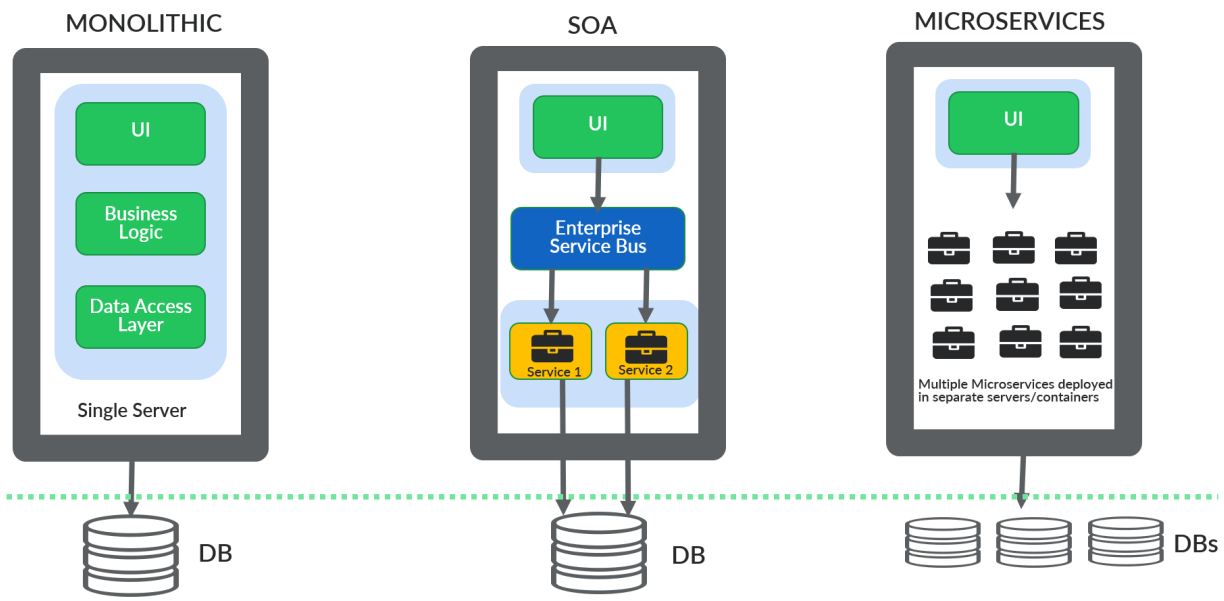


# **Section 1: Introduction to Microservices Architecture**

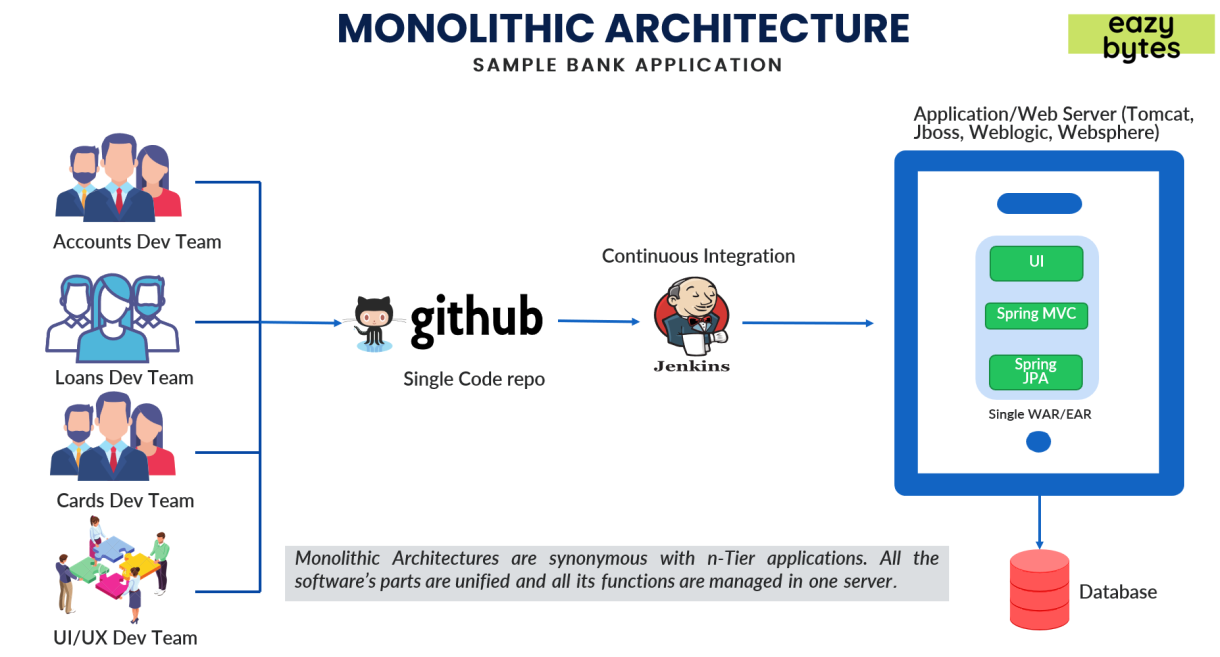
## **Evolution of Microservices architecture**



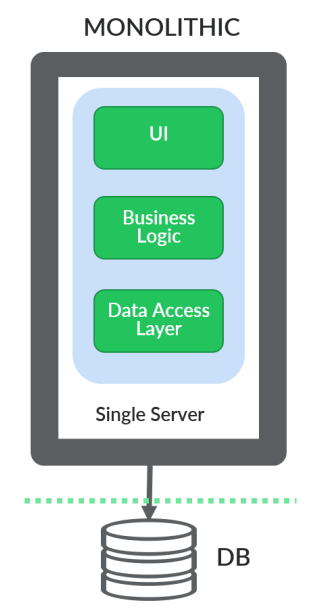
### **Kiến trúc nguyên khối (Monolithic Architecture)**

Monolith là một từ cổ để chỉ một khối đá khổng lồ. Trong công nghệ phần mềm, một mẫu nguyên khối đề cập đến một đơn vị phần mềm không thể chia tách. Khái niệm phần mềm nguyên khối nằm trong các thành phần khác nhau của ứng dụng được kết hợp thành một chương trình duy nhất trên một nền tảng duy nhất. Thông thường, ứng dụng nguyên khối bao gồm cơ sở dữ liệu, giao diện người dùng phía client và ứng dụng phía máy chủ. Tất cả các bộ phận của phần mềm được hợp nhất và tất cả các chức năng của phần mềm được quản lý ở một nơi. Chúng ta hãy xem xét cấu trúc của phần mềm nguyên khối một cách chi tiết.

Các thành phần của phần mềm nguyên khối được kết nối với nhau và phụ thuộc lẫn nhau, giúp phần mềm được khép kín. Kiến trúc này là một giải pháp truyền thống để xây dựng các ứng dụng, nhưng một số lập trình viên thấy nó đã lỗi thời. Tuy nhiên, trong nhiều trường hợp, một kiến trúc nguyên khối là một giải pháp hoàn hảo cho một số loại phần mềm.



Monolithic Architectures đồng nghĩa với các ứng dụng n-Tier. Tất cả các phần của phần mềm được thống nhất và tất cả các chức năng của nó được quản lý trong một máy chủ..



**Ưu điểm**

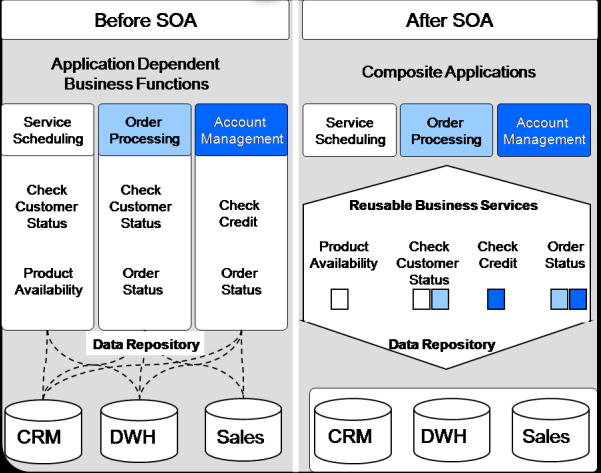
* Phát triển và triển khai đơn giản
* Hiệu suất tốt hơn  
  Nếu được xây dựng đúng cách, các ứng dụng nguyên khối thường có hiệu suất cao hơn các ứng dụng dựa trên microservice. Ứng dụng có kiến trúc microservice có thể cần thực hiện 40 lệnh gọi API đến 40 dịch vụ microservice khác nhau để load từng màn hình, điều này rõ ràng dẫn đến hiệu suất chậm hơn.

