Giới thiệu về Microservices (kiến trúc nhiều dịch vụ nhỏ)

11/10/2015 / Bởi **Trịnh Minh Cường** / trong **Java**

Microservices hiện được quan tâm trong giới phần mềm, công nghệ với nhiều bài viết, blog, thảo luận, truyền thông, hội thảo. Kỳ vọng về khả năng của Microservice đang lên đỉnh giống như một xu hướng thời trang đang lan rộng. Ngược lại, một số người cho rằng, microservices không có gì mới lạ, chẳng qua nó là SOA (kiến trúc hướng dịch vụ) được đánh bóng, đổi tên mà thôi.

Bài viết này được dịch từ bài viết "Introduction to Microservices" của tác giả Chris Richardson. Bài viết dài, nên nhiều chỗ tôi phải lược dịch. Một số chỗ có thể khó hiểu đối với người đọc VN, nên tôi thêm

chú thích. Một số quan điểm khá thiên kiến, trên thì khen Microservices, dưới thì liệt kê nhược điểm nhưng không giải thích rõ trường hợp nào thì là nhược điểm và đã có những giải pháp gì xử lý. Tuy nhiên phải công nhận đây là một bài viết tốt đáng đọc. Tiếp sau sẽ còn 6 bài trong chủ đề này.

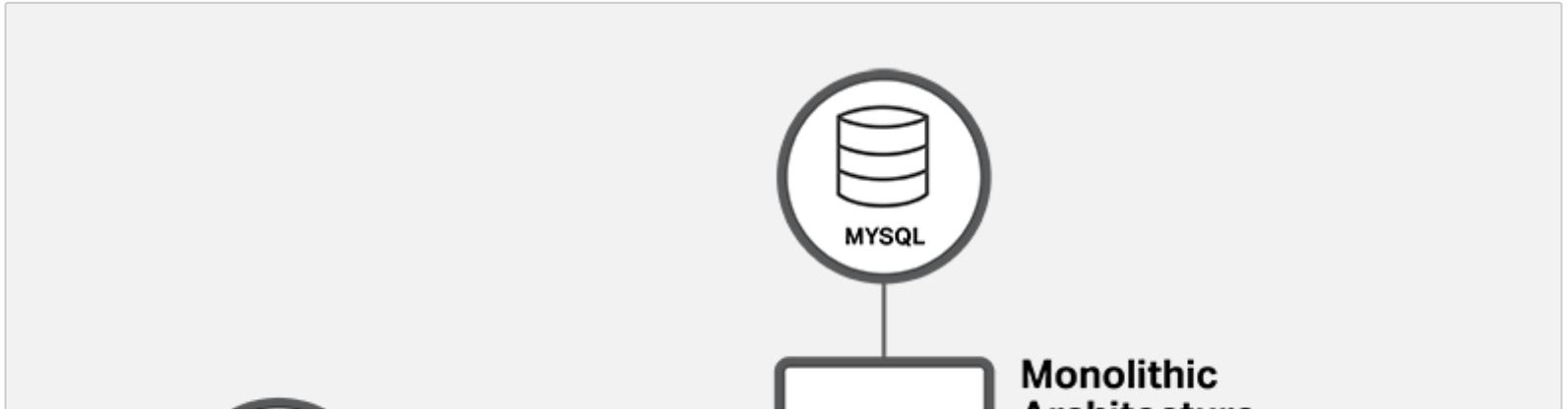
Mặc cho kỳ vọng cao, hay đánh giá bảo thủ, kiến trúc microservices vẫn đem lại lợi ích khi nó giúp phương pháp agile thực sự hiệu quả và xây dựng được giải pháp phần mềm doanh nghiệp rất phức tạp.

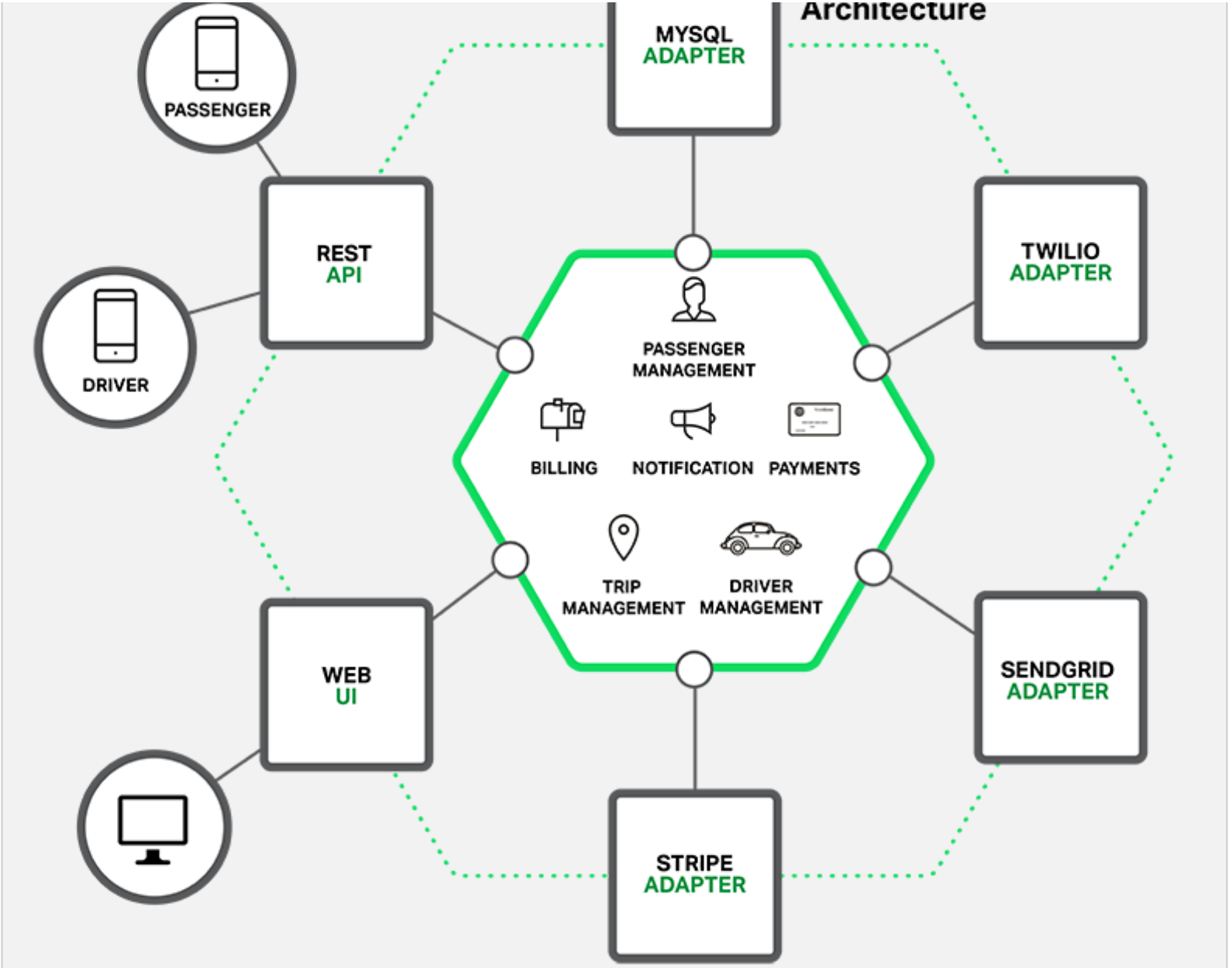
Bài viết này sẽ giải thích tại sao công ty, tổ chức của bạn nên cân nhắc áp dụng kiến trúc microservices

