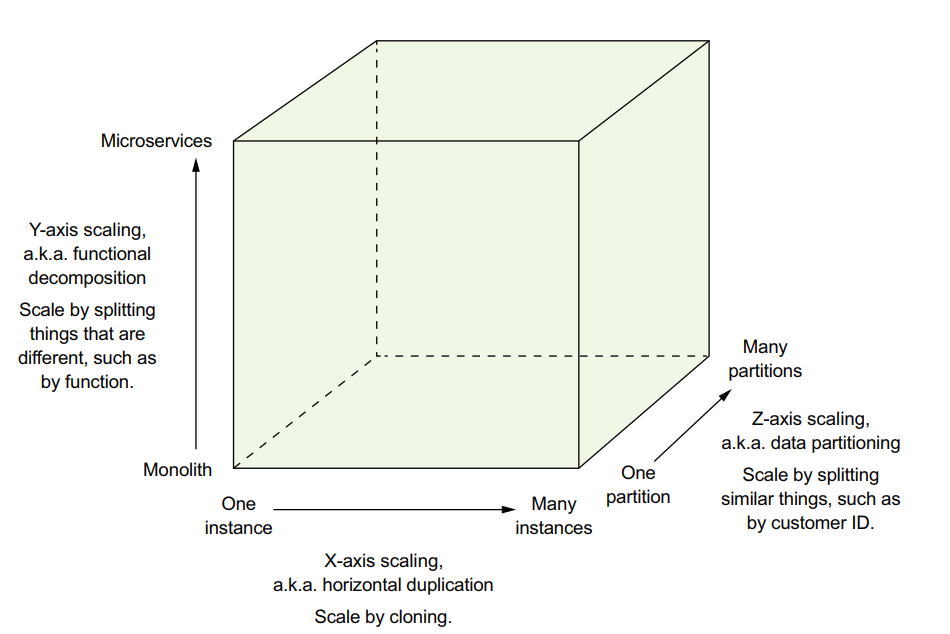
Microservice

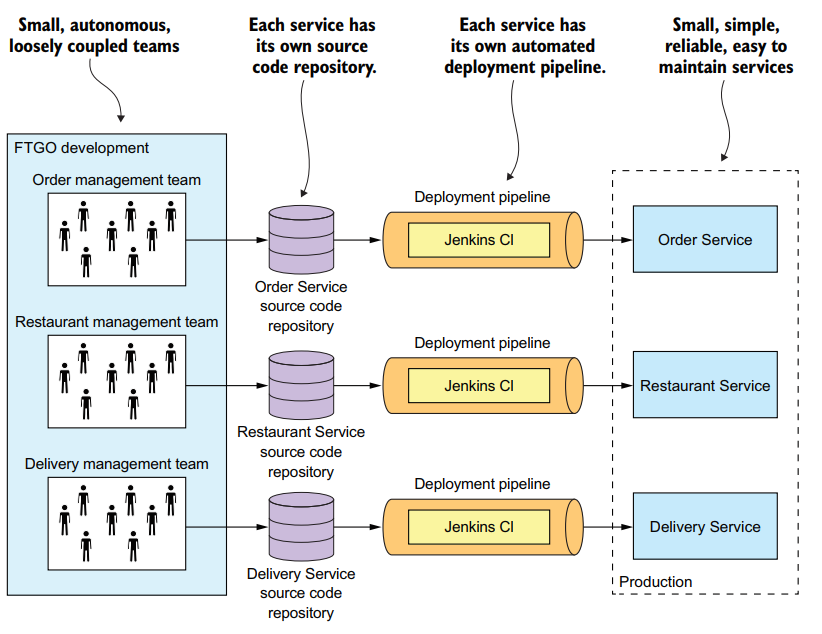
# Lợi ích của kiến trúc monolithic

* Dễ dàng phát triển
* Dễ dàng thực hiện thay đổi tới ứng dụng:Bạn có thể thay đổi code và database schema, build và deploy
* Đơn giản test
* Đơn giản deploy
* Dễ dàng scale



# Lợi ích của microservice

* Cho phép continuous delivery và deployment ứng dụng lớn, phức tạp.
  + Có 3 hướng mà kiến trúc micro cho phép continuous delivery/deployment
    - Dễ dàng test: Do mỗi service trong kiến trúc micro nhỏ, tự động test rất dễ viết và thực hiện. Kết quả là ứng dụng sẽ có ít bug hơn.
    - Dễ dàng deploy:Mỗi service có thể tự deploy độc lập.Nếu đội phát triển chịu trách nhiệm cho service đó cần deploy thay đổi tới service đó, họ có thẻ đơn giản deploy thay đổi của họ lên. Nó dễ dàng hơn việc deploy thay đổi liên tục tới production.
    - Cho phép đội phát triển có thể tự trị và phụ thuộc lỏng lẻo: Bạn có thể tổ chức đội kĩ sư thành các nhóm nhỏ. Mỗi nhóm tự chịu trách nhiệm cho việc phát triển và deploy một hoặc một vài service liên quan. Điều này làm cho tốc độ phát triển tăng lên đáng kể.
  + Việc này có một số lợi ích về business:
    - Giảm thời gian phát triển, cho phép business nhanh chóng nhận được phản hồi từ khách hàng
    - Cho phép business cung cấp dịch vụ đáng tin cậy cho khách hang ngày nay mong đợi.
    - Tăng sự hài long của nhân viên bởi vì sẽ có thêm thời gian để delivery các tính năng có giá trị.
* Service nhỏ, dễ bảo trì
  + Một lợi ích khác của kiến trúc micro là mỗi service tương đối nhỏ. Code sẽ dễ dàng hơn để dev có thể hiểu. Lượng code nhỏ sẽ không khiến IDE trở nên chậm, giúp cho dev phát triển hiệu quả hơn. Và mỗi service khởi động nhanh hơn một ứng dụng monolith, làm cho dev hiệu quả hơn và tốc độc triển khai nhanh hơn.



* Service có thể độc lập deploy
* Service có thể độc lập scale
  + Mỗi service trong kiến trúc micro có thể tự scale độc lập. Hơn nữa, mỗi service có thể deploy trên các phần cứng phù hợp nhất với yêu cầu tài nguyên của chúng. Nó hoàn toàn khác khi sử dụng khiến trúc monolithic.
* Cho phép độiphát triển trở nên tự động hóa cao
* Cho phép dễ dàng thử và áp dụng các công nghệ mới
  + Về nguyên tắc, khi ta phát triển một service mới, dev có thể chọn bất cứ ngôn ngữ hay framework nào phù hợp nhất với service đó.
  + Hơn nữa, vì cá dịch vụ là nhỏ, việc viết lại chúng sử dụng ngôn ngữ và công nghệ tốt hơn trở nên thực tế. Nếu việc thử nghiệm công nghệ mới thất bại, bạn có thể ném nó đi mà không gây ra rủi ro cho dự án. Nó hoàn toàn khác khi bạn làm việc với kiến trúc monolithic, khi mà bạn lựa chọn các công nghệ ban đầu bị hạn chế nghiêm trọng khả năng sử dụng ngôn ngữ hay framework trong tương lai.
* Cách ly lỗi tốt
  + Kiến trúc micro có khả năng cách li lỗi tốt. Ví dụ, nếu bộ nhớ bị rò rỉ trên 1 service thì chỉ có tác động tới service đó. Các service khác vẫn tiếp túc xử lý các yêu cầu bình thường. Ngược lại, nếu sử dụng kiến truc monolithic, một thành phần lỗi sẽ gây tác động tới toàn bộ hệ thống.

# Nhược điểm của kiến trúc microservice

Như ta biết, không có công nghệ nào là mũi tên bạc cả, và kiến trúc micro cũng vậy, nó cũng có một số nhược điểm và vấn đề.

Dưới đây là một số nhược điểm và vấn đề chính của kiến trúc micro:

* + Tìm ra tập các service sao cho đúng là một thách thức
    - Một thách thức trong việc sử dụng kiến trúc micro chính là việc không có bất cứ giải thuật rõ ràng được định nghĩa nào cho việc chia nhỏ hệ thống thành các service. Nếu bạn chia nhỏ hệ thống không chính xác, bạn sẽ xây dựng dành 1 distributed monolith(hệ thống monolith phân tán), hệ thống bao gồm các cặp service phải được deploy cùng nhau. Một distributed monolith có nhược điểm của cả kiến trúc monolith và kiến trúc micro.
  + Hệ thống phân tán phức tạp, khiến việc phát triển, test, deploy trở nên khó khăn.
    - Một vấn đề khác khi sử dụng kiến trúc micro chính là dev phải đối phó với sự phức tạp của việc tạo ra một hệ thống phân tán (distributed system). Các service phải sử dụng cơ chế giao tiếp giữa các tiến trình (interprocess communication mechanism).Nó trở nên phức tạp hơn một phương thức goi bình thường. Hơn nữa, service phải được thiết kế để xử lý lỗi một phần và đối phó với các dịch vụ từ xa có thể không có hoặc độ trễ cao.
    - Triển khai các use case trên vùng các service đòi hỏi việc sử dụng các kĩ thuật không quen thuộc. Mỗi service sẽ có database riêng của nó, gây ra một thách thức khi triển khai transaction và query trên vùng các service.
    - IDE và các tool phát triển tập trung vào xây dựng ứng dụng monolith chứ không cung cấp các hỗ trợ rõ rang cho việc phát triển các ứng dụng phân tán. Viết các automatic test gọi nhiều service là một thách thức. Đó là tất cả các vấn đề đặc trưng cho kiến trúc micro. Do đó, dev của bạn phải có các kĩ năng phát triển phần mềm phức tạp và delivery để thành công trong việc sử dụng microservice.
    - Để thành công trong việc deploy microservice, bạn phải sử dụng các công nghệ như sau:
      * Automated deployment tooling: như Netflix Spinnaker
      * Một of-the-shelf PaaS như Pivotal Cloud Foundry hay Red Had OpenShift.
      * Một Docker orchestration platform như Docker Swarm hay Kubernetes
  + Deploy các tính năng trên một vùng dịch vụ yêu cầu sự phối hợp cẩn thận
    - Một thách thức khác khi sử dụng kiến trúc micro là việc deploy các tính năng trên một vùng service đòi hỏi sự phối hợp cẩn thận giữa các đội phát triển. Bạn phải tạo ra một kế hoạch triển khai thứ tự deployment service dựa trên phụ thuộc giữa các service.
  + Khó khăn trong việc ra quyết định sử dụng kiến trúc micro vào ứng dụng.
    - Khi phát triển phiên bản đầu tiên của ứng dụng, bạn thường không có các vấn đề về kiến trúc cần giải quyết. Hơn nữa, việc sử dụng kiến trúc phân tán sẽ làm quá trình phát triển chậm đi. Đó có thể là một tình trạng khó xử cho startup, khi các vấn đề lớn thường là làm thế nào phát triển nhanh chóng các mô hình kinh doanh và các ứng dụng đi kèm. Sử dụng kiến trúc micro làm cho nó trở nên khó khăn để nhanh chóng. Một startup gần như chắc chắn sẽ bắt đầu với một ứng dụng monolithic.

Ta có thể thấy, kiến trúc micro có nhiều lợi ích, bên cạnh đó sẽ có những nhược điểm. Bởi vì các vấn đề đó, việc áp dụng kiến trúc cần cẩn trọng. Nhưng với các ứng dụng phức tạp, kiến trúc micro sẽ là lựa chọn đúng đắn.

Bạn phải giải quyết rất nhiều vấn đề về thiết kế và kiến trúc khi sử dụng kiến trúc micro. Hơn thế nữa, nhiều vấn đề trong đấy sẽ có nhiều giải pháp, mỗi cái lại khác nhau. Sẽ không có giải pháp nào là giải pháp duy nhất hoàn hảo.

# “Ngôn ngữ” Microservice architecture pattern

## Kiến trúc micro không phải là “viên đạn bạc”

Trở lại những năm 1986, Fred Brooks, tác giả của cuốn sách The Mythical Man-Month, từng nói rằng: Trong thế giới phần mềm, không có gì là “viên đạn bạc”. Tức không có công nghệ hay kĩ thuật nào khi áp dụng giúp cho bạn tang mười lần năng suất. Một thập kỉ sau, các dev vẫn còn tranh cãi về viên đạn bạc mà họ yêu thích, hoàn toàn bị thuyết phục rằng công nghệ yêu thích của họ thúc đẩy năng suất của họ.

Có một lập luận như sau: “Nếu bạn làm X, sau đó một con chó sẽ chết, do đó bạn phải làm Y”. Ví dụ, bất đồng bộ với reactive programming, hướng đối tượng với hướng chức năng, Java với Javascript, Rest với message. Tất nhiên, thức tế là nhiều sắc thái. Mọi công nghệ có nhược điểm và hạn chế thường bị bỏ qua bởi những người ủng hộ chúng. Vì vậy, việc áp dụng một công nghệ thường theo một chu kỳ Gartner hype cycle, đi qua 5 giai đoạn, bao gồm: peak of inflated expectations (đỉnh cao sự mong đợi), trough of disillusionment (máng của sự vỡ mộng) và kết thúc là plateau of productivity (cao nguyên năng suất).

Micro thích hợp cho ứng dụng của bạn dựa vào nhiều yếu tố đánh giá. Lời khuyên luôn luôn sử dụng kiến trúc này là lời khuyên tồi, nhưng cũng là lời khuyên tồi nếu tôi khuyên bạn không bao giờ sử dụng nó.

Lý do cơ bản cho những lập luận này về công nghệ là do con người chủ yếu được điều khiển bởi cảm xúc của họ. Trí óc con người như con voi và kĩ sĩ: Con voi tượng trưng cho phần cảm xúc của não người, hầu hết là cho ra quyết định. Còn kĩ sĩ đại diện cho phần lập luận hợp lý của não. Nó có thể thi thoảng ảnh hưởng tới con voi, nhưng luôn cung cấp các luận cứ cho các quyết định của con voi.

Chúng ta là cộng đồng các nhà phát triển. cần vượt qua được cảm xúc tự nhiên và tìm một cách tốt nhất để bàn luận và áp dụng công nghệ. Một cách tuyệt vời để bản lâunj và miêu tả công nghệ là sử dụng dạng pattern. Khi miêu tả một công nghệ trong dạng pattern, bạn phải nói về nhược điểm

## Pattern và ngôn ngữ pattern

Pattern là một giải pháp tái sử dụng để giải quyết vấn đề xuát hiện ở trong một ngữ cảnh cụ thể. Nó là ý tưởng có nguồn gốc từ kiến trúc thực tế và được chứng minh hữu dụng trong kiến trúc và thiết kế phần mềm. Khái niệm pattern được tạo ra bởi Christoguage Alexander. Ông ta cũng tạo ra khái niệm ngôn ngữ pattern, một taạp hợp cá pattern liên quan giải quyết các vấn đề trong một miền cụ thể.

Các tác phẩm của Alexander đã truyền cảm hứng cho cộng động phần mềm để áp dụng các khái niệm của pattern và ngôn ngữ pattern. Sách Design pattern: Elementss of Reusable Object-Oriented Software viết bởi Erich Gamma, Richard Helm… là một tập hợp các design pattern hướng đối tượng.

Hãy tưởng tượng nếu bạn đang xây dựng một hệ thống banking phải hỗ trợ hang ngàn chính sách. Mỗi chính sách định nghĩa giới hạn trên của số dư tài khoản và phí cho một tài khoản thấu chi. Bạn có thể giải quyết vấn đề đó sử dụng Strategy pattern, bao gồm 3 phần:

* Một interface strategy gọi là Overdraft đóng gói giải thuật overdraft
* Một hoặc nhiều class cụ thể cho interface đó, mỗi cái cho một ngữ cảnh cụ thể
* Class Account sử dụng giải thuật đó.

Strategy pattern là một design pattern hướng đối tượng, do vậy các thành phần của giải phátp đó là các class.

Một lý do khác tại sao pattern có giá trị bởi vì pattern được mô tả ngữ cảnh nó được áp dụng. Ý tưởng rằng một giải pháp là một cụ thể hóa của một ngữ cảnh nhất định và có thể không hoạt động tốt trong ngữ cảnh khác, là một sự cải tiến cách sử dụng công nghệ. Ví dụ, giải pháp giải quyết vấn đề scale của Netflix sẽ không là phương pháp tốt nhát cho một ứng dụng ít người dùng hơn.

Cấu trúc pattern thường được sử dụng bao gồm 3 phần đặc biệt:

* Forces
* Ngữ cảnh kết quả
* Pattern liên quan

Forces ở đây chính là các vấn đề bạn phải giải quyết khi giải quyết một vấn đề trong ngữ cảnh. Forces có thể xung đột, do đó có thể không giải quyết được tất cả chúng. Forces mà quan trọng hơn phụ thuộc vào ngữ cảnh. Bạn phải ưu tiên giải quyết một số forces hơn. Ví dụ, code phải dễ để hiểu và có hiệu năng tốt. Code viết theo kiểu reactive sẽ có hiệu năng cao hơn kiểu synchronous,nhưng khó hiểu hơn. Rõ rang liệt kê các forces rất hữu ích vì nó sẽ giúp làm rõ các vấn đề cần giải quyết

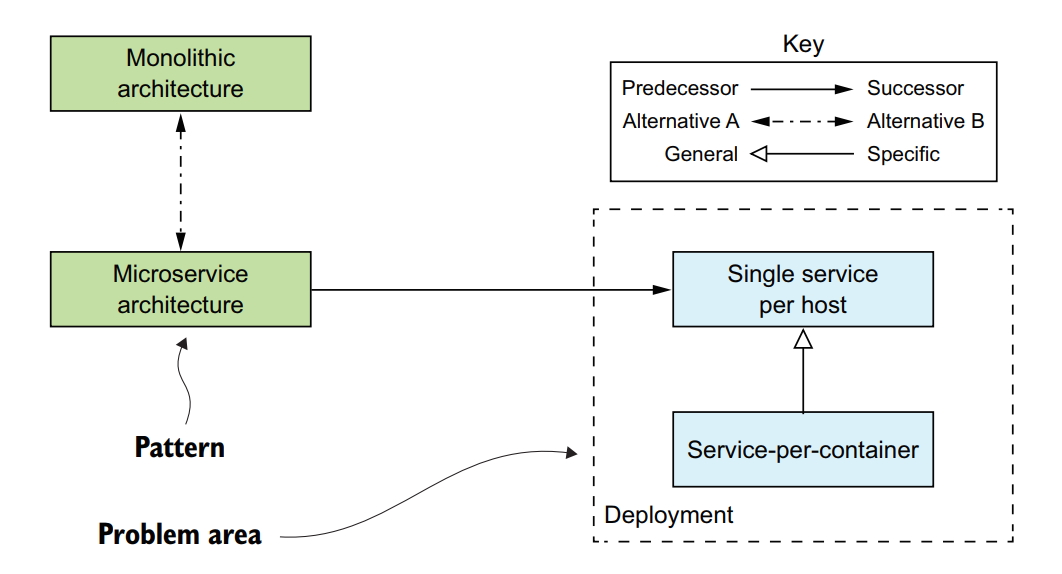
Ngữ cảnh kết quả: Kết quả của việc áp dụng pattern. Nó bao gồm 3 phần

* Lợi ích : lợi ích của pattern, bao gồm các force đã được giải quyết
* Nhược điểm: Nhược điểm của pattern, bao gồm các force chưa giải quyết được
* Các vấn đề: Các vấn đề mới xuất hiện khi áp dụng pattern.

Ngữ cảnh kết quả cong cung một cái nhìn đầy đủ và ít sai lệch về giải pháp. Từ đó cho phép ra quyết định dễ dàng hơn.

