**MẪU BÁO CÁO TIỂU LUẬN TỐT NGHIỆP**

**HÌNH THỨC TRÌNH BÀY**

* Tiểu luận phải được trình bày ngắn gọn, rõ ràng, mạch lạc, sạch sẽ, không được tẩy xóa, có đánh số trang, đánh số bảng biểu, hình vẽ, đồ thị.
* Tiêu đề chính (Mục lục, các chương…) bảng mã Unicode, Font chữ Times New Roman, size 18pt
* Chữ thường (normal): bảng mã Unicode, Font chữ Times New Roman, kích thước(size) 13pt
* Dãn dòng (line spacing) đặt ở chế độ 1.3-1.5 lines.
* Lề trên 2.7 cm, lề dưới 2.7 cm, lề trái 3.4 cm, lề phải 2 cm.
* Đánh số trang ở giữa bên dưới :

+ Vị trí: canh giữa

+ Kiểu số Times New Roman, kích thước 12

+ Các trang đầu đánh theo số La Mã

* Các bảng biểu trình bày theo chiều ngang khổ giấy thì đầu bảng là lề trái của trang.
* Số thứ tự của các chương, mục được đánh số bằng hệ thống số Ả-rập, không dùng số La mã. Các mục và tiểu mục được đánh số bằng các nhóm hai hoặc ba chữ số, cách nhau một dấu chấm: số thứ nhất chỉ số chương, chỉ số thứ hai chỉ số mục, số thứ ba chỉ số tiểu mục. Ví dụ:

**Chương 3 …**

**3.1. …**

**3.1.1. …**

**3.1.2. …**

**3.2. …**

Trong tiểu luận không được phép có 2 chương mục có cùng chỉ số

* Tiểu luận bản cuối cùng đã hiệu chỉnh sau khi bảo vệ phải đóng bìa cứng, in chữ nhũ vàng đủ dấu tiếng Việt trên bìa xanh nước biển (theo mẫu). Họ và tên các SV được ghi theo mã số sinh viên
* Qui ước ghi tài liệu tham khảo:
  + Bài đăng tạp chí: Tên tác giả, tên bài báo, tên tạp chí, tập, số, năm và các trang.
  + Ví dụ: Kumar S, Superconvergence of a ..., IMA Journal of Numerial Analysis (7), 1987, pp. 313 - 325.
  + Bài báo cáo hội nghị: Tên tác giả, Tên bài báo, Tên hội nghị, tên tuyển tập các báo cáo, nơi và thời gian tổ chức. Ví dụ: B.K. Paradopop, Fuzzy sets and fuzzy realational structures as Chu spaces, Proceedings of the First International Workshop on ..., Thessaloniki, Greece, Oct. 16-20, 1998.
  + Sách: Tên tác giả, tên sách, lần xuất bản, nhà xuất bản, nơi xuất bản, năm xuất bản. Ví dụ: A.N.Tikhonov, Solutions of Ill-Posed Problems, Willey, NewYork, 1997.

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC NÔNG LÂM TP HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**TIỂU LUẬN TỐT NGHIỆP**

**ỨNG DỤNG CÁC CÔNG NGHỆ REACT, NESTJS VÀ KUBERNETES TRONG KIẾN TRÚC MICROSERVICE ĐỂ XÂY DỰNG SÀN THƯƠNG MẠI ĐIỆN TỬ**

**Ngành : Công nghệ thông tin**

**Niên khoá : 2019 - 2023**

**Lớp : DH19DT**

**Sinh viên thực hiện : 1. Châu Văn Lộc MSSV: 19130121**

**2. Cao Huy Tấn Lộc MSSV: 19130120**

TP.HỒ CHÍ MINH, tháng năm

Ư/2014

LỜI CAM ĐOAN

Chúng tôi xin cam đoan đề tài “ỨNG DỤNG CÁC CÔNG NGHỆ REACT, NESTJS VÀ KUBERNETES TRONG KIẾN TRÚC MICROSERVICE ĐỂ XÂY DỰNG SÀN THƯƠNG MẠI ĐIỆN TỬ” là một công trình nghiên cứu độc lập, không sao chép các đề tài khác.

Đề tài là một sản phẩm do nhóm đã nỗ lực nghiên cứu, trong bài có sự tham khảo của một số tài liệu có nguồn gốc rõ ràng. Đề tài được nghiên cứu đảm bảo theo đúng mục tiêu đã được thuyết minh trước Hội đồng Khoa học nhà trường. Nhóm nghiên cứu cam đoan chịu hoàn toàn trách nhiệm về bản quyền.

Sinh viên thực hiện 1 Sinh viên thực hiện 2

Châu Văn Lộc Cao Huy Tấn Lộc

LỜI CẢM ƠN

Lời nói đầu, chúng em xin gửi lời cảm ơn chân thành đến quý Thầy, Cô Khoa Công nghệ thông tin, Trường Đại học Nông Lâm Thành phố Hồ Chí Minh đã truyền dạy kiến thức trong suốt thời gian học tập tại trường. Đây là hành trang không thể thiếu giúp chúng em hoàn thành khóa luận và tự tin lập nghiệp sau này.

Xin tỏ lòng biết ơn và gửi lời tri ân sâu sắc đến Ts. Nguyễn Thị Phương Trâm đã luôn kiên nhẫn, chỉ bảo, hướng dẫn chúng em trong suốt quá trình thực hiện khóa luận tốt nghiệp.

Xin chân thành cảm ơn!

DANH SÁCH CHỮ VIẾT TẮT

ANSI **A**merican **N**ational **S**tandards **I**nstitute

Học viện chuẩn hoá quốc gia Mỹ

ATM **A**utomatic **T**eller **M**achine

Máy rút tiền tự động

CA **C**ertification **A**uthority

Chứng thực

ISO **I**nternational **S**tandardizations **O**rganization

Tổ chức chuẩn hoá quốc tế

MAC **M**essage **A**uthentication **C**ode

Mã chứng thực thông điệp

UUCP **U**nix-to-**U**nix **C**opy **P**rotocol

Giao thức truyền nhận thông điệp trên Unix

DANH MỤC HÌNH ẢNH

[Hình 2.1 Mạch chỉnh lưu và bộ lọc **Error! Bookmark not defined.**](#_Toc98229800)

**DANH MỤC BẢNG**

Bảng 1.1: Tên bảng 19

**TÓM TẮT**

**MỤC LỤC**

MỞ ĐẦU

## LÝ DO CHỌN ĐỀ TÀI

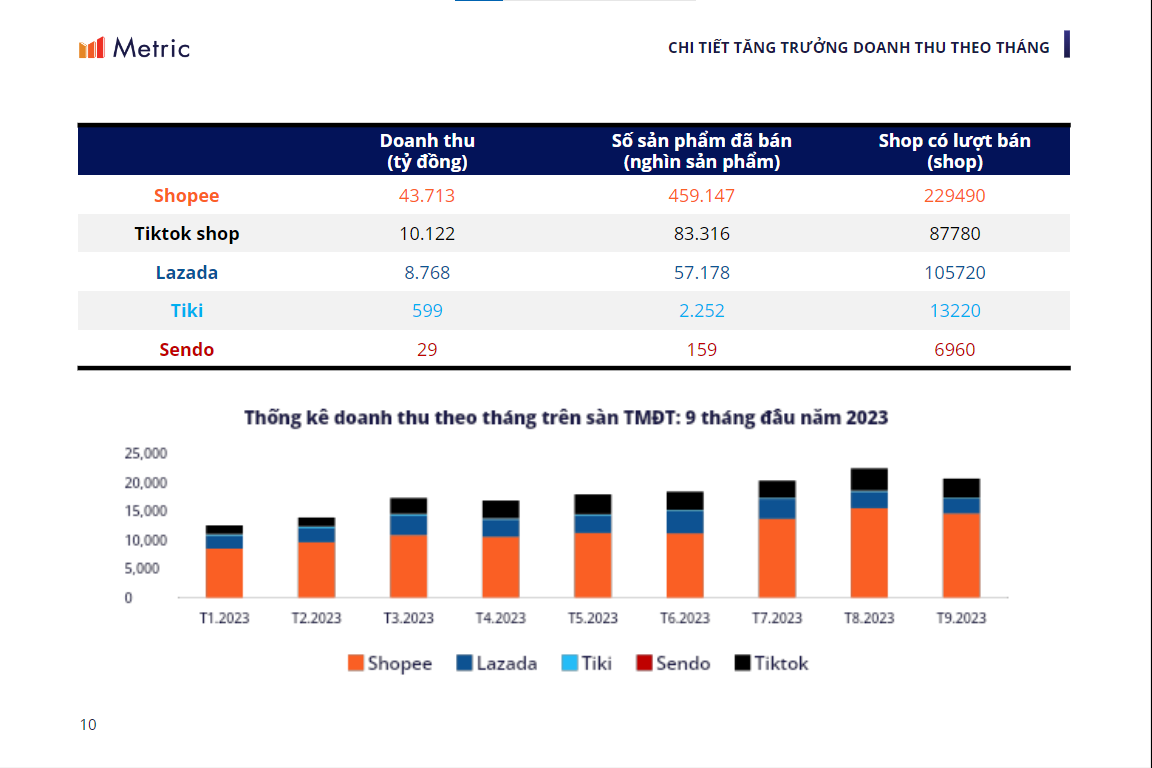
Thương mại điện tử đã và đang là xu hướng kinh doanh nằm trong top tăng trưởng trên thế giới, với quy mô và phạm vi trên gần như toàn bộ các lĩnh vực trong thương mại. Cuộc cách mạng về công nghệ thông tin đã làm thay đổi cuộc chơi về thương mại nói riêng cũng như các lĩnh vực khác nói chung.

Khoảng những năm 2000 – 2010, việc sở hữu một thiết bị kết nối mạng internet đã bắt đầu phổ biến, với những chiếc máy tính cá nhân có kết nối mạng internet, đến những chiếc laptop nhỏ gọn, tiện lợi. Qua đến giai đoạn từ 2010 đến nay, sự bùng nổ của các thiết bị di động cầm tay tiện lợi, và sự thay đổi chóng mặt của công nghệ internet giúp việc kết nối trở nên thuận tiện hơn, luôn sẵn sàng mọi lúc mọi nơi.

Với nền tảng công nghệ thông tin phát triển cộng với việc các thiết bị kết nối mạng trở nên phổ biến trong khoảng 2 thập kỉ vừa qua đã tác động tích cực lên ngành bán lẻ, giúp thay đổi thói quen của người tiêu dùng. Người tiêu dùng đã bắt đầu hình thành thói quen mua hàng trực tuyến trên các trang bán hàng điện tử hoặc các sàn thương mại điện tử. Nguyên nhân dẫn đến sự thay đổi này có thể được tóm gọn trong các ý sau

Thứ nhất, các trang thương mại điện tử đã xuất hiện rất nhiều, đa dạng sản phẩm, đa dạng thương hiệu và luôn có những ưu đãi hấp dẫn, kích thích người mua hàng chi tiêu. Thứ hai, khách hàng có thể mua hàng xuyên biên giới mà không cần phải đi đến tận nơi mua, chỉ cần click là đã có thể mua hàng, rất nhanh chóng và tiện lợi. Thứ ba, đối với các người bán, việc kinh doanh thương mại điện tử giúp bài toán chi phí của họ trở nên tối ưu, giảm các chi phí lớn như chi phí mặt bằng, chi phí nhân viên. Chỉ với một nguồn hàng ổn định, một kho lưu trữ và một hệ thống bán hàng hiệu quả, người bán đã có thể mang sản phẩm đến tận tay người tiêu dùng với chi phí thấp, giúp giá thành sản phẩm trở nên cạnh tranh hơn.

