# Chương 1: Khởi đầu dự án

## 1. Giới thiệu đề tài

Trong bối cảnh đô thị hóa diễn ra mạnh mẽ, các khu chung cư cao tầng ngày càng xuất hiện nhiều tại các thành phố lớn nhằm đáp ứng nhu cầu nhà ở ngày càng tăng của người dân. Song song với đó, việc quản lý vận hành và cung cấp các dịch vụ cho cư dân trở thành một thách thức lớn đối với ban quản lý.  
  
Quản lý cư dân, hợp đồng dịch vụ, hóa đơn thanh toán, bảo trì, vệ sinh, bảo vệ,... là những nghiệp vụ phức tạp, đòi hỏi tính chính xác, kịp thời và minh bạch cao. Tuy nhiên, nhiều chung cư hiện nay vẫn đang sử dụng các phương pháp quản lý thủ công hoặc phần mềm tập trung với kiến trúc monolithic, dẫn đến các hạn chế như:  
+ Dễ xảy ra sai sót, thất thoát thông tin.  
+ Khó mở rộng, khó tích hợp thêm dịch vụ mới.  
+ Thiếu tính tự động hóa, ảnh hưởng đến trải nghiệm của cư dân.  
  
Để giải quyết những hạn chế đó, đề tài "Xây dựng hệ thống quản lý hợp đồng dịch vụ tại chung cư bằng kiến trúc Microservices" được lựa chọn. Đề tài hướng tới việc xây dựng một hệ thống phần mềm hiện đại, linh hoạt, giúp nâng cao hiệu quả quản lý, tăng tính minh bạch và mang lại sự tiện lợi cho cư dân.

## 2. Lý do chọn đề tài

Việc áp dụng kiến trúc Microservices cho hệ thống quản lý chung cư giúp giải quyết triệt để những vấn đề tồn tại trong các mô hình cũ, đồng thời bắt kịp xu hướng chuyển đổi số. Các lý do chính để lựa chọn đề tài này gồm:  
+ Tính cấp thiết và thực tế cao: Đáp ứng nhu cầu quản lý ngày càng phức tạp tại các khu chung cư lớn.  
+ Nâng cao chất lượng dịch vụ: Giúp ban quản lý dễ dàng kiểm soát thông tin cư dân, hợp đồng, dịch vụ, hóa đơn và các cảnh báo liên quan.  
+ Tối ưu trải nghiệm cư dân: Cho phép cư dân tự quản lý hợp đồng, nhận thông báo tự động và thực hiện thanh toán online.  
+ Khả năng mở rộng và bảo trì dễ dàng: Nhờ kiến trúc Microservices, từng chức năng có thể phát triển, triển khai và mở rộng độc lập mà không ảnh hưởng đến toàn hệ thống.

## 3. Ý nghĩa và vai trò của hệ thống

Hệ thống đóng vai trò then chốt trong hoạt động nghiệp vụ của ban quản lý chung cư:  
+ Giảm tải công việc thủ công: Các quy trình như lập hợp đồng, xuất hóa đơn, gửi cảnh báo và quản lý dịch vụ đều được tự động hóa.  
+ Đảm bảo minh bạch: Thông tin dịch vụ, chi phí và hợp đồng được lưu trữ đầy đủ, rõ ràng và dễ tra cứu.  
+ Tăng tính kết nối và tương tác: Ban quản lý và cư dân có thể tương tác trực tuyến qua hệ thống, giảm thiểu khâu trung gian và giấy tờ.  
+ Cung cấp dữ liệu hỗ trợ ra quyết định: Hệ thống lưu trữ và phân tích dữ liệu giúp ban quản lý đánh giá hiệu quả vận hành và cải thiện dịch vụ.

## 4. Tính quan trọng và cấp thiết

Trong thực tế, nếu không có một hệ thống quản lý hợp đồng dịch vụ chuyên nghiệp:  
+ Ban quản lý sẽ gặp khó khăn trong việc kiểm soát số lượng lớn cư dân, dịch vụ và hóa đơn, dễ dẫn đến thất thoát, tranh chấp và giảm uy tín.  
+ Cư dân sẽ thiếu thông tin minh bạch, khó nắm bắt tình trạng hợp đồng, không được nhắc nhở kịp thời khi đến hạn thanh toán hoặc hết hạn dịch vụ, gây ra bất tiện và phiền hà.  
+ Hệ thống quản lý lỗi thời không thể đáp ứng kịp nhu cầu phát triển, mở rộng hoặc tích hợp các công nghệ, dịch vụ mới trong tương lai.  
  
Do đó, việc triển khai một hệ thống quản lý hợp đồng dịch vụ tại chung cư dựa trên kiến trúc Microservices là giải pháp tối ưu, cấp thiết và mang tính chiến lược nhằm nâng cao hiệu quả quản lý, đảm bảo quyền lợi cư dân và đáp ứng yêu cầu phát triển bền vững.

