

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN



BÁO CÁO BÀI TẬP LỚN
IoT và Ứng dụng

Theo dõi thông số trong nhà: ánh sáng, nhiệt độ, độ ẩm
và điều khiển các thiết bị

Trần Xuân Đạt - B21DCCN223

Giảng viên hướng dẫn: Nguyễn Quốc Uy

Hà Nội, 09/2024

MỤC LỤC

MỤC LỤC.....	1
CHƯƠNG 1: GIỚI THIỆU ĐỀ TÀI.....	4
1.1 Đặt vấn đề:.....	4
1.2. Mục tiêu bài tập lớn:.....	4
a. NestJS:.....	8
b. ReactJS.....	9
CHƯƠNG 2: GIAO DIỆN DỰ ÁN.....	11
2.1. Thiết kế tổng thể.....	11
2.2. Thiết kế chi tiết.....	12
a. Luồng dữ liệu.....	12
b. Sơ đồ tuần tự.....	13
c. Database.....	14
CHƯƠNG 3. SOURCE CODE.....	15
CHƯƠNG 4: KẾT QUẢ.....	21
CHƯƠNG 5: TÀI LIỆU THAM KHẢO.....	25

CHƯƠNG 1: GIỚI THIỆU ĐỀ TÀI

1.1. Đặt vấn đề

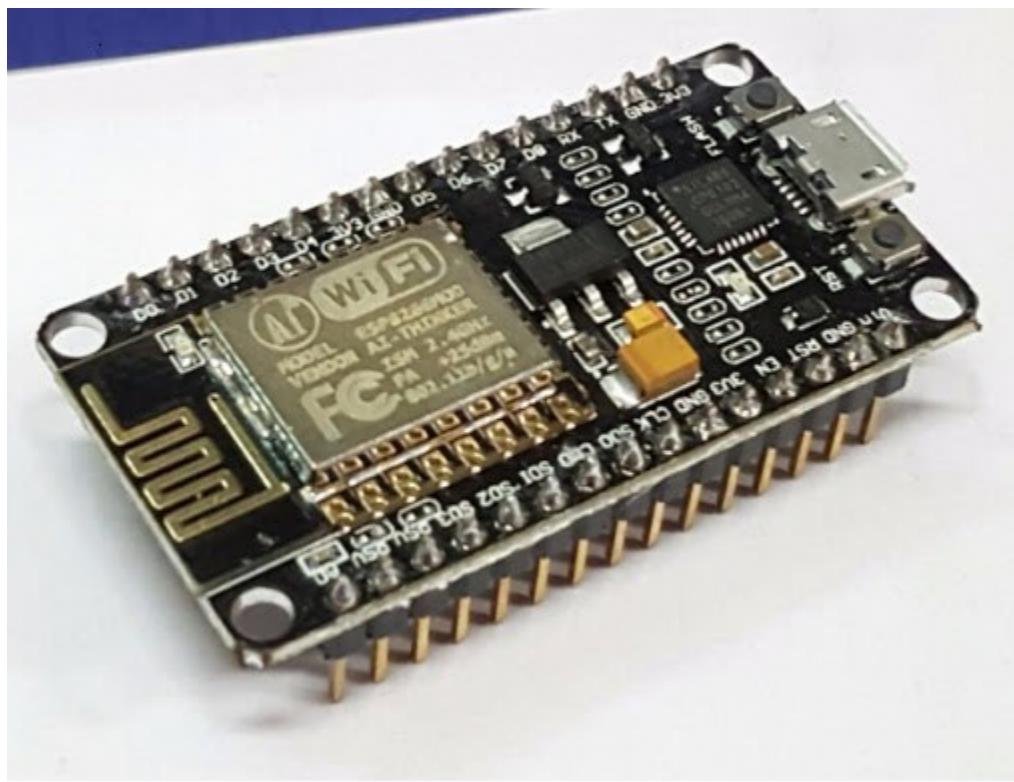
Một hệ thống thời gian thực cho phép truyền dữ liệu từ cảm biến để hiển thị cho người dùng. Người dùng có thể tương tác ngược lại với thiết bị.

1.2. Mục tiêu của bài tập lớn

Hệ thống sử dụng hai cảm biến để thông báo nhiệt độ và độ ẩm, ánh sáng cho người dùng thông qua trang web có thể truy cập với mạng LAN. Trang web sẽ cung cấp nhiệt độ và độ ẩm, ánh sáng với dashboard hiển thị dễ nhìn, cho phép người dùng có thể xem thông tin nhiệt độ và độ ẩm trực tiếp ở thời gian thực và có thể điều khiển led.

1.3. Các thiết bị được sử dụng

a. Kit WiFi NODEMCU ESP8266 CH340



KIT WIFI NODEMCU ESP8266 CH340 là một bảng phát triển dựa trên vi điều khiển ESP8266, tích hợp sẵn module WiFi. Đây là một công cụ tiện lợi dành cho những người yêu thích phát triển các ứng dụng IoT, cho phép kết nối với các mạng WiFi và gửi/nhận dữ liệu qua internet. Phiên bản này sử dụng chip CH340 để giao tiếp với máy tính thông qua cổng USB, giúp dễ dàng nạp code và giao tiếp dữ liệu.

ESP8266 là một vi điều khiển mạnh mẽ với khả năng kết nối mạng WiFi, tích hợp nhiều tính năng tiện lợi như GPIO (các chân vào/ra), SPI, I2C, UART, và ADC, giúp điều khiển các thiết bị ngoại vi như cảm biến, đèn LED, motor, và các loại module khác.

NodeMCU hỗ trợ lập trình bằng nhiều ngôn ngữ khác nhau, bao gồm Arduino IDE, MicroPython, và Lua, giúp lập trình viên dễ dàng tích hợp và triển khai các ứng dụng.

Các thông số kỹ thuật của KIT WIFI NODEMCU ESP8266 CH340:

- Vi xử lý (MCU): ESP8266
- CPU: Lõi Tensilica L106 32-bit RISC, xung nhịp lên tới 80 MHz.
- Bộ nhớ trong: 32 KB cache instruction và 80 KB DRAM.
- Flash: 4 MB.
-

Giao tiếp WiFi:

- Chuẩn WiFi: 802.11 b/g/n.
- Hỗ trợ mạng: AP (Access Point), STA (Station) và AP+STA.
- Bảo mật: WEP, WPA/WPA2.

Giao tiếp với máy tính: USB to UART: Sử dụng chip CH340 để chuyển đổi USB sang UART giúp nạp chương trình và giao tiếp dữ liệu với máy tính qua cổng USB.

Điện áp hoạt động: 3.3V.

Dòng điện:

- Dòng hoạt động: Khoảng 70 mA (có thể cao hơn khi truyền dữ liệu qua WiFi).
- Dòng nghỉ: < 10 µA (deep sleep mode).
- Kích thước: 50mm x 25mm.

Số chân GPIO: 11 chân I/O (có thể dùng làm PWM, I2C, SPI, ADC, UART).

ADC: 1 chân ADC với độ phân giải 10 bit.

