

## MỤC LỤC

LỜI CẢM ƠN .....	5
MỤC LỤC .....	7
DANH MỤC HÌNH ẢNH .....	9
TÓM TẮT .....	11
ABSTRACT .....	12
PHẦN GIỚI THIỆU .....	13
1. ĐẶT VẤN ĐỀ: .....	13
2. MỤC TIÊU ĐỀ TÀI .....	14
3. NỘI DUNG NGHIÊN CỨU .....	14
4. BỐ CỤC CỦA LUẬN VĂN .....	14
PHẦN NỘI DUNG .....	15
CHƯƠNG 1: MÔ TẢ BÀI TOÁN .....	15
CHƯƠNG 2: CƠ SỞ LÝ THUYẾT .....	18
1. DOCKER .....	18
1.1. Lý do Docker ra đời? .....	18
1.2. Docker là gì? .....	18
1.3. Tại sao chúng ta cần Docker? .....	18
1.4. Các khái niệm cơ bản .....	19
1.5. Một số lệnh cơ bản .....	23
1.6. Các khái niệm cốt lõi của Container .....	25
1.7. Cách tạo Docker Image .....	28
2. KUBERNETES .....	29
2.1. Lý do Kubernetes ra đời? .....	29
2.2. Kubernetes là gì? .....	30
2.3. Microservices .....	30
2.4. Kiến trúc của một cụm Kubernetes .....	33
2.5. Chạy một ứng dụng trong Kubernetes .....	35
2.6. Các khái niệm cơ bản .....	35
2.6.1. Kubectl .....	35
2.6.2. Pod .....	36
2.6.3. Replication và các Controller khác .....	39
2.6.4. Service .....	40
2.6.5. Volume .....	42
2.6.6. ConfigMap và Secret .....	43

2.6.6.1. <i>ConfigMap</i> .....	44
2.6.6.2. <i>Secret</i> .....	44
2.6.7. <i>Deployment</i> .....	44
2.6.8. <i>StatefulSet</i> .....	46
2.6.9. <i>Tự động mở rộng Pod và Cluster Node</i> .....	50
<b>3. COUCHDB.....</b>	<b>54</b>
3.1. <i>CouchDB là gì?</i> .....	54
3.2. <i>Tại sao chúng ta lại cần CouchDB ?</i> .....	54
3.3. <i>Mô hình dữ liệu</i> .....	54
3.4. <i>Các tính năng chính</i> .....	54
3.5. <i>Công nghệ Cluster</i> .....	55
<b>4. NFS .....</b>	<b>57</b>
4.1. <i>NFS là gì?</i> .....	57
4.2. <i>Những tính năng của NFS là gì?</i> .....	59
4.3. <i>Một vài tùy chọn của đặc quyền cần biệt.</i> .....	59
<b>CHƯƠNG 3: PHƯƠNG PHÁP THỰC HIỆN .....</b>	<b>60</b>
1. <b>TỔNG QUAN BÀI TOÁN.....</b>	<b>60</b>
2. <b>CÁC BƯỚC THỰC HIỆN .....</b>	<b>60</b>
<b>CHƯƠNG 4: PHÂN TÍCH VÀ ĐÁNH GIÁ .....</b>	<b>78</b>
1. <b>KẾT QUẢ ĐẠT ĐƯỢC .....</b>	<b>78</b>
2. <b>HẠN CHÉ .....</b>	<b>78</b>
3. <b>HƯỚNG PHÁT TRIỂN.....</b>	<b>78</b>
<b>TÀI LIỆU THAM KHẢO .....</b>	<b>79</b>

## **DANH MỤC HÌNH ẢNH**

*Hình 1: Docker*

*Hình 2: Tại sao chúng ta cần Docker?*

*Hình 3: Docker Engine*

*Hình 4: Phân biệt Container với Virtual Machine*

*Hình 5: Sơ đồ tổng quan*

*Hình 6: Container Life Cycle*

*Hình 7: SSH vào trong Container*

*Hình 8: Port Mapping*

*Hình 9: Volume*

*Hình 10: Các thành phần bên trong Monolithic Application và Microservices*

*Hình 11: Microservices*

*Hình 13: Kiến trúc của một cụm Kubernetes*

*Hình 14: Tất cả Container của một Pod chạy trên cùng một Node.*

*Hình 15: Mỗi Pod có một IP riêng. Các Pod giao tiếp với nhau bằng IP đó*

*Hình 16: Một Container không được chạy nhiều tiến trình.*

*Hình 17: Mô tả RelicationToken.*

*Hình 18: Service NodePort*

*Hình 19: Mô tả Deployment.*

*Hình 20: Scale StatefulSet.*

*Hình 21: Persistent Volume.*

*Hình 22: Persistent Volume Claims.*

*Hình 23: Horizontal controller*

*Hình 24: Minh họa AutoScaler*

*Hình 25: Quy trình tiến hành AutoScale*

*Hình 26: Cluster là gì?*

*Hình 27: Sơ đồ tổng quát quá trình triển khai CouchDB*

*Hình 28: Sơ đồ tổng quát quá trình triển khai ứng dụng Web*

*Hình 29: Minh họa Scale tự động 1*

*Hình 30: Minh họa Scale tự động 2*

## TÓM TẮT

Công nghệ Containers được tạo ra để khắc phục trường hợp khi một đoạn chương trình hoặc phần mềm chạy ổn định trên máy của lập trình viên nhưng lại không thể chạy trên Server vì thiếu các thư viện cũng như môi trường. Vì thế Containers được sinh ra để chứa toàn bộ các thành phần, môi trường để chạy phần mềm một cách nhẹ nhất, độc lập và có thể để ứng dụng chạy nhanh chóng và đáng tin cậy từ môi trường máy này sang môi trường máy khác.

Kubernetes là một hệ thống phần mềm cho phép chúng ta dễ dàng triển khai và quản lý các ứng dụng được đóng gói trên nó. Có thể nói Kubernetes là một hệ thống quản lý chuỗi Container. Nó theo dõi, quản lý và mở rộng tự động các Container.

