**BỘ CÔNG THƯƠNG**

**TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP TP.HCM**

A logo with a red and blue letter and a drop of water

Description automatically generated

**HỆ THỐNG QUẢN LÝ ĐĂNG KÝ HỌC PHẦN**

***Nhóm 31 - Sinh viên thực hiện***

|  |  |  |
| --- | --- | --- |
| **STT** | **Họ và tên** | **Mã số sinh viên** |
| 1 | Nguyễn Khánh An (NT) | 20122281 |
| 2 | Lê Hữu Hiệp | 20115701 |
| 3 | Bùi Gia Đại | 20102181 |

Mục lục

I. Giới thiệu 5

1.1. Giới thiệu chung 5

1.2. Thành viên và vai trò 5

1.3. Định nghĩa và từ viết tắt 5

1.4. Tài liệu tham khảo 5

II. CƠ SỞ LÝ THUYẾT 6

2.1. Tổng quan, lịch sử và nguồn gốc kiến trúc Microservices: 6

2.2. Các thành phần, cách hoạt động của Microservices: 7

2.3. Lý do lựa chọn áp dụng kiến trúc Microservices vào hệ thống 11

2.4. Ưu điểm, nhược điểm của kiến trúc Microservices 12

2.5. Các bước xây dựng kiến trúc Microservices 13

2.6. Đánh giá đặc điểm kiến trúc Microservices 14

III. Mục tiêu và ràng buộc của kiến trúc 14

3.1. Mục tiêu 14

3.2. Ràng buộc của kiến trúc 14

IV. Tổng quan kiến trúc (High-level Architectural Overview) 15

4.1. Sự phụ thuộc vào các hệ thống khác 15

4.2. Sự phụ thuộc vào các kho dữ liệu 15

V. Use-Case View 16

5.1. Tổng quát 16

5.4. Hiện thực hoá (Use-Case Realizations) 17

VI. Quan điểm logic (Logical View) 22

6.1. Tổng quan Frontend 22

6.2. Tổng quan backend 22

6.3. Mô hình phân tích 24

6.4. Các gói, hệ thống con, lớp hoặc tầng quan trọng 24

VII. Hiện thực hoá (Deployment View) 24

8.1. Sơ đồ triển khai 24

- Mục đích của API Gateway: 26

o Quản lý và điều hướng yêu cầu: API Gateway nhận tất cả các yêu cầu từ client và định tuyến chúng đến các dịch vụ backend phù hợp. 26

**Lợi ích của việc sử dụng Keycloak trong kiến trúc Microservices** 27

**Cách hoạt động của Keycloak trong Microservices** 27

**Lợi ích của Circuit Breaker:** 28

**Mục đích của Distributed Tracing:** 29

**Cách hoạt động của Distributed Tracing** 29

**Tổng quan về Kafka** 29

Các thành phần chính trong Kafka: 29

**Kiến trúc Hướng Sự kiện với Kafka** 29

**Lợi ích của Dockerizing an Application** 30

VIII. Data View 31

Tổng quan 31

Data Access Mechanisms 31

IX. Process View 33

X. Kích thước và hiệu suất 37

XI. Chất lượng 37

XII. Các mẫu và cơ chế chung (Common Patterns and Mechanisms) 37

Microservices Architecture Document

# Giới thiệu

## Giới thiệu chung

Hệ thống quản lý lớp học tín chỉ được thiết kế để đáp ứng nhu cầu đăng ký học phần, theo dõi tiến trình học tập, và quản lý thông tin học thuật của sinh viên tại một trường đại học. Hệ thống sẽ cho phép sinh viên dễ dàng tìm kiếm và đăng ký các môn học, lớp tín chỉ phù hợp với yêu cầu học tập của họ.

Hệ thống bao gồm nhiều dịch vụ nhỏ, độc lập, có thể triển khai riêng biệt và giao tiếp với nhau thông qua API. Sử dụng mô hình Microservices cho backend và mô hình MVC (Model-View-Controller) cho frontend.

## Định nghĩa và từ viết tắt

|  |  |
| --- | --- |
| **Thuật ngữ** | **Giải thích** |
| json |  |
| token |  |

## Tài liệu tham khảo

+ MVC Framework – Introduction: <https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.html>

+ Everything you need to know about MVC architecture:

<https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1>

+ Fundamentals of Software Architecture An Engineering Approach - Mark Richards & Neal Ford: <https://drive.google.com/file/d/1ggN8NPgwQN-aoKWyoizbjY4padyG4M6z/view?usp=sharing>

# CƠ SỞ LÝ THUYẾT

## Tổng quan, lịch sử và nguồn gốc kiến trúc Microservices:

A diagram of a user interface layer

Description automatically generated

*Hinh 1 Mô hình kiến trúc Microservices cơ bản*

Microservices đã trở thành một kiểu kiến trúc rất phổ biến và thu hút sự chú ý đáng kể trong những năm gần đây. Nó được đặt tên từ sớm và phổ biến qua một bài đăng blog của Martin Fowler và James Lewis có tựa đề “Microservices” được xuất bản vào tháng 3 năm 2014.

Microservices được cảm hứng mạnh mẽ từ các ý tưởng trong thiết kế theo miền (DDD), một quy trình thiết kế logic cho các dự án phần mềm. Một khái niệm cụ thể từ DDD, là bounded context, là nơi mỗi phần của hệ thống được đóng gói để làm việc mà không liên kết với các phần khác đã quyết định làm nên Microservices. Microservices hướng tới sự tách biệt cao, mô hình hóa vật lý cho khái niệm logic của bounded context, cho phép các ứng dụng phát triển linh hoạt và dễ bảo trì bằng cách chia nhỏ thành các dịch vụ độc lập. Bởi trong một bounded context, các phần bên trong, như mã nguồn và cấu trúc dữ liệu, được kết hợp với nhau để tạo ra công việc; nhưng chúng không bao giờ được kết nối với bất kỳ thứ gì bên ngoài bounded context, chẳng hạn như cơ sở dữ liệu hoặc định nghĩa lớp từ một bounded context khác. Điều này cho phép mỗi context chỉ xác định những gì nó cần thay vì phải phục vụ các thành phần khác.

Martin Fowler có một câu rất thú vị rằng: “The term ‘microservice’ is a label, not a description”: Đại ý là người ta đã vô tình đặt cái tên “microservice” cho một phong cách kiến trúc mới Microservices, để đối lập với phong cách kiến trúc thịnh hành thời điểm đó nhưng lại hết sức “cồng kềnh” là “Service-oriented Architecture” (kiến trúc hướng dịch vụ).

## Các thành phần, cách hoạt động của Microservices:

A diagram of a software application

Description automatically generated

*Hinh 2. Các thành phần kiến trúc Microservices*

Microservice được xây dựng dựa trên khái niệm "bounded context": Mỗi service mô hình hoá một domain hoặc một workflow. Nghĩa là mỗi Service chứa mọi thứ cần thiết để hoạt động trong ứng dụng, bao gồm các class, các thành phần con trong nó và Database.

Tính tái sử dụng và ghép nối:

* + Khi kiến trúc sư thiết kế một hệ thống ủng hộ việc sử dụng lại, họ cũng ủng hộ việc ghép nối để đạt được việc tái sử dụng đó, hoặc bằng kế thừa hoặc tổ hợp.
  + Tuy nhiên, nếu mục tiêu của kiến trúc sư yêu cầu mức độ tách rời cao thì họ thiên về sao chép hơn là tái sử dụng. Mục tiêu chính của Microservice là khả năng tách rời cao, mô hình hóa vật lý khái niệm logic về bối cảnh bị giới hạn.