Hiện nay, không khó để tìm thấy một sàn thương mại điện tử trên internet. Shopee, Lazada hay Tiktok shop ngay chính tại Việt Nam chính là ví dụ điển hình cho sự phát triển của ngành thương mại điện tử tại Việt Nam. Theo Metric, Tổng doanh thu của toàn thị trường TMĐT trong quý III/2023 đạt 63 nghìn tỷ đồng và đạt mức tăng trưởng 54,42% so với cùng kỳ năm ngoái. Năm 2023, tổng doanh thu 9 tháng đầu tiên đã cao hơn cả năm 2022 khoảng 10 nghìn tỷ đồng, tương đương 7%. Trong đó, Tiktok Shop mang về 25 nghìn tỷ đồng. Nếu chỉ tính trên doanh thu 4 sàn Shopee, Lazada, Tiki, Sendo (Tikok Shop ra mắt vào cuối tháng 4/2022), tỷ lệ tăng trưởng so với cùng kỳ là 33%. Đây là các con số rất ấn tượng, chứng minh rằng thị trường thương mại điện tử vẫn đang tăng trưởng rất mạnh tại Việt Nam [1] . Điều này chứng minh cho xu hướng không thể phủ nhận – Dịch chuyển từ thương mại truyền thống sang thương mại điện tử.



Hình 1: Chi tiết tăng trưởng doanh thu theo tháng của các sàn thương mại điện tử tại Việt Nam – 9 tháng đầu năm 2023

(Nguồn: <https://advertisingvietnam.com/metric-cong-bo-bao-cao-thi-truong-san-tmdt-quy-iii2023-p23110>)

Dù hiện tại, thương mại điện tử vẫn đang phát triển và đã đạt được nhiều bước tiến lớn, vẫn còn tồn đọng rất nhiều vấn đề khi tăng trưởng quá nhanh. Đầu tiên, khi lượng khách hàng tăng cao, vấn đề đảm bảo khả năng sử dụng khi có một lượng lớn khách hàng truy cập vào cùng một thời điểm vẫn còn nhiều hạn chế. Bên cạnh đó, các yếu tố về xây dựng hệ thống dữ liệu, kiến trúc hệ thống làm sao để đảm bảo tốc độ thực thi, sự chính xác của dữ liệu và đưa ra các con số thống kê cũng là một thách thức lớn.

Để giải quyết những vấn đề đó, chúng em đã bắt tay vào nghiên cứu giải pháp để làm sao xây dựng một sàn thương mại điện tử. Với yêu cầu là đảm bảo hệ thống vận hành trơn tru khi lượng truy cập tăng đột biến, mà vẫn mang lại trải nghiệm tuyệt vời dành cho khách hàng và người bán. Ngoài ra, khả năng mở rộng quy mô của sàn thương mại điện tử khi tăng trưởng và mở rộng cũng là tiêu chí hàng đầu để nhóm tìm hiểu các công nghệ có thể đáp ứng được điều này.

Sau quá trình nghiên cứu, nhóm đã quyết định lựa chọn kiến trúc Microservice để áp dụng vào bài toán xây dựng sàn thương mại điện tử để đáp ứng với các yêu cầu trên. Kiến trúc Microservice Kiến trúc Microservice có rất nhiều ưu điểm khi sử dụng cho hệ thống thương mại điện tử như

* Có thể sử dụng nhiều loại cơ sở dữ liệu.
* Hỗ trợ lập trình đa ngôn ngữ
* Có thể kiểm soát các vấn đề gây ra lỗi, khả năng chịu lỗi và cách ly lỗi mà không ảnh hưởng đến toàn bộ hệ thống.
* Cung cấp khả năng mở rộng và thu nhỏ services mà không ảnh hưởng đến hiệu suất.
* Dễ dàng giám sát an ninh của hệ thống và đảm bảo nhu cầu bảo mật / xác thực API.
* Có thể sử dụng các webservices để cải thiện hiệu suất của hệ thống mà không làm ảnh hưởng đến hoạt động của các services khác.

Kiến trúc Microservices có rất nhiều cải thiện về hiệu suất so với kiến trúc Monolith truyền thống. Theo bài báo “Designing microservice architectures for scalability and reliability in ecommerce” [2]

* Khả năng xử lý của một trang thương mại điện tử sử dụng Microservice đạt hiệu quả hơn 30% khi load script và 4% khi load phần tử hình ảnh.
* Khả năng chịu tải khi stress test với lần lượt 300, 500 và 1000 lần lặp kiểm tra dữ liệu với tối đa 25000 request đồng thời, tỷ lệ xuất hiện lỗi rất thấp (khoảng 3%)
* Những công nghệ hiện tại trong API Gateway, hệ thống điều phối container,... đang hỗ trợ rất tốt khi làm việc với microservices.
* Từ đó, nhóm quyết định chọn đề tài ứng dụng kiến trúc Microservices để xây dựng một sàn thương mại điện tử, tự động mở rộng hoặc thu nhỏ services đảm bảo hệ thống luôn luôn hoạt động ổn định.

Chính nhờ những ưu điểm bên trên, nhóm đã đi đến quyết định thực hiện đề tài “ Ứng dụng các công nghệ React, NestJS và Kubernetes trong kiến trúc Microservices để xây dựng sàn thương mại điện tử ” để hiện thực kết quả từ những nghiên cứu trên.

## MỤC TIÊU VÀ PHẠM VI NGHIÊN CỨU

Nhóm em tập trung vào mục tiêu tập trung vào việc ứng dụng kiến trúc Microservices và các công nghệ như NestJS, Kong API Gateway, Kubernetes và React để xây dựng một sàn thương mại điện tử có khả năng mở rộng và quản lý hiệu quả, tương tự Shopee. Đầu tiên, đề tài sẽ nghiên cứu sâu về kiến trúc Microservices, các khái niệm cơ bản, ưu điểm vượt trội như tính module hóa, khả năng mở rộng linh hoạt, khả năng phục hồi tốt cùng những thách thức đi kèm. Việc nắm vững kiến thức này sẽ tạo nền tảng vững chắc cho quá trình triển khai.

Tiếp theo, đề tài sẽ sử dụng NestJS, một framework Backend dựa trên Node.js, để xây dựng các service của sàn thương mại điện tử. Mỗi service sẽ đảm nhận một chức năng cụ thể như quản lý người dùng, sản phẩm, xử lý đơn hàng, thanh toán. Việc thiết kế các service nhỏ, độc lập sẽ giúp hệ thống dễ bảo trì, nâng cấp và tối ưu hóa tài nguyên.

Đề tài cũng sẽ nghiên cứu, triển khai Kong API Gateway để quản lý các yêu cầu từ client, điều phối tới service phù hợp, đồng thời cung cấp các tính năng bảo mật như xác thực, ủy quyền. Ngoài ra, đề tài sẽ sử dụng các dịch vụ khác như Load Balancer để phân phối lưu lượng truy cập hiệu quả, đảm bảo hệ thống hoạt động ổn định, và Message-queuing để các service giao tiếp không đồng bộ, tăng tính linh hoạt.

Cuối cùng, đề tài sẽ triển khai hệ thống trên Kubernetes, một nền tảng điều phối container, để tự động hóa việc triển khai, mở rộng và quản lý các service. Kubernetes sẽ giúp dễ dàng mở rộng hoặc thu nhỏ các service theo nhu cầu thực tế, đảm bảo hệ thống luôn sẵn sàng đáp ứng người dùng. Phía client, đề tài sẽ sử dụng React để xây dựng giao diện người dùng của sàn thương mại điện tử.

Tóm lại, đề tài tập trung nghiên cứu và ứng dụng kiến trúc Microservices, sử dụng NestJS, Kong API Gateway, Kubernetes và React để xây dựng một sàn thương mại điện tử có khả năng mở rộng, quản lý hiệu quả và đáp ứng nhu cầu người dùng tốt nhất.

## Ý NGHĨA KHOA HỌC VÀ THỰC TIỄN CỦA ĐỀ TÀI

# TỔNG QUAN ĐỀ TÀI

## Phân tích, đánh giá các công trình nghiên cứu đã có liên quan đến đề tài nghiên cứu

## Nêu những vấn đề còn tồn tại

## Chỉ ra những vấn đề mà tiểu luận cần tập trung nghiên cứu, giải quyết

# PHƯƠNG PHÁP VÀ NỘI DUNG NGHIÊN CỨU

## Cơ sở lí thuyết, lí luận, giả thiết khoa học

### Microservices

#### Giới thiệu về Microservices

Microservice là một kỹ thuật phát triển phần mềm, là một biến thể của kiến trúc hướng dịch vụ (SOA - xuất hiện trong những năm 1990).

SOA tập trung vào việc chia ứng dụng thành các dịch vụ nhỏ và độc lập, nhằm tạo sự linh hoạt và tái sử dụng trong phát triển phần mềm. Tuy nhiên, SOA thường gặp khó khăn trong việc triển khai thực tế do sự phức tạp của các dịch vụ lớn và sự phụ thuộc mạnh mẽ giữa các dịch vụ.

Bắt đầu vào khoảng giữa những năm 2000, một số công ty công nghệ lớn như Amazon, Netflix và Twitter đã bắt đầu áp dụng kiến trúc microservice để tăng cường tính mở rộng, khả năng module và triển khai liên tục trong hệ thống của họ. Việc triển khai thành công của những công ty này đã thu hút sự quan tâm và lan rộng kiến trúc microservice sang các lĩnh vực phát triển phần mềm khác. Đến những năm 2010, kiến trúc microservice đã trở nên phổ biến và trở thành một trong những phong cách kiến trúc phổ biến trong công nghiệp phát triển phần mềm. Các công ty và tổ chức phát triển phần mềm khắp thế giới đã bắt đầu chuyển từ kiến trúc monolithic truyền thống sang kiến trúc microservice để tận dụng các lợi ích vượt trội của kiến trúc này trong việc xây dựng các ứng dụng phức tạp và linh hoạt. Cho đến hiện nay, kiến trúc microservice tiếp tục phát triển và tiến xa hơn trong việc tạo ra các hệ thống phần mềm phức tạp, đáp ứng nhanh chóng và hiệu quả yêu cầu của thị trường và người dùng.

Trong kiến trúc microservices, các logic ứng dụng được phân chia thành các phần riêng biệt, được gọi là các microservices. Thay vì phải hiện thực tất cả các logic ứng dụng trong một ứng dụng nguyên khối (monolithic application), mỗi microservice độc lập triển khai một phần cụ thể của ứng dụng hoặc một khả năng kinh doanh. Các microservices có thể được viết bằng nhiều ngôn ngữ lập trình và được triển khai và mở rộng độc lập với nhau. Mỗi microservice hoạt động như một ứng dụng con độc lập, có khả năng giao tiếp với các microservices khác thông qua các giao thức giao tiếp nhẹ nhàng như HTTP, RPC hoặc các hàng đợi tin nhắn. Việc phân chia ứng dụng thành các microservices độc lập giúp cải thiện tính module và tách biệt các chức năng, giúp ứng dụng dễ hiểu, phát triển, kiểm thử và duy trì.



