ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
ĐẠI HỌC BÁCH KHOA THÀNH PHỐ HỒ CHÍ MINH
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Đồ án tốt nghiệp

Báo cáo đồ án tốt nghiệp

Tìm hiểu và hiện thực hệ thống trục tích hợp dữ liệu cho trường đại học

---

Giảng viên hướng dẫn: ThS. Nguyễn Cao Đạt
ThS. Nguyễn Quang Sang
Sinh viên thực hiện: Trần Mạnh Tuấn - 2213807
Phạm Lê Huy - 2252260

HỒ CHÍ MINH, THÁNG 5 2026

Lời cam đoan

Nhóm xin cam đoan Đồ án chuyên ngành này là công trình nghiên cứu của riêng nhóm.

Các kết quả nghiên cứu, số liệu và kết luận được trình bày trong báo cáo là trung thực. Mọi thông tin tham khảo, trích dẫn từ các tài liệu, sách báo, trang web và các nghiên cứu trước đây đều đã được chỉ rõ nguồn gốc trong danh mục tài liệu tham khảo theo đúng quy định.

Nếu phát hiện có bất kỳ sự gian lận nào, nhóm xin chịu hoàn toàn trách nhiệm trước Hội đồng và Nhà trường.

Thành phố Hồ Chí Minh, Tháng 05/2026

SVTH1	**SVTH2**
(Ký, ghi rõ họ tên)	(Ký, ghi rõ họ tên)
Trần Mạnh Tuấn	Phạm Lê Huy

Lời cảm ơn

Nhóm thực hiện xin bày tỏ lòng biết ơn chân thành và sâu sắc đến **ThS. Nguyễn Cao Đạt** và **ThS. Nguyễn Quang Sang** - hai giảng viên hướng dẫn đã luôn tận tình hỗ trợ, định hướng và đồng hành cùng nhóm trong suốt quá trình thực hiện đồ án tốt nghiệp.

Trong suốt quá trình nghiên cứu và phát triển hệ thống, nhóm đã nhận được rất nhiều góp ý quý báu từ thầy về cả mặt kỹ thuật lẫn phương pháp triển khai thực tế. Những buổi trao đổi và phản hồi chi tiết của thầy đã giúp nhóm nhận ra các hạn chế trong thiết kế ban đầu, từng bước hoàn thiện kiến trúc hệ thống, cải thiện tư duy tổ chức mã nguồn cũng như nâng cao chất lượng tài liệu kỹ thuật đi kèm.

Bên cạnh đó, sự định hướng của thầy còn giúp nhóm tiếp cận bài toán theo góc nhìn thực tiễn hơn, không chỉ dừng lại ở mức xây dựng một sản phẩm mang tính học thuật mà còn hướng tới khả năng triển khai trong môi trường thực tế với các yêu cầu về ổn định, bảo mật và khả năng mở rộng lâu dài. Những góp ý liên quan đến quy trình đồng bộ dữ liệu, kiểm soát thay đổi lược đồ và cơ chế phân quyền đã góp phần quan trọng trong việc hình thành kiến trúc tổng thể của trục tích hợp dữ liệu.

Nhóm cũng đặc biệt trân trọng sự hỗ trợ và động viên của thầy trong những giai đoạn gặp khó khăn về kỹ thuật cũng như áp lực tiến độ. Sự tận tâm và trách nhiệm của thầy không chỉ giúp nhóm hoàn thành đồ án đúng định hướng mà còn tạo động lực để các thành viên tiếp tục học hỏi, nghiên cứu và nâng cao năng lực chuyên môn của bản thân.

Cuối cùng, nhóm xin gửi lời cảm ơn chân thành đến quý thầy cô trong Khoa Khoa học và Kỹ thuật Máy tính, Trường Đại học Bách khoa - Đại học Quốc gia Thành phố Hồ Chí Minh đã truyền đạt những kiến thức nền tảng quý giá trong suốt quá trình học tập. Những kiến thức và kinh nghiệm đó chính là nền tảng quan trọng để nhóm có thể hoàn thành đồ án này.

Danh sách thành viên & Phân chia công việc

STT	Họ và tên	MSSV	Công việc
1	Phạm Lê Huy	2252260	Xây dựng giao diện trục tích hợp, phát triển module prisma, xây dựng cấu trúc prisma, MongoDB schema, phát triển logic schema thay đổi, lập lịch
2	Trần Mạnh Tuấn	2213807	Phát triển module đồng bộ dữ liệu, xác thực và phân quyền, xây dựng testcase, xử lý lỗi và ngoại lệ đối với các trường hợp đụng độ dữ liệu (data collision)

Mục lục

Lời cam đoan **3**

Lời cảm ơn **4**

Danh mục thuật ngữ và từ viết tắt **12**

1 Tổng quan **14**

2 Phân tích nghiệp vụ và đặc trưng dữ liệu của trường đại học **16**

3 Thiết kế kiến trúc **18**
3.1 Yêu Cầu Chức Năng và Phi Chức Năng 18
3.2 Sơ Đồ Kiến Trúc Tổng Thể 20
3.3 Giải Thích Chi Tiết Các Thành Phần 21
3.3.1 Admin Dashboard (Next.js Frontend) 21
3.3.2 Kong API Gateway 21
3.3.3 NestJS Backend (Monolith) 21
3.3.4 Tầng Lưu Trữ 23
3.3.5 Hệ Thống Nguồn Dữ Liệu 23
3.3.6 Giám Sát: Prometheus và Grafana 23
3.4 So Sánh Với Kiến Trúc Lý Thuyết Ban Đầu 25

4 Những vấn đề trong tích hợp dữ liệu và cách giải quyết **26**
4.1 Các dạng xung đột dữ liệu thường gặp 26
4.2 Cách giải quyết 28
4.2.1 Tầng tiếp nhận dữ liệu — `DataIntegrationService` 29
4.2.2 Tầng đồng bộ — `SyncEngineService` 30
4.2.3 Tổng hợp cơ chế phòng thủ 32

5 Cơ chế xác thực và phân quyền **33**
5.1 Luồng xác thực 33
5.2 Luồng phân quyền 34
5.2.1 Lớp 1 — Kiểm tra vai trò (`RolesGuard`) 34
5.2.2 Lớp 2 — Kiểm tra phạm vi bảng (`PermissionsGuard`) 35
5.3 Cơ chế cấp lại mã truy cập 36
5.4 Cấu trúc cơ sở dữ liệu phục vụ xác thực và phân quyền 36

6 Luồng xử lý dữ liệu **38**
6.1 Chiến lược đồng bộ dữ liệu và cơ chế ghi vào cơ sở dữ liệu 38
6.1.1 Tổng quan 38
6.1.2 Quy trình chung: Phân trang và tải dữ liệu 38
6.1.3 Chiến lược `upsert` — Mặc định 41
6.1.4 Chiến lược `incremental` — Bộ lọc tại phía máy chủ 42
6.1.5 Chiến lược `overwrite` — Xóa và ghi đè toàn bộ 43
6.2 Cơ Chế Retry Trong ETL Pipeline 45
6.2.1 Đặt Vấn Đề 45
6.2.2 Thiết Kế Retry Có Chủ Ý 45
6.2.3 Luồng Xử Lý Khi Retry 47
6.2.4 Đánh Giá 47
6.3 Phát hiện bản ghi mồ côi (Orphan Detection) 49
6.3.1 Đặt vấn đề 49
6.3.2 Giải pháp: Phép trừ tập hợp khóa chính 49
6.3.3 Quy trình triển khai 49
6.3.4 Thiết kế không tự động xóa 50
6.3.5 API quản lý bản ghi mồ côi 51
6.3.6 Giới hạn của phương pháp 52
6.4 Thông Báo Qua Email Sau Khi Pipeline Chạy Xong 53
6.4.1 Đặt vấn đề 53
6.4.2 Công nghệ sử dụng 53
6.4.3 Điểm kích hoạt 53
6.4.4 Nội dung email 53
6.4.5 Cấu hình biến môi trường 54
6.4.6 Luồng xử lý 55

7 Chiến Lược Sao Lưu Dữ Liệu Với MinIO và AWS S3 **56**
7.1 Đặt Vấn Đề 56
7.2 Kiến Trúc Hai Tầng: MinIO và AWS S3 56
7.2.1 Cấu Hình Triển Khai 57
7.3 Bốn Loại Trigger Backup 58
7.3.1 Trigger 1: Pre-Sync Backup 58
7.3.2 Trigger 2: Schema Change Backup 58
7.3.3 Trigger 3: Scheduled Weekly Backup 58
7.3.4 Trigger 4: Manual Backup 59

7.4 Cấu Trúc Lưu Trữ Trong MinIO 59
7.5 Chính Sách Lưu Giữ (Retention Policy) 61
7.6 Cơ Chế Đồng Bộ Lên AWS S3 62
7.6.1 Mục Tiêu 62
7.6.2 Luồng Hoạt Động 62
7.6.3 Trạng Thái Đồng Bộ 63
7.6.4 Fire-and-Forget cho Sync Hàng Loạt 63
7.7 API Quản Lý Backup 64
7.8 Cơ Chế Restore 64
7.9 Đánh Giá 65
7.9.1 Ưu Điểm 65
7.9.2 Hạn Chế và Hướng Phát Triển 65

8 Phân tích Hiệu năng và Capacity Planning 67
8.1 Tổng quan 67
8.2 Kiểm thử Data Collision — Mock Source Server 67
8.2.1 Kiến trúc Mock Server 67
8.2.2 Ma trận Test Case 68
8.2.3 Ví dụ kết quả kiểm thử — TC-03 Deduplication 69
8.2.4 Ví dụ kết quả kiểm thử — TC-11 Dead Letter 70
8.3 Kiểm thử Hiệu năng — PERF Test Cases 71
8.3.1 Thiết kế MemoryProfiler 71
8.3.2 PERF-01 — RAM Capacity Planning 72
8.3.3 PERF-02 — Incremental Sync với Server-side Filter 73
8.4 Phân tích RAM cho Sync Job 75
8.4.1 Vòng đời dữ liệu trong RAM 75
8.4.2 Ước lượng RAM theo kích thước bảng 75
8.5 Phân tích Throughput cho Export API (Outbound) 75
8.5.1 Kiến trúc Export API hiện tại 75
8.5.2 Bottleneck: OFFSET Pagination 75
8.5.3 Ước lượng Throughput theo Little's Law 76
8.6 Tóm tắt và Khuyến nghị Server 77

9 Quy trình CI/CD Với GitHub Actions 78
9.1 Repo chính của dự án 78
9.2 CI/CD Tự Động 78

9.2.1 Đặt vấn đề . . . 78
9.2.2 Kiến trúc pipeline . . . 78
9.2.3 File workflow GitHub Actions . . . 79
9.2.4 Cấu hình GitHub Secrets . . . 81
9.2.5 Lưu ý kỹ thuật . . . 81
9.3 Kiểm thử tự động (Unit Testing) . . . 82
9.4 Đánh giá . . . 82

10 Triển Khai Hệ Thống Lên AWS EC2 83
10.1 Tổng Quan Kiến Trúc Triển Khai . . . 83
10.2 Yêu Cầu Hạ Tầng . . . 83
10.2.1 Cấu Hình EC2 Instance . . . 83
10.2.2 Security Group . . . 84
10.3 Cài Đặt Môi Trường . . . 85
10.3.1 Cài Docker . . . 85
10.3.2 Cài Docker Compose Plugin và Buildx . . . 85
10.4 Triển Khai Ứng Dụng . . . 86
10.4.1 Clone Repository . . . 86
10.4.2 Cấu Hình Biến Môi Trường . . . 86
10.4.3 Build và Khởi Động . . . 87
10.4.4 Khởi Tạo Cơ Sở Dữ Liệu . . . 87
10.5 Cấu Hình CORS Cho Kong API Gateway . . . 87
10.6 Quản Lý Sau Triển Khai . . . 88
10.6.1 Tự Động Khởi Động Lại . . . 88
10.6.2 Vấn Đề Thay Đổi Public IP . . . 88
10.6.3 Kiểm Tra Trạng Thái Hệ Thống . . . 89

11 Giao diện người dùng và quản trị hệ thống 90
11.1 Bảng điều khiển trung tâm . . . 90
11.2 Quản lý lịch sử đồng bộ . . . 91
11.3 Quản lý sao lưu và phục hồi . . . 92
11.4 Quản lý phân quyền truy cập . . . 93
11.5 Quản lý và đối soát lược đồ dữ liệu . . . 94
11.6 Tra cứu và khai thác dữ liệu theo phạm vi phân quyền . . . 96

12 Tổng kết **98**
12.1 Những gì đã đạt được . . . 98
12.1.1 Kiến trúc và thiết kế hệ thống . . . 98
12.1.2 Luồng ETL và chiến lược đồng bộ . . . 98
12.1.3 Hệ thống phòng thủ Data Collision . . . 98
12.1.4 Schema Registry và giám sát thay đổi schema . . . 99
12.1.5 Sao lưu và phục hồi . . . 99
12.1.6 CI/CD và triển khai . . . 99
12.1.7 Giao diện quản trị . . . 99
12.2 Hạn chế và hướng phát triển . . . 100
12.3 Kết luận . . . 101

Phụ lục A: Đặc tả API **101**

Phụ lục B: Hướng dẫn cài đặt và triển khai **105**

Danh sách hình vẽ

3.1 Sơ đồ kiến trúc tổng thể . . . 20
3.2 Tương tác giữa các module backend và với các thành phần bên ngoài . . . 24
5.1 Luông xác thực và kiểm tra quyền truy cập . . . 34
5.2 Luồng xử lý cấp lại mã truy cập (Refresh Token) . . . 36
6.1 Cách thức hoạt động của hàm syncOneTable . . . 40
6.2 Luồng quyết định retry trong pipeline ETL . . . 47
6.3 Quy trình hai bước: phát hiện và xác nhận xóa bản ghi mồ côi . . . 51
6.4 Minh họa mail cảnh báo . . . 54
6.5 Luồng gửi email thông báo sau khi pipeline kết thúc . . . 55
7.1 Kiến trúc lưu trữ hai tầng: MinIO và AWS S3 . . . 57
7.2 Giao diện cấu trúc trong MINIO . . . 60
7.3 Các file Backup cho một bảng trong MINIO . . . 60
7.4 Giao diện lưu trữ file trên S3 . . . 62
9.1 Repo Github . . . 78
9.2 Luồng CI/CD từ push code đến deploy EC2 . . . 79
10.1 Kiến trúc triển khai trên EC2 . . . 83
10.2 Máy chủ EC2 trên AWS . . . 84
11.1 Giao diện bảng điều khiển trung tâm . . . 91

11.2 Giao diện quản lý lịch sử đồng bộ 92
11.3 Giao diện quản lý sao lưu và phục hồi dữ liệu 93
11.4 Giao diện quản lý phân quyền truy cập cho người dùng 94
11.5 Giao diện quản lý lược đồ dữ liệu 96
11.6 Giao diện đối với người dùng không phải là quản trị viên 97

Danh sách bảng

3.1 So sánh kiến trúc lý thuyết và thực tế 25
4.1 Các trường hợp xung đột kiểu dữ liệu điển hình 27
4.2 Quy tắc chuẩn hóa kiểu dữ liệu 31
4.3 Tổng hợp các cơ chế phòng thủ xung đột dữ liệu 32
6.1 Hiệu quả Incremental: Lọc tại đích (BE-side) và lọc tại nguồn (Server-side) 43
6.2 Chính sách retry theo loại lỗi 46
6.3 Biến môi trường cho NotificationService 54
7.1 Chính sách lưu giữ backup theo loại trigger 61
7.2 Các giá trị trạng thái đồng bộ S3 63
7.3 Các endpoint quản lý backup 64
8.1 Ma trận kiểm thử Data Collision Defense 68
8.2 Phân tích nguồn tiêu thụ RAM trong pipeline (PERF-01) 73
8.3 So sánh `upsert` vs `incremental` — kết quả thực nghiệm 74
8.4 Ước lượng peak RAM theo cấu hình bảng 75
8.5 Ước lượng latency theo độ sâu trang — OFFSET vs Keyset 76
8.6 Ước lượng RPS tối đa theo cấu hình pool và latency 77
8.7 Khuyến nghị cấu hình server cho Data Integration Hub 77
9.1 Danh sách GitHub Secrets cần cấu hình 81
9.2 Bộ unit test của backend 82
10.1 Cấu hình EC2 instance được sử dụng 83
10.2 Inbound rules của Security Group 84
10.3 Địa chỉ truy cập các dịch vụ sau triển khai 89
12.1 Hạn chế hiện tại và hướng phát triển 100

Danh mục thuật ngữ và từ viết tắt

Từ viết tắt/Thuật ngữ	Ý nghĩa
ACID	Atomicity, Consistency, Isolation, Durability - Các thuộc tính đảm bảo độ tin cậy của giao dịch cơ sở dữ liệu.
API	Application Programming Interface - Giao diện lập trình ứng dụng.
CI/CD	Continuous Integration/Continuous Deployment - Tích hợp liên tục và Triển khai liên tục.
Data Silo	Ốc đảo dữ liệu - Tình trạng dữ liệu bị cô lập, phân mảnh giữa các hệ thống.
ETL	Extract, Transform, Load - Quá trình trích xuất, biến đổi và tải dữ liệu.
Golden Record	Bản ghi lõi/Bản ghi vàng - Phiên bản dữ liệu duy nhất, đầy đủ và đáng tin cậy nhất sau khi đã qua chuẩn hóa từ nhiều nguồn.
HTTP	Hypertext Transfer Protocol - Giao thức truyền tải siêu văn bản.
JSON	JavaScript Object Notation - Định dạng trao đổi dữ liệu.
JWT	JSON Web Token - Định dạng token để xác thực và trao đổi thông tin.
MDM	Master Data Management - Quản trị dữ liệu chủ.
OIDC	OpenID Connect - Lớp xác thực người dùng trên nền tảng OAuth 2.0.
ORM	Object-Relational Mapping - Kỹ thuật ánh xạ dữ liệu giữa cơ sở dữ liệu quan hệ và đối tượng trong code.

RBAC	Role-Based Access Control - Phân quyền truy cập dựa trên vai trò.
Schema Registry	Nơi lưu trữ, quản lý cấu trúc siêu dữ liệu (metadata) của các bảng.
SSO	Single Sign-On - Cơ chế đăng nhập một lần.
UI	User Interface - Giao diện người dùng.
Upsert	Update or Insert - Thao tác cập nhật bản ghi nếu đã tồn tại, hoặc tạo mới nếu chưa tồn tại.
Prisma DMMF	Data Model Meta Format — một cấu trúc dữ liệu (JSON) nội bộ mà Prisma ORM tạo ra từ file schema.prisma

1 Tổng quan

Trong bối cảnh chuyển đổi số đang diễn ra mạnh mẽ tại các cơ quan quản lý và các cơ sở giáo dục đại học, dữ liệu ngày càng trở thành tài sản cốt lõi của nhà trường. Việc quản lý, tích hợp và xây dựng các quy chuẩn dùng chung cho dữ liệu có vai trò đặc biệt quan trọng, nhằm bảo đảm hoạt động phối hợp thông suốt giữa các đơn vị, đồng thời tạo nền tảng cho công tác thống kê, phân tích và hỗ trợ ban lãnh đạo ra quyết định.

Tuy nhiên, quá trình khai thác và chia sẻ dữ liệu trong nội bộ trường đại học hiện nay vẫn còn nhiều hạn chế. Mỗi đơn vị, phòng ban thường sử dụng một hệ thống phần mềm riêng để quản lý các lĩnh vực như nhân sự, đào tạo, tài chính, nghiên cứu khoa học hoặc sinh viên. Điều này dẫn đến hiện tượng dữ liệu bị cô lập giữa các hệ thống, mỗi nơi chỉ lưu giữ một phần thông tin và thiếu sự liên kết với các hệ thống khác. Bên cạnh đó, cùng một đối tượng nhưng có thể được lưu trữ nhiều lần ở các hệ thống khác nhau với định dạng, cấu trúc hoặc nội dung không thống nhất. Khi dữ liệu được chia sẻ thủ công giữa các đơn vị, việc cập nhật thường không được đồng bộ, dễ phát sinh sai lệch, trùng lặp và khó xác định đâu là phiên bản mới nhất.

Để giải quyết bài toán trên, đề tài hướng tới việc xây dựng một trục tích hợp dữ liệu cho trường đại học. Hệ thống đóng vai trò là đầu mối trung tâm, tiếp nhận dữ liệu từ các hệ thống nguồn, thực hiện chuẩn hóa, kiểm tra tính hợp lệ và lưu trữ vào cơ sở dữ liệu dùng chung của toàn trường. Các hệ thống thành phần sẽ không kết nối trực tiếp với nhau mà trao đổi thông tin thông qua trục tích hợp, từ đó giúp giảm sự phụ thuộc lẫn nhau và bảo đảm tính nhất quán của dữ liệu.

Kiến trúc được đề xuất theo mô hình Hub-and-Spoke kết hợp với API-led, trong đó trục tích hợp là trung tâm kết nối, còn các hệ thống quản lý của từng đơn vị là các vệ tinh. Một điểm quan trọng của hệ thống là xây dựng bộ đặc tả dữ liệu thống nhất, quy định rõ cấu trúc, ý nghĩa và kiểu dữ liệu của từng trường thông tin mà các hệ thống nguồn phải tuân thủ. Trên cơ sở đó, hệ thống cung cấp các giao diện lập trình ứng dụng thống nhất để các đơn vị có thể cập nhật và khai thác dữ liệu theo cùng một chuẩn.

Ngoài chức năng tích hợp dữ liệu, trục tích hợp còn hỗ trợ công tác quản trị dữ liệu thông qua các cơ chế theo dõi, ghi nhận lịch sử thay đổi, kiểm tra trạng thái đồng bộ và giám sát hoạt động của toàn hệ thống theo thời gian thực. Đồng thời, hệ thống cũng được thiết kế với các cơ chế bảo mật như xác thực, phân quyền, mã hóa dữ liệu và cổng quản lý truy cập nhằm bảo đảm an toàn cho dữ liệu khi trao đổi giữa các hệ thống.

Bên cạnh việc phục vụ nhu cầu chia sẻ dữ liệu trong nội bộ trường, trục tích hợp còn là nền tảng để đồng bộ dữ liệu với các đơn vị quản lý cấp trên như Đại học Quốc gia Thành phố Hồ Chí Minh hoặc Hệ thống cơ sở dữ liệu giáo dục đại học. Qua đó, nhà

trường có thể bảo đảm dữ liệu được cung cấp đầy đủ, chính xác và kịp thời theo đúng yêu cầu quản lý.

2 Phân tích nghiệp vụ và đặc trưng dữ liệu của trường đại học

1. **Tính chu kỳ và sự biến động theo thời gian**

   Dữ liệu trong trường đại học không phát sinh đồng đều mà phụ thuộc chặt chẽ vào lịch trình của năm học. Ở một số thời điểm, hệ thống có thể phải xử lý khối lượng dữ liệu rất lớn trong thời gian ngắn.

   Các giai đoạn có lưu lượng truy cập cao bao gồm:

   - Đăng ký học phần: Đây là thời điểm hệ thống chịu tải lớn nhất do số lượng truy vấn và cập nhật tăng đột biến trong thời gian ngắn.
   - Kỳ thi và nhập điểm: Cuối mỗi học kỳ, lượng dữ liệu điểm số từ các hệ thống đào tạo được chuyển về trục tích hợp rất lớn.
   - Tuyển sinh: Trong giai đoạn từ tháng 7 đến tháng 9, hệ thống phải tiếp nhận số lượng lớn hồ sơ sinh viên mới.

   Ngược lại, trong thời gian nghỉ hè hoặc giữa học kỳ, dữ liệu phát sinh rất ít và tần suất cập nhật thấp. Vì vậy, trục tích hợp cần được thiết kế để có khả năng mở rộng linh hoạt theo từng thời điểm, tránh quá tải vào giai đoạn cao điểm nhưng vẫn tiết kiệm tài nguyên khi nhu cầu thấp.

2. **Sự thay đổi vai trò của đối tượng**

   Một đặc trưng quan trọng của dữ liệu đại học là cùng một cá nhân có thể thay đổi vai trò trong suốt quá trình gắn bó với nhà trường nhưng vẫn phải được quản lý như một đối tượng duy nhất.

   Ví dụ, một người có thể trải qua các giai đoạn:

   Sinh viên đại học → Cựu sinh viên → Học viên cao học → Giảng viên hoặc nhân viên chức.

   Mỗi giai đoạn có thể được quản lý bởi một hệ thống khác nhau và sử dụng các mã định danh khác nhau, chẳng hạn mã số sinh viên đại học, mã học viên cao học hoặc mã viên chức. Nếu không có cơ chế liên kết, dữ liệu của cùng một người sẽ bị tách rời thành nhiều bản ghi khác nhau.

   Do đó, trục tích hợp cần xây dựng một định danh duy nhất cho mỗi cá nhân, có thể dựa trên căn cước công dân hoặc mã định danh nội bộ. Định danh này được sử

dụng để liên kết toàn bộ lịch sử và vai trò của đối tượng trong các hệ thống khác nhau.

3. **Tính đa dạng về cấu trúc dữ liệu**

   Dữ liệu của trường đại học rất đa dạng về nội dung và định dạng. Một hệ thống tích hợp phải có khả năng xử lý đồng thời nhiều loại dữ liệu khác nhau.

   Các nhóm dữ liệu tiêu biểu bao gồm:

   - Dữ liệu định danh: họ tên, ngày sinh, giới tính, quê quán, địa chỉ.
   - Dữ liệu học tập: học phần, số tín chỉ, điểm số, kết quả học tập.
   - Dữ liệu tài chính: học phí, học bổng, các khoản thu chi.
   - Dữ liệu nghiên cứu khoa học: bài báo, đề tài, minh chứng, tệp tài liệu.

   Trong đó, có những dữ liệu được tổ chức theo cấu trúc chặt chẽ dưới dạng bảng, nhưng cũng có những dữ liệu tồn tại dưới dạng tệp văn bản, hình ảnh hoặc tài liệu đính kèm. Vì vậy, trục tích hợp cần hỗ trợ cả dữ liệu có cấu trúc và dữ liệu phi cấu trúc.

4. **Tính phụ thuộc và yêu cầu nhất quán giữa các hệ thống**

   Dữ liệu của một đơn vị thường là đầu vào cho hoạt động của đơn vị khác. Do đó, sự thay đổi dữ liệu ở hệ thống nguồn phải được phản ánh kịp thời đến các hệ thống liên quan.

   Ví dụ, sinh viên chỉ được phép đóng học phí nếu đã hoàn tất đăng ký học phần. Khi thông tin đăng ký học phần thay đổi, dữ liệu tương ứng trong hệ thống tài chính, thư viện hoặc ứng dụng dành cho sinh viên cũng cần được cập nhật.

   Vì vậy, trục tích hợp cần hỗ trợ cơ chế xử lý theo sự kiện. Khi có một thay đổi xảy ra ở hệ thống nguồn, trục tích hợp sẽ tự động phát sinh các tác vụ đồng bộ đến các hệ thống phụ thuộc nhằm bảo đảm dữ liệu luôn nhất quán.

5. **Tính phân cấp và nhu cầu tổng hợp dữ liệu**

   Dữ liệu trong trường đại học thường được hình thành theo nhiều cấp độ, từ dữ liệu chi tiết đến dữ liệu tổng hợp để phục vụ công tác thống kê và báo cáo.

   Ví dụ:

   Điểm thành phần → Điểm học phần → Điểm trung bình tích lũy → Xếp loại học lực.

Trục tích hợp không chỉ lưu trữ dữ liệu chi tiết mà còn cần có khả năng tính toán, tổng hợp và chuẩn hóa dữ liệu ở mức cao hơn. Điều này đặc biệt quan trọng khi đồng bộ dữ liệu lên các hệ thống cấp trên, vì các hệ thống này thường chỉ yêu cầu dữ liệu tổng hợp thay vì toàn bộ dữ liệu chi tiết.

6. **Yêu cầu khác nhau về độ trễ đồng bộ**

   Không phải mọi loại dữ liệu đều cần được đồng bộ ngay lập tức. Tùy theo mức độ quan trọng của nghiệp vụ, mỗi loại dữ liệu sẽ có yêu cầu khác nhau về thời gian cập nhật.

