|  |
| --- |
| BAN CƠ YẾU CHÍNH PHỦ  **HỌC VIỆN KỸ THUẬT MẬT MÃ**  ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯  Logo HvKTMM |
| ĐỒ ÁN TỐT NGHIỆP  **NGHIÊN CỨU GIẢI PHÁP TRAEFIK CHO CÂN BẰNG TẢI VÀ REVERSE PROXY CHO MICROSERVICE** |
| Ngành: An toàn thông tin  Mã số: 7.48.02.02  *Sinh viên thực hiện*:  **Cao Minh Tiệp**  Lớp: AT14A  MSSV: AT140146  *Người hướng dẫn*:  **ThS.Nguyễn Xuân Thu**  Đại học Kỹ thuật - Hậu cần CAND    **Hà Nội, 2022** |

|  |
| --- |
| BAN CƠ YẾU CHÍNH PHỦ  **HỌC VIỆN KỸ THUẬT MẬT MÃ**  ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯  Logo HvKTMM |
| ĐỒ ÁN TỐT NGHIỆP  **NGHIÊN CỨU GIẢI PHÁP TRAEFIK CHO CÂN BẰNG TẢI VÀ REVERSE PROXY CHO MICROSERVICE** |
| Ngành: An toàn thông tin  Mã số: 7.48.02.02  *Sinh viên thực hiện*:  **Cao Minh Tiệp**  Lớp: AT14A  MSSV: AT140146  *Người hướng dẫn*:  **ThS.Nguyễn Xuân Thu**  Đại học Kỹ thuật - Hậu cần CAND    **Hà Nội, 2022** |

# 

# LỜI CẢM ƠN

Sau khoảng thời gian nghiên cứu và tìm hiểu, được sự giúp đỡ và dạy bảo tận tình của thầy giáo ThS. Nguyễn Xuân Thu em đã hoàn thành đồ án tốt nghiệp của mình. Em xin gửi lời cảm ơn sâu sắc và chân thành nhất tới thầy đã hết lòng giúp đỡ em trong suốt quá trình vừa qua. Em cũng gửi lời cảm ơn tới tất cả các thầy cô và bạn bè trong suốt 5 năm học tại học viện đã giúp em có được các kiến thức vững vàng để trang bị cho con đường tương lai sau khi ra trường.

Dù đã rất cố gắng và tích cực học tập, nhưng do khả năng nhìn nhận vấn đề vẫn còn đôi chút hạn chế, nên đồ án của em chắc chắn không tránh khỏi thiếu sót. Em rất mong nhận được sự đóng góp của thầy cô và bạn bè để có thể hoàn thiện tốt hơn đồ án của mình.

*Em xin chân thành cảm ơn!*

|  |  |
| --- | --- |
|  | **SINH VIÊN THỰC HIỆN ĐỒ ÁN**  **Cao Minh Tiệp** |

# LỜI CAM ĐOAN

Em xin cam đoan bản đồ án này do em tự nghiên cứu dưới sự hướng dẫn của thầy giáo ThS.Nguyễn Xuân Thu.

Để hoàn thành đồ án này, em chỉ sử dụng những tài liệu đã ghi trong mục tài liệu tham khảo, ngoài ra không sử dụng bất cứ tài liệu nào khác mà không được ghi.

Nếu sai, em xin chịu mọi hình thức kỷ luật theo quy định của Học viện.

**SINH VIÊN THỰC HIỆN**

**Cao Minh Tiệp**

# MỤC LỤC

[**LỜI CẢM ƠN i**](#_Toc106225151)

[**LỜI CAM ĐOAN ii**](#_Toc106225152)

[**MỤC LỤC iii**](#_Toc106225153)

[**DANH MỤC KÝ HIỆU VIẾT TẮT v**](#_Toc106225154)

[**DANH MỤC BẢNG BIỂU vi**](#_Toc106225155)

[**DANH MỤC HÌNH VẼ vii**](#_Toc106225156)

[**LỜI NÓI ĐẦU ix**](#_Toc106225157)

[**CHƯƠNG 1. TỔNG QUAN VỀ MICROSERVICE VÀ CÂN BẰNG TẢI TRONG MICROSERVICE 1**](#_Toc106225158)

[1.1. Microservice 1](#_Toc106225159)

[*1.1.1. Giới thiệu về Microservice* 1](#_Toc106225160)

[*1.1.2. Kiến trúc Microservice* 6](#_Toc106225161)

[1.2. Cân bằng tải trong Microservice 12](#_Toc106225162)

[*1.2.1. Cân bằng tải* 12](#_Toc106225163)

[*1.2.2. Cân bằng tải trong kiến trúc Microservice* 18](#_Toc106225164)

[1.3. Các phương pháp cân bằng tải trong Microservice 19](#_Toc106225165)

[*1.3.1. Phân loại các phương pháp cân bằng tải cho kiến trúc Microservice 19*](#_Toc106225166)

[*1.3.2.* *Kỹ thuật cân bằng tải chính cho kiến trúc Microservice* 20](#_Toc106225167)

[1.4. Kết luận chương 1 22](#_Toc106225168)

[**CHƯƠNG 2. NGHIÊN CỨU GIẢI PHÁP TRAEFIK CÂN BẰNG TẢI CHO MICROSERVICE 23**](#_Toc106225169)

[2.1. Giới thiệu tổng về Traefik 23](#_Toc106225170)

[*2.1.1. Khái niệm về Traefik* 23](#_Toc106225171)

[*2.1.2. Tầm quan trọng của Traefik đối với Microservice 23*](#_Toc106225172)

[*2.1.3. Các tính năng của Traefik 24*](#_Toc106225173)

[2.2. Kiến trúc của Traefik 24](#_Toc106225174)

[*2.2.1. Các điểm truy cập (EntryPoints) 25*](#_Toc106225175)

[*2.2.2. Các Bộ Định Tuyến (Routers) 27*](#_Toc106225176)

[*2.2.3. Services 30*](#_Toc106225177)

[*2.2.4. Các nhà cung cấp (Providers)* 33](#_Toc106225178)

[*2.2.5. Phần mềm trung gian (MiddleWare)* 36](#_Toc106225179)

[2.3. Phân Tích Source Code Về Load Balance 39](#_Toc106225180)

[*2.3.1. Hàm New 40*](#_Toc106225181)

[*2.3.2. Hàm Push 41*](#_Toc106225182)

[*2.3.3. Hàm Pop 41*](#_Toc106225183)

[*2.3.4. Hàm RegisterStatusUpdater 41*](#_Toc106225184)

[*2.3.5. Hàm nextServer 42*](#_Toc106225185)

[*2.3.6. Hàm addService 43*](#_Toc106225186)

[2.3. Phát triển, tùy biến một số tính năng Traefik 43](#_Toc106225187)

[*2.3.1. Thay đổi giao diện* 43](#_Toc106225188)

[*2.3.2. Tùy biến tính năng HealthCheck 48*](#_Toc106225189)

[2.4. Kết luận chương 2 51](#_Toc106225190)

[**CHƯƠNG 3. TRIỂN KHAI 52**](#_Toc106225191)

[3.1. Mô tả bài toán 52](#_Toc106225192)

[3.2. Xây dựng kịch bản 52](#_Toc106225193)

[*3.2.1. Xây dựng kịch bản cho bài toán 1* 52](#_Toc106225194)

[*3.2.2. Xây dựng kịch bản cho bài toán 2* 52](#_Toc106225195)

[3.3. Triển khai 53](#_Toc106225196)

[*3.3.1. Bài toán 1: Cân bằng tải cho server (Round Robin) 53*](#_Toc106225197)

[*3.2.2. Bài toán 2: Cân bằng tải cho service (Weight Round Robin) 57*](#_Toc106225198)

[3.4. Kết luận chương 3 64](#_Toc106225199)

[**KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN ĐỀ TÀI 65**](#_Toc106225200)

[**TÀI LIỆU THAM KHẢO 66**](#_Toc106225201)

# DANH MỤC KÝ HIỆU VIẾT TẮT

|  |  |  |
| --- | --- | --- |
| **Viết tắt** | **Tiếng anh** | **Tiếng Việt** |
| VIP | Virtual IP address | Địa chỉ IP ảo |
| LB | Load balancing | Cân bằng tải |
| SLB | Server load balancing | Cân bằng tải máy chủ |
| GSLB | Global server load balancing | Cân bằng tải máy chủ toàn cầu |
| SDN | Software-defined networking | Mạng do phần mềm xác định |
| TLS | Transport Layer Security | Bảo mật tầng vận chuyển |
| CI | Continuous Integration | Hội nhập liên tục |
| CD | Continuous Delivery | Chuyển giao liên tục |
| API | Application Programming Interface | Giao diện lập trình ứng dụng |

# DANH MỤC BẢNG BIỂU

[Bảng 2. 1:Các bộ kết hợp có sẵn 28](#_Toc106149206)

[Bảng 2. 2: Danh sách các nhà cung cấp hiện được hỗ trợ trong Traefik 34](#_Toc106149207)

[Bảng 2. 3: Danh sách phần mềm trung gian HTTP có sẵn 38](#_Toc106149208)

[Bảng 2. 4: Danh sách phần mềm trung gian TCP có sẵn 39](#_Toc106149209)

DANH MỤC HÌNH VẼ

[Hình 1. 1. Kiến trúc Microservice 1](#_Toc106148318)

[Hình 1. 2. Các kiểu kiến trúc 6](#_Toc106148319)

[Hình 1. 3. Biểu đồ load balancing 13](#_Toc106148320)

[Hình 1. 4. Mô hình cân bằng tải phía máy chủ 18](#_Toc106148321)

[Hình 1. 5. Mô hình cân bằng tải phía máy khách 19](#_Toc106148322)

[Hình 2. 1. Mô hình tổng quan của Traefik………………………… ………23](#_Toc106148326)

[Hình 2. 2. Kiến trúc của Traefik 25](#_Toc106148327)

[Hình 2. 3. Sơ đồ EntryPoints trong Traefik 25](#_Toc106148328)

[Hình 2. 4. Sơ đồ Router trong Traefik 27](#_Toc106148329)

[Hình 2. 5. Sơ đồ EntryPoints trong Traefik 30](#_Toc106148330)

[Hình 2. 6. Sơ đồ Provider trong Traefik 33](#_Toc106148331)

[Hình 2. 7. Sơ đồ MiddleWare trong Traefik 36](#_Toc106148332)

[Hình 2. 8. File cấu hính YAML của MiddleWare 37](#_Toc106148333)

[Hình 2. 9. File cấu hính TOML của MiddleWare 37](#_Toc106148334)

[Hình 2. 10. Giao diện của TCP ban đầu 44](#_Toc106148335)

[Hình 2. 11. Giao diện của TCP sau khi thay đổi 44](#_Toc106148336)

[Hình 2. 12. Giao diện của UDP ban đầu 44](#_Toc106148337)

[Hình 2. 13. Giao diện của UDP sau khi thay đổi 45](#_Toc106148338)

[Hình 2. 14. Giao diện của HTTP ban đầu 45](#_Toc106148339)

[Hình 2. 15. Giao diện của HTTP sau khi thay đổi 45](#_Toc106148340)

[Hình 2. 16. Giao diện của HTTP Routers ban đầu 46](#_Toc106148341)

[Hình 2. 17. Giao diện của HTTP Routers sau khi thay đổi 46](#_Toc106148342)

[Hình 2. 18. Giao diện của HTTP Services ban đầu 46](#_Toc106148343)

[Hình 2. 19. Giao diện của HTTP Services sau khi thay đổi 47](#_Toc106148344)

[Hình 2. 20. Giao diện của HTTP MiddleWares ban đầu 47](#_Toc106148345)

[Hình 2. 21. Giao diện của HTTP MiddleWares sau khi thay đổi 47](#_Toc106148346)

[Hình 2. 22. Kiểm tra trạng thái của Server sau 3 lần false 50](#_Toc106148347)

[Hình 2. 23. Kiểm tra trạng thái của Server sau 3 lần false 50](#_Toc106148348)

[Hình 3. 1. Kết quả thực thi kịch bản 1…………………………… ………..53](#_Toc106149056)

[Hình 3. 2. Kết quả thực thi kịch bản 2 54](#_Toc106149057)

[Hình 3. 3. Kết quả thực thi kịch bản 3 55](#_Toc106149058)

[Hình 3. 4. Kết quả thực thi kịch bản 4 56](#_Toc106149059)

[Hình 3. 5. File cấu hình cho kịch bản 5 58](#_Toc106149060)

[Hình 3. 6. Kết quả thực thi kịch bản 5 58](#_Toc106149061)

[Hình 3. 7. Số request đến server 1 của service tiepcao\_v1 59](#_Toc106149062)

[Hình 3. 8. Số request đến server 2 của service tiepcao\_v1 59](#_Toc106149063)

[Hình 3. 9. Số request đến server 1 của service tiepcao\_v2 60](#_Toc106149064)

[Hình 3. 10. Số request đến server 2 của service tiepcao\_v2 60](#_Toc106149065)

[Hình 3. 11. File cấu hình cho kịch bản 6 61](#_Toc106149066)

[Hình 3. 12. Kết quả thực thi kịch bản 6 62](#_Toc106149067)

[Hình 3. 13. Số request đến server 1 của service tiepcao\_v1 62](#_Toc106149068)

[Hình 3. 14. Số request đến server 2 của service tiepcao\_v1 63](#_Toc106149069)

[Hình 3. 15. Số request đến server 1 của service tiepcao\_v2 63](#_Toc106149070)

[Hình 3. 16. Số request đến server 2 của service tiepcao\_v2 64](#_Toc106149071)

# LỜI NÓI ĐẦU

Bạn có thể đã chọn kiến trúc Microservices vì nhiều lý do, đặc biệt là để tận dụng các tính năng của công nghệ đám mây. Kiến trúc Microservices cung cấp nhiều lợi thế như dịch vụ được kết hợp lỏng lẻo được phân phối, phát triển theo định hướng kinh doanh hoặc miền và sự nhanh nhẹn để phân phối liên tục, v.v.

Mọi thứ đang hoạt động tốt vào thời điểm này, nếu tất cả dịch vụ hoạt động trên một phiên bản trong môi trường Microservices của bạn.

Còn khi bạn cần mở rộng một vài dịch vụ để xử lý nhiều yêu cầu hơn. Phiên bản nào nhận được yêu cầu? hoặc Làm thế nào để khách hàng biết họ sẽ gửi yêu cầu đến phiên bản nào?

Phân phối công việc trong Microservices rất phức tạp nếu không có chiến lược phù hợp, bạn có nguy cơ khối lượng công việc sẽ không mở rộng hiệu quả.

Điện toán đám mây và ảo hóa cho phép các nhóm phần mềm mở rộng quy mô số lượng phiên bản hệ thống lưu trữ để đáp ứng với những thay đổi về khối lượng công việc. Ví dụ thêm hoặc bớt một thành phần như Microservice, tuy nhiên không tự động thay đổi cách phân phối công việc. Để kiểm soát phân phối công việc, bạn cần cân bằng tải.

Chiến lược cân bằng tải Microservices chủ yếu phụ thuộc vào cách bạn phát triển các Microservices và cách hoạt động của khám phá dịch vụ. Mong đợi bộ cân bằng tải của bạn phân phối công việc theo cách gần như ngẫu nhiên và không nhất thiết phải tính đến trạng thái hiện tại của tất cả các trường hợp.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **SINH VIÊN THỰC HIỆN ĐỒ ÁN**  Cao Minh Tiệp |  |  |  |  |

# CHƯƠNG 1. TỔNG QUAN VỀ MICROSERVICE VÀ CÂN BẰNG TẢI TRONG MICROSERVICE

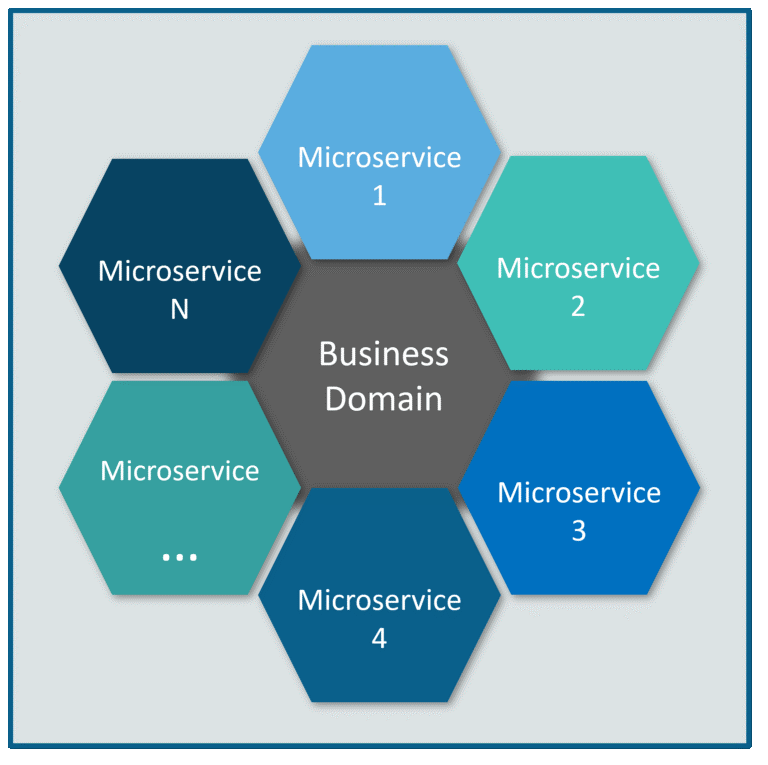
## Microservice

* + 1. ***Giới thiệu về Microservice***

1. *Khái niệm về Microservice*

Microservices là tên gọi của dịch vụ nhỏ thuộc dạng tách biệt đại diện cho 1 phần nhỏ tương ứng bên trong các Business domain của lập trình viên. Với kiến trúc Monolithic thì bạn sẽ sở hữu một server lớn với khả năng chịu mọi trách nhiệm giải quyết hầu hết các requests. Và việc này sẽ gây ra khá nhiều khó khăn trên các phương tiện đối với tất cả requests.

Chính vì vậy, Microservices được xem như giải pháp có thể cân bằng được tất cả các traffic dựa theo yêu cầu của doanh nghiệp. Và nếu như bạn đang nhận một lượng lớn các thanh toán thì hầu hết bạn sẽ có thể scale up thiết bị thanh toán và giữ cho dịch vụ nằm ở mức sử dụng 1 lượng nhỏ hơn so với các services.



Hình 1. . Kiến trúc Microservice

1. *Những lợi ích của Microservices*

Những lợi ích mà Microservices đem lại:

* Source code tinh gọn: do hệ thống cấu thành từ những dự án nhỏ, mỗi dự án sẽ đơn giản hơn và tập trung vào 1 hoặc 1 vài nghiệp vụ chính nên source code gọn nhẹ hơn. Giúp mang lại tính gọn nhẹ, dễ bảo trì và mở rộng.
* Bảo mật source code: vì developer nào làm việc ở dự án nào thì chỉ truy cập được source code trong dự án đó.
* Tồn tại độc lập: vì các dự án khác nhau và có thể có cách deploy riêng, và một service nào đó chết thì service khác vẫn hoạt động bình thường.
* Scale độc lập: theo những nhu cầu sử dụng của hệ thống mà có thể scale riêng service đó.

#### *Ưu và nhược điểm của Microservices*

* Ưu điểm

Một ứng dụng “nguyên khối” truyền thống chứa mã ở một vị trí duy nhất cho tất cả các services cần thiết để thực hiện các chức năng của nó.

Để bắt đầu, điều đó thường hoạt động tốt. Nhưng điều gì sẽ xảy ra nếu một services cụ thể cần được mở rộng để đáp ứng nhu cầu? Nói chung, một phiên bản mới của toàn bộ thành phần phía máy chủ sẽ cần được tạo và công việc sẽ cần được thực hiện trên cơ sở mã để đảm bảo services mới được mở rộng phù hợp với hệ sinh thái ứng dụng rộng lớn hơn. Đây không phải là một cách sử dụng tốt các nguồn lực.

Ngoài ra, nếu một ứng dụng phát triển theo thời gian, các nhóm lập trình viên độc lập có thể sẽ chịu trách nhiệm làm việc trên các chức năng khác nhau. Ngay cả khi có ý định tốt nhất, các nhóm này khó có thể làm việc trên cùng một cơ sở mã tại cùng một thời điểm trong khi duy trì giao tiếp mạnh mẽ. Ngay cả khi có thể giao tiếp hoàn hảo, rất nhiều thời gian và sự nhanh nhẹn chắc chắn sẽ bị mất đi.

Đây là những vấn đề mà kiến ​​trúc Microservices được thiết kế để giải quyết. Ngoài ra, bởi vì mỗi services có cơ sở mã riêng của nó, các Microservices có thể được các nhóm độc lập sửa đổi hoặc mở rộng quy mô riêng lẻ mà không gây ra các vấn đề về khả năng tương thích rộng hơn hoặc yêu cầu giao tiếp giữa các nhóm.

Các lợi thế khác của Microservices bao gồm:

* + Khả năng sử dụng các ngôn ngữ lập trình khác nhau cho các services khác nhau (còn gọi là lập trình đa ô vuông)
  + Cách ly lỗi đơn giản hơn nhiều
  + Dữ liệu được liên kết để mỗi services có thể áp dụng mô hình dữ liệu thích hợp nhất.
  + Cho phép phát triển nhanh, song song bởi các nhóm lập trình riêng biệt
  + Không có phụ thuộc chéo giữa các cơ sở mã.
* Nhược điểm

Các nhà phát triển phần mềm phải đối mặt với sự phức tạp của việc tạo ra một hệ thống phân tán:

* + Cần phải implement việc communication giữa các inter-services.
  + Việc Handle partial failure là rất phức tạp vì một luồng xử lý cần đi qua nhiều services.
  + Việc thực hiện những requests ở nhiều services khó khăn hơn, nó cũng đòi hỏi sự phối hợp cẩn thận giữa các teams.
  + Những khó khăn trong việc đảm bảo toàn vẹn CSDL nếu triển khai theo kiến trúc cơ sở dữ liệu phân vùng.
  + Việc triển khai và quản lý Microservices nếu làm theo cách thủ công so với ứng dụng một khối phức tạp hơn nhiều.
  + Phải xử lý sự cố khi kết nối chậm.

#### *Phân loại Microservice*

Có hai loại Microservice chính: Stateless Microservices (không trạng thái) và Stateful Microservices (trạng thái). Sự khác biệt cốt lõi giữa 2 kiểu khi xuống liệu các services có giữ lại hồ sơ giao tiếp giữa mỗi mô-đun hay không. Ở đây, chúng ta sẽ tìm hiểu những ưu và nhược điểm của mỗi loại.

* Stateless Microservices

Một trong những loại Microservice chính, Microservice không trạng thái, là lý tưởng cho các hệ thống phân tán. Trong trường hợp này, mỗi yêu cầu được coi là một đơn vị độc lập không có bối cảnh từ các lần lặp trước đó. Lợi ích là các thành phần không trạng thái có thể dễ dàng được thay thế, mà không có nguy cơ truyền lại các vấn đề ẩn giấu bên trong mã cũ.

Microfort Stervice không giữ hồ sơ trạng thái phiên giữa các yêu cầu. Điều này có nghĩa là nếu một phần của một cá thể services bị xóa, nó sẽ không ảnh hưởng đến logic xử lý cho services đó.

* Stateful Microservices

Trên Flipside, các services trạng thái là một loại Microservices mà lưu trữ thông tin phiên ngay tại đó trong mã. Bằng cách này, khi hai hoặc nhiều services giao tiếp, họ tự động giữ hồ sơ services được thực hiện.

Mặc dù có rất nhiều trường hợp sử dụng trong đó các services không trạng thái có ý nghĩa, nhưng có nhiều lý do mà các tổ chức cần giữ lại thông tin đó.

Ví dụ, một cái gì đó như ngân hàng trực tuyến không trạng thái sẽ không hoạt động, vì các services sẽ không giữ hồ sơ lịch sử về tiền gửi và rút tiền .

#### Cách các Microservice giao tiếp với nhau

Một số loại liên kết thường dùng của các ứng dụng Microservices hiện nay.

* Gửi Tin Bất Đồng Bộ – AMQP, STOMP, MQTT

Trong một số điều kiện, hoàn cảnh việc truyền tin bất đồng bộ là cần thiết (client không mong đợi response ngay lập tức, hoặc không cần response). Những giao thức truyền tin bất đồng bộ như AMQP, STOMP hay MQTT được sử dụng rộng rãi.

