**BỘ CÔNG THƯƠNG**

**TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP TP.HCM KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO MÔN HỌC**

**KIẾN TRÚC THIẾT KẾ PHẦN MỀM**

**Đề tài: CHƯƠNG TRÌNH HỆ THỐNG QUẢN LÝ LỚP HỌC TÍN CHỈ**

Sinh viên thực hiện: Trần Văn Hùng – 20061461

Nguyễn Thị Hoài Thương –20001595

Trương Thị Tường Vi – 20000435

Lớp: DHKTPM16A

Hồ Chí Minh, ngày tháng năm 2024

**Mục Lục**

[I. Tổng quan về các kiến trúc 4](#_Toc167052454)

[1. Microservices architecture style 4](#_Toc167052455)

[2. Pipeline architecture style 5](#_Toc167052456)

[3. Layer architecture style 6](#_Toc167052457)

[4. Microkernel architecture style 9](#_Toc167052458)

[5. Service-Based architecture style 11](#_Toc167052459)

[6. Event-Driven architecture style 14](#_Toc167052460)

[7. Space-Based architecture style 15](#_Toc167052461)

[8. Service-Oriented architecture style 16](#_Toc167052462)

[II. Đặc tả 18](#_Toc167052463)

[III. Lý do chọn kiến trúc 18](#_Toc167052464)

[IV. Cơ sở lý thuyết 19](#_Toc167052465)

[1. Ngôn ngữ lập trình JavaScript 19](#_Toc167052466)

[2. Xây dựng microservice với Node.js 19](#_Toc167052467)

[3. Xậy dựng giao diện người dùng(UI) với React.js 19](#_Toc167052468)

[4. Xây dựng API Gateway với RESTful API 20](#_Toc167052469)

[V. Phân tích thiết kế kiến trúc 20](#_Toc167052470)

[VI. Phân tích thiết kế hệ thống 21](#_Toc167052471)

[1. Quản lý Học thuật (Academic Service): 21](#_Toc167052472)

[1.1. Class diagram 21](#_Toc167052473)

[1.2. Use case diagram 22](#_Toc167052474)

[1.3. Đặc tả use case xem danh sách môn học 22](#_Toc167052475)

[2. Truy xuất danh sách môn học theo chuyên ngành. 23](#_Toc167052476)

[3. Hiện thị danh sách môn học 23](#_Toc167052477)

[2. Đăng ký học phần (Registration Service): 24](#_Toc167052478)

[3. Quản lý Xét tốt nghiệp (Graduation Service): 24](#_Toc167052479)

[3.1. Use case diagram 24](#_Toc167052480)

[4. Quản lý Tài khoản (Authentication Service): 25](#_Toc167052481)

[5. Quản lý Thông báo (Notification Service): 25](#_Toc167052482)

[6. Quản lý Cựu sinh viên (Alumni Service): 28](#_Toc167052483)

[VII. Hiện Thực 30](#_Toc167052484)

[VIII. Kiểm Thử 30](#_Toc167052485)

[IX. Kết luận 30](#_Toc167052486)

[X. Tài liệu tham khảo 30](#_Toc167052487)

# Tổng quan về các kiến trúc

## Microservices architecture style

Kiến trúc Microservices là một loại kiến trúc phần mềm, trong đó ứng dụng được chia thành các dịch vụ nhỏ, độc lập và triển khai riêng biệt. Mỗi dịch vụ tập trung vào một chức năng cụ thể và giao tiếp với nhau thông qua các giao thức soft như HTTP/REST hoặc message queues.

**Nguồn gốc:** Thuật ngữ "microservices" được phổ biến bởi Martin Fowler và James Lewis trong một bài blog vào năm 2014. Nó được lấy cảm hứng từ Domain-Driven Design (DDD), đặc biệt là khái niệm về "Bounded Context", trong đó mỗi dịch vụ thể hiện một ngữ cảnh nghiệp vụ độc lập.

Đặc điểm chính:

Tính độc lập: Mỗi dịch vụ là một đơn vị triển khai riêng biệt, có thể được phát triển, triển khai, mở rộng và bảo trì độc lập với các dịch vụ khác.

Tính nhỏ gọn: Mỗi dịch vụ tập trung vào một chức năng cụ thể, giúp dễ dàng hiểu, phát triển và bảo trì.

Tính linh hoạt: Các dịch vụ có thể được viết bằng các ngôn ngữ lập trình và sử dụng các công nghệ khác nhau, phù hợp với nhu cầu của từng dịch vụ.

Tính khả dụng cao: Do tính độc lập, việc một dịch vụ gặp sự cố sẽ không ảnh hưởng đến toàn bộ hệ thống.

Khả năng mở rộng: Dễ dàng mở rộng quy mô của từng dịch vụ độc lập để đáp ứng nhu cầu.

Ưu điểm:

Dễ dàng phát triển và bảo trì: Do tính nhỏ gọn và độc lập của các dịch vụ.

Triển khai nhanh chóng: Việc triển khai các dịch vụ nhỏ gọn và độc lập sẽ nhanh chóng và hiệu quả hơn.

Khả năng mở rộng linh hoạt: Mỗi dịch vụ có thể được mở rộng độc lập để đáp ứng nhu cầu.

Khả năng phục hồi cao: Việc một dịch vụ gặp sự cố sẽ không ảnh hưởng đến toàn bộ hệ thống.

Hỗ trợ nhiều công nghệ: Cho phép sử dụng các công nghệ khác nhau cho các dịch vụ khác nhau.

Nhược điểm:

Độ phức tạp cao: Quản lý nhiều dịch vụ độc lập có thể phức tạp hơn so với quản lý một ứng dụng nguyên khối.

Khó khăn trong kiểm thử: Việc kiểm thử hệ thống microservices có thể phức tạp hơn do sự phụ thuộc giữa các dịch vụ.

Yêu cầu kỹ năng cao: Cần đội ngũ phát triển có kỹ năng cao để xây dựng và quản lý hệ thống microservices.

## Pipeline architecture style

Kiến trúc Pipeline (Pipes and Filters) là một phong cách kiến trúc phần mềm, trong đó dữ liệu được xử lý theo một chuỗi các thành phần (filters) được kết nối với nhau bằng các kênh truyền dữ liệu (pipes). Mỗi filter thực hiện một nhiệm vụ cụ thể và độc lập với các filter khác.

**Đặc điểm chính:**

* **Tính tuyến tính:** Dữ liệu được xử lý theo một chuỗi tuyến tính các filter.
* **Tính độc lập:** Mỗi filter hoạt động độc lập với các filter khác.
* **Tính tái sử dụng:** Các filter có thể được tái sử dụng trong các pipeline khác nhau.
* **Dễ dàng kiểm thử:** Do tính độc lập của các filter.
* **Các thành phần:**
* **Filters:** Các thành phần xử lý dữ liệu, nhận dữ liệu từ pipe đầu vào và đưa kết quả ra pipe đầu ra.
* **Pipes:** Các kênh truyền dữ liệu kết nối các filter.

**Ưu điểm:**

Dễ dàng hiểu và triển khai: Do tính tuyến tính và độc lập của các thành phần.

Khả năng mở rộng: Dễ dàng thêm hoặc bớt filter trong pipeline.

Tính linh hoạt: Có thể thay đổi thứ tự các filter trong pipeline.

Dễ dàng kiểm thử: Có thể kiểm thử từng filter độc lập.

**Nhược điểm:**

Không phù hợp với các ứng dụng phức tạp: Do tính tuyến tính của pipeline.

Khó khăn trong xử lý lỗi: Việc xử lý lỗi trong pipeline có thể phức tạp.

## Layer architecture style

Kiến trúc theo tầng là một mô hình thiết kế phần mềm trong đó hệ thống được chia thành các tầng (layers), mỗi tầng đảm nhiệm một vai trò cụ thể và chỉ tương tác với các tầng liền kề. Mỗi tầng có trách nhiệm riêng biệt, giúp tách biệt các khía cạnh khác nhau của ứng dụng, từ giao diện người dùng đến lưu trữ dữ liệu.