   - Dữ liệu điểm thi hoặc báo cáo thống kê có thể chấp nhận độ trễ từ vài chục phút đến cuối ngày.
   - Dữ liệu về tình trạng sinh viên như thôi học, bảo lưu hoặc bị đình chỉ cần được cập nhật gần như ngay lập tức để kịp thời khóa hoặc thay đổi quyền truy cập của sinh viên trên các hệ thống khác.

   Vì vậy, trục tích hợp cần phân loại dữ liệu theo mức độ ưu tiên để lựa chọn cơ chế đồng bộ phù hợp, tránh lãng phí tài nguyên nhưng vẫn đáp ứng yêu cầu nghiệp vụ.

Dựa trên các nghiên cứu lý thuyết ở chương trước, chương này trình bày kiến trúc thực tế đã được hiện thực hóa cho hệ thống Trục tích hợp dữ liệu. Kiến trúc được điều chỉnh so với đề xuất ban đầu để phù hợp với phạm vi dự án capstone và các ràng buộc kỹ thuật thực tế.

3 Thiết kế kiến trúc

3.1 Yêu Cầu Chức Năng và Phi Chức Năng

Yêu Cầu Chức Năng

- **Tích hợp dữ liệu từ hệ thống nguồn:** Hệ thống kết nối và trích xuất dữ liệu từ API của hệ thống nguồn thông qua giao thức HTTP/REST, hỗ trợ phân trang và xử lý dữ liệu theo lô.
- **Đồng bộ theo nhiều chiến lược:** Hỗ trợ ba chiến lược đồng bộ — `upsert` (thêm mới hoặc cập nhật), `incremental` (chỉ xử lý bản ghi thay đổi theo `updatedAfter`), và `overwrite` (xóa toàn bộ rồi ghi lại) — được cấu hình riêng cho từng bảng thông qua Schema Registry.

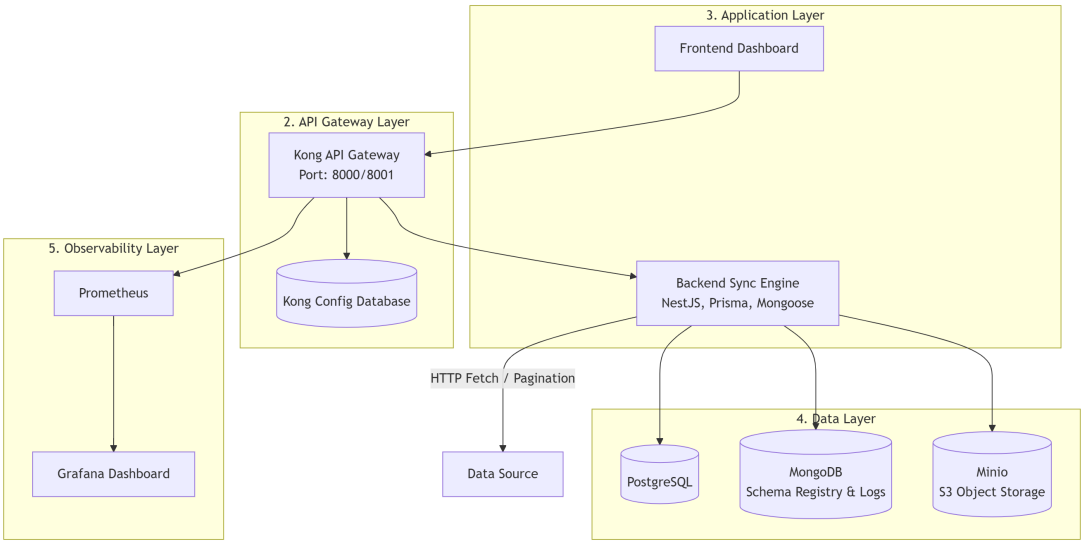
- **Lập lịch đồng bộ tự động:** Cho phép tạo, cấu hình và kích hoạt/vô hiệu hóa các tác vụ đồng bộ định kỳ thông qua biểu thức cron. Lịch được lưu trong MongoDB và nạp động vào bộ lập lịch khi khởi động.
- **Quản lý schema:** Tự động phát hiện thay đổi cấu trúc dữ liệu tại hệ thống nguồn thông qua so sánh hash. Khi schema thay đổi, bảng bị khóa khỏi chu kỳ đồng bộ cho đến khi quản trị viên xem xét và phê duyệt.
- **Sao lưu và phục hồi:** Tự động tạo snapshot trước mỗi lần sync, sau mỗi thay đổi schema và theo lịch hàng tuần. Hỗ trợ restore về bất kỳ điểm nào trong lịch sử backup.
- **Xác thực và phân quyền:** Mọi yêu cầu API đều phải qua xác thực JWT. Phân quyền theo vai trò kết hợp phạm vi bảng dữ liệu cụ thể.
- **Giám sát và cảnh báo:** Thu thập metrics qua Prometheus, hiển thị trên Grafana, và gửi email cảnh báo khi tác vụ đồng bộ thất bại.

Yêu Cầu Phi Chức Năng

- **Tính bảo mật:** Mật khẩu được băm bằng `bcrypt`. Token JWT ký bằng khóa bí mật. Toàn bộ giao tiếp qua Kong API Gateway với kiểm soát CORS và xác thực token tập trung.
- **Tính bền vững dữ liệu:** Backup đa lớp (MinIO nội bộ + AWS S3 đám mây) đảm bảo dữ liệu không bị mất ngay cả khi hạ tầng on-premise gặp sự cố.
- **Khả năng quan sát:** Toàn bộ sự kiện đồng bộ, lỗi và kết quả được ghi vào Event Log (MongoDB). Metrics được thu thập và hiển thị theo thời gian thực trên Grafana.
- **Khả năng bảo trì:** Backend được tổ chức theo mô hình module hóa của NestJS — mỗi module có trách nhiệm duy nhất, dễ dàng mở rộng hoặc thay thế độc lập.
- **Tính sẵn sàng:** Tất cả container được cấu hình `restart: always` — tự động khởi động lại sau sự cố hoặc khi máy chủ reboot.

3.2 Sơ Đồ Kiến Trúc Tổng Thể

So với kiến trúc lý thuyết API-led Connectivity ba lớp (System API, Process API, Experience API) ban đầu, kiến trúc thực tế được hợp nhất thành một **backend monolith** theo mô hình module hóa của NestJS. Quyết định này xuất phát từ phạm vi dự án capstone: số lượng nguồn dữ liệu, lưu lượng và team size chưa đủ để biện minh cho chi phí vận hành của kiến trúc microservice đầy đủ.



Hình 3.1: Sơ đồ kiến trúc tổng thể

3.3 Giải Thích Chi Tiết Các Thành Phần

3.3.1 Admin Dashboard (Next.js Frontend)

Giao diện quản trị được xây dựng bằng Next.js, đóng vai trò là điểm tương tác duy nhất cho quản trị viên. Các chức năng chính bao gồm: quản lý người dùng và phân quyền, theo dõi lịch sử đồng bộ qua Event Log, kích hoạt và giám sát tác vụ đồng bộ, quản lý Schema Registry, và thao tác với hệ thống backup.

Frontend giao tiếp hoàn toàn qua Kong API Gateway, không gọi trực tiếp vào backend. Biến `NEXT_PUBLIC_KONG_URL` được nhúng vào bundle tại thời điểm build — thay đổi địa chỉ yêu cầu build lại container.

3.3.2 Kong API Gateway

Kong là cổng vào duy nhất cho mọi yêu cầu từ frontend. Cấu hình được quản lý khai báo qua file `kong.yml` và đồng bộ vào Kong runtime thông qua công cụ `deck`. Các trách nhiệm của Kong:

- **Định tuyến:** Ánh xạ các đường dẫn (`/auth`, `/users`, `/integration`, v.v.) tới backend tại `http://backend:4000/api/v1`.
- **CORS:** Plugin CORS kiểm soát các origin được phép gọi API — quan trọng khi triển khai trên EC2 với Public IP cụ thể.
- **Metrics:** Plugin Prometheus thu thập số liệu về lưu lượng và độ trễ của từng route.

3.3.3 NestJS Backend (Monolith)

Backend là một ứng dụng NestJS duy nhất, được tổ chức thành các module độc lập. Mỗi module đóng gói controller, service và các dependency của riêng nó.

Auth Module và Users Module Hai module này hiện thực hóa cơ chế xác thực và phân quyền. Xác thực dựa trên JWT với Passport.js. Phân quyền theo mô hình vai trò hai cấp: vai trò cứng (`admin`, `reader`, `writer`) kết hợp phạm vi bảng dữ liệu linh hoạt qua `roleSettings` dạng regex pattern. Chi tiết được trình bày tại Chương **??**.

Schema Registry Module Lưu trữ metadata của từng bảng dữ liệu trong MongoDB, bao gồm: chiến lược đồng bộ (`syncStrategy`), khóa chính, danh sách trường, hash của cấu trúc hiện tại, trạng thái (`stable` / `changed` / `new`), và thời điểm đồng bộ gần nhất (`lastSyncTime`).

Mỗi lần pipeline chạy, Schema Registry so sánh hash cấu trúc trả về từ API nguồn với hash đang lưu. Nếu phát hiện thay đổi, bảng bị đánh dấu `changed`, emit event `schema.changed` để kích hoạt backup kiểm toán, và bị loại khỏi chu kỳ sync cho đến khi admin xem xét.

Data Integration Module Đây là module trung tâm, hiện thực hóa toàn bộ pipeline ETL. Khi được kích hoạt (thủ công hoặc qua lịch), module thực hiện tuần tự:

1. Lấy danh sách bảng `stable` từ Schema Registry.
2. Với mỗi bảng: tạo pre-sync backup, sau đó fetch dữ liệu từ API nguồn theo chiến lược đã cấu hình.
3. Ghi dữ liệu vào PostgreSQL thông qua Sync Engine.
4. Phát hiện orphan records (bản ghi tồn tại ở destination nhưng không còn ở source) nếu điều kiện cho phép.
5. Ghi kết quả vào Event Log và gửi email thông báo qua Notification Module.

Module hỗ trợ hai loại pipeline: **Full Sync** (toàn bộ bảng stable) và **Custom Sync** (danh sách bảng do admin chỉ định).

Job Scheduler Module Quản lý các tác vụ đồng bộ định kỳ. Mỗi tác vụ được lưu trong MongoDB với biểu thức cron, loại job (`FULL_SYNC`), trạng thái (`active/inactive`) và thời điểm chạy gần nhất.

Khi backend khởi động, toàn bộ tác vụ active được đăng ký động vào `SchedulerRegistry` của NestJS. Admin có thể tạo, sửa, bật/tắt và kích hoạt thủ công tác vụ qua API mà không cần restart server.

Backup Module Quản lý toàn bộ vòng đời backup: tạo snapshot (4 trigger), liệt kê, tải về, xóa và restore. Backup được lưu trên MinIO theo cấu trúc `{trigger}/{tableName}/{timestamp}.json`. Admin có thể đồng bộ thủ công lên AWS S3 để lưu trữ vĩnh viễn. Chi tiết được trình bày tại Chương **??**.

Event Log Module Ghi nhận mọi sự kiện đáng chú ý vào MongoDB: bắt đầu/kết thúc pipeline, thành công/thất bại của từng bảng, phát hiện orphan, restore backup. Event Log là nguồn dữ liệu chính để admin theo dõi lịch sử hoạt động của hệ thống.

Notification Module Gửi email cảnh báo qua SMTP (Gmail) khi tác vụ đồng bộ thất bại hoặc hoàn thành. Email chứa thông tin: tên tác vụ, thời gian, thời lượng thực thi, và chi tiết lỗi (nếu có). Module được đánh dấu `@Global` để có thể inject vào bất kỳ module nào mà không cần khai báo lại.

3.3.4 Tầng Lưu Trữ

PostgreSQL (qua Prisma ORM) Cơ sở dữ liệu quan hệ chính, lưu trữ toàn bộ dữ liệu nghiệp vụ đã được đồng bộ từ các hệ thống nguồn, cùng với thông tin người dùng và cấu hình retention policy. Prisma ORM đảm bảo type-safety và quản lý migration có kiểm soát.

MongoDB (qua Mongoose) Cơ sở dữ liệu document, lưu trữ ba loại dữ liệu vận hành: Schema Registry (metadata và cấu hình đồng bộ của từng bảng), Event Log (nhật ký hoạt động), và Scheduled Jobs (cấu hình tác vụ định kỳ). MongoDB Atlas được sử dụng dưới dạng dịch vụ cloud — không cần triển khai và quản lý MongoDB server.

MinIO và AWS S3 Lưu trữ backup theo kiến trúc hai tầng: MinIO (self-hosted, độ trễ thấp) và AWS S3 (cloud, lưu trữ vĩnh viễn). Chi tiết tại Chương **??**.

3.3.5 Hệ Thống Nguồn Dữ Liệu

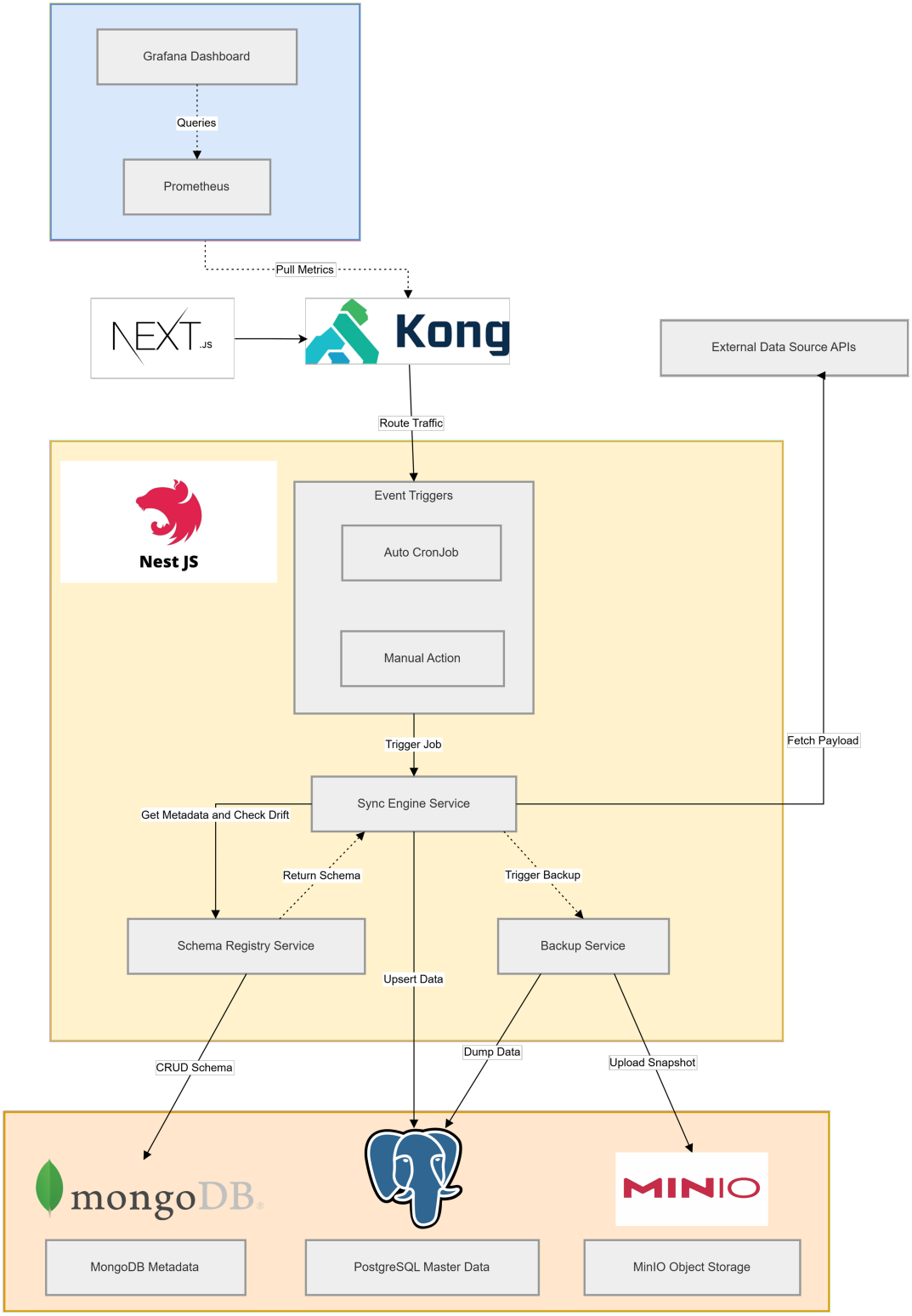
Trong phạm vi dự án, hệ thống nguồn là một REST API cung cấp dữ liệu sinh viên qua giao thức HTTP. API trả về dữ liệu theo định dạng chuẩn:

```
{
  "success": true,
  "metadata": { "totalRecords": 68411, "totalPages": 14 },
  "payload": [ { "id": 1, ... } ]
}
```

Hệ thống tích hợp đóng vai trò consumer thuần túy — chỉ đọc dữ liệu qua API, không kết nối trực tiếp vào cơ sở dữ liệu nguồn.

3.3.6 Giám Sát: Prometheus và Grafana

Prometheus thu thập metrics từ Kong (qua plugin) và backend (qua endpoint `/metrics`). Grafana hiển thị các dashboard theo dõi: số lượng request, tỷ lệ lỗi, độ trễ phản hồi, và



Hình 3.2: Tương tác giữa các module backend và với các thành phần bên ngoài

trạng thái các container. Grafana được cấu hình gửi cảnh báo qua email khi phát hiện bất thường.

3.4 So Sánh Với Kiến Trúc Lý Thuyết Ban Đầu

Bảng 3.1: So sánh kiến trúc lý thuyết và thực tế

Khía cạnh	**Lý thuyết**	**Thực tế**
Mô hình	API-led (3 lớp riêng biệt)	Monolith module hóa (NestJS)
Data Sharing Server	Server riêng (Process + Experience API)	Tích hợp vào backend monolith
Data Integration Server	Server riêng (System API)	Module trong cùng backend
Phân quyền	RBAC đầy đủ (5 bảng)	Role enum + roleSettings JSON (regex)
Nguồn dữ liệu	Nhiều CSDL (Đào tạo, Tài chính, Nhân sự)	Một REST API nguồn
Backup	Không đề cập	MinIO + AWS S3, 4 trigger
Schema quản lý	Không đề cập	Schema Registry (MongoDB)
Cảnh báo	Không đề cập	Email alert qua SMTP
Triển khai	Không đề cập	Docker Compose trên AWS EC2

Sự khác biệt lớn nhất so với thiết kế lý thuyết là việc hợp nhất Data Sharing Server và Data Integration Server thành một backend duy nhất. Quyết định này được đưa ra vì: (1) số lượng nguồn dữ liệu và lưu lượng hiện tại chưa đủ để biện minh cho độ phức tạp vận hành của microservices; (2) kiến trúc module hóa của NestJS đã cung cấp sự tách biệt trách nhiệm đủ tốt trong phạm vi một process duy nhất; (3) dễ dàng di chuyển sang microservices trong tương lai bằng cách tách từng module thành service độc lập khi cần.

4 Những vấn đề trong tích hợp dữ liệu và cách giải quyết

4.1 Các dạng xung đột dữ liệu thường gặp

Trong quá trình xây dựng trục tích hợp dữ liệu, hệ thống phải đối mặt với nhiều dạng xung đột phát sinh tại ranh giới giữa các hệ thống nguồn phân tán và cơ sở dữ liệu chủ (Master Data). Phần này trình bày các dạng xung đột đã được nhận diện và cơ chế phòng thủ tương ứng đã được hiện thực trong dự án.

1. **Xung đột cấu trúc phản hồi API**

   Dạng xung đột này xảy ra khi hệ thống nguồn trả về dữ liệu không tuân thủ định dạng chuẩn mà trục tích hợp kỳ vọng. Đây là dạng xung đột ở tầng tiếp nhận dữ liệu — nguy hiểm nhất vì nếu không được phát hiện sớm, dữ liệu sai định dạng sẽ lan truyền sâu vào luồng xử lý, gây ra các lỗi rất khó truy vết.

   Trong dự án, cấu trúc phản hồi (response) chuẩn được quy định như sau:

```
{
  "success": true,
  "metadata": { "totalPages": N, "currentPage": P, ... },
  "payload": [ { ... }, { ... } ]
}
```

   Trong đó:

   - `success`: Cho biết yêu cầu có được xử lý thành công hay không.
   - `timestamp`: Thời điểm hệ thống trả về dữ liệu.
   - `metadata`: Chứa thông tin mô tả như tên bảng dữ liệu, số lượng bản ghi, số trang và trang hiện tại.
   - `payload`: Chứa dữ liệu thực tế được trả về.

   Các vi phạm hợp đồng giao tiếp phổ biến được ghi nhận bao gồm:

   - Thiếu trường `success` hoặc `success = false` — hệ thống nguồn đang trong trạng thái lỗi nội bộ.

- Trường `payload` không tồn tại hoặc không phải kiểu mảng (`Array`) — lỗi tuần tự hóa (serialization) ở hệ thống nguồn.
- Sử dụng tên trường khác (`data`, `records`) thay vì `payload` — sự thiếu thống nhất giữa các hệ thống nguồn.

2. **Xung đột lược đồ dữ liệu**

   Xung đột lược đồ (schema) xảy ra khi hệ thống nguồn thay đổi cấu trúc bảng (thêm, sửa, xóa cột) mà không thông báo cho trục tích hợp. Hậu quả trực tiếp là các trường không xác định được đưa vào Prisma ORM gây lỗi *"Unknown field"*, hoặc ngược lại, các trường mới ở nguồn không được ánh xạ vào cơ sở dữ liệu chủ dẫn đến việc mất mát dữ liệu một cách thầm lặng.

3. **Xung đột kiểu dữ liệu**

   Xung đột kiểu dữ liệu phát sinh do sự không đồng nhất trong cách các hệ thống nguồn biểu diễn cùng một giá trị. Các trường hợp thực tế ghi nhận trong dự án bao gồm:

Bảng 4.1: Các trường hợp xung đột kiểu dữ liệu điển hình

Trường	Kiểu Prisma	Nguồn gửi	Hậu quả nếu không xử lý
`id`	`Int`	`"701"` (chuỗi)	Lỗi kiểu dữ liệu Prisma
`active`	`Boolean`	`"true"` (chuỗi)	Lỗi kiểu dữ liệu Prisma
`ngaySinh`	`DateTime`	`"1990-06-15T..."` (chuỗi)	Lỗi kiểu dữ liệu Prisma
`ngaySinh`	`DateTime`	`"not-a-date"` (chuỗi)	Crash hệ thống (Runtime)

4. **Xung đột khóa chính**

   Xung đột khóa chính xảy ra dưới hai hình thức:

   (a) **Trùng lặp trong cùng một lô dữ liệu (batch):** Hệ thống nguồn trả về nhiều bản ghi có cùng giá trị khóa chính trong một lần phân trang. Nếu lần lượt thực hiện lệnh `upsert` theo thứ tự xuất hiện, các bản ghi đầu sẽ bị ghi đè bởi bản ghi sau mà không có cảnh báo.

   (b) **Thiếu hoặc rỗng khóa chính:** Bản ghi không mang giá trị khóa chính hợp lệ, không thể thực hiện `upsert` và nếu không được xử lý sẽ bị bỏ qua một cách thầm lặng.

5. **Xung đột phân trang**

   Đây là dạng xung đột đặc thù của kiến trúc tích hợp theo lô. Khi luồng xử lý gửi yêu cầu lấy trang thứ p, hệ thống nguồn có thể trả về dữ liệu của một trang khác do lỗi bộ nhớ đệm (cache CDN) hoặc lỗi định tuyến của bộ cân bằng tải. Hậu quả là dữ liệu ở các trang bị bỏ qua, trong khi luồng xử lý không nhận ra sự sai lệch và vẫn báo cáo thành công.

6. **Xung đột ràng buộc cơ sở dữ liệu**

   Xung đột ràng buộc phát sinh khi dữ liệu từ nguồn vi phạm các quy tắc toàn vẹn được định nghĩa trong cơ sở dữ liệu chủ, ví dụ: giá trị vượt độ dài `VARCHAR`, vi phạm ràng buộc `NOT NULL`, hay vi phạm ràng buộc khóa ngoại. Nếu một bản ghi bị lỗi gây hủy (abort) toàn bộ lô dữ liệu, những bản ghi hợp lệ còn lại trong cùng lô đó sẽ không được đồng bộ.

4.2 Cách giải quyết

Để giải quyết sự đa dạng về cấu trúc dữ liệu giữa các hệ thống, trục tích hợp xây dựng một bộ đặc tả dữ liệu thống nhất và lưu trữ trong MongoDB. Bộ đặc tả này quy định rõ tên trường, kiểu dữ liệu, các ràng buộc bắt buộc, giá trị mặc định và mối liên hệ giữa các trường dữ liệu.

Khi một đơn vị gửi dữ liệu lên trục tích hợp, dữ liệu đó bắt buộc phải tuân thủ đúng cấu trúc đã được định nghĩa trước. Nếu dữ liệu không phù hợp, hệ thống sẽ từ chối tiếp nhận và trả về thông báo lỗi tương ứng. Nhờ đó, các phòng ban buộc phải sử dụng cùng một chuẩn dữ liệu, giúp hạn chế sự phân mảnh giữa các hệ thống, đồng thời hỗ trợ việc kiểm tra và phát hiện lỗi dễ dàng hơn.

Listing 1: Ví dụ đặc tả dữ liệu tổng quát trong MongoDB

```
{
  "tableName": "ten_bang",
  "dataFrom": "ten_he_thong",
  "dataFromApi": "/duong_dan_api",
  "dataFromMethod": "GET",
  "description": "Mo ta bang du lieu",

  "details": [
    {
```

```
      "name": "ten_truong",
      "type": "kieu_du_lieu",
      "length": 10
    },
    {
      "name": "trang_thai",
      "type": "boolean",
      "length": null
    }
  ],

  "primaryKey": ["khoa_chinh"],

  "fieldsCount": 2,
  "recordsCount": 100,

  "status": "stable",
  "syncStrategy": "overwrite"
}
```

Hệ thống hiện thực chiến lược phòng thủ đa tầng được chia thành hai lớp xử lý tương ứng với hai thành phần cốt lõi của luồng xử lý.

4.2.1 Tầng tiếp nhận dữ liệu — `DataIntegrationService`

Tầng tiếp nhận chịu trách nhiệm kiểm tra và lọc dữ liệu ngay tại điểm giao tiếp, trước khi bất kỳ bản ghi nào được đưa vào các bước xử lý sâu hơn.

Kiểm tra cấu trúc phản hồi API (Fail-Fast Validation). Mọi phản hồi từ hệ thống nguồn đều được kiểm tra bốn điều kiện bắt buộc ngay sau khi nhận:

```
if (!responseData
    || responseData.success !== true
    || !responseData.payload
    || !Array.isArray(responseData.payload)) {
  throw new Error('[API CONTRACT VIOLATION]...');
}
```

Chiến lược thất bại nhanh (*fail-fast*) đảm bảo rằng dữ liệu sai định dạng bị từ chối ngay tại điểm tiếp nhận, không lan truyền vào các tầng xử lý phía sau. Bảng dữ liệu bị lỗi được ghi nhận và hệ thống tiếp tục xử lý các bảng kế tiếp.

Bộ lọc trạng thái lược đồ (Schema Status Guard). Trước mỗi phiên đồng bộ, hệ thống kiểm tra trạng thái lược đồ của từng bảng trong MongoDB. Chỉ các bảng có trạng thái `stable` mới được phép đồng bộ. Các bảng ở trạng thái `changed` hoặc `new` sẽ bị bỏ qua và cảnh báo được ghi vào nhật ký, chờ quản trị viên xem xét và phê duyệt thủ công.

Bộ lọc trang lệch (Page Mismatch Guard). Sau mỗi lần nhận phản hồi phân trang, hệ thống so sánh số trang thực tế trong siêu dữ liệu (`metadata`) với số trang đang được yêu cầu. Nếu phát hiện sai lệch, vòng lặp phân trang bị dừng ngay lập tức để tránh ghi đè dữ liệu sai:

```
if (meta.currentPage && meta.currentPage !== currentPage) {
  // [PAGE MISMATCH] Expected page P, got Q. Skipping.
  break;
}
```

Bộ lọc dữ liệu rỗng. Nếu hệ thống nguồn trả về mảng `payload` rỗng trước khi đạt tới tổng số trang đã khai báo, vòng lặp phân trang sẽ dừng sớm để tránh lặp vô hạn do lỗi logic phân trang từ phía nguồn.

