HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN I

BÁO CÁO BÀI TẬP LỚN

ĐỀ TÀI: TÌM HIỂU VỀ API PATTERN VÀ TRIỂN KHAI VÀO HỆ THỐNG GIÁM SÁT PHÁT HIỆN HOẢ HOẠN

Môn học : **Phát triển phần mềm hướng dịch vụ**

Nhóm môn học : **99**

Họ và tên : **Doãn Mạnh Đạt - B20DCCN170**

Hà nội - 2025

MỤC LỤC

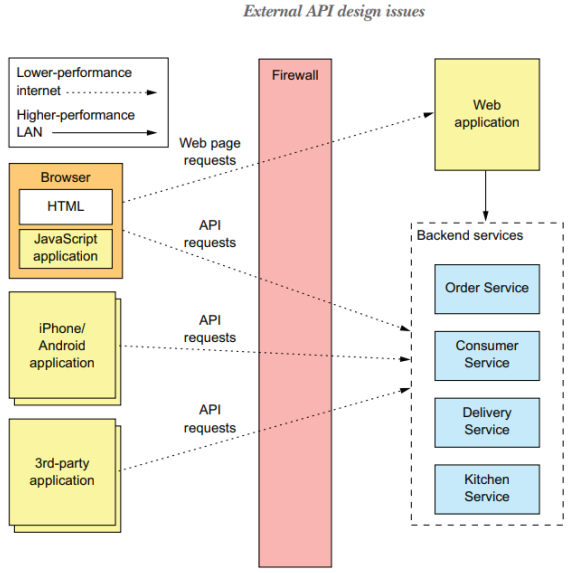
DANH MỤC HÌNH ẢNH

DANH MỤC BẢNG BIỂU

CHƯƠNG 1. GIỚI THIỆU ĐỀ TÀI

1.1. Bài toán thực tiễn

Trong quá trình chuyển đổi từ mô hình monolithic sang microservices, hệ thống FTGO [1] đối mặt với một vấn đề mang tính kiến trúc quan trọng: làm thế nào thiết kế cơ chế cung cấp API hiệu quả cho nhiều loại client khác nhau như ứng dụng mobile, ứng dụng web, dịch vụ partner bên ngoài hay các ứng dụng chạy nội bộ sau tường lửa. Việc để client giao tiếp trực tiếp với từng microservice nhanh chóng bộc lộ nhiều hạn chế: mỗi chức năng của giao diện người dùng thường yêu cầu dữ liệu tổng hợp từ nhiều service khác nhau, khiến số lượng request tăng lên đáng kể và gây ra tình trạng "chatty communication". Điều này đặc biệt nghiêm trọng đối với các thiết bị di động, vốn thường hoạt động trên mạng có độ trễ cao và băng thông thấp. Bên cạnh đó, mỗi loại client lại có yêu cầu dữ liệu rất khác nhau – ứng dụng web cần lượng dữ liệu lớn và phong phú, trong khi ứng dụng mobile chỉ cần payload tối giản để tối ưu hiệu năng và mức tiêu thụ tài nguyên. Việc đưa logic tổng hợp dữ liệu xuống client cũng tạo ra sự phụ thuộc chặt chẽ vào cách phân rã microservices ở backend; bất kỳ sự thay đổi nào trong cấu trúc API nội bộ đều buộc client phải cập nhật theo, làm gia tăng chi phí bảo trì và làm chậm tốc độ phát hành sản phẩm. Ngoài ra, một số service nội bộ sử dụng các giao thức IPC không phù hợp để expose trực tiếp ra bên ngoài, chẳng hạn gRPC hoặc messaging, càng khiến mô hình "client gọi thẳng service" trở nên kém khả thi.



Hình 1.1. Vấn đề của việc thiết kế API bên ngoài

1.2. Giải pháp đưa ra

Để giải quyết các vấn đề trên, đề xuất đưa ra là nhóm các giải pháp External API Patterns, trong đó hai mô hình quan trọng nhất là API Gateway Pattern và Backend for Frontend (BFF) Pattern.

API Gateway đóng vai trò như một lớp trung gian duy nhất đứng trước toàn bộ microservices, chịu trách nhiệm định tuyến (routing) request, tổng hợp dữ liệu từ nhiều service (API composition), chuyển đổi giao thức, và xử lý các chức năng biên như xác thực, phân quyền, ghi log, caching hay rate limiting. Mô hình này không chỉ giảm số lượng request giữa client và server mà còn che giấu mọi chi tiết phức tạp về cách tổ chức microservices ở phía sau.

Tuy nhiên, do mỗi loại client có nhu cầu và đặc điểm mạng khác nhau, API Gateway thường được mở rộng bằng mô hình Backend for Frontends. Theo đó, hệ thống không sử dụng một gateway chung mà tách thành nhiều gateway chuyên biệt, mỗi gateway được thiết kế riêng cho một nhóm client cụ thể như mobile, web hay đối tác bên thứ ba. Cách tiếp cận này giúp tối ưu hóa API theo đúng nhu cầu của từng client, tăng cường tính linh hoạt và hỗ trợ các nhóm frontend triển khai độc lập.

1.3. Thực thi giải pháp

Việc triển khai các giải pháp trên có thể thực hiện theo nhiều cách khác nhau. Nếu yêu cầu đơn giản, các nền tảng thương mại như AWS API Gateway có thể được sử dụng để tạo lớp dịch vụ trung gian mà gần như không cần viết mã. Trong các hệ thống phức tạp hơn, API Gateway có thể được xây dựng trực tiếp bằng các framework như Spring Cloud Gateway hoặc Express Gateway, cho phép lập trình viên toàn quyền kiểm soát quá trình routing, composition và xử lý giao thức. Đối với các nhu cầu API linh hoạt hơn nữa, đặc biệt khi muốn giảm hiện tượng over-fetching và under-fetching, GraphQL có thể được sử dụng như một nền tảng triển khai gateway tự nhiên: schema được khai báo rõ ràng, còn việc truy vấn dữ liệu từ nhiều service được thực hiện thông qua các resolver ánh xạ tới từng microservice tương ứng.

