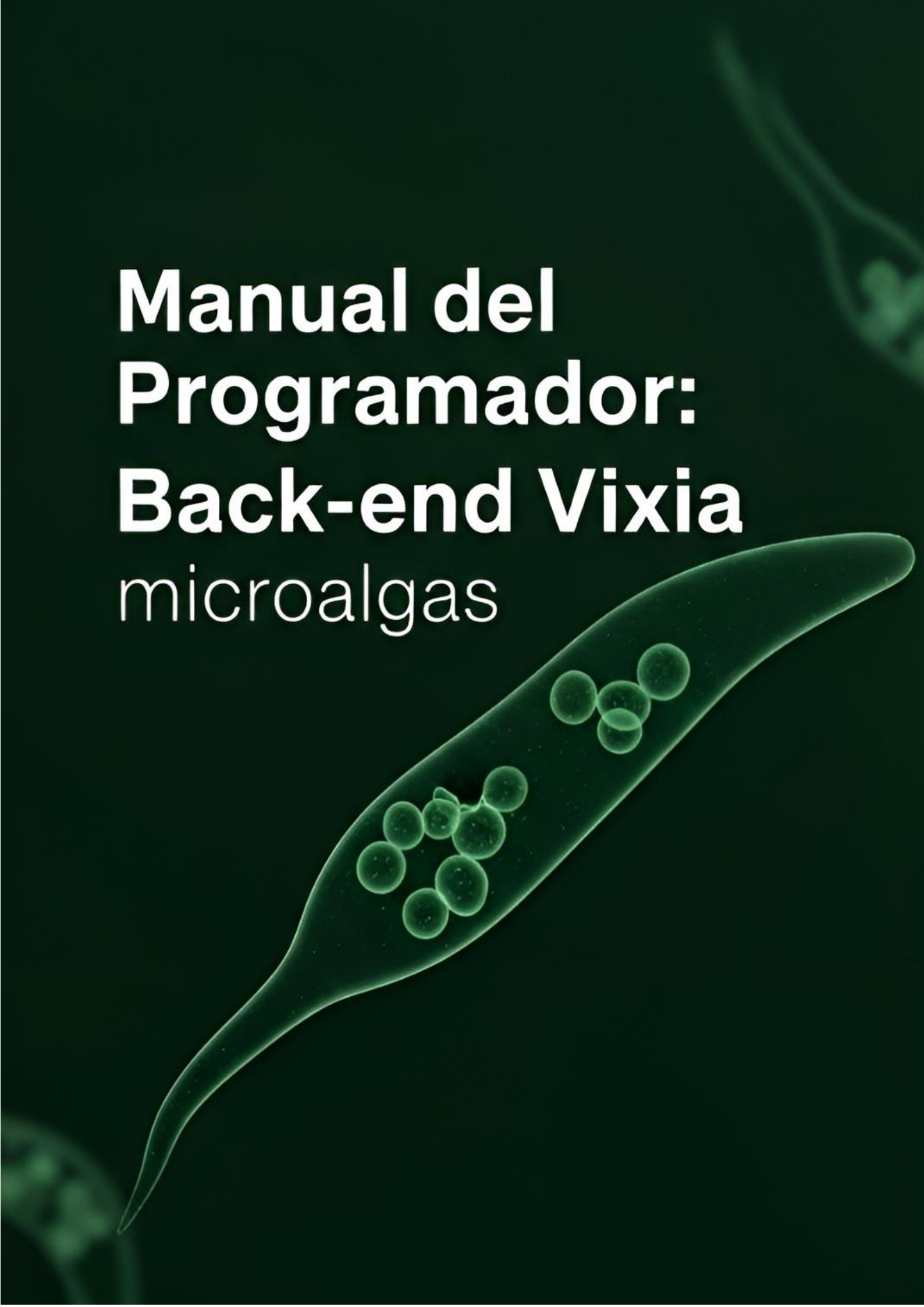
Manual del Programador: Back-end Vixia microalgas



