**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI**

**PHÂN HIỆU TẠI TP. HỒ CHÍ MINH**

**BỘ MÔN CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO ĐỒ ÁN TỐT NGHIỆP**

**ĐỀ TÀI: CHỨNG THỰC VÀ PHÂN QUYỀN TRONG HỆ THỐNG MICROSERVICES**

Giảng viên hướng dẫn: PHẠM THI VƯƠNG

Sinh viên thực hiện: PHẠM VĂN TỊNH

Lớp : CÔNG NGHỆ THÔNG TIN

Khoá :K57

Tp. Hồ Chí Minh, năm 2020

|  |  |
| --- | --- |
| TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI  PHÂN HIỆU TẠI TP. HỒ CHÍ MINH  **BỘ MÔN CÔNG NGHỆ THÔNG TIN**  --------------------------------------- | CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM  Độc lập – Tự do - Hạnh phúc  --------------------------------------------------- |

**NHIỆM VỤ THỰC HIỆN ĐỒ ÁN TỐT NGHIỆP**

Họ và tên sinh viên: PHẠM VĂN TỊNH

MSSV: 5751071043

Chuyên ngành: Công nghệ thông tin

Lớp: CQ.K57.CNTT

1. Tên đề tài đồ án tốt nghiệp nghiệp:

Chứng Thực và Phân Quyền trong hệ thống Microservices.

1. Nhiệm vụ thực hiện đồ án tốt nghiệp:

* Tìm hiểu cấu trúc Microservices.
* Xây dựng hệ thống xác thực phân quyền trên nền tảng kiến trúc Microservices.

1. Ngày bắt đầu thực tập tốt nghiệp: ngày … tháng …. năm …….
2. Ngày hoàn thành báo cáo thực tập tốt nghiệp: ngày …. tháng ….. năm …….
3. Họ tên giáo viên hướng dẫn: PHẠM THI VƯƠNG

|  |  |
| --- | --- |
| **Trưởng bộ môn** | *Tp. Hồ Chí Minh, ngày …. Tháng …..năm* …….  Giáo viên hướng dẫn  Phạm Thi Vương |
|  | |

**LỜI CẢM ƠN**

Trước tiên với tình cảm sâu sắc và chân thành nhất, cho phép em tỏ lòng biết ơn thầy PHẠM THI VƯƠNG (giảng viên trường Đại Học Sài Gòn) đã tạo điều kiện hỗ trợ, giúp đỡ em trong suốt quá trình học tập và nghiên cứu đề tài Xác Thực và Phân Quyền Trong Hệ Thống Microservices. Trong suốt thời gian từ khi bắt đầu thực hiện đồ án thực tập tốt nghiệp đến nay, em nhận được rất nhiều sự quan tâm giúp đỡ của thầy Phạm Thi Vương.

Với lòng biết ơn sâu sắc nhất, em xin gửi đến quý Thầy Cô ở bộ môn Công Nghệ Thông Tin trường Đại Học Giao Thông Vận Tải phân hiệu TPHCM đã truyền đạt vốn kiến thức cho chúng em trong suốt quá trình học tập tại trường. Nhờ có những lời hướng dẫn, dạy bảo của thầy cô nên đề tài nghiên cứu của em mới có thể hoàn thành tốt đẹp.

Một lần nữa, em xin chân thành cảm ơn thầy cô đã trực tiếp giúp đỡ, quan tâm, hướng dẫn em hoàn thành tốt bài báo cáo này trong thời gian qua.

Bài báo cáo thực tập thực hiện trong khoảng thời gian gần 3 tuần. Bước đầu đi vào thực tế của em còn nhiều hạn chế và còn nhiều bỡ ngỡ nên không tránh khỏi thiếu sót, em rất mong nhận được những ý kiến đóng góp quý báu của quý Thầy Cô để kiến thức của em trong lĩnh vực này được hoàn thiện hơn đồng thời có điều kiện bổ sung, nâng cao ý thức của mình.

Em xin chân thành cảm ơn!

|  |  |
| --- | --- |
|  | *Tp. Hồ Chí Minh, ngày …. Tháng …..năm* 2020  Sinh viên  Phạm Văn Tịnh |

**NHẬN XÉT CỦA GIÁO VIÊN**

|  |  |
| --- | --- |
|  | *Tp. Hồ Chí Minh, ngày …. Tháng …..năm* …….  Giáo viên hướng dẫn  Phạm Thi Vương |

**MỤC LỤC**

**DANH MỤC CHỮ VIẾT TẮT**

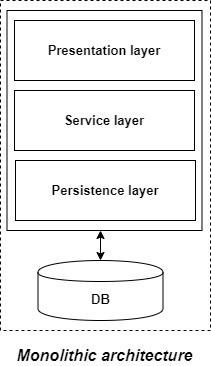
|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Danh từ** | **Mô tả** | **Ghi chú** |
| **1** | **Authentication** | Xác thực |  |
| **2** | **Authorization** | Ủy quyền/phân quyền |  |
| **3** | **OAuth** | Là tiêu chuẩn mở cho ủy quyền truy cập |  |
| **4** | **Monolithic Architecture** | Kiểu kiến trúc phần mềm cũ. Kiến trúc nguyên khối |  |
| **5** | **Microservices** | Kiểu kiến trúc phân tán nhiều dịch vụ nhỏ |  |
| **6** | **Docker Container** | Docker Container một đơn vị phần mềm tiêu chuẩn đóng gói mã và tất cả các phụ thuộc của nó để ứng dụng chạy nhanh hơn. |  |
| **7** | **JWT** | Json Web Token là một chuẩn mở (RFC 7519) định nghĩa một cách nhỏ gọn và khép kín để truyền một cách an toàn thông tin giữa các bên dưới dạng đối tượng Json |  |
| **8** | **Framework** | Là các đoạn code viết sẵn, cấu thành nên một bộ khung và các thư viện lập trình được đóng gói. |  |
| **9** | **Agile** | Là phương pháp phát triển phần mềm linh hoạt. |  |
| **10** | **Services** | Dịch vụ |  |
| **11** | **Cookie** | Http cookie, web cookie, internet cookie, trình duyệt cookie là những tập tin một trang web gửi đến máy người dùng và được lưu lại thông qua trình duyệt. |  |

* 1. **CHƯƠNG I - MỞ ĐẦU**
  2. **Mở đầu đề tài**

Hiện nay kiến trúc Microservices (kiến trúc phân tán), đang là chủ đề được cộng đồng Developer (lập trình viên, nhà phát triển phần mềm) vô cùng quan tâm. Chúng ta có thể tìm thấy khá nhiều tài nguyên giới thiệu và nói về tính chất cũng như lợi ích của Microservices, tuy nhiên không ai cũng hiểu và có cái nhìn khách quan về các ưu điểm, nhược điểm của Microservices và từ đó tìm ra phương pháp áp dụng khoa học và hiệu quả cho công việc, dự án của mình.

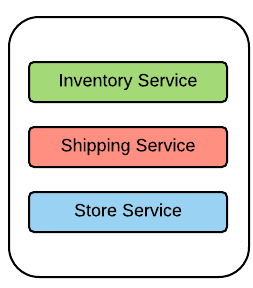
*Để hiểu Microservices, chúng ta cần tìm hiểu các ứng dụng nguyên khối là gì và điều gì đã khiến chúng ta chuyển từ các ứng dụng có cấu trúc Monolithic (cấu trúc nguyên khối) sang ứng dụng có cấu trúc Microservices (cấu trúc phân tán) trong thời gian gần đây.*

* 1. **Kiến trúc một khối (Monolithic Architecture)**



Các ứng dụng doanh nghiệp ngày nay đang được thiết kế để đáp ứng được số lượng lớn nghiệp vụ kinh doanh. Do đó một ứng dụng phần mềm cần cung cấp hàng trăm chức năng và tất cả những chức năng như vậy được gói gọn trong một ứng dụng nguyên khối duy nhất. ERP, CRM và các hệ thống phần mềm khác nhau là những ví dụ điển hình – chúng được xây dựng dưới dạng nguyên khối với hàng trăm chức năng. Việc triển khai, xử lý sự cố, mở rộng và nâng cấp các ứng dụng như vậy quả là một cơn ác mộng đối với bất kỳ doanh nghiệp nào.

Kiến trúc hướng dịch vụ (service-oriented architecture - SOA) được thiết kế để khắc phục các vấn đề phát sinh từ ứng dụng một khối (monolithic) bằng cách đưa ra khái niệm services. Do đó, với SOA, một ứng dụng sẽ được thiết kế dưới dạng kết hợp các service khác nhau. Khái niệm SOA không có nghĩa là biến từng service thành một khối riêng, nhưng hầu hết các ứng dụng triển khai theo SOA đều có hướng triển khai từng service dưới dạng một khối có cùng thời gian chạy (runtime). Vì vậy, tương tự ứng dụng một khối, các service này theo thời gian cũng tích lũy nhiều nghiệp vụ và chức năng khác nhau. Sự tăng trưởng này sẽ sớm biến những service đó thành những khối u nguyên khối, không khác gì ứng dụng thông thường.



Hình trên cho thấy một ứng dụng bán lẻ bao gồm nhiều dịch vụ. Tất cả các service này được triển khai trên cùng một ứng dụng runtime. Do đó nó cho thấy một số đặc điểm của một ứng dụng nguyên khối: nó phức tạp, được thiết kế, phát triển và triển khai trong cùng một đơn vị duy nhất; nó quá khó để áp dụng các phương pháp phát triển linh hoạt và phân phối nhanh; cập nhật một phần ứng dụng sẽ bắt buộc phải triển khai lại toàn bộ ứng dụng.

Ngoài ra, còn một số vấn đề đối với kiến trúc một khối: Nó sẽ khó được hỗ trợ nếu có xung đột về yêu cầu về tài nguyên (ví dụ như một service cần nhiều CPU hơn trong khi service khác lại cần nhiều bộ nhớ hơn). Một service không ổn định có thể làm cả ứng dụng bị chết và thông thường nó khó có thể đổi mới và áp dụng các công nghệ lập trình mới.