Xây dựng ứng dụng một khối (monolithic applications)

Tưởng tượng bạn phải xây dựng một dịch vụ gọi taxi qua di động cạnh tranh với Uber và Hailo. Sau một số buổi họp thu thập yêu cầu và phân tích thiết kế, bạn sẽ chọn công nghệ (technology stack) rồi tạo dự án đại loại như Rails, Spring Boot, Play, hay Maven. Dự án này sẽ có kiến trúc chia khối lục giác (hexagol architecture) hoặc ít hơn. Kiến trúc đa diện giúp ứng dụng chuyên biệt mô hình dữ liệu và đầu vào, đầu ra.

Trong lõi của ứng dụng là business logic được thể hiện bởi các khối dịch vụ, đối tượng cho từng vùng nghiệp vụ (domain objects) và các sự kiện (events: khách đặt xe, khách hủy xe, xe nhận khách...) Xung quanh lõi là bộ chuyển đổi (adapter) ví dụ như kết nối vào cơ sở dữ liệu, gửi nhận thông điệp (messaging), web service hoặc giao diện web front end.





Kiến trúc một khối có nhiều mặt, business logic ở trung tâm

Mặc dù có cấu trúc module hóa hợp lý, nhưng ứng dụng kiểu này sẽ đóng gói và cài đặt thành một khối (monolithic). Mã chạy cụ thể tùy thuộc vào ngôn ngữ lập trình hay thư viện framework. Ví dụ ứng dụng Java đóng trong file WAR, triển khai trên application server như Tomcat hay Jetty. Dùng framework khác, thì ứng dụng Java là một file tự đóng gói để chạy là JAR. Ứng Rails hay Node.js đóng gói theo cấu trúc thư mục phân cấp.

Ứng dụng được viết kiểu trên đây rất phổ biến. Chúng dễ viết, dễ thử nghiệm, dễ copy & paste bởi các công cụ lập trình IDE và dự án mẫu được tối ưu để tạo ra ứng dụng khối đơn nhất. */*Do cần thuyết phục lập trình viên đi theo công nghệ/framework nên nhà sản xuất phải làm sao lập trình chỉ cần New Project, ấn nút Build & Run là ứng dụng chạy được ngay và luôn*/*. Có thể kiểm thử tự động giao diện web với Selenium. Ứng dụng khối đơn nhất khá dễ triển khai. PHP, Python thì chỉ cần cập nhật mã. Một số khác có chức năng khởi động nóng (hot reload: dịch lại nạp lên chạy tiếp) như Node.js, Play Framework, Revel. Để tăng khả năng chịu tải thì bổ xung thêm web application server giống nhau sau bộ cân bằng tải (load balancer). */*Tất nhiên mở rộng cơ sở dữ liệu liên tục ghi thành nhiều bản không dễ dàng*/*

Địa ngục kiến trúc một khối

Đáng tiếc rằng, cách tiếp cận kiến trúc đơn nhất tuy dễ dàng nhưng bắt đầu bộc lộ nhiều khiếm khuyết. Ứng dụng thành công - số lượng người dùng tăng - yêu cầu tính năng mới tăng - dữ liệu tăng - logic phức tạp hơn - giao tiếp với hệ thống khác tăng kết quả một ứng dụng khủng. Sau mỗi kỳ phát triển (sprint), đội phát triển bổ xung vài tính năng mới, thêm code, thêm bảng, thêm logic... Chỉ sau vài năm, ứng dụng đơn giản sẽ kềnh càng như quái vật. Tôi có trao đổi với một lập trình viên, người từng viết công cụ phân tích sự phụ thuộc giữa hàng nghìn gói thư viện JAR trong ứng dụng hàng triệu dòng code. Chắc chắn một số lượng lớn man month và tiền tấn để tạo ra quái vật khủng đến vậy.

Ứng dụng một khối mà phình to sẽ rắc rối như một gia đình nhiều thế hệ đông con cái ở trong cùng một nhà. Nhà to đến mấy rồi cũng sẽ gặp vấn đề. Mọi nỗ lực tối ưu, phương pháp làm việc agile (mềm dẻo) đều không còn hiệu quả. Một thư viện được tham chiếu nhiều chỗ khi nâng cấp sẽ phải kiểm tra ở tất cả những điểm, nghiệp vụ mà nó được gọi. Ứng dụng đơn nhất dùng một ngôn ngữ lập trình duy nhất. Đội lập trình sẽ quen với sự thuận tiện, dễ dàng khi chỉ cần nắm sâu một ngôn ngữ, một công cụ có thể giải quyết hầu hết vấn đề. Họ dần lệ thuộc vào ngôn ngữ đó và trở nên thiên vị, ngại cởi mở, tích hợp với những công nghệ khác của ngôn ngữ khác. Thậm chí khi framework hay ngôn ngữ có nhược điểm cố hữu, tính đơn nhất một khối của ứng dụng sẽ bó buộc lập trình viên thử nghiệm đưa vào thay đổi đột phá ngoại lai.

Ứng dụng một khối có hơn 2 triệu dòng mã trên framework XYZ, liệu đội bạn có đủ dũng cảm, nguồn lực để viết lại toàn bộ trên framework ABC mới hơn, tốt hơn. Lập trình giỏi, sáng tạo cũng không muốn làm trong kiểu dự án mặc kẹt này. Tình trạng giữ thì khổ, xây thì khó, khiến dịch vụ - sản phẩm của bạn ì ạch kém cạnh tranh so với các dịch vụ mới nổi uyển chuyển, linh hoạt.