* Gửi Tin Đồng Bộ – REST, Thrift

Truyền tin đồng bộ (người gửi – client sẽ đợi một khoảng thời gian để nhận kết quả từ service). REST là sự lựa chọn ưu tiên hàng đầu vì nó cung cấp hệ thống truyền tin đơn giản qua giao thức HTTP dạng request – response. Vì thế, nhiều Microservices sử dụng HTTP với API.

* Các Kiểu Tin Nhắn – JSON, XML, Thrift, ProtoBuf, Avro

Với kiểu tin nhắn phù hợp cho Microservices và cũng là một yếu tố quan trọng. Với phần lớn các ứng dụng Microservices, thường sử dụng những kiểu tin nhắn dạng chữ như JSON và XML trên nền giao thức HTTP với API. Trong các trường hợp cần truyền tin dạng nhị phân, Microservices có thể dùng dạng Thrift, Proto hay Avro.

* Service Contracts – Định Nghĩa Service Interfaces – Swagger, RAML

Khi bạn đang có một nghiệp vụ được xây dựng như một services thì bạn cần định nghĩa và thông báo hợp đồng services (service contract thể hiện giao kèo của service). Vì chúng ta xây dựng Microservices trên kiểu kiến trúc REST nên ta có thể sử dụng cùng kiểu REST API để định nghĩa hợp đồng của Microservices. Vì thế Microservices sử dụng các ngôn ngữ định nghĩa REST API

#### Cách chọn kích thước cho Microservices

* Dựa trên số dòng mã
* Dựa trên những gì có thể được xây dựng hoặc xây dựng lại trong một Sprint duy nhất

Cả hai cách tiếp cận này đều thất bại vì về cơ bản chúng không sử dụng kiến ​​trúc Microservices: Chia ứng dụng thành các chức năng riêng lẻ của nó để tăng tốc độ phát triển và cải thiện tính nhanh nhẹn.

Một trong những cách tốt nhất để xác định quy mô services là sử dụng một khái niệm từ Domain-Driven Design (DDD) được gọi là Bối cảnh giới hạn. Bối cảnh giới hạn đặt ranh giới của các miền phụ trong một miền phức tạp dựa trên nhu cầu kinh doanh. Mỗi ngữ cảnh đư+ợc giới hạn xác định ngôn ngữ sẽ hợp lệ trong một miền phụ, ranh giới của miền phụ đó nằm ở đâu và cách nó sẽ tích hợp với các miền phụ khác.

Bối cảnh giới hạn chứa tất cả logic miền cần thiết để đáp ứng một nhu cầu hoặc chức năng cụ thể, làm cho chúng trở thành một phương tiện tuyệt vời để xác định kích thước phù hợp cho các services riêng lẻ trong kiến ​​trúc Microservices. Cách sắp xếp này gần giống với cách tiếp cận Unix: Do one thing and do it well.

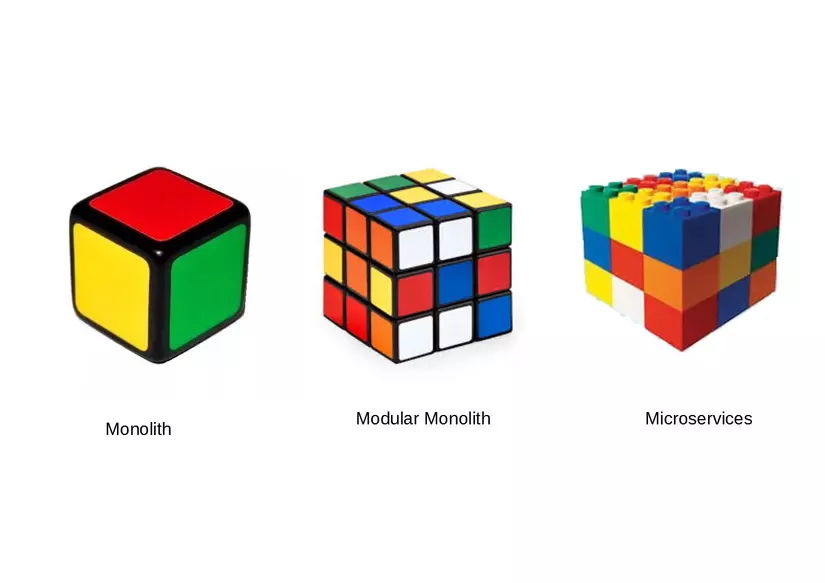
* + 1. ***Kiến trúc Microservice***

#### *Khái niệm về Kiến trúc Microservice*

Trong khoảng thời gian trở lại đây thì có rất nhiều công nghệ mới được ra mắt và đã ảnh hưởng rất nhiều đến việc phát triển phần mềm ví dụ như Cloud Computing, Containerization (Docker, Kubernetes), DevOps. Cùng với đó là sự xuất hiện của rất nhiều ngôn ngữ lập trình mới, ví dụ như Golang, Rust, Swift cùng với sự tiện lợi, dễ sử dụng giống như các ngôn ngữ như là JavaScript, Python. Một vài sự thay đổi về Software Development model, Waterfall software development model iần như bị loại bỏ và được thay thế bằng phương pháp phát triển phần mềm nhanh, lặp, tăng dần: Phát triển phần mềm Agile. Phần cứng máy tính ngày càng rẻ hơn, nhanh hơn và sự phát triển CPU Multi-Core, GPU. Những sự thay đổi về CSDL mới như là NoQuery, New Query nổi lên và trở thành xu hướng.

Để xử lý những vấn đề trên, cùng với những lợi thế đến từ Cloud Computing, Containerization, DevOps, modern Programming languages và đến từ nhu cầu phát triển phần mềm hiện đại (fast development, horizontal scaling), một Software Architecture Style được phát triển từ năm 2012: Microservice Architecture.

Kiến trúc Microservice cũng sử dụng cùng một kỹ thuật (Divide and conquer) để giải quyết sự phức tạo trong các hệ thống, và nó cũng giống như Monolithic architectrure nơi mà những hệ thống phức tạp được chia thành nhiều Microservices với khả năng giao tiếp thông qua những external interface. Điểm khác nhau chính giữa Modular Monolithich và Microservice Architecture là ở việc mọi Microservice đều có thể được deploy một cách độc lập, trong khi với Modular Monolith là các module phải deploy đồng thời. Các bạn có thể hiểu 1 cách đơn giản thông qua hình ảnh dưới đây



Hình 1. . Các kiểu kiến trúc

Monolithic application là một single unit (tightly coupled) giống như một miếng (cube) trong rubik. Modular application thì giống như Rubik nơi mà bao gồm rất nhiều các module nhỏ, tuy nhiên các module này không thể phân tách nhỏ mà phải deploy đồng thời. Còn Microservice thì giống như những miếng lego vậy, chúng ta có thể dễ dàng chia nhỏ và lắp chúng lại để thành một khối lớn. Do đó chúng ta có thể deploy riêng từng service, sau đó mới kết nối chúng vào với nhau, sẽ không làm giảm ảnh hưởng đến hệ thống hiện tại.

#### *Các thành phần chính trong kiến trúc Microservice*

Microservices tối đa hóa độ tin cậy của ứng dụng và tốc độ triển khai. Điều này đặc biệt quan trọng trong một thị trường đang vận động và phát triển nhanh hơn bao giờ hết. Kiến trúc chứa trong Microservices di chuyển ứng dụng đến bất cứ đâu mà không làm thay đổi hoặc phá vỡ môi trường, tạo điều kiện tăng tốc độ và giảm thời gian chết.

Dưới đây là các thành phần cốt lõi của Microservices:

* Clients: Ứng dụng khách thường phải sử dụng chức năng từ nhiều Microservice và yêu cầu cập nhật thường xuyên.
* Databases: Một service API cập nhật cơ sở dữ liệu Microservices bằng cách mang tất cả dịch vụ từ xa hỗ trợ truyền thông giữa các quá trình cho các ngăn xếp khác nhau.
* API gateway: API gateway là một mẫu thiết kế Microservices, một điểm nhập ứng dụng quan trọng cho các yêu cầu định tuyến và dịch giao thức.
* Identity providers: Một định danh Microservice phải cho phép truy cập từ máy chủ đến máy chủ và do người dùng điều khiển vào dữ liệu nhận dạng.
* Messaging formats: Microservices giao tiếp với các mẫu kiến trúc Microservice đồng bộ hoặc không đồng bộ.
* Static content: Clients nhận nội dung tĩnh thông qua dịch vụ lưu trữ dựa trên đám mây và mạng phân phối nội dung.
* Management: Thành phần này cho phép người dùng doanh nghiệp định cấu hình dịch vụ trong thời gian chạy.
* Service discovery: Một cơ chế khám phá dịch vụ để yêu cầu dịch vụ.

#### *Các đặc điểm của Microservice*

* Hợp phần hóa thông qua Services

Bản chất của Microservice ra lệnh chia nhỏ từng thành phần theo từng dịch vụ. Như vậy, nó có thể chỉ định, triển khai, tinh chỉnh và triển khai lại từng dịch vụ một cách độc lập mà không ảnh hưởng đến môi trường. Điều này có nghĩa là bạn thường có thể thay đổi một dịch vụ thay vì triển khai lại toàn bộ ứng dụng.

* Được tổ chức xung quanh Năng lực kinh doanh

Với cách tiếp cận nguyên khối, các nhóm công nghệ tập trung riêng biệt vào các khả năng liên quan đến công nghệ như giao diện người dùng, cơ sở dữ liệu và logic phía máy chủ. Mặt khác, Microservices tổ chức các nhóm chức năng chéo xoay quanh các khả năng và ưu tiên kinh doanh. Mỗi nhóm thiết kế các sản phẩm cụ thể dựa trên dịch vụ riêng lẻ giao tiếp bằng xe buýt tin nhắn.

* Sản phẩm không phải là Dự án

Trước đây, các nhà phát triển ứng dụng đã sử dụng mô hình dự án giao nhiệm vụ cho các nhóm xây dựng phần mềm, sau đó sẽ chuyển đến một tổ chức bảo trì sau khi hoàn thành. Kiến trúc Microservices ủng hộ một nhóm sở hữu sản phẩm trong suốt thời gian tồn tại của nó.

Điều này cho phép các nhà phát triển tương tác với sản phẩm của họ theo những cách mới, xem cách họ ứng xử trong quá trình sản xuất và tăng cường tiếp xúc với người dùng. Điều này có lợi cho các kỹ sư và doanh nghiệp tăng cường hợp tác và hiểu biết về các lĩnh vực của nhau.

* Điểm cuối thông minh và đường ống câm

Microservices có rất nhiều điểm chung với hệ thống UNIX truyền thống. Họ nhận được một yêu cầu, xử lý nó và tạo ra một phản hồi thích hợp. Fowler gọi cách tiếp cận này là "điểm cuối thông minh và đường ống câm". Cơ sở hạ tầng thường ngu ngốc vì nó chỉ đóng vai trò như một bộ định tuyến tin nhắn và tất cả các thiết bị thông minh đều nằm ở các điểm cuối sản xuất và tiêu thụ các điểm cuối.

* Quản trị phi tập trung

Quản trị phi tập trung là cấu trúc mặc định của kiến ​​trúc Microservice vì các nền tảng công nghệ đơn lẻ thường dẫn đến tiêu chuẩn hóa quá mức. Một lợi thế chính của Microservices so với monoliths là sử dụng các ngôn ngữ khác nhau ở những nơi chúng phù hợp nhất. Ví dụ: các services Spring Boot có thể xây dựng ứng dụng cho một thành phần, trong đó Spring Cloud bao gồm một thành phần khác.

* Quản lý dữ liệu phi tập trung

Hầu hết các Microservices đều cho phép mỗi thành phần quản lý cơ sở dữ liệu của riêng mình từ quản lý dữ liệu phi tập trung. Bạn luôn có thể sử dụng kho lưu trữ dữ liệu tốt nhất cho một dự án cụ thể trong khi loại bỏ công việc nâng cấp cơ sở dữ liệu được chia sẻ tốn nhiều thời gian.

* Tự động hóa cơ sở hạ tầng

Các chuyên gia CI / CD sử dụng tự động hóa cơ sở hạ tầng trong dịch vụ vi mô. Nó làm giảm khối lượng công việc của các nhà phát triển và cải thiện đáng kể hiệu quả của dòng thời gian triển khai.

* Thiết kế cho sự thất bại

Giống như các doanh nghiệp tốt nhất, dịch vụ vi mô có khả năng phục hồi. Vì dịch vụ độc đáo và đa dạng phải giao tiếp với nhau, nên có khả năng xảy ra thất bại. Đây là nhược điểm chính của Microservices so với monoliths vì giải pháp đòi hỏi thêm độ phức tạp.

Việc thiết lập ghi nhật ký và giám sát tinh vi là cần thiết để ngăn chặn sự cố ảnh hưởng đến người tiêu dùng. Mặc dù điều này đòi hỏi nhiều công việc hơn đối với các kỹ sư, nhưng nó có nghĩa là khả năng chống thất bại đã được phát triển tốt.

* Thiết kế Tiến hóa

Với một ngành công nghệ phát triển với tốc độ nhanh như hiện nay, thiết kế tiến hóa không còn là điều xa xỉ nữa; đó là một điều cần thiết. Nhiều loại thiết bị điện tử mới được tung ra thị trường mỗi năm và ứng dụng của bạn phải sẵn sàng để hỗ trợ chúng. Thiết kế không cấu trúc của Microservices có nghĩa là bạn có thể thay đổi ứng dụng mà không cần triển khai lại.

#### *Các trường hợp sử dụng kiến trúc Microservice*

Các thiết kế của Microservices rất nhanh nhẹn và linh hoạt và cho phép thời gian chết tối thiểu để bảo trì, sửa chữa và cập nhật.

Một số trường hợp sử dụng chính cho Microservices bao gồm:

* Xử lý dữ liệu: Các ứng dụng chạy trên kiến trúc Microservice có thể xử lý nhiều yêu cầu đồng thời hơn trong thời gian ngắn hơn, làm cho chúng hiệu quả hơn.
* Di chuyển trang web: Microservices tránh thời gian ngừng hoạt động di chuyển trang web, cấu trúc và / hoặc giao diện người dùng trang web.
* Phương tiện quy mô lớn: Nền tảng có thể xử lý vô số yêu cầu cho các miền phụ khác nhau mà không bị lỗi hoặc chậm trễ.
* Hóa đơn và giao dịch: Microservices có thể giúp thực hiện các giao dịch mạnh mẽ hơn với việc giảm lỗi ứng dụng, cho phép các công ty mở rộng quy mô mà không cần triển khai lại.
* Hệ thống lạc hậu: Phát triển một hệ thống container mới hiệu quả và hiệu quả hơn là cập nhật những nguyên khối cũ kỹ, thô kệch.

#### *Lợi ích kỹ thuật của Microservices*

Việc triển khai một kiến ​​trúc Microservices, hoặc phân hủy một khối cũ thành Microservices, có thể tăng tốc độ, tính linh hoạt và khả năng mở rộng, thường với chi phí đơn giản. Monoliths rất dễ xây dựng, triển khai và gỡ lỗi, nhưng khó mở rộng quy mô. Mặc dù kiến ​​trúc Microservice phức tạp hơn, nhưng có một số lợi ích cho các kỹ sư:

* Triển khai độc lập

Mỗi Microservice phải là một full-stack để hoạt động chính xác. Các nhóm phát triển có thể dễ dàng sửa lỗi trên một Microservice duy nhất mà không ảnh hưởng đến toàn bộ môi trường.

* Khả năng mở rộng

Các nhóm xây dựng ứng dụng với kiến ​​trúc Microservices có thể thay đổi từng dịch vụ mà không làm gián đoạn phần còn lại của ứng dụng. Các lập trình viên có thể làm việc song song để xác định dịch vụ nóng, cập nhật chức năng và mở rộng quy mô dịch vụ đã chọn.

* Tự do ngăn xếp công nghệ

Phụ thuộc vào ngăn xếp công nghệ đơn lẻ là một trong những đặc điểm hạn chế nhất của kiến ​​trúc nguyên khối. Với Microservices, các nhà phát triển có thể xây dựng một ứng dụng bằng ngôn ngữ lập trình, khuôn khổ, cơ sở dữ liệu và các công cụ front-end và backend. Điều này giúp giảm thiểu việc xâm nhập vào một chuỗi tất cả các giao dịch và không có ngăn xếp tiêu chuẩn nào.

* Cách ly lỗi

Một lỗi duy nhất thường có nghĩa là toàn bộ ứng dụng nguyên khối bị lỗi. Microservices cô lập các lỗi với các thành phần của chúng, vì vậy toàn bộ ứng dụng không bị ảnh hưởng.

* Năng suất

Microservices giảm thiểu các silo và thúc đẩy làm việc theo nhóm đa chức năng cho các lực lượng làm việc phân tán và từ xa ngày nay. Chúng có thể được kiểm tra độc lập trong khi các nhà phát triển xử lý các khía cạnh khác, hợp lý hóa việc đảm bảo chất lượng.

#### *Giá trị kinh doanh của Microservices*

Với giá trị kinh doanh, Microservices có rất nhiều lợi ích so với nguyên khối. Chúng thúc đẩy mối quan hệ cộng sinh giữa các nhà phát triển và lãnh đạo doanh nghiệp, dẫn đến kết quả tốt hơn. Các giá trị mang lại như sau :

* + Tốc độ đổi mới nhanh hơn. Thiết kế cải tiến của chúng giúp dễ dàng thay đổi và nâng cấp các thành phần ứng dụng.
  + Độ ổn định / khả năng phục hồi cao hơn. Microservices thực tế loại bỏ thời gian chết bởi vì, không giống như kiến trúc nguyên khối, dịch vụ là riêng biệt và các nhà phát triển có thể cập nhật từng thành phần một.
  + Phần mềm có thể mở rộng quy mô để theo kịp với nhu cầu kinh doanh. Các nhóm phát triển có thể làm việc trên các thành phần đơn lẻ mà không ảnh hưởng đến toàn bộ môi trường.
  + Chi phí thấp hơn và doanh thu tốt hơn. Microservices cần ít cơ sở hạ tầng tổng thể hơn, giảm thiểu thời gian chết và cải thiện độ ổn định để nâng cao trải nghiệm khách hàng và giảm nguyên mẫu ứng dụng xuống khung thời gian triển khai.
  1. **Cân bằng tải trong Microservice**
     1. ***Cân bằng tải***

#### *a. Khái niệm về Cân bằng tải*

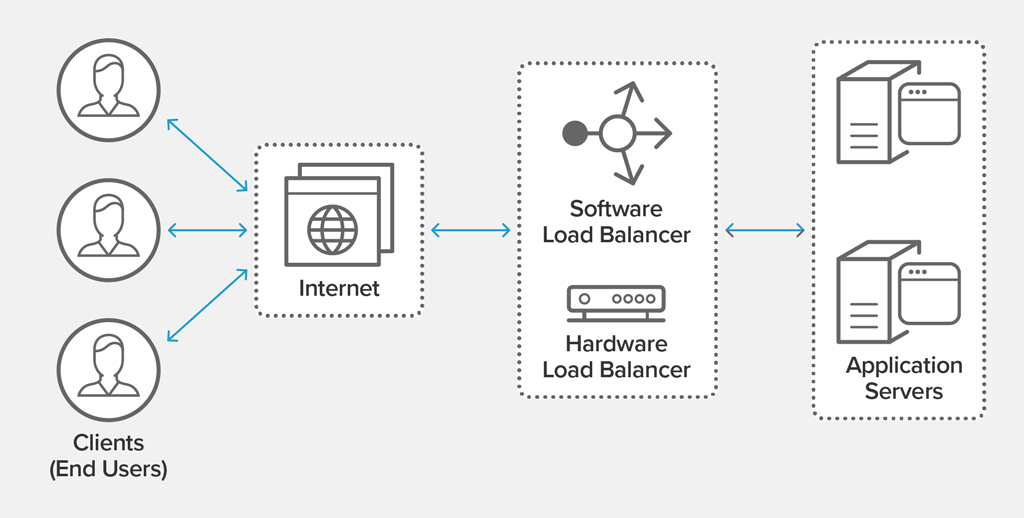
Cân bằng tải đề cập đến việc phân phối hiệu quả lưu lượng mạng đến trên một nhóm backend servers, còn được gọi là server farm hoặc server pool.

Các trang web có lưu lượng truy cập cao hiện đại phải phục vụ hàng trăm nghìn, nếu không phải hàng triệu, yêu cầu đồng thời từ người dùng hoặc khách hàng và trả lại văn bản, hình ảnh, video hoặc dữ liệu ứng dụng chính xác, tất cả đều theo cách nhanh chóng và đáng tin cậy. Để mở rộng quy mô chi phí hiệu quả nhằm đáp ứng những khối lượng lớn này, đối với các máy tính hiện đại cách tốt nhất là thường yêu cầu thêm nhiều máy chủ hơn.

Bộ cân bằng tải hoạt động như một "cảnh sát lưu lượng" ngồi trước máy chủ của bạn và định tuyến các yêu cầu của khách hàng trên tất cả các máy chủ có khả năng đáp ứng các yêu cầu đó theo cách tối đa hóa tốc độ và sử dụng dung lượng và đảm bảo rằng không có một máy chủ nào làm việc quá mức, điều này có thể làm giảm hiệu suất . Nếu một máy chủ duy nhất gặp sự cố, bộ cân bằng tải sẽ chuyển hướng lưu lượng truy cập đến các máy chủ trực tuyến còn lại. Khi một máy chủ mới được thêm vào nhóm máy chủ, bộ cân bằng tải sẽ tự động bắt đầu gửi yêu cầu đến nó.

Theo cách này, bộ cân bằng tải thực hiện các chức năng sau:

* Phân phối các yêu cầu của máy khách hoặc tải mạng một cách hiệu quả trên nhiều máy chủ
* Đảm bảo tính khả dụng và độ tin cậy cao bằng cách chỉ gửi yêu cầu đến các máy chủ trực tuyến
* Cung cấp sự linh hoạt để thêm hoặc bớt máy chủ khi nhu cầu ra lệnh



Hình 1. . Biểu đồ load balancing

#### *b. Cách cân bằng tải hoạt động*

Bộ cân bằng tải hoạt động như một proxy ngược. Nó cung cấp cho máy khách một địa chỉ IP ảo (VIP) đại diện cho ứng dụng. Máy khách kết nối với VIP và bộ cân bằng tải xác định liệu kết nối có được chuyển tiếp đến một phiên bản ứng dụng nhất định trên máy chủ hay không bằng cách sử dụng các thuật toán của nó. Trong suốt thời gian kết nối, bộ cân bằng tải quản lý và giám sát nó.

Thật vậy, nhiệm vụ của bộ cân bằng tải đôi khi được so sánh với nhiệm vụ của cảnh sát giao thông, vì nó được thiết kế để hướng các yêu cầu một cách có hệ thống đến những nơi thích hợp tại bất kỳ thời điểm nào, tránh tắc nghẽn tốn kém và các vấn đề ngoài kế hoạch. Cuối cùng, bộ cân bằng tải phải cung cấp hiệu suất và bảo mật cần thiết để hỗ trợ các môi trường công nghệ thông tin phức tạp và nhiều hoạt động xảy ra bên trong chúng.

Hãy xem xét tình huống của một đại lý thể thao đang đàm phán một thỏa thuận mới cho một vận động viên nổi tiếng. Người đại diện chấp nhận yêu cầu của vận động viên và chuyển nó đến một tổ chức cụ thể quan tâm. Nhóm cung cấp thông tin (một đề nghị) cho đại lý, sau đó họ sẽ gửi thông tin đó cho khách hàng. Điều này tiếp tục trong một thời gian cho đến khi tìm ra giải pháp.

Cân bằng tải máy chủ (SLB) là chức năng chính của cân bằng tải. Dựa trên vai trò của họ trong cuộc trò chuyện, người đại diện có thể cung cấp chức năng bổ sung. Họ có tùy chọn cho phép hoặc không cho phép một số chi tiết (bảo mật). Họ có thể muốn kiểm tra kỹ xem người mà họ đang nói chuyện cùng có phải là vận động viên đang được đề cập hay không (xác thực). Nếu giải đấu thể thao hiện tại không hoạt động, người đại diện có thể chuyển tiếp cuộc trò chuyện sang giải đấu khác tùy thuộc vào tình trạng sẵn có hoặc khu vực (GSLB).

Cân bằng tải là cách có thể mở rộng nhất để xử lý số lượng lớn các yêu cầu được tạo bởi quy trình làm việc đa ứng dụng, đa thiết bị ngày nay. Cân bằng tải, kết hợp với các hệ thống cung cấp quyền truy cập liền mạch vào nhiều ứng dụng, tệp và máy tính để bàn trong các nơi làm việc kỹ thuật số ngày nay, giúp nhân viên có trải nghiệm người dùng cuối nhất quán và đáng tin cậy hơn.