## 5. Mục tiêu

Mục tiêu chính của đề tài là xây dựng một hệ thống quản lý hợp đồng dịch vụ tại chung cư dựa trên kiến trúc Microservices, với các yêu cầu cụ thể như sau:  
+ Tự động hóa quy trình quản lý hợp đồng: Từ khâu định nghĩa dịch vụ, đăng ký, phê duyệt, đến lập hóa đơn và thanh toán.  
+ Tạo điều kiện cho cư dân tự quản lý thông tin: Cho phép cư dân đăng ký, tra cứu, cập nhật hợp đồng, nhận cảnh báo và thực hiện thanh toán online một cách thuận tiện.  
+ Đảm bảo khả năng mở rộng và tích hợp: Thiết kế hệ thống linh hoạt, có thể dễ dàng bổ sung hoặc thay đổi dịch vụ, tích hợp thêm các module mới hoặc các hệ thống thanh toán điện tử.  
+ Nâng cao hiệu quả quản lý: Hỗ trợ ban quản lý giảm tải công việc thủ công, kiểm soát chính xác thông tin, giảm rủi ro sai sót và tranh chấp.

## 6. Phạm vi

Phạm vi triển khai của đề tài tập trung vào các chức năng nghiệp vụ chính và không đi quá sâu vào các yếu tố bên ngoài (ví dụ như hạ tầng phần cứng, chi tiết bảo mật nâng cao, hoặc các cổng thanh toán thực tế). Cụ thể:  
+ Đối với ban quản lý: Quản lý cư dân, định nghĩa dịch vụ, duyệt/hủy tài khoản, lập hóa đơn và quản lý thanh toán.  
+ Đối với cư dân: Đăng ký, quản lý tài khoản, lập và tra cứu hợp đồng, cập nhật dịch vụ, nhận thông báo tự động.

+ Về hình thức thanh toán: Hệ thống hỗ trợ hai hình thức thanh toán gồm thủ công (cư dân nộp tiền mặt hoặc chuyển khoản và được ban quản lý xác nhận trên hệ thống) và thanh toán online thông qua cổng điện tử. Trong khuôn khổ đề tài thực tập, thanh toán online sẽ được mô phỏng bằng cách tích hợp môi trường sandbox của VNPay, giúp kiểm tra luồng thanh toán thật nhưng không phát sinh giao dịch thực tế.  
+ Về mặt kỹ thuật: Xây dựng backend Microservices bằng Spring Boot, RESTful APIs, MySQL; giao tiếp bất đồng bộ bằng RabbitMQ hoặc Kafka; sử dụng Eureka và Spring Cloud Gateway; phát triển giao diện web cho ban quản lý và cư dân.

## 7. Hướng giải quyết

Để đạt được các mục tiêu trên, đề tài sẽ triển khai các hướng giải quyết cụ thể:  
+ Áp dụng kiến trúc Microservices: Tách hệ thống thành nhiều dịch vụ nhỏ, độc lập, dễ triển khai và bảo trì.  
+ Sử dụng công nghệ Spring Boot: Tận dụng khả năng phát triển nhanh, dễ tích hợp và cộng đồng hỗ trợ mạnh mẽ.  
+ Giao tiếp thông qua RESTful APIs: Đảm bảo chuẩn hóa, dễ kết nối giữa frontend và backend.  
+ Triển khai giao tiếp bất đồng bộ: Sử dụng RabbitMQ hoặc Kafka giúp hệ thống xử lý luồng dữ liệu lớn, giảm độ trễ.  
+ Đảm bảo khả năng mở rộng: Có thể thêm mới dịch vụ mà không ảnh hưởng đến các chức năng khác.  
+ Phát triển giao diện web thân thiện: Hỗ trợ truy cập mọi lúc mọi nơi, tối ưu trải nghiệm người dùng cho cả ban quản lý và cư dân.

## 8. Đối tượng và phạm vi áp dụng

Hệ thống quản lý hợp đồng dịch vụ tại chung cư bằng kiến trúc Microservices được thiết kế để phục vụ hai nhóm đối tượng chính:  
  
+ Ban quản lý chung cư (BQL):  
Là đơn vị trực tiếp điều hành, chịu trách nhiệm kiểm soát, phê duyệt và giám sát tất cả các hoạt động liên quan đến hợp đồng dịch vụ của cư dân.  
Quyền và chức năng chính của BQL:  
+ Duyệt, cấp hoặc hủy tài khoản cư dân.  
+ Định nghĩa, cập nhật và quản lý danh mục dịch vụ (điện, nước, bảo trì, vệ sinh, gửi xe...).  
+ Lập và xuất hóa đơn dịch vụ hàng tháng.  
+ Quản lý tình trạng thanh toán, xử lý các vấn đề phát sinh.  
+ Gửi thông báo tự động cho cư dân khi hợp đồng sắp hết hạn hoặc đến kỳ thanh toán.  
Ràng buộc hoạt động của BQL:  
+ Chỉ được duyệt tài khoản khi cư dân cung cấp thông tin hợp lệ.  
+ Không được thay đổi nội dung hợp đồng của cư dân nếu không có yêu cầu chính thức.  
+ Không can thiệp trực tiếp vào quá trình thanh toán online.  
  