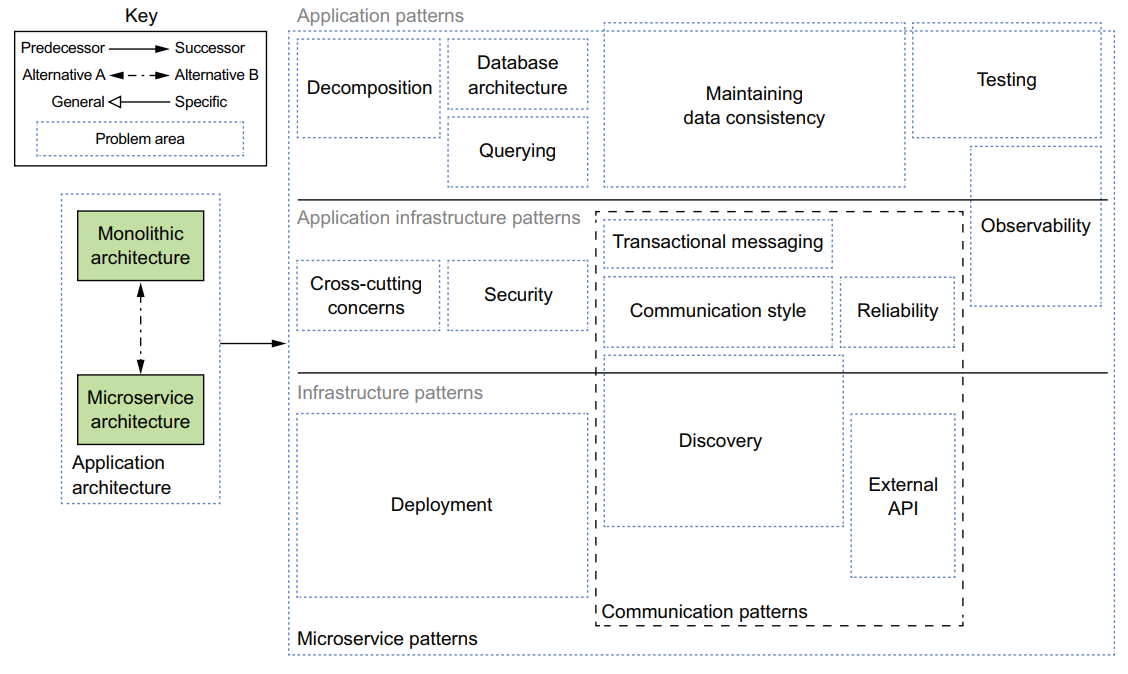
Các pattern liên quan: Có 5 kiểu. Nó miêu tả mối quan hệ giữa các pattern, bao gồm:

* Predecessor (tiền nhiệm): Một pattern tiền nhiệm là pattern thúc đẩy sự cần thiết cho pattern này. Ví dụ,pattern kiến trúc micro là tiền nhiệm của phần còn lại pattern trong ngôn ngữ pattern, ngoại trừ pattern kiến trúc monolithic
* Successor (kế thừa): Một pattern giải quyết vấn đề mà được chỉ ra bởi chính pattern này.Ví dụ, nếu bạn áp dụng pattern kiến trúc micro, bạn phải áp dụng hang loạt các pattern successor nó, bao gồm service discovery pattern,circuit breaker pattern.
* Alternative (thay thế): Một pattern cung cấp sự thay thế giải pháp cho pattern này. Ví dụ, kiến trúc mono và kiến trúc micro là 2 cách thiết kế ứng dụng, bạn sẽ chọn 1 trong hai.
* Generalization: Một pattern là giải pháp chung cho một vấn đề. Ví dụ, như Deploy a service as a container pattern là một pattern đặc biệt của Single service per host.



## Tổng quát về ngôn ngữ pattern kiến trúc microservices

Là một tập hợp các pattern giúp bạn kiến trúc ứng dụng sử dụng kiến trúc microservice.



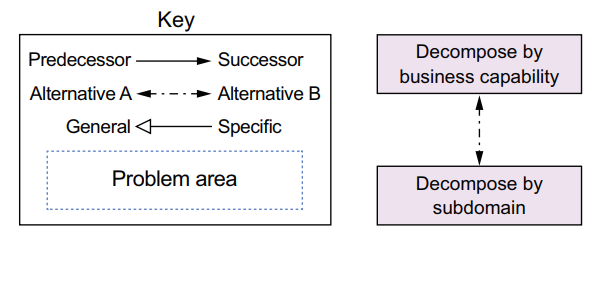
Hình trên sẽ giúp bạn quyết định khi nào sử dụng kiến trúc này. Nó miêu tả một kiến trúc monolithic và kiến trúc micro, với các lợi ích và khuyết điểm. Nếu kiến trúc micro phù hợp với ứng dụng của bạn, nó sẽ giúp bạn sử dụng một cách hiệu quả bằng cách giải quyết các vấn đề về thiết kế và kiến trúc khác.

Pattern này bao gồm một vài nhóm pattern. Bên trái là nhóm pattern kiến trúc ứng dụng, Monothilic và Microservice. Phần còn lại bao gồm các nhóm pattern là giải pháp để giải quyết các vấn đề được dùng bởi pattern kiến trúc microservice, được chia thành 3 lớp:

* Infrastructure patterns: Giải quyết các vấn đề về hạ tầng nằm ngoài việc phát triển.
* Application infrastructure: Giải quyết vấn đề kiến trúc có tác động tới việc phát triển.
* Application patterns: Giải quyết các vâsn đề mà đội phát triển phải đối mặt.

## Pattern cho việc phân tách ứng dụng thành các service

Phân tách hệ thống thành các tập service là một nghệ thuật, nhưng có một số chiến lược có thể giúp bạn. 2 pattern sau là 2 chiến lược khác nhau , bạn có thể sử dụng để đưa ra định nghĩa cho kiến trúc ứng dụng của bạn.



## Communication Patterns

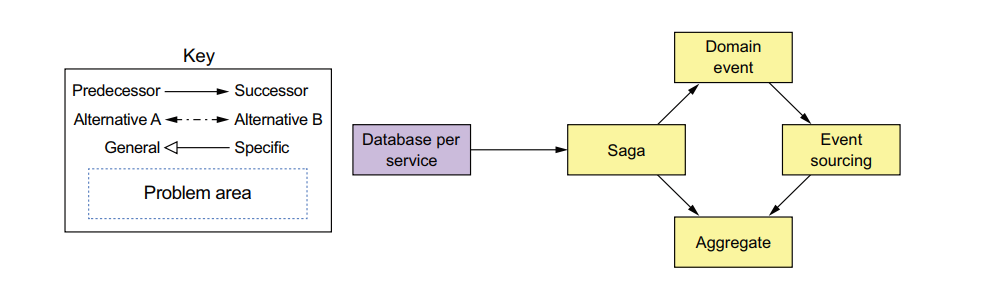
Ứng dụng được xây dựng sử dụng kiến trúc micro là hệ thống phân tán. Do đó, việc giao tiếp giữa các tiến trình IPC là một phần quan trọng của kiến trúc micro. Bạn phải thực hiện một loại các quyết định về kiến trúc và thiết kế về cách các service giao tiếp với nhau. Các pattern của loai này chia thành 5 nhóm:

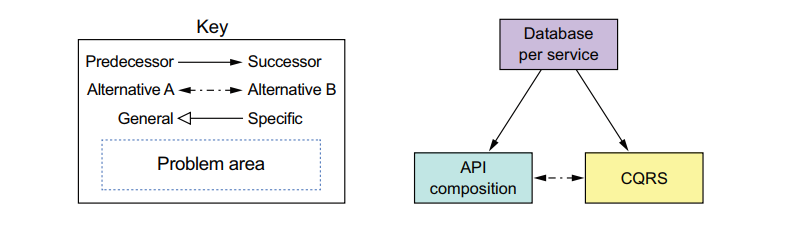
* Communication style: Cơ chế IPC nên sử dụng?
* Discovery: Làm thế nào để client của một service có thể xác định địa chỉ IP của instance service, nó phải tạo một HTTP request?
* Reliability: Bạn có thể đảm bảo sự giao tiếp giữa các service là tin cậy thậm chí service có thể unavailable
* Transactional messaging: Làm thế nào bạn tích hợp việc gửi message và việc publish sự kiện với database transaction để cập nhật dữ liệu kinh doanh.
* External API: Làm thế nào client của ứng dụng giao tiếp với các service

## Pattern nhất quán dữ liệu cho việc thực thi quản lý transaction

Để đảo bảo phụ thuộc lỏng lẻo, mỗi service sẽ có database của nó. Không may, điều này dẫn tới một số vấn đề. Các cách tiếp cận truyền thống của việc sử dụng các distributed transactions không phải là lựa chọn khả thi cho các ưgns dụng hiện đại. Thay vào đó, ứng dụng của bạn cần duy trì tính nhất quán dữ liệu bằng Saga pattern.

## Pattern cho việc query dữ liệu trong kiến trúc micro





Một vấn đề khác do việc sử dụng database riêng từng service là một vài câu truy vấn cần phải join dữ liệu thuộc nhiều service khcá nhau. Mỗi data của service chỉ có thể truy cập thông qua API của nó, bạn không thể sử dụng các query phân tán được. Như hình trên chính là một vài pattern bạn có thể dùng cho việc này.

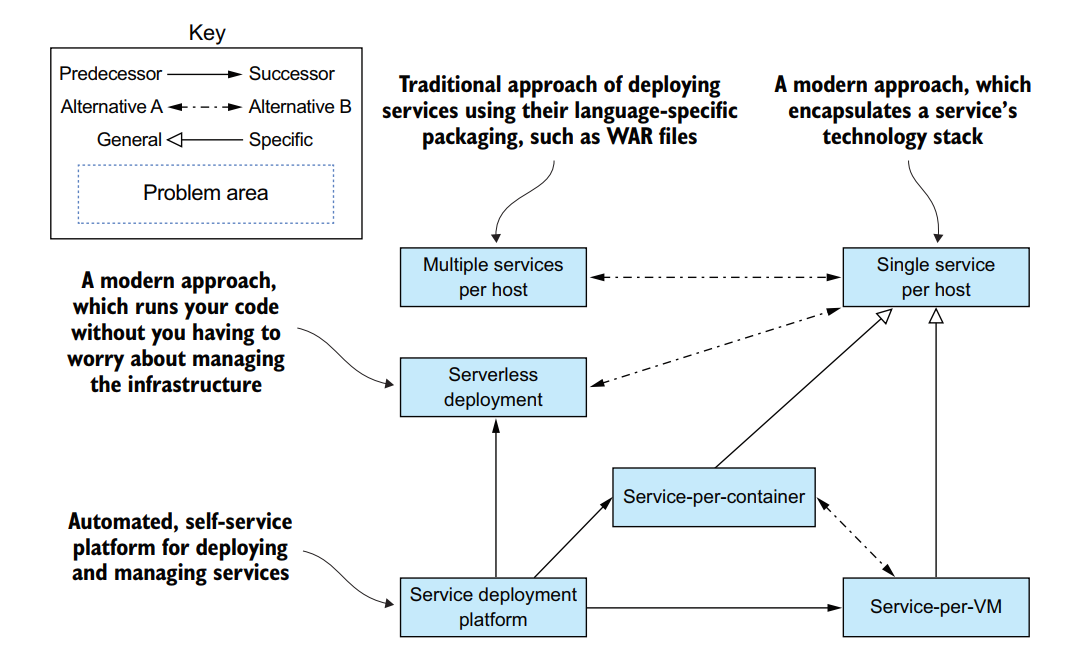
Thi thoảng bạn có thể sử dụng API composition pattern, nó sẽ gọi API của một hoặc nhiều dữ liệu và tập hợp kết quả. Hoặc có thể sử dụng CQRS pattern (Command query responsibility segregation),nó sẽ duy trì một hoặc nhiều bản sao query của dữ liệu.

## Service deployment pattern

Deploy một ứng dụng monolithic không phải lúc nào cũng dễ dàng, nhưng nó đơn giản theo nghĩa bạn chỉ có 1 ứng dụng cần deploy. Bạn phải chạy nhiều instance của nó đằng sau cân bằng tải.

Khi so sánh với kiến trúc micro, thì việc deploy ứng dụng sử dụng kiến trúc micro phức tạp hơn nhiều. Có thể có tới 10 hoặc 100 service dưới nhiều ngôn ngữ lập trình khác nhau, Nhiều phần bạn phải quản lý.

Theo truyền thống, và thường thủ công, cách deploy ứng dụng đóng gói trong mỗi ngôn ngữ cụ thể. Bạn cần một kiến trúc deploy tự động cao. Tốt nhất, bạn nên sử dụng một nền tảng triển khai cung cấp cho các nhà phát triển một giao diễn dẽ dàng cho việc deploy và quản lý service. Nền tảng deploy thường dựa trên công nghệ máy ảo, container hoặc serverless.



## Observability pattern cung cấp quản sát về hành vi của ứng dụng

Một phần quan trọng của hoạt động ứng dụng là hiểu được các hành vi tại thời điểm runtime và các vấn đề mắc phải như request thất bại hay độ trễ cao. Mặc dùng hiểu và xử lý sự cố ứng dụng monolithic không phải luôn luôn dễ dàng, nó giúp các request được xử lý đơn giản. Mỗi request tới được cân bằng tải tới một instance ứng dụng cụ thể, tạo một số goij tới database và trả về response. Ví dụ, bạn cần hiểu cách request cụ thể được xử lý, bạn nhìn vào log file của instance ứng dụng mà xử lý request đó.

Ngược lại, việc biết và chuẩn đoán vấn đề trong kiến trúc micro phức tạp hơn nhiều. Một request có thể được lăn xung quanh nhiều service trước khi có kết quả trả về client. Do đó, sẽ không có duy nhất một log file nào để kiểm tra. Tương tự, các vấn đề với trễ còn khó hơn để chuẩn đoán bởi vì có rất nhiều nghi phạm.

Một số pattern hay thuộc nhóm này

* Health check API: kiểm tra tình trạng sức khỏe của service
* Log aggregation: Log các hành động service và ghi lại log vào một server log chung, cung cấp việc tìm kiếm và cảnh báo
* Distributed tracing: Gán cho một request ID duy nhất và đánh vết request dọc các luồng service.
* Exception tracking: báo cáo exception tới một service phụ trách tracking exception, cảnh báo đội phát triển.
* Application metrics: Duy trì độ đo, như là các số liệu.
* Audit loggind: Log lại hành động của user.

## Pattern cho việc tự động test các service

Việc test các service sẽ dễ dàng khi sử dụng kiến trúc micro do chúng nhỏ hơn cả một ứng dụng monolithic. Một số pattern sau:

* Consumer-driven contract test: Xác minh rằng một service đã đáp ứng mong đợi của client
* Consumer-side contract test: Xác minh xem client của servie có thể giao tiếp với service
* Service component test: Test service độc lập

## Pattern cho xử lý Cross-cutting concerns

Trong kiến trúc micro, có nhiều mối quan tâm mà mọi service phải triển khai, bao gồm các pattern observability và discovery. Chúng cũng phải triển khai Externalized Configuration pattern, cung cấp các tham số cấu hình. Khi phát triển 1 service mới, tốn nhiều thời gian cho những thứ đó lại từ đầu. Một cách tiếp cận tốt đó là việc áp dụng Chassis pattern và xây dựng service trên framework có thể xử lý các vấn đề đó.

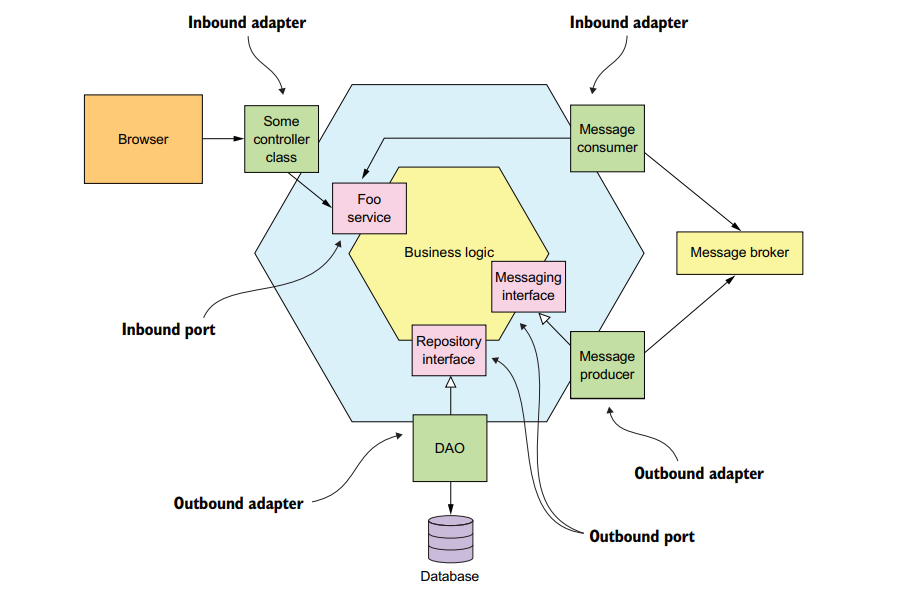
## Security pattern

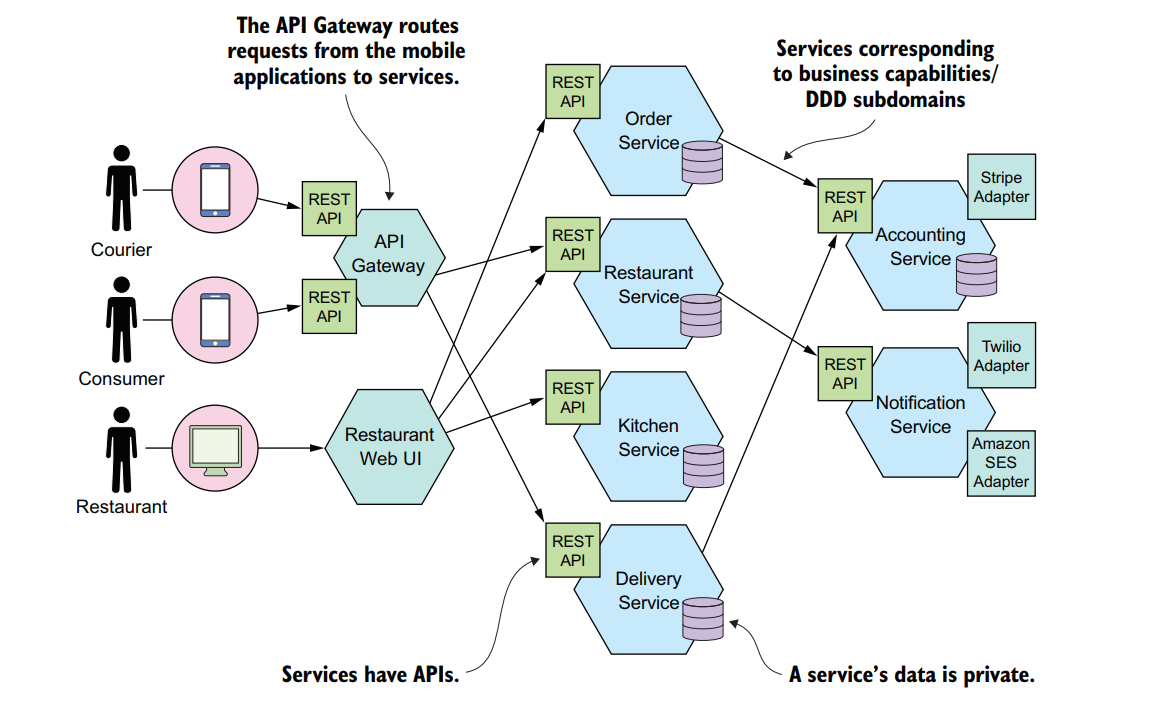
Trong kiến trúc micro, user thường được xác thực qua API gateway, truyền thông tin về user, như là định danh hay quyền tới service nó gọi. Một giải pháp thường thấy là sử dụng Access token pattern. API gateway truyền một access token như JWT tới service để xác minh token và lấy thông tin về user.

# Chiến thuật phân tách (Decomposition strategies)

Kiến trúc Microservice là một kiểu architectural style. Nó cấu trúc implement view như là tập cácc thành phần : executables hay WAR files. Các component là các service, và các connector là các giao thức trao đổi cho phép chúng cộng tác. Mỗi service lại có logical view architecture của nó, là một kiến trúc hexagonal.