Do tính chất đơn mục đích nên quy mô dịch vụ trong Microservice nhỏ hơn so với các kiến trúc phân tán khác. Các thành phần của kiến trúc Microservice cơ bản bao gồm:

* Service: chứa logic nghiệp vụ của microservice, chỉ có service mới được truy cập vào database
* Database: một tập hợp các bảng chứa dữ liệu phục vụ cho logic nghiệp vụ của microservice.
* API gateway: cung cấp một điểm truy cập thống nhất cho các Microservice, giúp quản lý, điều phối và bảo mật giao tiếp giữa các service với nhau trong kiến trúc Microservices, đồng thời cung cấp một nền tảng để triển khai các chính sách quản lý API và tối ưu giao tiếp. Để tìm hiểu sâu hơn chúng ta sẽ tìm hiểu cấu trúc hoạt động của kiến trúc.
* Lớp API: Hầu hết các hình ảnh về vi dịch vụ đều bao gồm một lớp API nằm giữa những người sử dụng hệ thống (giao diện người dùng hoặc lệnh gọi từ các hệ thống khác), nhưng đây là tùy chọn. Điều này phổ biến vì nó cung cấp một vị trí tốt trong kiến trúc để thực hiện các nhiệm vụ hữu ích thông qua gián tiếp dưới dạng proxy hoặc liên kết với các cơ sở hoạt động, chẳng hạn như dịch vụ đặt tên. Mặc dù lớp API có thể được sử dụng cho nhiều mục đích khác nhau nhưng không nên sử dụng nó làm công cụ hòa giải hoặc điều phối nếu kiến trúc sư muốn giữ đúng triết lý cơ bản của kiến trúc này: tất cả logic thú vị trong kiến trúc này sẽ diễn ra bên trong một bối cảnh bị giới hạn và việc đặt sự phối hợp hoặc logic khác vào một bộ hòa giải sẽ vi phạm quy tắc đó.
* Tái sử dụng hoạt động (Operational Reuse)

A diagram of a service plan

Description automatically generated

*Hinh 3. Mẫu sidecar trong microservice*

Trong kiến trúc microservices, việc xử lý các vấn đề vận hành như giám sát và ghi nhật ký đòi hỏi một cách tiếp cận khéo léo. Một giải pháp phổ biến là sử dụng mẫu một bên(sidecar). Mẫu Sidecar là một mẫu thiết kế tập trung vào việc mở rộng chức năng của dịch vụ chính bằng cách gắn một thùng chứa phụ,được gọi là "sidecar" cho nó. Trong đó các vấn đề vận hành chung được xử lý bởi một thành phần bên riêng, có thể được sở hữu bởi các nhóm cá nhân hoặc một nhóm cơ sở hạ tầng chia sẻ. Khi đến lúc nâng cấp, các nhóm chỉ cần cập nhật thành phần bên chung, và tất cả các dịch vụ microservices sẽ nhận được các cải tiến mới. Việc xây dựng một mạng dịch vụ cho phép kiểm soát thống nhất trên toàn bộ kiến trúc cho các vấn đề như ghi nhật ký và giám sát. Khám phá dịch vụ được sử dụng để tăng tính linh hoạt vào các kiến trúc microservices bằng cách cho phép triển khai các phiên bản mới của các dịch vụ và xử lý vấn đề quy mô hoặc linh hoạt.

Giao diện người dùng (Frontends): Microservices đề cao việc tách biệt các thành phần, bao gồm cả giao diện người dùng và backend. Ban đầu, microservices đề xuất giao diện người dùng là một phần của ngữ cảnh giới hạn, nhưng trong thực tế, các ràng buộc bên ngoài và yêu cầu phân vùng từ các ứng dụng web khiến việc này khó khăn. Do đó, có hai phong cách giao diện người dùng phổ biến cho microservices:

* Giao diện người dùng nguyên khối: Giao diện người dùng duy nhất gọi qua lớp API để thỏa mãn các yêu cầu của người dùng. Điều này phổ biến trong các ứng dụng web hiện đại sử dụng các framework JavaScript để xây dựng giao diện người dùng duy nhất.

A diagram of a computer application

Description automatically generated

*Hinh 4. Kiến trúc microservice với giao diện người dùng nguyên khối*

* Microfrontends (user interfaces uses microfrontends): Sử dụng các thành phần tại cấp độ giao diện người dùng để tạo ra một mức độ tách biệt và đồng bộ giống như các dịch vụ backend. Mỗi dịch vụ phát ra giao diện người dùng của riêng mình, mà frontend sẽ phối hợp với các thành phần giao diện người dùng khác được phát ra. Điều này giúp cô lập ranh giới dịch vụ từ giao diện người dùng đến backend, thống nhất toàn bộ miền trong một nhóm duy nhất. Các nhà phát triển có thể triển khai mẫu microfrontend bằng cách sử dụng các framework web dựa trên thành phần như React hoặc sử dụng các framework mã nguồn mở hỗ trợ mẫu này.

A diagram of a computer component

Description automatically generated

*Hinh 5. Mẫu microfrontend trong microservice*

+ Giao tiếp (Communication): Trong kiến trúc microservices, các kiến trúc sư và nhà phát triển đối mặt với việc xác định độ phân cấp phù hợp, ảnh hưởng đến cả việc cô lập dữ liệu và giao tiếp. Quyết định về kiểu giao tiếp đồng bộ hoặc không đồng bộ đóng vai trò quan trọng trong việc giữ cho các dịch vụ được tách biệt nhưng vẫn được phối hợp một cách có ích. Cơ bản, các kiến trúc sư phải quyết định về việc giao tiếp đồng bộ hoặc không đồng bộ. Giao tiếp đồng bộ đòi hỏi người gọi phải chờ đợi một phản hồi từ người được gọi. Trong khi đó, giao tiếp không đồng bộ thường sử dụng sự kiện và tin nhắn, sử dụng kiến trúc dựa trên sự kiện. Trong việc tương tác giữa các dịch vụ, choreography và orchestration là hai mô hình phổ biến.

* Choreography không có trung tâm điều phối trung ương, mỗi dịch vụ gọi các dịch vụ khác khi cần thiết.

A diagram of a computer

Description automatically generated  
*Hình 6: Sử dụng choreography trong microservices để quản lý điều hướng*

* Orchestraton tạo ra một trung tâm điều phối để xử lý các quy trình kinh doanh phức tạp.

A diagram of a service

Description automatically generated

*Hinh 6. Sử dụng tính năng phối hợp trong microservice*

=> Cả hai mô hình này đều có nhược điểm và lợi ích riêng, và việc lựa chọn phụ thuộc vào yêu cầu cụ thể của dự án và mức độ phức tạp của quy trình kinh doanh. Quan trọng nhất là tìm ra cách biểu diễn sự liên kết giữa các dịch vụ một cách hợp lý, phù hợp với cả mục tiêu kiến trúc và miền.

+ Giao dịch và Sagas:

* Là hai khía cạnh quan trọng trong kiến trúc microservices. Mục tiêu chính của các kiến trúc sư là tách biệt tối đa giữa các dịch vụ, nhưng điều này thường gây khó khăn khi cần phối hợp các giao dịch giữa chúng. Vấn đề nảy sinh khi tính nguyên tử của các giao dịch, một khái niệm dễ áp dụng trong ứng dụng monolithic, trở nên phức tạp hơn trong môi trường phân tán.
* Việc xây dựng giao dịch vượt qua các ranh giới dịch vụ không chỉ vi phạm nguyên tắc tách rời mà còn tạo ra loại kết nối động tồi tệ nhất, gọi là sự đồng hiện của giá trị. Vì vậy, lời khuyên tốt nhất là “tránh thực hiện giao dịch qua các dịch vụ mà thay vào đó tập trung vào việc điều chỉnh mức độ chi tiết của các thành phần”. Thường, việc xây dựng các giao dịch xuyên ranh giới là một dấu hiệu cho thấy kiến trúc đã quá chi tiết.
* Tuy nhiên, có những trường hợp ngoại lệ khi mà việc phối hợp giao dịch giữa các dịch vụ là cần thiết. Trong những tình huống này, một mẫu phân tán phổ biến được sử dụng là mô hình saga. Trong mô hình này, một dịch vụ trung gian đóng vai trò hòa giải giữa các cuộc gọi dịch vụ và điều phối giao dịch. Người hòa giải phải đảm bảo rằng giao dịch chỉ được coi là thành công nếu tất cả các phần của nó đều thành công. Trong trường hợp xảy ra lỗi, người hòa giải phải phối hợp các dịch vụ để hoàn tác các thay đổi đã thực hiện.
* Tóm lại, trong kiến trúc microservices, việc thực hiện các giao dịch giữa các dịch vụ là một thách thức. Tuy nhiên, việc sử dụng mô hình saga có thể giúp giải quyết vấn đề này một cách linh hoạt, mặc dù việc sử dụng giao dịch nên được hạn chế và chỉ sử dụng khi thực sự cần thiết.