#### *c. Các dạng cân bằng tải*

Có một số cách tiếp cận cân bằng tải để giải quyết các vấn đề mạng:

* Bộ cân bằng tải mạng

Cân bằng tải mạng là việc phân bổ lưu lượng ở cấp độ truyền tải thông qua các quyết định định tuyến dựa trên các yếu tố mạng như địa chỉ IP và các cổng đích. Cân bằng tải như vậy là TCP (cấp 4) và không tính đến bất kỳ tham số cấp ứng dụng nào như loại nội dung, dữ liệu cookie, tiêu đề, vị trí, hành vi ứng dụng, v.v. Cân bằng tải mạng thực hiện các bản dịch địa chỉ mạng mà không cần kiểm tra nội dung của các gói rời rạc, chỉ tập trung vào thông tin lớp mạng và định hướng lưu lượng cho phù hợp.

* Cân bằng tải ứng dụng

Còn được gọi là bộ cân bằng tải Lớp 7, cao nhất trong mô hình OSI, phân phối các yêu cầu dựa trên một số tiêu chí ở cấp ứng dụng. Bộ cân bằng tải L7 đánh giá nhiều loại dữ liệu hơn đáng kể, bao gồm tiêu đề HTTP và phiên SSL, đồng thời phân phối tải của máy chủ dựa trên phán đoán dựa trên sự kết hợp của nhiều biến. Bộ cân bằng tải ứng dụng điều chỉnh lưu lượng máy chủ theo cách này, dựa trên cách sử dụng và hành vi cá nhân.

* Bộ cân bằng tải máy chủ toàn cầu

Với ngày càng nhiều ứng dụng được lưu trữ trong các trung tâm dữ liệu đám mây trên toàn cầu, GSLB mở rộng khả năng của L4 và L7 chung trên một số trung tâm dữ liệu, cho phép phân phối tải toàn cầu hiệu quả hơn mà không ảnh hưởng đến trải nghiệm của người dùng cuối. Bộ cân bằng tải đa điểm hỗ trợ khôi phục nhanh chóng và hoạt động của công ty liền mạch trong trường hợp máy chủ gặp sự cố hoặc thảm họa tại bất kỳ trung tâm dữ liệu nào, vì các trung tâm dữ liệu bổ sung ở bất kỳ nơi nào trên thế giới đều có thể được sử dụng cho hoạt động kinh doanh liên tục.

#### *d. Các thuật toán cân bằng tải*

* Round Robin

Cân bằng tải round-robin là một trong những thuật toán cân bằng tải đơn giản và được sử dụng nhiều nhất. Các yêu cầu của máy khách được phân phối đến các máy chủ ứng dụng theo vòng quay. Ví dụ: nếu bạn có ba máy chủ ứng dụng: máy khách đầu tiên yêu cầu máy chủ ứng dụng đầu tiên trong danh sách, máy khách thứ hai yêu cầu máy chủ ứng dụng thứ hai, máy khách thứ ba yêu cầu máy chủ ứng dụng thứ ba, máy chủ thứ tư cho máy chủ ứng dụng đầu tiên. và như thế.

Thuật toán cân bằng tải này không xem xét các đặc điểm của máy chủ ứng dụng, tức là nó giả định rằng tất cả các máy chủ ứng dụng đều giống nhau với cùng tính khả dụng, tính toán và các đặc tính xử lý tải.

* Weighted Round Robin

Weighted Round Robin được xây dựng dựa trên thuật toán cân bằng tải Round-robin đơn giản để tính đến các đặc điểm khác nhau của máy chủ ứng dụng. Quản trị viên ấn định trọng số cho từng máy chủ ứng dụng dựa trên tiêu chí họ lựa chọn để chứng minh khả năng xử lý lưu lượng của máy chủ ứng dụng. Nếu máy chủ ứng dụng 1 mạnh gấp đôi máy chủ ứng dụng 2 (và máy chủ ứng dụng 3), thì máy chủ ứng dụng 1 được cung cấp với trọng lượng cao hơn và máy chủ ứng dụng 2 và 3 có cùng trọng lượng. Nếu có năm yêu cầu máy khách tuần tự, hai đầu tiên chuyển đến máy chủ ứng dụng 1, thứ ba chuyển đến máy chủ ứng dụng 2, thứ tư đến máy chủ ứng dụng 3 và thứ năm tới máy chủ ứng dụng 1.

* Least Connection

Cân bằng Least Connection là một thuật toán cân bằng tải động trong đó các yêu cầu của máy khách được phân phối đến máy chủ ứng dụng với số lượng kết nối hoạt động ít nhất tại thời điểm nhận được yêu cầu của máy khách. Trong trường hợp máy chủ ứng dụng có thông số kỹ thuật tương tự, máy chủ ứng dụng có thể bị quá tải do các kết nối tồn tại lâu hơn; thuật toán này xem xét tải kết nối đang hoạt động.

* Weighted Least Connection

Weighted Least Connection được xây dựng dựa trên thuật toán cân bằng tải Least Connection để tính đến các đặc điểm khác nhau của máy chủ ứng dụng. Quản trị viên ấn định trọng số cho từng máy chủ ứng dụng dựa trên tiêu chí họ lựa chọn để chứng minh khả năng xử lý lưu lượng của máy chủ ứng dụng. LoadMaster đang đưa ra các tiêu chí cân bằng tải dựa trên các kết nối đang hoạt động và trọng số của máy chủ ứng dụng.

* Resource Based (Adaptive)

Resource Based (Adaptive) là một thuật toán cân bằng tải yêu cầu một tác nhân được cài đặt trên máy chủ ứng dụng báo cáo về tải hiện tại của nó cho bộ cân bằng tải. Tác nhân được cài đặt giám sát trạng thái sẵn có của máy chủ ứng dụng và tài nguyên. Bộ cân bằng tải truy vấn đầu ra từ tác nhân để hỗ trợ các quyết định cân bằng tải.

* Resource Based (SDN Adaptive)

SDN Adaptive là một thuật toán cân bằng tải kết hợp kiến ​​thức từ Layers 2, 3, 4 và 7 và đầu vào từ Bộ điều khiển SDN để đưa ra quyết định phân phối lưu lượng tối ưu hơn. Điều này cho phép thông tin về trạng thái của các máy chủ, trạng thái của các ứng dụng đang chạy trên chúng, sức khỏe của cơ sở hạ tầng mạng và mức độ tắc nghẽn trên mạng đều đóng một phần trong việc đưa ra quyết định cân bằng tải.

* Fixed Weighting

Trọng số cố định là một thuật toán cân bằng tải trong đó quản trị viên chỉ định trọng số cho từng máy chủ ứng dụng dựa trên tiêu chí họ chọn để chứng minh khả năng xử lý lưu lượng của máy chủ ứng dụng. Máy chủ ứng dụng có trọng số cao nhất sẽ nhận được tất cả lưu lượng truy cập. Nếu máy chủ ứng dụng có trọng số cao nhất bị lỗi, tất cả lưu lượng sẽ được chuyển hướng đến máy chủ ứng dụng có trọng số cao nhất tiếp theo.

* Weighted Response Time

Weighted Response Time là một thuật toán cân bằng tải trong đó thời gian phản hồi của các máy chủ ứng dụng xác định máy chủ ứng dụng nào nhận được yêu cầu tiếp theo. Thời gian phản hồi của máy chủ ứng dụng để kiểm tra tình trạng được sử dụng để tính toán trọng lượng của máy chủ ứng dụng. Máy chủ ứng dụng đang phản hồi nhanh nhất sẽ nhận được yêu cầu tiếp theo.

* Source IP Hash

Thuật toán cân bằng tải Source IP Hash kết hợp địa chỉ IP nguồn và đích của máy khách và máy chủ để tạo khóa băm duy nhất. Khóa được sử dụng để cấp phát máy khách cho một máy chủ cụ thể. Vì khóa có thể được tạo lại nếu phiên bị hỏng, yêu cầu của máy khách được chuyển hướng đến cùng một máy chủ mà nó đã sử dụng trước đó. Điều này hữu ích nếu điều quan trọng là khách hàng phải kết nối với một phiên vẫn đang hoạt động sau khi ngắt kết nối.

* URL Hash

URL Hash là một thuật toán cân bằng tải để phân phối các lần ghi đồng đều trên nhiều trang web và gửi tất cả các lần đọc đến trang web sở hữu đối tượng.

#### *e. Lợi ích của Cân bằng tải*

Chuyển hướng lưu lượng truy cập đến máy chủ hoạt động mạnh nhất tại thời điểm đó có những lợi ích sau:

* Big Data

Có thể kiểm tra thông tin chi tiết hữu ích từ dữ liệu lớn do người dùng trên toàn thế giới cung cấp để giúp các doanh nghiệp đưa ra quyết định sáng suốt hơn, tốt hơn.

* Nâng cao hiệu quả

Cân bằng tải làm giảm tải trọng gia tăng trên máy chủ và duy trì các hoạt động và phản hồi trơn tru, mang lại trải nghiệm tốt hơn cho khách hàng.

* Phân tích dự đoán

Sự tắc nghẽn lưu lượng có thể được dự đoán bởi bộ cân bằng tải phần mềm trước khi chúng xảy ra trong thế giới thực.

* Khả năng phục hồi

Với ít hoặc không có thời gian ngừng hoạt động, các thành phần bị lỗi và hoạt động kém có thể được thay thế kịp thời, cung cấp thông tin về thiết bị nào cần được bảo dưỡng.

* Bảo vệ

Load Balancer bổ sung thêm một lớp bảo mật cho trang web và ứng dụng của bạn mà không yêu cầu bất kỳ thay đổi nào.

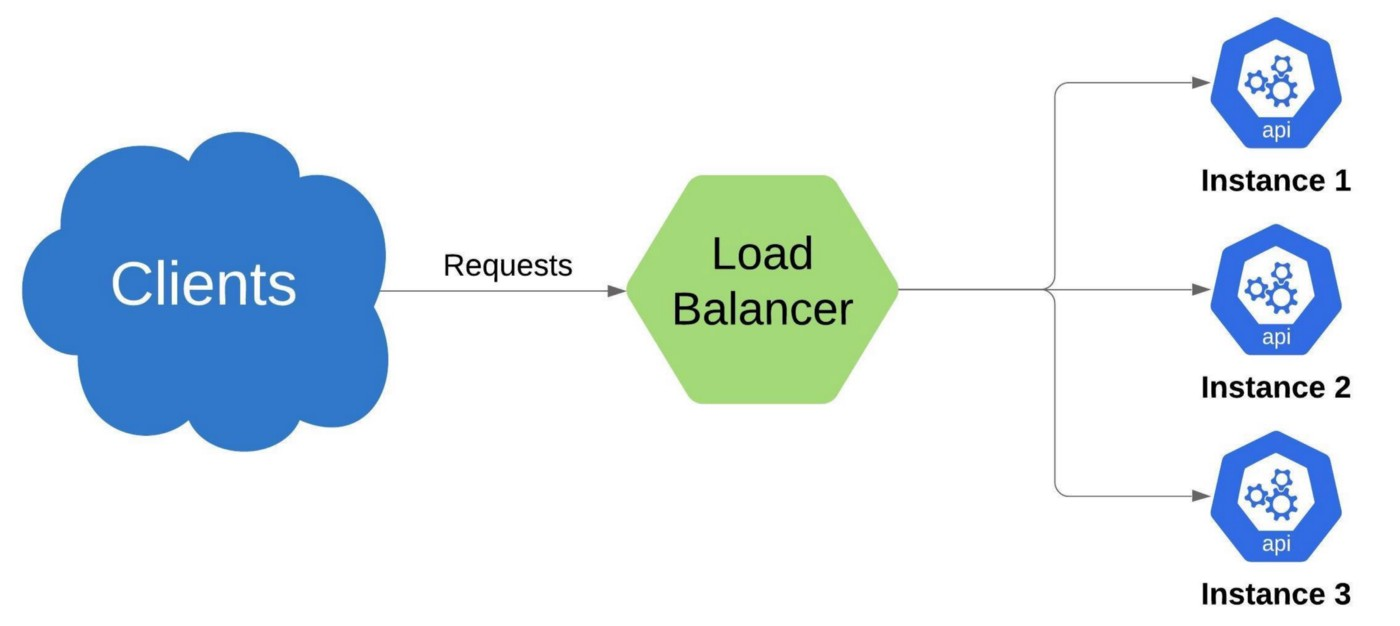
* Khả năng mở rộng

Cân bằng tải cho phép bạn thay đổi cơ sở hạ tầng máy chủ bất kỳ lúc nào mà không ảnh hưởng

* + 1. ***Cân bằng tải trong kiến trúc Microservice***

Trong kiến trúc Microservices có hai loại cân bằng tải; chúng là cân bằng tải phía máy chủ và cân bằng tải phía máy khách. Dưới đây là chi tiết về 2 loại cân bằng tải:

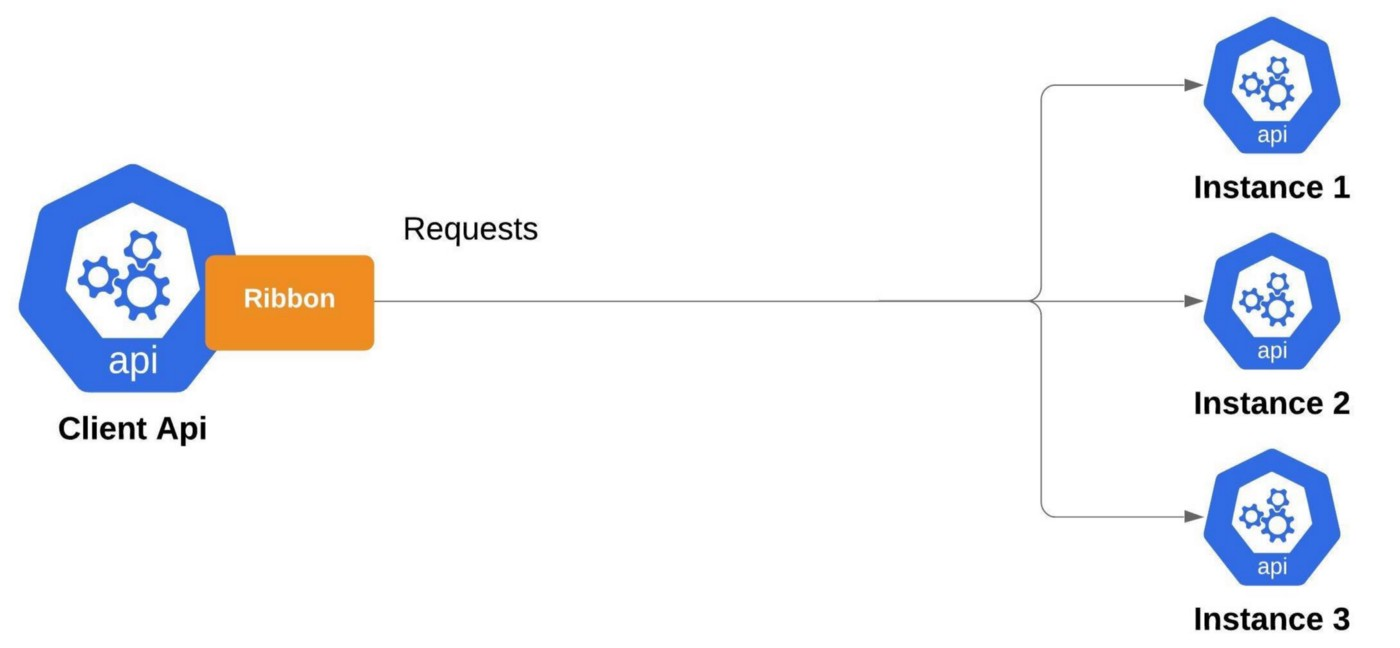
#### *Cân bằng tải phía máy chủ*



Hình 1. . Mô hình cân bằng tải phía máy chủ

Cân bằng tải phía máy chủ là một cân bằng tải cổ điển. Lưu lượng được phân phối bởi một bộ phân phối tải đặt phía trước các máy chủ và phân phối đến các máy chủ sẽ thực hiện công việc chính một cách bình đẳng hoặc theo các quy tắc nhất định. Ví dụ như các bộ cân bằng tải phía máy chủ được sử dụng phổ biến nhất nginx, netscaler, v.v.

#### *Cân bằng tải phía máy khách*



Hình 1. . Mô hình cân bằng tải phía máy khách

Trong cân bằng tải phía máy khách, máy khách xử lý việc cân bằng tải. Trong trường hợp này, api máy khách nên biết tất cả các phiên bản của địa chỉ api máy chủ thông qua mã hóa cứng hoặc bằng sổ đăng ký services.

Với phương pháp này, bạn có thể thoát khỏi tắc nghẽn và điểm thất bại duy nhất. Nếu bạn sử dụng services khám phá, bạn không phải biết bất kỳ thông tin nào về api máy chủ ngoại trừ tên đã đăng ký api, cơ chế đăng ký máy chủ sẽ cung cấp tất cả thông tin về api máy chủ.

* 1. **Các phương pháp cân bằng tải trong Microservice**

### **Phân loại các phương pháp cân bằng tải cho kiến trúc Microservice**

Theo nguyên tắc về việc liệu yêu cầu có thể được phân bổ trong thời gian thực hay không theo trạng thái máy chủ, các thuật toán cân bằng tải chủ đạo hiện tại có thể được chia thành hai danh mục: thuật toán cân bằng tải tĩnh và thuật toán cân bằng tải động. Thuật toán cân bằng tải tĩnh chủ yếu bao gồm Polling Method, Random Method, Source Address Hashing Method, Consistent Hash Method, Weighted Polling Method, etc. pháp băm nhất quán, Phương pháp thăm dò có trọng số, v.v.

Các thuật toán cân bằng tải động chủ yếu bao gồm Minimum Connection Number Method, Fastest Response Method, etc.

Theo các công nghệ khác nhau để phân bổ các yêu cầu dịch vụ, tải phổ biến công nghệ cân bằng cũng có thể được chia thành phương pháp ngẫu nhiên, phương pháp thăm dò, băm phương pháp, phương pháp tối ưu hóa và các phương pháp dựa trên trí tuệ nhân tạo.

* + 1. ***Kỹ thuật cân bằng tải chính cho kiến trúc Microservice***

#### *a. Công nghệ cân bằng tải dựa trên polling method*

Kỹ thuật cân bằng tải dựa trên polling method có thể thiết lập một nhóm máy chủ như {S0, S1, S2 Si. Sn Sn}, trong đó SI đại diện cho một dịch vụ. Khi khách hàng gửi yêu cầu, máy chủ sẽ được yêu cầu từ S0 đến SN lần lượt, cho đến khi kết thúc một vòng, sau đó tiến hành đến vòng yêu cầu tiếp theo. Những lợi thế của phương pháp bỏ phiếu là đơn giản và nguyên tắc kỹ thuật hiệu quả, trong khi những nhược điểm là tải trọng mạng và cuộc gọi liên kết các vấn đề tồn tại trong khung Microservice không được xem xét, điều này sẽ dẫn đến tải không đều đến các máy chủ và các nút làm việc.

Để giải quyết các vấn đề trên, các nhà nghiên cứu trong nước đã được cải thiện và tóm tắt nó. Bằng cách thu thập tải yêu cầu được cảm nhận bởi chuỗi Microservice, phân tích chuỗi gọi Microservice được yêu cầu và mô hình phân tích tải được xây dựng bởi Sử dụng mối quan hệ của chuỗi gọi Microservice. Sau đó, chức năng chi phí được thiết kế Để vượt qua tất cả các kết hợp gọi dịch vụ và chọn đường dẫn tối ưu với thời gian ngắn nhất thời gian cần thiết và lần lượt thực hiện lịch trình cân bằng tải, giúp giảm hiệu quả Sự chậm trễ trung bình của các yêu cầu và cải thiện đáng kể sự sẵn có của dịch vụ . Chiến lược khác là sử dụng công nghệ cân bằng tải dựa trên SDN, thông qua Phương pháp ghép mạng SDN và kiến ​​trúc Microservice, để nhận ra nhiều hơn Kiểm soát tốt và khả năng quản lý của mạng, và cấy ghép tốt hơn cân bằng tải Công nghệ cho Microservice. Đồng thời, khả năng theo dõi của SDN cho Liên kết dịch vụ được sử dụng để thực hiện thuật toán cân bằng tải dựa trên phân tích liên kết, để phân tích tải liên kết tổng thể và cung cấp chức năng giới hạn hiện tại để ngăn chặn Truy cập từ việc ảnh hưởng đến tính khả dụng cao của hệ thống quá nhiều.

#### *Công nghệ cân bằng tải dựa trên thuật toán băm*

Công nghệ cân bằng tải dựa trên thuật toán băm được sử dụng rộng rãi trong lĩnh vực tải lập lịch và tối ưu hóa trong các hệ thống phân tán, điện toán đám mây và Microservice. Tại hiện tại, nó chủ yếu bao gồm công nghệ cân bằng tải dựa trên chiến lược mô đun băm và công nghệ cân bằng tải dựa trên thuật toán băm phù hợp. Nguyên tắc công nghệ cân bằng tải dựa trên chiến lược mô đun băm là thiết kế một hàm băm thích hợp để tính toán số lượng yêu cầu và nút dịch vụ để xác định nút nào yêu cầu sẽ được phân phối cho. Đặc điểm của nó đơn giản và hiệu quả và có thể đảm bảo rằng các yêu cầu tương tự của hệ thống được xử lý cùng máy, giảm di chuyển dữ liệu và giao tiếp nút. Tuy nhiên, trong thực tế môi trường, nếu số lượng nút thay đổi động do sự thất bại của hệ thống, Các kết quả tính toán sẽ được thay đổi rất nhiều và một số lượng lớn việc di chuyển dữ liệu sẽ được tạo ra, giảm sự sẵn có của dịch vụ hệ thống. Công nghệ cân bằng tải dựa trên thuật toán băm phù hợp là một cải thiện trước đây.

Nguyên tắc thuật toán như sau: Đầu tiên, giá trị băm tương ứng của nút được tính toán và gán cho một vòng tròn 0-232. Thứ hai, tìm giá trị băm của yêu cầu dịch vụ và ánh xạ nó thành cùng một vòng tròn. Cuối cùng, việc tra cứu được thực hiện theo chiều kim đồng hồ từ vị trí của yêu cầu và lần đầu tiên mút tìm thấy là nút đã truy cập. Kỹ thuật cân bằng tải dựa trên sự nhất quán Thuật toán băm có khả năng mở rộng bên tốt. Hơn nữa, khác với thay đổi động của số lượng các nút dựa trên chiến lược mô đun băm, nó có thể giải quyết một cách hiệu quả vấn đề của các nút dữ liệu thay đổi và quyền sở hữu dữ liệu và tránh một số lượng lớn dữ liệu các vấn đề di cư gây ra bởi các vấn đề trên. Tuy nhiên, kỹ thuật cân bằng tải dựa trên thuật toán băm nhất quán có Không có biện pháp cân bằng tải, điều này sẽ gây ra độ lệch dữ liệu giữa các nút. Theo quan điểm của Các vấn đề trên, các học giả có liên quan đã thực hiện rất nhiều cải tiến, chủ yếu bao gồm: Một kỹ thuật cân bằng tải dựa trên sự phân chia và tham lam. Bằng cách chia vòng băm, Các máy chủ có thể cung cấp dịch vụ tương ứng được chia. Ngoài ra, một máy chủ vật lý là ánh xạ tới nhiều nút ảo theo các chức năng của nó. Khi một yêu cầu đến, nó đầu tiên tìm kiếm một phân vùng phù hợp và sau đó ánh xạ các đặc điểm của yêu cầu thành một hàm băm giá trị. Theo ý tưởng thuật toán tham lam, nó chọn một nút dịch vụ giữa mức thấp nhất tải và tải trọng trung bình ngược chiều kim đồng hồ .