Ví dụ:





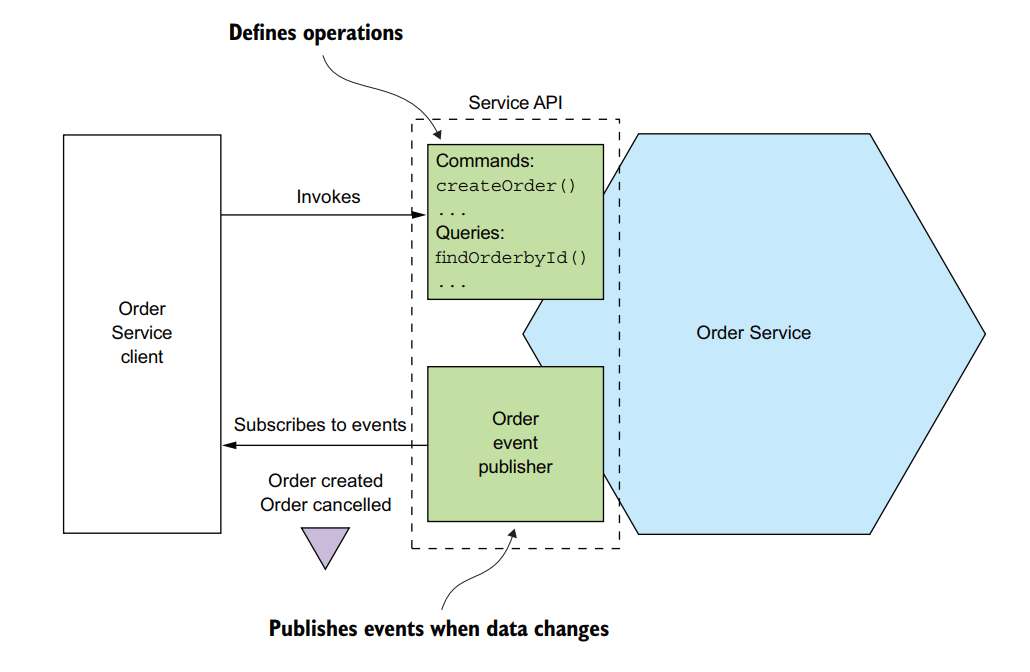
Vấn đề quan trọng chính là các service trong kiến trúc này phụ thuộc lỏng lẻo (1 tính chất trong SOLID). Do đó sẽ có các giới hạn cách chúng làm việc với nhau. Vậy service là gì

## Service là gì?

Service là một thành phần độc lập, thành phần phần mềm độc lập triển khai, triển khai các chức năng hữu ích. Service có API cung cấp truy cập của client tới các chức năng. Có 2 kiểu hành vi chính: command và querry. API bao gồm command, query và event. Command như là vieẹc createOrder() thực thi hành động cập nhất dữ liệu. Query như findOrderById() dùng để lấy dữ liệu. Service cũng publish các event như OrderCreated, sẽ được tiêu thụ bởi client của chúng.

API của service đóng gói việc thực thi nội bộ. Không giốn monolith, dev không thể viết code mà bỏ qua API của nó. Điều này khiến ch kiến trúc đảm bảo tính module hóa.

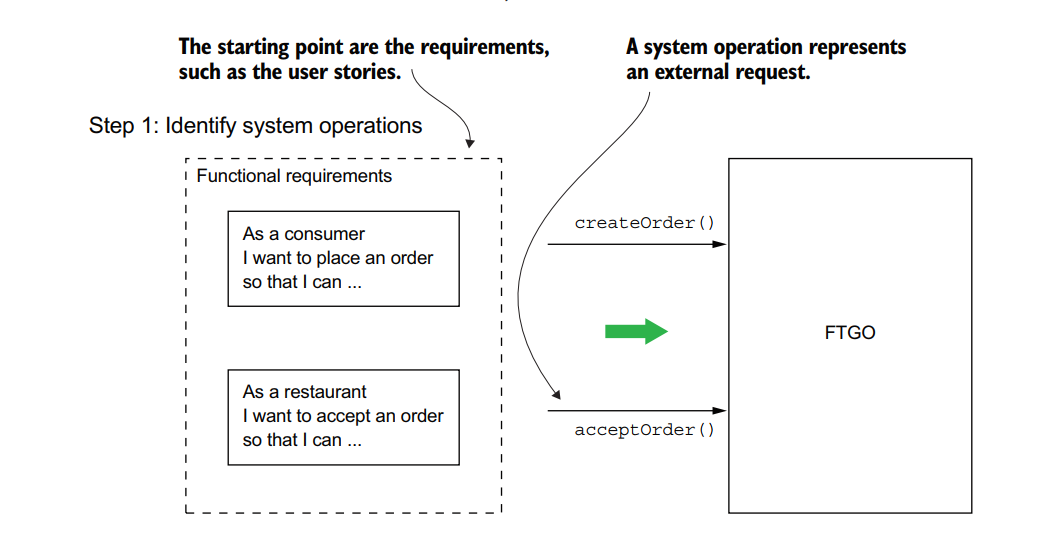
Mỗi service trong kiến trúc cũng có kiến trúc của nó, technology stack. Nhưng một service là một kiến trúc hexagonal. API của nó được triển khai bởi các apdapter, các apdater tương tcá với business logic của service.Các hoạt động adapter gọi business logic, và các event adapter được publish bởi business logic.

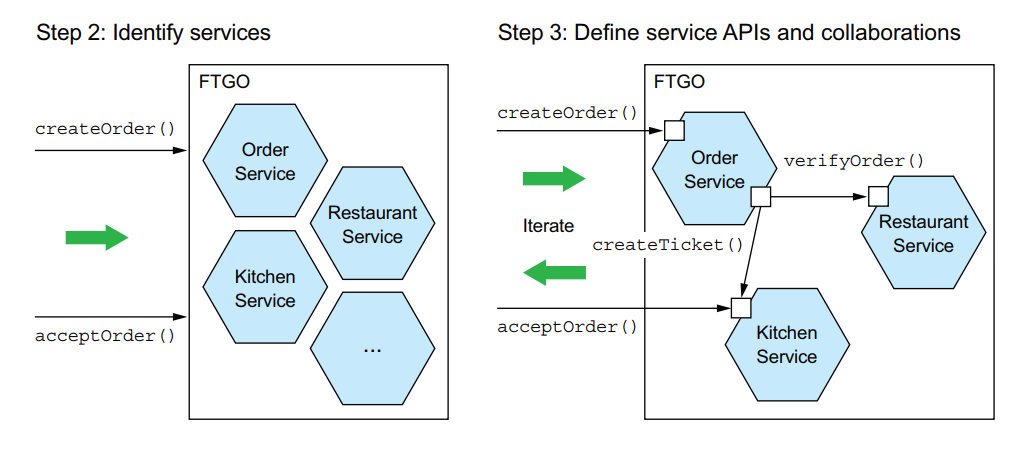


## Định nghĩa kiến trúc micro cho ứng dụng

3 step:

* Xác định các hoạt động hệ thống
* Xác định các service
* Định nghĩa các API service và cộng tác service





Ứng dụng tồn tại để xử lý các request, do vậy, bước đầu tiên chúng ta cần là cô đọng các yêu cầu của ứng dụng vào các yêu cầu key. Nhưng thay cho việc miêu tả các yêu cầu theo các thuật ngữa cụ thể về công nghệ như REST hay message, ta sẽ sử dụng các kí pháp trừu tượng của hệ thống. Một hoạt động hệ thống ( system operation) là một trừu tượng của một yêu cầu mà ứng dụng phải xử lý. Nó có thể là một command, cập nhật dữ liệu, hay một query lấy dữ liệu. Hành vi của mỗi command được định nghĩa trong một mô hình miền trừu tường (abstract domain model), có nguồn gốc từ các yêu cầu. Các hoạt động hệ thống trở thành các kịch bản kiến trúc minh họa cách các dịch vụ cộng tác.

Bước 2 là đi xác định việc phân nhỏ chúng vào các service. Có một vài chiến thuật bạn có thể sử dụng. Một trong số đó có nguồn gốc từ việc định nghĩa các service trùng với các cấu rtúc business.Một chiến thuật khác là tổ chức các service xung quanh domain driven design sub domains Kết quả thu được là các service được tổ chức xung quanh business concept hơn là technical concepts.

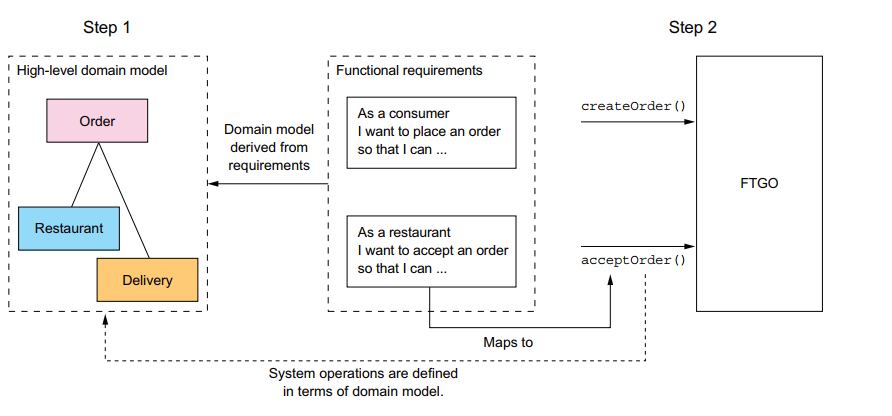
Bước 3 là đi xác định các API của mỗi service. Để làm điều đó, bạn chỉ định mỗi hoạt động hệ thống được xác định trong bước 1 vào service. Một service có thể thực hiện một hoạt động hoàn toàn của chính nó. Ngoài ra, nó có thể cần cộng tác với service khác. Trong trường hợp đó, bạn xác định cách service chúng cộng tác với nhau , nó thường đòi hỏi các service hỗ trợ các hoạt động bổ sung. Bạn sẽ cũng cần quyết định cơ chế giao tiếp liên tiến trình (IPC) để cụ thể hóa mỗi API service.

Có rất nhiều chướng ngại vật khi phân tách. Đầu tiên phải kể đến độ trễ mạng. Bạn phải biết rằng một khi phân tách cụ thể sẽ không áp dụng thực tế do có quá nhiều round-trips giữa các service. Một chướng ngại vật khác chính là giao tiếp đồng bộ giữa các service làm giảm tính sẵn sàng.Thứ 3 phải nói đến là yêu cấu duy trì tính nhất quán dữ liệu dọc các service. Bạn nên sử dụng saga. Thứ tư và cũng là cuối cùng được gọi là gold classes, đuọc sử dụng trong cả ứng dụng.May mắn thay, bạn có thể sử dụng các khái niệm từ DDD (Domain driven design) để loại bỏ các gold class.

Phần đầu tiên tiếp theo sau đây sẽ miêu tả các định danh các hoạt động của ứng dụng. Sau đó, chúng ta sẽ xem xét các chiến lược và hướng dẫn để phân tách ứng dụng ra các service, và giải quyết các chướng ngại vật khi phân tách. Cuối cùng, ta sẽ bàn vèe cách định nghĩa các API của service.

## Xác định các hoạt động của hệ thống

Bước đầu tiên trong việc định nghĩa kiến trúc ứng dụng là xác định các hoạt động của ứng dụng. Điểm khởi đầu của các yêu cầu ứng dụng, bao gồm các user story và các mối liên hệ user scenarios. Hoạt động hệ thống được xấc định và định nghĩa sử dụng tiến trình hai bước như hình sau. Bước một ta tạo ra high-level domain model bao gồm các class chính cung cấp các vốn từ để miêu tả các hoạt động hệ thống. Bước hai là xác định các hoạt động và miêu tả hành vi của chúng theo khái niệm của domain model.

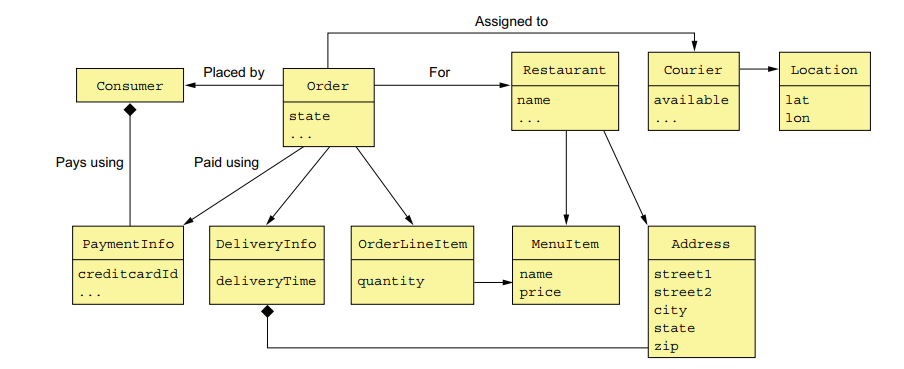


Domain model có nguồn gốc từ các danh từ của user story, và các hoạt động hệ thống có nguoòn gốc từ các động từ. Bạn có thể định nghĩa domain model sử dụng kĩ thuật Event Storming. Hành vi của mỗi hoạt động được miêu tả trong ảnh hưởng của nó lên một hay nhiều domain object và mối quan hệ giữa chúng. Một hoạt động hệ thống có thể tạo,cập nhật, xóa domain object, cũng như tạo và phá hủy mối quan hệ giữa chúng.

### Tạo ra high-level domain model

Bước đầu tiên trong quá trình định nghĩa hoạt động hệ thống chính là tạo ra high-level domain model cho ứng dụng. Domain model này sẽ đơn giản hơn thứ chúng ta muốn triển khai. Ứng dụng thậm chí sẽ không có một domain model duy nhất bởi vì mỗi service sẽ có domain model riêng của nó. High-level domain model hữuu ích trong giai đoạn này bởi vì nó định nghĩa ra các từ vựng cho việc miêu tả hành vi hoạt động hệ thống.

Domain model được tạo sử dụng kĩ thuật tiêu chuẩn như là phân tích danh từ trong story và scenarios và nói chuyện với chuyên gia. Ví dụ, với Place Order story, ta có thể mở rộng story thành nhiều user scenarios



Nhìn trên , ta sẽ thấy, mỗi class sẽ có trách nhiệm của nó

* Consumer: Là một người tiêu thụ, đặt hàng
* Order: Miêu tả order và theo dõi trạng thái của order đó

….

### Xác định các hoạt động hệ thống

Một khi bạn đã xác định được high-level domain model, bước tiếp theo là ta xác định các request mà ứng dụng phải xử lý. Có hai loại:

* Command: Các hoạt động hệ thống liên quan tới việc tạo, sửa, xóa dữ liệu
* Queries:Các hoạt động hệ thống được dữ liệu

Khi định nghĩa các system command, ta nên phân tích các động từ trong user story và scenarios. Ví dụ với Place Order story, nghe tới tên ta cũng biết sẽ có một hoạt động là Create Order.Ví dụ:

|  |  |  |  |
| --- | --- | --- | --- |
| Actor | Story | Command | Description |
| Consumer | Create Order | CreateOrder() | Tạo ra một order |
| Restaurant | Accept Order | AcceptOrder() | Chỉ ra rằng nhà hàng chấp nhận order đó và cam kết rằng họ sẽ chuẩn bị order đó theo thời gian đã chỉ định |
| Restaurant | Order Ready for Pickup | NoteOrderReadyForPickup() | Chỉ ra rằng Order đã sẵn sàng nhận |
| Courier | Update Location | NoteUpdatedLocation() | Cập nhật vị trí hiện tại của người giao hàng |
| Courier | Delivery picked up | NoteDeliveryPickedUp() | Chi ra rằng người giao hàng đã nhận order |
| Courier | Delivery delivered | NoteDeliveryDelivered() | Người giao hàng đã hoàn thành việc giao hàng |

Một command có các đặc tả chỉ ra các tham số và trả về, và các hành vi của domain model class. Đặc tả hành vi bao gồm : nhiều kiện đầu vào, điều kiện đầu ra. Ví dụ:

|  |  |
| --- | --- |
| Operation | CreateOrder(consumer id,payment method,delivery address, delivery time, restaurant id, order line items) |
| Return | OrderId,…. |
| Pre-condition | Khách hàng tồn tại và có thể đặt hàng  Mỗi sản phẩm phải trùng với các sản phẩm trong menu của nhà hàng  Địa chỉ giao hàng và thời gian có thể được phục vụ bởi nhà hàng |
| Post-condition | Thẻ credit của khách hàng được cấp quyền để trả hóa đơn  Order được tạo trong trạng thái PENDING\_ACCEPTANCE |

|  |  |
| --- | --- |
| Operation | AcceptOrder(restaurantId, orderId, readyByTime) |
| Return |  |
| Pre-condition | Trạng thái của order là PENDING\_ACCEPTANCE  Có người giao hàng đang rảnh |
| Post-condition | Trạng thái của order được thay đổi thành ACCEPTED  Order.readyByTime được thay đổi thành giá trị readyByTime  Người giao hàng được giao việc đi giao hàng |

Hầu hết các hoạt động của hệ thống là command, thi thoảng là query, nhằm lấy dữ liệu, cũng rất quan trọng.VD:

* findAvailableRestaurants(deliveryAddress, deliveryTime)- Tìm các nhà hàng có thể giao tới các địa chỉ cụ thể vào thời điểm cụ thể.
* findRestaurantMenu(id)-Lấy thông tin về nhà hàng bao gồm các món trong menu

Một khi các hoạt động hệ thống được xác dịnh, bước tiếp theo ta cần làm là định nghĩa ra các service. Có rất nhiều chiến thuật phân tách bạn có thể sử dụng. Mỗi chúng tập trung tấn công vấn đề từ góc nhièn khác nhau và sử dụng các thuật ngữ riêng. Nhưng với tất cả các chiến thuật, kết quả cuối cùng vẫn như nhau: một kiến trúc bao gồm tập service chủ yếu được tổ chức xung quanh khái niệm business hơn là khái niệm công nghệ.

## Định nghĩa service bằng cách áp dụng pattern phân tách theo business capability

Một chiến thuật cho việc tạo ra kiến trúc micro chính là phân tách theo khả năng kinh doanh (business capacibility). Khái niệm business capability là một thứ gì mà một business làm để tạo ra giá trị.Tập các khả năng cho một business phụ thuộc vào loại hình của business. Ví dụ, các khả năng của công ty bảo hiểm bao gồm quản lý bảo hiểm, bồi thường, hóa đơn.

### Khả năng kinh doanh định nghĩa tổ chức làm gì

Khả năng kinh doanh của tổ chức “chụp” lại kinh doanh của tổ chức là gì. Nhìn chung, chúng ổn định, trái ngược với cách tổ chức hoạt động kinh doanh, chúng thay đổi liên tục. Đó là đặc trưng ngày nay, với sự phát triển nhanh chóng của công nghệ để tự động hóa cá quá trình kinh doanh. Ví dụ chính là việc sử dụng cây ATM thay cho việc ta phải ra tận ngân hàng để xử lý với nhân viên ngân hàng.

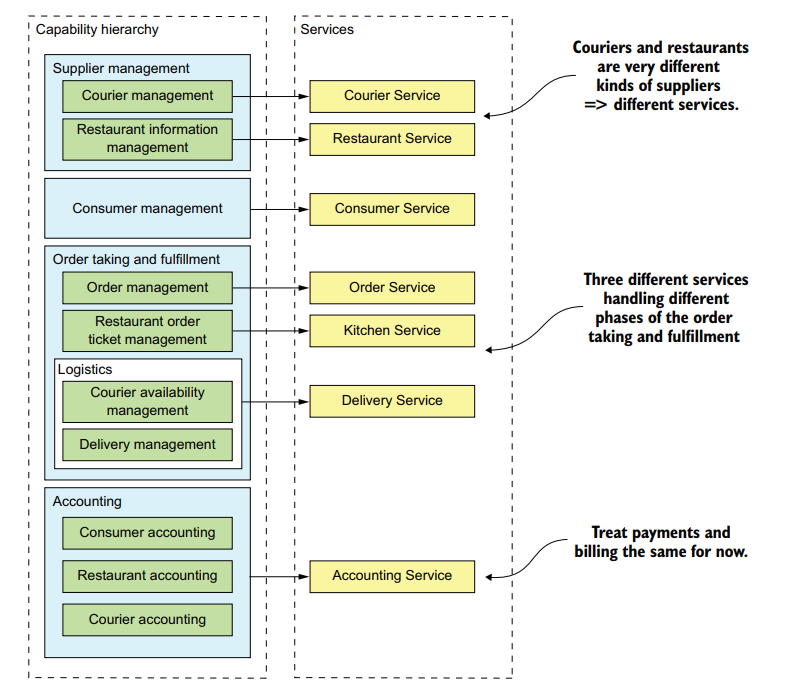
### Định nghĩa các khả năng kinh doanh

Khả năng kinh doanh được định nghĩa thông qua việc phân tích mục đích,cơ cấu tổ chức và quá trình kinh doanh của tổ chức đó. Mỗi khả năng kinh doanh có thể được coi như một service, ngoại trừ nó hướng kinh doanh hơn hướng kĩ thuật. Đặc điểm của nó bao gồm các thành phần, đầu vào, đầu ra…. Ví dụ: đầu vào của khả năng kinh doanh bảo hiểm là ứng dụng của người dùng, và các đầu ra bao gồm sự phê duyệt và giá.

Một khả năng kinh doanh thường tập trung vào một đối tượng kinh doanh cụ thể.VÍ dụ: đối tượng kinh doanh bồi thường tập trung vào khả năng quản lý bồi thường. Một khả năng thường có thẻ được phân chia thành các khả năng con. VÍ dụ, khả năng quản lý bồi thường có một vài khả năng con, bao gồm quản lý thông tin bồi thường, duyệt bồi thường, quản lý thanh toán bồi thường.