## Lý do lựa chọn áp dụng kiến trúc Microservices vào hệ thống

**1. Khả năng mở rộng linh hoạt (Scalability)**

Cho phép mở rộng từng phần của hệ thống một cách độc lập. Nếu một dịch vụ cần xử lý nhiều yêu cầu hơn, chúng ta có thể mở rộng chỉ dịch vụ đó mà không cần mở rộng toàn bộ hệ thống. Ví dụ, nếu có nhiều sinh viên truy cập vào hệ thống để đăng ký học phần trong thời gian cao điểm, chỉ cần mở rộng phần dịch vụ đăng nhập mà không ảnh hưởng đến các phần khác.

**2. Tính linh hoạt trong phát triển và triển khai (Flexibility in Development and Deployment)**

Mỗi dịch vụ trong hệ thống (đăng ký học phần, quản lý sinh viên, quản lý môn học, thanh toán học phí,...) có thể được phát triển - kiểm thử - triển khai độc lập. Điều này giúp giảm thời gian triển khai các tính năng mới hoặc sửa lỗi. Ứng dụng cụ thể là ta có thể cập nhập riêng lẻ mà không cần dừng lại toàn bộ hệ thống.

**3. Khả năng chịu lỗi cao (Fault Isolation)**

Lỗi trong một dịch vụ sẽ không ảnh hưởng đến các dịch vụ khác. Điều này giúp hệ thống vẫn hoạt động bình thường dù có sự cố ở một phần cụ thể.Ví dụ, xuất hiện lỗi trong dịch vụ thanh toán học phí sẽ không ảnh hưởng đến đăng ký học phần hoặc tra cứu thông tin môn học.

**4. Tăng cường bảo trì và nâng cấp (Maintainability and Upgradability)**

Mỗi dịch vụ có thể được phát triển bằng công nghệ, mô hình kiến trúc phù hợp nhất. Dễ dàng thêm tính năng mới của từng dịch vụ cụ thể. Các dịch vụ nhỏ hơn, tách biệt dễ bảo trì và nâng cấp.

**5. Tái sử dụng dịch vụ (Reusability)**

Các dịch vụ như xác thực người dùng, quản lý sinh viên, có thể được tái sử dụng trong các ứng dụng khác như hệ thống quản lý điểm, hệ thống thư viện.

**6. Khả năng quan sát và giám sát (Observability and Monitoring)**

Dễ dàng theo dõi hiệu suất và hành vi của từng dịch vụ riêng biệt, giúp phát hiện và khắc phục sự cố nhanh chóng.

**7. Hỗ trợ tích hợp liên tục và triển khai liên tục (CI/CD)**

Hỗ trợ tốt hơn cho CI/CD, cho phép tự động xây dựng, kiểm thử và triển khai từng dịch vụ một cách nhanh chóng và độc lập.

**8. Phù hợp với DevOps và Agile**

Khuyến khích sự cộng tác chặt chẽ giữa các nhóm phát triển và vận hành, phù hợp với phương pháp Agile, cho phép phản ứng nhanh chóng với thay đổi và cải tiến liên tục.

## Ưu điểm, nhược điểm của kiến trúc Microservices

1. **Ưu điểm:**

* Cho phép lập trình viên linh động hơn trong việc lựa chọn ngôn ngữ, công cụ và nền tảng để phát triển và triển khai các microservice (tuy nhiên trong một hệ thống, việc lựa chọn các ngôn ngữ khác nhau để phát triển các microservice không được khuyến khích).
* Dễ dàng thực hiện tự động tích hợp và tự động triển khai (CI-CD) bằng cách sử dụng một số công cụ như Jenkins, Hudson …
* Mỗi microservice có kích thước nhỏ, giúp cho các lập trình viên dễ tiếp cận, đọc hiểu source code. Do vậy các thành viên mới tham gia team sẽ hòa nhập và đóng góp cho team nhanh hơn.
* Dễ dàng mở rộng và tích hợp với các dịch vụ của bên thứ ba.
* Cô lập lỗi tốt hơn, khi một microservice bị lỗi và ngừng hoạt động thì các microservice khác vẫn có thể hoạt động bình thường. Với mô hình nguyên khối, một lỗi nhỏ có thể làm cả hệ thống ngừng hoạt động.
* Khi cần thay đổi một thành phần, thì chỉ cần sửa đổi, cập nhật và triển khai lại thành phần đó chứ không cần triển khai lại toàn bộ hệ thống.

1. **Nhược điểm:**

* Nhược điểm của kiến trúc microservice đến từ bản chất của hệ thống phân tán.
* Việc triển khai hệ thống microservice phức tạp hơn nhiều so với việc triển khai hệ thống nguyên khối.
* Cần tính toán kích cỡ của một microservice. Nếu một microservice quá lớn, bản thân nó trở thành một ứng dụng theo kiến trúc nguyên khối.
* Nếu một microservice quá nhỏ thì độ phức tạp của hệ thống tăng lên rất nhiều, làm cho hệ thống trở lên khó hiểu, lúc này việc quản lý giám sát và triển khai hệ thống sẽ khó khăn hơn.
* Khi ứng dụng ngày càng lớn lên, số lượng microservice ngày càng nhiều, các lập trình viên thường có xu hướng sử dụng sự hỗ trợ từ các công cụ mã nguồn mở, hoặc của bên thứ 3, việc sử dụng, tích hợp các công cụ này làm cho hệ thống khó kiểm soát và có thể bị dính các mã độc làm cho hệ thống kém an toàn.

## Các bước xây dựng kiến trúc Microservices

**Bước 1:** **Thiết kế microservice**

Thiết kế microservice: Xác định các chức năng của ứng dụng của bạn có thể được chia thành các dịch vụ độc lập, riêng biệt. Mỗi microservice phải có một trách nhiệm duy nhất và được liên kết lỏng lẻo với các microservice khác. Chọn các framework phù hợp để xây dựng microservice trong Java như Spring Boot, Micronaut hoặc Vert.x. Các khung này cung cấp các công cụ và tính năng để xây dựng các dịch vụ vi mô có khả năng mở rộng và linh hoạt. Xác định API cho từng microservice, bao gồm các tham số đầu vào, phản hồi đầu ra và cơ chế xử lý lỗi. Sử dụng các phương pháp tiếp cận REST ful hoặc theo hướng sự kiện để liên lạc giữa các vi dịch vụ.

**Bước 2: Thiết lập giao tiếp giữa các microser**

Thiết lập giao tiếp giữa các microservice: Quyết định các giao thức và mô hình liên lạc giữa các microservice. Các tùy chọn phổ biến bao gồm HTTP/REST, gRPC và các trình môi giới tin nhắn như Apache Kafka hoặc RabbitMQ. Triển khai các cơ chế giao tiếp bằng cách sử dụng các khung và thư viện đã chọn. Ví dụ: bạn có thể sử dụng Spring Cloud, Micronaut Discovery và/hoặc Eureka để khám phá dịch vụ và Spring Cloud Gateway hoặc Netflix Zuul cho chức năng cổng API. Triển khai các cơ chế xử lý lỗi và khả năng chịu lỗi, chẳng hạn như ngắt mạch, thử lại và dự phòng, để đảm bảo khả năng phục hồi và độ tin cậy trong microservice giao tiếp.