#### *Công nghệ cân bằng tải dựa trên trí tuệ nhân tạo*

Chiến lược khác là sử dụng công nghệ cân bằng tải dựa trên SDN với sự phát triển nhanh chóng của công nghệ trí tuệ nhân tạo, thuật toán tối ưu hóa dựa trên Trí tuệ nhân tạo dần dần được áp dụng cho lĩnh vực cân bằng tải. Tải trọng điển hình Công nghệ cân bằng dựa trên Thuộc địa ong nhân tạo và công nghệ cân bằng tải trọng Dựa trên thuật toán tối ưu hóa khuẩn lạc. Công nghệ cân bằng tải dựa trên Thuộc địa ong nhân tạo có những lợi thế của dễ dàng Thực hiện và phạm vi ứng dụng rộng. Trong thuật toán này, yêu cầu dịch vụ là được coi là một con ong, và nút tải được lấy làm nguồn mật ong. Thuật toán giải quyết vấn đề cân bằng tải với ý tưởng về thuật toán thuộc địa ong nhân tạo: các nút là Sắp xếp theo tải dịch vụ từ cao đến thấp. Nếu tải của một nút quá cao, nhiệm vụ trong tải sẽ bị xóa và tác vụ sẽ được di chuyển sang nút có tải thấp hoặc ưu tiên cao và ít nhiệm vụ hơn. Tải trọng mô phỏng và ưu tiên nhiệm vụ được phát cho tất cả nhiệm vụ tính toán . Trong công nghệ cân bằng tải dựa trên thuật toán tối ưu hóa khuẩn lạc, heuristic hàm chức năng và pheromone được đặt đầu tiên, và sau đó chúng được khởi tạo theo Tài nguyên dịch vụ của nút. Để đạt được hiệu quả của cân bằng tải, Pheromone sẽ không chỉ xem xét tải các nút mà còn xem xét tính toán và Khả năng giao tiếp của các nút. Công nghệ này có thể đảm bảo tránh rơi vào vấn đề giải pháp tối ưu cục bộ và cải thiện hiệu quả độ tin cậy của hệ thống và sự sẵn có của dịch vụ .

## Kết luận chương 1

Chúng ta có cái nhìn tổng quan về Microservice và kiến trúc của hệ thống Microservice .

Với kiến trúc Microservice, mỗi ứng dụng nằm trong vùng chứa riêng của nó. Không có thời gian và tài nguyên bị lãng phí khi quản lý các tác động cập nhật ứng dụng này lên ứng dụng khác. Do đó, dịch vụ vi mô rất quan trọng trong việc giúp các công ty đạt được mục tiêu kinh doanh của họ - cho dù đó là phát triển nhanh hơn, phần mềm tốt hơn, tài năng hàng đầu hay chi phí thấp hơn.

Mặc dù phải mất thời gian, nỗ lực và tài nguyên để chuyển đổi một kiến trúc nguyên khối sang kiến trúc Microservices, nhưng phần thưởng có thể rất xứng đáng.

# CHƯƠNG 2. NGHIÊN CỨU GIẢI PHÁP TRAEFIK CÂN BẰNG TẢI CHO MICROSERVICE

## 2.1. Giới thiệu tổng về Traefik

* + 1. ***Khái niệm về Traefik***



Hình 2. . Mô hình tổng quan của Traefik

Traefik là một Reverse-proxy đời mới, và cũng là load balancer để làm cho việc deploy hệ thống Microservice được trở lên dễ dàng hơn. Tích hợp rất nhiều các thành phần infrastructure như Docker, Swarm mode, Kubernetes, Marathon, Consul, Etcd, Rancher, Amazon ECS... Và tính tự động là điểm quan trọng nhất trong các config với traefik.

### **2.1.2.** **Tầm quan trọng của Traefik đối với Microservice**

#### *a. Traefik với Microservice*

Microservice nó là một kiểu kiến trúc hệ thống để dựng lên application của chúng ta. Tại đó, application được định nghĩa dưới dạng một tập hợp các services. Mỗi service sẽ đảm nhiệm một phần chức năng trong hệ thống. Để người dùng truy cập vào hệ thống Microservice thì bạn cần một reverse proxy. Và reverse proxy chính là một trong số các vấn đề mà chúng ta gặp phải trong kiến trúc này. Traefik sinh ra để giải quyết việc đó.

#### *b. Traefik với reverse-proxy*

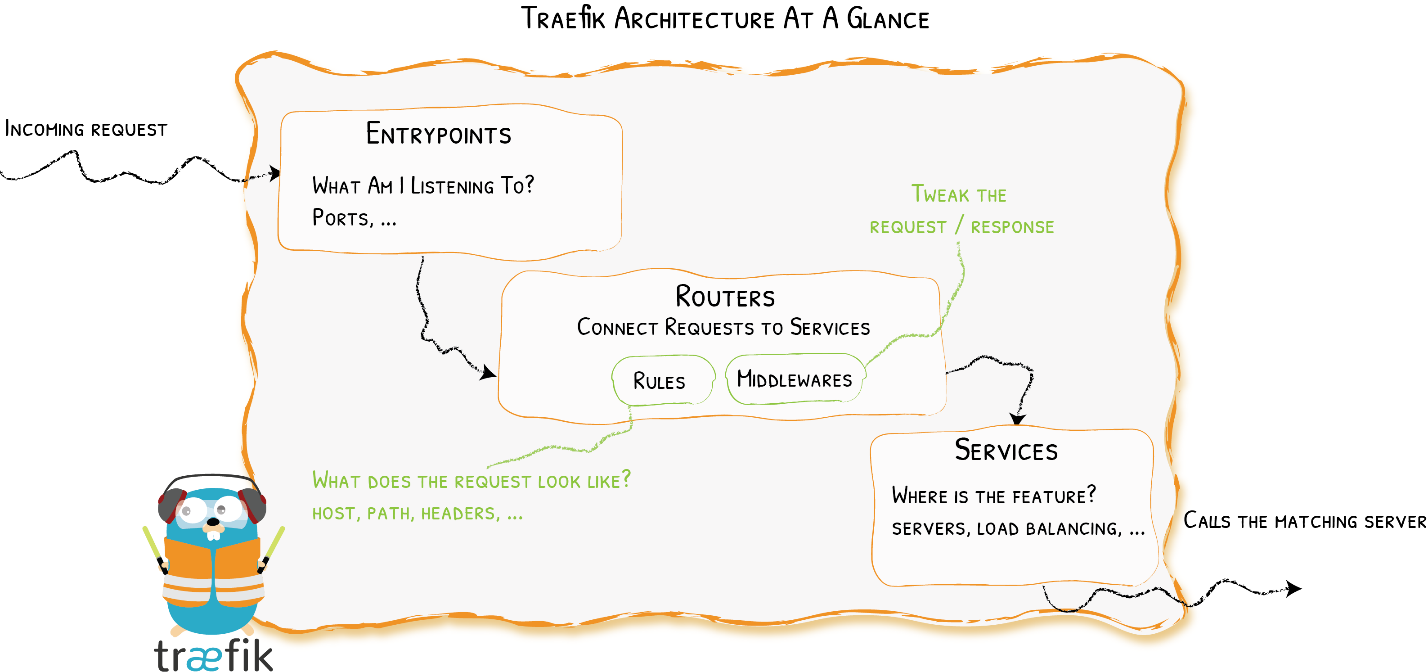
Traefik là một reverse-proxy hiện đại và có thể xử lý môi trường vùng chứa động. Nó là một ứng dụng nhỏ được viết bằng GO phù hợp với những thách thức mới. Bạn có thể sử dụng nó như một giao diện người dùng trong nhiều môi trường khác nhau. Những cái đơn giản hơn là Docker và Docker Swarm, những cái phức tạp hơn là Apache Mesos hoặc Kubernetes. Bạn thậm chí có thể đọc siêu dữ liệu từ các services thư mục như etcd hoặc Consul.

### **2.1.3. Các tính năng của Traefik**

Dưới đây là một số tính năng nổi trội của phương pháp Traefik:

* Traefik có tính năng Watch, vì vậy có thể liên tục cập nhật cấu hình của nó mà không cần phải khởi động lại
* Hỗ trợ nhiều giải thuật load balancing
* Cung cấp HTTPS cho các Microservices của bạn bằng cách tận dụng Let's Encrypt
* Dễ dàng sử dụng cũng như quản lý thông qua giao diện clean Web UI của nó
* Có sẵn Websocket, HTTP / 2, GRPC
* Chống quá tải với Circuit breakers
* Có lưu trữ nhật ký truy cập (JSON, CLF)
* Tốc độ sử dụng xử lý nhanh
* Có Rest API cho bạn sử dụng để để update các config
  1. **Kiến trúc của Traefik**

Xác định điểm truy cập (ở dạng cơ bản nhất, chúng là số cổng). Sau khi được kết nối với các điểm truy cập này, routers phân tích các yêu cầu đến để xem liệu chúng có khớp với một bộ quy tắc(rule) hay không. Nếu có, bộ định tuyến có thể chuyển đổi yêu cầu bằng cách sử dụng các phần mềm trung gian (middleware) trước khi chuyển tiếp chúng đến services của bạn.

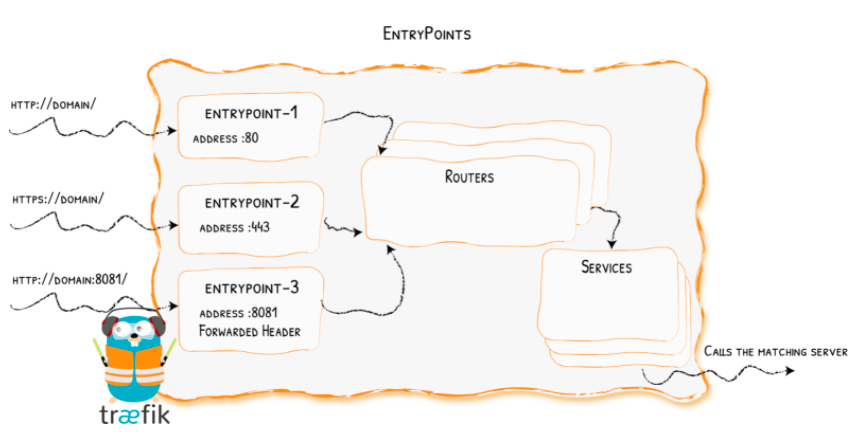


Hình 2. . Kiến trúc của Traefik

Trách nhiệm của từng thành phần trong kiến trúc:

* Các nhà cung cấp (providers) khám phá services sống trên cơ sở hạ tầng của bạn (IP, sức khỏe, ...)
* Các điểm truy cập (entrypoints) lắng nghe lưu lượng đến (cổng, ...)
* Routers phân tích các yêu cầu (máy chủ, đường dẫn, tiêu đề, SSL, ...)
* Services chuyển tiếp yêu cầu đến services của bạn (cân bằng tải, ...)
* Các phần mềm trung gian (Middlewares) có thể cập nhật yêu cầu hoặc đưa ra quyết định dựa trên yêu cầu (xác thực, giới hạn tốc độ, tiêu đề, ...)

### **Các điểm truy cập (EntryPoints)**



Hình 2. . Sơ đồ EntryPoints trong Traefik

EntryPoints là các điểm nhập mạng vào Traefik. Chúng xác định cổng sẽ nhận các gói tin và có lắng nghe TCP hay UDP hay không.

* Cấu hình

EntryPoints là một phần của cấu hình tĩnh. Chúng có thể được xác định bằng cách sử dụng đối số tệp (YAML hoặc TOML) hoặc CLI.

* Các lựa chọn HTTP (HTTP options)

Toàn bộ phần này dành riêng cho các tùy chọn, được khóa bằng điểm nhập, sẽ chỉ áp dụng cho định tuyến HTTP.

* Chuyển hướng (Redirect)

*Entrypoint*: Phần này là một tiện ích để cho phép chuyển hướng (vĩnh viễn) tất cả các yêu cầu đến trên một điểm vào (ví dụ: cổng 80) đến một điểm vào khác (ví dụ: cổng 443) hoặc một cổng rõ ràng (: 443).

* Các phần mềm trung gian (Middleware)

Danh sách middleware được thêm vào trước theo mặc định vào danh sách middleware của mỗi bộ định tuyến được liên kết với điểm vào được đặt tên.

* TLS (Bảo mật tầng vận chuyển)

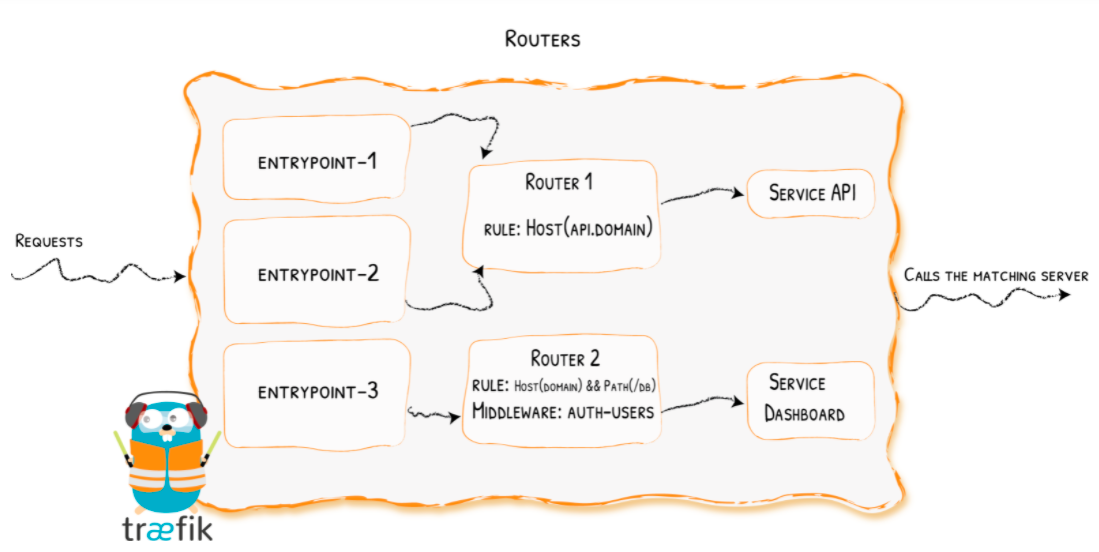
Phần này nói về cấu hình TLS mặc định được áp dụng cho tất cả các bộ định tuyến được liên kết với điểm nhập được đặt tên. Nếu phần TLS (tức là bất kỳ trường nào của nó) do người dùng xác định, thì cấu hình mặc định hoàn toàn không áp dụng. Phần TLS giống như phần TLS trên bộ định tuyến HTTP.

* Các lựa chọn UDP (UDP Options)

Toàn bộ phần này dành riêng cho các tùy chọn, được khóa bằng điểm vào, sẽ chỉ áp dụng cho định tuyến UDP.

*Time out:* Thời gian chờ xác định thời gian chờ trong một phiên không hoạt động trước khi giải phóng các tài nguyên liên quan. Giá trị Thời gian chờ phải lớn hơn 0.

### **Các Bộ Định Tuyến (Routers)**



Hình 2. . Sơ đồ Router trong Traefik

Một bộ định tuyến chịu trách nhiệm kết nối các yêu cầu đến với các services có thể xử lý chúng. Trong quá trình này, các bộ định tuyến có thể sử dụng các middleware để cập nhật yêu cầu hoặc hành động trước khi chuyển tiếp yêu cầu tới services

#### *Cấu hình HTTP Routers*

* Các điểm truy cập **(**Entrypoints)

Nếu không được chỉ định, các bộ định tuyến HTTP sẽ chấp nhận các yêu cầu từ tất cả các điểm nhập đã xác định. Nếu bạn muốn giới hạn phạm vi bộ định tuyến trong một tập hợp các điểm vào, hãy đặt tùy chọn entryPoints.

* Quy tắc (Rule)

Rules là một tập hợp các cấu hình khớp nhau với các giá trị, xác định xem một yêu cầu cụ thể có khớp với các tiêu chí cụ thể hay không. Nếu quy tắc được xác minh, bộ định tuyến sẽ hoạt động, gọi middleware và sau đó chuyển tiếp yêu cầu tới services.

Bảng 2. : **Các bộ kết hợp có sẵn**

|  |  |
| --- | --- |
| **Quy định** | **Sự mô tả** |
| Headers(`key`, `value`) | Kiểm tra xem có khóa keyđược xác định trong tiêu đề hay không, với giá trịvalue |
| HeadersRegexp(`key`, `regexp`) | Kiểm tra xem có khóa keyđược xác định trong tiêu đề hay không, với giá trị khớp với biểu thức chính quyregexp |
| Host(`example.com`, ...) | Kiểm tra xem miền yêu cầu (giá trị tiêu đề máy chủ) có nhắm mục tiêu đến một trong các miền đã cho hay không domains. |
| HostHeader(`example.com`, ...) | Giống như Host, chỉ tồn tại vì lý do lịch sử. |
| HostRegexp(`example.com`, `{subdomain:[a-z]+}.example.com`, ...) | Khớp với miền yêu cầu. Xem "Cú pháp Regexp" bên dưới. |
| Method(`GET`, ...) | Kiểm tra xem phương thức yêu cầu có phải là một trong các phương thức đã cho methods( GET,,,, POST)PUTDELETEPATCHHEAD |
| Path(`/path`, `/articles/{cat:[a-z]+}/{id:[0-9]+}`, ...) | Khớp đường dẫn yêu cầu chính xác. Xem "Cú pháp Regexp" bên dưới. |
| PathPrefix(`/products/`, `/articles/{cat:[a-z]+}/{id:[0-9]+}`) | Khớp đường dẫn tiền tố yêu cầu. Xem "Cú pháp Regexp" bên dưới. |
| Query(`foo=bar`, `bar=baz`) | Khớp các tham số Chuỗi truy vấn. Nó chấp nhận một chuỗi các cặp key = value. |
| ClientIP(`10.0.0.0/16`, `::1`) | Khớp nếu IP của ứng dụng khách yêu cầu là một trong các IP / CIDR đã cho. Nó chấp nhận các định dạng IPv4, IPv6 và CIDR. |

* Priority (Quyền ưu tiên)

Để tránh chồng chéo đường dẫn, các tuyến đường được sắp xếp, theo mặc định, theo thứ tự giảm dần bằng cách sử dụng độ dài quy tắc. Mức độ ưu tiên trực tiếp bằng độ dài của quy tắc và do đó độ dài dài nhất có mức độ ưu tiên cao nhất.

Giá trị 0 cho mức độ ưu tiên bị bỏ qua: ưu tiên = 0 có nghĩa là sắp xếp theo độ dài quy tắc mặc định được sử dụng.

* Services

Mỗi yêu cầu cuối cùng phải được xử lý bởi một services, đó là lý do tại sao mỗi định nghĩa bộ định tuyến nên bao gồm một mục tiêu services, về cơ bản là nơi yêu cầu sẽ được chuyển đến.

Nói chung, một services được chỉ định cho một bộ định tuyến nên đã được xác định, nhưng có những ngoại lệ đối với các provider dựa trên nhãn.

* TLS

Khi một phần TLS được chỉ định, nó sẽ hướng dẫn Traefik rằng bộ định tuyến hiện tại chỉ dành riêng cho các yêu cầu HTTPS (và bộ định tuyến nên bỏ qua các yêu cầu HTTP (không phải TLS)). Traefik sẽ chấm dứt các kết nối SSL (có nghĩa là nó sẽ gửi dữ liệu được giải mã đến các services).

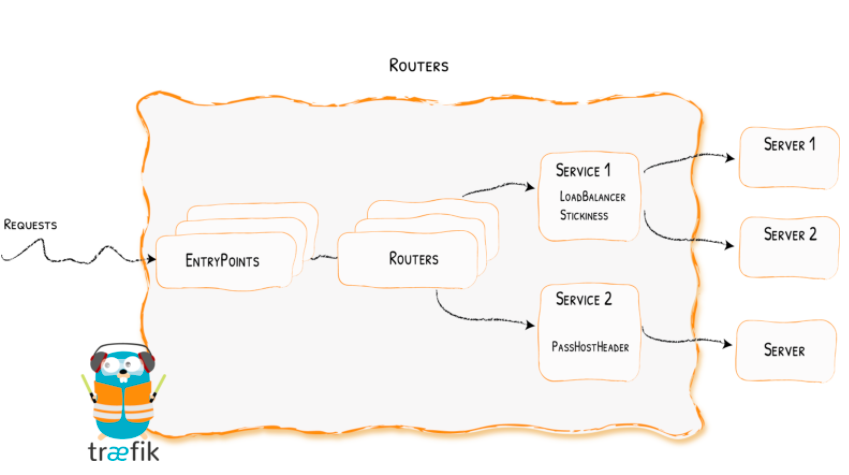
#### *TCP Routers*

Nếu cả bộ định tuyến HTTP và bộ định tuyến TCP lắng nghe cùng một điểm vào, bộ định tuyến TCP sẽ áp dụng trước bộ định tuyến HTTP. Nếu không tìm thấy tuyến nào phù hợp cho các bộ định tuyến TCP, thì các bộ định tuyến HTTP sẽ tiếp quản.

#### *UDP Routers*

Tương tự với TCP, vì UDP là lớp truyền tải, không có khái niệm về yêu cầu, do đó không có khái niệm về tiền tố đường dẫn URL để khớp với gói UDP đến. Hơn nữa, vì hiện tại không có hỗ trợ TLS tốt cho nhiều máy chủ, nên không có khái niệm SNI Máy chủ nào phù hợp với cả hai máy. Do đó, không có tiêu chí nào có thể được sử dụng làm quy tắc để khớp các gói đến nhằm định tuyến chúng. Vì vậy, các "bộ định tuyến" UDP tại thời điểm này hầu như chỉ là bộ cân bằng tải ở dạng này hay dạng khác.

### **Services**



Hình 2. . Sơ đồ EntryPoints trong Traefik

Các services chịu trách nhiệm định cấu hình cách tiếp cận các services thực tế mà cuối cùng sẽ xử lý các yêu cầu đến

#### *Cân bằng tải server (Server Load Balance)*

* Server

Server khai báo một phiên bản chương trình của bạn. Tùy chọn url trỏ đến một phiên bản cụ thể.

Các đường dẫn trong url của máy chủ không có hiệu lực. Nếu bạn muốn các yêu cầu được gửi đến một đường dẫn cụ thể trên máy chủ của mình, hãy định cấu hình bộ định tuyến để sử dụng phần mềm trung gian tương ứng (ví dụ: phần mềm trung gian AddPrefix hoặc ReplacePath).

* Load-balancing

Hiện tại chỉ có phương pháp cân bằng tải Round Robin được hộ trỡ.

* Phiên cố định (Stiky Sessions)

Khi các phiên cố định được bật, tiêu đề Set-Cookie được đặt trên phản hồi ban đầu để cho khách hàng biết máy chủ nào xử lý phản hồi đầu tiên. Trong các yêu cầu tiếp theo, để giữ cho phiên hoạt động với cùng một máy chủ, máy khách phải gửi cookie với giá trị được đặt.

*Lưu ý:*

* Độ kết dính đa tầng

Khi chuỗi hoặc trộn các bộ cân bằng tải (ví dụ: bộ cân bằng tải của các máy chủ là một trong những "con" của bộ cân bằng tải của các dịch vụ), để khả năng hoạt động ổn định, tùy chọn cần phải được chỉ định ở tất cả các cấp độ yêu cầu. Điều đó có nghĩa là khách hàng cần gửi một cookie với càng nhiều cặp khóa / giá trị càng có nhiều cấp độ dính.

* Máy chủ chỉ định và không lành mạnh

Nếu máy chủ được chỉ định trong cookie trở nên không lành mạnh, yêu cầu sẽ được chuyển tiếp đến máy chủ mới (và cookie sẽ theo dõi máy chủ mới).

* Tên cookie

Tên cookie mặc định là chữ viết tắt của sha1 (ví dụ: \_1d52e).

* Cờ Secure & HTTPOnly & SameSite

Theo mặc định, cookie sở thích được tạo mà không có các cờ đó. Tuy nhiên, người ta có thể thay đổi điều đó thông qua cấu hình.

SameSite có thể là không, lỏng lẻo, nghiêm ngặt hoặc trống rỗng.

* Kiểm tra tình trạng (Health Check)

Cấu hình Health Check kiểm tra tình trạng để loại bỏ các máy chủ không lành mạnh khỏi vòng xoay cân bằng tải. Traefik sẽ coi các máy chủ của bạn hoạt động tốt miễn là chúng trả về mã trạng thái giữa 2XX và 3XX cho các yêu cầu kiểm tra tình trạng (thực hiện trong mỗi khoảng thời gian).