**Các tầng trong kiến trúc theo tầng**

* **Presentation Layer (Tầng trình bày):**

Chức năng: Hiển thị dữ liệu cho người dùng và nhận các tương tác từ người dùng.

Vai trò: Tạo giao diện người dùng (UI), xử lý các sự kiện từ người dùng và hiển thị thông tin.

Ví dụ: Các trang web HTML/CSS, ứng dụng di động, giao diện đồ họa (GUI) của ứng dụng desktop.

* **Application Layer (Tầng ứng dụng):**

Chức năng: Điều phối các hoạt động giữa các tầng và xử lý logic ứng dụng không liên quan đến nghiệp vụ.

Vai trò: Đóng vai trò cầu nối giữa tầng trình bày và tầng logic nghiệp vụ, quản lý phiên làm việc của người dùng.

Ví dụ: Các bộ điều khiển trong mô hình MVC (Model-View-Controller).

* **Business Logic Layer (Tầng logic nghiệp vụ):**

Chức năng: Chứa các quy tắc nghiệp vụ và logic xử lý chính của ứng dụng.

Vai trò: Xử lý các hoạt động nghiệp vụ, áp dụng các quy tắc và logic của nghiệp vụ cụ thể.

Ví dụ: Các dịch vụ nghiệp vụ (business services), các lớp xử lý nghiệp vụ trong ứng dụng.

* **Data Access Layer (Tầng truy cập dữ liệu):**

Chức năng: Truy xuất và lưu trữ dữ liệu từ các nguồn dữ liệu như cơ sở dữ liệu, hệ thống tập tin.

Vai trò: Cung cấp các phương thức để thao tác với dữ liệu, đảm bảo rằng logic nghiệp vụ không bị ràng buộc vào chi tiết của lưu trữ dữ liệu.

Ví dụ: ORM (Object-Relational Mapping) frameworks, DAO (Data Access Objects).

* **Database Layer (Tầng cơ sở dữ liệu):**

Chức năng: Lưu trữ và quản lý dữ liệu thực tế.

Vai trò: Chứa các hệ quản trị cơ sở dữ liệu (DBMS), đảm bảo tính toàn vẹn và bảo mật của dữ liệu.

Ví dụ: MySQL, PostgreSQL, MongoDB.

**Ưu điểm của kiến trúc theo tầng**

**Tính mô-đun hóa cao:**

Lợi ích: Giúp dễ dàng bảo trì và nâng cấp từng phần của hệ thống mà không ảnh hưởng đến các phần khác.

Ví dụ: Khi cần nâng cấp giao diện người dùng, chỉ cần thay đổi tầng trình bày mà không cần thay đổi các tầng khác.

**Tái sử dụng mã nguồn:**

Lợi ích: Các tầng dưới có thể được sử dụng lại trong nhiều ứng dụng khác nhau, giảm công sức phát triển.

Ví dụ: Một tầng truy cập dữ liệu có thể được dùng lại trong nhiều ứng dụng có cơ sở dữ liệu tương tự.

**Dễ bảo trì và kiểm thử:**

Lợi ích: Việc phân chia rõ ràng các tầng giúp dễ dàng kiểm thử từng phần riêng lẻ và xác định lỗi nhanh chóng.

Ví dụ: Các công cụ kiểm thử tự động có thể được áp dụng riêng cho từng tầng, tăng hiệu quả kiểm thử.

**Tính linh hoạt:**

Lợi ích: Dễ dàng thay đổi hoặc nâng cấp một tầng mà không cần thay đổi toàn bộ hệ thống.

Ví dụ: Khi có yêu cầu thay đổi trong quy tắc nghiệp vụ, chỉ cần thay đổi tầng logic nghiệp vụ mà không cần ảnh hưởng đến tầng giao diện người dùng.

**Quản lý phức tạp dễ dàng:**

Lợi ích: Giúp quản lý và phát triển các hệ thống phức tạp bằng cách chia nhỏ các chức năng thành các tầng rõ ràng.

Ví dụ: Một dự án lớn có thể được chia thành nhiều nhóm, mỗi nhóm phụ trách một tầng cụ thể.

**Nhược điểm của kiến trúc theo tầng**

**Hiệu suất:**

Hạn chế: Việc giao tiếp qua lại giữa các tầng có thể tạo ra overhead, làm giảm hiệu suất tổng thể của hệ thống.

Ví dụ: Mỗi lần dữ liệu phải chuyển từ tầng này sang tầng khác có thể gây ra độ trễ.

**Độ phức tạp:**

Hạn chế: Khi có nhiều tầng, việc thiết kế và duy trì giao tiếp giữa các tầng có thể trở nên phức tạp.

Ví dụ: Cần phải thiết kế rõ ràng các giao diện (interfaces) giữa các tầng để đảm bảo tính tương thích và hiệu quả.

**Thiếu linh hoạt trong một số trường hợp:**

Hạn chế: Kiến trúc theo tầng có thể cứng nhắc và không phù hợp với các ứng dụng yêu cầu hiệu suất cao hoặc thời gian thực.

Ví dụ: Các ứng dụng thời gian thực như trò chơi hoặc hệ thống điều khiển cần phản hồi nhanh có thể không phù hợp với kiến trúc này.

**Khó khăn trong việc điều chỉnh:**

Hạn chế: Khi một tầng thay đổi, có thể cần điều chỉnh các tầng liên quan để đảm bảo tính tương thích.

Ví dụ: Nếu có thay đổi trong tầng cơ sở dữ liệu, các tầng truy cập dữ liệu và logic nghiệp vụ cũng cần được cập nhật tương ứng.

**Kết luận**

Kiến trúc theo tầng là một mô hình thiết kế phần mềm phổ biến và mạnh mẽ, mang lại nhiều lợi ích về mô-đun hóa, bảo trì, tái sử dụng mã nguồn và quản lý phức tạp. Tuy nhiên, nó cũng đi kèm với những thách thức về hiệu suất, độ phức tạp và tính linh hoạt. Việc lựa chọn áp dụng kiến trúc theo tầng cần dựa trên nhu cầu cụ thể của dự án và khả năng cân nhắc giữa các ưu và nhược điểm.

## Microkernel architecture style

Kiến trúc Microkernel (Microkernel Architecture) là một kiểu thiết kế hệ điều hành trong đó chức năng cốt lõi của hệ điều hành được giữ ở mức tối thiểu trong nhân (kernel). Các chức năng và dịch vụ bổ sung được thực thi dưới dạng các module riêng biệt trong không gian người dùng. Nhân microkernel chỉ chịu trách nhiệm cho các chức năng cơ bản như giao tiếp giữa các tiến trình (IPC), quản lý bộ nhớ cơ bản và xử lý ngắt.

**Các đặc điểm chính của kiến trúc Microkernel**

**Nhân nhỏ gọn:** Chỉ chứa các thành phần tối thiểu cần thiết cho việc quản lý tài nguyên cơ bản và giao tiếp giữa các tiến trình.

**Các dịch vụ ngoài nhân:** Các dịch vụ như quản lý hệ thống tập tin, giao tiếp mạng và giao diện người dùng được thực thi trong không gian người dùng dưới dạng các module riêng biệt.

**Giao tiếp qua cơ chế nhắn tin:** Các module dịch vụ và nhân giao tiếp với nhau thông qua các thông điệp (messages), tăng tính mô-đun và giảm sự phụ thuộc.

**Ưu điểm của kiến trúc Microkernel**

**Tính bảo mật cao:**

Lợi ích: Các dịch vụ chạy trong không gian người dùng, giảm nguy cơ ảnh hưởng đến toàn hệ thống khi một dịch vụ bị tấn công hoặc lỗi.

Ví dụ: Nếu dịch vụ quản lý hệ thống tập tin bị lỗi, nó sẽ không làm sụp đổ toàn bộ hệ thống vì không ảnh hưởng trực tiếp đến nhân.