+ Cư dân:  
Là những người đang sinh sống và sử dụng dịch vụ tại chung cư.  
Quyền và chức năng chính của cư dân:  
+ Đăng ký tài khoản và chờ BQL phê duyệt.  
+ Lập hợp đồng dịch vụ mới hoặc điều chỉnh hợp đồng đang sử dụng.  
+ Tra cứu thông tin hợp đồng, hóa đơn, lịch sử thanh toán.  
+ Nhận thông báo tự động từ hệ thống.  
+ Thanh toán online các khoản phí dịch vụ.  
Ràng buộc của cư dân:  
+ Chỉ được quản lý hợp đồng gắn với phòng mà mình đang thuê hoặc sở hữu.  
+ Khi đăng ký tài khoản, phải khai báo đầy đủ thông tin xác thực (CCCD, số điện thoại, email).  
+ Chỉ được điều chỉnh dịch vụ trong phạm vi và thời gian quy định.  
  
Về phạm vi áp dụng, hệ thống được tập trung cho các khu chung cư cao tầng tại các thành phố lớn, nơi có số lượng cư dân đông và nhu cầu dịch vụ đa dạng. Trong mô hình quản lý, mỗi phòng (căn hộ) được xem là đơn vị đại diện chính cho hợp đồng dịch vụ; cư dân (chủ hộ) là người đứng tên thực hiện giao dịch, thanh toán và nhận thông báo.  
Các dịch vụ, hợp đồng, hóa đơn đều được gắn với phòng, nhưng trách nhiệm thanh toán và quyền điều chỉnh thuộc về cư dân đang cư trú tại phòng đó. Điều này giúp đảm bảo tính minh bạch, dễ kiểm soát và phù hợp với nghiệp vụ thực tế.  
  
Mối quan hệ ràng buộc giữa hai đối tượng:  
+ BQL là bên cung cấp dịch vụ và kiểm soát thông tin, cư dân là bên sử dụng dịch vụ và thực hiện nghĩa vụ thanh toán.  
+ Cư dân chỉ được đăng ký, lập hoặc điều chỉnh hợp đồng sau khi BQL phê duyệt tài khoản.  
+ Mọi hóa đơn được BQL phát hành dựa trên dữ liệu hợp đồng mà cư dân đã xác nhận.  
+ BQL gửi thông báo, cư dân nhận và thực hiện các nghĩa vụ tương ứng.  
+ Cư dân không thể tự ý thay đổi điều khoản hợp đồng nếu chưa có sự đồng thuận từ BQL.  
Mối quan hệ này giúp đảm bảo minh bạch, hạn chế tranh chấp, đồng thời tăng cường tính chuyên nghiệp và trách nhiệm của cả hai bên.

## 9. Phương pháp nghiên cứu

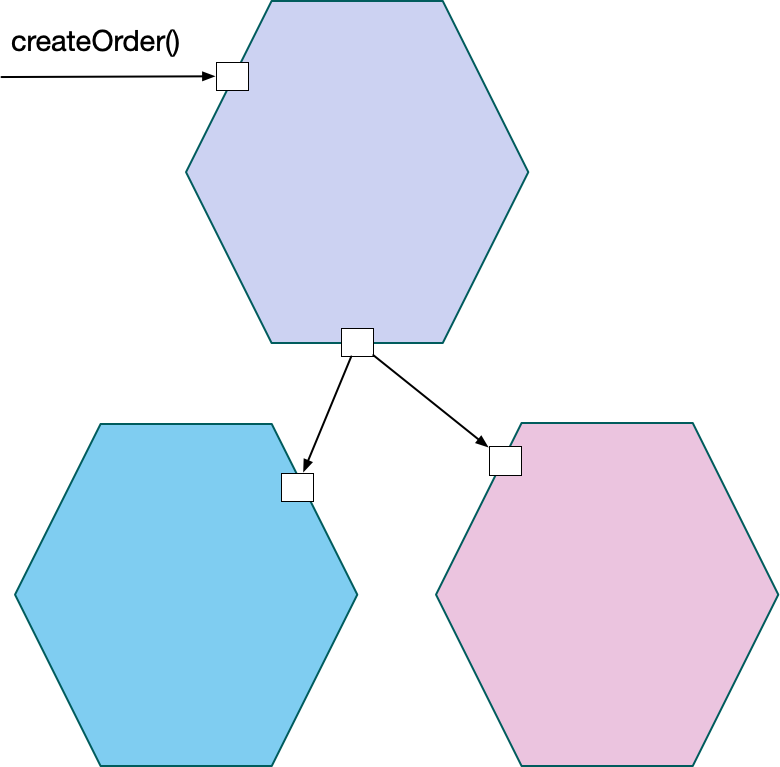
Đề tài được thực hiện thông qua phương pháp nghiên cứu kết hợp, bao gồm:

* **Nghiên cứu lý thuyết**: Tìm hiểu các kiến trúc phần mềm hiện đại, đặc biệt là kiến trúc Microservices, cùng các công nghệ như Spring Boot, RESTful APIs, MySQL, RabbitMQ/Kafka, Eureka và Spring Cloud Gateway. Ngoài ra, nghiên cứu các giải pháp giao tiếp bất đồng bộ và phương pháp tích hợp cổng thanh toán điện tử (VNPay sandbox).
* **Khảo sát thực tế**: Tìm hiểu nghiệp vụ quản lý hợp đồng dịch vụ tại một số chung cư lớn, các vấn đề thường gặp trong quy trình lập hợp đồng, quản lý hóa đơn và thanh toán. Thu thập các yêu cầu thực tế từ ban quản lý và nhu cầu của cư dân.
* **Phân tích và mô hình hoá**: Phân tích nghiệp vụ, xác định các chức năng chính, thiết kế sơ đồ kiến trúc tổng thể và mô hình dữ liệu, từ đó chia nhỏ hệ thống thành các vi dịch vụ độc lập.
* **Thử nghiệm và phát triển**: Xây dựng hệ thống demo, triển khai các chức năng chính, thử nghiệm quy trình lập hợp đồng, tạo hóa đơn và thanh toán (bao gồm mô phỏng thanh toán online). Đánh giá tính ổn định, khả năng mở rộng và mức độ đáp ứng yêu cầu nghiệp vụ.

Phương pháp tiếp cận này giúp đảm bảo hệ thống vừa đáp ứng được yêu cầu thực tiễn, vừa có tính khoa học, hiện đại và dễ dàng phát triển, mở rộng trong tương lai.

# Chương 2: Cơ sở lý thuyết

## 1. Kiến trúc Microservices



Hình 1: Ảnh mô tả kiến trúc Microservice

**Microservices — còn được gọi là kiến trúc microservice — là một kiểu kiến trúc phần mềm cấu trúc một ứng dụng thành tập hợp gồm hai hoặc nhiều dịch vụ mà:**

+ “Có thể triển khai độc lập”.

+ “Kết nối lỏng lẻo”.

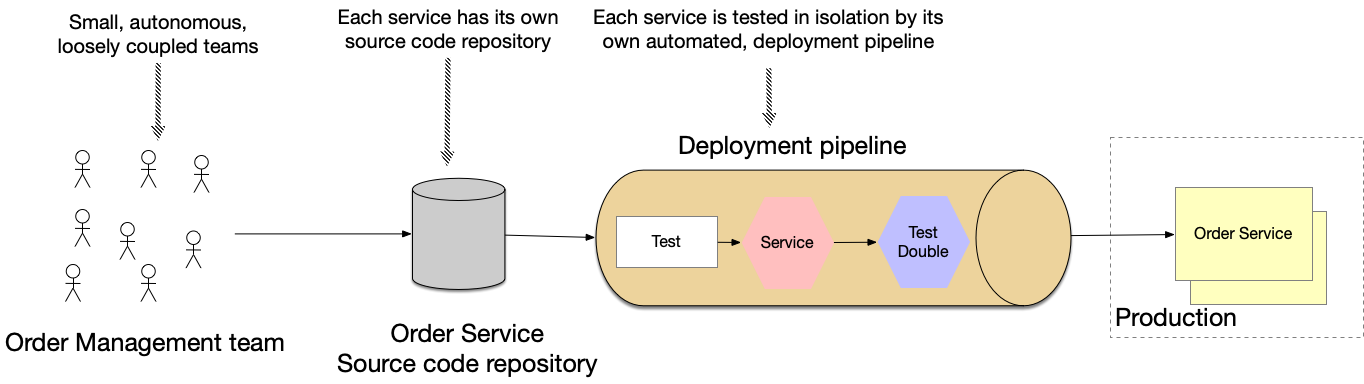
Các dịch vụ thường được tổ chức xoay quanh các năng lực nghiệp vụ. Mỗi dịch vụ thường được sở hữu bởi một nhóm nhỏ, riêng biệt.