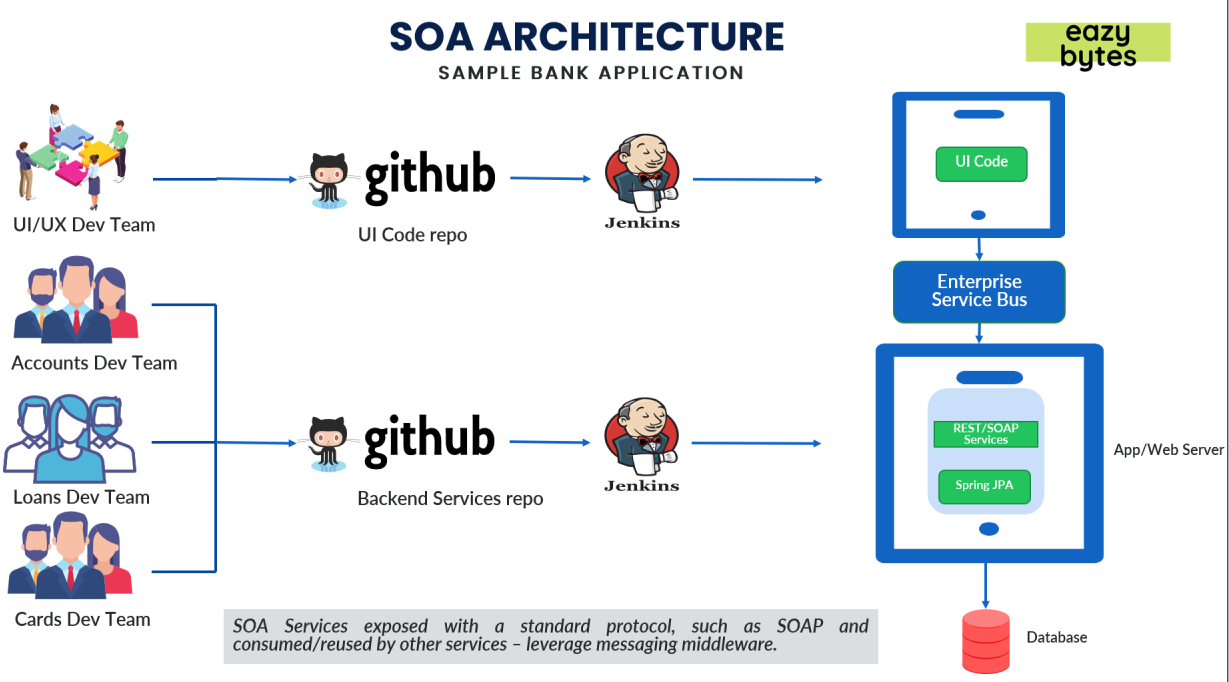
**Nhược điểm**

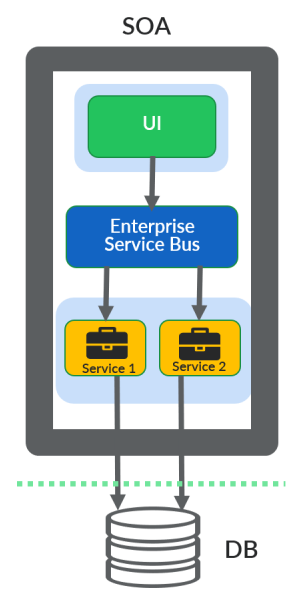
* Khó áp dụng công nghệ mới
* Hạn chế khi thay đổi
* Cơ sở mã đơn lẻ và khó bảo trì
* Cập nhật nhỏ và phát triển tính năng luôn cần triển khai đầy đủ

### **Kiến trúc hướng dịch vụ - SOA (Service-Oriented Architecture)**

SOA (Service-Oriented Architecture) hay thường được hiểu chính là kiến trúc hướng dịch vụ và là một mẫu kiến trúc sắp xếp phân tách ứng dụng lớn thành các dịch vụ (service). SOA là cấp độ cao hơn của phát triển ứng dụng thường dùng trong quy trình nghiệp vụ và giao tiếp chuẩn để giúp che đi sự phức tạp kỹ thuật. Cùng so sánh việc trước và sau khi sử dụng SOA sẽ như thế nào bằng hình minh hoạ bên dưới:







**Ưu điểm của SOA**

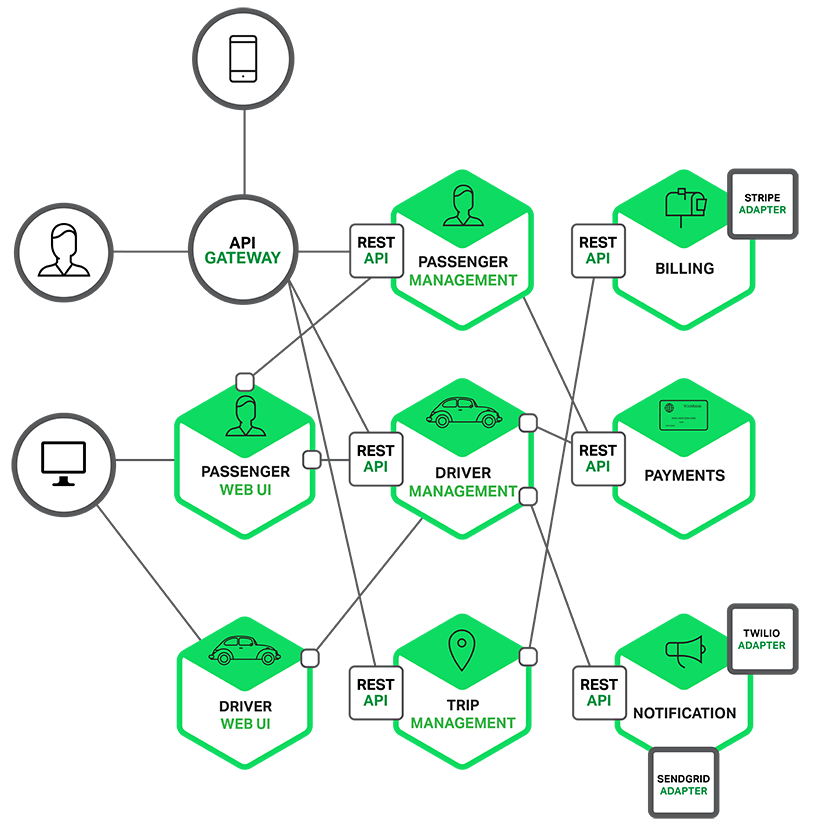
* Khả năng sử dụng lại dịch vụ
* Khả năng bảo trì tốt hơn
* Độ tin cậy cao hơn
* Phát triển song song