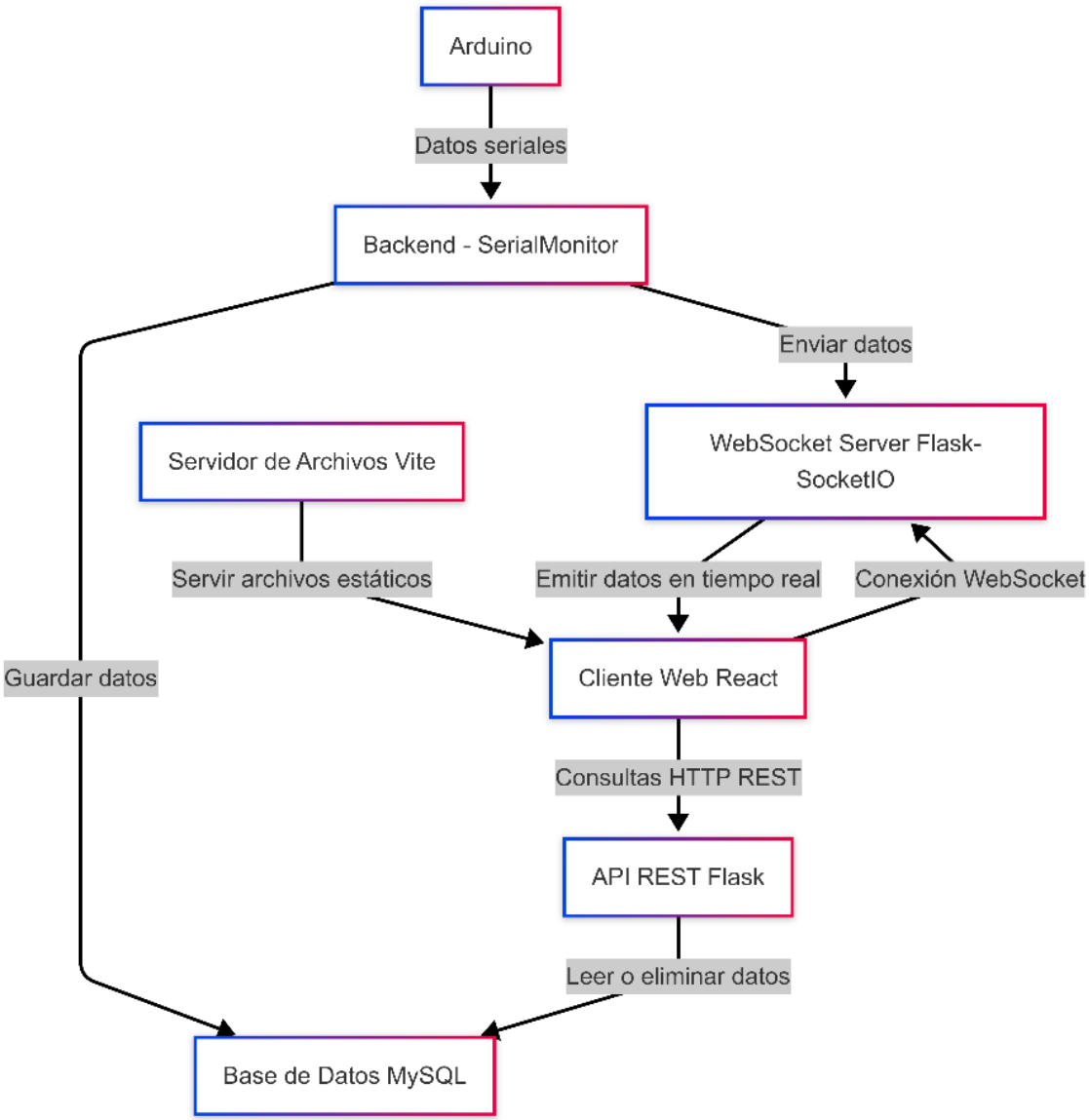
Contenido

1. Introducción al Proyecto

1.1. Visión General

La API de monitoreo de sensores es una aplicación backend que gestiona la recolección, el procesamiento, el almacenamiento y la transmisión en tiempo real de datos provenientes de un dispositivo Arduino equipado con sensores. Su función principal es recibir datos de espectrómetro, temperatura y pH a través de comunicación serial. Estos datos son luego persistidos en una base de datos MySQL y transmitidos a clientes conectados mediante WebSockets para un monitoreo en vivo. La API también soporta la gestión de la calibración del "blanco" y la realización de mediciones manuales.