4.2.2 Tầng đồng bộ — `SyncEngineService`

Tầng đồng bộ xử lý từng lô dữ liệu đã qua tầng tiếp nhận, áp dụng các cơ chế làm sạch và chuẩn hóa trước khi ghi vào PostgreSQL.

Lọc trường dữ liệu không xác định. Hệ thống truy vấn Prisma DMMF (*Data Model Meta Format*) tại thời gian thực thi (runtime) để lấy danh sách các trường hợp lệ của từng bảng. Bất kỳ trường nào từ nguồn không nằm trong danh sách này đều bị loại bỏ trước khi tiến hành ghi cơ sở dữ liệu:

```
const validFields = this.getValidScalarFields(modelName);
const filteredRecord = Object.fromEntries(
  Object.entries(record).filter(([k]) => validFields.has(k))
);
```

Cơ chế này loại bỏ hoàn toàn lỗi *"Unknown field"* của Prisma mà không cần can thiệp thủ công khi hệ thống nguồn tùy tiện thêm cột mới.

Loại bỏ trùng lặp khóa chính. Trước khi `upsert`, toàn bộ lô dữ liệu được xử lý qua hàm loại bỏ trùng lặp sử dụng cấu trúc `Map` với từ khóa là giá trị khóa chính. Bản ghi xuất hiện sau cùng với cùng khóa chính sẽ được giữ lại. Các bản ghi có khóa chính `null` hoặc `undefined` không bị loại bỏ tại bước này mà được chuyển tiếp sang vòng lặp `upsert` để cố ý tạo ra lỗi và được ghi nhận vào nhật ký lỗi.

Chuẩn hóa kiểu dữ liệu. Dựa trên siêu dữ liệu từ Prisma DMMF, hệ thống tự động chuyển đổi các giá trị sai kiểu từ nguồn về kiểu dữ liệu đúng:

Bảng 4.2: Quy tắc chuẩn hóa kiểu dữ liệu

Kiểu Prisma	Nguồn gửi	Sau chuẩn hóa
`DateTime`	`"1990-06-15T00:00:00Z"`	Đối tượng `Date`
`DateTime`	`"not-a-date"`	`null`
`Int`	`"701"`	`701`
`Float`	`"3.14"`	`3.14`
`Boolean`	`"true"` / `"1"`	`true`
`Boolean`	`"false"`	`false`

Cập nhật bất biến (Idempotent Upsert). Thay vì thực hiện `INSERT` hoặc `UPDATE` riêng lẻ, hệ thống sử dụng cơ chế `upsert` của Prisma cho mọi bản ghi. Với cơ chế này, việc chạy lại luồng xử lý nhiều lần trên cùng một tập dữ liệu không tạo ra các bản ghi trùng lặp mà chỉ cập nhật thông tin mới nhất, đảm bảo tính bất biến của hệ thống.

Nhật ký lỗi (Dead Letter Log). Mọi bản ghi gây lỗi trong quá trình `upsert` — dù là thiếu khóa chính, vi phạm ràng buộc `VARCHAR`, hay lỗi cơ sở dữ liệu khác — đều được bắt giữ riêng lẻ trong khối `try/catch` và ghi vào mảng `errors` của nhật ký sự kiện trong MongoDB. Luồng xử lý không bị hủy; các bản ghi hợp lệ còn lại trong cùng lô vẫn tiếp tục được đồng bộ. Quản trị viên có thể truy vết chính xác từng bản ghi lỗi thông qua giá trị khóa chính và thông báo lỗi được lưu kèm.

4.2.3 Tổng hợp cơ chế phòng thủ

Bảng 4.3: Tổng hợp các cơ chế phòng thủ xung đột dữ liệu

Dạng xung đột	**Vị trí xử lý**	**Cơ chế giải quyết**
Vi phạm cấu trúc phản hồi	`DataIntegration`	Fail-fast validation, từ chối ngay tại đầu vào
Lược đồ thay đổi chưa duyệt	`DataIntegration`	Bộ lọc trạng thái lược đồ (`status = stable`)
Trang phân trang lệch	`DataIntegration`	Bộ lọc trang lệch, dừng vòng lặp
Dữ liệu rỗng bất thường	`DataIntegration`	Bộ lọc dữ liệu rỗng, dừng vòng lặp
Trường lạ từ nguồn	`SyncEngineService`	Lọc trường hợp lệ qua Prisma DMMF
Trùng khóa chính trong lô	`SyncEngineService`	Loại bỏ trùng lặp bằng cấu trúc `Map`
Sai kiểu dữ liệu	`SyncEngineService`	Chuẩn hóa kiểu dữ liệu qua Prisma DMMF
Chạy lại luồng xử lý	`SyncEngineService`	Cơ chế `upsert` bất biến
Lỗi bản ghi đơn lẻ	`SyncEngineService`	Lưu nhật ký lỗi (*Dead Letter Log*) vào MongoDB

5 Cơ chế xác thực và phân quyền

5.1 Luồng xác thực

Xác thực là quá trình kiểm tra và xác minh danh tính của người dùng trước khi cho phép truy cập vào hệ thống. Trục tích hợp dữ liệu sử dụng JSON Web Token (JWT) kết hợp với thư viện Passport.js để thực hiện cơ chế xác thực phi trạng thái (*stateless*).

Quy trình xác thực diễn ra theo các bước sau:

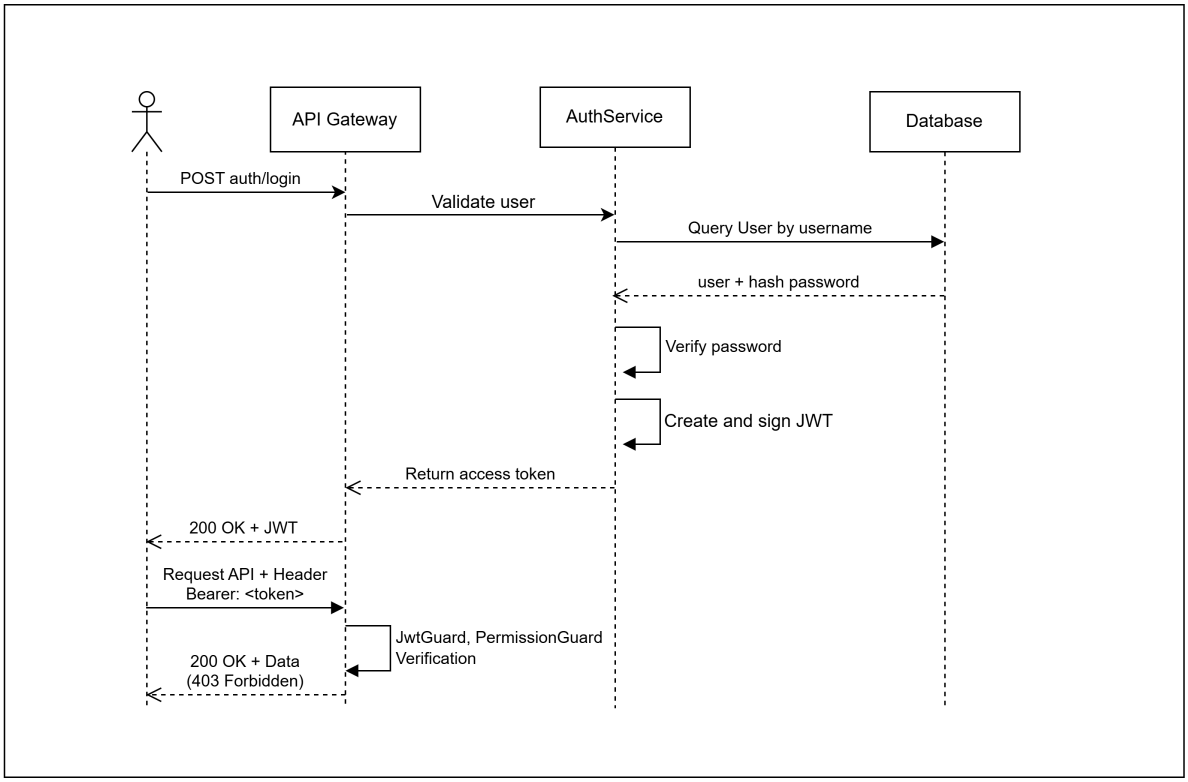
1. Người dùng gửi yêu cầu đăng nhập kèm theo tên đăng nhập và mật khẩu đến điểm cuối API `POST /auth/login`.
2. Dịch vụ xác thực truy vấn bảng `users` trong PostgreSQL để tìm người dùng theo tên đăng nhập. Mật khẩu được xác minh thông qua hàm `bcrypt.compare` bằng cách đối chiếu với giá trị băm đã lưu trong cột `passwordHash`.
3. Nếu thông tin hợp lệ, hệ thống khởi tạo một mã truy cập JWT với phần dữ liệu tải trọng (payload) chứa các thông tin sau:
   - `sub`: Định danh người dùng (ID).
   - `username`: Tên đăng nhập.
   - `role`: Vai trò của người dùng (`admin` | `reader` | `writer`).
   - `roleSettings`: Cấu hình phạm vi quyền hạn dưới định dạng JSON (được trình bày chi tiết tại Mục 5.2).
4. Mã truy cập được ký số bằng khóa bí mật `JWT_SECRET` và có thời gian tồn tại được quy định thông qua biến môi trường `JWT_EXPIRES_IN`.
5. Sau khi nhận được mã truy cập, người dùng đính kèm mã này vào mọi yêu cầu truy xuất tài nguyên tiếp theo. Hệ thống hỗ trợ hai hình thức truyền mã truy cập:
   - Qua Tiêu đề HTTP (HTTP Header):

     ```
     Authorization: Bearer <token>
     ```

   - Qua tham số truy vấn (URL Query - áp dụng cho các kết nối luồng dữ liệu liên tục như SSE):

     ```
     GET /integration/stream-logs?token=<token>
     ```

Việc sử dụng JWT giúp hệ thống tuân thủ hoàn toàn kiến trúc phi trạng thái — máy chủ không cần lưu trữ thông tin phiên làm việc (session); mọi thông tin cần thiết để xác thực và phân quyền đều được nhúng gọn bên trong mã thông báo.



Hình 5.1: Luồng xác thực và kiểm tra quyền truy cập

5.2 Luồng phân quyền

Sau khi hệ thống xác thực người dùng thành công, luồng yêu cầu tiếp tục đi qua hai lớp bảo vệ độc lập để thực hiện kiểm tra quyền truy cập một cách tuần tự.

5.2.1 Lớp 1 — Kiểm tra vai trò (`RolesGuard`)

Hệ thống định nghĩa ba vai trò cố định, được lưu trực tiếp trên bảng `users`:

- `admin`: Quản trị viên hệ thống, có toàn quyền truy cập tài nguyên và được đặc cách bỏ qua mọi bước kiểm tra phân quyền phía sau.
- `reader`: Người dùng chỉ có quyền đọc dữ liệu trong phạm vi được chỉ định.
- `writer`: Người dùng có quyền cả đọc và ghi dữ liệu trong phạm vi được chỉ định.

`RolesGuard` chịu trách nhiệm đọc danh sách vai trò cho phép từ bộ trang trí (*decorator*) `@Roles()` gắn trên các trình điều khiển (controller) hoặc từng điểm cuối cụ thể. Sau đó, hệ thống đối chiếu với trường `role` lấy từ tải trọng JWT. Nếu không khớp, yêu cầu sẽ bị từ chối ngay lập tức với mã lỗi 403 Forbidden.

5.2.2 Lớp 2 — Kiểm tra phạm vi bảng (`PermissionsGuard`)

Ngoài vai trò, mỗi người dùng (ngoại trừ `admin`) được cấp một cấu hình `roleSettings` lưu dưới định dạng JSON trong cơ sở dữ liệu:

Listing 2: Cấu trúc cấu hình roleSettings

```
{
  "writeScopes": ["^tcns_.*", "^dm_gioi_tinh$"],
  "readScopes":  ["^nguoi_hoc$", "^dm_.*"]
}
```

Mỗi phần tử trong mảng `writeScopes` và `readScopes` là một biểu thức chính quy (*regular expression*) dùng để so khớp với tên bảng được truyền qua tham số động `:table` của URL yêu cầu. Logic kiểm tra được thực thi như sau:

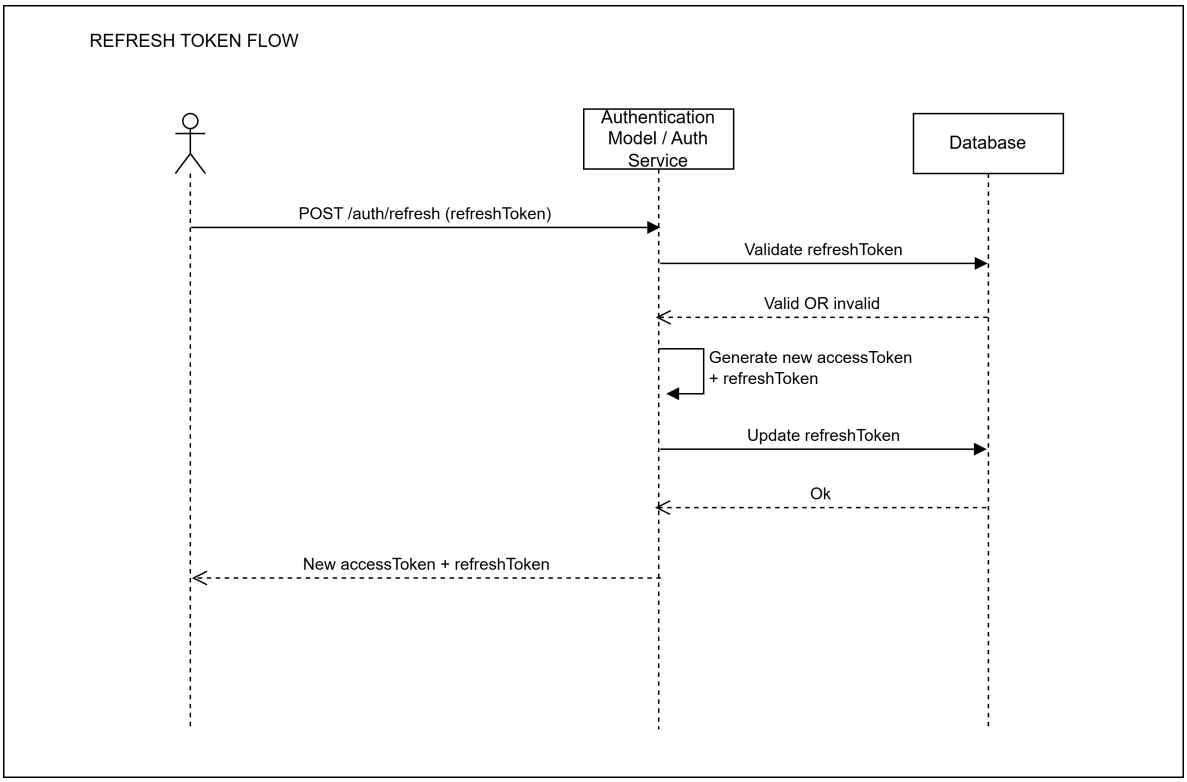
- **Admin**: Bỏ qua hoàn toàn bước kiểm tra này.
- **Thao tác đọc** (Phương thức `GET`):
  - `reader`: Tên bảng phải khớp với ít nhất một biểu thức trong `readScopes`.
  - `writer`: Tên bảng phải khớp với ít nhất một biểu thức trong `readScopes` **hoặc** `writeScopes`.
- **Thao tác ghi** (Phương thức `POST`, `PUT`, `PATCH`, `DELETE`): Chỉ tài khoản `writer` được phép thực hiện, và tên bảng bắt buộc phải khớp với ít nhất một biểu thức trong `writeScopes`.

Việc ứng dụng biểu thức chính quy mang lại tính linh hoạt rất cao trong cấu hình. Ví dụ: biểu thức `tcns_.*` cấp quyền truy cập hàng loạt cho toàn bộ các bảng có tiền tố `tcns_`, trong khi biểu thức `dm_gioi_tinh$` giới hạn quyền truy cập chặt chẽ vào đúng một bảng duy nhất.

5.3 Cơ chế cấp lại mã truy cập

Khi mã truy cập ngắn hạn hết hiệu lực, người dùng chỉ cần gửi lại mã cũ đến điểm cuối `POST /auth/refresh-token`. Hệ thống sẽ xác minh chữ ký và tính toàn vẹn của mã thông qua phương thức `jwtService.verify()`, sau đó trích xuất dữ liệu gốc để ký lại và phát hành một mã truy cập hoàn toàn mới.

Cơ chế này giữ được sự tối giản về mặt logic và không yêu cầu phải lưu trữ thêm *Refresh Token* trong cơ sở dữ liệu, đáp ứng tốt yêu cầu về mô hình phi trạng thái của toàn bộ hệ thống.



Hình 5.2: Luồng xử lý cấp lại mã truy cập (Refresh Token)

5.4 Cấu trúc cơ sở dữ liệu phục vụ xác thực và phân quyền

Khác với mô hình kiểm soát truy cập dựa trên vai trò truyền thống (RBAC - Role-Based Access Control) thường đòi hỏi nhiều bảng trung gian phức tạp, hệ thống trục tích hợp đã hợp nhất toàn bộ thông tin xác thực và phân quyền vào một bảng duy nhất:

- `users`: Lưu trữ thông tin định danh bao gồm tên đăng nhập, mật khẩu đã băm

(`passwordHash`), họ tên, thư điện tử, kết hợp với vai trò hệ thống (`role`) và cấu hình phạm vi quyền (`roleSettings`).

Lối thiết kế này đạt được sự cân bằng tối ưu giữa tính đơn giản và độ linh hoạt: trường vai trò kiểu liệt kê (enum) đảm bảo tính nhất quán cho các thao tác cấp cao, trong khi trường `roleSettings` lưu dạng JSON cho phép tùy biến phạm vi quyền chi tiết đến từng bảng cụ thể mà không cần phải thực hiện di cư dữ liệu (migration) làm thay đổi cấu trúc bảng.

6 Luồng xử lý dữ liệu

6.1 Chiến lược đồng bộ dữ liệu và cơ chế ghi vào cơ sở dữ liệu

6.1.1 Tổng quan

Sau khi dữ liệu được lấy về từ API nguồn, hệ thống cần quyết định *cách thức ghi* vào cơ sở dữ liệu đích thông qua Prisma ORM. Thay vì áp dụng một phương thức duy nhất cho mọi bảng, hệ thống thiết kế ba chiến lược riêng biệt, mỗi chiến lược phù hợp với đặc thù của từng loại dữ liệu:

- `upsert` — chiến lược mặc định, phù hợp với các bảng nghiệp vụ lớn, dữ liệu có sự thay đổi thường xuyên.
- `incremental` — chiến lược đồng bộ tăng tiến, giúp giảm tải băng thông mạng và cơ sở dữ liệu bằng cách chỉ yêu cầu và ghi nhận những bản ghi có sự thay đổi kể từ lần đồng bộ gần nhất.
- `overwrite` — chiến lược ghi đè toàn bộ, phù hợp với các bảng danh mục nhỏ, cần đảm bảo hệ thống đích hoàn toàn đồng nhất 100% với hệ thống nguồn.

Chiến lược được cấu hình tĩnh trong *Schema Registry* (MongoDB) tại trường `syncStrategy`, và được áp dụng tự động tại hàm `syncOneTable()` trong mã nguồn. Thứ tự ưu tiên khi chọn chiến lược diễn ra như sau:

$$\texttt{schema.syncStrategy} \xrightarrow{\text{nếu không có}} \texttt{upsert} \quad \text{(mặc định)}$$

Mọi thay đổi về chiến lược đều được áp dụng ngay ở lần đồng bộ tiếp theo mà không cần khởi động lại dịch vụ, nhờ vào cơ chế truy vấn Schema Registry tại thời gian thực thi.

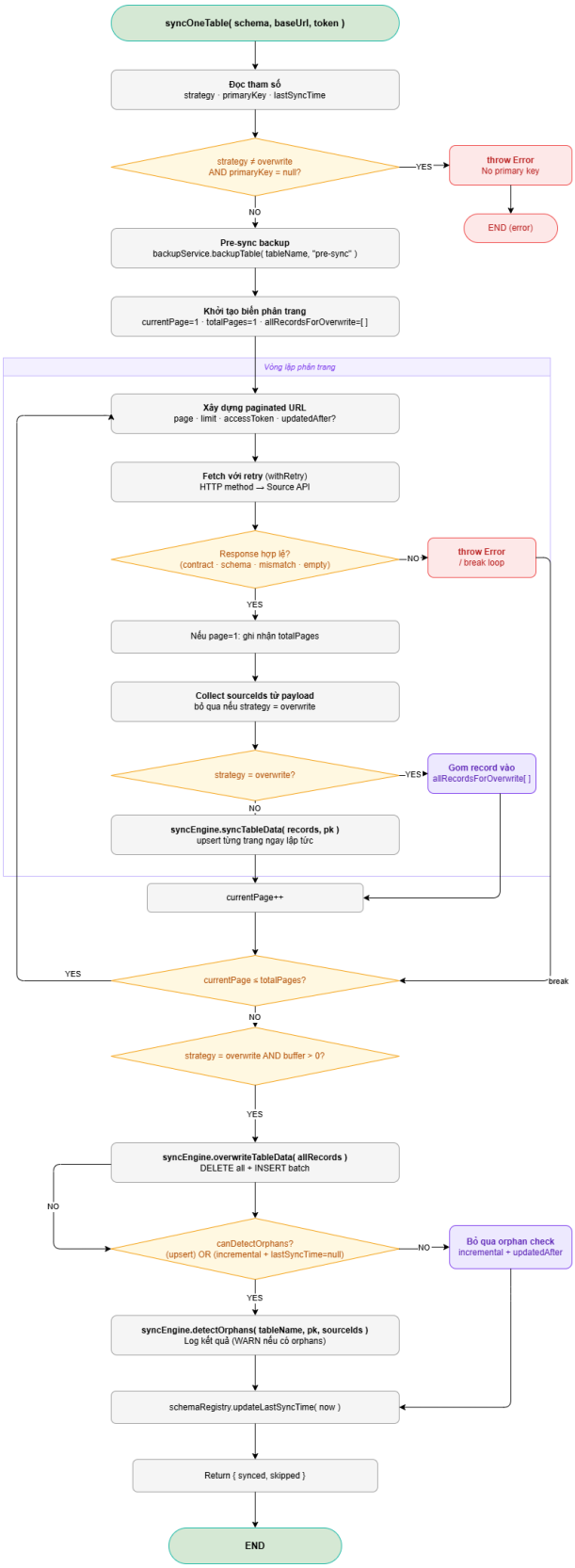
6.1.2 Quy trình chung: Phân trang và tải dữ liệu

Bất kể áp dụng chiến lược nào, hệ thống đều thực hiện **cơ chế phân trang** khi gọi API nguồn, với kích thước mỗi trang cố định là 5.000 bản ghi (`BATCH_LIMIT = 5000`). URL truy vấn được tạo động theo biểu mẫu:

```
{baseUrl}{dataFromApi}?page={currentPage}&limit=5000
    [&accessToken={token}]
    [&updatedAfter={lastSyncTime}]   ← chỉ áp dụng khi incremental
```

Hệ thống tiến hành đọc giá trị `metadata.totalPages` từ phản hồi của trang đầu tiên để xác định tổng số trang cần duyệt, sau đó lặp cho đến khi tải hết toàn bộ. Mỗi lần tải đều được bao bọc trong cơ chế **thử lại (retry)** với tối đa 3 lần thao tác nhằm đối phó với lỗi đường truyền mạng.

Hình mô tả luồng xử lý tổng quát của một bảng dữ liệu, từ thời điểm bắt đầu cho đến khi cập nhật trường `lastSyncTime`.



Hình 6.1: Cách thức hoạt động của hàm syncOneTable

6.1.3 Chiến lược `upsert` — Mặc định

Nguyên lý hoạt động Đây là chiến lược **an toàn và mang tính tổng quát** cao nhất. Với mỗi trang dữ liệu nhận được, hệ thống sẽ tiến hành `upsert` toàn bộ các bản ghi vào PostgreSQL thông qua Prisma:

- Nếu bản ghi chứa `primaryKey` đã tồn tại → thực hiện lệnh **UPDATE** cho các trường còn lại.
- Nếu bản ghi chưa tồn tại → thực hiện lệnh **INSERT** bản ghi mới.

Thao tác này đảm bảo tính **bất biến (idempotent)**: hệ thống có thể chạy lại nhiều lần với cùng một tập dữ liệu đầu vào mà vẫn cho ra cùng một kết quả cuối cùng, không gây ra hiện tượng trùng lặp hay mất mát dữ liệu.

Chi tiết luồng ghi cơ sở dữ liệu Trước khi gọi lệnh `upsert`, mỗi lô bản ghi bắt buộc phải vượt qua bốn lớp tiền xử lý:

1. **Lọc trường dữ liệu (DB Safety Net)** — Loại bỏ các trường không nằm trong định nghĩa lược đồ Prisma. Điều này giúp ngăn chặn lỗi do hệ thống nguồn tự ý bổ sung cột mới mà chưa được thực thi migration trên PostgreSQL.
2. **Loại bỏ trùng lặp (Deduplication)** — Nếu dữ liệu nguồn trả về nhiều bản ghi có cùng khóa chính trong cùng một trang, hệ thống chỉ giữ lại bản ghi *xuất hiện sau cùng*. Các bản ghi có khóa chính bị `null` không bị loại bỏ ở bước này, mà sẽ được giữ lại để đẩy vào nhật ký lỗi (Dead Letter Log).
3. **Chuẩn hóa kiểu dữ liệu (Type Normalization)** — Chuyển đổi định dạng từ chuẩn JSON về đúng kiểu dữ liệu khai báo trong Prisma:
   - `DateTime`: chuỗi ISO 8601 → Đối tượng `Date`.
   - `Int` / `Float`: chuỗi chữ số → Kiểu số thực.
   - `Boolean`: `"true"` / `"1"` → `true`.
4. **Upsert bất biến và lưu nhật ký lỗi** — Thực hiện lệnh `prisma[model].upsert()` trong vòng lặp. Nếu một bản ghi bị lỗi (vi phạm ràng buộc dữ liệu duy nhất, sai kiểu...), thông báo lỗi được ghi chú vào *Dead Letter Log* trong MongoDB thay vì bắt buộc dừng toàn bộ tiến trình.

Sau khi hoàn tất, kết quả tổng quan sẽ được ghi vào nhật ký sự kiện MongoDB với các chỉ số báo cáo chi tiết: tổng số bản ghi cần đồng bộ, số lượng bản ghi bị loại bỏ trùng lặp, số bản ghi thành công/thất bại, và mảng chi tiết chứa nguyên nhân lỗi của từng bản ghi.

6.1.4 Chiến lược `incremental` — Bộ lọc tại phía máy chủ

Nguyên lý hoạt động Chiến lược `incremental` tận dụng khả năng **lọc dữ liệu từ phía API nguồn** thông qua việc đính kèm tham số `updatedAfter`, yêu cầu nguồn chỉ truyền trả những bản ghi đã có sự thay đổi kể từ lần đồng bộ cuối cùng. Cách tiếp cận này giúp tối ưu hóa hiệu năng trên cả hai khía cạnh:

- **Băng thông mạng**: Hệ thống nguồn chỉ trả về một tập con các bản ghi thay vì toàn bộ dữ liệu bảng.
- **Tải cơ sở dữ liệu**: Hệ thống đích chỉ cần thực hiện `upsert` đúng số lượng bản ghi thực sự mới hoặc bị thay đổi.

Mốc thời gian dùng để tạo bộ lọc là trường `lastSyncTime` trong tài liệu cấu hình (Schema Registry), được hệ thống cập nhật tự động sau mỗi lần đồng bộ thành công.

Quản lý tham số `updatedAfter` Tham số `updatedAfter` được gắn trực tiếp vào URL truy vấn khi và chỉ khi chiến lược được đặt là `incremental` *và* bảng đã được đồng bộ ít nhất một lần trước đó (`lastSyncTime` khác `null`).