*Những đặc điểm này là khởi đầu đẫn đến kiến trúc microservices ra đời.*

* 1. **Nhược điểm của các ứng dụng nguyên khối:**
* Nó trở nên quá lớn về kích thước theo thời gian và dẫn đến khó quản lý.
* Chúng ta cần triển khai lại toàn bộ hệ thống ngay cả khi có một thay đổi nhỏ.
* Khi kích thước của ứng dụng tăng lên, thời gian khởi động và triển khai của nó củng tăng lên.
* Đối với bất kỳ nhà phát triển nào mới tham gia dự án, rất khó để hiểu logic của ứng dụng nguyên khối lớn ngay cả khi trách nhiệm chỉ liên quan đến một chức năng duy nhất.
* Ngay cả khi một ứng dụng phải đối mặt với lượng truy cập lớn, chúng ta cần phải triển khai các phiên bản của toàn bộ ứng dụng lên trên nhiều máy chủ. Nó rất không hiệu quả và chiếm nhiều tài nguyên không cần thiết. Do đó, tỷ lệ hiệu quả không khả dụng đối với ứng dụng có cấu trúc nguyên khối.
* Rất khó để áp dụng bất kỳ công nghệ mới nào phù hợp với chức năng cụ thể vì nó ảnh hưởng đến toàn bộ ứng dụng, cả thời gian và chi phí.
* Nó không đáng tin cậy bởi vì một lỗi trong bất kỳ mô-đun nào có thể làm hỏng toàn bộ ứng dụng có cấu trúc nguyên khối.
  1. **Ưu điểm của các ứng dụng nguyên khối:**
* Đơn giản để phát triển khi so sánh với microservices nơi mà các nhà phát triển cần có kỹ năng đạt yêu cầu để xác định và phát triển dịch vụ.
* Dể dàng triển khai.
* Tương đối dể dàng và đơn giản để phát triển so với kiến trúc Microservices.
* Các vấn đề về độ trễ và bảo mật mạng tương đối ít so với kiến trúc Microservices
  1. **CHƯƠNG II - MICROSERVICES**
  2. **Kiến trúc Microservices**

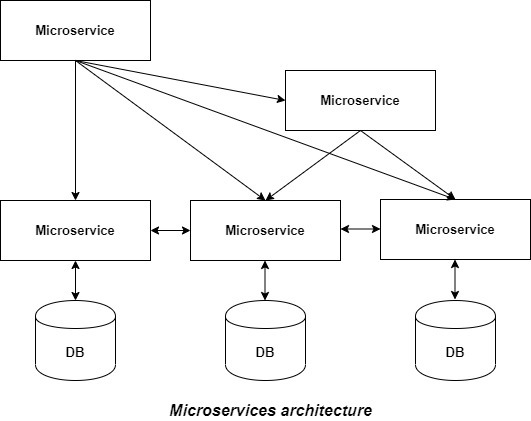
Hầu hết các định nghĩa về kiến trúc Microservices giải thích nó như là một khái niệm kiến trúc tập trung vào việc tách các service sẵn có trong kiến trúc một khối thành các service độc lập. Tuy nhiên, microservices không chỉ làm những công việc phân chia như thế.

Hãy xem xét điều đó bằng cách nhìn các chức năng trong kiến trúc một khối bằng cách xác định khả năng nghiệp vụ cần có từ ứng dụng – đó là trả lời câu hỏi ứng dụng cần làm gì, có ích hay không? Sau đó những khả năng đó có thể được triển khai trên các nền tảng công nghệ khác nhau, nhưng mỗi service sẽ giải quyết một phạm vi kinh doanh rất cụ thể và hạn chế.

Bằng cách này, kiến trúc hệ thống bán lẻ như hình 1 có thể được mô tả lại như sau:



Vậy Microservices là một phong cách phát triển kiến trúc, trong đó ứng dụng được tạo thành từ các dịch vụ nhỏ hơn giao tiếp với nhau bằng các giao thức trọng lượng nhẹ (light weight protocols) như HTTP. Theo Sam Newman: “Microservices are the small services that work together” Microservices là các dịch nhỏ hoạt động cùng với nhau.



Kiến trúc Microservices có tác động đáng kể đến mối quan hệ giữa ứng dụng và cơ sở dữ liệu. Thay vì chia sẻ một cơ sở dữ liệu với các dịch vụ nhỏ khác, mỗi microservice có cơ sở dữ liệu riêng. Nó thường dẫn đến sự trùng lặp một số dữ liệu nhưng có cơ sở dữ liệu riêng trên từng microservice là điều cần thiết nếu chúng ta muốn hưởng lợi từ kiến trúc này vì nó đảm bảo “khớp nối lỏng lẻo”. Một lợi thế khác của việc có cơ sở dữ liệu riêng cho mỗi microservice là mỗi microservice có thể sử dụng loại cơ sở dữ liệu phù hợp nhất cho nhu cầu của chính nó.

* 1. **Kiến trúc hướng dịch vụ (SOA) và kiến rúc Microservices**
     1. **Kiến trúc hướng dịch vụ Service Oriented Architecture (SOA)**

Kiến trúc hướng dịch vụ (SOA) là tập hợp một số lượng lớn các dịch vụ trong đó các dịch vụ giao tiếp với nhau thông qua một giao thức truyền thông trên mạng. Các nguyên tắc cơn bản của SOA đó là độc lập với các nhà cung cấp, sản phẩm và công nghệ.

Về cơ bản SOA là tập hợp toàn bộ các dịch vụ kết nối “mềm dẻo” với nhau và có giao tiếp. Chúng được định nghĩa một cách rõ ràng, hoàn toàn độc lập với nền tảng hệ thống và có thể tái sử dụng. Đây là cấp độ cao hơn của việc phát triển ứng dụng có kiến trúc một khối.

* + 1. **Sự khác biệt giữa SOA và Microservices**

|  |  |
| --- | --- |
| **SOA** | **MICROSERVICES** |
| Là tập hợp một số lượng lớn các dịch vụ giao tiếp với nhau. | Là kiến trúc có số lượng lớn các dịch vụ và chia thành các dịch vụ nhỏ hoặc các thành phần có thể chia sẻ. |
| Hỗ trợ nhiều giao thức giao tiếp khác nhau. | Hỗ trợ các giao thức giao tiếp nhẹ như HTTP/REST/gRPC. |
| Sử dụng Container để chứa các dịch vụ là ít phổ biến. | Sử dụng Container để chứa các dịch vụ hầu như là chắc chắn. |
| Sử dụng cơ sở dữ liệu truyền thống. | Sử dụng các cơ sở dữ liệu không liên quan đến nhau. |
| Sử dụng các dịch vụ ESB để liên lạc | Không sử dụng các dịch vụ ESB. Microservices có một hệ thống nhắn tin đơn giản. |
| Có tiêu chuẩn quản trị hệ chung. | Quản trị thỏa mái với tập trung nhiều hơn vào mọi người. |

* 1. **Các nguyên tắc của Microservices**
* **Single responsibility**: (trách nhiệm duy nhất) Đây là một nguyên tắc được xác định là một phần của thiết kế SOILD. Tức là, mỗi service cần có phạm vi và bị giới hạn về nghiệp vụ cụ thể, không phụ thuộc lẫn nhau. Điều đó giúp chúng ta đáp ứng được sự phát triển linh hoạt và nhanh chóng trong cung cấp các service.
* **Built around business capabilities**: (được xây dựng xung quanh khả năng kinh doanh) trong thế giới ngày nay, nơi có rất nhiều công nghệ tồn tại, luôn có một công nghệ phù hợp để thực hiện một chức năng cụ thể. Nhưng trong các ứng dụng nguyên khối, đó là nhược điểm lớn, vì chúng ta không thể sử dụng công nghệ khác nhau cho từng chức năng và do đó, cần phải linh hoạt trong các lĩnh vực cụ thể. Một microservice sẽ không bao giờ hạn chế việc áp dụng các công nghệ hoặc lưu trữ dữ liệu phù hợp nhất để giải quyết vấn đề kinh doanh, tức mỗi microservice có thể sử dụng công nghệ khác nhau dựa trên các yêu cầu kinh doanh.
* **Design for failure**: (thiết kế cho sự thất bại) Microservices phải được thiết kế với các trường hợp thất bại có thể xảy ra. Microservices phải khai thác các lợi thế của kiến trúc này đó là khi một service gặp lỗi thì sẽ không ảnh hưởng đến toàn bộ hệ thống, các chức năng khác phải vẫn có thể truy cập được đối với người dùng. Nhưng đối với trường hợp là ứng dụng được xây dựng theo kiến trúc nguyên khối, sự thất bại của một mô-đun sẽ dẫn đến sự sụp đổ của cả một hệ thống.
  1. **Ưu điểm của Microservices**
* Microservices rát dể quản lý vì kích thước tương đối nhỏ.
* Nếu có bất kỳ cập nhật nào trong một dịch vụ, thì chúng ta chỉ cần triển khai lại dịch vụ đó.
* Microservice khép kín và do đó, được triển khai độc lập. Thời gian khởi động và triển khai tương đối ít hơn.
* Nhà phát triển có thể rất dể dàng tham gia dự án bởi vì chỉ cần hiểu một dịch vụ cụ thể cung cấp chức năng mà nhà phát triển sẽ làm việc chứ không phải toàn bộ hệ thống.
* Nếu một service cụ thể đang đối mặt với một lượng truy cập lớn do người dung sử dụng chức năng đó vượt quá thì chúng ta có thể mở rộng quy mô service đó. Do đó, kiến trúc Microservices hỗ trợ mở rộng theo chiều ngang.
* Mỗi service có thể bị hỏng do một số lỗi, thì nó sẽ không ảnh hưởng đến các service khác và toàn bộ hệ thống sẽ được nguyên vẹn, tiếp tục cung cấp các chức năng khác cho người dùng.
  1. **Nhược điểm của Microservices**
* Là một hệ thống phân tán, nó phức tạp hơn nhiều so với các ứng dụng nguyên khối. Độ phức tạp của nó tăng lên cùng với sự gia tăng số lượng của service.
* Các nhà phát triển cần phải có các kỹ năng được yêu cầu làm việc với kiến trúc Microservices để có thể xác định các nhiệm vụ của các service và cách liên lạc giữa chúng.
* Việc triển khai các service rất phức tạp.
* Các microservice rất tốn kém về mặt sử dụng mạng vì chúng cần tương tác với nhau và tất cả các liên lạc này đều chịu ảnh hưởng bởi độ trễ của mạng.
* Các ứng dụng có kiến trúc Microservices kém an toàn hơn ứng dụng có kiến trúc nguyên khối do giao tiếp giữa các dịch vụ thông qua mạng.
* Gỡ lỗi là khó khăn khi kiểm soát luồng dữ liệu được chạy nhiều microservice và để chỉ ra lý do tại sao và chính chính lỗi đã xảy ra là một nhiệm vụ khó khăn.
  1. **Các khái niệm chính trong Microservices**
     1. **Thiết kế Microservices:**

Chúng ta có thể làm một trong hai điều khi nói về kiến trúc Microservices: Một là xây dựng ứng dụng của mình từ đầu và hai là chuyển đổi ứng dụng của mình hiện có thành microservice. Dù bằng cách nào, điều quan trọng là bạn phải quyết định đúng kích thước, phạm vi và khả năng của các microservice. Đây có lẽ là điều khó nhất mà bất kể nhà phát triển nào cần gặp phải khi thực hiện kiến trúc Microservices trong thực tế.

Dưới đây là một số quan niệm sai lầm về kiến trúc Microservices:

* Số dòng code/kích thước team: Có một số tranh luận về việc quyết định kích thước của microservice được dựa trên số dòng code của việc triển khai hoặc kích thước của một team tham gia vào xây dựng. Đây được coi là những số liệu không thực tế và tệ hại.
* Micro là một thuật ngữ bị hiểu sai: Nhiều lập trình viên có xu hướng nghĩ rằng họ nên cố làm cho service nhỏ nhất có thể. Đây là một hiểu nhầm, không phải càng nhỏ thì càng tốt.
* Trong bối cảnh web service, các service thường được triển khai ở các mức độ chi tiết khác nhau – từ một vài chức năng đến vài chục chức năng. Việc có các service và đặt chúng dưới dạng microservice sẽ không đem lại bất kỳ lợi ích nào đối với ứng dụng có kiến trúc Microservices.
  + 1. **Message trong Microservices:**

Trong kiến trúc một khối, các chức năng nghiệp vụ của các mô-đun khác nhau sẽ gọi nhau bằng cách sử dụng hàm gọi hoặc các method gọi hàm của ngôn ngữ lập trình. Trong SOA, điều này được chuyển sang cách request/response các message service có tính lỏng lẽo hơn, chủ yếu dựa trên SOAP với nhiều giao thức khác nhau như HTTPS, JMS.