Trong đề tài “Mở Rộng Tự Động Quy Mô Ứng Dụng Web Với Kubernetes”, chúng tôi sẽ triển khai một cụm Kubernetes với 3 Node trên 3 máy ảo Ubuntu để tiến hành triển khai Database NoSQL (CouchDB) và ứng dụng Web (React App). Quản lý, theo dõi và mở rộng quy mô của nó với Kubernetes.

## **ABSTRACT**

Containers technology was created to overcome the case when a piece of program or software runs stably on the developer's machine but cannot run on the Server because of the lack of libraries and environments. So Containers were born to contain all the components, the environment to run the software in the lightest, independent way and be able to let the application run quickly and reliably from one machine environment to another. .

Kubernetes is a software system that allows us to easily deploy and manage applications packaged on it. It can be said that Kubernetes is a Container chain management system. It tracks, manages and AutoScaler Containers.

In the topic "Autoscaling Web Applications with Kubernetes: Online Shopping", we will deploy a Kubernetes Cluster with 3 Nodes on 3 Ubuntu virtual machines to deploy NoSQL Database (CouchDB) and Web application (React App). ). Manage, track and scale it with Kubernetes.

## PHẦN GIỚI THIỆU

### 1. ĐẶT VẤN ĐỀ:

Ở xã hội ngày nay, công nghệ thông tin được coi là một trong những ngành có vai trò quan trọng trong việc phát triển đất nước. Công nghệ thông tin giờ đây đã có mặt ở nhiều phương diện trong cuộc sống hàng ngày của chúng ta, từ thương mại đến giải trí và thậm chí cả văn hóa, xã hội và giáo dục... Đặc biệt là trong tình hình dịch bệnh như hiện nay, khi mà chúng ta cần hạn chế tiếp xúc để đảm bảo an toàn sức khỏe bản thân cũng như ngăn chặn dịch bệnh lây lan rộng trong cộng đồng. Việc ứng dụng công nghệ thông tin để giúp người dân nắm bắt thông tin tình hình dịch bệnh hay chỉ đơn giản là giúp người dân dễ dàng mua nhu yếu phẩm, đồ dùng sinh hoạt phục vụ đời sống cá nhân là rất cần thiết và tiện lợi.

Xây dựng một ứng dụng Web mua sắm online trong tình hình hiện nay là một lựa chọn đúng đắn. Và việc xây dựng một ứng dụng Web thì thời gian hoạt động liên tục của máy chủ là yếu tố vô cùng quan trọng đối với người sử dụng, đặc biệt là đối với một doanh nghiệp. Khi xảy ra thời gian chết hệ thống hoặc lưu lượng truy cập quá mức khiến máy chủ bị quá tải thì việc có một máy chủ dự phòng là cần thiết. Tuy nhiên việc đó khiến chúng ta phải đổi mới với vấn đề về chi phí. Có thể chúng ta sẽ phải tăng gấp đôi chi phí cho một máy chủ vật lý chính và một máy chủ vật lý dự phòng, ngoài ra còn có không gian đặt máy chủ vật lý và các hệ thống khác kèm theo.

Bên cạnh đó, việc ứng dụng được chạy trên máy chủ vật lý dẫn đến một vấn đề. Không thể xác định ranh giới tài nguyên cho các ứng dụng trong một máy chủ vật lý và điều này gây ra sự cố phân bổ tài nguyên. Ví dụ nếu nhiều ứng dụng cùng chạy trên một máy chủ vật lý, có thể có những trường hợp một ứng dụng sẽ chiếm phần lớn tài nguyên hơn và kết quả là các ứng dụng khác sẽ hoạt động kém đi. Một giải pháp cho điều này sẽ là chạy từng ứng dụng trên một máy chủ vật lý khác nhau. Nhưng giải pháp này không tối ưu vì tài nguyên không được sử dụng đúng mức và một lần nữa chúng ta lại phải đổi mới với vấn đề chi phí để có thể duy trì nhiều máy chủ vật lý cùng lúc. Để giải quyết những vấn đề trên, Kubernetes là một câu trả lời thỏa đáng.

Kubernetes là một nền tảng nguồn mở, khả chuyền, có thể mở rộng để quản lý các ứng dụng được đóng gói và các Service, giúp thuận lợi trong việc cấu hình và tự động hóa việc triển khai ứng dụng. Kubernetes cung cấp một framework để chạy các hệ phân tán một cách mạnh mẽ. Nó đảm nhiệm việc nhân rộng và chuyền đổi dự phòng cho ứng dụng, cung cấp các mẫu Deployment và hơn thế nữa.

Trong luận văn nay, chúng tôi sẽ đi sâu vào tìm hiểu về Kubernetes cũng như cách đóng gói một ứng dụng Web thành Container và mở rộng tự động ứng dụng Web khi lưu lượng truy cập tăng cao làm hệ thống bị quá tải. Quá trình nghiên cứu tiến hành qua 4 giai đoạn: 1. Tìm hiểu các kiến thức về Kubernetes và Autoscaler (mở rộng tự động) trong Kubernetes, 2. Cài đặt, cấu hình Kubernetes Cluster, 3. Triển khai CouchDB Cluster và ứng dụng Web lên Kubernetes, 4. Tiến hành scale tự động CouchDB và ứng dụng Web.

## 2. MỤC TIÊU ĐỀ TÀI

Với đề tài “Mở rộng tự động quy mô ứng dụng Web với Kubernetes”, các mục tiêu được đề ra là:

- Hiểu được các kiến thức về Kubernetes, Docker, CouchDB Cluster.
- Triển khai thành công CouchDB Cluster và ứng dụng Web lên Kubernetes.
- Biết cách tạo image với Docker, đóng gói ứng dụng Web và triển khai trên Kubernetes.
- Hiểu công dụng của AutoScaler trong Kubernetes và cách scale tự động ứng dụng Web.

## 3. NỘI DUNG NGHIÊN CỨU

Nội dung nghiên cứu bao gồm các phần:

- Phân tích và nghiên cứu kiến thức.
- Tiến hành cài đặt Kubernetes Cluster trên Ubuntu.
- Triển khai CouchDB Cluster, cài đặt Scale tự động CouchDB Cluster.
- Triển khai ứng dụng Web, cài đặt Scale tự động ứng dụng Web.
- Kiểm tra và đánh giá sự hoạt động của CouchDB Cluster và ứng dụng Web sau khi Scale.