**Tính mô-đun và dễ bảo trì:**

Lợi ích: Các dịch vụ có thể được phát triển, kiểm thử và nâng cấp độc lập với nhân, giúp việc bảo trì và nâng cấp hệ thống dễ dàng hơn.

Ví dụ: Nâng cấp hoặc thay thế dịch vụ mạng mà không cần thay đổi hoặc khởi động lại nhân hệ điều hành.

**Khả năng mở rộng:**

Lợi ích: Dễ dàng thêm các dịch vụ mới mà không cần thay đổi cấu trúc của nhân.

Ví dụ: Thêm dịch vụ mới như hỗ trợ Bluetooth bằng cách bổ sung một module riêng biệt mà không ảnh hưởng đến các phần khác của hệ thống.

**Tính ổn định cao:**

Lợi ích: Nhân nhỏ gọn và ít chức năng dẫn đến ít khả năng xảy ra lỗi nghiêm trọng trong nhân, tăng tính ổn định của hệ điều hành.

Ví dụ: Lỗi trong dịch vụ quản lý bộ nhớ người dùng sẽ không làm sụp đổ hệ thống vì nó không chạy trong không gian nhân.

**Nhược điểm của kiến trúc Microkernel**

**Hiệu suất thấp hơn:**

Hạn chế: Việc giao tiếp giữa các dịch vụ qua cơ chế nhắn tin có thể tạo ra overhead, làm giảm hiệu suất tổng thể so với các hệ điều hành nhân đơn khối (monolithic kernel).

Ví dụ: Các tác vụ yêu cầu giao tiếp liên tục giữa các dịch vụ như xử lý tệp và giao tiếp mạng có thể bị chậm hơn do thời gian cần thiết cho việc nhắn tin qua lại.

**Độ phức tạp trong thiết kế:**

Hạn chế: Thiết kế và triển khai một hệ điều hành microkernel đòi hỏi sự phân chia rõ ràng giữa các chức năng, tạo ra thách thức trong việc lập trình và kiểm thử.

Ví dụ: Cần thiết kế và duy trì các giao diện (interfaces) chuẩn hóa giữa nhân và các dịch vụ người dùng để đảm bảo tính tương thích và hiệu quả.

**Tăng overhead cho các dịch vụ hệ thống:**

Hạn chế: Do các dịch vụ chạy trong không gian người dùng, mỗi khi cần truy cập tài nguyên hệ thống, phải thực hiện nhiều lượt chuyển đổi ngữ cảnh giữa không gian người dùng và không gian nhân.

Ví dụ: Các thao tác như truy xuất đĩa hoặc xử lý ngắt có thể tốn nhiều thời gian hơn do cần chuyển đổi ngữ cảnh liên tục giữa các dịch vụ và nhân.

**Ví dụ thực tế**

Minix: Một hệ điều hành giáo dục sử dụng kiến trúc microkernel, nổi tiếng với tính đơn giản và bảo mật cao.

QNX: Một hệ điều hành thương mại thời gian thực (RTOS) sử dụng microkernel, được sử dụng rộng rãi trong các hệ thống nhúng và công nghiệp ô tô.

Mach: Nền tảng microkernel được sử dụng trong các hệ điều hành như NeXTSTEP và ảnh hưởng đến thiết kế của macOS.

**Kết luận**

Kiến trúc Microkernel mang lại nhiều lợi ích về bảo mật, tính mô-đun, dễ bảo trì và ổn định cao. Tuy nhiên, nó cũng đi kèm với những thách thức về hiệu suất và độ phức tạp trong thiết kế. Việc lựa chọn áp dụng kiến trúc Microkernel phụ thuộc vào nhu cầu cụ thể của hệ thống và khả năng cân nhắc giữa các ưu và nhược điểm. Hệ điều hành Microkernel là một lựa chọn lý tưởng cho các hệ thống đòi hỏi tính bảo mật cao và khả năng mở rộng linh hoạt, nhưng cần cân nhắc kỹ lưỡng trong các hệ thống yêu cầu hiệu suất cao.

## Service-Based architecture style

Kiến trúc dựa trên dịch vụ (Service-Based Architecture - SBA) là một mô hình thiết kế phần mềm trong đó các ứng dụng được chia thành các dịch vụ nhỏ, mỗi dịch vụ đảm nhiệm một chức năng cụ thể và giao tiếp với nhau qua các giao thức định sẵn. Các dịch vụ này có thể chạy trên cùng một máy chủ hoặc phân tán trên nhiều máy chủ khác nhau, và thường sử dụng các giao thức như HTTP/HTTPS để trao đổi dữ liệu. SBA thường được sử dụng trong các hệ thống phân tán và quy mô lớn để tận dụng tính linh hoạt và khả năng mở rộng.

**Đặc điểm chính của kiến trúc dựa trên dịch vụ**

**Phân tách chức năng:** Ứng dụng được chia thành nhiều dịch vụ nhỏ, mỗi dịch vụ đảm nhiệm một chức năng cụ thể như xác thực người dùng, quản lý đơn hàng, xử lý thanh toán.

**Giao tiếp qua mạng:** Các dịch vụ giao tiếp với nhau qua mạng bằng cách sử dụng các giao thức như HTTP/HTTPS, gRPC hoặc các hệ thống nhắn tin như Kafka, RabbitMQ.

**Độc lập và có thể triển khai riêng:** Mỗi dịch vụ có thể được phát triển, triển khai và mở rộng một cách độc lập, giúp tăng khả năng linh hoạt và đáp ứng nhanh với các thay đổi.

**Sử dụng các API:** Các dịch vụ cung cấp các API (giao diện lập trình ứng dụng) để cho phép các dịch vụ khác hoặc các ứng dụng người dùng tương tác với chúng.

**Ưu điểm của kiến trúc dựa trên dịch vụ**

**Khả năng mở rộng:**

Lợi ích: Dễ dàng mở rộng từng dịch vụ một cách độc lập để đáp ứng nhu cầu tăng trưởng mà không cần thay đổi toàn bộ hệ thống.

Ví dụ: Khi lượng truy cập tăng, có thể triển khai thêm nhiều instance của một dịch vụ cụ thể như dịch vụ xác thực người dùng mà không ảnh hưởng đến các dịch vụ khác.

**Tính mô-đun và tái sử dụng:**

Lợi ích: Các dịch vụ có thể được tái sử dụng trong nhiều ứng dụng khác nhau, giảm công sức phát triển và bảo trì.

Ví dụ: Dịch vụ xử lý thanh toán có thể được sử dụng bởi nhiều ứng dụng khác nhau trong cùng một tổ chức.

**Tính linh hoạt và dễ bảo trì:**

Lợi ích: Các dịch vụ có thể được phát triển, triển khai và nâng cấp một cách độc lập, giảm thiểu rủi ro và thời gian dừng hệ thống.

Ví dụ: Có thể nâng cấp dịch vụ quản lý người dùng mà không cần ngừng các dịch vụ khác như quản lý đơn hàng hay xử lý thanh toán.

**Khả năng phát triển phân tán:**

Lợi ích: Cho phép các nhóm phát triển khác nhau làm việc đồng thời trên các dịch vụ khác nhau, tăng hiệu quả phát triển.

Ví dụ: Một nhóm có thể làm việc trên dịch vụ giao tiếp với khách hàng trong khi nhóm khác làm việc trên dịch vụ xử lý đơn hàng.

**Khả năng chịu lỗi:**

Lợi ích: Khi một dịch vụ gặp sự cố, các dịch vụ khác vẫn có thể tiếp tục hoạt động, giúp hệ thống chịu lỗi tốt hơn.

Ví dụ: Nếu dịch vụ xử lý thanh toán gặp sự cố, dịch vụ quản lý sản phẩm và quản lý người dùng vẫn hoạt động bình thường.