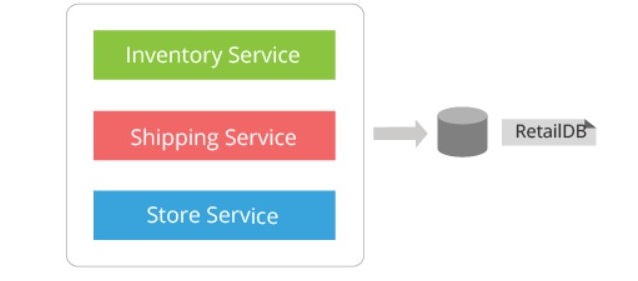
* Tin nhắn đồng bộ(Synchronous Messaging) – REST, Thrift: đối với tin nhắn đồng bộ (client yêu cầu phản hồi kịp thời từ các service và thời gian đợi để nhận được nó) trong kiến trúc Microservices, REST là lựa chọn dễ nhất vì nó cung cấp một kiểu message đơn giản với các request, response HTTP. Do đó hầu hết các microservice đều sử dụng HTTP bên cạnh các tài nguyên khác (mỗi chức năng đại diện cho một tài nguyên và các hoạt động sẽ được thực hiện bởi các tài nguyên khác).



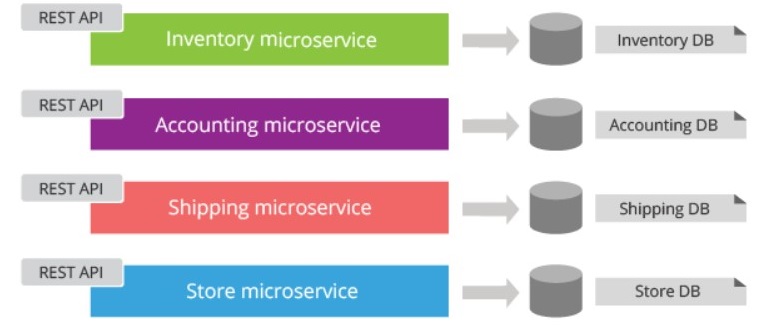
Thift được sử dụng như một thay thế cho REST/HTTP trong đồng bộ tin nhắn.

* Tin nhắn không đồng bộ (Asynchronous) – AMQP, STOMP, MQTT: Đối với một số hoàn cảnh chúng ta bắt buộc phải sử dụng các tin nhắn không đồng bộ (client không muốn phản hồi ngay lập tức hoặc hoàn toàn không phản hồi). Trong những tình huống này, các giao thức tin nhắn không đồng bộ như AMQP, STOPM hay MQTT được sử dụng phổ biến.
* Message Format – JSON, XML, Thrift, ProtoBuf, Avro: Một yếu tố quan trọng khác là định dạng của message. Các ứng dụng nguyên khối truyền thống sử dụng các định dạng nhị phân phức tạp, còn SOA/web service lại sử dụng các message dựa trên các định dạng message phức tạp (SOA) hay schema(xsd). Hầu hết các ứng dụng dựa trên kiến trúc Microservices sử dụng các định dạng message dựa trên văn bản đơn giản như JSON hay XML trên HTTP REST API. Trong trường hợp chúng ta cần định dạng message nhị phân (do message văn bản có thể trở nên dài dòng), microservices có thể tận dụng định dạng message nhị phân như binary Thrift, ProtoBuf hay Avro.
* Service Contracts – Khai báo Service Interface – Swagger, RAML, Thrift, Thrift IDL: Khi bạn có một nghiệp vụ có thể làm thành một service, bạn sẽ cần làm những bản service contract (đại loại một văn bản mà các nhà phát triển muốn những client khác gọi đến phải tuân theo những nguyên tắc trong đó để được trích xuất dữ liệu). Trong ứng dụng có kiến trúc một khối truyền thống, chúng ta hầu như không cần làm bởi vì các service bên trong có thể gọi nhau qua Framwork hoặc ngôn ngữ nền tảng. Trong SOA/web service, WSDL thường được dùng để làm service contract, nhưng WSDL không phải là giải pháp trong một hệ thống có sử dụng kiến trúc Microservices dùng REST để giao tiếp giữa các service.
  + Bởi vì xây dựng các ứng dụng có kiến trúc Microservices dựa trên kiến trúc REST nên chúng ta có thể sử dụng các kỹ thuật định nghĩa API trong REST để tạo một service contract. Do đó, có thể sử dụng Swagger và RAML để định nghĩa service contract.
  + Đối với các hệ thống không dựa trên HTTP/REST, chẳng hạn như Thrift, chúng ta có thể sử dụng Interface Definition Languages (IDL) như Thrift IDL.
    1. **Quản lý dữ liệu phân tán**

Trong kiến trúc một khối, ứng dụng lưu trữ dữ liệu trong các cơ sở dữ liệu đơn và tập trung để thực hiện các chức năng/khả năng khác nhau của ứng dụng



Trong kiến trúc Microservices, các chức năng được phân tán trên nhiều microservice và nếu chúng ta sử dụng cùng một cơ sở dữ liệu tập trung thì nó rất khó đảm bảo tính lỏng lẽo giữa các service bởi nếu database schema có sự thay đổi ở một service nào đó có thể làm ảnh hưởng hoặc có thể làm chết một vài service khác. Do đó, mỗi service cần có cơ sở dữ liệu riêng.



Dưới đây là các khía cạnh chính của việc thực hiện quản lý dữ liệu phân tán trong kiến trúc Microservices:

* Mỗi microservice có thể có một cơ sở dữ liệu riêng để duy trì dữ liệu cần thực hiện chức năng ngiệp vụ của nó.
* Một microservice cụ thể chỉ có thể truy cập vào cơ sở dữ liệu riêng của nó, chứ không phải cơ sở dữ liệu của các service khác.
* Trong một số tình huống, bạn có thể phải cập nhật một số cơ sở dữ liệu cho một giao dịch. Trong trường hợp như vậy, cơ sở dữ liệu của các service khác chỉ được cập nhật thông qua API của nó.

Việc quản lý dữ liệu phân tán sẽ cung cấp cho chúng ta các service tách rời hòan toàn và tự do lựa chọn các kỹ thuật quản lý dữ liệu khác nhau(SQL hoặc NoSQL,… các hệ thống quản trị cơ sở dữ liệu khác nhau cho mỗi service). Tuy nhiên, đối với các trường hợp sử dụng các giao dịch phức tạp liên quan đến nhiều service, các giao dịch phải được thực hiện bằng cách sử dụng API được cung cấp từ mỗi service là logic tại client hoặc tầng trung gian (Gateway).

* + 1. **Quản trị phân tán**

Kiến trúc Microservices rất thích hợp để quản trị phân tán. Quản trị trong IT được hiểu là các quy trình đảm bảo việc sử dụng các công nghệ hiệu quả và cho phép một tổ chức đạt được mục tiêu của mình. Trong SOA, quản trị SOA hướng dẫn phát triển các service có thể sử dụng lại, thiết lập cách thức các service được thiết kế, phát triển và cách các service đó thay đổi theo thời gian. Nó thiết lập các thỏa thuận giữa bên cung cấp service và người sử dụng service, nó cho biết người dùng biết những gì họ mong muốn và cho bên cung cấp biết những gì họ bắt buộc phải cung cấp. Quản trị SOA có hai loại quản trị được dùng phổ biến:

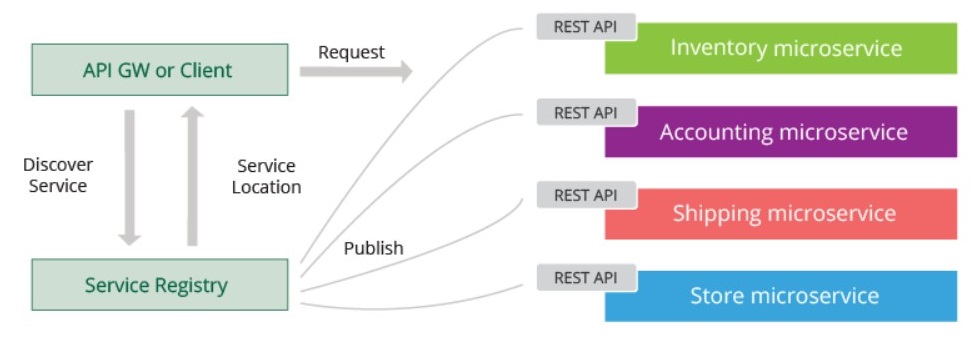
* Quản trị Design-time: xác định và kiểm soát cách tạo, thiết kế và phát triển service theo tiêu chuẩn đã cho trước.
* Quản trị Run-time: xác định khả năng thực thi các service theo tieu chuẩn đã cho trước trong suốt quá trình thi hành.

Vậy quản trị trong microservice có ý nghĩa là gì? Trong kiến trúc Microservices, microservice được xây dựng dưới các service độc lập và tách rời hoàn toàn với các nền tảng công nghệ khác nhau. Do đó không cần phải xác định một tiêu chuẩn chung cho thiết kế và phát triển service. Quản trị phân tán hệ thống có kiến trúc Microserivces có thể được tóm tắt như sau:

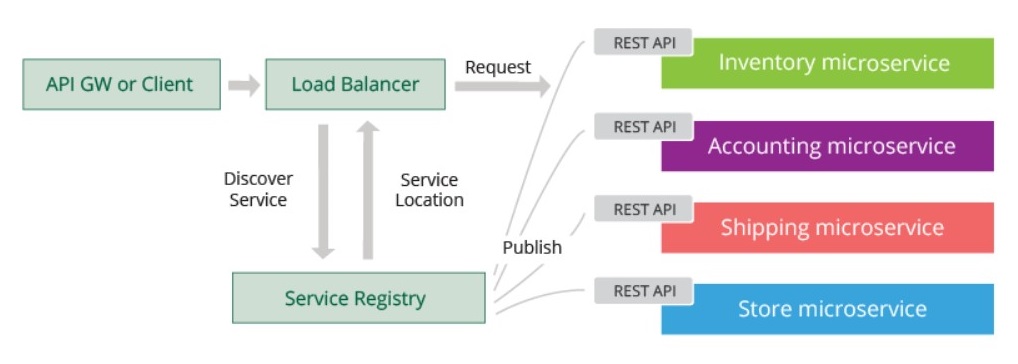
* Các service có thể tự đưa ra những nguyên tắc về thiết kế và cách thực hiện của riêng nó.
* Kiến trúc Microservices khuyến khích việc chia service chung hoặc có thể tái sử dụng.
* Các khía cạnh quản trị run-time như SALs, điều tiết, giám sát, các yêu cầu chung về bảo mật và các service discovery sẽ không được triển khai trên mỗi service. Thay vào đó, chúng nên được ở trong các mô-đun chuyên dụng (thường ở tầng API-gateway).
  + 1. **Service Registry và Service Discovery**

Trong kiến trúc Microservices, số lượng các service mà chúng ta cần để làm việc là khá nhiều. Chúng thay đổi vị trí linh hoạt do tính chất cần phát triển nhanh của microservice. Do đó, cần tìm vị trí của microservice trong suốt thời gian thực thi. Giải pháp cho vấn đề này là Service Registry.

* Service Registry: là nơi chứa các metadata của các microservice instance (bao gồm vị trí location, host post,…) Các microservices instance được đăng ký với service registry khi khởi động và sẽ hủy đăng ký khi bị tắt. Các thành phần khác cần thông tin của microservice nào đó sẽ tìm thông qua service registry.
* Service Discovery: Để tìm các microservice đang hoạt động và vị trí của nó, chúng ta cần đến cơ chế Service Discovery. Có 2 loại cơ chế Service Discovery là client-side discovery và server-side discovery:
  + Client- side discovery: Theo cách tiếp cận này, client hoặc API-gateway sẽ có được vị trí của một service instance bằng cách truy vấn một service registry.