Diagrama de Arquitectura Conceptual



1.2. Tecnologías Utilizadas

- **Lenguaje de Programación**: Python (versión recomendada: 3.9+)
- **Framework Web (API REST)**: Flask
- **Librería WebSockets**: Flask-SocketIO
- **Base de Datos**: MySQL
- **ORM/Conexión BD**: Módulo Flask SQLAlchemy
- **Comunicación Serial**: Pyserial
- **Manejo de Datos**: pandas
- **Variables de Entorno**: Python-dotenv
- **Concurrencia**: Threading, Asyncio
- **Control de Versiones**: Git

2. Configuración del Entorno de Desarrollo

Esta sección detalla los pasos para configurar un entorno de desarrollo funcional para la API.

2.1. Requisitos Previos

Antes de clonar el repositorio y configurar el proyecto, asegúrese de tener instalados los siguientes componentes:

- **Sistema Operativo**: Compatible con Python y MySQL (Windows, macOS, Linux).
- **Git**: Herramienta de control de versiones.
    - **Descarga e instalación**: https://git-scm.com/downloads
    - **Verificación**: git --version
- **Python**: Versión 3.9 o superior.
    - **Descarga e instalación**: https://www.python.org/downloads/
    - **Verificación**: python --version o python3 --version
- **MySQL Server**: Servidor de base de datos MySQL.
    - **Descarga e instalación**: https://mariadb.org/download
        - Durante la instalación, configure un usuario (ej. root o uno específico para la API) y su contraseña.
    - **Verificación del cliente**: mysql --version
- **IDE/Editor de Código**: Se recomienda un editor como Visual Studio Code o PyCharm, con extensiones para Python.
- **Drivers de Arduino/Serial**: Los drivers necesarios para que el sistema operativo reconozca el dispositivo Arduino al conectarlo vía USB.

2.3. Clonación del Repositorio

Abra una terminal y ejecute los siguientes comandos para clonar el código fuente del proyecto:

```
git clone <URL_del_repositorio_aqui>
cd <nombre_del_directorio_del_proyecto>
```

2.3. Creación y Activación del Entorno Virtual

Es fundamental utilizar un entorno virtual para aislar las dependencias del proyecto y evitar conflictos con otras instalaciones de Python.

```
python -m venv
# En Windows:
.\venv\Scripts\activate
# En macOS/Linux:
source venv/bin/activate
```

2.4. Instalación de Dependencias Python

Con el entorno virtual activado, instale todas las librerías necesarias utilizando el archivo requirements.txt:

```
pip install -r requirements.txt
```

2.5. Configuración de Variables de Entorno (.env)

Las variables de entorno deben estar contempladas en un archivo llamado .env en la raíz del directorio del proyecto. Este archivo contendrá configuraciones esenciales y credenciales sensibles. Este archivo no debe ser subido al control de versiones.

```
# Auth config
SALT_KEY=your_salt_key
DB_PATH=bd/users/usersDB.json

# Database config
DB_HOST = localhost
DB_PORT = db_port
DB_NAME = your_bd_name
DB_USER = your_user_name
DB_PASSWORD = your_bd_pass

# Arduino config
VID = "2341"
PID = "0042"
BAUD_RATE = 9600
TIMEOUT = 1
TIME_BETWEEN_MEASURAMENTS = 1
TIME_LIGHT = 1
TIME_DARK = 1
LIGHT_WHITE = 100
LIGHT_RED = 0
LIGHT_BLUE = 0
```

Nota: Reemplace con los valores correspondientes a su configuración local. Los valores VID y PID de su Arduino pueden obtenerse en el Administrador de Dispositivos (Windows) o mediante comandos como *"lsusb –v"* (Linux) o *"ioreg -p IOUSB –l"* (macOS) cuando el Arduino está conectado.