**Bước 3: Xử lý việc lưu trữ dữ liệu**

Triển khai lưu trữ dữ liệu bằng các công nghệ thích hợp, chẳng hạn như cơ sở dữ liệu quan hệ như MySQL hoặc PostgreSQL, cơ sở dữ liệu NoSQL như MongoDB hoặc Cassandra hoặc kho dữ liệu trong bộ nhớ như Redis hoặc Hazelcast. - Sử dụng các framework và thư viện thích hợp để truy cập dữ liệu, chẳng hạn như JPA hoặc Spring Data, để tương tác với cơ sở dữ liệu từ microservice Java.

**Bước 4: Quản lý triển khai**

Chọn chiến lược triển khai cho vi dịch vụ của bạn, chẳng hạn như đóng gói bằng Docker hoặc ảo hóa bằng máy ảo. Việc container hóa thường được sử dụng trong các kiến trúc microservice do tính linh hoạt và khả năng mở rộng của nó. - Sử dụng các công cụ điều phối vùng chứa như Kubernetes hoặc Docker Swarm để quản lý việc triển khai, mở rộng quy mô và giám sát các vi dịch vụ. Triển khai các cơ chế ghi nhật ký, giám sát và truy tìm để đảm bảo khả năng quan sát và khắc phục sự cố dịch vụ vi mô trong sản xuất.

**Bước 5: Kiểm tra và triển khai microservices**

Viết bài kiểm tra đơn vị, bài kiểm tra tích hợp và bài kiểm tra đầu cuối cho từng microservice để đảm bảo chức năng của nó và độ tin cậy. Sử dụng các phương pháp tích hợp liên tục và triển khai liên tục (CI/CD) để tự động xây dựng, kiểm tra, và triển khai microservices vào môi trường sản xuất. Giám sát và tối ưu hóa hiệu suất cũng như độ tin cậy của microservice trong sản xuất để đảm bảo hoạt động trơn tru hoạt động.

## Đánh giá đặc điểm kiến trúc Microservices

Mặc dù microservice rất phổ biến và mạnh mẽ nhưng có lẽ đây cũng là kiến trúc khó triển khai nhất.

Kiến trúc này được đánh giá cao nhất ở khả năng mở rộng, khả năng chịu lỗi và tính linh hoạt. Tuy nhiên, kiến trúc này được đánh giá là chi phí cao, việc áp dụng phức tạp và gặp nhiều vấn đề liên quan đến hiệu suất.

Kiến trúc microservices mang lại nhiều tính năng cực đoan trên thang điểm đánh giá, từ mức độ hỗ trợ cho các thực hành kỹ thuật hiện đại đến khả năng chịu lỗi và tin cậy. Mặc dù gặp phải những vấn đề về hiệu suất và tính khả dụng khi có quá nhiều giao tiếp giữa các dịch vụ, nhưng với sự độc lập và mục đích đơn giản của các dịch vụ, kiến trúc này thường dẫn đến khả năng chịu lỗi cao. Với triết lý tập trung vào miền và tính tách rời cực đoan, microservices tạo ra nhiều thách thức nhưng cũng mang lại nhiều lợi ích lớn nếu được triển khai đúng cách.

# Mục tiêu và ràng buộc của kiến trúc

## Mục tiêu

* **An toàn:** Hệ thống cần phải an toàn để bảo đảm quyền riêng tư thông tin cá nhân.
* **Khả năng mở rộng tương lai**: Hệ thống cần có khả năng mở rộng để đáp ứng nhu cầu ngày càng tăng của sinh viên. Kiến trúc microservice sẽ cho phép chúng ta dễ dàng mở rộng các dịch vụ riêng lẻ mà không ảnh hưởng đến toàn bộ hệ thống.
* **Khả năng phục hồi**: Hệ thống cần có khả năng phục hồi sau sự cố. Có hỗ trợ chuyển đổi dữ liệu dự phòng (tái sử dụng dữ liệu cho mục đích khác). Kiến trúc microservice sẽ cho phép chúng ta dễ dàng cô lập các lỗi và phục hồi các dịch vụ bị ảnh hưởng mà không làm gián đoạn toàn bộ hệ thống.
* **Hiệu suất**: Hệ thống cần có hiệu suất cao để đáp ứng nhu cầu của người dùng. Kiến trúc microservice sẽ cho phép chúng ta tối ưu hóa hiệu suất của từng dịch vụ riêng lẻ cụ thể. Hoạt động 24/7.
* **Bảo mật**: Hệ thống cần đảm bảo bảo mật dữ liệu của người dùng. Kiến trúc microservice sẽ cho phép chúng ta dễ dàng triển khai các biện pháp bảo mật cho từng dịch vụ riêng lẻ. Authentication: đăng nhập tài khoản và mật khẩu. Thông tin cần được mã hoá.
* **Dễ bảo trì**: Hệ thống cần dễ bảo trì để giảm thiểu chi phí vận hành. Kiến trúc microservice sẽ cho phép chúng ta dễ dàng sửa chữa và nâng cấp các dịch vụ riêng lẻ mà không ảnh hưởng đến toàn bộ hệ thống.
* **Thanh toán**: Lưu trữ các hành động quan trọng, nhạy cảm như hoá đơn, thao tác huỷ đăng ký, ấn học lại(huỷ điểm cũ).
* **Tính không phủ nhận (Non-repudiation):** Cung cấp bằng chứng cho thấy một hành động cụ thể đã xảy ra.

## Ràng buộc của kiến trúc

* **Triển khai phức tạp:** Việc triển khai hệ thống microservice phức tạp hơn nhiều so với việc triển khai hệ thống nguyên khối.
* **Cần tính toán kích cỡ của một microservice:** Nếu một microservice quá lớn, bản thân nó trở thành một ứng dụng theo kiến trúc nguyên khối.
* **Quản lý khó khăn:** Nếu một microservice quá nhỏ thì độ phức tạp của hệ thống tăng lên rất nhiều, làm cho hệ thống trở lên khó hiểu, lúc này việc quản lý giám sát và triển khai hệ thống sẽ khó khăn hơn.
* **Nguy cơ về bảo mật:** Khi ứng dụng ngày càng lớn lên, số lượng microservice ngày càng nhiều => các lập trình viên thường sẽ sử dụng sự hỗ trợ từ các công cụ mã nguồn mở, hoặc của bên thứ 3, việc sử dụng, tích hợp các công cụ này làm cho hệ thống khó kiểm soát và có thể bị dính các mã độc làm cho hệ thống kém an toàn.
* **Yêu cầu cao về cơ sở hạ tầng:** Vì đằng sau mỗi dịch vụ là một cơ sở hạng tầng riêng biệt nên việc cơ sở hạng tầng sẽ tăng trưởng theo các dịch vụ. Chi phí vận hành nhiều dịch vụ cũng là gánh nặng lớn.
* **Cần nhân lực trình độ cao:** Cần đầu tư vào các công cụ và đào tạo nhân lực để quản lý và vận hành hệ thống phân tán.
* **Việc đồng bộ dữ liệu:** Việc quản lý giao dịch hệ thống phân tán như của kiến trúc microservice rất phức tạp. Điều này nằm ở việc dữ liệu cần nhất quán với nhau giữa nhiều dịch vụ.
* **Độ trễ:**Vì các dịch vụ tách rời nhau nên chúng em đang hướng đến giao tiếp qua mạng. Việc giao tiếp qua mạng có thể ảnh hưởng bởi nhiều yếu tố gây ra độ trễ.
* **Quá tải:**Khả năng nguy cơ quá tải đến từ việc cấu hình và quản lý không đúng cách làm số lượng lớn yêu cầu qua lại giữa các dịch vụ.