### Khả năng triển khai độc lập:

* + 1. **Định nghĩa đơn giản về khả năng triển khai độc lập:**

Định nghĩa đơn giản của ‘independently deployable’ (khả năng triển khai độc lập) là một dịch vụ được đóng gói thành một đơn vị có thể triển khai hoặc có thể thực thi. Ví dụ về một đơn vị có thể triển khai hoặc thực thi bao gồm: tệp JAR có thể thực thi, tệp WAR, tệp thực thi của hệ điều hành, một image của Docker container, hoặc một tệp Zip định nghĩa một hàm AWS Lambda. Ngược lại, một tệp JAR thông thường cần được đóng gói cùng với các tệp JAR khác thì không phải là một đơn vị có thể triển khai hoặc thực thi. Mặc dù điều này đáp ứng định nghĩa theo đúng nghĩa đen của ‘independently deployable’, nó lại hoàn toàn không đủ.

* + 1. **Định nghĩa hoàn chỉnh về triển khai độc lập :**



Hình : Ảnh định nghĩa cách 1 microservice được coi là độc lập

**Dịch vụ triển khai độc lập** là một dịch vụ được đóng gói dưới dạng một đơn vị có thể triển khai hoặc thực thi được, và sẵn sàng cho môi trường sản xuất sau khi đã được kiểm thử độc lập. Một dịch vụ như vậy sẽ có kho mã nguồn và quy trình triển khai riêng. Quy trình triển khai này sẽ kiểm thử dịch vụ một cách độc lập bằng cách sử dụng **test doubles** (đối tượng kiểm thử) cho các cộng tác viên của nó, cùng với kiểm thử hợp đồng hướng đến người tiêu dùng (consumer+driven contract testing). Kết quả cuối cùng từ quy trình triển khai là một dịch vụ có thể và nên được triển khai vào môi trường sản xuất.

Nếu chúng ta cần kiểm thử dịch vụ của mình với các dịch vụ khác để xác minh rằng nó đã sẵn sàng cho sản xuất, thì nó không phải là dịch vụ triển khai độc lập. Hơn nữa, chúng ta có thể muốn cân nhắc đặt các dịch vụ đó vào một kho lưu trữ duy nhất. Điều này đảm bảo rằng đầu ra của một quy trình triển khai duy nhất thực sự sẵn sàng cho sản xuất và cũng loại bỏ sự phức tạp của việc phát triển trên nhiều kho lưu trữ.

Một lợi ích quan trọng của dịch vụ triển khai độc lập là nó giúp đẩy nhanh quy trình triển khai. Nó loại bỏ sự cần thiết của các bài kiểm thử end+to+end (kiểm thử đầu cuối) chậm chạp, dễ hỏng và phức tạp của nhiều dịch vụ. Nó cũng loại bỏ sự cần thiết của việc các nhóm phải phối hợp và có khả năng cản trở lẫn nhau.

* + 1. **Vấn đề phát sinh:**

Một trở ngại đối với việc triển khai các dịch vụ một cách độc lập chính là các bài kiểm thử chấp nhận người dùng (User Acceptance Tests + UAT) ở cấp độ hệ thống. Các bài kiểm thử chấp nhận này thường được viết từ góc nhìn của người dùng và thường bao trùm nhiều dịch vụ. Việc triển khai trực tiếp các bài kiểm thử như vậy sẽ đòi hỏi phải kiểm thử nhiều dịch vụ cùng lúc.

Để các dịch vụ có thể được triển khai độc lập, chúng ta phải thay thế các bài kiểm thử chấp nhận người dùng cấp hệ thống bằng các bài kiểm thử chấp nhận người dùng cấp độ dịch vụ.

* + 1. **Yêu cầu về đặc tả rõ ràng cho dịch vụ triển khai độc lập:**

Để một dịch vụ có thể được kiểm thử độc lập, nó cần có hai yếu tố chính:

+ Một đặc tả (specification) được định nghĩa rõ ràng.

+ Một bộ kiểm thử (test suite) xác minh rằng hành vi của nó tuân thủ đặc tả đó.

Nếu chúng ta không thể đặc tả chính xác hành vi của một dịch vụ, có lẽ chúng ta nên xem xét lại liệu nó có hợp lý để tồn tại như một dịch vụ riêng biệt hay không.

Các dịch vụ cũng phải hợp tác thông qua các API ổn định để tránh việc phải thay đổi đồng loạt một cách thường xuyên. Đây là một khía cạnh của đặc điểm khác của kiến trúc microservice – kết nối lỏng lẻo (loosely coupled) – mà tôi sẽ mô tả chi tiết hơn trong phần tiếp theo.