3. Base de Datos MySQL

3.1. Creación de la Base de Datos

Abra un cliente MySQL (ej. la línea de comandos o MySQL Workbench) y ejecute los siguientes comandos para crear la base de datos:

```
CREATE DATABASE IF NOT EXISTS `your_database_name`;
USE `your_database_name`;
```

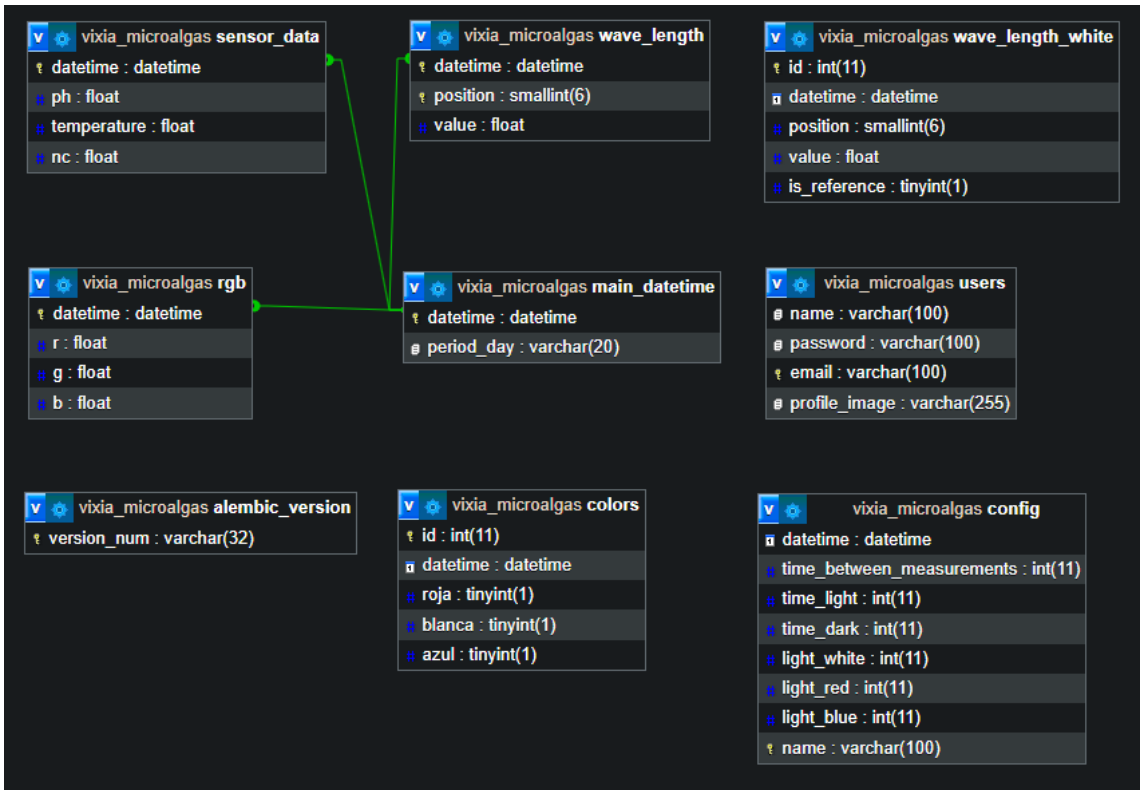
3.2. Estructura de Tablas (Esquema SQL)

La API interactúa con varias tablas para almacenar los datos y configuraciones. Se ha implementado en la base de datos Object Relational Mapping (ORM). A continuación, se presenta el esquema para las tablas principales.

La clase database se encarga de inicializar y conectarse con la base de datos según la configuración.

El módulo **database.queries** maneja las consultas a las tablas.

El módulo **database.models** contempla los modelos a partir de los que el ORM creará las tablas automáticamente al iniciar el proyecto



4. Ejecución del Proyecto

4.1. Conexión del Arduino

El arduino se conecta vía usb al servidor backend y se comunica por serial. Asegúrese de que el firmware correcto esté cargado en el Arduino, diseñado para enviar datos en el formato h<data>,<temp>,<ph>a.

Verifique que el puerto serial sea reconocido por su sistema operativo.

4.2. Arranque de la API Flask

Con el entorno virtual activado, inicie la aplicación Flask:

```
python app.python
```

La API se iniciará por defecto en http://localhost:5000. La clase SerialMonitor se inicializa dentro de la aplicación Flask, por lo que su hilo de monitoreo del puerto serial debería iniciarse automáticamente con la aplicación.