## 4. BỐ CỤC CỦA LUẬN VĂN

Bố cục luận văn được chia thành các phần:

- **Phần giới thiệu:** Đặt vấn đề, đưa ra mục tiêu đề tài
- **Phần nội dung:** Mô tả bài toán, các cơ sở lý thuyết được áp dụng, phương pháp thực hiện.
- **Phần kết luận:** Kết quả đạt được, các mặt hạn chế và phương hướng phát triển tiếp theo.

## **PHẦN NỘI DUNG**

### **CHƯƠNG 1: MÔ TẢ BÀI TOÁN**

Hiện nay, xây dựng một ứng dụng Web mua sắm online trong tình hình dịch bệnh cần hạn chế tiếp xúc là rất cần thiết. Người dùng chỉ cần chọn mặt hàng cần mua, cho vào giỏ hàng và nhấn đặt hàng, hệ thống sẽ xử lý đơn hàng và vận chuyển đến tận nơi của người dùng. Tuy nhiên, vấn đề đặt ra là làm thế nào để xử lý một lượng dữ liệu lớn và đang tăng lên hàng ngày một từ nhiều khu vực khác nhau hay thậm chí là trên toàn quốc. Bên cạnh đó là việc máy chủ phải hoạt động liên tục để có thể đáp ứng nhu cầu của người dùng bất cứ lúc nào và phải giải quyết thế nào khi lưu lượng truy cập quá lớn khiến máy chủ bị quá tải không thể đáp ứng hết các yêu cầu từ người dùng. Vì khi máy chủ xảy ra sự cố, các dữ liệu được lưu trữ có thể biến mất và nó làm ảnh hưởng đến quá trình mua sắm của người dùng. Chính vì vậy, để giải quyết vấn đề này thì cần một hệ thống có khả năng mở rộng tốt và có khả năng chịu lỗi cao. Từ những yêu cầu trên em đã đưa ra một mô hình sử dụng Kubernetes trong việc tự động mở rộng ứng dụng Web và Database khi lưu lượng truy cập cao hoặc khi có Container bị lỗi không thể khởi động. Đây là giải pháp có thể giúp giải quyết được vấn đề đã đặt ra ban đầu.

Quá trình nghiên cứu và thực hiện trải qua 4 giai đoạn:

#### ***Giai đoạn 1: Tìm hiểu sơ lược về Docker và Kubernetes.***

Docker là công nghệ giúp đóng gói ứng dụng cùng toàn bộ thư viện và nền tảng thực thi lại thành một gói duy nhất có thể đem chạy ở bất kì đâu, bất kì môi trường nào.

Docker là một nền tảng mở để triển khai, di chuyển và chạy các ứng dụng. Với Docker, chúng ta có thể quản lý cơ sở hạ tầng của mình giống như quản lý các ứng dụng của mình. Bằng cách tận dụng các phương pháp luận của Docker để vận chuyển, kiểm thử và triển khai code một cách nhanh chóng, có thể giảm đáng kể độ trễ giữa việc viết code và chạy code.

#### ***Kubernetes có thể làm những gì?***

Với Kubernetes, chúng ta có thể đóng gói và chạy các ứng dụng. Sau đó việc chúng ta cần làm là quản lý các Container chạy các ứng dụng và đảm bảo rằng không có thời gian chết. Nếu có một Container gặp sự cố không thể khởi động lại, một Container khác cần được khởi động lên để thay thế.

Kubernetes cho phép chúng ta chạy các ứng dụng phần mềm của mình trên hàng nghìn nút máy tính như thể tất cả các nút đó là một máy tính khổng lồ duy nhất. Nó loại bỏ cơ sở hạ tầng cơ bản và bằng cách đó, đơn giản hóa việc phát triển, triển khai và quản lý cho cả nhóm phát triển và hoạt động.

### **Kubernetes cung cấp:**

- **Service discovery và cân bằng tải:** Trong Kubernetes, Container có thể sử dụng DNS hoặc địa chỉ IP của riêng nó. Nếu lưu lượng truy cập đến một Container tăng cao, Kubernetes có thể cân bằng tải và phân phối lưu lượng mạng để việc triển khai được ổn định hơn.
- **Điều phối bộ nhớ:** Kubernetes cho phép tự động gắn kết (mount) một hệ thống lưu trữ mà chúng ta chọn như local storages, public cloud providers, v.v..
- **Tự động rollouts và rollbacks:** Chúng ta có thể mô tả trạng thái mong muốn cho các Container được triển khai và nó có thể thay đổi trạng thái thực tế sang trạng thái mong muốn với tần suất được kiểm soát. Ví dụ, bạn có thể tự động hóa Kubernetes để tạo mới các Container cho việc triển khai của bạn, xoá các Container hiện có và áp dụng tất cả các resource của chúng vào Container mới.
- **Đóng gói tự động:** Lập trình viên cung cấp cho Kubernetes một Cluster gồm các Node mà nó có thể sử dụng để chạy các tác vụ được đóng gói (Containerized Task). Lập trình viên cho Kubernetes biết mỗi Container cần bao nhiêu CPU và bộ nhớ (RAM). Kubernetes có thể điều phối các Container đến các Node để tận dụng tốt nhất các Resource.
- **Tự phục hồi:** Kubernetes khởi động lại các Containers bị lỗi, thay thế các Container, xoá các Container không phản hồi lại cấu hình health check do người dùng xác định và không cho các client biết đến chúng cho đến khi chúng sẵn sàng hoạt động.
- **Quản lý cấu hình và bảo mật:** Kubernetes cho phép chúng ta lưu trữ và quản lý các thông tin nhạy cảm như: password, OAuth Token và SSH Key. Chúng ta có thể triển khai và cập nhật lại Secret và cấu hình ứng dụng mà không cần build lại các Container Image và không để lộ Secret trong cấu hình Stack.

### ***Giai đoạn 2: Tiến hành cài đặt Docker và Kubernetes.***

Ở giai đoạn này, tiến hành tạo 3 máy ảo Ubuntu, cấu hình mạng trên 3 máy. Cài đặt và cấu hình sau khi đã tìm hiểu những khái niệm căn bản về Docker và Kubernetes. Hệ điều hành mà em lựa chọn là Ubuntu 20.04.