Để phổ biến các thay đổi trạng thái (ví dụ: tất cả các máy chủ của services này đều không hoạt động) trở lên, HealthCheck cũng phải được bật trên (các) services chính của services này.Dưới đây là các tùy chọn có sẵn cho cơ chế healtch check :

* *Path* được thêm vào server URL dể thiết lập điểm cuối của kiểm tra tình trạng
* *scheme*, nếu được xác định, sẽ thay thế lược đồ server URL cho điểm cuối kiểm tra tình trạng
* *hostname*, nếu được xác định, sẽ áp dụng tên máy chủ tiêu đề Host cho yêu cầu kiểm tra tình trạng.
* *port*, nếu được xác định, sẽ thay thế cổng server URL cho điểm cuối kiểm tra tình trạng.
* *interval* xác định tần suất của các cuộc gọi kiểm tra sức khỏe.
* *timeout* xác định khoảng thời gian tối đa mà Traefik sẽ đợi yêu cầu kiểm tra sức khỏe trước khi xem xét máy chủ bị lỗi (không lành mạnh).
* *headers* xác định tiêu đề tùy chỉnh được gửi đến điểm cuối kiểm tra sức khỏe.
* *followRedirects* xác định liệu chuyển hướng có nên được tuân theo trong các cuộc gọi kiểm tra tình trạng hay không (mặc định: true).

*Lưu ý:*

* Định dạng cho interval và timeout

Khoảng thời gian và thời gian chờ phải được đưa ra ở định dạng được hiểu theo thời gian. Khoảng thời gian phải lớn hơn thời gian chờ. Nếu cấu hình không phản ánh điều này, khoảng thời gian sẽ được đặt thành thời gian chờ + 1 giây.

* Phục hồi máy chủ

Traefik tiếp tục theo dõi tình trạng của các máy chủ không lành mạnh. Nếu một máy chủ đã khôi phục (trả lại phản hồi 2xx -> 3xx một lần nữa), nó sẽ được thêm trở lại nhóm xoay bộ cân bằng tải.

* Truyền tải máy chủ (Server Transport)

Server Transport cho phép tham chiếu cấu hình ServersTransport để giao tiếp giữa Traefik và các máy chủ của bạn.

* Chuyển tiếp phản hồi (Response Forwarding)

Phần này là về cách Traefik chuyển tiếp phản hồi từ máy chủ phụ trợ đến máy khách.

Dưới đây là các tùy chọn có sẵn cho cơ chế Response Forwarding:

*Khoảng thời gian xả (FlushInterval):* *Chỉ định khoảng thời gian giữa các lần gửi tới máy khách trong khi sao chép nội dung phản hồi. Đó là thời lượng tính bằng mili giây, mặc định là 100. Giá trị âm có nghĩa là sẽ giảm ngay sau mỗi lần viết cho máy khách. FlushInterval bị bỏ qua khi ReverseProxy nhận ra một phản hồi là phản hồi phát trực tuyến; đối với những phản hồi như vậy, các bản ghi sẽ được chuyển cho khách hàng ngay lập tức.*

#### *Máy chủ vận chuyển (ServersTransport)*

ServersTransport cho phép định cấu hình truyền tải giữa Traefik và các máy chủ của bạn.

#### *Weighted Round Robin (Service)*

WRR có thể cân bằng tải các yêu cầu giữa nhiều services dựa trên trọng số. Chiến lược này chỉ khả dụng để cân bằng tải giữa các services chứ không phải giữa các máy chủ.

**HeathCheck:** HealthCheck cho phép tự kiểm tra sức khỏe tự động cho services này, tức là bất cứ khi nào một trong các services con của nó được báo cáo là ngừng hoạt động, services này sẽ biết về nó và tính đến nó (tức là nó bỏ qua con bị lỗi) khi chạy thuật toán cân bằng tải. Ngoài ra, nếu services gốc của services này cũng đã bật HealthCheck, thì services này sẽ báo cáo cho services gốc bất kỳ thay đổi trạng thái nào.

#### *Phản chiếu service (Mirroring)*

Mirroring có thể phản chiếu các yêu cầu được gửi đến một services đến các services khác. Xin lưu ý rằng theo mặc định, toàn bộ yêu cầu được lưu vào bộ nhớ đệm trong khi nó đang được sao chép.

**HeathCheck:** HealthCheck cho phép tự động kiểm tra sức khỏe tự động cho services này, tức là nếu trình xử lý chính của services trở nên không thể truy cập được, thông tin sẽ được truyền tải lên cấp mẹ của nó.

#### *e. Chuyển đổi dự phòng service (Failover)*

Công việc của dịch vụ chuyển đổi dự phòng là chuyển tiếp tất cả các yêu cầu tới dịch vụ dự phòng khi dịch vụ chính không thể truy cập được.

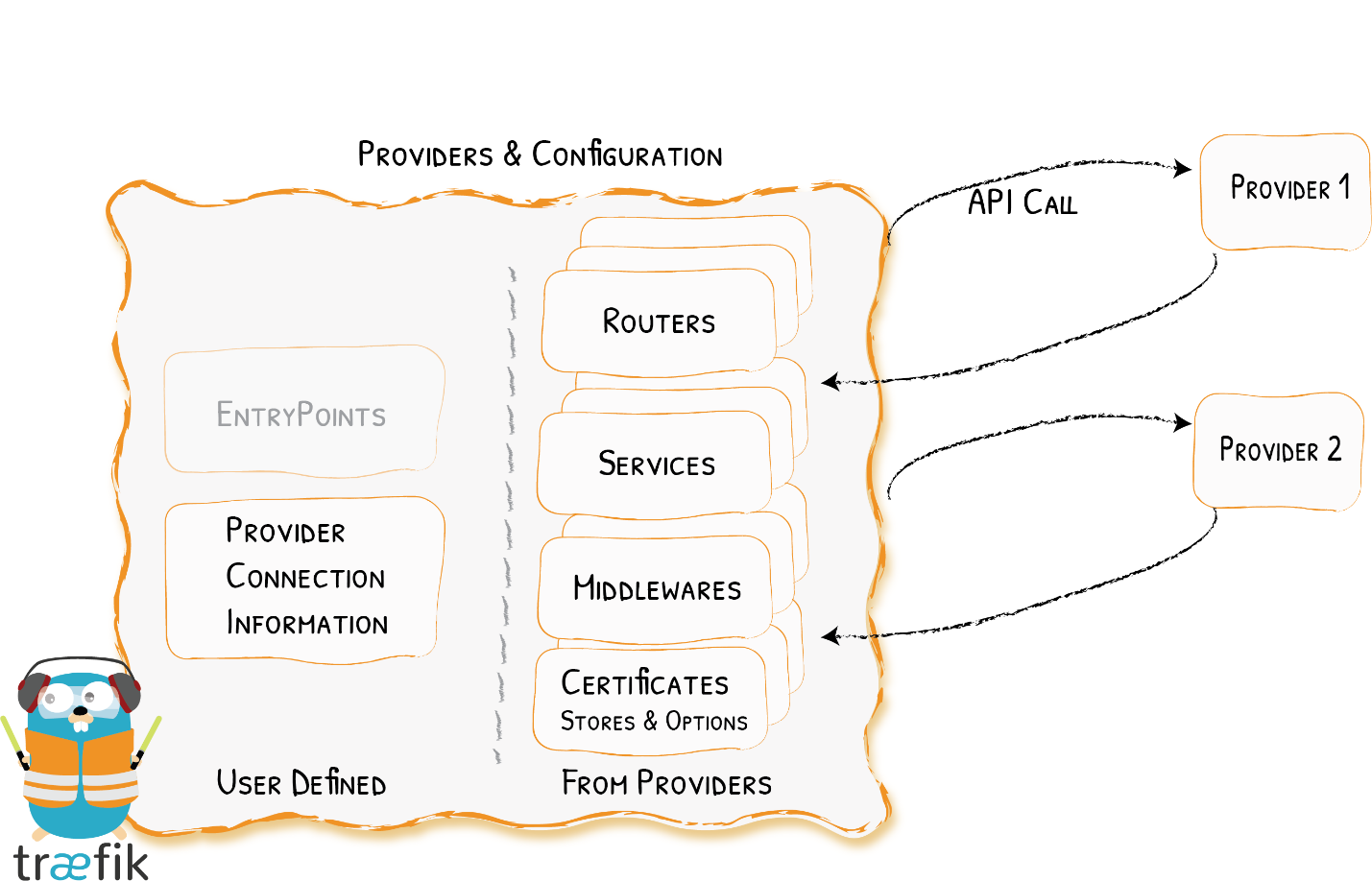
Liên quan đến HealthCheck: Dịch vụ chuyển đổi dự phòng dựa vào hệ thống HealthCheck để nhận thông báo khi dịch vụ chính của nó không thể truy cập được, có nghĩa là HealthCheck cần được kích hoạt và hoạt động trên dịch vụ chính. Tuy nhiên, HealthCheck không cần phải được bật trên chính dịch vụ chuyển đổi dự phòng để nó hoạt động. Nó chỉ được yêu cầu để truyền tải thông tin lên khi bản thân quá trình chuyển đổi dự phòng gặp sự cố (tức là cả chính và dự phòng của nó đều bị hỏng).

*Lưu ý:* Chiến lược này hiện chỉ có thể được xác định với File Provider*.*

**HeathCheck:** HealthCheck cho phép tự kiểm tra sức khỏe tự động cho dịch vụ này, tức là nếu dịch vụ chính và dịch vụ dự phòng không thể truy cập được, thì thông tin sẽ được truyền tải lên đến cấp độ gốc của nó.

*Lưu ý :*

* Nếu HealthCheck được bật cho một dịch vụ nhất định, nhưng bất kỳ dịch vụ nào trong số các dịch vụ con của nó không được kích hoạt, thì việc tạo dịch vụ sẽ không thành công.
* HealthCheck trên dịch vụ Chuyển đổi dự phòng hiện chỉ có thể được xác định với File Provider.
  + 1. ***Các nhà cung cấp (Providers)***



Hình 2. . Sơ đồ Provider trong Traefik

Các provider là các thành phần cơ sở hạ tầng, cho dù là bộ điều phối, công cụ vùng chứa, provider đám mây hay các cửa hàng khóa-giá trị. Ý tưởng là Traefik truy vấn các API của provider để tìm thông tin liên quan về định tuyến và khi Traefik phát hiện ra thay đổi, nó sẽ tự động cập nhật các tuyến.

#### *Bộ điều phối (Orchestrators)*

Mặc dù mỗi provider là khác nhau, nhưng bạn có thể coi mỗi provider thuộc một trong bốn loại:

* Dựa trên nhãn: mỗi vùng chứa được triển khai có một tập hợp các nhãn được đính kèm với nó
* Dựa trên khóa-giá trị: mỗi vùng chứa được triển khai cập nhật kho khóa-giá trị với thông tin liên quan
* Dựa trên chú thích: một đối tượng riêng biệt, với các chú thích, xác định các đặc điểm của vùng chứa
* Dựa trên tệp: sử dụng tệp để xác định cấu hình

#### *Không gian tên Nhà cung cấp (Provider Namespace)*

Khi bạn khai báo các đối tượng nhất định trong cấu hình động Traefik, chẳng hạn như middleware, services, TLS options hoặc server transports, chúng nằm trong không gian tên của provider của chúng. Ví dụ: nếu bạn khai báo một middleware bằng cách sử dụng nhãn Docker, nó nằm trong không gian tên provider Docker.

Nếu bạn sử dụng nhiều provider và muốn tham chiếu đến một đối tượng được khai báo trong một provider khác (ví dụ: tham chiếu đến một đối tượng nhiều provider như middleware ), thì tên đối tượng phải được đặt sau dấu phân tách @ và tên nhà cung cấp.

<resource-name>@<provider-name>

**Bảng 2. 2: Danh sách các nhà cung cấp hiện được hỗ trợ trong Traefik**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nhà cung cấp** | **Kiểu** | **Kiểu cấu hình** | **Tên nhà cung cấp** |
| [Docker](https://doc.traefik.io/traefik/providers/docker/) | Bộ điều phối | Nhãn | docker |
| [Kubernetes IngressRoute](https://doc.traefik.io/traefik/providers/kubernetes-crd/) | Bộ điều phối | Tài nguyên tùy chỉnh | kubernetescrd |
| [Kubernetes Ingress](https://doc.traefik.io/traefik/providers/kubernetes-ingress/) | Bộ điều phối | Xâm nhập | kubernetes |
| [Kubernetes Gateway API](https://doc.traefik.io/traefik/providers/kubernetes-gateway/) | Bộ điều phối Bộ điều phối | Tài nguyên API Gateway | kubernetesgateway |
| [Consul Catalog](https://doc.traefik.io/traefik/providers/consul-catalog/) | Bộ điều phối | Nhãn | consulcatalog |
| [ECS](https://doc.traefik.io/traefik/providers/ecs/) | Bộ điều phối | Nhãn | ecs |
| [Marathon](https://doc.traefik.io/traefik/providers/marathon/) | Bộ điều phối | Nhãn | marathon |
| [Rancher](https://doc.traefik.io/traefik/providers/rancher/) | Bộ điều phối | Nhãn | rancher |
| [File](https://doc.traefik.io/traefik/providers/file/) | Thủ công | Định dạng YAML/TOML | file |
| [Consul](https://doc.traefik.io/traefik/providers/consul/) | Key/Value | Key/Value | consul |
| [Etcd](https://doc.traefik.io/traefik/providers/etcd/) | Key/Value | Key/Value | etcd |
| [ZooKeeper](https://doc.traefik.io/traefik/providers/zookeeper/) | Key/Value | Key/Value | zookeeper |
| [Redis](https://doc.traefik.io/traefik/providers/redis/) | Key/Value | Key/Value | redis |
| [HTTP](https://doc.traefik.io/traefik/providers/http/) | Thủ công | Định dạng JSON | http |

2.2.4.3. Hạn chế phạm vi khám phá dịch vụ

Theo mặc định, Traefik tạo các tuyến đường cho tất cả các vùng chứa được phát hiện.

Nếu bạn muốn giới hạn phạm vi khám phá dịch vụ Traefik, tức là không cho phép tạo tuyến cho một số vùng chứa, bạn có thể làm như vậy theo hai cách khác nhau:

* Tùy chọn cấu hình chung exposedByDefault.

Danh sách các nhà cung cấp hỗ trợ các tính năng này:

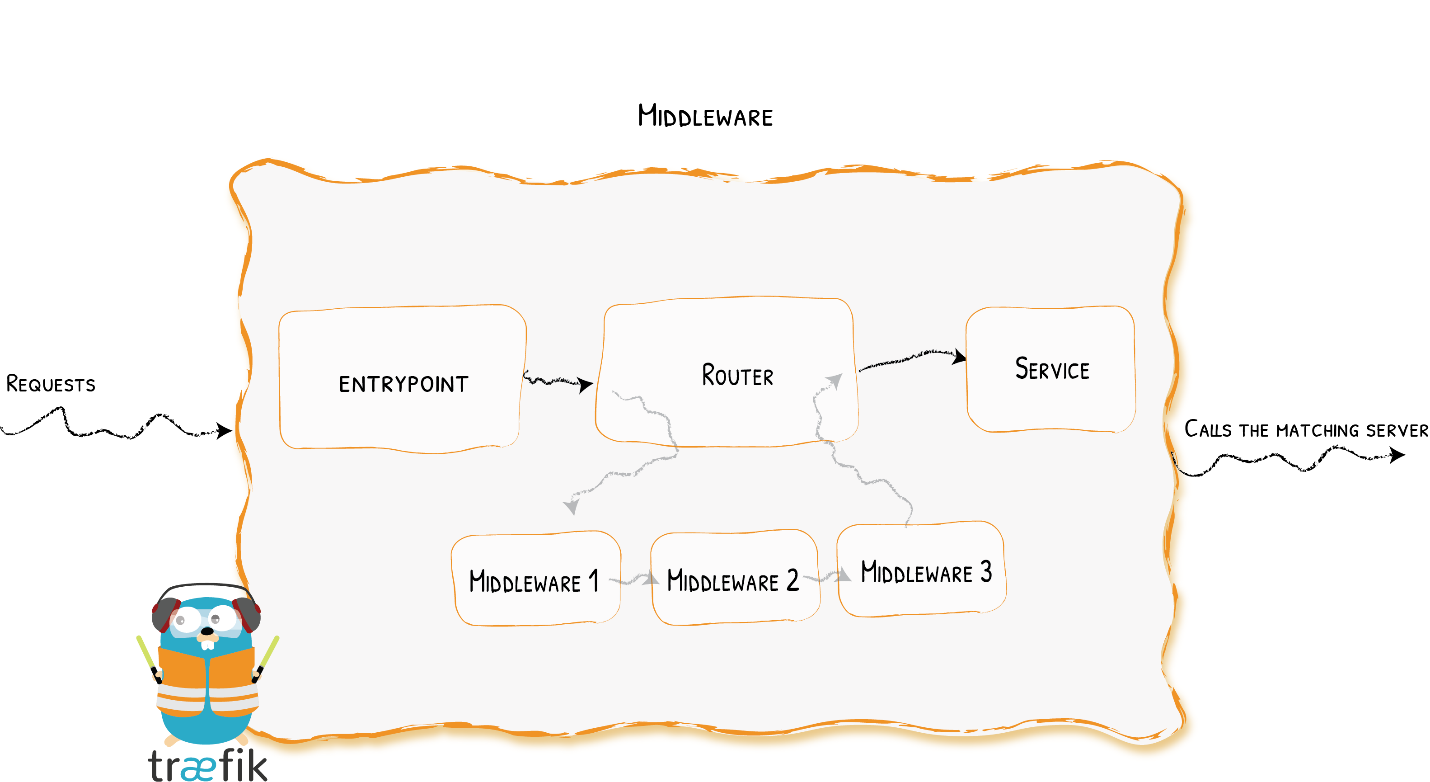
* + Docker
  + Consul Catalog
  + Rancher
  + Marathon
* Một cơ chế chi tiết tốt hơn dựa trên các ràng buộc.

Danh sách các nhà cung cấp hỗ trợ các ràng buộc:

* + Docker
  + Consul Catalog
  + Rancher
  + Marathon
  + Kubernetes CRD
  + Kubernetes Ingress
  + Kubernetes Gateway

* + 1. ***Phần mềm trung gian (MiddleWare)***

Điều chỉnh yêu cầu



Hình 2. . Sơ đồ MiddleWare trong Traefik

Được gắn với các router, middlewares là một phương tiện để điều chỉnh các yêu cầu trước khi chúng được gửi đến dịch vụ của bạn (hoặc trước khi câu trả lời từ dịch vụ được gửi đến khách hàng).

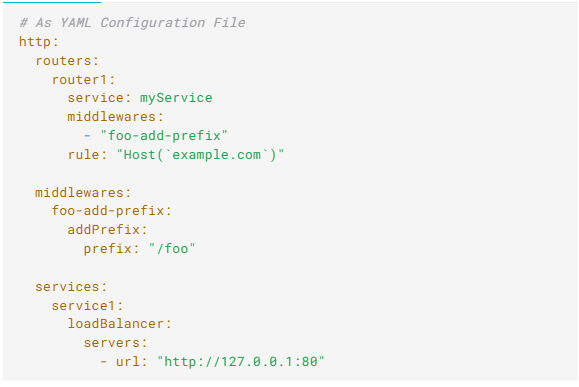
Có một số middleware có sẵn trong Traefik, một số có thể sửa đổi yêu cầu, tiêu đề, một số phụ trách chuyển hướng, một số thêm xác thực, v.v.

Middleware sử dụng cùng một giao thức có thể được kết hợp thành chuỗi để phù hợp với mọi tình huống.

#### Cấu hình

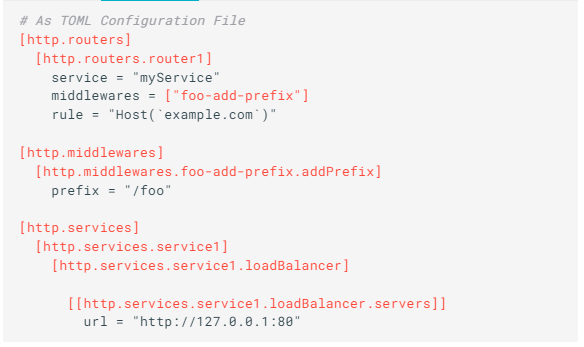
Trong Docker:

File (YAML)



Hình 2. . File cấu hính YAML của MiddleWare

File (TOML)



Hình 2. . File cấu hính TOML của MiddleWare

#### b. Phần mềm trung gian có sẵn

* Phần mềm trung gian HTTP có sẵn

Bảng 2. **: Danh sách phần mềm trung gian HTTP có sẵn**

|  |  |  |
| --- | --- | --- |
| **Phần mềm trung gian** | **Mục đích** | **Vùng hoạt động** |
| AddPrefix | Thêm tiền tố đường dẫn | Công cụ sửa đổi đường dẫn |
| BasicAuth | Thêm xác thực cơ bản | Bảo mật, Xác thực |
| Buffering | Tạo bộ đệm cho yêu cầu / phản hồi | Yêu cầu vòng đời |
| Chain | Kết hợp nhiều phần mềm trung gian | Misc |
| CircuitBreaker | Ngăn chặn việc gọi các dịch vụ không lành mạnh | Yêu cầu vòng đời |
| Compress | Nén phản hồi | Công cụ sửa đổi nội dung |
| ContentType | Xử lý tự động phát hiện loại nội dung | Misc |
| DigestAuth | Thêm xác thực thông báo | Bảo mật, Xác thực |
| Errors | Xác định các trang lỗi tùy chỉnh | Yêu cầu vòng đời |
| ForwardAuth | Xác thực ủy quyền | Bảo mật, Xác thực |
| Headers | Thêm / Cập nhật tiêu đề | Bảo vệ |
| IPWhiteList | Giới hạn các IP máy khách được phép | Bảo mật, Vòng đời yêu cầu |
| InFlightReq | Giới hạn số lượng kết nối đồng thời | Bảo mật, Vòng đời yêu cầu |
| PassTLSClientCert | Thêm chứng chỉ khách hàng trong tiêu đề | Bảo vệ |
| RateLimit | Giới hạn tần suất cuộc gọi | Bảo mật, Vòng đời yêu cầu |
| RedirectScheme | Chuyển hướng dựa trên lược đồ | Yêu cầu vòng đời |
| RedirectRegex | Chuyển hướng dựa trên regex | Yêu cầu vòng đời |
| ReplacePath | Thay đổi đường dẫn của yêu cầu | Công cụ sửa đổi đường dẫn |
| ReplacePathRegex | Thay đổi đường dẫn của yêu cầu | Công cụ sửa đổi đường dẫn |
| Retry | Tự động thử lại trong trường hợp có lỗi | Yêu cầu vòng đời |
| StripPrefix | Thay đổi đường dẫn của yêu cầu | Công cụ sửa đổi đường dẫn |
| StripPrefixRegex | Thay đổi đường dẫn của yêu cầu | Công cụ sửa đổi đường dẫn |

* Phần mềm trung gian TCP có sẵn

Bảng 2. **: Danh sách phần mềm trung gian TCP có sẵn**

|  |  |  |
| --- | --- | --- |
| **Phần mềm trung gian** | **Mục đích** | **Vùng hoạt động** |
| InFlightConn | Giới hạn số lượng kết nối đồng thời. | Bảo mật, Vòng đời yêu cầu |
| IPWhiteList | Giới hạn các IP máy khách được phép. | Bảo mật, Vòng đời yêu cầu |

## 2.3. Phân Tích Source Code Về Load Balance

Load balancing (Cân bằng tải) là việc phân phối hiệu quả lưu lượng truy cập đến trên một nhóm backend servers, hay còn được gọi là server farm hoặc server pool.

Các website hiện đại có lưu lượng truy cập cao phải phục vụ hàng trăm nghìn, thậm chí hàng triệu các requests đồng thời từ người dùng hoặc khách hàng, đồng thời phải phản hồi chính xác lại các văn bản, hình ảnh, video hoặc dữ liệu ứng dụng, tất cả đều được thực hiện rất nhanh chóng và đáng tin cậy. Để đáp ứng hiệu quả khối lượng lớn các yêu cầu và phản hồi một cách tiết kiệm nhất, best practice tốt nhất thường sẽ là sử dụng thêm servers.

Một cân bằng tải (load balancer) hoạt động như " cảnh sát giao thông " (traffic cop) ở phía trước server và định tuyến các request của client trên tất cả các servers có khả năng thực hiện các request đó, sao cho tối ưu về tốc độ và hiệu suất nhất và đảm bảo rằng không có server nào phải hoạt động quá mức. Nếu một server đơn lẻ bị hỏng, cân bằng tải (load balancer) sẽ chuyển hướng lưu lượng truy cập đến các server trực tuyến còn lại. Khi một server mới được thêm vào nhóm máy chủ, bộ cân bằng tải sẽ tự động bắt đầu gửi yêu cầu đến máy chủ mới thêm này.