Hình 2.1: Minh họa về kiến trúc Microservices

*(Nguồn:* [*https://microservices.io/i/Microservice\_Architecture.png*](https://microservices.io/i/Microservice_Architecture.png)*)*

#### Những đặc trưng của Microservices

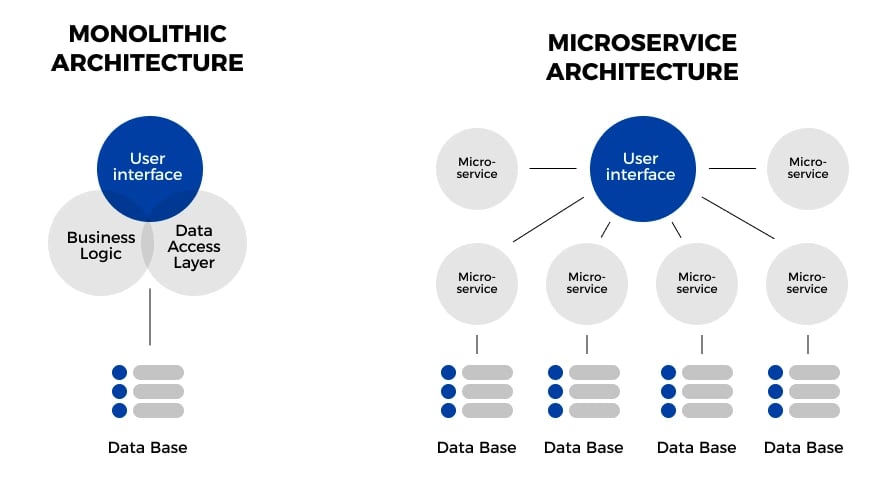
Kiến trúc Microservices cung cấp một cách tiếp cận linh hoạt và hiệu quả để quản lý và phát triển các ứng dụng phức tạp. Đối với các sàn thương mại điện tử, nơi mà số lượng người dùng và các giao dịch có thể lên đến con số rất lớn trong thời gian ngắn, khả năng chịu tải và mở rộng kịp thời là cực kỳ quan trọng. Microservices cho phép từng service trong hệ thống hoạt động độc lập, qua đó:

1. **Mở rộng linh hoạt**: Các services có thể được mở rộng một cách độc lập, giúp tối ưu hóa tài nguyên và quản lý load balancing hiệu quả hơn so với kiến trúc monolithic truyền thống.
2. **Độc lập về Database**: Trong kiến trúc Microservices, mỗi dịch vụ quản lý cơ sở dữ liệu của riêng mình, điều này làm giảm đi sự phụ thuộc lẫn nhau giữa các dịch vụ về mặt dữ liệu và cải thiện hiệu suất và tính ổn định của hệ thống.
3. **Đa ngôn ngữ lập trình**: Hỗ trợ phát triển bằng nhiều ngôn ngữ lập trình khác nhau, giúp tận dụng tối đa sức mạnh và chuyên môn của từng ngôn ngữ để giải quyết các vấn đề cụ thể.
4. **Cách ly và chịu lỗi**: Sự cô lập giữa các services giúp đảm bảo rằng sự cố ở một dịch vụ không làm ảnh hưởng đến toàn bộ hệ thống, qua đó giảm thiểu downtime và mất mát dữ liệu.
5. **Bảo mật và giám sát**: Việc phân tách các dịch vụ cũng giúp dễ dàng triển khai các giải pháp bảo mật tốt hơn, từ việc xác thực người dùng cho đến việc mã hóa dữ liệu.

**Ứng dụng kiến trúc Microservices trong thương mại điện tử đòi hỏi sự đầu tư kỹ lưỡng vào cơ sở hạ tầng công nghệ, đặc biệt là các công nghệ hỗ trợ tính năng mở rộng tự động và quản lý tài nguyên hiệu quả. Việc này bao gồm:**

1. **Load Balancing và API Gateway**: Các giải pháp Load Balancer và API Gateway đóng vai trò quan trọng trong việc phân phối các request một cách đồng đều giữa các dịch vụ, đồng thời cung cấp một điểm truy nhập thống nhất cho các client. API Gateway giúp điều phối các request đến service phù hợp, trong khi đó Load Balancer đảm bảo các service không bị quá tải, cải thiện độ ổn định và hiệu suất của hệ thống.
2. **Message Queuing**: Hệ thống hàng đợi tin nhắn (message queuing) cho phép các service giao tiếp một cách không đồng bộ, giúp giảm bớt sự phụ thuộc lẫn nhau giữa các services và cải thiện khả năng chịu lỗi của hệ thống. Khi một services gặp vấn đề, các yêu cầu có thể được lưu vào hàng đợi và xử lý sau, mà không làm ảnh hưởng đến trải nghiệm người dùng cuối.
3. **Container Orchestration (Điều phối Container)**: Công nghệ điều phối container, như Kubernetes, cung cấp khả năng Automation Devops, mở rộng và quản lý vòng đời của các microservices. Sử dụng containerization không chỉ giúp đơn giản hóa việc phát triển và triển khai service mà còn đảm bảo tính nhất quán và cô lập môi trường giữa phát triển và sản xuất, giúp giảm thiểu các vấn đề liên quan đến "nó hoạt động trên máy tôi, nhưng không hiểu vì sao lại không hoạt động trên máy bạn".

#### So sánh kiến trúc Microservices và kiến trúc nguyên khối (Monolithic)



Hình 2.2: So sánh giữa Monolithic và Microservices

*(Nguồn:* [*https://www.openlegacy.com/hs-fs/hubfs/Picture1.webp?width=889&height=478&name=Picture1.webp*](https://www.openlegacy.com/hs-fs/hubfs/Picture1.webp?width=889&height=478&name=Picture1.webp)*)*

Chương này sẽ so sánh hai kiến trúc phần mềm phổ biến là **Microservices** và **Monolithic architecture**. Đây là hai mô hình phát triển ứng dụng có ảnh hưởng đáng kể đến cách thiết kế, triển khai, và quản lý các hệ thống phần mềm.

* + - * **Định nghĩa và Cấu trúc:**
  + **Monolithic Architecture**: Trong mô hình này, tất cả các thành phần của ứng dụng (bao gồm cả frontend, backend, và database layers) được phát triển và triển khai như một đơn thể duy nhất. Mô hình này thường đơn giản hơn để phát triển và triển khai ban đầu, nhưng có thể gặp khó khăn khi quy mô tăng lên hoặc khi cần cập nhật và bảo trì.
  + **Microservices Architecture**: Trong mô hình Microservices, ứng dụng được chia thành nhiều dịch vụ nhỏ, mỗi dịch vụ hoạt động độc lập và thực hiện một chức năng cụ thể của hệ thống. Các Microservices này giao tiếp với nhau thông qua API (Application Programming Interface) và thường được triển khai một cách độc lập.
* **Scalability và Maintenance:**
* **Monolithic Architecture**: Việc mở rộng quy mô (scaling) và bảo trì ứng dụng monolithic có thể trở nên phức tạp khi kích thước ứng dụng tăng lên. Mọi thay đổi, dù nhỏ, đều yêu cầu xây dựng lại và triển khai lại toàn bộ ứng dụng, điều này có thể dẫn đến downtime đáng kể và tăng rủi ro trong quản lý phiên bản.
* **Microservices Architecture**: Các dịch vụ trong mô hình Microservices có thể được mở rộng độc lập, cho phép quản lý quy mô hiệu quả hơn. Bảo trì và cập nhật từng Microservice riêng biệt cũng dễ dàng hơn, giúp giảm thiểu rủi ro và downtime. Tuy nhiên, quản lý nhiều dịch vụ đồng thời đòi hỏi các công cụ và kỹ năng quản lý phức tạp.
* **Độ phức tạp và Cost of Development:**
* **Monolithic Architecture**: Phát triển ban đầu thường nhanh hơn và đơn giản hơn vì tất cả các thành phần của ứng dụng được tích hợp chặt chẽ. Tuy nhiên, chi phí và độ phức tạp có thể tăng lên đáng kể khi ứng dụng phát triển kích thước và tính năng.
* **Microservices Architecture**: Phát triển dựa trên Microservices đòi hỏi kỹ thuật cao hơn và ban đầu có thể tốn kém hơn do cần thiết lập một hạ tầng phức tạp để quản lý các dịch vụ độc lập. Tuy nhiên, trong dài hạn, mô hình này thường mang lại lợi ích về khả năng mở rộng và dễ dàng bảo trì hơn.
* **Thích hợp với từng loại ứng dụng:**
* **Monolithic Architecture** thường phù hợp với các dự án nhỏ đến trung bình, nơi mà độ phức tạp của sản phẩm không cao và không yêu cầu mở rộng quy mô lớn.
* **Microservices Architecture** được ưu tiên cho các dự án lớn hơn, có yêu cầu cao về quy mô và tính linh hoạt, đặc biệt là trong môi trường phát triển năng động và phức tạp.

#### Những khó khăn khi triển khai Microservices

* **Sự phức tạp trong quản lý và điều hành:**

Khi chia nhỏ ứng dụng thành nhiều service, có thể dẫn đến sự phức tạp về quản lý và điều hành. Số lượng service tăng lên và có thể trở nên khó kiểm soát, đặc biệt khi phải xử lý các sự phụ thuộc giữa các service.

* **Tăng chi phí**:

Với kiến trúc Microservices, giao tiếp giữa các service phải thông qua mạng, điều này có thể tăng chi phí và ảnh hưởng đến hiệu suất của hệ thống cũng như xuất hiện nhiều rủi ro tiềm tàng về mạng.

* **Triển khai phức tạp hơn**:

Triển khai và quản lý các service đòi hỏi sự phối hợp và quản lý khá phức tạp và phải đảm bảo rằng các service hoạt động một cách đồng nhất và tương tác tốt với nhau.

#### Kết luận về Microservices

Microservices mang lại tính linh hoạt, khả năng phát triển và triển khai độc lập, cũng như tích hợp và khả năng tái sử dụng tốt hơn so với kiến trúc monolithic. Tuy nhiên, việc triển khai microservices có thể đòi hỏi quản lý và phát triển phức tạp hơn so với monolithic, do sự tách biệt và đa dạng của các dịch vụ. Trong đề tài này, hệ thống áp dụng kiến trúc Microservice để xây dựng hệ thống quản lý bán hàng, có khả năng mở rộng cao, đồng thời tích hợp nhiều công cụ quản lý hệ thống và công cụ DevOps giúp quản lý và giám sát các dịch vụ trong hệ thống một cách trực quan và hiệu quả, đồng thời cũng đảm bảo tính ổn định của hệ thống.

### RESTful API, API Gateway và KongAPI

#### RESTful API

##### Khái niệm

RESTful API là một tiêu chuẩn dùng trong việc thiết kế API cho các ứng dụng web (thiết kế Web services) để tiện cho việc quản lý các resource. Nó chú trọng vào tài nguyên hệ thống (tệp văn bản, ảnh, âm thanh, video, hoặc dữ liệu động…), bao gồm các trạng thái tài nguyên được định dạng và được truyền tải qua HTTP.



Hình: Mô tả về thiết kế của RESTful API

(Nguồn: <https://viblo.asia/p/restful-api-la-gi-1Je5EDJ4lnL>)

API (Application Programming Interface) là một tập hợp các quy tắc và cơ chế mà qua đó, một ứng dụng hoặc một thành phần có thể tương tác với một ứng dụng hoặc thành phần khác. API có thể trả về dữ liệu mà bạn cần cho ứng dụng của mình dưới các định dạng dữ liệu phổ biến như JSON hoặc XML.

REST (REpresentational State Transfer) là một kiểu kiến trúc dùng để xây dựng API, nó chuyển đổi cấu trúc dữ liệu và sử dụng các phương thức HTTP đơn giản để tạo ra giao tiếp giữa các máy. Thay vì sử dụng một URL để xử lý thông tin người dùng, REST gửi một yêu cầu HTTP như GET, POST, DELETE, v.v. đến một URL để xử lý dữ liệu.

RESTful API là một tiêu chuẩn để thiết kế các API cho các ứng dụng web nhằm quản lý các tài nguyên (resources). RESTful là một trong những kiểu thiết kế API phổ biến nhất hiện nay, cho phép các ứng dụng (web, di động, v.v.) giao tiếp với nhau.

Chức năng quan trọng nhất của REST là quy định cách sử dụng các phương thức HTTP (như GET, POST, PUT, DELETE...) và cách định dạng các URL để quản lý các tài nguyên trong ứng dụng web. RESTful không quy định logic code của ứng dụng và không bị giới hạn bởi ngôn ngữ lập trình, bất kỳ ngôn ngữ hoặc framework nào cũng có thể được sử dụng để thiết kế một RESTful API.