### **Chi tiết máy ảo:**

- Máy Master: IP là 192.168.1.100, 30GB bộ nhớ, RAM 4GB, 2 CPU.
- Máy Worker 1: IP là 192.168.1.101, 30GB bộ nhớ, RAM 3GB, 2 CPU.
- Máy Worker 2: IP là 192.168.1.102, 30GB bộ nhớ, RAM 3GB, 2 CPU.

### ***Giai đoạn 3: Triển khai CouchDB Cluster lên Kubernetes, cài đặt scale tự động CouchDB.***

Tìm hiểu các phiên bản CouchDB được Docker hỗ trợ sau đó tạo Container trên Kubernetes và tiến hành cấu hình Cluster cho CouchDB. Cài đặt scale tự động khi có Container bị lỗi hoặc lưu lượng truy cập quá lớn khiến Server bị quá tải.

### ***Giai đoạn 4: Triển khai ứng dụng Web lên Kubernetes, cài đặt scale tự động ứng dụng Web.***

Ở giai đoạn này, đầu tiên cần biết cách tạo Docker Image và cách đẩy Image lên Docker Hub. Tiến hành tạo Docker Image cho ứng dụng Web. Sau đó đẩy Image đã tạo lên Docker Hub và tiến hành tạo Container trên Kubernetes. Cuối cùng là cài đặt scale tự động khi có Container bị lỗi hoặc lưu lượng truy cập quá lớn khiến Server bị quá tải.

## CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

### 1. DOCKER

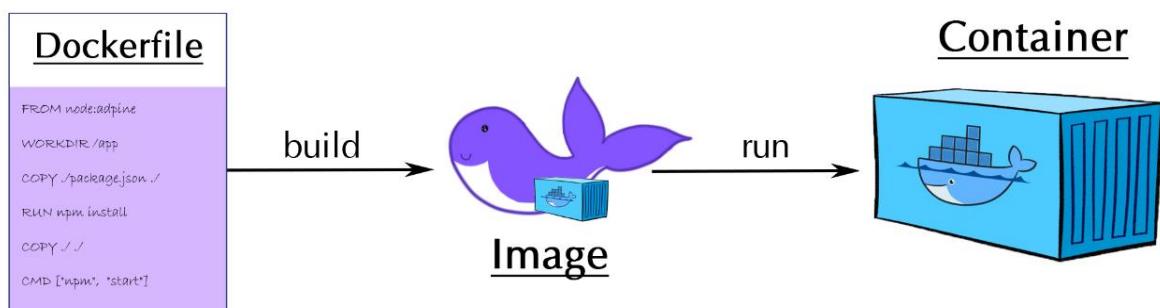
#### 1.1. Lý do Docker ra đời?

Khi phát triển ứng dụng bằng bất kì ngôn ngữ nào chúng ta đều phải cài đặt những phần mềm đi kèm theo nó bao gồm các thư viện, nền tảng thực thi. Ví dụ như khi lập trình Java thì chúng ta phải cài đặt JDK, Maven, khi lập trình Angular thì phải cài đặt Node.js, lập trình C# thì phải cài đặt .NET Core hoặc .NET Framework.

Vấn đề xảy ra khi ta triển khai ứng dụng lên một máy khác ở một môi trường khác thì không phải bất kì ngôn ngữ nào cũng có thể build ra file thực thi chạy trực tiếp bằng ngôn ngữ máy. Do đó khả năng cao là chúng ta phải cài lại từ đầu những phần mềm hỗ trợ này và phải cài đúng Version của Project mà ta đang làm. Vấn đề đó sẽ càng phức tạp hơn nếu phải cài đặt trên nhiều máy khác nhau thay vì một máy. Do đó mà việc triển khai ứng dụng khó khăn hơn và kém hiệu quả. Đó là lý do mà Docker ra đời, Docker mang đến giải pháp cho vấn đề này.

#### 1.2. Docker là gì?

Docker là công nghệ giúp đóng gói ứng dụng cùng toàn bộ thư viện và nền tảng thực thi lại thành một gói duy nhất có thể chạy ở bất kì đâu, bất kì môi trường nào cho dù là Linux, MacOS hay là Windows, việc duy nhất chúng ta cần làm là cài đặt Docker.



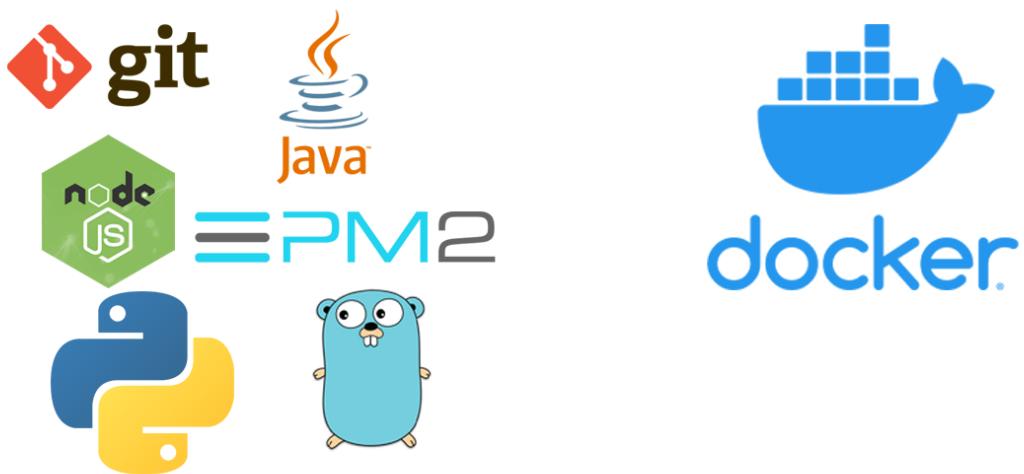
Hình 1: Docker

#### 1.3. Tại sao chúng ta cần Docker?

Như đã nói ở trên, việc triển khai ứng dụng sẽ dễ dàng hơn rất nhiều với Docker. Chúng ta không cần phải nhớ hệ thống nào chạy bằng ngôn ngữ gì, phải cài đặt những thư viện gì, nền tảng gì, phiên bản bao nhiêu. Chúng ta chỉ cần cài Docker và mọi thứ sẽ được giải quyết.