Tóm lại, một load balancer sẽ thực hiện các chức năng chính sau đây:

- Phân phối các client requests hoặc network load một cách hiệu quả trên nhiều servers.

- Đảm bảo tính khả dụng và độ tin cậy cao bằng cách chỉ gửi các yêu cầu đến các máy chủ trực tuyến.

- Thêm vào hoặc loại bớt các servers theo yêu cầu một cách linh hoạt và dễ dàng.

### **2.3.1. Hàm New**

Vị trí: Nằm trong file wrr.go của folder loadbalacner

*func New(sticky \*dynamic.Sticky, hc \*dynamic.HealthCheck) \*Balancer {*

*balancer := &Balancer{*

*status: make(map[string]struct{}),*

*wantsHealthCheck: hc != nil,*

*}*

*if sticky != nil && sticky.Cookie != nil {*

*balancer.stickyCookie = &stickyCookie{*

*name: sticky.Cookie.Name,*

*secure: sticky.Cookie.Secure,*

*httpOnly: sticky.Cookie.HTTPOnly,*

*}*

*}*

*return balancer*

*}*

Mục đích: Dùng để tạo khởi tạo một load balancer mới

### **2.3.2. Hàm Push**

Vị trí: Nằm trong file wrr.go của folder loadbalacner

*func (b \*Balancer) Push(x interface{}) {*

*h, ok := x.(\*namedHandler)*

*if !ok {*

*return*

*}*

*b.handlers = append(b.handlers, h)*

*}*

Mục đích: Hàm Push thực hiện heap.Interface dùng để đẩy một Item vào Heap

### **2.3.3. Hàm Pop**

Vị trí: Nằm trong file wrr.go của folder loadbalacner

*func (b \*Balancer) Pop() interface{} {*

*h := b.handlers[len(b.handlers)-1]*

*b.handlers = b.handlers[0 : len(b.handlers)-1]*

*return h*

*}*

Mục đích: Hàm Pop thực hiện heap.Interface dùng để đẩy một Item ra khỏi Heap

### **2.3.4. Hàm RegisterStatusUpdater**

Vị trí: Nằm trong file wrr.go của folder loadbalacner

*func (b \*Balancer) RegisterStatusUpdater(fn func(up bool)) error {*

*if !b.wantsHealthCheck {*

*return errors.New("healthCheck not enabled in config for this weighted service")*

*}*

*b.updaters = append(b.updaters, fn)*

*return nil*

*}*

Mục đích: Hàm RegisterStatusUpdater thêm Server vào danh sách các hook được chạy khi trạng thái của Balancer thay đổi.

### **2.3.5. Hàm nextServer**

Vị trí: Nằm trong file wrr.go của folder loadbalacner

*func (b \*Balancer) nextServer() (\*namedHandler, error) {*

*b.mutex.Lock()*

*defer b.mutex.Unlock()*

*if len(b.handlers) == 0 {*

*return nil, fmt.Errorf("no servers in the pool")*

*}*

*if len(b.status) == 0 {*

*return nil, errNoAvailableServer*

*}*

*var handler \*namedHandler*

*for {*

*handler = heap.Pop(b).(\*namedHandler)*

*b.curDeadline = handler.deadline*

*handler.deadline += 1 / handler.weight*

*heap.Push(b, handler)*

*if \_, ok := b.status[handler.name]; ok {*

*break*

*}*

*}*

*log.WithoutContext().Debugf("Service selected by WRR: %s", handler.name)*

*return handler, nil*

*}*

Mục đích: Hàm nextServer dùng để gọi đến Server kế tiếp trong pool

### **2.3.6. Hàm addService**

Vị trí: Nằm trong file wrr.go của folder loadbalacner

func (b \*Balancer) AddService(name string, handler http.Handler, weight \*int) {

w := 1

if weight != nil {

w = \*weight

}

if w <= 0 { // non-positive weight is meaningless

return

}

h := &namedHandler{Handler: handler, name: name, weight: float64(w)}

b.mutex.Lock()

h.deadline = b.curDeadline + 1/h.weight

heap.Push(b, h)

b.status[name] = struct{}{}

b.mutex.Unlock()

}

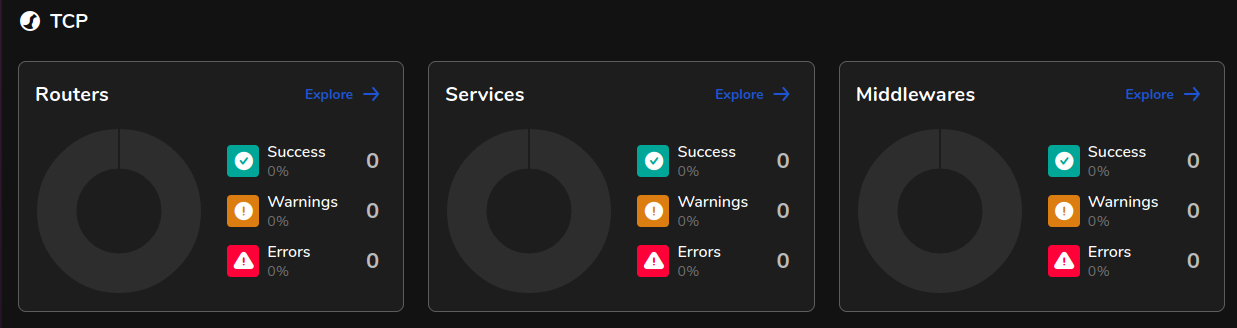
Mục đích: Hàm AddService thêm một handler. Một handler với trọng số không dương sẽ bị bỏ qua

* 1. **Phát triển, tùy biến một số tính năng Traefik**
     1. ***Thay đổi giao diện***

Về mặt giao diện của Traefik trên web, nội dụng trong giao diện đã được việt hóa.

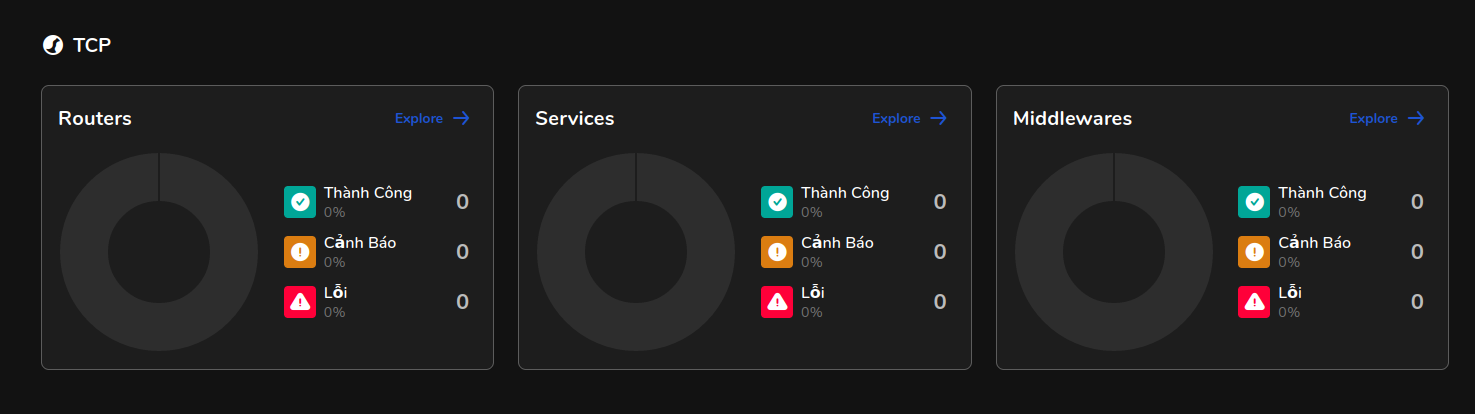
#### TCP

Giao diện của TCP trước khi thay đổi



Hình 2. . Giao diện của TCP ban đầu

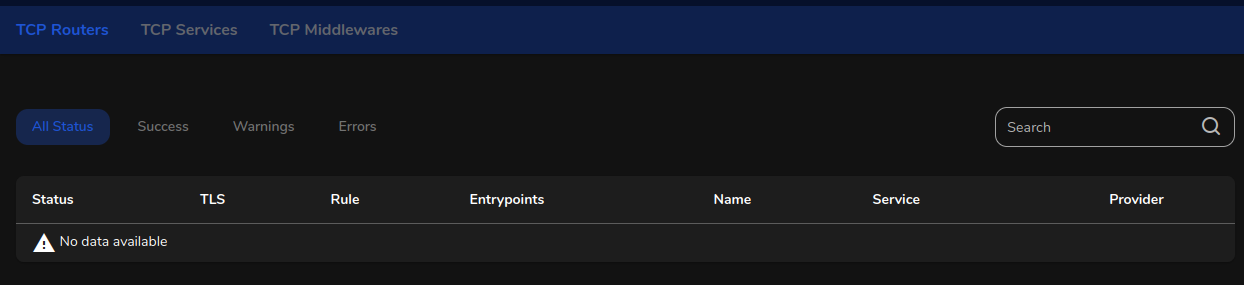
Giao diện của TCP sau khi thay đổi



Hình 2. . Giao diện của TCP sau khi thay đổi

#### b. TCP Routers

Giao diện của TCP Routers trước khi thay đổi



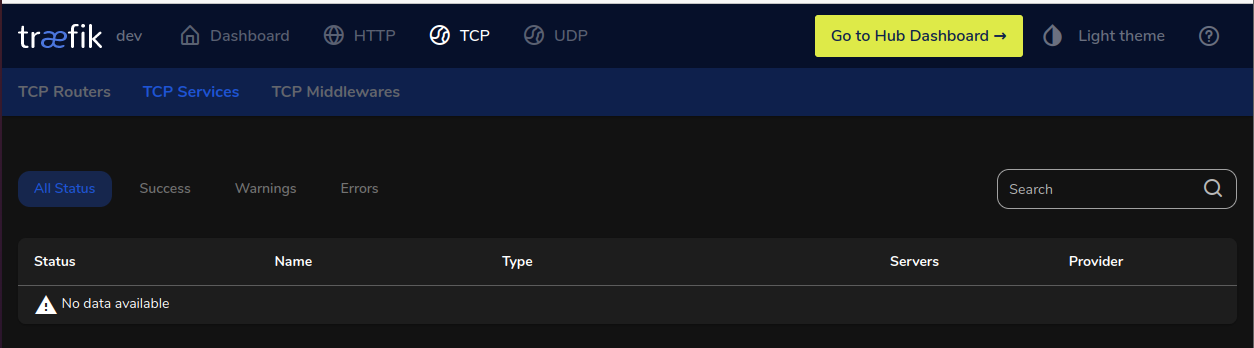
Hình 2. . Giao diện của TCP Routers ban đầu

Giao diện của TCP Routers sau khi thay đổi

Hình 2. . Giao diện của TCP Routers sau khi thay đổi

#### c. TCP Services

Giao diện của TCP Services trước khi thay đổi



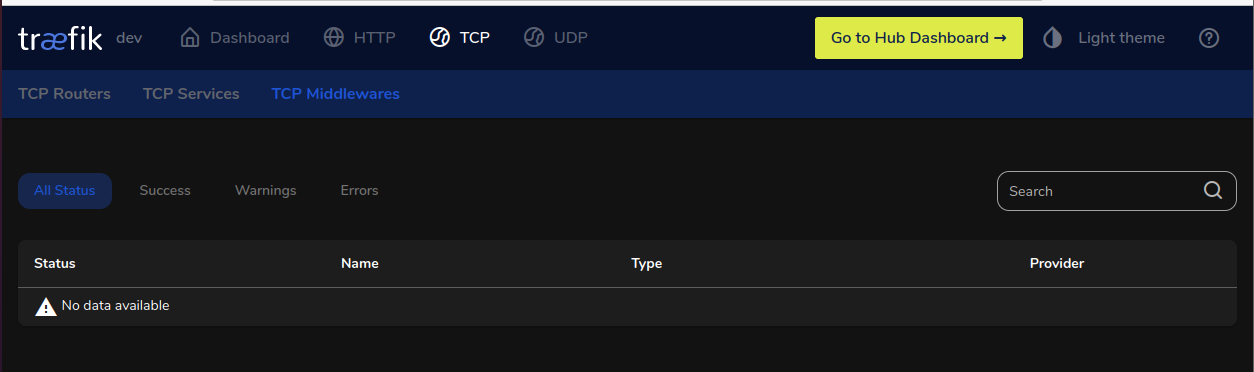
Hình 2. . Giao diện của TCP Services ban đầu

Giao diện của TCP Services sau khi thay đổi

Hình 2. . Giao diện của TCP Services sau khi thay đổi

#### d. TCP MiddleWares

Giao diện của TCP MiddleWares trước khi thay đổi



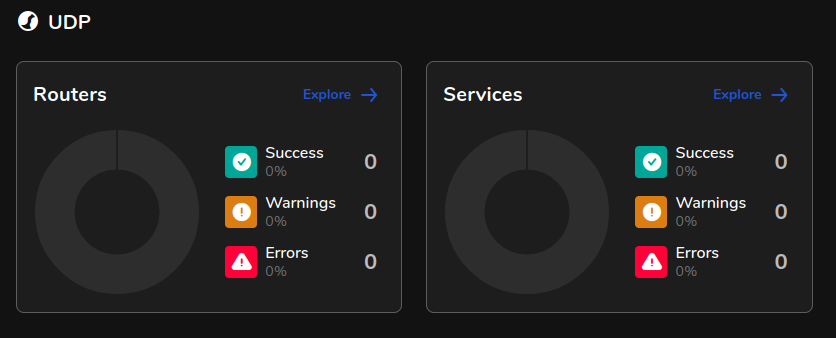
Hình 2. . Giao diện của TCP MiddleWares ban đầu

Giao diện của TCP MiddleWares sau khi thay đổi

Hình 2. . Giao diện của TCP MiddleWares sau khi thay đổi

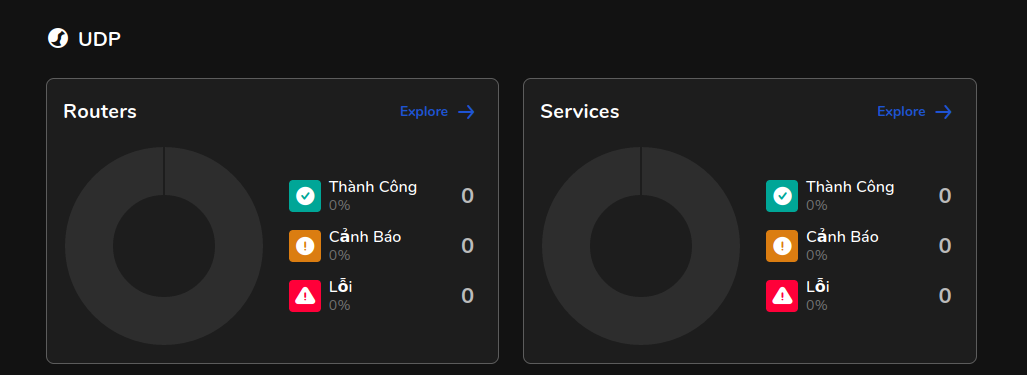
#### e. UDP

Giao diện của UDP trước khi thay đổi



Hình 2. . Giao diện của UDP ban đầu

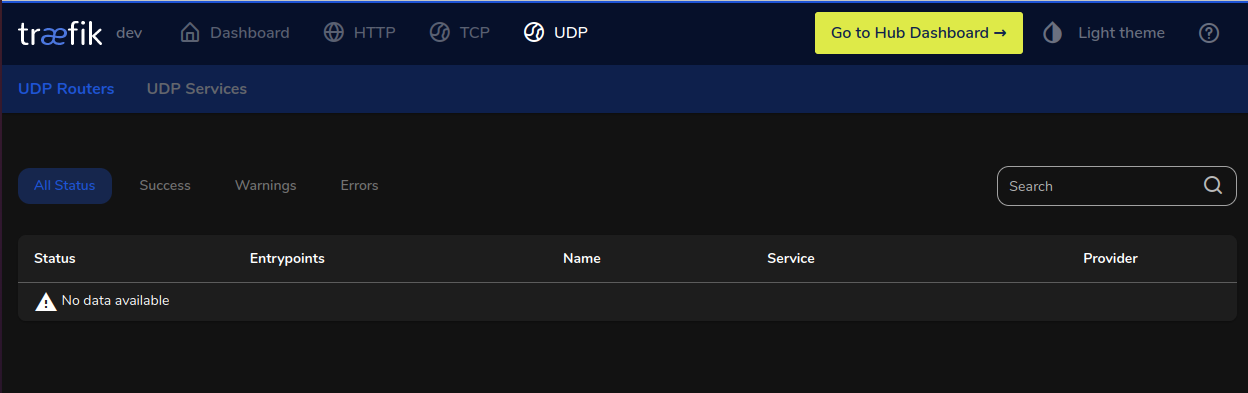
Giao diện của UDP sau khi thay đổi



Hình 2. . Giao diện của UDP sau khi thay đổi

#### f. UDP Routers

Giao diện của UDP Routers trước khi thay đổi



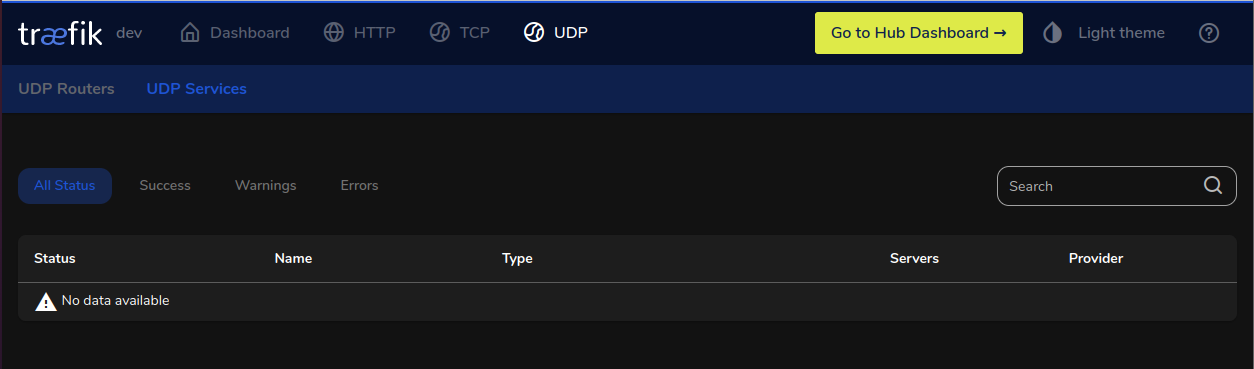
Hình 2. . Giao diện của UDP Routers ban đầu

Giao diện của UDP Routers sau khi thay đổi

Hình 2. . Giao diện của UDP Routers sau khi thay đổi

#### g. UDP Services

Giao diện của UDP Services trước khi thay đổi



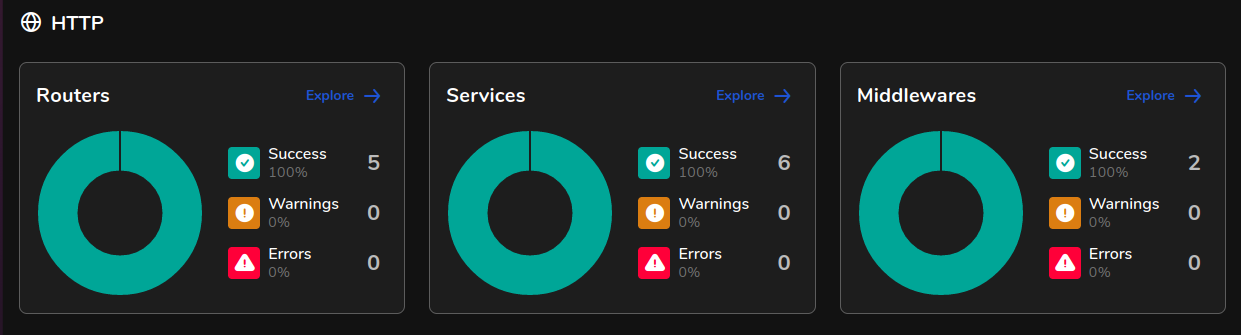
Hình 2. . Giao diện của UDP Services ban đầu

Giao diện của UDP Services sau khi thay đổi

Hình 2. . Giao diện của UDP Services sau khi thay đổi

#### h. HTTP

Giao diện của HTTP trước khi thay đổi



Hình 2. . Giao diện của HTTP ban đầu

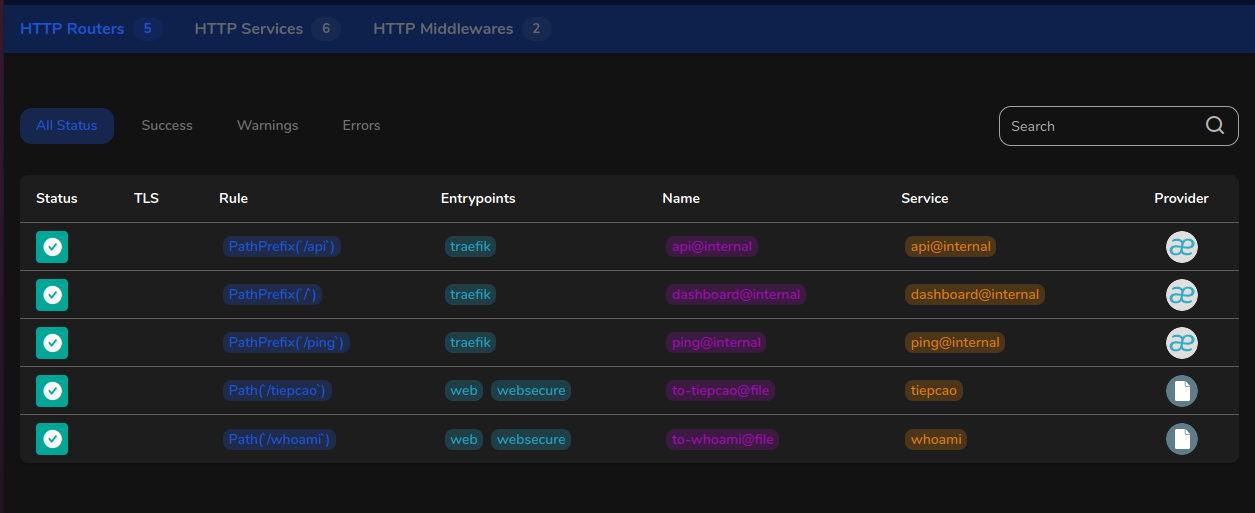
Giao diện của HTTP sau khi thay đổi



Hình 2. . Giao diện của HTTP sau khi thay đổi

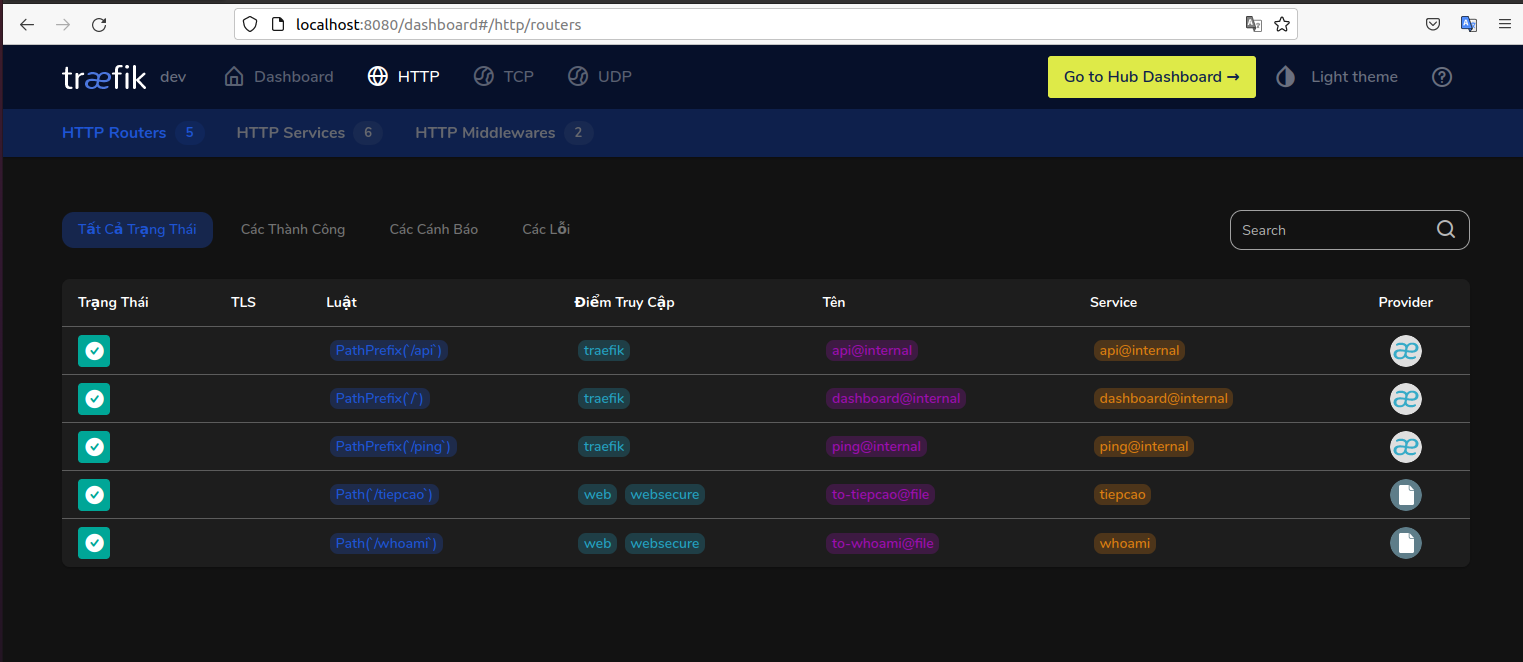
#### i. HTTP Routers

Giao diện của HTTP Routers trước khi thay đổi



Hình 2. . Giao diện của HTTP Routers ban đầu

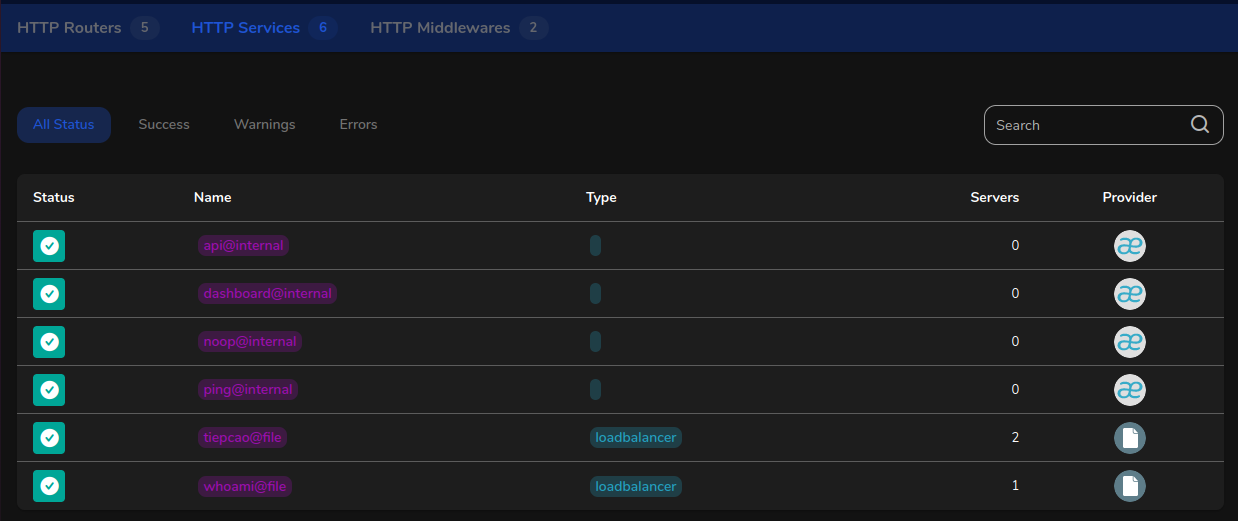
Giao diện của HTTP Routers sau khi thay đổi



Hình 2. . Giao diện của HTTP Routers sau khi thay đổi

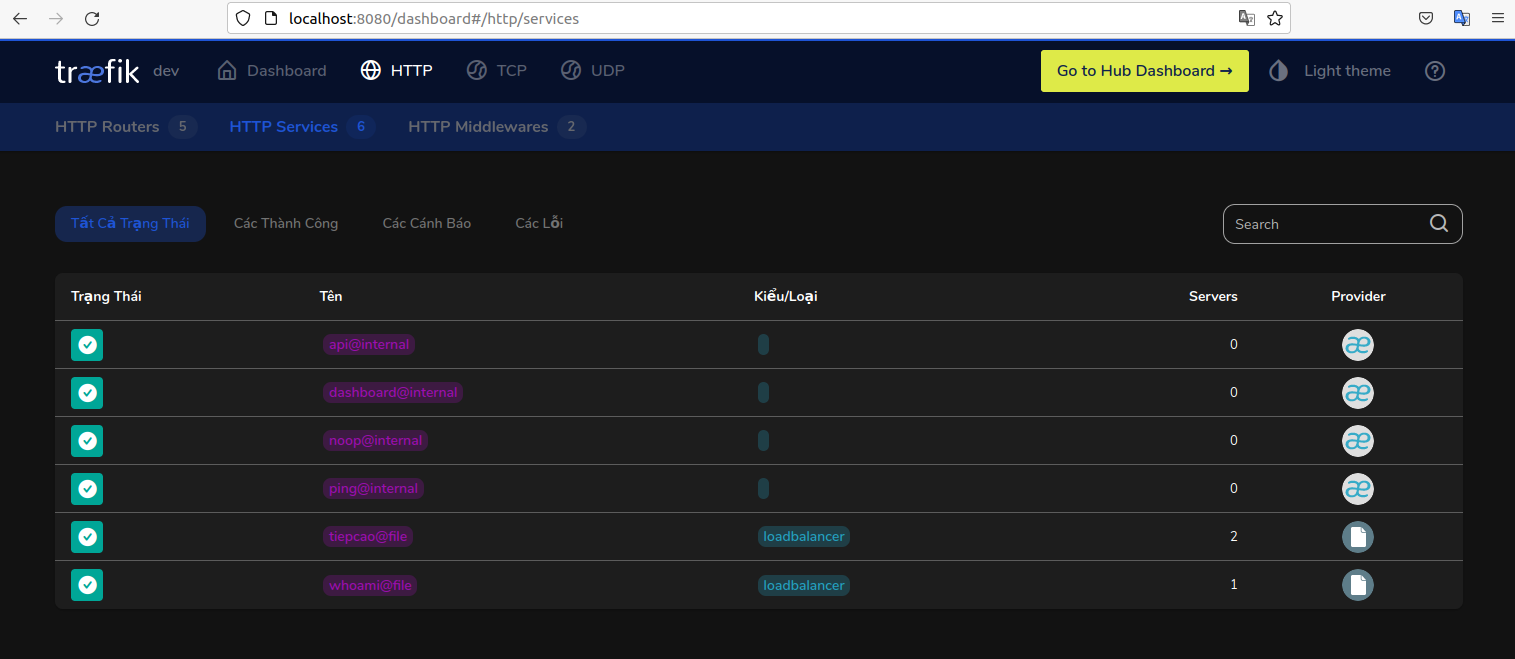
#### j. HTTP Services

Giao diện của HTTP Services trước khi thay đổi



Hình 2. . Giao diện của HTTP Services ban đầu

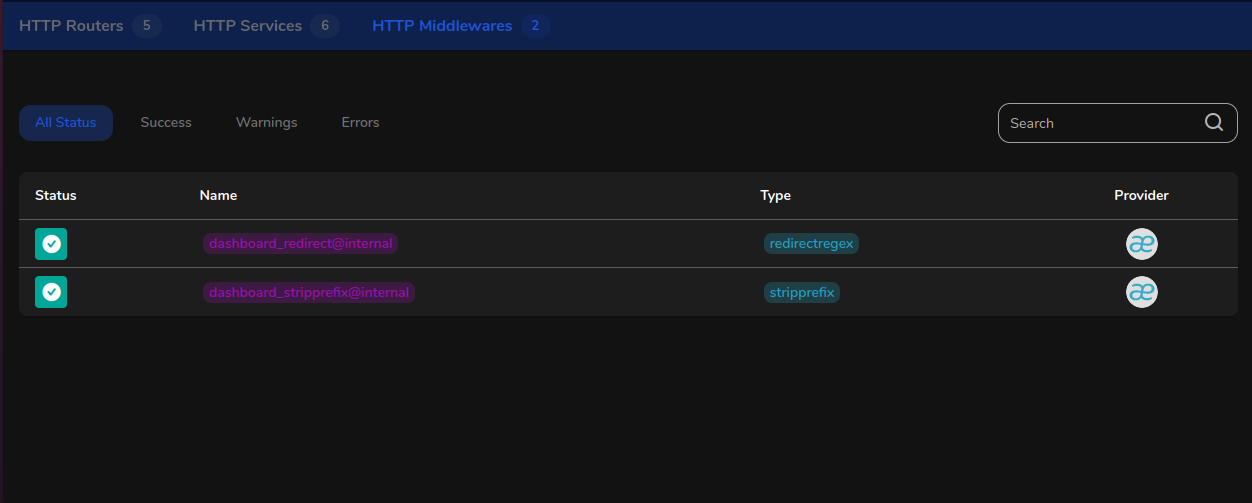
Giao diện của HTTP Services sau khi thay đổi



Hình 2. . Giao diện của HTTP Services sau khi thay đổi

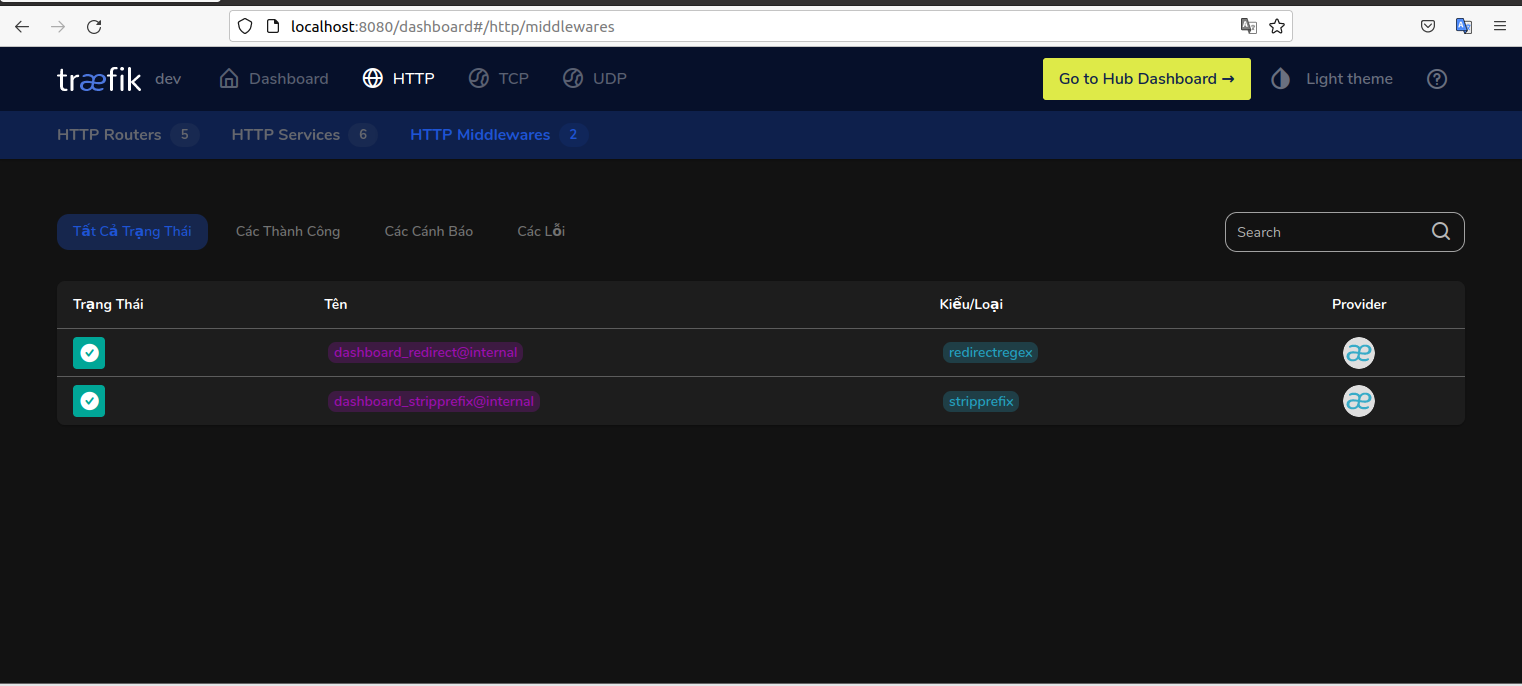
#### k. HTTP MiddleWares

Giao diện của HTTP MiddleWares trước khi thay đổi



Hình 2. . Giao diện của HTTP MiddleWares ban đầu

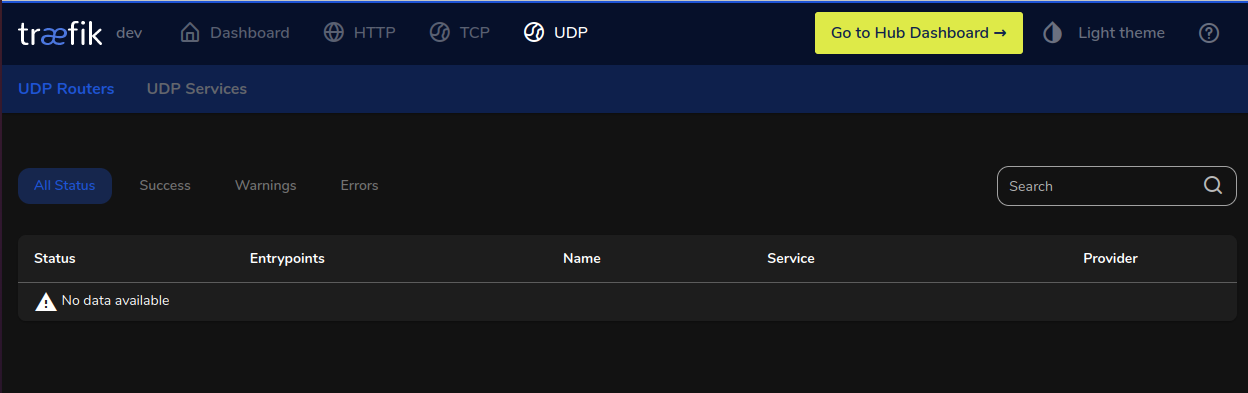
Giao diện của HTTP MiddleWares sau khi thay đổi



Hình 2. . Giao diện của HTTP MiddleWares sau khi thay đổi

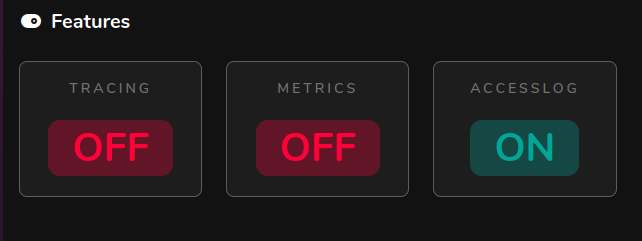
#### l. Features

Giao diện của Features trước khi thay đổi



Hình 2. 33. Giao diện của Features ban đầu

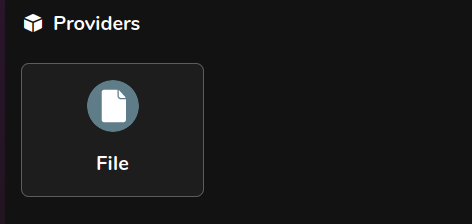
Giao diện của Features sau khi thay đổi



Hình 2. 34. Giao diện của Features sau khi thay đổi

#### m. Providers

Giao diện của Providers trước khi thay đổi



Hình 2. 35. Giao diện của Providers ban đầu

Giao diện của Providers sau khi thay đổi

Hình 2. 36. Giao diện của Providers sau khi thay đổi

* + 1. **Tùy biến tính năng HealthCheck**

Cấu hình HealthCheck kiểm tra tình trạng để loại bỏ các máy chủ không lành mạnh khỏi vòng xoay cân bằng tải. Traefik sẽ coi các máy chủ của bạn hoạt động tốt miễn là chúng trả về mã trạng thái giữa 2XX và 3XX cho các yêu cầu kiểm tra tình trạng (thực hiện trong mỗi khoảng thời gian (interval)). HealthCheck cho phép tự kiểm tra sức khỏe tự động cho services, tức là bất cứ khi nào một trong các services con của nó được báo cáo là ngừng hoạt động, services này sẽ biết về nó và tính đến nó (tức là nó bỏ qua con bị lỗi) khi chạy thuật toán cân bằng tải. Healthcheck chỉ kiểm tra tình trạng một lần duy nhất mà không cần biết trạng thái gần đây của máy chủ. Nếu như máy chủ đó không ổn định, thường xuyên bật tắt thì dẫn đến lãng phí tài nguyên khi HealthCheck liên tục phải phân phối lại tài nguyên cho máy chủ đó. Để khắc phục tình trạng đó cần phải kiểm tra tính ổn định của máy chủ đó mỗi khi quyết định bật tắt nó. Dưới đây là hai hàm dùng để kiểm tra trạng thái của máy chủ trước khi HealthCheck quyết định bật tắt nó.

#### a. Tính năng Check Down

Mục đích: Kiểm tra trạng thái của máy chủ trước khi cho máy chủ ngừng hoạt động

Bổ sung hàm *checkStatusForDown* vào hàm file healthcheck.go:

*checkStatusForDown(serverURL \*url.URL, firstArr [lenOfStatus]bool) (returnArr [lenOfStatus]bool, status bool) {*

*returnArr = firstArr*

*var count int = 0*

*for i := lenOfStatus - 1; i > 0; i-- {*

*returnArr[i] = returnArr[i-1]*

*}*

*returnArr[0] = false*

*for \_, val := range returnArr {*

*if !val {*

*count++*

*if count == lenOfStatus {*

*return returnArr, false*

*}*

*}*

*}*

*return returnArr, true*

*}*

Kết quả: Khi Server phản hồi bị lỗi trong 3 lần liên tục sẽ bị xem như là không đủ điều kiện hoạt động và sẽ được đổi sang trạng thái Down.

#### b. Tính năng Check Up

Mục đích: Kiểm tra trạng thái của máy chủ trước khi cho máy chủ hoạt động trở lại

Bổ sung hàm *checkStatusForUp* vào hàm checkHealth

*checkStatusForUp(serverURL \*url.URL, firstArr [lenOfStatus]bool) (returnArr [lenOfStatus]bool, status bool) {*

*returnArr = firstArr*

*for i := lenOfStatus - 1; i > 0; i-- {*

*returnArr[i] = returnArr[i-1]*

*}*

*returnArr[0] = true*

*for \_, val := range returnArr {*

*if !val {*

*return returnArr, false*

*}*

*}*

*return returnArr, true*

*}*

Kết quả: Khi Server phản hồi bình thường trong 3 lần liên tục sẽ được xem như là đủ điều kiện để hoạt động trở lại.

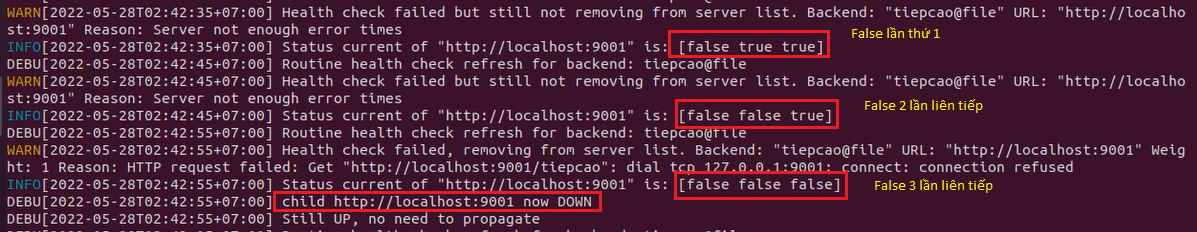
Ví dụ minh họa:

Chuẩn bị:

* + Cài đặt 1 services gồm 2 server.
  + Server 1, port = 9001
  + Server 2, port = 9002
  + Cài đặt biến n = 3

Thực hành:

*Case 1:* Kiểm tra trạng thái server(port = 9001) sau n lần kiểm tra fail liên tiếp.

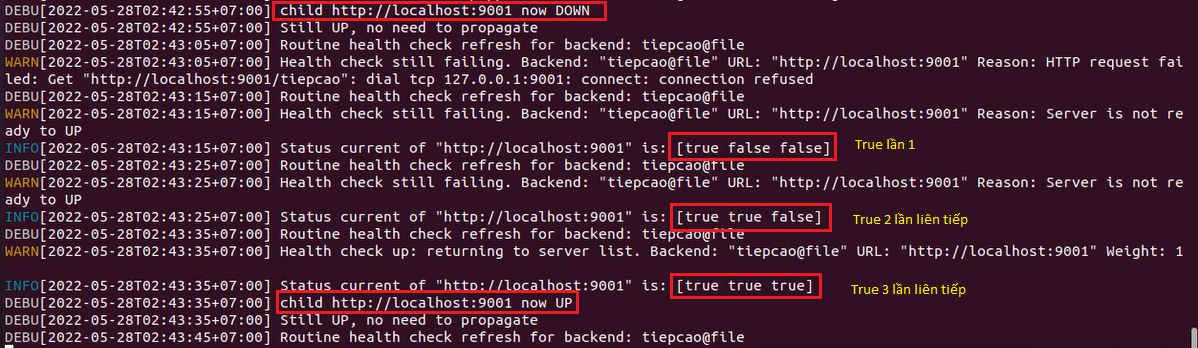


Hình 2. . Kiểm tra trạng thái của Server sau 3 lần false

Sau khi kiểm tra 3 lần false liên tiếp, server sẽ bị loại khỏi danh sách hoạt động.

*Case 2:* Kiểm tra trạng thái server(port = 9001) sau n lần kiểm tra success liên tiếp.

Điều kiện tiên quyết: Server đang trong trạng thái DOWN



Hình 2. . Kiểm tra trạng thái của Server sau 3 lần false

Sau khi kiểm tra 3 lần true liên tiếp, server sẽ được đưa lại vào danh sách hoạt động.

Kết luận: Sau khi thêm hai hàm *checkStatusForDown và checkStatusForUp,* HealthCheck sẽ kiểm tra những trạng thái gần nhất của máy chủ để quyết định máy chủ đó có đủ điện để được đưa vào hoạt động hay không và máy chủ đó có nên cho ngừng hoạt động hay không. Số lần kiểm tra trạng thái gần nhất của máy chủ phụ thuộc vào lựa chọn của người dùng khi cấu hình.

## Kết luận chương 2

Chương 2 đã trình bày về Traefik và tầm quan trọng của phương pháp Traefik trong Microservices. Chúng ta đã cũng tìm hiểu về cách hoạt động cũng như thành phần cấu tạo bên trong. Vì đây là một mã nguồn mở nên chúng ta có thể tùy chỉnh theo hệ thống mà chúng ta sử dụng sao cho phù hợp. Traefik hộ trỡ rất nhiều cho kiến trúc Microservices.

# CHƯƠNG 3. TRIỂN KHAI PHƯƠNG PHÁP TRAEFIK

* 1. **Mô tả bài toán**
* **Bài toán 1**

Một công ty công nghệ tại Nhật Bản sử dụng 1 phần mềm dùng để phân tích và mô hình hóa dữ liệu. Trong mô hình của ứng dụng có 1 service với nhiệm vụ là lấy dữ liệu từ database và có 5 server để xử lý cái request đến service đó.

* **Bài toán 2**

Một công ty công nghệ tại Nhật Bản sử dụng 1 phần mềm dùng để phân tích và mô hình hóa dữ liệu. Trong mô hình của ứng dụng có 2 service với nhiệm vụ là lấy dữ liệu từ database.Mỗi Serivce có 2 server để xử lý cái request đến service đó.

* 1. **Xây dựng kịch bản**
     1. ***Xây dựng kịch bản cho bài toán 1***

Bài toán đặt ra là chúng ta phải xây dựng các kịch bản để hiểu rõ hơn về việc cân bằng tải cho các server. Trong bài toán này ta sẽ thực nghiệm và đánh giá thông qua 4 kịch bản sau:

* *Kịch bản 1: Load balancing ở trường hợp bình thường (chưa sử dụng health check)*
* *Kịch bản 2: Khi có một số server bỗng dưng bị sập (chưa sử dụng health check)*
* *Kịch bản 3: Khi có một số server bỗng dưng bị sập (Sử dụng health check mặc định)*
* *Kịch bản 4: Khi có một số server bỗng dưng bị sập (Sử dụng health check đã được cải tiến)*
  + 1. ***Xây dựng kịch bản cho bài toán 2***

Bài toán đặt ra là chúng ta phải xây dựng các kịch bản để hiểu rõ hơn về việc cân bằng tải cho các service. Trong bài toán này ta sẽ thực nghiệm và đánh giá thông qua 2 kịch bản sau:

* *Kịch bản 5: Phân phối request đến service 1 và service 2 với tỉ lệ 50/50*
* *Kịch bản 6: Phân phối request đến service 1 và service 2 với tỉ lệ 80/20 (service 1/service 2)*

## Triển khai

### **Bài toán 1: Cân bằng tải cho server (Round Robin)**

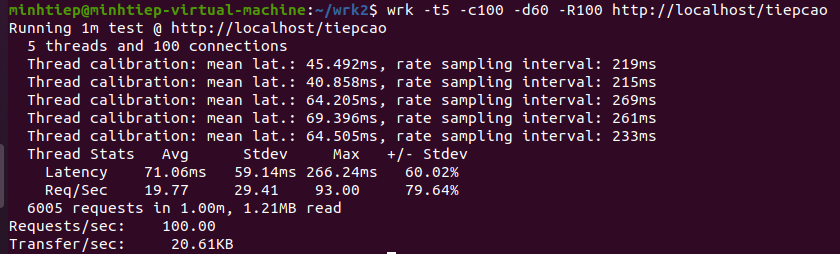
#### *Kịch bản 1: Load balancing ở trường hợp bình thường (không sử dụng health check)*

Chuẩn bị:

* + Cài đặt 1 services gồm 5 servers
  + Số Thread: 5
  + Số lượng connection: 100
  + Trong thời gian: 60s
  + Số lượng request/s : 100

Thực hành:

* + Bước 1: Khởi chạy Traefik
  + Bước 2: Triển khai các server
  + Bước 3: Thực hiện câu lệnh gửi request đi.
    - Tiến hành triển khai câu lệnh :



Hình 3. . Kết quả thực thi kịch bản 1

Kết quả: Tổng cộng 6005 requests in 1.00m, 1.21 MB read

* Requests/sec: 100.00
* Transfer/sec: 20.61 KB
* Số lượng request không phản hồi: 0 request

Kết luận: Khi chạy với kịch bản trên server chạy gần đạt được hiệu suất đề ra.

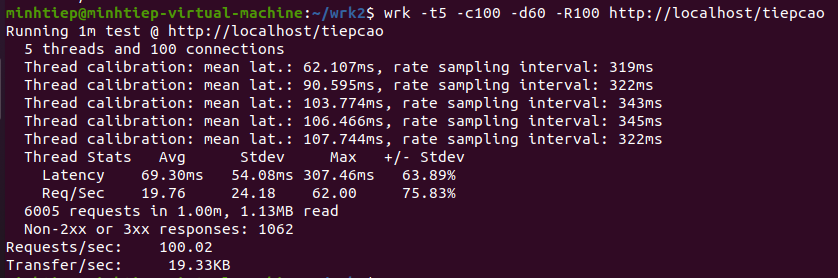
#### *Kịch bản 2: Khi có một số server bỗng dưng bị sập (không sử dụng health check )*

Chuẩn bị:

* + Cài đặt 1 services gồm 5 servers, 2 server bị sập (Server ở cổng 9001, 9002 sẽ bị sập sau khi hoạt động trong vòng 30s kể từ khi bắt đầu)
  + Số Thread: 5
  + Số lượng connection: 100
  + Trong thời gian: 60s
  + Setup a client call 100 request/s

Thực hành:

* + Bước 1: Khởi chạy Traefik
  + Bước 2: Triển khai các server
  + Bước 3: Thực hiện câu lệnh gửi request đi.
    - Tiến hành triển khai câu lệnh :



Hình 3. . Kết quả thực thi kịch bản 2

Kết quả: Tổng cộng 6005 requests in 1.00m, 1.13MB read

* Requests/sec: 100.02
* Transfer/sec: 19.33 KB
* Số lượng request không phản hồi: 1062 request

Kết luận: Bởi vì không sử dụng tính năng health check nên các request vẫn liên tục được gửi đến những server không hoạt động. Dẫn đến số lượng request không được phản hồi rất nhiều

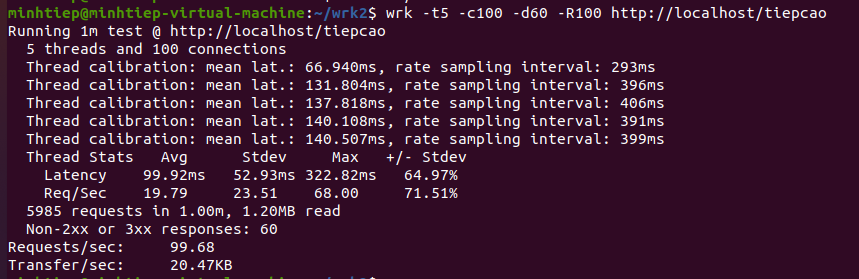
#### *Kịch bản 3: Khi có một số server bỗng dưng bị sập (Sử dụng health check mặc định)*

Chuẩn bị:

* + Cài đặt 1 services gồm 5 servers, 2 server bị sập (Server ở cổng 9001, 9002 sẽ bị sập sau khi hoạt động trong vòng 30s kể từ khi bắt đầu)
  + Số Thread: 5
  + Số lượng connection: 100
  + Trong thời gian: 60s
  + Setup a client call 100 request/s
  + interval = 10s
  + timeout = 3s

Thực hành:

* + Bước 1: Khởi chạy Traefik
  + Bước 2: Triển khai các server
  + Bước 3: Thực hiện câu lệnh gửi request đi.
    - Tiến hành triển khai câu lệnh :



Hình 3. . Kết quả thực thi kịch bản 3

Kết quả: Tổng cộng 5985 requests in 1.00m, 1.20MB read

* Requests/sec: 99.68
* Transfer/sec: 20.47 KB
* Số lượng request không phản hồi: 60 request

Kết luận: Bởi vì Traefik đã sử dụng chức năng health check nên khi 1 server bị sập sẽ được phát hiện trong thời gian quy định. Vì thế các request sẽ không gửi đến các server đang ngừng hoạt động nên các số lượng request không phản hồi được giảm thiểu đáng kể.

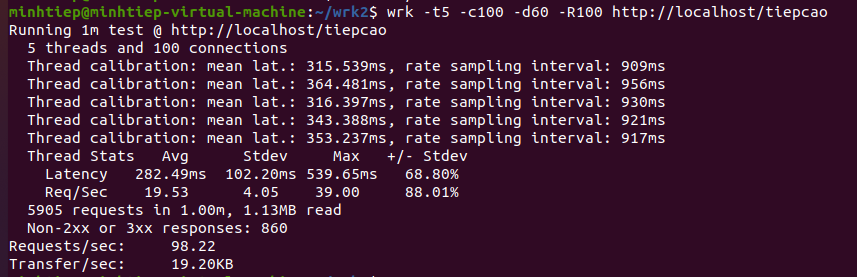
#### *Kịch bản 4: Khi có một số server bỗng dưng bị sập (Sử dụng health check đã được cải tiến)*

Chuẩn bị:

* Cài đặt 1 services gồm 5 servers, 2 server bị sập(Server ở cổng 9001, 9002 sẽ bị sập sau khi hoạt động trong vòng 30s kể từ khi bắt đầu)
* Số Thread: 5
* Số lượng connection: 100
* Trong thời gian: 60s
* Setup a client call 100 request/s
* interval = 10s
* timeout = 3s

Thực hành:

* Bước 1: Khởi chạy Traefik
* Bước 2: Triển khai các server
* Bước 3: Thực hiện câu lệnh gửi request đi.
* Tiến hành triển khai câu lệnh :



Hình 3. . Kết quả thực thi kịch bản 4

Kết quả: Tổng cộng 5905 requests in 1.00m, 1.13MB read

* Requests/sec: 98.22
* Transfer/sec: 19.20 KB
* Số lượng request không phản hồi: 860 request

Kết luận: Sau khi dùng heathcheck đã được sửa vì Traefik sẽ kiểm tra trạng thái server nhiều lần hơn đối với khi sử dụng health check mặc định nhưng khi kiểm tra đủ số lượng yêu cầu thì request sẽ không gửi đến những server bị lỗi nữa. Vì vậy số lượng request không phản hồi sẽ nhiều hơn khi sử dụng healthcheck mặc định và ít hơn so với khi chúng ta không sử dụng health check.

### **3.2.2. Bài toán 2: Cân bằng tải cho service (Weight Round Robin)**

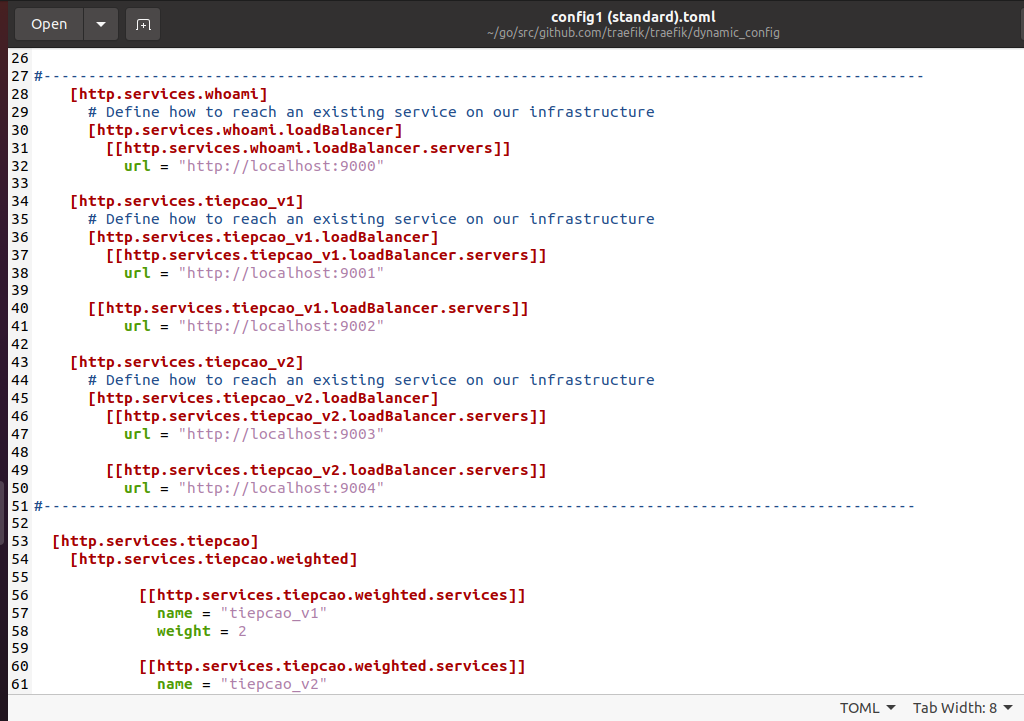
#### *Kịch bản 5: Cân bằng tải cho các Service ở trường hợp 50/50*

Chuẩn bị:

* + Cài đặt 2 Service. Mỗi service gồm 2 servers
  + Cài đặt weight round robin: 50/50.
  + Số Thread: 5
  + Số lượng connection: 100
  + Trong thời gian: 60s
  + Số request/s : 100

Thực hành:

* + Bước 1: Khởi chạy Traefik
  + Bước 2: Triển khai các server
  + Bước 3: Thực hiện câu lệnh gửi request đi.
    - Tiến hành triển khai câu lệnh :

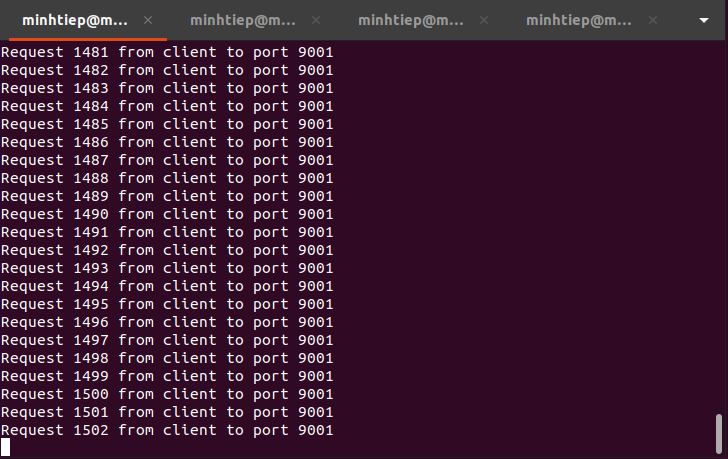


Hình 3. . File cấu hình cho kịch bản 5

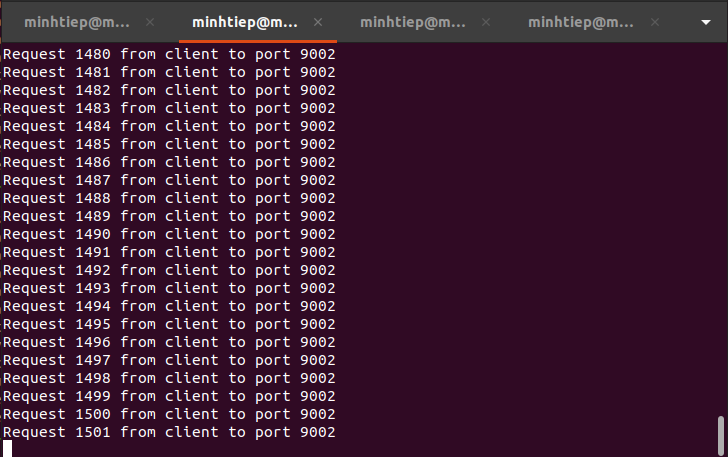
* Tiến hành triển khai câu lệnh :



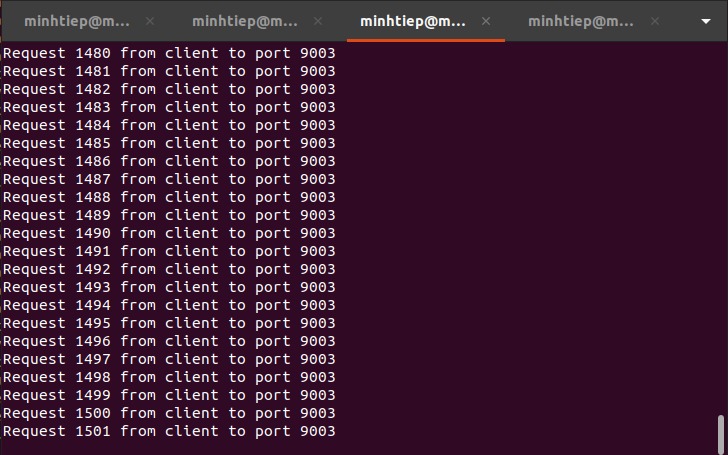
Hình 3. . Kết quả thực thi kịch bản 5



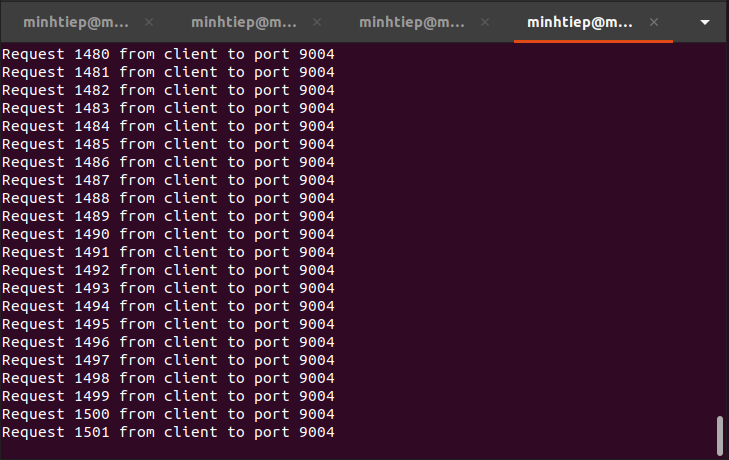
Hình 3. . Số request đến server 1 của service tiepcao\_v1



Hình 3. . Số request đến server 2 của service tiepcao\_v1



Hình 3. . Số request đến server 1 của service tiepcao\_v2



Hình 3. . Số request đến server 2 của service tiepcao\_v2

Kết luận: Trong trường hợp sử dụng thuật toán Weight Round Robin với tham số là 50/50, mỗi service đều được phân bố bằng nhau. Trong mỗi service có sử dụng thuật toán Round Robin vì vậy số request đến các server là đều nhau. Suy ra thuật toán hoạt động chính xác trong trường hợp này.

#### *Kịch bản 6: Cân bằng tải cho các Service ở trường hợp 80/20*

Chuẩn bị:

* + Cài đặt 2 Service. Mỗi Service gồm 2 servers
  + Setup round robin: 80/20.
  + Số Thread: 5
  + Số lượng connection: 100
  + Trong thời gian: 60s
  + Số request/s : 100

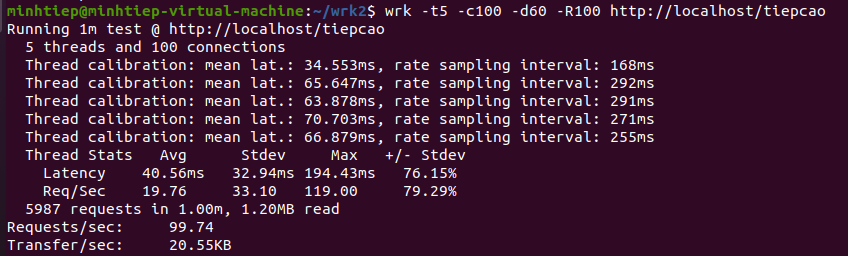
Thực hành:

* + Bước 1: Khởi chạy Traefik
  + Bước 2: Triển khai các server
  + Bước 3: Thực hiện câu lệnh gửi request đi.
    - Tiến hành triển khai câu lệnh :

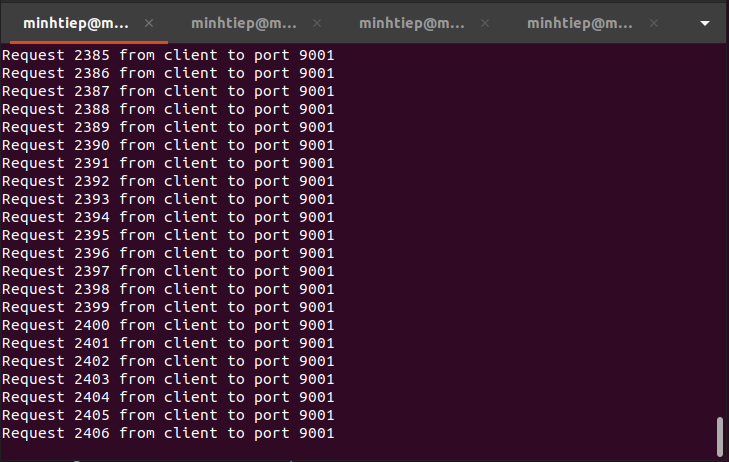


Hình 3. . File cấu hình cho kịch bản 6

* Tiến hành triển khai câu lệnh :



Hình 3. . Kết quả thực thi kịch bản 6



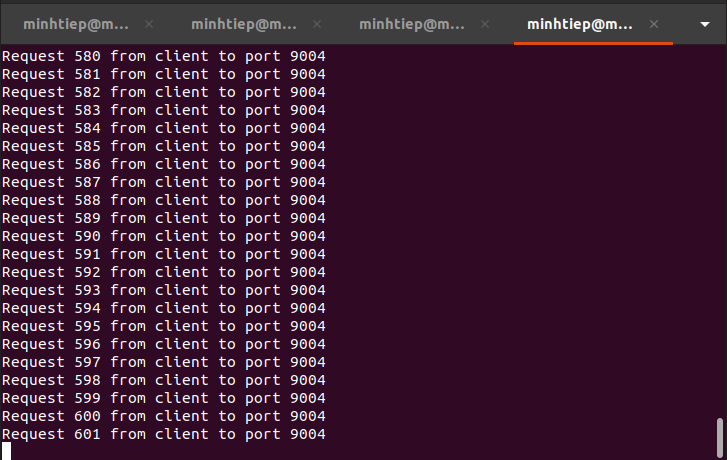
Hình 3. . Số request đến server 1 của service tiepcao\_v1



Hình 3. . Số request đến server 2 của service tiepcao\_v1



Hình 3. . Số request đến server 1 của service tiepcao\_v2



Hình 3. . Số request đến server 2 của service tiepcao\_v2

Kết luận: Trong trường hợp sử dụng thuật toán Weight Round Robin với tham số là 80/20, mỗi service đều được phân bố theo tỉ lệ 80/0. Trong mỗi service có sử dụng thuật toán Round Robin vì vậy số request đến các server trong một service là như nhau. Suy ra thuật toán đã hoạt động chính xác trong trường hợp này.

## Kết luận chương 3

Thông qua 6 kịch bản đã được trình bày tại chương 3, ta đã có cái nhìn tổng quan hơn về cơ chế và cách thức hoạt động của Traefik, đồng thời, hiểu rõ và áp dụng phương pháp Traefik vào trong từng bài toán thực tế cụ thể.

# KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN ĐỀ TÀI

Trên đây là toàn bộ nội dung đề tài tìm hiểu và nghiên cứu giải pháp Traefik cho cân bằng tải và reverse proxy cho Microservice. Qua ba chương, đồ án đã đạt được một số mục tiêu đề ra, cụ thể như sau:

Chương 1 đã hệ thống được tìm hiểu về Microservices và cân bằng tải trong kiến trúc Microservices như là cách khái niệm, cách thức giao tiếp trong kiến trúc và những phương pháp cân bằng tải trong Microservices. Qua đó giúp chúng ta có kiến thức cơ bản để bước vào tìm hiểu chương 2 và 3 của đồ án.

Khi bước vào chương 2, chúng ta đi sâu vào phân tích và phát triển phương pháp cân bằng tải Traefik. Trong chương này chúng ta sẽ hiểu rõ hơn về cách thức phương pháp Traefik cân bằng tải cũng và cách quản lý các thành phần của hệ thống.

Chương cuối cùng dựa trên những hiểu biết đã có của 2 chương trước. Sau khi đặt được mục tiêu ban đầu là triển khai phương pháp cân bằng tải Traefik cho cân Microservices, ta sẽ áp dụng những phát triển đã có ở chương 2 vào các bài toán thực tế cụ thể. Điều đó làm cho chúng ta có thể áp dụng Traefik vào nhiều hệ thống hơn thông qua việc tùy chỉnh cấu hình của nó.

Bên cạnh những điều mà Traefik đã có sẵn, vẫn còn rất nhiều tiềm năng để phát triển và ứng dụng nó rộng rãi hơn. Trong thực tế chúng ta cần thay đổi những thuật toán cân bằng tải để phù hợp với các dự án và Traefik có hỗ trợ nhiều thuật toán cân bằng tải nên chúng ta có thể tích hợp thêm nhiều lựa chọn thuật toán cân bằng tải hơn cho người dùng. Điều đó sẽ tăng thêm tính linh hoạt cho phương pháp cân bằng tải Traefik.

# TÀI LIỆU THAM KHẢO

***Tài liệu tiếng anh:***

[1] Hao Wang, Yong Wang, Guanying Liang, Yunfan Gao (2021*), Research on load balancing technology for Microservice architecture*

[2] Chris Richardson (2018*),* *Microservices Patterns*

[3] Gerard Blokdyk (2017), *Load Balancing: Beginner's Guide - Third Edition*

[4] Yipei Niu, Fangming Liu, Zongpeng Li (2048), *Load Balancing across Microservices*

***Tài liệu online:***

[1] https://traefik.io

[2] https://middleware.io/blog/Microservices-architecture/

[3] https://iximiuz.com/en/posts/traefik-canary-deployments-with-weighted-load-balancing/

[4] <https://www.atatus.com/glossary/load-balancer/>

[5] <https://traefik.io/glossary/reverse-proxy/>

[6] [https://kemptechnologies.com/load-balancer/load-balancing-algorithms- techniques/](https://kemptechnologies.com/load-balancer/load-balancing-algorithms-%20%20%20%20%20%20%20%20techniques/)

[7] <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>

[8] <https://github.com/traefik/traefik/wiki/Awesome-Traefik>

[9] <https://community.traefik.io/>

[10] <https://medium.com/@utnas/microservices-architecture-using-traefik-reverse-proxy-and-ambassador-as-gateway-606a77d8f7c7>

[11] <https://medium.com/must-know-computer-science/system-design-load-balancing-1c2e7675fc27>

[12] https://www.infopulse.com/blog/the-importance-of-microservices-architecture-for-modern-applications