* + Server- side discovery: Với phương pháp này, client/API gateway sẽ gửi một request đến một mô-đun (ví dụ như load blancer) chạy trên một vị trí đã biết. Mô-đun đó sẽ gọi đến service registry và xác định vị trí mà request cần đến.



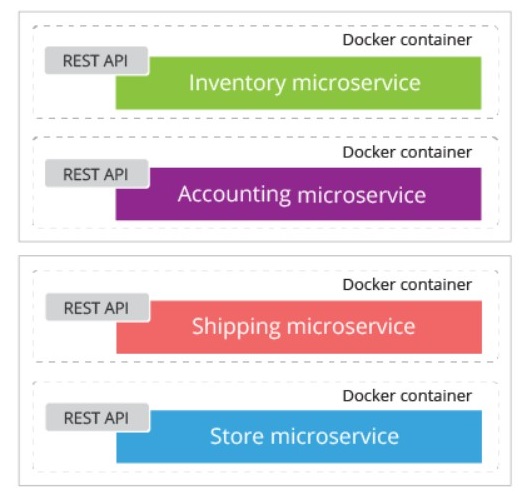
Các microservice có thể tận dụng các giải pháp triển khai như Kubernetes cho service-side discovery.

* + 1. **Deployment**

Khi nói đến kiến trúc Microservices, việc triển khai các service đóng vai trò quan trọng và có yêu cầu chính như sau:

* Khả năng triển khai/gỡ cài đặt độc lập đối với các service khác.
* Phải có khả năng tăng quy mô cho từng service.
* Triển khai nhanh chóng.
* Lỗi trong một service sẽ không được làm ảnh hưởng đến bất kỳ service nào khác. Docker(một công cụ mã nguồn mở cho phép các lập trình viên và quản trị viên hệ thống triển khai các container trong môi trường Linux, Window, Mac,…) cung cấp một công cụ tuyệt vời để triển khai các service theo các yêu cầu trên. Các ý chính bao gồm:
  + Package của service dưới dạng một Docker image.
  + Triển khai mội service instance bằng một container.
  + Việc tăng quy mô được thực hiện trên việc thay đổi số lượng container instance.
  + Xây dựng, triển khai và khởi động một service sẽ nhanh hơn nhiều khi sử dụng Docker container (so với các máy ảo Virtual Machince)

Kubernetes là một công cụ có khả năng mở rộng các tính năng của Docker bằng các cho phép quản lý một cụm Linux/Window container dưới dạng một hệ thống duy nhất, quản lý và chạy các Docker container trên nhiều máy chủ, cung cấp các vị trí cùng cấp của các container, service discovery và các kiểm soát khác. Như chúng ta có thể thấy, hầu hết các tính năng này đều rất cần thiết trong hệ thống microservice. Do đó, sử dụng Kubernetes(dựa trên Docker) để triển khai hệ thống Microservices đã trở thành một giải pháp mạnh mẽ, đặc biệt là đối với các hệ thống có quy mô lớn.



* + 1. **Bảo mật**

Bảo mật là một yêu cầu quan trọng và phổ biến khi chúng ta sử dụng kiến trúc Microservices trong thực tế. Trước khi tìm hiểu bảo mật trong Microservices, hãy xem xét cách mà chúng ta thường triển khai bảo mật ở các ứng dụng có cấu trúc nguyên khối:

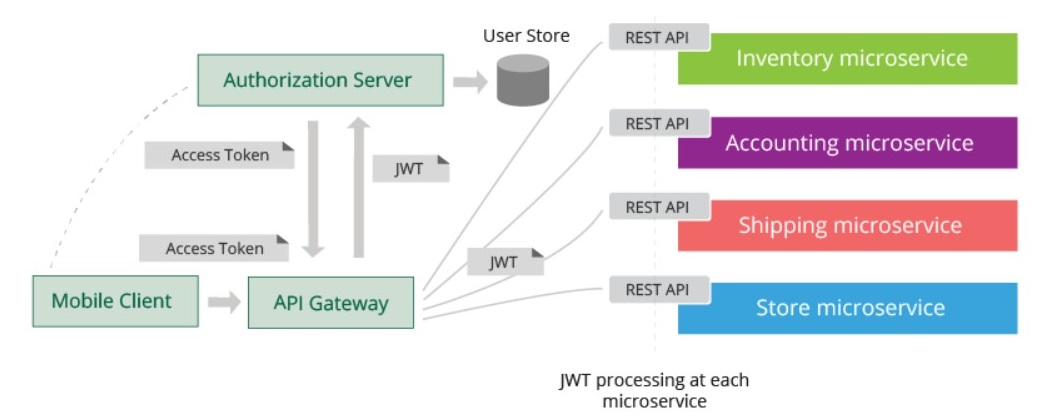
* Trong một ứng dụng nguyên khối điển hình, bảo mật là tìm ra “ai là người gọi”, “người gọi có thể làm gì” và “chúng ta truyền thông tin đó như thế nào”.
* Điều này được thực hiện tại một mô-đun bảo mật chung, nằm ở đầu xử lý yêu cầu và mô-đun đó chứa yêu cầu cần thiết với việc sử dụng vùng lưu trữ người dùng bên dưới.

Vậy thì chúng ta có thể trực tiếp sử dụng lại các nguyên tắc này với kiến trúc Microservices không? Câu trả lời là có, nhưng đòi hỏi một mô-đun bảo mật được triển khai ở từng service có thể giao tiếp với vùng lưu trữ thông tin người dùng và truy xuất thông tin khi cần thiết. Đó là cách giả quyết rất tệ trong bảo mật microservice.

Thay vào đó, có thể tận dụng các tiêu chuẩn bảo mật API đang được phổ biến như OAuth2.0, OpenID Connect để tìm giải pháp tốt hơn cho vấn đề bảo mật. Trước khi đi sâu vào vấn đề đó, trước tiên chúng ta cần tóm tắt mục đích của từng tiêu chuẩn và cách chúng ta có thể sử dụng chúng.

* OAuth2.0: Là một giao thức ủy quyền truy cập. Client xác thực với một server ủy quyền và nhận được mà token, được gọi là “Access token”. Một access token sẽ không có thông tin nào về user/client. Nó chỉ có một tham chiếu đến thông tin người dùng chỉ có thể truy xuất bởi server ủy quyền. Do đó được gọi là “by-reference token” và nó đủ an toàn ngay khi trong trường hợp mạng internet được công khai.
* OpenID Connect: Hoạt động tương tự OAuth, nhưng ngoài Access Token, server ủy quyền sẽ phát hành mã token ID có chứa thông tin về người dùng. Điều này thường được triển khai bằng JWT (JSON Web Token) và nó được đăng ký bởi server ủy quyền. Điều này đảo bảo sự tin tưởng giữa server ủy quyền và client. Do đó, JWT còn được gọi là “by-value token” vì nó chứa thông tin người dùng và rõ ràng không an toàn khi sử dụng bên ngoài mạng nội bộ.

Bây giờ, hãy xem cách chúng ta có thể sử dụng các tiêu chuẩn này để bảo mật các service trong ví dụ:



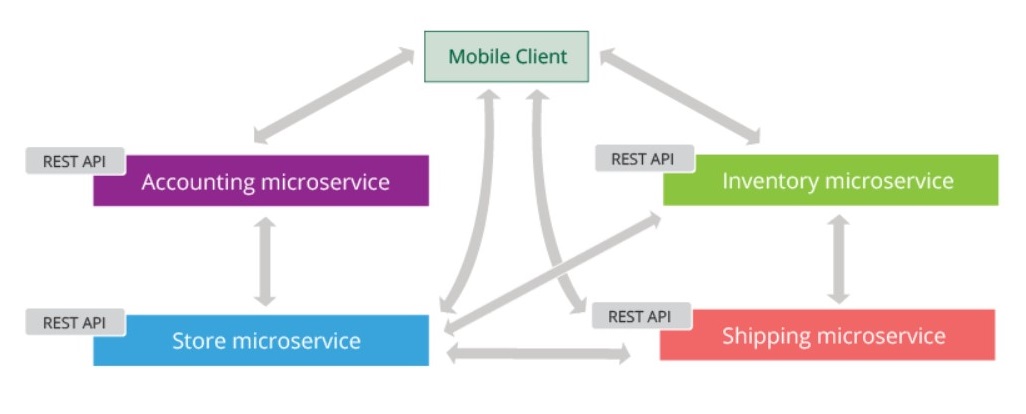
Như đã mô tả trong hình trên, các bước chính cần có liên quan đến việc triển khai bảo mật trong hệ thống Microservices là:

* Để lại việc xác thực cho OAuth 2.0 và OpenID Connect (server ủy quyền), nhờ vậy microservice sẽ cung cấp quyền truy cập thành công khi ai đó có quyền sử dụng dữ liệu.
* Sử API-gateway trong đó có một entry-point duy nhất cho tất cả các yêu cầu của máy khách.
* Client nối với server ủy quyền và nhận access token. Sau đó sẽ gửi các access token kèm theo trong mỗi lần request API – gateway.
* Token sẽ được giải mã ở gateway-API-gateway trích xuất access token và gửi nó đến server ủy quyền để truy xuất JWT.
* Cổng gateway sẽ dùng JWT kèm theo mỗi lần request đến tầng microservice.
* JWT chứa các thông tin cần thiết để lưu trữ các phiên người dùng,… Nếu mỗi service đều có thể hiểu một JSON token thì chúng ta có thể phân phối cơ chế nhận dạng của mình cho phép vận chuyển danh tính người dùng trên toàn hệ thống.
* Ở mỗi lớp microservice, chúng ta có thể có một mô-đun nhỏ để xử lý JWT.
  + 1. **Inter-Service/Process Communication**

Trong kiến trúc Microservices, các ứng dụng được xây dựng như một tập hợp các service độc lập. Do đó để biết khi nào cần đến nghiệp vụ nào thì chúng ta cần phải có các cấu trúc giao tiếp giữa các service hay quy trình khác nhau. Trừ khi microservice sử dụng các chuẩn giao thức (như HTTP) và các định dạng message (như JSON,…) thì yêu cầu thích hợp với một giao thức khác là yêu cầu tối thiểu khi nhắc đến việc giao tiếp giữa các microservice. Một cách tiếp cận khác trong giao tiếp microservice là sử dụng bus message hoặc cổng gateway với khả năng định tuyến tối thiểu và chỉ hoạt động như một “dumb pipe” không triển khai bất kỳ nghiệp vụ nào trên gateway. Dựa trên những phương pháp này, chúng ta có một số kiểu giao tiếp như sao:

* + - 1. **Point – to – Point Style – Gọi service trực tiếp**

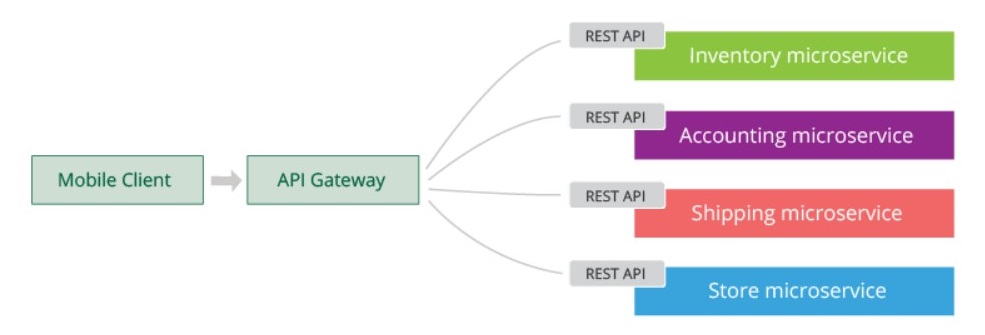
Trong kiểu point-to-point, toàn bộ logic định tuyến message nằm trên mỗi điểm cuối và các service có thể giao tiếp trực tiếp. Mỗi microservice sẽ đưa ra một REST API và một microservice cụ thể hoặc một client bên ngoài để có thể gọi một microservice khác thông qua REST API của nó.

