**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH**

**KHOA ĐÀO TẠO CHẤT LƯỢNG CAO**

**NGÀNH CÔNG NGHỆ THÔNG TIN**

-----🙞🙜🕮🙞🙜-----



**TIỂU LUẬN CHUYÊN NGÀNH**

**TÌM HIỂU KIẾN TRÚC MICROSERVICE, TRIỂN KHAI ỨNG DỤNG TRÊN KIẾN TRÚC ĐÓ VỚI CÔNG NGHỆ JAVA**

GVHD: Nguyễn Thanh Phước

SVTH: Đoàn Huỳnh Thiện 15110129

Đặng Tiến Hưng 15110060

**TP. Hồ Chí Minh, tháng 12 năm 2018**

# NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN

………………………………………………………………………………………….………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

Chữ ký giảng viên hướng dẫn

# NHẬN XÉT CỦA GIẢNG VIÊN PHẢN BIỆN

………………………………………………………………………………………….………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

………………………………………………………………………………………….

Chữ ký giảng viên phản biện

# LỜI CẢM ƠN

Nhóm xin chân thành cảm ơn sự hướng dẫn và chỉ bảo tận tình của anh Nguyễn Thanh Phước - người đã luôn hướng dẫn, quan tâm, động viên, thông cảm, tạo điều kiện thuận lợi cho nhóm rất nhiều trong quá trình thực hiện đề tài. Các định hướng và sự hiểu biết về khoa học của anh chính là tiền đề để tôi hoàn thành được đề tài này. Đồng thời, nhóm cũng xin được gửi lời cảm ơn đến các thầy cô giáo trong khoa Công nghệ thông tin trường Đại học Sư Phạm Kỹ Thuật đã giảng dạy và truyền đạt những kiến thức, kinh nghiệm quý báu cho tôi trong suốt khóa học.

Nhóm cũng xin được cảm ơn các tác giả của các công trình nghiên cứu, tài liệu, bài báo đã được tôi sử dụng, trích dẫn trong luận văn vì đã cung cấp nguồn tư liệu quý báu và các kiến thức liên quan để tôi thực hiện luận văn.

# MỤC LỤC

[NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN i](#_Toc532459880)

[NHẬN XÉT CỦA GIẢNG VIÊN PHẢN BIỆN ii](#_Toc532459881)

[LỜI CẢM ƠN iii](#_Toc532459882)

[MỤC LỤC iv](#_Toc532459883)

[LIỆT KÊ HÌNH v](#_Toc532459884)

[CHƯƠNG 1: MỞ ĐẦU 1](#_Toc532459885)

[1.1 Đặt vấn đề 1](#_Toc532459886)

[1.2 Lý do chọn đề tài 1](#_Toc532459887)

[1.3 Mục tiêu đề tài 2](#_Toc532459888)

[CHƯƠNG 2: NỘI DUNG 3](#_Toc532459889)

[2.1 Microservices 3](#_Toc532459890)

[2.1.1 Định nghĩa 3](#_Toc532459891)

[2.1.2 Scale Cube 4](#_Toc532459892)

[2.1.3 Đặc điểm 6](#_Toc532459893)

[2.1.4 Ưu điểm 8](#_Toc532459894)

[2.1.5 Nhược điểm 8](#_Toc532459895)

[2.2 Docker 9](#_Toc532459896)

[2.2.1 Định nghĩa 9](#_Toc532459897)

[2.2.2 Các thành phần cơ bản 9](#_Toc532459898)

[2.2.3 Một số khái niệm 10](#_Toc532459899)

[2.2.4 So sánh Docker với Virtual Machine 10](#_Toc532459900)

[2.2.5 Các thành phần và công nghệ ảo hóa ứng dụng trong Docker 13](#_Toc532459901)

[2.2.6 Cách ảo hóa với Docker 19](#_Toc532459902)

[2.3 Gateway 24](#_Toc532459903)

[2.4 Eureka 25](#_Toc532459904)

[2.4.1 Định nghĩa 25](#_Toc532459905)

[2.4.2 Cách sử dụng Eureka 25](#_Toc532459906)

[2.5 Áp dụng kiến trúc Microservices vào một bài toán thực tế 28](#_Toc532459907)

[2.5.1 Phương án giải quyết 29](#_Toc532459908)

[2.5.2 Use case 30](#_Toc532459909)

[2.5.3 Kiến trúc microservice 36](#_Toc532459910)

[2.5.4 Product Service 36](#_Toc532459911)

[2.5.5 Review Service 39](#_Toc532459912)

[2.5.6 Order Service 40](#_Toc532459913)

[2.5.7 Authentication Service 43](#_Toc532459914)

[2.5.8 Application 44](#_Toc532459915)

[2.5.9 Triển khai 45](#_Toc532459916)

[CHƯƠNG 3: KẾT LUẬN 47](#_Toc532459917)

[TÀI LIỆU THAM KHẢO 48](#_Toc532459918)

# DANH MỤC HÌNH

[**Hình 2.1** Microservices và Monolithic Architecture 8](#_Toc532465199)

[**Hình 2.2** The Scale Cube 9](#_Toc532465200)

[**Hình 2.3** Mở rộng theo việc clone các service đằng sau bộ cân bằng tải 9](#_Toc532465201)

[**Hình 2.4** Mở rộng theo chức năng 10](#_Toc532465202)

[**Hình 2.5** Mở rộng theo trục X và Y 10](#_Toc532465203)

[**Hình 2.6** Mở rộng theo dữ liệu 11](#_Toc532465204)

[**Hình 2.7** Đặc tính của microservices 11](#_Toc532465205)

[**Hình 2.8** DevOps 12](#_Toc532465206)

[**Hình 2.9** Mô hình máy chủ truyền thống 15](#_Toc532465207)

[**Hình 2.10** Mô hình máy ảo VMs 16](#_Toc532465208)

[**Hình 2.11** Mô hình ảo hóa Container 17](#_Toc532465209)

[**Hình 2.12** Hệ thống file cắt lớp Container 17](#_Toc532465210)

[**Hình 2.13** Khác biệt giữa Docker và VMs 18](#_Toc532465211)

[**Hình 2.14** Kiến trúc Docker 20](#_Toc532465212)

[**Hình 2.15** Quy trình thực thi của một hệ thống sử dụng Docker 21](#_Toc532465213)

[**Hình 2.16** Tạo Spring Boot project 24](#_Toc532465214)

[**Hình 2.17** Cây thư mục Spring Boot project 25](#_Toc532465215)

[**Hình 2.18** Tạo Dockerfile trong project 27](#_Toc532465216)

[**Hình 2.19** Tạo thành công image cho project 28](#_Toc532465217)

[**Hình 2.20** Khởi chạy image 29](#_Toc532465218)

[**Hình 2.21** Microservice không sử dụng API Gateway 29](#_Toc532465219)

[**Hình 2.22** Microservice sử dụng API Gateway 30](#_Toc532465220)

[**Hình 2.23** Use Case Product Service 35](#_Toc532465221)

[**Hình 2.24** Use Case Authentication Service 36](#_Toc532465222)

[**Hình 2.25** Use Case Review Service 37](#_Toc532465223)

[**Hình 2.26** Use Case Order Service 38](#_Toc532465224)

[**Hình 2.27** Use Case Payment Service 39](#_Toc532465225)

[**Hình 2.28** Use Case Application 40](#_Toc532465226)

[**Hình 2.29** Kiến trúc microservice 41](#_Toc532465227)

[**Hình 2.30** Cấu trúc code Product Service 42](#_Toc532465228)

[**Hình 2.31** Cấu trúc code Review Service 44](#_Toc532465229)

[**Hình 2.32** Cấu trúc code Order Service 46](#_Toc532465230)

[**Hình 2.33** Cấu trúc code của Authentication Service 48](#_Toc532465231)

[**Hình 2.34** Cấu trúc code Application 49](#_Toc532465232)

[**Hình 2.35** Microservice triển khai trên docker 51](#_Toc532465233)

# DANH MỤC BẢNG

# 

[**Bảng 3.1** Mô tả Use Case Product Service 31](#_Toc532465347)

[**Bảng 3.2** Mô tả Use Case Authentication Service 32](#_Toc532465348)

[**Bảng 3.3** Mô tả Use Case Review Service 32](#_Toc532465349)

[**Bảng 3.4** Mô tả Use Case Order Service 33](#_Toc532465350)

[**Bảng 3.5** Mô tả Use Case Payment Service 34](#_Toc532465351)

[**Bảng 3.6** Mô tả Actor 35](#_Toc532465352)

[**Bảng 3.7** Mô tả Use Case Application 35](#_Toc532465353)

[**Bảng 3.8** Endpoint GET Product Service 38](#_Toc532465354)

[**Bảng 3.9** Endpoint POST Product Service 38](#_Toc532465355)

[**Bảng 3.10** Endpoint PUT Product Service 38](#_Toc532465356)

[**Bảng 3.11** Endpoint DELETE Product Service 38](#_Toc532465357)

[**Bảng 3.12** Endpoint GET Review Service 40](#_Toc532465358)

[**Bảng 3.13** Endpoint POST Review Service 40](#_Toc532465359)

[**Bảng 3.14** Endpoint PUT Review Service 40](#_Toc532465360)

[**Bảng 3.15** Endpoint DELETE Review Service 40](#_Toc532465361)

# CHƯƠNG 1: MỞ ĐẦU

## 1.1 Đặt vấn đề

Dịch vụ (service) ra đời và ngày càng phát triển đã giúp các nhà xây dựng và phát triển phần mềm tạo ra các hệ thống có khả năng thích ứng cao với nhiều môi trường khác nhau, tăng khả năng tái sử dụng. Các hệ thống được phát triển nhanh chóng và giảm được sự phức tạp, hạ giá thành khi xây dựng và triển khai. Tuy nhiên việc không quan tâm đến kích thước của các dịch vụ trong hệ thống đang đặt ra bài toán khó cho các nhà xây dựng và phát triển phần mềm là làm sao giảm chi phí khi xây dựng hệ thống với các dịch vụ, làm sao tránh ảnh hưởng đến cả hệ thống khi muốn thay đổi một số chức năng của hệ thống... Phạm vi của dịch vụ càng lớn hệ thống càng trở lên phức tạp, khó phát triển, kiểm thử và bảo trì. Chính những điều này đang làm cho việc xây dựng và phát triển hệ thống phần mềm dựa trên dịch vụ đang vượt khỏi khả năng kiểm soát của các kiểu kiến trúc phần mềm hiện có và cần phải có một kiểu kiến trúc mới để giải quyết vấn đề này.

Kiến trúc microservices là một kiểu kiến trúc phần mềm mới và đang rất phát triển hiện nay. Trong đó các dịch vụ được chia nhỏ để thực hiện một chức năng duy nhất của hệ thống. Việc chia nhỏ các dịch vụ trong kiến trúc microservices giúp cho hệ thống đơn giản hơn, dễ phát triển hơn, giảm chi phí xây dựng, tăng khả năng thích ứng công nghệ. Kiến trúc microservices được coi là lời giải ưu việt cho bài toán xây dựng và phát triển hệ thống dựa trên dịch vụ hiện nay. Nó đã và đang được nghiên cứu và ứng dụng rộng rãi bởi các công ty lớn như Netflix, Ebay, Amazon, Twitter, Paypal, Gilt, Soundcloud, ...Đặc biệt là hiện nay khi các sản phẩm phần mềm đóng gói đang dần đượcthay thế bởi các phần mềm dịch vụ thì kiến trúc microservices sẽ là đề tài ngày càng được quan tâm.

## 1.2 Lý do chọn đề tài

Xuất phát từ những ý nghĩa thực tiễn như vậy, nhóm đã thực hiện đề tài “Tìm hiểu kiến trúc microservice, triển khai ứng dụng trên kiến trúc đó với công nghệ Java” để tìm hiểu và áp dụng kiến trúc microservices để triển khai ứng dụng.

## 1.3 Mục tiêu đề tài

Tìm hiểu các công nghệ và kiến thức liên quan tới microservices:

* Docker
* Spring Boot
* Angular 6
* Gateway
* Eureka

Áp dụng các kiến thức về microservices để thiết kế một website bán hàng dựa vào kiến trúc microsevices.

# CHƯƠNG 2: NỘI DUNG

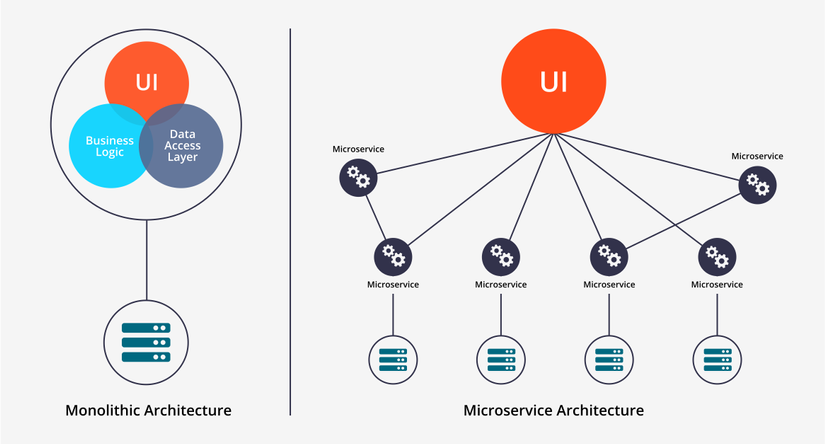
## 2.1 Microservices

### 2.1.1 Định nghĩa

Microservices là kiến trúc tập hợp các service nhỏ, độc lập hoạt động cùng nhau.

Mỗi microservice sẽ chịu trách nhiệm riêng, các team có thể phát triển chúng độc lập với các microservice khác.

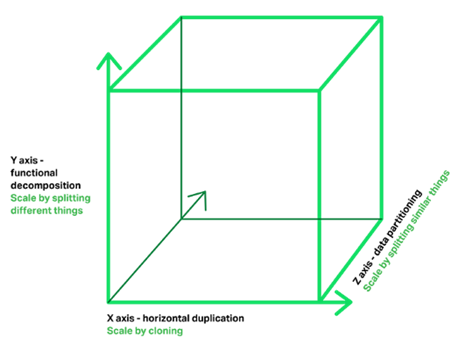
Chúng giao tiếp với nhau qua rất nhiều phương thức có thể là Rest API hoặc là GRPC hoặc Lambda hoặc bất cứ phương thức nào có thể giao tiếp được.



**Hình 2.1** Microservices và Monolithic Architecture

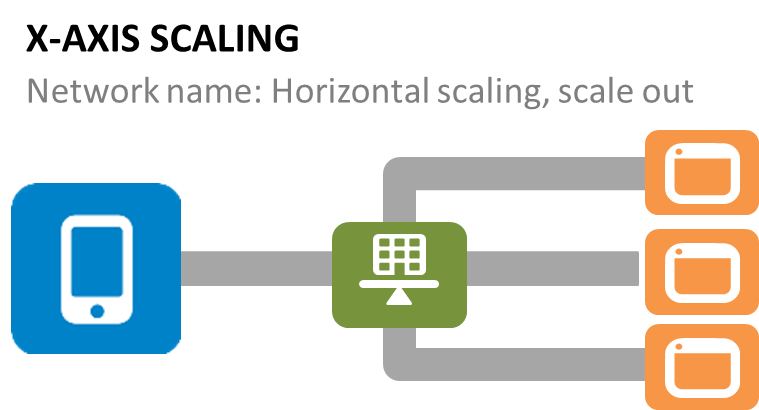
### 2.1.2 Scale Cube

Một trong những ưu điểm lớn nhất của microservices là về khả năng mở rộng.

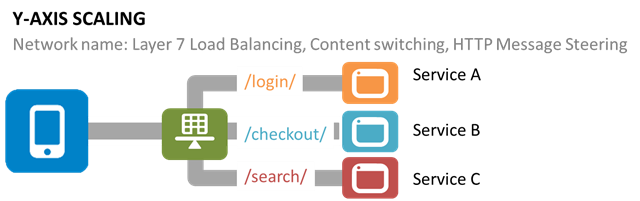


**Hình 2.2** The Scale Cube

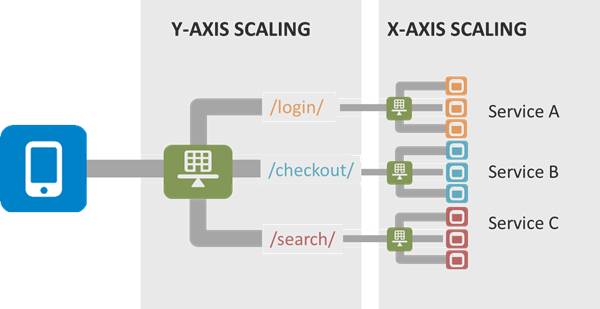
Kiến trúc microservice tương đương trục Y của 3 chiều mở rộng chịu tải (Scale Cube).



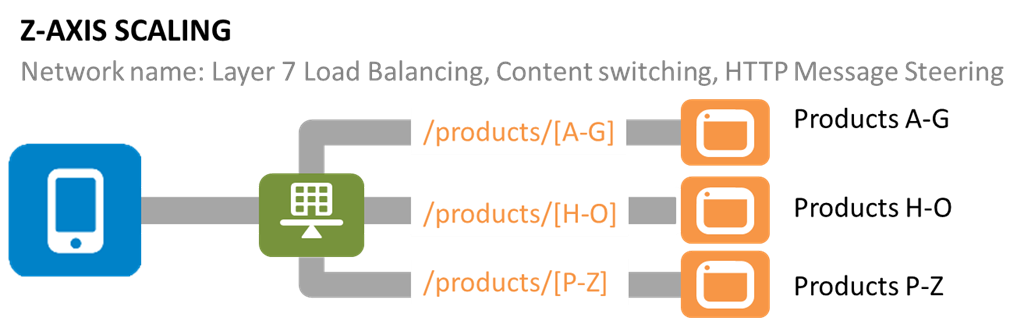
**Hình 2.3** Mở rộng theo việc clone các service đằng sau bộ cân bằng tải



**Hình 2.4** Mở rộng theo chức năng

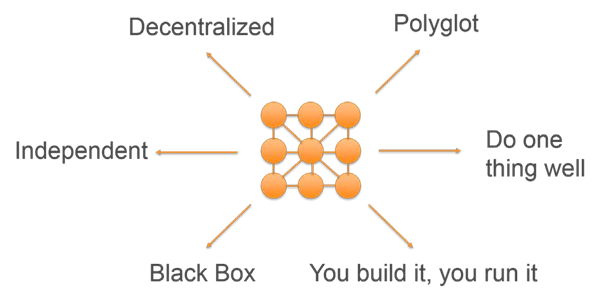


**Hình 2.5** Mở rộng theo trục X và Y



**Hình 2.6** Mở rộng theo dữ liệu

### 2.1.3 Đặc điểm



**Hình 2.7** Đặc tính của microservices

#### 2.1.3.1 Decentralized

Microservices là một hệ thống phân tán về:

Database – Data model

Develop

Deploy

Manage

Operate

#### 2.1.3.2 Independent

Mỗi service đều có thể thay đổi, nâng cấp hoặc thậm chí thay thế mà không ảnh hưởng tới chức năng các service khác.

#### 2.1.3.3 Do one thing well

Mỗi service sẽ được phát triển dựa trên các chức năng của 1 domain nhất định.

Như monolithic có quá nhiều thức để focus đến khi ứng dụng lớn, phức tạp sẽ bị break => tính hỗn độn, phức tạp, lộn xộn của - 1 trong 3 thách thức phát triển phần mềm (Heterogeneity, Delivery, Sercurity & Trust).

#### 2.1.3.4 Polygot

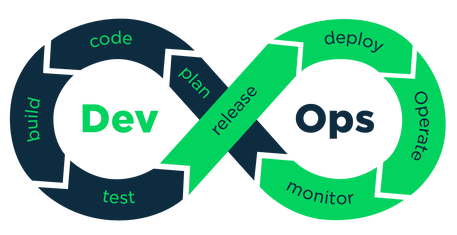
Microservices cho phép các team phát triển tự do chọn tool, ngôn ngữ, data stored.

#### 2.1.3.5 Black box

Service bị ẩn đi, mọi giao tiếp đều thông qua API.

#### 2.1.3.6 You build it, you run it

Chuỗi các hoạt động phát triển và vận hành ứng dụng do 1 team.



**Hình 2.8** DevOps

### 2.1.4 Ưu điểm

 Sử dụng đa ngôn ngữ, mỗi service có thể được phát triển bằng bất kỳ ngôn ngữ lập trình nào.

 Nếu 1 component của hệ thống có vấn đề, component đó sẽ bị cô lập và phần còn lại của hệ thống vẫn hoạt động bình thường.

Có khả năng mở rộng dễ dàng, chỉ cần mở rộng những service nào cần mở rộng.

Có thể deploy 1 service độc lập với các phần còn lại của hệ thống. Điều này giúp chúng ta deploy nhanh hơn, cũng đồng nghĩa với việc có thể đưa ra tính năng mới đến khách hàng nhanh hơn.

 Giảm thiểu sự gia tăng phức tạp của một hệ thống lớn.

Tối ưu cho việc thay thế các service, các service nhỏ nên việc viết lại khi được yêu cầu không là vấn đề vì chi phí thay thế rất thấp.

### 2.1.5 Nhược điểm

Thuật ngữ Microservice tạo ấn tượng cảm giác về kích thước của nó. Trên thực tế nhiều developer ủng hộ chỉ xây dựng service từ dưới 100 dòng code. Service nhỏ là tốt, nhưng nó không phải mục tiêu chính của microservices. Mục tiêu của mircoservices là phân nhỏ đầy đủ ứng dụng để tạo điều kiện phát triển và triển khai ứng dụng nhanh chóng.

Ứng dụng microservices là một hệ thống phân tán. Các developer cần phải lựa chọn phát triển mỗi dịch vụ nhỏ giao tiếp với các dịch vụ khác bằng cách nào messaging hay là RPC. Hơn nữa, họ cũng phải viết code để xử lý việc thất bại giữa chừng (partial failure) vì điểm đến của request có thể chậm hoặc không khả dụng. Việc này phức tạp hơn nhiều so với ứng dụng nguyên khối nơi các module gọi nhau thông qua các method/procedure cấp ngôn ngữ.

Phải đảm bảo giao dịch phân tán (distributed transaction) cập nhật dữ liệu đúng đắn (all or none) vào nhiều dịch vụ nhỏ khác nhau khó hơn rất nhiều.

Test một ứng dụng microservices cũng phức tạp hơn nhiều. Testing một service trong kiến trúc microservices đôi khi yêu cầu phải chạy cả các dịch vụ nhỏ khác mà nó phụ thuộc. Do đó khi phân rã ứng dụng một khối thành microservices cần luôn kiểm tra mức độ ràng buộc giữa các dịch vụ.

Việc triển khai microservices phức tạp hơn rất nhiều.

## 2.2 Docker

### 2.2.1 Định nghĩa

Docker là một open platform cung cấp cho người sử dụng những công cụ và service để người sử dụng có thể đóng gói và chạy chương trình của mình trên các môi trường khác nhau một cách nhanh nhất.

Docker cho phép các lập trình viên, quản trị hệ thống dùng để xây dựng, vận chuyển và chạy các ứng dụng phân tán. Ban đầu viết bằng Python, hiện tại đã chuyển sang Go-lang.

Docker đưa ra một giải pháp mới cho vấn đề ảo hóa, thay vì tạo ra các máy ảo con chạy độc lập kiểu hypervisors (tạo phần cứng ảo và cài đặt hệ điều hành lên đó), các ứng dụng sẽ được đóng gói lại thành các Container riêng lẻ. Các Container này chạy chung trên nhân hệ điều hành qua LXC (Linux Containers), chia sẻ chung tài nguyên của máy mẹ, do đó, hoạt động nhẹ và nhanh hơn các máy ảo dạng hypervisors.

### 2.2.2 Các thành phần cơ bản

- Docker Engine: là thành phần chính của Docker, quản lý việc tạo image, chạy container, dùng image có sẵn hay tải image chưa có về, kết nối vào container, thêm, sửa, xóa image và container, .....

- Docker Hub: là dịch vụ cloud để chia sẻ ứng dụng và tự động hóa chuỗi các công việc liên tục, có thể thao tác pull/push với các images. Nó giống như github, bitbucket.

- Docker Image: là file ảnh, file nền của một hệ điều hành, một nền tảng, một ngôn ngữ (vd: ubuntu image, ruby image, rails image, mysql image…). Từ các image này, bạn sẽ dùng nó để tạo ra các container. Các image là dạng read only.

- Docker Container: Hoạt động giống như một thư mục (directory), chứa tất cả những thứ cần thiết để một ứng dụng có thể chạy được. Mỗi một docker container được tạo ra từ một docker image. Các thao tác với một container : chạy, bật, dừng, di chuyển, và xóa.

+ Các container này sẽ dùng chung tài nguyên của hệ thống (RAM, Disk, Network…), chính nhờ vậy, những container sẽ rất nhẹ, việc khởi động, kết nối, tương tác sẽ rất nhanh gọn.

### 2.2.3 Một số khái niệm

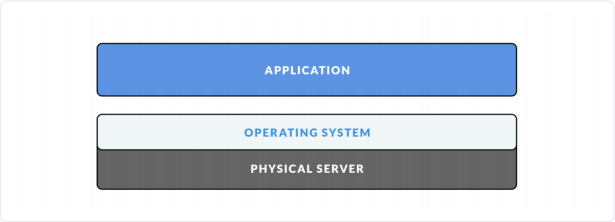
Dockerfile : là một file chứa tập hợp các lệnh để Docker có thể đọc và thực hiện để đóng gói một image theo yêu cầu người dùng.

Orchestration : là các công cụ, dịch vụ dùng để điều phối và quản lý nhiều containers sao cho chúng làm việc hiệu quả nhất.

Docker Compose: chạy ứng dụng bằng cách định nghĩa cấu hình các Docker container thông qua tệp cấu hình.

### 2.2.4 So sánh Docker với Virtual Machine

Ngày xưa, mô hình máy chủ thường là 1 máy chủ vật lý + 1 hệ điều hành (OS) + 1 application.

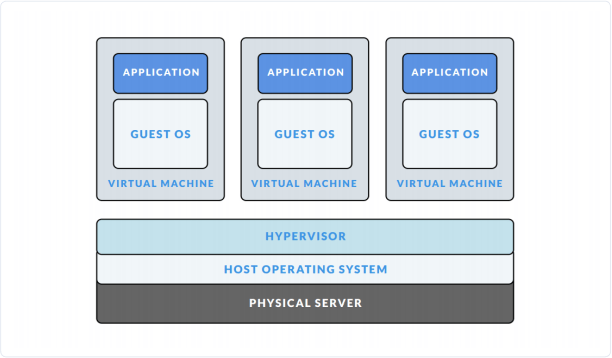


**Hình 2.9** Mô hình máy chủ truyền thống

Khi ứng dụng phát triển lên, mô hình này nảy sinh ra nhiều vấn đề, ví dụ:

* Lãng phí tài nguyên: mặc dù cấu hình máy khỏe, ổ cứng dung lượng lớn, nhưng hệ thống lại không tận dụng được hết lợi thế này;
* Khó khăn trong việc mở rộng hệ thống: muốn mở rộng phải thuê thêm server, cấu hình, cân bằng tải (load balacing), …

Lúc này, công nghệ ảo hóa (vitualization) ra đời.

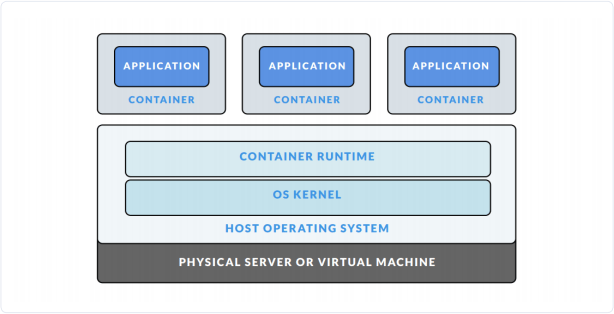


**Hình 2.10** Mô hình máy ảo VMs

Với công nghệ ảo hóa, trên cùng 1 máy chủ vật lý, có thể tạo ra nhiều OS, tức là sẽ chạy được nhiều application. Vậy là tài nguyên của máy được tận dụng tốt hơn. Tuy nhiên, việc ảo hóa này lại nảy sinh vấn đề mới:

* Ngốn tài nguyên: khi chạy 1 máy ảo, nó sẽ luôn chiếm 1 phần tài nguyên cố định. Vd: máy chủ có 512GB SSD, 16GB RAM. Tạo ra 4 máy ảo Linux, mỗi máy cấp 64GB SSD và 2GB RAM. Như vậy, sẽ mất 256 GB SSD để chứa 4 máy ảo, và khi chạy cùng 4 máy ảo lên cùng lúc, chúng sẽ chiếm 8GB RAM. Mặc dù chỉ chạy lên để không đó thôi, chưa dùng gì cả nhưng nó vẫn chiếm từng đó;
* Tốn thời gian thực thi: thời gian khởi động, shutdown của các máy ảo sẽ lâu, thường là hàng phút;
* Cồng kềnh: việc phải chịu tải cho cả 1 nhóm máy ảo như vậy thì server không thế chạy hết hiệu suất được.

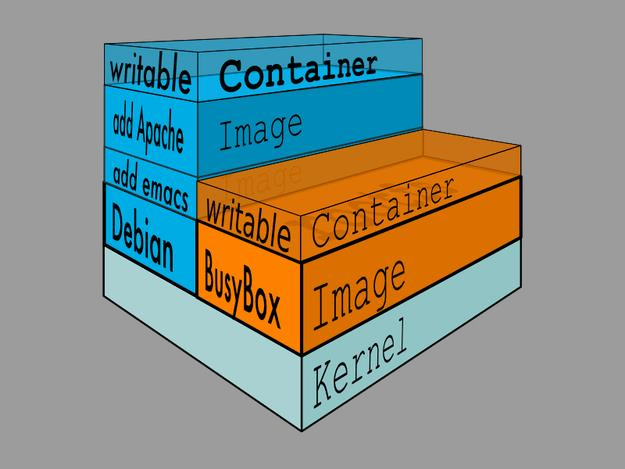
Bước tiến hóa tiếp theo, người ta phát minh ra containerlization.



**Hình 2.11** Mô hình ảo hóa Container

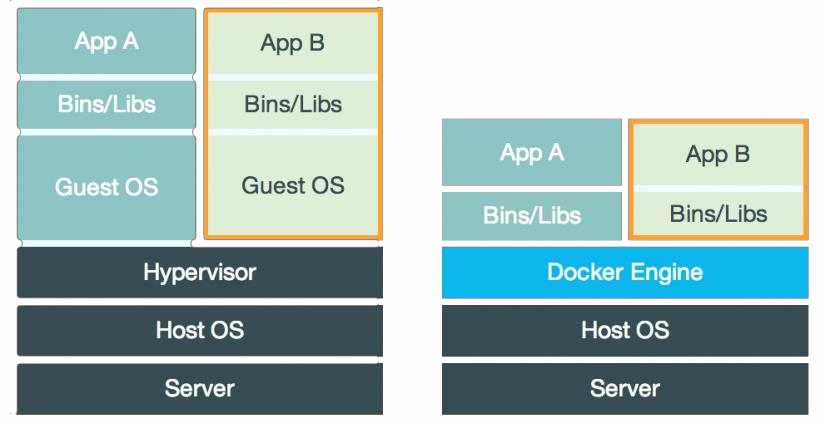
Với công nghệ này, trên một máy chủ, ta sẽ sinh ra được nhiều máy con (giống với ảo hóa), nhưng điều đặc biệt là các máy con (Guess OS) này đều dùng chung phần nhân của máy mẹ (host OS) và chia sẻ với nhau tài nguyên của máy mẹ (RAM chẳng hạn). Như vậy việc tận dụng tài nguyên sẽ được tối ưu hơn.

Ngoài ra, việc sử dụng hệ thống file cắt lớp (layer file system) sẽ khiến việc tối ưu tài nguyên hiệu quả hơn.



**Hình 2.12** Hệ thống file cắt lớp Container

Cụ thể, mỗi máy con (container) mới, nó sẽ được xây dựng dựa trên 1 file ảnh (image) dạng chỉ đọc (read-only). Trong mỗi máy con sẽ có thêm 1 lớp bọc có-thể- ghi-được (writabe-layer), các thay đổi trong máy con sẽ được ghi lên đây. Như vậy, từ 1 image ban đầu, ta có thể tạo nhiều máy con mà chỉ tốn rất ít dung lượng ổ đĩa.



**Hình 2.13** Khác biệt giữa Docker và VMs

Điểm khác biệt chính là các containers sử dụng chung kernel với Host OS nên các thao tác bật, tắt rất nhẹ nhàng, nhanh chóng.

* Ưu điểm: nhanh, nhẹ, có thể chia sẻ dễ dàng qua DockerHub;
* Nhược điểm : mới, cập nhật thay đổi thường xuyên.

### 2.2.5 Các thành phần và công nghệ ảo hóa ứng dụng trong Docker

#### 2.2.5.1 Các thành phần

Docker Images:

* + Là file ảnh, file nền của một hệ điều hành, một nền tảng, một ngôn ngữ (vd: ubuntu image, ruby image, rails image, mysql image…);
  + Từ các image này, bạn sẽ dùng nó để tạo ra các container;
  + Các image là dạng file-chỉ-đọc (read only file);
  + Tự bạn cũng có thể tạo image cho riêng mình;
  + Một image có thể được tạo từ nhiều image khác (vd: bạn tạo 1 image chạy ubuntu, có cài sẵn ruby 2.3 và rails 5, image này của bạn được tạo nên bởi 3 image khác).

Docker Images: Là kho chứa images. Người dùng có thể tạo ra các images của mình và tải lên đây hoặc tải về các images được chia sẻ.

Docker container: hoạt động giống như một thư mục (directory), chứa tất cả những thứ cần thiết để một ứng dụng có thể chạy được. Mỗi một docker container được tạo ra từ một docker image. Các thao tác với một container: chạy, bật, dừng, di chuyển, và xóa.

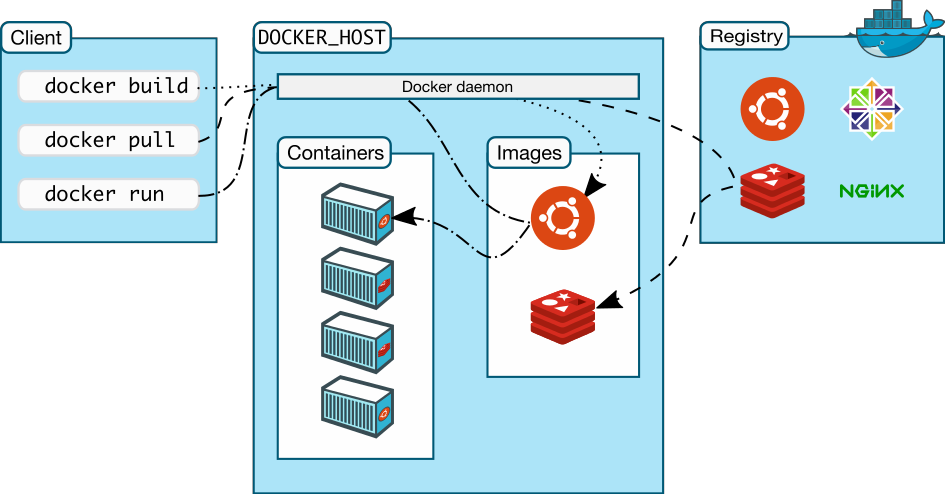
* + Là một máy ảo, được cấu thành từ 1 image và được đắp thêm 1 lớp “trang trí” writable-file-layer. Các thay đổi trong máy ảo này (cài thêm phần mềm, tạo thêm file…) sẽ được lưu ở lớp trang trí này;
  + Các container này sẽ dùng chung tài nguyên của hệ thống (RAM, Disk, Network…), chính nhờ vậy, những container của bạn sẽ rất nhẹ, việc khởi động, kết nối, tương tác sẽ rất nhanh gọn;
  + Nếu ánh xạ sang hướng đối tượng, thì image chính là class, còn container chính là instance-1 thể hiện của class đó. Từ 1 class ta có thể tạo ra nhiều instance, tương tự, từ 1 image ta cũng có thể tạo ra được nhiều container hoàn toàn giống nhau.

Dockerfile: là một file chứa tập hợp các lệnh để Docker có thể đọc và thực hiện để đóng gói một image theo yêu cầu người dùng.

Orchestration: là các công cụ, dịch vụ dùng để điều phối và quản lý nhiều containers sao cho chúng làm việc hiệu quả nhất.

#### 2.2.5.2 Kiến trúc của Docker

Docker sử dụng kiến trúc client-server. Docker client sẽ nói liên lạc với các Docker daemon, các Docker daemon sẽ thực hiện các tác vụ build, run và distribuing các Docker container. Cả Docker client và Docker daemon có thể chạy trên cùng 1 máy, hoặc có thể kết nối theo kiểu Docker client điều khiển các docker daemon như hình dưới. Docker client và daemon giao tiếp với nhau thông qua socker hoặc RESTful API.



**Hình 2.14** Kiến trúc Docker

*Docker Daemon:* Như thế hiện trên biểu đồ phía trên, Docker daemon chạy trên các máy host. Người dùng sẽ không tương tác trực tiếp với các daemon, mà thông qua Docker Client.

*Docker Client:* Là giao diện người dùng của Docker, nó cung cấp cho người dùng giao diện dòng lệnh và thực hiện phản hồi với các Docker Daemon.

*Docker Images:* Là một template chỉ cho phép đọc, ví dụ một image có thể chứa hệ điều hành Ubuntu và web app. Images được dùng để tạo Docker container. Docker cho phép chúng ta build và cập nhật các image có sẵn một cách cơ bản nhất, hoặc bạn có thể download Docker images của người khác.

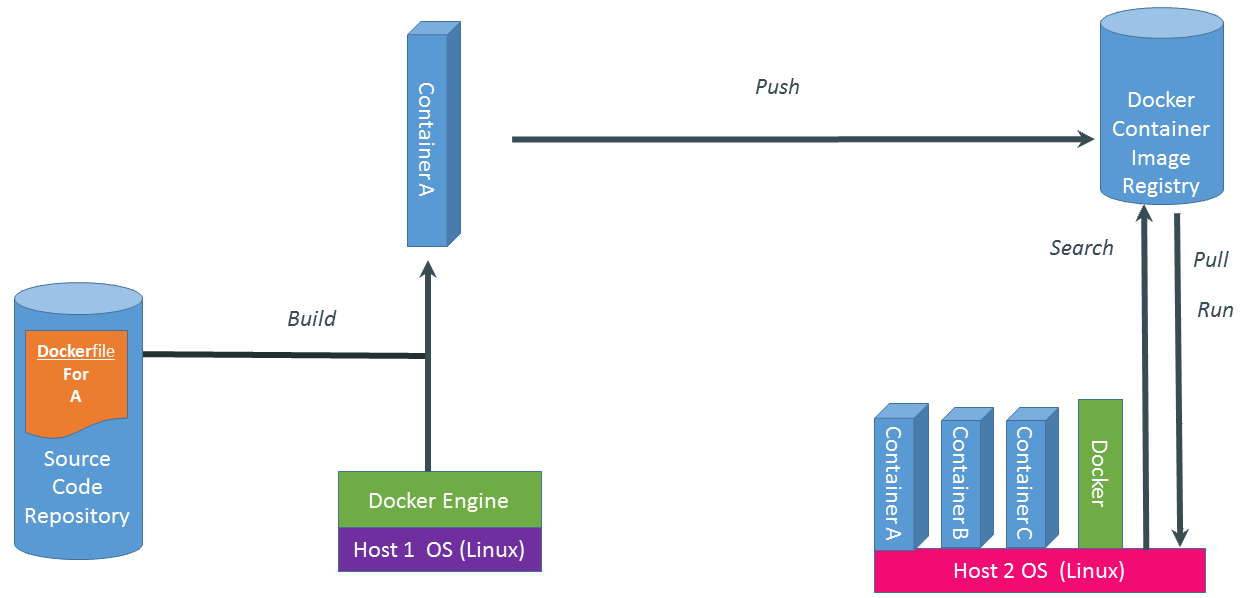
*Docker Container:* Một Docker container giữ mọi thứ chúng ta cần để chạy một app. Mỗi container được tạo từ Docker image. Docker container có thể có các trạng thái run, started, stopped, moved và deleted.

#### 2.2.5.3 Ưu điểm hình thức đóng gói thành Container

Việc đóng gói thành các container này có thể giải quyết được nhiều vấn đề mà ta chưa đề cập tới.

* + - * Ví dụ như trước kia ta không thể dùng chung Port, thì ở đây 2 ứng dụng với 2 container khác nhau. Ta có thể cấu hình Port trùng nhau cho ứng dụng này;
      * Tiếp theo là về việc quản lí phiên bản. Ta khó có thể cài 2 phiên bản của 1 phần mềm trên cùng 1 máy hypervisor. Tuy nhiên với Container, ta có thể cài mỗi phiên bản trên 1 Container và chạy một cách trơn tru;
      * Khả năng khởi động nhanh của Docker cũng là một lợi thế rất lớn;
      * Tiếp theo nói về tài nguyên, Docker sẽ ngốn ít tài nguyên hơn các máy hypervisor.

#### 2.2.5.4 Quy trình thực thi của một hệ thống sử dụng Docker.



**Hình 2.15** Quy trình thực thi của một hệ thống sử dụng Docker

Như trong hình vẽ, một hệ thống Docker được thực thi với 3 bước chính:

*Build -> Push -> Pull,Run.*

Cụ thể hơn về nguyên lí 3 bước này:

− *Build*:

* Đầu tiên chúng ta sẽ tạo một dockerfile, trong dockerfile này chính là code của chúng ta.
* Dockerfile này sẽ được Build tại một máy tính đã cài đặt Docker Engine.
* Sau khi build ta sẽ thu được Container, trong Container này chứa bộ thư viện và ứng dụng của chúng ta.

*Push*:

* Sau khi có được Container, chúng ta thực hiện push Container này lên đám mây và lưu trữ ở đó.
* Việc push này có thể thực hiện qua môi trường mạng Internet.

*Pull, Run:*

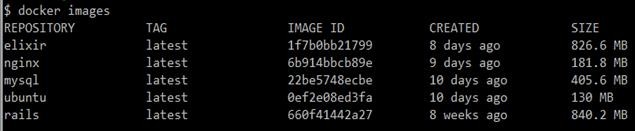
* Giả sử một máy tính muốn sử dụng Container chúng ta đã push lên đám mây (máy đã cài Docker Engine) thì bắt buộc máy phải thực hiện việc Pull container này về máy. Sau đó thực hiện Run Container này.

Đó chính là quy trình 3 bước rất đơn giản và rõ ràng miêu tả hoạt động của một hệ thống sử dụng Docker.

#### 2.2.5.5 Các lệnh cơ bản

**Hiển thị danh sách các images**

|  |
| --- |
| *docker images* |



**Tải image về local**

|  |
| --- |
| *docker pull <name\_image:tag>* |

(Phần :tag là option, nếu để trống thì mặc định là version latest).

Ví dụ: *docker pull ubuntu*  download ubuntu latest.

*docker pull ubuntu:14.04*  download ubuntu 14.04.

Truy cập trang <https://hub.docker.com/>, nơi lưu trữ các images tập trung để tìm images cần dùng.

**Chạy một image**

|  |
| --- |
| *docker run –name <tên\_container> -v <thư mục trên máy tính>:<thư mục trong container> -p<port\_máy tính>:<port\_container> <image name> bash* |

Ví dụ: *docker run –name eva\_nginx -p 80:80 -d nginx*

* + docker run : lệnh chạy của docker;
  + –name: đặt tên cho container ở đây là eva\_nginx . Name này là duy nhất, không thể tạo trùng, nếu không đặt thì docker tự genate;
  + -p mở port container ra ngoài IP public;
  + -d bật chế độ chạy background;
  + nginx: tên images.

**Liệt kê các container**

|  |
| --- |
| *docker ps -a* |

Câu lệnh trên sẽ liệt kê toàn bộ các container hiện có.

|  |
| --- |
| *docker ps* |

Câu lệnh trên sẽ liệt kê toàn bộ các container hiện đang chạy.

**Dừng container đang chạy**

|  |
| --- |
| *docker stop <container\_id hoặc name\_container>* |

Câu lệnh sẽ dừng các container có id là “container\_id” hoặc name là “name\_container”.

|  |
| --- |
| *docker stop $(docker ps –a –q)* |

Câu lệnh dừng toàn bộ container đang chạy.

**Khởi động lại container đã dừng**

|  |
| --- |
| *docker start <container\_id hoặc name\_container>* |

**Xóa container không còn sử dụng**

|  |
| --- |
| *docker rm <container\_id hoặc name\_container>* |

Câu lệnh sẽ xóa các container có id là “container\_id” hoặc name là “name\_container”.

|  |
| --- |
| *docker rm $(docker ps -a -q)* |

Câu lệnh xóa toàn bộ container.

**Truy cập vào 1 container đang chạy**

|  |
| --- |
| *docker exec -it <container\_id hoặc name\_container> bash* |

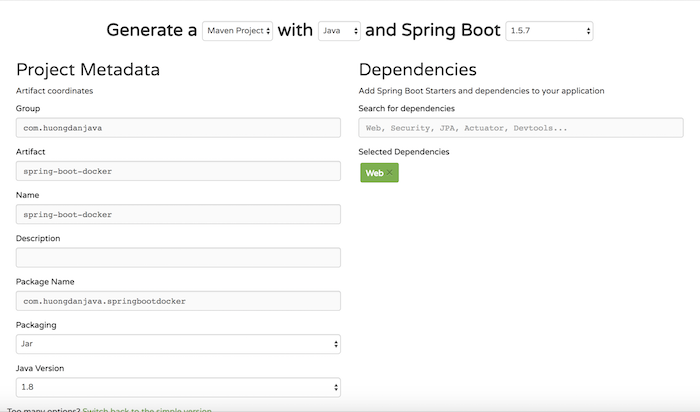
Ngoài ra, còn rất nhiều câu lệnh khác. Sử dụng lệnh *docker -h* để biết thêm chi tiết.

### 2.2.6 Cách ảo hóa với Docker

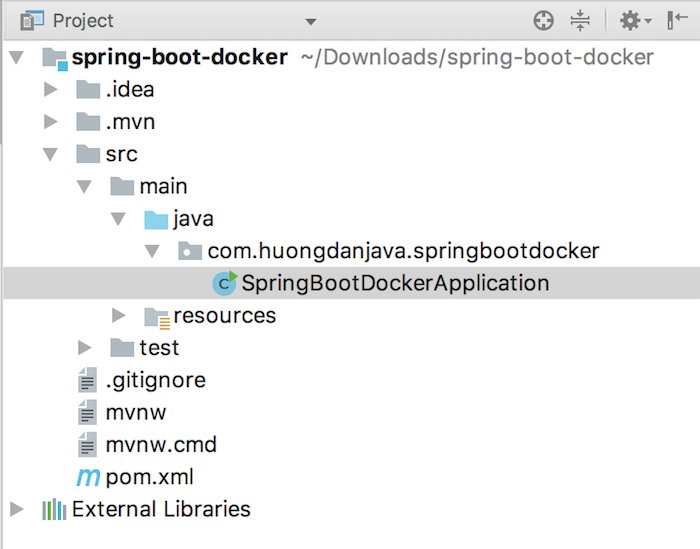
Phần này sẽ hướng dẫn ảo hóa 1 ứng dụng bằng cách build 1 image, image sẽ là 1 webservice viết bằng java. Với các bước tương tự có thể build các image khác tùy vào mục đích sử dụng.

**Bước 1: Tạo 1 app bằng Spring Boot.**

Chúng tôi sử dụng Spring Boot Initializr Web để tạo một Spring Boot project:



**Hình 2.16** Tạo Spring Boot project



**Hình 2.17** Cây thư mục Spring Boot project

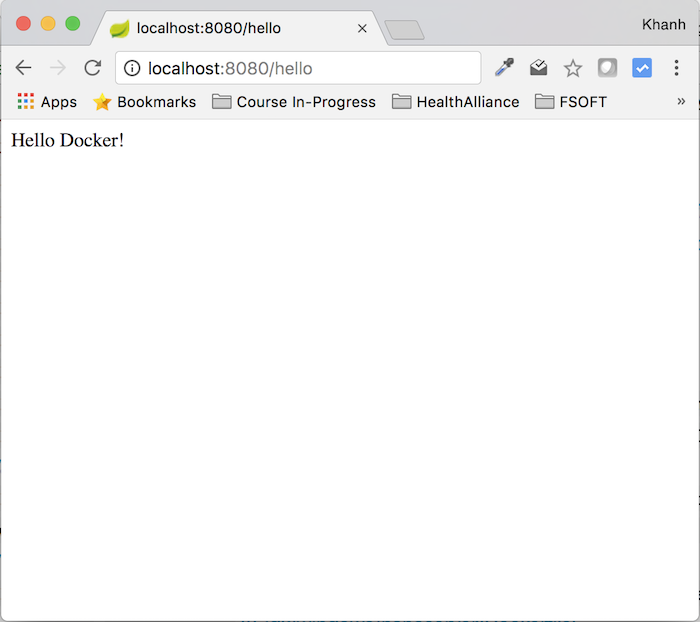
Mở tập tin SpringBootDockerApplication:

|  |
| --- |
| package com.huongdanjava.springbootdocker;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  public class SpringBootDockerApplication {  public static void main(String[] args) {  SpringApplication.run(SpringBootDockerApplication.class, args);  }  } |

Và modify nó một chút để thêm một RESTful Web Service return về chuỗi “Hello Docker!” như sau:

|  |
| --- |
| package com.huongdanjava.springbootdocker;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  @SpringBootApplication  @RestController  public class SpringBootDockerApplication {  @RequestMapping("/hello")  public String helloDocker() {  return "Hello Docker!";  }  public static void main(String[] args) {  SpringApplication.run(SpringBootDockerApplication.class, args);  }  } |

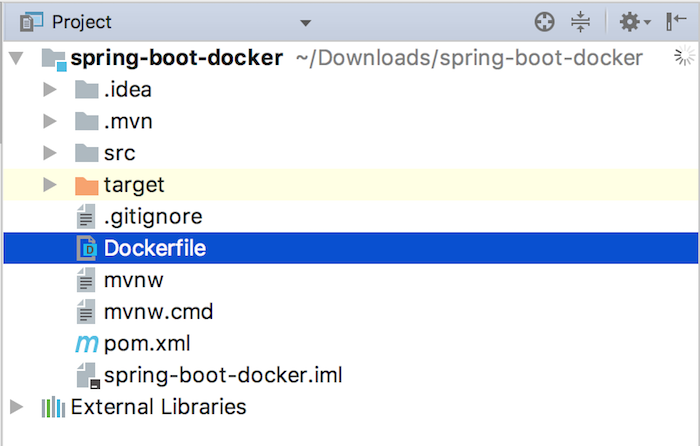
Kết quả:



Bây giờ, chúng tôi viết một Dockerfile để deploy ứng dụng lên Docker Container.

**Bước 2: Viết Dockerfile**

Tạo mới một tập tin Dockerfile nằm trong thư mục của project.



**Hình 2.18** Tạo Dockerfile trong project

Bây giờ sửa Dockerfile này để xây dựng một Docker Image.

Đầu tiên là sử dụng một Image trên Docker Hub để build Image cho project.

Image đó có tên là openjdk:8 nằm trong repository openjdk. Do đó, câu lệnh FROM sẽ có nội dung như sau:

|  |
| --- |
| FROM openjdk:8 |

Tiếp theo, copy tập tin spring-boot-docker-0.0.1-SNAPSHOT.jar nằm trong thư mục target của project vào Docker Container.

|  |
| --- |
| ADD target/spring-boot-docker-0.0.1-SNAPSHOT.jar app.jar |

Và cuối cùng thực thi câu lệnh để chạy ứng dụng Spring Boot của chúng ta mỗi khi Docker Container của chúng ta được chạy.

|  |
| --- |
| ENTRYPOINT exec java -jar app.jar |

Nội dung toàn bộ của Dockerfile như sau:

|  |
| --- |
| FROM openjdk:8  ADD target/spring-boot-docker-0.0.1-SNAPSHOT.jar app.jar  ENTRYPOINT exec java -jar app.jar |

**Bước 3: Deploy ứng dụng lên Docker Container**

Để deploy ứng dụng Spring Boot lên Docker Container từ Dockerfile, trước tiên phải tạo Docker Image từ Dockerfile.

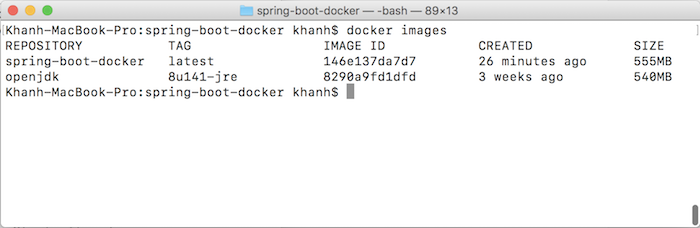
Mở Terminal lên và đi đến thư mục của project, sau đó hãy nhập dòng lệnh sau:

|  |
| --- |
| mvn clean package && docker build -t spring-boot-docker . |

Câu lệnh trên có 2 phần:

* Phần đầu tiên là chúng ta sẽ dùng Maven để build ứng dụng Spring Boot thành file jar.
* Phần tiếp theo là dùng Docker để build Docker Image từ Dockerfile.

Sau khi chạy câu lệnh trên, kiểm tra tất cả Docker Image, các bạn sẽ thấy các Docker Image như sau:



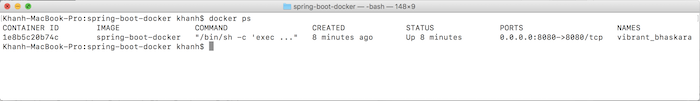
**Hình 2.19** Tạo thành công image cho project

Trong đó có Docker Image mà chúng ta vừa tạo.

Và bây giờ thì chạy Docker Container từ Docker Image vừa được tạo ra.

|  |
| --- |
| docker run -p 8080:8080 -t spring-boot-docker |

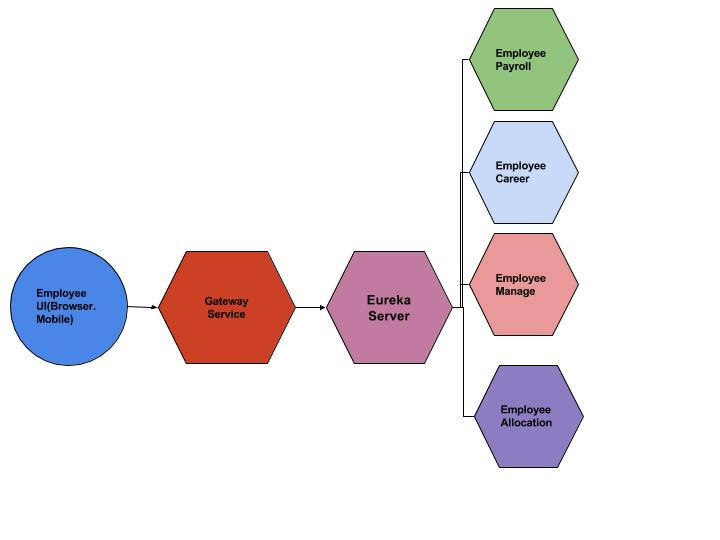
Ở đây, có 2 số 8080 thì số phía trước là port của máy, số phía sau là port của docker. Có thể kiểm tra kết quả bằng cách vào <http://localhost:8080/hello> .



**Hình 2.20** Khởi chạy image

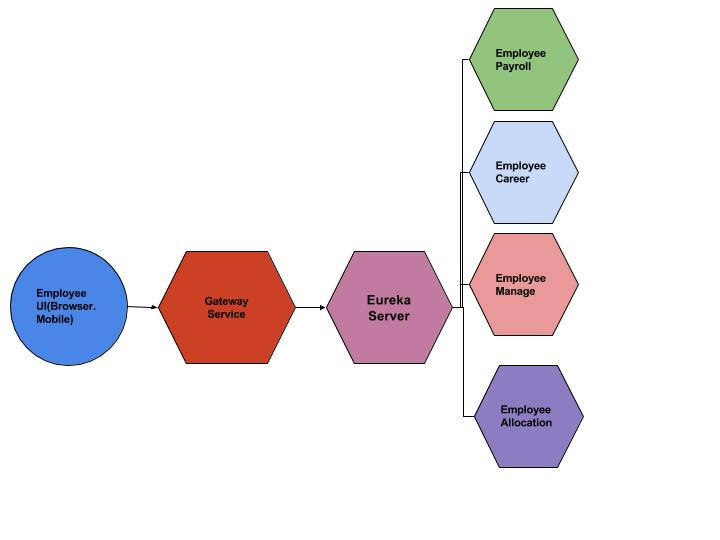
## 2.3 Gateway

Mấu chốt của microservices là tạo ra service không phụ thuộc có thể mở rộng và triển khai một cách độc lập. Trong thực tế, một hệ thống có từ 50-100 microservices là rất phổ biến. Hãy tưởng tượng một dịch vụ có 50 microservices, khi developer UI muốn lấy data từ 50 microservices thì phải biết chi tiết tất cả các API, URL, port để gọi chúng. Chắc chắn đó không phải là 1 thiết kế tốt. Vì vậy chúng ta phải cần một API Gateway.



**Hình 2.21** Microservice không sử dụng API Gateway

API Gateway làm nhiệm vụ định tuyến các yêu cầu, kết hợp và chuyển đổi các giao thức. Tất cả yêu cầu từ Client đều đi qua cổng kết nối API. Sau đó API Gateway định tuyến các yêu cầu này tới microservice phù hợp. API Gateway sẽ xử lý một yêu cầu người dùng bằng cách gọi đến một loạt microservice rồi tổng hợp các kết quả.



**Hình 2.22** Microservice sử dụng API Gateway

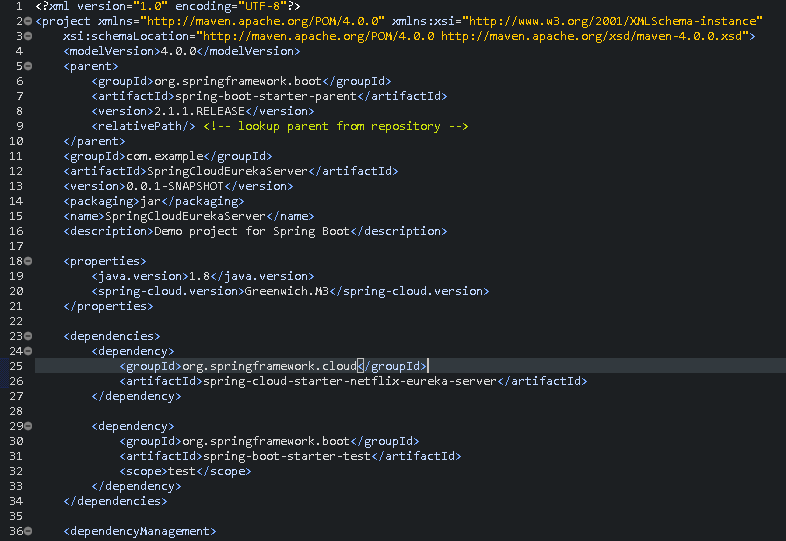
## 2.4 Eureka

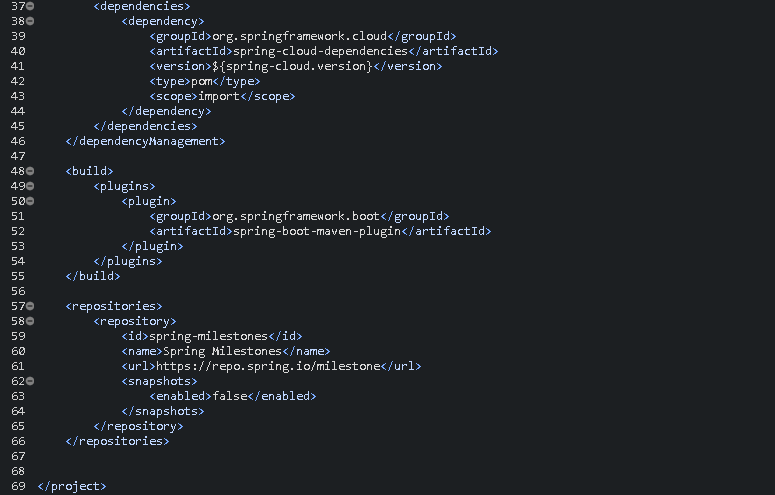
### 2.4.1 Định nghĩa

Eureka Server chịu trách nhiệm lưu trữ và cung cấp thông tin của các service trong một hệ thống Microservice. Khi một service đăng ký thông tin với server, nó sẽ cung cấp các thông tin như host, port, trạng thái của nó. Và nó cũng thường xuyên gửi heartbeat message để thông báo tình trạng của mình cho server.

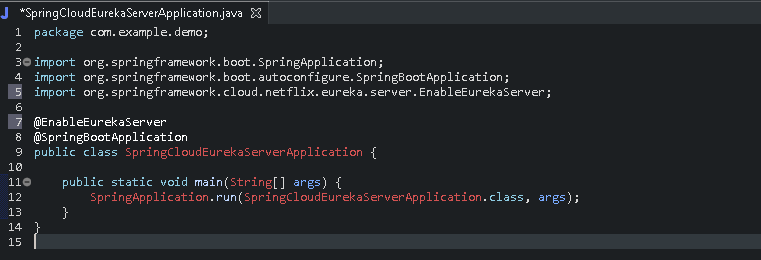
### 2.4.2 Cách sử dụng Eureka

Tạo một project Spring boot với file **pom.xml**

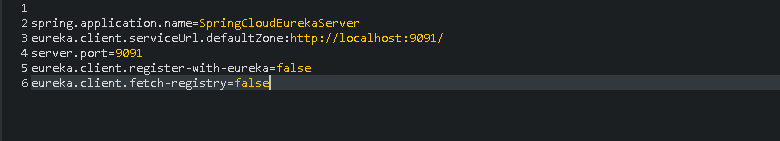




Để ứng dụng trở thành một **Service Registration**( Đăng ký dịch vụ) bạn cần sử dụng **@EnableEurekaServer**

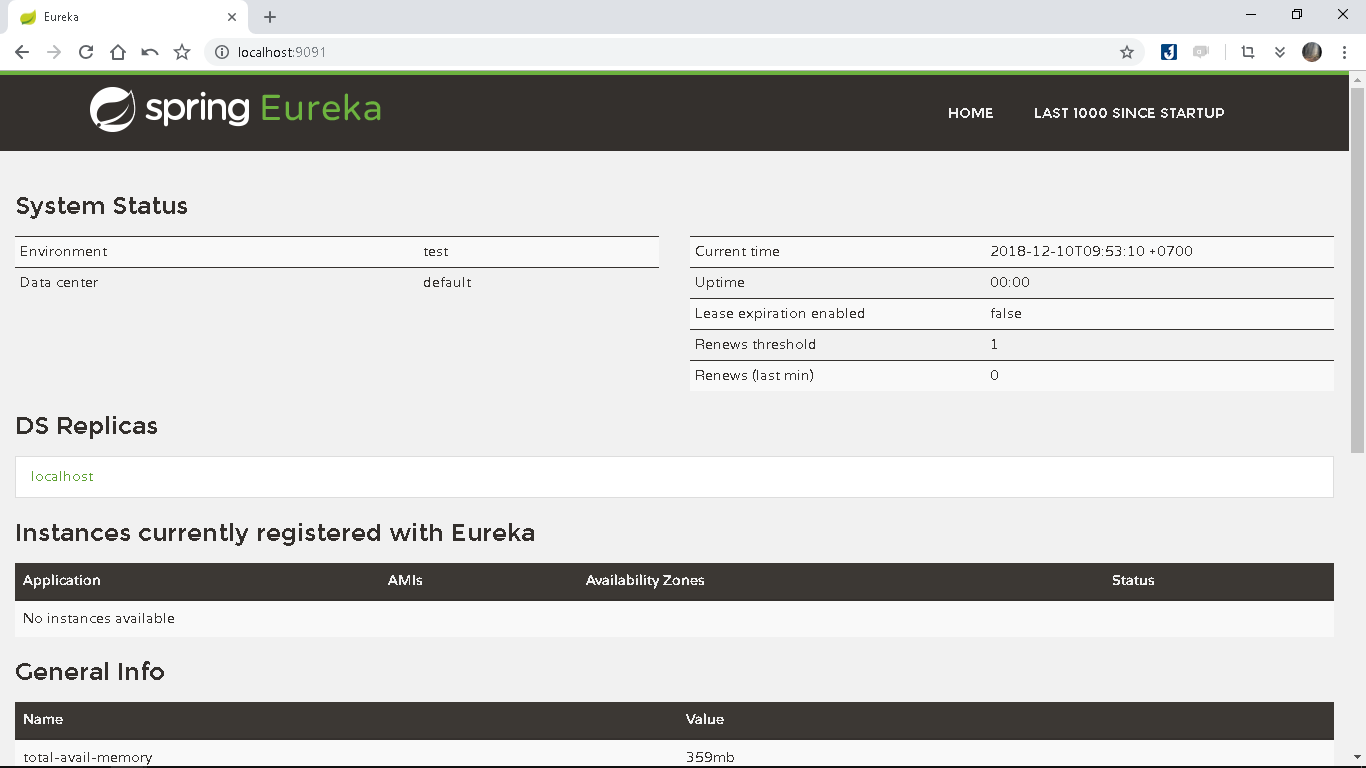


Tạo file bootstrap.properties

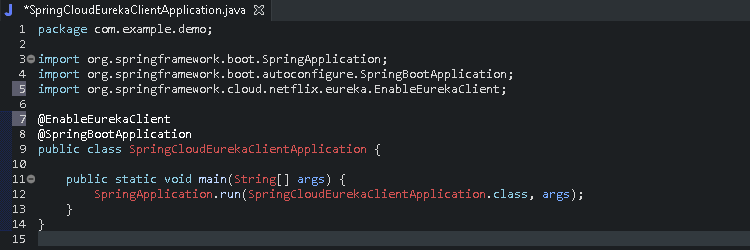


Khi chạy ứng dụng trực tiếp trên Eclipse, Eureka Monitor( Trình giám sát của Eureka) cho phép bạn nhìn thấy danh sách các dịch vụ đang hoạt động và đã đăng ký với Eureka này.

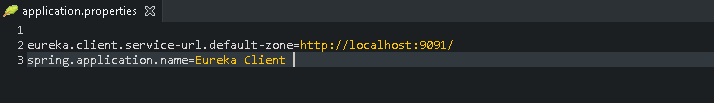
Truy cập vào URL: [**http://localhost:9091/**](http://localhost:9091/)



Để đăng ký dịch vụ từ Eureka-Client lên Eureka-Server. Chúng ta sẽ sử dụng annotation **@EnableEurekaClient**



Khai báo thông tin của Eureka Server trong tập tin cấu hình application.properties



Khai báo property eureka.client.service-url.default-zone với giá trị URL trỏ đến Eureka Server và tên của dịch vụ.

## 2.5 Áp dụng kiến trúc Microservices vào một bài toán thực tế

Với những kiến thức cơ bản đã tìm hiểu. Bây giờ chúng tôi sẽ áp dụng những kiến thức này để xây dựng một ứng dụng dựa trên kiến trúc microsevices. Bài toán đặt ra ở đây là xây dựng một website bán hàng với các yêu cầu:

* Có khả năng mở rộng nhanh chóng mà không ảnh hưởng tới ứng dụng đang chạy.
* Có khả năng chịu tải cao.
* Website vẫn hoạt động bình thường khi có một tính năng bị lỗi.
* Có các chức năng như một website thương mại điện tử:
* Chức năng của admin:
* Thêm, sửa, xóa các mặt hàng, loại mặt hàng.
* Quản lý đơn hàng của khách.
* Quản lý tài khoản khách hàng.
* Chức năng của khách hàng:
* Đăng ký tài khoản.
* Xem, tìm kiếm sản phẩm.
* Đặt hàng.
* Thanh toán online.

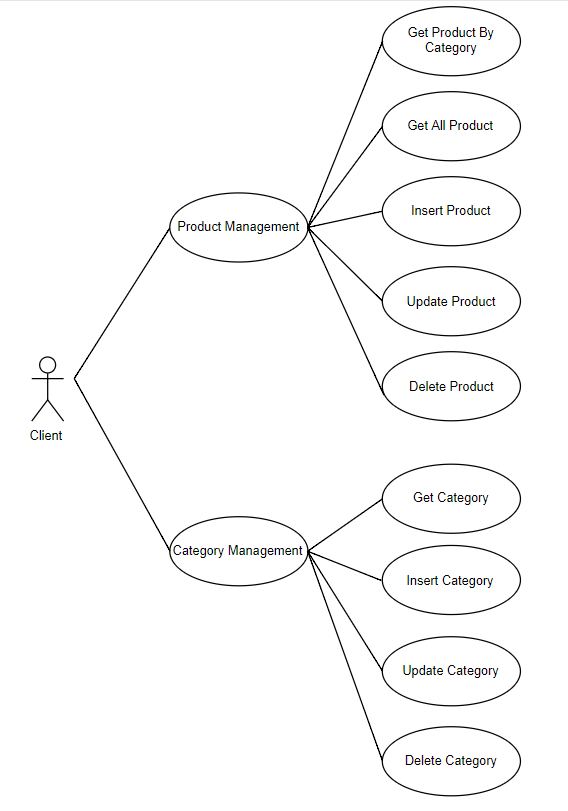
### 2.5.1 Phương án giải quyết

Để triển khai ứng dụng với các yêu cầu mà khách hàng đã đề ra. Nhóm tôi đã phân tích và đưa ra mô hình phù hợp cho bài toán là áp dụng kiến trúc Microservices. Lý do chúng tôi áp dụng mô hình Microservices là:

* Mỗi service đều có thể thay đổi, nâng cấp hoặc thậm chí thay thế mà không ảnh hưởng tới chức năng các service khác.
* Khi có một service bị lỗi, các service vẫn hoạt động bình thường.
* Với microservice thì việc tạo ra nhiều bản sao để tăng tải cho website rất dễ dàng.

### 2.5.2 Use case

#### 2.5.2.1 Product Service

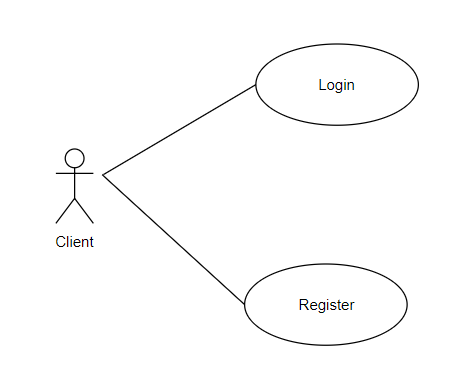


**Hình 2.23** Use Case Product Service

**Bảng 3.1** Mô tả Use Case Product Service

|  |  |
| --- | --- |
| Use Case | Mô tả |
| Get All Product | Lấy tất cả sản phẩm |
| Get Product By Category | Lấy sản phẩm theo loại |
| Insert Product | Thêm sản phẩm. |
| Update Product | Cập nhật sản phẩm. |
| Delete Product | Xóa 1 sản phẩm. |
| Get All Category | Lấy tất cả loại sản phẩm |
| Get Category By Id | Lấy một loại sản phẩm |
| Insert Category | Thêm loại sản phẩm. |
| Update Category | Cập nhật loại sản phẩm |
| Delete Category | Xóa 1 loại sẩn phẩm |

#### 2.5.2.2 Authentication Service

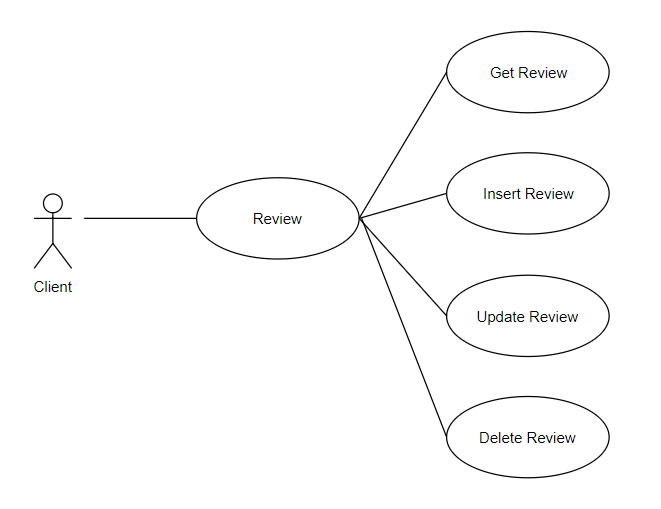


**Hình 2.24** Use Case Authentication Service

**Bảng 3.2** Mô tả Use Case Authentication Service

|  |  |
| --- | --- |
| Use Case | Mô tả |
| Login | Đăng nhập vào hệ thống. |
| Register | Đăng ký tài khoản |

#### 2.5.2.3 Review Service

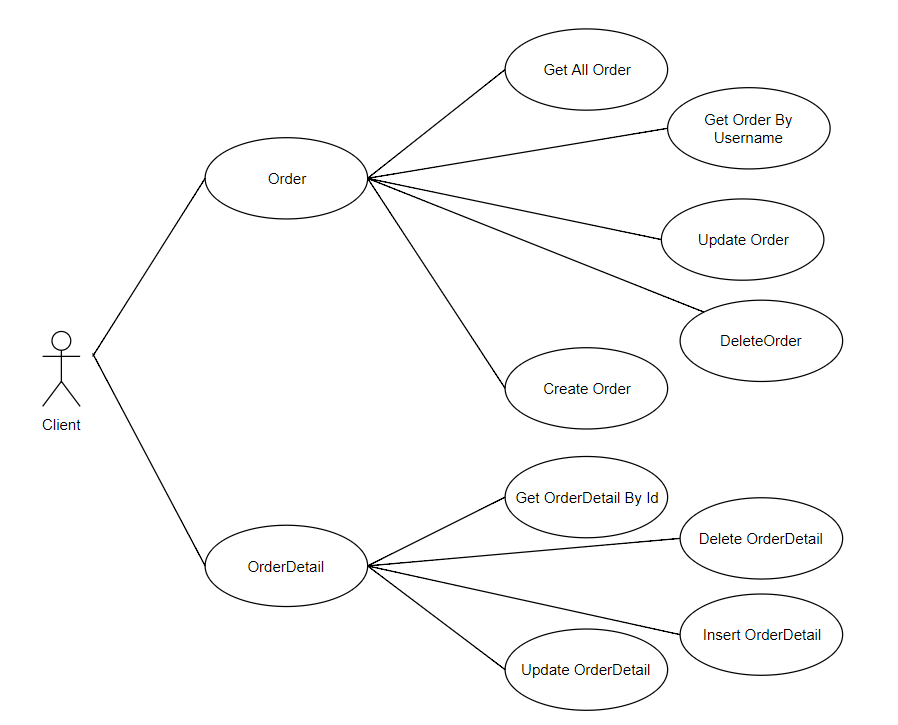


**Hình 2.25** Use Case Review Service

**Bảng 3.3** Mô tả Use Case Review Service

|  |  |
| --- | --- |
| Use Case | Mô tả |
| Get Review | Lấy những comment của sản phẩm từ database. |
| Insert Review | Thêm 1 bình luận. |
| Update Review | Cập nhật 1 bình luận. |
| Delete Review | Xóa 1 bình luận. |

#### 2.5.2.4 Order Service

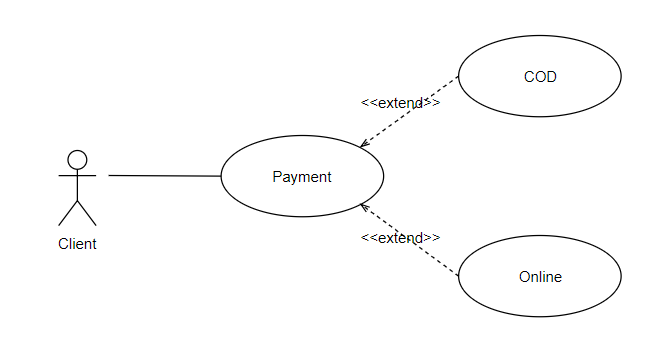


**Hình 2.26** Use Case Order Service

**Bảng 3.4** Mô tả Use Case Order Service

|  |  |
| --- | --- |
| Use Case | Mô tả |
| Get All Order | Lấy tất cả order |
| Get Order By Username | Lấy Order theo username |
| Create Order | Thêm 1 order mới. |
| Update Order | Cập nhật 1 order. |
| Delete Order | Xóa 1 order. |
| Get OrderDetail | Lấy tất cả sản phẩm hiện có của 1 order. |
| Insert OrderDetail | Thêm sản phẩm vào 1 order. |
| Update OrderDetail | Cập nhật sản phẩm của 1 order. |
| Delete OrderDetail | Xóa sản phẩm ra khỏi 1 order. |

#### 2.5.2.5 Payment Service

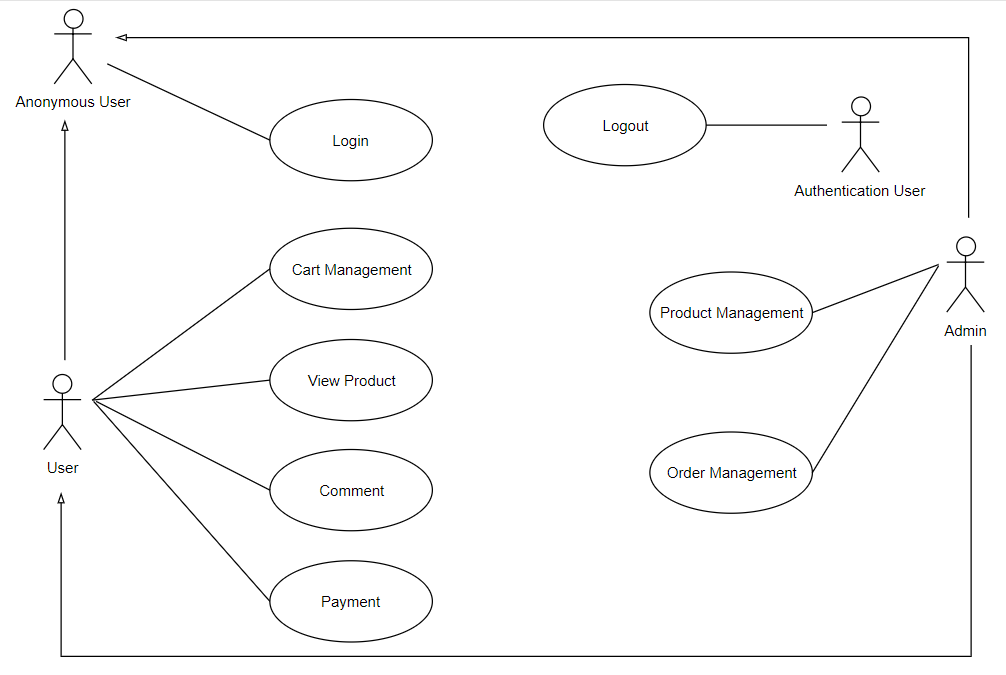


**Hình 2.27** Use Case Payment Service

**Bảng 3.5** Mô tả Use Case Payment Service

|  |  |
| --- | --- |
| Use Case | Mô tả |
| Payment | Thanh toán đơn hàng. |
| COD | Thanh toán khi giao hàng |
| Online | Thanh toán trực tuyến |

#### 2.5.2.6 Application



**Hình 2.28** Use Case Application

**Bảng 3.6** Mô tả Actor

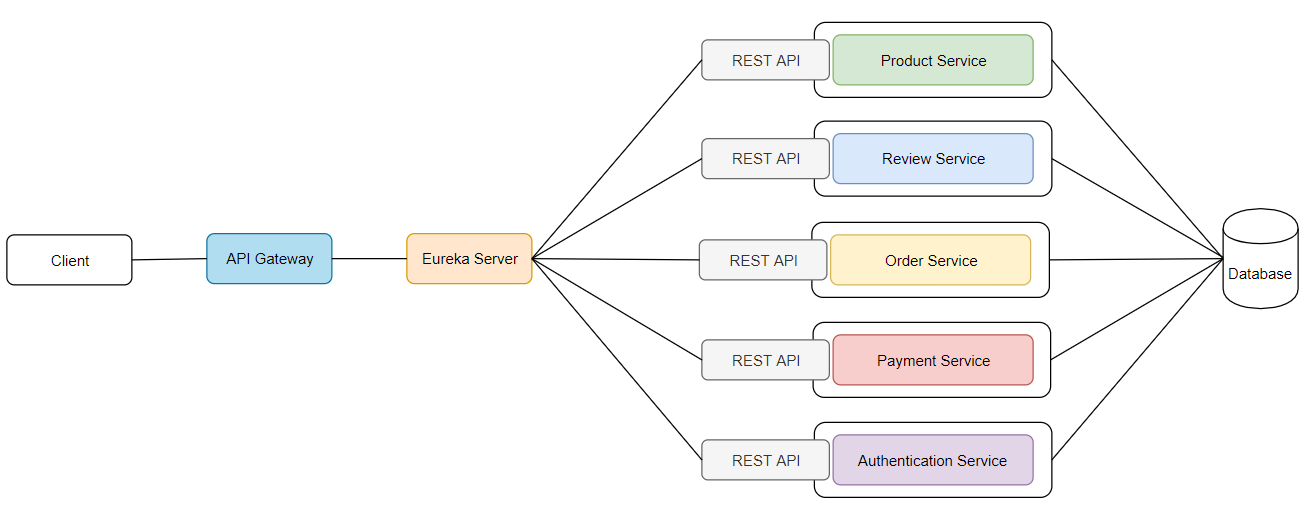
|  |  |
| --- | --- |
| Tên | Mô tả |
| Admin | Quản lý của cửa hàng |
| User | Người dùng |
| Anonymous User | Người dùng chưa đăng nhập |
| Authentication User | Người dùng đã đăng nhập |

**Bảng 3.7** Mô tả Use Case Application

|  |  |
| --- | --- |
| Use Case | Mô tả |
| Login | Cho phép người dùng đăng nhập vào website. |
| Logout | Cho phép người dùng đăng xuất khỏi website |
| Product Management | Cho phép admin có quyền thêm, sửa, xóa các sản phẩm của website. |
| Order Management | Cho phép admin có quyền quản lý các order đã được đặt, lịch sử order. |
| Cart Management | Cho phép user thêm, sửa, xóa sản phẩm khỏi giỏ hàng. |
| View Product | Cho phép user xem các sản phẩm, các loại sản phẩm của website. |
| Comment | Cho phép user xem, thêm, sửa, xóa bình luận. |
| Payment | Cho phép user thanh toán trực tiếp. |

### 2.5.3 Kiến trúc microservice

Sau quá trình tìm hiểu và phân tích thì nhóm tôi đã đưa thiết kế kiến trúc microservice sau cho bài toán.

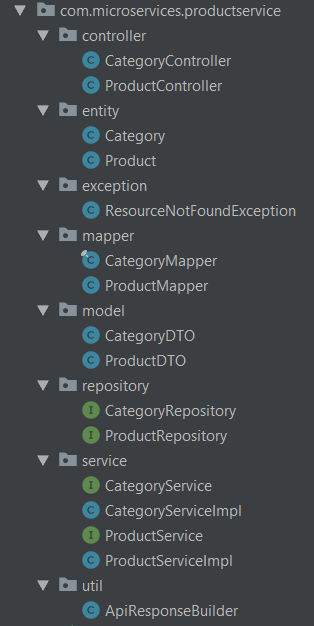


**Hình 2.29** Kiến trúc microservice

### 2.5.4 Product Service

Product Service là service cho phép người dùng lấy, thêm, sửa, xóa các sản phẩm.

#### 2.5.4.1 Cấu trúc code



**Hình 2.30** Cấu trúc code Product Service

Thư mục entity là các thực thể tồn tại trong service.

Thư mục repository thực hiện việc tạo database dựa vào các class entity, và cho phép người dùng khai báo thêm các query để truy vấn database thông qua các entity.

Thư mục model có nhiệm vụ chuyển đổi data lấy từ database chuyển thành đối tượng tùy mục đích sử dụng. DTO có nghĩa là Data Transfer Object.

Thư mục mapper sẽ map entity thành DTO hoặc ngược lại.

Thư mục exception chứa class dùng để trả về khi gặp lỗi NotFound.

Thư mục service chứa các hàm dùng để thực thi các chức năng của service.

Thư mục controller sẽ đưa ra các endpoint để người dùng có thể gọi.

#### 2.5.4.2 Danh sách endpoint

**Bảng 3.8** Endpoint GET Product Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| GET /product | Lấy danh sách tất cả các product từ database. |
| GET /product/X | Lấy product có id “X”. |
| GET /product/category/X | Lấy danh sách các sản phẩm thuộc category có id “X”. |
| GET /category | Lấy danh sách tất cả các category từ database. |
| GET /category/X | Lấy category có id “X”. |

**Bảng 3.9** Endpoint POST Product Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| POST /product | Thêm mới 1 product vào database. |
| POST /category | Thêm mới 1 category vào database. |

**Bảng 3.10** Endpoint PUT Product Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| PUT /product/X | Chỉnh sửa thông tin 1 product có id “X”. |
| PUT /category/X | Chỉnh sửa thông tin 1 category có id “X”. |

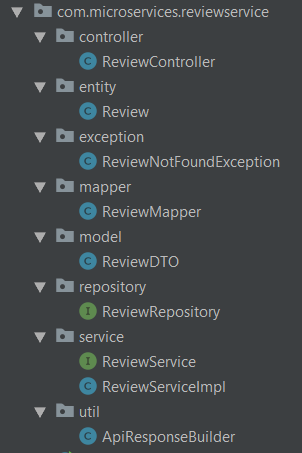
**Bảng 3.11** Endpoint DELETE Product Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| DELETE /product/X | Xóa 1 product có id “X” khỏi database. |
| DELETE /category/X | Xóa 1 category có id “X” khỏi database. |

### 2.5.5 Review Service

Review Service là service cho phép người dùng lấy, thêm, sửa, xóa bình luận các sản phẩm.

#### 2.5.5.1 Cấu trúc code



**Hình 2.31** Cấu trúc code Review Service

Thư mục entity là các thực thể tồn tại trong service.

Thư mục repository thực hiện việc tạo database dựa vào các class entity, và cho phép người dùng khai báo thêm các query để truy vấn database thông qua các entity.

Thư mục model có nhiệm vụ chuyển đổi data lấy từ database chuyển thành đối tượng tùy mục đích sử dụng. DTO có nghĩa là Data Transfer Object.

Thư mục mapper sẽ map entity thành DTO hoặc ngược lại.

Thư mục exception chứa class dùng để trả về khi gặp lỗi NotFound.

Thư mục service chứa các hàm dùng để thực thi các chức năng của service.

Thư mục controller sẽ đưa ra các endpoint để người dùng có thể gọi.

#### 2.5.5.2 Danh sách endpoint

**Bảng 3.12** Endpoint GET Review Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| GET /product/X | Lấy danh sách tất cả các bình luận từ database. |

**Bảng 3.13** Endpoint POST Review Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| POST / | Thêm một bình luận vào databsse. |

**Bảng 3.14** Endpoint PUT Review Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| POST /X | Chỉnh sửa một bình luận có id “X”. |

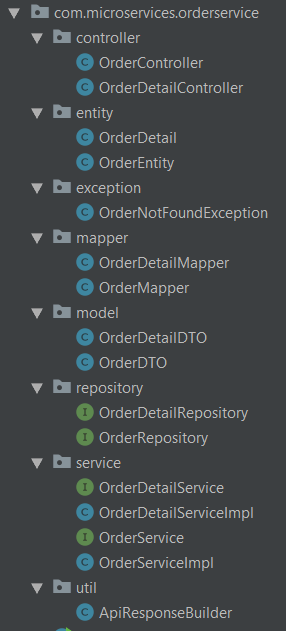
**Bảng 3.15** Endpoint DELETE Review Service

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| POST /X | Xóa một bình luận có id “X” khỏi database. |

### 2.5.6 Order Service

Order Service là service cho phép người dùng thêm sản phẩm vào giỏ hàng, tạo order.

#### 2.5.6.1 Cấu trúc code



**Hình 2.32** Cấu trúc code Order Service

Thư mục entity là các thực thể tồn tại trong service.

Thư mục repository thực hiện việc tạo database dựa vào các class entity, và cho phép người dùng khai báo thêm các query để truy vấn database thông qua các entity.

Thư mục model có nhiệm vụ chuyển đổi data lấy từ database chuyển thành đối tượng tùy mục đích sử dụng. DTO có nghĩa là Data Transfer Object.

Thư mục mapper sẽ map entity thành DTO hoặc ngược lại.

Thư mục exception chứa class dùng để trả về khi gặp lỗi NotFound.

Thư mục service chứa các hàm dùng để thực thi các chức năng của service.

Thư mục controller sẽ đưa ra các endpoint để người dùng có thể gọi.

#### 2.5.6.2 Danh sách endpoint

* GET

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| GET /order | Lấy danh sách tất cả các order database. |
| GET /order/username/X | Lấy order của username “X” |
| GET /order/active/X | Lấy danh sách các order có trạng thái “X”. Nếu X là true thì trả về danh sách các order đã được đặt hàng và ngược lại. |
| GET /orderdetail/X | Lấy danh sách các sản phẩm của order có id “X”. |

* POST

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| POST /order | Thêm một order vào database. |
| POST /orderdetail | Thêm một sản phẩm vào order. |

* PUT

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| PUT /order/X | Chỉnh sửa order có id “X” |
| PUT /orderdetail/X | Chinh sửa lại thông tin của 1 orderdetail của order có id “X”. |

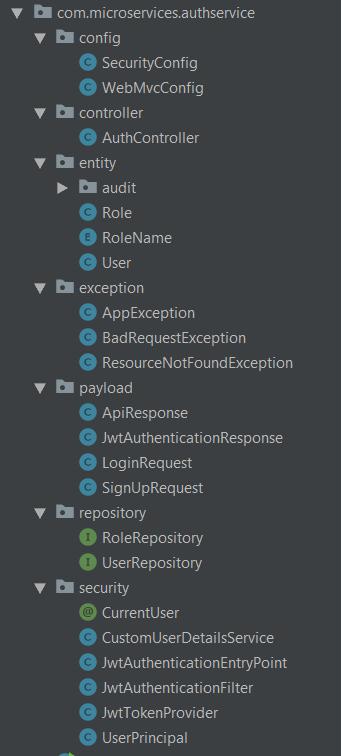
* DELETE

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| DELETE /order/X | Xóa một order có id “X” khỏi database. |
| DELETE /orderdetail/X | Xóa một orderdetail có id “X” khỏi database. |

### 2.5.7 Authentication Service

Authentication Service là service cho phép người dùng đăng kí tài khoản, đăng nhập, cấp quyền.

#### 2.5.7.1 Cấu trúc code



**Hình 2.33** Cấu trúc code của Authentication Service

#### 2.5.7.2 Danh sách endpoint

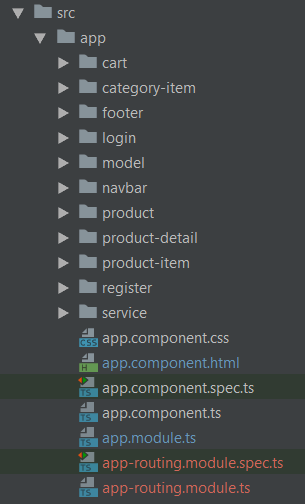
* POST

|  |  |
| --- | --- |
| Endpoint | Mục đích |
| POST /api/auth/signin | Đăng nhập và trả về token có phân quyền. |
| POST / api/auth/signup | Đăng ký tài khoản. |

### 2.5.8 Application

Đây chính là UI sử dụng Angular 6 để người dùng có thể sử dụng.

#### 2.5.8.1 Cấu trúc code



**Hình 2.34** Cấu trúc code Application

Thư mục cart (component cart) cho phép người dùng xem thông tin giỏ hàng.

Thư mục category-item (component category-item) cho phép người dùng xem các sản phẩm của một category.

Thư mục footer (component footer) là phần cuối cùng của website.

Thư mục login (component login) là trang login của website.

Thư mục model là các đối tượng được sử dụng.

Thư mục navbar (component navbar) là phần trên cùng của website.

Thư mục product (component product) là trang chủ. Component này bao gồm 3 component nhỏ: navbar, product-item, footer.

Thư mục product-detail (component product-detail) là trang thông tin chi tiết một sản phẩm.

Thư mục product-item (component product-item) là phần show các sản phẩm ở trang chủ.

Thư mục register (component register) là trang register của website.

Thư mục service dùng để gọi các api từ backend.

### 2.5.9 Triển khai

Việc triển khai có một vai trò trọng yếu và gồm các yêu cầu sau:

* Khả năng triển khai/ gỡ xuống độc lập mà không ảnh hưởng đến dịch vụ khác
* Có thể mở rộng theo cấp microservices, chỉ mở rộng microservices cần thiết.
* Phát triển và triển khai microservices nhanh chóng.
* Một microservice ngắt kết nối hay sập thì không ảnh hưởng các microservices khác.

Docker cung cấp một công cụ tuyệt vời để triển khai microservices đáp ứng đủ các yêu cầu trên. Để có thể deploy các microservice lên docker:

**Bước 1:** Build file jar cho từng microservice.

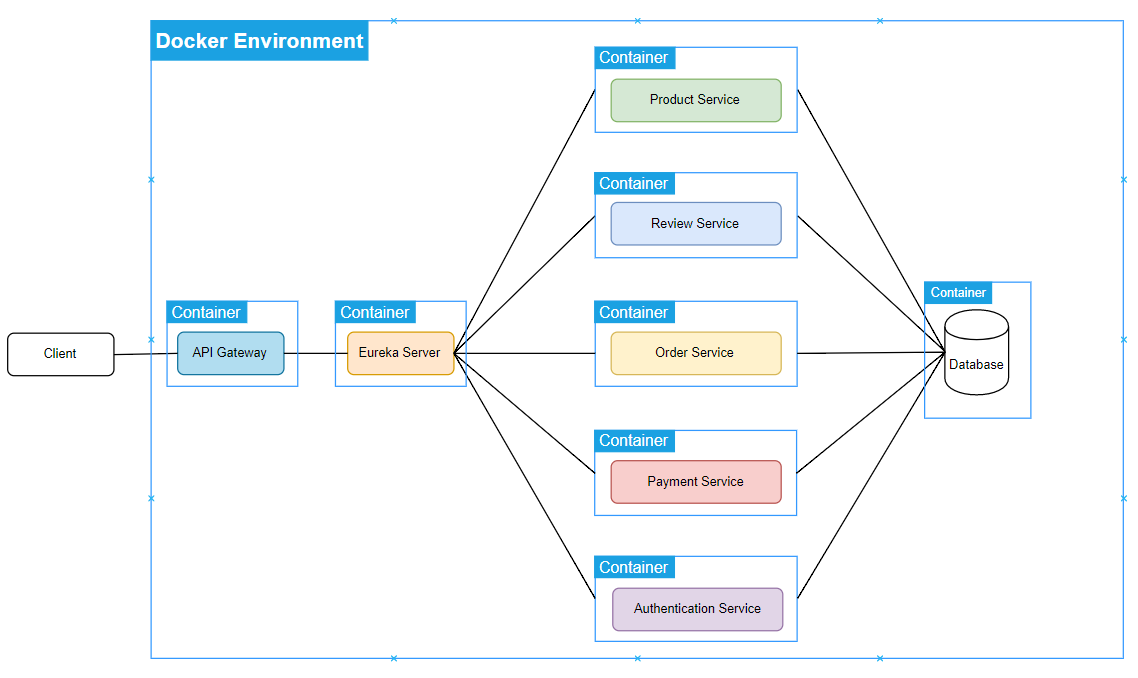
**Bước 2:** Tạo Dockerfile cho mỗi microservice. Dockerfile có nhiệm vụ đóng gói mỗi microservice thành một Docker Image.

**Bước 3**: Tạo docker-compose.

Chúng ta không thể chạy từng file Dockerfile để đóng gói mỗi microservice thành image, tiếp theo start từng image. Việc đấy sẽ tốn rất nhiều thời gian. Vì vậy chúng ta sẽ sử dụng docker-compose.

Compose là công cụ giúp định nghĩa và khởi chạy multi-container Docker applications. Trong Compose, chúng ta sử dụng Compose file để cấu hình application’s services. Chỉ với một câu lệnh là có thể dễ dàng create và start toàn bộ các services phục vụ cho việc chạy ứng dụng.

Hệ thống sau khi triển khai:



**Hình 2.35** Microservice triển khai trên docker

# CHƯƠNG 3: KẾT LUẬN

Thông qua đề tài nhóm tôi muốn tập trung nghiên cứu, xây dựng và triển khai một hệ thống quản lý phân tán, phân chia hệ thống thành nhiều các service theo hướng microservices đảm bảo cho khả năng phát triển mở rộng lâu dài. Một số điểm chính đề tài đã đạt được cụ thể như sau:

* Tìm hiểu về microservice.
* Tìm hiểu về các công nghệ giúp trong việc triển khai kiến trúc microservice: Docker, Eureka, Gateway,…
* Thiết kế và triển khai ứng dụng theo hướng microservice, sử dụng framework Spring Boot của Java.
* Xây dựng UI với công nghệ Angular 6.

Về phần triển khai, nhóm đã thiết kế được mô hình microservices. Ứng dụng những công nghệ như docker để triển khai các microservice.

# TÀI LIỆU THAM KHẢO

1. Pattern: Microservice Architecture, <https://microservices.io/patterns/microservices.html>

2. Giới thiệu về Microservices (kiến trúc nhiều dịch vụ nhỏ), <https://techmaster.vn/posts/33594/gioi-thieu-ve-microservices>

3. Microservices Thực Tiễn: Từ Thiết Kế Đến Triển Khai, <https://techmaster.vn/posts/34410/microservices-thuc-tien-tu-thiet-ke-den-trien-khai>

4. Microservices with Spring Boot — Intro to Microservices (Part 1), <https://medium.com/omarelgabrys-blog/microservices-with-spring-boot-intro-to-microservices-part-1-c0d24cd422c3>

5. Microservices with Spring Boot — Creating our Microserivces & Gateway (Part 2), <https://medium.com/omarelgabrys-blog/microservices-with-spring-boot-creating-our-microserivces-gateway-part-2-31f8aa6b215b>

6. Docker là gì và làm gì?, <https://viblo.asia/p/docker-la-gi-va-lam-gi-gGJ592RGKX2>

7. Microservices Communication: Zuul API Gateway, <https://dzone.com/articles/microservices-communication-zuul-api-gateway-1>

8. <https://docs.docker.com/>