# Cài Đặt Thủ Công

# Với Docker



*Hình 2: Tại sao chúng ta cần Docker?*

Ngoài ra khi nhắc đến Docker thì không thể không nhắc đến Microservices. Docker có 3 tính chất sau:

- Immutable (bất biến): khi đóng gói ứng dụng và chạy ở bất kì đâu thì nó sẽ hoạt động giống như khi chạy trên máy local, không thay đổi.
- Lightweight (nhẹ): vì chỉ khoảng vài chục Mega Bytes, nhẹ hơn máy ảo rất nhiều nên việc tạo Container rất nhanh chóng.
- Stateless: Container không tạo ra thay đổi dữ liệu bên trong nó, điều này là do tính bất biến của nó. Có thể hiểu rằng vì không chứa dữ liệu bên trong nên các Container cực kỳ tự do và linh hoạt. Chúng ta có thể nhân bản các Container lên 100 để đáp ứng đủ nhu cầu sử dụng hoặc có thể shutdown nó ở máy này và build nó lên ở máy khác mà vẫn chẳng ảnh hưởng đến hệ thống ban đầu.

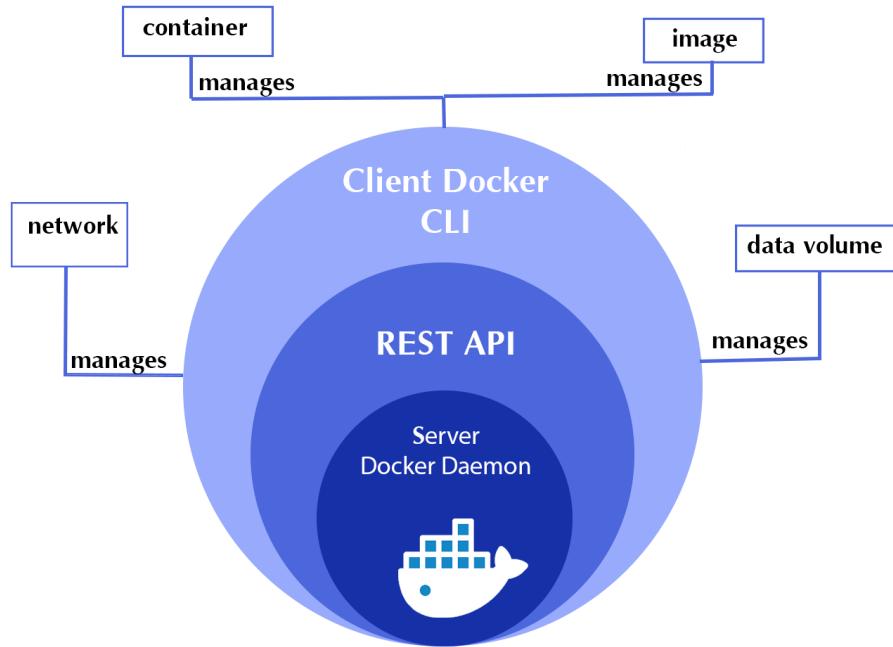
Đây chính là lý do khiến Docker hoàn hảo trong việc triển khai Microservices.

## 1.4.Các khái niệm cơ bản

### 1.4.1. Docker Engine

Docker Engine bao gồm 2 module chính:

- **Docker Daemon** và **Exposed APIs**: là module cốt lõi đóng vai trò là Server chạy ngầm ở bên dưới hệ thống.
- **Docker Client (cli)**: đóng vai trò là Client. Chúng ta dùng bộ câu lệnh này để giao tiếp với Docker Daemon thông qua giao thức HTTPS.



*Hình 3: Docker Engine*

Các Container sử dụng chung Kernel với hệ điều hành gốc được cài ở trên máy chủ vật lý nên chúng bị phụ thuộc vào hệ điều hành đó. Hệ điều hành Windows chỉ chạy được Container dành cho Windows như là .NET CORE, .NET Framework. Do bị giới hạn như vậy nên mỗi lần cài lên Windows, Docker sẽ yêu cầu người dùng kích hoạt Hypervisor hoặc Virtual Box và cài lên đó một máy ảo Linux. Lúc này Docker Daemon sẽ được cài lên máy ảo Linux đó, còn Docker CLI vẫn được cài trên máy Windows của người dùng.

#### 1.4.2. Image

Docker đóng gói toàn bộ ứng dụng cùng thư viện và nền tảng thực thi của chúng thành một package duy nhất. Những package mà Docker tạo ra chính là Images.

#### 1.4.3. Container

Khi chạy một Image, chúng ta có Container. Về cơ bản, Container hoạt động chính xác như một máy ảo với đầy đủ tính năng để cài đặt và chạy bất cứ phần mềm nào bên trong nó như là file system, network interface, cây tiến trình..

Tuy nhiên, Container không phải là máy ảo. Sự khác biệt lớn nhất là bởi vì Container không có nhân hệ điều hành (OS kernel) của riêng nó mà tất cả các Container đều dùng chung bộ nhân của máy chủ vật lý. Container chỉ đơn thuần là các tiến trình chạy trên hệ điều hành và được Docker quản lý.

**\* *Phân biệt Virtual Machine (máy ảo) với Container:***

Máy ảo sử dụng công nghệ ảo hóa ở Hardware Abstraction Layer (HAL) – lớp trừu tượng phần cứng, máy chủ vật lý sẽ cài đặt một phần mềm ảo hóa chuyên dụng như là Vmware, VirtualBox hoặc là Hypervisor. Chúng có nhiệm vụ phân chia tài nguyên vật lý cho các máy ảo được cài bên trên. Mỗi máy ảo có một hệ điều hành độc lập ở bên trong nó, dung lượng có thể lên đến hàng chục Gigabytes, chúng sử dụng phần nhân (kernel) của riêng nó, do đó chúng ta có thể cài đặt máy ảo Ubuntu chạy trên hệ điều hành Windows hoặc MacOS chạy trên hệ điều hành Windows mà không gặp bất cứ vấn đề gì.