**Nhược điểm của kiến trúc dựa trên dịch vụ**

**Độ phức tạp trong quản lý:**

Hạn chế: Việc quản lý nhiều dịch vụ phân tán có thể phức tạp hơn, yêu cầu các công cụ và kỹ thuật quản lý phù hợp.

Ví dụ: Cần sử dụng các hệ thống giám sát và quản lý dịch vụ như Kubernetes để quản lý việc triển khai và vận hành các dịch vụ.

**Hiệu suất giao tiếp:**

Hạn chế: Giao tiếp qua mạng giữa các dịch vụ có thể gây ra độ trễ và giảm hiệu suất so với các hệ thống nguyên khối (monolithic).

Ví dụ: Các dịch vụ phải thực hiện nhiều lượt gọi API qua mạng có thể gặp phải vấn đề về độ trễ và băng thông.

**Khả năng gỡ lỗi và theo dõi:**

Hạn chế: Khó khăn hơn trong việc gỡ lỗi và theo dõi các vấn đề trong hệ thống phân tán với nhiều dịch vụ.

Ví dụ: Khi gặp lỗi, cần theo dõi log và thông tin từ nhiều dịch vụ khác nhau để xác định nguyên nhân gốc rễ.

**Đảm bảo nhất quán dữ liệu:**

Hạn chế: Khó đảm bảo tính nhất quán dữ liệu khi các dịch vụ phân tán và cập nhật dữ liệu đồng thời.

Ví dụ: Khi nhiều dịch vụ cùng cập nhật một nguồn dữ liệu, có thể gặp phải các vấn đề về tính nhất quán và xung đột dữ liệu.

**Chi phí phát triển và vận hành:**

Hạn chế: Yêu cầu đầu tư vào các công cụ và kỹ thuật quản lý dịch vụ, cũng như kỹ năng của đội ngũ phát triển và vận hành.

Ví dụ: Cần có các công cụ chuyên dụng cho việc theo dõi, triển khai, và quản lý các dịch vụ phân tán, như Docker, Kubernetes, và các giải pháp CI/CD.

**Ví dụ thực tế**

Amazon: Sử dụng kiến trúc dịch vụ để xử lý các chức năng khác nhau như xử lý đơn hàng, thanh toán và quản lý kho hàng, cho phép mở rộng và phát triển các dịch vụ một cách độc lập.

Netflix: Áp dụng kiến trúc dịch vụ để quản lý việc phát trực tuyến, hệ thống đề xuất, và các dịch vụ liên quan đến khách hàng, giúp duy trì tính sẵn sàng cao và khả năng chịu lỗi tốt.

**Kết luận**

Kiến trúc dựa trên dịch vụ mang lại nhiều lợi ích về tính mô-đun, khả năng mở rộng, và tính linh hoạt. Tuy nhiên, nó cũng đi kèm với những thách thức về quản lý độ phức tạp, hiệu suất, và đảm bảo nhất quán dữ liệu. Việc lựa chọn áp dụng kiến trúc này phụ thuộc vào nhu cầu cụ thể của dự án và khả năng cân nhắc giữa các ưu và nhược điểm. SBA là lựa chọn lý tưởng cho các hệ thống yêu cầu khả năng mở rộng linh hoạt và phát triển phân tán, nhưng cần có sự chuẩn bị kỹ lưỡng về mặt quản lý và công nghệ.

## Event-Driven architecture style

Kiến trúc Event-Driven (EDA) sử dụng các sự kiện để kích hoạt và giao tiếp giữa các dịch vụ được tách rời. Một sự kiện là một thay đổi trạng thái hoặc cập nhật, như việc đặt một mặt hàng vào giỏ hàng trên trang web thương mại điện tử. EDA bao gồm các thành phần xử lý sự kiện không đồng bộ, nhận và xử lý sự kiện.

**Ưu điểm**

Tính linh hoạt và mở rộng: EDA cho phép các dịch vụ hoạt động độc lập và không đồng bộ, giúp hệ thống có thể mở rộng dễ dàng bằng cách thêm các thành phần mới mà không ảnh hưởng đến toàn bộ hệ thống.

Tính đáp ứng: Các dịch vụ trong EDA có thể phản ứng ngay lập tức với các sự kiện, cải thiện khả năng đáp ứng của hệ thống đối với các thay đổi và yêu cầu mới.

Khả năng chịu lỗi: Bằng cách phân tách các dịch vụ, một lỗi trong một dịch vụ sẽ ít có khả năng ảnh hưởng đến toàn bộ hệ thống, giúp tăng khả năng chịu lỗi.

Tính mô-đun: EDA thúc đẩy thiết kế mô-đun, nơi các thành phần có thể được phát triển, kiểm tra và triển khai độc lập, giúp dễ dàng bảo trì và nâng cấp.

**Nhược điểm**

Phức tạp trong quản lý: Quản lý và giám sát một hệ thống EDA có thể phức tạp hơn do số lượng lớn các thành phần và luồng dữ liệu không đồng bộ.

Độ trễ không đồng bộ: Mặc dù EDA hỗ trợ xử lý không đồng bộ, nhưng đôi khi việc truyền và xử lý sự kiện có thể gây ra độ trễ, ảnh hưởng đến hiệu suất hệ thống.

Khó khăn trong debug và kiểm tra: Do tính chất không đồng bộ và phân tán của EDA, việc tìm lỗi và kiểm tra hệ thống có thể trở nên khó khăn hơn.

Yêu cầu về hạ tầng: EDA thường yêu cầu hạ tầng phức tạp hơn, bao gồm các dịch vụ tin nhắn và quản lý sự kiện, điều này có thể tăng chi phí và yêu cầu kỹ thuật.

## Space-Based architecture style

Kiến trúc Space-Based (SBA) được thiết kế để xử lý các vấn đề liên quan đến khả năng mở rộng và độ tin cậy cao trong các hệ thống có tải công việc không đều và nhu cầu xử lý đồng thời cao. SBA loại bỏ các điểm nút cổ chai và tối ưu hóa hiệu suất thông qua việc phân tán và song song hóa.

**Ưu điểm**

Khả năng mở rộng cao: SBA cho phép mở rộng hệ thống dễ dàng bằng cách thêm hoặc bớt các nút mà không cần phải tạm dừng hoặc ảnh hưởng đến hoạt động của hệ thống.

Tính sẵn sàng cao: Bằng cách phân tán dữ liệu và xử lý trên nhiều nút, SBA giúp hệ thống duy trì hoạt động ngay cả khi một hoặc một vài nút gặp sự cố.

Hiệu suất cao: Việc xử lý song song và không có điểm nút cổ chai giúp tối ưu hóa hiệu suất tổng thể của hệ thống, đặc biệt là trong các tình huống tải cao hoặc biến động.

Tính linh hoạt: SBA hỗ trợ việc triển khai và cấu hình linh hoạt, cho phép hệ thống điều chỉnh dễ dàng theo yêu cầu kinh doanh và kỹ thuật.

**Nhược điểm**

Độ phức tạp trong phát triển và bảo trì: Việc thiết kế, phát triển và bảo trì một hệ thống SBA có thể phức tạp hơn do sự phân tán và tương tác giữa các thành phần.

Độ trễ đồng bộ: Dữ liệu phân tán có thể gây ra độ trễ khi cần đồng bộ hóa dữ liệu giữa các nút, ảnh hưởng đến hiệu suất hệ thống trong một số tình huống.

Chi phí hạ tầng: SBA yêu cầu một hạ tầng phức tạp và mạnh mẽ để hỗ trợ việc phân tán và song song hóa, điều này có thể làm tăng chi phí triển khai và vận hành.

Khó khăn trong debug và kiểm tra: Việc xác định và khắc phục lỗi trong một hệ thống phân tán có thể trở nên khó khăn hơn, đòi hỏi các công cụ và kỹ thuật phức tạp hơn.