****

Rõ ràng mô hình này hoạt động cho các ứng dụng microservice tương đối đơn giản, nhưng khi số lượng service tăng lên, điều này sẽ trở nên phức tạp vô cùng. Sau tất cả, đó chính là lý do cho việc sử dụng ESB trong triển khai SOA truyền thống, đó là loại bỏ các liên kết tích hợp point-to-point lộn xộn. Hạn chế của kiểu giao tiếp này được liệt kê như sau:

* Các yêu cầu phi chức năng, chẳng hạn như xác thực người dùng cuối, điều chỉnh, giám sát,… cần phải thực hiện ở mỗi cấp độ microservice.
* Do việc sao chép các common function, mỗi microservice có thể trở nên phức tạp.
* Không có trung tâm lưu trữ các yêu cầu phi chức năng: giám sát, truy tìm hoặc bảo mật cần được áp dụng ở mỗi cấp độ service.
* Thông thường kiểu giao tiếp trực tiếp được coi là “microservice anti-pattern” khi dùng trong các hệ thống có kiến trúc Microservices quy mô lớn.
  + - 1. **API-Gateway Style**

Ý tưởng chính đằng sau kiển API-Gateway là sử dụng một message gateway nhẹ làm điểm vào chính cho tất cả client thực hiện các yêu cầu phi chức năng phổ biến ở cấp độ gateway. Nói chung, một API-Gateway cho phép chúng ta sử dụng API được quản lý qua REST/HTTP. Do đó, chúng ta có thể đưa ra các chức năng kinh doanh của mình được triển khai dưới dạng microservice thông qua API-Gateway dưới dạng một API được quản lý. Trên thực tế, đây là sự kết hợp giữa quản lý hệ thống Microservices và API.



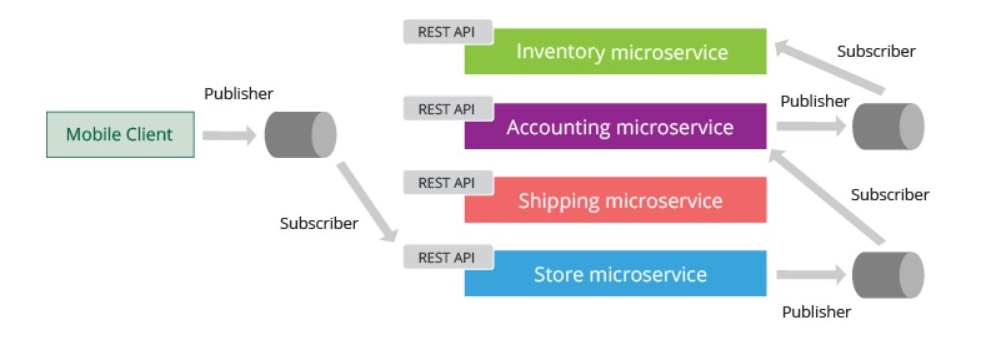
Trong kịch bản kinh doanh bán lẻ ban đầu, như được mô tả như hình trên, tất cả các microservice được hiển thị thông qua một API-Gateway và đó là điểm vào duy nhất cho tất cả các client. Kiểu API-Gateway sẽ đem lại các lợi ích như sau:

* Có khả năng cung cấp các khái niệm trừu tượng cần thiết ở cấp độ gateway cho các microservice
* Định tuyến/chuyển đổi message nhẹ ở cấp độ gateway – bạn có thể thực hiện lọc, định tuyến và chuyển đổi thông báo cơ bản gateway layer.
* Nơi tập trung để áp dụng các yêu cầu phi chức năng: bảo mật, giám sát và điều tiết – thay vì thực hiện các tính năng chung này ở mỗi lớp microservice.
* Với việc sử dụng mẫu API-gateway, microservice sẽ trở nên nhẹ hơn nữa vì tất cả các yêu cầu phi chức năng được thực hiện ở cấp độ gateway.

Kiểu API-Gateway cũng có thể là mẫu được sử dụng rộng rãi nhất trong hầu hết các triển khai hệ thống Microservices.

* + - 1. **Message Broker Style**

Trong các tình huống message không đồng bộ, chẳng hạn như các request một chiều sử dụng hàng đợi. Điều đó có nghĩa là một microservice có thể là nguồn cung cấp message và nó có thể gửi các message không đồng bộ đến một hàng đợi hoặc topic. Kiểu giao tiếp này sẽ tách biệt nguồn cung cấp message và nơi cần sử dụng message. Nó sử dụng một khu vực trung gian để lưu trữ các message cho đến khi những service cần đến có thể xử lý những message đó. Như vậy, các microservice gửi message sẽ không hề biết những microservice nào sử dụng message.

****

Giao tiếp giữa các người dùng/nhà phát triển được thông qua một message broker dựa trên các tiêu chuẩn message không đồng bộ, chẳng hạn như AMQP, MQTT, …

* + 1. **Transactions**

Các giao tiếp được hỗ trợ như thế nào trong microservice? Trên thực tế thì đây là một nhiệm vụ phức tạp. Kiến trúc microservice khuyến khích các service không có giao tiếp. Lý do là một service nên hoàn toàn khép kín và chỉ có một nhiệm vụ duy nhất. Do đó, trong hầu hết các trường hợp, giao tiếp chỉ được áp dụng ở phạm vi các microservice (tức là không phải trên nhiều microservice).

Tuy nhiên, nếu có một yêu cầu bắt buộc là phải có các giao tiếp trên nhiều service, thì các kịch bản như vậy có thể thực hiện bằng việc bổ sung các “hoạt động bù” ở cấp độ microservice. Có nghĩa là một microservice cụ thể chỉ có một nhiệm vụ và nếu một microservice cụ thể không thực hiện được nhiệm vụ, chúng ta có thể coi đó là một thất bại của toàn bộ ứng dụng. Sau đó, tất cả các hoạt động khác phải được hoàn tác bằng cách hoạt động bù tương ứng các microservice đó.

* + 1. **Thiết kế khi gặp lỗi**

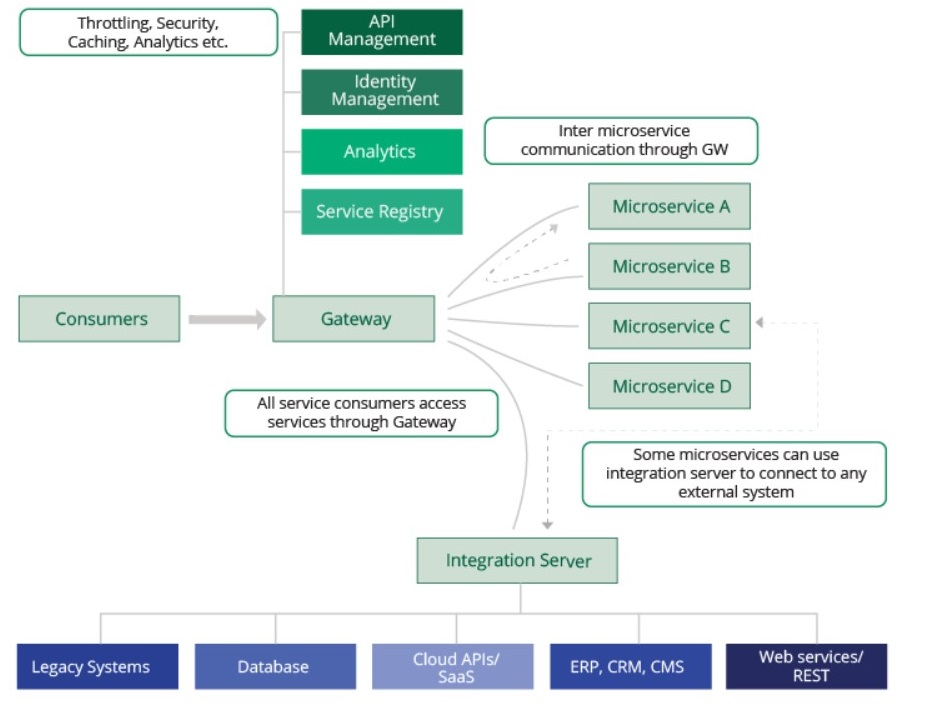
Kiến trúc Microservices là một bộ service phân tán và so với thiết kế nguyên khối, nó làm tăng khả năng gặp sự cố ở mỗi service. Một microservice cụ thể có thể thất bại do sự cố mạng, không có tài nguyên cơ bản,… Một microservice không có sẵn hoặc không phản hồi sẽ không được phép làm chết cả ứng dụng. Do đó, bản thân mỗi microservice nên có khả năng chịu lỗi, có thể phục hổi và nhờ đó client sẽ đảm đương nhiệm vụ xử lý lỗi. Hơn nữa, vì các service có thể thất bại bất cứ lúc nào, điều quan trọng là có thể phát hiện (giám sát theo thời gian thực) các lỗi một cách nhanh chóng và nếu có thể, các service này có thể tự động khôi phục.

Dưới đây là một số phương pháp xử lý lỗi trong hệ thống Microservices:

* Circuit Breaker: Khi muốn gọi đến một service, chúng ta sẽ muốn cài đặt một thành phần giám sát lỗi với mỗi lần goi, và khi số lỗi đạt ngưỡng nhất định thì thành phần đó sẽ dừng lại tất cả các request của service. Sau một số lượng request nhất định ở trạng thái mở hãy thay đổi mạch trở lại trạng thái đóng. Thiết kế này khá hữu ích để tránh tiêu thụ tài nguyên không cần thiết, yêu cầu phản hồi chậm trễ do timeout và cũng cung cấp cho chúng ta cách để giám sát hệ thống(dựa vào trạng thái đóng hay mở mạch hoạt động).
* Bulkhead: Mẫu thiết kế Bulkhead là cách thiết kế để cách ly các thành phần khác nhau trong ứng dụng để khi một service bị lỗi sẽ không ảnh hưởng đến bất kì service nào khác.
* Timeout: Mẫu thiết kế timeout là một cơ chế cho phép chúng ta ngừng chờ phản hồi từ microservice khi chúng ta nghĩ rằng nó sẽ không thực hiện được. Tại đây, chúng ta có thể chỉ định khoảng thời gian mà chúng ta muốn chờ.

Ngoài ra, cổng gateway có thể được sử dụng làm trung tâm lưu trữ mà tại đó chúng ta có thể có được trạng thái và giám sát từng service, vì mỗi service đều được gọi thông qua cổng gateway.