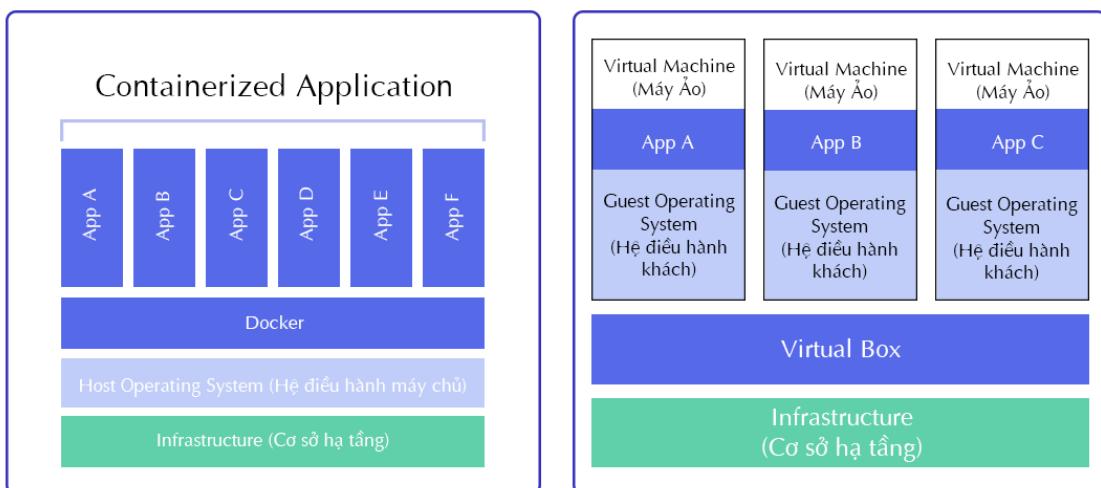
Container sử dụng công nghệ ảo hóa ở Software Abstraction Layer (SAL) – lớp trừu tượng phần mềm, bản thân Docker chính là công nghệ ảo hóa đó. Sự khác biệt so với Virtual Machine đó là các Container tận dụng lại phần nhân (kernel) của hệ điều hành máy chủ vật lý. Do đó không tốn dung lượng để cài thêm hệ điều hành, việc đó dẫn đến dung lượng của Container chỉ khoảng vài chục Megabytes nên cực kì nhẹ. Nhưng đổi lại nó không có sự tự do như Virtual Machine. Chúng ta không thể chạy Container Ubuntu trên hệ điều hành Windows được. Dó đó khi cài Docker lên Windows, chúng ta cần phải kích hoạt Hypervisor hoặc Virtual Box để setup một máy ảo Linux trên đó để cài Docker chạy bên trong. Từ đó chúng ta mới có thể chạy được Container dòng Linux trên hệ điều hành Windows.

### Container

Software Abstraction Layer (SAL) - lớp trùu tượng phần mềm, chia sẻ cùng một nhân hệ điều hành Nhẹ (Mega Bytes)

### Virtual Machine (Máy Ảo)

Hardware Abstraction Layer (HAL) – lớp trùu tượng phần cứng, chạy với hệ điều hành riêng Nặng (Giga Bytes)



*Hình 4: Phân biệt Container với Virtual Machine*

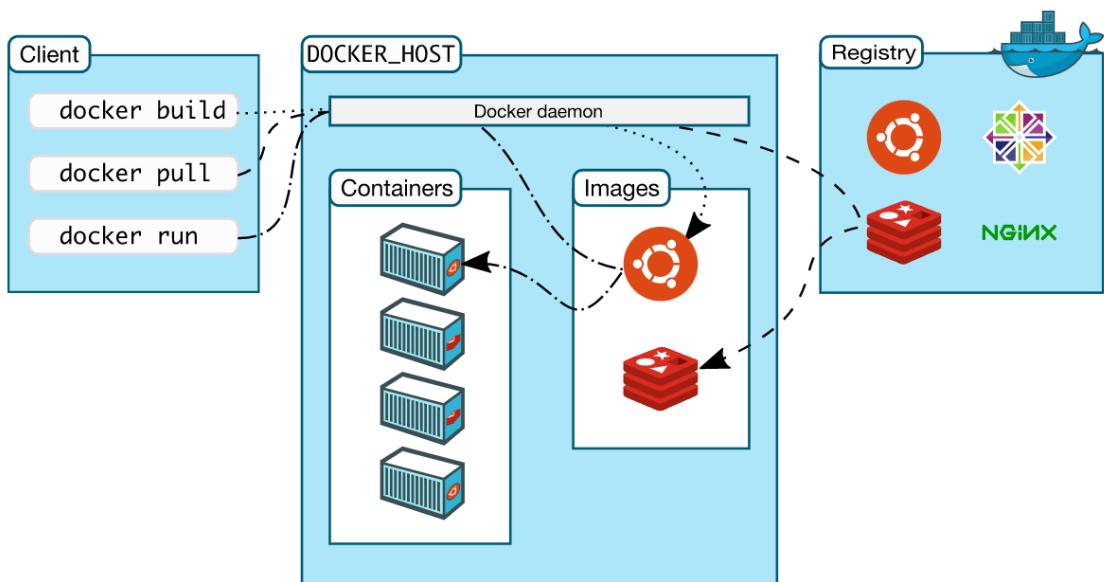
Docker được viết bằng ngôn ngữ Golang và Docker tận dụng những tính năng được cung cấp bởi nhân Linux như là Namespace để phân chia, tách biệt các Container với nhau, Control Group để tối ưu hóa tài nguyên phần cứng, .v.v..

#### **1.4.4. Registry**

Docker đóng gói các ứng dụng thành các Images, vậy làm cách nào để chúng ta tải những Images này lên mạng và chia sẻ đến các máy khác. Nơi dùng để lưu trữ các Images này chúng ta gọi là Container Registry.