**Kết luận**

Space-Based Architecture là một lựa chọn mạnh mẽ cho các hệ thống cần xử lý đồng thời cao và khả năng mở rộng linh hoạt. Tuy nhiên, nó đòi hỏi sự đầu tư đáng kể vào hạ tầng và kỹ thuật, cũng như sự phức tạp trong quản lý và bảo trì.

## Service-Oriented architecture style

Service-Oriented Architecture (SOA) là một hình thức kiến trúc phần mềm đặc trưng bởi việc kết hợp linh hoạt các dịch vụ mà chúng tạo thành cùng tương tác với nhau thông qua giao tiếp chuẩn. Mỗi dịch vụ trong SOA thực hiện một nhiệm vụ hay quy trình cụ thể và độc lập với cấu trúc hệ thống chung.

SOA không chỉ tập trung vào việc phát triển ứng dụng mà còn chú trọng đến quy trình nghiệp vụ. Nó giúp che giấu sự phức tạp kỹ thuật bằng cách sử dụng giao tiếp chuẩn của mình. Nói một cách đơn giản, SOA là một kiến trúc phần mềm tổ chức các thành phần thành những dịch vụ, mỗi dịch vụ thực hiện một nhiệm vụ cụ thể.

SOA kết nối các thành phần thông qua [cổng giao tiếp](https://fptshop.com.vn/tin-tuc/for-gamers/cac-cong-ket-noi-tren-mainboard-139884) và chúng có thể tương tác lẫn nhau mà không cần quan tâm đến nền tảng công nghệ. Điều này giúp hệ thống mở rộng và tích hợp dễ dàng hơn.

**Ưu nhược**

Tính linh hoạt và dễ mở rộng: SOA tách biệt các chức năng khác nhau thành các dịch vụ độc lập. Điều này làm cho việc thay đổi và mở rộng hệ thống trở nên dễ dàng hơn, vì bạn có thể chỉ cần thay đổi hoặc thêm mới các dịch vụ mà không cần phải ảnh hưởng đến toàn bộ hệ thống.

Tính tái sử dụng cao: Các dịch vụ trong SOA có thể được sử dụng lại trong nhiều ứng dụng khác nhau, giúp tối ưu hóa việc phát triển và giảm thiểu việc viết lại mã.

Tính phân tán: SOA cho phép các dịch vụ hoạt động độc lập và có thể được triển khai trên các máy chủ khác nhau, thậm chí là trên nền tảng đám mây. Điều này giúp tăng cường khả năng chịu lỗi và mở rộng của hệ thống.

Tính tái sử dụng cơ sở hạ tầng: Các dịch vụ SOA có thể được triển khai trên cơ sở hạ tầng hiện có mà không cần phải thay đổi hoặc cải thiện nhiều.

Tính đa dạng công nghệ: SOA không ràng buộc với bất kỳ ngôn ngữ lập trình hoặc công nghệ cụ thể nào. Điều này cho phép bạn sử dụng các công nghệ phù hợp nhất cho nhu cầu cụ thể của bạn.

Tính tương thích và tính tái sử dụng tài nguyên: SOA khuyến khích việc sử dụng các tiêu chuẩn mở, giúp tăng tính tương thích và khả năng tái sử dụng tài nguyên.

Quản lý dịch vụ hiệu quả: SOA cung cấp các công cụ quản lý dịch vụ để theo dõi, đo lường và quản lý hiệu suất của các dịch vụ, từ đó tối ưu hóa hoạt động của hệ thống.

**Nhược điểm**

Độ phức tạp: SOA có thể tạo ra một môi trường phức tạp với nhiều dịch vụ, giao tiếp mạnh mẽ và các lớp trung gian. Việc quản lý và triển khai một hệ thống SOA có thể đòi hỏi sự kiểm soát kỹ lưỡng để tránh việc mất quản lý và rối loạn.

Chi phí triển khai và quản lý: Xây dựng và triển khai một hệ thống SOA có thể đòi hỏi đầu tư lớn về thời gian, công sức và tài nguyên. Ngoài ra, việc quản lý một môi trường SOA cũng đòi hỏi chi phí quản lý cao, bao gồm việc đảm bảo hiệu suất, bảo mật và khả năng mở rộng của các dịch vụ.

Vấn đề tương tác và hiệu suất: Việc giao tiếp giữa các dịch vụ trong một hệ thống SOA có thể tạo ra những vấn đề về hiệu suất nếu không được thiết kế và triển khai đúng cách. Các cuộc gọi dịch vụ từ xa có thể tăng thời gian đáp ứng và tăng tải cho hệ thống.

Khó khăn trong việc theo dõi và gỡ lỗi: Với nhiều dịch vụ phân tán, việc theo dõi và gỡ lỗi các vấn đề trong hệ thống có thể trở nên phức tạp hơn. Việc định vị và sửa lỗi các lỗi liên quan đến giao tiếp giữa các dịch vụ có thể đòi hỏi kỹ năng và công cụ phức tạp.

Khả năng bảo mật và kiểm soát truy cập: Việc quản lý quyền truy cập và bảo mật trong một môi trường SOA phân tán có thể là một thách thức. Việc đảm bảo tính bảo mật của dịch vụ và dữ liệu khi chúng đi qua mạng cũng là một vấn đề quan trọng.

# Đặc tả

**Mục tiêu:** Quản lý hệ thống đăng ký học phần của sinh viên theo cơ chế tín chỉ, đáp ứng đầy đủ nhu cầu của sinh viên và nhà trường.

**Mô tả:**

Trường đại học cần quản lý hệ thống đăng ký học phần của sinh viên theo cơ chế tín chỉ. Sinh viên được biên chế ở các khoa theo ngành học đăng ký. Mỗi học kỳ, nhà trường chọn một số môn học của từng ngành học cho sinh viên đăng ký học. Sinh viên đăng ký theo học vào một lớp nhất định (lớp tín chỉ) với số sinh viên tối đa được qui định cho từng lớp. Môn học được mở có thể là môn học tự chọn. Mỗi môn học có một hoặc nhiều môn học tiên quyết. Sinh viên không được đăng ký môn học mà có môn tiên quyết chưa học. Mỗi học kỳ sinh viên được đăng ký tối đa 30 tín chỉ. Nếu quá số này, hệ thống sẽ không cho đăng ký. Để phòng ngừa trường hợp sinh viên đăng ký xong rồi hủy bỏ, nhà trường yêu cầu sinh viên phải xác nhận trước khi đăng ký. Vào ngày mở đăng ký, sinh viên sẽ đăng nhập vào hệ thống và sẽ nhìn thấy danh sách các môn học mà mình có khả năng đăng ký. Sinh viên chọn các môn học và tiến hành đăng ký. Trường hợp các lớp đã đầy, sinh viên sẽ được đưa vào một danh sách dự bị để nhà trường cân nhắc có mở thêm lớp hay không. Nếu không mở thêm lớp, sinh viên sẽ bị hủy đăng ký môn đó. Sau khi đăng ký thành công, một email thông báo sẽ được gửi cho sinh viên xác nhận việc đăng ký và nhận quyết định đóng học phí. Hệ thống cho phép sinh viên xem thông tin học tập của mình (số tín chỉ đã đạt, số môn đã học, điểm môn học, điểm trung bình tích lũy…), thời khóa biểu theo tuần và các tiện ích khác. Sau khi số tín chỉ đã đạt theo từng ngành, sinh viên có quyền đăng ký xét tốt nghiệp. Nếu mọi tiêu chuẩn đều thỏa mãn, sinh viên sẽ được cấp bằng tốt nghiệp và sẽ được đưa vào danh sách các cựu sinh viên. Thông tin về bằng cấp sẽ được công khai trên trang web của nhà trường. Thông tin của cựu sinh viên sẽ được lưu giữ để theo dõi quá trình làm việc (nếu sinh viên đồng ý), làm các cuộc survey, cũng như nhiều hoạt động khác. Ngoài ra hệ thống cần đảm bảo tính bảo mật và an toàn thông tin cho sinh viên, thân thiện với người dùng, dễ sử dụng và dễ dàng quản lý, được cập nhật thường xuyên để đáp ứng nhu cầu thay đổi của nhà trường và sinh viên.