### Từ khả năng kinh doanh tới các service

Một khi đã hoành thành việc định nghĩa khả năng kinh doanh, bạn sau đó định nghĩa các service cho mỗi khả năng kinh doanh của tập các nhóm kinh doanh có liên quan. VD:



Lợi ích chính của việc tổ chức các service xung quanh các khả năng là bởi vì chúng ổn định, kết quả của kiến trúc sẽ ổn định lâu dài. Các thành phần riêng lẻ của kiến trúc có thể phát triển như cách các khía cạnh của kinh doanh thay đổi, nhưng kiến trúc vẫn không thay đổi.

Ngoài ra còn có chiến thuật phân tách theo miền con, các bạn nên tìm hiểu đọc nó.

## Hướng dẫn phân tách

Chúng ta có thể sử dụng một vài nguyên tắc trong thiết kế hướng đối tượng khi áp dụng pattern kiến trúc micro. Đầu tiên chính là Single Responsibility Principle (SRP), cho việc dịnh nghĩa trách nhiệm một lớp, thứ hai là Common Closure Principle (CCP) cho việc tổ chức lớp vào các gói.

### Single Reponsibility Principle

Một trong những mực tiêu chính của thiết và kiến trúc phần mềm là xác định trách nhiệm của mỗi thành phần phần mềm. Khái niệm SRP được viết như sau

*A class should have only one reason to change*

Robert C.Martin

Mỗi trách nhiệm mà một class có một lý dó tiểm ẩn cho việc thay đổi. Nếu clas có nhiều trách nhiệm thay đổi độc lập, class đó sẽ không ổn định. Bằng cách theo SRP, bạn định nghĩa các class mà mỗi trong chúng chỉ có một trách nhiệm và đó là lý do duy nhất để thay đổi.

Chúng ta có thể áp dụng SRP khi định nghĩa kiến trúc micro và tạo ra các service liên kết, nhỏ mà mỗi trong số chúng có một trách nhiệm. Nó sẽ giảm kích thước của service và tang tính ổn định.

### Common closure principle

*The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package*

Robert C.Martin

Ý tưởng ở đây nếu hai class thay đổi theo sát bước bước bởi vì cùng một lý do, thì chúng sẽ thuộc về cùng q package. Có lẽ, ví dụ, những class triển khai một khía cạnh khác nhau của một quy tắc kinh doanh cụ thể. Mục đích là khi quy tắc kinh doanh thay đổi, dev chỉ cần thay đổi code ở ít package nhất.

Chúng ta có thể áp dụng CCP khi tạo ra kiến trúc micro và đóng gói các thành phần thay đổi cùng lý do vào cung 1 service. Điều này sẽ giúp giảm số lượng service cần thay đổi và deploy khi một vài yêu cầu thay đổi. Việc thay đổi cũng sẽ chỉ ảnh ưởng tới 1 đội và 1 service.

## Chướng ngại vật trong việc phân tách ứng dụng ra các service

Mặt khác, các chiến lược của việc tạo ra kiến trúc micro bằng cách định nghĩa các service tương ứng với khả năng kinh doanh hoặc miền con có một vài trở ngại:

* Trễ mạng
* Giảm tính sẵn có do giao tiếp đồng bộ
* Duy trì tính nhất quán dữ liệu dọc các service
* Đạt được cái nhìn nhất quán của dữ liệu
* God class chặn việc phân tách

### Trễ mạng

Trễ mạng là mối quan tâm luôn hiện diện trong hệ thống phân tán.Thi thoảng, bạn có thể giảm độ trễ tới mức chấp nhận được bằng cách triển khai batch API cho việc lấy nhiều đối tượng trong một round trip. Nhưng trong các tình huống khách, giải pháp là kết hợp các servce, thay thế IPC bởi gọi phương thức hoặc gọi chức năng

### Giảm tính sẵn có do giao tiếp đồng bộ

Vấn đề khác là làm sao để thực hiện giao tiếp liên service theo cách mà không giảm tính sẵn sàng. Ví dụ, một cấch đơn giản nhất để triển khai hoạt động createOrder() là dành cho Order service để đồng bộ gọi service khác sử dụng REST. Hạn chế của việc sử dụng giao thức như REST là giảm sự sẵn có của Order service. Nó sẽ không thể tạo ra order nếu không có các service khác.

### Duy trì tính nhất quán dữ liệu dọc các service

Một thách thức khác là việc duy trì tính nhất quán dữ liệu dọc các service. Một vài hoạt động của hệ thống cần cập nhật dữ liệu trong nhiều service.Ví dụ, khi nhà hàng chấp nhận một order,cập nhật sẽ xuất hiện ở cả Kitchen Service và Delivery Service. Kitchen Service thay đổi trạng thái của Ticket, Delivery Service lập lịch giao hàng. Cả hai phải cập nhật thành công nhanh nhất có thể.

Các giải pháp truyền thống là sử dụng một cơ chế quản lý distributed transaction gồm 2 giai đoạn, dựa trên cam kết. Trong phầu sau, tôi sẽ nói về saga. Saga là một chuỗi của các local transaction được định địa chỉ sử dụng messaging. Saga phức tạp hơn ACID transactions truyền thống nhưng chúng làm việc tốt trong nhiều tình huống.

### Đạt được cái nhìn nhất quán của dữ liệu

Trong ứng dụng monolithic, các tính chất của ACID transaction đảm bảo rằng query sẽ trả lại một cái nhìn nhất quán của database. Ngược lại, trong kiến trúc micro, mặc dù mỗi database của service là nhất quán, bạn không thể đạt được một cái nhìn nhất quán tổng quát của dữ liệu.Nếu b ạn cần một cái nhìn nhất quán của một vài dữ liệu, thì nó phải nằm trong một serive duy nhất, để ngăn chặn việc phân tách cạnh tranh.

### God class chặn việc phân tách

God class là những lớp cồng kềnh được sử dụng trong suốt ứng dụng. Một god class thường triển khai logic kinh doanh cho nhiều khía cạnh khách nhau của ứng dụng. Nó thường có rất nhiều trường được ánh xạ vào bảng trong cơ sở dữ liệu với nhiều cột. Hầu hét các ứng dụng sẽ có một trong các class này, mỗi trong số chúng đại diện cho một khái niệm nó là trung tâm của domain: accounts trong banking, order trong e-commerce, policy trong bảo hiểm. Bởi vì god class bó trạng thái và hành vì với nhau cho nhiều khía cạnh khác nhau của ứng dụng, nó là một trở ngại không thể vượt qua trong việc chia nhỏ bất cứ bunisness logic thành các service.

Một giải pháp là đóng gói god class vào một thư viện và tạo ra một Order database trung tâm. Tất cả service xử lý order sử dụng thư viện đó và truy cập vào database. Vấn đề với giải pháp này là nó vi phạm một nguyên tắc quan trọng của kiến trúc micro và kết quả không mong muốn, liên kết chặt. Ví dụ, bất cứ thay đổi nào tới Order schema yêu cầu các đội phải cập nhật code.

Một giải pháp khác là đóng gói Order databsse trong Order service, sẽ được gọi bởi các service khác để lấy và cập nhật dữ liệu order. Vấn đề với thiết kế này đó là Order service sẽ là một dịch vụ dữ liệu với một domain model thiếu máu chứa ít hoặc không có business logic. May thay, DDD cung cấp một giải pháp.

Một cách tiếp cận tốt hơn là áp dụng DĐ và đối xử mỗi service như là một sub-domain riêng rẽ với domain model của chúng. Nghĩa là mỗi service trong ứng dụng mà có bất cứ thứ gì cần làm với god class sẽ có domain model riêng của nó với một phiên bản của god class.

## Định nghĩa ra các API cho service

Bước tiếp theo sẽ là định nghĩa các API của từng service: Các hoạt động và sự kiện của nó. Một hoạt động API service tồn tại cho một hoặc 2 lý do: một vài hoạt động tương ứng với hoạt động hệ thống. Chúng có thể được gọi bởi client ngoại và có thẻ là bởi các service khác. Các hoạt động khác tồn tại để hỗ trợ cộng tác giữa các service. Những hoạt động này chỉ được gọi bởi các service khác.

Service publish các sự kiện chính cho phpé nó phối hợp với các service khác. Ứng dụng có thể sử dụng event để thông báo với các client ngoại, nó có thể là Websocket.

Điểm bắt đầu cho việc định nghĩa này là việc ánh xạ mỗi hoạt động hệ thống tới một dịch vụ. Sau đó, chúng ta quyết định xem service nào cần cộng tác với service khác để triển khai hoạt động hệ thống. Nếu cộng tác cần thiết, chúng ta cần xác định API nào mà service đó cần cung cấp để hỗ trợ việc cộng tác.Đầu tiên hay nhìn vào cách gán hoạt động hệ thống tới service.

### Gán hoạt động hệ thống vào service

Bước đầu tiên cần quyết định service nào là điểm khởi đầu cho một request.Nhiều hoạt động hệ thống map tới một service, nhưng thi thoảng việc ánh xạ là sẽ không rõ ràng. Sau khi đã gán hết các hoạt động vào service, bước tiếp theo ta sẽ quyết định xem các service cộng tác để xử lý mỗi hoạt động hệ thống.

### Xác định API được yêu cầu để hỗ trợ công tác giữa các service

Thi thoảng một vài hoạt động của hệ thống đã được xử lý hoàn toàn bởi một service. Nhưng một vài hoạt động cần việc cộng tác giữa các service. Dữ liệu cần thiết xử lý một trong những yêu cầu này có thể, được rải rác xung quanh nhiều service. Ví dụ để thực hiện được createOrder, Order Service cần gọi các service sau để xác nhận pre-condition và post-condition đúng:

* Consumer Service: Xác minh rằng consumer có thể tạo ra một order và đạt được thông tin thanh toán
* Restaurant Service, Kitchen Service,..

# Giao tiếp liên tiến trình trong kiến trúc micro

Kiến trúc micro cấu trúc ứng dụng thành các tập service.Các service phải thường cộng tác để xử lý yêu cầu. Bởi vì các service instance chạy trên nhiều máy, chúng phải tương tác sử dụng IPC. Việc chọn cơ chế IPC là một quyết định kiến trúc quan trọng. Nó có thể tác động tới tính sẵn có của ứng dụng.

## Tổng quan về giao tiếp tiến trình trong kiến trúc micro

Có nhiều công nghệ bạn có thể chọn, cơ chế đồng bộ trên nền HTTP như REST hoặc gRPC, hay cơ chế bất đồng bộ dựa trên thông điệp như AMQP, STOMP. Trước khi đi vào chi tiết, chúng ta cùng xem một vài vấn đề khi thiết kế.

### Kiểu tương tác(phong cách tương tác)

Đầu tiên ta nên suy nghĩ về phong cách tương tác giữa service và client của nó trước khi lựa chọn cơ chế IPC. Điều này giúp bạn tập trung vào yêu cầu và tránh bị sa lầy trong chi tiết của công nghệ IPC cụ thể. Phong cách tương tác khá là đa dạng. Chúng được chia thành 2 nhóm:

* One-to-one: Mỗi request từ client được xử lý chính xác bởi 1 service
* One-to-many: Mỗi request được xử lý bởi nhiều service.

Hoặc ta cũng có thể chia chúng theo phương pháp giao tiếp đồng bộ hay bất đồng bộ:

* Synchronous
* Asynchronous

Ta có bảng sau:

|  |  |  |
| --- | --- | --- |
|  | One-to-one | One-to-many |
| Đồng bộ | Request/response | - |
| Bất đồng bộ | Bất đồng bộ request/response  Thông báo 1 hchiều | Publish/subscribe  Publish/async response |

### Định nghĩa các API trong kiến trúc micro

API hay giao diện là trung tâm trong phát triển phần mềm. Mỗi module trong ứng dụng có giao diện và nó định nghĩa tập các hoạt động mà client có thể gọi. Trong ứng dụng monolithic, giao diện được quy định sử dụng một ngôn ngữ lập trình như Java interface.Một Java interface cụ thể một tập các phương thức mà client có thể gọi. Việc triển khai các phương thức đó được ẩn đi khỏi client.

API và giao diện là phần quan trọng trong kiến trúc micro. APi service là một hợp đồng giữa service và client. API bao gồm các hoạt động, mà client có thể gọi, và các event, được publish bởi service. Một hoạt động có tên, tham số, kiểu trả về. Và mộ event có kiểu và tập các trường được publish lên message channel.

Bất kể cơ chế IPC nào bạn chọn, điều quan trọng là bạn phải xác định chính xác API sử dụng kiểu ngôn ngữ định nghĩa giao diện nào (interface definition language). Đầu tiên bạn nên viết ra các định nghĩa của giao diện, sau đó bạn review giao diện đó với dev client.Sau hai quá trình này, thì bạn mới nên viết các triển khai cho chúng.

### Sự phát triển API

API luôn thay đổi theo thời gian khi các tính năng mới được thêm vào, tính năng hiện tại bị thay đổi, và có lẽ các tính năng cũ bị loại bỏ. Trong ứng dụng mono thì điều này vô cùng đơn giản, nhưng khi ta sử dụng kiến trúc micro thì nó là một thách thức. Một client là một service, thường được phát triển bởi mộ đội khác. Client có thể thậm chí là một ứng dụng ngoài của tổ chức. Bạn thơngf không thể ép buộc tất cả client phải nâng cập theo chân bạn. Bởi vì ứng dụng hiện đại thường không bao giờ tắt để bảo trì, bạn sẽ thực hiện một nâng cập rolling cho dịch vụ của bạn, do đó, cả 2 phiên bản của service sẽ vẫn chạy đồng thời. Thông thường ta sẽ sử dụng Sematic version để đánh phiên bản.

### Cấu trúc định dạng thông điệp

Bản chất của IPC là việc trao đổi thông điệp. Thông điều thường chứa dữ liệu, và một quyết định thiết kế quan trọng là định dạng dữ liệu đó. Có hai nhóm chính của việc định dạng thông điệp: text và binary.

#### Thông điệp dạng text-based

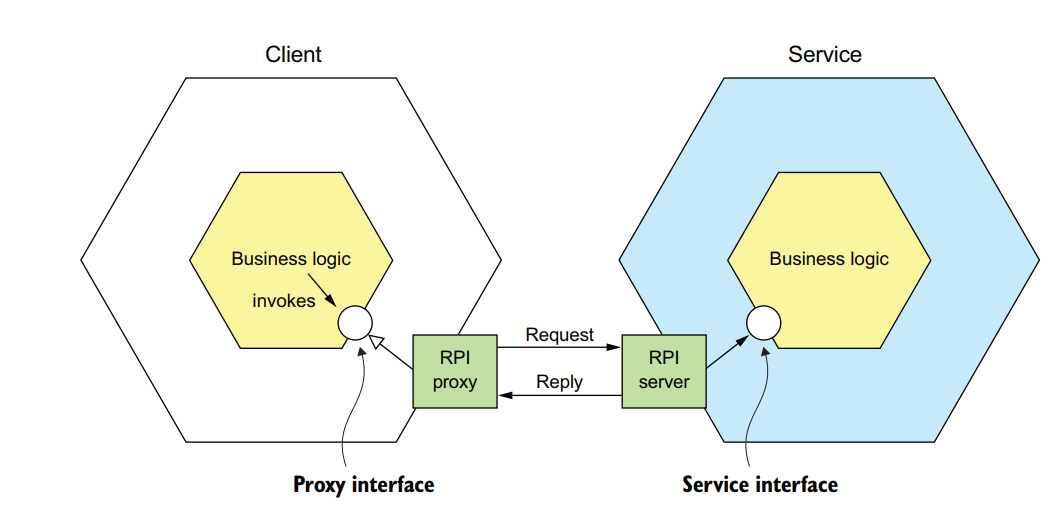
Như là JSON hay XML. Lợi ích của dạng này là nó dễ đọc. Dạng này cho phép consumer lấy thành phần cần thiết và bỏ lại thứ không liên quan.

#### Thông điệp dạng binary-based

Có một vài định dạng khá là phổ biến như Protocol Bufferss và Arvo. Cả 2 dạng đều cung cấp IDL type cho việc định nghĩa cấu trúc của message. Trình biên dịch sau đó sẽ sinh ra code để serialize và deserialize thông điệp.

## Giao tiếp sử dụng pattern gọi thủ tục từ xa đồng bộ

Khi sử dụng cơ chế này, client gửi một request tới service, và service xử lý request đó và gửi lại response. Một vài client sẽ block chờ response, và một vài khác sẽ có kiến trúc reactive, none-blocking. Nhưng không giống sử dụng thông điệp, client giả định rằng response sẽ trả về trong thời gian mong đợi.



Như hình trên ta thấy cách RPI làm việc. Business logic bên client gọi một proxy interface, được triển khai bởi RPI proxy adapter class. RPI proxy tạo một request tới service. Request được xử lý bởi RPI server adapter class, nó sẽ gọi business logic của service qua interface. Nó sau đó gửi lại phản hồi tới RPI proxy, rồi proxy này sẽ trả kết quả về client. Ngày nay ta có thể sử dụng REST hoặc gRPC để triển khai pattern này.

Sử dụng REST sẽ có những lợi ích và nhược điểm:

Lợi ích:

* Đơn giản quen thuộc
* Có thể test dễ dàng API với trình duyệt
* Hỗ trợ trực tiếp giao tiếp phong cách request/response
* Không đòi hỏi môi giới trung gian, đơn giản hóa kiến trúc hệ thống

Khó khăn:

* Chỉ hỗ trợ một kiểu request/response tỏng giao tiếp phong cách
* Giảm tính sẵn có.
* Client cần biết URL của service instance.
* Lấy nhiều tài nguyên trong một request là thách thức
* Khó khăn trong việc ánh xạ các hoạt động cập nhật tới HTTP

Sử dụng gRPC sẽ có những lợi ích và nhược điểm:

Lợi ích

* Đơn giản để thiết kế một API mà có tập hợp nhiều hoạt động cập nhật
* Cơ chế IPC hiệu quả, nhỏ gọn, đặc biệt khi trao đổi thông điệp lớn
* Cho phép khả năng tương tác giữa client và service được biết ở nhiều loại ngôn ngữ

Khó khăn:

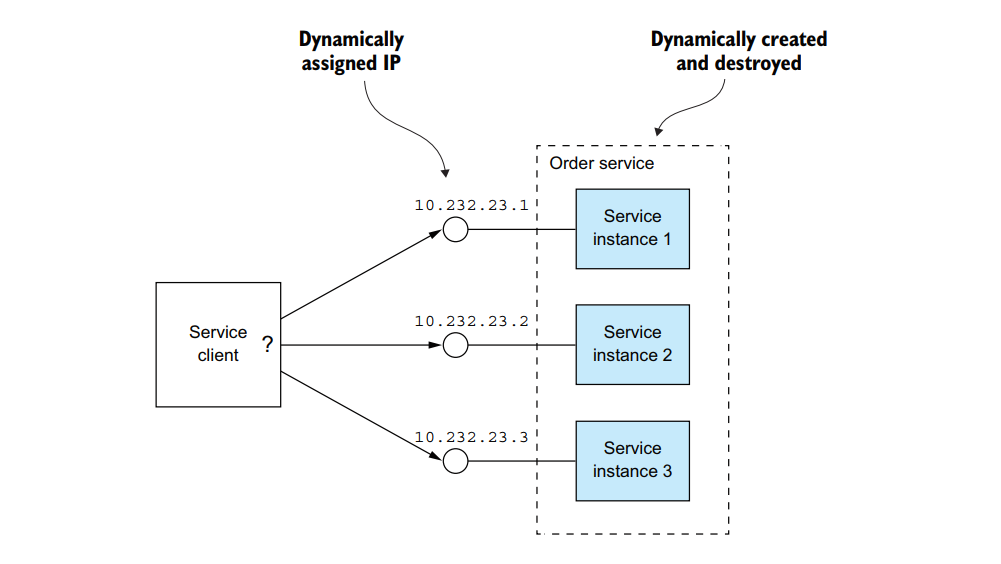
* Client dùng Javascript sẽ khó khăn để tương tác
* Tường lửa cũ có thể không hỗ trợ HTTP/2

### Xử lý lỗi sử dụng Circuit breaker pattern

Bất cứ khi nào một dịch vụ tạo ra một request đồng bộ tới service khác, luôn có một nguy cơ của sự thất bại một phần. Bởi vì client và server chạy độc lập, service có thể không đáp ứng kịp thời yêu cầu của client. Service này có thể bị lỗi hoặc do đang bảo trì hoặc bị quá tải và response chậm.