### Tính kết nối lỏng lẻo:

Thực tế có hai loại liên kết (coupling) khác nhau:

+ Liên kết thời gian chạy (runtime coupling) + ảnh hưởng đến tính khả dụng (availability).

+ Liên kết thời gian thiết kế (design+time coupling) + ảnh hưởng đến tốc độ phát triển (development velocity).

Giảm thiểu liên kết thời gian thiết kế và giảm thiểu liên kết thời gian chạy là hai trong số năm "lực lượng vật chất tối" (dark matter forces) định hình kiến trúc microservice.

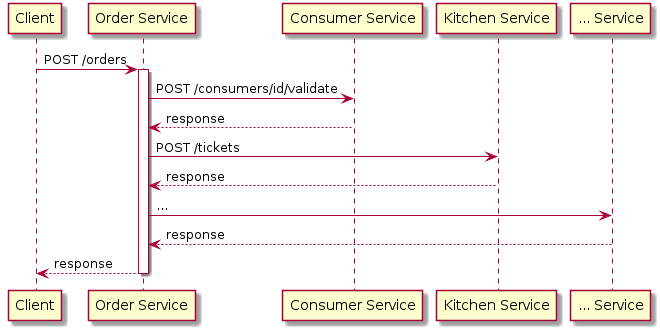
* + 1. **Liên kết thời gian chạy và tính khả dụng**

Liên kết thời gian chạy (runtime coupling) giữa các dịch vụ là mức độ mà tính khả dụng (availability) của một dịch vụ bị ảnh hưởng bởi tính khả dụng của một dịch vụ khác. Hay nói chính xác hơn, đó là mức độ mà tính khả dụng của một hoạt động được triển khai bởi một dịch vụ bị ảnh hưởng bởi tính khả dụng của một dịch vụ khác.

* + 1. **Liên kết thời gian chạy làm giảm tính khả dụng**

Ví dụ, hãy tưởng tượng rằng thao tác hệ thống createOrder() được triển khai bởi điểm cuối HTTP POST /orders trong Dịch vụ Đặt hàng (Order Service). Dịch vụ Đặt hàng xử lý yêu cầu HTTP POST bằng cách gọi các dịch vụ khác, chờ chúng phản hồi, rồi sau đó gửi phản hồi cho client của nó.

Trong thiết kế này, Dịch vụ Đặt hàng không thể phản hồi yêu cầu POST cho đến khi các dịch vụ khác phản hồi lại nó. Dịch vụ Đặt hàng (hoặc thao tác createOrder()) được gọi là có liên kết thời gian chạy với các dịch vụ khác đó. Kết quả là, tính khả dụng của thao tác createOrder() bị giảm đi vì tất cả các dịch vụ liên quan đều phải khả dụng.



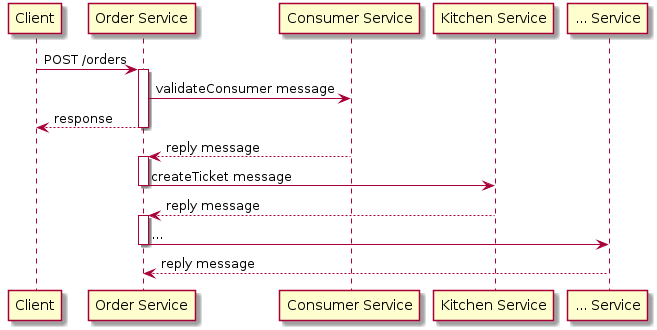
Hình : Ảnh minh họa cách liên kết thời gian chạy làm giảm tính sẵn sàng của service

* + 1. **Giảm thiểu liên kết thời gian chạy:**

## Giảm thiểu liên kết thời gian chạy

**Giảm thiểu liên kết thời gian chạy** là một trong những "lực hút của vật chất tối" chống lại việc phân tách. Một cách để giảm liên kết thời gian chạy của một thao tác là **giảm số lượng dịch vụ thực hiện nó**. Trên thực tế, chúng ta có thể loại bỏ hoàn toàn liên kết thời gian chạy bằng cách làm cho một thao tác trở thành cục bộ (local) đối với một dịch vụ duy nhất. Tuy nhiên, không phải lúc nào cũng có thể tạo ra một kiến trúc microservice mà tất cả các thao tác đều là cục bộ. Điều đó có thể vi phạm các "lực đẩy của năng lượng tối", vốn khuyến khích việc phân tách.