# Lý do chọn kiến trúc

# Cơ sở lý thuyết

## Ngôn ngữ lập trình JavaScript

JavaScript là ngôn ngữ lập trình được nhà phát triển sử dụng để tạo trang web tương tác. Từ làm mới bảng tin trên trang mạng xã hội đến hiển thị hình ảnh động và bản đồ tương tác, các chức năng của JavaScript có thể cải thiện trải nghiệm người dùng của trang web. Là ngôn ngữ kịch bản phía máy khách, JavaScript là một trong những công nghệ cốt lõi của World Wide Web. Ngôn ngữ này cũng hỗ trợ tái sử dụng đoạn mã một cách dễ dàng, vì là một ngông ngữ bập cao và cũng hỗ trợ các kỹ thuật lập trình hướng đối tượng.

JavaScript hỗ trợ lập trình không đồng bộ và Non-blocking I/O, rất hữu ích trong việc xây dựng và phát triển các services. Vì các services có thể chờ đợi yêu cầu các dịch vụ khác từ phía client mà không cần chờ đợi phải đồng bộ.

JavaScript có nhiều thư viện và framework mạnh mẽ như Express.js, NestJS cho phát triển backend của microservices, các thư viện như Axios hoặc Fetch API cho việc giao tiếp giữa các services. Điều này hỗ trợ lập trình viên gia tăng tốc độ phát triển ứng dụng.

## Xây dựng microservice với Node.js

Việc phát triển kiến trúc microservices với Node.js sẽ cung cấp một nền tảng linh hoạt và hiệu suất cao. Với khả năng xử lý bất đồng bộ và Non-blocking I/O giúp cho việc xây dựng các microserviecs có khả năg mở rộng và phản hồi nhanh. Cộng đồng phát triển rộng rãi của Node.js cung cấp cho các nhà phát triển một hệ sinh thái đa dạng của của các thư viện và framework, giúp cái thiện hiệu suất xây dựng và tăng khả năng tái sử dụng đoạn mã. Sự linh hoạt và hiệu suất của Node.js làm cho nó trở thành một công cụ rất hữu ích trong việc xây dựng và phát triển kiến trúc microservices.

## Xậy dựng giao diện người dùng(UI) với React.js

React.js xây dựng giao diện người dùng linh hoạt và hiệu quả trong việc phát triển mô hình microservices, thư viện này giúp tải sử dụng và quản lý cách trạng thái, tách biệt các thành phần của giao diện từ đó giảm thiểu sự phụ thuộc giữa các thành phần làm tăng tính năng sử dụng lại các đoạn mã.

Khả năng phản ứng nhanh của React.js cho phép các giao diện người dùng tự động cập nhật khi cso sự thay đổi trong dữ liệu hoặc trạng thái, giúp người dùng có một trải nghiệp mượt mà khi sử dụng. Việc tích hợp các thư viện và framework khác như Redux và Axios giúp dễ dàng quả lý trạng thái và tương tác các microservice khác nhau.

## Xây dựng API Gateway với RESTful API

API Gateway là một phần trung tâm của hệ thống trong mô hình microservices, chịu trách nghiệm diều hướng các yêu cầu từ người dùng đến các dịch vụ microservices cụ thể. Đóng vai trò như một “cổng vào” cho các dịnh vụ. API Gateway chịu trách nghiệm định tuyến các yêu cầu đến các dịnh vụ microservices tương ứng. Bằng cách này, nó giúp giảm bớt sự phức tạp của việc gọi trực tiếp đến các dịch vụ, thay vào đó phía client chỉ cần gửi yêu cầu đến API Gateway và nó sẽ điều hướng yêu cầu đến dịch vụ phù hợp.

RESTful API cung cấp một cách tiếp cận linh hoạt và mạnh mẽ để quản lý và điều hướng các yêu cầu từ người dùng đến các dịnh vụ microservices. Bằng cach sử dụng RESTful API có thể xác định các tài nguyên và địa chỉ URL của các dịch vụ microservices, điều này cho phép API Gateway dễ dàng định tuyến các yêu cầu đến các dịnh vụ tương ứng. Ngoài ra, RESTful API cũng cung cấp các tính năng như xác thực ủy quyền, quản lý phiên bản và giao diện người dùng, giúp API Gateway trở nên mạnh mẽ và linh hoạt hơn trong việc quản lý hệ thống microservices.