Hiện nay trên thị trường có những nhà cung cấp dịch vụ lưu trữ Image khác nhau như:

- Docker Hub
- Amazon Elastic Container Registry (ECR)
- Google Container Registry (GCR)
- Azure Container Registry (ACR)



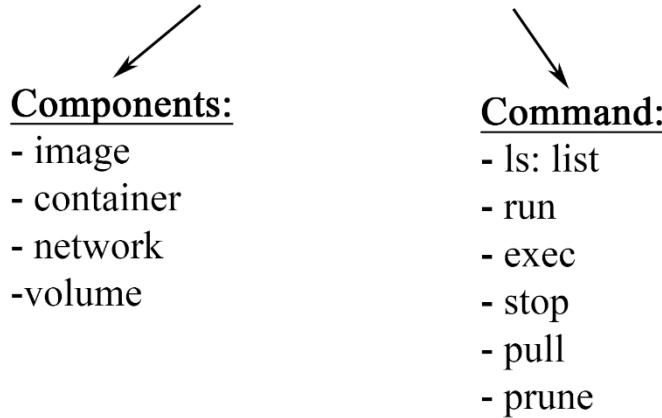
*Hình 5: Sơ đồ tổng quan*

Client là bộ câu lệnh CLI chúng ta dùng để tương tác với Docker Daemon. Các Images sẽ được lưu trữ trên Docker Registry và được tải, lưu (pull) về máy Local. Khi chúng ta chạy những Images đó trên máy Local chúng ta sẽ có Container.

### 1.5.Một số lệnh cơ bản

\*Cú pháp chung của một câu lệnh Docker:

```
docker <component> <command>
```



#### \*Các câu lệnh về Image:

Lệnh pull: Lệnh pull là lệnh dùng để tải các Image từ Registry về. Mỗi Image sẽ có nhiều <tag> khác nhau dùng để đánh Version của Image đó, nếu không để <tag> thì mặc định sẽ là “latest”.

```
$ docker image pull <image>
$ docker image pull <image>:<tag>
```

Lệnh push: Ngược lại với lệnh pull, dùng để tải Image đã build ở máy Local lên Registry.

```
$ docker image push <image>:<tag>
```

Lệnh ls: liệt kê ra toàn bộ Image đang có trong Local.

```
$ docker image ls | docker images
```

Lệnh prune: xóa những Image không còn được dùng nữa.

```
$ docker image prune
```

Một số lệnh có thể viết tắt như lệnh pull hoặc lệnh push vì những lệnh này đặc trưng dành riêng cho đối tượng Image nên chúng ta không sợ bị nhầm lẫn với các đối tượng khác.

```
$ docker pull <image>:<tag>
$ docker push <image>:<tag>
```

#### \*Các câu lệnh về Container:

Lệnh run: dùng để chạy một Container.

```
$ docker container run <image>
```

Lệnh ls: liệt kê ra các Container đang chạy.

```
$ docker container ls | docker container ls  
-a
```

```
$ docker ps | docker ps -a
```

Lệnh stop: dùng để dừng Container đang chạy.

```
$ docker container stop <container_id>
```

Lệnh prune: xóa toàn bộ Container đã Shut Down không còn được dùng nữa.

```
$ docker container prune
```

Lệnh exec: là một lệnh đặc biệt, dùng để chạy một câu lệnh ở bên trong một Container bất kì.

```
$ docker container exec <container_id>  
<command>
```

Một số lệnh có thể viết tắt như lệnh run, lệnh stop, lệnh exec vì những lệnh này đặc trưng cho đối tượng Container.

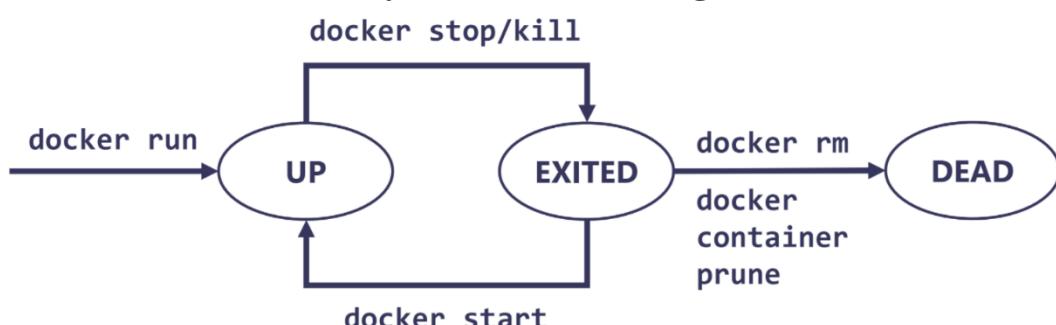
```
$ docker run
```

```
$ docker stop
```

```
$ docker exec
```

## 1.6.Các khái niệm cốt lõi của Container

### 1.6.1. Container Life Cycle (Chu trình sống của Container)



Hình 6: Container Life Cycle

Khi chạy một Image, ta có Container, Container lúc này ở trạng thái sống là “Up”, khi stop/kill thì Container ở trạng thái kết thúc là “Exited”, khi dùng lệnh start để chạy lại Container thì nó sẽ sống lại vài ở trạng thái “Up”. Khi một Container ở trạng thái “Exited” mà ta dùng lệnh remove hoặc prune thì nó sẽ rơi vào trạng thái “Dead”, lúc này Docker sẽ xóa sạch để giải phóng bộ nhớ.

Chu trình sống của Container được quyết định bởi tiến trình có PID = 1 (tiến trình chủ đạo- main process) chạy ở bên trong nó. Container sẽ còn sống khi PID 1 còn sống.

Khi dùng lệnh “docker stop”, Docker sẽ gửi một tín hiệu SIGTERM (Terminating Signal) đến để kết thúc tiến trình PID 1 ở bên trong Container, sau đó Container sẽ kết thúc với EXITED (0).

Trong vòng 10 giây sau khi chạy lệnh “docker stop”, nếu vì một lỗi nào đó mà Container chưa dừng lại thì Container sẽ gửi đi tín hiệu thứ 2 là SIGKILL (kill signal) để ép buộc tiến trình PID 1 phải dừng ngay lập tức với EXITED (137).