Pattern này thường sẽ xử lý bằng cách thêo gới số lượng các request thành công, thất bại, và nếu tỉ lệ thất bại vượt ngưỡng, các request gửi tương lai sẽ bị thất bại ngay lập tức.Sau một thời gian timeout, client thử một lần nữa, nếu thành công, đóng circuit breaker.

### Sử dụng service discovery

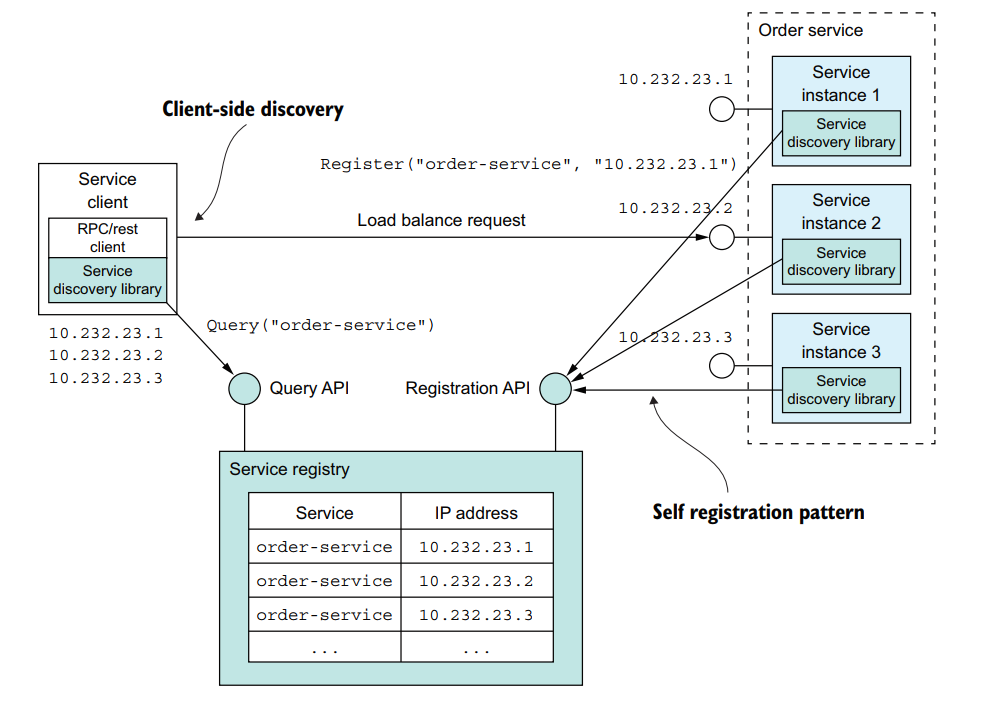


Chúng ta không thể cấu hình tĩnh client với các địa chỉ IP của service. Do đó, ứng dụng nên sử dụng một cơ chế service discovery động. Nó là một khái niệm đơn giản: Thành phần quan trọng của nó là service registry, là một cơ sở dữ liệu của các địa chỉ mạng của các service instance của ứng dụng. Nó có cơ chế cập nhật khi instance chạy, dừng. Khi client gọi service, service discovery tìm dữ liệu trong database để lấy được danh sách các instance có sẵn và điều hướng request tới một trong số chúng.Có hai cách chính để triển khai service discovery:

* Service và client tương tác trực tiếp với service registry
* Kiến trúc deployment xử lý service discovery.

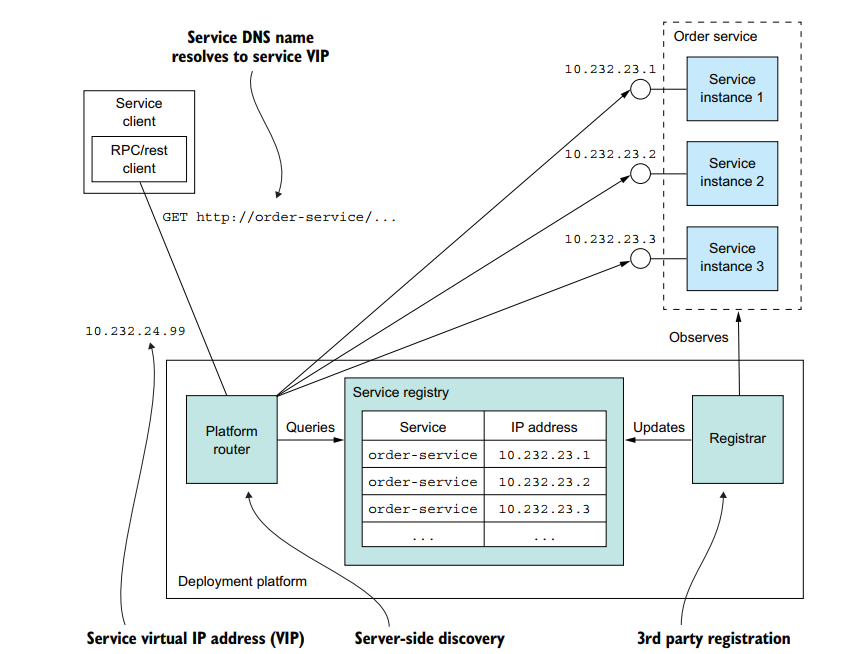
Áp dụng Service discovery pattern cấp ứng dụng.

Cách để triển khai service discovery là cho client và service tương tác với service registry như hình sau



Kết hợp bởi 2 pattern, đầu tiên là Self registration pattern. Service instance gọi API đăng kí của service registry để đăng kí vị trí mạng. Thứ hai là client-side discovery pattern. Khi service client muốn gọi một service, nó tìm kiếm trong service registry để lấy được danh sách các instances service. Để tăng hiệu năng, client có thẻ cache chúng. Client sau đó ử dụng giải thuật cân bằng tải như round-robin hay random để lựa chọn instance và gửi request tới đó.

Áp dụng service discovery pattern được cung cấp bởi nền tảng



Nhiều môi trường deployment như Docker và Kubernetes có sẵn service registry và cơ chế service discovery.Nền tảng deploy mnet cho mỗi service một DNS name, địa chỉ IP ảo và VIP address. Client tạo một request tới DNS name/VIP, và nền tảng deployment tự động điều hướng request tới một trong các instance service sẫn có.

## Giao tiếp sử dụng pattern thông điệp bất đồng bộ

Ứng dụng dựa trên thông điệp thường sử dụng message broker, hoạt động như một nhà trung gian giữa các service. Một client tạo ra một request tới service thông qua gửi một thông điệp. Nếu service instance trả lời, nó sẽ gửi lại một thông điệp cụ thể cho client. Do là bất đồng bộ, client không cần block để chờ phản hồi.

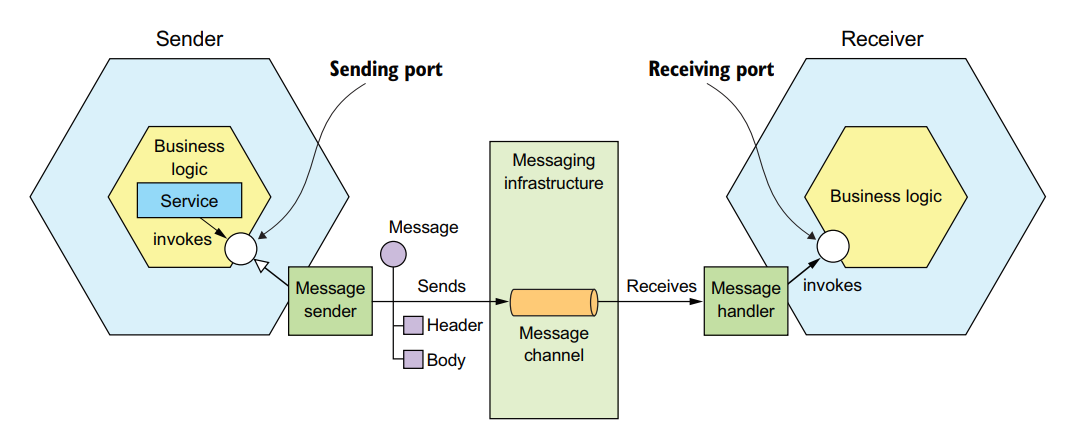
### Tổng quan về thông điệp

Thông điệp được trao thổi qua các message channel. Bên gửi viết thông điệp tới channel, và bên nhận sẽ đọc thông điệp từ channel.

Thông điệp bao gồm phần header và body.Phần header bao gồm các cặp name-value, metadata miêu tả dữ liệu được gửi. Nhằm thêm dữ liệu, header chứa một giá trị duy nhất message id được sinh ra bởi bên gửi hoặc kiến trúc message, và có thể có thêm return address. Phần body là phần dữ liệu được gửi, có thể là dạng text hoặc binary.Có một vài kiểu thông điệp:

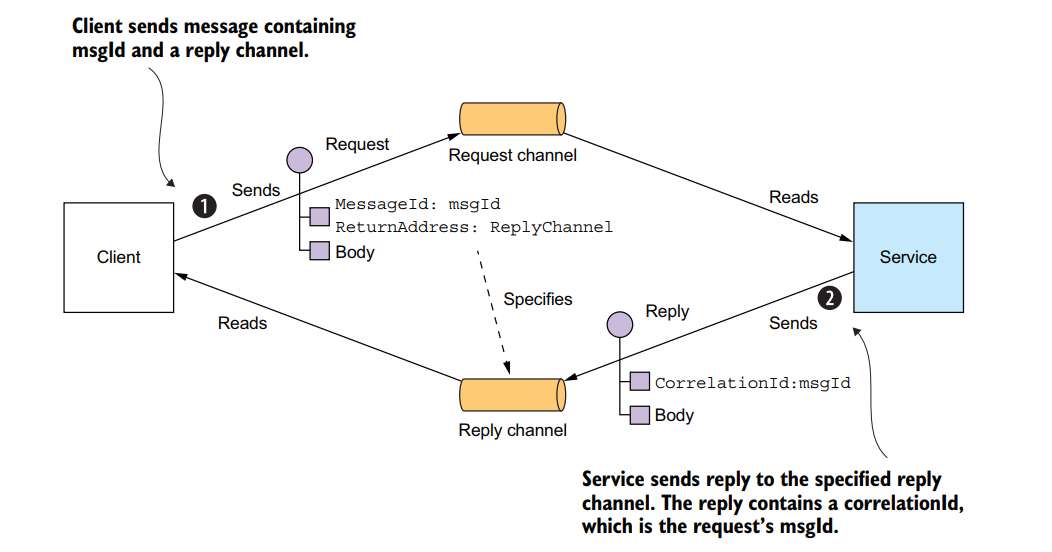
* Document: một thông điệp chỉ chứa dữ liệu . Bên nhận quyết địch cách hiểu nó. Câu trả lời cho một command là một ví dụ.
* Commad:Thông điệp tương đương với một RPC request. Nó cụ thể hoạt động gọi và tham số đi kèm
* Event: Một thông điệp chỉ ra rằng có gì đã xảy ra ở bên gửi. Một event thường là một domain event, đại diện cho một thay đổi trạng thái của một domain object như Order,Customer.

Như ta biết, thông điệp được trao đổi qua kênh (channel). Business logic bên sender gọi sending port interface, nó đóng gói cơ chế giao tiếp. Sending port được triển khai bởi một message sender adapter class, thứ sẽ gửi thông điệp tới bên nhận thông qua message channel. Message channel là một trừu tượng hóa của kiến trúc thông điệp. Message handler apdater class bên nhận được gọi để xử lý thông điệp. Nó gọi receiving port interface được triển khai bởi business logic của bên tiêu thụ. Channel có thể xử lý nhiều bên gửi và bên nhận.



Có 2 kiểu channel: point-to-point,publish-subscribe.Point-to-point chuyển một thông điệp tới chính xác một consumer mà đọc nó từ channel. Service sử dụng point to point channel cho tương tác one-to-one. Kiểu còn lại sẽ chuyển mỗi thông điệp tới tất cả các consumer được đính kèm. Service sử dụng publish-subscribe cho tương tác one-to-many.

### Triển khai sử dụng thông điệp



Khi client và service tương tác sử dụng cơ chế request/response hoặc asynchoronous request/response, client gửi một request và service sẽ phản hồii lại. Thông điệp là không đồng bộ, nó cng cấp asynchronous request/response. Nhưng client có thể block tới khi phản hồi trả về.

Client và service triển khai asynchronous request/response bằng cách trao đổi các cặp thông điệp. Như hình trên, client gửi một thông điệp command, cụ thể hoạt động thực hiện và các tham số, tới point-to-point channel. Service xử lý request và gửi lại thông điệpphản hồi chứa kết quả, tới point-to-point channel.

Client phải nói cho service nơi để gửi lại thông điệp phản hồi. Client sẽ gửi thông điệp mà chứa reploy channel trong header. Bên server viết phàn hổi chứa correlation id cùng giá trị với message indentifier, để phản hồi channel. Client sử dụng correlation id để khớp phản hồi với request.

#### Triển khai thông báo một chiều

Triển khai thông báo 1 chiều đơn giản sử dụng thông báo bất đồng bộ. Client gửi thông điệp, như là một thông điệp command, tới point-to-point channel. Service đăng kí channel và xử lý các thông điệp tới. Nó không gửi lại phản hồi

#### Triển khai publish/subscribe

Thông điệp hỗ trợ kiểu tương tác publish/subscribe. Client publish một thông điệp tới publish-subscribe channel, được đọc bởi nhiều consumer.

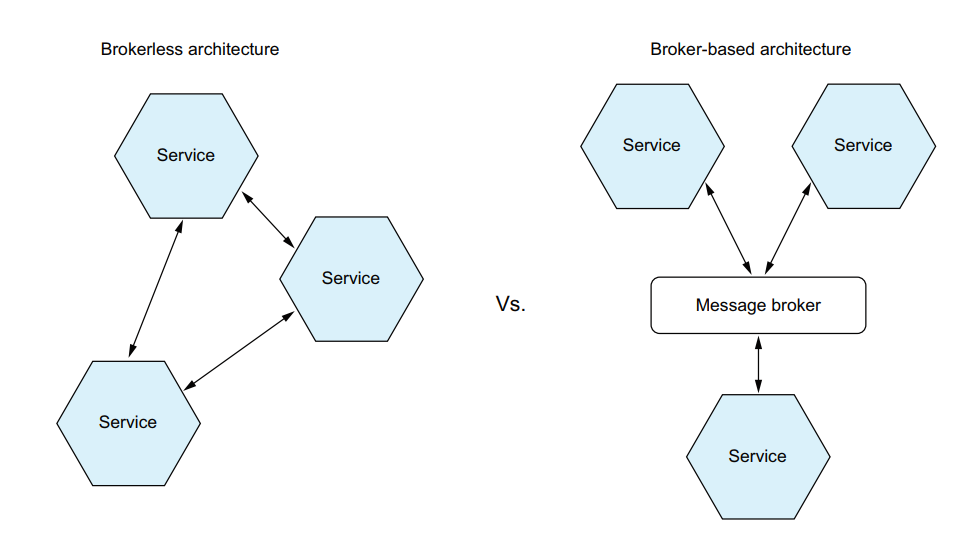
#### Triển khai publish/async response

Kiểu tương tác này là sự kết hợp của publish/subscribe và request/response. Client publish một thông điệp cụ thể reply channel trong header tới publish-subscibe channel. Consumer viết thông điệp phản hồi chứa correlation id tới reply channel.Client tập hợp các response sử dụng correlation id khớp thông điệp phản hồi với request.

Mỗi service trong ứng dụng sẽ có một API bất đồng bộ sẽ sử dụng một hoặc nhiều cách triển khai. Service có API bất đồng bộ cho việc gọi hoạt động sẽ có một message channel cho các request. Tương tự, service publish event sẽ publish chúng nên event message channel.

### Sử dụng message broker

Ứng dụng dựa trên thông điệp thường sử dụng một message broker, một service cơ sở hạ tầng thông qua đó các service giao tiếp với nhau. Kiến trúc dựa trên broker không là kiến trúc thông điệp duy nhất. Bạn có thể sử dụng brokerless-based, trong đó các service giao tiếp với nhau trực tiếp.



Trong chủ đề này, ta chỉ tập trung vào broker-based message.

#### Tổng quan về Broker-based messaging

Message broker là một trung gian qua đó các thông điệp chảy. Bên gửi viết thông điệp tới message broker, và message broker chuyển nó tới bên nhận. Một lợi ích khi sử dụng message broker là bên gửi không cần biết về vị trí mạng của bên tiêu thụ. Lợi ihcsh khác rằng message broker đệm thông điệp tới khi bên tiêu thụ có thể xử lý chúng.

Một số message broker open source:

* ActiveMQ
* RabbitMQ
* Apache Kafka

#### Triển khai message channel sử dụng message broker

Mỗi message broker triển khai message channel theo các cách khác nhau. Ta có bảng sau:

|  |  |  |
| --- | --- | --- |
| Message broker | Point-to-point channel | Publish-subscriber channel |
| JMS | Queue | Topic |
| Apache Kafka | Topic | Topic |
| AMQP-based brokers, như RabbitMQ | Exchange và Queue | Fnaout exchange và queue per consumer |
| AWS kinesis | Stream | Stream |
| AWSS SQS | Queue | - |

#### Lợi ích và nhược điểm của broker-based message

Có nhiều lợi ích khi dùng broker-based message

* Phụ thuộc lỏng lẻo (loose coupling): Client tạo request bằng cách đơn giản gửi một thông điệp tới channel thích hợp.
* Đệm thông điệp (message buffering): Đệm các thông điệp tới khi chúng có thể được xử lý.
* Giao tiếp linh hoạt
* Giao tiếp tiến trình rõ ràng

Ngoài ra, có một số nhược điểm:

* Tiểm ẩn nút thắt hiệu năng.
* Tiểm ẩn lỗi điểm duy nhất.
* Phức tạp một số hành động bổ sung.

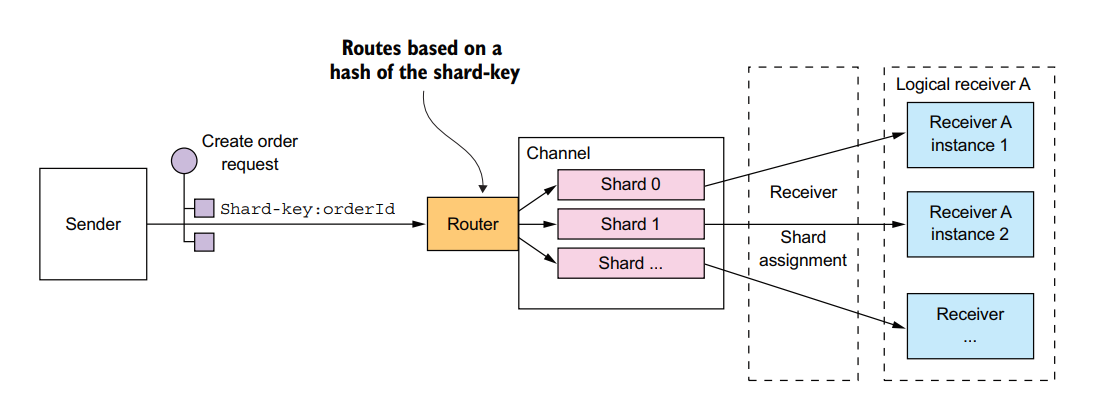
### Cạnh tranh về bên nhận và thứ tự thông điệp

Một thách thức là làm thế nào để mở rộng bên nhận thông điệp trong khi vẫn giữ thứ tự thông điệp. Đó là yêu cầu cơ bản khi có nhiều instances của service xử lý thông điệp song song. Hơn nữa, thậm trí một instance có thể sẽ sử dụng đa luồng. Sử dụng đa luồng và đa instance để xử lý đồng thời thông điệp tang thông lượng của ứng dụng. Nhưng thách thức với xử lý thông điệp đồng thời đảm bảo mỗi thông điệpc được xử lý đúng một lần và đúng thứ tự.

Ví dụ, giả sử có 3 instance của một service đọc thông điệp từ cùng một point-to-point channel và bên gửi publish event Order Created, Order Updated, Order Cancel tuần tự. Triển khai của thông điệp đồng thời chuyển mỗi message tới bên nhận khác nhau. NBởi vì do dễ mạng hoặc bất cứ lý do gây trễ, thông điệp có thể bị xử lý sai thứ tụ, gây ra kết quả không mong muốn. Về lý thuyết, instance của service có thể xử lý thông điệp Order Cancelled trước khi tiến trình service khác xử lý Order Created.