**Nhược điểm của SOA**

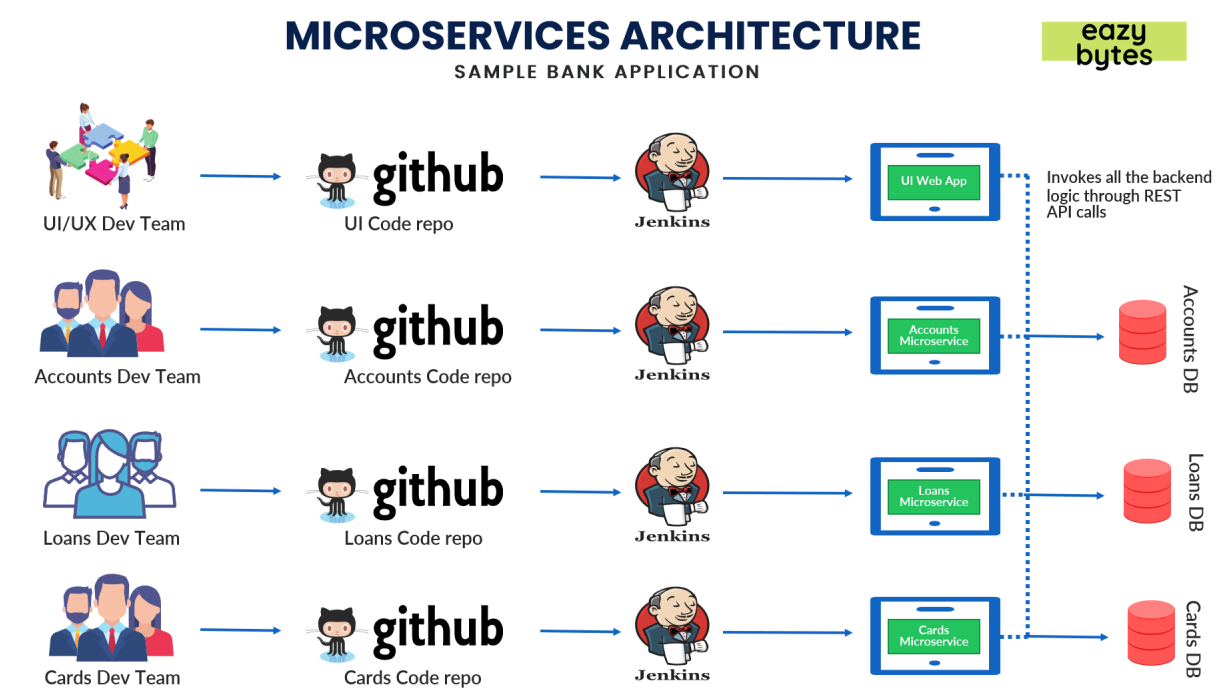
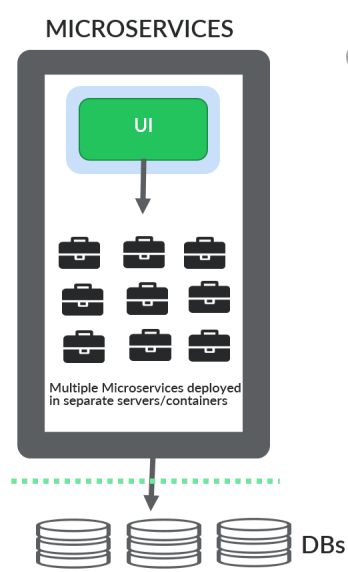
* Quản lý phức tạp
* Chi phí đầu tư cao
* Quá tải. Trong SOA, tất cả các đầu vào được xác nhận trước khi một dịch vụ tương tác với một dịch vụ khác. Khi sử dụng nhiều dịch vụ, điều này làm tăng thời gian phản hồi và giảm hiệu suất tổng thể.

### **Kiến trúc Microservice (Microservices Architecture)**

Microservice là một loại kiến trúc phần mềm hướng dịch vụ, tập trung vào việc xây dựng một loạt các thành phần tự quản lý tạo nên ứng dụng. Không giống như các ứng dụng nguyên khối được xây dựng dưới dạng một đơn vị không thể chia tách, các ứng dụng microservice bao gồm nhiều thành phần độc lập output ra các API.



Kiến trúc Microservices đã trở thành một xu hướng quan trọng trong lĩnh vực phát triển phần mềm trong vài năm qua và vẫn tiếp tục phát triển mạnh mẽ. Đây là một kiến trúc phần mềm phân tán, trong đó ứng dụng được chia thành các thành phần nhỏ độc lập gọi là microservices. Mỗi microservice thực hiện một chức năng cụ thể và có thể được triển khai, quản lý và mở rộng độc lập.



**Ưu điểm**

* Dễ phát triển, thử nghiệm và triển khai
* Tăng sự nhanh nhẹn
* Khả năng mở rộng theo chiều ngang
* Phát triển song song

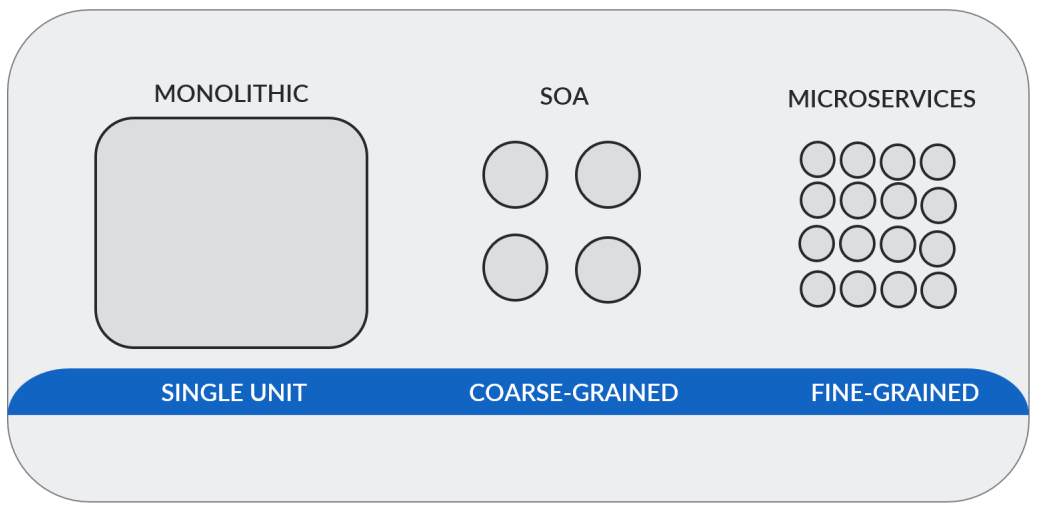
**Nhược điểm**

* Độ phức tạp
* Chi phí cơ sở hạ tầng
* Lo ngại về an ninh

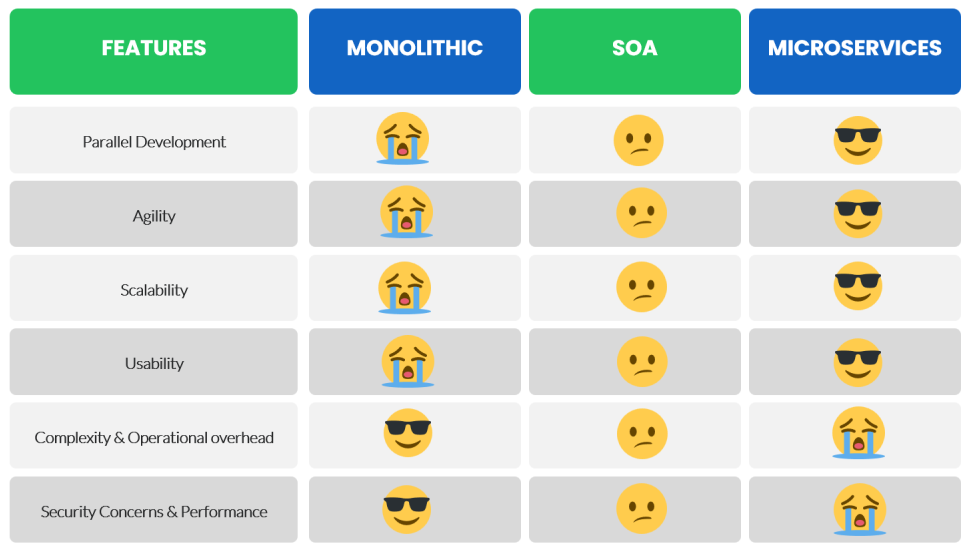
### **Kết luận cuối cùng**

* **Kiến trúc nguyên khối** (Monolithic Architecture) : Dự án Start-up nhỏ, Nguồn lực ít
* **Kiến trúc hướng dịch vụ** (SOA - Service Oriented Architecture) : Ứng dụng doanh nghiệp với nghiệp vụ phức tạp (Ngân hàng ...)
* **Kiến trúc Microservices** (Microservices Architecture) : Hệ thống cực lớn quy mô phức tạp, Đội ngũ phát triển đa năng

## **Monolithic Vs Soa Vs Microservices Comparision**



Single Unit, Coarse-Grained và Fine-Grained đề cập đến cách chia nhỏ và tổ chức các thành phần trong kiến trúc phần mềm. Single Unit là sự tổ chức ứng dụng thành một đơn vị duy nhất, trong khi Coarse-Grained và Fine-Grained tập trung vào mức độ chi tiết của việc chia nhỏ thành phần. Coarse-Grained chia nhỏ thành phần thành các đơn vị lớn, trong khi Fine-Grained chia nhỏ thành phần thành các đơn vị nhỏ độc lập.



