos con credenciales de acceso que se incorporarían posteriormente al archivo de configuración .env.

A continuación, se habilitó el acceso a través de la red agregando la IP 0.0.0.0/0, lo cual permite conexiones desde cualquier origen —una configuración común durante la etapa de desarrollo. Posteriormente, se generó una URI de conexión desde la opción “Connect your application”, la cual fue descompuesta en partes para ser utilizadas como variables de entorno: el usuario, la contraseña, la dirección del clúster, el nombre de la base de datos lógica y los parámetros de conexión.



8. **Variables archivo .env**
   MONGO_USER
   MONGO_PASSWORD
   MONGO_CLUSTER
   MONGO_DB
   MONGO_PARAMS

9. **Endpoints de la API Flask (app.py)**

Método	Ruta	Descripción
POST	/upload-data	Subir lectura de sensor
POST	/upload-image	Subir imagen para almacenar en GridFS
GET	/data	Obtener todos los datos de sensores
GET	/images	Listar todas las imágenes almacenadas
GET	/image/<file_id>	Descargar imagen específica por ID
GET	/sensor-data/<tipo_sensor>	Filtrar datos de sensores por tipo

10. **Implementación y Ajustes Prácticos**

El sistema fue diseñado desde una arquitectura modular que separa la lógica de persistencia de datos (gridfs_utils.py) del manejo de rutas y peticiones HTTP (app.py). Esta separación tenía como objetivo:

- Evitar duplicación de código.
- Facilitar el mantenimiento y escalabilidad.
- Garantizar coherencia en el acceso a la base de datos y a GridFS.

10.1Desviaciones durante la implementación

Durante el proceso de desarrollo e integración, surgieron limitaciones que impidieron aplicar completamente esta arquitectura:

- **Falta de integración con el sistema MQTT**: Aunque gridfs_utils.py estaba preparado para ser utilizado tanto desde el módulo HTTP como desde un cliente MQTT (por ejemplo, para almacenar datos de sensores desde un broker), dicha integración no se completó.
- **Restricciones de tiempo**: Se priorizó tener rutas funcionales en app.py y una API mínima operativa, por lo que algunas funciones diseñadas para validar y espaciar los datos (como should_store_data) no fueron aprovechadas desde todos los puntos de entrada posibles.
- **No conexión con hilos de sincronización**: La función sincronizar_periodicamente() fue implementada correctamente, pero no fue activada dentro del flujo de ejecución de app.py, quedando como un componente listo, pero no utilizado.

11. Conclusión

Este sistema ofrece una solución robusta y adaptable para el monitoreo de cultivos hidropónicos. Integra recolección de datos en tiempo real, almacenamiento redundante, arquitectura modular y servicios REST. Aunque algunos componentes no fueron activados completamente, el sistema se encuentra listo para futuras expansiones.