Giải pháp cho vấn đề là là sử dụng message broker như Apache Kafka hay AWS Kinesis để sử dụng shared channel:

1. Shared channel bao gồm 1 hoặc hai shard, mỗi trong số chúng hoạt động như một channel
2. Bên gửi cụ thể shard key trong phần header, broker sử dụng shard key đó để gán thông điệp tới shard/partition cụ thể. Nó có thể, ví dụ, chọn shard bằng canhs tính mã hash của shard key lấy phần dư với số lượng shard.
3. Message broker nhóm các instance của bên nhận và coi nó như là một bên nhận logic. Apache Kafka sử dụng thuật ngữ consumer group. Message broker gán mỗi shard tới một bên nhận duy nhất. Nó gán lại shard khi bên nhận khởi động và tắt.



Ví dụ như hình trên, mỗi thông báo sự kiện Order có thêm trường orderId làm shard key. Mỗi sự kiện cho một thứ tự củ thể được publish tới cùng shard, được đọc bởi một instance duy nhất của bên nhận. Kết quả đạt được, các thông báo đảm bảo thứ tự của nó trong xử lý.

### Xử lý trùng thông báo

Đây là một khó khăn bạn cần xử lý khi sử dụng thông điệp.Ý tưởng của Message broker sẽ chuyển mỗi thông điệp một lần, đảm bảo duy nhất 1 lần gây ra sự tốn kém. Thay vào đó, message broker đảo bảo ít nhất thông điệp được chuyển đi một lần.

Trong trường hợp có lỗi ở client hay network, thông điệp có thể bị chuyển tới nhiều lần. Để xử lý vấn đề này, bạn nên dùng message broker nào mà duy trì thứ tự khi chuyển thông điệp. Có hai hướng xử lý:

* Viết xử lý cho thông điệp không thay đổi.
* Theo dõi thông điệp và loại bỏ bản sao

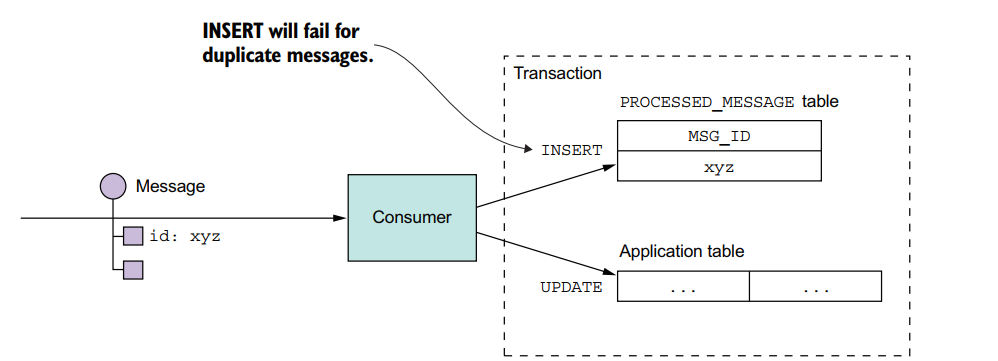
#### Viết xử lý cho thông điệp không thay đổi

Nếu phần logic xử lý thông điệp không đổi, sau đó thông điệp bị trùng lặp thì không ảnh hưởng. Logic ứng dụng là bất biến nếu gọi nhiều lần với cùng đầu vào và không có thêm ảnh hưởng. Ví dụ, hủy một order đã được hủy rồi là một hành động bất biến.

Không may, logic của ứng dụng thường không bất biến. Hoặc có lẽ bạn sử dụng dùng một message broker không duy trì thứ tự khi chuyển lại thông điệp. Trùng hoặc sai thứ tự thông điệp gây ra bug. Trong tình huống này, bạn phải viếc các xử lý thông điệp theo dõi thông điệp và loại bỏ cái trùng

#### Theo dõi thông điệp và loại bỏ các bản sao

Giải pháp cho vấn đề này là bên tiêu thụ theo dõi các thông điệp mà nó phải xử lý sử dụng message id và loại bỏ các cái trùng. Ta có thể chứa message id cua rmỗi thông điệp mà bên tiêu thụ đã xử lý vào một bảng trong database.



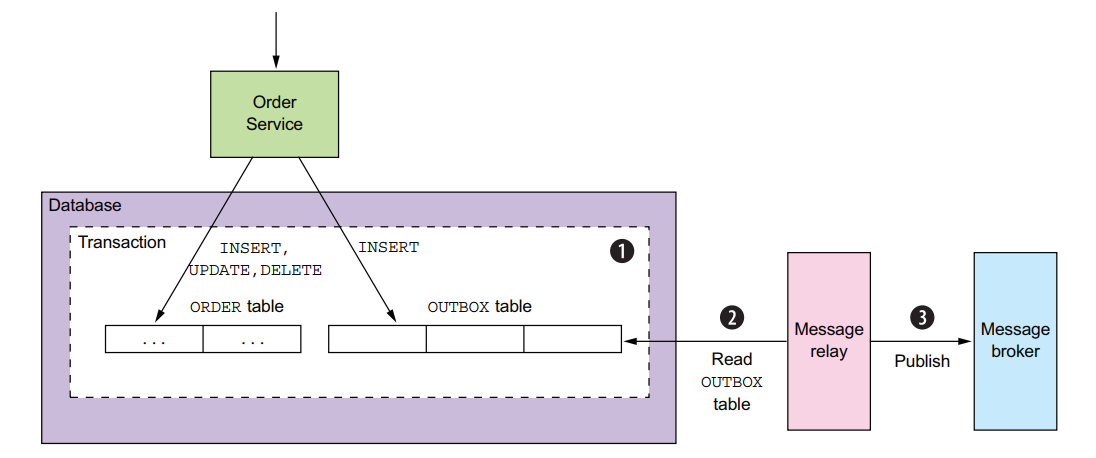
Khi Consumer xử lý message, nó ghi lại message id trong bảng của database như là một phần của transaction. Một cách khác cho việc xử lý thông điệp là ghi lại các message id trong bảng của ứng dụng thay vì một bảng chuyên dụng. Cấch tiếp cận này đặc biệt hữuu ích khi sử dụng database dạng NoSQL.

### Thông điệp transaction

Một service thường publish thông điệp như là một phần của transaction cập nhật database. Giải pháp truyền thông ta sẽ sử dụng một distributed transaction bao trùm database và message broker. Nhưng nó không phải là lựa chọn tốt cho các ứng dụng hiện đại ngày nay. Hơn nữa, nhiều broker hiện đại ngày nay như Apache Kafka không hỗ trợ distributed transaction.Ta sẽ sử dụng các cơ chế khác nhau để phù hợp với ứng dụng

#### Sử dụng bảng trong database như là một hàng đợi thông điệp (message queue)

Ví thử bạn sử dụng cơ sở dữ liệu quan hệ. Một cách đơn giản để chắc chắn publish thông điệp là áp dụng Transactional outbox pattern. Pattern nfay sử dụng bảng trong database như là một hàng đợi thông điệp tạm thời.



Hình trên ta có thể thấy, Order service có một bảng OUTBOX trong database.Như là một phần của transaction, service gửi thông điệp bằng cách thêm chúng vào bảng OUTBOX. Tính Atomicity được đảo bảo bởi vì đây là ACID transaction local.OUTBOX hoạt động như một hàng đợi tạm thời. Message relay là thày phần đọc bảng OUTBOX và publish thông điệp lên message broker.

Bạn có thể đạt được điều tương tự với một vài cơ sở dữ liệu NoSQL. Mỗi thực thể kinh doanh được chứa như là một bản ghi trong database có thộc tính danh sách các thông điệp được publish. Khi service cập nhật thực thể trong database, nó nối thông điệp vào list đó. Đó là tính atomic bởi vì nó hoàn toàn làm việc vơi smột hoạt động cơ sở dữ liệu.Có hai cách để publish message từ database lên message broker.

#### Sử *dụng Polling publisher pattern*

Nếu ứng dụng sử dụng cơ sở dữ liệu quan hệ, một cách đơn giản để publish là để MessageRelay thăm dò bảng cho các thông điệp chưa được publish. Định kì nó sẽ truy vấn bảng: Select \* from OUTBOX ORDER BY … ASC.

Tiếp theo, MessageRelay publish các thông điệp đó tới message broker, gửi mỗi chúng tới đúng channel. Cuối cùng, xóa thông điệp đó khỏi OUTBOX

BEGIN

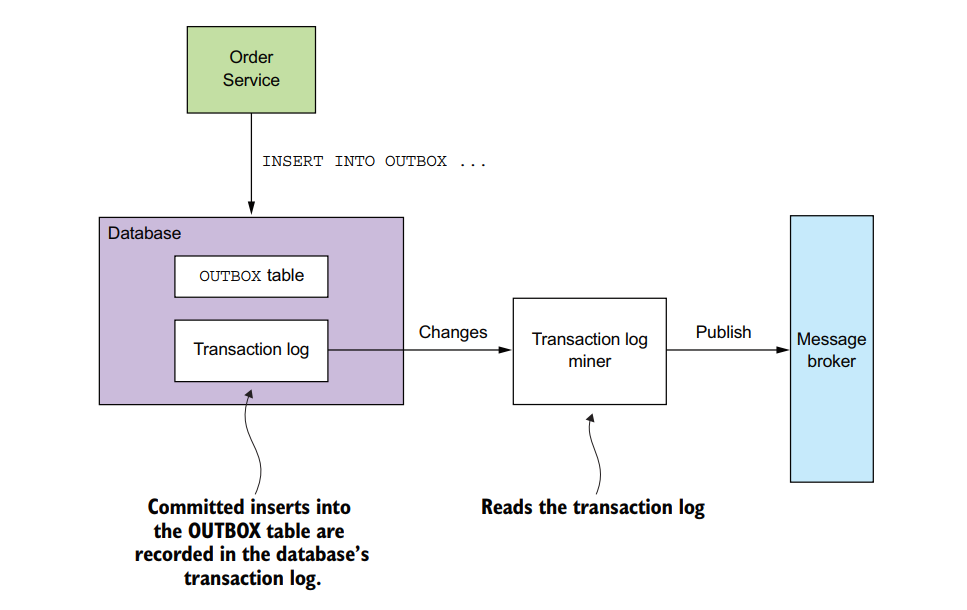
DELETE FROM OUTBOX WHERE ID in (….)

COMMIT

Thăm dò database là phương pháp đơn giản làm việc khá tốt với quy mô nhỏ. Nhược điểm là thường xuyên thăm dò gây ra tốn kém.

#### Sử dụng Transaction log tailing pattern

Một giải pháp tinh vi hơn là cho messageRelay vào đoạn cuối của database transaction log. Mọi cập nhật đã được commit bởi ứng dụng sẽ được biểu diễn lại như một thực thể trong log. Transaction log miner có thể đọc transaction log và publish thay đổi như là một thông báo tới message broker.



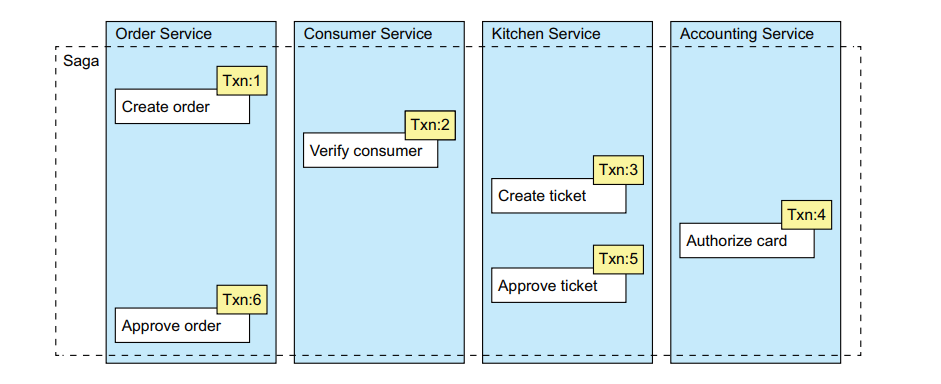
Transaction log minner đọc các thực thể transaction log. Nó biến đổi mỗi thực thể log tương ứng thành một thông điệp và publish nó lên message broker. Cách tiếp cận này có thể sử dụng để publish thông điệp đã ghi vào OUTBOX trong RDBMS hoặc nối vào bản ghi trong NoSQL database. Một số open source hỗ trợ như: Debezium, LinkedIn Databus, DynamoDB streams, Eventuate Tram.

# Quản lý transaction sử dụng Saga

Saga là cơ chế duy trì tính nhất quán dữ liệu trong kiến trúc micro mà không phải sử dụng bất kì distributed transaction. Bạn định nghĩa ra saga cho mỗi command hệ thống mà cần cập nhật dữ liệu trên nhiều service. Saga là một chuỗi các transaction local. Mỗi transaction đó cập nhật dữ liệu bên trong service sử dụng ACID transaction.

Hoạt động hệ thống thiét lập bước đầu của saga. Một giao dịch hoàn thành sẽ kích hoạt thực thi của local transaction tiếp theo. Một lợi ích của thông điệp bất đồng bộ là nó đảm bảo cho tất cả các bước của saga được thực thi, thậm trí một hoặc nihều phần saga tạm thời không có sẵn.

Ví dụ về sag: Create Order saga

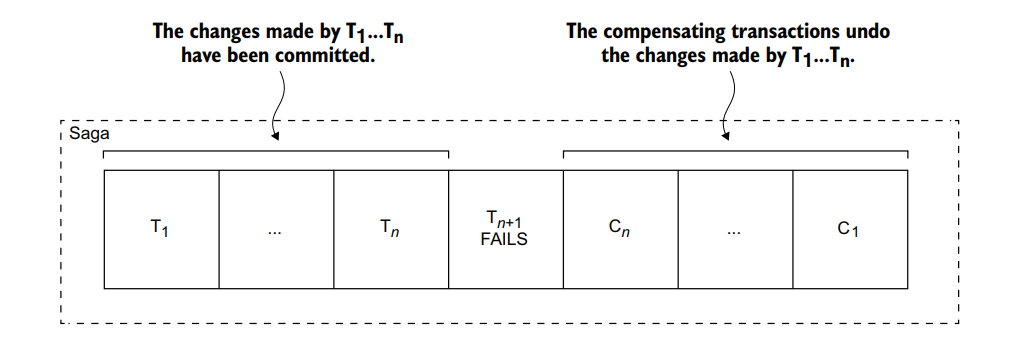


Order service triển khai createOrder() sử dụng saga. Local transaction đầu tiên của ssage được thiết lập bởi request bên ngoài để tạo ra một order. 5 local transactions còn lại mỗi cái được kích hoạt bởi sự hoàn thành của cái trước nó. Bao gồm:

* Order Service: Tạo ra order có trạng thái APPROVAL\_PENDING
* Consumer Service: Xác minh xem consumer có thể order không
* Kitchen Service: Kiểm tra tính đúng đắn các thông tin order và tạo ra Ticket với trạng thái CREATE\_PENDING.
* Account Service:Cấp quyền credit card cho consumer.
* Kitchen Service: Thay đổi trạng thái của Ticket sang AWAITING\_ACCEPTANCE.
* Order Service: Thay đổi trạng thái của Order về APPROVED

## Saga sử dụng cơ chế compensating transaction để rollback thay đổi

Giả sử ta có transaction của saga bị lỗi, tac động của n transaction trước đó cần rollback. Mỗi bước, Ti có mốt tương ứng compensating transaction Cj , thứ sẽ undo lại tác dụng của Ti Để undo lại tác động của n bước, saga phải thực hiện mỗi Ci ­theo thứ tự, như hình sau:



Saga thực thi compensation transaction theo thứ tự ngược của transaction :Cn,….C1 .Khi Ci  hoàn thành, sẽ kích hoạt Ci-1 .

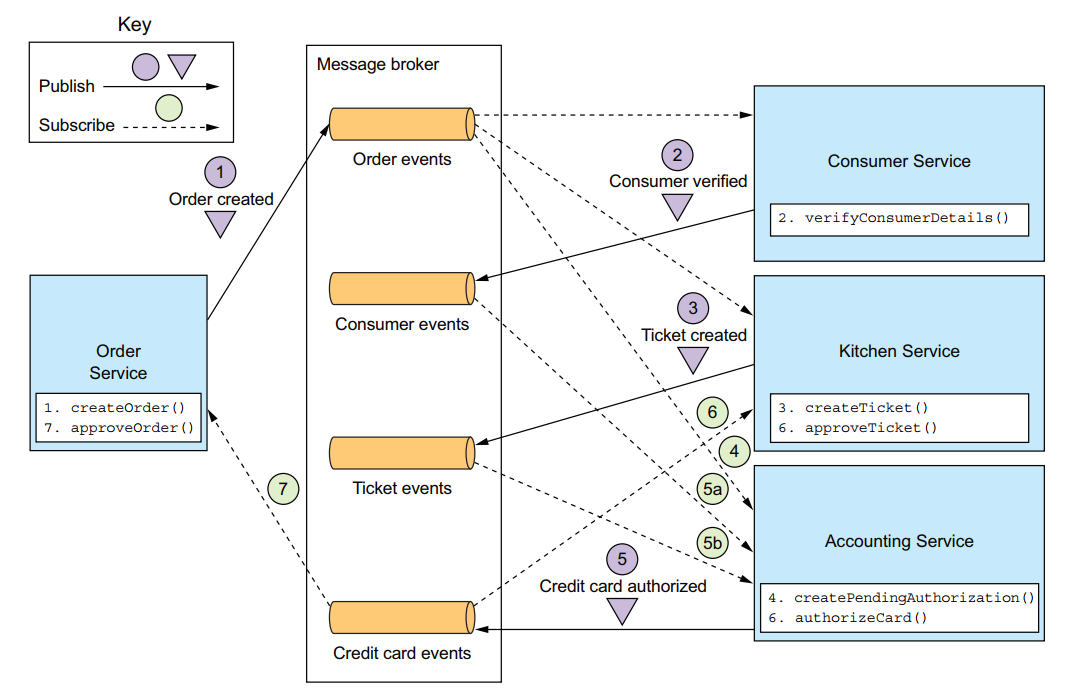
## Điều phối saga

Triển khai một saga bao gồm logic điều phối các bước trong saga. Khi saga được khởi tạo bởi hệ tống, logic điều phối phải chọn và nói với các thành phần tham gia saga để thực thi local transaction. Một khi transaction hoàn thành, điều phối luồng saga lựa chọn và gọi thành phần saga tiếp theo. Quá trình tiếp tục tới khi saga thực hiện hết các bước. Nếu có local transaction nào lỗi, saga phải thực thi compensating transaction theo thứ tự ngược lại. Có 2 hướng logic điều phối saga:

* Choreography: Phân bổ việc ra quyết định và trình tự các thành phần tham gia saga. Chúng gaio tiếp bằng cách trao đổi sự kiện.
* Orchestration: Tập trung logic điều phối saga trong một class, gọi là saga orchestrator class. Một saga orchestrator gửi thông điệp lệnh cho các thành phần saga nói cho chúng biết phải làm gì.