Trong ứng dụng một khối, sự chặt chẽ là ưu điểm tự nhiên xuất phát từ kiến trúc, nhưng nó tiềm ẩn nguy cơ ràng buộc cứng nhắc đóng bê tông (tight coupling). Chi phí, thời gian, nỗ lực phát triển - sửa lỗi - kiểm thử một chức năng sẽ tăng tỷ lệ bậc 2 theo độ lớn của ứng dụng. Nói cách khác đi, mã nguồn khó đọc, khó bảo trì tỷ lệ bậc 2 theo số ràng buộc, tham chiếu được tạo ra hết sức dễ dãi khi phát triển.

Ở trên tôi có nói đến khả năng biên dịch nóng - khởi động lại (hot reload) khi code đổi, hay cân bằng tải bằng cách thêm nhiều ứng dụng web giống sau sau bộ cân bằng tại, nhưng khi ứng dụng to khủng, việc biên dịch nóng kéo dài hơn, khởi động lại sẽ chậm đi, copy phiên bản mới ra các server sẽ lâu hơn. Có những ứng dụng thời gian khởi động kéo dài từ 12-40 phút. Việc lập trình và gỡ rối tệ như để thay một con ốc trên đoàn tầu hỏa đang chạy, mà chúng ta phải dừng cả đoàn tầu rồi khởi động lại.

Khởi động lại ứng dụng một khối giống như dừng một đoàn tầu

Gần đây, bạn nghe nói nhiều hơn về triển khai đều đặn (continous deployment). Những ứng dụng SaaS (Software application as Service) tiên tiến, cần phải cập nhật vài lần trong một ngày. Quá khó để triển khai lại cả một ứng dụng cực lớn chỉ vì một số nâng cấp nhỏ. Hoạt động bị ngưng trệ, kiểm thử lại sau triển khai sẽ lâu công hơn. Kết quả là triển khai đều đặn khó áp dụng với ứng dụng một khối.

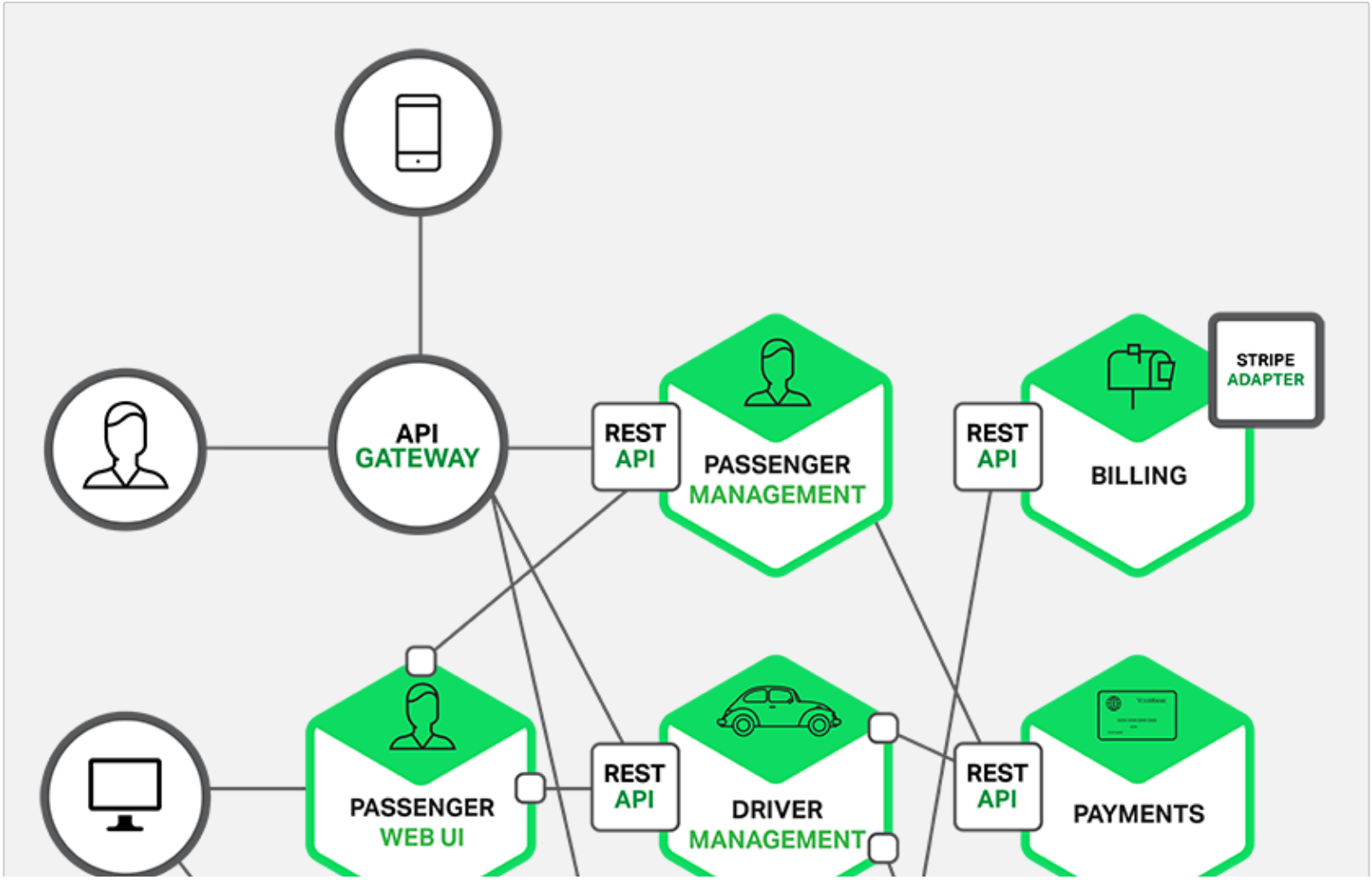
Khả năng mở rộng chịu tải ứng dụng một khối sẽ khó khi các thành phần khác nhau tranh chấp, dị biệt nhu cầu dùng tài nguyên hệ thống. Ví dụ module xử lý ảnh cần triển khai trên Amazon EC2 tối ưu CPU, sẽ khó cho module lưu bộ nhớ tạm (cache) cần rất nhiều bộ nhớ đang ra phải triển khai trên EC2 tối ưu bộ nhớ.