Phát triển song song, Linh hoạt, Khả năng mở rộng, Khả năng sử dụng, Độ phức tạp và chi phí hoạt động, Mối quan tâm về bảo mật & Hiệu suất

# **Phần 2: [Optional] Microservices & Spring (Match Made in Heaven)**

## **Why Spring is the best framework for building microservices**

Spring là framework phát triển phổ biến nhất để xây dựng các ứng dụng và dịch vụ web dựa trên java. Từ Ngày đầu tiên, Spring đang làm việc để xây dựng mã của chúng tôi dựa trên các nguyên tắc như khớp nối lỏng lẻo bằng cách sử dụng phép nội xạ phụ thuộc. Trong những năm qua, Spring framework đang phát triển bằng cách duy trì sự phù hợp trên thị trường.



* **01:** Xây dựng các small service bằng SpringBoot cực kỳ dễ dàng & nhanh chóng
* **02:** Spring Cloud cung cấp công cụ cho dev xây dựng nhanh một số pattern phổ biến trong Microservices
* **03:** Cung cấp các tính năng sẵn sàng sản xuất như metrics, security, embedded servers
* **04:** Spring Cloud giúp triển khai microservice lên cloud dễ dàng
* **05:** Có một cộng đồng lớn các nhà phát triển Spring có thể trợ giúp và thích nghi dễ dàng

## **What is spring boot?**

Spring Boot giúp dễ dàng tạo các ứng dựa trên Spring cấp độ sản xuất(production-grade), độc lập mà bạn có thể "chỉ cần chạy".

* **STAND ALONE SPRING APPs:** Tạo các ứng dụng Spring độc lập/dịch vụ REST cực kỳ nhanh chóng và dễ dàng
* **NO NEED DEPLOY INTO A SERVER:** Máy chủ **Embed Tomcat, Jetty or Undertow** có sẵn và việc triển khai diễn ra trực tiếp
* **STARTER PROJECTS:** Starters projects là một tập hợp các bộ mô tả dependency thuận tiện mà bạn có thể sử dụng để xây dựng các ứng dụng Spring của mình.
* **AUTO CONFIGURATIONS:** Tự động cấu hình các thư viện/bean của Spring và bên thứ 3 bất cứ khi nào có thể
* **PROD READY FEATURES:** Hỗ trợ sẵn có các tính năng product như số liệu, kiểm tra tình trạng và cấu hình bên ngoài
* **SIMPLE CONFIGURATION:** Cung cấp nhiều annotation để thực hiện các cấu hình đơn giản và không yêu cầu cấu hình XML

## **What is spring cloud?**

* Spring Cloud cung cấp các công cụ để các nhà phát triển nhanh chóng xây dựng một số pattern phổ biến của Microservices
* **SPRING CLOUD CONFIG:** Đảm bảo rằng bất kể bạn đưa ra bao nhiêu phiên bản microservice; chúng sẽ luôn có cùng cấu hình.
* **SERVICE REGISTRATION & DISCOVERY:** Các service mới sẽ được đăng ký và sau này consumer có thể gọi chúng thông qua tên hợp lý thay vì vị trí thực tế
* **ROUTING & TRACING:** Đảm bảo rằng tất cả lệnh gọi đến vi dịch vụ của bạn đều đi qua một "front door" (cửa trước) duy nhất trước khi service mục tiêu được gọi và service tương tự sẽ được truy tìm.
* **LOAD BALANCING:** Cân bằng tải phân phối hiệu quả lưu lượng mạng đến nhiều backend server hoặc nhóm server
* **SPRING CLOUD SECURITY:** Cung cấp các tính năng liên quan đến bảo mật dựa trên token trong các Microservice/Spring Boot
* **SPRING CLOUD NETFLIX:** Spring Cloud Netflix chủ yếu tập trung vào việc tích hợp các dịch vụ của Netflix vào các ứng dụng Spring. Ví dụ:   
  - Service Discovery (Eureka): Là một dịch vụ đăng ký và phát hiện dịch vụ.   
  - Circuit Breaker (1-lystrix)  
  - Intelligent Routing (Zuul) Là một cổng (gateway) và proxy dịch vụ. Nó cho phép quản lý, bảo mật và điều hướng các yêu cầu từ bên ngoài tới các microservice bên trong hệ thống.   
  - Client Side Load Balancing (Ribbon):Là một thư viện cân bằng tải mặt trước (load balancing) cho các cuộc gọi giữa các microservice. Nó giúp phân phối tải đều giữa các phiên bản của một dịch vụ để tăng khả năng chịu tải và độ tin cậy.

## **Các phương pháp xác định ranh giới và kích thước phù hợp để xây dựng microservice**

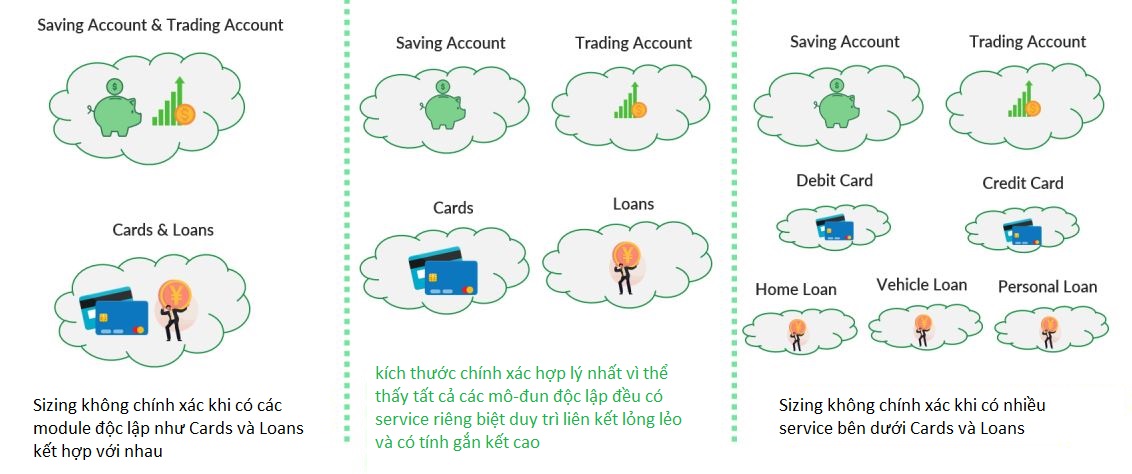
Một trong những khía cạnh thách thức nhất trong việc xây dựng một hệ thống microservice thành công là xác định các ranh giới microservice phù hợp và xác định kích thước của từng microservice

**Dưới đây là các cách tiếp cận phổ biến nhất trong ngành:**

* **Domain-Driven Sizing:** - Vì nhiều sửa đổi hoặc cải tiến của chúng tôi được thúc đẩy bởi nhu cầu kinh doanh, nên chúng tôi có thể xác định kích thước/xác định ranh giới của các microservice phù hợp chặt chẽ với khả năng Kinh doanh & Thiết kế theo hướng domain. Nhưng quá trình này mất rất nhiều thời gian và cần có kiến thức tốt về domain.
* **Event Storming Sizing:** Tiến hành một cuộc họp tương tác giữa các bên liên quan khác nhau để xác định danh sách các sự kiện quan trọng trong hệ thống như ‘Completed Payment’, ‘Search for a Product’, v.v. Dựa trên các sự kiện, chúng tôi có thể xác định ‘Commands’, 'Reactions’ và có thể cố gắng nhóm chúng thành các domain-driven service.