Cách khác để giảm liên kết thời gian chạy trong khi vẫn thỏa mãn các lực của năng lượng tối là **thiết kế các dịch vụ tự chứa (self+contained services)**. Một dịch vụ tự chứa sẽ phản hồi một yêu cầu đồng bộ với một kết quả cục bộ (partial outcome) và sau đó hoàn thành thao tác một cách bất đồng bộ. Ví dụ, Dịch vụ Đặt hàng (Order Service) có thể phản hồi yêu cầu HTTP POST /orders bằng một phản hồi 202 Accepted và sau đó bắt đầu một **"Create Order Saga"** để hoàn tất thao tác. Cách tiếp cận này cải thiện tính khả dụng của Dịch vụ Đặt hàng. Nhược điểm là nó làm cho client trở nên phức tạp hơn vì nó phải có khả năng xử lý các kết quả cục bộ và bằng cách nào đó xác định kết quả cuối cùng của thao tác.



Hình : Ảnh minh họa về Liên kết thời gian chạy (Runtime Coupling) trong quá trình tạo đơn hàng

* + 1. **Liên kết thời gian thiết kế và tốc độ phát triển:**

Mức độ liên kết thời gian thiết kế (design+time coupling) giữa một cặp các phần tử phần mềm – từ các lớp (classes) cho đến các dịch vụ (services) – là khả năng chúng cần phải thay đổi cùng nhau vì cùng một lý do. Liên kết thời gian thiết kế giữa các dịch vụ trong kiến trúc microservice đặc biệt có vấn đề.

* + - 1. **Liên kết thời gian thiết kế làm giảm tốc độ phát triển**

Nếu hai dịch vụ có liên kết lỏng lẻo (loosely coupled), thì một thay đổi ở dịch vụ này hiếm khi đòi hỏi một thay đổi ở dịch vụ kia. Tuy nhiên, nếu hai dịch vụ có liên kết chặt chẽ (tightly coupled), thì một thay đổi ở dịch vụ này thường đòi hỏi một thay đổi ở dịch vụ kia. Những loại thay đổi đồng bộ (lockstep changes) như vậy rất tốn kém vì chúng thường liên quan đến các thay đổi API gây hỏng (breaking API changes).

Ví dụ, hãy tưởng tượng rằng Dịch vụ Đặt hàng (Order Service) và Dịch vụ Khách hàng (Customer Service) có liên kết chặt chẽ. Mỗi khi cần thực hiện một thay đổi gây hỏng đối với Dịch vụ Khách hàng, chuỗi các bước sẽ như sau:

Thay đổi Dịch vụ Khách hàng để thêm một phiên bản chính mới của API. Dịch vụ này phải triển khai cả phiên bản API cũ và mới cho đến khi tất cả các client đã được di chuyển sang.

Di chuyển Dịch vụ Đặt hàng sang phiên bản API mới.

Loại bỏ phiên bản API cũ khỏi Dịch vụ Khách hàng.

Điều tồi tệ hơn nữa là, khá thường xuyên các dịch vụ này thuộc sở hữu của các đội khác nhau, điều này đòi hỏi các đội đó phải phối hợp các thay đổi. Nói cách khác, liên kết thời gian thiết kế giữa các dịch vụ làm suy yếu tính tự chủ của đội (team autonomy).

* + - 1. **Giảm thiểu liên kết thời gian thiết kế**

Giảm thiểu liên kết thời gian thiết kế là một trong những "lực hút của vật chất tối" chống lại việc phân tách. Có một vài cách khác nhau để giảm thiểu liên kết thời gian thiết kế giữa các dịch vụ:

+ Thiết kế các phân miền (subdomain) có liên kết lỏng lẻo: Các phân miền có liên kết lỏng lẻo có thể được đóng gói thành các dịch vụ khác nhau. Liên kết thời gian thiết kế lỏng lẻo thường đạt được bằng cách mỗi phân miền có một API ổn định giúp đóng gói việc triển khai bên trong của nó.

+ Đóng gói các phân miền có liên kết chặt chẽ vào cùng một dịch vụ: Nếu hai phân miền có liên kết chặt chẽ với nhau, thì việc đóng gói chúng lại cùng trong một dịch vụ sẽ tránh được liên kết thời gian thiết kế giữa các dịch vụ.

* + 1. **Ưu điểm:**

- Tăng khả năng mở rộng và linh hoạt cho hệ thống.  
- Dễ dàng phát triển, bảo trì và triển khai từng phần riêng biệt.  
- Cho phép sử dụng công nghệ khác nhau cho từng dịch vụ nếu cần.  
- Cải thiện tính chịu lỗi, tránh ảnh hưởng toàn hệ thống khi một dịch vụ gặp sự cố.

* + 1. **Nhược điểm:**

- Quản lý phức tạp hơn do nhiều service, nhiều cơ sở dữ liệu.  
- Yêu cầu hệ thống giám sát, logging và cơ chế giao tiếp giữa các dịch vụ phức tạp hơn.  
- Đòi hỏi thiết kế hạ tầng mạng, bảo mật, quản lý phiên bản tốt.