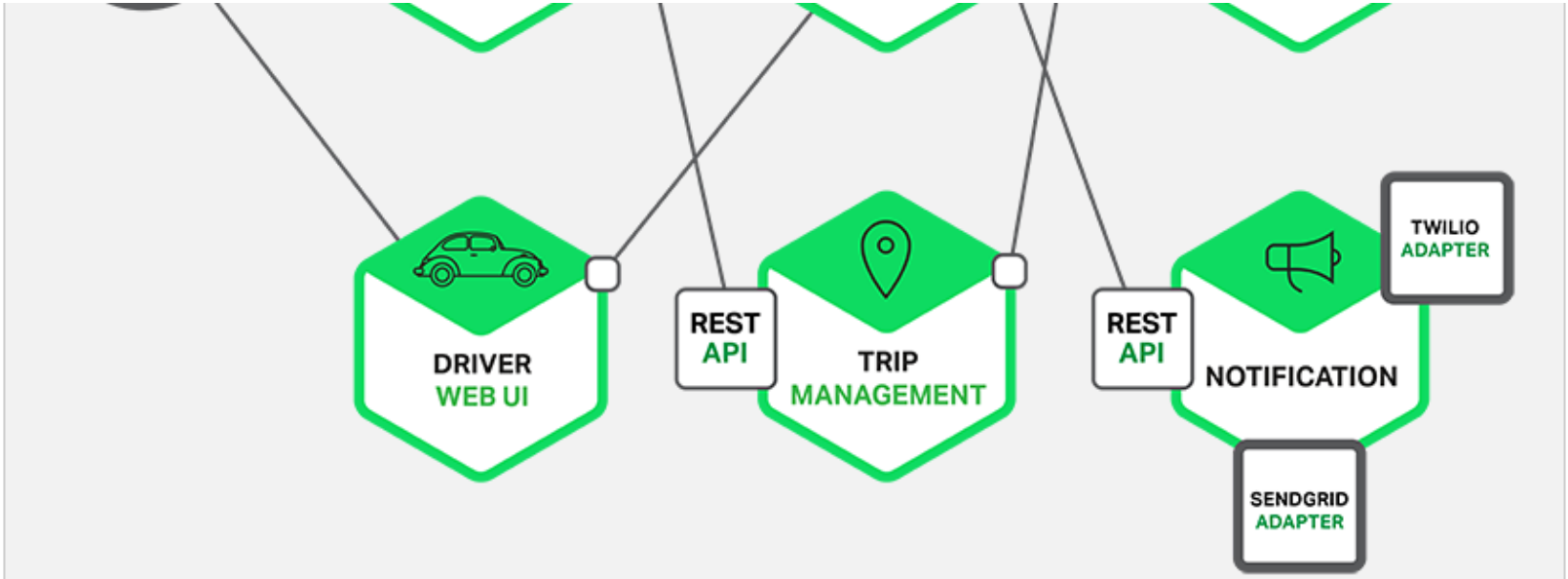
Tóm lại, bám vào kiến trúc một khối, một vé xuống địa ngục là cao hơn lên thiên đường.

Microservice - đơn giản hóa sự phức tạp

Nhiều tập đoàn như Amazon, eBay, Netflix đã giải quyết vấn đề ứng dụng một khối bằng kiến trúc microservices (nhiều dịch vụ nhỏ). Ý tưởng là chia nhỏ ứng dụng lớn ra thành các dịch vụ nhỏ kết nối với nhau.

Mỗi dịch vụ nhỏ thực hiện một tập các chức năng chuyên biệt như quản lý đơn hàng, quản lý khách hàng. Mỗi dịch vụ là một ứng dụng nhỏ có kiến trúc đa diện lõi là business logic kết nối ra các adapter khác nhau. Một số dịch vụ nhỏ lộ ra giao tiếp lập trình API cho dịch vụ nhỏ khác hay ứng dụng client gọi tới. Khi vận hành, mỗi dịch vụ nhỏ được chạy trong một máy ảo (virtual machine) hoặc Docker container (ảo hóa tầng ứng dụng).





Microservices của một công ty điều hành taxi kiểu Uber, Hailo

Mỗi vùng chức năng giờ được thực thị bởi một dịch vụ nhỏ. Ứng dụng web cũng có thể chia nhỏ hơn chuyên cho từng đối tượng người dùng (một cho hành khách taxi, một cho tài xế). Thiết kế giao diện cho từng đối tượng người dùng giúp tối ưu trải nghiệm tốt hơn, tốc độ nhanh hơn, dễ tương thích hơn trong khi chức năng tối giản hơn.

Mỗi dịch vụ đằng sau (back end service) lộ ra REST API (hiện nay còn nhiều lựa chọn khác như Google Protobuf, Apache Thrift, Apache Avro tốn ít băng thông hơn REST JSON)