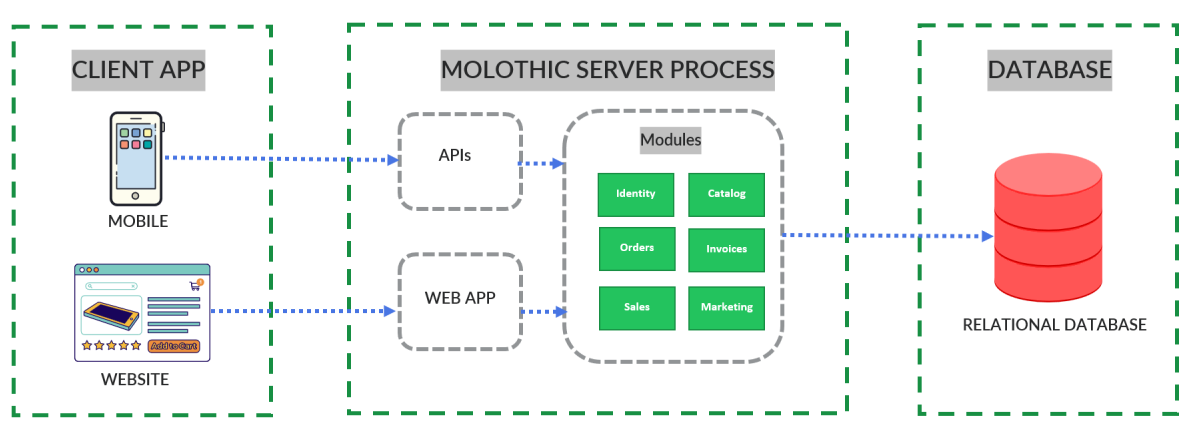
## **Sizing & identifying boundaries with a Bank App use case**

Bây giờ, hãy lấy một ví dụ về Bank application cần được xây dựng dựa trên kiến trúc vi dịch vụ và cố gắng xác định **ranh giới và kích thước** các dịch vụ.



## **Sizing & identifying boundaries with a Ecommerce migration use case**

Bây giờ, hãy lấy một tình huống trong đó một E-Commerce startup (công ty khởi nghiệp Thương mại điện tử) đang tuân theo kiến trúc nguyên khối(monolithic) và cố gắng hiểu những thách thức với nó là gì.



Vấn đề mà nhóm **Ecommerce** đang gặp phải do thiết kế monolithic truyền thống

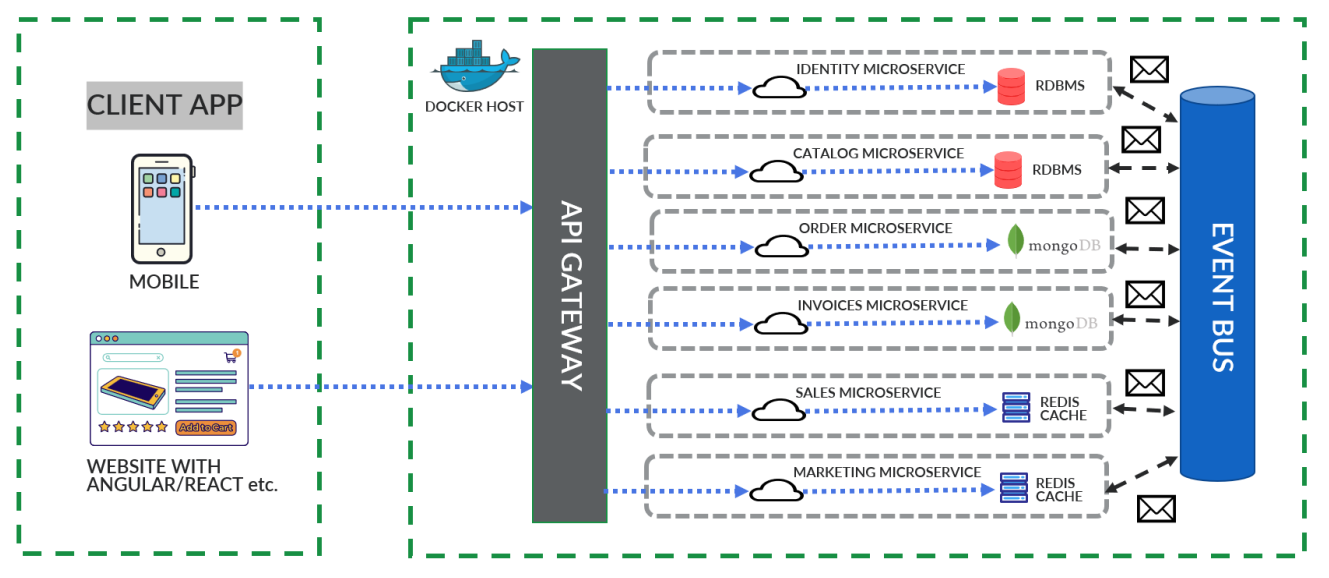
**Ngày đầu tiên**

* Dễ dàng xây dựng, thử nghiệm, triển khai, khắc phục sự cố và mở rộng quy mô trong quá trình khởi chạy và khi quy mô nhóm nhỏ

**Sau vài ngày, ứng dụng/trang web này trở thành một siêu hit và bắt đầu phát triển rất nhiều. Bây giờ nhóm có vấn đề dưới đây,**

* Ứng dụng đã trở nên cực kỳ phức tạp đến mức không một người nào hiểu được nó.
* Bạn sợ thực hiện các thay đổi - mỗi thay đổi đều có tác dụng phụ ngoài ý muốn và tốn kém.
* Các tính năng/bản sửa lỗi mới trở nên khó thực hiện, tốn thời gian và tốn kém.
* Mỗi lần phát hành càng nhỏ càng tốt và yêu cầu triển khai đầy đủ toàn bộ ứng dụng.
* Một thành phần không ổn định có thể làm hỏng toàn bộ hệ thống.
* Các công nghệ và khuôn khổ mới không phải là một lựa chọn.
* Rất khó để duy trì các nhóm nhỏ bị cô lập và thực hiện các phương pháp phân phối nhanh.

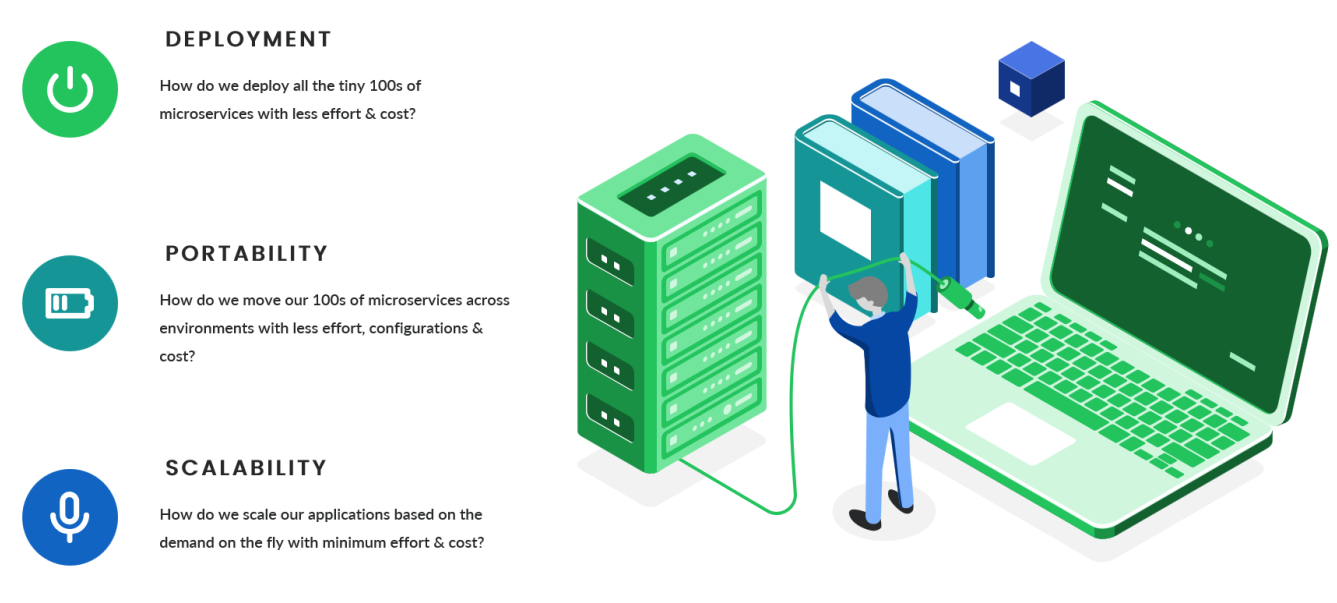
=> Vì vậy, **Ecommerce** đã quyết định và áp dụng thiết kế dựa trên cloud-native dưới đây bằng cách tận dụng kiến trúc Microservices để giúp dự án của họ trở nên dễ dàng và ít rủi ro hơn trước những thay đổi liên tục.