# Tổng quan kiến trúc (High-level Architectural Overview)

## Sự phụ thuộc vào các hệ thống khác

Hệ thống quản lý lớp học tín chỉ có thể phụ thuộc vào các hệ thống khác để thực hiện đầy đủ các chức năng. Các phụ thuộc chính có thể bao gồm:

* Hệ thống Xác thực người dùng (Authention service): Hệ thống này đảm bảo rằng chỉ những người dùng được ủy quyền mới có thể truy cập vào hệ thống. Các dịch vụ xác thực có thể bao gồm OAuth2, OpenID Connect hoặc LDAP.
* Hệ thống Thanh toán (Credit payment service): Để xử lý học phí và các khoản thanh toán khác, hệ thống có thể tích hợp với các cổng thanh toán như Stripe, PayPal, hoặc các ngân hàng, Momo, ZaloPay, ViettelPay,…

## Sự phụ thuộc vào các kho dữ liệu

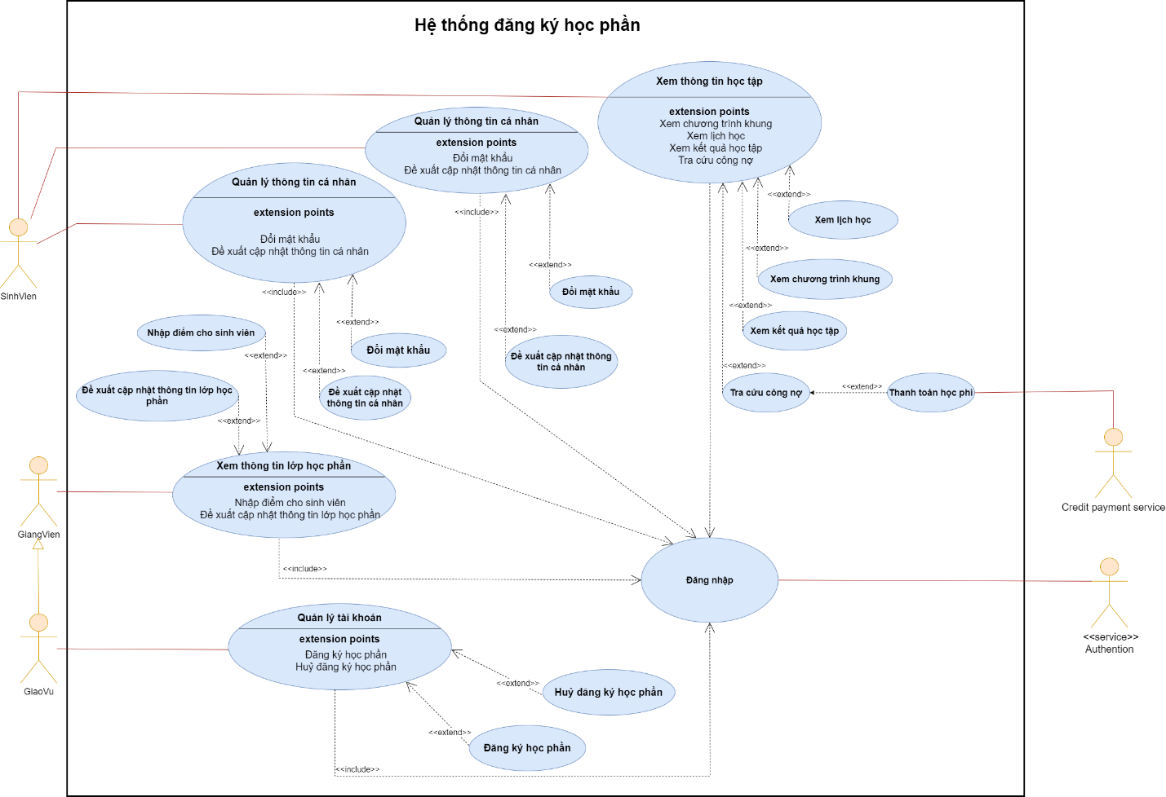
Hệ thống cần truy cập và lưu trữ dữ liệu trên các kho dữ liệu khác nhau. Các phụ thuộc chính bao gồm:

* CSDL Quan hệ (RDBMS): Sử dụng MySQL hoặc PostgreSQL để lưu trữ dữ liệu chính về sinh viên, giảng viên, khóa học, đăng ký học phần, và điểm số.
* Redis: Sử dụng cho caching để cải thiện hiệu suất của hệ thống, đặc biệt là trong các tác vụ đọc dữ liệu thường xuyên.
* CSDL NoSQL: Có thể sử dụng MongoDB hoặc Cassandra cho các dữ liệu phi cấu trúc hoặc các dữ liệu cần khả năng mở rộng linh hoạt.

# Use-Case View

## Tổng quát

Hệ thống quản lý lớp học tín chỉ được thiết kế để đáp ứng nhu cầu đăng ký học phần, theo dõi tiến trình học tập, và quản lý thông tin học thuật của sinh viên tại một trường đại học. Hệ thống sẽ cho phép sinh viên dễ dàng tìm kiếm và đăng ký các môn học, lớp tín chỉ phù hợp với yêu cầu học tập của họ. Hệ thống bao gồm nhiều dịch vụ nhỏ, độc lập, có thể triển khai riêng biệt và giao tiếp với nhau thông qua API. Sử dụng mô hình Microservices cho backend và mô hình MVC (Model-View-Controller) cho frontend.



*Hinh 7. Use case tổng quát*

Dành cho sinh viên

* Quản lý thông tin cá nhân: Đổi mật khẩu, đề xuất cập nhập thông tin cá nhân
* Xem thông tin học tập: Xem lịch học, xem chương trình khung, tra cứu công nợ, xem kết quả học tập
* Đăng ký học phần: Xem lịch học, huỷ đăng ký học phần
* Thanh toán học phí
* Đăng nhập

1. **Dành cho giảng viên**

* Đề xuất cập nhật thông tin lớp học phần
* Nhập điểm cho sinh viên
* Đăng nhập

1. **Dành cho giáo vụ**

* Quản lý lớp học phần: Mở lớp học phần, Cập nhập thông tin cá nhân
* Quản lý thông tin sinh viên: Đặt lại mật khẩu tài khoản, Cập nhập thông tin sinh viên
* Đăng nhập

**Link sơ đồ chi tiết bằng web-app chúng em thực hiện:** [*diagram*](https://app.diagrams.net/?libs=general;uml#G1S8fkjcpy5NA5ih3cbpZGq4OyIK45jkIs)

## Hiện thực hoá (Use-Case Realizations)

* **Đăng nhập**

A screenshot of a video game

Description automatically generated

*Hinh 8. Use case đăng nhập*

* Mô tả tóm tắt: Người dùng đăng nhập và hệ thống.
* **Đăng xuất**

A screenshot of a video game

Description automatically generated

*Hinh 9. Use case đăng xuất*

* Mô tả tóm tắt: Người dùng đăng xuất ra hệ thống.
* **Xem thông tin cá nhân: xem chương trình khung, xem lịch học, tra cứu công nợ, xem kết quả học tập, thanh toán học phí**

A group of white ovals with black text

Description automatically generated

*Hinh 10. Use case xem thông tin cá nhân*

* Mô tả tóm tắt: Sinh viên xem thông tin cá nhân của mình sau khi đăng nhập. Các thông tin có thể xem: kết quả học, chương trình khung theo khoá và ngành của bản thân, lịch học, tra cứu công nợ. Ngoài ra sinh viên còn có thể thanh toán trực tuyến.
* **Quản lý thông tin cá nhân: đổi mật khẩu, đề xuất cập nhập thông tin cá nhân**

A close-up of a computer screen

Description automatically generated

*Hinh 11. Case quản lý thông tin cá nhân*

* Mô tả tóm tắt: Sinh viên quản trị đối với tài khoản cá nhân bản thân: đổi mật khẩu và đề xuất cập nhật thông tin cá nhân.
* **Quản lý đăng ký học phần: đăng ký học phần, huỷ đăng ký học phần. Thanh toán học phí**