4.3. Verificación de la Comunicación Serial

Una vez que la API esté en funcionamiento, el hilo monitor_serial intentará abrir el puerto y comenzar a leer los datos. Revise la consola de la API para confirmar mensajes como "Puerto X abierto.", "Medición automática..." o "Medición manual...", indicando que la comunicación serial está activa.

5. Uso de la API y Comunicación

Esta sección detalla cómo interactuar con la API a través de sus endpoints REST y la comunicación en tiempo real vía WebSockets.

5.1. Endpoints REST (HTTP)

La API Flask expone varios endpoints para gestionar la autenticación, configuración y consulta de datos.

- **/:** GET - Retorna un mensaje de bienvenida ("The app is on").
- **/file_path**: POST - Recibe una ruta de archivo.
- **/register**: POST - Registra un nuevo usuario. Espera un JSON con los campos "name", "password" y "email".
- **/login**: POST - Inicia sesión para un usuario existente. Espera un JSON con los campos "email" y "password".
- **/check_auth**: GET - Verifica el estado de autenticación. Requiere un JWT válido.
- data_routes.py (Rutas de Datos):
- **/get_hours**: POST - Obtiene las horas de los datos para una fecha específica. Espera datos en formato JSON.
- **/get_comparation**: POST - Obtiene la comparación de datos. Espera datos en formato JSON.
- **/ph**: POST - Obtiene datos de pH.
- **/temp**: POST - Obtiene datos de temperatura.
- **/wake_up**: GET - Activa/desactiva el monitor (probablemente relacionado con las luces).
- **/get_manual**: GET - Dispara una medición manual desde el Arduino.
- **/get_config**: GET - Obtiene la configuración.
- **/change_config**: POST - Cambia la configuración. Espera datos en formato JSON.
- **/get_name**: GET - Obtiene el nombre del proceso.
- **/get_all_names**: GET - Obtiene todos los nombres de los procesos.
- **/proc**: GET - Obtiene el proceso de datos para un nombre específico.
- **/export**: GET - Exporta todos los datos a Excel para una fecha específica.
- **/export/proc**: GET - Exporta los datos de un proceso específico a Excel.
- **/delete**: DELETE - Elimina datos por fecha

5.2. Comunicación WebSockets (SocketIO)

El servidor utiliza Flask-SocketIO para la comunicación en tiempo real con los clientes. Los clientes (frontend) deben conectarse a ws://localhost:5000/socket.io/ (ajuste el host y puerto según la configuración).

Eventos del Lado del Cliente (Frontend) a la API

- **Connect**: Se emite cuando un cliente establece una conexión exitosa con el servidor SocketIO. El servidor lo utiliza para enviar el estado inicial del sistema.
- **Disconnect**: Se emite cuando un cliente cierra su conexión con el servidor SocketIO.

Emisiones del Servidor (Backend) al Cliente (Frontend)

- **Arduino_data**:

Propósito: Envía los últimos datos de medición procesados del Arduino.

Frecuencia: Se emite con cada nueva medición (automática o manual) después de promediar 10 mediciones de oscuridad y 10 de luz.

Datos (JSON):

JSON

```
{
    "datetime": "YYYY-MM-DD HH:MM:SS",
    "colors": null,
    "rgb": {"r": 200, "g": 150, "b": 100},
    "data": {"ph": 7.2, "temperature": 24.5},
    "wave_length": [/* array de flotantes del espectro medido */],
    "x": [/* array de longitudes de onda (WAVELENGTHS) */],
    "nc": 12345
}
```

- **Lights_state:**

Propósito: Envía el estado actual de las luces (roja, azul, blanca) del Arduino.

Frecuencia: Se emite cada vez que el estado de una luz cambia.

Datos (JSON): {"roja": 1, "azul": 0, "blanca": 0} (1 indica encendido, 0 indica apagado).

- **Manual_mode:**

Propósito: Indica si el sistema está esperando o realizando un ciclo de medición manual.

- **Wake_up_state:**

Propósito: Envía el estado de "actividad" del monitor de Arduino. Si es false, la API no estará procesando activamente nuevas mediciones automáticas (esto ocurre, por ejemplo, durante una medición manual).