# **Session 3: How do we build, deploy, scale our microservices using** **Docker (Challenge 2)**

## **Introduction to challenges while building, deploying microservices**

Xây dựng và triển khai microservices có thể là một công việc thử thách nhưng cũng rất đáng để làm. Dưới đây là một số thách thức thường gặp:



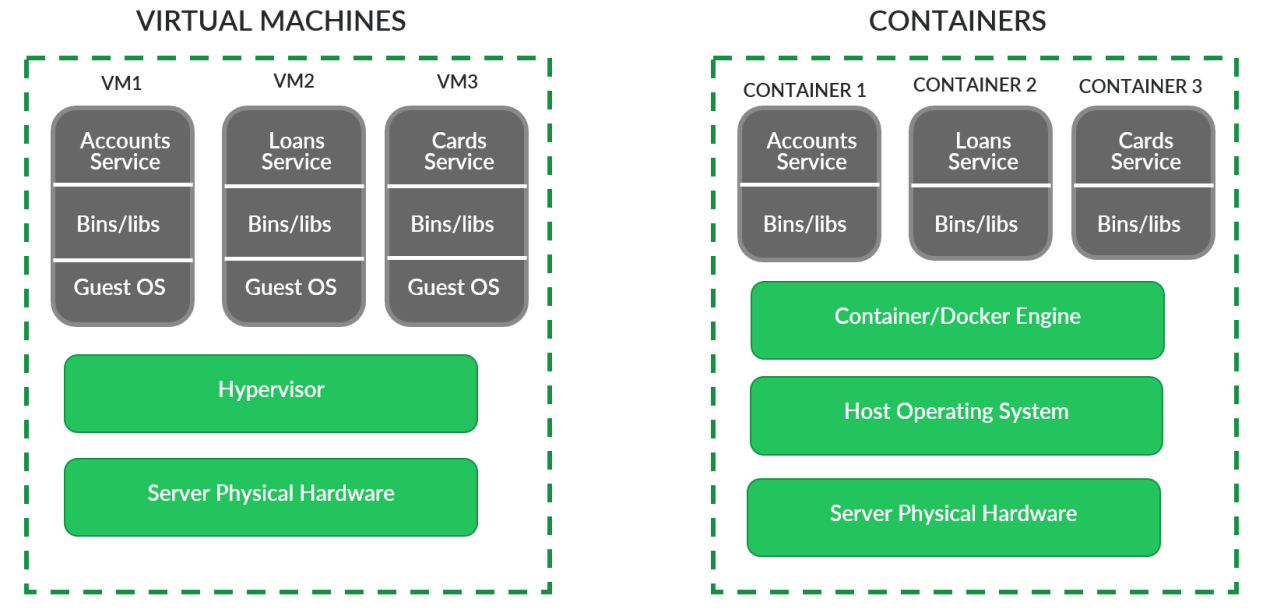
* **DEPLOYMENT:** Làm cách nào để chúng tôi triển khai tất cả 100 microservice với ít nỗ lực và chi phí hơn?
* **PORTABILITY**: Làm cách nào để chúng tôi di chuyển hơn 100 microservice qua các môi trường với ít nỗ lực, cấu hình và chi phí hơn?
* **SCALABILITY:** Làm cách nào để chúng tôi mở rộng quy mô các ứng dụng của mình dựa trên nhu cầu nhanh chóng với nỗ lực và chi phí tối thiểu?

## **Containerization technology**

Công nghệ container (containerization technology) là một phương pháp giúp đóng gói ứng dụng và tất cả các thành phần phụ thuộc của nó, bao gồm các thư viện và cấu hình hệ thống, vào một gói duy nhất được gọi là container. Mỗi container hoạt động độc lập và cô lập với các container khác, cho phép ứng dụng chạy một cách đáng tin cậy và nhất quán trên nhiều môi trường máy tính.

Công nghệ container cho phép các ứng dụng chạy trên môi trường máy tính của bạn mà không cần lo ngại về sự khác biệt trong cấu hình hệ thống. Mỗi container bao gồm mọi thứ cần thiết để chạy ứng dụng, giúp tránh các xung đột giữa các phần mềm và đảm bảo tính ổn định của hệ thống.

Một trong những công nghệ container phổ biến nhất là Docker, nhưng có nhiều công nghệ container khác nhau như Podman, LXC và rkt. Containerization giúp tăng cường tính di động, linh hoạt và hiệu quả trong việc triển khai và quản lý ứng dụng, đồng thời cung cấp cách tiếp cận hiện đại để xây dựng và chạy các ứng dụng trên các môi trường máy tính hiện đại.



Sự khác biệt chính giữa virtual machines và container. Các Container không cần Guest Os cũng như trình ảo hóa(hypervisor) để chỉ định tài nguyên; thay vào đó, họ sử dụng container engine.

* Máy ảo: Máy ảo là một môi trường ảo hoàn chỉnh, bao gồm hệ điều hành và ứng dụng, chạy trên một lớp ảo hóa hypervisor. Mỗi VM có riêng lịch trình tài nguyên của nó (bộ nhớ, CPU, ổ cứng) và hoạt động như một máy tính độc lập.
* Container: Container là một phần cách ly ứng dụng và các phụ thuộc của nó, chia sẻ cùng một hạt nhân hệ điều hành với máy chủ chủ. Container chia sẻ hạt nhân, nhưng có môi trường gốc riêng biệt, nơi ứng dụng chạy.
* Container thường nhẹ nhàng, khởi động nhanh chóng và tiết kiệm tài nguyên hơn so với máy ảo, nhưng có cơ chế cách ly ít hơn. VMs cung cấp cách cách ly hoàn toàn và có thể chạy các hệ điều hành khác nhau, nhưng tốn nhiều tài nguyên hơn và khởi động chậm hơn

## **Definition of containers**

### **Container là gì?**

**Container** là một môi trường biệt lập lỏng lẻo cho phép xây dựng và chạy các **software package** (gói phần mềm). **Software package** này bao gồm mã và tất cả các thành phần phụ thuộc để chạy các ứng dụng một cách nhanh chóng và đáng tin cậy trên bất kỳ môi trường máy tính nào. Nó được gọi là **packages container image**.

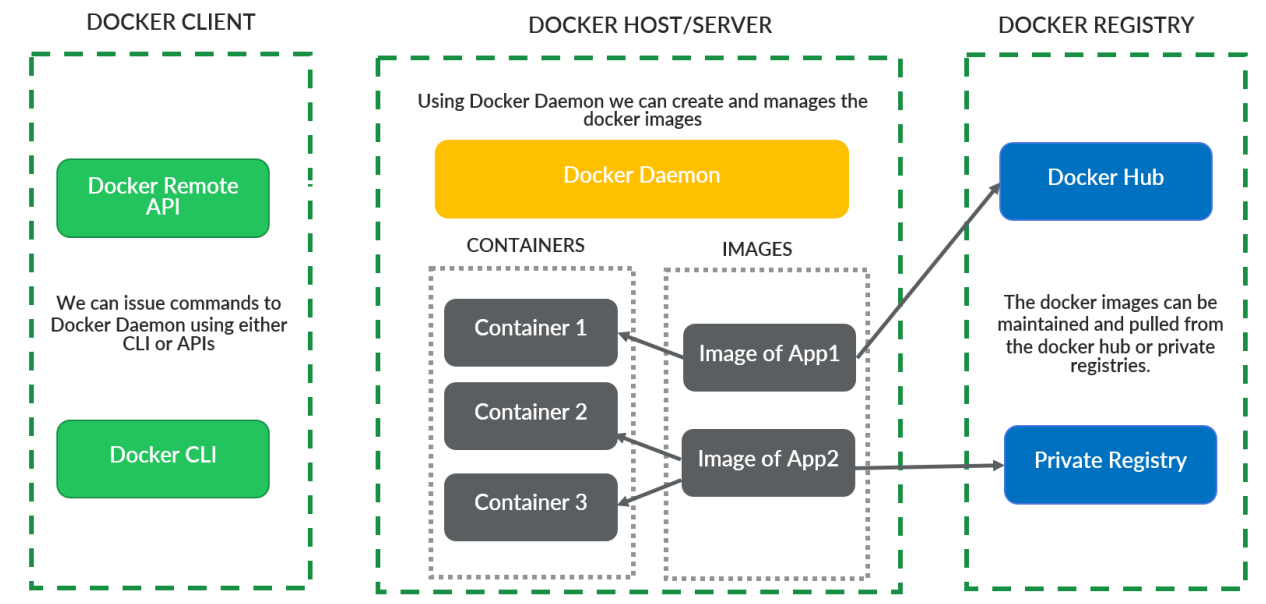
### **Software containerization**

**Software containerization** (Bộ chứa phần mềm) là một phương pháp ảo hóa hệ điều hành được sử dụng để triển khai và chạy **container** mà không cần sử dụng máy ảo (VM). **Container** có thể chạy trên phần cứng vật lý, trên đám mây, máy ảo và trên nhiều hệ điều hành.

### **Docker là gì?**

Docker là một trong những công cụ sử dụng ý tưởng về các tài nguyên bị cô lập để tạo ra một bộ công cụ cho phép đóng gói các ứng dụng với tất cả các phụ thuộc được cài đặt và chạy ở bất cứ đâu bạn muốn.

## **Introduction to Docker & its architecture**



Docker là một công nghệ containerization (đóng gói ứng dụng vào container) phổ biến và được sử dụng rộng rãi trong việc triển khai ứng dụng trên các môi trường khác nhau. Docker cung cấp một cách đơn giản và hiệu quả để đóng gói ứng dụng và các thành phần liên quan của nó vào trong một container, giúp tăng tính di động, quản lý và triển khai.

Kiến trúc của Docker bao gồm các thành phần sau:

* **Docker daemon:** Là một tiến trình chạy trên máy chủ, quản lý các container và cung cấp các API để tương tác với Docker.
* **Docker client:** Là một công cụ dòng lệnh hoặc giao diện người dùng đồ họa (GUI) để tương tác với Docker daemon.
* **Docker image:** Là một tập hợp các lệnh và tập tin được đóng gói trong một định dạng chuẩn, định nghĩa cách để tạo ra một container.
* **Docker container:** Là một thực thể của Docker image, chứa tất cả các thành phần cần thiết để chạy ứng dụng.
* **Docker registry:** Là một kho lưu trữ để lưu trữ các Docker image, cho phép các nhà phát triển chia sẻ các image với nhau.

Khi một Docker image được tạo ra, nó có thể được lưu trữ trên Docker registry hoặc chia sẻ với các nhà phát triển khác. Khi một container được tạo ra từ một image, nó có thể được khởi chạy trên bất kỳ máy chủ nào có Docker daemon chạy trên đó.

Kiến trúc của Docker cung cấp tính di động và khả năng triển khai dễ dàng cho các ứng dụng, đồng thời giúp tối ưu hóa sử dụng tài nguyên của hệ thống.

## **Understanding Docker Hub & Installing Docker**

Docker Hub là một dịch vụ kho lưu trữ ảnh Docker trực tuyến. Nó cung cấp cho người dùng một nơi để lưu trữ, quản lý và chia sẻ các ảnh Docker, và cũng là một nguồn tài nguyên phong phú để tìm kiếm các ảnh Docker chất lượng từ cộng đồng Docker.

Để sử dụng Docker Hub, người dùng cần đăng ký tài khoản và đăng nhập vào trang web Docker Hub. Sau đó, họ có thể tìm kiếm các ảnh Docker sẵn có trên Docker Hub hoặc tải lên các ảnh Docker của riêng mình để chia sẻ với cộng đồng.

Để cài đặt Docker, bạn có thể thực hiện các bước sau:

* Truy cập trang web chính thức của Docker và tải xuống phiên bản Docker phù hợp với hệ điều hành của bạn.
* Cài đặt Docker trên hệ thống của bạn bằng cách chạy tệp cài đặt và làm theo các hướng dẫn trên màn hình.
* Sau khi cài đặt xong, kiểm tra phiên bản Docker bằng cách chạy lệnh sau trong terminal hoặc command prompt:

docker version

* Nếu phiên bản Docker hiện tại được hiển thị, điều đó có nghĩa là Docker đã được cài đặt thành công trên hệ thống của bạn.

Sau khi cài đặt Docker, bạn có thể sử dụng Docker CLI (command-line interface) để quản lý các container và ảnh Docker trên hệ thống của bạn.

## **Creating Docker image definition using a Dockerfile**

Để tạo Docker image cho dự án Spring Boot, bạn cần sử dụng Dockerfile. Dockerfile là một tệp văn bản đơn giản chứa các hướng dẫn để xây dựng Docker image của ứng dụng của bạn. Dưới đây là cách tạo một Dockerfile cho dự án Spring Boot cụ thể:

### **Bước 1: Chuẩn bị các tệp cần thiết:**

Trước tiên, bạn nên đảm bảo rằng tệp .jar của ứng dụng Spring Boot đã được xây dựng thành công và có sẵn để triển khai. Đảm bảo rằng tệp .jar này được đặt trong thư mục gốc của dự án.

Vào thư mục của từng service:

mvn clean install

### **Bước 2: Tạo Dockerfile:**

Tạo một tệp có tên "Dockerfile" trong thư mục gốc của dự án Spring Boot và thêm các hướng dẫn sau vào Dockerfile:

#Start with a base image containing Java runtime

FROM openjdk:17-jdk-slim as build

#Information around who maintains the image

MAINTAINER eazybytes.com

# Add the application's jar to the container

COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar

#execute the application

ENTRYPOINT ["java","-jar","/accounts-0.0.1-SNAPSHOT.jar"]

### **Bước 3: Xây dựng Docker image:**

Sau khi bạn đã tạo Dockerfile, tiếp theo là xây dựng Docker image bằng cách chạy lệnh sau trong cửa sổ dòng lệnh:

docker build -t your-image-name:tag .

Trong đó:

* "your-image-name" là tên bạn muốn đặt cho image của bạn.
* "tag" là phiên bản hoặc tag của image (ví dụ: latest, v1.0, ...). Nếu không chỉ định tag, mặc định sẽ là "latest".
* Dấu chấm ở cuối lệnh chỉ định thư mục hiện tại làm nguồn cho việc xây dựng image.

docker build -t 1995mars/account .