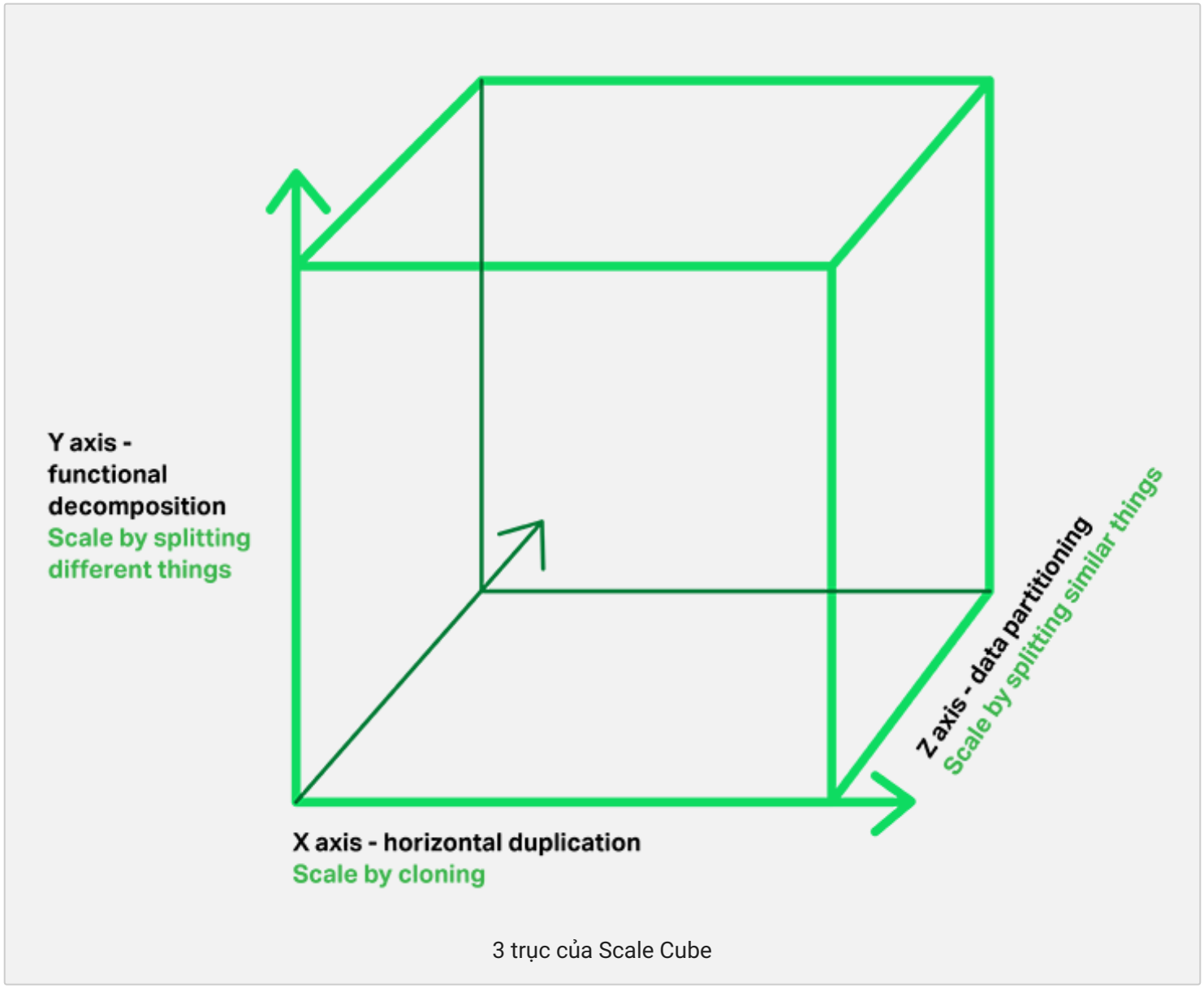
Các dịch vụ sẽ gọi / sử dụng API cung cấp bởi dịch vụ khác. Ví dụ dịch vụ quản lý tài xe sử dụng Notification Server để chủ động báo tài xế đang rảnh đón khách hàng tiềm năng. Phần giao diện (UI services) sẽ gọi đến các dịch vụ khác để lấy dữ liệu hiển thị. Hiện nay, pattern reactive cho phép dịch vụ có thể thông báo hoặc chủ động gửi dữ liệu mới để giao diện cập nhật. Đặc điểm của kết nối giữa các dịch vụ có thể là:

- Synchronous (đồng bộ - gọi xong chờ)
- Asynchronous (bất đồng bộ - gọi xong chạy tiếp. Khi có kết quả thì xử lý),

Cách gọi:
- REST (tập lệnh gửi qua HTTP để truy vấn, thao tác dữ liệu. Kiểu dữ liệu XML, JSON, JSONb)
- RPC (remote procedure call -lệnh gọi từ xa. Kiểu dữ liệu binary, Thrift, Protobuf, Avro)
- SOAP (Simple Object Access Protocol)

Một số dịch API REST có thể lộ ra cho thiết bị di động của hành khách và tài xế kết nối. Ứng dụng của người dùng cuối sẽ không được kết nối trực tiếp vào dịch vụ đằng sau. Thay vào đó có một cổng API (API gateway) đứng giữa. Cổng API có một số nhiệm vụ như phân tải, lưu tạm (cache), kiểm tra quyền truy cập, đo và theo dõi (API metering and monitoring).

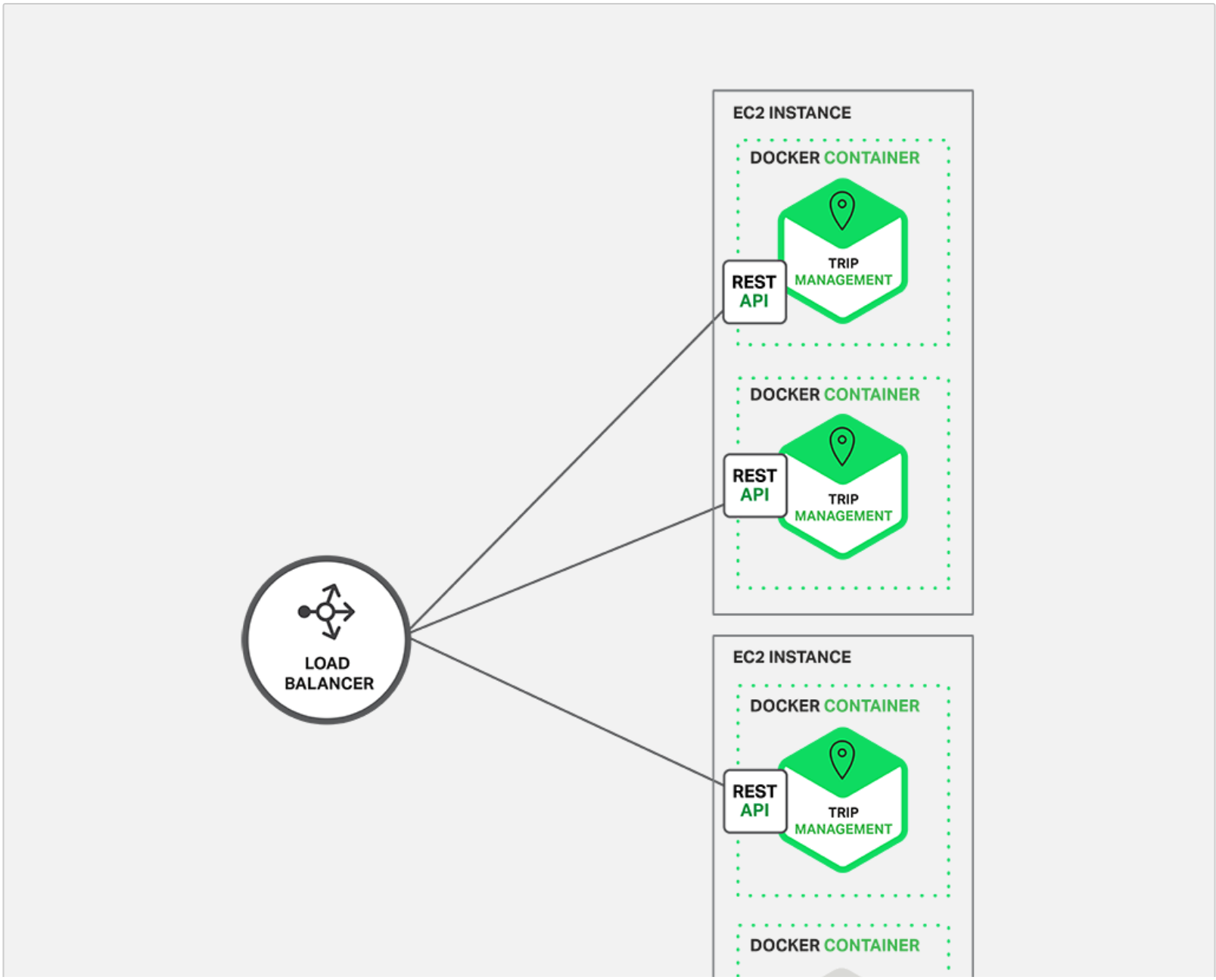
Kiến trúc microservice tương đương trục Y của 3 chiều mở rộng chịu tải (Scale Cube):
trục X : bổ xung thêm web application sau bộ phân tải
trục Y: chia nhỏ ứng dụng một khối thành nhiều dịch vụ nhỏ
trục Z: phân vùng dữ liệu để xử lý song song trên từng vùng