Listing 3: Thêm `updatedAfter` vào URL phân trang

```
const updatedAfterParam =
  strategy === 'incremental' && lastSyncTime
    ? `&updatedAfter=${encodeURIComponent(lastSyncTime.
      toISOString())}`
    : '';
```

API nguồn sau khi tiếp nhận tham số này sẽ tiến hành lọc và chỉ trả về các bản ghi thỏa mãn điều kiện `updatedAt > lastSyncTime`. Toàn bộ số lượng bản ghi trả về sẽ được hệ thống tiến hành `upsert` thẳng vào cơ sở dữ liệu mà không cần phải thực hiện bước kiểm tra điều kiện thêm một lần nữa.

Trường hợp đồng bộ lần đầu tiên Khi `lastSyncTime = null` (tức bảng chưa từng được đồng bộ), tham số `updatedAfter` **sẽ không được thêm** vào URL truy vấn. Khi đó, API nguồn sẽ trả về toàn bộ dữ liệu hiện có — hành vi này tương đương với phương thức `upsert` (tải toàn phần). Đây là một cơ chế **khởi tạo ban đầu (bootstrapping)** tự động, giúp quản trị viên không cần phải thiết lập cấu hình thủ công cho lần chạy đầu tiên.

Bảng 6.1: Hiệu quả Incremental: Lọc tại đích (BE-side) và lọc tại nguồn (Server-side)

Tiêu chí	Lọc tại hệ thống đích	Lọc tại hệ thống nguồn (Mới)
Dữ liệu qua mạng	Toàn bộ bảng (100%)	Chỉ tải bản ghi có thay đổi
Số trang cần tải	`totalPages` đầy đủ	Giảm tỷ lệ thuận theo thay đổi
Xử lý tại backend	Lọc `updatedAt` cho từng dòng	Upsert thẳng vào DB
Lượt ghi vào DB	Chỉ ghi bản ghi thay đổi	Chỉ ghi bản ghi thay đổi
Lợi thế cốt lõi	Giảm thao tác ghi DB	Giảm cả thao tác ghi DB và băng thông mạng

So sánh hiệu năng chiến lược Incremental

6.1.5 Chiến lược `overwrite` — Xóa và ghi đè toàn bộ

Nguyên lý hoạt động Chiến lược `overwrite` được thiết kế đặc thù cho các **bảng danh mục (dimension tables)** với kích thước nhỏ, nơi có yêu cầu cốt lõi là: hệ thống đích phải *hoàn toàn giống hệt* hệ thống nguồn sau mỗi lần chạy. Thay vì cập nhật riêng lẻ từng dòng, hệ thống thực hiện các bước sau:

1. **Tải toàn bộ** dữ liệu từ nguồn, **lưu đệm toàn bộ vào bộ nhớ RAM** — hệ thống tuyệt đối không thực hiện bất kỳ thao tác ghi cơ sở dữ liệu nào trong giai đoạn phân trang.
2. **Sau khi tải xong tất cả các trang**, hệ thống mở một giao dịch duy nhất (**transaction**) để thực hiện:
   (a) `deleteMany({})` — xóa trống toàn bộ dữ liệu bảng hiện tại.
   (b) `createMany(...)` — chèn lại toàn bộ bản ghi mới được tải về.

Toàn bộ quá trình xóa và chèn được bao bọc trong khối `prisma.$transaction()`, đảm bảo **tính nguyên tử (atomicity)**: nếu quá trình chèn dữ liệu thất bại, thao tác xóa bảng trước đó sẽ lập tức được hoàn tác, ngăn ngừa việc bảng dữ liệu rơi vào trạng thái trống rỗng nửa vời.

Lý do lưu đệm toàn bộ dữ liệu trước khi ghi Nếu hệ thống thực hiện thao tác "xóa rồi chèn"tuần tự theo từng trang, bảng dữ liệu đích sẽ bị xóa sạch ngay sau trang đầu tiên và chỉ chứa một lượng nhỏ dữ liệu của trang đó trong khi chờ tải trang tiếp theo. Điều này gây ra **khoảng thời gian gián đoạn (window downtime)**, ảnh hưởng nghiêm trọng đến các truy vấn đọc đang diễn ra đồng thời của các hệ thống khác. Việc lưu đệm toàn bộ dữ liệu vào RAM rồi mới ghi đè trong một giao dịch duy nhất giúp rút ngắn thời gian gián đoạn xuống mức tối thiểu (chỉ bằng với thời gian thực thi của giao dịch cơ sở dữ liệu).

Miễn trừ cơ chế phát hiện bản ghi mồ côi Vì bản chất của `overwrite` là thực hiện `DELETE all` trước khi chèn mới, mọi bản ghi bị xóa ở hệ thống nguồn hiển nhiên cũng sẽ bị xóa ở hệ thống đích. Do đó, hệ thống được lập trình để **bỏ qua hoàn toàn** bước kiểm tra bản ghi mồ côi cho những bảng sử dụng chiến lược này.

6.2 Cơ Chế Retry Trong ETL Pipeline

6.2.1 Đặt Vấn Đề

Trong quá trình đồng bộ dữ liệu từ hệ thống nguồn *myBK* qua REST API, pipeline ETL phải đối mặt với hai nhóm lỗi có bản chất hoàn toàn khác nhau:

- **Lỗi tạm thời (Transient Errors):** Phát sinh do điều kiện môi trường — mạng không ổn định, server nguồn quá tải nhất thời, timeout do latency cao. Những lỗi này *có thể tự khỏi* sau một khoảng thời gian ngắn.
- **Lỗi nghiệp vụ (Business Errors):** Phát sinh do vi phạm hợp đồng dữ liệu — payload sai định dạng, trang phân trang không khớp, khóa chính bị thiếu. Những lỗi này *không thay đổi* dù retry bao nhiêu lần, vì nguyên nhân gốc rễ nằm ở phía dữ liệu, không phải môi trường.

Một cơ chế retry áp dụng đồng nhất cho cả hai nhóm sẽ gây ra hai hệ quả tiêu cực: lãng phí tài nguyên khi retry lỗi nghiệp vụ không thể phục hồi, và che khuất các lỗi thực sự cần được xử lý bởi con người.

6.2.2 Thiết Kế Retry Có Chủ Ý

Hệ thống áp dụng nguyên tắc *Selective Retry*: chỉ retry các lỗi tạm thời ở tầng giao tiếp mạng, và để lỗi nghiệp vụ thất bại ngay lập tức (*fail-fast*) để pipeline có thể ghi nhận và xử lý đúng cách.

Phân Loại Lỗi và Quyết Định Retry Bảng 6.2 tổng hợp chính sách retry theo từng tình huống lỗi trong pipeline.

Bảng 6.2: Chính sách retry theo loại lỗi

Tình huống	**Retry**	**Lý do**
Network timeout / `ECONNREFUSED`	Có	Lỗi tạm thời do môi trường mạng. Server nguồn có thể đã phục hồi sau vài giây.
API trả về `5xx` (server nguồn lỗi)	Có	Server nguồn có thể đang khởi động lại hoặc quá tải nhất thời. Retry sau 10 giây có khả năng thành công.
`[API CONTRACT VIOLATION]`	Không	Payload sai cấu trúc (`success != true`, thiếu `payload[]`). Retry sẽ nhận lại cùng response sai — không có ý nghĩa.
`[SCHEMA CONTRACT VIOLATION]`	Không	Payload chứa field chưa định nghĩa trong metadata. Đây là lỗi cấu hình cần admin xử lý thủ công.
`[PAGE MISMATCH]`	Không	Server nguồn trả về sai số trang phân trang. Retry vẫn nhận trang sai — cần điều tra phía nguồn.
DB constraint violation (upsert)	Không	Record vi phạm ràng buộc cơ sở dữ liệu sẽ tiếp tục thất bại trong mọi lần retry. Chuyển sang *Dead Letter Log* để xử lý riêng.

Tham Số Retry Pipeline được cấu hình với các tham số sau:

- **Số lần retry tối đa:** 3 lần (tổng cộng 4 lần thử bao gồm lần đầu tiên).
- **Khoảng chờ giữa các lần retry:** 10 giây (fixed delay).
- **Phạm vi áp dụng:** Chỉ tầng HTTP fetch — không áp dụng cho tầng ghi cơ sở dữ liệu.

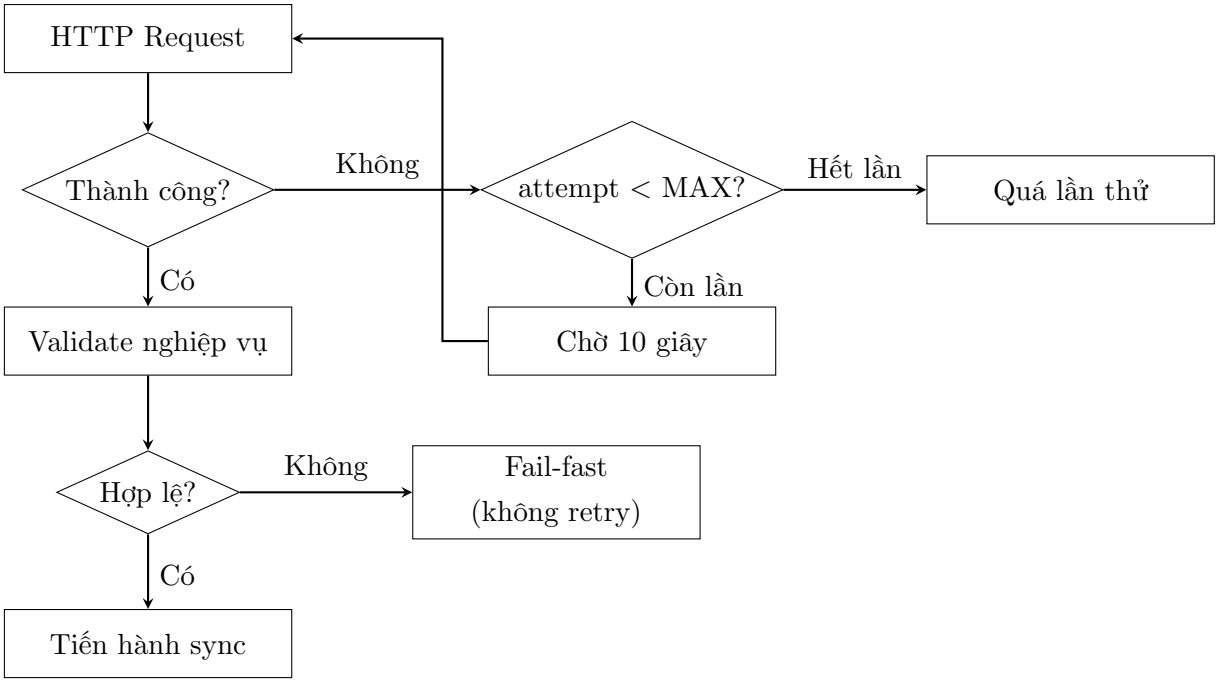
Cài Đặt Retry được đóng gói trong phương thức `withRetry<T>()` — một generic wrapper nhận vào một hàm bất đồng bộ và tên ngữ cảnh để ghi log. Phương thức này

được gọi quanh mỗi HTTP request trong vòng lặp phân trang.

Lỗi nghiệp vụ được ném ra *bên ngoài* `withRetry()` — cụ thể là sau khi response đã được nhận thành công từ mạng — do đó chúng không bị bắt bởi vòng lặp retry

6.2.3 Luồng Xử Lý Khi Retry

Hình 6.2 mô tả luồng quyết định khi một HTTP request thất bại.



Hình 6.2: Luồng quyết định retry trong pipeline ETL

6.2.4 Đánh Giá

Thiết kế retry có chủ ý mang lại các lợi ích sau:

1. **Tăng độ bền:** Pipeline tự phục hồi khi gặp lỗi mạng tạm thời mà không cần can thiệp thủ công, đặc biệt quan trọng trong môi trường ngrok với độ ổn định không cao.

2. **Phát hiện lỗi nhanh (Fail-Fast):** Lỗi nghiệp vụ được báo cáo ngay lập tức thay vì bị che khuất bởi nhiều lần retry vô ích, giúp rút ngắn thời gian phát hiện sự cố từ phía nguồn.

3. **Tiết kiệm tài nguyên:** Không lãng phí 30 giây chờ (3 lần × 10 giây) cho các lỗi xác định không thể phục hồi.

4. **Log có ngữ cảnh:** Mỗi lần retry được ghi kèm số thứ tự, tên bảng, số trang và thông báo lỗi gốc, giúp vận hành dễ dàng truy vết nguyên nhân.

Hạn chế: Cơ chế hiện tại sử dụng *fixed delay* (10 giây cố định). Trong môi trường production với tải cao, chiến lược *exponential backoff with jitter* sẽ hiệu quả hơn vì tránh được hiện tượng nhiều client retry đồng thời. Đây là hướng cải tiến cho phiên bản tiếp theo.

6.3 Phát hiện bản ghi mồ côi (Orphan Detection)

6.3.1 Đặt vấn đề

Các chiến lược đồng bộ `upsert` và đồng bộ tăng tiến (`incremental`) chỉ xử lý hai chiều biến động của dữ liệu: thêm mới và cập nhật. Chiều thứ ba - xóa bản ghi tại hệ thống nguồn - hoàn toàn nằm ngoài phạm vi xử lý của hai chiến lược này.

Khi một bản ghi bị xóa tại hệ thống nguồn, nó không còn xuất hiện trong phản hồi của API. Tuy nhiên, bản ghi tương ứng tại hệ thống đích (PostgreSQL) vẫn tồn tại nguyên vẹn vì không có lệnh `DELETE` nào được phát ra. Theo thời gian, những bản ghi còn lại này tích lũy ngày càng nhiều, tạo ra sự phân mảnh lớn so với dữ liệu nguồn thực tế.

Định nghĩa chính thức: một **bản ghi mồ côi** (orphan record) là bản ghi tồn tại trong cơ sở dữ liệu đích nhưng không xuất hiện trong bất kỳ trang dữ liệu nào được trả về từ API nguồn sau một chu kỳ đồng bộ toàn phần.

6.3.2 Giải pháp: Phép trừ tập hợp khóa chính

Trong số các phương án xử lý bản ghi mồ côi, hệ thống lựa chọn phương pháp **Phép trừ tập hợp khóa chính (ID Set Difference)** — phương án phù hợp nhất với ràng buộc của dự án (API nguồn không hỗ trợ luồng sự kiện thay đổi hay sự kiện xóa).

Ý tưởng cốt lõi:

$$\text{Orphan IDs} = \text{Destination IDs} \setminus \text{Source IDs} \tag{6.1}$$

Tức là: lấy tập hợp tất cả khóa chính đang tồn tại ở hệ thống đích, trừ đi tập hợp khóa chính thu thập được từ toàn bộ các trang API nguồn. Phần còn lại chính là danh sách các bản ghi không còn tồn tại ở nguồn.

6.3.3 Quy trình triển khai

Việc phát hiện bản ghi mồ côi được tích hợp trực tiếp vào cuối hàm `syncOneTable()` - sau khi vòng lặp phân trang hoàn thành và toàn bộ khóa chính từ nguồn (`source IDs`) đã được thu thập:

1. **Thu thập khóa chính từ nguồn:** Trong mỗi vòng lặp phân trang, mọi giá trị khóa chính từ `payload` được thêm vào một tập hợp `sourceIds`. Bước này không tốn thêm tài nguyên mạng vì dữ liệu đã được tải sẵn.

2. **Truy vấn khóa chính tại đích:** Sau khi tải xong tất cả các trang, hệ thống thực hiện một truy vấn `SELECT` chỉ lấy cột khóa chính từ bảng đích - tránh việc tải toàn bộ bản ghi lên bộ nhớ.

3. **Tính toán hiệu tập hợp:** So sánh hai tập hợp, hệ thống trả về danh sách khóa chính chỉ tồn tại ở hệ thống đích.

4. **Cảnh báo:** Nếu phát hiện bản ghi mồ côi, hệ thống ghi nhật ký mức độ cảnh báo (`WARN`) với danh sách mã định danh đầy đủ (hiển thị tối đa 20 mã đầu tiên nếu danh sách quá dài).

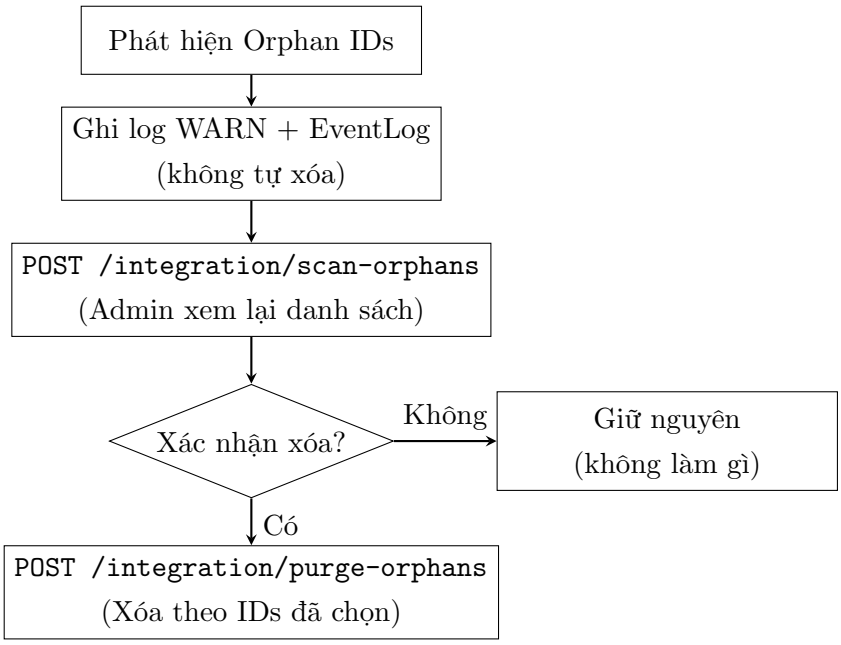
6.3.4 Thiết kế không tự động xóa

Một quyết định thiết kế quan trọng là hệ thống **chỉ cảnh báo, không tự động xóa** khi phát hiện bản ghi mồ côi. Quyết định này xuất phát từ ba rủi ro thực tiễn:

1. **Bản ghi mồ côi giả tạo do lỗi API:** Nếu máy chủ nguồn trả về thiếu trang do lỗi phân trang hoặc quá thời gian chờ (timeout), các bản ghi thuộc trang bị thiếu sẽ không có trong `sourceIds` và bị nhận diện nhầm là bản ghi mồ côi. Việc tự động xóa trong trường hợp này đồng nghĩa với làm mất dữ liệu hợp lệ.

2. **Dữ liệu được tạo thủ công:** Một số bảng có thể chứa bản ghi được nhập tay bởi quản trị viên phục vụ mục đích vận hành nội bộ. Những bản ghi này không tồn tại ở nguồn nhưng hoàn toàn hợp lệ tại đích.

3. **Tính không thể phục hồi:** Không giống như thao tác chèn hoặc cập nhật, thao tác `DELETE` không thể hoàn tác (rollback) dễ dàng nếu không có cơ chế sao lưu đầy đủ.



Hình 6.3: Quy trình hai bước: phát hiện và xác nhận xóa bản ghi mồ côi

6.3.5 API quản lý bản ghi mồ côi

Hệ thống cung cấp hai điểm cuối (endpoint) riêng biệt để quản trị viên thực hiện quy trình xác nhận hai bước:

Bước 1 — Quét bản ghi mồ côi (Scan)

Listing 4: API quét bản ghi mồ côi — chỉ đọc

```
POST /api/v1/integration/scan-orphans
Content-Type: application/json

{ "tables": ["nguoi_hoc"] }
```

Điểm cuối này tải toàn bộ dữ liệu từ API nguồn, thực hiện phép trừ tập hợp, và trả về danh sách khóa chính cụ thể. Không có thao tác ghi hoặc xóa nào được thực hiện lên cơ sở dữ liệu.

Listing 5: Cấu trúc phản hồi mẫu từ API scan-orphans

```
[{
  "tableName": "nguoi_hoc",
  "primaryKey": "id",
  "orphanCount": 1,
```

```
  "orphanIds": [99999]
}]
```

Listing 6: API xóa bản ghi mồ côi — yêu cầu truyền ID tường minh

Bước 2 — Xóa có xác nhận (Purge)

```
POST /api/v1/integration/purge-orphans
Content-Type: application/json

{
  "tableName": "nguoi_hoc",
  "primaryKey": "id",
  "ids": [99999]
}
```

Quản trị viên bắt buộc phải sao chép danh sách `orphanIds` từ kết quả của bước 1 và truyền lại một cách tường minh vào yêu cầu này. Cơ chế này đảm bảo không thể xảy ra việc vô tình xóa hàng loạt do lỗi lập trình hay gọi API không có chủ đích.

Mỗi lần xóa được ghi nhận vào nhật ký sự kiện với đầy đủ thông tin: tên bảng, khóa chính, số lượng ID được yêu cầu xóa, và số lượng bản ghi thực tế đã bị xóa.

6.3.6 Giới hạn của phương pháp

Phương pháp trừ tập hợp khóa chính có hiệu quả cao trong điều kiện hiện tại của dự án, song cần lưu ý một giới hạn quan trọng: orphan detection tự động trong sync chỉ chạy khi fetch toàn bộ (upsert hoặc incremental lần đầu), muốn kiểm tra orphan trên incremental cần gọi POST /integration/scan-orphans riêng — endpoint đó luôn fetch toàn bộ source. Với cấu trúc dữ liệu hiện tại (ví dụ: bảng `nguoi_hoc` chứa hơn 68.000 bản ghi), thời gian quét mất khoảng 3 phút — đây là mức chi phí chấp nhận được cho một tác vụ vận hành định kỳ, nhưng không phù hợp nếu yêu cầu kiểm tra theo thời gian thực. Hướng cải tiến lâu dài là yêu cầu hệ thống nguồn cung cấp điểm cuối API `/changes?since=<timestamp>` để chỉ trả về danh sách các khóa chính bị xóa trong một khoảng thời gian cụ thể.

6.4 Thông Báo Qua Email Sau Khi Pipeline Chạy Xong

6.4.1 Đặt vấn đề

Sau khi một pipeline đồng bộ dữ liệu hoàn thành — dù thành công hay thất bại — hệ thống cần chủ động thông báo đến quản trị viên mà không yêu cầu họ phải theo dõi log liên tục. Cơ chế thông báo qua email giải quyết nhu cầu này: kết quả của mỗi lần chạy pipeline được gửi tự động đến địa chỉ email cấu hình sẵn ngay khi job kết thúc.

6.4.2 Công nghệ sử dụng

Hệ thống sử dụng thư viện **Nodemailer** tích hợp trong `NotificationService` (NestJS) để gửi email qua giao thức **SMTP**. Mặc định kết nối tới Gmail (`smtp.gmail.com:587`) với xác thực TLS, nhưng có thể đổi sang bất kỳ SMTP server nào qua biến môi trường.

Kết nối SMTP được khởi tạo một lần khi module load. Nếu biến môi trường `SMTP_USER` hoặc `SMTP_PASSWORD` chưa được cấu hình, service tự động tắt tính năng email mà không làm crash ứng dụng — đảm bảo môi trường phát triển không bắt buộc phải có SMTP.

6.4.3 Điểm kích hoạt

`NotificationService` được gọi tại cuối cả hai pipeline đồng bộ trong `DataIntegrationService`:

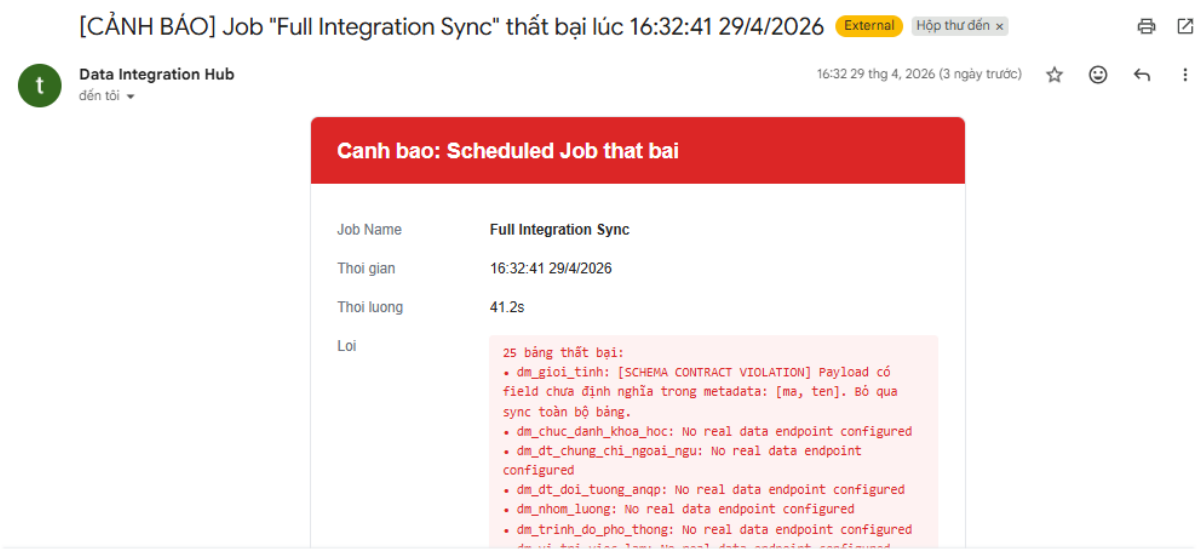
- **Full Integration Pipeline** — đồng bộ toàn bộ các bảng có trạng thái `stable`.
- **Custom Integration Pipeline** — đồng bộ một tập bảng do admin chỉ định.

Tùy kết quả, một trong hai phương thức được gọi:

- `sendJobSuccessSummary()` — khi toàn bộ bảng đồng bộ thành công (`status = done`).
- `sendJobFailureAlert()` — khi có ít nhất một bảng thất bại (`status = partial_success` hoặc `failed`).

6.4.4 Nội dung email

Mỗi email được render dưới dạng **HTML** với bố cục bảng, gồm các trường sau:



Hình 6.4: Minh họa mail cảnh báo

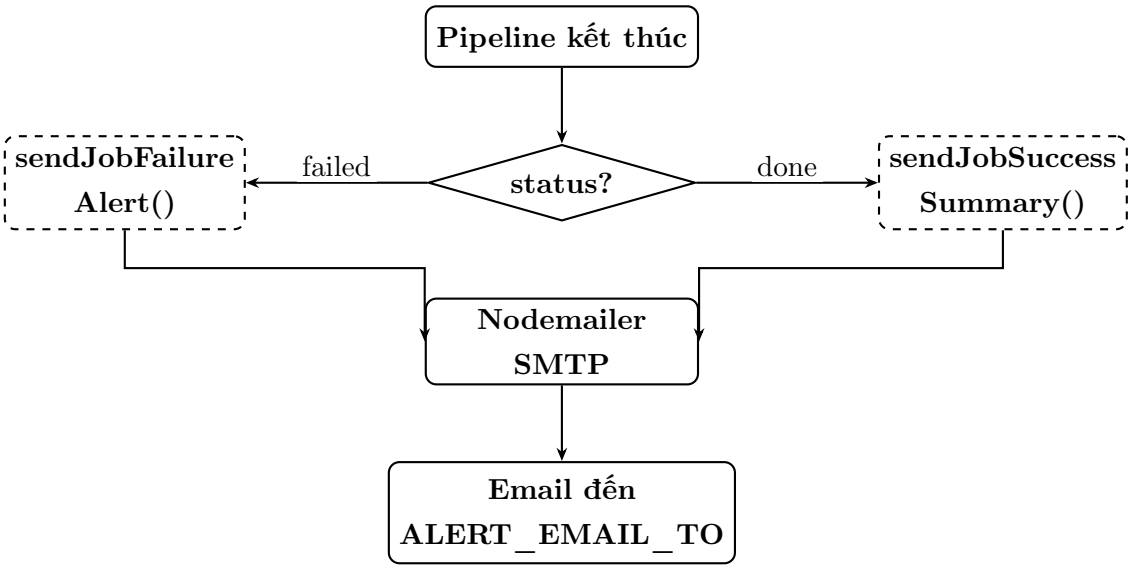
6.4.5 Cấu hình biến môi trường

Bảng 6.3: Biến môi trường cho NotificationService

Biến	Mặc định	Mục đích
`SMTP_HOST`	`smtp.gmail.com`	SMTP server
`SMTP_PORT`	587	Cổng SMTP (TLS)
`SMTP_USER`	—	Địa chỉ email gửi
`SMTP_PASSWORD`	—	App Password của Gmail
`ALERT_EMAIL_TO`	—	Địa chỉ nhận cảnh báo

Với Gmail, `SMTP_PASSWORD` phải là **App Password** (16 ký tự) được tạo tại *Google Account* → *Security* → *2-Step Verification* → *App passwords* — không dùng mật khẩu tài khoản thông thường.