### **Bước 4: Chạy container từ Docker image:**

Khi Docker image đã được xây dựng, bạn có thể chạy container bằng cách chạy lệnh sau:

docker run -p 8080:8080 your-image-name:tag

Trong đó, "8080:8080" là cổng mà ứng dụng trong container sẽ lắng nghe và bạn có thể truy cập ứng dụng của mình qua cổng 8080 trên máy localhost.

docker run -p 8080:8080 1995mars/account

## **Deep dive of important Docker commands**

Docker cung cấp một loạt các lệnh để quản lý các container, ảnh (image), mạng (network) và khối lưu trữ (volume). Sau đây là một số lệnh Docker quan trọng:

* **docker run:** Lệnh này được sử dụng để tạo và chạy một container từ một ảnh Docker. Ví dụ: docker run -it ubuntu bash sẽ tạo một container mới từ image Ubuntu và mở một phiên bản terminal bash bên trong container.
* **docker pull:** Lệnh này được sử dụng để tải về một image Docker từ Docker Hub hoặc một registry khác. Ví dụ: docker pull nginx sẽ tải về image Nginx mới nhất từ Docker Hub.
* **docker build:** Lệnh này được sử dụng để xây dựng một image Docker từ Dockerfile. Ví dụ: docker build -t myimage . sẽ xây dựng một image Docker từ Dockerfile được đặt trong thư mục hiện tại và đặt tên là "myimage".
* **docker ps:** Lệnh này được sử dụng để liệt kê tất cả các container đang chạy trên hệ thống. Ví dụ: docker ps sẽ liệt kê tất cả các container đang chạy trên hệ thống.
* **docker stop:** Lệnh này được sử dụng để dừng một container đang chạy. Ví dụ: docker stop mycontainer sẽ dừng container có tên "mycontainer".
* **docker rm:** Lệnh này được sử dụng để xóa một container. Ví dụ: docker rm mycontainer sẽ xóa container có tên "mycontainer".
* **docker rmi:** Lệnh này được sử dụng để xóa một image Docker. Ví dụ: docker rmi myimage sẽ xóa image Docker có tên "myimage".
* **docker network:** Lệnh này được sử dụng để quản lý các mạng Docker. Ví dụ: docker network create mynetwork sẽ tạo một mạng Docker mới có tên "mynetwork".
* **docker volume:** Lệnh này được sử dụng để quản lý các khối lưu trữ Docker. Ví dụ: docker volume create myvolume sẽ tạo một khối lưu trữ Docker mới có tên "myvolume".
* **docker exec:** Lệnh này được sử dụng để thực thi một lệnh bên trong một container đang chạy. Ví dụ: docker exec -it mycontainer bash sẽ mở một phiên bản terminal bash bên trong container có tên "mycontainer".

Trên đây là một số lệnh Docker quan trọng, tuy nhiên có nhiều lệnh khác nữa để quản lý các container và image Docker. Bạn nên tìm hiểu và sử dụng các lệnh này để quản lý các ứng dụng của mình trên Docker một cách hiệu quả.

## **Introduction to Buildpacks**

Buildpacks là một công nghệ được sử dụng trong quá trình triển khai ứng dụng và các dịch vụ cloud để tự động hóa việc đóng gói ứng dụng và tạo các môi trường chạy mà không cần phải quan tâm đến cơ sở hạ tầng cụ thể. Buildpacks đảm bảo rằng ứng dụng của bạn được đóng gói một cách chính xác, có thể chạy trên các môi trường khác nhau một cách nhất quán, giúp giảm thiểu sự cố, tối ưu hóa hiệu suất và tăng tính di động.

Cơ chế hoạt động của Buildpacks hoạt động như sau:

* Phát hiện ngôn ngữ: Buildpacks xác định ngôn ngữ lập trình được sử dụng trong mã nguồn của ứng dụng.
* Đóng gói ứng dụng: Sau khi xác định ngôn ngữ, Buildpacks sẽ tạo ra môi trường chạy chính xác dựa trên yêu cầu của ứng dụng. Nó sẽ tự động tải xuống và cài đặt các phụ thuộc cần thiết, bao gồm phiên bản của ngôn ngữ lập trình, thư viện và công cụ cần thiết khác.
* Xây dựng ứng dụng: Sau khi xác định và chuẩn bị môi trường chạy, Buildpacks sẽ tiến hành xây dựng mã nguồn thành các thành phần thực thi có thể chạy được.
* Đóng gói: Cuối cùng, Buildpacks sẽ đóng gói ứng dụng đã xây dựng thành các container hoặc các định dạng triển khai ứng dụng khác nhau, sẵn sàng cho việc triển khai lên các môi trường cloud hoặc nền tảng PaaS (Platform as a Service).

Lợi ích chính của Buildpacks bao gồm:

* Tích hợp nền tảng: Buildpacks giúp bạn triển khai ứng dụng một cách dễ dàng và nhất quán trên các môi trường khác nhau, bao gồm các nhà cung cấp dịch vụ đám mây khác nhau như AWS, Azure, Google Cloud, Heroku, v.v.
* Tự động hóa: Buildpacks loại bỏ sự cần thiết phải cấu hình và tùy chỉnh môi trường chạy bằng cách thực hiện một phần lớn công việc tự động hóa, giảm nguy cơ sai sót do con người và tăng tính nhất quán.
* Độ tin cậy: Khi sử dụng Buildpacks, ứng dụng của bạn sẽ chạy trên một môi trường có độ tin cậy cao, giúp giảm thiểu sự cố và giữ cho ứng dụng của bạn luôn ổn định.
* Hiệu suất tối ưu: Các Buildpacks đảm bảo rằng các ứng dụng được xây dựng một cách tối ưu, điều này có thể dẫn đến hiệu suất tốt hơn và tiết kiệm tài nguyên.

Buildpacks đã trở thành một công nghệ phổ biến trong cộng đồng phát triển phần mềm và đang được sử dụng rộng rãi trong các dự án ứng dụng và hệ thống phức tạp.

Trong Spring Java, Buildpacks là một cơ chế để xây dựng và đóng gói ứng dụng Java thành các container hỗ trợ, chẳng hạn như Docker images. Buildpacks giúp tự động hóa quy trình xây dựng và triển khai ứng dụng, đồng thời đảm bảo rằng ứng dụng của bạn có thể chạy một cách đáng tin cậy và hiệu quả trên các môi trường khác nhau.

Các tính năng chính của Buildpacks trong Spring Java bao gồm:

* Xây dựng tự động: Buildpacks xác định ngôn ngữ lập trình và các thành phần phụ thuộc trong mã nguồn của ứng dụng Java và tự động tạo môi trường chạy phù hợp.
* Đóng gói container: Sau khi xây dựng ứng dụng, Buildpacks sẽ đóng gói ứng dụng thành các container như Docker images. Container là một cách phổ biến để triển khai ứng dụng đáng tin cậy và di động trên nhiều môi trường.
* Linh hoạt và nhất quán: Buildpacks cho phép bạn triển khai ứng dụng Java một cách nhất quán trên các nền tảng khác nhau, bao gồm các nhà cung cấp đám mây và các môi trường PaaS.
* Tối ưu hóa hiệu suất: Buildpacks giúp tối ưu hóa hiệu suất của ứng dụng Java bằng cách cung cấp môi trường chạy tối ưu và loại bỏ những thành phần không cần thiết.

Để sử dụng Buildpacks trong dự án Spring Java, bạn có thể sử dụng các công cụ hỗ trợ như Spring Boot CLI hoặc các framework triển khai ứng dụng Spring Boot như Spring Cloud, Spring Cloud Data Flow và Spring Cloud Services. Các công cụ này giúp bạn triển khai ứng dụng và sử dụng Buildpacks một cách dễ dàng và hiệu quả.

## **Creating docker image of Loans microservice using Buildpacks**

**Buildpacks** tạo ra các image mà không cần phải tạo ra **Dockerfile**

**Trong file pom.xml**

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
</build>

**Chạy lệnh command:**

mvn spring-boot:build-image

Lệnh mvn spring-boot:build-image được sử dụng để build một Docker image từ một ứng dụng Spring Boot bằng cách sử dụng plugin "spring-boot-maven-plugin".

Khi chạy lệnh này, plugin sẽ đóng gói ứng dụng của bạn dưới dạng một file JAR và sau đó sử dụng Dockerfile được tạo ra tự động để build một image Docker. Image này sẽ có kích thước nhỏ hơn so với việc build image bằng cách sử dụng một hệ thống build khác, vì nó chỉ bao gồm các thành phần cần thiết để chạy ứng dụng Spring Boot.

## **Pushing Docker images from your local to remote Docker hub repository**