6. Estructura del Código y Convenciones

6.1. Organización de Directorios

La estructura del proyecto sigue un diseño modular para organizar las diferentes funcionalidades:

```
backend/
├── app.py
├── config.py
├── arduino/
│   ├── SerialMonitor.py
│   ├── monitor_instance.py
│   └── util.py
├── auth/
│   └── auth.py
├── database/
│   ├── db.py
│   ├── db_instance.py
│   ├── executor_instance.py
│   ├── models.py
│   └── queries.py
├── routes/
│   ├── auth_routes.py
│   └── data_routes.py
├── utils/
│   ├── lib.py
│   └── datos_fic_db.py
└── websockets/
    ├── socketio_intance.py
    └── socketio_routes.py
```

6.2. Clase SerialMonitor

Esta clase constituye el núcleo de la interacción con el Arduino. Su propósito es encapsular toda la lógica relacionada con la comunicación serial, incluyendo la lectura de datos, el procesamiento de mensajes, el almacenamiento en base de datos y la emisión de información a través de WebSockets. Cuando este recibe un dato del Arduino, tanto lo almacena en la base de datos como lo emite por WebSockets.

Funcionalidades clave:

- **__init__(self, baud_rate, app, socketio)**: Inicializa el monitor de comunicación, cargando las variables de entorno necesarias (como el VID/PID del Arduino), y configura la conexión serial y el entorno WebSocket.
- **start()**: Abre el puerto serial e inicia un hilo en segundo plano que se encarga de monitorear continuamente la comunicación con el Arduino.
- **stop():** Cierra el puerto serial y detiene de forma segura el hilo de monitoreo.

- **monitor_serial():** Funciona como el bucle principal del sistema, leyendo constantemente los datos del puerto serial, decodificándolos y enviándolos para su procesamiento.
- **messages_handler()**: Se encarga de procesar mensajes de texto específicos enviados por el Arduino, como estados de luces o mensajes relacionados con la calibración inicial.
- **process_buffer()**: Extrae y valida bloques de datos delimitados por los caracteres "h" (inicio) y "a" (fin), asegurando que los mensajes recibidos sean correctos antes de su análisis.
- **handle_arduino_message():** Interpreta los datos extraídos del mensaje (espectro, temperatura y pH), identificando si corresponden a mediciones automáticas o manuales, y agrupándolos según el contexto.
- **save_batch(self, timestamp, tipo):** Calcula los promedios de mediciones realizadas tanto en oscuridad como con luz. Luego, guarda estos datos en la base de datos y emite el evento arduino_data vía WebSocket.
- **save_manual_batch(self, timestamp):** Similar a save_batch, pero diseñada para mediciones manuales. Marca el estado del sistema como inactivo temporalmente mientras se realiza la medición, evitando interferencias.
- **send_command(self, command):** Envía comandos al Arduino a través del puerto serial, incluyendo la activación de una medición manual mediante el comando "M".

6.3. Funciones Auxiliares

- **calculate_nc(wave_length):** Calcula un "número de componentes" utilizando una fórmula específica basada en un valor del espectro medido.
- **calculate_rgb(wave_length, wave_length_white):** Calcula los valores RGB a partir del espectro medido, utilizando un espectro de referencia "blanco" obtenido de la base de datos (get_reference_wavelength_white()).

7. Pruebas

Se recomienda implementar pruebas unitarias y de integración para asegurar la robustez de la API. Si existen, pueden ejecutarse mediante el comando “pytest”.

Es crucial añadir tests para la lógica de procesamiento de mensajes del Arduino, los cálculos de RGB/NC, y las interacciones con la base de datos.

8. Despliegue

Para un entorno de producción, la ejecución de la API requiere consideraciones adicionales:

Servidor WSGI: No utilice flask run o python app.py en producción. Se debe emplear un servidor WSGI (Web Server Gateway Interface) como Gunicorn o uWSGI.

```
pip install gunicorn
gunicorn -w 4 -b 0.0.0.0:5000 app:app
```

WebSockets en Producción: Para escalar los WebSockets en entornos con múltiples instancias o procesos, Flask-SocketIO puede requerir un back-end de mensajes como Redis.

Contenedorización (Docker): Se recomienda encarecidamente contenerizar la aplicación utilizando Docker para asegurar un despliegue consistente y reproducible en diferentes entornos.

9. Futuras Implementaciones y Consideraciones

Esta sección describe posibles mejoras y extensiones para el proyecto:

9.1. Mejoras Potenciales

Autenticación y Autorización: Implementar sistemas más robustos para proteger todos los endpoints de la API.

Validación de Datos: Mejorar la validación de los datos entrantes del Arduino y de las peticiones HTTP para mayor seguridad y estabilidad.

Manejo de Errores: Implementar una estrategia global de manejo de excepciones en Flask para proporcionar respuestas API consistentes y descriptivas.

Optimización de BD: Añadir índices a las tablas de MySQL para mejorar el rendimiento de las consultas, especialmente a medida que crece el volumen de datos.

Visualización de Datos Históricos: Expandir los endpoints y la lógica para consultar y graficar series de datos históricas de la base de datos.

9.2. Problemas Conocidos

Formato de Mensaje Arduino: Cualquier modificación en el firmware del Arduino que altere el formato de los mensajes (h…a) requerirá una actualización correspondiente en las funciones process_buffer y handle_arduino_message de la clase SerialMonitor.

Puerto Serial: Asegúrese de que el puerto serial del Arduino esté libre y no esté siendo utilizado por otra aplicación (ej. el IDE de Arduino) al intentar iniciar la API.

Reconexión Automática: Actualmente, la clase SerialMonitor no implementa una lógica de reconexión automática si la conexión serial con el Arduino se pierde. Considere añadir esta funcionalidad para mejorar la resiliencia.

10. Resolución de Problemas Comunes (Troubleshooting)

¿Por qué aparece el error "Error al abrir el puerto X: [Errno 2] No such file or directory"?

Este error ocurre cuando no se puede establecer la conexión con el Arduino. Posibles causas y soluciones:

- Verifique que el Arduino esté correctamente conectado al puerto USB.
- Asegúrese de que los valores VID y PID en el archivo .env coincidan con los de su dispositivo.
- Compruebe que el puerto serial no esté siendo utilizado por otra aplicación.
- En **Linux/macOS**, puede ser necesario otorgar permisos para acceder al puerto serial.

¿Qué significa "Error al convertir datos a números: ValueError"?

Este error indica que los datos recibidos desde el Arduino no tienen el formato esperado. Posibles causas:

- El Arduino podría estar enviando información incompleta, malformada o corrupta.
- Revise el código del **firmware del Arduino** para asegurarse de que el formato de salida sea correcto.
- Verifique la **calidad de la conexión serial** (puerto suelto, interferencias, velocidad incorrecta, etc.).

La API no arranca por un error de base de datos, ¿qué hago?

Si la API lanza un error relacionado con MySQL, siga estos pasos:

- Asegúrese de que el **servidor MySQL** esté en ejecución.
- Verifique que las variables en el archivo .env (MYSQL_USER, MYSQL_PASSWORD, MYSQL_DB) sean correctas.
- Confirme que el usuario tenga los **permisos adecuados** sobre la base de datos.
- Revise si la base de datos y las tablas existen. Si no, asegúrese de que se creen automáticamente al iniciar la API (según la configuración del proyecto).

¿Por qué los WebSockets no se conectan o no reciben datos?

Si el cliente no recibe datos en tiempo real, puede deberse a:

- La API Flask no está corriendo o **Flask-SocketIO no está bien configurado**.
- La URL del WebSocket en el cliente es incorrecta. Debe tener el siguiente formato
- Consulte la **consola del navegador** en el cliente para ver si hay errores relacionados con la conexión WebSocket.

Las mediciones manuales no se registran o la API parece "colgada", ¿qué revisar?

Esto puede ocurrir si la secuencia de medición manual no se está ejecutando correctamente:

- Verifique que el cliente haya enviado correctamente el **comando "M"** al Arduino.
- Revise los **logs de la API** para confirmar que waiting_for_manual se establece en True.
- Compruebe que se están recibiendo las mediciones en las listas dark_med y light_med.
- Recuerde que se requieren **10 mediciones en oscuridad y 10 en luz** para completar la medición manual y guardarla.