3 trục của Scale Cube

Biểu đồ dưới mô tả dịch vụ điều xe có thể được triển khai với Docker chạy trên Amazon EC2. Để tăng khả năng sẵn sàng, các Docker container chạy trên các máy ảo trên mây độc lập. Nginx làm nhiệm vụ phân tải, phân phối đều đặn các yêu

cầu đến từng dịch vụ.

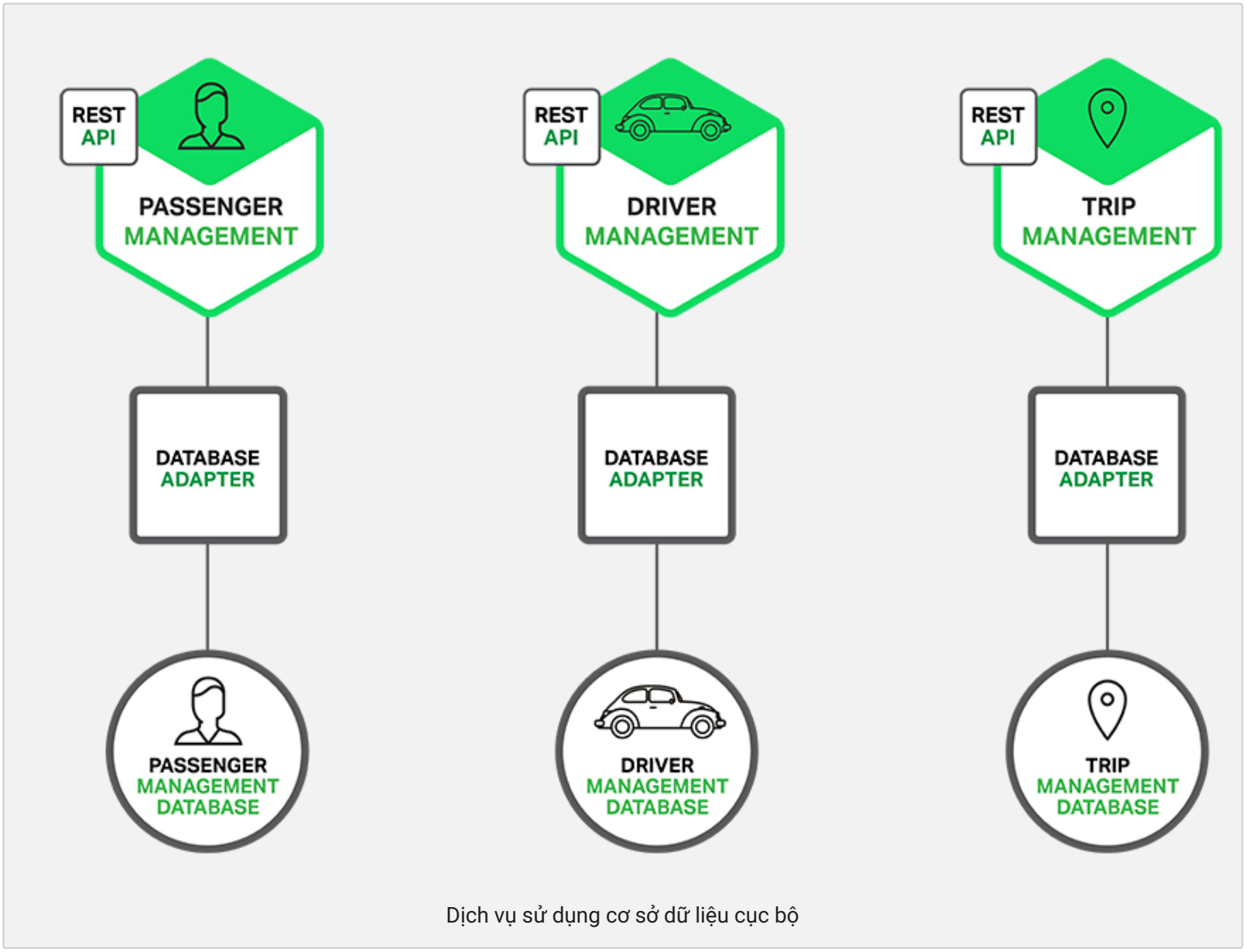


Các dịch vụ chạy trong Docker Container khác nhau, phía trước là bộ phân tải

Kiến trúc microservices ảnh hưởng lớn đến quan hệ ứng dụng và cơ sở dữ liệu. Thay vì dùng chung một cơ sở dữ liệu giữa các dịch vụ, mỗi dịch vụ sẽ có CSDL riêng. */*Cách này đi ngược lại tập quán tập trung hóa cơ sở dữ liệu. Hệ quả là sẽ có dư thừa dữ liệu, cơ chế foreign key ràng buộc quan hệ dữ liệu không thể áp dụng với bảng ở 2 cơ sở dữ liệu tách biệt. Thiết kế này sẽ gây sốc đối với nhiều lập trình đã quá quen với mô hình client - server, ở đó cơ sở dữ liệu luôn là một trung tâm, tập hợp mọi bảng.*/*

Tuy nhiên, lưu dữ liệu ở từng dịch vụ rất quan trọng nếu bạn muốn kiến trúc microservice thực sự hiệu quả vì nó đảm bảo loose coupling (ít ràng buộc)

*/*Chú ý đây chỉ là khuyến nghị của tác giả, trong thực tế, vài dịch vụ vẫn có thể dùng chung một CSDL khi tính toàn vẹn dữ liệu (ACID Atomicity, Consistency, Isolation, Durability) cần ưu tiên cao nhất.*/*



Dịch vụ sử dụng cơ sở dữ liệu cục bộ

Từng dịch vụ nhỏ có thể tùy chọn công nghệ lưu trữ dữ liệu tối ưu nhất ~ polygot persistence architecture. Ví dụ: dịch vụ điều xe cần phải dùng CSDL hỗ trợ việc truy vấn theo tọa độ tốt nhất. Dịch vụ cache thì dùng công nghệ lưu tạm ngay trong bộ nhớ, in memory key-value storage như Redis.