# Phân tích thiết kế kiến trúc

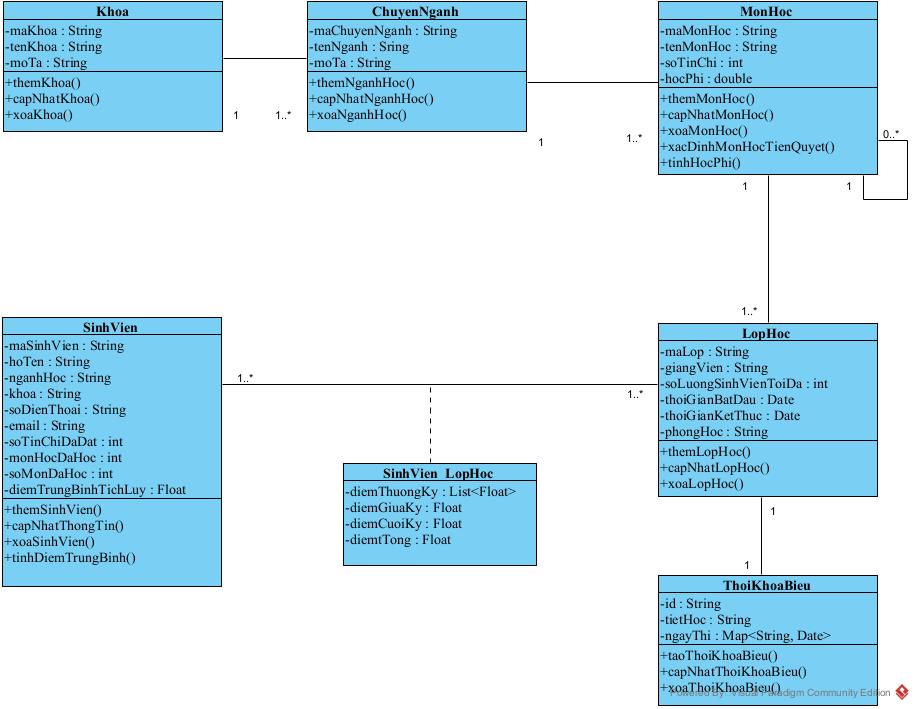
A diagram of a company

Description automatically generated

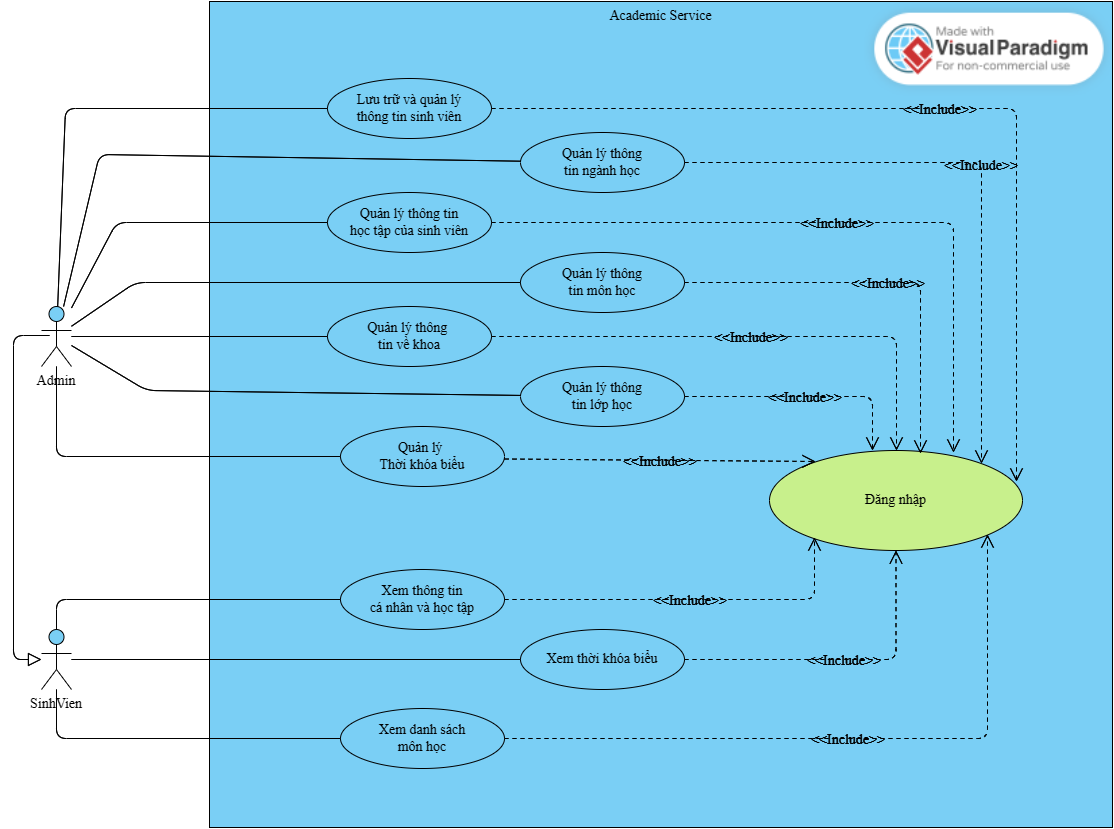
# Phân tích thiết kế hệ thống

## Quản lý Học thuật (Academic Service):

### Class diagram



### Use case diagram



### Đặc tả use case xem danh sách môn học

|  |  |
| --- | --- |
| **Use case:** UC001\_ Xem danh sách môn học | |
| **Mục đích:**Xem danh sách môn học của sinh viên | |
| **Mô tả sơ lược:**chức năng đăng nhập giúp sinh viên xem danh sách môn học | |
| **Actor chính:** Sinh viên | |
| **Actor phụ:** Không | |
| **Điều kiện trước (Pre-condition):**  - Sinh viên đăng nhập thành công | |
| **Điều kiện sau** **(Post-condition):**  - Hệ thống hiển thị danh sách môn học của sinh viên | |
| **Luồng sự kiện chính (main flow):** | |
| Actor | System |
| 1. Sinh viên chọn chức năng xem danh sách môn học | * 1. Lấy mã chuyên ngành của sinh viên |
|  | Truy xuất danh sách môn học theo chuyên ngành. |
|  | Hiện thị danh sách môn học |
| **Luồng sự kiện thay thế (alternate flow):** | |
| **Luồng sự kiện ngoại lệ (exception flow):** | |

* Activity diagram

A diagram of a system

Description automatically generated

* Sequence diagram

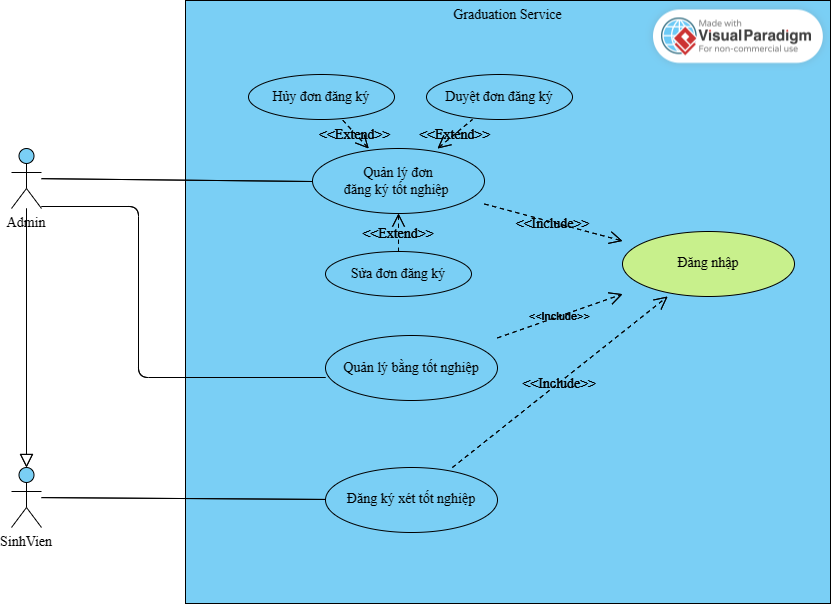
A diagram of a diagram

Description automatically generated

## Đăng ký học phần (Registration Service):

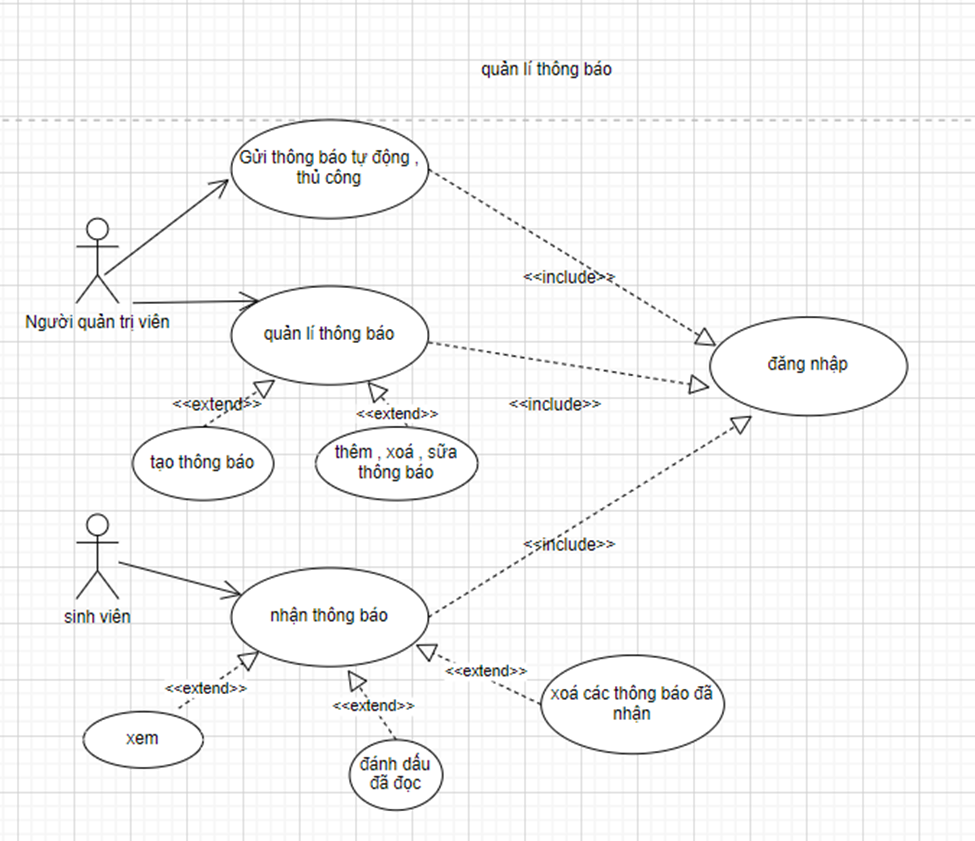
## Quản lý Xét tốt nghiệp (Graduation Service):