1.4. Kết luận

Việc để client giao tiếp trực tiếp với tất cả các service không chỉ gây ra hiệu năng kém mà còn làm tăng mức độ phụ thuộc vào cấu trúc backend, từ đó cản trở sự phát triển nhanh và linh hoạt của hệ thống. External API Patterns – đặc biệt là API Gateway và Backend for Frontend – trở thành giải pháp thiết yếu giúp đơn giản hóa giao tiếp client–server, tối ưu hóa trải nghiệm người dùng và đảm bảo kiến trúc backend luôn

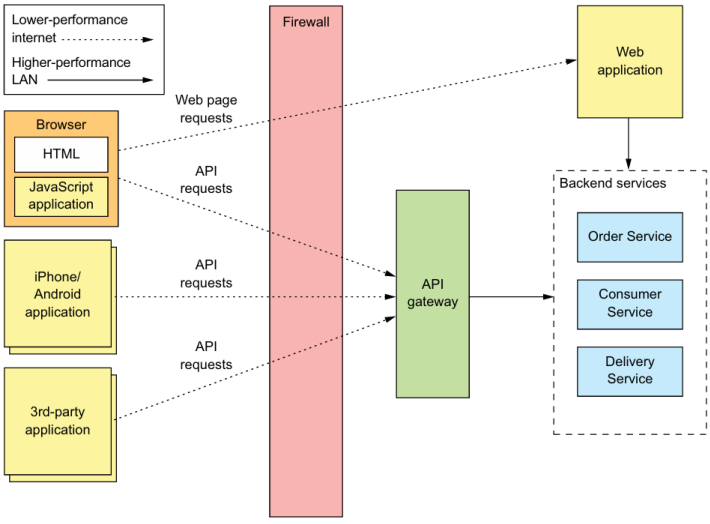
được bảo toàn và dễ mở rộng. Đây là cách tiếp cận phù hợp và gần như tiêu chuẩn đối với mọi hệ thống microservices hiện đại.

CHƯƠNG 2. MẪU API BÊN NGOÀI

Trong bối cảnh hệ thống microservices có nhiều loại client với nhu cầu truy cập dữ liệu rất khác nhau, chương 2 trình bày hai mô hình kiến trúc quan trọng dùng để thiết kế API: API Gateway Pattern và Backend for Frontend (BFF) Pattern. Đây là hai giải pháp được sử dụng rộng rãi trong các hệ thống phân tán để giảm tải cho client, che giấu độ phức tạp của backend, đồng thời tăng tính linh hoạt trong việc phát triển giao diện và khả năng mở rộng của toàn hệ thống.

2.1. API Gateway Pattern

API Gateway Pattern được mô tả như một điểm truy cập duy nhất cho toàn bộ client khi tương tác với các microservice phía sau. Thay vì để client phải biết vị trí, ngôn ngữ, giao thức hoặc hình thức triển khai của từng microservice, gateway cung cấp một API thống nhất và thân thiện hơn, phù hợp với từng tác vụ mà client yêu cầu. Gateway thực hiện nhiều chức năng quan trọng như routing đến đúng service, tổng hợp dữ liệu từ nhiều nguồn (API composition), xử lý xác thực, phân quyền, caching, logging, rate limiting hay chuyển đổi giao thức từ HTTP/JSON sang các cơ chế IPC nội bộ như gRPC hoặc messaging. Nhờ đó, gateway giúp cải thiện hiệu năng, giảm độ trễ, đồng thời đảm bảo client không bị phụ thuộc vào việc phân rã hệ thống backend. Điều này đặc biệt quan trọng với các hệ thống có tần suất thay đổi cao hoặc có nhiều nhóm backend hoạt động độc lập.

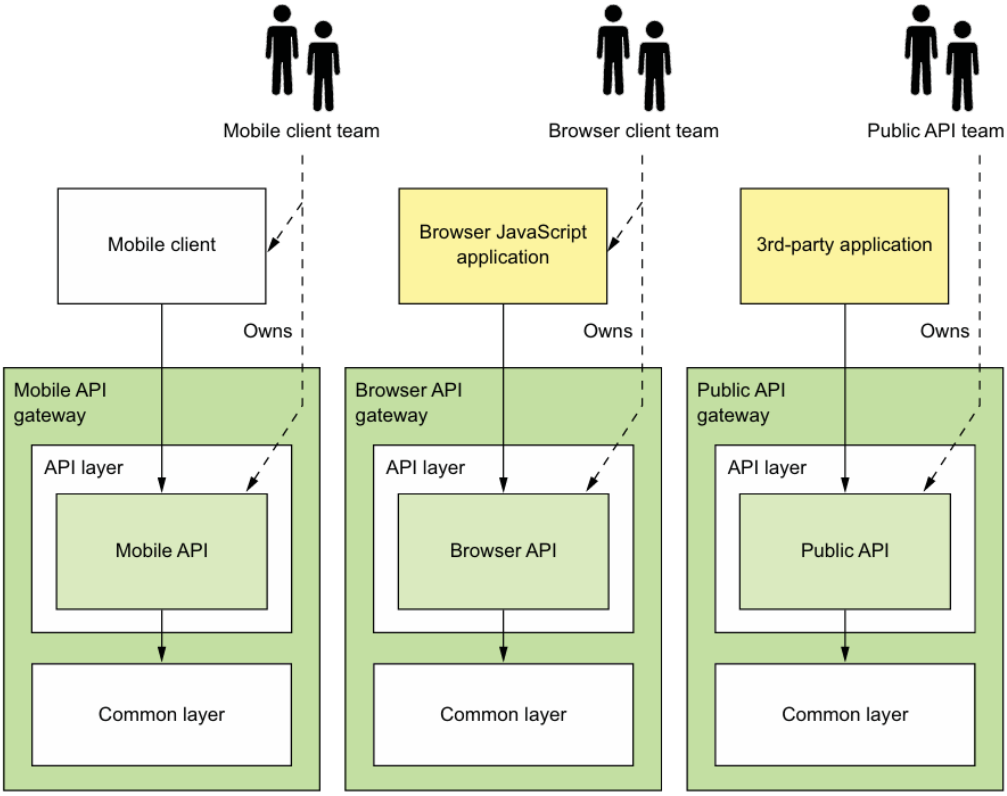


Hình 1.1. Cấu trúc của hệ thống sử dụng API Gateway

2.2. BFF Pattern

Tuy nhiên, việc sử dụng một API Gateway chung cho mọi loại client cũng có những hạn chế nhất định. Các ứng dụng mobile thường cần payload nhỏ và ít chi tiết