* + 1. **Mẫu thiết kế phổ biến:**

- **API Gateway Pattern**: Làm điểm truy cập duy nhất, hỗ trợ routing, authentication và load balancing.

- **Service Discovery Pattern**: Giúp các dịch vụ tự động tìm thấy nhau mà không cần cấu hình thủ công.

- **Circuit Breaker Pattern**: Ngăn chặn lỗi lan rộng khi dịch vụ downstream bị lỗi.

- **Database per Service**: Mỗi dịch vụ quản lý database riêng, tránh phụ thuộc lẫn nhau.

- **Event-driven Pattern**: Các dịch vụ giao tiếp bằng sự kiện (event), giúp tách biệt, giảm kết dính.

## 2. Spring Boot

Spring Boot là framework nổi bật trong hệ sinh thái Spring, giúp phát triển các ứng dụng Java dễ dàng, nhanh chóng và cấu hình tối giản.  
  
Các kiến thức cần nắm vững:  
+ Cách khởi tạo dự án Spring Boot, sử dụng auto+configuration và embedded servers (Tomcat, Jetty...).  
+ Các starter dependencies giúp giảm cấu hình, chỉ cần thêm đúng starter sẽ có đầy đủ tính năng.  
+ Sự khác biệt giữa Spring Boot và Spring MVC: Spring Boot hỗ trợ chạy độc lập, tự động cấu hình; còn Spring MVC chỉ tập trung vào mô hình MVC trong Spring Framework.  
+ Cách xây dựng các ứng dụng độc lập, có thể chạy ngay bằng Spring Boot.

## 3. RESTful APIs

RESTful APIs được sử dụng để giao tiếp giữa các dịch vụ hoặc giữa frontend và backend.  
  
Các kiến thức cơ bản:  
+ Nguyên tắc thiết kế REST: dựa trên tài nguyên (Resource+based), Stateless (không lưu trạng thái), giao diện đồng nhất (Uniform Interface).  
+ Sử dụng các phương thức HTTP: GET, POST, PUT, DELETE để thực hiện các thao tác CRUD (Create, Read, Update, Delete).  
+ Định dạng dữ liệu trao đổi phổ biến: JSON.

## 4. MySQL

MySQL là hệ quản trị cơ sở dữ liệu quan hệ phổ biến, dễ sử dụng và được hỗ trợ rộng rãi.  
  
Các điểm cần lưu ý:  
+ Kiến thức cơ bản về mô hình dữ liệu quan hệ, cách thiết kế bảng, quan hệ và ràng buộc dữ liệu.  
+ Thiết kế schema riêng cho từng vi dịch vụ, đảm bảo mỗi service có cơ sở dữ liệu độc lập (Database per service).  
+ Tích hợp MySQL với Spring Boot, sử dụng Spring Data JPA để thao tác dữ liệu dễ dàng.

## 5. RabbitMQ hoặc Kafka (Message Broker)

RabbitMQ hoặc Kafka giúp xây dựng cơ chế giao tiếp bất đồng bộ giữa các dịch vụ, giúp hệ thống trở nên linh hoạt và ít phụ thuộc hơn.  
  
Các nội dung chính:  
+ Hiểu cơ chế giao tiếp bất đồng bộ (asynchronous communication).  
+ Nguyên lý hoạt động: Producer, Consumer, Queue (RabbitMQ) hoặc Topic (Kafka).  
+ Cách sử dụng RabbitMQ (hoặc Kafka) để gửi và nhận thông báo/sự kiện giữa các dịch vụ, giúp giảm độ kết dính trực tiếp.

## 6. Eureka (Discovery Service)

Service Discovery cho phép các dịch vụ tự động tìm thấy nhau mà không cần cấu hình địa chỉ thủ công.  
  
Các khái niệm cần nắm:  
+ Vai trò của Eureka Server và Eureka Client trong kiến trúc phân tán.  
+ Cách các vi dịch vụ đăng ký địa chỉ của mình lên Eureka Server.  
+ Cách các dịch vụ khác tra cứu và giao tiếp với nhau thông qua Eureka.

## 7. Spring Cloud Gateway (API Gateway)

Spring Cloud Gateway đóng vai trò là điểm truy cập duy nhất (Single Entry Point) của toàn bộ hệ thống.  
  
Vai trò và ứng dụng:  
+ Định tuyến (routing) yêu cầu từ người dùng đến các dịch vụ backend.  
+ Hỗ trợ xử lý bảo mật (authentication, authorization), cân bằng tải (load balancing), ghi log.  
+ Cách cấu hình Spring Cloud Gateway để kiểm soát luồng yêu cầu và bảo vệ hệ thống nội bộ.

Service: thông tin người đùng và tài khoản + hợp đồng và dịch vụ + hóa đơn + thanh toán (thủ công và vnpay) + thông báo tự động

Chương 3: