**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 HIỆU 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 15110

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

**Mục lục**

[1. Microservices 3](#_Toc532068824)

[1.1 Định nghĩa? 3](#_Toc532068825)

[1.2 Scale Cube 4](#_Toc532068826)

[1.3 Đặc điểm 7](#_Toc532068827)

[1.3.1 Decentralized (phân tán) 7](#_Toc532068828)

[1.3.2 Independent 7](#_Toc532068829)

[1.3.3 Do one thing well 7](#_Toc532068830)

[1.3.4 Polygot 8](#_Toc532068831)

[1.3.5 Black box 8](#_Toc532068832)

[1.3.6 You build it, you run it 8](#_Toc532068833)

[1.4 Ưu điểm 8](#_Toc532068834)

[1.5 Nhược điểm 9](#_Toc532068835)

[2. Docker 10](#_Toc532068836)

[2.1 Định nghĩa 10](#_Toc532068837)

[2.2 Các thành phần cơ bản 11](#_Toc532068838)

[2.3 Một số khái niệm 12](#_Toc532068839)

[2.4 So sánh Docker với Virtual Machine 12](#_Toc532068840)

[2.5 Các thành phần và công nghệ ảo hóa ứng dụng trong Docker 15](#_Toc532068841)

[2.5.1 Các thành phần 15](#_Toc532068842)

[2.5.2 Kiến trúc của Docker 16](#_Toc532068843)

[2.5.3 Ưu điểm hình thức đóng gói thành Container 17](#_Toc532068844)

[2.5.4 Quy trình thực thi của một hệ thống sử dụng Docker. 18](#_Toc532068845)

[2.5.5 Các lệnh cơ bản 19](#_Toc532068846)

[2.6 Cách ảo hóa với Docker 21](#_Toc532068847)

[3. Gateway 27](#_Toc532068848)

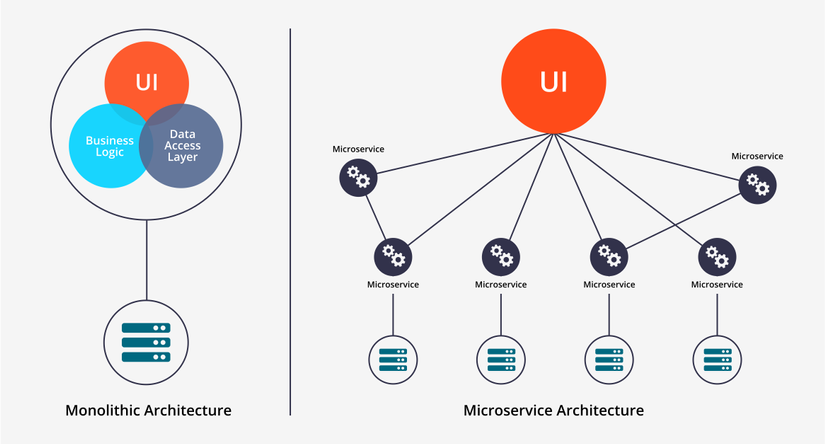
# 1. Microservices

## 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.

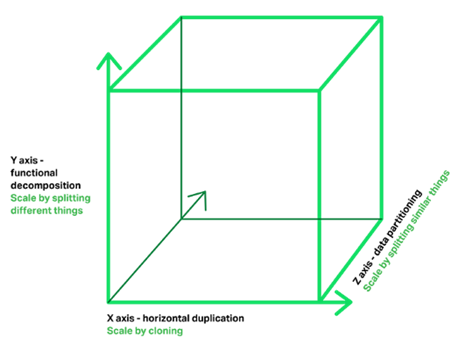
Chúng giao tiếp với nhau qua rất nhiều phương thức có thể là Rest API hoặc là GRPC hoặc Lambda hoặc bất cứ phương thức nào có thể giao tiếp được.



Hình 1.1 Microservices và Monolithic Architecture

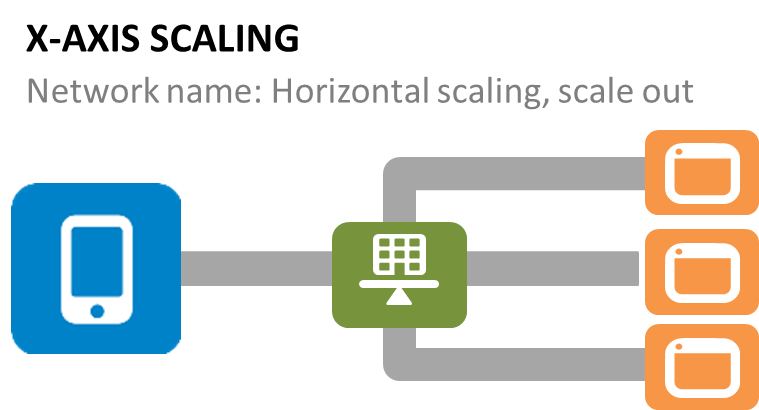
## 1.2 Scale Cube

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

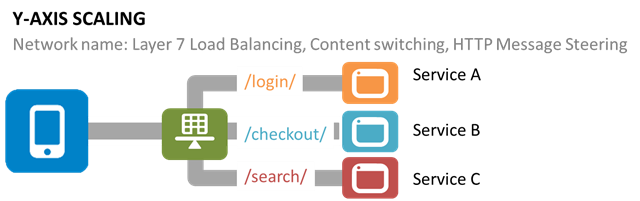


Hình 1.2 The Scale Cube

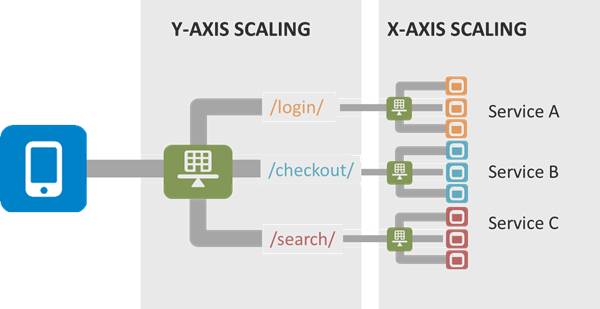
Kiến trúc microservice tương đương trục Y của 3 chiều mở rộng chịu tải (Scale Cube).



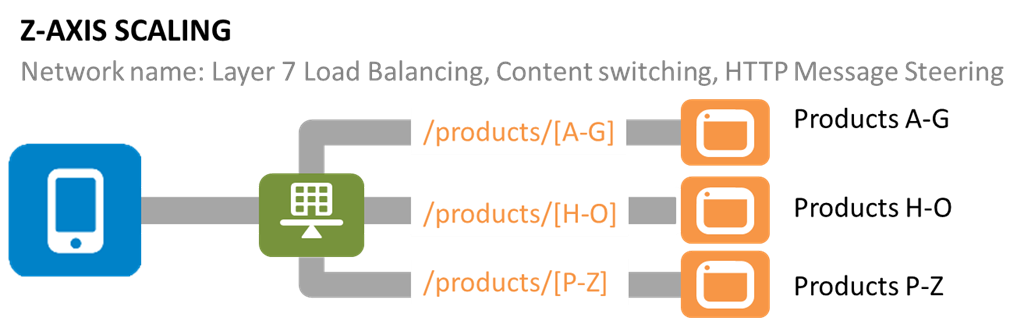
Hình 1.3 Trục X: Mở rộng theo việc clone các service đằng sau bộ cân bằng tải



Hình 1.4 Trục Y: Mở rộng theo chức năng

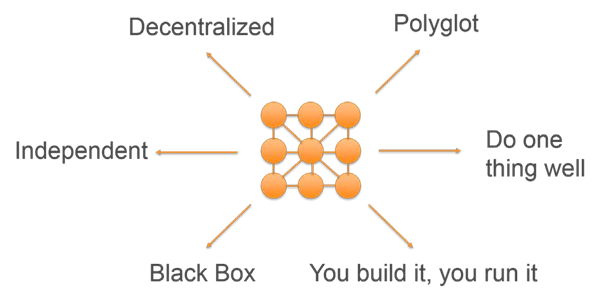


Hình 1.5 Mở rộng theo trục X và Y



Hình 1.6 Trục Z: Mở rộng theo dữ liệu

## 1.3 Đặc điểm



Hình 1.7 Đặc tính của microservices

### 1.3.1 Decentralized

Microservices là một hệ thống phân tán về:

Database – Data model

Develop

Deploy

Manage

Operate

### 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.

### 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).

### 1.3.4 Polygot

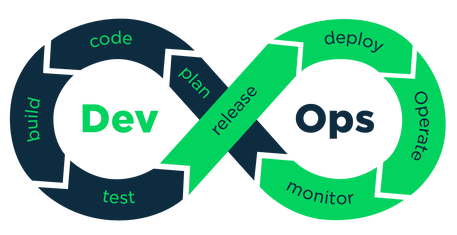
Microservices cho phép các team phát triển tự do chọn tool, ngôn ngữ, data stored.

### 1.3.5 Black box

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

### 1.3.6 You build it, you run it

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



Hình 1.8 DevOps

## 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.

## 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.

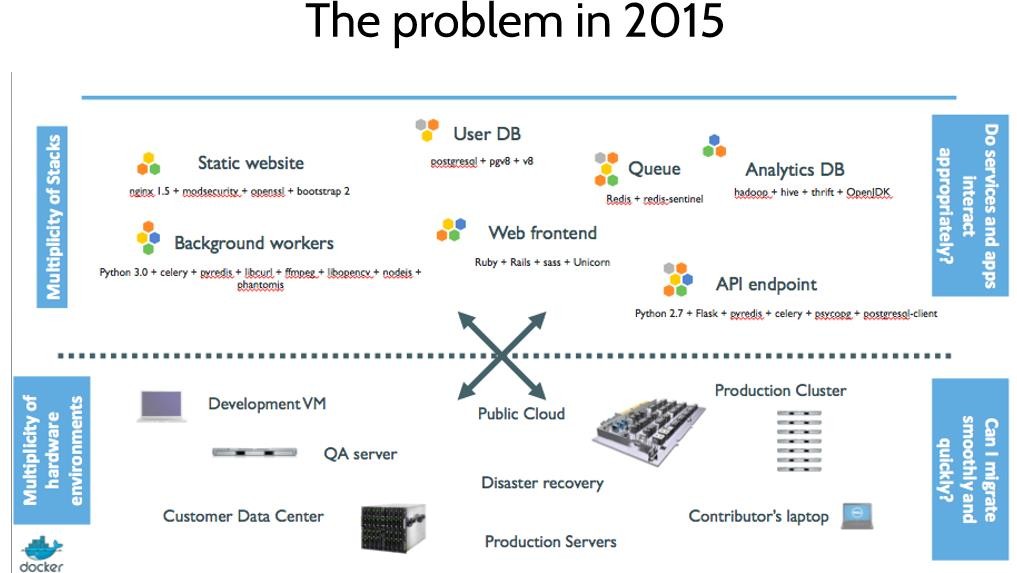
# 2. Docker

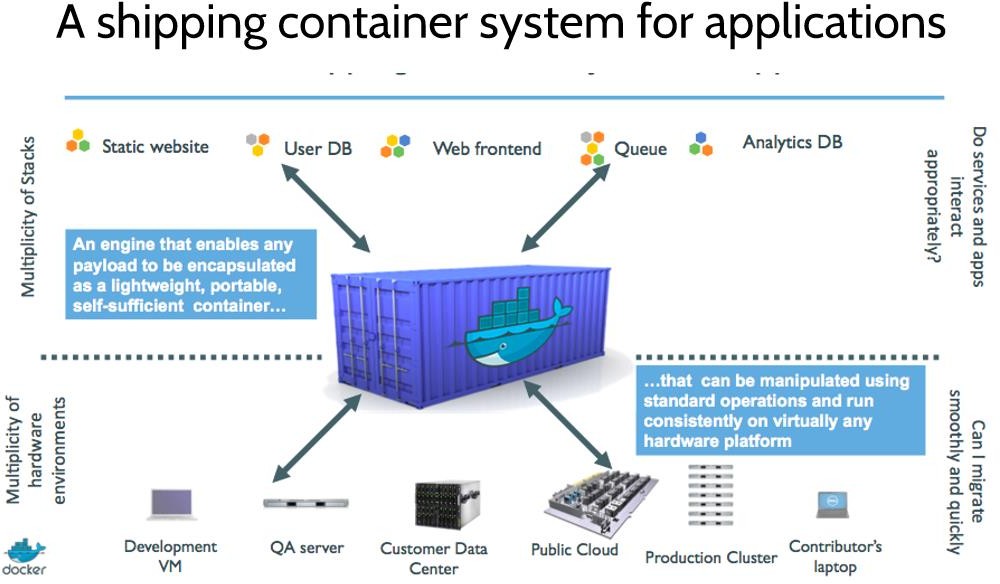
## 2.1 Định nghĩa

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 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.

## 2.3 Một số khái niệm

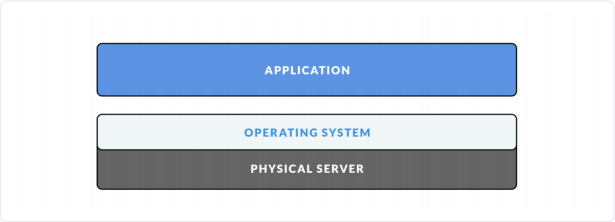
Dockerfile : là một file chứa tập hợp các lệnh để Docker có thể đọc và thực hiện để đóng gói một image theo yêu cầu người dùng.

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.

## 2.4 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.

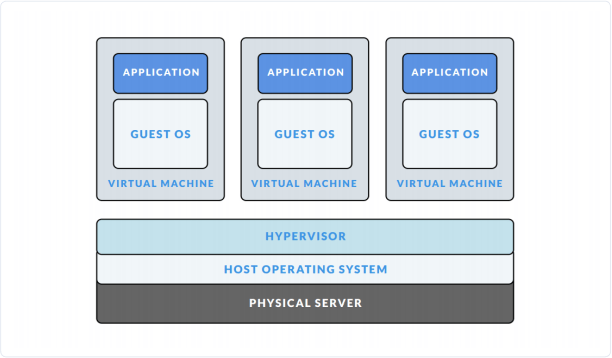


Hình 2.1 Mô hình máy chủ truyền thống

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.

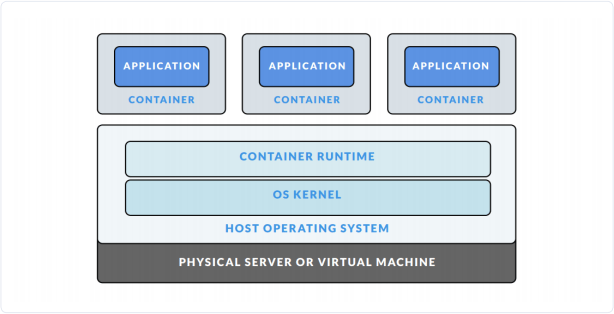


Hình 2.2 Mô hình máy ảo VMs

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.

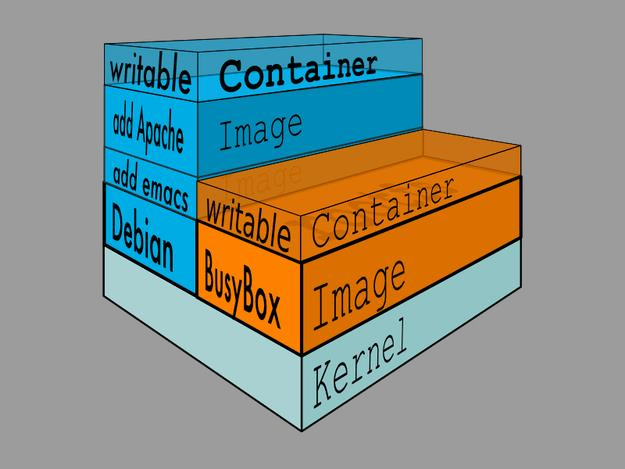
Bước tiến hóa tiếp theo, người ta phát minh ra containerlization.



Hình 2.3 Mô hình ảo hóa Container

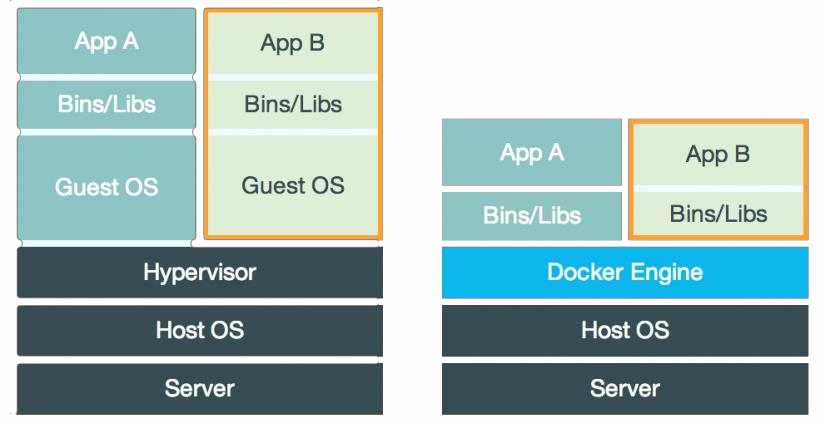
Với công nghệ này, trên một máy chủ, ta sẽ sinh ra được nhiều máy con (giống với ảo hóa), nhưng điều đặc biệt là các máy con (Guess OS) này đều dùng chung phần nhân của máy mẹ (host OS) và chia sẻ với nhau tài nguyên của máy mẹ (RAM chẳng hạn). Như vậy việc tận dụng tài nguyên sẽ được tối ưu hơn.

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



Hình 2.4 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 2.5 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.

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

### 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.

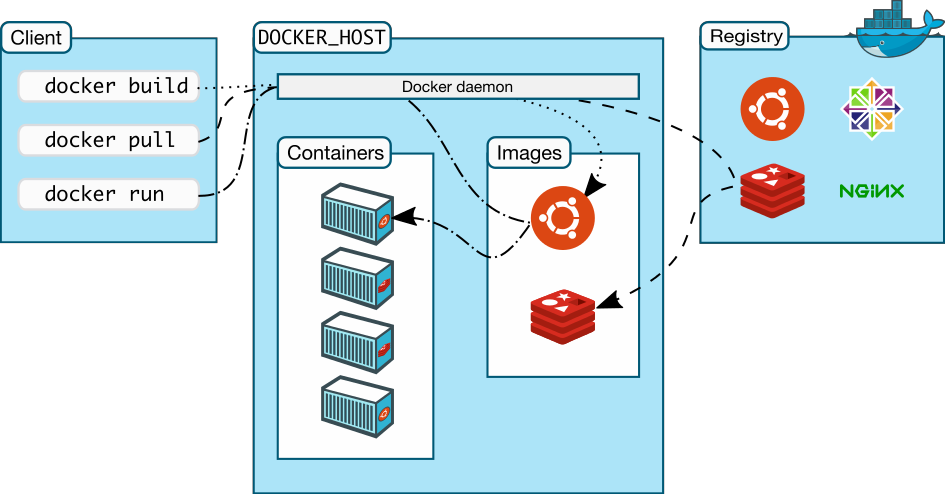
* + Là một máy ảo, được cấu thành từ 1 image và được đắp thêm 1 lớp “trang trí” writable-file-layer. Các thay đổi trong máy ảo này (cài thêm phần mềm, tạo thêm file…) sẽ được lưu ở lớp trang trí này;
  + Các container này sẽ dùng chung tài nguyên của hệ thống (RAM, Disk, Network…), chính nhờ vậy, những 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.

### 2.5.2 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 2.6 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.

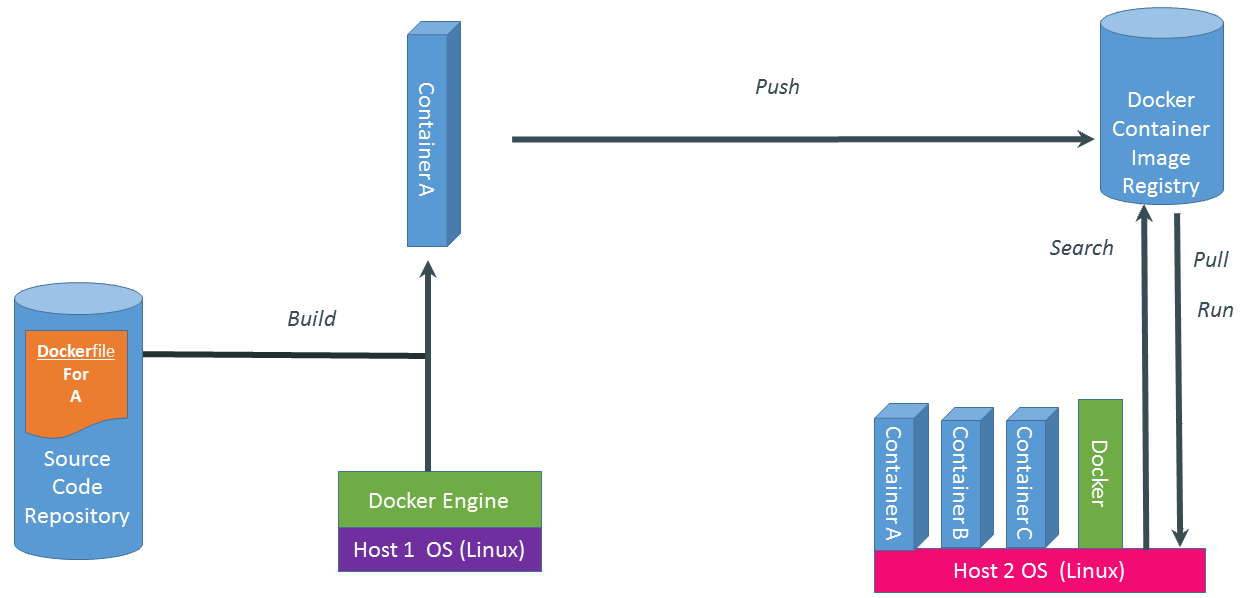
*Docker Container:* Một Docker container giữ mọi thứ chúng ta cần để chạy một app. Mỗi container được tạo từ Docker image. Docker container có thể có các trạng thái run, started, stopped, moved và deleted.

### 2.5.3 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.

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



Hình 2.7 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:*

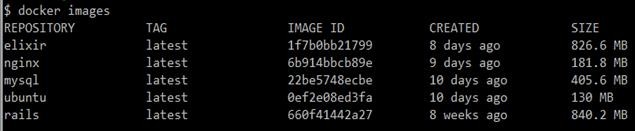
Giả sử một máy tính muốn sử dụng Container chúng ta đã push lên đám mây (máy đã cài Docker Engine) thì bắt buộc máy phải thực hiện việc Pull container này về máy. Sau đó thực hiện Run Container này.

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

### 2.5.5 Các lệnh cơ bản

#### 2.5.5.1 Hiển thị danh sách các images

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



#### 2.5.5.2 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.

#### 2.5.5.3 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.

#### 2.5.5.4 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.

#### 2.5.5.5 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.

#### 2.5.5.6 Khởi động lại container đã dừng

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

#### 2.5.5.7 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.

#### 2.5.5.8 Truy cập vào 1 container đang chạy

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

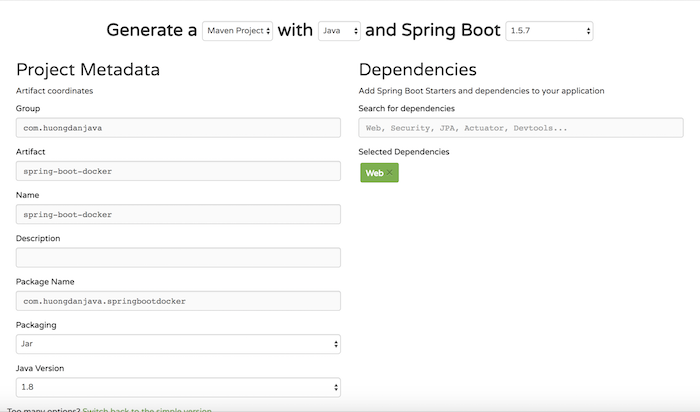
Ngoài ra, còn rất nhiều câu lệnh khác. Sử dụng lệnh *docker -h* để biết thêm chi tiết.

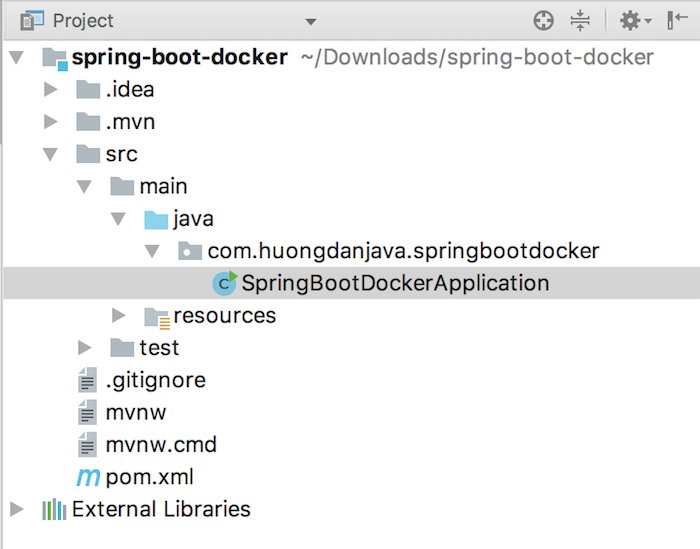
## 2.6 Cách ảo hóa với Docker

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 2.8 Tạo Spring Boot project



Hình 2. 9 Cây thư mục Spring Boot project

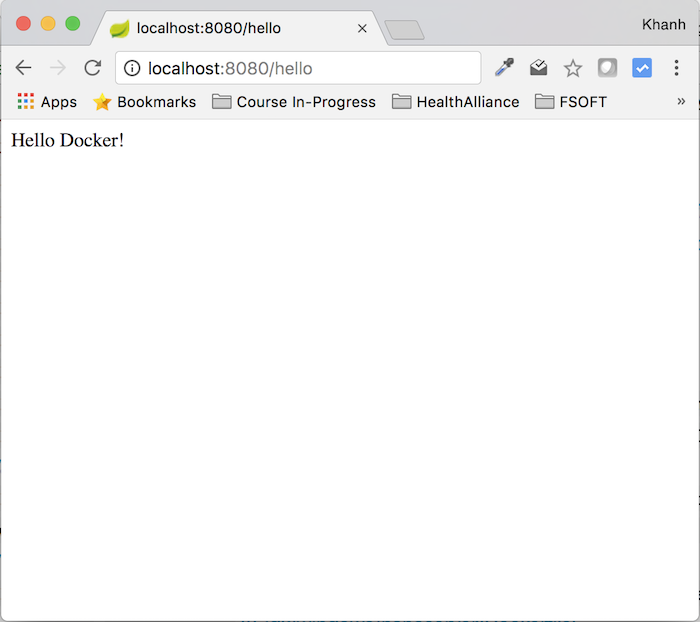
Mở tập tin SpringBootDockerApplication:

|  |
| --- |
| 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);  }  } |

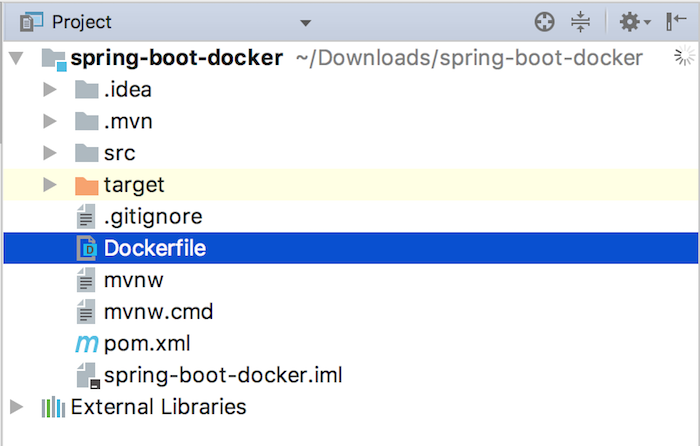
Kết quả:



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 2.10 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 của mình 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.

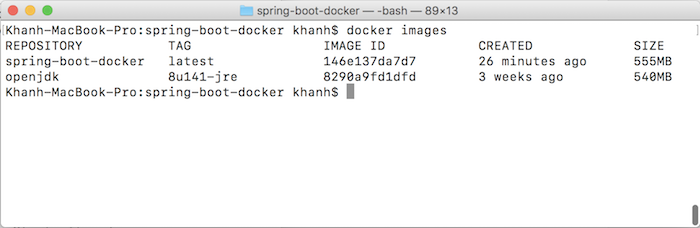
Mở Terminal lên và đi đến thư mục của project, sau đó hãy nhập dòng lệnh sau:

|  |
| --- |
| 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:



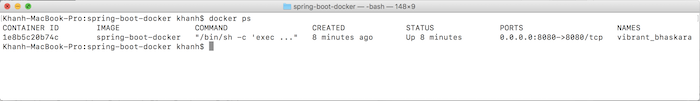
Hình 2.11 Tạo thành công image cho project

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 |

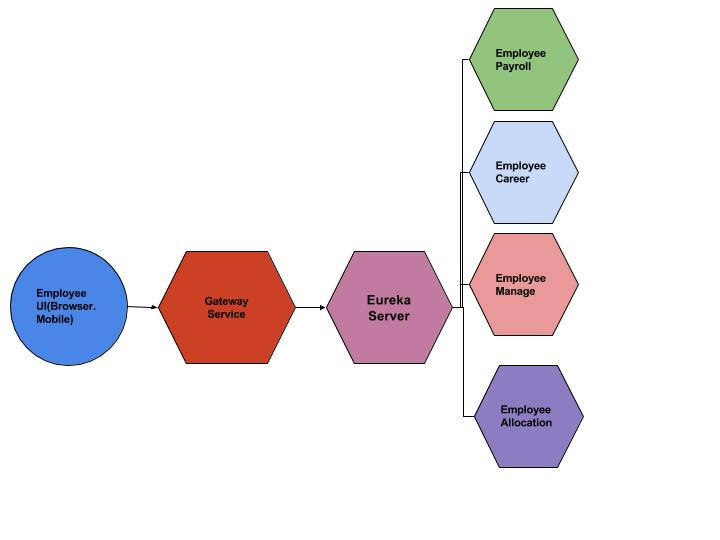
Ở đây, có 2 số 8080 thì số phía trước là port của máy, số phía sau là port của docker. Có thể kiểm tra kết quả bằng cách vào <http://localhost:8080/hello> .



Hình 2.12 Khởi chạy image

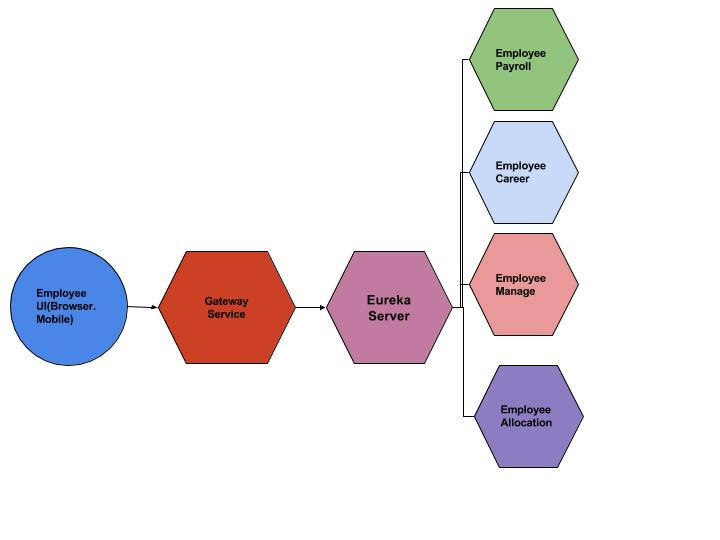
# 3. Gateway

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 3. 1 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 3. 2 Microservice sử dụng API Gateway

# 4. Eureka

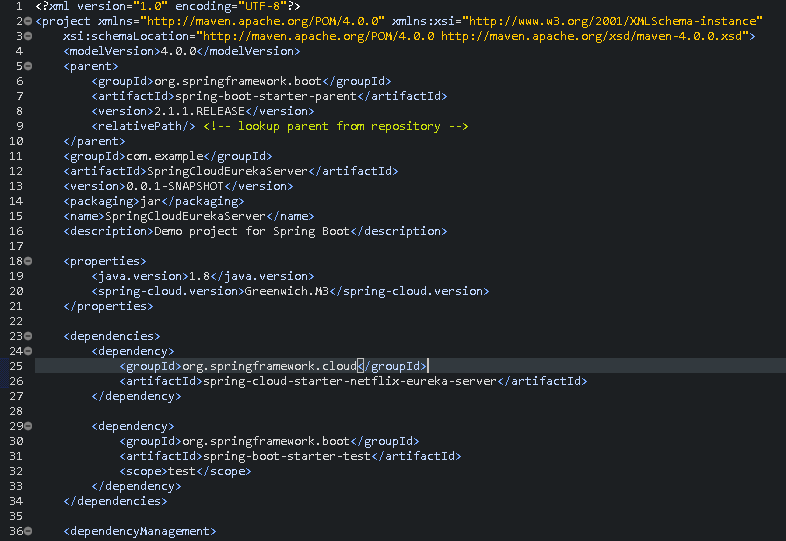
## 4.1 Định nghĩa

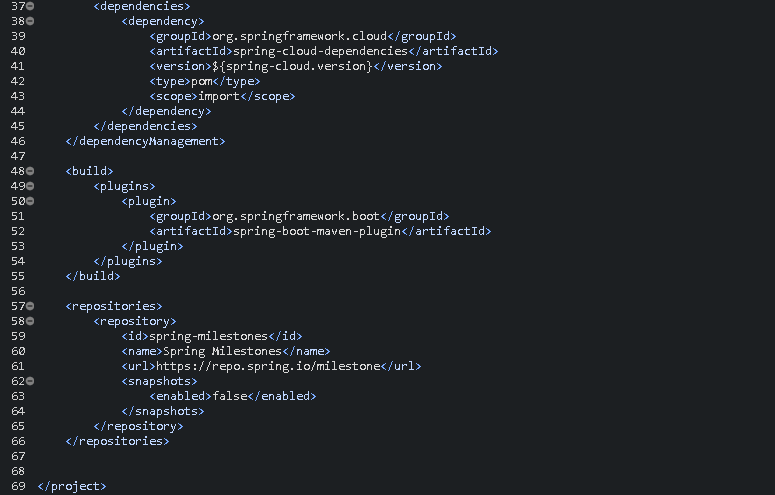
Eureka Server chịu trách nhiệm lưu trữ và cung cấp thông tin của các service trong một hệ thống Microservice. Khi một service đăng ký thông tin với server, nó sẽ cung cấp các thông tin như host, port, trạng thái của nó. Và nó cũng thường xuyên gửi heartbeat message để thông báo tình trạng của mình cho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

Ví dụ: Trong thực tế khi số người sử dụng ứng dụng **chat** tăng lên, số lượng Chat Server cũng tang lên, và chúng có cách để chia sẻ với nhau qua trạng thái người dùng.

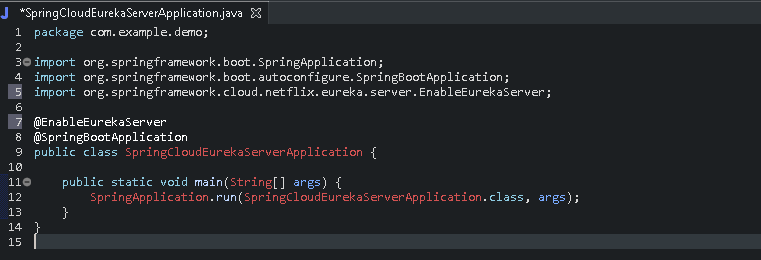
## 4.2 Cách sử dụng Eureka

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

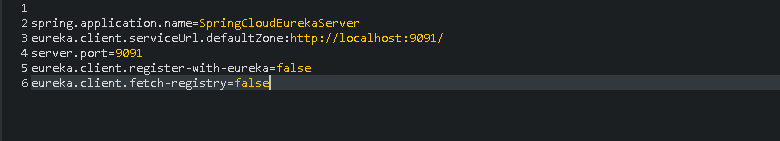




Để ứng dụng trở thành một **Service Registration**( Đăng ký dịch vụ) bạn cần sử dụng **@EnableEurekaServer**

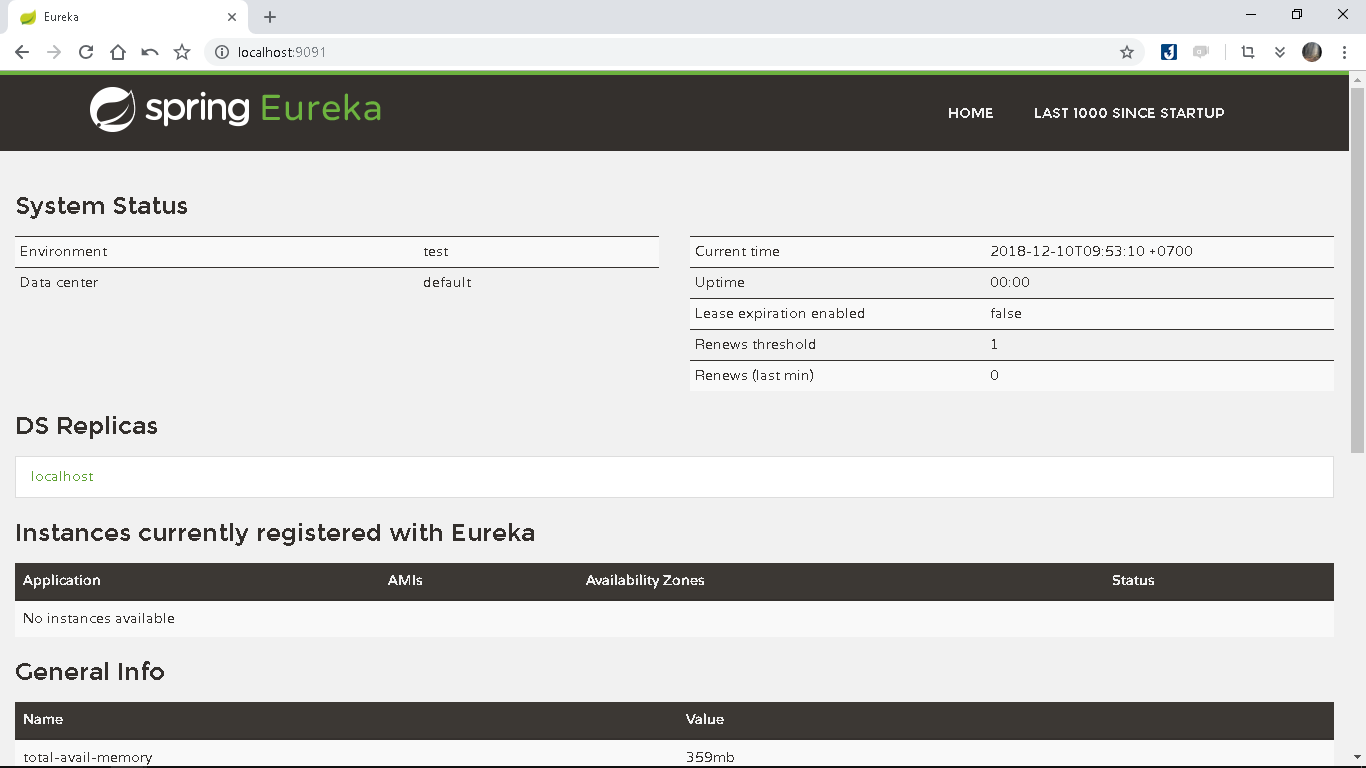


Tạo file bootstrap.properties

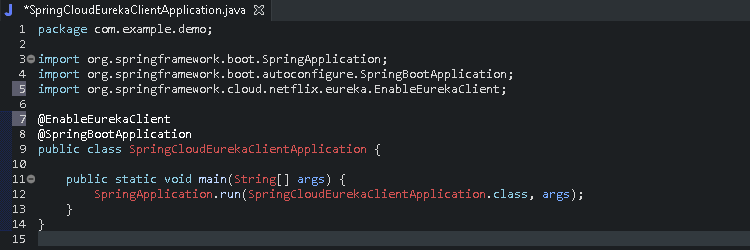


Khi chạy ứng dụng trực tiếp trên Eclipse, Eureka Monitor( Trình giám sát của Eureka) cho phép bạn nhìn thấy danh sách các dịch vụ đang hoạt động và đã đăng ký với Eureka này.

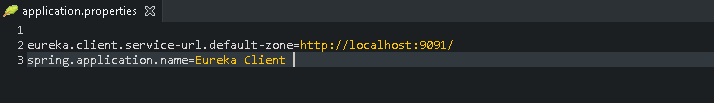
Truy cập vào URL: [**http://localhost:9091/**](http://localhost:9091/)



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



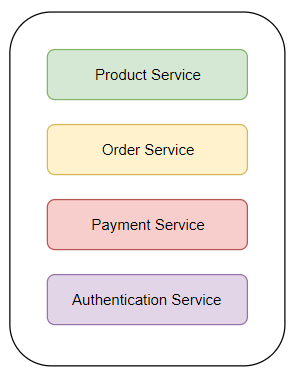
Khai báo property eureka.client.service-url.default-zone với giá trị URL trỏ đến Eureka Server và tên của dịch vụ.

# 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 dựa trên kiến trúc microservices.

## 5.1 Giới thiệu.

Bắt đầu thiết kế với kiến trúc monolithic (nguyên khối) để ứng dụng cho bài toán này thì thiết kế như sau:



Hình 5.1 Kiến trúc Monolithic

Với mô hình trên cỏ thể thấy tất cả service của một ứng dụng. Những service này được triển khai cùng một lúc vào ứng dụng. Dù bên trong có gồm các services thì đây vẫn là một ứng dụng có kiến trúc monolithic. Nó sẽ có một số tính chất sau:

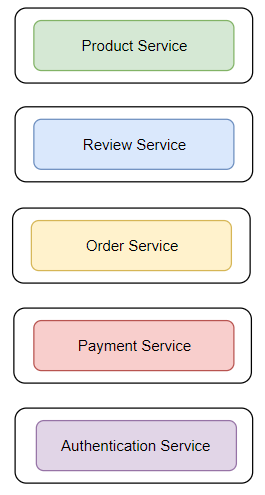
* Ứng dụng monolithic phức tạp và to gây khó khăn cho việc bảo trì, nâng cấp và thêm tính năng mới.
* Phải triển khai lại toàn bộ hệ thống dù chỉ cập nhật hay nâng cấp một phần.
* Độ tin cậy: một service không ổn định có thể sập cả hệ thống.
* Khó đổi mới: ứng dụng monolithic phải sử dụng chung công nghệ nên khó thay đổi hay áp dụng công nghệ mới.

Với những tính chất bị giới hạn trên của kiến trúc nguyên khối nên chúng tôi sẽ phát triển ứng dụng theo kiến trúc microservices.

## 5.2 Kiến trúc Microservices

Nền tảng kiến trúc microservices là xây dựng một ứng dụng mà ứng dụng này là tổng hợp của nhiều services nhỏ và độc lập có thể chạy riêng biệt, phát triển và triển khai độc lập.

Ý tưởng là khi nhìn vào kiến trúc monolithic, ta có thể nhận biết, xác định các yêu cầu và khả năng cần thiết để đáp ứng một nghiệp vụ. Sau đó từng năng lực nghiệp vụ này sẽ được xây dựng thành những service nhỏ, độc lập. Những services này có thể sử dụng các nền tảng công nghệ khác nhau và phục vụ một mục đích cụ thể và có giới hạn.



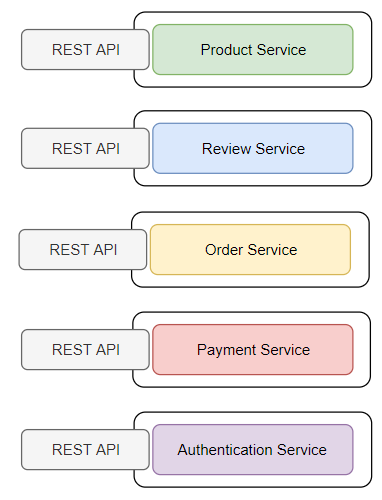
Hình 5.2 Kiến trúc microservices

Có thể thấy, với kiến trúc microservices mỗi service nằm ở 1 khối khác nhau, chứ không nằm chung 1 khối tổng thể như kiến trúc monolothic. Điều này giúp cho ứng dụng có thể scale một cách dễ dàng, có thể sử dụng đa ngôn ngữ cho ứng dụng.

## 5.3 Liên lạc giữa Microservices

Để client có thể lấy data từ phía backend, chúng tôi sẽ sử dụng REST để truyền tin từ phía backend tới client.

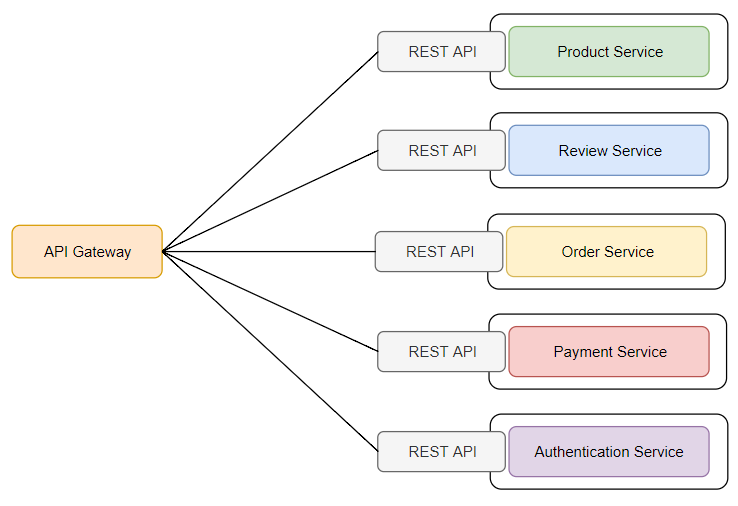
REST là sự lựa chọn hàng đầu vì nó cung cấp hệ thống truyền tin đơn giản qua giao thức HTTP dạng request - response. Do đó, nhiều microservices sử dụng HTTP với API. Mỗi chức năng xuất ra API.



Hình 5.3 Sử dụng REST API cho microservices

## 5.4 API Gateway

Trong kiến trúc microservice, developer frontend muốn lấy data từ các microsevice thì phải biết chi tiết URL, port. Đó không phải là thiết kế tốt. Để giải quyết vấn đề này, chúng tôi sẽ sử dụng API Gateway.



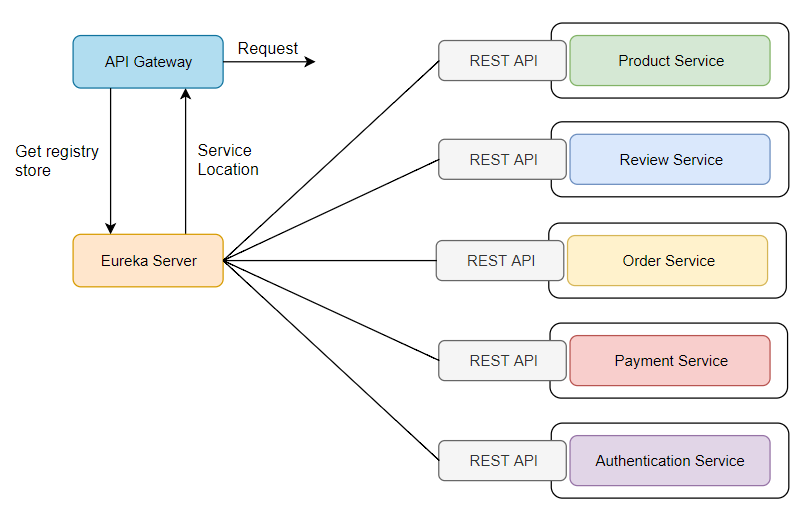
Hình 5.4 Tất cả microservice được xuất qua API Gateway

Trong ví dụ bán hàng này, tất cả microservices đều được xuất qua API Gateway và đây là một chốt đầu vào duy nhất cho người dùng.

## 5.5 Quản lý các Microservice

Quá trình xây dựng một ứng dụng theo kiểu Microservices sẽ có nhiều thử thách và một trong những thử thách đó là làm thế nào quản lý các service được tạo ra, bao gồm việc bảo trì, mở rộng và khả năng high availability của chúng. Ứng dụng càng lớn thì việc quản lý các service này sẽ lại càng khó khăn hơn.

Để giải quyết vấn đề này, chúng tôi sẽ sử dụng Eureka server, nơi dùng để để các microservice đăng ký và quản lý thông tin của chúng.



Hình 5.5 Microservice được quản lý bởi Eureka Server

## 5.6 Triển khai

Việc triển khai có một 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, lập trình viên có thể dễ dàng create và start toàn bộ các services phục vụ cho việc chạy ứng dụng.

# 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>