

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC

ĐỒ ÁN TỐT NGHIỆP

Kiến trúc microservice và vận dụng triển khai hệ thống đấu giá trực tuyến

PHẠM ANH ĐỨC

duc.pa195859@sis.hust.edu.vn

Ngành Toán Tin

Giảng viên hướng dẫn: TS. Vũ Thành Nam

Bộ môn:

Toán Tin

Viện:

Toán ứng dụng và Tin học

Chữ ký của GVHD

HÀ NỘI, 7/2023

ĐỒ ÁN TỐT NGHIỆP

NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN

1. Mục đích và nội dung của đồ án:

.....

.....

.....

.....

2. Kết quả đạt được:

.....

.....

.....

.....

3. Ý thức làm việc của sinh viên:

.....

.....

.....

.....

Hà Nội, ngày tháng năm

Giáo viên hướng dẫn

Ký và ghi rõ họ tên

ĐỒ ÁN TỐT NGHIỆP

Lời cảm ơn

Trong suốt quá trình làm đồ án, em đã nhận được sự giúp đỡ và hỗ trợ nhiệt tình từ thầy Vũ Thành Nam. Nhờ những kiến thức và kinh nghiệm của thầy, em đã có thể hiểu rõ hơn về các khái niệm về kiến trúc microservice và kỹ thuật thiết kế, xây dựng và triển khai hệ thống dựa trên kiến trúc microservice mà điển hình là hệ thống quản lý đơn từ. Ngoài ra, thầy cũng đã giúp em giải quyết những vấn đề và thắc mắc trong quá trình thực hiện đồ án. Em rất cảm ơn thầy đã giúp em không chỉ hoàn thành đồ án mà còn nâng cao kiến thức và kỹ năng của mình.

Tóm tắt nội dung đồ án

Đồ án giới thiệu về ứng dụng thân thiện đám mây, các yêu cầu của ứng dụng thân thiện đám mây và microservice. Để giúp phát triển và triển khai microservice, quy trình phát triển, triển khai liên tục, container, container orchestration các mẫu thiết kế microservice cũng được đề cập trong đồ án. Để áp dụng kiến trúc microservice cho bài toán thực tế, cách thiết kế service theo Domain driven design được sử dụng để lựa chọn, phân tách các nghiệp vụ. Cuối cùng, đồ án đã triển khai thử nghiệm hệ thống xây dựng theo kiến trúc microservice trên Kubernetes cho kết quả khả quan.

Sinh viên thực hiện

Ký và ghi rõ họ tên

MỤC LỤC

Phần mở đầu	5
Chương 1 Ứng dụng thân thiện đám mây	6
1.1. Containers	7
1.2. Kiến trúc microservice	8
1.3. Mô hình logic kiến trúc microservice	10
1.4. Một số mẫu thiết kế microservice	11
1.5. Container Orchestration	14
1.6. CI/CD	15
1.7. Domain Driven Design (DDD)	15
Chương 2 Phân tích thiết kế hệ thống đấu giá trực tuyến	18
2.1. Mô tả nghiệp vụ đấu giá	18
2.2. Thiết kế các service dựa trên Domain Driven Design	21
2.3. Mô hình logic hệ thống	22
2.4. Thiết kế các service	23
2.5. Thiết kế giao diện người dùng	26
Chương 3 Triển khai hệ thống	29
3.1. Lựa chọn các cấu phần kỹ thuật	29
3.1. Triển khai CI/CD	39
3.2. Triển khai sử dụng Docker & Docker Compose	40
3.3. Triển khai sử dụng Kubernetes	43
Kết luận	50
Tài liệu tham khảo	51

DANH SÁCH HÌNH VẼ

Hình 1. So sánh kiến trúc monolith và microservice	8
Hình 2. Mô hình logic kiến trúc microservice cơ bản.....	11
Hình 3. So sánh hai cách gọi API	12
Hình 4. Truyền thông đồng bộ và bất đồng bộ giữa các service.....	14
Hình 5. Nghiệp vụ đấu giá truyền thống	20
Hình 6. Mô hình logic hệ thống	22
Hình 7. Giao diện trang chính để xem các sản phẩm đang đấu giá	27
Hình 8. Giao diện tạo phiên đấu giá và tạo sản phẩm đấu giá	27
Hình 9. Giao diện chi tiết cuộc đấu giá.....	28
Hình 10. Khảo sát các thư viện, framework phổ biến để phát triển giao diện web.	29
Hình 11. Kiến trúc máy ảo và docker	31
Hình 12. Minh họa API của auction service trên Postman	34
Hình 13. Kiến trúc Kubernetes.....	38
Hình 14. Hình ảnh triển khai liên tục trên Github Actions	40
Hình 15. Các service khi được docker hóa thành các image	41

DANH SÁCH BẢNG

Bảng 1. Thiết kế dữ liệu cho user service	23
Bảng 2. Thiết kế API cho user service.....	23
Bảng 3. Thiết kế dữ liệu cho product service.....	24
Bảng 4. Thiết kế API cho product service	24
Bảng 5. Thiết kế dữ liệu cho auction service	24
Bảng 6. Thiết kế dữ liệu cho auction service (tiếp theo)	25
Bảng 7. Thiết kế API cho auction service.....	25

Phần mở đầu

Hiện nay, công nghệ thông tin tác động đến sự phát triển của tất cả các lĩnh vực, từ kinh tế, văn hoá, chính trị, xã hội cho đến giáo dục. Xu hướng số hóa mọi hoạt động, thủ tục đã trở thành một xu hướng được trong thế giới hiện đại, giúp tăng năng suất, tiết kiệm thời gian và giảm công sức của con người. Việc sử dụng công nghệ thông tin giúp cho các hoạt động giao dịch diễn ra nhanh chóng, tiết kiệm thời gian và chi phí, đồng thời giảm thiểu tình trạng sai sót và mất mát thông tin. Cùng với đó là sự phát triển về các công nghệ, kỹ thuật giúp tăng hiệu năng, tiết kiệm chi phí và tài nguyên tính toán cũng như đảm bảo tính ổn định cho hệ thống. Một trong những nền tảng của kỹ thuật này là hệ thống phân tán và điển hình là kiến trúc microservice. Để tìm hiểu ưu nhược điểm của microservice và những phương pháp, cách thực hiện triển khai xung quanh kiến trúc này, tôi đã tìm hiểu và áp dụng kiến trúc để xây dựng và triển khai hệ thống đấu giá trực tuyến.

Hệ thống được xây dựng trên công nghệ:

- Microsoft ASP.NET Core: Xây dựng các API.
- VueJS: Xây dựng giao diện ứng dụng web.
- Microsoft SQL Server: Hệ quản trị cơ sở dữ liệu.
- Docker: Xây dựng các containers.
- Kubernetes: Quản lý việc hoạt động giữa các containers
- Github Actions: Giúp đưa CI/CD vào quy trình phát triển ứng dụng
- RabbitMQ: Hỗ trợ trao đổi thông điệp giữa các services

Bố cục của đồ án gồm 3 phần chính:

- Phần 1: Ứng dụng thân thiện đám mây
- Phần 2: Phân tích và triển khai hệ thống đấu giá trực tuyến
- Phần 3: Triển khai hệ thống

Chương 1 Ứng dụng thân thiện đám mây

Điện toán đám mây (cloud computing) là một xu hướng công nghệ nổi bật trên thế giới trong những năm gần đây và đã có những bước phát triển nhảy vọt cả về chất lượng, quy mô cung cấp và loại hình dịch vụ, với một loạt các nhà cung cấp nổi tiếng như Google, Amazon, Salesforce, Microsoft,... Điện toán đám mây là mô hình điện toán mà mọi giải pháp liên quan đến công nghệ thông tin đều được cung cấp dưới dạng các dịch vụ qua mạng Internet, giải phóng người sử dụng khỏi việc phải đầu tư nhân lực, công nghệ và hạ tầng để triển khai hệ thống. Từ đó điện toán đám mây giúp tối giản chi phí và thời gian triển khai, tạo điều kiện cho người sử dụng nền tảng điện toán đám mây tập trung được tối đa nguồn lực vào công việc chuyên môn. Lợi ích của điện toán đám mây mang lại không chỉ gói gọn trong phạm vi người sử dụng nền tảng điện toán đám mây mà còn từ phía các nhà cung cấp dịch vụ điện toán. Theo định nghĩa của Viện Quốc gia Tiêu chuẩn và Công nghệ Mỹ (US NIST), điện toán đám mây là mô hình cho phép truy cập trên mạng tới các tài nguyên được chia sẻ (ví dụ: hệ thống mạng, máy chủ, thiết bị lưu trữ, ứng dụng và các dịch vụ) một cách thuận tiện và theo nhu cầu sử dụng. Những tài nguyên này có thể được cung cấp một cách nhanh chóng hoặc thu hồi với chi phí quản lý tối thiểu hoặc tương tác tối thiểu với nhà cung cấp dịch vụ.

Nhờ vào các đặc điểm và lợi ích này, ứng dụng cloud native trở thành một xu hướng quan trọng trong việc xây dựng ứng dụng hiện đại và hỗ trợ sự chuyển đổi và tối ưu hóa trong môi trường đám mây.

Một tập hợp các nguyên tắc, phương pháp phát triển ứng dụng thân thiện đám mây được giới thiệu bởi Cloud Native Computing Foundation (CNCF) trong dự án Cloud Native Landscape. Phương pháp này gồm 12 ý chính:

1. **Code Base:** Mỗi microservice có một mã nguồn riêng, được lưu trữ trong kho mã nguồn của riêng nó. Được theo dõi bằng hệ thống quản lý phiên bản, có thể triển khai lên nhiều môi trường (Staging, Production).
2. **Dependencies:** Mỗi microservice cô lập và đóng gói các phụ thuộc riêng của nó, giúp chấp nhận các thay đổi mà không ảnh hưởng đến toàn bộ hệ thống.
3. **Configurations:** Thông tin cấu hình được di chuyển ra khỏi microservice và được quản lý bên ngoài mã nguồn thông qua một công cụ quản lý cấu hình. Cùng một phiên bản triển khai có thể tự động áp dụng các cấu hình khác nhau cho các môi trường khác nhau.
4. **Backing Services:** Các tài nguyên phụ (lưu trữ dữ liệu, bộ đệm, message broker) nên được tiếp cận thông qua một URL có địa chỉ. Việc này giúp phân tách tài nguyên từ ứng dụng, giúp nó có thể thay thế và thích ứng với các tài nguyên khác.
5. **Build, Release, Run:** Mỗi phiên bản triển khai phải tuân thủ nguyên tắc chia rõ ràng giữa giai đoạn xây dựng (build), phát hành (release) và chạy (run). Mỗi phiên

ĐỒ ÁN TỐT NGHIỆP

- bản nên được đánh dấu bằng một ID duy nhất và hỗ trợ khả năng quay trở lại phiên bản trước. Các hệ thống CI/CD hiện đại hỗ trợ nguyên tắc này.
6. **Processes:** Mỗi microservice nên thực thi trong một quá trình riêng biệt, cách ly với các dịch vụ khác đang chạy.
 7. **Port Binding:** Mỗi microservice nên tự chứa các giao diện giao tiếp và chức năng của nó trên một cổng riêng biệt. Việc này cung cấp sự cách ly giữa các microservice khác nhau.
 8. **Concurrency:** Khi cung tăng, cần mở rộng hệ thống theo chiều ngang, tăng các instance của một service.
 9. **Disposability:** Instance của các service có khả năng hủy bỏ dễ dàng.
 10. **Dev/Prod Parity:** Đảm bảo các môi trường ổn định, tương đồng trong quá trình phát triển, triển khai sản phẩm.
 11. **Logging:** Đảm bảo việc ghi log, event để dễ dàng sửa lỗi trong giai đoạn triển khai.
 12. **Admin Process:** Tách rời các hoạt động quản trị ra thành một tiến trình riêng để đảm bảo ranh giới giữa việc quản trị và thực thi ứng dụng.

Những trụ cột này giúp định hình triết lý và phương pháp xây dựng ứng dụng cloud native thành công, tối ưu hóa tính mở rộng và tính sẵn sàng của ứng dụng, đồng thời giúp đơn giản hóa việc triển khai và quản lý hệ thống đám mây.

1.1. Containers

Container là một cách tiếp cận để phát triển phần mềm. Một ứng dụng hoặc service cùng các thư viện và cấu hình được đóng gói cùng nhau dưới dạng image. Ứng dụng trong image có thể được unit test và có thể triển khai dễ dàng trên hệ điều hành máy chủ.

Giống như việc vận chuyển hàng hóa bằng container, container có thể được vận chuyển bằng tàu hỏa, tàu thủy, xe công-te-nơ mà không quan tâm hàng hóa bên trong container là gì. Container phần mềm cũng vậy, chúng bao gồm mọi thứ cần thiết để triển khai độc lập trên đa môi trường mà ít cần thay đổi nhiều (về code, configuration).

Các container cũng tạo ra sự phân tách giữa ứng dụng và hệ điều hành chúng chạy trên. Container khi chạy cũng sử dụng ít tài nguyên hơn máy ảo. Container mang lại sự cách ly, sự di động, dễ dàng mở rộng và điều khiển trong quy trình phát triển và triển khai ứng dụng. Có lẽ ý nghĩa quan trọng nhất là sự tách biệt môi trường phát triển Dev và Ops.

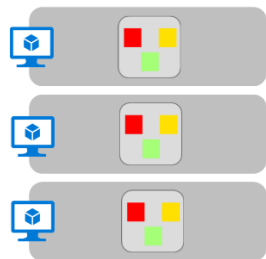
Container cũng giúp đáp ứng yêu tố **2. Dependencies** và **10. Dev/Prod Parity** trong 12 yếu tố quan trọng nêu trên.

1.2. Kiến trúc microservice

Trước khi kiến trúc microservice trở nên phổ biến, các doanh nghiệp thường tổ chức các phần mềm, ứng dụng của họ dựa trên kiến trúc monolith. Ứng dụng theo kiến trúc monolith (monolith application) là kiểu ứng dụng 1 tier trong đó các mô-đun khác nhau được kết hợp thành một chương trình duy nhất. Ví dụ: khi xây dựng một ứng dụng thương mại điện tử, thì ứng dụng đó phải có kiến trúc mô-đun với nhiều mô-đun khác nhau phục vụ cho nghiệp vụ mà ứng dụng đó đáp ứng. Trong một ứng dụng monolith, các mô-đun được xác định bằng cách sử dụng các gói (ví dụ như Java package) và các thư viện ngoài (ví dụ như Java JAR).

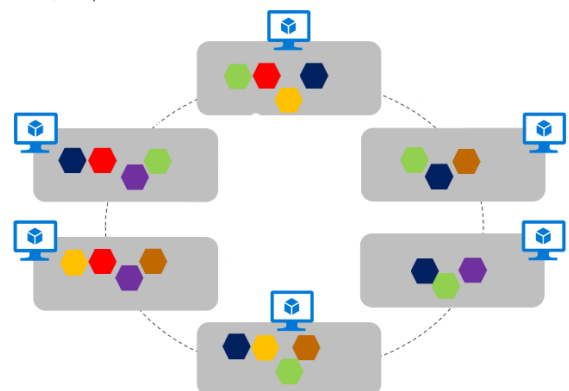
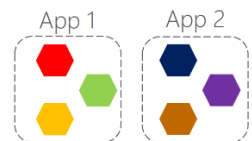
Triển khai kiểu Monolithic

- Phương pháp truyền thống chạy ứng dụng trên một vài tiến trình và đóng gói trong các lớp và thư viện
- Scale bằng cách chạy cả ứng dụng trên các máy chủ, máy ảo.



Triển khai kiểu Microservice

- Một ứng dụng theo kiến trúc Microservice thường phân tách chức năng dưới dạng các service thành phần.
- Scale bằng cách triển khai từng service độc lập trên các máy chủ, máy ảo



Hình 1. So sánh kiến trúc monolith và microservice

Một số ưu điểm của ứng dụng monolith có thể kể đến như:

- Có thể xây dựng kiểm thử đầu cuối (end-to-end testing) bằng các phần mềm như Selenium.
- Cách triển khai dễ dàng chỉ với việc copy những phần đã được build lên máy chủ triển khai (deployment server).
- Tất cả các module trong một ứng dụng monolith đều chạy trên cùng một nhóm process do đó có thể chia sẻ bộ nhớ và tài nguyên, do đó dễ dàng sử dụng các công cụ như logging, caching và áp dụng các tiêu chuẩn bảo mật.
- Có thể cung cấp tốc độ, khả năng tính toán tốt do các module có thể gọi nhau trực tiếp thay vì sử dụng các phương thức gọi từ xa.

ĐỒ ÁN TỐT NGHIỆP

Tuy vậy, xây dựng ứng dụng theo kiến trúc monolith lại có những thách thức:

- Sau khi ứng dụng đã phát triển, kích thước ứng dụng trở nên rất lớn do đó thời gian, tốc độ build sẽ chậm, ứng dụng trở nên khó gỡ lỗi (debug) do có quá nhiều thành phần phụ thuộc nhau, một phần mã nguồn thay đổi có thể ảnh hưởng lớn đến cả hệ thống. Do đó thời gian phát triển, thời gian đưa ra bản cập nhật mới thường kéo dài.
- Các module chạy trên cùng tiến trình, do đó nếu xảy ra lỗi (ví dụ như memory leak) thì cả hệ thống sẽ gặp nguy hiểm.
- Khi muốn áp dụng đổi mới sáng tạo vào hệ thống, ví dụ như sử dụng các ngôn ngữ hay framework, thư viện mới sẽ gây áp lực về mặt thời gian và tiền bạc để viết lại toàn bộ hệ thống cũ.

Đối với hệ thống xây dựng theo kiến trúc microservice, mỗi microservice thường triển khai một tập hợp các tính năng hoặc chức năng riêng biệt. Mỗi microservice là một ứng dụng nhỏ có kiến trúc và logic nghiệp vụ riêng. Kiến trúc microservices thay đổi mối quan hệ giữa ứng dụng và cơ sở dữ liệu. Thay vì chia sẻ một cơ sở dữ liệu duy nhất với các dịch vụ khác, mỗi service có một cơ sở dữ liệu riêng.

Kiến trúc microservice giải quyết một số hạn chế của kiến trúc monolith, các ưu điểm của kiến trúc microservice có thể kể đến như:

- Có thể tách toàn bộ hệ thống thành các phần nhỏ, dễ quản lý, đáp ứng được nghiệp vụ đề ra. Thường các service sẽ được phân chia theo phương pháp hướng nghiệp vụ (Domain Driven Design). Mỗi service có một vùng hoạt động riêng, trao đổi với các service khác thông qua cơ chế trao đổi thông điệp (ví dụ như RPC). Do đó mỗi service có thể được phát triển nhanh chóng, dễ hiểu, dễ bảo trì.
- Có thể phân chia các team để phát triển từng service độc lập với nhau, dễ dàng kiểm thử, theo dõi.
- Có thể áp dụng các ngôn ngữ, framework khác nhau phù hợp với nghiệp vụ, tạo thành một ứng dụng kiểu polygot. Ví dụ sử dụng python cho các nghiệp vụ có sử dụng trí tuệ nhân tạo như xác định giả mạo khuôn mặt, phân loại khách hàng nhưng cũng sử dụng các ngôn ngữ và framework hỗ trợ mạnh cho các ứng dụng doanh nghiệp như ASP.NET hay Java Spring.
- Kiến trúc microservice cho phép người vận hành mở rộng quy mô của hệ thống độc lập với nhau. Giả sử chỉ có một vài tác vụ, nghiệp vụ quá tải (ví dụ như tác vụ thanh toán), người vận hành có thể chỉ định mở rộng service thanh toán lên nhiều instance hơn và giữ các service khác để tối ưu việc sử dụng phần cứng, do đó tiết kiệm chi phí và tối ưu hóa lợi nhuận.

Tuy vậy, kiến trúc microservice cũng có một số mặt hạn chế:

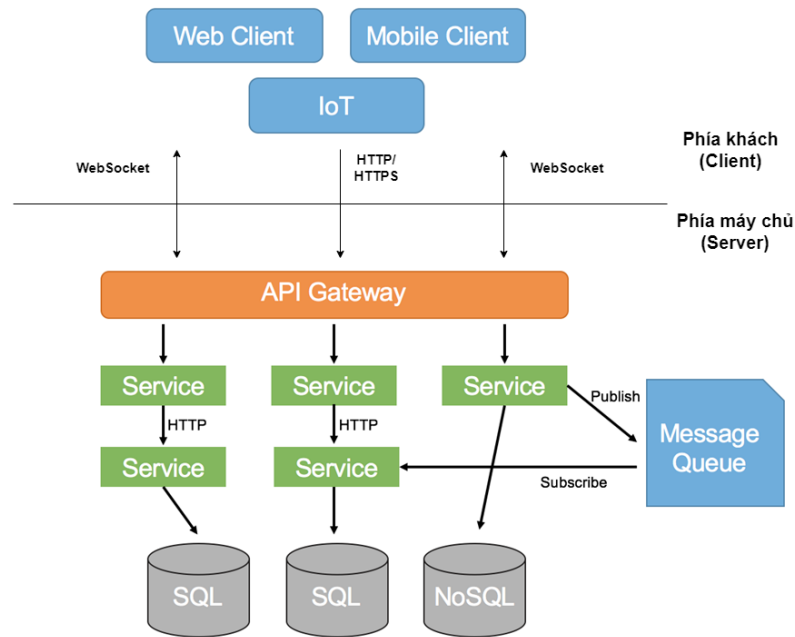
ĐỒ ÁN TỐT NGHIỆP

- Khó khăn chính khi áp dụng kiến trúc microservice là sự phức tạp của hệ thống phân tán. Người phát triển ứng dụng dựa trên kiến trúc này phải chọn phương thức trao đổi phù hợp giữa các service và phải xử lý các lỗi gặp phải trong quá trình truyền nhận. Ví dụ như lỗi mất, lỗi bản tin, service tham gia trong quá trình giao tiếp không tồn tại (bị sập).
- Một khó khăn khác của kiến trúc này là việc xử lý giao dịch trên các service (distributed transaction). Việc cập nhật các thực thể khi thực thi một nghiệp vụ nào đó thường xuyên diễn ra và thường chúng là các thao tác đơn (atomic), nghĩa là khi một thao tác lỗi thì giao dịch sẽ không được thực hiện. Việc này thường được đảm bảo bởi chức năng xử lý giao dịch có sẵn của cơ sở dữ liệu khi sử dụng kiến trúc monolith có một cơ sở dữ liệu. Tuy vậy trong kiến trúc microservice, mỗi service thường có cơ sở dữ liệu riêng, một giao dịch có thể liên đới đến nhiều service, do đó việc xử lý lỗi giao dịch giữa các service là việc khó khăn (việc đảm bảo rollback đúng, consistent giữa các service).
- Việc triển khai hệ thống microservice cũng gây ra không ít khó khăn. Một ứng dụng thiết kế theo kiến trúc microservice thường có nhiều service và khi triển khai có một hoặc nhiều instance (để đảm bảo tính ổn định và sẵn sàng – High Availability). Do đó người vận hành cần phải triển khai các phương thức nhận diện service (Service Discovery Mechanism) để cung cấp địa chỉ của 1 instance của một service cho service khác.

1.3. Mô hình logic kiến trúc microservice

Một mô hình kiến trúc microservice cơ bản thường bao gồm:

- Các Service. Những service này có thể được viết bằng đa dạng các ngôn ngữ, framework khác nhau nhưng chỉ cần chung phương thức giao tiếp. Kiến trúc kiểu vậy được gọi là kiến trúc Polygot.
- Message Queue: Để đáp ứng nhu cầu truyền thông điệp giữa các service và tăng khả năng chịu tải.
- Database: Có thể sử dụng các loại cơ sở dữ liệu quan hệ hoặc phi quan hệ
- API Gateway: Là một mẫu thiết kế giúp trao đổi thông tin giữa client và hệ thống.
- Client: là người dùng hệ thống sử dụng các thiết bị, trình duyệt khác nhau.



Hình 2. Mô hình logic kiến trúc microservice cơ bản

1.4. Một số mẫu thiết kế microservice

Một phương pháp quản lý dữ liệu cơ bản trong kiến trúc microservice là việc mỗi service có một database (database per service pattern). Lý do dẫn đến việc này là:

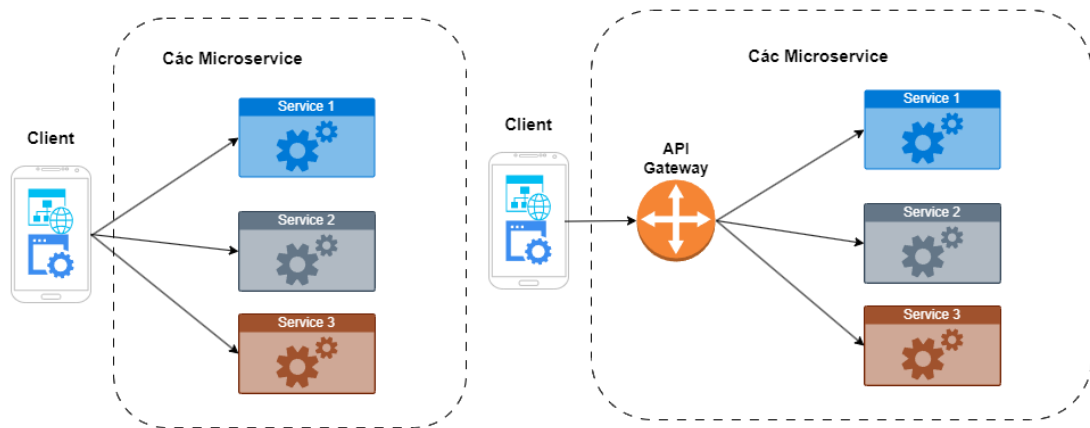
- Các services phải không phụ thuộc chặt chẽ để có thể phát triển, triển khai và mở rộng độc lập với nhau.
- Một vài nghiệp vụ yêu cầu phải lấy dữ liệu mà được sở hữu bởi nhiều services.
- Một vài nghiệp vụ cần join dữ liệu từ nhiều service.
- Database cần được nhân bản và phân mảnh để có thể mở rộng tốt hơn.

Khi áp dụng mẫu thiết kế này:

- Database chỉ có thể được truy cập trực tiếp bởi chính service sở hữu nó
- Tốc độ truy vấn dữ liệu trong service tốt hơn.
- Khó hỗ trợ transaction. Để sử dụng transaction cần áp dụng mẫu thiết kế Saga.
- Xây dựng toán tử join dữ liệu cho các database ở các service là việc khó khăn.

1.4.1. External API

Để đáp ứng nhu cầu sử dụng API của client, hệ thống cần cung cấp các endpoint để client truy cập.



Hình 3. So sánh hai cách gọi API

Gọi API trực tiếp

Một nghiệp vụ phía client thường có liên đới giữa nhiều service, do đó để thực hiện nghiệp vụ, client cần tạo API request đến nhiều service. Cách gọi trực tiếp này có thể phù hợp cho các ứng dụng ít service. Tuy vậy trong ứng dụng cần thời gian xử lý nhanh, cách này lại có một số hạn chế. Hạn chế chính là việc phải gọi quá nhiều service để hiển thị một UI nào đó, nhất là trong môi trường internet, việc gọi nhiều lần sẽ tăng độ trễ, ứng dụng chậm xử lý thao tác, do đó không thân thiện với người sử dụng. Một hạn chế khác là một số phương thức như AMQP và các phương thức sử dụng mã nhị phân như gRPC chưa hỗ trợ rộng rãi ở phía client, các request cũng đều phải chuyển về HTTP/HTTPS để thân thiện hơn với phía client. Ngoài ra việc đảm bảo an toàn, bảo mật cho tất cả các service cũng là vấn đề khó khăn khi các service được truy cập từ bên ngoài.

Cách gọi thông qua API Gateway

Từ những hạn chế của phương pháp trao đổi trực tiếp giữa client và service, API Gateway được đề xuất và là kiểu thiết kế tiêu biểu cho mẫu External API. Mẫu thiết kế API Gateway cung cấp cho phía client một địa chỉ duy nhất cho một nhóm các service có liên quan đến nhau. API Gateway còn có thể cung cấp các tính năng như caching hay TLS/SSL cho phép các service và API Gateway giao tiếp nội bộ sử dụng HTTP và giao tiếp giữa API Gateway với client sử dụng HTTPS. Thêm vào đó, tính năng xác thực có thể được cài đặt ở API Gateway, giữa API Gateway và các service không cần xác thực, giảm tính phức tạp của hệ thống.

Tuy vậy API gateway cũng có những hạn chế nhất định, điều dễ thấy nhất ở kiến trúc này đó chính là việc chính Gateway có thể là điểm nghẽn, bị quá tải và không thể xử lý các request. Việc này có thể tránh bằng cách chia nhỏ các gateway cho từng nhóm các service có liên quan tới nhau.

ĐỒ ÁN TỐT NGHIỆP

1.4.2. Service Discovery

Services thường cùng hoạt động để hoàn thành một nghiệp vụ nào đó. Trong kiến trúc monolith, các service có thể gọi lẫn nhau qua các method một cách dễ dàng. Trong kiến trúc phân tán cổ điển, các services thường có số lượng instance nhất định, chạy trên các host, port định sẵn do đó dễ dàng gọi nhau sử dụng các giao thức phổ biến như HTTP hay RPC. Tuy vậy trong kiến trúc microservice hiện đại, thân thiện với đám mây, các service thường chạy lúc cao tải và giảm bớt số lượng instance lúc không có nhiều tải với mục đích tiết kiệm chi phí, lúc này việc tìm kiếm các service đang chạy là một việc không dễ. Do đó cần có cách để một service biết địa chỉ của service khác để gọi.

Trong mẫu thiết kế Service Discovery có ba mẫu chính:

- Client-side discovery
- Server-side discovery
- Self-registered

1.4.3. Kiểu giao tiếp

Trong một ứng dụng monolith chạy trên một tiến trình duy nhất, các thành phần gọi lẫn nhau bằng cách sử dụng các lệnh gọi hàm hoặc phương thức (Local Procedure Call – LPC). Giữa các thành phần có thể có ràng buộc chặt (strongly coupled) nếu khởi tạo một object mới hoặc có ràng buộc lỏng hơn (loosely coupled) khi nhận đối tượng bằng tham số truyền vào (sử dụng Dependency Injection). Dù bằng cách nào, các đối tượng đều chạy trong cùng một tiến trình. Thách thức lớn nhất khi chuyển đổi từ ứng dụng monolith sang ứng dụng dựa trên microservice nằm ở việc thay đổi cơ chế giao tiếp giữa các thành phần. Chuyển đổi trực tiếp từ LPC thành lệnh gọi Remote Procedure Call (RPC) giữa các thành phần sẽ gây ra sự thiếu hiệu quả trong môi trường hệ thống phân tán.

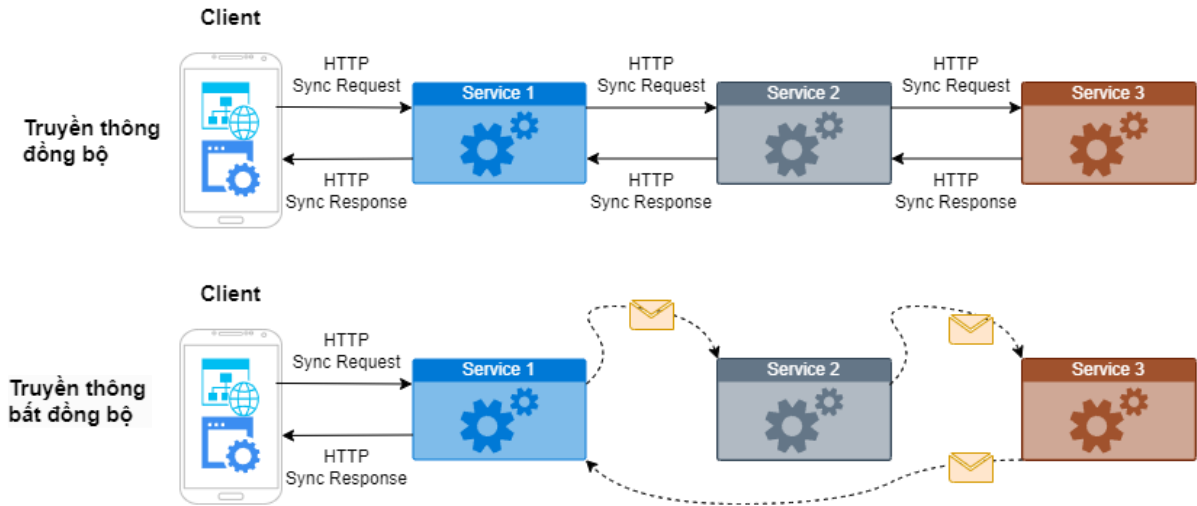
Có một vài giải pháp thường được đề cập khi thiết kế hệ thống truyền thông trong hệ thống phân tán. Một là cô lập các nghiệp vụ microservice càng nhiều càng tốt, điều này sẽ giúp hạn chế việc dịch vụ này cần gọi dịch vụ khác để thực thi nghiệp vụ. Sau đó, sử dụng giao tiếp không đồng bộ giữa các microservice nội bộ. Cuối cùng là tổng hợp lại thông tin từ một vài dịch vụ cần thiết rồi trả về client.

Hai giao thức thường được sử dụng là truyền thông đồng bộ (ví dụ HTTP) và truyền thông điệp không đồng bộ (ví dụ AMQP).

- HTTP là một giao thức đồng bộ. Client gửi yêu cầu và chờ phản hồi từ service. Điểm quan trọng ở đây là client chỉ có thể tiếp tục nhiệm vụ của mình khi nhận được phản hồi của máy chủ HTTP.
- Các giao thức khác như AMQP (một giao thức được hỗ trợ bởi nhiều hệ điều hành và môi trường đám mây) là không đồng bộ. Client gửi tin nhắn đến service và

ĐỒ ÁN TỐT NGHIỆP

thường không đợi phản hồi. Thông điệp thường được gửi đến một broker và được cho vào hàng đợi như RabbitMQ.



Hình 4. Truyền thông đồng bộ và bất đồng bộ giữa các service

1.5. Container Orchestration

Trong môi trường phân tán, việc quản lý và điều phối các container trở nên phức tạp hơn. Các container có thể chạy trên nhiều máy chủ và cần được phân phối cân bằng tải, điều khiển và theo dõi một cách hiệu quả. Sẽ rất khó khăn nếu một nhóm người vận hành hệ thống gồm vài trăm đến vài chục nghìn container. Từ hạn chế này, container orchestration được ra đời.

Container orchestration là quá trình quản lý, triển khai và điều phối các container trong một môi trường ứng dụng phân tán. Nó giúp tổ chức và quản lý các container trong một hệ thống phân tán lớn, đảm bảo sự linh hoạt, khả năng mở rộng và độ tin cậy cao.

Các hệ thống container orchestration giúp tự động hóa việc triển khai, mở rộng và quản lý các ứng dụng được đóng gói trong container. Các công cụ container orchestration như Kubernetes (K8s) cung cấp các tính năng như quản lý tài nguyên, điều phối các ứng dụng, tự động phục hồi khi có lỗi, tự động mở rộng và cân bằng tải tài nguyên.

Container orchestration giúp đơn giản hóa việc triển khai và quản lý ứng dụng trong môi trường phân tán. Nó cho phép tự động hóa các tác vụ quản lý container như triển khai, cập nhật, sao lưu và khôi phục, theo dõi và quản lý tài nguyên.

Container orchestration cũng hỗ trợ việc phân phối tải tài nguyên, giúp cân bằng tải các container và đảm bảo hiệu suất cao cho ứng dụng. Nó cung cấp khả năng mở rộng linh hoạt, cho phép thêm hoặc giảm số lượng container để đáp ứng nhu cầu tải.

1.6. CI/CD

CI/CD là phương pháp phát hành bản cập nhật mới thường xuyên cho khách hàng bằng cách đưa tự động hóa vào các giai đoạn phát triển ứng dụng. CI/CD thường được hiểu là tích hợp liên tục, phân phối liên tục và triển khai liên tục. CI/CD là một giải pháp cho các vấn đề bổ sung code mới có thể gây ra lỗi cho quá trình phát triển và vận hành.

"CI" trong CI/CD luôn đề cập đến việc tích hợp liên tục, đây là quy trình tự động hóa dành cho nhà phát triển. Áp dụng CI có nghĩa là các thay đổi code mới cho một ứng dụng thường xuyên, thử nghiệm và tích hợp vào một kho lưu trữ dùng chung (git repository). Đây là một giải pháp cho vấn đề có quá nhiều branch đang phát triển cùng lúc dễ dẫn đến xung đột.

"CD" trong CI/CD đề cập đến phân phối liên tục và/hoặc triển khai liên tục, là những khái niệm đôi khi được sử dụng thay thế cho nhau. Cả hai đều nói về việc tự động hóa các giai đoạn tiếp theo của quy trình phát triển phần mềm, nhưng đôi khi chúng được sử dụng riêng để minh họa mức độ tự động hóa đang diễn ra.

Phân phối liên tục thường có nghĩa là các thay đổi của nhà phát triển đối với ứng dụng sẽ tự động được kiểm tra lỗi và tải lên kho lưu trữ (như GitHub), sau đó nhóm vận hành có thể triển khai vào môi trường production. Cuối cùng, mục đích của việc phân phối liên tục là để đảm bảo rằng việc triển khai mã mới tốn ít công sức.

Triển khai liên tục (có thể gọi là "CD") có thể đề cập đến việc tự động phát hành các thay đổi của nhà phát triển từ repository tới môi trường production. Nó giải quyết vấn đề phải thực hiện thủ công quá trình triển khai ứng dụng. Nó được xây dựng dựa trên những lợi ích của việc phân phối liên tục bằng cách tự động hóa giai đoạn tiếp theo trong quy trình.

1.7. Domain Driven Design (DDD)

Theo Microsoft, Microservices nên được thiết kế dựa trên nghiệp vụ và khả năng của nghiệp vụ đó, không nên thiết kế ứng dụng microservice theo lớp như lớp truy cập dữ liệu hoặc lớp truyền thông. Ngoài ra, các service không được phụ thuộc chặt với nhau (loosely coupled) và có tính độc lập chức năng. Một microservice được gọi là độc lập chức năng nó có một mục đích duy nhất, được xác định rõ ràng, ví dụ như quản lý tài khoản người dùng hoặc theo dõi lịch sử chuyển hàng hóa. Một dịch vụ nên đóng gói các tri thức về nghiệp vụ và trong suốt về nghiệp vụ đó. Ví dụ: khách hàng có thể đặt vé máy bay mà không cần biết thuật toán tạo vé, lập lịch vé.

ĐỒ ÁN TỐT NGHIỆP

Domain Driven Design (DDD) cung cấp một khung giúp người phát triển hệ thống thiết kế đúng, chuẩn các microservice. DDD có hai giai đoạn, strategic và tactical. Strategic DDD tập trung thiết kế cấu trúc một hệ thống lớn. Tactical DDD giúp đảm bảo rằng kiến trúc tập trung vào quá trình kinh doanh, giải quyết vấn đề của hệ thống. Tactical DDD cung cấp một tập hợp các mẫu thiết kế có thể sử dụng để tạo domain. Các mẫu này bao gồm các thực thể, các dịch vụ trong domain. Các mẫu chiến thuật này sẽ giúp thiết kế các dịch vụ siêu nhỏ vừa kết hợp lỏng lẻo vừa gắn kết.

Một số nguyên tắc của DDD:

1. Ngôn ngữ phong cách phản ánh: Dùng ngôn ngữ chung và rõ ràng để mô tả các khái niệm trong miền kinh doanh. Sử dụng thuật ngữ và ngữ cảnh mà các chuyên gia trong miền đó sử dụng.
2. Phân lớp miền: Xác định các lớp trong miền, bao gồm Entity (đối tượng), Value Object (đối tượng giá trị), Aggregate (tập hợp đối tượng), Repository (kho lưu trữ), và Service (dịch vụ). Các lớp này được sắp xếp sao cho phù hợp với các quy tắc và quy định của miền.
3. Aggregate: Aggregate là một nhóm các đối tượng liên quan với nhau và được coi là một đối tượng duy nhất trong quá trình thực thi. Aggregate bảo vệ tính toàn vẹn của dữ liệu và quản lý xử lý các yêu cầu liên quan.
4. Repository: Repository làm nhiệm vụ cung cấp giao diện để truy xuất và lưu trữ dữ liệu trong miền. Nó cung cấp một cách trừu tượng để tương tác với cơ sở dữ liệu.
5. Ubiquitous Language (Ngôn ngữ phổ biến): Đảm bảo rằng cả nhóm phát triển và các thành viên liên quan trong dự án sử dụng cùng một ngôn ngữ chung để thảo luận và hiểu các khái niệm trong miền.

Trong Domain-Driven Design (DDD), việc chọn các dịch vụ (services) phải dựa trên phân tích và hiểu rõ về miền ngữ cảnh của ứng dụng. Các dịch vụ nên được xác định để phục vụ các khái niệm và quy trình kinh doanh quan trọng trong miền đó. Một số cách chọn dịch vụ theo DDD như:

1. Xác định quy trình kinh doanh quan trọng: ví dụ như hoạt động, luồng công việc, quy trình tương tác với người dùng, và các quy tắc kinh doanh quan trọng khác.
2. Xác định các khái niệm trong miền: Phân tích và xác định các đối tượng (entities), đối tượng giá trị (value objects), tập hợp đối tượng (aggregates), hoặc các yếu tố quan trọng khác trong miền.
3. Xác định phạm vi và trách nhiệm của mỗi dịch vụ: Dựa trên các khái niệm và quy trình kinh doanh đã xác định, hãy xác định phạm vi và trách nhiệm của từng dịch vụ. Mỗi dịch vụ nên đảm nhận một phạm vi rõ ràng và chỉ xử lý các nhiệm vụ liên quan đến phạm vi đó.
4. Tránh việc tạo ra dịch vụ quá lớn: Hạn chế kích thước của mỗi dịch vụ bằng cách tập trung vào một phạm vi cụ thể. Điều này giúp giữ cho mỗi dịch vụ nhỏ gọn, dễ quản lý và hiểu được.

ĐỒ ÁN TỐT NGHIỆP

5. Thiết kế giao diện dịch vụ (Service Interface Design): Thiết kế giao diện dịch vụ dựa trên các yêu cầu và khái niệm của miền. Giao diện dịch vụ nên phản ánh rõ ràng các hành vi và hoạt động quan trọng liên quan đến miền kinh doanh.

Chương 2 Phân tích thiết kế hệ thống đấu giá trực tuyến

2.1. Mô tả nghiệp vụ đấu giá

2.1.1. Nghiệp vụ đấu giá truyền thống

Đấu giá kiểu Anh hay đấu giá tăng dần là hình thức được nhiều người biết đến nhất. Phương thức này phổ biến nhất trong các cuộc đấu giá, tại đó những người mua đưa ra các mức giá chào mua bằng việc hô giá công khai (open outcry) theo trình tự tăng dần cho đến khi không có mức giá chào mua nào cao hơn được đưa ra. Mức giá chào mua cao nhất sau cùng này sẽ được người tổ chức đấu giá chấp nhận.

Tùy theo quy định của từng sàn đấu giá mà mỗi lần trả người tham gia đấu giá phải nâng giá món hàng đấu giá lên ít nhất là bao nhiêu.

Theo quy định của pháp luật Việt Nam, cụ thể, căn cứ *Điều 40 Luật Đấu giá tài sản 2016* có quy định các hình thức và phương thức đấu giá:

1. Tổ chức đấu giá tài sản thỏa thuận với người có tài sản đấu giá lựa chọn một trong các hình thức sau đây để tiến hành cuộc đấu giá:

- a) Đấu giá trực tiếp bằng lời nói tại cuộc đấu giá;
- b) Đấu giá bằng bỏ phiếu trực tiếp tại cuộc đấu giá;
- c) Đấu giá bằng bỏ phiếu gián tiếp;
- d) Đấu giá trực tuyến.

2. Phương thức đấu giá bao gồm:

- a) Phương thức trả giá lên;
- b) Phương thức đặt giá xuống.

3. Hình thức đấu giá, phương thức đấu giá phải được quy định trong Quy chế cuộc đấu giá và công bố công khai cho người tham gia đấu giá biết.

4. Chính phủ quy định chi tiết điểm d khoản 1 Điều này.

Nghiệp vụ đấu giá bỏ phiếu trực tiếp tại cuộc đấu giá được Điều 42 Luật Đấu giá tài sản 2016 quy định được trích dưới đây:

2. Việc trả giá trong trường hợp đấu giá theo phương thức trả giá lên được thực hiện như sau:

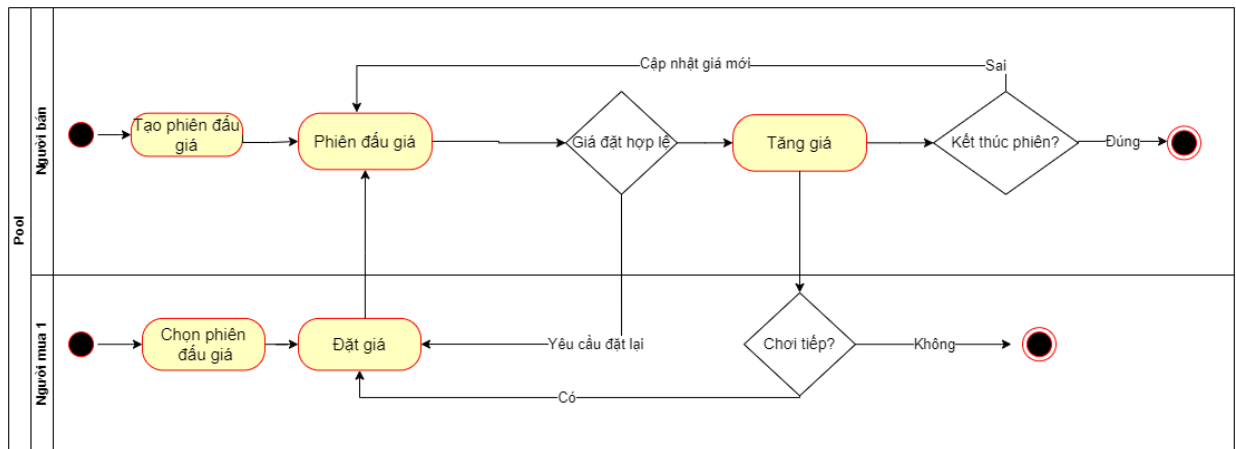
ĐỒ ÁN TỐT NGHIỆP

- a) Người tham gia đấu giá được phát một tờ phiếu trả giá, ghi giá muốn trả vào phiếu của mình. Hết thời gian ghi phiếu, đấu giá viên yêu cầu người tham gia đấu giá nộp phiếu trả giá hoặc bỏ phiếu vào hòm phiếu; kiểm đếm số phiếu phát ra và số phiếu thu về; công bố từng phiếu trả giá, phiếu trả giá cao nhất với sự giám sát của ít nhất một người tham gia đấu giá;
- b) Đấu giá viên công bố giá cao nhất đã trả của vòng đấu giá đó và đề nghị người tham gia đấu giá tiếp tục trả giá cho vòng tiếp theo. Giá khởi điểm của vòng đấu giá tiếp theo là giá cao nhất đã trả ở vòng đấu giá trước liền kề;
- c) Cuộc đấu giá kết thúc khi không còn ai tham gia trả giá. Đấu giá viên công bố người trả giá cao nhất và công bố người đó là người trúng đấu giá;
- d) Trường hợp có từ hai người trở lên cùng trả mức giá cao nhất, đấu giá viên tổ chức đấu giá tiếp giữa những người cùng trả giá cao nhất để chọn ra người trúng đấu giá. Nếu có người trả giá cao nhất không đồng ý đấu giá tiếp hoặc không có người trả giá cao hơn thì đấu giá viên tổ chức bốc thăm để chọn ra người trúng đấu giá.

3. Việc chấp nhận giá trong trường hợp đấu giá theo phương thức đặt giá xuống được thực hiện như sau:

- a) Người tham gia đấu giá được phát một tờ phiếu chấp nhận giá, ghi việc chấp nhận, giá khởi điểm mà đấu giá viên đưa ra vào phiếu của mình. Hết thời gian ghi phiếu, đấu giá viên yêu cầu người tham gia đấu giá nộp phiếu chấp nhận giá hoặc bỏ phiếu vào hòm phiếu; kiểm đếm số phiếu phát ra và số phiếu thu về;
 - b) Đấu giá viên công bố việc chấp nhận giá của từng người tham gia đấu giá với sự giám sát của ít nhất một người tham gia đấu giá;
 - c) Đấu giá viên công bố người chấp nhận giá khởi điểm và công bố người đó là người trúng đấu giá. Trường hợp không có người nào chấp nhận giá khởi điểm thì đấu giá viên công bố mức giảm giá và tiến hành việc bỏ phiếu với mức giá đã giảm;
 - d) Trường hợp có từ hai người trở lên cùng chấp nhận giá khởi điểm hoặc giá đã giảm thì đấu giá viên tổ chức bốc thăm để chọn ra người trúng đấu giá.
4. Người có tài sản đấu giá và tổ chức đấu giá tài sản thỏa thuận cách thức tiến hành bỏ phiếu và số vòng đấu giá quy định tại khoản 2, khoản 3 Điều này."

Mô hình cơ bản của nghiệp vụ đấu giá truyền thống được mô tả theo biểu đồ dưới đây:



Hình 5. Nghiệp vụ đấu giá truyền thống

Do hình thức đấu giá truyền thống yêu cầu người mua và người bán đều tham dự cùng một phiên đấu giá tại một địa điểm cụ thể, vì vậy phiên đấu giá chưa mang tính tiện lợi trong thời đại số. Do đó cần có hình thức đấu giá dễ dàng hơn cho cả người bán và người mua.

2.1.2. Nghiệp vụ đấu giá trực tuyến

Từ nghiệp vụ đấu giá truyền thống, tôi mô tả các bước trong nghiệp vụ đấu giá trực tuyến:

1. Đăng ký và xác thực: Người dùng muốn tham gia đấu giá trực tuyến cần đăng ký tài khoản trên hệ thống. Họ cung cấp thông tin cá nhân và thông tin liên quan khác và hoàn thành quy trình xác thực nếu có. Điều này giúp hệ thống dễ dàng quản lý cuộc đấu giá, người tham gia đặt giá
2. Đăng bài đấu giá: Người bán đăng thông tin về sản phẩm hoặc dịch vụ mà họ muốn đấu giá. Thông tin này bao gồm mô tả chi tiết, ảnh, giá khởi điểm và các điều kiện đấu giá khác. Người bán cũng có thể thiết lập thời gian kết thúc đấu giá và các quy tắc đặc biệt khác nếu cần thiết.
3. Nhận lượt đặt giá: Người mua tham gia đấu giá bằng cách đặt giá cho sản phẩm hoặc dịch vụ mà họ quan tâm. Họ đặt giá bằng hoặc cao hơn giá hiện tại và bước tăng. Hệ thống sẽ ghi lại lượt đặt giá và hiển thị giá cao nhất hiện tại cho tất cả người dùng xem.
4. Theo dõi và nâng giá: Trong suốt quá trình đấu giá, người mua có thể theo dõi tình trạng của đấu giá và có thể nâng giá khi cần thiết để giữ vị trí dẫn đầu. Họ có thể nhận thông báo và cập nhật về các lượt đặt giá mới nhất và có thể thực hiện các hành động tương ứng.
5. Kết thúc đấu giá: Khi thời gian đấu giá kết thúc, hệ thống sẽ xác định người mua có giá cao nhất và công bố người chiến thắng. Sau đó, người mua và người bán liên hệ để hoàn tất quá trình thanh toán và giao hàng dựa trên các điều kiện đấu giá được thiết lập trước đó.

ĐỒ ÁN TỐT NGHIỆP

6. Đánh giá và phản hồi: Sau khi giao dịch hoàn tất, người mua và người bán có thể đánh giá và phản hồi về nhau. Điều này cung cấp thông tin quan trọng về độ tin cậy và chất lượng của người dùng tham gia đấu giá trong tương lai.

2.2. Thiết kế các service dựa trên Domain Driven Design

Trong nghiệp vụ đấu giá có một vài khái niệm chính như:

1. Người dùng (User): Đại diện cho các người dùng tham gia đấu giá, bao gồm người bán và người mua.
2. Sản phẩm (Product): Đại diện cho các mặt hàng được đấu giá.
3. Đấu giá (Auction): Quy trình đấu giá, bao gồm việc tạo đấu giá, đặt giá, và xác định người chiến thắng.

Dựa trên các khái niệm và quy trình kinh doanh trên, tôi xác định các dịch vụ sau:

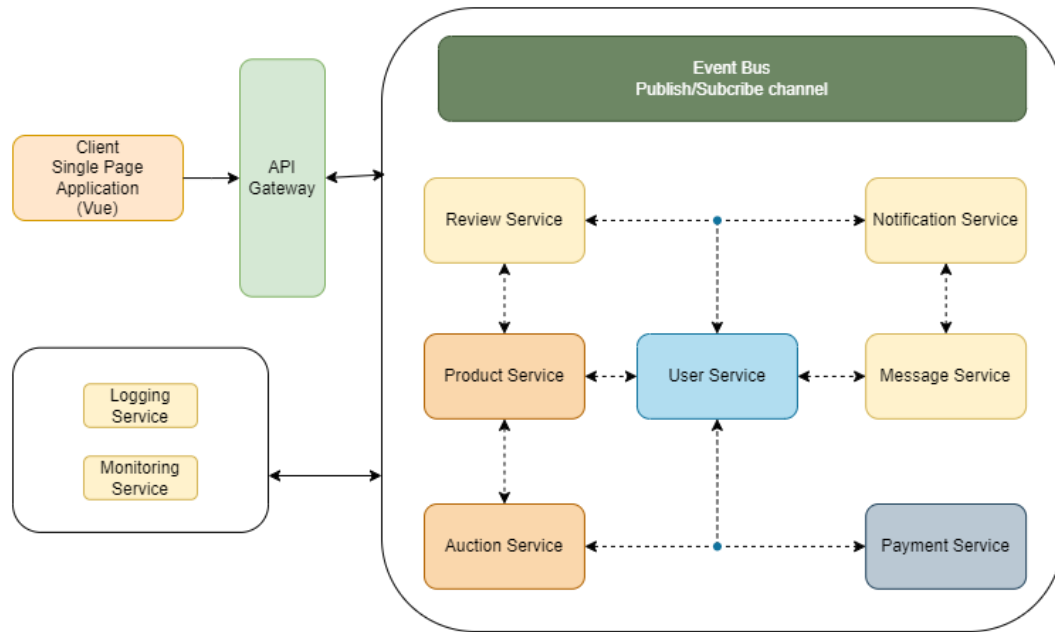
1. User Service: Dịch vụ này quản lý thông tin người dùng, bao gồm việc đăng ký, xác thực, cập nhật thông tin người dùng, và quản lý quyền truy cập.
2. Product Service: Dịch vụ này quản lý thông tin về sản phẩm được đấu giá, bao gồm việc tạo sản phẩm mới, cập nhật thông tin sản phẩm, và xóa sản phẩm.
3. Auction Service: Dịch vụ này quản lý quá trình đấu giá, bao gồm việc tạo mới phiên đấu giá, đặt giá, xác định người chiến thắng, và cập nhật trạng thái đấu giá.
4. Payment Service: Dịch vụ này xử lý quá trình thanh toán cho các giao dịch đấu giá thành công. Nó có thể giao tiếp với các hệ thống thanh toán bên ngoài để xử lý các giao dịch thanh toán và cung cấp thông tin về trạng thái thanh toán.

Ngoài ra để hệ thống hoạt động với đầy đủ chức năng của một hệ thống công nghệ thông tin, tôi bổ sung thêm một số service như:

5. NotificationService: Quản lý thông báo cho người dùng
6. MessageService: Quản lý việc nhắn tin giữa người dùng hệ thống

2.3. Mô hình logic hệ thống

Mô hình logic hệ thống được xây dựng gồm Client giao tiếp với hệ thống thông qua API Gateway. Khi client cần giao tiếp với service nào, API Gateway sẽ đóng vai trò trung gian giữa client và service đó. Các service trong hệ thống cũng có thể giao tiếp với nhau thông qua kết nối gián tiếp nhờ vào Event Bus. Những kết nối gián tiếp được minh họa bởi đường nét đứt. Để hệ thống hoạt động ổn định, tôi cũng bổ sung thêm thiết kế về giám sát và ghi lại log.



Chú giải

←→ Kết nối trực tiếp

←- - - -> Kết nối gián tiếp (qua Message Queue)

Hình 6. Mô hình logic hệ thống

ĐỒ ÁN TỐT NGHIỆP

2.4. Thiết kế các service

Để độc lập về dữ liệu cũng như thuận tiện cho việc phát triển và triển khai, tôi áp dụng mẫu thiết kế Database Per Service. Bên cạnh việc thiết kế database, việc thiết kế API cho từng service cũng rất quan trọng, giúp đảm bảo rằng các microservice có thể tương tác và làm việc cùng nhau một cách hiệu quả.

2.4.1. User service

Thiết kế về dữ liệu:

Bảng 1. Thiết kế dữ liệu cho user service

User Service		
Tên trường	Diễn giải	Kiểu dữ liệu
Id	Mã người dùng	Guid/Text
Name	Họ và tên	Text
Email	Địa chỉ thư điện tử	Text
PhoneNumber	Số điện thoại	Text
Address	Địa chỉ nhà/cơ quan	Text
Role	Quyền	Integer

Thiết kế API:

Bảng 2. Thiết kế API cho user service

UserService				
STT	Tên	Mô tả	Dữ liệu vào	Dữ liệu trả về
1	AddAnUser	Thêm mới người dùng vào hệ thống.	name, email, phoneNumber, password, address	–
2	Login	Đăng nhập vào hệ thống	email, password	trạng thái, token.
3	GetUserById	Lấy thông tin người dùng theo id	id, token	thông tin người dùng

2.4.2. Product Service

Thiết kế về dữ liệu:

ĐỒ ÁN TỐT NGHIỆP

Bảng 3. Thiết kế dữ liệu cho product service

Product Service		
Tên trường	Diễn giải	Kiểu dữ liệu
ProductId	Mã sản phẩm	Integer
ProductName	Tên sản phẩm	Text
Stock	Số lượng trong kho	Integer
Image	Nơi lưu ảnh	Text
OwnerId	Người sở hữu	Guid/Text

Thiết kế API:

Bảng 4. Thiết kế API cho product service

ProductService				
STT	Tên	Mô tả	Dữ liệu vào	Dữ liệu trả về
1	AddAProduct	Thêm mới sản phẩm vào hệ thống.	productName, stock image, ownerId	trạng thái: thành công/thất bại
2	ListProduct	Lấy danh sách các sản phẩm	–	danh sách các sản phẩm
3	GetProductBy UserId	Lấy danh sách sản phẩm theo người dùng	userId	danh sách các sản phẩm

2.4.3. Auction Service

Thiết kế về dữ liệu

Bảng 5. Thiết kế dữ liệu cho auction service

Auction Service: Auction		
Tên trường	Diễn giải	Kiểu dữ liệu
Id	Mã cuộc đấu giá	Integer
Name	Tên cuộc đấu giá	Text
StartDate	Ngày bắt đầu	Datetime
EndDate	Ngày kết thúc	Datetime
OwnerId	Người sở hữu	Text
IncrementStep	Bước tăng sau mỗi lần đặt giá	Float

ĐỒ ÁN TỐT NGHIỆP

InitialPrice	Giá khởi điểm	Float
ProductId	Mã sản phẩm đấu giá	Integer
LastBidPrice	Giá cuối cùng tại thời điểm gọi truy vấn	Float

Bảng 6. Thiết kế dữ liệu cho auction service (tiếp theo)

Auction Service: Bidding		
Tên trường	Diễn giải	Kiểu dữ liệu
Id	Mã lần đấu giá	Integer
UserId	Người tham gia đấu giá	Text
AuctionId	Mã cuộc đấu giá	Integer
ActionDate	Thời điểm đấu giá	Datetime
BiddingPrice	Số tiền đấu giá	Float

Thiết kế API

Bảng 7. Thiết kế API cho auction service

AuctionService				
STT	Tên	Mô tả	Dữ liệu vào	Dữ liệu trả về
1	GetAllAuctions	Lấy danh sách các cuộc đấu giá	–	danh sách các cuộc đấu giá
2	GetAuctionsBy UserId	Lấy danh sách các cuộc đấu giá theo người dùng	userId	danh sách các cuộc đấu giá
3	GetAnAuction	Lấy thông tin chi tiết cuộc đấu giá	auctionId	thông tin chi tiết cuộc đấu giá
4	AddAnAuction	Tạo cuộc đấu giá mới	name, ownerId, productId, startDate, endDate, initialPrice, incrementStep	trạng thái: thành công/thất bại
5	AddABidding	Đặt giá vào cuộc đấu giá	userId, auctionId, biddingPrice	trạng thái: thành công/thất bại
6	GetABidding	Xem chi tiết cuộc đặt giá	biddingId	lần đặt giá

ĐỒ ÁN TỐT NGHIỆP

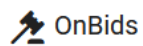
7	GetBiddingsBy AuctionId	Lấy danh sách những lần đặt giá theo cuộc đấu giá	auctionId	các lần đặt giá
8	GetBiddingsBy AuctionId AndUserId	Lấy danh sách những lần đặt giá theo cuộc đấu giá và theo user. API này phục vụ mục đích xem lịch sử lần đấu giá của user	auctionId, user	các lần đặt giá

2.5. Thiết kế giao diện người dùng

Giao diện người dùng (User Interface - UI) là phần của hệ thống mà người dùng tương tác trực tiếp để sử dụng các chức năng và tương tác với dữ liệu. Giao diện người dùng bao gồm các thành phần như màn hình, nút bấm, trường nhập liệu, danh sách, biểu đồ và các phần tử tương tác khác, nhằm cung cấp trải nghiệm tương tác tốt cho người dùng. Dựa trên nghiệp vụ đã được phân tích ở trên, tôi thiết kế các giao diện sau:

1. **BiddingDetail:** Giao diện này được sử dụng để xem chi tiết về cuộc đấu giá đang diễn ra.
2. **CreateAuction:** Giao diện này cho phép người dùng tạo một cuộc đấu giá mới.
3. **Home:** Đây là trang chủ, trang LandingPage hiển thị các cuộc đấu giá đang diễn ra để người dùng dễ dàng theo dõi.
4. **MyProduct:** Giao diện này hiển thị danh sách các sản phẩm mà người dùng đang sở hữu, giúp quản lý sản phẩm một cách thuận tiện.
5. **MyAuction:** Giao diện này hiển thị danh sách các cuộc đấu giá do người dùng tạo và quản lý.
6. **SignIn:** Trang đăng nhập vào hệ thống, cho phép người dùng đăng nhập vào tài khoản cá nhân.
7. **SignUp:** Trang tạo tài khoản mới trong hệ thống, giúp người dùng đăng ký và tạo tài khoản cá nhân.
8. **Payment:** Giao diện này để người dùng tạo thanh toán trên hệ thống

ĐỒ ÁN TỐT NGHIỆP



Hình 7. Giao diện trang chính để xem các sản phẩm đang đấu giá



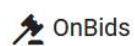
The screenshot shows the "Tạo phiên đấu giá" (Create Bidding Round) form. It has a back arrow at the top left. The form fields are: "Tên phiên đấu giá" (Bidding Round Name), "Ngày bắt đầu" (Start Date) with a date picker icon, "Ngày kết thúc" (End Date) with a date picker icon, "Giá khởi điểm" (Starting Price), and "Bước tăng giá" (Bid Increment). At the bottom, there are two buttons: "CHỌN SẢN PHẨM" (Select Product) and "TẠO" (Create).



The screenshot shows the "Tạo sản phẩm" (Create Product) form. It has a back arrow at the top left. The form fields are: "Tên sản phẩm" (Product Name), "Số lượng" (Quantity), and "Ảnh sản phẩm (upload ảnh rồi điền link)" (Product Image (upload image then fill link)). At the bottom, there are two buttons: "ĐÓNG" (Close) in red and "TẠO" (Create) in green. Below the form, there are buttons for "CHỌN SẢN PHẨM" (Select Product) and "TẠO" (Create).


Hình 8. Giao diện tạo phiên đấu giá và tạo sản phẩm đấu giá

ĐỒ ÁN TỐT NGHIỆP



<>

Thử nghiệm



Mã sản phẩm: 1
Tên sản phẩm: Máy ảnh Canon 5D Classic

Bắt đầu: 11/7/2023
Kết thúc: 14/7/2023

Giá khởi điểm: 100 VND Bước giá: 1 VND Giá hiện tại: 100 VND

Đặt giá Hủy

Hình 9. Giao diện chi tiết cuộc đấu giá

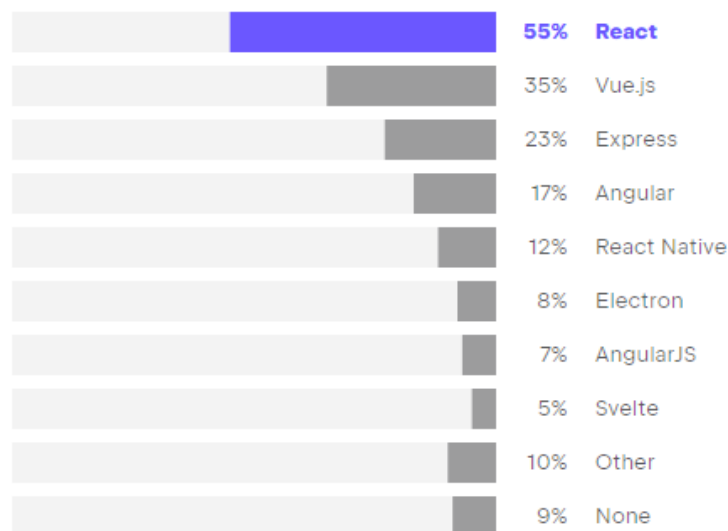
Chương 3 Triển khai hệ thống

3.1. Lựa chọn các cấu phần kỹ thuật

3.1.1. Frontend framework

Cùng với sự phát triển của công nghệ, các ứng dụng chạy trên máy tính, cần yêu cầu cài đặt, tính di động thấp đang dần được thay thế bởi các ứng dụng chạy trên nền tảng web thông qua trình duyệt. Trước đây, các ứng dụng web sử dụng bộ công cụ HTML, CSS và Javascript để đem lại trải nghiệm người dùng tốt hơn ứng dụng web cũ chỉ gồm HTML và CSS. Tuy vậy trong quá trình phát triển ứng dụng, việc sử dụng ngôn ngữ Javascript để lập trình các thành phần web gây không ít khó khăn cho người lập trình. Do đó hầu hết các ứng dụng web gần đây đều sử dụng một khung hỗ trợ do (thường được gọi là framework). Các framework này hỗ trợ người lập trình phát triển ứng dụng nhanh hơn, tối ưu hơn và thường các framework được phát hành dưới dạng mã nguồn mở.

Dựa trên khảo sát năm 2022 của JetBrains, 29,269 người phát triển ứng dụng được hỏi về framework họ sử dụng cho kết quả như biểu đồ sau:



Hình 10. Khảo sát các thư viện, framework phổ biến để phát triển giao diện web.

React là thư viện phổ biến nhất trong phát triển frontend với các lợi thế như:

- **Virtual DOM:** Việc áp dụng Document Object Model ảo cho phép React cập nhật các thành phần hiển thị rất nhanh, chỉ những thay đổi nào cần cập nhật React sẽ cập nhật để tránh lãng phí việc xử lý. Điều này nâng cao trải nghiệm người dùng và tốc độ xử lý của hệ thống.

ĐỒ ÁN TỐT NGHIỆP

- **React Component:** Các nhà phát triển có thể sử dụng lại các thành phần mà họ đã viết, tiết kiệm thời gian phát triển.

Tuy vậy React lại thiếu tài liệu hướng dẫn dễ đọc cho những người mới. React cũng thiếu các cấu trúc cơ bản của việc lập hiển thị Array (người dùng sẽ phải map các thành phần dưới dạng React để hiển thị).

Các ưu điểm và khuyết điểm của React được framework Vue kế thừa và chỉnh sửa.

- Vue rất rõ ràng và dễ sử dụng, các tài liệu của Vue viết chi tiết và thân thiện với người phát triển hơn.
- Dễ dàng sử dụng lại các thành phần, dễ dàng bổ sung các Component cho các Component khác.

3.1.2. Containerization Engine

Docker là một dự án mã nguồn mở để tự động hóa việc triển khai các ứng dụng dưới dạng các containers, các containers này dễ dàng chạy trên các nền tảng đám mây hoặc chạy trên hạ tầng có sẵn của tổ chức. Docker cũng là một công ty thúc đẩy và phát triển công nghệ này, họ hợp tác với các nhà cung cấp dịch vụ điện toán đám mây, Linux và Windows.

Tuy Docker Containers có thể chạy mọi nơi, mọi chỗ nhưng chúng cũng có một số hạn chế nhất định. Các Windows Image chỉ có thể chạy được trên máy Windows (Windows/Windows Server), các Linux Image chỉ chạy được trên máy Linux và có thể chạy trên máy Windows khi sử dụng ảo hóa Hyper-V hoặc sử dụng Windows Subsystem for Linux.

Docker Containers khi so sánh với máy ảo có một số ưu điểm như:

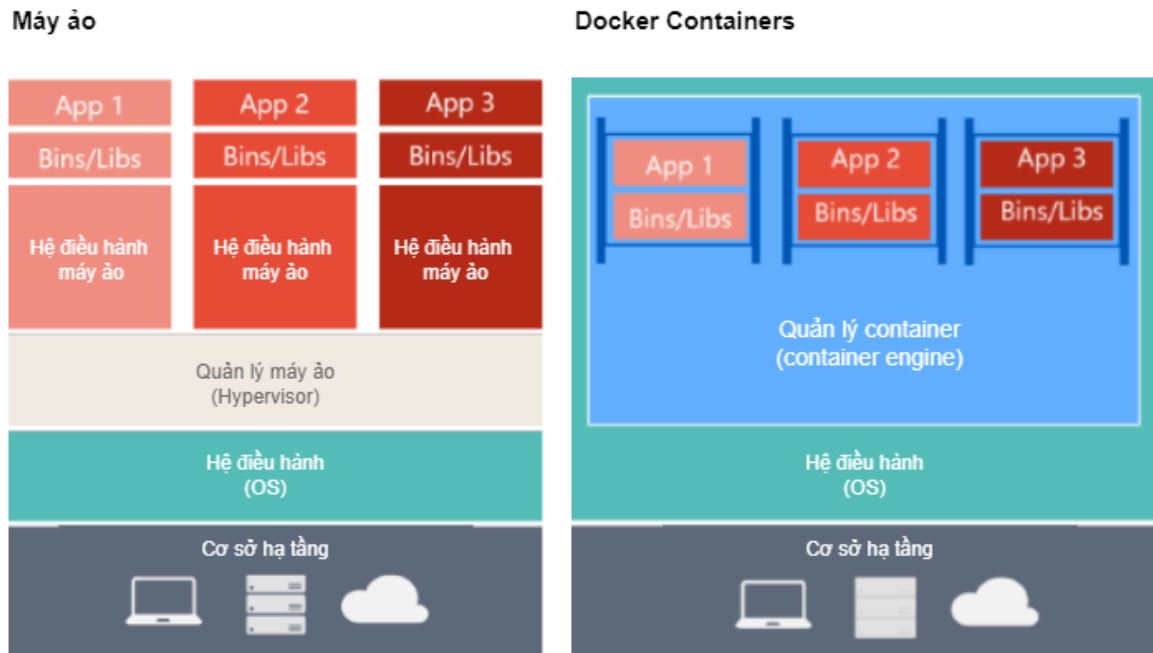
- Yêu cầu ít tài nguyên, các containers sử dụng chung nhân hệ điều hành
- Thời gian xây dựng image, thời gian khởi động container nhanh hơn, chạy được nhiều việc hơn so với việc sử dụng máy ảo. Ví dụ ta chỉ có thể chạy dc 3 instance (trên 3 máy ảo khác nhau) nhưng lại có thể chạy dc 6 docker containers chứa 3 instance trên cùng một máy, do đó giảm chi phí vận hành.

Các từ khóa được sử dụng trong hệ sinh thái Docker:

- **Container image:** Một gói có các thư viện, framework, các package, thông tin cần thiết để tạo được container. Thông thường, một image thường được phân lớp. Các lớp xếp chồng lên nhau, có thể một image được xây dựng dựa trên image khác. Một image không bao giờ thay đổi sau khi được xây dựng.
- **Dockerfile:** File văn bản bao gồm thông tin, hướng dẫn tạo Docker image
- **Build:** Quá trình tạo ra image từ các thông tin, hướng dẫn xây dựng từ Dockerfile

ĐỒ ÁN TỐT NGHIỆP

- **Container:** Một thực thể được tạo ra từ Docker image, tương tự như tiến trình là thực thể của chương trình. Một container bao gồm nội dung của Docker image, môi trường thực thi. Khi mở rộng một service lên, người ta thường thêm nhiều thực thể container của cùng một docker image.



Hình 11. Kiến trúc máy ảo và docker

- **Volume:** là filesystem (tập tin hệ thống) mà docker container có thể sử dụng để đọc hoặc viết. Vì images chỉ có thể đọc nhưng đa số các chương trình đều cần ghi dữ liệu, volume thêm khả năng viết cho container. Volume lưu trong hệ điều hành và được chương trình Docker quản lý.
- **Tag:** Là nhãn hoặc đánh dấu cho các image để phân biệt phiên bản của cùng một loại image.
- **Repository:** là tập hợp các Docker images, được phiên bản hóa sử dụng tag. Thường được sử dụng cho nhiều phiên bản khác nhau của image. Ví dụ như image nginx có các phiên bản gốc, debian bullseye, alpine (kích thước nhẹ),...
- **Registry:** là kho lưu trữ các repository. Registry mặc định là Docker Hub. Các company thường có Private Registry lưu các images nội bộ.
- **Compose:** là công cụ dưới dạng CLI và file dưới dạng YML/YAML sử dụng để chạy ứng dụng gồm nhiều container.
- **Cluster:** là một bộ các máy chạy docker nhưng thể hiện ra ngoài như là một máy chạy docker duy nhất. Một ứng dụng chạy trên cluster có thể được mở rộng thành nhiều instance do đó có thể tăng khả năng chịu tải cũng như có tính sẵn sàng cao. Docker cluster có thể được tạo bằng Kubernetes, Docker Swarm, Azure Service Fabric,...

ĐỒ ÁN TỐT NGHIỆP

- **Orchestrator:** Công cụ giúp quản lý các cluster và máy chạy docker thông qua CLI hoặc GUI. Qua đó hỗ trợ quản lý mạng, cấu hình, cân bằng tải, service discovery, tính sẵn sàng cao, cấu hình máy chạy Docker, v.v. Orchestrator chạy, phân phối, thay đổi quy mô và hồi phục các service trên các node.

3.1.3. API Gateway

Có nhiều giải pháp API Gateway được xây dựng, cả dưới mã nguồn mở và dưới dạng Platform as a Service (PaaS).

Giải pháp PaaS có thể kể đến như Amazon API Gateway, Google API Gateway, Azure API Management. Các giải pháp này chạy trên các hệ thống đám mây nên yêu cầu các service cũng cần triển khai trên cloud của các nhà cung cấp này và thường tính phí theo lời gọi API do đó chưa phù hợp với phạm vi của đồ án.

Bên cạnh giải pháp PaaS, các giải pháp API Gateway dưới dạng mã nguồn mở cũng khá phổ biến như Ocelot, Envoy, HAProxy. Trong đồ án này, tôi sử dụng Ocelot vì Ocelot nhanh, nhẹ, cung cấp các tính năng cơ bản cho API Gateway và cung cấp thêm một vài tính năng khác như xác thực. Ocelot có thể được chạy dưới dạng container, do đó có thể chạy cùng môi trường phát triển với các service (các API Gateway PaaS chạy trên môi trường của nhà cung cấp chúng).

Để sử dụng Ocelot cho đồ án, tôi thiết lập cài đặt như sau:

```
{
  "GlobalConfiguration": {
    "BaseUrl": "http://localhost:80"
  },
  "Routes": [
    {
      "UpstreamPathTemplate": "/AuctionService/{everything}",
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "UpstreamHttpMethod": ["Get", "Post", "Options", "Put", "Delete"],
      "DownstreamHostAndPorts": [
        {
          "Host": "auctions",
          "Port": 80
        }
      ]
    },
    {
      "UpstreamPathTemplate": "/UserService/{everything}",
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "UpstreamHttpMethod": ["Get", "Post", "Options", "Put", "Delete"],
      "DownstreamHostAndPorts": [
        {
          "Host": "users",
          "Port": 80
        }
      ]
    }
  ]
}
```

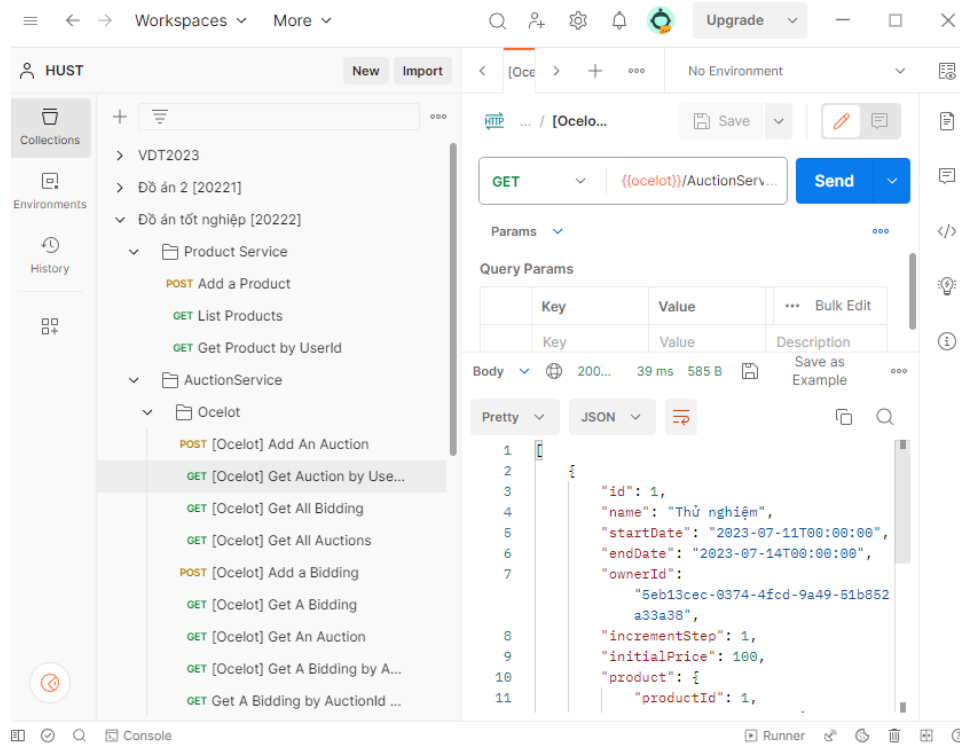
```
    },  
    {  
      "UpstreamPathTemplate": "/ProductService/{everything}",  
      "DownstreamPathTemplate": "/api/{everything}",  
      "DownstreamScheme": "http",  
      "UpstreamHttpMethod": ["Get", "Post", "Options", "Put", "Delete"],  
      "DownstreamHostAndPorts": [  
        {  
          "Host": "products",  
          "Port": 80  
        }  
      ]  
    }  
  ]  
}
```

3.1.4. Services

Tất cả các service tôi dự định xây dựng không có các tác vụ phức tạp cần sử dụng một ngôn ngữ cụ thể nào đó, ví dụ như với tác vụ Machine Learning và Deep Learning thì xây dựng các microservice bằng Python được ưu tiên hơn cả. Do các tác vụ là web thông thường và đã có kinh nghiệm lập trình trên ngôn ngữ C# trước đó, tôi lựa chọn sử dụng framework .NET 6 của Microsoft. Với bản chất mô-đun hóa và kích thước bé của .NET 6 giúp kết hợp hoàn hảo khi sử dụng trong các containers. So sánh với .NET Framework, khi triển khai và chạy, image được build bởi .NET 6 nhỏ hơn nhiều so với .NET Framework. Về tốc độ xử lý, theo Microsoft, tốc độ xử lý của .NET 6 mới nhanh hơn tới 10 lần so với .NET Framework cũ. Hơn thế nữa .NET 6 cũng chạy được đa các nền tảng, trong khi .NET Framework chỉ chạy được trên hệ điều hành Windows/Windows Server.

Các service tôi xây dựng sử dụng REST API để trao đổi thông tin với bên ngoài.

ĐỒ ÁN TỐT NGHIỆP



Hình 12. Minh họa API của auction service trên Postman

3.1.5. Database

Do đã lựa chọn phát triển các service sử dụng công nghệ .NET của Microsoft. Để có được sự tương thích tốt nhất, tôi đã lựa chọn sử dụng hệ quản trị cơ sở dữ liệu Microsoft SQL Server. Ngoài ra, Microsoft là một công ty công nghệ lớn, cung cấp hỗ trợ tốt cho sản phẩm của họ. SQL Server được cung cấp với tài liệu phong phú, cộng đồng lớn, các diễn đàn thảo luận và tài liệu hướng dẫn từ Microsoft.

Để tiết kiệm thời gian cài đặt cũng như tăng tính tương thích trong các môi trường khác nhau, tôi sử dụng Database trong container.

```
mssql:
  container_name: "mssql"
  image: 'mcr.microsoft.com/mssql/server'
  ports:
    - '1433:1433'
  environment:
    - ACCEPT_EULA=Y
    - SA_PASSWORD=Secret1234
  volumes:
    - 'sqlserver:/var/opt/mssql'
  networks:
    - internal_network
```

3.1.6. Message Queue

Để đáp ứng được tốc độ xử lý tốt cũng như việc bên gửi không cần biết bên nhận, bên nhận có thể có nhiều service xử lý một tác vụ, tôi sử dụng phương thức truyền thông không đồng bộ. Có rất nhiều công cụ hỗ trợ làm broker cho việc truyền thông bất đồng bộ giữa các service như Azure Service Bus, RabbitMQ, Kafka. Tuy vậy, để đơn giản việc cài đặt cũng như do kích thước của đồ án tốt nghiệp chưa yêu cầu việc xử lý và queue nhiều thông điệp, tôi sử dụng RabbitMQ cho giải pháp truyền thông giữa các service.

Để cài đặt RabbitMQ, tôi sử dụng Docker cung cấp sẵn bởi RabbitMQ theo hướng dẫn trên trang chủ (<https://www.rabbitmq.com/download.html>).



```
1 docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management
```

Ngoài ra, để thuận tiện hơn cho việc phát triển, tôi chạy RabbitMQ trong docker-compose.

```
rabbitmq:
  container_name: "rabbitmq"

  image: "rabbitmq:3.12-management"
  ports:
    - 5672:5672
    - 15672:15672
  environment:
    RABBITMQ_DEFAULT_USER: "admin"
    RABBITMQ_DEFAULT_PASS: "secret"
  networks:
    - internal_network
```

3.1.7. Công cụ CI/CD

Trong đồ án tốt nghiệp này, tôi đã áp dụng kỹ thuật CI và CD để tăng hiệu quả phát triển và triển khai ứng dụng. Hiện nay có nhiều công cụ hỗ trợ như Jenkin, TeamCity, Gitlab, Github Actions, CircleCI, TravisCI,...

ĐỒ ÁN TỐT NGHIỆP

Jenkin là một công cụ thường được nhiều DevOps sử dụng để thực hiện các thao tác liên tục tích hợp mã nguồn. Jenkins viết dựa trên Java, hỗ trợ các cách cấu hình dựa trên giao diện người dùng (GUI) hoặc giao diện dòng lệnh (CLI). Jenkins thường được áp dụng cho các dự án có những đặc điểm như:

- Code lưu ở một nơi lưu trữ riêng biệt (Code Repository)
- Nhà phát triển và vận hành muốn kiểm soát toàn bộ quá trình CI/CD
- Cần máy chủ đặt tại cơ quan (do vấn đề bảo mật,...)

Bên cạnh Jenkins, một công cụ CI/CD mới nổi và cũng khá dễ sử dụng chính là Github Actions. Github Actions là công cụ đáp ứng hầu hết các nhu cầu CI/CD của nhà phát triển. Github Actions là công cụ có sẵn trên Github, hầu hết người có repository trên Github đều có thể sử dụng với ít bước cài đặt. Một số đặc điểm của dự án áp dụng được Github như:

- Codebase lưu trên Github
- Không muốn tập trung sâu vào CI/CD và các tùy chỉnh phức tạp.
- Dễ dàng tương tác với Docker để xây dựng Docker images và đưa lên Repository như DockerHub.

Nhờ các đặc điểm của Github Actions, tôi đã lựa chọn sử dụng công cụ CI/CD này phục vụ việc phát triển và xây dựng hệ thống. Một số thành phần của Github Actions tôi sử dụng trong đồ án như:

- **Workflows:** Là thủ tục tự động, được tạo bởi các job, chạy định kỳ hoặc dựa vào các trigger của một event nào đó.
- **Events:** Là hành động được tạo sẵn và có thể được sử dụng để chạy **workflows** tự động. Use case cho **Events** là khi người lập trình đẩy code lên một nhánh nào đó, một workflow sẽ được chạy để test code.
- **Runners:** Là các server (hay các môi trường thực thi) để chạy các **workflows**. Các runners này được Github cung cấp và người chạy CI/CD cũng có thể dùng các server riêng để chạy. Đối với runners cung cấp bởi github, hiện họ hỗ trợ các hệ điều hành như Ubuntu, Debian, Windows,...
- **Jobs:** Là tập hợp các bước thực hiện trên 1 runner. Nếu một workflow bao gồm nhiều job thì các job này sẽ được chạy song song.
- **Actions:** Là một cụm các jobs được dựng sẵn, có thể hiểu chúng như thư viện cung cấp các jobs sẵn.

3.1.8. Container Orchestration

Các công cụ container orchestration là những phần mềm quan trọng giúp quản lý và điều phối các container trong môi trường phức tạp của hệ thống. Hai công cụ container orchestration phổ biến nhất hiện nay là Kubernetes, Docker Swarm.

ĐỒ ÁN TỐT NGHIỆP

Kubernetes là một công cụ container orchestration mạnh mẽ và phổ biến nhất trong cộng đồng Cloud Native và DevOps. Nó hỗ trợ quy mô lớn và có khả năng xử lý môi trường phức tạp. Tuy nhiên, việc cấu hình ban đầu và khởi đầu có thể phức tạp đối với người mới bắt đầu. Kubernetes cung cấp nhiều tính năng bảo mật và khả năng khắc phục lỗi, đồng thời hỗ trợ tự động phục hồi, giúp đảm bảo ứng dụng hoạt động ổn định và tin cậy.

Docker Swarm là một công cụ container orchestration dễ sử dụng và dễ triển khai, đặc biệt là với người dùng đã quen thuộc với Docker. Nó được tích hợp sẵn trong Docker, không cần phải cài đặt phần mềm bổ sung. Docker Swarm hỗ trợ hiệu suất tốt và tối ưu hóa cho các ứng dụng nhẹ, nhưng quy mô lớn của nó không bằng Kubernetes. Nó thích hợp cho việc triển khai các ứng dụng đơn giản, nhưng có thể phức tạp hơn khi đối mặt với các tình huống phức tạp hơn.

Do tính phổ biến, tôi sử dụng Kubernetes trong việc triển khai các container. Kubernetes bao gồm nhiều thành phần chính làm việc cùng nhau để quản lý việc triển khai và vận hành các ứng dụng container. Một số thành phần quan trọng của Kubernetes thường hay được sử dụng như:

1. Control Plane (Nền tảng quản trị):

- **Controller Manager (CM):** Quản lý các bộ điều khiển (controllers) trong hệ thống, đảm bảo trạng thái mong muốn của các tài nguyên (ví dụ: replica sets, deployments) được duy trì.
- **Etcd:** Là cơ sở dữ liệu phân tán được sử dụng để lưu trữ trạng thái của cụm Kubernetes. Etcd giữ thông tin cấu hình và trạng thái của các tài nguyên, giúp đảm bảo trạng thái đồng nhất của hệ thống.
- **Cloud Control Manager:** Giao tiếp và điều khiển các nguồn tài nguyên của các nhà cung cấp cloud để cung cấp cho cluster.
- **API Server (api):** Là thành phần chính cung cấp API để giao tiếp với Kubernetes. Tất cả tác vụ của người dùng và các công cụ khác đều thông qua API Server.
- **Scheduler (sched):** Định vị các pod (nhóm container) lên các nút làm việc dựa trên các yêu cầu và ràng buộc đã định sẵn.

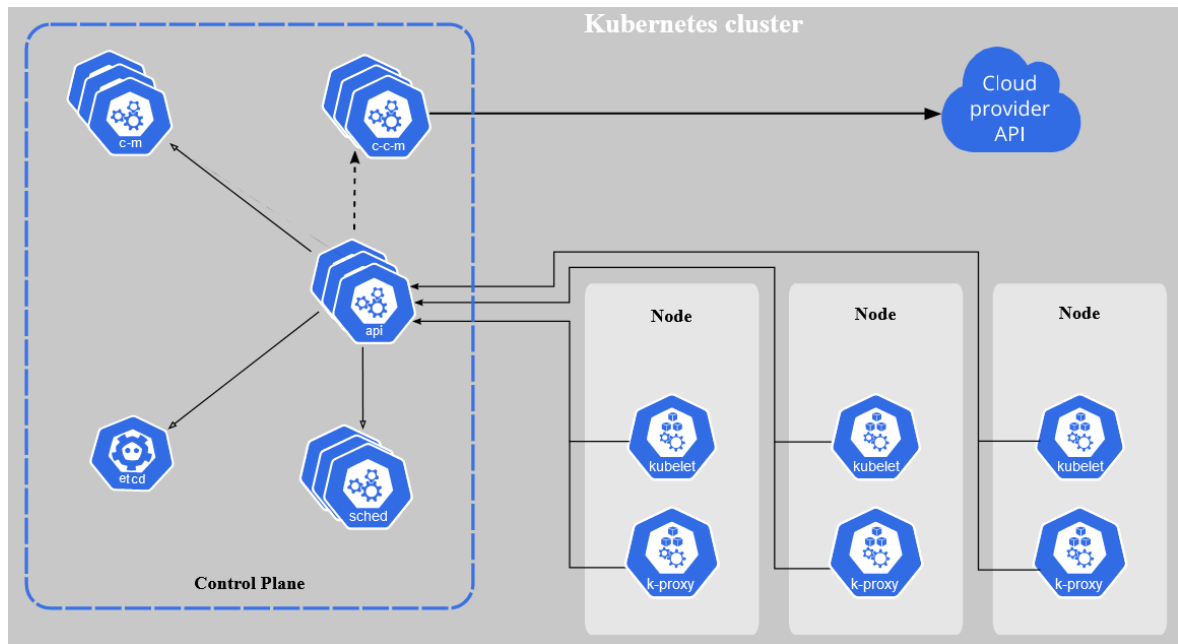
2. Worker Nodes : Đây là các nút trong hệ thống chịu trách nhiệm chạy các container và ứng dụng thực tế. Mỗi nút làm việc chứa các thành phần sau:

- **Kubelet:** Là thành phần chịu trách nhiệm liên lạc với API Server và đảm bảo pod (nhóm container) được chạy đúng cách trên nút.
- **Container Runtime:** Là phần mềm quản lý và chạy các container, chẳng hạn như Docker hoặc containerd.
- **Kube-proxy:** Đảm bảo mạng trong cụm Kubernetes, quản lý định tuyến và cân bằng tải giữa các pod.

3. Pods: Là đơn vị cơ bản nhất trong Kubernetes, chứa một hoặc nhiều container chạy cùng nhau trên cùng một nút. Các container trong cùng một pod chia sẻ cùng một không gian mạng và có thể giao tiếp trực tiếp với nhau thông qua localhost.

ĐỒ ÁN TỐT NGHIỆP

4. **Replica Sets và Deployments:** Định nghĩa số lượng và trạng thái mong muốn của các pod. Replica Sets duy trì số lượng pod đúng theo yêu cầu, trong khi Deployments cho phép thực hiện rolling updates và rollbacks của ứng dụng.
5. **Services:** Định nghĩa một cách trừu tượng các pod và cung cấp một địa chỉ IP cố định và cổng liên lạc để cho phép các ứng dụng trong cụm tương tác với nhau.
6. **Volumes:** Cung cấp một cơ chế để lưu trữ dữ liệu và chia sẻ dữ liệu giữa các container trong cùng một pod.
7. **Namespace:** Giúp chia cụm thành các phạm vi cô lập logic, giúp quản lý và phân quyền tài nguyên dễ dàng.
8. **ConfigMap:** Cung cấp giải pháp cấu hình ứng dụng cho các container. ConfigMap có thể chứa các cặp khóa-giá trị (key-value pairs) hoặc cũng có thể chứa dữ liệu không có cấu trúc (data literals). Dữ liệu trong ConfigMap có thể được sử dụng bởi các Pods, Deployments, StatefulSets, DaemonSets và các tài nguyên khác trong Kubernetes.



Hình 13. Kiến trúc Kubernetes

Cài đặt Kubernetes

Để phát triển và thử nghiệm các ứng dụng trên Kubernetes, có các công cụ phát triển nổi bật là Kind, Minikube.

Kind là một công cụ container orchestration nhẹ, tiện, sử dụng Docker để tạo và quản lý các cụm Kubernetes trên máy tính cá nhân. Với Kind, việc triển khai các cụm Kubernetes trở nên dễ dàng và nhanh chóng, giúp tạo ra môi trường phát triển cụ thể để

ĐỒ ÁN TỐT NGHIỆP

thử nghiệm ứng dụng. Điểm mạnh của Kind là tính đơn giản và hỗ trợ triển khai, xóa cluster nhanh, dễ dàng. Kind cung cấp môi trường phát triển cũng như môi trường thử nghiệm giống như môi trường triển khai thực tế, giúp đảm bảo tích hợp, triển khai không lỗi ẩn (lỗi đến lúc triển khai mới phát hiện ra). Với việc sử dụng Docker, Kind giúp tiết kiệm thời gian và tài nguyên, không cần phải cài đặt thêm môi trường máy ảo. Ngoài ra, Kind cũng hỗ trợ cluster đa node và kèm tính sẵn sàng cao (HA).

Để cài đặt Kind, tôi sử dụng môi trường Ubuntu. Các bước cài đặt:

1. Download Kind binary:

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
```

2. Cấp quyền chạy:

```
chmod +x ./kind
```

3. Chuyển vào thư mục bin:

```
sudo mv ./kind /usr/local/bin/kind
```

3.1. Triển khai CI/CD

Về việc triển khai liên tục, do hạn chế về nhiều mặt nên tôi chưa thể thiết lập một máy chủ có địa chỉ IP tĩnh để thực hiện việc triển khai. Tuy vậy, tôi cũng đã thiết kế một quy trình CD cơ bản để phục vụ việc tạo Docker Images khi phiên bản mới của mã nguồn được phát hành. Sau khi tạo xong, các Docker Images này sẽ được đưa lên DockerHub và sau đó được triển khai thử công trên máy chủ đích sử dụng Kubernetes.

```
name: cd-build-push-on-tag
on:
  push:
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
```


ĐỒ ÁN TỐT NGHIỆP

```
- name: Login to Docker Hub
  uses: docker/login-action@v2
  with:
    username: ${ secrets.DOCKERHUB_USERNAME }
    password: ${ secrets.DOCKERHUB_TOKEN }
- name: Build and push auctions-service
  uses: docker/build-push-action@v4
  with:
    push: true
    tags: ducpa01/datn-auctions:latest
    context: "${defaultContext}:src/services/auctions"
- name: Build and push products-service
  uses: docker/build-push-action@v4
  with:
    push: true
    tags: ducpa01/datn-products:latest
    context: "${defaultContext}:src/services/products"

- name: Build and push users-service
  uses: docker/build-push-action@v4
  with:
    push: true
    # tags: ducpa01/datn-users:${github.ref_name}
    tags: ducpa01/datn-users:latest
    context: "${defaultContext}:src/services/users"

- name: Build and push API Gateway
  uses: docker/build-push-action@v4
  with:
    push: true
    tags: ducpa01/datn-apigw:latest
    context: "${defaultContext}:src/ApiGateway"
```



Hình 14. Hình ảnh triển khai liên tục trên Github Actions

3.2. Triển khai sử dụng Docker & Docker Compose

ĐỒ ÁN TỐT NGHIỆP

Trong đồ án tốt nghiệp, tôi sử dụng Docker để xây dựng và chạy các service. Để tạo Docker image, tôi xây dựng Dockerfile cho các dịch vụ. Một *Dockerfile* cơ bản như dưới đây:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 as build-env
WORKDIR /src
COPY *.csproj .
RUN dotnet restore
COPY . .
RUN dotnet publish -c Release -o /publish

FROM mcr.microsoft.com/dotnet/aspnet:6.0 as runtime
WORKDIR /publish
COPY --from=build-env /publish .
ENTRYPOINT ["dotnet", "auctions.dll"]
```

Dockerfile này đã được ứng dụng kỹ thuật multi-stages build giúp giảm thiểu tối đa kích thước image, tối ưu tốc độ build hơn do đã có các caching layers.

Trong quá trình COPY file vào Docker Container, có những folder, file không cần thiết và không nên để trong container tránh làm tăng dung lượng. Đây là file *.dockerignore* tôi đã tạo:

```
bin/
obj/
out/
TestResults/
```

Sau khi build xong, tôi có thể kiểm tra các images đã build bằng câu lệnh “docker images”:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ducpa01/datn-users	latest	00d0cf2b2771	4 days ago	300MB
ducpa01/datn-auctions	latest	6a2c882f3f72	4 days ago	300MB
ducpa01/datn-products	latest	d7b8c0e58cfd	4 days ago	299MB
ducpa01/products	latest	182038bb632e	5 days ago	299MB
ducpa01/datn-apigw	latest	5aee8f20256c	2 weeks ago	218MB
apigw	latest	5aee8f20256c	2 weeks ago	218MB

Hình 15. Các service khi được docker hóa thành các image

Sau khi đã có các images của các services, tôi tạo docker compose để thử nghiệm giao tiếp giữa các service.

ĐỒ ÁN TỐT NGHIỆP

```
version: '3.7'

services:
  apigw:
    container_name: apigw
    image: ducpa01/datn-apigw:latest
    ports:
      - 5000:80
    volumes:
      - "./configuration.json:/publish/configuration/configuration.json:ro"
    networks:
      - internal_network
  auctions:
    container_name: datn-auctions
    image: ducpa01/datn-auctions:latest
    ports:
      - 5078:80
    networks:
      - internal_network
    depends_on:
      - rabbitmq
      - mssql
  users:
    container_name: datn-users
    image: ducpa01/datn-users:latest
    ports:
      - 5180:80
    networks:
      - internal_network
    depends_on:
      - rabbitmq
      - mssql
  products:
    container_name: datn-products
    image: ducpa01/datn-products:latest
    ports:
      - 5105:80
    networks:
      - internal_network
    depends_on:
      - rabbitmq
      - mssql

  rabbitmq:
    container_name: "rabbitmq"

    image: "rabbitmq:3.12-management"
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      RABBITMQ_DEFAULT_USER: "admin"
      RABBITMQ_DEFAULT_PASS: "secret"
    networks:
      - internal_network
  mssql:
    container_name: "mssql"
    image: 'mcr.microsoft.com/mssql/server'
    ports:
      - '1433:1433'
    environment:
```

ĐỒ ÁN TỐT NGHIỆP

```
- ACCEPT_EULA=Y
- SA_PASSWORD=Secret1234
volumes:
- 'sqlserver:/var/opt/mssql'
networks:
- internal_network
volumes:
  sqlserver:
networks:
  internal_network:
```

3.3. Triển khai sử dụng Kubernetes

Các bước triển khai lên Kubernetes

1. Dựng một cụm Kubernetes (K8s Cluster) sử dụng Kind
2. Đóng gói ứng dụng sử dụng Docker Container. Điều này giúp đảm bảo ứng dụng có thể chạy độc lập với môi trường và dễ dàng chạy trên các cụm Kubernetes.
3. Xây dựng cấu hình Kubernetes (K8s Manifest). Đây là các file dưới dạng YAML hoặc JSON dùng để mô tả các tài nguyên trong cụm Kubernetes, ví dụ như Pod, Deployment, Service, ConfigMap, v.v...
4. Triển khai ứng dụng sử dụng lệnh **kubectl apply**.

Tạo Kubernetes Cluster

Do Kind cần forward một số port để có thể truy cập từ máy host bên ngoài, tôi phải xây dựng file config để tạo cùng với cluster.

```
#kind-cluster.config.yml
apiVersion: kind.x-k8s.io/v1alpha4
kind: Cluster
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 31080
    hostPort: 80
    listenAddress: "0.0.0.0"
    protocol: tcp
- role: worker
```

Sau khi đã có file config, để tạo một cluster thỏa mãn yêu cầu đã cho, tôi chạy lệnh:

```
kind create cluster --name datn --config kind-cluster.config.yml
```

ĐỒ ÁN TỐT NGHIỆP

Lệnh này sẽ tạo một cluster có tên là *datn* và forward port 31080 ra bên ngoài dưới port 80. Các ứng dụng ngoài cluster có thể truy cập port 80 để sử dụng dịch vụ.

ConfigMap

Để thuận tiện cho việc cấu hình API Gateway, tôi xây dựng ConfigMap như sau:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: apigw-configmap
data:
  configuration.json: |
    {
      "GlobalConfiguration": {
        "BaseUrl": "http://localhost:80"
      },
      "Routes": [
        {
          "UpstreamPathTemplate": "/AuctionService/{everything}",
          "DownstreamPathTemplate": "/api/{everything}",
          "DownstreamScheme": "http",
          "UpstreamHttpMethod": ["Get", "Post", "Options", "Put", "Delete"],
          "DownstreamHostAndPorts": [
            {
              "Host": "auctions-service",
              "Port": 80
            }
          ]
        },
        {
          "UpstreamPathTemplate": "/UserService/{everything}",
          "DownstreamPathTemplate": "/api/{everything}",
          "DownstreamScheme": "http",
          "UpstreamHttpMethod": ["Get", "Post", "Options", "Put", "Delete"],
          "DownstreamHostAndPorts": [
            {
              "Host": "users-service",
              "Port": 80
            }
          ]
        },
        {
          "UpstreamPathTemplate": "/ProductService/{everything}",
          "DownstreamPathTemplate": "/api/{everything}",
          "DownstreamScheme": "http",
          "UpstreamHttpMethod": ["Get", "Post", "Options", "Put", "Delete"],
          "DownstreamHostAndPorts": [
            {
              "Host": "products-service",
              "Port": 80
            }
          ]
        }
      ]
    }
  }
```

Xây dựng ConfigMap cho các service khác

ĐỒ ÁN TỐT NGHIỆP

Các service khác cũng cần cấu hình một vài dịch vụ như cơ sở dữ liệu và RabbitMQ, tôi xây dựng config map như sau:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: services-configmap
data:
  appsettings.json: |
    {
      "Logging": {
        "LogLevel": {
          "Default": "Information",
          "Microsoft.AspNetCore": "Warning"
        }
      },
      "AllowedHosts": "*",
      "ConnectionStrings": {
        "DataContext": "Data Source=.db",
        "Sqlserver": "Server=mssql-service;Database=datn;User
Id=sa;Password=Secret1234;TrustServerCertificate=true"
      },
      "Spring": {
        "RabbitMq": {
          "Host": "rabbitmq",
          "Port": 5672,
          "Username": "admin",
          "Password": "secret"
        }
      }
    }
  }
```

Services

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auctions-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: auctions
    spec:
      containers:
        - name: auctions
          image: ducpa01/datn-auctions:latest
          ports:
            - containerPort: 80
          env:
            - name: ASPNETCORE_URLS
              value: http://*:80
          volumeMounts:
            - name: services-config-volume
              mountPath: /publish/appsettings.json
              readOnly: true
              subPath: appsettings.json
```

ĐỒ ÁN TỐT NGHIỆP

```
resources:
  limits:
    memory: "128Mi"
    cpu: "500m"

  volumes:
    - name: services-config-volume
      configMap:
        name: services-configmap
  selector:
    matchLabels:
      app: auctions
---
apiVersion: v1
kind: Service
metadata:
  name: auctions-service
spec:
  type: ClusterIP
  ports:
    - port: 80
  selector:
    app: auctions
```

API Gateway

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apigw-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: apigw
  template:
    metadata:
      labels:
        app: apigw
    spec:
      containers:
        - name: apigw-deployment
          image: ducpa01/datn-apigw:latest
          ports:
            - containerPort: 80
          volumeMounts:
            - name: apigw-config-volume
              mountPath: /publish/configuration
              readOnly: true
          resources:
            limits:
              memory: "128Mi"
              cpu: "500m"

      volumes:
        - name: apigw-config-volume
          configMap:
            name: apigw-configmap
```

ĐỒ ÁN TỐT NGHIỆP

```
---
apiVersion: v1
kind: Service
metadata:
  name: apigw-service
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 31080
      protocol: TCP
      name: http
  selector:
    app: apigw
```

Microsoft SQL Server

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mssql
spec:
  serviceName: "mssql"
  replicas: 1
  selector:
    matchLabels:
      app: mssql
  template:
    metadata:
      labels:
        app: mssql
    spec:
      securityContext:
        fsGroup: 10001
      containers:
        - name: mssql
          image: 'mcr.microsoft.com/mssql/server'
          ports:
            - containerPort: 1433
              name: tcpsql
          env:
            - name: ACCEPT_EULA
              value: "Y"
            - name: MSSQL_SA_PASSWORD
              value: "Secret1234"
          volumeMounts:
            - name: mssql
              mountPath: "/var/opt/mssql"
      volumeClaimTemplates:
        - metadata:
            name: mssql
          spec:
            accessModes:
              - ReadWriteOnce
            resources:
              requests:
                storage: 500Mi
```


ĐỒ ÁN TỐT NGHIỆP

```
apiVersion: v1
kind: Service
metadata:
  name: mssql-service
spec:
  type: ClusterIP
  ports:
    - port: 1433
  selector:
    app: mssql
```

RabbitMQ

Để dựng lên RabbitMQ cluster đảm bảo tính nhất quán dữ liệu khi có nhiều instance, tôi sử dụng RabbitMQ Cluster Operator đã được cung cấp sẵn bởi RabbitMQ. Lệnh dưới đây được sử dụng để cài RabbitMQ Cluster Operator:

```
kubectl apply -f https://github.com/rabbitmq/cluster-operator/releases/latest/download/cluster-operator.yml
```

Sau khi đã có operator, tôi tiến hành dựng lên một cluster với cấu hình, tài nguyên như sau:

```
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
  name: rabbitmq
spec:
  resources:
    requests:
      cpu: 250m
      memory: 200Mi
    limits:
      cpu: 500m
  replicas: 1
  rabbitmq:
    additionalConfig: |
      default_user=admin
      default_pass=secret
```

Để kiểm tra thông tin, số lượng các pod đang chạy, tôi sử dụng **kubectl get all** để nhận được các thông tin. Ngoài ra Kubernetes còn hỗ trợ giao diện web để giám sát quá trình triển khai.

ĐỒ ÁN TỐT NGHIỆP

```

vm1@vm1:~/hust/DATN-BiddingSystem/deploy/k8s$ kubectl get all

```

NAME	READY	STATUS	RESTARTS	AGE
pod/apigw-deployment-bc8bff8cc-2qfvh	1/1	Running	0	10m
pod/apigw-deployment-bc8bff8cc-k5rwd	1/1	Running	0	10m
pod/auctions-deployment-6bbbf7958b-6vhw	1/1	Running	0	14h
pod/auctions-deployment-6bbbf7958b-9kprf	0/1	Pending	0	118s
pod/auctions-deployment-6bbbf7958b-xjdlj	0/1	Pending	0	118s
pod/mssql-0	1/1	Running	0	15h
pod/products-deployment-7857d7956-zmfc	1/1	Running	0	14h
pod/rabbitmq-server-0	1/1	Running	0	13m
pod/users-deployment-7cf6db4646-7fq48	1/1	Running	0	2m22s
pod/users-deployment-7cf6db4646-s7m8v	1/1	Running	0	12m

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/apigw-service	NodePort	10.96.176.83	<none>	80:31080/TCP	10m
service/auctions-service	ClusterIP	10.96.85.240	<none>	80/TCP	14h
service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	15h
service/mssql-service	ClusterIP	10.96.42.250	<none>	1433/TCP	14h
service/products-service	ClusterIP	10.96.192.109	<none>	80/TCP	14h
service/rabbitmq	ClusterIP	10.96.201.213	<none>	15672/TCP, 15692/TCP, 5672/TCP	13m
service/rabbitmq-nodes	ClusterIP	None	<none>	4369/TCP, 25672/TCP	13m
service/rabbitmq-service	ClusterIP	10.96.9.147	<none>	5672/TCP	39m
service/users-service	ClusterIP	10.96.66.194	<none>	80/TCP	12m

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/apigw-deployment	2/2	2	2	10m
deployment.apps/auctions-deployment	1/3	3	1	14h
deployment.apps/products-deployment	1/1	1	1	14h
deployment.apps/users-deployment	2/2	2	2	12m

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/apigw-deployment-bc8bff8cc	2	2	2	10m
replicaset.apps/auctions-deployment-6bbbf7958b	3	3	1	14h
replicaset.apps/products-deployment-7857d7956	1	1	1	14h
replicaset.apps/users-deployment-7cf6db4646	2	2	2	12m

NAME	READY	AGE
statefulset.apps/mssql	1/1	15h
statefulset.apps/rabbitmq-server	1/1	13m

NAME	ALLREPLICASREADY	RECONCILESUCCESS	AGE
rabbitmqcluster.rabbitmq.com/rabbitmq	True	True	13m

Như vậy, toàn bộ hệ thống đấu giá trực tuyến đã được triển khai trên Cluster sử dụng Kubernetes và Docker. Hệ thống này có thể dễ dàng mở rộng bằng cách thay đổi tham số các Replica tùy theo nhu cầu thực tế. Do hạn chế về thời gian và thiết bị, tôi chưa thể thử nghiệm cài đặt thêm Cluster trên những máy tính khác nhau mà mới chỉ dừng lại ở việc phát triển, triển khai thử nghiệm cluster trên máy tính cá nhân.

Kết luận

Đề tài đã trình bày về kiến trúc microservice, ứng dụng kiến trúc microservice trong xây dựng hệ thống đấu giá trực tuyến, tạo ra một ứng dụng thân thiện với môi trường đám mây. Bằng cách áp dụng kiến thức cơ bản về microservice, phân tích và thiết kế hệ thống, tôi đã tiến hành phân tích, thiết kế và xây dựng hệ thống đấu giá trực tuyến cơ bản. Sau đó, nhờ việc sử dụng Docker và Kubernetes, tôi đã thành công triển khai ứng dụng và tích hợp quá trình triển khai liên tục, giúp việc cập nhật và phát triển ứng dụng diễn ra một cách hiệu quả và nhanh chóng.

Để đảm bảo tính ổn định và tin cậy của hệ thống, tôi đã xây dựng mô hình logic của hệ thống logging và monitoring, giúp theo dõi hoạt động của ứng dụng và dễ dàng xác định và khắc phục sự cố trong quá trình triển khai. Qua quá trình nghiên cứu và thực hiện, tôi đã đạt được mục tiêu đề ra ban đầu và tạo ra một hệ thống đấu giá trực tuyến cơ bản và dễ dàng mở rộng theo chiều ngang.

Một số hạn chế của đồ án:

- Chưa dựng được cluster trên các máy tính khác nhau
- Chưa áp dụng được các kỹ thuật nâng cao khi sử dụng Kubernetes
- Chưa hỗ trợ Websocket để cập nhật thông tin lên UI liên tục
- Chưa thiết kế được giao diện đẹp mắt, thân thiện với người dùng

Hướng phát triển trong tương lai:

- Sử dụng Websocket để liên tục cập nhật giá vừa được đặt.
- Tăng tính bảo mật cho hệ thống sử dụng RefreshToken.
- Áp dụng kỹ thuật giao dịch phân tán sử dụng mẫu thiết kế Saga.
- Dựng và quản lý nhiều cluster trên các máy khác nhau.

Tài liệu tham khảo

- [1] Sam Newman (2021). Building Microservices: Designing Fine-Grained Systems, O'Reilly Media.
- [2] Chris Richardson (2018). Microservices Patterns: With examples in Java, Manning.
- [3] Kevin Hoffman (2017). Building Microservices with ASP.NET Core: Develop, Test, and Deploy Cross-Platform Services in the Cloud, O'Reilly Media.
- [4] Giới thiệu về Microservice, Google Cloud.
<https://cloud.google.com/architecture/microservices-architecture-introduction>
- [5] So sánh Microsoft .NET và Microsoft .NET Framework.
<https://learn.microsoft.com/en-us/dotnet/architecture/microservices/net-core-net-framework-containers/net-core-container-scenarios>
- [6] Luật đầu giá tài sản 2016.
- [7] Domain driven design, Microsoft. <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model>
- [8] Giới thiệu về Docker, Microsoft Learn. <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined>
- [9] Giới thiệu về Kubernetes. <https://kubernetes.io/docs/concepts/overview/>