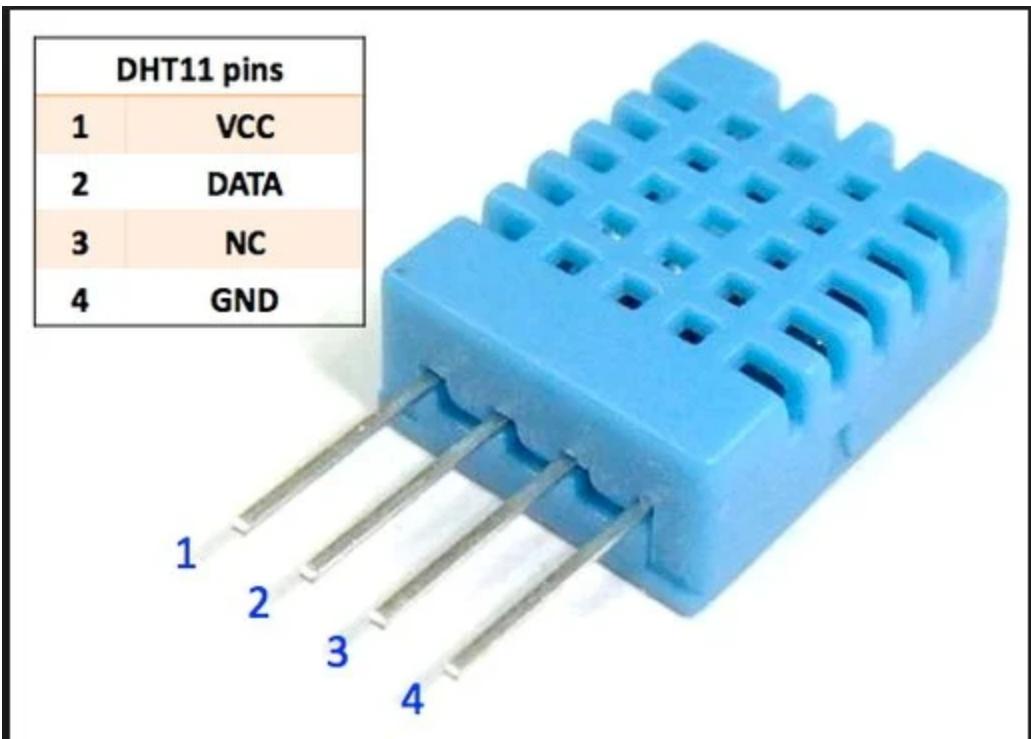
Chế độ tiết kiệm năng lượng: Hỗ trợ các chế độ ngủ (sleep) và deep sleep để tiết kiệm pin.

Giao thức hỗ trợ: UART, SPI, I2C, 1-Wire, PWM.

Lập trình: Hỗ trợ lập trình qua Arduino IDE, Lua, MicroPython.

Cổng kết nối: Giao tiếp với các thiết bị ngoại vi qua các chân GPIO (General Purpose Input Output).

b. Cảm biến nhiệt độ độ ẩm DHT11



DHT11 là một loại cảm biến phổ biến được sử dụng để đo nhiệt độ và độ ẩm trong các dự án điện tử. Đây là cảm biến có giá thành thấp, dễ sử dụng, và được tích hợp bộ chuyển đổi tín hiệu số, giúp việc giao tiếp với vi điều khiển trở nên đơn giản. DHT11 thường được sử dụng trong các ứng dụng như hệ thống nhà thông minh, điều khiển môi trường, và các dự án liên quan đến IoT (Internet of Things).

Các thông số kỹ thuật của DHT11:

- Phạm vi đo nhiệt độ: 0°C đến 50°C, với độ sai lệch $\pm 2^{\circ}\text{C}$.
- Phạm vi đo độ ẩm: 20% đến 90% RH (độ ẩm tương đối), với độ sai lệch $\pm 5\%$ RH.

Độ phân giải:

- Nhiệt độ: 1°C.
- Độ ẩm: 1% RH.

Tần số lấy mẫu: Khoảng 1 Hz (cứ mỗi giây lấy một mẫu).

Nguồn điện hoạt động: 3.3V đến 5V.

Dòng tiêu thụ: 0.5 - 2.5 mA khi đo, 100 - 150 μA khi nghỉ.

Giao tiếp: Tín hiệu số qua một chân dữ liệu (single-wire digital interface).

Kích thước: 15.5 mm x 12 mm x 5.5 mm.

Cấu tạo của DHT11:

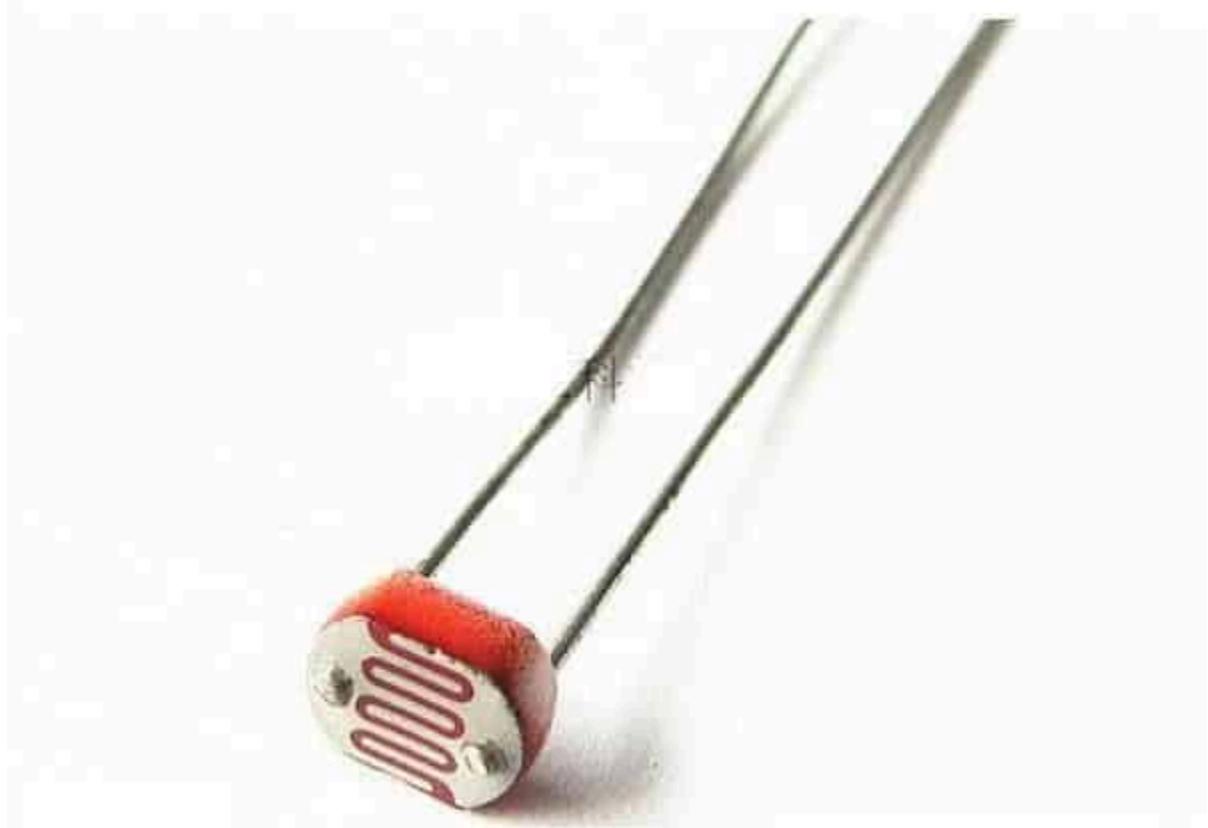
- Cảm biến nhiệt độ: Sử dụng một điện trở nhiệt để đo sự thay đổi của nhiệt độ.
- Cảm biến độ ẩm: Sử dụng một lớp polymer có khả năng hấp thụ và phóng thích độ ẩm từ không khí, làm thay đổi điện dung và từ đó tính toán được độ ẩm tương đối.

Bên cạnh đó, cảm biến DHT11 còn có một bộ vi điều khiển tích hợp để xử lý tín hiệu và chuyển đổi dữ liệu analog thành tín hiệu số, sau đó gửi đến vi điều khiển qua một dây dữ liệu.

Cách kết nối DHT11: Cảm biến DHT11 có 4 chân, trong đó thường sử dụng 3 chân:

- VCC: Chân cấp nguồn (3.3V - 5V).
- DATA: Chân dữ liệu, giao tiếp với vi điều khiển.
- GND: Chân nối đất.
- NC: Không kết nối (Not Connected).

c. Cảm biến quang trở



Cảm biến quang trở (LDR - Light Dependent Resistor) là một loại cảm biến ánh sáng thụ động, hoạt động dựa trên nguyên lý thay đổi điện trở khi cường độ ánh sáng chiếu vào nó thay đổi. Cụ thể:

- Nguyên lý hoạt động: Khi ánh sáng mạnh chiếu vào, điện trở của quang trở giảm xuống. Ngược lại, khi ánh sáng yếu hoặc không có ánh sáng, điện trở của quang trở tăng lên. Điều này cho phép quang trở đo được mức độ cường độ ánh sáng môi trường.
- Ứng dụng: Cảm biến quang trở thường được sử dụng trong các thiết bị như đèn tự động bật/tắt theo ánh sáng môi trường, hệ thống đo cường độ ánh sáng, hoặc các ứng dụng điều khiển ánh sáng dựa vào điều kiện ánh sáng hiện tại.
- Cấu tạo: Quang trở thường được làm từ các chất bán dẫn như cadmium sulfide (CdS). Khi ánh sáng chiếu vào, các electron trong vật liệu bán dẫn được kích thích, làm giảm điện trở của cảm biến.
- Tính chất: Quang trở có đặc tính phi tuyến tính, có nghĩa là sự thay đổi điện trở không tỉ lệ thuận với sự thay đổi của cường độ ánh sáng

d. Led

LED 2 chân (Light Emitting Diode) là một loại diode phát sáng khi có dòng điện đi qua theo chiều thuận. Nó là một linh kiện điện tử phổ biến, được sử dụng rộng rãi trong các thiết bị điện tử để chiếu sáng hoặc làm đèn báo hiệu.

Cấu tạo: 2 chân:

- Chân dương (Anode): Đây là chân dài hơn, được kết nối với cực dương (+) của nguồn điện.
- Chân âm (Cathode): Chân này ngắn hơn và được kết nối với cực âm (-) của nguồn điện.

Nguyên lý hoạt động:

- Khi có dòng điện đi từ Anode sang Cathode (theo chiều thuận của diode), các electron sẽ chuyển động và tái hợp với các lỗ trống trong cấu trúc vật liệu bán dẫn của LED. Quá trình này giải phóng năng lượng dưới dạng ánh sáng.
- LED là một diode nên chỉ phát sáng khi dòng điện đi theo một hướng nhất định (từ chân dương sang chân âm). Nếu kết nối ngược chiều, LED sẽ không sáng.

Đặc điểm:

- Tiêu thụ năng lượng thấp: LED sử dụng ít năng lượng hơn so với các loại đèn khác.
- Tuổi thọ cao: LED có tuổi thọ dài, có thể hoạt động hàng nghìn giờ.
- Kích thước nhỏ gọn: Thường rất nhỏ và dễ dàng tích hợp vào các mạch điện tử.
- Ánh sáng đa dạng: LED có thể phát ra nhiều màu sắc khác nhau (đỏ, xanh lá, xanh dương, trắng, vàng, v.v.) tùy thuộc vào loại vật liệu bán dẫn được sử dụng

1.4. Công nghệ sử dụng

a. NestJS

NestJS là một framework phát triển ứng dụng backend mạnh mẽ và tiên bội, được xây dựng trên nền tảng **Node.js** và sử dụng **TypeScript** làm ngôn ngữ chính. NestJS cung cấp một kiến trúc mô-đun và linh hoạt, giúp việc phát triển các ứng dụng server-side trở nên dễ dàng hơn, đặc biệt với các dự án có tính mở rộng cao.

NestJS kết hợp các yếu tố từ lập trình hướng đối tượng (OOP), lập trình hướng hàm (FP), và lập trình phản ứng (Reactive Programming), giúp các lập trình viên dễ dàng quản lý các ứng dụng lớn, phức tạp với cấu trúc module rõ ràng.

Tính năng chính của NestJS

1. **TypeScript Native:** NestJS sử dụng TypeScript, cho phép lập trình viên tận dụng tính an toàn kiểu dữ liệu và khả năng phát triển hiện đại. Tuy nhiên, NestJS cũng hỗ trợ JavaScript, cho phép lập trình viên chuyển đổi dễ dàng nếu cần.
2. **Kiến trúc mô-đun:** Ứng dụng NestJS được chia thành các module riêng biệt, giúp tổ chức mã nguồn dễ dàng và hỗ trợ quản lý các thành phần khác nhau của ứng dụng một cách độc lập và có thể tái sử dụng.
3. **Hỗ trợ nhiều framework khác:** NestJS dựa trên Node.js và sử dụng các thư viện như **Express** hoặc **Fastify** làm nền tảng HTTP. Lập trình viên có thể dễ dàng chuyển đổi giữa các framework này.
4. **Dependency Injection (DI):** NestJS sử dụng mô hình Dependency Injection, giúp quản lý và khởi tạo các phụ thuộc một cách dễ dàng và tự động. Điều này giúp mã nguồn trở nên sạch sẽ và dễ duy trì hơn.
5. **Khả năng mở rộng:** Với kiến trúc mô-đun, NestJS dễ dàng mở rộng và hỗ trợ thêm các module từ bên ngoài, hoặc tạo ra các module mới cho ứng dụng.
6. **GraphQL và REST:** NestJS hỗ trợ cả hai chuẩn API phổ biến là **REST** và **GraphQL**, giúp các nhà phát triển có thể lựa chọn phù hợp với nhu cầu dự án của mình.
7. **WebSocket và Microservices:** Ngoài việc hỗ trợ các yêu cầu HTTP truyền thống, NestJS còn hỗ trợ **WebSocket** và **Microservices**, cho phép phát triển các ứng dụng thời gian thực hoặc phân tán.
8. **Middleware, Guards, Pipes, Interceptors và Filters:** Các khái niệm này giúp quản lý luồng xử lý yêu cầu trong NestJS, tương tự như các tính năng có trong các framework như Spring hay Angular. Chúng cho phép lập trình viên thao tác với yêu cầu và phản hồi trước và sau khi nó đi qua các thành phần của ứng dụng.

Cấu trúc của NestJS

NestJS tổ chức ứng dụng thành các thành phần chính sau:

- **Module:** Một tập hợp các thành phần liên quan nhau, chẳng hạn như controllers, services, và providers. Mỗi ứng dụng NestJS đều có ít nhất một module, thường là **AppModule**.
- **Controller:** Xử lý các yêu cầu HTTP từ client và gửi phản hồi. Controllers thường giao tiếp với các service để lấy dữ liệu hoặc xử lý logic nghiệp vụ.
- **Service:** Chứa logic nghiệp vụ chính của ứng dụng. Các service thường được sử dụng trong controllers thông qua dependency injection.
- **Providers:** Các service hoặc class giúp tạo ra các đối tượng hoặc xử lý logic mà các thành phần khác cần sử dụng.
- **Middleware:** Xử lý các yêu cầu trước khi chúng được gửi tới controller.
- **Guard, Interceptor, Pipe:** Các cơ chế để bảo vệ, chuyển đổi, hoặc thao tác trên dữ liệu và luồng điều khiển.

b. ReactJS

ReactJS là một thư viện JavaScript mạnh mẽ được phát triển và duy trì bởi Facebook, cho phép xây dựng các giao diện người dùng (UI) một cách hiệu quả. React được thiết kế để giúp việc phát triển các ứng dụng web trở nên dễ dàng hơn, đặc biệt là khi cần xử lý giao diện phức tạp và cập nhật động (dynamic UI).

ReactJS được giới thiệu lần đầu vào năm 2013 và nhanh chóng trở thành một trong những công cụ phổ biến nhất trong cộng đồng phát triển front-end. Nó cho phép lập trình viên tạo ra các ứng dụng web với khả năng tương tác tốt, sử dụng các thành phần (components) có thể tái sử dụng.

Đặc điểm chính của ReactJS:

- Component-Based Architecture (Kiến trúc dựa trên thành phần): React cho phép chia giao diện thành các component độc lập, có thể tái sử dụng. Mỗi component là một khôi UI tự quản lý và có thể bao gồm các thành phần nhỏ hơn. Các component trong React có thể được gói gọn và tái sử dụng ở nhiều nơi trong ứng dụng, giúp tăng tính tổ chức và dễ dàng mở rộng ứng dụng.
- Virtual DOM: React sử dụng một cơ chế gọi là Virtual DOM, giúp cập nhật giao diện nhanh hơn. Thay vì cập nhật trực tiếp DOM thực, React cập nhật Virtual DOM trước, sau đó so sánh sự thay đổi giữa Virtual DOM và DOM thực (quá trình này gọi là reconciliation) và chỉ cập nhật những phần thay đổi, giảm thiểu thao tác trên DOM. Điều này giúp cải thiện hiệu suất của ứng dụng, đặc biệt là với các ứng dụng có giao diện phức tạp và nhiều thay đổi theo thời gian thực.
- JSX (JavaScript XML): JSX là một cú pháp mở rộng của JavaScript, cho phép viết HTML-like trong các file JavaScript. Điều này giúp việc phát triển UI trở nên trực quan hơn, đồng thời tận dụng được sức mạnh của JavaScript để xây dựng các cấu trúc phức

tập. JSX cung cấp khả năng kết hợp HTML với logic JavaScript, giúp code dễ đọc và quản lý hơn.

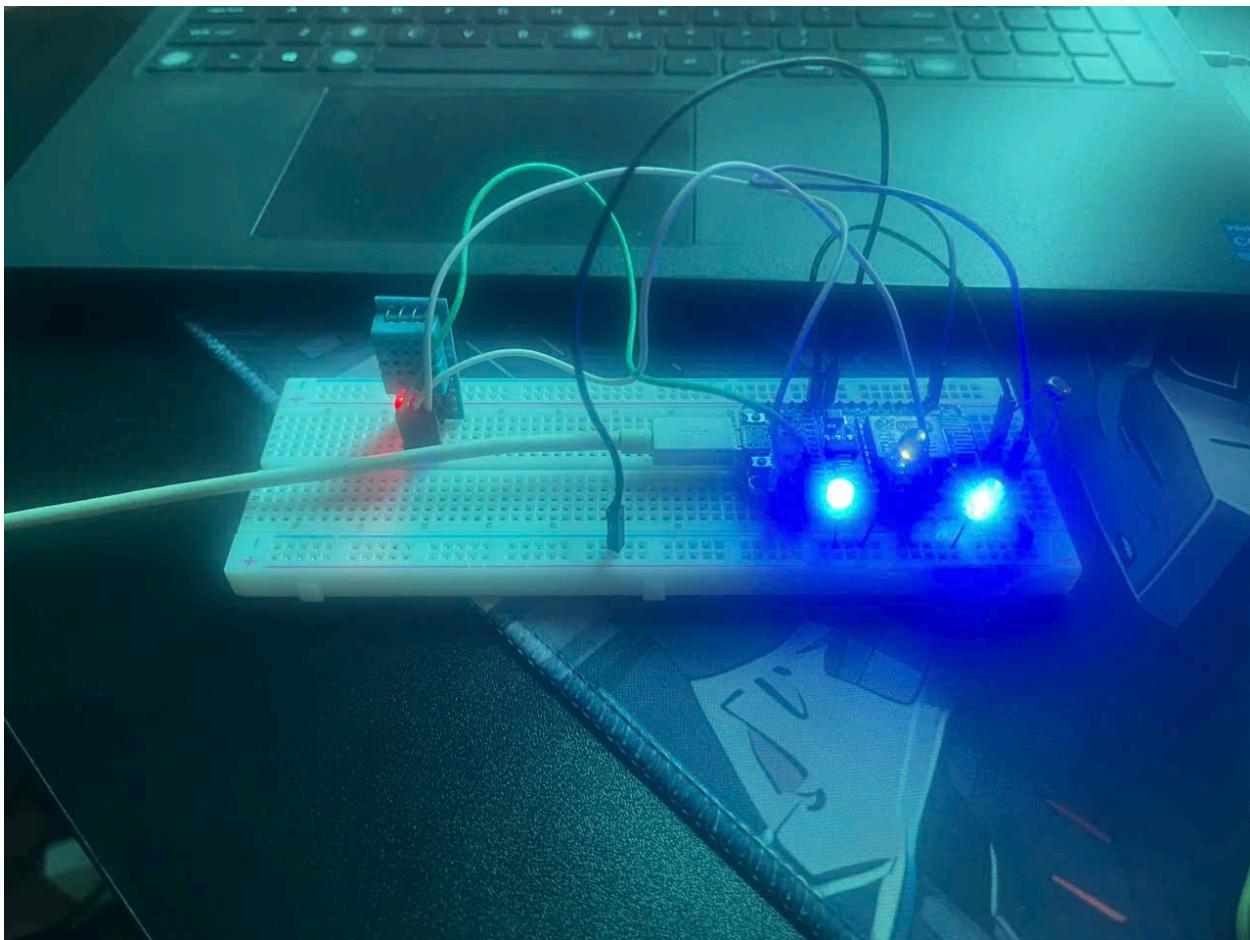
- One-Way Data Binding (Ràng buộc dữ liệu một chiều): React sử dụng one-way data binding, nghĩa là dữ liệu chỉ chảy theo một hướng, từ cha đến con. Điều này giúp việc quản lý dữ liệu trở nên dễ đoán và đơn giản hơn, giảm thiểu lỗi. Các component nhận dữ liệu từ cha thông qua props và có thể quản lý trạng thái riêng của chúng thông qua state.
- State Management (Quản lý trạng thái): State trong React giúp theo dõi trạng thái nội bộ của một component. Khi state thay đổi, React sẽ tự động cập nhật lại giao diện của component tương ứng. Bên cạnh state, React còn hỗ trợ quản lý dữ liệu giữa các component thông qua props, giúp truyền dữ liệu từ component cha sang component con.
- Hooks: Hooks được giới thiệu trong React phiên bản 16.8, cho phép sử dụng state và các tính năng khác của React mà không cần dùng class.
 - Các hook phổ biến bao gồm:
 - useState: Quản lý trạng thái trong functional component.
 - useEffect: Quản lý các side effects như gọi API, đồng bộ hóa dữ liệu, hoặc xử lý hành động sau khi render.
 - useContext, useReducer: Quản lý state toàn cục hoặc phức tạp hơn.

Hooks mang lại sự linh hoạt và giúp code trở nên ngắn gọn hơn khi sử dụng functional component.

- Routing và SPA (Single Page Application): Với React Router, các ứng dụng có thể được xây dựng dưới dạng Single Page Application (SPA), nơi người dùng có thể điều hướng qua nhiều trang mà không cần tải lại toàn bộ trang web. Điều này giúp tăng trải nghiệm người dùng và hiệu suất ứng dụng.
- Community và Ecosystem (Cộng đồng và hệ sinh thái): React có một cộng đồng lớn và mạnh mẽ. Bên cạnh React Router cho điều hướng, còn có Redux hoặc Context API để quản lý state toàn cục, cùng với hàng loạt công cụ hỗ trợ khác giúp phát triển ứng dụng React dễ dàng và nhanh chóng hơn.

CHƯƠNG 2: GIAO DIỆN DỰ ÁN

2.1. Thiết kế tổng thể

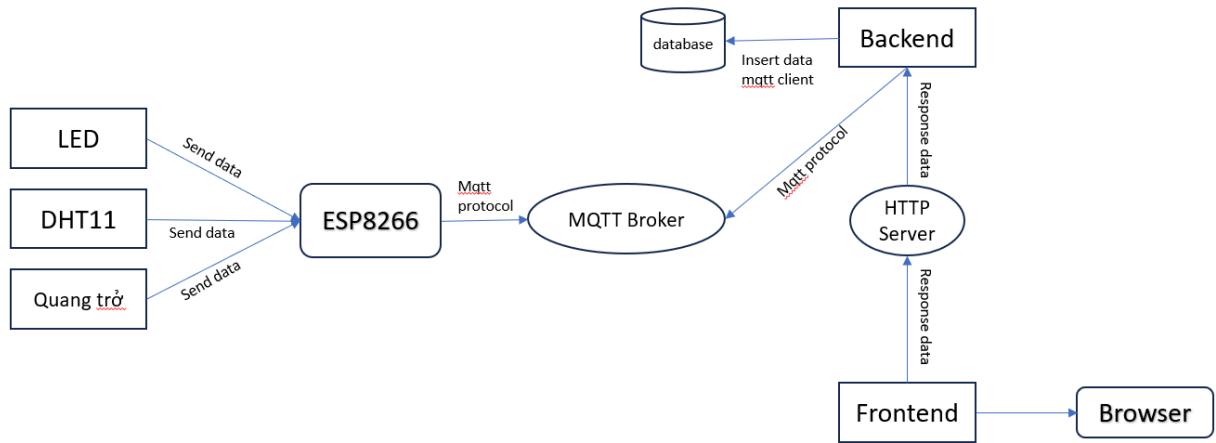


Giao diện Web sẽ gồm 4 trang:

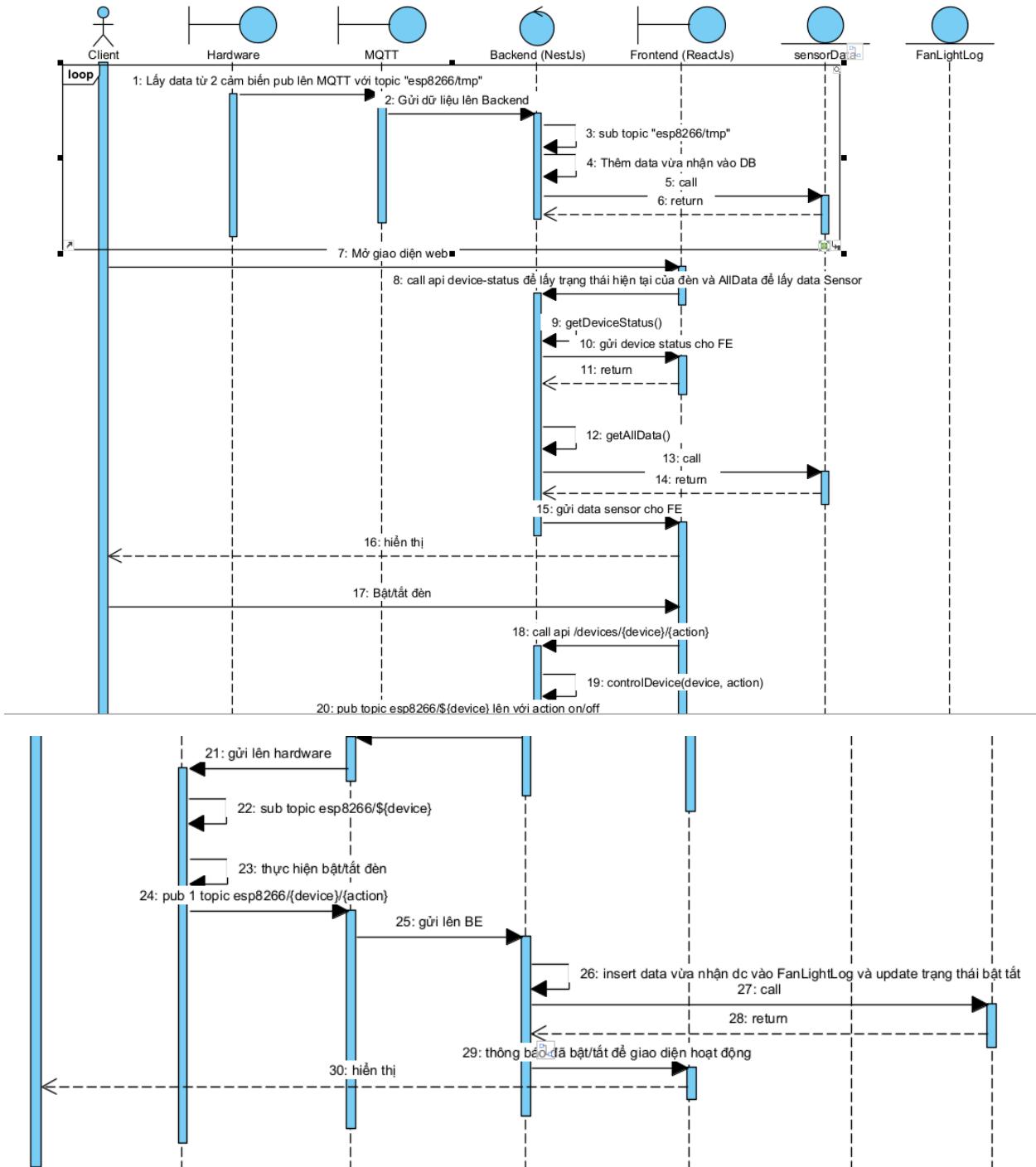
- Home: Gồm biểu đồ của các giá trị thời gian thực, 3 ô hiển thị giá trị thời gian thực và các nút điều khiển thiết bị
- Data Sensor: 1 bảng gồm các cột thông tin về sensor và có thể tìm kiếm sắp xếp theo trường, thứ tự ...
- Action History: 1 bảng lưu trữ các hành động bật/tắt các thiết bị, có thể tìm kiếm và sắp xếp
- Profile: chứa các thông tin cá nhân, link GitHub, báo cáo

2.2. Thiết kế chi tiết

a. Luồng dữ liệu



b. Sơ đồ tuần tự



c. Database

Gồm 2 bảng:

The screenshot shows the database structure for the 'esp8266' database. It includes:

- Tables:** fanlightlog, sensordata
- Views:**
- Stored Procedures:**
- Functions:**

- Fanlightlog: chứa data trạng thái bật tắt của led
- Sensordata: chứa data nhiệt độ, độ ẩm, ánh sáng

Dữ liệu sample của 2 bảng:

- Fanlightlog:

	id	device	state	timestamp
	40	fan	off	2024-09-12 00:37:38
	41	40	off	2024-09-12 00:38:13
	42	led	on	2024-09-12 00:38:25
	43	fan	off	2024-09-12 00:38:36
	44	fan	on	2024-09-12 00:38:41
	45	fan	on	2024-09-12 00:38:48
	46	fan	on	2024-09-12 00:40:19
	47	fan	on	2024-09-12 00:40:44
	48	led	on	2024-09-12 00:41:39
	49	led	on	2024-09-12 00:42:21
	50	led	on	2024-09-12 00:43:07

fanlightlog 1 ×

- Sensordata:

	id	temperature	humidity	light	createdAt
	1	30.8	98	405	2024-09-17 17:41:33
	2	30.8	98	432	2024-09-17 17:41:38
	3	30.8	98	457	2024-09-17 17:41:43
	4	30.8	98	457	2024-09-17 17:41:48
	5	30.8	98	456	2024-09-17 17:41:53
	6	30.8	98	456	2024-09-17 17:41:58
	7	30.8	98	458	2024-09-17 17:42:03
	8	30.8	98	439	2024-09-17 17:42:08
	9	30.8	98	450	2024-09-17 17:42:13
	10	30.8	98	430	2024-09-17 17:42:19
	11	30.8	98	428	2024-09-17 17:42:23

CHƯƠNG 3: SOURCE CODE

Một số các chức năng chính của backend:

- Setup MQTT:

```
You, 1 hour ago | 1 author (You)
1 import { Injectable } from '@nestjs/common';
2 import { Sequelize } from 'sequelize-typescript';
3 import { Op } from 'sequelize';
4 import * as mqtt from 'mqtt';
5 import { SensorData } from '../model/sensor-data.model';
6 import { FanLightLog } from '../model/fan-light-log.model';
7
8 You, 1 hour ago | 1 author (You)
9 @Injectable()
10 export class MqttService {
11   private client;
12   private deviceConnected = false;
13   private deviceStates = {
14     led: 'off', // Mock state for LED
15     fan: 'off', // Mock state for Fan
16     tem: 'off', // Mock state for Tem
17   };
18
19   constructor(private sequelize: Sequelize) {
20     this.client = mqtt.connect('mqtt://broker.emqx.io', {
21       username: 'emqx',
22       password: 'Xuandat1106',
23     });
24
25     this.client.on('connect', () => {
26       console.log('Connected to MQTT broker');
27       // Subscribe to the necessary topics
28       this.client.subscribe('mcu8266/tmp');
29       this.client.subscribe('esp8266/led');
30       this.client.subscribe('esp8266/fan');
31       this.client.subscribe('esp8266/tem');
32       this.client.subscribe('esp8266/status'); // Subscribe to the new status topic
33       this.client.subscribe('device/led/state');
34       this.client.subscribe('device/fan/state');
35       this.client.subscribe('device/tem/state');
36     });
37
38     this.client.publish('esp8266/status', 'disconnected');
39
40     this.client.on('message', (topic, message) => [
41       this.client.publish('esp8266/status', 'connected');
42       if (topic === 'mcu8266/tmp') {
43         this.handleSensorData(message.toString());
44
45       } else if (topic === 'esp8266/led' || topic === 'esp8266/fan' || topic === 'esp8266/tem') {
46         this.logFanLightAction(topic, message.toString());
47
48       } else if (topic === 'esp8266/status') {
49         if (message.toString() === 'connected') {
50           this.deviceConnected = true;
51         } else if (message.toString() === 'disconnected') {
52           this.deviceConnected = false;
53         }
54
55       }
56     ]);
57   }
}
| You, 24 hours ago * add device status and esp status, create contro...
```

- Bật tắt LED:

- ControlDevice():

```

async controlDevice(device: 'led' | 'fan' | 'tem', action: 'on' | 'off') {
  if (!this.deviceConnected) {
    console.log(`Cannot control ${device}. Device is disconnected.`);
    return {
      message: `Device is disconnected. Cannot perform action on ${device}.`,
      success: false,
    };
  }

  const topic = `esp8266/${device}`;
  this.client.publish(topic, action);
  this.deviceStates[device] = action;
  console.log(`Sent command to ${device}: ${action}`);
  return {
    message: `Successfully sent ${action} command to ${device}.`,
    success: true,
  };
}

```

- devices.controller.ts:

```

src > controller > ts devices.controller.ts > DevicesController > controlDevice
You, 5 seconds ago | 1 author (You)
1 import { Controller, Post, Param, Get } from '@nestjs/common';
2 import { MqttService } from '../mqtt/mqtt.service';
3 import { ApiTags,ApiOperation, ApiResponse, ApiParam } from '@nestjs/swagger';
4
5 You, 5 seconds ago | 1 author (You)
6 @ApiTags('devices')
7 @Controller('devices')
8 export class DevicesController {
9   constructor(private readonly mqttService: MqttService) {}
10
11   @ApiOperation({ summary: 'Control device' })
12   @ApiResponse({ status: 200, description: 'Device controlled successfully' })
13   @ApiResponse({ status: 400, description: 'Device is disconnected' })
14   @Post(':device/:action')
15   async controlDevice(
16     @Param('device') device: 'led' | 'fan',
17     @Param('action') action: 'on' | 'off',
18   ) {
19     // Kiểm tra trạng thái kết nối
20     const deviceStatus = await this.mqttService.getDeviceStatus();
21
22     if (deviceStatus.status === 'disconnected') {
23       return {
24         statusCode: 400,
25         message: `Cannot control ${device}. Device is disconnected.`,
26       };
27     }
28
29     // Điều khiển thiết bị nếu đã kết nối
30     const result = await this.mqttService.controlDevice(device, action);
31
32     return {
33       statusCode: 200,
34       message: result.message,
35       success: result.success,
36     };
37   }
38 }

```

- Import dữ liệu hành động bật tắt vào DB:

- logFanLightAction():

```
async logFanLightAction(device: string, state: string) {
  const currentTimestamp = new Date(); // Lấy thời gian hiện tại

  // Kiểm tra giá trị của state để chắc chắn không bị null
  if (!state) {
    throw new Error('State is null or undefined');      You, 2 weeks ago • init api and
  }

  // Lưu dữ liệu vào cơ sở dữ liệu
  await FanLightLog.create({
    device: device.substring(8),
    state: state, // Lưu hành động bật/tắt
    timestamp: currentTimestamp,
  });

  // Trả về phản hồi thành công
  return {
    message: `Device ${device} turned ${state}`,
    device,
    state,
    timestamp: currentTimestamp,
  };
}

async getFanLightLogData(
  page: number,
  limit: number,
): Promise<{ rows: FanLightLog[]; count: number }> {
  try {
    const offset = (page - 1) * limit;
    const { rows, count } = await FanLightLog.findAndCountAll({
      offset,
      limit,
    });
    return { rows, count };
  } catch (error) {
    console.error('Error fetching fan light log data:', error);
    throw error;
  }
}
```

- device-data.service.ts

```
You, 10 hours ago | 1 author (You)
1 import { Injectable } from '@nestjs/common';
2 import { FanLightLog } from '../model/fan-light-log.model'; // Assuming the model path
3 import { Op } from 'sequelize';
4
5 You, 10 hours ago | 1 author (You)
6 @Injectable()
7 export class DeviceDataService {
8   // Service method to get fan and light log data with pagination
9   //get all plain data
10  async getAllPlainData(): Promise<any> {
11    try {
12      return await FanLightLog.findAll();
13    } catch (error) {
14      console.error('Error fetching fan light log data:', error);
15      throw error;
16    }
17  }
18
19  async getFanLightLogData(
20    page: number,
21    limit: number,
22  ): Promise<{ rows: FanLightLog[]; count: number }> {
23    try {
24      const offset = (page - 1) * limit;
25      const { rows, count } = await FanLightLog.findAndCountAll({
26        offset,
27        limit,
28      });
29      return { rows, count };
30    } catch (error) {
31      console.error('Error fetching fan light log data:', error);
32      throw error;
33    }
34  }
35}
```

- Sensor:
 - sensor.controller.ts:

```
You, 10 hours ago | 1 author (You)
1 import { Controller, Get, Query, Param } from '@nestjs/common';
2 import { ApiTags,ApiOperation, ApiResponse, ApiParam } from '@nestjs/swagger';
3 import { SensorDataService } from '../service/sensor-data.service';
4 import { SensorDataResponse } from 'src/response/sensorData.interface';
5 import { SensorData } from 'src/model/sensor-data.model';      You, 10 hours ago * rel
6
7 You, 10 hours ago | 1 author (You)
8 @ApiTags('Sensor')
9 @Controller('SensorData')
10 export class SensorDataController {
11   constructor(private readonly sensorDataService: SensorDataService) {}
12
13   // Get all sensor data with pagination
14   @Get()
15   @ApiOperation({ summary: 'Get all sensor data' })
16   @ApiResponse({ status: 200, description: 'Success' })
17   async getAllSensorData(
18     @Query('page') page: string = '1',
19     @Query('limit') limit: string = '10',
20   ): Promise<SensorDataResponse> {
21     const pageNumber = parseInt(page, 10);
22     const limitNumber = parseInt(limit, 10);
23     const { rows, count } = await this.sensorDataService.paginateSensorData(
24       pageNumber,
25       limitNumber,
26     );
27     return {
28       columns: ['id', 'temperature', 'humidity', 'light', 'timestamp'],
29       rows,
30       count,
31       page: pageNumber,
32       limit: limitNumber,
33     };
34   }
35
36   // get all data
37   @Get('allData')
38   @ApiOperation({ summary: 'Get all sensor data' })
39   @ApiResponse({ status: 200, description: 'Success' })
40   async getAllData(): Promise<SensorData[]> {
41     const rows = await this.sensorDataService.getAllData();
42     return rows;
43   }

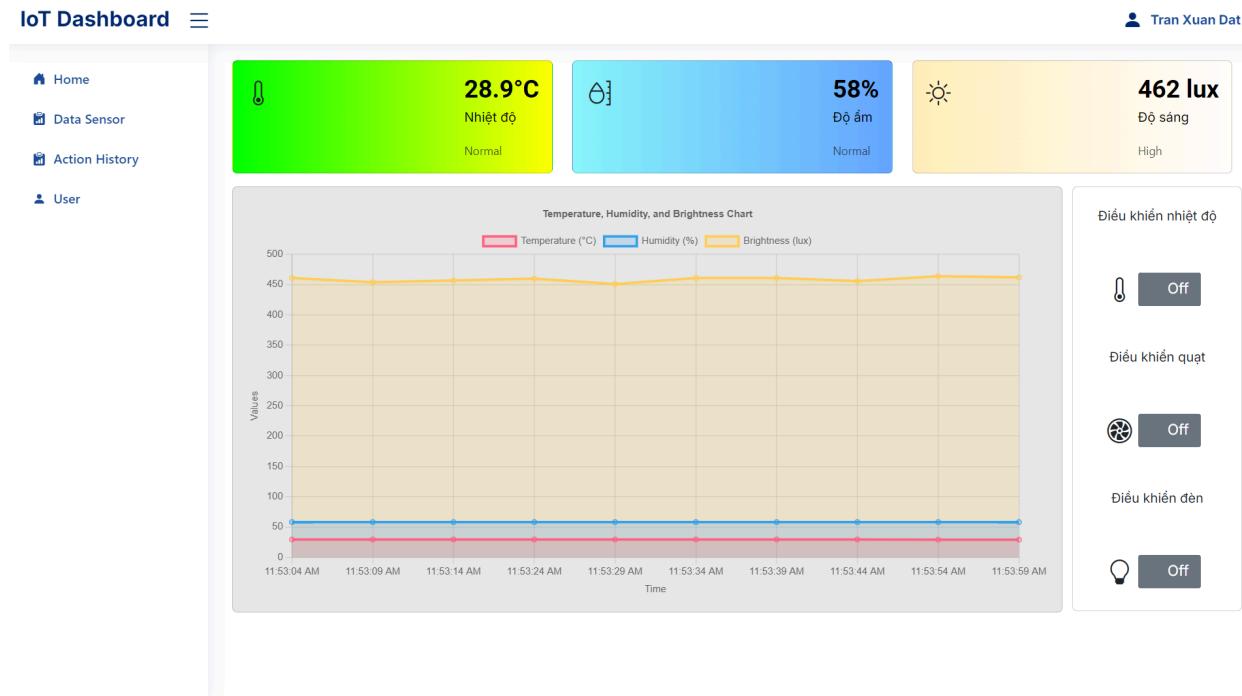
```

- sensor-data.service.ts

```
1 100, 12 hours ago | 1 author (You)
2 import { Injectable } from '@nestjs/common';
3 import { Op } from 'sequelize';
4 import { SensorData } from '../model/sensor-data.model';
5
6 You, 12 hours ago | 1 author (You)
7 @Injectable()
8 export class SensorDataService {
9   async getAllData() {
10     return await SensorData.findAll();
11   }
12
13   async paginateSensorData(pageNumber: number, limitNumber: number): Promise<{ rows: SensorData[], count: number }> {
14     const offset = (pageNumber - 1) * limitNumber;
15     const { rows, count } = await SensorData.findAndCountAll({
16       offset,
17       limit: limitNumber,
18     });
19     return { rows, count };
20   }
21
22   async sortSensorData(sortField: string, sortOrder: 'ASC' | 'DESC'): Promise<{ rows: SensorData[], count: number }> {
23     const { rows, count } = await SensorData.findAndCountAll({
24       order: [[sortField, sortOrder]],
25     });
26     return { rows, count };
27   }
28
29   async searchSensorData(query: string, field?: string): Promise<SensorData[]> {
30     let whereCondition;
31     ...
32
33     if (field) {
34       whereCondition = {
35         [field]: {
36           [Op.like]: `%%${query}%%`,
37         },
38       };
39     } else {
40       whereCondition = {
41         [Op.or]: [
42           { temperature: { [Op.like]: `%%${query}%%` } },
43           { humidity: { [Op.like]: `%%${query}%%` } },
44           { light: { [Op.like]: `%%${query}%%` } },
45           { createdAt: { [Op.like]: `%%${query}%%` } },
46         ],
47       };
48     }
49
50     const logs = await SensorData.findAll({
51       where: whereCondition,
52     });
53     return logs;
54   }
55 }
```

CHƯƠNG 4: KẾT QUẢ

Giao diện Home:



Giao diện DataSensor

The IoT Dashboard Data Sensor interface shows a list of sensor readings. The sidebar includes Home, Data Sensor, Action History, and User. The main area has a search bar and a table titled "Data Sensor" displaying 10 rows of data. The table columns are ID, Temperature, Humidity, Light, and Time. The data shows repeated entries for ID 1, Temperature 30.8, Humidity 98, Light 405, and Time 17-09-2024 17:41:33.

ID	Temperature	Humidity	Light	Time
1	30.8	98	405	17-09-2024 17:41:33
2	30.8	98	432	17-09-2024 17:41:38
3	30.8	98	457	17-09-2024 17:41:43
4	30.8	98	457	17-09-2024 17:41:48
5	30.8	98	456	17-09-2024 17:41:53
6	30.8	98	456	17-09-2024 17:41:58
7	30.8	98	458	17-09-2024 17:42:03
8	30.8	98	439	17-09-2024 17:42:08
9	30.8	98	450	17-09-2024 17:42:13
10	30.8	98	430	17-09-2024 17:42:19

Giao diện Action History

IoT Dashboard ⚙

Tran Xuan Dat

Home Data Sensor Action History User

Action History

All Search by all...

ID	Device	State	Timestamp
1	fan	on	10-09-2024 14:00:54
2	led	on	10-09-2024 14:01:06
3	led		10-09-2024 14:26:34
4	led	AVISO	10-09-2024 14:26:39
5	led	ALARMA	10-09-2024 14:26:44
6	led	AVISO	10-09-2024 14:26:49
7	led	ALARMA	10-09-2024 14:26:54
8	led	AVISO	10-09-2024 14:26:59
9	led	ALARMA	10-09-2024 14:27:04
10	led	AVISO	10-09-2024 14:27:09

Rows: 10 Previous 1 2 3 ... 28 Next

Giao diện User

IoT Dashboard ⚙

Tran Xuan Dat

Home Data Sensor Action History User



Trần Xuân Đạt

Email: xuandattran2003@gmail.com
Lớp tín chỉ: D21CNPM04
Mã sinh viên: B21DCCN223
Nhóm môn học: Lớp IoT và Ứng dụng - Nhóm 5
Link GitHub UI: <https://github.com/xuandat7/IoTDashboardUI>
Link GitHub API: <https://github.com/xuandat7/IoTDashboardAPI>
Link PDF: <https://docs.google.com/document/u/0/>

Data Sensor được gửi về backend mỗi 5s:

```

Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 416 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 412 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 415 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 412 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 416 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 417 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 422 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 415 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 415 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 424 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 423 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 382 }
Executing (default): INSERT INTO `SensorData` (`id`, `temperature`, `humidity`, `light`) VALUES (DEFAULT,?, ?, ?);
Data inserted successfully: { temperature: 28.8999962, humidity: 57, light: 389 }

```

Sử dụng Swagger để thể hiện API

The screenshot displays the Swagger UI for the ESP8266 API. The interface is organized into sections:

- devices**: Contains a POST endpoint for controlling a device.
- ActionHistory**: Contains two GET endpoints for getting history action data with pagination and plain data.
- Sensor**: Contains two GET endpoints for getting all sensor data.
- ESP8266**: Contains a GET endpoint for getting device status.
- devices status**: Contains a GET endpoint for getting the status of all devices.

Thử get device-status

devices status

GET /deviceStatus/device-status Get the status of all devices

Parameters

No parameters

Cancel

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:3001/deviceStatus/device-status' \
-H 'accept: */*
```

Request URL

http://localhost:3001/deviceStatus/device-status

Server response

Code Details

200 Response body

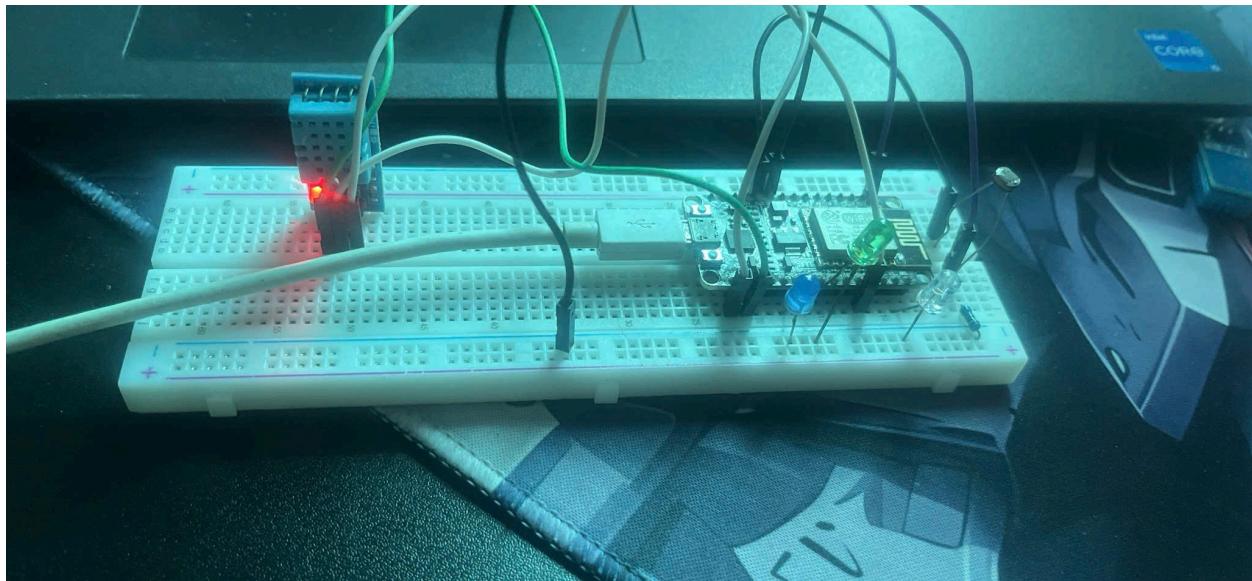
```
{
  "led": {
    "state": "off",
    "connected": true
  },
  "fan": {
    "state": "off",
    "connected": true
  },
  "tem": {
    "state": "off",
    "connected": true
  }
}
```

Download

Response headers

```
access-control-allow-origin: *
content-length: 118
content-type: application/json; charset=utf-8
date: Wed, 18 Sep 2024 05:00:27 GMT
etag: W/76-M28E49V0iyKSeZG607L7mSoH"
x-powered-by: Express
```

Tương ứng với trạng thái đèn:



CHƯƠNG 5: TÀI LIỆU THAM KHẢO

- ESP8266 NodeMCU MQTT Publish Subscribe DHT22 Readings with Arduino IDE (<https://microcontrollerslab.com/esp8266-nodemcu-mqtt-publish-subscribe-dht22-readings/>)

- ESP8266 NodeMCU MQTT – Publish DHT11/DHT22 Temperature and Humidity Readings (Arduino IDE)
(<https://randomnerdtutorials.com/esp8266-nodemcu-mqtt-publish-dht11-dht22-arduino/>)
- NestJS documentation (<https://docs.nestjs.com/v5/>)
-