6.4.6 Luồng xử lý



Hình 6.5: Luồng gửi email thông báo sau khi pipeline kết thúc

7 Chiến Lược Sao Lưu Dữ Liệu Với MinIO và AWS S3

7.1 Đặt Vấn Đề

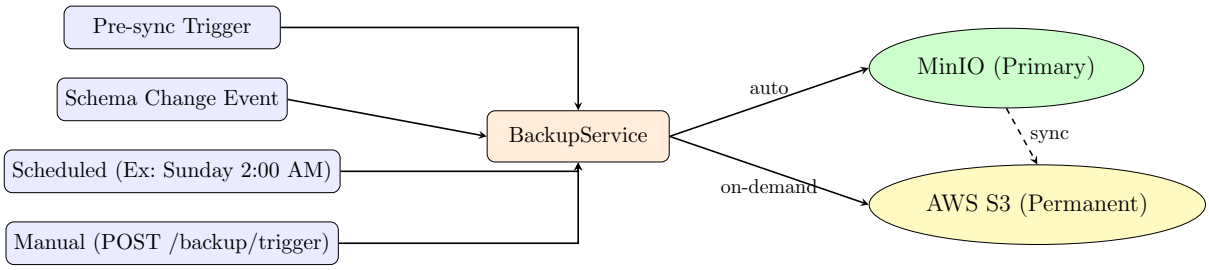
Trong hệ thống tích hợp dữ liệu, mỗi lần đồng bộ đều có khả năng ghi đè hoặc biến đổi dữ liệu tại destination. Khi xảy ra các sự kiện bất thường — lỗi pipeline, thay đổi schema không kiểm soát, hoặc dữ liệu nguồn bị hỏng — cần có khả năng phục hồi về trạng thái trước đó một cách đáng tin cậy. Yêu cầu này đặt ra bốn bài toán cụ thể:

1. **Backup phòng ngừa:** Tạo snapshot trước mỗi lần sync để có điểm phục hồi ngay cả khi pipeline ghi dữ liệu sai.
2. **Backup kiểm toán:** Lưu giữ vĩnh viễn snapshot tại thời điểm schema thay đổi — phục vụ truy vết lịch sử cấu trúc dữ liệu.
3. **Backup định kỳ:** Đảm bảo luôn có bản sao dữ liệu hoàn chỉnh theo lịch, độc lập với hoạt động sync.
4. **Lưu trữ vĩnh viễn trên cloud:** Nhân bản các backup quan trọng lên AWS S3 để đảm bảo tính bền vững và khả năng phục hồi ngay cả khi infrastructure nội bộ gặp sự cố.

7.2 Kiến Trúc Hai Tầng: MinIO và AWS S3

Hệ thống sử dụng kiến trúc lưu trữ hai tầng:

- **Tầng 1 — MinIO (primary):** Object storage self-hosted triển khai trên Docker, tương thích với giao thức Amazon S3. Mọi backup được ghi vào MinIO ngay lập tức. MinIO phục vụ cho các tác vụ restore và truy vấn thường xuyên nhờ độ trễ thấp trong môi trường nội bộ.
- **Tầng 2 — AWS S3 (secondary/permanent):** Dịch vụ object storage của Amazon Web Services, đóng vai trò kho lưu trữ vĩnh viễn. Admin có thể chủ động đẩy (sync) các file backup từ MinIO lên S3 để bảo toàn dữ liệu ngoài môi trường on-premise.



Hình 7.1: Kiến trúc lưu trữ hai tầng: MinIO và AWS S3

7.2.1 Cấu Hình Triển Khai

MinIO được tích hợp vào `docker-compose.yml` cùng với phần còn lại của hệ thống:

Listing 7: Cấu hình MinIO trong docker-compose.yml

```yaml
services:
  minio:
    image: minio/minio:latest
    container_name: data-hub-minio
    command: server /data --console-address ":9001"
    ports:
      - "9000:9000"   # S3 API endpoint
      - "9001:9001"   # Web Console
    environment:
      MINIO_ROOT_USER: ${MINIO_ACCESS_KEY}
      MINIO_ROOT_PASSWORD: ${MINIO_SECRET_KEY}
    volumes:
      - minio_data:/data
    restart: always
```

Cấu hình AWS S3 được truyền vào backend qua biến môi trường:

Listing 8: Biến môi trường AWS S3

```
AWS_ACCESS_KEY_ID=<access-key>
AWS_SECRET_ACCESS_KEY=<secret-key>
AWS_REGION=ap-southeast-2
AWS_S3_BUCKET=data-integration-hub-bucket
```

7.3 Bốn Loại Trigger Backup

7.3.1 Trigger 1: Pre-Sync Backup

Đây là lớp bảo vệ quan trọng nhất. Ngay trước khi `syncOneTable()` bắt đầu ghi dữ liệu vào PostgreSQL, một snapshot đầy đủ của bảng được tạo và lưu lên MinIO:

Listing 9: Pre-sync backup trong syncOneTable()

```
1 await this.backupService.backupTable(schema.tableName, 'pre-
    sync');
2 // Sau do moi bat dau fetch va sync
```

Chỉ những bảng có `status = 'stable'` mới đến được bước `syncOneTable()` — pipeline lọc bảng không ổn định trước khi vào vòng lặp sync.

7.3.2 Trigger 2: Schema Change Backup

Khi `SchemaRegistryService` phát hiện một bảng thay đổi cấu trúc, hệ thống emit event `schema.changed`. `BackupService` lắng nghe event này và tạo snapshot ngay lập tức — trước khi bảng bị đánh dấu `changed` và khóa khỏi chu kỳ sync tiếp theo:

Listing 10: Xử lý event schema.changed

```
1 @OnEvent('schema.changed')
2 async handleSchemaChanged(payload: { tableName: string }) {
3   await this.backupTable(payload.tableName, 'schema-change');
4 }
```

Đây là snapshot cuối cùng của dữ liệu tương thích với schema cũ, có giá trị kiểm toán cao và được giữ vĩnh viễn.

7.3.3 Trigger 3: Scheduled Weekly Backup

Mỗi Chủ nhật lúc 2:00 AM, hệ thống tự động backup toàn bộ bảng `stable` và thực hiện cleanup các file quá hạn:

Listing 11: Scheduled backup mỗi Chủ nhật 2:00 AM

```
1 @Cron('0 2 * * 0')
2 async handleWeeklyBackup() {
3   await this.backupAllStable('scheduled');
4   await this.cleanupOldBackups();
```

```
}
```

7.3.4 Trigger 4: Manual Backup

Admin có thể kích hoạt backup bất kỳ lúc nào qua API `POST /backup/trigger`, chỉ định một hoặc nhiều bảng cụ thể, hoặc bỏ trống để backup toàn bộ bảng stable.

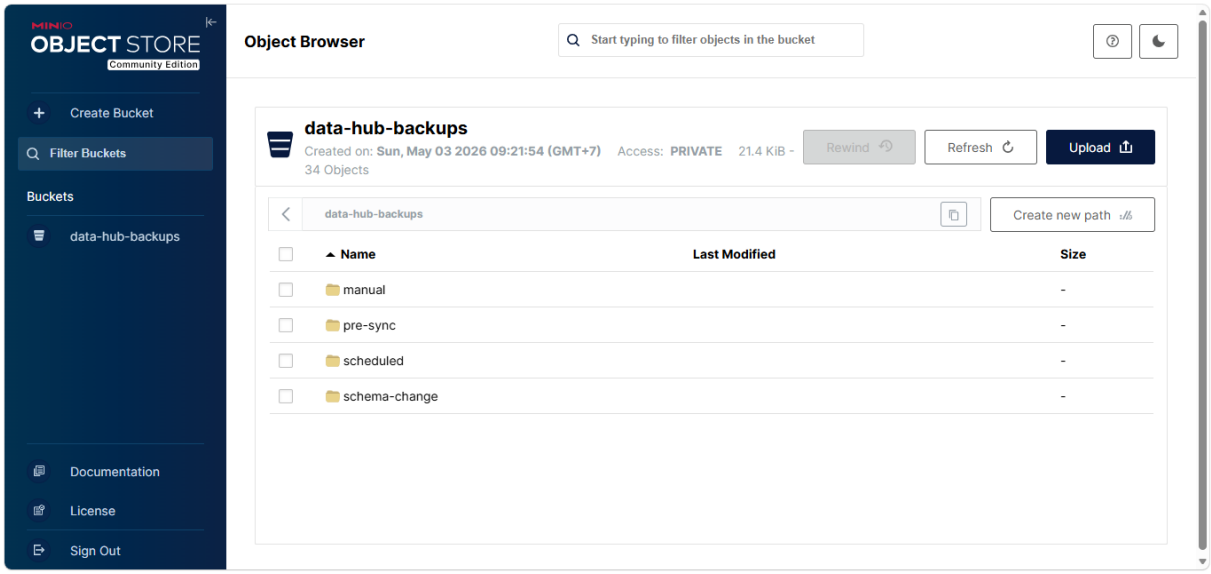
7.4 Cấu Trúc Lưu Trữ Trong MinIO

Tất cả backup được tổ chức trong bucket `data-hub-backups` theo cấu trúc phân cấp thống nhất:

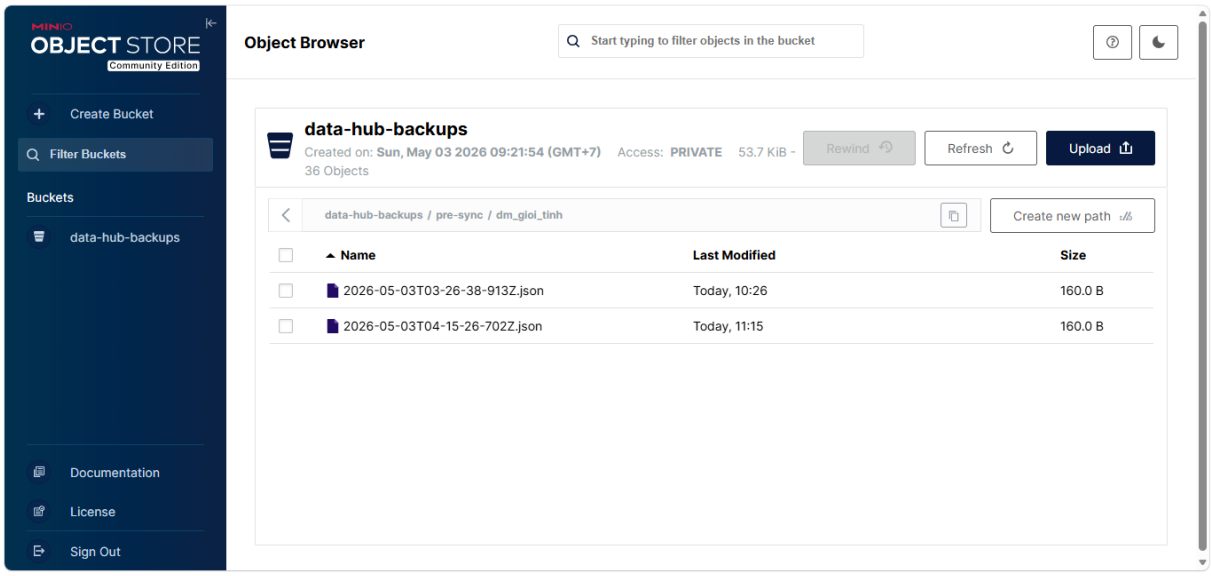
`{trigger}/{tableName}/{ISO-timestamp}.json`

Listing 12: Cấu trúc thư mục trong bucket MinIO

```
data-hub-backups/
|-- pre-sync/
|   |-- nguoi_hoc/
|   |   |-- 2026-04-20T08-30-00-000Z.json
|   |   `-- 2026-04-21T09-15-22-451Z.json
|   `-- nh_dao_tao/
|       `-- 2026-04-21T09-15-25-112Z.json
|
|-- schema-change/
|   `-- nguoi_hoc/
|       |-- 2026-03-15T14-22-10-334Z.json
|       `-- 2026-04-10T09-05-44-218Z.json
|
|-- scheduled/
|   `-- nguoi_hoc/
|       |-- 2026-04-13T02-00-05-000Z.json
|       `-- 2026-04-20T02-00-03-000Z.json
|
`-- manual/
    `-- nguoi_hoc/
        `-- 2026-04-20T15-00-00-000Z.json
```



Hình 7.2: Giao diện cấu trúc trong MINIO



Hình 7.3: Các file Backup cho một bảng trong MINIO

Mỗi file backup là một JSON object gồm phần `meta` và `data`:

Listing 13: Cấu trúc file backup JSON

```
{
  "meta": {
    "tableName": "nguoi_hoc",
    "trigger": "pre-sync",
    "backedUpAt": "2026-04-21T02:15:30.000Z",
    "recordCount": 68411
  },
  "data": [
    { "id": 1, "ho": "Nguyen Van", "ten": "A", ... }
  ]
}
```

JSON được chọn thay vì CSV vì giữ nguyên kiểu dữ liệu (`DateTime`, `Boolean`, `null`) và tương thích trực tiếp với Prisma `createMany()` khi restore.

7.5 Chính Sách Lưu Giữ (Retention Policy)

Bảng 7.1: Chính sách lưu giữ backup theo loại trigger

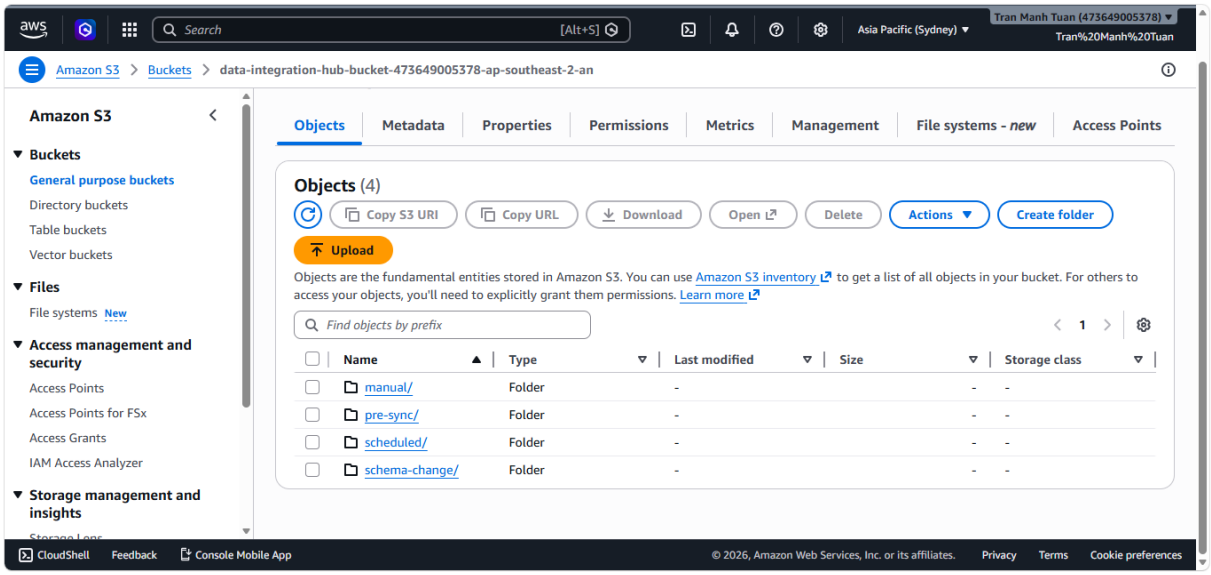
Loại trigger	**Thời gian giữ**	**Lý do**
`pre-sync`	30 ngày	Đủ để phát hiện và rollback lỗi sync trong chu kỳ vận hành.
`scheduled`	60 ngày	Lưu 8 bản weekly — đủ truy vết biến động 2 tháng gần nhất.
`schema-change`	Vĩnh viễn	Audit trail lịch sử thay đổi cấu trúc dữ liệu.
`manual`	30 ngày	Backup thủ công thường phục vụ mục đích ngắn hạn.

Cleanup được thực thi tự động sau mỗi weekly backup và có thể gọi thủ công qua `POST /backup/cleanup`.

7.6 Cơ Chế Đồng Bộ Lên AWS S3

7.6.1 Mục Tiêu

MinIO là storage nội bộ chạy trên cùng máy chủ với hệ thống. Trong trường hợp máy chủ gặp sự cố nghiêm trọng (mất điện, hỏng ổ đĩa), toàn bộ dữ liệu MinIO có thể bị mất cùng lúc. AWS S3 đóng vai trò lớp lưu trữ thứ hai — địa lý phân tán, độ bền **99.999999999%** (11 số 9) theo cam kết SLA của Amazon.



Hình 7.4: Giao diện lưu trữ file trên S3

7.6.2 Luồng Hoạt Động

Đồng bộ lên S3 được thiết kế theo mô hình **on-demand** (theo yêu cầu) thay vì tự động sau mỗi backup, nhằm tránh tạo ra bottleneck I/O trong các chu kỳ sync tần suất cao:

1. Admin gọi API sync (một file hoặc toàn bộ).
2. `S3Service` kiểm tra file đã tồn tại trên S3 chưa bằng `HeadObjectCommand`.
3. Nếu chưa có: tải file JSON từ MinIO về bộ nhớ, upload lên S3 bằng `PutObjectCommand`.
4. Nếu đã có: bỏ qua — tránh upload trùng lặp không cần thiết.

7.6.3 Trạng Thái Đồng Bộ

API `GET /backup/list` trả về thêm trường `s3Status` cho mỗi file backup, phản ánh tình trạng đồng bộ giữa MinIO và S3:

Bảng 7.2: Các giá trị trạng thái đồng bộ S3

Giá trị	**Ý nghĩa**
`not_synced`	File chỉ tồn tại trên MinIO, chưa được upload lên S3.
`synced`	File đã được upload lên S3 và `lastModified` trên MinIO $\leq$ `LastModified` trên S3 — hai phiên bản đồng nhất.
`out_of_date`	File tồn tại trên S3 nhưng phiên bản trên MinIO mới hơn — cần sync lại.

Trạng thái được tính bằng cách so sánh `lastModified` của từng object: nếu thời điểm chỉnh sửa trên MinIO $\leq$ thời điểm upload lên S3 thì file được coi là `synced`; ngược lại là `out_of_date`. Để tránh gọi `HeadObject` cho từng file (chi phí $O(n)$ request), hệ thống sử dụng `ListObjectsV2` một lần duy nhất để lấy toàn bộ danh sách S3 và xây dựng `Map<key, LastModified>` trước khi đối chiếu.

7.6.4 Fire-and-Forget cho Sync Hàng Loạt

Khi sync toàn bộ bucket (có thể lên đến hàng trăm file), tác vụ mất nhiều phút để hoàn thành. Nếu xử lý đồng bộ (*blocking*), HTTP client hoặc API Gateway sẽ timeout trước khi nhận được response.

Hệ thống giải quyết bằng mô hình **fire-and-forget**: endpoint trả về response 200 ngay lập tức, trong khi tác vụ sync tiếp tục chạy nền:

Listing 14: Fire-and-forget cho sync-s3/all

```
@Post('sync-s3/all')
syncAllToS3() {
  // Khong await - tra response ngay, sync chay nen
  this.backupService.syncAllToS3().catch(() => {});
  return {
```

```
    message: 'S3 sync dang chay nen. Kiem tra log de xem tien
        trinh.'
  };
}
```

7.7 API Quản Lý Backup

Bảng 7.3: Các endpoint quản lý backup

Method	Endpoint	Mô tả
POST	/backup/trigger	Backup thủ công một hoặc nhiều bảng.
GET	/backup/list?prefix=...	Liệt kê file backup kèm s3Status.
POST	/backup/download	Trả về presigned URL (1 giờ) để tải file từ MinIO.
DELETE	/backup/file	Xóa một file backup khỏi MinIO.
POST	/backup/restore	Restore một bảng từ file backup về PostgreSQL.
POST	/backup/cleanup	Xóa các file backup quá hạn theo retention policy.
POST	/backup/sync-s3	Đồng bộ một file cụ thể từ MinIO lên S3.
POST	/backup/sync-s3/all	Đồng bộ toàn bộ bucket lên S3 (chạy nền).

7.8 Cơ Chế Restore

Khi cần phục hồi dữ liệu, hệ thống thực hiện theo chiến lược **overwrite**: xóa toàn bộ bảng rồi chèn lại từ file backup. Trước khi restore, hệ thống tự động tạo thêm một

snapshot `manual` của trạng thái hiện tại — đây là lớp bảo vệ phòng trường hợp admin restore nhầm file.

Do bảng `nguoi_hoc` có hơn 68.000 bản ghi, việc insert toàn bộ trong một transaction sẽ vượt quá timeout 5.000 ms của Prisma interactive transaction. Hệ thống giải quyết bằng cách chia nhỏ thành các batch 1.000 bản ghi và insert tuần tự bên ngoài transaction:

Listing 15: Batch INSERT khi restore

```
const BATCH_SIZE = 1000;
for (let i = 0; i < data.length; i += BATCH_SIZE) {
  const batch = data.slice(i, i + BATCH_SIZE);
  await this.prisma[modelName].createMany({
    data: batch,
    skipDuplicates: true
  });
}
```

7.9 Đánh Giá

7.9.1 Ưu Điểm

- **Bảo vệ nhiều lớp:** Ba trigger tự động đảm bảo luôn có điểm phục hồi gần nhất bất kể nguyên nhân sự cố.
- **Lưu trữ địa lý phân tán:** Kết hợp MinIO nội bộ và AWS S3 đám mây — sự cố tại một nơi không ảnh hưởng đến phiên bản còn lại.
- **Audit trail đầy đủ:** Backup `schema-change` lưu vĩnh viễn cho phép tái hiện lịch sử dữ liệu theo từng mốc thay đổi cấu trúc.
- **Khả năng quan sát:** Trường `s3Status` trong API list cho phép admin theo dõi tình trạng đồng bộ của từng file mà không cần truy cập trực tiếp vào AWS Console.

7.9.2 Hạn Chế và Hướng Phát Triển

Phương án hiện tại thực hiện `SELECT *` toàn bộ bảng trước mỗi lần sync. Với bảng `nguoi_hoc` có 68.411 bản ghi, mỗi file backup có kích thước khoảng 50–80 MB. Hai hướng cải tiến được đề xuất: (1) sử dụng *incremental backup* — chỉ lưu delta so với lần trước; (2) nén bằng `gzip` trước khi upload để giảm dung lượng khoảng 60–70%.

Cơ chế đồng bộ S3 hiện hoạt động theo mô hình on-demand. Hướng phát triển tiếp theo là tự động hóa: sau mỗi lần backup thành công, hệ thống tự đẩy file mới lên S3 mà không cần admin can thiệp thủ công.

8 Phân tích Hiệu năng và Capacity Planning

8.1 Tổng quan

Hệ thống Data Integration Hub vận hành theo hai chiều dữ liệu với đặc tính tải hoàn toàn khác nhau:

- **Inbound (Sync Job)**: Lấy dữ liệu từ Source API và ghi vào PostgreSQL theo lịch định kỳ. Tải có thể dự đoán, bottleneck chính là **RAM** và **network I/O**.
- **Outbound (Export API)**: Phục vụ các hệ thống tiêu thụ truy vấn dữ liệu qua REST API phân trang. Tải không dự đoán được, bottleneck chính là **DB query latency** và **concurrency**.

Để kiểm tra hiệu năng có kiểm soát, nhóm xây dựng một **Mock Source Server** (Express.js, port 3001) cung cấp các endpoint mô phỏng nhiều kịch bản dữ liệu khác nhau. Server này phục vụ đồng thời hai mục đích: kiểm thử cơ chế phòng thủ *Data Collision* và đo lường hiệu năng pipeline.

8.2 Kiểm thử Data Collision — Mock Source Server

8.2.1 Kiến trúc Mock Server

Mock server được thiết kế với một helper dùng chung `buildResponse()` chuẩn hóa mọi response theo đúng API Contract mà NestJS kỳ vọng:

Listing 16: Helper `buildResponse()` của Mock Server

```
function buildResponse(tableName, allData, page, limit,
                       overrideCurrentPage = null) {
  const totalRecords = allData.length;
  const totalPages   = Math.ceil(totalRecords / limit) || 1;
  const startIndex   = (page - 1) * limit;
  const payload      = allData.slice(startIndex, startIndex +
      limit);

  return {
    success:   true,
    timestamp: new Date().toISOString(),
    metadata:  { tableName, totalRecords, totalPages,
```

```
                    currentPage: overrideCurrentPage ?? page,
                      limit },
    payload,
  };
}
```

8.2.2 Ma trận Test Case

Bảng 8.1 tổng hợp toàn bộ 15 test case được thiết kế để kiểm tra từng lớp cơ chế phòng thủ. Mỗi test case trỏ vào một endpoint riêng biệt trên Mock Server và được cấu hình qua Schema Registry MongoDB.

Bảng 8.1: Ma trận kiểm thử Data Collision Defense

TC	Tên	Kịch bản / Cơ chế kiểm tra	Kỳ vọng
TC-01	Happy Path	Dữ liệu chuẩn, đúng contract, không có lỗi	Sync 100% thành công
TC-02	Schema Filter	Payload có 5 fields lạ không khai báo trong Prisma DMMF	Fields bị lọc, upsert thành công
TC-03	Dedup	id=801 lặp ×3, id=804 lặp ×2 trong 1 batch	Giữ record cuối, log *"2 duplicates removed"*
TC-04	Type Norm (Int/Bool)	`id="701"` (string), `active="true"` (string)	Normalize thành `Int` và `Boolean` đúng kiểu
TC-04b	Type Norm (DateTime)	ISO string, date-only string, chuỗi sai định dạng	ISO → Date; chuỗi sai → `null` (không crash)
TC-05	Contract: thiếu `success`	Response dùng key `data` thay vì `payload`, thiếu `success`	Throw *[API CONTRACT VIOLATION]*, skip bảng
TC-06	Contract: `success: false`	Hệ thống nguồn trả lỗi nội bộ	Throw *[API CONTRACT VIOLATION]*, skip bảng

(xem trang sau)

(tiếp theo)

TC	Tên	Kịch bản / Cơ chế kiểm tra	Kỳ vọng
TC-07	Contract: payload object	`payload` là `{}` thay vì `[]`	Throw *[API CONTRACT VIOLATION]*, skip bảng
TC-08	Contract: thiếu `payload`	Response chỉ có `metadata`, không có `payload`	Throw *[API CONTRACT VIOLATION]*, skip bảng
TC-09	Page Mismatch	Server luôn trả `currentPage: 1` dù request page 2	*[PAGE MISMATCH]* → dừng loop ở trang 2
TC-10	Empty Payload	`totalPages: 3` nhưng page 2 trả `payload: []`	*[EMPTY PAYLOAD]* → dừng loop sớm
TC-11	Dead Letter: thiếu PK	3/6 records có `id: null` hoặc không có field `id`	3 records lỗi vào Dead Letter Log; 3 còn lại sync thành công
TC-12	Dead Letter: DB Constraint	`ma = "THIS_STRING_EXCEEDS_VARCHAR5"`	Record vi phạm vào Dead Letter Log; batch không bị abort
TC-13	Schema Status Gate	Set `status="changed"` trong Schema Registry	Bảng bị skip, không fetch API
TC-14	Bad Endpoint	Set `dataFromApi` sai đường dẫn trong Schema Registry	HTTP 404, retry 3 lần, fail fast
TC-15	Idempotent Upsert	Chạy sync bảng `dm_gioi_tinh` 2 lần liên tiếp	Row count PostgreSQL không tăng ở lần 2

8.2.3 Ví dụ kết quả kiểm thử — TC-03 Deduplication

Listing 17: Payload TC-03: id=801 lặp 3 lần trong cùng 1 batch

```
const dupData = [
```

```
  { id: 801, ma: "D1A", ten: "Dup801 v1 overwrite", active: true },
  { id: 802, ma: "N01", ten: "Binh thuong #1",       active: true },
  { id: 801, ma: "D1B", ten: "Dup801 v2 overwrite", active: true },
  { id: 803, ma: "N02", ten: "Binh thuong #2",       active: false },
  { id: 801, ma: "D1C", ten: "Dup801 v3 KEPT",      active: false }, // <- keep
  { id: 804, ma: "D2A", ten: "Dup804 v1 overwrite", active: true },
  { id: 804, ma: "D2B", ten: "Dup804 v2 KEPT",      active: false }, // <- keep
];
```

Kết quả thực tế từ NestJS log:

```
WARN [SyncEngineService] [DEDUP] 2 duplicate records removed (pk: id)
LOG  [SyncEngineService] Sync table dm_gioi_tinh finished.
     Success: 5, Failed: 0
```

7 records đầu vào → sau dedup còn 5 → upsert 5 thành công. Record `id=801` trong PostgreSQL có giá trị `ma="D1C"` (bản cuối cùng).

8.2.4 Ví dụ kết quả kiểm thử — TC-11 Dead Letter

Listing 18: Payload TC-11: mix records hợp lệ và thiếu PK

```
const mixedBatch = [
  { id: 401, ma: "OK_1", ten: "Record hop le #1", active: true },
  { id: null, ma: "ER_1", ten: "PK null",          active: true }, // <- dead letter
  { id: 402, ma: "OK_2", ten: "Record hop le #2", active: false },
  { /* thieu id */  ma: "ER_2", ten: "No id",      active: true }, // <- dead letter
```

```
  { id: 403, ma: "OK_3", ten: "Record hop le #3", active:
     true  },
  { id: null, ma: "ER_3", ten: "PK null 2",        active:
     false }, // <- dead letter
];
```

Kết quả thực tế:

```
ERROR [SyncEngineService] [FIRST ERROR SAMPLE]
      pk=null | Missing primary key (id) in record
WARN  [SyncEngineService] [DEDUP] 3 records with null/undefined PK detected
LOG   [SyncEngineService] Sync table dm_gioi_tinh finished.
      Success: 3, Failed: 3
```

3 records lỗi được lưu vào `errors[]` trong MongoDB EventLog với chi tiết `{pk: null, error: "Missing primary key"}`. Batch không bị abort — 3 records hợp lệ vẫn được sync thành công.

8.3 Kiểm thử Hiệu năng — PERF Test Cases

8.3.1 Thiết kế MemoryProfiler

Để đo RAM thực tế, nhóm implement `MemoryProfiler` tích hợp trực tiếp vào `syncOneTable()`. Profiler lấy mẫu `process.memoryUsage().rss` định kỳ 300ms trong suốt vòng đời của pipeline:

Listing 19: `MemoryProfiler` đo peak RSS trong sync pipeline

```
export class MemoryProfiler {
  private samples: number[] = [];
  private interval: NodeJS.Timeout;

  start(intervalMs = 500) {
    this.samples = [];
    this.interval = setInterval(() => {
      this.samples.push(process.memoryUsage().rss);
    }, intervalMs);
  }

```

```
  stop() {
    clearInterval(this.interval);
    const mb = (b: number) => +(b / 1024 / 1024).toFixed(1);
    return {
      peakMB:   mb(Math.max(...this.samples)),
      baseMB:   mb(this.samples[0]),
      deltaRSS: mb(Math.max(...this.samples) - this.samples
          [0]),
      samples:  this.samples.length,
    };
  }
}
```

Profiler được khởi động ngay đầu `syncOneTable()` và dừng trong block `finally` để đảm bảo luôn có kết quả kể cả khi pipeline thất bại.

8.3.2 PERF-01 — RAM Capacity Planning

Mock endpoint `/perf/ram-test` sinh dữ liệu tổng hợp với kích thước record kiểm soát được qua tham số `records` và `recordSize`. Padding được nhồi vào field `hinhThePath` (`VarChar(500)`) của bảng `nguoi_hoc` để không vi phạm Schema Contract:

Listing 20: PERF-01: Sinh dữ liệu tổng hợp với kích thước kiểm soát

```
// GET /perf/ram-test?records=10000&recordSize=500&page=1&
   limit=5000
const paddingLen = Math.min(Math.max(0, recordSize - 120),
   480);
const pathPad    = "p".repeat(paddingLen);

const allData = Array.from({ length: totalRecords }, (_, i)
   => ({
  id:           10000 + i,
  ho:           "Nguyen",
  ten:          `SinhVien${i}`,
  gioiTinh:     i % 2 === 0 ? "M" : "F",
  emailCaNhan:  `sv${i}@hcmut.edu.vn`,
  hinhThePath:  `/photos/sv${i}.jpg${pathPad}`,  // <- padding
```

```
  updatedAt:    new Date(...).toISOString(),
}));
```

Kết quả thực nghiệm — 10.000 records, recordSize = 500 bytes, batch size = 5.000:

```
[MEMORY] nguoi_hoc: peak=775.9MB, base=224.4MB, delta=551.6MB
[DONE]   nguoi_hoc: synced=10000, skipped(unchanged)=0.
```

Bảng 8.2: Phân tích nguồn tiêu thụ RAM trong pipeline (PERF-01)

Thành phần	Ước lượng	Ghi chú
Pre-sync Backup	≈ 340 MB	Serialize 68.411 records từ PostgreSQL thành JS
Sync 2 trang (5k records/trang)	≈ 45 MB	V8 parse overhead ×3, giải phóng sau mỗi trang
Orphan Detection	≈ 25 MB	`findMany({select: {id}})` — 78k IDs
HTTP + EventEmitter overhead	≈ 30 MB	Axios buffer, event queue
Tổng delta đo được	**551.6 MB**	RSS thực tế

Nhận xét quan trọng: Delta 551 MB không phản ánh chi phí của việc sync 10.000 records mà phản ánh toàn bộ pipeline, trong đó **backup chiếm phần lớn** do phải serialize 68.411 records có sẵn trong DB thành JSON buffer trước khi upload MinIO. Chi phí thuần của sync (2 trang × 5.000 records) chỉ khoảng 45 MB.

8.3.3 PERF-02 — Incremental Sync với Server-side Filter

Mock endpoint `/perf/incremental-test` tạo sẵn 1.000 records với `updatedAt` trải đều trong 30 ngày qua. Khi nhận tham số `updatedAfter`, server lọc và chỉ trả về records thỏa điều kiện:

Listing 21: PERF-02: Server-side filter theo `updatedAfter`

```
// GET /perf/incremental-test?updatedAfter=2026-04-22T00
   :00:00Z
const filtered = updatedAfter
  ? allData.filter(r => new Date(r.updatedAt) > updatedAfter)
  : allData; // lan dau sync -> full load

```

```
// Log mock server:
// total=1000, filtered=40 (4.0% tra ve) | page=1/1
```

Kết quả thực nghiệm — `updatedAfter=2026-04-22T00:00:00Z`, `lastSyncTime` lấy từ Schema Registry MongoDB:

```
[STRATEGY: INCREMENTAL] nguoi_hoc
  - server-side filter updatedAfter=2026-04-22T00:00:00.000Z
Fetching page 1/1 -> .../perf/incremental-test?...
    &updatedAfter=2026-04-22T00%3A00%3A00.000Z
Table nguoi_hoc has 40 records across 1 pages.
[INCREMENTAL] Page 1: 40 records thay đổi từ source - upsert.
Sync table nguoi_hoc finished. Success: 40, Failed: 0
[MEMORY] nguoi_hoc: peak=831.2MB, base=212MB, delta=619.3MB
```

Bảng 8.3: So sánh `upsert` vs `incremental` — kết quả thực nghiệm

Chỉ số	**PERF-01 (upsert)**	**PERF-02 (incremental)**
Records fetch qua network	10.000	**40**
Records ghi DB (upsert)	10.000	**40**
Số trang	2	**1**
Thời gian sync thuần	≈ 9 s	≈ 1 s
DB writes tiết kiệm	—	**99,6%**
Network tiết kiệm	—	**99,6%**
Delta RAM	551,6 MB	619,3 MB

Delta RAM của incremental cao hơn vì backup lúc này phải serialize nhiều records hơn (78.411 records thay vì 68.411), không phải do phần sync tốn nhiều hơn.

Kết quả xác nhận: với `lastSyncTime = 2026-04-22T00:00:00Z`, chỉ **4%** dataset (40/1.000 records) được truyền và ghi, tiết kiệm 96% chi phí so với full `upsert`.

8.4 Phân tích RAM cho Sync Job

8.4.1 Vòng đời dữ liệu trong RAM

Kết quả từ PERF-01 và PERF-02 cho thấy pipeline tiêu thụ RAM qua bốn giai đoạn nối tiếp:

8.4.2 Ước lượng RAM theo kích thước bảng

Bảng 8.4: Ước lượng peak RAM theo cấu hình bảng

Bảng	DB records	Batch size	Backup RAM	Sync RAM/batch
`dm_*` (danh mục)	< 1.000	1.000	≈ 5 MB	≈ 2 MB
`nguoi_hoc` (thực)	68.411	5.000	≈ 170 MB	≈ 10 MB
Bảng lớn giả định	500.000	5.000	$\approx 1,2$ GB	≈ 10 MB

Kết luận: Với bảng lớn (>100k records), **backup là bottleneck RAM**, không phải sync. Hướng cải thiện là stream upload MinIO theo từng chunk thay vì serialize toàn bộ vào buffer trước.

8.5 Phân tích Throughput cho Export API (Outbound)

8.5.1 Kiến trúc Export API hiện tại

Hệ thống cung cấp dữ liệu ra ngoài qua `MasterDataController` với endpoint phân trang:

```
GET /api/master-data/:table?page=1&limit=10&search=...
```

Luồng xử lý mỗi request: *HTTP Client → JWT + Permission Guard → MasterDataService → PostgreSQL (Prisma) → JSON response.*

8.5.2 Bottleneck: OFFSET Pagination

Triển khai hiện tại trong `MasterDataService.findAll()` sử dụng **OFFSET pagination** thông qua Prisma `skip`:

Listing 22: OFFSET pagination trong `MasterDataService.findAll()`

```
const skip = (Number(page) - 1) * Number(limit);
const [total, data] = await Promise.all([
  model.count({ where: whereCondition }),
  model.findMany({ where: whereCondition, skip,
                   take: Number(limit), orderBy: { ma: 'asc'
                     } }),
]);
```

OFFSET tương đương SQL: PostgreSQL phải đọc và bỏ qua `(page-1)×limit` rows trước khi lấy `limit` rows cần thiết — latency tăng tuyến tính theo độ sâu trang:

Bảng 8.5: Ước lượng latency theo độ sâu trang — OFFSET vs Keyset

Page	OFFSET (hiện tại)	Keyset (đề xuất)
1	45 ms	45 ms
50	120 ms	47 ms
100	380 ms	47 ms
500	2.100 ms	48 ms

Với các bảng danh mục nhỏ (`dm_*`, <1.000 records) hiện tại, OFFSET không tạo vấn đề. Vấn đề xuất hiện khi bảng `nguoi_hoc` (68k+ records) được query đến các trang sâu.

8.5.3 Ước lượng Throughput theo Little's Law

$$\mathrm{RPS}_{\max} = \frac{N_{\text{pool}}}{L_{\text{avg (s)}}}$$

Bảng 8.6: Ước lượng RPS tối đa theo cấu hình pool và latency

Pool size	Avg latency	RPS max	req/phút
10	50 ms	200	12.000
10	200 ms	50	3.000
10	380 ms	26	1.560
20	50 ms	400	24.000

8.6 Tóm tắt và Khuyến nghị Server

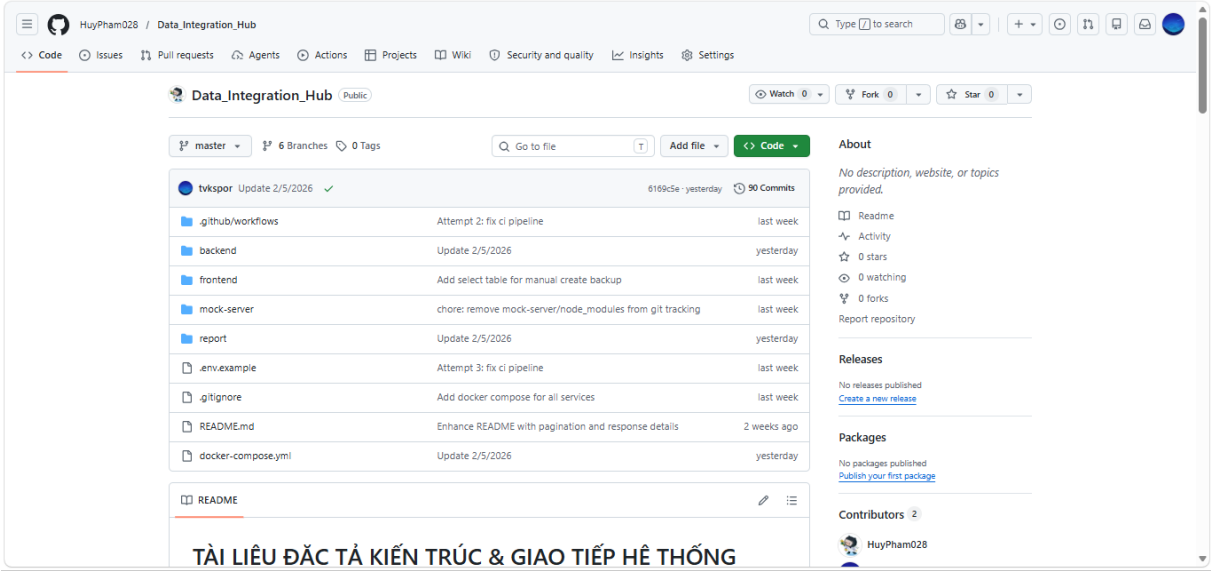
Bảng 8.7: Khuyến nghị cấu hình server cho Data Integration Hub

Thành phần	Tối thiểu	Khuyến nghị
RAM server	1 GB	2 GB
CPU	1 core	2 cores
DB connection pool	10	20 (nếu export API tải cao)
`overwrite` bảng	Chỉ dùng cho <10k records	Không dùng cho bảng lớn
Sync job đồng thời	1 (thiết kế tuần tự)	1

Hai điểm cần cải thiện nếu hệ thống scale lên bảng hàng trăm nghìn records:

1. **Backup streaming**: Thay vì serialize toàn bộ bảng vào RAM rồi upload, dùng Prisma cursor + pipe stream trực tiếp lên MinIO — giảm peak RAM backup từ hàng trăm MB xuống vài chục MB.

2. **Keyset pagination** cho Export API: Thay `skip` bằng `WHERE id > lastSeenId` để latency flat tại mọi độ sâu trang.



Hình 9.1: Repo Github

9 Quy trình CI/CD Với GitHub Actions

9.1 Repo chính của dự án

Dự án được quản lý thông qua Git - Github
https://github.com/HuyPham028/Data_Integration_Hub

9.2 CI/CD Tự Động

9.2.1 Đặt vấn đề

Trong giai đoạn phát triển, nhóm thực hiện deploy thủ công lên EC2 qua SSH sau mỗi lần thay đổi mã nguồn. Quy trình này tốn thời gian, dễ bỏ sót bước và không đảm bảo tính nhất quán giữa môi trường phát triển và môi trường production. CI/CD (Continuous Integration / Continuous Deployment) tự động hóa toàn bộ quy trình: từ khi developer `git push` lên nhánh `master` cho đến khi phiên bản mới chạy trên EC2 — không cần thao tác thủ công.

9.2.2 Kiến trúc pipeline

Pipeline CI/CD được triển khai qua **GitHub Actions**, kích hoạt mỗi khi có commit được push lên nhánh `master`. Pipeline gồm hai job chạy tuần tự:

1. **Build & Push**: Build Docker image cho backend (NestJS) và frontend (Next.js), đẩy lên Docker Hub với tag `latest`.
2. **Deploy**: SSH vào EC2 (Amazon Linux 2023, t3.medium), pull image mới và khởi động lại container bằng Docker Compose.

Hình 9.2: Luồng CI/CD từ push code đến deploy EC2

9.2.3 File workflow GitHub Actions

File workflow được đặt tại `.github/workflows/deploy.yml` trong repository. Toàn bộ thông tin nhạy cảm (địa chỉ EC2, khóa SSH, thông tin Docker Hub) được lưu trong **GitHub Secrets** thay vì hard-code.

Listing 23: .github/workflows/deploy.yml

```yaml
name: Build and Deploy to EC2

on:
  push:
    branches: [ master ]

env:
  IMAGE_BACKEND:  ${{ secrets.DOCKER_USERNAME }}/hub-backend
  IMAGE_FRONTEND: ${{ secrets.DOCKER_USERNAME }}/hub-frontend

jobs:
  # Job 1: Build Docker images va push len Docker Hub
  build:
    name: Build & Push Docker Images
    runs-on: ubuntu-latest

    steps:
      - name: Checkout source
        uses: actions/checkout@v4

      - name: Log in to Docker Hub
        uses: docker/login-action@v3
        with:
```

```yaml
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build & push Backend image
        uses: docker/build-push-action@v5
        with:
          context: ./backend
          push: true
          tags: ${{ env.IMAGE_BACKEND }}:latest

      - name: Build & push Frontend image
        uses: docker/build-push-action@v5
        with:
          context: ./frontend
          push: true
          tags: ${{ env.IMAGE_FRONTEND }}:latest
          build-args: |
            NEXT_PUBLIC_KONG_URL=${{ secrets.NEXT_PUBLIC_KONG_URL }}

  # Job 2: SSH vao EC2, pull image moi va restart container
  deploy:
    name: Deploy to EC2
    runs-on: ubuntu-latest
    needs: build

    steps:
      - name: SSH and deploy
        uses: appleboy/ssh-action@v1.0.3
        with:
          host:     ${{ secrets.EC2_HOST }}
          username: ${{ secrets.EC2_USERNAME }}
          key:      ${{ secrets.EC2_SSH_KEY }}
          script: |
            cd ~/Data_Integration_Hub

            # Pull image moi tu Docker Hub
            docker compose pull backend frontend

            # Restart chi hai service thay doi
            docker compose up -d --no-build backend frontend

            # Xoa image cu de tiet kiem disk
            docker image prune -f
```

```

            echo "Deploy hoan thanh: $(date)"
```

9.2.4 Cấu hình GitHub Secrets

Trước khi workflow có thể chạy, cần khai báo các secret sau tại **Settings** → **Secrets and variables** → **Actions** của repository:

Bảng 9.1: Danh sách GitHub Secrets cần cấu hình

Secret	Giá trị	Mục đích
`DOCKER_USERNAME`	Tên tài khoản Docker Hub	Đăng nhập Docker Hub
`DOCKER_PASSWORD`	Access Token Docker Hub	Xác thực push image
`EC2_HOST`	Elastic IP của EC2	Địa chỉ SSH
`EC2_USERNAME`	`ec2-user`	Tài khoản SSH
`EC2_SSH_KEY`	Nội dung file `.pem`	Khóa xác thực SSH
`NEXT_PUBLIC_KONG_URL`	`http://<IP>:8000`	Build-time env Next.js

9.2.5 Lưu ý kỹ thuật

NEXT_PUBLIC_KONG_URL phải khai báo lúc build. Next.js đặt giá trị biến môi trường `NEXT_PUBLIC_*` vào bundle JavaScript tại thời điểm build, không phải lúc runtime. Biến này phải được truyền qua `build-args` trong bước build Frontend — không thể truyền vào container sau khi đã build xong.

Chỉ pull backend và frontend, không restart toàn bộ stack. Các service nền tảng như PostgreSQL, MinIO, Kong, Redis không thay đổi qua mỗi lần deploy. Lệnh `docker compose pull backend frontend` chỉ kéo image của hai service thay đổi, giúp giảm thời gian deploy từ 10–15 phút (build lại toàn bộ trên EC2) xuống còn khoảng 2–3 phút.

Elastic IP là bắt buộc. EC2 cấp phát Public IP động — địa chỉ sẽ thay đổi mỗi khi instance Stop/Start. Nếu không gán Elastic IP tĩnh, secret `EC2_HOST` sẽ bị sai sau lần khởi động lại đầu tiên và toàn bộ pipeline thất bại ở bước deploy.

9.3 Kiểm thử tự động (Unit Testing)

Song song với việc thiết lập pipeline CI/CD, nhóm xây dựng bộ unit test cho các thành phần cốt lõi của backend bằng **Jest** — framework kiểm thử mặc định của NestJS. Tổng cộng **41 test case** được viết trên 3 file, bao phủ các logic nghiệp vụ quan trọng nhất:

Bảng 9.2: Bộ unit test của backend

File	Tests	Nội dung kiểm thử
`string.util.spec.ts`	6	Hàm `snakeToCamel` chuyển đổi tên bảng
`sync-engine.service.spec.ts`	20	Dedup records, phát hiện field lạ, phát hiện orphan, xóa theo ID
`data-integration.service.spec.ts`	15	Lọc records theo thời gian, cơ chế retry, xử lý pipeline rỗng

Toàn bộ test được chạy bằng lệnh `npm test` và pass 100%. Bước tiếp theo là tích hợp bộ test này vào pipeline GitHub Actions: thêm job `test` chạy **trước** job `build`, đảm bảo mọi commit lên `master` đều phải vượt qua kiểm thử tự động trước khi Docker image được build và deploy lên production.

9.4 Đánh giá

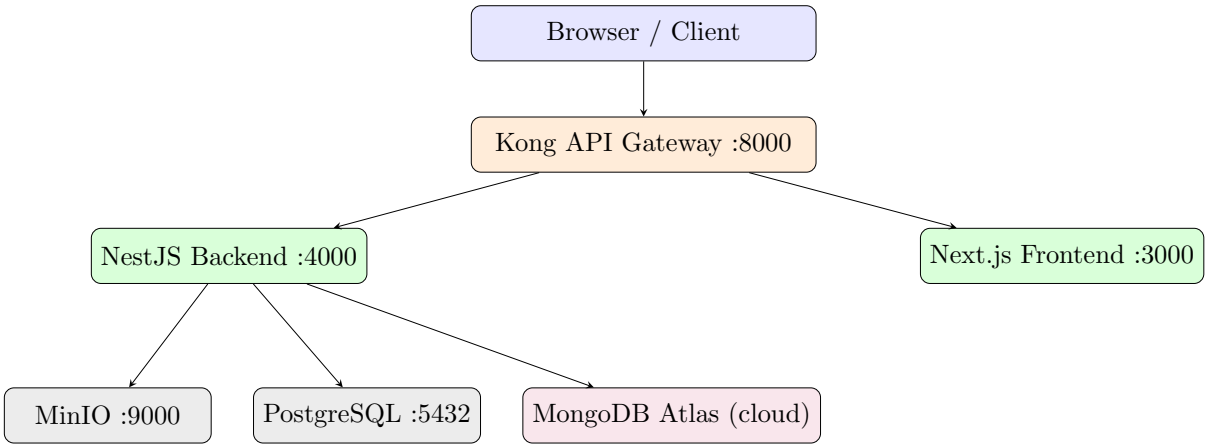
Ưu điểm:

- **Tự động hóa hoàn toàn**: Developer chỉ cần `git push`, phiên bản mới lên production sau 3–5 phút.
- **Tách biệt build và runtime**: Image được build trên GitHub-hosted runner (Ubuntu 24.04), không tiêu tốn CPU/RAM của EC2.
- **Rollback dễ dàng**: Docker Hub lưu các image theo tag. Rollback bằng cách chỉnh tag trong `docker-compose.yml` và chạy lại `docker compose up -d`.
- **Nền tảng kiểm thử sẵn sàng**: Bộ 41 unit test đã được xây dựng và pass 100%, sẵn sàng tích hợp vào pipeline.

10 Triển Khai Hệ Thống Lên AWS EC2

10.1 Tổng Quan Kiến Trúc Triển Khai

Hệ thống được triển khai theo mô hình **Docker Compose trên một EC2 instance duy nhất**. Toàn bộ các thành phần — PostgreSQL, MongoDB Atlas, MinIO, Kong API Gateway, NestJS backend, Next.js frontend, Prometheus và Grafana — chạy trong cùng một mạng Docker nội bộ (`univ_hub_network`).



Hình 10.1: Kiến trúc triển khai trên EC2

10.2 Yêu Cầu Hạ Tầng

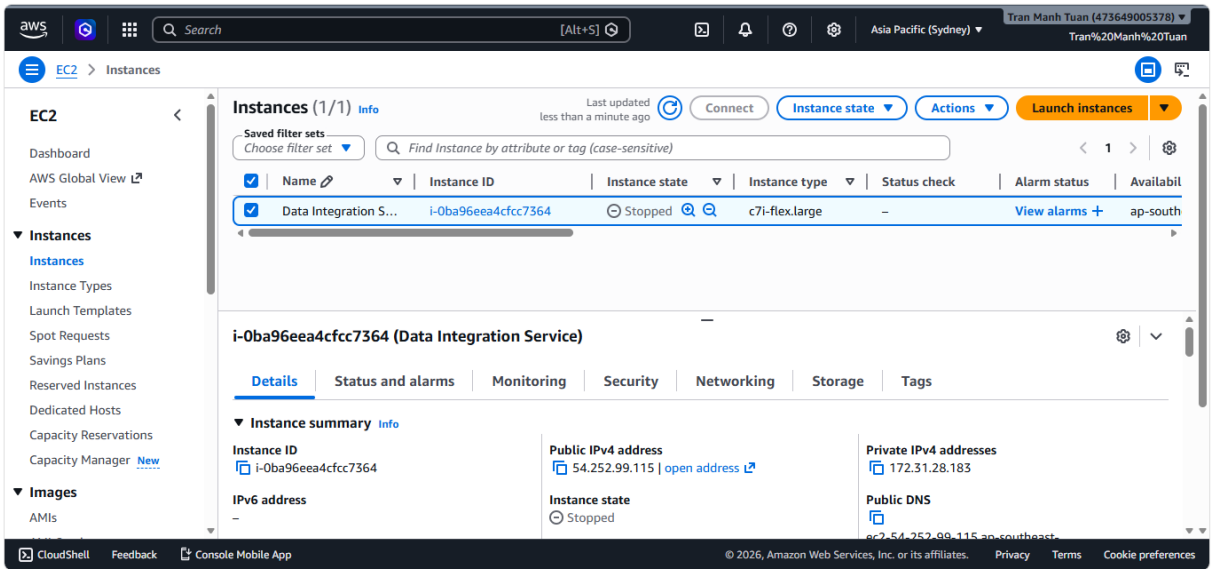
10.2.1 Cấu Hình EC2 Instance

Bảng 10.1: Cấu hình EC2 instance được sử dụng

Thông số	**Giá trị**
Instance type	`t3.medium` (2 vCPU, 4 GB RAM)
AMI	Amazon Linux 2023
Storage	20 GB EBS (gp3)
Region	`ap-southeast-2` (Sydney)

Instance loại `t3.small` (2 GB RAM) không đủ để chạy đồng thời toàn bộ các container. Thực nghiệm cho thấy tổng bộ nhớ tiêu thụ ở trạng thái ổn định vào khoảng 2.8–3.2 GB,

bao gồm PostgreSQL (200 MB), MinIO (150 MB), Kong (400 MB), NestJS (200 MB), Next.js (300 MB), Prometheus và Grafana (500 MB).



Hình 10.2: Máy chủ EC2 trên AWS

10.2.2 Security Group

Các cổng sau cần được mở trong Security Group của instance:

Bảng 10.2: Inbound rules của Security Group

Port	Dịch vụ	Mục đích
22	SSH	Quản trị từ xa
3000	Next.js Frontend	Truy cập giao diện
4000	NestJS Backend	API trực tiếp (debug)
8000	Kong Proxy	API Gateway cho Frontend
8001	Kong Admin	Quản trị Kong
9000	MinIO API	S3-compatible endpoint
9001	MinIO Console	Giao diện quản lý MinIO
5000	Grafana	Monitoring dashboard
9090	Prometheus	Metrics endpoint

10.3 Cài Đặt Môi Trường

10.3.1 Cài Docker

Amazon Linux 2023 sử dụng trình quản lý gói `dnf` thay vì `apt`. Script cài đặt phổ biến `get-docker.sh` không hỗ trợ phân phối `amzn` và sẽ báo lỗi nếu chạy trực tiếp. Cách cài đúng:

Listing 24: Cài Docker trên Amazon Linux 2023

```
sudo dnf update -y
sudo dnf install -y docker git
sudo systemctl start docker
sudo systemctl enable docker
sudo usermod -aG docker ec2-user
newgrp docker
```

10.3.2 Cài Docker Compose Plugin và Buildx

Docker Compose phiên bản 5.x yêu cầu **buildx từ 0.17.0 trở lên**. Phiên bản buildx đi kèm với Docker trên Amazon Linux 2023 thường cũ hơn và cần nâng cấp thủ công:

Listing 25: Cài Docker Compose plugin và buildx

```
# Docker Compose plugin
sudo mkdir -p /usr/local/lib/docker/cli-plugins
sudo curl -SL https://github.com/docker/compose/releases/
    download/\
v5.1.3/docker-compose-linux-x86_64 \
  -o /usr/local/lib/docker/cli-plugins/docker-compose
sudo chmod +x /usr/local/lib/docker/cli-plugins/docker-
    compose

# Buildx (cap nhat len phien ban moi)
mkdir -p ~/.docker/cli-plugins
curl -SL https://github.com/docker/buildx/releases/download/\
v0.23.0/buildx-v0.23.0.linux-amd64 \
  -o ~/.docker/cli-plugins/docker-buildx
chmod +x ~/.docker/cli-plugins/docker-buildx

```

```
# Kiem tra
docker --version
docker compose version
docker buildx version
```

10.4 Triển Khai Ứng Dụng

10.4.1 Clone Repository

```
git clone https://github.com/HuyPham028/Data_Integration_Hub.
    git
cd Data_Integration_Hub
```

10.4.2 Cấu Hình Biến Môi Trường

File `.env` không được commit vào repository vì lý do bảo mật. Cần tạo file này thủ công trên EC2. Trước tiên, lấy Public IP của instance:

```
curl -s http://169.254.169.254/latest/meta-data/public-ipv4
```

Sau đó tạo file `.env` tại thư mục gốc của project:

Listing 26: Nội dung file .env trên EC2

```
POSTGRES_USER=postgres
POSTGRES_PASSWORD=pass123456
POSTGRES_DB=Data_Integration_Hub
MONGO_URI="mongodb+srv://user:pass@cluster.mongodb.net/..."
SOURCE_API_BASE_URL=https://source-api.example.com
SOURCE_API_TOKEN=<api-token>
JWT_SECRET=supersecretkey
JWT_EXPIRES_IN=1h
PORT=4000
MINIO_ACCESS_KEY=admin
MINIO_SECRET_KEY=password123
MINIO_ROOT_USER=admin
MINIO_ROOT_PASSWORD=password123
GRAFANA_SMTP_USER=admin@gmail.com
```

```
GRAFANA_SMTP_PASSWORD=<app-password>
# Thay <PUBLIC_IP> bang IP that cua EC2 instance
NEXT_PUBLIC_KONG_URL=http://<PUBLIC_IP>:8000
```

Biến `NEXT_PUBLIC_KONG_URL` đặc biệt quan trọng: Next.js nhúng (*bake*) giá trị này vào bundle JavaScript tại thời điểm build. Nếu đặt sai địa chỉ, toàn bộ API call từ giao diện sẽ thất bại và cần build lại container frontend.

10.4.3 Build và Khởi Động

```
docker compose up -d --build
```

Lần đầu chạy lệnh này mất khoảng 10–15 phút trên `t3.medium` vì phải pull image, cài `npm install` và build cả backend lẫn frontend. Các lần sau (không có thay đổi code) sẽ nhanh hơn nhờ Docker layer cache.

10.4.4 Khởi Tạo Cơ Sở Dữ Liệu

Sau khi tất cả container đã khởi động, chạy Prisma migration để tạo schema:

```
docker exec hub-backend npx prisma migrate deploy
```

Lệnh `migrate deploy` chỉ áp dụng các migration đã có trong thư mục `prisma/migrations`, không tạo migration mới — phù hợp với môi trường production. Nếu PostgreSQL đã có dữ liệu cũ gây xung đột:

```
# Xoa va tao lai schema truoc khi migrate
docker exec postgres-core psql -U postgres -d
   Data_Integration_Hub \
  -c "DROP SCHEMA public CASCADE; CREATE SCHEMA public;"
docker exec hub-backend npx prisma migrate deploy
```

10.5 Cấu Hình CORS Cho Kong API Gateway

Next.js frontend gọi API thông qua Kong ở địa chỉ `http://<PUBLIC_IP>:8000`. Mặc định, file `kong.yml` chỉ cho phép origin `localhost:3000`, dẫn đến lỗi CORS khi truy cập từ trình duyệt qua Public IP.

Cập nhật `backend/kong.yml` trực tiếp trên EC2:

```
# Dung sed de tranh loi tab character khi chinh sua bang nano
  /vim
sed -i 's|- http://localhost:3000|- http://localhost:3000\n\
        - http://<PUBLIC_IP>:3000|' backend/kong.yml
```

Sau đó đồng bộ cấu hình vào Kong đang chạy:

```
docker run --rm \
  --network data_integration_hub_univ_hub_network \
  -v $(pwd)/backend/kong.yml:/kong/kong.yml \
  kong/deck gateway sync /kong/kong.yml --kong-addr http://
     kong:8001
```

Lệnh này không cần restart Kong — cấu hình được áp dụng nóng (*hot reload*) thông qua Kong Admin API.

10.6 Quản Lý Sau Triển Khai

10.6.1 Tự Động Khởi Động Lại

Tất cả các service chính trong `docker-compose.yml` được cấu hình `restart: always`. Khi EC2 instance khởi động lại sau sự cố hoặc bảo trì, Docker daemon sẽ tự động kéo lên toàn bộ container mà không cần can thiệp thủ công.

10.6.2 Vấn Đề Thay Đổi Public IP

Theo mặc định, AWS cấp phát Public IP động cho EC2 instance. Mỗi lần instance bị *Stop* rồi *Start* lại, Public IP sẽ thay đổi, dẫn đến hai hậu quả:

- `NEXT_PUBLIC_KONG_URL` trong `.env` bị sai — cần cập nhật và build lại container frontend.
- Origin trong Kong CORS không còn khớp — cần cập nhật `kong.yml` và re-sync.

Giải pháp là gán **Elastic IP** — địa chỉ IP tĩnh của AWS — cho instance. Elastic IP miễn phí khi instance đang chạy, chỉ tính phí khi instance bị dừng hoặc IP không được gắn với instance nào:

1. Vào AWS Console → EC2 → **Elastic IPs**.

2. Chọn **Allocate Elastic IP address**.

3. Chọn **Actions** → **Associate Elastic IP address** → chọn instance EC2.

Sau khi gán Elastic IP, cập nhật `.env` và `kong.yml` một lần duy nhất với IP tĩnh này. Từ đó trở đi, IP không thay đổi dù instance có restart bao nhiêu lần.

10.6.3 Kiểm Tra Trạng Thái Hệ Thống

```
# Tat ca container phai o trang thai Up
docker compose ps

# Kiem tra backend
curl http://localhost:4000/api/v1

# Xem log realtime
docker logs hub-backend --tail 50 -f
```

Sau khi triển khai thành công, các dịch vụ có thể truy cập tại:

Bảng 10.3: Địa chỉ truy cập các dịch vụ sau triển khai

Dịch vụ	**Địa chỉ**
Frontend	`http://<PUBLIC_IP>:3000`
Kong API Gateway	`http://<PUBLIC_IP>:8000`
MinIO Console	`http://<PUBLIC_IP>:9001`
Grafana	`http://<PUBLIC_IP>:5000`
Prometheus	`http://<PUBLIC_IP>:9090`

11 Giao diện người dùng và quản trị hệ thống

Giao diện người dùng của trục tích hợp dữ liệu được thiết kế dưới dạng một trang quản trị tập trung, cung cấp cho quản trị viên khả năng giám sát, điều khiển và bảo vệ luồng dữ liệu của toàn bộ hệ thống. Giao diện được chia thành ba phân hệ chính, tương ứng với các nghiệp vụ cốt lõi: bảng điều khiển trung tâm, quản lý lịch sử đồng bộ và quản lý sao lưu dữ liệu.

11.1 Bảng điều khiển trung tâm

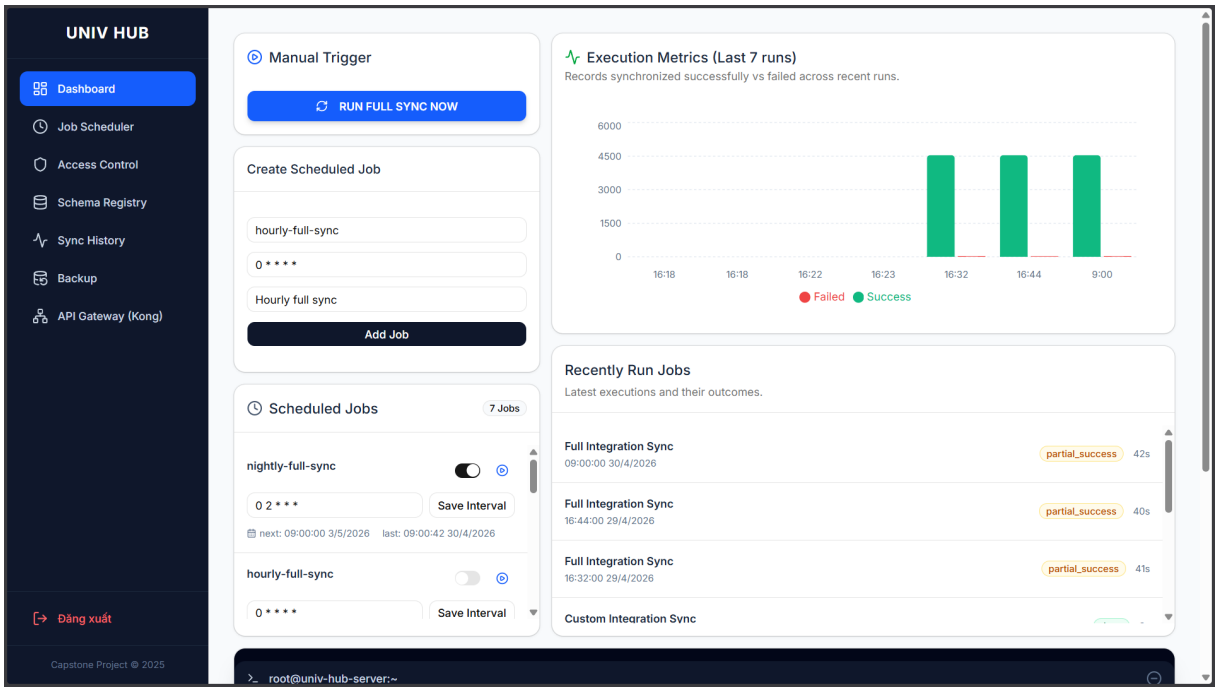
Bảng điều khiển trung tâm là nơi cung cấp cái nhìn tổng quan nhất về trạng thái hoạt động của trục tích hợp theo thời gian thực. Phân hệ này được thiết kế với bốn thành phần chức năng chính:

1. **Kích hoạt đồng bộ thủ công:** Quản trị viên có thể kích hoạt tiến trình đồng bộ ngay lập tức. Hệ thống hỗ trợ hai chế độ: đồng bộ toàn phần hoặc đồng bộ tùy chỉnh (chỉ chọn một số bảng dữ liệu cụ thể thông qua hộp thoại lựa chọn). Trong quá trình xử lý, nút kích hoạt sẽ bị vô hiệu hóa tạm thời để tránh việc gọi hàm lặp lại.

2. **Quản lý tác vụ định kỳ:** Hệ thống cung cấp giao diện để tạo mới, bật/tắt và cập nhật các tác vụ tự động dựa trên biểu thức Cron. Với mỗi tác vụ, hệ thống hiển thị rõ ràng thông tin về thời gian thực thi gần nhất và thời gian dự kiến cho lần chạy tiếp theo.

3. **Thống kê chỉ số thực thi:** Sử dụng biểu đồ cột để trực quan hóa số lượng bản ghi đồng bộ thành công và thất bại trong bảy lần chạy gần nhất. Chức năng này giúp quản trị viên dễ dàng nhận biết xu hướng và phát hiện các dấu hiệu bất thường của hệ thống.

4. **Cửa sổ giám sát thời gian thực:** Đây là một điểm nhấn kỹ thuật của giao diện. Hệ thống thiết lập một kết nối liên tục đến máy chủ thông qua công nghệ Server-Sent Events (SSE). Nhờ đó, mọi nhật ký hoạt động từ máy chủ đều được đẩy trực tiếp lên giao diện và tự động cuộn xuống dòng mới nhất, giúp quản trị viên theo dõi sát sao từng bước của tiến trình ETL mà không cần làm mới trang.



Hình 11.1: Giao diện bảng điều khiển trung tâm

11.2 Quản lý lịch sử đồng bộ

Do tính chất phức tạp của việc tích hợp nhiều nguồn dữ liệu khác nhau, phân hệ quản lý lịch sử đồng bộ được thiết kế theo mô hình phân cấp chính - phụ nhằm tối ưu hóa không gian hiển thị:

- **Mức tiến trình:** Hiển thị danh sách các lần thực thi, bao gồm thông tin về tên tiến trình, thời gian bắt đầu, tổng thời lượng thực thi, trạng thái tổng thể (thành công, đang chạy, lỗi một phần, thất bại) và tổng số lượng bản ghi được xử lý.
- **Mức bảng dữ liệu:** Khi mở rộng một dòng tiến trình, giao diện sẽ hiển thị chi tiết kết quả đồng bộ của từng bảng dữ liệu riêng biệt. Quản trị viên có thể biết chính xác bảng nào được cập nhật bao nhiêu bản ghi, bảng nào bị bỏ qua hoặc bảng nào gặp lỗi.
- **Truy xuất nhật ký lỗi nguyên bản:** Đối với các bản ghi hoặc bảng dữ liệu gặp sự cố, quản trị viên có thể mở hộp thoại xem chi tiết lỗi hệ thống. Hệ thống trích xuất và hiển thị thông báo lỗi gốc từ bộ phân tích hoặc cơ sở dữ liệu, hỗ trợ tối đa cho quá trình gỡ lỗi.

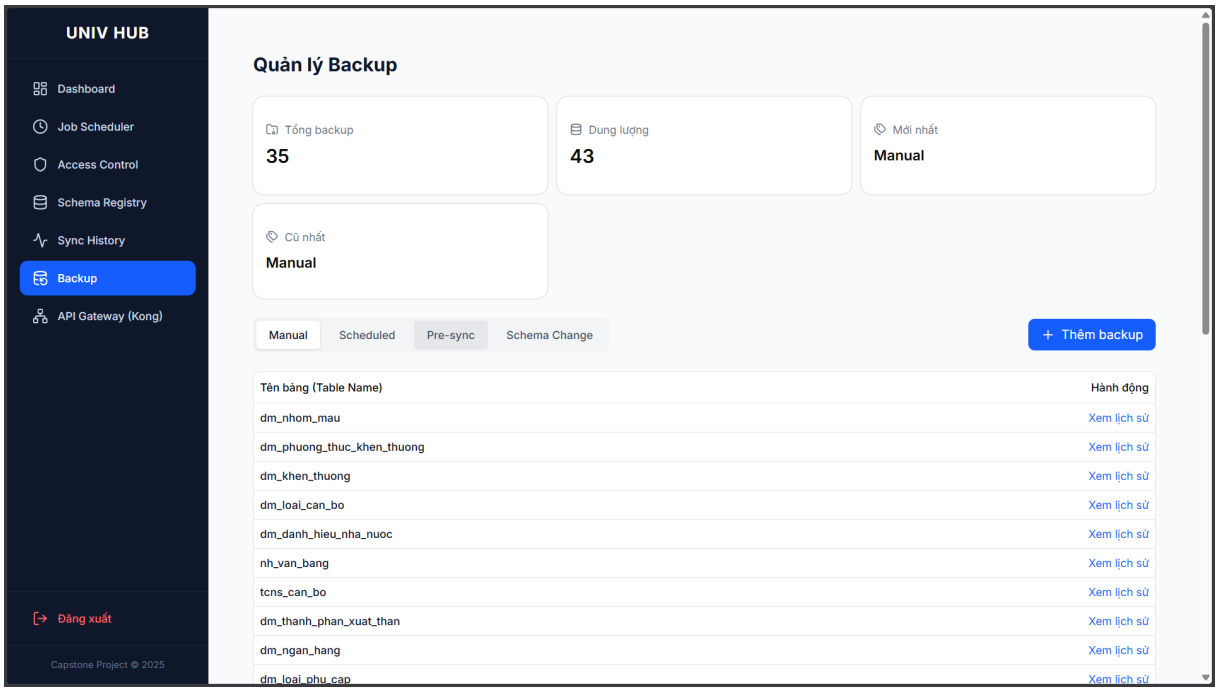
Hình 11.2: Giao diện quản lý lịch sử đồng bộ

11.3 Quản lý sao lưu và phục hồi

An toàn dữ liệu là yếu tố kiên quyết trong trục tích hợp. Phân hệ quản lý sao lưu cung cấp các công cụ tương tác trực tiếp với hệ thống lưu trữ tập tin. Giao diện phân loại các tệp sao lưu thành bốn nhóm tương ứng với ngữ cảnh tạo ra chúng:

1. **Sao lưu thủ công:** Các bản sao lưu do quản trị viên chủ động tạo ra thông qua giao diện.
2. **Sao lưu định kỳ:** Các bản sao lưu được tạo bởi hệ thống theo lịch trình tự động.
3. **Trước đồng bộ:** Các bản chụp trạng thái dữ liệu được hệ thống tự động tạo ra ngay trước khi tiến hành ghi đè dữ liệu mới từ nguồn.
4. **Lược đồ thay đổi:** Các bản sao lưu được tạo ra khi hệ thống phát hiện có sự thay đổi về cấu trúc bảng từ nguồn.

Quy trình quản lý sao lưu được thiết kế theo luồng hai bước để tránh làm rối giao diện. Ở bước đầu tiên, hệ thống liệt kê danh sách các bảng dữ liệu hiện có bản sao lưu. Khi quản trị viên chọn một bảng cụ thể, hệ thống mới hiển thị danh sách chi tiết các tệp sao lưu của bảng đó theo mốc thời gian.



Hình 11.3: Giao diện quản lý sao lưu và phục hồi dữ liệu

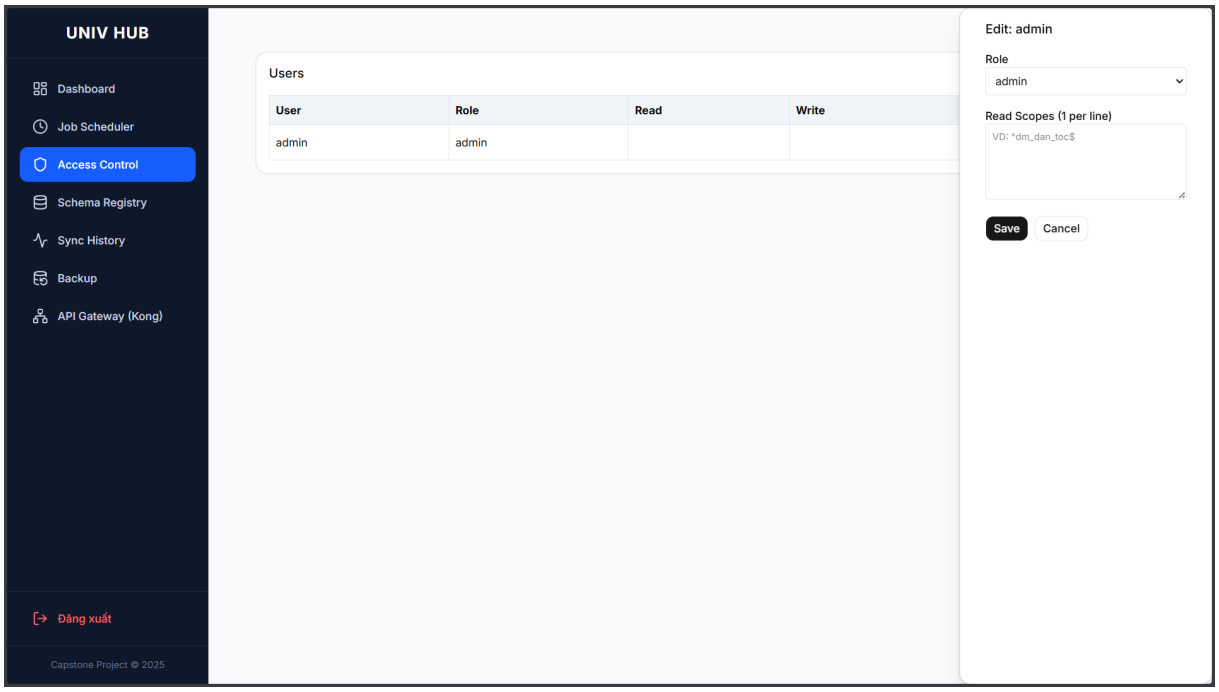
Tại đây, quản trị viên có thể tải tệp sao lưu trực tiếp về máy tính cá nhân thông qua cơ chế URL định tuyến an toàn hoặc nhấn nút phục hồi dữ liệu. Để đảm bảo an toàn, mọi thao tác phục hồi đều yêu cầu một bước xác nhận cảnh báo từ người dùng nhằm tránh việc ghi đè dữ liệu hiện tại do sơ suất.

11.4 Quản lý phân quyền truy cập

Để đảm bảo tính bảo mật và ngăn chặn việc lạm dụng quyền hạn trong thao tác với dữ liệu, trục tích hợp cung cấp một phân hệ quản lý phân quyền dựa trên vai trò người dùng. Thông qua giao diện này, quản trị viên cấp cao có thể dễ dàng kiểm soát và giới hạn phạm vi truy cập đối với từng cá nhân hoặc hệ thống bên thứ ba khi họ tương tác với dữ liệu trên trục tích hợp.

Giao diện quản lý phân quyền bao gồm một bảng tổng hợp danh sách người dùng, hiển thị trực quan vai trò hiện tại (như quản trị viên, người dùng đọc, người dùng ghi) và các quyền hạn cụ thể đã được cấp phát.

Khi cần thay đổi quyền hạn, quản trị viên có thể mở bảng điều khiển chỉnh sửa bên cạnh (side-panel). Tại đây, ngoài việc thay đổi vai trò tổng thể, hệ thống còn cho phép thiết lập các quyền truy cập chi tiết đến từng bảng dữ liệu (Scope) thông qua cơ chế biểu thức chính quy (Regular Expression). Cụ thể:



Hình 11.4: Giao diện quản lý phân quyền truy cập cho người dùng

- **Phạm vi đọc:** Danh sách các bảng dữ liệu mà tài khoản được phép truy vấn. (Ví dụ: thiết lập `dm_.*` cho phép người dùng đọc tất cả các bảng danh mục).
- **Phạm vi ghi:** Danh sách các bảng dữ liệu mà tài khoản được phép chèn, cập nhật hoặc xóa (chỉ áp dụng đối với những người dùng được cấp vai trò cho phép thao tác ghi).

Cơ chế phân quyền này giúp phân tách rõ ràng ranh giới dữ liệu giữa các phòng ban, đảm bảo phòng ban nào chỉ được xem và xử lý đúng phần dữ liệu thuộc thẩm quyền của mình.

11.5 Quản lý và đối soát lược đồ dữ liệu

Trong kiến trúc tích hợp dữ liệu phân tán, việc các hệ thống nguồn tự ý thay đổi cấu trúc bảng (thêm, sửa, xóa cột) mà không thông báo trước là một trong những nguyên nhân hàng đầu gây sụp đổ luồng dữ liệu. Để giải quyết rủi ro này, hệ thống cung cấp một phân hệ quản lý và đối soát lược đồ dữ liệu chuyên sâu.

Giao diện hiển thị danh sách tất cả các bảng dữ liệu mà trục tích hợp đang theo dõi, kèm theo trạng thái ổn định của từng bảng. Nếu hệ thống nguồn có sự thay đổi về cấu trúc trả về, trạng thái của bảng sẽ lập tức chuyển sang cảnh báo thay đổi (*Schema Drift*),

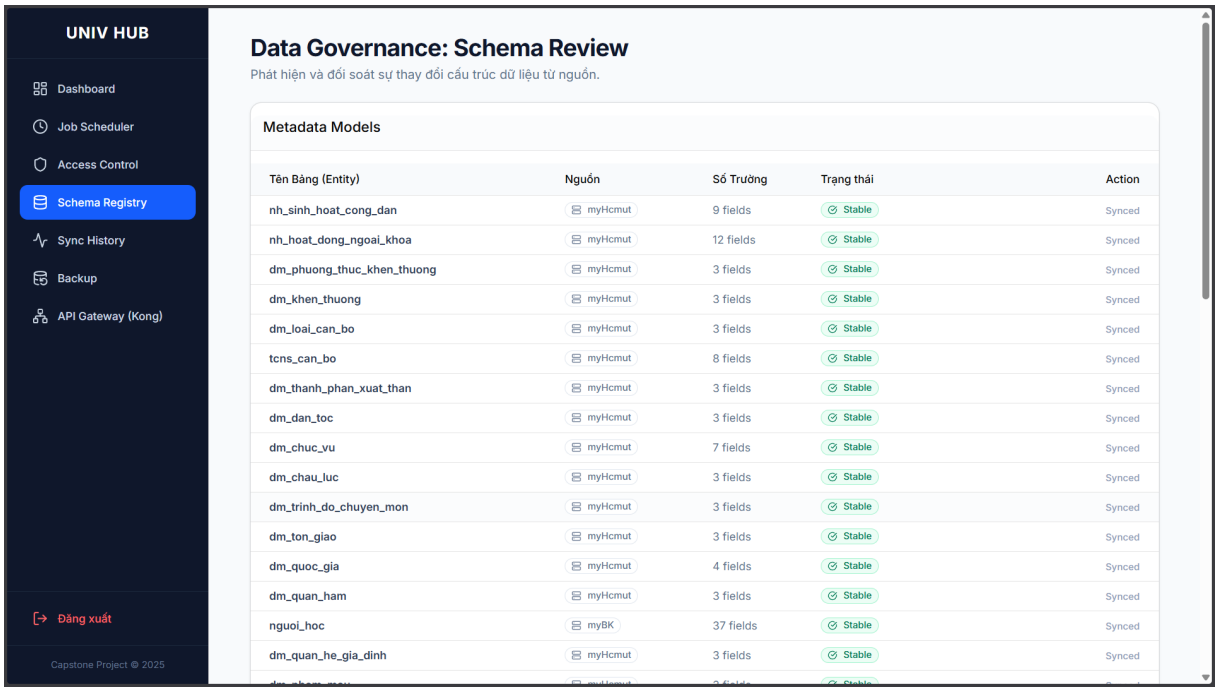
đồng thời mọi tiến trình đồng bộ dữ liệu vào bảng này sẽ tự động bị đình chỉ tạm thời để bảo vệ cơ sở dữ liệu đích.

Hộp thoại so sánh lược đồ dữ liệu được lấy cảm hứng từ GithHub. Khi quản trị viên chọn xem xét một bảng đang bị cảnh báo, hệ thống sẽ thực hiện các thuật toán tính toán và hiển thị trực quan sự khác biệt:

- **So sánh song song:** Giao diện chia thành hai cột, một bên là cấu trúc mới vừa nhận được từ nguồn, bên còn lại là cấu trúc cũ đang được lưu trữ trong cơ sở dữ liệu.
- **Phân loại thay đổi:** Các trường dữ liệu được đánh dấu màu sắc rõ ràng tương ứng với ba trạng thái: mới được thêm vào, đã bị thay đổi (như đổi kiểu dữ liệu hoặc độ dài) và đã bị xóa bỏ.
- **Cảnh báo rủi ro cao:** Nếu hệ thống phát hiện có trường dữ liệu bị xóa hoặc thay đổi kiểu (có nguy cơ gây mất dữ liệu hiện có), một thông báo cảnh báo màu đỏ sẽ xuất hiện, yêu cầu quản trị viên hết sức cẩn trọng.

Hơn thế nữa, để tự động hóa tối đa quy trình cập nhật, giao diện còn tích hợp tính năng tự động sinh mã nguồn (Auto-generate Code). Hệ thống sẽ chuyển đổi các kiểu dữ liệu từ cấu trúc của hệ thống nguồn sang định dạng chuẩn của Prisma ORM và tạo sẵn đoạn mã khai báo mô hình dữ liệu (Prisma Model). Quản trị viên chỉ cần nhấn nút sao chép, dán vào mã nguồn của trục tích hợp và chạy lệnh đồng bộ cơ sở dữ liệu mà không cần phải tự viết tay lại từ đầu.

Sau khi quá trình cập nhật cấu trúc cơ sở dữ liệu hoàn tất một cách an toàn, quản trị viên sẽ nhấn nút xác nhận trên giao diện. Lúc này, trạng thái của bảng sẽ trở lại trạng thái ổn định và các luồng đồng bộ dữ liệu sẽ tự động hoạt động trở lại bình thường.



Hình 11.5: Giao diện quản lý lược đồ dữ liệu

11.6 Tra cứu và khai thác dữ liệu theo phạm vi phân quyền

Bên cạnh các chức năng quản trị dành cho quản trị viên hệ thống, trục tích hợp còn cung cấp một giao diện chuyên biệt dành cho người dùng thông thường nhằm phục vụ nhu cầu tra cứu và khai thác dữ liệu đã được đồng bộ. Toàn bộ phạm vi dữ liệu hiển thị trên giao diện đều được kiểm soát chặt chẽ dựa trên cơ chế phân quyền đã được thiết lập trước đó.

Khi đăng nhập vào hệ thống, người dùng chỉ nhìn thấy danh sách những bảng dữ liệu thuộc phạm vi được phép truy cập. Các bảng nằm ngoài phạm vi phân quyền hoàn toàn không xuất hiện trên giao diện, từ đó giảm thiểu nguy cơ truy cập trái phép hoặc lộ lọt thông tin giữa các phòng ban.

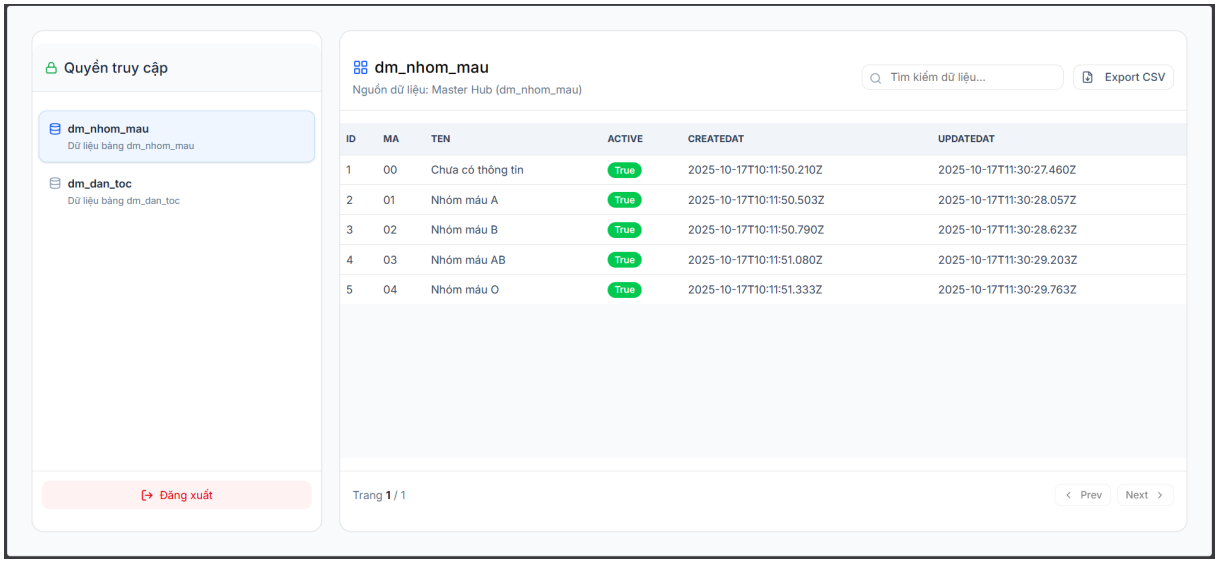
Giao diện đọc dữ liệu được thiết kế theo hướng đơn giản và tập trung vào khả năng khai thác dữ liệu nhanh chóng. Hệ thống hiển thị dữ liệu dưới dạng bảng với khả năng phân trang nhằm đảm bảo hiệu năng ngay cả khi số lượng bản ghi lớn. Người dùng có thể thực hiện tìm kiếm trực tiếp theo từ khóa để nhanh chóng lọc ra các dòng dữ liệu cần thiết mà không phải duyệt thủ công toàn bộ bảng dữ liệu.

Đối với những tài khoản chỉ có quyền đọc, mọi thao tác thay đổi dữ liệu như thêm mới, cập nhật hoặc xóa đều bị vô hiệu hóa hoàn toàn trên giao diện cũng như tại tầng kiểm tra quyền ở phía máy chủ. Điều này giúp đảm bảo dữ liệu trên trục tích hợp luôn

được bảo vệ trước các thao tác ngoài ý muốn.

Ngoài ra, hệ thống còn hỗ trợ phân biệt rõ ràng giữa người dùng chỉ được phép đọc dữ liệu và người dùng được phép ghi dữ liệu. Với các tài khoản có quyền ghi, giao diện sẽ hiển thị thêm các chức năng thao tác dữ liệu tương ứng với đúng phạm vi bảng được cấp quyền, trong khi các tài khoản thông thường vẫn duy trì giao diện tối giản chỉ phục vụ nhu cầu tra cứu.

Cơ chế này giúp trục tích hợp vừa đảm bảo khả năng chia sẻ dữ liệu linh hoạt giữa các hệ thống thành viên, vừa duy trì được tính an toàn và cô lập dữ liệu giữa các đơn vị sử dụng khác nhau.



Hình 11.6: Giao diện đối với người dùng không phải là quản trị viên

12 Tổng kết

12.1 Những gì đã đạt được

Đề tài đặt ra bài toán thực tiễn: dữ liệu của một trường đại học bị phân tán trên nhiều hệ thống nguồn riêng biệt, không có cơ chế tích hợp tập trung, gây khó khăn cho việc quản lý, tra cứu và bảo đảm tính nhất quán. Sau quá trình nghiên cứu và triển khai, nhóm đã xây dựng được một hệ thống **Data Integration Hub** hoàn chỉnh giải quyết bài toán đó theo các khía cạnh sau:

12.1.1 Kiến trúc và thiết kế hệ thống

Nhóm áp dụng mô hình **Hub-and-Spoke** kết hợp kiến trúc **API-led Connectivity** để tập trung toàn bộ luồng dữ liệu qua một điểm trung tâm duy nhất thay vì tích hợp điểm-điểm phân tán. Hệ thống được hiện thực hóa dưới dạng **NestJS modular monolith** — một lựa chọn thực tiễn phù hợp với quy mô dự án, giúp đơn giản hóa việc triển khai và vận hành so với kiến trúc microservices phân tán. Kong API Gateway đóng vai trò lớp bảo vệ phía trước, xử lý routing, CORS và xác thực trước khi request đến các module nội bộ.

12.1.2 Luồng ETL và chiến lược đồng bộ

Pipeline ETL hỗ trợ ba chiến lược đồng bộ linh hoạt theo đặc thù từng loại bảng:

- **Upsert** — chiến lược mặc định, phù hợp với bảng nghiệp vụ thay đổi thường xuyên.
- **Incremental** — chỉ đồng bộ các bản ghi có `updatedAt` sau lần sync gần nhất, tiết kiệm băng thông và thời gian.
- **Overwrite** — xóa và ghi lại toàn bộ, dùng cho bảng danh mục nhỏ cần đảm bảo destination = source tuyệt đối.

Cơ chế phân trang batch 5.000 records kết hợp với **retry có chọn lọc** (chỉ retry lỗi mạng, không retry lỗi nghiệp vụ) giúp pipeline vừa ổn định vừa không che giấu bug logic.

12.1.3 Hệ thống phòng thủ Data Collision

Nhóm thiết kế và kiểm thử một hệ thống phòng thủ hai lớp để xử lý dữ liệu không nhất quán từ các nguồn khác nhau:

- **Lớp 1 — DataIntegrationService:** Kiểm tra sớm (fail-fast) với Schema Status Guard, API Contract Validator và Page Mismatch Detector. Lỗi nghiêm trọng dừng pipeline ngay lập tức để không làm hỏng dữ liệu đích.
- **Lớp 2 — SyncEngineService:** Lọc field lạ, loại bỏ trùng lặp primary key, chuẩn hóa kiểu dữ liệu (Int, Boolean, DateTime). Các bản ghi vi phạm được chuyển vào **Dead Letter Log** thay vì abort toàn batch — đảm bảo partial success thay vì all-or-nothing.

Bộ 15 test case (TC-01 đến TC-15) đã kiểm thử đầy đủ các tình huống xung đột dữ liệu và xác nhận cơ chế hoạt động đúng như thiết kế.

12.1.4 Schema Registry và giám sát thay đổi schema

Schema Registry trên MongoDB theo dõi cấu trúc của từng bảng dữ liệu nguồn. Khi source API thay đổi schema, hệ thống tự động phát hiện thông qua so sánh `hashValue`, đánh dấu bảng về trạng thái `changed` (Schema Drift), lưu lịch sử vào `oldDetails` và dừng pipeline cho bảng đó chờ Admin xác nhận. Cơ chế này ngăn pipeline ghi sai dữ liệu khi schema chưa được cập nhật tương ứng ở tầng PostgreSQL.

12.1.5 Sao lưu và phục hồi

Hệ thống sao lưu hai tầng với **MinIO** và **AWS S3** được kích hoạt theo bốn trigger: tự động trước mỗi lần sync, theo lịch hàng tuần, khi schema thay đổi và thủ công. Quá trình restore hoàn toàn tự động qua API: tải file JSON từ MinIO → snapshot hiện tại → DELETE all → INSERT batch 1.000 records.

12.1.6 CI/CD và triển khai

Quy trình CI/CD trên **GitHub Actions** bao gồm hai job: Build & Push (đóng gói Docker image lên Docker Hub) và Deploy (SSH vào EC2, pull image mới, khởi động lại container). 41 unit test tự động (Jest) được tích hợp trong pipeline để phát hiện regression trước khi deploy. Hệ thống được triển khai lên **Amazon EC2 t3.medium** với cấu hình Docker Compose đầy đủ bao gồm backend, frontend, Kong Gateway, PostgreSQL, MongoDB, MinIO và Grafana.

12.1.7 Giao diện quản trị

Admin Dashboard (Next.js) cung cấp đầy đủ công cụ để vận hành hệ thống mà không cần truy cập trực tiếp vào database hay terminal:

- Kích hoạt và theo dõi pipeline đồng bộ theo thời gian thực (SSE log stream).
- Xem lịch sử đồng bộ phân cấp theo từng bảng.
- Quản lý backup, restore và dọn dẹp dữ liệu cũ.
- Phân quyền truy cập theo role và pattern regex trên từng bảng.
- Xem diff schema và phê duyệt thay đổi trực tiếp trên giao diện.

12.2 Hạn chế và hướng phát triển

Mặc dù hệ thống đã đáp ứng đầy đủ các yêu cầu đặt ra, vẫn còn một số hạn chế cần được khắc phục trong các phiên bản tiếp theo:

Bảng 12.1: Hạn chế hiện tại và hướng phát triển

Hạn chế	**Hướng khắc phục**
Pipeline ETL chạy tuần tự từng bảng, thời gian đồng bộ tỷ lệ tuyến tính với số bảng	Triển khai worker pool song song (Promise.allSettled với concurrency limit) để đồng bộ nhiều bảng cùng lúc
Export API nguồn dùng OFFSET pagination, gây chậm với bảng lớn	Chuyển sang Keyset pagination (cursor-based) ở phía source API để đảm bảo $O(1)$ mỗi trang
Cơ chế phát hiện schema drift phụ thuộc vào `hashValue` từ source API	Tích hợp engine tự phát hiện thay đổi bằng cách so sánh `details` nội bộ, không phụ thuộc nguồn ngoài
Restore dùng DELETE all → INSERT, có khoảng trống dữ liệu tạm thời	Áp dụng kỹ thuật Blue/Green restore: ghi vào bảng tạm, sau đó swap atomic
Chưa có cơ chế audit log cho hành động của Admin	Bổ sung audit trail ghi lại mọi thao tác của admin (ai, khi nào, thao tác gì) vào MongoDB
Unit test chưa bao phủ toàn bộ các module	Tăng coverage với integration test thực (không mock) cho các module Backup, SchemaRegistry

12.3 Kết luận

Dự án **Data Integration Hub** đã chứng minh rằng việc xây dựng một hệ thống tích hợp dữ liệu tập trung, an toàn và có khả năng vận hành độc lập là hoàn toàn khả thi với stack công nghệ hiện đại. Từ kiến trúc Hub-and-Spoke, cơ chế phòng thủ dữ liệu hai lớp, hệ thống backup thông minh đến giao diện quản trị trực quan — mỗi thành phần được thiết kế có chủ đích để giải quyết một vấn đề cụ thể trong bài toán tích hợp dữ liệu đại học.

Quan trọng hơn, hệ thống không chỉ hoạt động đúng trong điều kiện lý tưởng mà còn được kiểm thử với các tình huống dữ liệu xấu thực tế: API trả sai cấu trúc, dữ liệu trùng lặp, schema thay đổi đột ngột, database constraint vi phạm. Đây là yếu tố quyết định sự tin cậy của một hệ thống tích hợp trong môi trường sản xuất.

Nhóm hi vọng đề tài này đặt nền tảng cho một hướng phát triển tiếp theo: mở rộng hệ thống thành nền tảng tích hợp dữ liệu có thể tái sử dụng cho nhiều trường đại học, hướng tới mô hình **Data Mesh** nơi mỗi phòng ban tự quản lý domain dữ liệu của mình nhưng vẫn tuân theo một chuẩn tích hợp chung.

Phụ lục A: Đặc tả API (API Specification)

Hệ thống **University Integration Hub** tuân theo tiêu chuẩn thiết kế RESTful API và được tài liệu hóa dựa trên chuẩn **OpenAPI 3.0 (OAS 3.0)**. Định dạng dữ liệu trao đổi (Request/Response) chủ yếu là `application/json`.

Dưới đây là danh sách tổng hợp các API endpoint cốt lõi của hệ thống, được phân nhóm theo từng nghiệp vụ quản trị và xử lý dữ liệu.

Method	**Endpoint / Đường dẫn**	**Mô tả chức năng**
	1. Nhóm Xác thực (Auth)	
POST	`/api/v1/auth/login`	Đăng nhập hệ thống
POST	`/api/v1/auth/register`	Đăng ký tài khoản (Payload: username, email, password)
GET	`/api/v1/auth/me`	Lấy thông tin tài khoản đang đăng nhập
POST	`/api/v1/auth/refresh-token`	Cấp lại Access Token mới bằng Refresh Token

(Tiếp tục Phụ lục A: Đặc tả API)

Method	Endpoint / Đường dẫn	Mô tả chức năng
2. Nhóm Quản lý Người dùng (Users)		
GET	`/api/v1/users`	Lấy danh sách người dùng trong hệ thống
POST	`/api/v1/users`	Tạo tài khoản người dùng mới
GET	`/api/v1/users/{id}`	Lấy thông tin chi tiết người dùng theo ID
PUT	`/api/v1/users/{id}`	Cập nhật thông tin người dùng
PUT	`/api/v1/users/{id}/role`	Thay đổi vai trò (Role) của người dùng
GET	`/api/v1/users/permissions`	Lấy danh sách toàn bộ cấu hình quyền hạn
GET	`/api/v1/users/{id}/permissions`	Lấy danh sách quyền hạn của một người dùng
POST	`/api/v1/users/{id}/permissions`	Cấp/gán quyền hạn mới cho người dùng
3. Nhóm Cấu hình Nguồn Dữ liệu (Source Config)		
GET	`/api/v1/source-configs`	Lấy danh sách cấu hình các hệ thống nguồn
POST	`/api/v1/source-configs`	Tạo cấu hình kết nối hệ thống nguồn mới
GET	`/api/v1/source-configs/{sourceId}`	Lấy chi tiết một cấu hình nguồn theo ID
PUT	`/api/v1/source-configs/{sourceId}`	Cập nhật cấu hình hệ thống nguồn
DELETE	`/api/v1/source-configs/{sourceId}`	Xóa cấu hình hệ thống nguồn
PATCH	`/api/v1/source-configs/{sourceId}/toggle-active`	Bật/Tắt trạng thái hoạt động của nguồn
4. Nhóm Quản lý Cấu trúc Dữ liệu (Schema Registry)		

(Tiếp tục Phụ lục A: Đặc tả API)

Method	Endpoint / Đường dẫn	Mô tả chức năng
POST	`/api/v1/schema-registry/import`	Import/Đăng ký cấu trúc dữ liệu (Schema) mới
GET	`/api/v1/schema-registry`	Lấy danh sách các Schema đã đăng ký
PUT	`/api/v1/schema-registry/{tableName}`	Cập nhật cấu hình Schema của một bảng
POST	`/api/v1/schema-registry/run-detector`	Khởi chạy công cụ dò tìm Schema Drift
PUT	`/api/v1/schema-registry/{tableName}/resolve`	Đánh dấu đã xử lý cảnh báo Schema Drift
PUT	`/api/v1/schema-registry/{tableName}/reject`	Từ chối thay đổi Schema Drift của bảng
5. Nhóm Đồng bộ Dữ liệu (Data Integration & NguoiHoc)		
POST	`/api/v1/integration/run-full-sync`	Khởi chạy tiến trình đồng bộ toàn phần
POST	`/api/v1/integration/run-custom-sync`	Khởi chạy tiến trình đồng bộ tùy chỉnh
GET	`/api/v1/nguoi-hoc/trigger-sync`	Kích hoạt thủ công luồng đồng bộ Người Học
POST	`/api/v1/integration/scan-orphans`	Quét tìm dữ liệu rác/mồ côi (Orphaned records)
POST	`/api/v1/integration/purge-orphans`	Xóa bỏ các dữ liệu rác/mồ côi đã quét được
GET	`/api/v1/integration/stream-logs`	Xem log quá trình đồng bộ thời gian thực (SSE)
6. Nhóm Lập lịch Tự động (Job Scheduler)		
GET	`/api/v1/jobs`	Lấy danh sách các tiến trình đồng bộ tự động

(Tiếp tục Phụ lục A: Đặc tả API)

Method	Endpoint / Đường dẫn	Mô tả chức năng
POST	`/api/v1/jobs`	Tạo mới lịch chạy đồng bộ (Cronjob)
PUT	`/api/v1/jobs/{id}`	Cập nhật cấu hình tiến trình đồng bộ
PUT	`/api/v1/jobs/{id}/toggle`	Bật/Tắt tiến trình chạy tự động
POST	`/api/v1/jobs/{id}/trigger`	Kích hoạt chạy ngay một tiến trình
7. Nhóm Dữ liệu Danh mục (Master Data & Tables)		
GET	`/api/v1/tables/allowed`	Lấy danh sách các bảng được phép thao tác
GET	`/api/v1/api/master-data/allowed`	Lấy danh sách danh mục được phép truy cập
GET	`/api/v1/api/master-data/{table}`	Lấy toàn bộ dữ liệu của một bảng danh mục
POST	`/api/v1/api/master-data/{table}`	Thêm mới bản ghi vào bảng danh mục
GET	`/api/v1/api/master-data/{table}/{id}`	Lấy chi tiết một bản ghi danh mục
PATCH	`/api/v1/api/master-data/{table}/{id}`	Cập nhật một phần bản ghi danh mục
DELETE	`/api/v1/api/master-data/{table}/{id}`	Xóa bản ghi trong danh mục
8. Nhóm Sao lưu & Phục hồi (Backup)		
GET	`/api/v1/backup/list`	Lấy danh sách các bản sao lưu hiện có
POST	`/api/v1/backup/trigger`	Kích hoạt tiến trình tạo bản sao lưu ngay lập tức
POST	`/api/v1/backup/download`	Tải xuống một bản sao lưu
POST	`/api/v1/backup/restore`	Phục hồi hệ thống từ một bản sao lưu

(Tiếp tục Phụ lục A: Đặc tả API)

Method	Endpoint / Đường dẫn	Mô tả chức năng
DELETE	`/api/v1/backup/file`	Xóa một file sao lưu cụ thể
GET	`/api/v1/backup/retention`	Xem cấu hình vòng đời lưu trữ (Retention Policy)
PATCH	`/api/v1/backup/retention`	Cập nhật vòng đời lưu trữ bản sao lưu
POST	`/api/v1/backup/cleanup`	Chạy dọn dẹp các bản sao lưu đã cũ
POST	`/api/v1/backup/sync-s3`	Đồng bộ bản sao lưu hiện tại lên Amazon S3
POST	`/api/v1/backup/sync-s3/all`	Đồng bộ toàn bộ bản sao lưu lên Amazon S3
9. Nhóm Nhật ký Hệ thống (Event Log)		
GET	`/api/v1/event-logs`	Lấy danh sách nhật ký sự kiện (Audit Logs)

Lưu ý: Cấu trúc dữ liệu chi tiết của các Request Payload (như `CreateUserDto`, `LoginDto`, `UpdateSchemaRegistryDto`,...) và định dạng Response được đặc tả trực quan thông qua giao diện Swagger UI trên môi trường triển khai thực tế.

Phụ lục B: Hướng dẫn cài đặt và triển khai hệ thống

Phụ lục này cung cấp các hướng dẫn chi tiết để thiết lập môi trường, cấu hình và khởi chạy hệ thống **University Integration Hub** trên môi trường phát triển (Local Development) cũng như môi trường thực tế (Production).

B.1. Yêu cầu hệ thống (Prerequisites)

Để khởi chạy toàn bộ các thành phần của hệ thống (Backend, Frontend, Mock Server, Database), máy chủ cần được cài đặt sẵn các phần mềm sau:

- **Node.js:** Phiên bản v18.x hoặc cao hơn.
- **Trình quản lý gói:** `npm` (đi kèm Node.js) hoặc `yarn`.

- **Docker & Docker Compose:** Dành cho việc khởi chạy nhanh cơ sở dữ liệu và triển khai Production.
- **Cơ sở dữ liệu:** PostgreSQL 14+ (có thể thiết lập chạy qua Docker container).

B.2. Cấu trúc thư mục mã nguồn (Project Structure)

Hệ thống được tổ chức theo cấu trúc phân tách rõ ràng các service. Dưới đây là các thành phần chính yếu cấu thành nên dự án:

```
Data_Integration_Hub/
|-- backend/            # Mã nguồn Backend (TypeScript / NestJS)
|-- frontend/           # Mã nguồn giao diện quản trị (Dashboard)
|-- mock-server/        # Máy chủ giả lập nguồn dữ liệu ngoài (server.js)
|-- docker-compose.yml  # Cấu hình hạ tầng triển khai qua Docker
`-- README.md           # Tài liệu hướng dẫn chung của dự án
```

B.3. Cấu hình biến môi trường (.env)

Trước khi khởi chạy, cần tạo file `.env` tại thư mục gốc của `backend/` và cấu hình các thông số kết nối. Dưới đây là mẫu cấu hình tiêu biểu cần thiết:

```
# 1. Thông tin cấu hình Server
PORT=3000
NODE_ENV=development

# 2. Thông tin cấu hình Database (PostgreSQL)
DATABASE_URL="postgresql://postgres:password@localhost:5432/integration_hub"

# 3. Cấu hình bảo mật và phân quyền (JWT)
JWT_SECRET="your_super_secret_key_here"
JWT_EXPIRATION="24h"

# 4. Cấu hình kết nối hệ thống ngoài
MOCK_SERVER_URL="http://localhost:4000"
```

B.4. Khởi chạy trên môi trường phát triển (Local Development)

Các bước dưới đây hướng dẫn khởi chạy từng dịch vụ độc lập để thuận tiện cho việc theo dõi log và gỡ lỗi (debug) trong quá trình lập trình.

Bước 1: Khởi chạy Cơ sở dữ liệu

Sử dụng Docker Compose để dựng nhanh container cho PostgreSQL và Redis (nếu có):

```
$ docker-compose up -d db redis
```

Bước 2: Khởi chạy Integration Backend

Di chuyển vào thư mục backend, cài đặt các gói phụ thuộc và khởi chạy:

```
$ cd backend
$ npm install
$ npm run start:dev
```

Hệ thống backend sẽ bắt đầu lắng nghe các API Request tại cổng 3000.

Bước 3: Khởi chạy Mock Server (Giả lập hệ thống nguồn)

Để mô phỏng API của các trường đại học nhằm mục đích kiểm thử đồng bộ dữ liệu:

```
$ cd mock-server
$ npm install
$ node server.js
```

Mock Server sẽ hoạt động tại cổng 4000.

Bước 4: Khởi chạy Frontend Dashboard

```
$ cd frontend
$ npm install
$ npm run dev
```

Giao diện quản trị viên sẽ được phục vụ trên trình duyệt.

B.5. Hướng dẫn triển khai bằng Docker (Production)

Để triển khai toàn bộ hệ thống lên máy chủ thật một cách liền mạch, hệ thống đã được đóng gói sẵn với `docker-compose.yml`. Tại thư mục gốc của dự án (`Data_Integration_Hub/`), chỉ cần thực thi lệnh sau:

```
$ docker-compose up -d --build
```

Lệnh này sẽ tự động thực hiện chuỗi quy trình:

- Build (đóng gói) các Docker image cho Backend và Frontend.
- Khởi tạo các container Cơ sở dữ liệu và ánh xạ Volume để lưu trữ dữ liệu vĩnh viễn (persistent data).
- Thiết lập mạng lưới nội bộ (Docker Network) để các dịch vụ giao tiếp với nhau một cách bảo mật, không lộ port ra bên ngoài.
- Tự động khởi chạy lại ứng dụng (`restart: always`) nếu máy chủ bị khởi động lại hoặc hệ thống gặp sự cố văng lỗi (crash).

Tài liệu

[1] Andrea Cali, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Data integration under integrity constraints. *Information Systems*, 29(2):147–163, 2004.

[2] DAMA International. DAMA-DMBOK: Data management body of knowledge. Technics Publications, 2017.

[3] Zhamak Dehghani. *Data Mesh: Delivering Data-Driven Value at Scale*. O'Reilly Media, 2022.

[4] Docker, Inc. Docker documentation: Develop with docker. https://docs.docker.com, 2023.

[5] Weiguo Fan, Hongjun Lu, Stuart E. Madnick, and David Cheung. Discovering and reconciling value conflicts for numerical data integration. *Information Systems*, 26(8):635–656, 2001.

[6] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[7] William H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, 4th edition, 2005.

[8] Walaa S. Ismail, Mona M. Nasr, Torky I. Sultan, and Ayman E. Khedr. Semantic conflicts reconciliation as a viable solution for semantic heterogeneity problems. *International Journal of Computer Science and Information Security*, 8(5), 2010.

[9] Michael B. Jones, John Bradley, and Nat Sakimura. RFC 7519: Json web token (JWT). Internet Engineering Task Force (IETF), https://www.rfc-editor.org/rfc/rfc7519, 2015.

[10] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, 2nd edition, 2002.

[11] Kong Inc. Kong gateway documentation. https://docs.konghq.com, 2023.

[12] Maurizio Lenzerini. Data integration: A theoretical perspective. *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 233–246, 2002.

[13] Ee-Peng Lim and Roger H. L. Chiang. The integration of relationship instances from heterogeneous databases. *Journal of Database Management*, 7(4):3–16, 1996.

[14] Meta Platforms, Inc. Jest: Delightful javascript testing framework. https://jestjs.io/docs, 2023.

[15] MinIO, Inc. Minio high performance object storage documentation. https://min.io/docs, 2023.

[16] MuleSoft. Api-led connectivity: A blueprint for building scalable and agile enterprise applications. White Paper, MuleSoft Inc., 2018.

[17] NestJS Contributors. Nestjs: A progressive node.js framework — official documentation. https://docs.nestjs.com, 2023.

[18] Prisma Data, Inc. Prisma orm — next-generation node.js and typescript orm. https://www.prisma.io/docs, 2023.

[19] Thomas C. Redman. *Data Quality for the Information Age*. Artech House, 1996.

[20] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

[21] The PostgreSQL Global Development Group. Postgresql 17 documentation. https://www.postgresql.org/docs/17, 2023.