hơn so với ứng dụng web, trong khi các dịch vụ partner bên ngoài có thể yêu cầu các chính sách bảo mật hoặc giới hạn truy cập khác biệt hoàn toàn. Để giải quyết hạn chế này, mô hình Backend for Frontend (BFF) được giới thiệu như một sự mở rộng tự nhiên của API Gateway. BFF cho phép mỗi nhóm client có một gateway riêng được thiết kế và tối ưu hóa đúng với nhu cầu sử dụng của họ. Điều này mang lại sự linh hoạt mà một API Gateway chung khó có thể đáp ứng được. Ví dụ, một Mobile BFF có thể hỗ trợ các payload nhẹ, được nén tối đa, hạn chế số lượng request và phù hợp với môi trường mạng di động; trong khi Web BFF có thể cung cấp dữ liệu phong phú hơn, tổ chức phức tạp hơn, hoặc hỗ trợ caching mạnh hơn để tối ưu cho trình duyệt. Đồng thời, mỗi nhóm frontend có thể tự phát triển và triển khai BFF của riêng mình mà không ảnh hưởng đến các nhóm khác, từ đó tăng khả năng tự chủ và tốc độ phát triển.



Hình 2.1. Mô hình Backend from Frontend - BFF

2.3. Kết luận

Từ góc nhìn thực tiễn, hai mô hình API Gateway và BFF không loại trừ nhau mà thường được kết hợp lại trong các hệ thống microservices hiện đại. API Gateway có thể đóng vai trò "entry point" cho toàn bộ yêu cầu từ bên ngoài, trong khi phía sau nó, mỗi loại client vẫn có thể đi qua một BFF chuyên biệt. Điều này tạo ra một kiến trúc API tầng hóa, giúp tăng cường tính mềm dẻo của hệ thống mà vẫn giữ được khả năng kiểm soát tập trung đối với các chức năng biên như bảo mật hay giới hạn lưu lượng. Chương

2 đã cung cấp nền tảng lý thuyết quan trọng cho việc hình thành một chiến lược API hiệu quả trong hệ thống microservices.

CHƯƠNG 3. TRIỂN KHAI CỔNG API

Chương 3 tập trung vào cách triển khai thực tế một API Gateway, bao gồm hai hướng tiếp cận: sử dụng các nền tảng gateway có sẵn (off-the-shelf) và tự xây dựng gateway dựa trên các framework.

3.1. Các nền tảng API Gateway

Sử dụng giải pháp gateway thương mại hoặc dịch vụ quản lý (managed service), tiêu biểu như AWS API Gateway. Với phương pháp này, thay vì lập trình gateway từ đầu, người phát triển chỉ cần định nghĩa các tài nguyên (resource), các phương thức HTTP, và ánh xạ chúng tới các backend cụ thể như một HTTP service nội bộ hoặc các dịch vụ cloud khác. Điểm mạnh của giải pháp này là hầu như không cần viết mã, giúp giảm đáng kể chi phí phát triển và vận hành. Các chức năng như xác thực, giới hạn lưu lượng, logging, caching, hoặc mapping request đều được cung cấp sẵn. Tuy nhiên, nhược điểm của cách tiếp cận này là khả năng mở rộng về mặt logic khá hạn chế, đặc biệt khi cần thực hiện API composition phức tạp hoặc khi gateway cần xử lý các tác vụ kết hợp nhiều bước. Việc tùy chỉnh hành vi gateway thường phải tuân theo cách triển khai của nhà cung cấp dịch vụ và không linh hoạt bằng việc tự xây dựng.

3.2. Tự triển khai API Gateway

Sử dụng các framework ứng dụng, chẳng hạn như Spring Cloud Gateway đối với Java hoặc Express/Node.js đối với môi trường JavaScript. Đây là hướng đi ví dụ cụ thể về FTGO API Gateway. Khi tự xây dựng gateway, nhóm phát triển có toàn quyền kiểm soát logic routing, composition, chuyển đổi dữ liệu và xử lý các chức năng biên. Với Spring Cloud Gateway, nhà phát triển có thể cấu hình rule định tuyến bằng DSL của framework hoặc viết controller để xử lý các endpoint cần kết hợp dữ liệu từ nhiều service. Framework này được xây dựng trên Spring WebFlux và Project Reactor, vốn hỗ trợ mô hình lập trình reactive, cho phép thực hiện nhiều lời gọi service song song theo cách không chặn (non-blocking). Điều này đặc biệt hữu ích khi gateway phải tổng hợp kết quả từ nhiều microservice, giúp giảm đáng kể thời gian phản hồi cho client. Một điểm quan trọng khác trong việc triển khai API Gateway là tính ổn định và khả năng phục hồi khi tương tác với các microservice. Gateway không chỉ cần routing hiệu quả mà còn phải xử lý lỗi một cách an toàn. Gateway cũng phải tích hợp đầy đủ với kiến trúc microservices bằng cách sử dụng dịch vụ khám phá (service discovery), quan sát hệ thống (observability) bao gồm logging, metrics và distributed tracing. Nhờ đó, gateway hoạt động như một microservice thực thụ, có khả năng scale độc lập và giám sát được toàn bộ luồng gọi dịch vụ bên dưới.

CHƯƠNG 4. TRIỂN KHAI CA SỬ DỤNG

4.1. Ý tưởng

Trong bối cảnh các hệ thống IoT ngày càng phổ biến vào đời sống và công nghiệp, bài toán giám sát và cảnh báo cháy trở thành một trong những ứng dụng gần gũi, dễ triển khai nhưng lại mang ý nghĩa thực tiễn rất lớn. Các cảm biến nhiệt độ và khói hiện nay được sử dụng rộng rãi trong gia đình, chung cư, nhà máy, kho hàng… nhằm phát hiện sớm các nguy cơ cháy nổ để giảm thiểu thiệt hại về người và tài sản.

Chính vì tính ứng dụng cao và trực tiếp này, báo cáo bài tập lớn sẽ lựa chọn Use Case: Hệ thống giám sát – phát hiện cháy theo thời gian thực làm đề tài nghiên cứu và xây dựng thử nghiệm. Đây là một trong những Use Case tiêu biểu nhất cho:

- Ứng dụng IoT thực tế
- Luồng xử lý theo thời gian thực (near real-time)
- Phối hợp nhiều công nghệ và giao thức khác nhau
- Áp dụng API Pattern – đặc biệt là Translation Protocol

Hệ thống cảnh báo cháy là ví dụ điển hình của việc cần chuyển đổi giao thức (Protocol Translation). Các thiết bị IoT thực tế thường khai báo dữ liệu qua MQTT hoặc giao thức nhẹ; trong khi các hệ thống Web/API hiện đại vận hành dựa trên HTTP/REST. Để đồng bộ hai thế giới này, kiến trúc hệ thống cần một lớp dịch – *Translation Service* – nhằm chuyển đổi dữ liệu từ MQTT về REST JSON chuẩn. Đây là một trong những mô hình API Pattern phổ biến và quan trọng nhất trong các hệ thống phân tán.

