Ecosistema de Tecnologías IoT y Aplicaciones en la Industria

Docente orientador: Esp. Lic. Gustavo Hernán Siciliano

Integrantes:

- Est. Bruno Massacane
- Est. Emiliano Ginarte Delgado
- Est. Maximiliano Calahorra
- Est. Ramiro Xavier Ferreiro

Índice

Introducción

El presente informe describe el desarrollo de un sistema IoT enfocado en la medición y visualización de datos ambientales, específicamente temperatura y humedad. Este proyecto integra hardware, protocolos de comunicación y software moderno para capturar datos en tiempo real, procesarlos y presentarlos de forma accesible a través de una interfaz web.

El objetivo principal es crear una solución funcional que abarque todo el flujo de datos: desde la adquisición con sensores físicos, su transmisión mediante MQTT, procesamiento intermedio con Node-RED, almacenamiento en una base de datos MongoDB a través de una API desarrollada en NestJS, y la visualización final mediante un frontend construido con Next.js.

Este trabajo busca demostrar la capacidad de integrar tecnologías modernas de manera eficiente en un entorno IoT completo, con un enfoque en la escalabilidad, modularidad y facilidad de visualización de la información.

Arquitectura del sistema

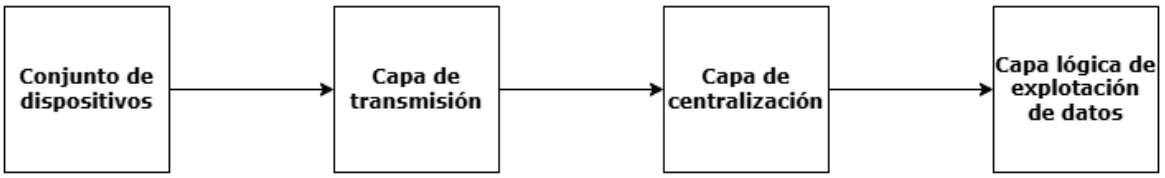

Figura 1: Diagrama de arquitectura simplificado.

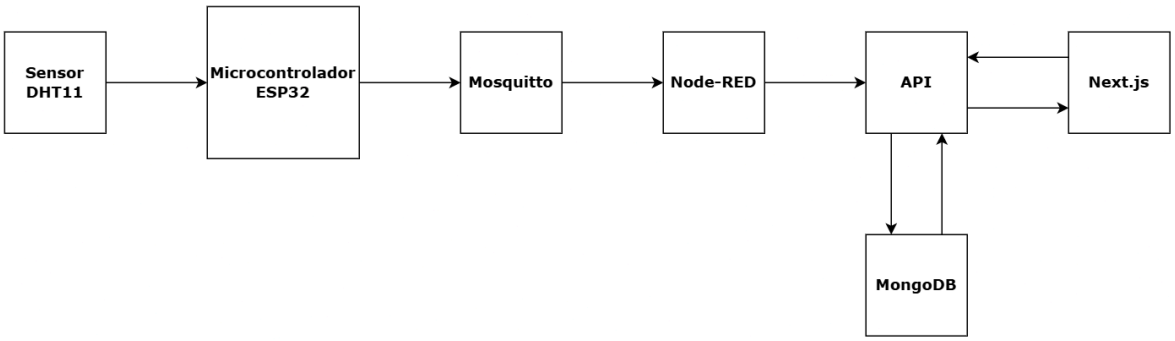

Figura 2: Diagrama de arquitectura detallado.

El sistema implementado sigue una arquitectura orientada al flujo de datos en tiempo real. A continuación, se detalla cada uno de los componentes involucrados:

1. **Sensor físico:** Se emplea un sensor DHT11 para la medición de temperatura y humedad, conectado a un microcontrolador ESP32. El ESP32 realiza lecturas periódicas del sensor a través de su interfaz digital y actúa como nodo publicador dentro del sistema, enviando los datos obtenidos a canales específicos del broker mediante el protocolo MQTT.

2. **Broker MQTT (Mosquitto):** El **broker MQTT**, implementado mediante **Mosquitto**, cumple el rol de **gestor central de mensajes** en el sistema. Su función principal es **recibir los datos publicados en determinados tópicos** y **reenviarlos a todos los clientes suscritos a esos mismos tópicos**.

   Utiliza el protocolo **MQTT**, un estándar de mensajería ligero basado en el modelo **publish/subscribe**, ideal para dispositivos de recursos limitados. La conexión se establece sobre TCP/IP, y en nuestro caso se añade una capa de seguridad mediante **autenticación por usuario y contraseña**, restringiendo el acceso solo a dispositivos autorizados.

3. **Node-RED: Node-RED** actúa como intermediario entre el broker MQTT y la API del sistema. Mediante flujos visuales, se configuran **nodos MQTT de entrada** que se suscriben a tópicos específicos y reciben los datos del microcontrolador en tiempo real.

   Luego, estos mensajes son procesados por **nodos de función**, que transforman el contenido para adaptarlo al formato requerido por la API. Finalmente, se utiliza un **nodo HTTP request** para realizar una solicitud POST con los datos

transformados al **endpoint GraphQL** del backend.

Este enfoque permite desacoplar la recolección de datos del proceso de almacenamiento, facilitando la **escalabilidad** y **modularidad** del sistema.

4. **API Backend**: Desarrollada con NestJS y GraphQL, expone un endpoint para registrar las mediciones, validarlas y guardarlas en una base de datos MongoDB.

5. **Base de Datos (MongoDB)**: Se almacena cada medición con su fecha y hora, temperatura y humedad, permitiendo consultas posteriores.

6. **Frontend (Next.js)**: Una interfaz web permite visualizar en tiempo real la última medición, las últimas 10 entradas en una tabla y sus respectivos promedios, con gráficos para mejorar la comprensión de los datos.

Componentes del proyecto

El proyecto integra distintas tecnologías con funciones específicas que permiten lograr el objetivo anteriormente descrito.

Sensor y Microcontrolador

El componente físico del sistema está compuesto por un módulo sensor DHT11 junto a un microcontrolador ESP32. A continuación, se detalla el funcionamiento y características técnicas de ambos elementos.

Módulo Sensor DHT11

Para este proyecto se utilizó un **módulo DHT11**, el cual incluye el sensor junto con componentes pasivos adicionales montados sobre una pequeña placa. Este diseño modular facilita la conexión al microcontrolador y mejora la estabilidad del sistema.

El módulo cuenta con **tres pines de conexión**:

- **VCC**: alimentación (3.3V o 5V)
- **GND**: tierra
- **DATA**: línea de datos digital

Además, incorpora:

- Una **resistencia pull-up** entre VCC y DATA que asegura un estado lógico alto cuando no hay transmisión, evitando errores en la lectura.
- Un **condensador cerámico SMD** que actúa como filtro de desacoplo, estabilizando la alimentación y minimizando el ruido eléctrico.

Protocolo de Comunicación del DHT11

El sensor DHT11 utiliza un **protocolo digital propietario** de una sola vía para transmitir las lecturas de temperatura y humedad. A diferencia de protocolos estándares como I²C o UART, este protocolo transmite los datos mediante una **secuencia de pulsos temporizados**, que deben ser interpretados correctamente por el microcontrolador.

Para decodificar esta señal, es necesario utilizar una **librería específica** en el firmware del microcontrolador (por ejemplo, `<DHT.h>` en el entorno Arduino), que se encarga de calcular los tiempos entre pulsos y reconstruir los valores de medición.

Microcontrolador ESP32

El encargado de interactuar con el sensor y transmitir los datos es el **ESP32**, un microcontrolador de alto rendimiento con doble núcleo, arquitectura Xtensa LX6 a 240 MHz, y conectividad integrada **Wi-Fi** y **Bluetooth**. Este dispositivo cuenta con:

- RAM y memoria Flash superiores al estándar (típicamente 512 KB de RAM + 4 MB de Flash),
- Entradas analógicas de 12 bits (ADC),
- Capacidad de **multitarea** gracias a su sistema operativo FreeRTOS,
- Varios periféricos digitales (PWM, I2C, SPI, UART, etc.).

Frente a alternativas como el **Arduino Uno**, el ESP32 resulta considerablemente **más potente, más económico y más flexible**, siendo ideal para proyectos que requieren conectividad inalámbrica.

Código del ESP32

El código cargado en el microcontrolador **ESP32** es el encargado de realizar las lecturas del sensor DHT11, conectarse a una red Wi-Fi, y enviar los datos recolectados a un **broker MQTT** para su posterior procesamiento.

Inicialización y configuración

En el `setup()`, se establece la **velocidad del puerto serie a 115200 baudios**, lo cual permite la depuración del sistema y el monitoreo desde un monitor serial.

Conectividad Wi-Fi

Para habilitar la conectividad del ESP32, se hace uso de la **librería `WiFi.h`**, que permite conectarse a una red inalámbrica local especificando:

- El **SSID** (nombre de la red),
- Y la **contraseña de acceso**.

El dispositivo permanece en bucle hasta lograr una conexión exitosa.

Sincronización horaria (NTP)

Una vez conectado a Internet, el ESP32 sincroniza la hora local mediante el protocolo NTP utilizando el servidor "pool.ntp.org". Para el huso horario de Argentina, se establece un desplazamiento de UTC-3.

La función `getLocalTime()` se usa para obtener la hora actual desde el servidor. Esta información se usará posteriormente para incluir una marca temporal precisa en cada publicación de datos.

Configuración del protocolo MQTT

La comunicación con el broker MQTT se realiza utilizando la **librería** `PubSubClient.h`, ampliamente usada para dispositivos IoT debido a su bajo consumo de recursos.

En el código se define:

- La **dirección IP del broker** (local, en este caso),
- El **puerto** (por defecto, 1883),
- El **usuario y contraseña** necesarios para autenticarse en el broker,
- Y el **tópico** o canal al cual el dispositivo se suscribirá y publicará datos.

También se configura un **callback** (función de retorno) que se activa automáticamente cuando se recibe un mensaje en el tópico suscripto.

Lectura del sensor y publicación de datos

Se emplea la **librería** `DHT.h` para poder interpretar correctamente la señal digital del sensor DHT11. Cada cierto intervalo (2 segundos en este caso), el ESP32:

- Lee la **temperatura** y la **humedad** del entorno,
- Calcula el **índice de calor** (heat index) en grados Celsius,
- Convierte estos valores a texto plano (usando `dtostrf` para dar formato flotante),
- Y los **publica en distintos tópicos MQTT** (`humidity` y `temperature`).

Estos valores serán recibidos por el broker y utilizados en etapas posteriores del flujo de datos.

Prototipado

Cabe aclarar que esta implementación corresponde a una **versión de prototipado**, ejecutada sobre una **protoboard** para facilitar el desarrollo. La **fuente de alimentación** del ESP32 es la **PC mediante USB**, aunque el sistema está diseñado para escalar fácilmente a una versión definitiva montada sobre una **PCB (placa de circuito impreso)** y alimentada desde un **cargador de celular convencional**.

MQTT y Broker Mosquitto

¿Qué es MQTT?

MQTT (Message Queuing Telemetry Transport) es un protocolo de mensajería ligero, diseñado específicamente para dispositivos con recursos limitados y redes de baja confiabilidad o ancho de banda reducido. Está basado en el modelo de **publicador/suscriptor**, lo que permite una comunicación asincrónica y desacoplada entre dispositivos.

Su estructura básica involucra tres componentes:

- **Publicador**: Cliente que envía (publica) mensajes a un tópico.
- **Suscriptor**: Cliente que recibe mensajes de un tópico al cual está suscripto.
- **Broker**: Servidor que recibe todos los mensajes de los publicadores, y los redirige a los suscriptores correspondientes.

Funciones principales del broker MQTT

El **broker** es el eje central del sistema MQTT, y cumple funciones críticas como:

- **Gestión de conexiones** de todos los clientes, tanto publicadores como suscriptores.
- **Recepción de mensajes** publicados en los distintos tópicos.
- **Ruteo y distribución de mensajes**, reenviando los datos a todos los clientes suscritos al tópico correspondiente.

Flujo básico de comunicación MQTT

1. **Conexión inicial**: Un cliente se conecta al broker mediante una conexión TCP/IP, y se establece una sesión MQTT.
2. **Suscripción**: El cliente envía una solicitud para suscribirse a uno o varios tópicos.
3. **Publicación**: Un cliente (que puede o no ser el mismo) publica un mensaje en un tópico.
4. **Distribución**: El broker recibe el mensaje y lo distribuye automáticamente a todos los clientes suscriptos a ese tópico.

Calidad de Servicio (QoS)

MQTT permite ajustar la **calidad del servicio (Quality of Service - QoS)** con que los mensajes son entregados:

- **QoS 0**: Entrega "como sea posible". No se garantiza que el mensaje llegue.
- **QoS 1**: El mensaje se entrega al menos una vez, aunque puede duplicarse.
- **QoS 2**: El mensaje se entrega exactamente una vez, sin duplicados.

Esto permite encontrar un equilibrio entre rendimiento y confiabilidad según las necesidades del sistema.

Seguridad en MQTT

El protocolo permite la autenticación mediante:

- **Nombre de usuario y contraseña** (como en nuestro caso),

- Y también admite encriptación utilizando **TLS/SSL** para garantizar confidencialidad y protección ante ataques.

Implementación en el proyecto

En nuestro sistema, se utilizó el **broker MQTT Mosquitto**, una de las implementaciones más populares y livianas, ideal para ambientes de desarrollo y producción.

- Fue desplegado utilizando **Docker**, lo que facilitó una instalación rápida, replicable y aislada del entorno del sistema operativo.
- Se configuró el broker para requerir **autenticación mediante nombre de usuario y contraseña**, reforzando la seguridad del ecosistema IoT.
- Los mensajes son publicados desde el **ESP32** en tópicos específicos (como `temperature` y `humidity`) y son consumidos posteriormente por **Node-RED** para su procesamiento y envío a la API.

Node-RED (Intermediario de Datos)

Node-RED es una herramienta de desarrollo basada en flujos visuales, diseñada para la integración de hardware, APIs y servicios online. En nuestro sistema cumple el rol de **intermediario entre el broker MQTT (Mosquitto) y la API del backend**, permitiendo el procesamiento y redirección de los datos obtenidos por el microcontrolador.

Estructura del flujo en Node-RED

El flujo construido en Node-RED está compuesto por los siguientes nodos principales:

- **Nodo MQTT (input)**:
    - Este nodo se configura como suscriptor, conectado al broker Mosquitto.
    - Se define el host (IP del broker), el puerto (1883 en nuestro caso), y se establece la autenticación mediante **usuario y contraseña**.
    - El nodo se suscribe a tópicos específicos, por ejemplo: `temperature` y `humidity`, que son los mismos utilizados por el ESP32 para publicar los datos.
- **Nodo Function (procesamiento del payload)**:
    - Una vez recibido el mensaje desde el broker, el nodo `function` se encarga de transformar el `payload` del mensaje.
    - Se ajusta el contenido al formato requerido por la **API GraphQL**. Esto incluye:
        - Definir la estructura de la **mutación GraphQL**.

        - Convertir los datos del sensor (humedad y temperatura) a JSON válido.
        - Añadir campos adicionales si es necesario (por ejemplo, timestamp o identificador del dispositivo).
- **Nodo HTTP Request**:
    - Este nodo envía una **solicitud HTTP POST** a la **API** implementada en NestJS, con el payload generado por el nodo `function`.
    - Se configura la cabecera HTTP (`Content-Type: application/json`) y la URL del endpoint del servidor GraphQL.
    - El nodo también permite capturar la respuesta del servidor, lo cual puede ser útil para debugging o registros.
- **Nodo Debug (opcional)**:
    - Utilizado durante el desarrollo para visualizar las salidas de cada nodo.
    - Ayuda a verificar que los datos están siendo recibidos correctamente y enviados con el formato adecuado.

Ventajas del uso de Node-RED en el sistema

- **Facilidad de integración**: Node-RED permite conectar diferentes tecnologías (MQTT, HTTP, bases de datos) sin necesidad de código complejo.
- **Visualización y trazabilidad**: Al ser visual, es sencillo entender el flujo de los datos desde que llegan por MQTT hasta que se almacenan en la API.
- **Extensibilidad**: Es posible añadir fácilmente nuevos sensores, tópicos o condiciones de procesamiento simplemente editando el flujo.
- **Control total sobre el formato de datos**: Gracias al nodo `function`, se tiene control total sobre cómo transformar los datos antes de enviarlos al backend.

Resumen del flujo general

1. **MQTT Input**: Se reciben los mensajes en tiempo real desde los tópicos `temperature` y `humidity`.
2. **Function Node**: Se empaquetan los valores en una estructura GraphQL válida.
3. **HTTP Request**: Se realiza una solicitud POST al servidor NestJS.
4. **(Opcional) Debug**: Se verifica el estado de la operación y se depuran errores si es necesario.

API Backend

El backend del sistema fue implementado en **NestJS**, un framework de Node.js que facilita la construcción de aplicaciones escalables y bien estructuradas. Se utiliza **GraphQL** como motor de consulta, lo que permite al frontend solicitar sólo los datos que necesita.

Para la comunicación con la base de datos **MongoDB**, se utiliza **Mongoose**, una librería de modelado de datos para MongoDB que permite definir esquemas fuertemente tipados y aplicar validaciones a nivel de documento. Se definió un esquema para `Medicion`, el cual especifica:

- `fechaHora`: tipo `Date`, requerido
- `temperatura`: tipo `Number`, requerido
- `humedad`: tipo `Number`, requerido

Los datos son validados automáticamente por Mongoose antes de ser insertados en la base de datos, lo que garantiza consistencia en las mediciones almacenadas.

La API expone resolvers para:

- Consultar la última medición
- Calcular el promedio de temperatura de las últimas N
- Calcular el promedio de humedad de las últimas N
- Obtener una medición con determinada fecha y hora
- Obtener todas las mediciones
- Obtener las últimas N mediciones
- Obtener las mediciones entre un rango de fecha y hora (extremos incluidos)
- Obtener las mediciones entre un rango de temperatura (extremos incluidos)
- Obtener las mediciones entre un rango de humedad (extremos incluidos)
- Insertar una nueva medición
- Eliminar una medición por su fecha y hora

Node.js

¿Qué es?

Un **entorno de ejecución para JavaScript del lado del servidor**, construido sobre el motor V8 de Chrome.

¿Para qué se usa?

- Servidores web, APIs REST, WebSockets.
- Aplicaciones en tiempo real (chat, IoT).
- Scripts de automatización, CLI.

Ventajas

- Asincronía con modelo de eventos no bloqueante.
- Ecosistema enorme (npm).
- JavaScript en backend y frontend (stack unificado).
- Ideal para microservicios y aplicaciones en tiempo real.

Desventajas

- Poco adecuado para procesamiento intensivo en CPU.
- El modelo asincrónico puede complicar el código (aunque Promises y async/await ayudan).
- Callback hell (si no se gestiona bien).

---

npm

¿Qué es?

Gestor de paquetes oficial de Node.js. Permite instalar, compartir y gestionar dependencias. Usa el archivo `package.json` para gestionar dependencias, scripts, versiones, entre otras cuestiones.

¿Para qué se usa?

- Instalar librerías JS (como Express, NestJS, React).
- Ejecutar scripts de desarrollo.
- Publicar tus propios paquetes.

Ventajas

- Repositorio enorme.
- Integración directa con Node.js.
- Scripts personalizados (npm run).

Desventajas

- A veces lento.
- Problemas de seguridad si se usan paquetes sin mantenimiento.
- node_modules puede crecer demasiado.

TypeScript

¿Qué es?

Un **superset de JavaScript con tipado estático** y orientado a objetos. Compila a JavaScript puro para ejecutarse en cualquier entorno JavaScript (como Node.js o navegadores).

¿Para qué se usa?

- Evita errores comunes en JavaScript.
- Tener un código más mantenible, claro y predecible.
- Facilitar el desarrollo a gran escala.

Ventajas

- Tipado estático + autocompletado.
- Mejor experiencia con editores (por ejemplo, VSCode).
- Excelente integración con frameworks modernos.
- Reduce bugs en producción.
- Detección de errores en tiempo de compilación.

Desventajas

- Necesita una fase de compilación.
- Puede ser más verboso que JavaScript.
- Requiere aprendizaje adicional para quienes vienen de JavaScript puro.

NestJS

¿Qué es?

Un **framework de desarrollo backend** para Node.js, basado en TypeScript y principios de programación orientada a objetos y funcional.

¿Para qué se usa?

- Crear APIs REST y GraphQL.
- Backend para aplicaciones web o móviles.
- Integración con bases de datos, microservicios, autenticación, etcétera.
- Proyectos empresariales.

Ventajas

- Inspirado en Angular (modular, inyección de dependencias, decoradores).
- Arquitectura modular y escalable basada en controladores, servicios y proveedores.
- Integración fluida con TypeScript.
- Soporta REST, GraphQL, MQTT, WebSockets, microservicios, y más.
- Inyección de dependencias y testing incluidos.
- Ideal para construir APIs robustas, escalables y mantenibles.

Desventajas

- Curva de aprendizaje más alta para quienes vienen de Express o JavaScript puro.
- A veces es demasiado “estructurado” para proyectos simples.
- Abstracción compleja en casos muy personalizados.

GraphQL

¿Qué es?

GraphQL es un lenguaje de consulta, un estilo de arquitectura y un conjunto de herramientas para crear y manipular las API. A diferencia de las APIs REST, donde se deben desarrollar varios endpoints que devuelven datos específicos, GraphQL permite a los clientes pedir exactamente los datos que necesitan en una única solicitud.

Algunos conceptos claves de esta tecnología son:

1. **Esquema**: Define los tipos de datos y las relaciones entre ellos en la API. Es como el contrato entre el cliente y el servidor.

2. **Consultas (Queries)**: Son solicitudes para leer datos desde el servidor. Con GraphQL, podés especificar qué campos de datos querés para evitar recibir más o menos información de la que necesitás.
3. **Mutaciones (Mutations)**: Se utilizan para modificar datos en el servidor, como crear, actualizar o eliminar registros.
4. **Suscripciones (Subscriptions)**: Permiten a los clientes suscribirse a eventos en tiempo real en el servidor. Por ejemplo, podés usar una suscripción para recibir actualizaciones en tiempo real cuando cambien ciertos datos.
5. **Resolvers**: Son funciones en el servidor que se encargan de obtener y devolver los datos para cada campo de una consulta o mutación.

¿Para qué se usa?

Al igual que REST, es un estilo de arquitectura de API (Application Programming Interface) que permite el intercambio de datos entre diferentes servicios o aplicaciones en un modelo cliente-servidor.

Las APIs facilitan el acceso a los datos y las operaciones de datos de la siguiente manera:

1. Un cliente envía una solicitud de API a uno o varios puntos de conexión de un servidor.
2. El servidor proporciona una respuesta que contiene datos, el estado de los datos o códigos de error.

Esta arquitectura permite crear, modificar, actualizar y eliminar datos en una aplicación, servicio o módulo independiente a través de la API.

Los equipos de frontend y backend utilizan estas arquitecturas de API para crear aplicaciones modulares y accesibles. El uso de una arquitectura API ayuda a mantener la seguridad, la modularidad y la escalabilidad de los sistemas. También, los hace más eficientes y fáciles de integrar con otros sistemas.

Es una buena estrategia de arquitectura cuando contamos con alguna de las siguientes condiciones:

- Ancho de banda limitado y deseo de minimizar la cantidad de solicitudes y respuestas.
- Varios orígenes de datos y deseo de combinarlos en un punto de conexión.
- Las solicitudes de los clientes varían significativamente y se esperan respuestas muy diferentes.

Ventajas

- **Flexibilidad y eficiencia**: Permite a los clientes solicitar exactamente los datos que necesitan, lo que reduce el consumo de ancho de banda y mejora la eficiencia de las aplicaciones, especialmente en entornos móviles y con redes de baja latencia.

- **Menos endpoints**: A diferencia de las APIs REST, donde diferentes endpoints pueden devolver diferentes conjuntos de datos, con GraphQL se puede acceder a todos los datos necesarios a través de un único endpoint, simplificando la estructura de la API.
- **Fuerte tipado**: El esquema en GraphQL proporciona un fuerte tipado, lo que facilita el desarrollo, depuración y mantenimiento de las APIs, al tiempo que mejora la experiencia de desarrollo. Gracias a este nivel de detalle en los esquemas, el sistema puede identificar automáticamente los errores de las solicitudes y proporcionar mensajes de error útiles.
- **Ecosistema creciente**: Hay una gran cantidad de herramientas, bibliotecas y servicios que soportan GraphQL, incluyendo clientes populares como Apollo y Relay, lo que facilita su adopción en proyectos nuevos y existentes.
- **Resultados en formato JSON**: Es el formato de intercambio de datos más popular que todos los lenguajes, plataformas y sistemas entienden.
- **Neutralidad del lenguaje y la base de datos**: Esto genera que tenga un alto nivel de interoperabilidad con cualquier aplicación.

Desventajas

- **Complejidad en el backend**: Implementar un servidor GraphQL puede ser más complejo que una API REST tradicional, especialmente si se necesita manejar consultas anidadas o resolver datos provenientes de múltiples fuentes. Los resolvers, en particular, pueden volverse complicados de manejar.
- **Over-fetching de datos**: Aunque GraphQL está diseñado para evitar solicitar más datos de los necesarios, en ciertas situaciones, una consulta mal diseñada o muy amplia puede generar un impacto negativo en el rendimiento del servidor, ya que puede pedir muchos datos a la vez.
- **Problemas de seguridad**: Permite a los clientes definir sus propias consultas, lo que puede abrir la puerta a problemas de seguridad como el abuso de la API con consultas demasiado complejas o costosas. Los desarrolladores deben implementar medidas para limitar la profundidad y la complejidad de las consultas, como la paginación y la limitación de la cantidad de elementos solicitados.
- **Nuevas herramientas y aprendizaje**: Migrar de REST a GraphQL puede requerir que los desarrolladores aprendan nuevas herramientas y paradigmas, lo cual puede incrementar la curva de aprendizaje y complicar la transición, especialmente en equipos que ya están familiarizados con REST.
- **Cacheo más complicado**: En REST, el cacheo a nivel de HTTP es más sencillo porque los endpoints suelen ser estáticos. Con GraphQL, dado que todas las consultas se realizan a través de un único endpoint y las respuestas pueden variar ampliamente según la consulta, implementar un cacheo eficiente es más complejo.
- **Sobrecarga en el cliente**: Al darle al cliente la responsabilidad de definir las consultas, puede haber una sobrecarga en términos de lógica y procesamiento en el lado del cliente, especialmente si no gestiona adecuadamente la construcción de consultas.
- **Fragmentación del ecosistema**: Aunque el ecosistema de GraphQL ha crecido, aún existe una fragmentación en términos de herramientas y convenciones, lo que

puede dificultar la elección de la mejor herramienta o biblioteca para un proyecto determinado.

Apollo Server

¿Qué es?

Es una biblioteca open-source para crear **servidores GraphQL** en Node.js. Permite construir un **endpoint GraphQL** fácilmente, integrándose con frameworks como Express, Koa, Fastify o NestJS.

¿Para qué se usa?

- Construir y exponer un **API GraphQL**.
- Recibir consultas y mutaciones desde clientes.
- Unificar múltiples fuentes de datos en un único esquema GraphQL.

Ventajas

- Configuración rápida y simple.
- Compatible con middleware de Node.js.
- Integra herramientas como **Apollo Studio, GraphQL Playground, tracing**, etcétera.
- Buen soporte para **modularización del esquema** y resolvers.
- Permite aplicar lógica avanzada (auth, catching, contextos, etcétera).

Desventajas

- Puede ser complejo escalar en arquitecturas grandes sin planificación.
- Necesita configuración extra para autenticación y control de errores avanzados.
- Algunas funciones avanzadas (monitoring, analytics) requieren **Apollo GraphOS** (plan gratuito con límites).
- Si se usa con REST legacy o microservicios, puede necesitar adaptadores o capas intermedias.

Mongoose

¿Qué es?

Una **librería ODM** (Object Data Modeling) para trabajar con MongoDB desde Node.js usando esquemas definidos.

¿Para qué se usa?

- Modelar datos en MongoDB como si fueran objetos.
- Validar, sanitizar, y establecer relaciones entre documentos.
- Crear middleware antes/después de ciertas operaciones.

Ventajas

- Organización clara de los datos.
- Validaciones integradas.
- Middleware poderoso (pre/post hooks).
- Acceso a una amplia variedad de métodos predefinidos para operar la base de datos.

Desventajas

- Agrega una capa de abstracción (menos control directo sobre MongoDB).
- A veces puede complicar consultas avanzadas.
- Poco flexible si necesitás un esquema muy dinámico.

Ejemplo de esquema (en TypeScript)

```
const SensorSchema = new mongoose.Schema({
  temperatura: Number,
  humedad: Number,
  fecha: Date
});
```

Figura 3: Ejemplo de esquema Mongoose.

Base de Datos

Se eligió **MongoDB** como sistema de almacenamiento debido a su flexibilidad para manejar estructuras dinámicas de datos y su compatibilidad con JSON.

Cada documento insertado representa una medición, con los siguientes campos:

- `fechaHora`: Timestamp ISO

- `temperatura`: Float (°C)
- `humedad`: Entero (%)

Las consultas a esta base permiten obtener tanto datos históricos como métricas agregadas, como promedios.

MongoDB

¿Qué es?

Es una base de datos **NoSQL orientada a documentos**, que guarda datos en formato BSON (JSON binario).

¿Para qué se usa?

- Almacenar datos semi-estructurados (JSON).
- Aplicaciones que necesitan alta disponibilidad y escalabilidad.
- Proyectos ágiles o con datos que cambian frecuentemente o en tiempo real, como en IoT.

Ventajas

- No requiere esquema fijo como las bases de datos relacionales, por lo que es muy flexible.
- Escalable horizontalmente (sharding), ideal para grandes volúmenes de datos.
- Fácil de usar para desarrolladores.

Desventajas

- No tan fuerte en transacciones complejas (aunque mejoró mucho).
- Rendimiento inferior a bases relacionales en ciertos casos.
- La desnormalización puede llevar a duplicidad de datos.

MongoDB Atlas

- **Plataforma en la nube** oficial de MongoDB.
- Provee instancias seguras de MongoDB en minutos administradas por AWS (Amazon Web Services), GCP (Google Cloud Computing) o Azure.
- Incluye herramientas para backups, monitoreo, escalabilidad y seguridad.

Frontend

El frontend fue desarrollado con **Next.js**, un framework basado en React que permite tanto renderizado del lado del cliente como del servidor. La aplicación consulta a la API GraphQL para obtener datos de temperatura y humedad en tiempo real.

Las principales funcionalidades de la interfaz incluyen:

- Visualización de la **última medición** usando un **gauge o gráfico tipo velocímetro**, gracias a la librería `react-gauge-chart`, que proporciona una visualización clara e inmediata del valor actual.

- **Tabla** con las últimas 10 mediciones, incluyendo fecha, hora, temperatura y humedad.

- **Promedios** calculados a partir de esas mediciones, presentados en forma textual.

El diseño busca ofrecer una experiencia visual amigable e informativa, orientada a usuarios que necesiten monitorear las condiciones ambientales de manera rápida. Por lo tanto, la interfaz no solo es capaz de presentar la información, sino también de actualizarse periódicamente con la intención de reflejar las nuevas mediciones registradas por el sensor.

yarn

¿Qué es?

Un **gestor de paquetes alternativo a npm**, para solucionar problemas como velocidad, seguridad y determinismo.

¿Para qué se usa?

- Instalar dependencias más rápido y con mayor control.
- Alternativamente más estable y predecible frente a npm.

Ventajas

- Instalación paralela, por lo tanto, más rápida.
- `yarn.lock` asegura versiones exactas.
- Mejores mensajes de error y scripts más potentes.

Desventajas

- Incompatibilidades ocasionales con proyectos hechos con npm.
- Duplicación de herramientas (si ya usas npm bien configurado).
- Algunos paquetes pueden comportarse diferente por diferencias internas.

React

¿Qué es?

Una **librería de JavaScript para construir interfaces de usuario**.

¿Para qué se usa?

- Construir aplicaciones web SPA (Single Page Applications).
- Crear componentes reutilizables e interactivos.
- Gestionar la UI (User Interface) basada en estado y props.

Ventajas

- Reutilización de componentes.
- Virtual DOM para rendimiento eficiente.
- Comunidad y ecosistema enormes.
- Integración con librerías modernas (Redux, Tailwind, etcétera).

Desventajas

- Solo la “V” de MVC (Modelo-Vista-Controlador). Son necesarias otras librerías para routing, estado, etcétera.
- Curva de aprendizaje con hooks y state management.
- Re-renderización puede causar errores si no se gestiona bien el estado.

---

react-gauge-chart

¿Qué es?

Una **librería de visualización en React** para crear **medidores tipo gauge** (indicadores semicirculares).

¿Para qué se usa?

- Mostrar valores como velocidad, temperatura, batería, de sensores (IoT), rendimiento, porcentaje de uso, entre otras cosas, en formato visual.
- Dashboards e interfaces visuales.

Ventajas

- Fácil de integrar en React.
- Personalización de colores, niveles y animación.
- Ligera y visualmente atractiva.

Desventajas

- No tan flexible como usar D3 directamente. Está basado en él.
- Estilo visual limitado a gauge.
- Pocas actualizaciones recientes (dependiendo del mantenedor).

Next.js

¿Qué es?

Un **framework de React** para producción que ofrece funcionalidades como SSR (Server-Side Rendering), SSG (Static Site Generation) y routing automático.

¿Para qué se usa?

- Aplicaciones web modernas con buen rendimiento.
- Web apps optimizadas para SEO (Searching Engineering Optimization).
- Sitios híbridos (páginas estáticas + dinámicas).
- Landing pages.
- Dashboards.
- Aplicaciones web completas con autenticación y consumo de APIs.

Ventajas

- SSR y SSG listos para usar.
- Rutas automáticas basadas en archivos.
- Integración fácil con APIs (páginas `/api/`).
- Ideal para sitios rápidos, SEO-friendly y con experiencia fluida.

Desventajas

- Más complejo que React puro para proyectos simples.
- Algunas abstracciones son difíciles de personalizar.
- Gestión de datos (data fetching) puede variar entre versiones y modos.