##### Cách hoạt động của RESTful API



Hình: Sơ đồ hoạt động của RESTful API

REST hoạt động chủ yếu dựa vào giao thức HTTP. Các hoạt động cơ bản nêu trên sẽ sử dụng những phương thức HTTP riêng.

* GET (SELECT): Trả về một Resource hoặc một danh sách Resource.
* POST (CREATE): Tạo mới một Resource.
* PUT (UPDATE): Cập nhật thông tin cho Resource.
* DELETE (DELETE): Xoá một Resource.

Các bước hoạt động của RESTful API

1. **Client Gửi Yêu Cầu (Request)**

Quá trình tương tác bắt đầu khi client (máy khách) gửi một yêu cầu HTTP tới server (máy chủ). Yêu cầu này bao gồm các thành phần sau:

* **Phương Thức HTTP**

Xác định loại hành động mà client muốn thực hiện. Các phương thức phổ biến bao gồm GET (để lấy dữ liệu), POST (để tạo mới), PUT (để cập nhật), và DELETE (để xóa).

* **URI (Uniform Resource Identifier)**

Xác định tài nguyên mà client muốn tương tác. URI dùng để xác định tài tài nguyên bởi chính xác nơi lấy nó hoặc tên của nó, và tất nhiên là có thể bằng cả hai. Tài nguyên, hay chính xác hơn với trường hợp này là tài nguyên trên mạng, nó có thể là tập tin nhạc, ảnh, văn bản, css, json. Ví dụ: https://api.example.com/users/123.

* **Headers**

Cung cấp thông tin bổ sung về yêu cầu, chẳng hạn như kiểu nội dung (Content-Type), định dạng mà client chấp nhận (Accept), và thông tin xác thực (Authorization).

* **Body**

Chứa dữ liệu cần thiết cho các phương thức POST và PUT, chẳng hạn như thông tin người dùng yêu cầu

1. **Server Nhận và Xử Lý Yêu Cầu**

Khi server nhận được yêu cầu từ client, nó sẽ thực hiện các bước sau để xử lý yêu cầu:

* **Xác Thực (Authentication)**

Kiểm tra xem yêu cầu có đến từ một nguồn hợp lệ và người dùng có được phép truy cập hay không.

* **Ủy Quyền (Authorization)**

Xác định liệu người dùng có quyền thực hiện hành động yêu cầu trên tài nguyên cụ thể hay không.

* **Xử Lý Yêu Cầu (Processing)**

Thực hiện hành động yêu cầu dựa trên phương thức HTTP và URI. Ví dụ, nếu là yêu cầu GET, server sẽ truy xuất thông tin tài nguyên từ cơ sở dữ liệu hoặc hệ thống lưu trữ khác.

* **Tạo Phản Hồi (Response)**

Sau khi xử lý yêu cầu, server sẽ tạo ra một phản hồi để gửi lại cho client. Phản hồi này bao gồm một mã trạng thái HTTP, các header, và một body (nếu cần).n tạo hoặc cập nhật.

1. **Client Nhận và Xử Lý Phản Hồi**

Khi client nhận được phản hồi từ server, nó sẽ thực hiện các bước sau:

* **Kiểm Tra Status code HTTP**

Mã trạng thái HTTP cho biết kết quả của yêu cầu. Ví dụ: 200 OK cho biết yêu cầu đã thành công, 404 Not Found cho biết tài nguyên không tồn tại, và 500 Internal Server Error cho biết có lỗi xảy ra trên server.

* **Xử Lý Headers**

Headers trong phản hồi cung cấp thông tin về dữ liệu trả về và cách xử lý nó, chẳng hạn như kiểu nội dung (Content-Type), kích thước nội dung (Content-Length), và thông tin về caching (Cache-Control).

* **Xử Lý Body**

Nếu phản hồi chứa body, client sẽ xử lý dữ liệu theo yêu cầu. Ví dụ, nếu nội dung là JSON, client có thể phân tích cú pháp và sử dụng dữ liệu này trong ứng dụng của mình.

##### So sánh giữa RESTful API và SOAP API

Cả 2 loại SOAP và RESTful đều cho phép phía client gửi request đến server để query, nhưng chúng được thực hiện bằng những cách khác nhau. Sự khác nhau chính giữa SOAP và REST là cách mà client giao tiếp server thông qua SOAP sẽ bị hạn chế bởi nhiều quy tắc và format được thiết kế chính xác, trong khi với REST cho phép việc giao tiếp thông qua giao thức HTTP và ít các quy tắc rườm rà hơn.

1. **RESTful API**

REST (Representational State Transfer) là một phong cách kiến trúc cho các hệ thống phân tán, dựa trên một tập hợp các nguyên tắc và ràng buộc. Các nguyên tắc chính của REST bao gồm:

* **Định danh tài nguyên duy nhất (Unique Resource Identification)**

Mỗi tài nguyên được định danh bởi một URI (Uniform Resource Identifier).

* **Đại diện tài nguyên (Resource Representation)**

Tài nguyên có thể được đại diện dưới nhiều định dạng như JSON, XML, HTML.

* **Stateless (Không Trạng Thái)**

Mỗi yêu cầu từ client đến server phải chứa đủ thông tin để server có thể xử lý yêu cầu đó độc lập với các yêu cầu khác.

* **Giao diện đồng nhất (Uniform Interface)**

Sử dụng các phương thức HTTP chuẩn như GET, POST, PUT, DELETE để thao tác với tài nguyên.

* **Cacheable (Có Thể Lưu Trữ Tạm Thời)**

Phản hồi từ server có thể được đánh dấu là cacheable để client có thể lưu trữ và tái sử dụng.

1. **SOAP**

SOAP là một giao thức dựa trên XML để trao đổi thông điệp giữa các hệ thống. Các đặc điểm chính của SOAP bao gồm:

* **Định dạng XML (XML-based Protocol)**

Tất cả các thông điệp SOAP đều được định dạng bằng XML, cung cấp tính linh hoạt cao và khả năng mở rộng.

* **Giao thức độc lập (Protocol Agnostic)**

SOAP có thể sử dụng nhiều giao thức vận chuyển khác nhau, bao gồm HTTP, SMTP, TCP.

* **Hỗ trợ WS- Standards\***

SOAP hỗ trợ nhiều tiêu chuẩn web service mở rộng như WS-Security, WS-AtomicTransaction, WS-ReliableMessaging.

* **Cấu trúc thông điệp phức tạp (Complex Message Structure)**

SOAP cho phép định nghĩa các thông điệp phức tạp và hỗ trợ các kiểu dữ liệu phong phú thông qua XML Schema.

* **Stateful hoặc Stateless**

SOAP có thể hoạt động trong môi trường có trạng thái (stateful) hoặc không trạng thái (stateless) tùy vào yêu cầu của ứng dụng.

1. **So Sánh Chi Tiết**

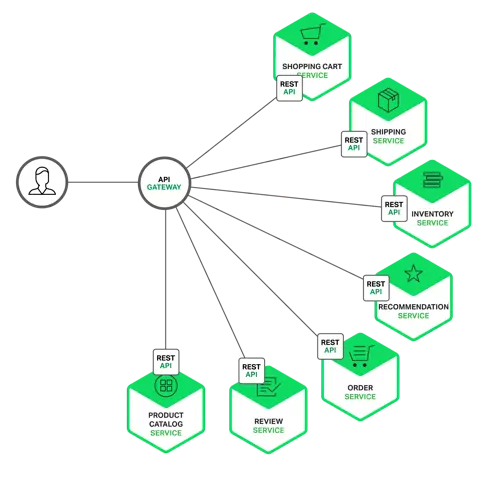
|  |  |  |
| --- | --- | --- |
| **Tiêu chí** | **RESTful API** | **SOAP API** |
| Kiến trúc và giao thức |  | Chuẩn hóa giao tiếp giữa client và server về format, structure và method. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

#### API Gateway

##### Khái niệm

API là viết tắt của Application Program Interface, chứa tập hợp các công cụ để xây dựng các ứng dụng. API Gateway là một công cụ quản lý API nằm giữa các máy khách (client) và nhóm các backend service. API gateway hoạt động như một reverse proxy, chấp nhận tất cả các lệnh gọi API, tìm nạp và tổng hợp các tài nguyên thích hợp trước khi gửi phản hồi cho mỗi yêu cầu API.

API Gateway có thể coi là một cổng trung gian, nó là cổng vào duy nhất tới hệ thống microservices của chúng ta, api gateway sẽ nhận các requests từ phía client, chỉnh sửa, xác thực và điều hướng chúng đến các API cụ thể trên các services phía sau.



Hình: Sơ đồ hệ thống sử dụng API Gateway

*(Nguồn:* [*https://images.viblo.asia/7c5add32-77c0-43c4-bb76-f87ec53badaa.png*](https://images.viblo.asia/7c5add32-77c0-43c4-bb76-f87ec53badaa.png)*)*

Ngoài nhiệm vụ chính là proxy request thì một hệ thống API Gateway thường sẽ đảm nhận luôn vài vai trò khác như bảo mật API, điều khiển (monitoring), phân tích (analytics) số lượng requests cũng như tình trạng hệ thống phía sau.

##### Cách hoạt động của API Gateway

1. **Tiếp nhận yêu cầu (Request):** Khi một yêu cầu gửi từ client đến hệ thống thông qua mạng, API Gateway sẽ tiếp nhận yêu cầu này.
2. **Xác thực và Quyền Truy Cập:** API Gateway kiểm tra xem yêu cầu này có hợp lệ không, xác thực người dùng nếu cần thiết và kiểm tra xem họ có quyền truy cập vào tài nguyên hay chức năng yêu cầu không.
3. **Định Tuyến (Routing):** Sau khi xác thực, API Gateway quyết định nơi yêu cầu sẽ được chuyển tiếp trong hệ thống. Nó có thể định tuyến yêu cầu đến dịch vụ backend cụ thể dựa trên loại yêu cầu, URI hoặc bất kỳ yêu cầu khác.
4. **Chuyển Tiếp Yêu Cầu:** API Gateway chuyển tiếp yêu cầu từ client đến dịch vụ backend tương ứng. Điều này có thể bao gồm việc gửi yêu cầu HTTP, SOAP hoặc giao thức khác tới dịch vụ backend.
5. **Xử Lý Yêu Cầu Backend:** Dịch vụ backend tiếp nhận yêu cầu từ API Gateway và xử lý nó theo yêu cầu của client. Điều này có thể liên quan đến truy vấn cơ sở dữ liệu, xử lý logic kinh doanh, hoặc bất kỳ tác vụ nào khác mà dịch vụ thực hiện.
6. **Nhận Phản Hồi (Response):** Sau khi dịch vụ backend hoàn thành xử lý, nó trả về phản hồi cho API Gateway. Phản hồi có thể là dữ liệu, mã lỗi, thông báo hoặc bất kỳ thông tin nào mà client cần biết.
7. **Xử Lý Phản Hồi:** API Gateway kiểm tra phản hồi từ dịch vụ backend và thực hiện xử lý dựa trên nội dung của phản hồi. Điều này có thể bao gồm việc ghi log, định dạng lại dữ liệu hoặc thậm chí là xử lý mã lỗi.
8. **Trả Lời Yêu Cầu Client:** Sau khi hoàn tất xử lý phản hồi, API Gateway chuyển tiếp phản hồi từ dịch vụ backend tới client ban đầu. Phản hồi có thể bao gồm dữ liệu được định dạng lại theo yêu cầu của client.

##### Ưu điểm khi sử dụng API Gateway trong hệ thống microservice

* ***Che giấu cấu trúc của hệ thống microservices với bên ngoài***

Khách hàng sẽ tương tác với hệ thống của chúng ta thông qua API Gateway, không phải gọi trực tiếp đến từng dịch vụ cụ thể. Các endpoints của các dịch vụ chỉ được gọi nội bộ, tức là giữa các dịch vụ với nhau hoặc từ API Gateway. Do đó, phía client không cần và cũng không thể biết được cách thức phân chia các dịch vụ ở backend. Việc này giúp lập trình viên dễ dàng hơn trong việc refactor code frontend.

* ***Code frontend gọn gàng hơn:***

Thay vì phải theo dõi nhiều endpoints, tất cả chỉ việc gọi đến API Gateway, giúp phần code frontend trở nên gọn gàng hơn. Điều này đặc biệt hữu ích khi hệ thống ngày càng mở rộng và phức tạp.

* ***Dễ dàng theo dõi và quản lý lưu lượng truy cập:***

Hầu hết các hệ thống API Gateway phổ biến hiện nay đều đi kèm với tính năng theo dõi và quản lý lượng traffic thông qua GUI hoặc các APIs của hệ thống Gateway. Ví dụ như với Kong (phiên bản Enterprise), việc quản lý lưu lượng trở nên đơn giản hơn rất nhiều.

* ***Caching yêu cầu và cân bằng tải***

API Gateway cũng kiêm luôn vai trò của một load balancer, giúp phân phối các yêu cầu đến các dịch vụ backend, giảm thiểu rủi ro quá tải hệ thống. Việc caching các yêu cầu cũng giúp cải thiện hiệu suất tổng thể.

* ***Thêm một lớp bảo mật cho hệ thống***:

API Gateway giúp ngăn chặn các cuộc tấn công bằng cách thêm một lớp bảo vệ chống lại các loại tấn công như DDoS, SQL Injection,... Điều này giúp nâng cao mức độ an toàn của hệ thống.

* ***Thay thế dịch vụ xác thực***

API Gateway thường cung cấp nhiều cơ chế xác thực khác nhau, giúp chúng ta có thể sử dụng để xác thực người dùng, tiết kiệm thời gian và làm hệ thống trở nên đơn giản hơn. Ví dụ, Kong API Gateway hỗ trợ nhiều cơ chế xác thực hữu ích.

##### Nhược điểm khi sử dụng API Gateway

### NestJS Framework

#### Tổng quan về NestJS

##### NestJS là gì

Nest (NestJS) là một framework Node.js dùng để xây dựng các ứng dụng ở phía server (server-side applications) có tính mở rộng và hiệu quả. NestJS được xây dựng và được hỗ trợ rất mạnh khi sử dụng Typescript (NestJS vẫn cho phép phát triển bằng Javascript) và tổng hợp rất nhiều các tính chất của OOP và Functional Programming.



Hình: Logo NestJS

(Nguồn: [docs.nestjs.com/assets/logo-small-gradient.svg](https://docs.nestjs.com/assets/logo-small-gradient.svg))

##### Ưu điểm của NestJS

NestJS tương thích với cả Typescript và Javascript thuần, default là Typescript. Điều này giúp NestJS có thể được dùng bởi rất nhiều lập trình viên mà không gặp khó về ngôn ngữ mà họ sử dụng.

Kiến trúc của NestJS hỗ trợ rất tốt các pattern design khác nhau như Service Repository Pattern hay Dependency Injection một cách dễ dàng. Design Pattern xuyên suốt NestJS là Dependency Injection, giúp cho việc import các module trở nên vô cùng dễ dàng

NestJS có rất nhiều module hỗ trợ, từ việc hot reload, logger cho đến GraphQL, Websocket rồi cqrs Pattern, Microservices,... Ta chỉ cần NestJS để làm tất cả mọi thứ.

Module microservices của NestJS hỗ trợ đủ loại kết nối: RabbitMQ, gRPC, Kafka, Redis,... đây là một trong những điều làm NestJS trở nên vô cùng hiệu quả khi xây dựng các ứng dụng dựa trên kiến trúc Microservice. Lập trình viên chỉ cần tập trung vào code cho phần business, còn infra thì NestJS đã implement sẵn chỉ việc dùng.

##### Lợi ích của NestJS trong xây dựng ứng dụng trên kiến trúc Microservice

NestJS cung cấp nhiều tính năng vượt trội giúp dễ dàng triển khai kiến trúc microservice:

1. Modularity (Tính Module):

NestJS cho phép chia ứng dụng thành các module nhỏ, mỗi module chứa các thành phần như controller, service, middleware và provider. Sự modularity này giúp tổ chức mã nguồn một cách cấu trúc và dễ quản lý, điều này rất quan trọng trong một hệ thống microservice với nhiều dịch vụ độc lập.

2. Dependency Injection (DI):

NestJS sử dụng dependency injection để quản lý sự phụ thuộc giữa các thành phần của ứng dụng, giúp giảm sự rắc rối trong việc quản lý các module và dịch vụ. Điều này làm cho việc thêm, thay đổi hoặc loại bỏ các thành phần trở nên dễ dàng hơn, đặc biệt là trong một môi trường microservice có sự phức tạp cao.

3. Scalability (Khả năng Mở Rộng):

NestJS hỗ trợ mô hình mở rộng dễ dàng, cho phép mở rộng các dịch vụ một cách độc lập mà không ảnh hưởng đến các phần khác của hệ thống. Điều này làm cho việc mở rộng và quản lý các dịch vụ trở nên dễ dàng hơn trong kiến trúc microservice so với kiến trúc monolithic.

5. Built-in Support for Microservices (Hỗ Trợ Tích Hợp Cho Microservices):

NestJS cung cấp các công cụ và cơ chế tích hợp sẵn để xây dựng và quản lý microservices. NestJS có thể dễ dàng triển khai và xây dựng các services, giao tiếp giữa các services, và quản lý việc mở rộng và load balancing một cách hiệu quả.

6. Flexibility (Độ Linh Hoạt):

NestJS cung cấp một loạt các tính năng và công nghệ để tùy chỉnh và mở rộng ứng dụng theo cách phù hợp nhất với nhu cầu cụ thể của dự án microservice. Từ GraphQL đến WebSockets và các công nghệ giao tiếp khác, NestJS cung cấp một nền tảng đa dạng cho việc phát triển các dịch vụ microservice.

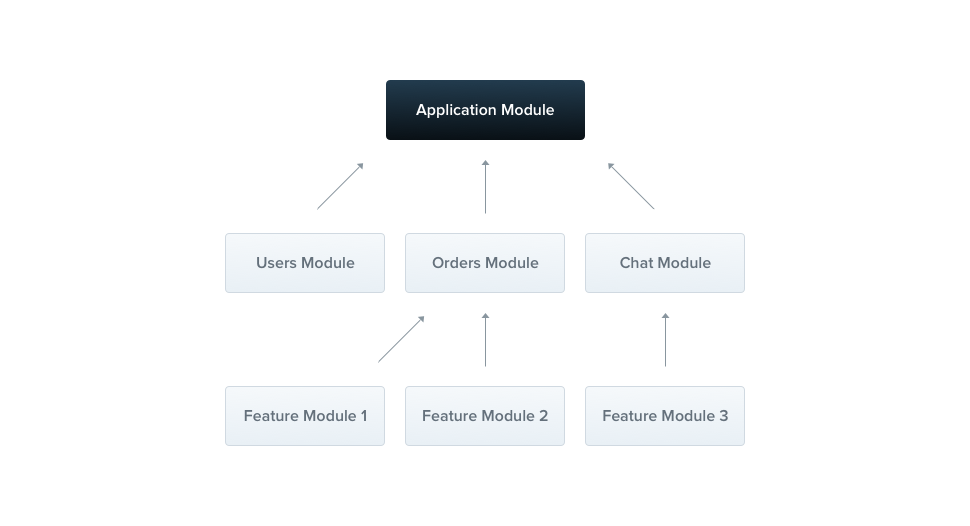
7. Ease of Testing (Dễ Kiểm Thử):

Nhờ vào cấu trúc modularity và dependency injection, việc kiểm thử các dịch vụ trong một ứng dụng microservice được thực hiện dễ dàng hơn. Có thể tách biệt và kiểm thử từng dịch vụ một cách độc lập mà không cần phải triển khai toàn bộ hệ thống.

#### Các thành phần nền tảng của NestJS framework

##### Module

Module là khái niệm quan trọng của NestJS, chịu trách nhiệm tổ chức và đóng gói các thành phần của ứng dụng. Một ứng dụng NestJS thường bao gồm nhiều module, mỗi module chứa một tập hợp các thành phần như controllers, services, providers, và middleware. Các module giúp tách biệt logic của ứng dụng thành các phần nhỏ hơn, dễ quản lý và bảo trì.



Hình: Mô tả cấu trúc của một Module

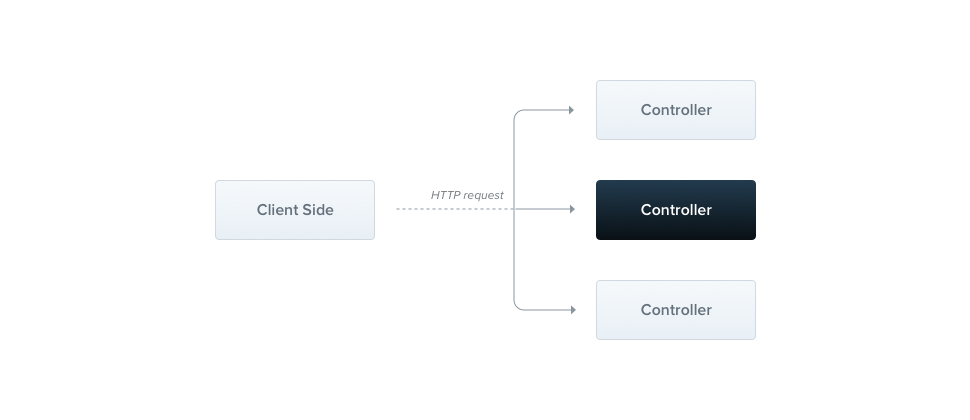
(Nguồn: <https://docs.nestjs.com/assets/Modules_1.png>)

Module có thể được khai báo thông qua decorator @Module(), trong đó được xác định các controllers, providers, và imports cần thiết cho module đó. Sự phụ thuộc giữa các module được quản lý thông qua imports và exports, giúp tổ chức ứng dụng một cách có cấu trúc và dễ hiểu.

##### Controller

Controller là thành phần xử lý các yêu cầu đến từ client thông qua các endpoint HTTP hoặc các event khác. Mỗi controller định nghĩa một hoặc nhiều route, và xác định hành động cần thực hiện khi một request đến.

Các action trong controller có thể trả về dữ liệu hoặc response tương ứng với request, hoặc gọi các service khác để xử lý logic.



Hình: Mô tả cách hoạt động của Controller

(Nguồn: <https://docs.nestjs.com/assets/Controllers_1.png>)

Các controller được đánh dấu bằng decorator @Controller(), và các route được định nghĩa bằng các decorator như @Get(), @Post(), @Put(), @Delete(), vv. NestJS cung cấp cơ chế tiện lợi để xác định endpoint và xử lý dữ liệu từ request.

##### Service

Service là thành phần chứa logic nghiệp vụ của ứng dụng. Trong NestJS, mọi logic nghiệp vụ nên được đặt trong service, trong khi controller chỉ nên xử lý việc gọi service và trả về dữ liệu.

Các service có thể chứa các phương thức để thao tác với cơ sở dữ liệu, gửi yêu cầu HTTP, hoặc thực hiện các tác vụ khác liên quan đến logic của ứng dụng.

Các service được đánh dấu bằng decorator @Injectable(), và sau đó có thể được inject vào các controller hoặc các service khác thông qua dependency injection.

##### Middleware

Middleware là các hàm trung gian được thực thi trước khi request đến được xử lý bởi một controller hoặc sau khi response được trả về từ controller. Middleware thường được sử dụng để thực hiện các nhiệm vụ như xác thực, ghi log, xử lý lỗi, hoặc thực hiện các tác vụ tiền xử lý và sau xử lý cho request và response.

Middleware có thể được định nghĩa bằng cách sử dụng decorator @Middleware() hoặc thông qua sử dụng hàm trung gian trực tiếp trong pipeline của ứng dụng.

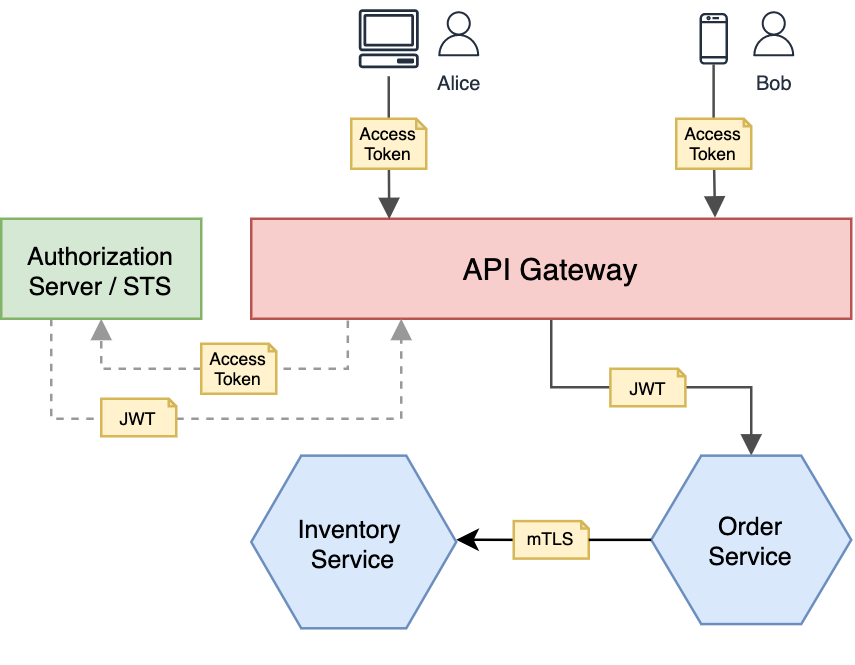
#### Cơ chế bảo mật của NestJS

Bảo mật là một phần quan trọng không thể thiếu trong việc phát triển ứng dụng, đặc biệt là khi xây dựng các dịch vụ Microservice. Có thể nói, cơ chế bảo mật chính là điều kiện tiên quyết giúp ứng dụng Microservice hoạt động hiệu quả.

NestJS cung cấp một loạt các cơ chế bảo mật để giúp bảo vệ ứng dụng khỏi các cuộc tấn công và lỗ hổng bảo mật. Dưới đây là một số cơ chế bảo mật chính trong NestJS:

##### Authentication (Xác Thực)

NestJS hỗ trợ nhiều phương thức xác thực khác nhau như JWT (JSON Web Tokens), OAuth, Basic Auth, và Passport. Có thể chọn phương thức phù hợp với nhu cầu cụ thể của ứng dụng và triển khai xác thực thông qua middleware hoặc interceptor.



*Hình: Bảo mật Microservices với JWT*

*(Nguồn:* [*https://medium.com/ballerina-techblog/securing-microservices-with-jwt-a16b738b110f*](https://medium.com/ballerina-techblog/securing-microservices-with-jwt-a16b738b110f)*)*

##### Authorization (Ủy Quyền)

NestJS cung cấp các cơ chế để kiểm tra quyền truy cập của người dùng vào các tài nguyên và hành động trong ứng dụng. Bằng cách sử dụng middleware hoặc interceptor, NestJS có thể thực hiện kiểm tra quyền truy cập trước khi cho phép request được xử lý, đảm bảo rằng chỉ những người dùng có quyền mới có thể truy cập để sử dụng các tài nguyên đó.

##### CORS (Cross-Origin Resource Sharing)

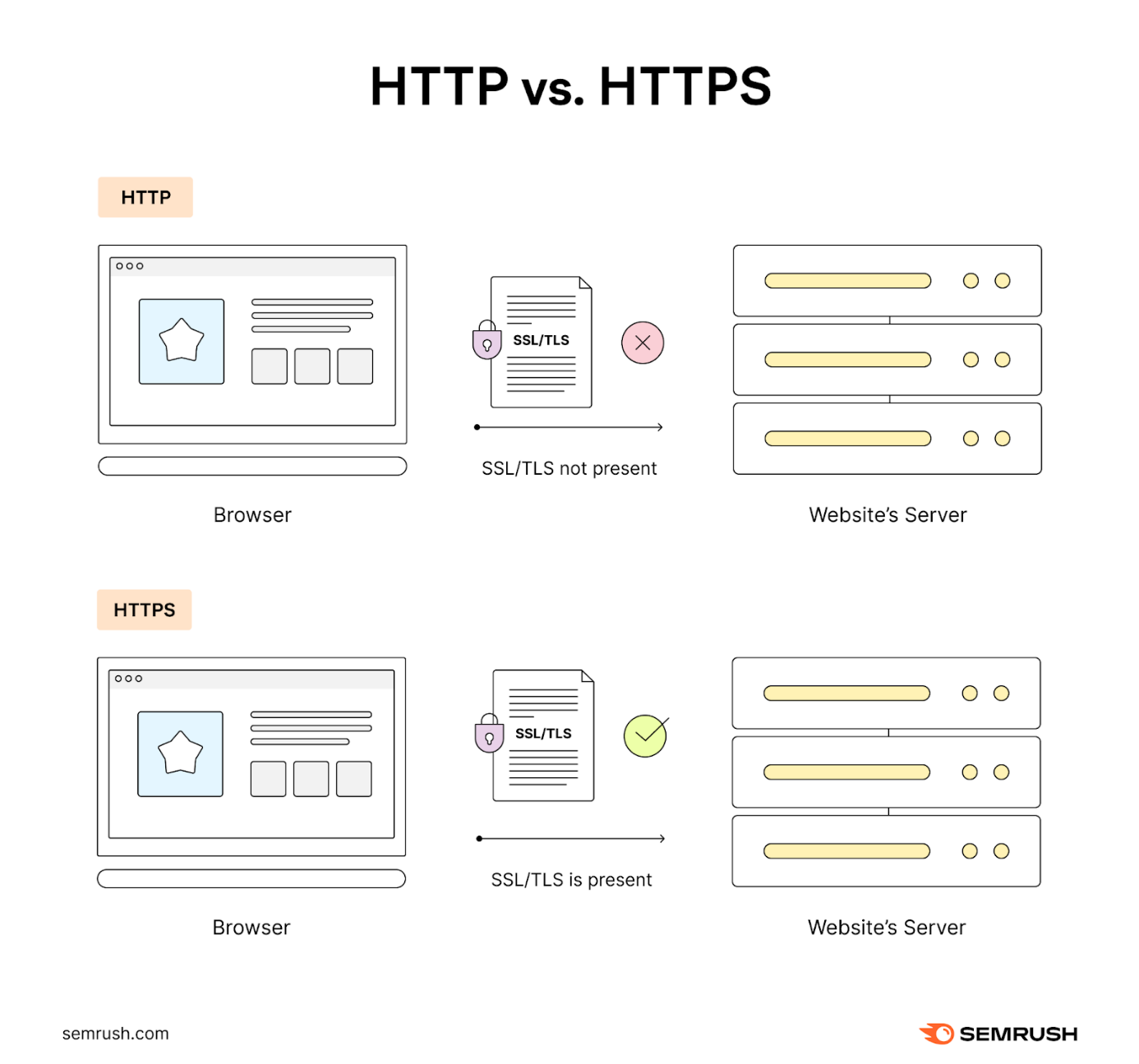
NestJS cung cấp cơ chế CORS để kiểm soát việc truy cập tài nguyên từ các trang web khác nguồn gốc. Bằng cách cấu hình CORS, có thể xác định các trang web nào được phép truy cập vào các API của ứng dụng và xác định các tùy chọn như phương thức, tiêu đề, và quyền truy cập.

##### CSRF (Cross-Site Request Forgery) Protection

NestJS cung cấp các cơ chế bảo vệ chống lại cuộc tấn công CSRF bằng cách sử dụng token CSRF. Khi sử dụng middleware CSRF, NestJS sẽ tạo ra và gửi một token CSRF với mỗi yêu cầu và kiểm tra token này khi yêu cầu được gửi đi, đảm bảo rằng yêu cầu chỉ được chấp nhận từ các trang web đã xác định.

##### HTTPS (HTTP Secure)

NestJS hỗ trợ việc triển khai ứng dụng qua HTTPS, giúp mã hóa dữ liệu truyền đi giữa client và server và đảm bảo tính bảo mật của thông tin nhạy cảm như tên người dùng, mật khẩu và dữ liệu cá nhân.



*Hình: Cơ chế bảo mật của HTTP và HTTPS*

(Nguồn: <https://www.semrush.com/blog/what-is-https/>)

NestJS cho phép triển khai các cơ chế bảo mật thông qua middleware và interceptor. Điều này cho phép bạn kiểm soát các yêu cầu và phản hồi trước khi chúng được xử lý hoặc trả về, giúp đảm bảo tính bảo mật của ứng dụng.

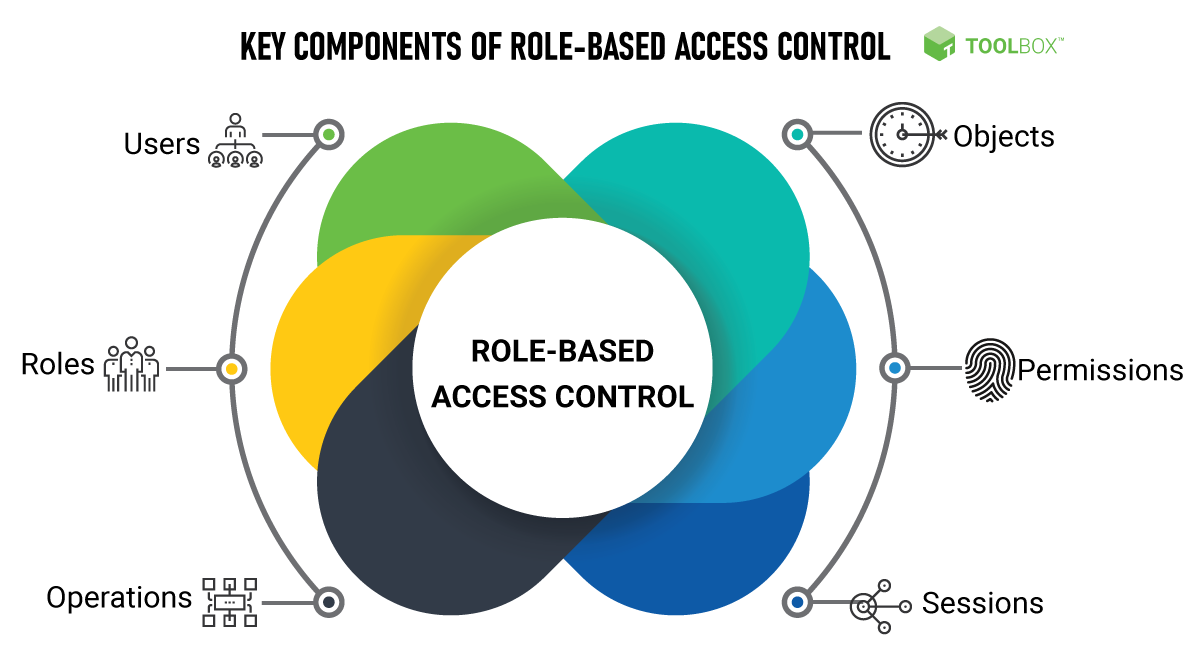
##### Role-Based Access Control (RBAC)

Kiểm soát truy cập dựa trên vai trò (Role-Based Access Control gọi tắt là RBAC) đề cập đến ý tưởng gán quyền cho người dùng dựa trên vai trò của họ trong một tổ chức. RBAC cung cấp cách tiếp cận quản lý truy cập đơn giản, dễ quản lý và ít xảy ra lỗi.