Việc chọn đề tài giám sát cháy giúp báo cáo vừa đảm bảo tính thực tế, vừa thuận tiện trong việc minh họa rõ ràng về chủ đề API Pattern, đặc biệt là Translation Protocol Pattern. Ở tầng này, thiết bị chỉ thực hiện nhiệm vụ thu thập và truyền tải thông tin, chưa tham gia bất kỳ bước xử lý nghiệp vụ nào. Điều này đảm bảo tính đơn giản, dễ thay thế và dễ mở rộng thiết bị.

4.2. Cấu trúc xây dựng hệ thống

4.2.1. Tầng thu thập dữ liệu môi trường – IoT Devices

Phần cứng IoT được thiết kế sử dụng các thành phần như ESP8266, DHT11, cảm biến lửa và còi báo Buzzer 5V, relay 5V,..giúp hệ thống giám sát môi trường hiệu quả và phản ứng nhanh khi phát hiện nguy cơ cháy. Các cảm biến được lập trình để:

- Đọc giá trị nhiệt độ, độ ẩm theo chu kỳ.
- Giám sát và phát hiện lửa bằng cảm biến hồng ngoại.

- Phản ứng khi có dấu hiệu phát hiện lửa, hoặc nhiệt độ cao.
- Đóng gói dữ liệu "thô" thu được và gửi lên server MQTT theo topic:
    - mqtt_topic_pub ="fire_detection/sensor_data
    - mqtt_topic_sub ="fire_detection/control-device

Việc sử dụng giao thức MQTT giúp giảm tải băng thông, phù hợp với thiết bị tài nguyên hạn chế và yêu cầu thời gian thực.

4.2.2. Tầng chuyển đổi giao thức – Translation Service

Translation Service là thành phần quan trọng của hệ thống, có nhiệm vụ trung gian giữa thiết bị phần cứng và tầng xử lý nghiệp vụ. Đây cũng là nơi áp dụng rõ nhất API Pattern Translation Protocol.

Thành phần này thực hiện bốn nhóm chức năng chính:

- (1) Tiếp nhận dữ liệu từ MQTT Broker

Service đăng ký (subscribe) vào các topic MQTT tương ứng với các thiết bị. Dữ liệu đầu vào tại bước này mang tính "thô", thường không đồng nhất và phụ thuộc vào cách đóng gói của từng loại thiết bị.

- (2) Xử lý và chuẩn hóa dữ liệu đầu vào

Dữ liệu được kiểm tra định dạng, chuyển đổi kiểu giá trị, tách – gom trường dữ liệu và thực hiện chuyển đổi từ cấu trúc MQTT payload thành một cấu trúc JSON theo domain model của hệ thống.

- (3) Suy luận trạng thái dựa trên ngưỡng kỹ thuật

Tầng này có thể đánh giá ngưỡng hoặc phân loại điều kiện cảnh báo như:

- Trạng thái bình thường (NORMAL)
- Cảnh báo nguy cơ cháy (FIRE_WARNING)
- Cảnh báo cháy nghiêm trọng (FIRE_DANGER)

Việc trích xuất và suy luận sớm này giúp Backend giảm tải và tăng tính thống nhất của dữ liệu.

- (4) Chuyển đổi giao thức và gửi dữ liệu về Backend bằng REST

Sau khi chuẩn hóa, Translation Service gửi dữ liệu qua REST API đến Backend theo mô hình: "/api/sensor-data". Từ đó, toàn bộ hệ thống phía sau không phải xử lý MQTT hoặc phụ thuộc vào cách thiết bị định dạng dữ liệu.

Qua cơ chế này, Translation Service đóng vai trò tách biệt rõ ràng giữa tầng thiết bị IoT và tầng ứng dụng, đảm bảo khả năng mở rộng và tính linh hoạt của hệ thống.

4.2.3. Tầng xử lý nghiệp vụ - Backend logic server

Backend được xây dựng dựa trên Node.js và Express, có nhiệm vụ tiếp nhận dữ liệu từ Translation Service và thực hiện các xử lý nghiệp vụ chuyên sâu. Chức năng chính bao gồm:

- (1) Tiếp nhận dữ liệu chuẩn hóa từ Translation Service

  Backend không tương tác trực tiếp với MQTT, mà chỉ nhận dữ liệu đã được dịch và chuẩn hóa.

- (2) Lưu trữ và quản lý dữ liệu

  Backend có thể ghi nhận dữ liệu sự kiện vào cơ sở dữ liệu để phục vụ báo cáo hoặc phân tích dài hạn.

- (3) Đánh giá tình trạng và sinh cảnh báo

  Backend có thể thực hiện các bước kiểm tra nâng cao như:

  - Kiểm tra liên tục sự thay đổi nhiệt độ bất thường
  - So sánh dữ liệu theo thời gian để phát hiện xu hướng
  - Kết hợp nhiều cảm biến để giảm cảnh báo sai

- (4) Cung cấp REST API cho giao diện người dùng

  Các endpoint được thiết kế đảm bảo tính nhất quán, dễ truy vấn và mở rộng, ví dụ:

Phương thức	Endpoint	Chức năng
GET	/api/sensor-data	Lấy dữ liệu mới nhất từ cảm biến
GET	/api/history	Lấy lịch sử hoạt động
POST	/api/control-device	Điều khiển thiết bị IoT từ Website
GET	/api/devices	Health check

Bảng 4.1. Các endpoint của hệ thống

Bảng 4.2.

Nhờ đó, Backend trở thành nền tảng cung cấp dữ liệu trung tâm cho các hệ thống giao diện và dịch vụ khác.

4.2.4. Tầng hiển thị và tương tác người dùng – Frontend UIUX

Giao diện được xây dựng bằng HTML, CSS, Bootstrap và JavaScript nhằm bảo đảm:

- Dễ sử dụng, dễ triển khai và dễ tùy chỉnh
- Có tính trực quan trong hiển thị các trạng thái
- Dễ dàng kết nối với REST API từ Backend

Frontend thực hiện các chức năng:

- Hiển thị nhiệt độ, trạng thái khói và mức độ cảnh báo theo thời gian thực
- Tô màu cảnh báo theo mức độ nghiêm trọng
- Cho phép xem lịch sử các sự kiện
- Tổng hợp thông tin từ nhiều thiết bị giám sát

Cơ chế cập nhật có thể là pull định kỳ hoặc push qua WebSocket. Frontend là cầu nối cuối cùng giữa hệ thống kỹ thuật và người vận hành, giúp thông tin được trình bày rõ ràng, kịp thời và dễ đánh giá.

4.2.5. Luồng hoạt động chính – Main Flow

Bước 1: Thiết bị IoT thu thập dữ liệu cảm biến (nhiệt độ, khói).
Bước 2: Thiết bị gửi bản tin chứa dữ liệu lên MQTT Broker.
Bước 3: Translation Service subscribe topic và nhận dữ liệu từ Broker.
Bước 4: Translation Service chuyển đổi dữ liệu từ MQTT sang JSON chuẩn theo domain model.
Bước 5: Translation Service gửi dữ liệu chuẩn hóa đến Backend qua REST API.
Bước 6: Backend ghi nhận, xử lý và lưu dữ liệu vào hệ thống.
Bước 7: Backend cung cấp API để giao diện Dashboard truy vấn trạng thái.
Bước 8: Giao diện người dùng hiển thị trạng thái theo thời gian gần thực, người dùng có thể thao tác trên giao diện chính để tương tác ngược lại với thiết bị IoT.

4.3. Triển khai hệ thống

Dưới đây là hình ảnh mã nguồn của phần dữ liệu thô thực nhận từ thiết bị Iot, mã nguồn của Translation Protocol Pattern và phần dữ liệu sau chuẩn hoá, cuối cùng là giao diện chính của hệ thống.

4.3.1. Thu thập dữ liệu

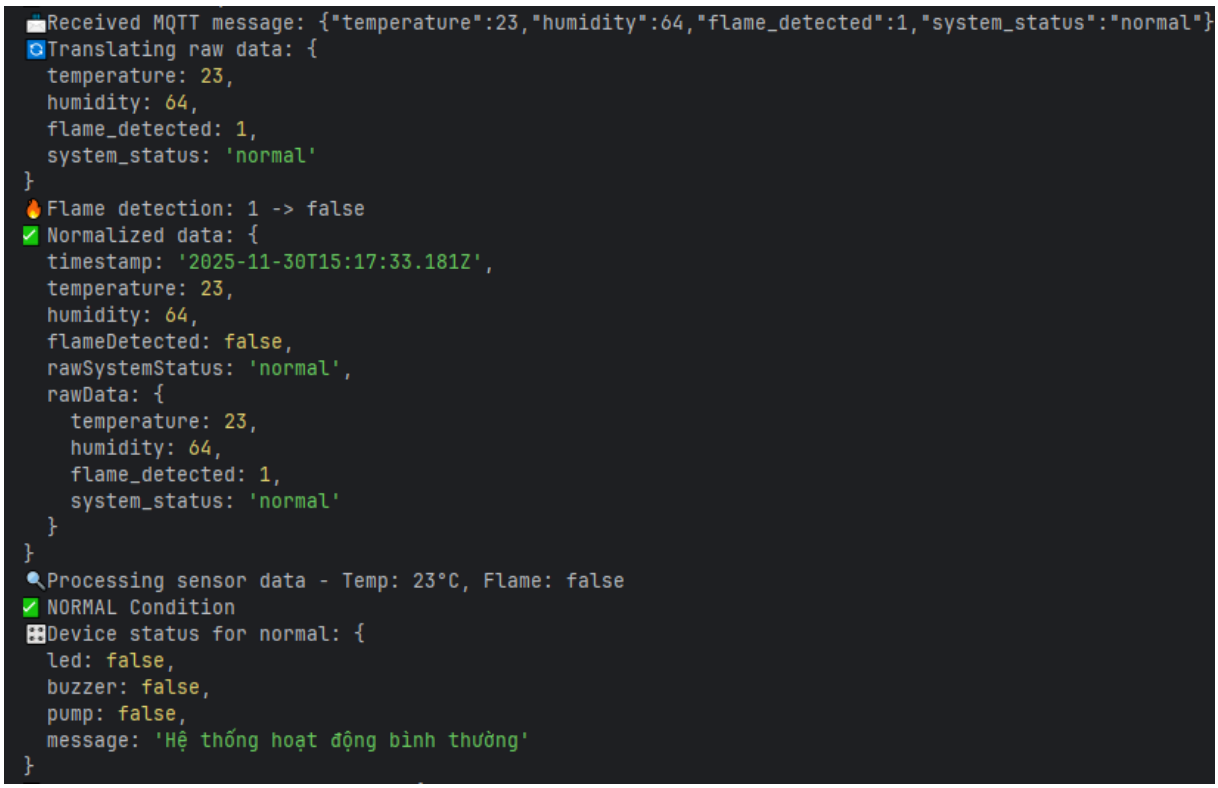
```
22:04:48.687 -> .......
22:04:52.445 -> WiFi connected
22:04:52.445 -> IP address: 192.168.0.102
22:04:52.445 -> Attempting MQTT connection...connected
22:04:52.934 -> Message published
22:04:52.934 -> Temp: 23.0C | Humi: 64.0% | Flame: Not detected | Status: normal
22:04:52.934 -> ------------------------------------
22:04:57.975 -> Message published
22:04:57.975 -> Temp: 23.0C | Humi: 64.0% | Flame: Not detected | Status: normal
22:04:57.975 -> ------------------------------------
22:05:02.935 -> Message published
22:05:02.935 -> Temp: 23.0C | Humi: 64.0% | Flame: Detected | Status: warning
22:05:02.935 -> ------------------------------------
22:05:07.958 -> Message published
22:05:07.958 -> Temp: 23.0C | Humi: 64.0% | Flame: Not detected | Status: normal
22:05:07.958 -> ------------------------------------
22:05:12.947 -> Message published
22:05:12.947 -> Temp: 23.0C | Humi: 64.0% | Flame: Not detected | Status: normal
22:05:12.947 -> ------------------------------------
22:05:17.935 -> Message published
22:05:17.935 -> Temp: 23.0C | Humi: 64.0% | Flame: Not detected | Status: normal
22:05:17.935 -> ------------------------------------
```

Hình 4.1. Dữ liệu thô thu thập bởi cảm biến

4.3.2. Mã nguồn và dữ liệu sau chuẩn hoá

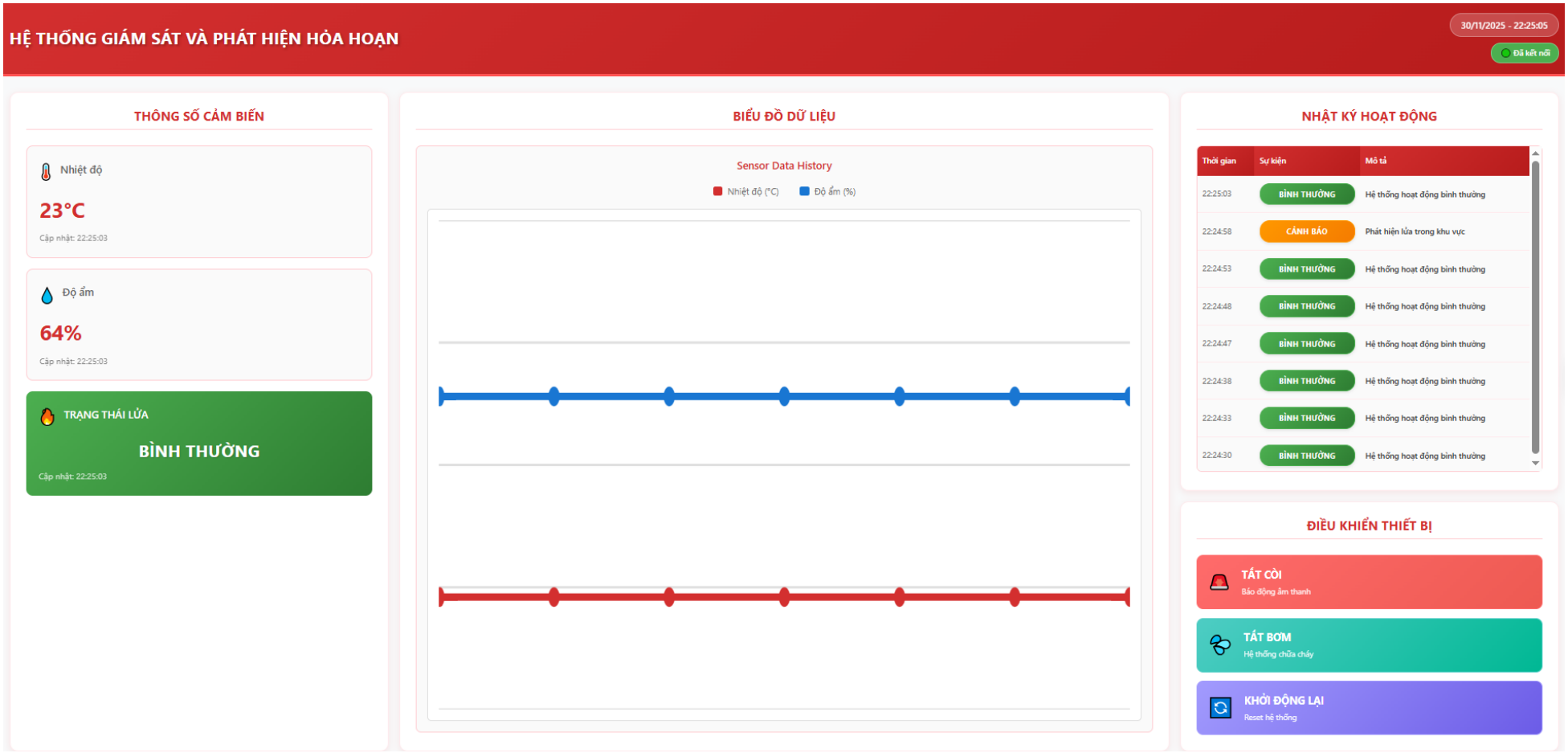
```
export class TranslationService {
    normalizeRawData(rawData) {
        console.log('🔄 Translating raw data:', rawData);

        //Chuẩn hóa dữ liệu thô từ ESP8266
        const normalized = {
            timestamp: new Date().toISOString(),
            temperature: this.validateTemperature(rawData.temperature),
            humidity: this.validateHumidity(rawData.humidity),
            flameDetected: this.normalizeFlameData(rawData.flame_detected),
            rawSystemStatus: rawData.system_status,
            rawData: rawData // Giữ nguyên dữ liệu gốc để debug
        };

        console.log('✅ Normalized data:', normalized);
        return normalized;
    }

    validateTemperature(temp) {
        if (temp === null || temp === undefined || temp < -50 || temp > 150) {
            console.warn('⚠️ Invalid temperature value:', temp);
            return 0;
        }
        return parseFloat(temp.toFixed(1));
    }

    validateHumidity(humidity) {
        if (humidity === null || humidity === undefined || humidity < 0 || humidity > 100) {
            console.warn('⚠️ Invalid humidity value:', humidity);
            return 0;
        }
        return parseFloat(humidity.toFixed(1));
    }

    normalizeFlameData(flameValue) {
        // ESP8266: 0 = có lửa, 1 = không có lửa
        // Chuẩn hóa: true = có lửa, false = không có lửa
        const normalized = flameValue === 0;
        console.log(`🔥 Flame detection: ${flameValue} -> ${normalized}`);
        return normalized;
    }
}
```

Hình 4.2. Mã nguồn xử lý chuyển đổi giao thức

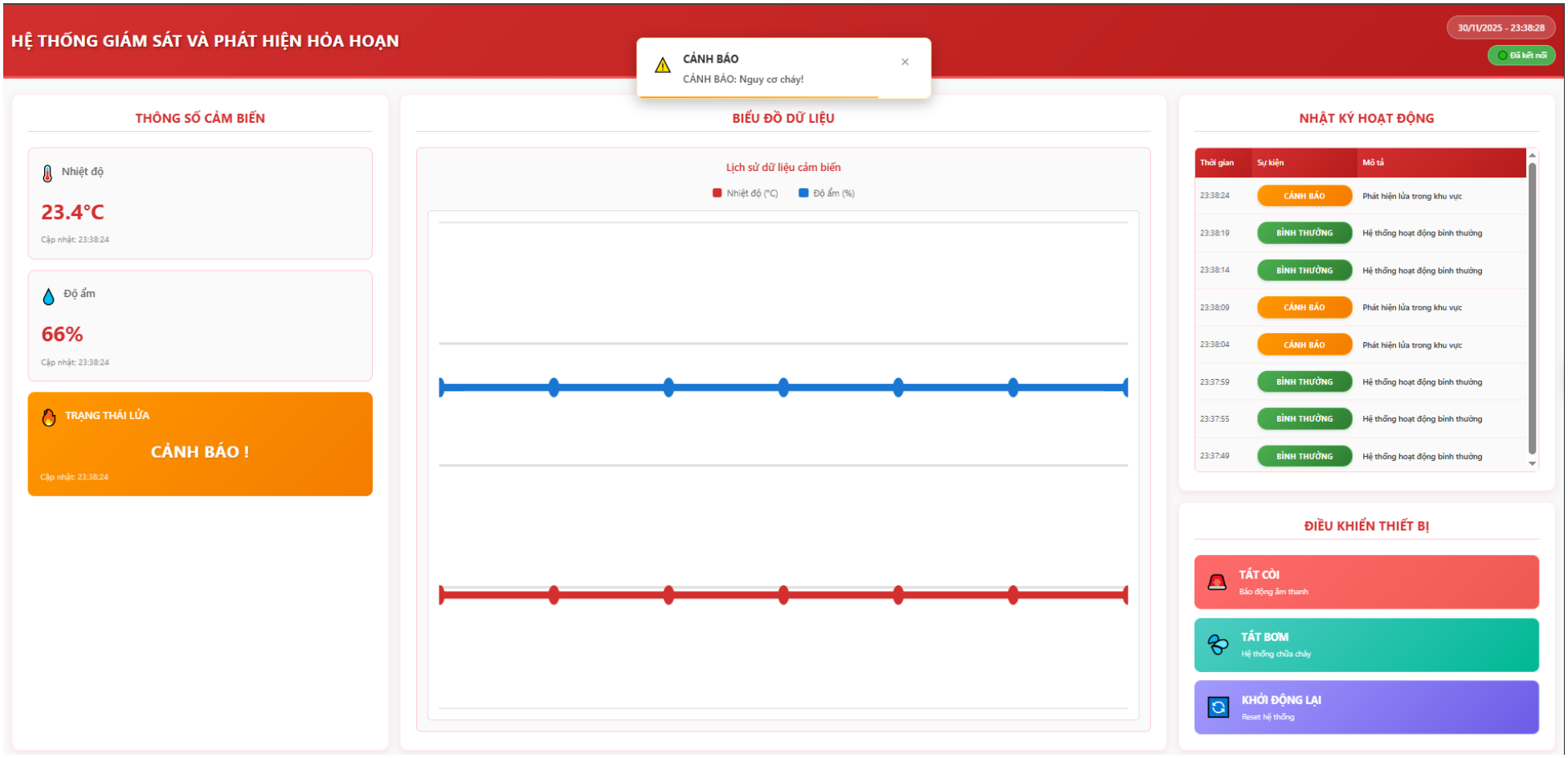


Hình 4.3. Dữ liệu sau khi chuẩn hoá

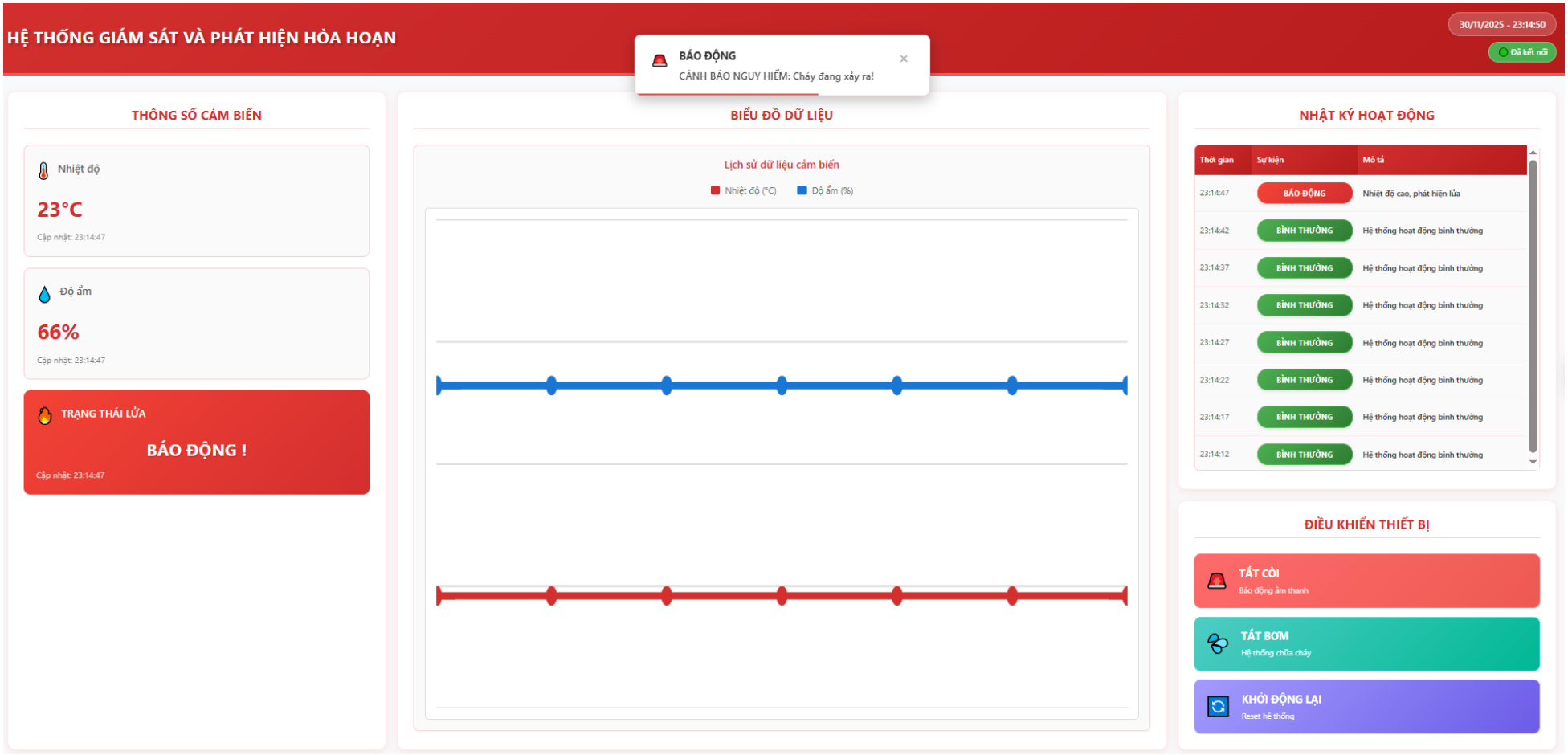
4.3.3. Giao diện chính của hệ thống



Hình 4.4. Hệ thống ở trạng thái bình thường



Hình 4.5. Hệ thống ở trạng thái cảnh báo



Hình 4.6. Hệ thống ở trạng thái báo động

CHƯƠNG 5. KẾT LUẬN

5.1. API Pattern

Trong một hệ thống microservices hiện đại, các client bên ngoài thường truy cập vào các dịch vụ thông qua một API Gateway, thành phần đóng vai trò trung gian giữa client và toàn bộ kiến trúc bên trong. API Gateway cung cấp cho mỗi loại client một tập API được thiết kế riêng, tối ưu với nhu cầu sử dụng của họ, đồng thời thực hiện một loạt các chức năng quan trọng như định tuyến yêu cầu (request routing), tổng hợp dữ liệu từ nhiều dịch vụ (API composition), chuyển đổi giao thức (protocol translation), cũng như triển khai các chức năng biên (edge functions) như xác thực, phân quyền, kiểm soát lưu lượng và logging. Một ứng dụng có thể sử dụng một API Gateway duy nhất hoặc áp dụng mô hình Backend for Frontend (BFF)**,** trong đó mỗi nhóm client (mobile, web, partner, v.v.) có một API Gateway riêng được phát triển và vận hành độc lập. Lợi ích lớn nhất của mô hình BFF là tạo ra sự tự chủ cao cho các nhóm phát triển frontend, giúp họ có thể tự thiết kế và triển khai API phù hợp nhất với trải nghiệm người dùng mà không phụ thuộc trực tiếp vào nhóm backend.

Việc triển khai API Gateway có thể dựa trên nhiều công nghệ khác nhau, từ các sản phẩm thương mại sẵn có cho tới việc tự xây dựng gateway bằng framework. Một lựa chọn phổ biến và dễ sử dụng là Spring Cloud Gateway, vốn hỗ trợ mạnh mẽ việc định tuyến dựa trên bất kỳ thuộc tính nào của request, như HTTP method hay đường dẫn. Framework này cho phép route request trực tiếp đến các backend services hoặc chuyển request vào các handler tùy chỉnh—nơi có thể thực thi logic đặc thù hoặc các quy trình tổng hợp dữ liệu phức tạp. Spring Cloud Gateway được xây dựng trên nền Spring Framework 5 và Project Reactor, sử dụng mô hình lập trình phản ứng (reactive programming) với kiểu dữ liệu như Mono, giúp hệ thống đạt hiệu năng cao và khả năng mở rộng tốt khi xử lý lượng lớn request song song.

5.2. Usecase minh hoạ sử dụng Translation Protocol

Trong phạm vi đề tài này, một trong những nội dung trọng tâm là áp dụng API Pattern – Translation Protocol vào một bài toán thực tế: hệ thống giám sát – phát hiện cháy dựa trên IoT. Đây là một Use Case minh họa rõ ràng cách mà một hệ thống cần chuyển đổi giao thức (protocol translation) giữa hai thế giới hoàn toàn khác biệt:

- Thế giới thiết bị IoT, vốn sử dụng giao thức MQTT dạng publish/subscribe,
- Và thế giới ứng dụng web/server, thường được xây dựng dựa trên HTTP/REST API.

Translation Protocol là thành phần chịu trách nhiệm "dịch" (translate) dữ liệu từ định dạng payload MQTT gốc – vốn không đồng nhất và phụ thuộc vào thiết bị – thành một domain model chuẩn hóa mà Backend có thể xử lý dễ dàng. Việc translation này bao gồm:

1. Tiếp nhận dữ liệu MQTT từ cảm biến nhiệt độ/khói;
2. Phân tích và xác thực payload dựa trên format gốc của thiết bị;
3. Ánh xạ (mapping) dữ liệu sang cấu trúc JSON thống nhất cho toàn hệ thống;
4. Suy luận trạng thái (NORMAL, WARNING, DANGER) dựa trên các tiêu chuẩn kỹ thuật;
5. Chuyển đổi giao thức từ MQTT sang REST thông qua API:

Chính nhờ lớp Translation Protocol này, phần Backend bên trong có thể hoạt động như một microservice thuần túy REST, không bị phụ thuộc vào MQTT hoặc cách thức đóng gói dữ liệu của thiết bị. Điều này hoàn toàn phù hợp với nguyên tắc “tách biệt mối quan tâm” (Separation of Concerns) trong thiết kế hệ thống phân tán.

5.3. Kết luận chung

Từ góc nhìn tổng thể, đề tài đã minh họa thành công ứng dụng của kiến trúc microservices kết hợp API Pattern trong một hệ thống IoT thực tế. Cụ thể:

- API Gateway và các mô hình như BFF cho thấy khả năng tối ưu hóa kiến trúc hướng client.
- Spring Cloud Gateway đại diện cho xu hướng hiện đại trong việc triển khai gateway theo mô hình phản ứng (reactive).
- Translation Protocol trong Use Case minh họa một cách rõ ràng về cách hệ thống cần chuyển đổi dữ liệu và giao thức để kết nối IoT với thế giới REST API.

Sự kết hợp giữa MQTT, Translation Protocol và REST Backend cho thấy tính đúng đắn của kiến trúc, đồng thời khẳng định rằng các API Pattern không chỉ mang tính lý thuyết mà hoàn toàn có thể ứng dụng trực tiếp vào các hệ thống thực tiễn. Đây cũng là minh chứng rõ ràng cho triết lý quan trọng trong thiết kế hệ thống hiện đại: API không chỉ là giao diện truy cập dữ liệu, mà còn là lớp chuyển đổi, bảo vệ và tối ưu hóa toàn bộ kiến trúc phía sau.

TÀI LIỆU THAM KHẢO

[1]: C. Richardson, *Microservices Patterns*. Manning Publications, 2018.