#### Saga dựa trên Choreography

Khi sử dụng choreography, không có trung tâm điều phối nói cho các thành phần tham gia saga phải làm gì. Thay vào đó, các thành phần saga subscribe sự kiện nhau và phản hồi chúng. Ví dụ với Create Order Saga

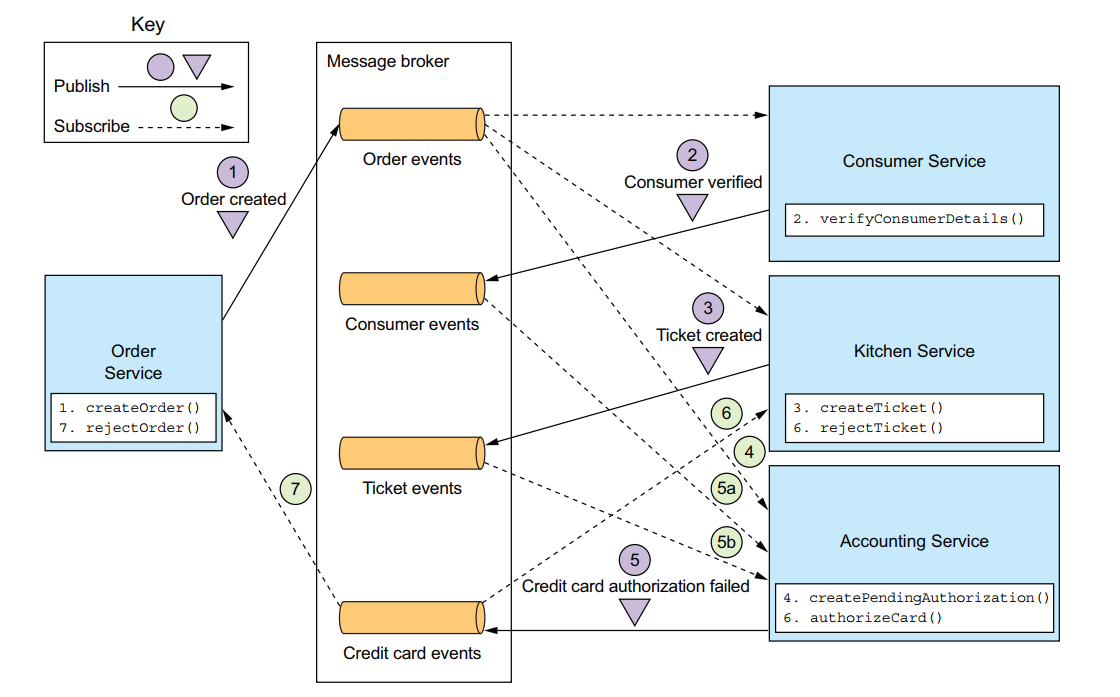


Luồng mong muốn:

1. Order Service tạo ra một Order với trạng thái APPROVAL\_PENDING và publish sự kiện OrderCreated.
2. Consumer Service tiêu thụ sự kiện OrderCreated, xác minh xem consumer có thể tạo ra order không, và publish sự kiện ConsumerVerified.
3. Kitchen Service tiêu thụ sự kiện OrderCreated , kiểm tra thông tin Order, tạo ra Ticket với trạng thái CREATE\_PENDING, publish sự kiện Ticket Created.
4. Accounting Service tiêu thụ sự kiện OrderCreated và tạo ra CreditCardAuthorization trong trạng thái PENDING.
5. Accounting Service tiêu thụ sự kiện TicketCreated và ConsumerVerified, xử lý thay toán thẻ của consumer, và publish sự kiện CreditCardAuthorized.
6. Kitchen Service tiêu thụ sự kiện CreditCardAuthorized và thay đổi trạng thái của Ticket sang AWAITING\_ACCEPTANCE
7. Order Service nhận được sự kiện CreditCardAuthorized, thay đổi trạng thái của Order thành APPROVED và publish sự kiện OrderApproved.

Nếu khi Accounting Service không cấp quyền được cho credit card của khách hàng, luồng sự kiện sẽ diễn biến như sau:

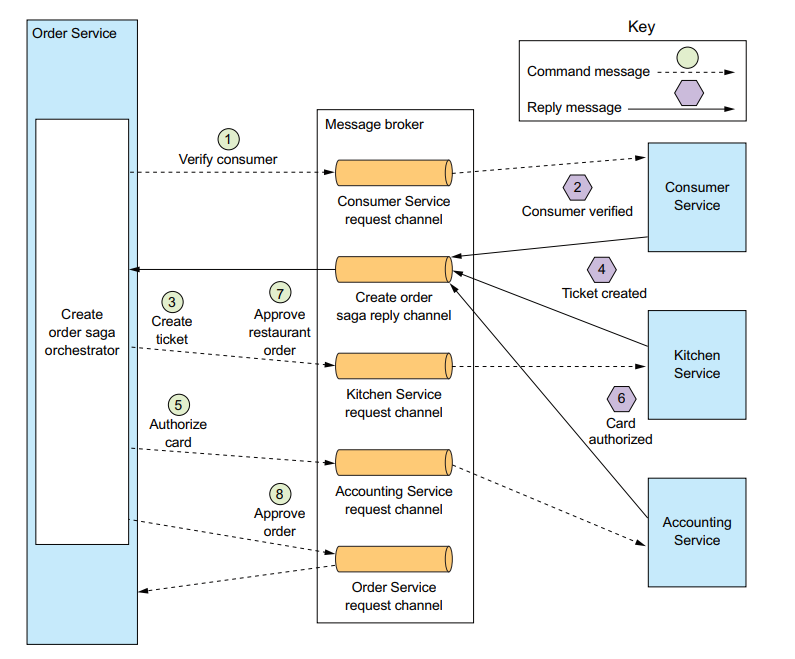
1. Order Service tạo ra một Order với trạng thái APPROVAL\_PENDING và publish sự kiện OrderCreated.
2. Consumer Service tiêu thụ sự kiện OrderCreated, xác minh xem consumer có thể tạo ra order không, và publish sự kiện ConsumerVerified.
3. Kitchen Service tiêu thụ sự kiện OrderCreated , kiểm tra thông tin Order, tạo ra Ticket với trạng thái CREATE\_PENDING, publish sự kiện Ticket Created.
4. Accounting Service tiêu thụ sự kiện OrderCreated và tạo ra CreditCardAuthorization trong trạng thái PENDING.
5. Accounting Service tiêu thụ sự kiện TicketCreated và ConsumerVerified, xử lý thay toán thẻ của consumer, và publish sự kiện Credit Card Authorization Failed.
6. Kitchen Service tiêu thụ sự kiện Credit Card Authorization Failed và thay đổi trạng thái của Ticket sang REJECTED.
7. Order Service nhận được sự kiện C Credit Card Authorization Failed, thay đổi trạng thái của Order thành REJECTED.



Có một số vấn đề khi triển khai cách này. Đầu tiên là việc phải đảm bảo rằng thành phần tham gia saga cập nhật database của nó và publish sự kiện như là một phần của database transaction. Mỗi bước trong saga cập nhật database và publish sự kiện. Do đó, để giao tiếp tin cậy, các thành phần tham gia saga phải sử dụng thông điệp transaction. Thứ hai, đó là vấn đề đảm bảo rằng thành phần tham gia phải có khả năng ánh xạ mỗi sự kiện mà nó nhận được với dữ liệu của nó.

#### Saga dựa trên Orchestration

Với phương pháp này, ta tập trung việc xử lý vào một class, class đó có trách nhiệm nói cho các thành phần tham gia saga biết phải làm gì. Saga Orchestration giao tiếp với các thành phần sử dụng command/async reply. Để thực hiện một bước saga, nó gửi một thông điệp lệnh tới thành phần nói cho nó biết làm gì. Sau khi thành phầ nđó hoàn thành công việc, nó gửi lại một thông điệp phản hồi cho orchestrator. Orchestrator xử lý thông điệp đó và quyết định xem sẽ làm gì tiếp theo.



Ta sẽ sử dụng lại ví dụ với phần trước, nhưng sử dụng Orchestration. Khi này, saga orchestrator là CreateOrderSaga class, nó sẽ gọi các thành phần tham gia saga sử dụng asynchronous request/response. Class này sẽ theo dõi quá trình xử lý và gửi các thông điệp lệnh tới các thành phần. CreateOrderSaga đọc các thông điệp phản hồi từ các kênh reply và quyết định xử lý tiếp như nào.

Ta có thể thấy, luồng mong muốn của quá trình trên sẽ là :

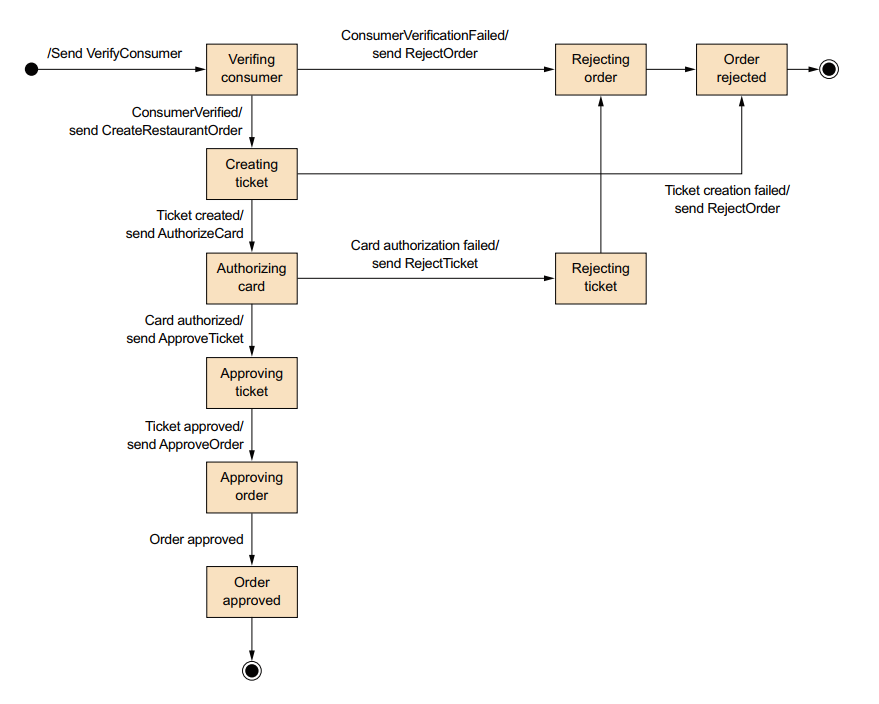
1. Saga Orchestrator gửi lệnh Verify Consumer tới Consumer Service.
2. Consumer Service phản hồi với thông điệp Consumer Verified.
3. Saga Orchestrator gửi lệnh Create Ticket tới Kitchen Service
4. Kitchen Service phản hồi với thông điệp Ticket Created.
5. Saga Orchestrator gửi thông điệp Authorize Card tới Accounting Service.
6. Accounting Service phản hồi với thông điệp Card Authorized.
7. Saga Orchestrator gửi lệnh Approve Ticket tới Kitchen Service
8. Saga Orchestrator gửi lệnh Approve Order tới Order Service.

Đó chỉ là ngữ cảnh mong muốn, nhưng trên thực tế ta sẽ có rất nhiều ngữ cảnh xảy ra với saga. Do đó, việc mô hình hóa saga như máy trạng thái để mô tả hết các ngữ cảnh có thể là điều nên làm.

#### Mô hình hóa saga orchestrator như là máy trạng thái

Máy trạng thái bao gồm một tập các trạng thái và tập các chuyển tiếp trạng thái được kích hoạt bởi các sự kiện. Mỗi trạng thái có thể có một hành động. Chuyển đổi giữa các trạng thái được kích hoạt bởi sự hoàn thành của local transaction bởi thành phần tham gia saga. Trạng thái hiện tại và đầu ra cụ thể của local transaction quyết định trạng thái dịch chuyển và hành động thực hiện nếu có.

Ví dụ với Create Order saga ở trên:



Mỗi bước trong saga orchestrator bao gồm việc service cập nhật database và publish thông điệp. Ví dụ, Order Service tạo một Order và Create Order Orchestrator và gửi thông điệp tới thành phần tham gia saga đầu tiên. Ví dụ như Kitchen Service, xử lý thông điệp lệnh bằng việc cập nhật database của nó và gửi một thông điệp phản hồi. Order Service xử lý thông điệp đó bằng cách cập nhật trạng thái của Saga Orchestrator và gửi một thông điệp lệnh tới thành phần tiếp theo.

## Xử lý sự thiếu hụt tính Isolation

Isolation chính là chữ I trong ACID. Isolation đảm bảo rằng kết quả của việc thực hiện nhiều transaction đồng thời là giống nhau nếu chúng thực hiện theo đúng một thứ tự.

Khi sử dụng saga, ta sẽ có khó khăn vê tính Isolation. Bỏi vì việc cập nhật bởi mỗi local transaction trong saga có thể nhìn thấy ngay lập tức bởi saga khác một khi transaction commit. Điều này gây ra hai vấn đề. Một,saga khác có thể thay đổi dữ liệu truy cập bởi saga này khi nó đang thực hiện. Và saga khác cũng có thể đọc dữ liệu trước khi saga hoàn thành công việc, do đó có thể tiếp xúc với dữ liệu không nhất quán. Do đó, saga là ACD:

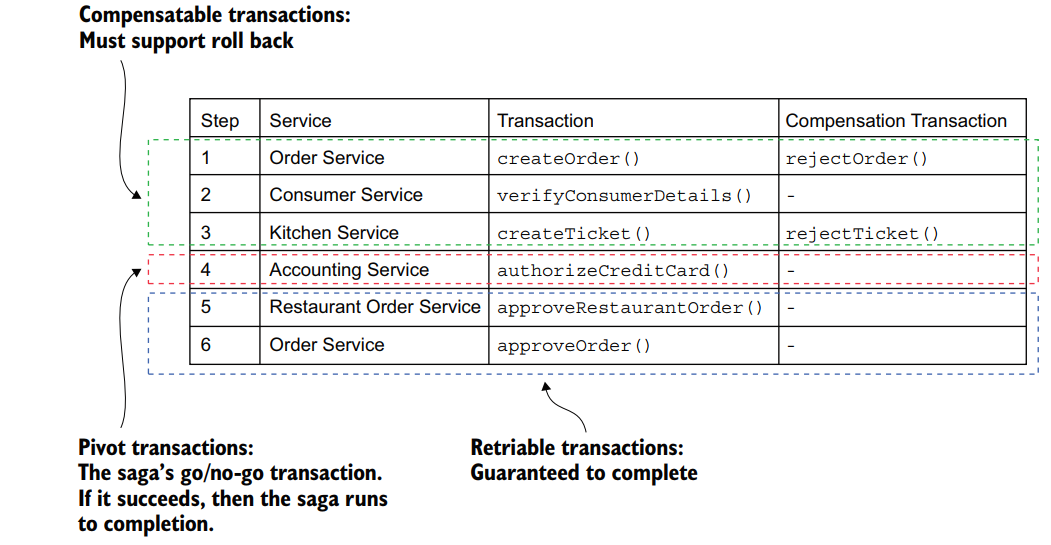
* Atomicity:Saga đảm bảo tất cả transaction được thực hiện hoặc tất cả thay đổi bị hoàn tác.
* Consistency: Tính toàn vẹn tham chiếu bên trong một service được xử lý bởi local database. Tính toàn vẹn tham chiếu qua các dịch vụ được xử lý bởi các dịch vụ đó.
* Durability : Được xử lý bởi local database.

Do thiếu Isolation, gây ra dị thường (anomalies). Trên thực tế, dev chấp nhận việc giảm tính isolation để đổi lại được hiệu năng cao hơn. Để xử lý vấn đề này, ta sử dụng chiến thuật countermeasures. Bằng cách sử dụng countermeasures, bạn có thể viết logic kinh doanh dựa trên saga hoạt động chính xác hơn, giảm sự thiếu hụt tính isolation.

#### Sử dụng Countermeasures xử lý việc thiếu tính isolation

Saga bao gồm 3 kiểu của transaction:

* Compensatable transaction
* Pivot transaction
* Retriable transaction



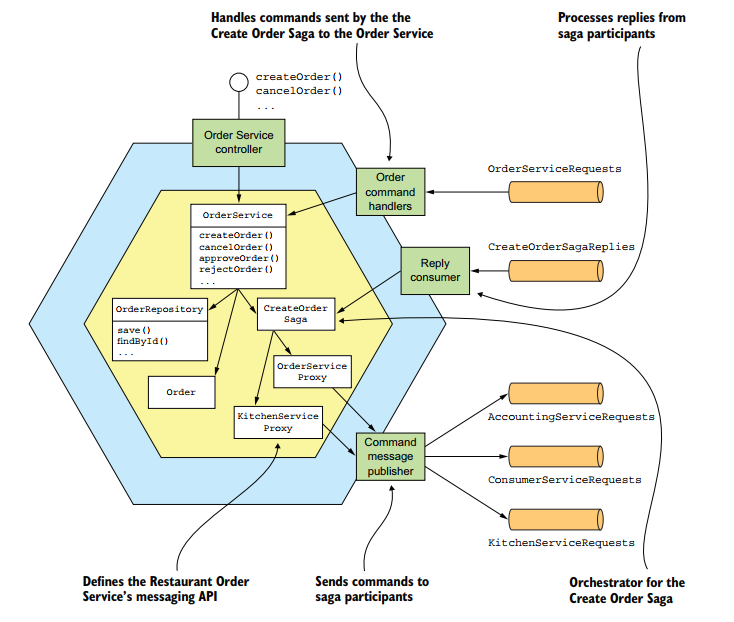
#### Chiến thuật countermeasure: sử dụng semantic lock

Khi sử dụng semantic lock countermeasure, các transaction loại compensatable thiết lập cờ ở bất kì bản ghi nào nó tạo mới hoặc update. Giá trị cờ chỉ ra rằng bản ghi đó vẫn chưa được committed và có thể tiềm ẩn thay đổi. Cờ có thể là một khóa chặn transaction khác không thể truy cập bản ghi hay cảnh báo bản ghi này có sự nghi ngờ. Cờ này sẽ được tắt khi saga hoàn thành thành công hoặc saga rollback. Bạn cần phải quyết định trên từng trường hợp cơ bản saga xử lý thế nào khi bản ghi đó đang bị lock. Ví dụ, khi cancelOrder(). Client có thể gọi một hoạt động để hủy Order đang trọng trạng thái APPROVAL\_PENDING.

Có nhiều cách để xử lý. Một là ta báo với client thử lại sau và trả lỗi về. Cách này rất đơn giản để thực hiện nhưng nó khiến client trở lên phức tạp vì client phải triển khai logic thử lại. Hướng khác là ta sẽ block cancelOrder() tới khi lock được mở. Lợi ích của sử dụng semantic lock là tái tạo lại được tính Isolation của ACID.

Ngoài ra còn một số cách như sử dụng commutative update, pessimistic view, reread value, version file, by value,…

## Ví dụ thiết kế cho Order Service và Create Order Saga



# Triển khai query trong kiến trúc microservice

Có hai hướng triển khai query khi sử dụng kiến trúc micro:

* Sử dụng API composition pattern
* Sử dụng CQRS pattern

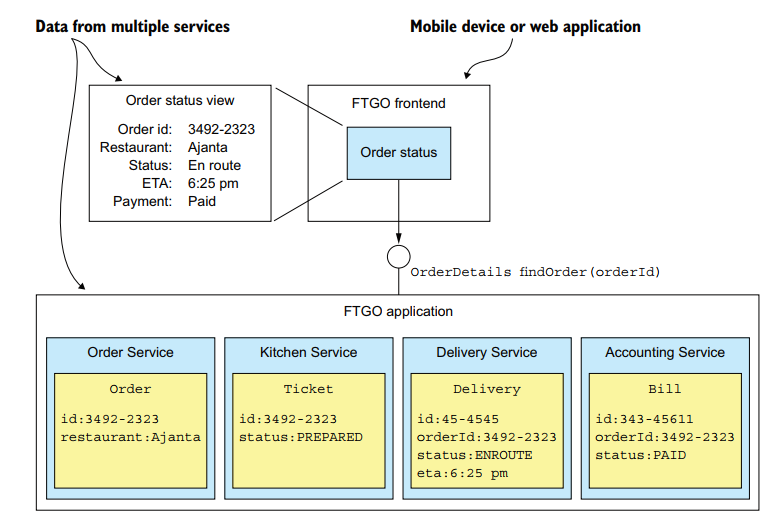
## API composition pattern

Giả sử ta có hoạt động findOrder() để lấy được order theo khóa chính. Input của phương thức là OrderId và output là OrderDetails chứa các thông tin về order.Giả sử ta có client là một project front-end hay thiết bị di động lấy dữ liệu từ phương thức này và hiển thị ra các thông tin trạng thái Order. Nó bao gồm các thông tin cơ bản về order như trạng thái, trạng thái thanh toán, trạng thái của order ở nhà hàng, trạng thái giao hàng,…

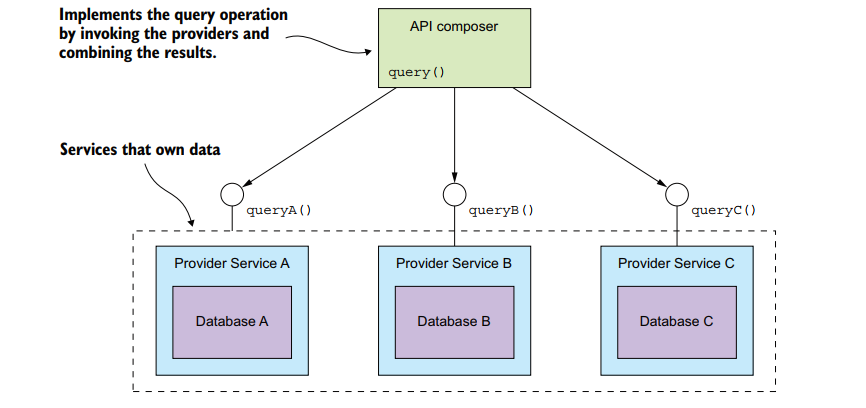
Bởi vì dữ liệu nằm ở một cơ sở dữ liệu nên với ứng dụng loại mono có thể dễ dàng lấy thông tin này thông qua câu lệnh SELECT để join các bảng với nhau. Đối với ứng dụng micro, data rải rác ở các service:

* Order Service: các thông tin cơ bản của order
* Kitchen Service: Trạng thái order từ góc nhìn của nhà hàng, và thời gian dự kiến nấu xong
* Delivery Service: Trạng thái giao hàng, thời gian dự kiến giao hàng, vị trí hiện tại
* Account Service: Trạng thái thanh toán

Bất cứ client nào cần thông tin chi tiết của order cần phải “hỏi” tất cả các service này.