Khi sử dụng RBAC, cần phân tích nhu cầu của người dùng và nhóm họ thành các vai trò dựa trên trách nhiệm chung. Sau đó, chỉ định một hoặc nhiều vai trò cho từng người dùng và một hoặc nhiều quyền cho từng vai trò.

Với RBAC, việc quản lý quyền truy cập trở nên dễ dàng hơn miễn là tuân thủ nghiêm ngặt các yêu cầu về vai trò. RBAC giúp:

* Tạo phân quyền có hệ thống
* Dễ dàng kiểm tra các đặc quyền của người dùng và khắc phục các sự cố
* Nhanh chóng thêm và thay đổi vai trò, cũng như triển khai
* Giảm khả năng xảy ra lỗi khi gán quyền cho người dùng
* Tích hợp người dùng bên thứ ba bằng cách cung cấp cho họ các vai trò được xác định trước
* Tuân thủ hiệu quả hơn quy định về bảo mật và quyền riêng tư



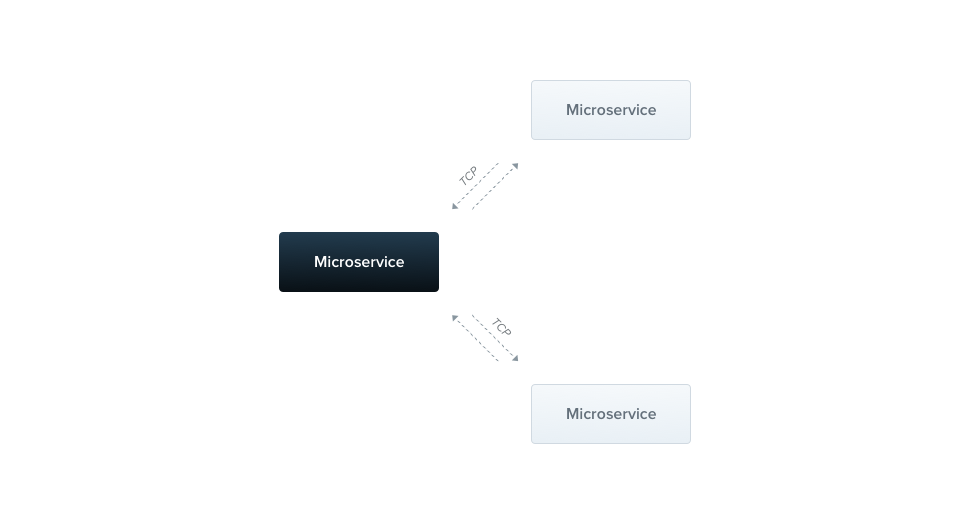
Hình: Các thành phần quan trọng của RBAC

(Nguồn: [*https://whitehat.vn/threads/kiem-soat-truy-cap-dua-tren-vai-tro-role-based-access-control-la-gi.17301/*](https://whitehat.vn/threads/kiem-soat-truy-cap-dua-tren-vai-tro-role-based-access-control-la-gi.17301/))

NestJS không có sẵn cơ chế RBAC nhưng có thể triển khai nó bằng cách sử dụng middleware hoặc interceptor để kiểm tra vai trò của người dùng và xác định quyền truy cập của họ vào các tài nguyên cụ thể.

#### Khả năng hỗ trợ mạnh mẽ cho kiến trúc Microservice

NestJS hỗ trợ các ứng dụng phát triển dựa trên kiến trúc Microservices rất tốt. Hầu hết các khái niệm của NestJS như dependency injection, decorators, pipes, guard và interceptors đều được áp dụng vào ứng dụng microservice.



*Hình: Cách giao tiếp giữa các microservice thông qua phương thức TCP*

(Nguồn: <https://docs.nestjs.com/assets/Microservices_1.png>)

Về cơ bản, trong NestJS, các công nghệ thường được sử dụng trong microservice đều được hỗ trợ rất mạnh, như Message Queuing (RabbitMQ), hệ thống message pub/sub phân tán (Redis, Kafka)

### Kubernetes

#### Container và giải pháp điều phối container

Image

##### Khái niệm

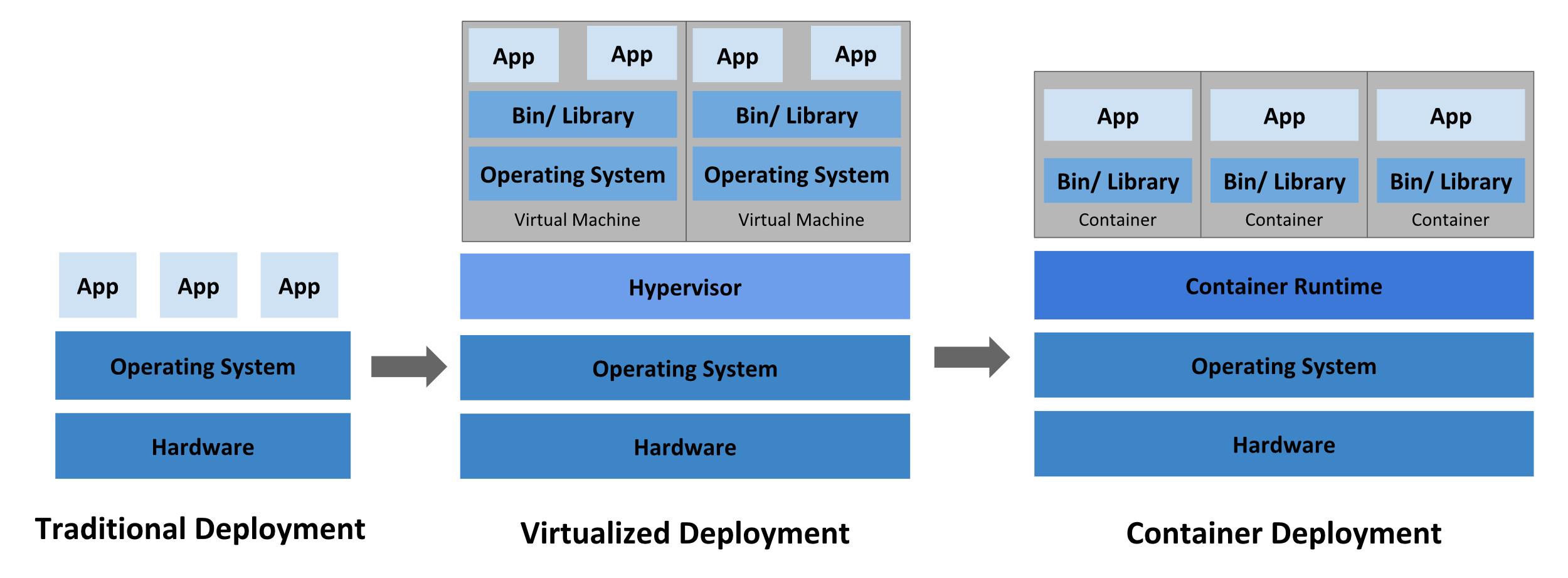
Container Image là một bản đóng gói (package) của phần mềm, bao gồm tất cả các phần để chạy ứng dụng đó như: Source code, thư viện, các config và dependency. Image đảm bảo việc triển khai ứng dụng, giúp ứng dụng có thể chạy một cách nhất quán và đồng bộ trên các môi trường triển khai khác nhau, từ môi trường phát triển đến môi trường thực tế

##### Cấu trúc

##### Môi trường Container (Container Environment)

Container là một run-time environment mà ở đó người dùng có thể chạy một ứng dụng độc lập. Những container này rất gọn nhẹ và cho phép chạy ứng dụng trong đó rất nhanh chóng và dễ dàng.

Công nghệ container cho phép đóng gói ứng dụng và tài nguyên liên quan của chúng vào một gói duy nhất gọi là container. Mỗi container hoạt động như một môi trường độc lập, chứa tất cả những gì cần thiết để chạy ứng dụng, bao gồm mã nguồn, thư viện, biến môi trường và các phụ thuộc khác.



Hình 2.3: Sự tiến hóa của công nghệ phát triển ứng dụng

(Nguồn: [kubernetes.io/images/docs/Container\_Evolution.svg](https://kubernetes.io/images/docs/Container_Evolution.svg))

Một số lợi ích của việc sử dụng container bao gồm:

* **Tính di động**: Container đảm bảo rằng ứng dụng được đóng gói cùng với mọi thứ cần thiết để chạy trên bất kỳ môi trường nào mà không bị phụ thuộc vào các yếu tố bên ngoài.
* **Tính cô lập**: Mỗi container hoạt động trong một môi trường độc lập, không tác động trực tiếp đến các container khác hoặc hệ thống máy chủ. Điều này đảm bảo tính an toàn và độ tin cậy của ứng dụng.
* **Tính nhất quán**: Containers đảm bảo rằng ứng dụng chạy tương tự trên mọi môi trường. Điều này giúp giảm thiểu sự khác biệt và xung đột giữa môi trường phát triển, kiểm thử và triển khai.
* **Tính tái sử dụng**: Containers có thể được xây dựng một lần và triển khai trên nhiều môi trường khác nhau. Điều này giúp tăng hiệu suất và giảm thời gian triển khai.

Các công nghệ container phổ biến nhất hiện nay là Docker và rkt. Docker đã trở thành tiêu chuẩn công nghiệp và cung cấp một cách tiện lợi để tạo, triển khai và quản lý container. Docker sử dụng Docker Engine để quản lý các quy trình container, cung cấp tính năng như quản lý tài nguyên, cô lập và mạng. Nhờ vào công nghệ container, việc triển khai ứng dụng trở nên dễ dàng, linh hoạt và nhất quán hơn.

#### Kubernetes

##### Giới thiệu

Kubernetes là một nền tảng nguồn mở, có khả năng mở rộng và di động, dùng để quản lý các containerized workloads và service được đóng gói trong container. Nền tảng này hỗ trợ cả cấu hình khai báo và tự động hóa, thuận tiện cho việc quản lý. Kubernetes sở hữu một hệ sinh thái lớn và phát triển nhanh chóng. Các dịch vụ, hỗ trợ và công cụ liên quan đến Kubernetes đều có sẵn rộng rãi.