A screenshot of a computer screen

Description automatically generated

*Hinh 12 Use case quản lý đăng ký học phần*

* Mô tả tóm tắt: Sinh viên có thể đăng ký học phần và huỷ đăng ký học phần ở kì chuẩn bị theo quy định của nhà trường.
* **Xem thông tin lớp học phần: đề xuất cập nhập thông tin lớp học phần, nhập điểm cho sinh viên**

A screenshot of a computer screen

Description automatically generated

*Hinh 13. Use case xem thông tin lớp học phần*

* Mô tả tóm tắt: Giảng viên có thể nhập điểm các sinh viên hoặc đề xuất cập nhật thông tin lớp học phần mà bản thân đang giảng dạy.
* **Quản lý lớp học phần: cập nhật thông tin lớp học phần, mở lớp học phần**

A screenshot of a computer screen

Description automatically generated

*Hinh 14. Use case quản lý lớp học phần*

* Mô tả tóm tắt: Giáo vụ có quyền mở lớp mới hoặ cập nhật thông tin lớp học phần.
* **Quản lý tài khoản: thêm tài khoản, cập nhật thông tin tài khoản, đặt lại mật khẩu tài khẩu**

A screenshot of a computer screen

Description automatically generated

Hinh 15. Use case quản lý tài khoản

* Mô tả tóm tắt: Giáo vụ có quyền thêm tài khoản đăng nhập mới, đặt lại mật khẩu tài khoản (không xem mật khẩu hiện tại), cập nhật thông tin tài khoản.

# Quan điểm logic (Logical View)

## Tổng quan Frontend

Tổng quan logic của “Hệ thống đăng ký học phần” được tạo thành từ ba gói chính:

A diagram of a model

Description automatically generated

*Hinh 16. Kiến trúc fontend*

* **Model**: bao gồm các lớp kiểm soát trực tiếp dữ liệu, logic và quy tắc của Hệ thống Trực tuyến và được hiển thị trong dạng xem.
* **View**: Phần này bao gồm các lớp tạo ra các biểu diễn thông tin đầu ra cho người dùng dựa trên các bản cập nhật mô hình.
* **Controller**: bao gồm các lớp có thể gửi lệnh đến mô hình để thay đổi trạng thái của mô hình, như cũng như các lệnh tới chế độ xem liên quan của nó để sửa đổi cách trình bày của mô hình.

## Tổng quan backend

A diagram of a cloud service

Description automatically generated*Hinh 17. Mô hình mircoservices*

* API Gateway: Đóng vai trò là cổng vào duy nhất, quản lý và tối ưu hóa giao tiếp giữa client và các dịch vụ backend.
* Các Service Con (Microservices): Độc lập, tự quản lý và dễ dàng mở rộng, cung cấp các chức năng cụ thể.
* Eureka: Cung cấp khả năng tự động phát hiện và đăng ký dịch vụ, hỗ trợ cân bằng tải.
* Docker*:* Tạo ra môi trường nhất quán cho phát triển và triển khai, cải thiện tính di động và hiệu quả tài nguyên*.* A screenshot of a computer

  Description automatically generated A screenshot of a computer program

  Description automatically generated

*Hinh 18. Package backend và package của một service*

Các packages được ánh xạ thành các tầng của kiến trúc như sau:

* Gói repositories chứa các lớp của tầng Data Access.
* Gói services chứa các lớp của tầng Business Logic.
* Gói controllers chứa các lớp của tầng Presentation (sử dụng REST API).

Các gói phụ trợ khác gồm có:

* Gói models là nơi chứa các entities. Các lớp này nhận nhiệm vụ ánh xạ các bảng trong cơ sở dữ liệu đồng thời cũng là các đối tượng mang dữ liệu trao đổi giữa các tầng.
* Gói configs chứa các lớp dùng cho việc cấu hình ứng dụng.
* Gói enums chứa các tập hằng số cho các đối tượng. Trong ví dụ này, đó là hai tập hằng số chỉ các trạng thái lớp học(sinhvien), lớp học phần (lopHocPhan),…
* Gói pks quản lý các entity có khóa phức hợp, đảm bảo tính toàn vẹn dữ liệu và hỗ trợ các thao tác CRUD cho các entity này.
* Gói dto đóng gói và truyền dữ liệu giữa các lớp hoặc dịch vụ, tách biệt dữ liệu khỏi logic kinh doanh và cung cấp cấu trúc rõ ràng cho dữ liệu được truyền.

## Mô hình phân tích

A computer screen shot of a computer

Description automatically generated

*Hinh 19. Mô hình ERD*

# Hiện thực hoá (Deployment View)

## Sơ đồ triển khai

A screenshot of a computer

Description automatically generated

*Hinh 20. Sơ đồ triển khai*

1. **Inter-Process Communication (IPC)**

**Mục đích:** Cho phép các tiến trình giao tiếp và chia sẻ tài nguyên với nhau.Các tiến trình này có thể đang chạy trên cùng một máy tính hoặc các máy tính khác nhau trên cùng một mang giúp tăng hiệu suất, khả năng mở rộng, và bảo mật của hệ thống.

**Lợi ích:**

* Chia sẻ tài nguyên: IPC cho phép các tiến trình chia sẻ tài nguyên như bộ nhớ, dữ liệu và thiết bị phần cứng.
* Tăng hiệu suất: Các tiến trình có thể hoạt động song song và giao tiếp với nhau, giúp tăng hiệu suất và khả năng đáp ứng của hệ thống.
* Tăng cường khả năng mở rộng: Hệ thống có thể mở rộng dễ dàng bằng cách thêm các tiến trình mới mà không cần phải thay đổi các tiến trình hiện có.
* Tính bảo mật: IPC cung cấp các cơ chế để bảo mật dữ liệu khi truyền giữa các tiến trình, ngăn chặn truy cập trái phép.

1. **Service Discovery using Netflix Eureka**

**Mục đích:** Netflix Eureka là một dịch vụ phát hiện (service discovery) cho các ứng dụng microservices. Nó cho phép các dịch vụ trong hệ thống tự động phát hiện và giao tiếp với nhau mà không cần cấu hình thủ công các điểm cuối (endpoints).

**Lợi ích:**

* Tự động phát hiện dịch vụ: Eureka giúp các dịch vụ tự động tìm thấy nhau mà không cần cấu hình thủ công, giảm thiểu rủi ro lỗi cấu hình.
* Cân bằng tải động: Với thông tin về các dịch vụ khả dụng, hệ thống có thể phân phối yêu cầu đến các dịch vụ một cách đồng đều, giúp cân bằng tải.
* Khả năng chịu lỗi: Eureka hỗ trợ các cơ chế để dịch vụ phát hiện và loại bỏ các dịch vụ không khả dụng, cải thiện khả năng chịu lỗi của hệ thống.
* Mở rộng dễ dàng: Khi thêm các dịch vụ mới vào hệ thống, chúng tự động đăng ký với Eureka và trở nên khả dụng ngay lập tức mà không cần thay đổi cấu hình của các dịch vụ khác.

**Cách hoạt động của Netflix Eureka:**

* Eureka Server: Trung tâm đăng ký và quản lý các dịch vụ. Các dịch vụ sẽ đăng ký và gửi thông tin tình trạng hoạt động đến đây.
* Eureka Client: Mỗi dịch vụ trong hệ thống có một Eureka Client để đăng ký bản thân với Eureka Server và để lấy thông tin về các dịch vụ khác khi cần giao tiếp.

1. **Implement API Gateway using Spring Cloud Gateway**

**Mục đích:**Quản lý và điều hướng yêu cầu: API Gateway nhận tất cả các yêu cầu từ client và định tuyến chúng đến các dịch vụ backend phù hợp.