### Use case diagram



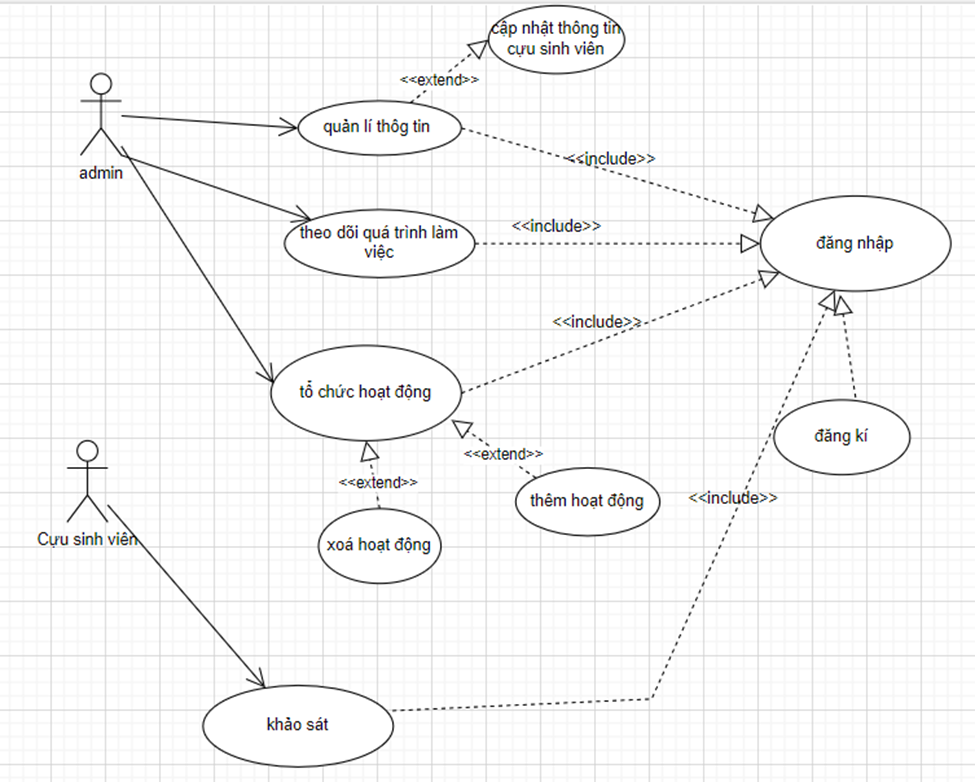
## Quản lý Tài khoản (Authentication Service):

## Quản lý Thông báo (Notification Service):

**Đặc tả usecase quản lí thông báo:**

|  |  |
| --- | --- |
| **Use case:** Quản lý Thông báo | |
| **Mục đích:** | Đảm bảo sinh viên nhận được thông báo kịp thời và chính xác về các vấn đề liên quan đến học tập, đăng ký môn học, đóng học phí, xét tốt nghiệp và các thông tin khác. |
| **Mô tả:** | Use case này bao gồm các chức năng gửi, nhận, quản lý và lưu trữ các thông báo giữa hệ thống và sinh viên. Các thông báo có thể được gửi qua email, tin nhắn trong hệ thống hoặc các kênh liên lạc khác. |
| **Tác nhân:** | Sinh viên , hệ thống , quản trị viên |
| **Điều kiện trước:** | Sinh viên đã đăng ký tài khoản và cung cấp thông tin liên lạc chính xác.  Hệ thống đã được cấu hình để gửi thông báo qua các kênh liên lạc được lựa chọn. |
| **Điều kiện sau:** | Sinh viên nhận được thông báo kịp thời và chính xác.  Sinh viên có thể xem lại lịch sử thông báo.  Quản trị viên có thể theo dõi trạng thái gửi thông báo và quản lý nội dung thông báo. |
| **Người dùng** | **Hệ thống** |
| **Luồng sự kiện chính (Basic flows)** | |
| Sinh viên xem thông báo: Đăng nhập vào hệ thống và truy cập vào mục thông báo. |  |
| Sinh viên quản lý thông báo: Đánh dấu thông báo đã đọc, xóa thông báo hoặc thay đổi cài đặt thông báo |  |
| Quản trị viên tạo thông báo: Soạn thảo nội dung thông báo, chọn đối tượng nhận và lên lịch gửi. | Hệ thống tạo thông báo tự động: Tạo thông báo dựa trên các sự kiện (đăng ký môn học, đóng học phí, kết quả học tập) hoặc theo lịch trình. |
| Quản trị viên gửi thông báo: Gửi thông báo đến một hoặc nhiều sinh viên. | Hệ thống gửi thông báo: Gửi thông báo đến sinh viên qua các kênh liên lạc đã được cấu hình (email, tin nhắn trong hệ thống, ứng dụng di động). |
|  | Hệ thống lưu trữ thông báo: Lưu trữ lịch sử thông báo để sinh viên và quản trị viên có thể xem lại |
|  | Hệ thống theo dõi trạng thái gửi thông báo: Ghi nhận thông báo đã gửi thành công hoặc không thành công. |
| **Luồng sự kiện phụ (Alternative Flows)** | |
| 6.1 Sinh viên không nhận được thông báo: Kiểm tra lại cài đặt thông báo hoặc liên hệ với quản trị viên. | 6.2 Gửi thông báo không thành công: Ghi lại lỗi và có thể thử gửi lại sau |
|  | 6.3 Thông báo bị lỗi: Hiển thị thông báo lỗi cho sinh viên hoặc quản trị viên. |
| 6.4 quản trị viên xác nhận và kết thúc usecase |  |

## Quản lý Cựu sinh viên (Alumni Service):

**Đặt tả usecase:**

|  |  |
| --- | --- |
| **Use case:** UC006\_Quản lý thông tin và hoạt động của cựu sinh viên | |
| **Mục đích:** | Duy trì mối quan hệ với cựu sinh viên, cập nhật thông tin, hỗ trợ và tạo cơ hội kết nối giữa cựu sinh viên và nhà trường. |
| **Mô tả:** | Use case này cho phép quản trị viên quản lý thông tin cựu sinh viên, theo dõi quá trình làm việc, thực hiện khảo sát và tổ chức các hoạt động khác. |
| **Tác nhân:** | Quản trị viên cựu sinh viên |
| **Điều kiện trước:** | Sinh viên đã tốt nghiệp và được cấp bằng.  Thông tin của cựu sinh viên đã được lưu trữ trong hệ thống. |
| **Điều kiện sau:** | Thông tin cựu sinh viên được cập nhật đầy đủ và chính xác.  Nhà trường duy trì mối quan hệ tốt với cựu sinh viên. |
| **Người dùng** | **Hệ thống** |
| **Luồng sự kiện chính (Basic flows)** | |
| Quản trị viên đăng nhập vào hệ thống quản lý cựu sinh viên. |  |
| Quản trị viên xem danh sách cựu sinh viên. | Hệ thống hiển thị danh sách các cựu sinh viên của trường |
| Quản trị viên chọn một cựu sinh viên để xem chi tiết. | Hệ thống hiển thị chi tiết thông tin các cựu sinh viên ( tên , năm tốt nghiệp , …) |
| Quản trị viên xem và cập nhật thông tin của cựu sinh viên (nếu cần). |  |
| Quản trị viên có thể thực hiện các tác vụ sau:  +Theo dõi quá trình làm việc của cựu sinh viên (nếu được phép).  +Gửi khảo sát hoặc thư mời tham gia sự kiện.  +Tạo báo cáo thống kê về cựu sinh viên. | Hệ thống xác nhận và thực hiện các yêu cầu của người dùng. |
| Quản trị viên lưu thay đổi và thoát khỏi hệ thống. |  |
| **Luồng sự kiện phụ (Alternative Flows)** | |
|  | 2.1 Nếu không tìm thấy cựu sinh viên, hệ thống hiển thị thông báo tương ứng. |
| 4.1 Nếu thông tin cựu sinh viên không chính xác, quản trị viên cập nhật và lưu thay đổi. |  |
|  | 4.3 Hệ thống hiển thị xác nhận , kết thúc usecase |

**Q**

# Hiện Thực

# Kiểm Thử

# Kết luận

# Tài liệu tham khảo