Phía bề nổi, kiến trúc microservice tương tự như SOA (kiến trúc hướng dịch vụ). Cả hai đều có một tập các dịch vụ. Điểm khác là microservice không dùng chuẩn do các tập đoàn lớn như IBM, Microsoft, Oracle đặt ra như WS-* hay Enterprise Service Bus. Nó hướng đến các chuẩn có tính cạnh tranh cao hơn như Protobuf, Thrift hoặc cởi mở hơn, dễ đọc như JSON. Microservices do các công ty start up khởi xướng từ 2010, trong khi SOA do tập đoàn lớn đề xuất từ 1990 trong giải pháp thương mại của họ và ưu tiên dùng thông điệp dạng XML. Microservice không áp dụng một số phần của SOA như canonical schema (phân cấp thông điệp từ mức tập đoàn - công ty - phòng ban - bộ phận). Có thể thấy Microservice gọn hơn, đa dạng hơn trong giao thức - chuẩn dữ liệu.

Ưu điểm của Microservices

1- Giảm thiểu sự gia tăng phức tạp rối rắm hệ thống lớn.
2- Chia nhỏ ứng dụng một khối cồng kềnh thành các dịch vụ nhỏ dễ quản lý, bảo trì nâng cấp, tự do chọn, nâng cấp công nghệ mới.
3- Mỗi dịch vụ nhỏ sẽ định ra ranh giới rõ ràng dưới dạng RPC hay API hướng thông điệp.
4- Microservice thúc đẩy tách rạch ròi các khối chức năng (loose coupling - high cohesion), điều rất khó thực hiện với ứng dụng một khối. Nếu muốn loose coupling - high cohesion trong ứng dụng một khối, sẽ phải thiết kế theo Design Pattern (Gang Of Four) và liên tục tái cấu trúc (refactor)

Mỗi dịch vụ nhỏ sẽ phát triển dễ hơn, nhanh hơn, dễ viết mã kiểm thử tự động.

Một số dịch vụ có thuê ngoài phát triển mà vẫn bảo mật hệ thống - mã nguồn phần dịch vụ còn lại. Đội phát triển có nhiều lựa chọn công nghệ mới, framework, CSDL mới, đa dạng để nâng cấp từng dịch vụ nhỏ, chọn môi trường tối ưu nhất để chạy. Các dịch vụ có thể bật tắt để kiểm nghiệm so sánh A|B, tăng tốc quá trình cải tiến giao diện. Triển khai đều đặn khả thi với microservice. Dịch vụ nhỏ đóng gói trong Docker container có thể chuyển từ môi trường phát triển sang môi trường chạy thật không phải cấu hình thủ công lại, không phải copy file quá lớn.

Nhược điểm của microservices

Nhược điểm đầu tiên của microservices cũng chính từ tên gọi của nó. Microservice nhấn mạnh kích thước nhỏ gọn của dịch vụ. Một số lập trình đề xuất dịch vụ siêu nhỏ cỡ dưới 100 dòng code. Chia quá nhiều sẽ dẫn đến manh mún, vụn vặt, khó kiểm soát. Việc lưu dữ liệu cục bộ bên trong những dịch vụ quá nhỏ sẽ khiến dữ liệu phân tán quá mức cần thiết.

Nhược điểm tiếp của microservice đến từ đặc điểm hệ thống phân tán (distributed system):
1- Phải xử lý sự cố khi kết nối chậm, lỗi khi thông điệp không gửi được hoặc thông điệp gửi đến nhiều đích đến vào các thời điểm khác nhau.

2- Đảm bảo giao dịch phân tán (distributed transaction) cập nhật dữ liệu đúng đắn (all or none) vào nhiều dịch vụ nhỏ khác nhau khó hơn rất nhiều, đôi khi là không thể so với đảm bảo giao dịch cập nhật vào nhiều bảng trong một cơ sở dữ liệu trung tâm.

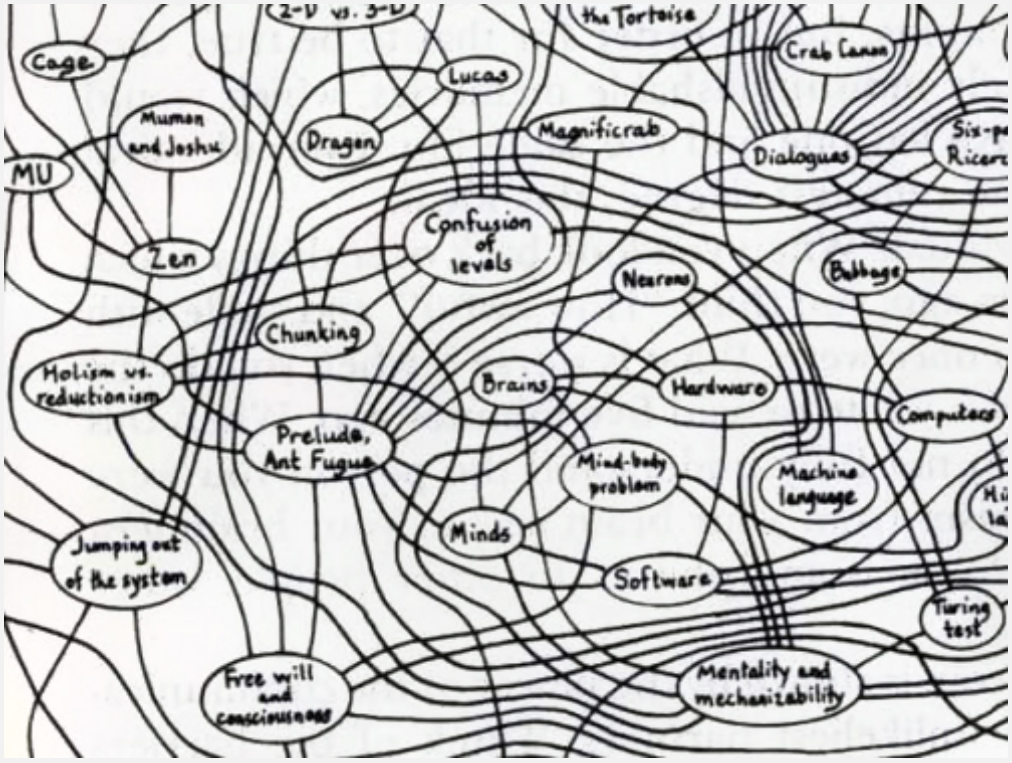
3- Theo nguyên tắc CAP (CAP theorem) thì giao dịch phân tán sẽ không thể thỏa mãn cả 3 điều kiện: consistency (dữ liệu ở điểm khác nhau trong mạng phải giống nhau), availablity (yêu cầu gửi đi phải có phúc đáp), partition tolerance (hệ thống vẫn hoạt động được ngay cả khi mạng bị lỗi). Những công nghệ cơ sở dữ liệu phi quan hệ (NoSQL) hay môi giới thông điệp (message broker) tốt nhất hiện nay cũng chưa vượt qua nguyên tắc CAP.