* + Tích hợp dịch vụ: Nó cung cấp một điểm duy nhất để tích hợp và tương tác với nhiều dịch vụ backend khác nhau.
  + Bảo mật: API Gateway có thể xử lý xác thực, ủy quyền và các chính sách bảo mật khác.
  + Giảm tải: Nó có thể thực hiện caching, nén và tối ưu hóa lưu lượng để giảm tải cho các dịch vụ backend.
  + Kiểm soát lưu lượng: Quản lý lưu lượng truy cập, hạn chế tần suất và bảo vệ dịch vụ khỏi các tấn công DDoS.

**Lợi ích:**

* + Dễ dàng cấu hình và mở rộng: Spring Cloud Gateway tích hợp chặt chẽ với hệ sinh thái Spring, dễ dàng cấu hình và mở rộng theo nhu cầu.
  + Hỗ trợ các tính năng mạnh mẽ: Cung cấp nhiều tính năng sẵn có như routing, filtering, và các chính sách bảo mật.
  + Hiệu suất cao: Xây dựng trên nền tảng Spring WebFlux, nó hỗ trợ lập trình phản ứng (reactive programming) và có khả năng xử lý lượng lớn yêu cầu đồng thời với hiệu suất cao.
  + Quản lý tập trung: Tất cả các yêu cầu API được quản lý và kiểm soát từ một điểm duy nhất, dễ dàng theo dõi và giám sát.
  + Tích hợp dễ dàng: Dễ dàng tích hợp với các dịch vụ khác trong hệ sinh thái Spring Cloud như Eureka, Config Server, và Sleuth.

**Cách hoạt động**: Spring Cloud Gateway hoạt động như một bộ reverse proxy để điều hướng các yêu cầu từ client đến các dịch vụ backend. Dưới đây là quy trình hoạt động cơ bản:

* Client Request: Một yêu cầu từ client được gửi đến API Gateway.
* Routing: API Gateway sử dụng các định nghĩa route để quyết định yêu cầu nào sẽ được chuyển tiếp đến dịch vụ backend nào. Các route được xác định bởi các predicates (điều kiện) và có thể được định nghĩa trong file cấu hình hoặc qua code.
* Filters: Trước khi yêu cầu được chuyển tiếp, nó có thể đi qua một hoặc nhiều bộ lọc (filters). Bộ lọc có thể thay đổi yêu cầu, thêm các header, thực hiện xác thực, ghi log, v.v.
* Forwarding: Yêu cầu sau khi được xử lý sẽ được chuyển tiếp đến dịch vụ backend phù hợp.
* Response Handling: Kết quả từ dịch vụ backend được trả về API Gateway, nơi nó có thể đi qua thêm các bộ lọc phản hồi trước khi được trả về client.

1. **Security Microservices using JWT(Json web token)**

**Mục Tiêu**

* Bảo Mật: Đảm bảo rằng chỉ có những người dùng đã được xác thực mới có thể truy cập các tài nguyên bảo mật của microservices.
* Xác Thực và Ủy Quyền: Sử dụng JWT (JSON Web Token) để xác thực người dùng và ủy quyền cho họ truy cập các dịch vụ.
* Độc Lập và Tích Hợp: Đảm bảo các microservices có thể hoạt động độc lập nhưng vẫn có thể tích hợp dễ dàng với hệ thống xác thực tập trung.

**Lợi Ích**

* Tính Di Động: JWT là một tiêu chuẩn mở và có thể được sử dụng trong nhiều ngôn ngữ và nền tảng khác nhau.
* Không Trạng Thái: JWT không yêu cầu lưu trữ trạng thái phía server, giúp giảm tải và tăng hiệu suất.
* An Toàn: JWT được ký số để đảm bảo tính toàn vẹn và xác thực, có thể mã hóa để bảo vệ thông tin nhạy cảm.
* Tính Linh Hoạt: Dễ dàng tùy chỉnh và mở rộng, có thể chứa các thông tin khác nhau tùy thuộc vào yêu cầu của hệ thống.
* Dễ Dàng Tích Hợp: JWT có thể dễ dàng tích hợp với các dịch vụ API Gateway, giúp bảo vệ toàn bộ hệ thống microservices.

**Cách Hoạt Động**

* Xác Thực Người Dùng: Người dùng gửi yêu cầu đăng nhập kèm theo thông tin xác thực (ví dụ: tên đăng nhập và mật khẩu) đến dịch vụ xác thực. Dịch vụ xác thực kiểm tra thông tin và, nếu hợp lệ, tạo ra một JWT chứa thông tin người dùng và quyền hạn.
* Tạo và Trả Về JWT: JWT được ký số bằng khóa bí mật của server để đảm bảo tính toàn vẹn, sau đó được trả về cho người dùng và lưu trữ phía client (thường là trong local storage hoặc cookies).
* Gửi Yêu Cầu Kèm JWT: Khi người dùng muốn truy cập vào một microservice bảo mật, họ gửi yêu cầu kèm theo JWT trong header của HTTP request (thường là Authorization: Bearer <JWT>).
* Xác Minh JWT: Microservice nhận yêu cầu và kiểm tra JWT, được giải mã và kiểm tra tính hợp lệ (chữ ký số, thời hạn sử dụng, v.v.).
* Nếu JWT hợp lệ, thông tin người dùng và quyền hạn được trích xuất từ JWT.
* Ủy Quyền: Microservice kiểm tra quyền hạn của người dùng dựa trên thông tin trong JWT. Nếu người dùng có quyền truy cập, yêu cầu được xử lý và phản hồi tương ứng. Nếu không, trả về lỗi ủy quyền (thường là HTTP 403 Forbidden).

1. **Implement Circuit Breaker**

**Circuit Breaker** (bộ ngắt mạch) là một mẫu thiết kế được sử dụng trong kiến trúc microservices để tăng độ tin cậy và khả năng chịu lỗi của hệ thống. Nó giúp ngăn chặn các lỗi lan rộng trong hệ thống bằng cách theo dõi các cuộc gọi đến các dịch vụ phụ thuộc và "ngắt mạch" nếu một dịch vụ nào đó không phản hồi đúng cách trong một khoảng thời gian nhất định.

**Lợi ích của Circuit Breaker*:***

* + Tăng cường độ tin cậy: Ngăn chặn các lỗi lan rộng trong hệ thống bằng cách cô lập các dịch vụ bị lỗi.
  + Giảm thiểu thời gian chờ: Tránh lãng phí tài nguyên hệ thống cho các yêu cầu đến các dịch vụ không khả dụng.
  + Cải thiện trải nghiệm người dùng: Cung cấp phản hồi nhanh chóng khi dịch vụ phụ thuộc không khả dụng, thay vì để người dùng phải chờ đợi.

**Circuit Breaker hoạt động dựa trên ba trạng thái chính:**

* + **Closed (Đóng)**:
    - Trong trạng thái này, các yêu cầu được gửi đi bình thường.
    - Nếu số lượng lỗi vượt quá ngưỡng quy định trong một khoảng thời gian cụ thể, Circuit Breaker chuyển sang trạng thái Open.
  + **Open (Mở)**:
    - Trong trạng thái này, các yêu cầu tiếp theo sẽ ngay lập tức bị từ chối hoặc nhận được một phản hồi thay thế mà không cần gửi tới dịch vụ phụ thuộc.
    - Circuit Breaker ở trạng thái này trong một khoảng thời gian nhất định trước khi chuyển sang trạng thái Half-Open.
  + **Half-Open (Nửa Mở)**:
    - Trong trạng thái này, một số yêu cầu được phép đi qua để kiểm tra xem dịch vụ phụ thuộc đã phục hồi hay chưa.
    - Nếu những yêu cầu thử nghiệm này thành công, Circuit Breaker chuyển lại trạng thái Closed. Nếu thất bại, nó quay lại trạng thái Open.

1. **Event Driven Architecture using Kafka**

**Event-Driven Architecture (EDA)** là một phong cách kiến trúc phần mềm trong đó các hành động và trạng thái của hệ thống được truyền tải thông qua các sự kiện. Kafka, một nền tảng truyền tải luồng dữ liệu phân tán, là một trong những công nghệ phổ biến nhất được sử dụng để thực hiện EDA.