Hình 2.4: Logo của Kubernetes

(Nguồn: <https://kubernetes.io/images/nav_logo2.svg>)

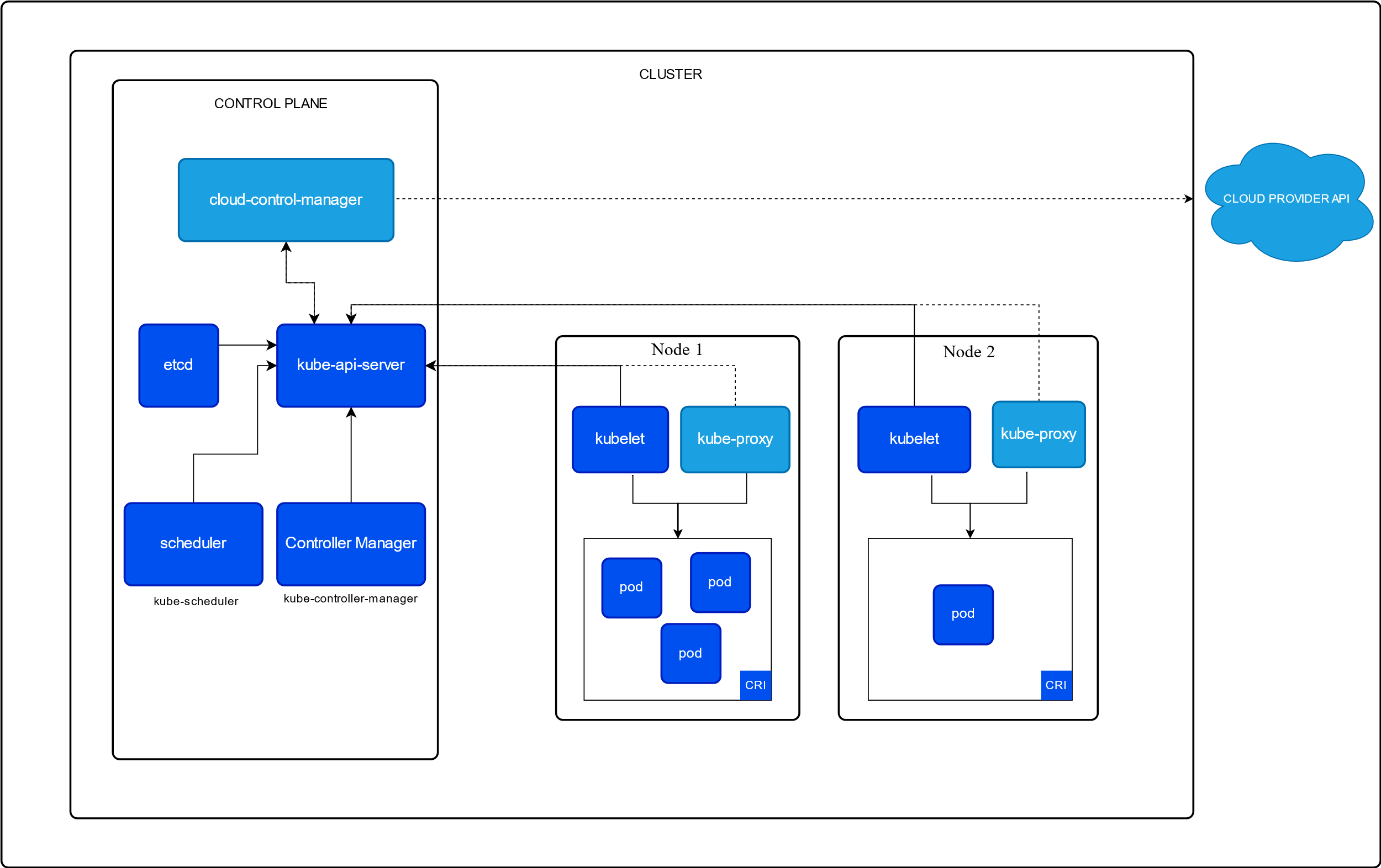
Tên gọi Kubernetes có nguồn gốc từ tiếng Hy Lạp, có nghĩa là thuyền trưởng hoặc phi công. Tên viết tắt "K8s" được hình thành từ việc đếm tám chữ cái nằm giữa chữ "K" và "s".

Kubernetes được phát triển ban đầu bởi Google vào năm 2014 và sau đó được công bố mã nguồn mở. Nền tảng này đã trải qua sự phát triển và mở rộng đáng kể và đã trở thành một trong những công cụ quản lý container phổ biến nhất trong ngành công nghệ.

Nguyên nhân chính dẫn đến sự ra đời của Kubernetes là để giải quyết các thách thức trong việc triển khai và quản lý các ứng dụng container phức tạp trên quy mô lớn. Trước khi có Kubernetes, việc triển khai và quản lý các ứng dụng container đòi hỏi nhiều công việc thủ công và phức tạp, gây ra sự phụ thuộc vào cấu hình hệ thống và gặp khó khăn trong việc mở rộng và khôi phục sau lỗi.

##### Kiến trúc Cluster

**Cluster Architecture** trong Kubernetes là cấu trúc tổ chức và triển khai các thành phần của một hệ thống Kubernetes. Nó bao gồm các máy chủ (nodes), thành phần điều phối (control plane components) và mạng lưới liên kết chúng.



Hình 2.5 Sơ đồ mô tả về Kiến trúc Cluster

(Nguồn: [kubernetes.io/images/docs/kubernetes-cluster-architecture.svg](https://kubernetes.io/images/docs/kubernetes-cluster-architecture.svg))

Trong một cụm Kubernetes, có hai loại máy chủ chính:

1. **Master Nodes**: Master nodes là các máy chủ chịu trách nhiệm điều phối và quản lý toàn bộ cụm Kubernetes. Chúng là trung tâm quản lý và chứa các thành phần điều phối chính của Kubernetes. Các thành phần quan trọng trên master nodes bao gồm:

* **API Server**: API server là thành phần trung tâm cho phép tương tác với cụm Kubernetes thông qua API.
* **Scheduler**: Scheduler quyết định và lên lịch cho việc triển khai các ứng dụng container trên các worker nodes dựa trên yêu cầu tài nguyên và các ràng buộc khác.
* **Controller Manager**: Controller manager là thành phần chịu trách nhiệm theo dõi trạng thái của cụm Kubernetes và thực hiện các tác vụ điều phối như khôi phục sau lỗi, mở rộng và tự động cân bằng tải.
* **etcd**: etcd là một cơ sở dữ liệu phân tán được sử dụng để lưu trữ trạng thái của cụm Kubernetes. Nó giúp đảm bảo tính nhất quán và đồng bộ giữa các thành phần của hệ thống.

1. **Worker Nodes**: Worker nodes là các máy chủ chạy ứng dụng container và cung cấp tài nguyên tính toán để chạy các ứng dụng đó. Các thành phần quan trọng trên worker nodes bao gồm:

* **Kubelet**: Kubelet là một agent chạy trên mỗi worker node, quản lý và điều khiển các container trên node đó.
* **Container Runtime**: Container runtime (như Docker) là môi trường chạy container trên worker node. Nó thực hiện công việc tạo, khởi chạy và quản lý các container.
* **Kube Proxy**: Kube Proxy là một proxy mạng giữa các container trên worker node và mạng lưới Kubernetes, cho phép các container tương tác với nhau và với các dịch vụ khác trong cụm.

Mạng lưới trong Cluster Architecture cung cấp kết nối giữa các server và container trong cụm Kubernetes. Nó đảm bảo tính nhất quán và liên tục giữa các thành phần của hệ thống.

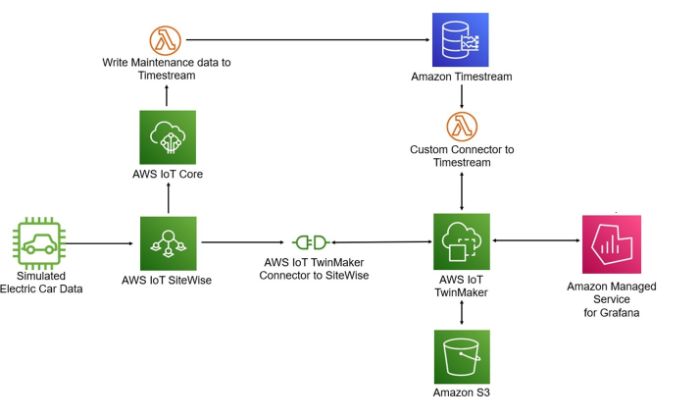
Cluster Architecture trong Kubernetes cho phép quản lý và mở rộng cụm dễ dàng. Nó cung cấp tính năng như tự động phát hiện và khôi phục sau lỗi, mở rộng ngang tự động và cân bằng tải để đảm bảo tính sẵn sàng và hiệu suất cao cho ứng dụng container.

# GIẢI PHÁP CHO BÀI TOÁN/VẤN ĐỀ/MÔ HÌNH

## Phát biểu mô hình/bài toán trong đề tài: cụ thể, rõ ràng, có thể phát biểu bằng ngôn ngữ tự nhiên, hay mô hình toán học, …

## Giải pháp cụ thể để giải quyết mô hình/bài toán:

* Đối với đề tài theo hướng xây dựng ứng dụng phần mềm: phân tích thiết kế để đưa ra sơ đồ kiến trúc tổng thể của hệ thống để giải quyết ứng dụng phần mềm.



Hình 3.2. Sơ đồ kiến trúc ứng dụng minh họa kiến trúc ứng dụng

* + Đối với đề tài theo hướng quản trị mạng: phân tích thiết kế giải pháp (thiết kế tổng thể, thiết kế thành phần, thiết kế giao tiếp người dùng, …).

### Mô tả chương trình

#### Phân tích các chức năng hệ thống bao gồm class diagram, use case, activity diagram, sequence diagram, …. Trong phần này cần chỉ rõ các vấn đề và giải pháp: Trình bày các vấn đề gặp phải trong quá trình làm đồ án và giải pháp để giải quyết vấn đề,

### Phân tích và đánh giá kết quả đề xuất triển khai của ứng dụng, …

# KẾT QUẢ, KẾT LUẬN VÀ KIẾN NGHỊ

## KẾT QUẢ

Mô tả ngắn gọn công việc nghiên cứu khoa học đã tiến hành, các kết quả nghiên cứu khoa học hoặc kết quả thực nghiệm. Đối với các đề tài ứng dụng có kết quả là sản phẩm phần mềm phải chỉ rõ các chức năng mà chương trình đã dạt được, phạm vi ứng dụng, ưu nhựơc điểm của chương trình

* Đối với đề tài theo hướng nghiên cứu cơ bản: kết quả đạt được của tiểu luận được trình bày dưới dạng bài báo khoa học và chương trình phần mềm là giải thuật đã lựa chọn trong tiểu luận.
* Đối với đề tài theo hướng xây dựng ứng dụng phần mềm: kết quả đạt được là sản phẩm phần mềm hoàn chỉnh, kèm theo hướng dẫn sử dụng chương trình (phụ lục)
* Đối với đề tài theo hướng quản trị mạng: kết quả đạt được là triển khai giải pháp trong môi trường thực tế hoặc trên phần mềm giả lập.

## 5.2. KẾT LUẬN

* Phần này phải căn cứ vào các dẫn liệu khoa học thu được trong quá trình nghiên cứu của tiểu luận hoặc đối chiếu với kết quả nghiên cứu của các tác giả khác thông qua các tài liệu tham khảo.
* Trình bày những kết quả đạt được, những đóng góp mới và những đề xuất mới.
* Phần kết luận cần ngắn gọn, không có lời bàn và bình luận thêm.

5.3. KIẾN NGHỊ

* Kiến nghị về sử dụng kết quả nghiên cứu và những hướng nghiên cứu tiếp theo

**TÀI LIỆU THAM KHẢO**

1. Trần Thu Hà (chủ biên), Giáo trình Điện tử cơ bản, NXB Đại học Quốc gia TPHCM, 2013.
2. Lê Xuân Thế, Dụng cụ Bán dẫn và Vi mạch, NXB Giáo dục, 2009.
3. Trang web [Chuyên trang thực hành điện tử, sửa chữa điện tử gia dụng, tự học điện tử!!: Sơ đồ mạch nguồn sử dụng ic ổn áp họ 78xx và 79xx (bachkhoadientu.com)](https://www.bachkhoadientu.com/2018/07/so-do-mach-nguon-su-dung-ic-on-ap-on-ap-ho-78xx-79xx.html), truy cập ngày 26/04/2021.
4. Dey, S., Saha, S., Singh, A. K., & McDonald-Maier, K. (2021). FoodSQRBlock: Digitizing food production and the supply chain with blockchain and QR code in the cloud. *Sustainability*, *13*(6), 3486.

# PHỤ LỤC (nếu có)

* + - 1. [METRIC công bố Báo cáo Thị trường sàn TMĐT quý III/2023 | Advertising Vietnam](https://advertisingvietnam.com/metric-cong-bo-bao-cao-thi-truong-san-tmdt-quy-iii2023-p23110)
      2. J. Thönes, "Microservices," in IEEE Software, vol. 32, no. 1, pp. 116-116, Jan.-Feb. 2015, doi: 10.1109/MS.2015.11