4- Kiểm thử tự động một dịch vụ trong kiến trúc microservices đôi khi yêu cầu phải chạy cả các dịch vụ nhỏ khác mà nó phụ thuộc. Do đó khi phân rã ứng dụng một khối thành microservices cần luôn kiểm tra mức độ ràng buộc giữa các dịch vụ mềm dẻo hơn hay cứng nhắc - lệ thuộc hơn. Nếu ràng buộc ít đi, lỏng leo hơn, bạn đi đúng hướng và ngược lại.

5- Nếu các dịch vụ nhỏ thiết kế phục thuộc vào nhau theo chuỗi. A gọi B, B gọi C, C gọi D. Nếu một mắt xích có giao tiếp API thay đổi, liệu các mắt xích khác có phải thay đổi theo không? Nếu có thì việc bảo trì, kiểm thử sẽ phức tạp tương tự ứng dụng một khối. Thiết kế dịch vụ tốt sẽ giảm tối đa ảnh hưởng lan truyền đến các dịch vụ khác.

thay đổi lan truyền trong ứng dụng khiến cho việc nâng cấp, kiểm tra khó khăn hơn

6- Cuộc họp có 2 người sẽ có 1 bắt tay, 3 người có 3 bắt tay. 4 người có 6 bắt tay, 5 có 10 bắt tay. Tổ hợp chập 2 của 5 = 10, công thức tổng quát = n! /((n-2)! * 2!). Thực thế không phải dịch vụ nào trong microservice cũng sẽ giao tiếp với tất cả những dịch vụ còn lại. Nhưng nếu không có quy tắc phân luồng - quản lý - đo đếm - theo dõi (manage - meter - monitor) , thì số lượng kết nối giữa các dịch vụ nhỏ gia tăng tùy tiện, chất lượng kết nối không kiểm soát. Hệ thống chậm nhưng không thể biết đoạn nghẽn cổ chai ở đâu?



Kết nối giữa các dịch vụ có thể rối rắm như thế này

7- Triển khai dịch vụ microservices nếu làm thủ công theo cách đã làm với ứng dụng một khối phức tạp hơn rất nhiều. Ứng dụng một khối bổ xung các server mới giống hệt nhau đằng sau bộ cần bằng tại. Trong khi ở kiến trúc microservice, các dịch vụ nhỏ nằm trên nhiều máy ảo hay Docker container khác nhau, hoặc một dịch vụ có nhiều thực thể phân tán ra nhiều. Theo Adrian Crockcroft, Hailo có 160 dịch vụ, NetFlix có hơn 600 dịch vụ. Trong dịch vụ đám mây, các máy ảo, docker container, thực thể có thể linh động bật tắt, dịch chuyển. Vậy cần thiết phải có một cơ chế phát hiện dịch vụ (service discovery mechanism) để cập nhật tự động địa chỉ IP và cổng, mô tả, phiên bản của mỗi dịch vụ.

Kết luận

Kiến trúc một khối sẽ hữu hiệu đối với ứng dụng đơn giản, ít chức năng. Nó bộc lộ nhiều nhược điểm khi ứng dụng phát triển lớn nhiều chức năng. Kiến trúc microservices chia nhỏ kiến trúc một khối ra các dịch vụ nhỏ. Microservices sẽ hiệu quả, phù hợp cho những ứng dụng phức tạp, liên tục phát triển nếu được thiết kế đúng và tận dụng các công nghệ quản lý, vận hành tự động.

> *Hiện nay các khóa học hướng dẫn triển khai kiến trúc microservices đang được xây dựng ở Techmaster.*
> *- Node.js xây dựng web site tốc độ cao*
> *- Docker (giáo trình đang làm dở)*
> *- Golang sẽ ra trong tháng 12/2015*
> *- Cơ sở dữ liệu phi quan hệ Postgresql - MongoDB 1/2016*

Monster Pixel *Ho Chi Minh* *$700 - $1,000*

Swift iOS Objective-C

Junior QC Engineer (Tester, Manual Test , Test Apps)

Công ty Cổ phần Công nghệ và Dịch vụ Moca *Ho Chi Minh* *Up to $600*

QC Tester Manual Test Test Apps

Tìm anh tài Brse tại Japan và Hà Nội

CÔNG TY CỔ PHẦN DIGI DINOS *Ha Noi* *$1,000 - $2,500*

BrSE

PHP vs Node.js - Cuộc chiến giữa hai công nghệ lập trình web
13/10/2015 Hồ Sỹ Hùng

BLOG HOME

7 Quy tắc cấu trúc ứng dụng Node.js
24/07/2017 Đinh Thiên Phúc

Bởi Trịnh Minh Cường

Mình bắt đầu lập trình Pascal từ năm 1993, với chiếc máy PC 2Mb RAM, 40Mb ổ cứng. Đến nay, đã hoàn thành gần 60 dự án phần mềm lớn nhỏ. Mình dạy lớp iOS, Node.js tại Techmaster. Ngoài ra mình còn làm cố vấn công nghệ cho các nhóm khởi nghiệp, kiêm quản lý dự án tại Techmaster. Mình bơi mỗi ngày 2000 mét.