**Tổng quan về Kafka**

Apache Kafka là một nền tảng truyền tải dữ liệu luồng mạnh mẽ và mở rộng, được thiết kế để xử lý các luồng dữ liệu thời gian thực. Nó cung cấp các tính năng sau:

* + **Durability**: Lưu trữ sự kiện một cách đáng tin cậy.
  + **Scalability**: Dễ dàng mở rộng quy mô xử lý dữ liệu khi khối lượng tăng lên.
  + **High-throughput**: Khả năng xử lý một lượng lớn dữ liệu với độ trễ thấp.
  + **Fault-tolerance**: Khả năng chịu lỗi và duy trì hoạt động ổn định trong các tình huống gặp sự cố.

Các thành phần chính trong Kafka*:*

* + **Producer**: Gửi dữ liệu (sự kiện) đến các chủ đề (topics).
  + **Consumer**: Nhận và xử lý dữ liệu từ các chủ đề.
  + **Broker**: Máy chủ Kafka quản lý lưu trữ và truyền tải các sự kiện.
  + **Topic**: Kênh logic nơi các sự kiện được lưu trữ.
  + **Partition**: Mỗi chủ đề được chia thành nhiều phân vùng để tăng cường khả năng mở rộng và hiệu suất.
  + **Zookeeper**: Được sử dụng để quản lý và điều phối Kafka, đảm bảo tính nhất quán và quản lý cấu hình.

**Quy trình hoạt động cơ bản:**

1. **Sản xuất sự kiện (Event Production)**:
   * Các dịch vụ hoặc ứng dụng phát sinh ra các sự kiện (event) khi một hành động xảy ra. Ví dụ, khi một đơn hàng được đặt, một sự kiện "OrderCreated" sẽ được gửi đến Kafka.
2. **Chủ đề (Topics)**:
   * Các sự kiện được gửi tới các chủ đề cụ thể trong Kafka. Mỗi loại sự kiện có thể có một hoặc nhiều chủ đề liên quan. Ví dụ, các sự kiện liên quan đến đơn hàng có thể được gửi tới chủ đề "orders".
3. **Tiêu thụ sự kiện (Event Consumption)**:
   * Các dịch vụ khác lắng nghe và xử lý các sự kiện từ các chủ đề mà chúng quan tâm. Ví dụ, một dịch vụ xử lý thanh toán sẽ lắng nghe các sự kiện từ chủ đề "orders" để bắt đầu quá trình thanh toán khi nhận được sự kiện "OrderCreated".
4. **Quản lý trạng thái và tương tác (State Management and Interaction)**:
   * Các dịch vụ tiêu thụ sự kiện có thể cập nhật trạng thái của chúng và phát sinh các sự kiện mới dựa trên kết quả xử lý. Ví dụ, sau khi xử lý thanh toán thành công, dịch vụ có thể phát sinh sự kiện "PaymentProcessed".
5. **Dockerize the application**

**Dockerize the application** là quá trình tạo một Docker container cho ứng dụng . Docker container là một môi trường độc lập và nhẹ, bao gồm tất cả các thành phần cần thiết để chạy ứng dụng, bao gồm cả mã nguồn, runtime, thư viện, và các phụ thuộc khác. Điều này giúp ứng dụng của bạn trở nên nhất quán và dễ dàng triển khai trên nhiều môi trường khác nhau.

**Lợi ích của Dockerizing an Application**

1. **Portability**: Docker containers có thể chạy trên bất kỳ máy chủ nào có cài đặt Docker, bất kể hệ điều hành hoặc môi trường phát triển.
2. **Isolation**: Mỗi container chạy trong một môi trường riêng biệt, giúp tránh xung đột về thư viện hoặc cấu hình giữa các ứng dụng.
3. **Consistency**: Môi trường chạy của ứng dụng luôn được giữ nguyên, bất kể nơi nó được triển khai (development, staging, production).
4. **Scalability**: Dễ dàng mở rộng ứng dụng bằng cách triển khai nhiều instance của container.

# Data View

## Tổng quan

A screenshot of a computer screen

Description automatically generated

*Hinh 21 Tổng quan data view*

# Process View

1. **Sơ đồ Activity**

**Đăng nhập**

A diagram of a diagram

Description automatically generated with medium confidence

*Hinh 22. Activity Đăng nhập*

**Đăng xuất**A screenshot of a computer screen

Description automatically generated

*Hinh 23. Activity đăng xuất*

**Xem thông tin sinh viên**

A diagram of a diagram

Description automatically generated

*Hinh 24. Activity xem thông tin sinh viên*

**Đăng ký học phần**

**Hủy đăng ký học phần**

A screenshot of a diagram

Description automatically generated

*Hinh 25. Activity Huỷ đăng ký học phần*

**Đổi mật khẩu**  
A screenshot of a diagram

Description automatically generated

*Hinh 26. Activity đổi mật khẩu*

**Cập nhật thông tin cá nhân**

A screenshot of a diagram

Description automatically generated

*Hinh 27. Activity cập nhật thông tin cá nhân*

**Thanh toán trực tuyến**

A screenshot of a chat

Description automatically generated

*Hinh 28. Activity thanh toán trực tuyến*

# Kích thước và hiệu suất

Hệ thống đăng ký học phần là trang website trực tuyến sẽ được sử dụng tại trường học và kho dữ liệu lưu trữ sẽ dự tính đặt tại trường. Người sử dụng hệ thống sẽ không cần phải cài đặt bất kỳ thành phần nào.

* Theo như nghiệp vụ thì hệ thống sẽ có lượt truy cập ít vào các thời điểm bình thường nhưng sẽ phải chịu áp lực lớn khi trong giai đoạn bắt đầu mở đăng ký. Do lượng sinh viên truy cập vào đăng ký cùng một lúc sẽ dễ
* Mặc dù không thể đo được kích thước của hệ thống nhưng ước tính lên tới hàng chục terabyte (TB). Kích thước cụ thể sẽ bao gồm dữ liệu về sinh viên, môn học, lớp học, giảng viên, và các giao dịch đăng ký học phần.
* Mọi thao tác phải được hệ thống thực hiện nhanh chóng, chính xác và các thông tin thanh toán học phí, đăng ký học phần, lưu điểm phải được lưu chính xác. Các truy vấn và cập nhật cơ sở dữ liệu phải được tối ưu hóa để xử lý số lượng lớn yêu cầu đồng thời.

Thường thì một kì sinh viên sẽ phải đăng ký trung bình là 5 môn. Số lượng truy cập đồng thời cao nhất rơi vào khoảng 28 nghìn người. => Vậy là hệ thống cần xử lý 140,000 giao dịch đăng ký trong khoảng thời gian ngắn. Vậy thì hệ thống đăng ký học phần muốn đạt hiệu quả cao thì cần đáp ứng đủ nhu cầu trên mà không gây ra sự cố về hiệu xuất.

A screenshot of a computer

Description automatically generated

Hinh 29. *Ảnh thực tế trang web đăng ký học phần trường iuh vào lúc dkhp*

# Chất lượng

Kiến trúc hệ thống tuân thủ các yêu cầu chất lượng sau:

* Hệ thống có thể đảm bảo rằng tất cả các chuyển đổi thông tin đều được lưu lại để duy trì mức độ bảo mật cao nhất tính toàn vẹn của hệ thống.
* Để đảm bảo an toàn dữ liệu, cơ sở dữ liệu sẽ được sao lưu hàng ngày.
* Trang web hệ thống có khả năng hiển thị chính xác trên bất kỳ thiết bị nào có trình duyệt web, bất kể kích thước màn hình (ví dụ: thiết kế đáp ứng). Ứng dụng hệ thống tương thích với nhiều trình duyệt phổ biến như: Google Chrome, Cốc Cốc, Microsoft Edge, Safari, Opera, UC Browser.