* + 1. **Microservices trong kiến trúc doanh nghiệp hiện đại**

****

Hình trên minh họa một kiến trúc CNTT doanh nghiệp cấp cao. Ở đây, chúng ta sử dụng một kiến trúc lai bao gồm cả microservice và các hệ thống hiện có. Có những quyết định thiết kế chính mà chúng ta cần phải đưa ra khi giới thiệu kiến trúc Microservice cho tổ chức của mình:

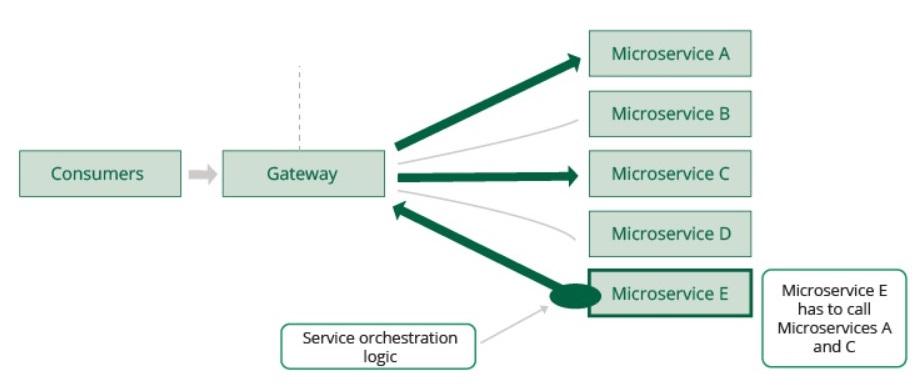
* Sử dụng kiến trúc Microservices để làm giải pháp bất cứ khi nào cần và cố gắng đạt được toàn bộ sức mạnh mà kiến trúc Microservices mang lại.
* Tích hợp doanh nghiệp vẫn là bắt buộc: vì chúng ta sẽ có một cách tiếp cận hỗn hợp, nên vì thế vẫn sẽ cần tích hợp tất cả hệ thống và dịch vụ nội bộ của mình với việc sử dụng một phần mềm tích hợp, chẳng hạn ESB.
* Chúng ta không thể vứt bỏ hầu hết các hệ thống hiện có, bởi các microservice mới có thể cần phải gọi đến các hệ thống nguyên khối để tạo điều kiện cho các yêu cầu kinh doanh khác nhau.
* Các “ESB” mới: Mặc dù phần mềm tích hợp, chẳng hạn như ESB, vẫn có thể cần thiết cho kiến trúc doanh nghiệp hiện đại, thì một tổ chức vẫn nên tìm kiếm phần mềm tích hợp nhẹ, hiệu suất cao và có thể mở rộng thay vì các framework tích hợp hạng nặng.
* Quản lý API: microservice có thể được đưa ra thông qua cổng gateway và tất cả các kỹ thuật quản lý API có thể được áp dụng ở lớp đó. Tất cả các yêu cầu khác như bảo mật, điều chỉnh, lưu trữ và giám sát phải được thực hiện ở cổng gateway. Hơn nữa, các service không dựa trên kiến trúc Microservices (SOA truyền thống) cũng có thể được đưa ra thông qua cổng API gateway.
  + 1. **Tích hợp Microservice**

Các câu hỏi thường gặp nhất trong kiến trúc Microservices là “microservice có thể nói chuyện với nhau không?”, “làm thế nào để xây dựng các microservice mới bằng cách tận dụng một microservice hiện có?” hoặc “làm thế nào để chúng ta soạn thảo/tích hợp microservice và hình thành các service?”.

Trên thực tế, kiến trúc Microservices thúc đẩy xây dựng một microservice với phạm vi kinh doanh hạn chế và tập trung. Do đó, khi nói đến việc xây dựng các giải pháp CNTT trên kiến trúc Microservices, không thể tránh khỏi việc chúng ta phải sử dụng các microservice hiện có. Sự tương tác giữa các microservice có thể được thực hiện theo kiểu point-to-point thông thường. Tuy nhiên, cách tiếp cận đó trở nên khá dễ phá vỡ (quá nhiều tương tác, khó quản lý, duy trì, mở rộng và khắc phục sự cố) khi nói đến các giải pháp microservice. Do đó, chúng ta cần tuân thủ các nguyên tắc tốt nhất về tích hợp microservice để loại bỏ các nhược điểm của tương tác kiểu point-to-point đó là:

* Sử dụng một cổng gateway để đưa ra các microservice: Sử dụng một cổng gateway ở trước tất cả các microservice và tất cả khách hàng chỉ sử dụng các microservice thông qua cổng gateway.
* Không có gọi trực tiếp giữa các microservice: microservice không thể gọi trực tiếp các microservice, tất cả các cuộc gọi phải đi qua cổng gateway.
* Micro-integration thông qua một máy chủ tích hợp.
  + - 1. **Phối hợp tại Microservice Layer**

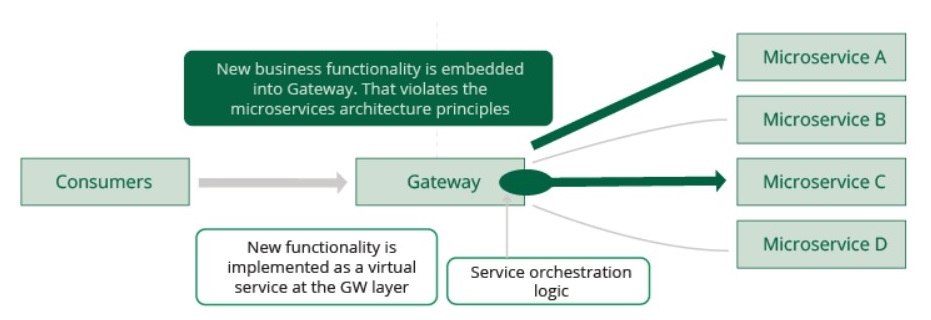
Khi chúng ta cần phải gọi nhiều microservices để hỗ trợ một yêu cầu kinh doanh nhất định, chúng ta có thể xây dựng một microservice khác (một lần nữa giải quyết vấn phạm vi yêu cầu kinh doanh hạn chế) sẽ điều phối các request service đến các microservice cần thiết và tổng hợp phản hồi cuối cùng và gửi lại cho khách hàng ban đầu.



Ví dụ, ở hình trên mô tả một kịch bản trong đó chúng ta có một vài microservices là A,B,C và D. Bây giờ chúng ta muốn đưa ra một chức năng mới đòi hỏi phải gọi microservice A và C một cách tuần tự và cung cấp một phản hồi tổng hợp. Để làm được điều này, chúng ta có thể xây dựng một microservice mới (microservice E) và có thể phối hợp một cách hợp lý có chứa dịch vụ A và C được nhúng vào microservice E. Tất cả các yêu cầu của microservice được thực hiện thông qua cổng gateway. Nếu microservice E phải được mở rộng một cách độc lập, điều đó có thể được thực hiện bằng cách mở rộng quy mô microservice E, A và C theo yêu cầu.

* + - 1. **Phối hợp tại Gateway Layer**

Cách tiếp cận khả thi khác là thực hiện cùng một kịch bản phối hợp bằng cách đưa các phối hợp hợp lý đến cấp độ cổng gateway. Trong trường hợp này, chúng ta không phải đưa ra một microservice mới, nhưng sẽ cần một lớp service ảo được lưu trữ trong cổng gateway đảm nhiệm việc điều phối.



Ví dụ, như hình ở trên, service gọi đến microservice A và C có thể được triển khai bên trong cổng gateway (hầu hết các cài đặt microservice gateway đều hỗ trợ tính năng này).

Khi việc mở rộng quy mô kinh doanh mới được nói đến, chúng ta cần phải mở rộng quy mô của cổng gateway và microservice A và C. Với điều này, cổng gateway sẽ trở nên hơi nguyên khối vì nó cũng chịu trách nhiệm định tuyến cho tất cả yêu cầu khác.

* + - 1. **Micro-Integration**

Khi chúng ta phải xây dựng các giải pháp tích hợp, thường thì việc sử dụng một máy chủ tập trung có chứa các tích hợp lý được xem xét hàng đầu. Khái niệm tích hợp vi mô (micro-integration) là một framework tích hợp nhẹ có thể được sử dụng để xây dựng các giải pháp tích hợp; nó có thể tích hợp microservice và các service/hệ thống khác. Và chúng ta có thể mở rộng chúng tại thời điểm runtime chứ không như phương pháp xây dựng máy chủ tích hợp thông thường, nơi mà chúng ta chỉ có thể mở rộng quy mô theo những kịch bản được lựa chọn sẵn.

* + - 1. **Choreography Style**

Một cách tiếp cận khả thi khác là xây dựng các tương tác giũa các microservice bằng cách sử dụng kiểu message không đồng bộ, chẳng hạn như MQTT hoặc Kafka. Trong trường hợp này, không có thành phần trung tâm nào đảm nhiệm tương tác giữa các service. Các service khác nhau có thể thực hiện message bằng các giao thức messaging protocol.

* + 1. **WSO2 Microservices Framework cho Java (WSO2 MSF4J)**

Có khá nhiều thư viện và framework để xây dựng hệ thống có kiến trúc Microservices, nhưng hầu hết trong số đó không thực sự tuân thủ các nguyên tắc cốt lõi của microservice.

WSO2 cung cấp một microservice framework nhẹ, nhanh và thân thiện với container.

WSO2 Microservices Framwork for Java (WSO2 MSF4J) cung cấp tùy chọn tốt nhất để tạo microservice trong Java với ý định triển khai dựa trong container. Các dịch vụ được phát triển bằng WSO2 MSF4J có thể khởi động chỉ trong vài mili giây trong Docker và có thể dễ dang tạo Docker image.

* + 1. **Tổng kết**

Khi muốn kết hợp kiến trúc Microservices trong môi trường CNTT doanh nghiệp hiện đại ngày nay, các khía cạnh chính chúng ta cần quan tâm tổng kết lại như sau:

* Microservice không phải là thuốc chữa bách bệnh – nó không giải quyết tất cả các như cầu CNTT của doanh nghiệp, vì vậy chúng ta cần phải sử dụng nó với các kiến trúc hiện có khác.
* Dùng SOA hợp lý.
* Hầu hết tất cả các doanh nghiệp sẽ không thể chuyển đổi toàn bộ hệ thống CNTT của họ sang hệ thống có kiến trúc Microservices. Thay vào đó, họ sẽ sử dụng Microservices để giải quyết một số trường hợp khi họ có thể tận dụng sức mạnh của kiến trúc Microservices.
* Tích hợp doanh nghiệp sẽ không bao giờ biến mất – điều đó có nghĩa là chúng ta cần phải có phần mềm tích hợp, chẳng hạn ESB để phục vụ cho tất cả các nhu cầu tích hợp.
* Tất cả các yêu cầu kinh doanh phải được đưa ra dưới dạng API bằng cách tận dụng các kỹ thuật quản lý API.
* Tương tác giữa các microservice nên được thực hiện qua cổng gateway.
* Việc phối hợp service giữa các microservice có thể được yêu cầu cho một số yêu cầu kinh doanh và có thể được triển khai bên trong một microservice khác hoặc ở tại lớp gateway.
  1. **CHƯƠNG III – XÂY DỰNG HỆ THỐNG XÁC THỰC & PHÂN QUYỀN DỰA TRÊN KIẾN TRÚC MICROSERVICES**
  2. **Cơ sở lý thuyết**
     1. **Giới thiệu về server xác thực và phân quyền (Single Sign - On)**

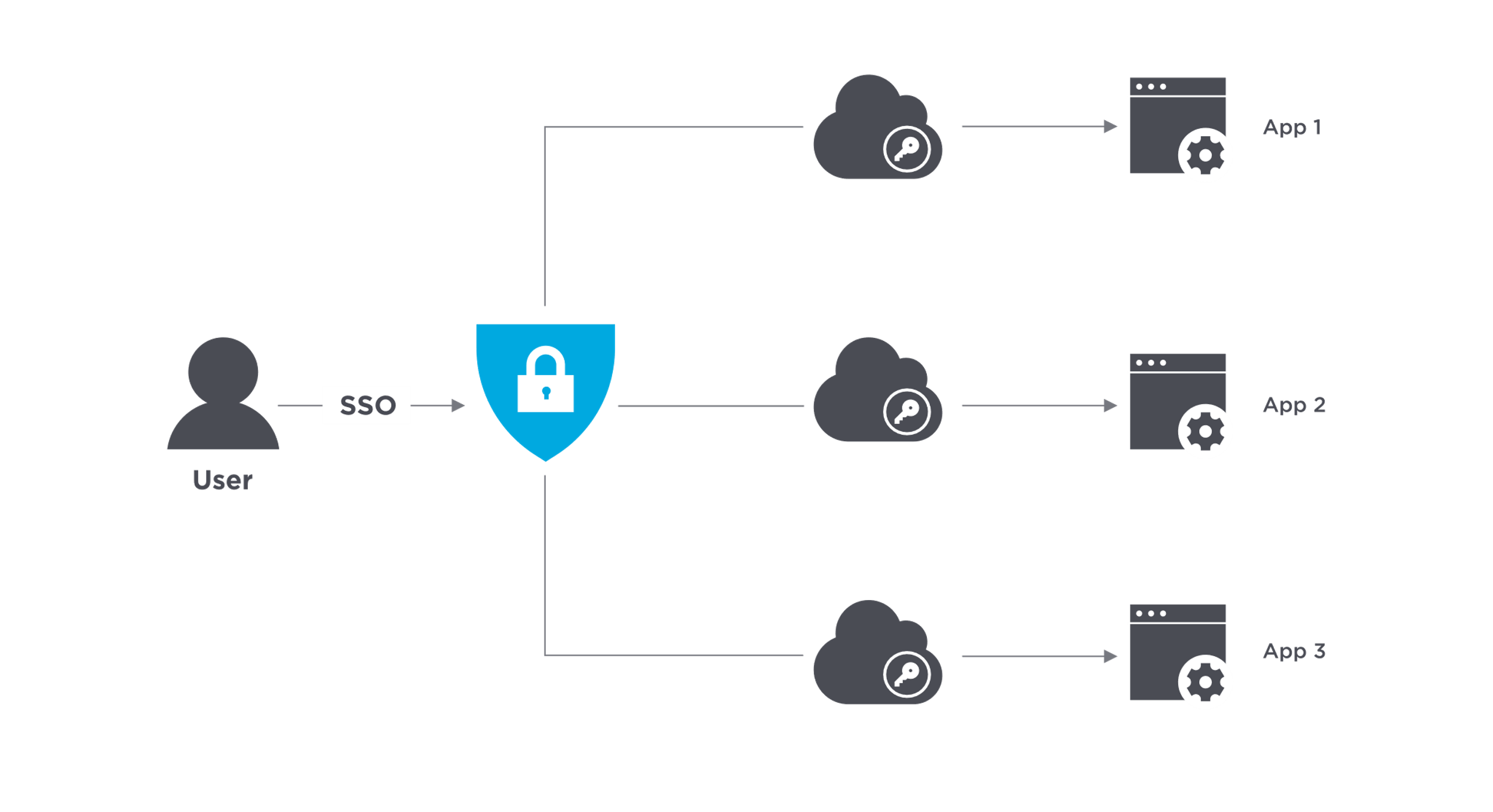
Single Sign-On giúp các công ty làm việc hiệu quả hơn và tiết kiệm cả thời gian và tiền bạc.

Về phía người dùng, Single Sign-On tiết kiệm thời gian và năng lượng vì người dùng không phải đăng nhập nhiều lần vào nhiều ứng dụng. Điều này cung cấp trải nghiệm người dùng hợp lý và khiến phần mềm ít gặp phải sự cố truy cập tài khoản do phải xử lý nhiều mật khẩu và tài khoản người dùng khác nhau.

Single Sign-On cũng là tuyệt vời đối với quản trị viên. Nó tiết kiệm thời gian của quản trị viên và nhân viên trợ giúp kỹ thuật bằng cách giảm số lượng yêu cầu được gửi đến bộ phận trợ giúp liên quan đến các vấn đề truy cập thất bại.

* + 1. **Lợi ích của Single Sign-On**

Single Sign-On (SSO) trong doanh nghiệp đề cập đến khả năng nhân viên đăng nhập một lần với một bộ thông tin đăng nhập để có quyền truy cập vào tất cả các ứng dụng, trang web và dữ liệu mà họ có quyền. Single Sign-On giải quyết các vấn đề chính cho doanh nghiệp bằng cách cung cấp:

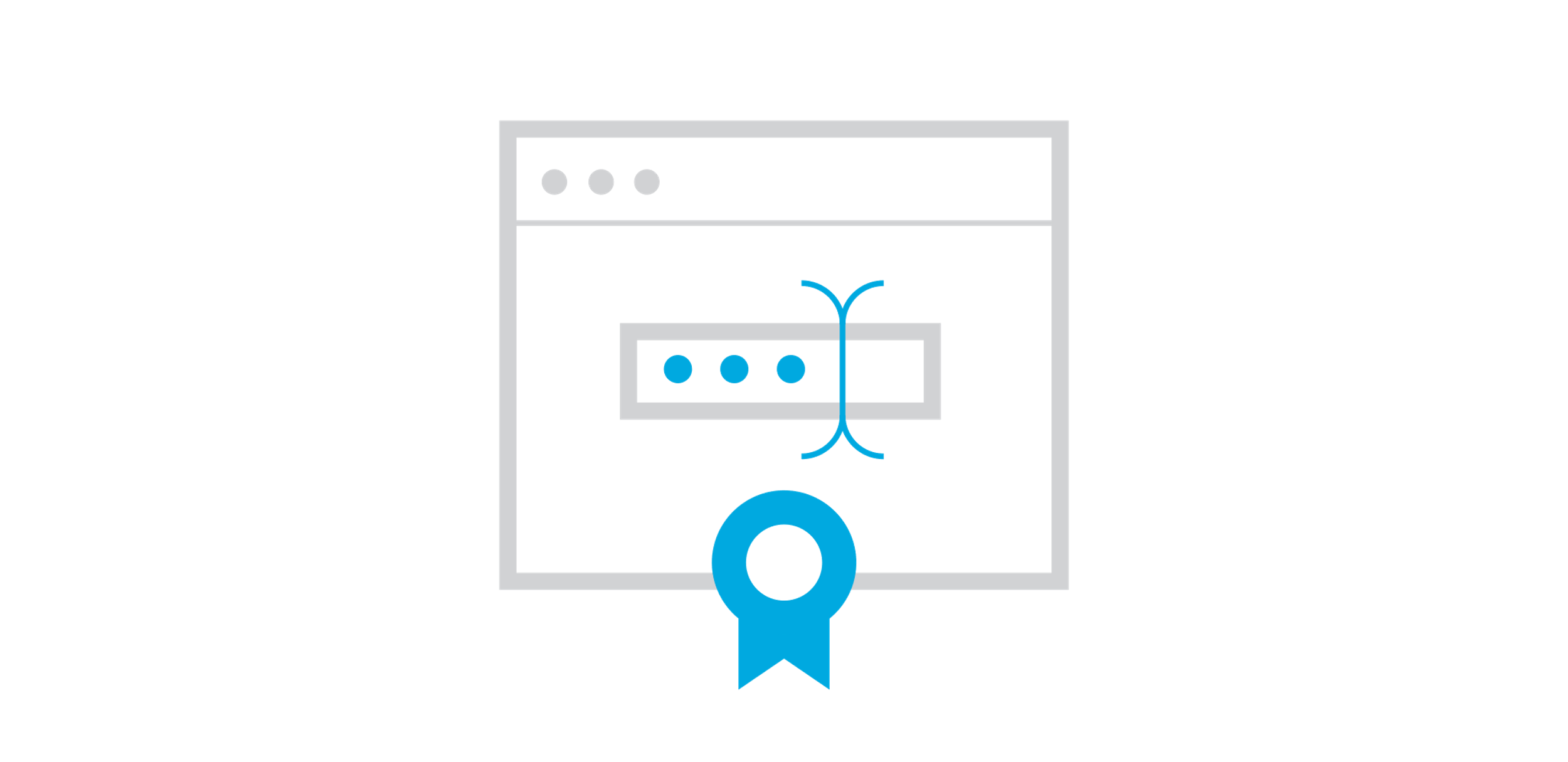


* + - 1. **Bảo mật**

Tên người dùng và mật khẩu là mục tiêu chính của tội phạm mạng. Mỗi khi người dùng đăng nhập vào ứng dụng mới, đó là cơ hội cho tin tặc. Single Sign-On giảm số lượng bề mặt tấn công vì người dùng chỉ đăng nhập một lần mỗi ngày và chỉ sử dụng một bộ thông tin đăng nhập.

Giảm đăng nhập vào một bộ thông tin đăng nhập sẽ cải thiện bảo mật doanh nghiệp. Khi người dùng phải sử dụng mật khẩu riêng cho từng ứng dụng mà họ thường không sử dụng. Trên thực tế, 59% người dùng sử dụng mật khẩu giống hoặc tương tự trên nhiều tài khoản. Do đó, nếu một hacker có quyền truy cập thông qua một trang web được bảo mật kém, họ có khả năng truy cập các hệ thống khác.

Khi Single Sign-On là một phần của giải pháp quản lý truy cập và nhận dạng (IAM), nó sử dụng một thư mục trung tâm kiểm soát quyền truy cập của người dùng vào tài nguyên ở mức chi tiết hơn.

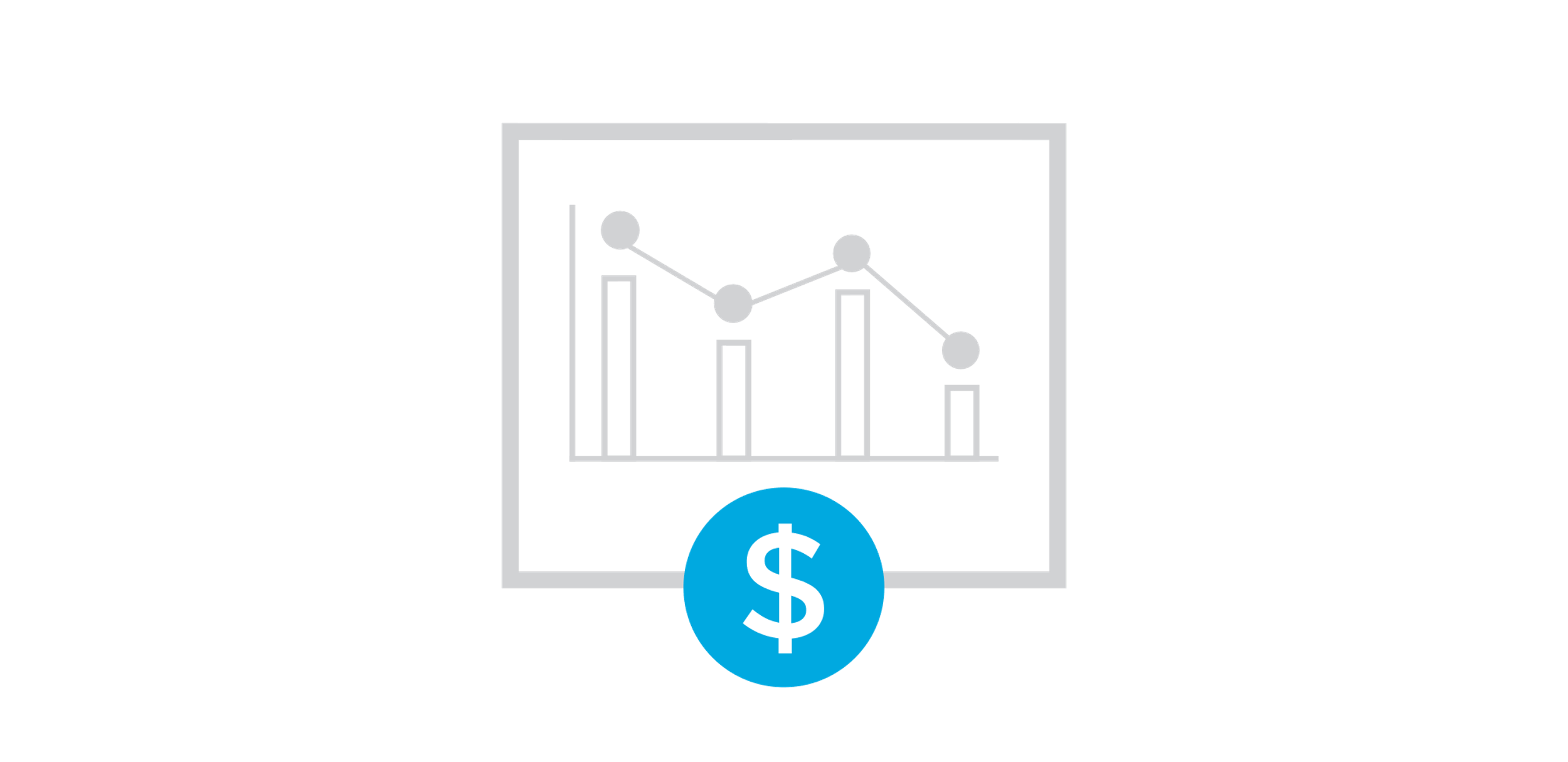


* + - 1. **Cải thiện khả năng sử dụng**

Single Sign-On giúp tiết kiệm thời gian, do đó cải thiện năng suất của người dùng.

* + - 1. **Chi phí**

Single Sign-On là một phần của hệ thống thông tin hợp nhất, tận dụng một thư mục trung tâm để cung cấp và từ chối người dùng, làm cho quá trình này nhanh hơn và rẻ hơn. Chính sách có thể được xác định dựa trên vai trò người dùng, vị trí và các đặt điểm người dùng khác.



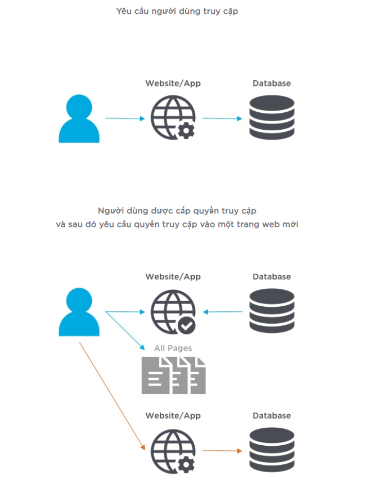
* + 1. **Nguyên tắc Single Sign-On hoạt động**

Như đã đề cập ở phần một, Single Sign-On (SSO) là thuộc tính của quản lý truy cập và nhận dạng (IAM) cho phép người dùng xác thực an toàn với nhiều ứng dụng và trang web bằng cách đăng nhập một lần chỉ với một bộ thông tin đăng nhập (tên người dùng và mật khẩu).

* + - 1. **Xác thực hoạt động khi không có Single Sign-On**

Nếu không có Single Sign-On, mỗi trang web sẽ duy trì cơ sở dữ liệu người dùng và thông tin đăng nhập của riêng họ. Đây là những gì sẽ xảy ra khi chúng ta cố gắng đăng nhập vào một ứng dụng hoặc trang web:

* Trang web trước tiên kiểm tra xem chúng ta đã được xác thực chưa. Nếu có, nó cung cấp cho chúng ta quyền truy cập vào trang web.
* Nếu không có, nó sẽ yêu cầu chúng ta đăng nhập và nó sẽ kiểm tra tên người dùng và mật khẩu dựa trên những thông tin trong cơ sở dữ liệu người dùng của nó.
* Sau khi đăng nhập, trang web sẽ chuyển dữ liệu xác thực khi chúng ta di chuyển qua trang khác và để xác minh rằng chúng ta được xác thực mỗi khi chúng ta đến trang mới.



Dữ liệu xác thực thường được chuyển dưới dạng cookie, session data hoặc token.

* + - 1. **Xác thực hoạt động khi có Single Sign-On**

Xác thực với Single Sign-On dựa trên mối quan hệ tin cậy giữa các tên miền (trang web). Với SSO, đây là điều sẽ xảy ra khi chúng ta cố gắng truy cập vào một ứng dụng hoặc trang web:

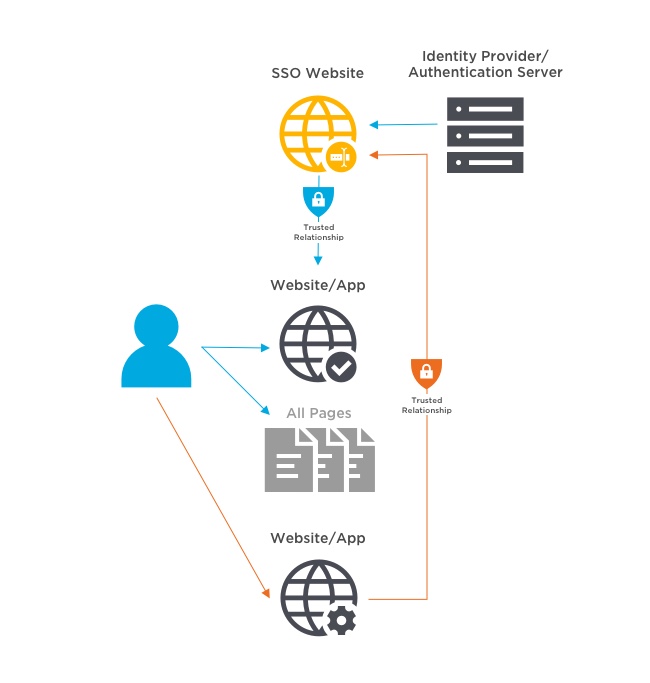
* Trước tiên, trang web sẽ kiểm tra xem bạn đã được xác thực bởi giải pháp Single Sign-On chưa, trong trường hợp đó, nó cho phép chúng ta truy cập vào trang web.
* Nếu không có, nó sẽ gửi chúng ta đến giải pháp Single Sign-On để đăng nhập.
* Chúng ta nhập tên người dùng/mật khẩu duy nhất mà bạn đã đăng ký hoặc được cấp trước đó.
* Giải pháp SSO yêu cầu xác thực từ nhà cung cấp nhận dạng hoặc hệ thống xác thực mà hệ thống đang sử dụng. Nó xác minh danh tính và thông báo cho giải pháp SSO.
* Giải pháp SSO chuyển dữ liệu xác thực đến trang web và đưa người dùng quay lại trang web đó.
* Sau khi đăng nhập, trang web sẽ chuyển dữ liệu xác thực với người dùng khi người dùng di chuyển qua trang mới để chứng minh rằng người dùng đã được xác thực.

****

Trong SSO, dữ liệu xác thực có dạng là tokens.

Trang web chuyển hướng người dùng đến trang SSO để đăng nhập. Người dùng đăng nhập bằng tên người dùng và mật khẩu.

Trang web SSO xác minh danh tính người dùng với nhà cung cấp nhận dạng.

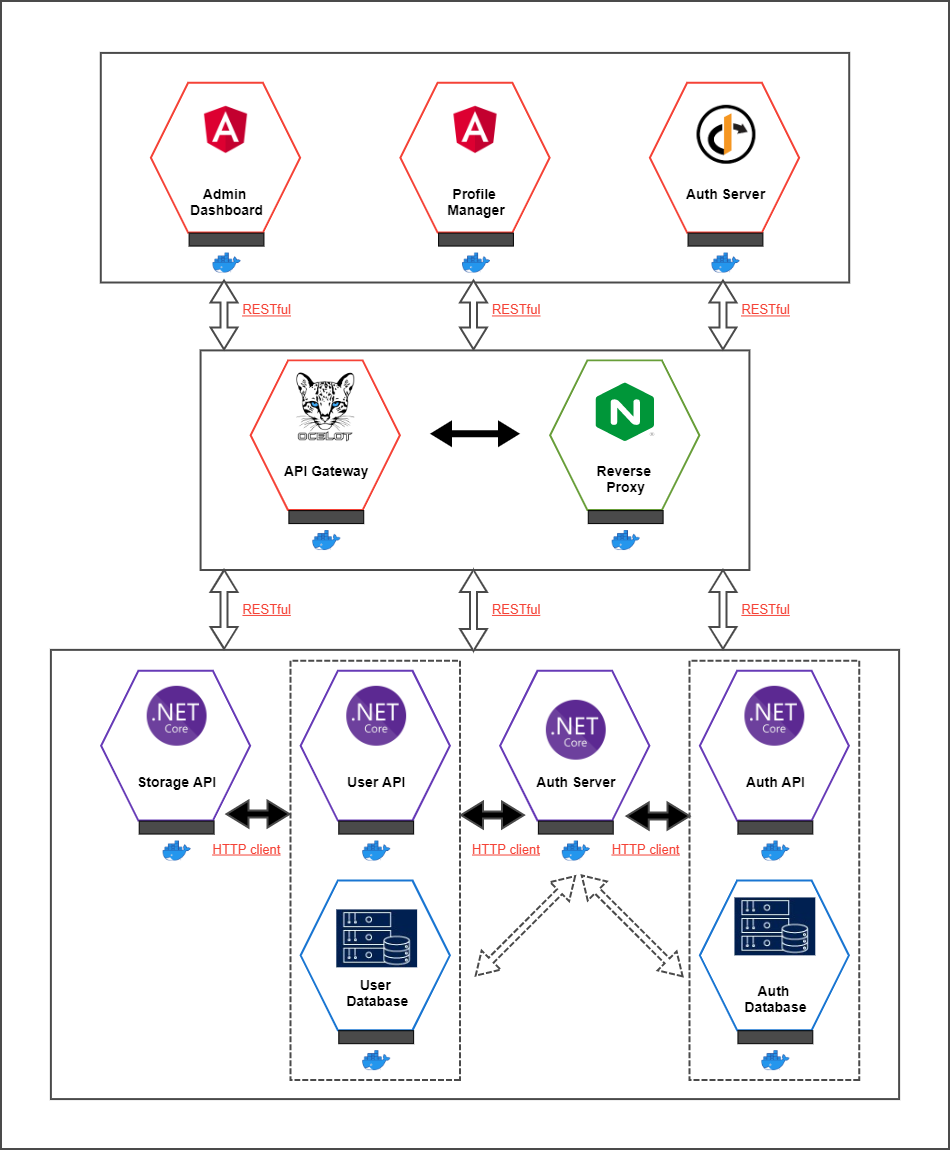


Khi người dùng cố gắng truy cập một trang web khác, trang web mới sẽ kiểm tra bằng giải pháp SSO. Vì người dùng đã xác thực, nó xác minh danh tính của người dùng cho trang web mà không yêu cầu đăng nhập bổ sung.

* 1. **Hệ thống xác thực và phân quyền (SSO) dựa trên kiến trúc Microservices**

Các công nghệ/Farmework có sử dụng trong hệ thống:

* .NET Core 3.1
* Angular 10.0.x
* Sql Server 2019
* Ocelot 16.0.x
* Nginx 1.19.x
* Swagger
* Identity Server 4
* ASP NET Identity
* Docker
* Node.js



* 1. **Tét**
  2. **Àgh**
  3. **Gffdhg**
  4. **agfhh**