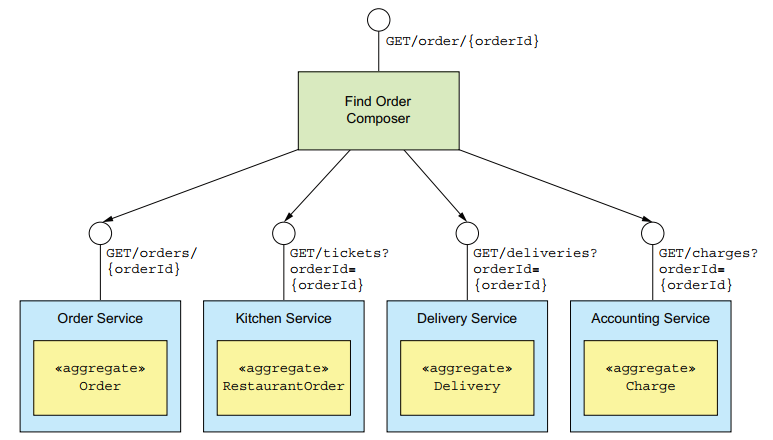


API composition pattern hoạt động bằng cách gọi tất cả các service mà chứa dữ liệu cần và kết hợp kết quả lại. Có hai kiểu:

* API composer: Triển khai query bằng cách query các service cung cấp. 
* Provider service: Tạo ra một service sở hữu dữ liệu mà query trả về.

API composer triển khai query bằng cách lấy dữ liệu từ các service cung cấp và kết hợp chúng vào thành kết quả. API composer có thể .

Ta thử triển khai phương thức findOrder () trên.Sẽ có dạng như sau:



Ta thấy, API composer là một service cung cấp REST endpoint. Provider service cũng triển khai REST API. Nhưng dạng cũng như việc sử dụng gRPC. Find Order Composer triển khai REST dưới enpoint GET /order/{orderId}. Nó gọi 4 service để lấy được dữ liệu và kết hợp chúng bằng cách gọi các service với tham số orderId.

Sử dụng cách này có các lợi ích và hạn chế sau:

* Lợi ích
  + Đơn giản, trực quan
* Hạn chế
  + Gây ra quá tải, tốn tài nguyên tính toán và tài nguyên mạng.
  + Giảm sự sẵn sàng
  + Thiếu hụt tính nhất quán của dữ liệu

## Sử dụng CQRS pattern

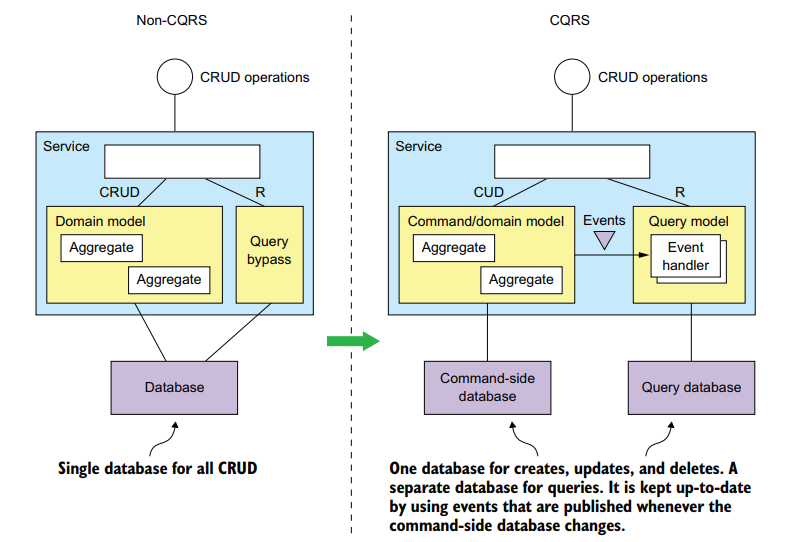
Giả sử ta có một phương thức findOrderHistory() với đầu vào là consumerId,pagination,filter và đầu ra là dữ liệu bạn cần về lịch sử Order của khách hàng có phân trang và lọc. Phương thức này được gọi bởi Module View triển khai Order History. View hiển thị tổng quát của từng order, bao gồm số order, trạng thái order, tổng tiền, thời gian dự kiến.

Ở mặt khác, ta thấy phương thức này tương tự phương thức findOrder() trên . Sự khác biệt nằm ở phương thức này là nó trả về một danh sách order, chứ không phải một. Phương thức này không hề đơn giản. Ví dụng, một tiêu chí lọc của nó là keyword mà trùng với một item trong menu. Chỉ có 2 service chứa menu item là KitchenService và OrderService, trong khi đó Delivery Service hay Accounting Service không chứa thông tin này, dó đó ta cũng không thể lọc dữ liệu sử dụng keyword. Sử dụng phương pháp API composition cũng có thể đạt được kết quả như mong muốn nhưng nó gây ra việc cần kết nối dữ liệu quá lớn.

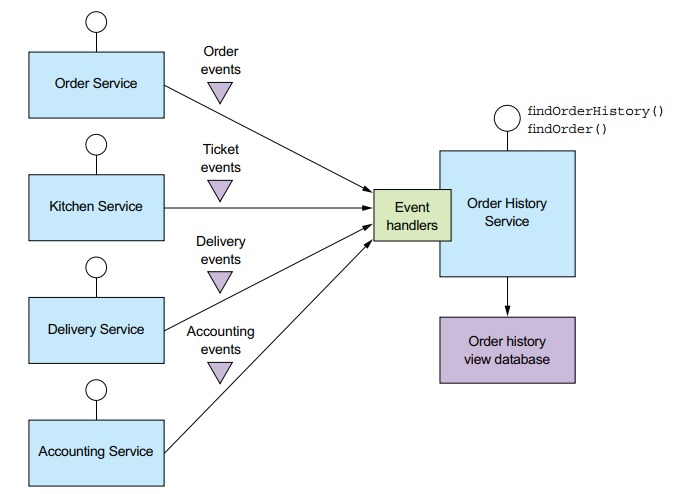
Giải pháp cho vấn đề này ta ta sẽ “chia để trị” (separate concerns).

### Tổng quan về CQRS

CQRS tách command và query ra riêng biệt. Phần command và data model triển khai hoạt động tạo, cập nhật, xóa . Phần query và data model triển khai query,được đồng bộ với phần command data model bằng cách subscribe sự kiện được publish bởi phần command.



CQRS không chỉ áp dụng cho service, nó còn có thể áp dụng cho việc định nghĩa query service. Query service có một API bao gồm chỉ các hoạt động query. Không có command. Nó triển khai hoạt động query bằng việc query database mà luôn được giữ mới nhất thông qua việc subscribe sự kiện được publish bởi một hay nhiều service.



### Lợi ích và điểm yếu của CQRS

* Lợi ích
  + Cho phép triển khai các query hiệu quả trong kiến trúc micro
  + Cho phép triển khai hiệu quả của các query đa dạng
  + Cho phép query trong ứng dụng loại event sourcing-based
  + Cải thiện việc phân tách mối quan tâm
* Điểm yếu
  + Phức tạp
  + Đối phó với nhiều vấn đề khó xử lý.

# External API pattern

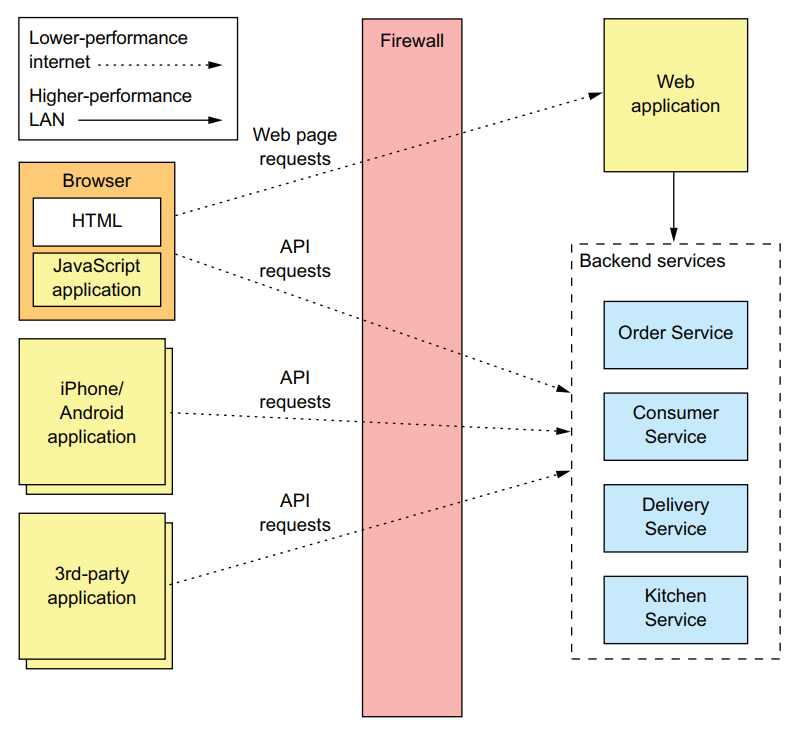
## Vấn đề của việc thiết kế External API

Như nhiều ứng dụng, chúng ta điều tạo ra các API, như REST API. Client của chúng có thể là ứng dụng di động, JavaScript chạy trên trình duyệt hay các ứng dụng phát triển bởi đối tác. Trong kiến trúc micro, API mà được để lộ cho client là các API của ứng dụng mono. Nhưng khi phát triển kiến trúc micro, không có chỉ một API, bởi vì mỗi service lại có API riêng của nó. Ta cần quyết định xem API sẽ được phơi bày cho client.

Công việc thiết kế external API là một thách thức do sự đa dạng client. Client khác nhau sẽ yêu cầu dữ liệu khách nhau . Ứng dụng web sẽ hiển thị nhiều thông tin hơn là ứng dụng di động. Mặt khác, các client truy cập service thông qua các kiểu mạng khác nhau. Client bên trong firewall sử dụng mạng LAN hiệu năng cao, client ngoài firewall sử dụng đường truyền internet hay mạng di động có hiệu năng thấp hơn.

Một cách tiếp cận thiết kế API là cho client gọi service trực tiếp.Nghe có vẻ đơn giản, nhưng hiếm khi được sử dụng trên khiến trúc micro do:

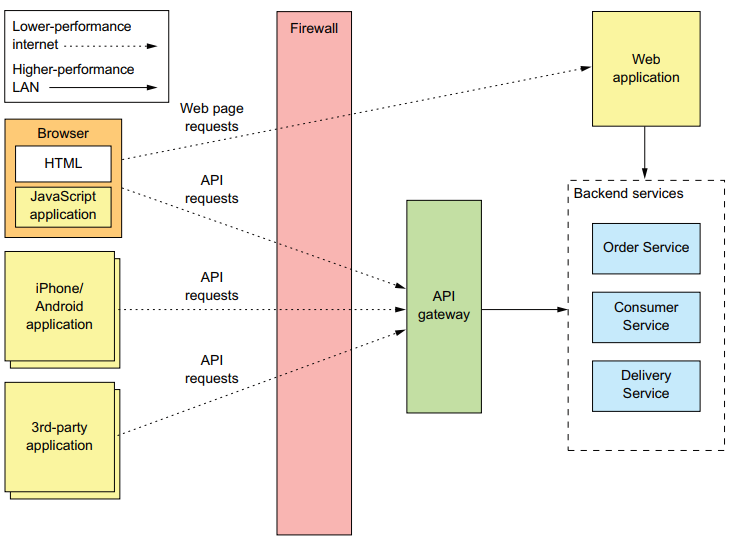
* Cần nhiều request để lấy dữ liệu cần thiết, giảm trải nghiệm người dùng
* Thiếu sự đóng gói cho client biết về mỗi service và khiến khó khăn trong việc thay đổi kiến trúc và API
* Một vài service sử dụng cơ chế IPC gây khó khăn cho client sử dụng, đặc biệt khi các client nằm ngoài firewall.



## API gateway pattern

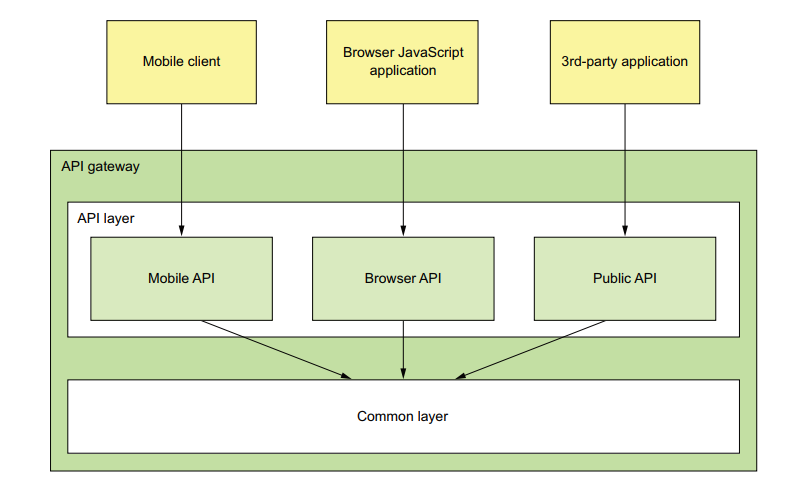
API gateway là một service là entrypoint để vào trong ứng dụng từ bên ngoài. Nó chịu trách nhiệm điều hướng request, API composition, và các chức năng khác như xác thực.

Như trên ta nói, việc gọi nhiều request trực tiếp tới các service là điều không nên. Thay vào đó, client chỉ cần gửi một request tới API gateway. Tương tự việc ta sử dụng pattern Façade. APi gateway đóng gói kiến trúc nội bộ ứng dụng và cung cấp API cho client. Nó có nhiều trách nhiệm như việc xác thực, theo dói, giới hạn số lượng request trên đơn vị thời gian.



Tất cả request đầu tiên phải đi vào API gateway, rồi sẽ được điều hướng tới service thích hợp. API gateway xử lý các request sử dụng API composition pattern và bởi việc gọi nhiều service và tổng hợp lại kết quả.

API gateway kiến trúc dạng layer,module, bao gồm 2 layer: API layer và common layer. API layer bao gồm một hoặc nhiều module API độc lập. Mỗi API module triển khai một API cho một client cụ thể.



Commong layer triển khai các thức năng chung, bao gồm các chức năng như xác thực. API module triển khai mỗi chức năng API trong một hoặc 2 cách. Một vài chức năng API ánh xạ trực tiếp tới một chức năng của service. API module triển khai các chức năng bằng cách điều hướng các request tới đúng service. Có thể sử dụng các file cấu hình để miêu tả luật điều hướng.

API module triển khai các API phức tạp sử dụng API composition, chứa các custom code. Mỗi API triển khai xử lý các request bằng cách gọi các service và kết hợp lại thành kết quả trả về client.

### Lợi ích và hạn chế của API gateway

* Lợi ích:Đóng gói cấu trúc nội của ứng dụng. Thay vì phải gọi cụ thể service, client nói chuyện với gateway là đủ. APi gateway cung cấp mỗi client với cụ thể API, điều này giúp giảm số lượng round-trip (vòng-chuyến) giữa các client và ứng dụng. Đơn giản hóa code client.
* Hạn chế: API gateway là thành phần có tính có sẵn cao phải được phát triển, deploy và quản lý. APi gateway có thể trở thành nút thắt cổ chai trong quá trình phát triển. Dev phải cập nhất API gateway để lộ các API service. API gateway phải được cập nhật lightweight. Không thì các dev phải đợi sự cập nhật đó.

## Triển khai API gateway

Có 2 cách để triển khai API gateway:

* Sử dụng off-the-shelf API gateway product/servic
* Tự phát triển API gateway sử dụng API gateway framework hoặc web framework

Có một vài off-the-shelf service và sản phầm triển khai tính năng API gateway. Ví dụ như AWS API gateway, AWS application load balancer hay Kong hoặc Traefix. Trong tài liệu này tôi sẽ chỉ đề cập tới cách tự triển khai API gateway

### Tự phát triển API gateway

Phát triển API gateway không phải là điều khó khăn. Nó đơn giản là một ứng dụng web điều hướng request tới service khác. Bạn có thể sử dụng các framework yêu thích. Nhưng sẽ có 2 vấn đề ta cần giải quyết:

* Triển khai một cơ chế cho việc định nghĩa luật đinhj tuyến để giảm thiểu code phức tạp.
* Triển khai chính xác hành vi HTTP proxy, bao gồm HTTP header được xử lý.

Tốt nhất ta nên sử dụng mộ framework được thiết kế cho việc này, giúp giảm code bạn cần phải viết. Trong tài liệu này chúng ta sẽ đi tìmhiểu về Netflix Zuul, open source bởi Netflix và Spring cloud gateway, phát triển bởi Pivotal.

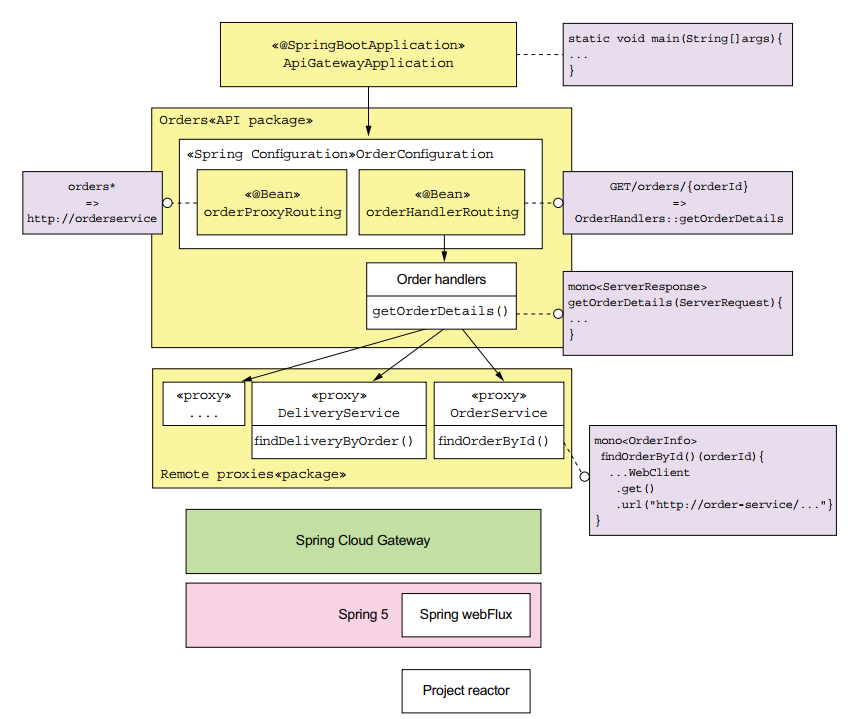
#### Sử dụng Netflix zuul

Zuul framework được phát triển để triển khai edge function như định tuyến, giới hạn request, xác thực. Zuul sử dụng khái niệm filter, request interceptor tương tự servlet filter trong java spring hay NodeJS express middleware. Zuul xử lý HTTP request bằng cách tập hợp một chuỗi các filter chuyển đổi request, gọi backend service, biến đổi response trước khi gửi lại client. Zuul xử lý định tuyến và các chức năng cạnh (edge function). Bạn có thể mở rộng Zuul bằng cách định nghĩa Spring MVC controller triển khai API composition. Nhưng giới hạn chính của Zuul là ở việc nó chỉ có thể triển khai định tuyến dựa trên đường dẫn.

#### Spring cloud gateway

Spring cloud gateway xây dựng trên các framework , bao gồm spring framework 5, spring boot 2, spring webflux. Spring cloud gateway cung cấp một cách đơn giản nhưng hoàn thiện để làm :

* Định tuyến request tới backend service
* Triển khai xử lý request thực hiện API composition
* Xử lý edge function như xác thực.



API gateway bao gồm các gói:

* ApiGatewayMan package: định nghĩa chương trình main cho API gateway
* Một hay nhiều API package: Mỗi API package triển khai một tập các API enpoint
* Proxy package: Bao gồm các lớp proxy được sử dụng bởi API package để gọi service