Tabla resumen de tecnologías utilizadas

Tecnología	¿Qué es?	¿Para qué se usa?	Ventajas	Desventajas
Sensor DHT11	Sensor digital de temperatura y humedad	Medir variables ambientales en proyectos electrónicos	Bajo costo, fácil de usar, integración directa con microcontroladores	Precisión y resolución limitadas, tiempos de respuesta lentos
Microcontrolador ESP32	MCU de alto rendimiento con Wi-Fi y Bluetooth integrados	Ejecutar lógica embebida, conectarse a redes, leer sensores y comunicarse con otros sistemas	Potente, económico, ideal para IoT, conectividad nativa, multitarea con FreeRTOS	Necesita conocimientos de programación embebida, consumo energético más alto que microcontroladores básicos
MQTT	Protocolo de mensajería ligero, basado en publicador/suscriptor	Comunicación entre dispositivos IoT y servidores	Ligero, ideal para redes inestables o de bajo ancho de banda, QoS configurables	Requiere infraestructura (broker), no incluye encriptación por defecto
Mosquitto	Broker MQTT open-source	Recibir y reenviar mensajes MQTT entre dispositivos y sistemas	Ligero, rápido, muy usado en entornos IoT, fácil de configurar con Docker	Gestión básica de usuarios por defecto, se necesita configurar seguridad manualmente
Node-RED	Plataforma visual basada en flujos para orquestación de datos	Intermediar entre dispositivos IoT, APIs y servicios	Desarrollo sin código complejo, integración rápida, visualización del flujo	Rendimiento limitado en sistemas muy grandes, depende del diseño de flujos para escalar
Node.js	Entorno de ejecución JS del lado del servidor	Ejecutar lógica de backend, crear APIs, manejar conexiones en tiempo real	Rápido, asincrónico, gran comunidad, unifica JS en todo el stack	No ideal para procesos intensivos en CPU
npm	Gestor de paquetes para Node.js	Instalar y gestionar dependencias de JS/TS	Enorme ecosistema de librerías, integración directa con Node.js	node_modules pesado, posibles conflictos entre paquetes
TypeScript	Superset de JavaScript con tipado estático	Evitar errores en tiempo de desarrollo y facilitar el mantenimiento	Tipado fuerte, autocompletado, más robustez	Requiere compilación, más complejo que JS puro
NestJS	Framework backend modular en TypeScript	Crear APIs REST/GraphQL escalables y mantenibles	Arquitectura robusta, integración con múltiples tecnologías, ideal para grandes proyectos	Curva de aprendizaje elevada
GraphQL	Lenguaje de consulta para APIs que permite pedir exactamente los	Para construir APIs flexibles, eficientes y fuertemente tipadas entre cliente y	Solicitudes precisas, menos endpoints, fuerte tipado, buen soporte de herramientas.	Mayor complejidad, riesgo de abusos, cacheo difícil, curva de aprendizaje elevada.

	datos necesarios mediante un único endpoint.	servidor.		
Apollo Server	Librería para crear APIs GraphQL en Node.js	Crear endpoints GraphQL, resolver queries y mutaciones	Simple, extensible, herramientas integradas (Playground, tracing)	Puede ser complejo escalar sin estructura adecuada
Mongoose	ODM (Object Data Modeling) para MongoDB en Node.js	Modelar datos y realizar validaciones	Esquemas claros, validación integrada, middleware potente	Puede ocultar detalles del funcionamiento interno de MongoDB
MongoDB	Base de datos NoSQL orientada a documentos	Almacenamiento flexible de datos JSON	Escalable, flexible, sin esquema fijo	Consultas complejas menos eficientes
MongoDB Atlas	Plataforma en la nube para MongoDB	Hospedar MongoDB fácilmente y escalar en producción	Backup, escalado automático, monitoreo incluido	Puede tener costos si se escala o se requieren features premium
yarn	Gestor de paquetes alternativo a npm	Instalar y gestionar dependencias JS	Más rápido que npm, manejo mejorado de versiones, resolución de conflictos más predecible	Posibles incompatibilidades con proyectos que usan solo npm
React	Librería JS para construir interfaces de usuario	Crear interfaces reactivas y componentes reutilizables	Virtual DOM, comunidad inmensa, SSR con Next.js	Solo maneja la vista (necesita librerías extra para routing, estado, etc.)
react-gauge-chart	Librería React para gráficas tipo gauge	Mostrar indicadores visuales tipo velocímetro	Visual atractivo, fácil integración	Poca personalización
Next.js	Framework de React para producción	Crear aplicaciones web con SSR, SSG y API routes	SEO friendly, performance optimizada, routing automático	Curva de aprendizaje si venís solo de React

Conclusión

A lo largo de este proyecto se logró desarrollar un sistema completo de monitoreo ambiental basado en tecnologías IoT, capaz de captar, procesar, almacenar y visualizar en tiempo real datos de temperatura y humedad.

La integración de hardware, protocolos de mensajería liviana (MQTT), herramientas de flujo como Node-RED, una API robusta construida con NestJS y GraphQL, y una base de datos no relacional como MongoDB, permitió crear una solución eficiente y escalable. El uso de Mongoose facilitó la validación y persistencia de los datos, asegurando su consistencia desde el backend.

Por otro lado, la interfaz desarrollada en Next.js brinda una experiencia visual clara y dinámica, incluyendo gráficos tipo gauge y tablas actualizadas, lo que permite al usuario obtener una visión inmediata del estado ambiental.

Este trabajo no solo sirvió para aplicar conocimientos técnicos en un entorno real, sino también para comprender de manera profunda el flujo de datos en arquitecturas distribuidas modernas. Además, dejó planteadas posibles líneas de mejora, como la incorporación de alertas inteligentes, múltiples sensores, o análisis históricos más complejos.

En resumen, el proyecto cumplió con los objetivos propuestos y sienta una base sólida para futuras ampliaciones o adaptaciones en otros contextos de monitoreo.