### 1.6.2. Thực thi câu lệnh bên trong Container

*Cú pháp:*

```
$ docker exec <container_id> <command>
```

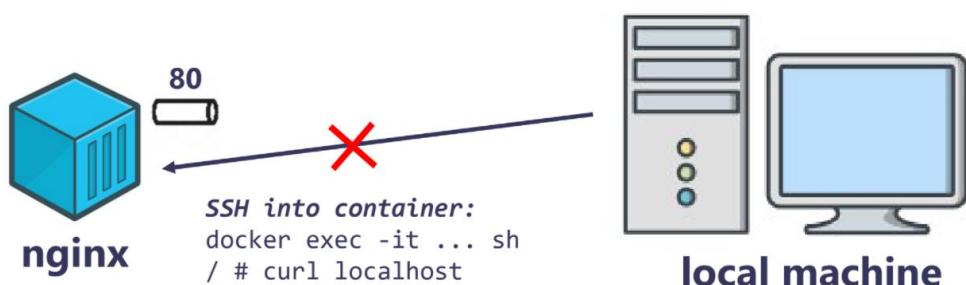
*Ví dụ:*

- ✓ docker exec <container\_id> echo Hello World
- ✗ docker exec <container\_id> echo \$PATH
- ✓ docker exec <container\_id> sh -c "echo \$PATH"
- ✓ docker exec <container\_id> cat /etc/os-release

Để SSH vào bên trong bất kỳ Container nào, sử dụng “exec shell” hoặc bash với cờ -it

```
$ docker exec -it <container_id> sh  
$ docker exec -it <container_id> bash
```

### 1.6.3. Port Mapping



Hình 7: SSH vào trong Container

Trong thực tế, khi triển khai một ứng dụng lên Web chúng ta không thể nào để người dùng SSH trực tiếp vào bên trong Container vì người dùng không có kiến thức chuyên môn và việc SSH vào Container như vậy sẽ kém bảo mật. Do đó chúng ta sẽ cần dùng đến Port Mapping.

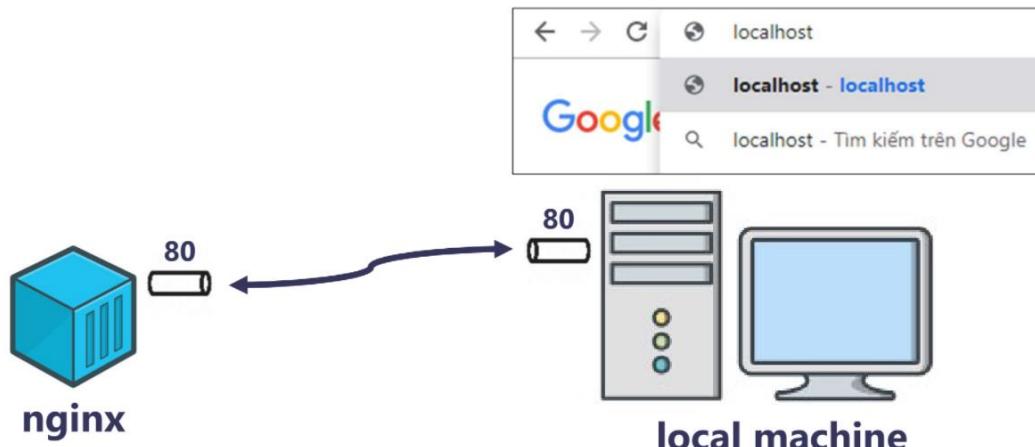
Chúng ta sẽ mở port 80 ở trên máy Local của chúng ta và kết nối với port 80 ở bên trong Container ra bên ngoài port 80 ở máy Local của chúng ta. Lúc này khi hai port đã kết nối với nhau thì ở máy Local chúng ta chỉ cần mở trình duyệt và truy cập trực tiếp vào localhost ở port 80.

### Cú pháp:

```
$ docker run -p  
<target_port>:<container_port> ...  
☞ <target_port>: port ở máy Local.  
☞ <container_port>: port ở trong Container.
```

### Ví dụ:

```
$ docker run -p 3000:3000 react-app
```



Hình 8: Port Mapping

#### 1.6.4. Lưu trữ dữ liệu bên trong Container - Bind Mount Volume

- “Image is immutable. Container is stateless.”

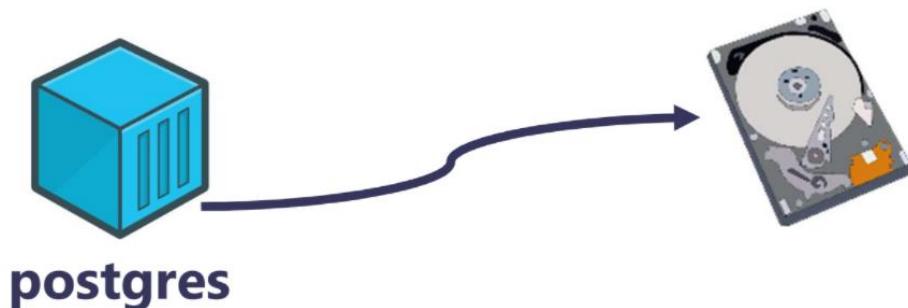
Các Image có tính bất biến, không thể chỉnh sửa được. Do vậy khi chạy một Container bất kỳ, cho dù chỉnh sửa dữ liệu, nội dung bên trong của Container đó như thế nào đi nữa thì nó cũng chẳng ảnh hưởng đến Image gốc ban đầu và nó cũng chẳng ảnh hưởng đến Container khác. Việc này dẫn đến một vấn đề đó là dữ liệu bên trong Container không được bảo toàn. Chỉ cần tắt Container đi thì mọi dữ liệu sẽ mất hết.

**Vậy làm cách nào để giữ lại dữ liệu bên trong Container !?**

=> Sử dụng Volume.

### ❖ Volume:

Volume là phần bộ nhớ mà Docker dùng để lưu trữ dữ liệu cho Container. Nói một cách dễ hiểu thì Volume là một thư mục ảo do Docker tạo ra và quản lý. Việc mà chúng ta cần làm là lấy Volume đó gắn vào Container. Thao tác gắn đó được gọi là “Bind Mount”



Hình 9: Volume

**Cú pháp:**

```
$ Docker volume create [volume_name]  
$ docker run -v [local_dir/volume]:  
[container_dir] ...
```

**Ví dụ:**

```
$ docker volume create pgdata  
$ docker run -v  
pgdata:/var/lib/postgresql/data -p  
5432:5432 postgres  
$ docker run -v  
/usr/data:/var/lib/postgresql/data -p  
5432:5432 postgres
```

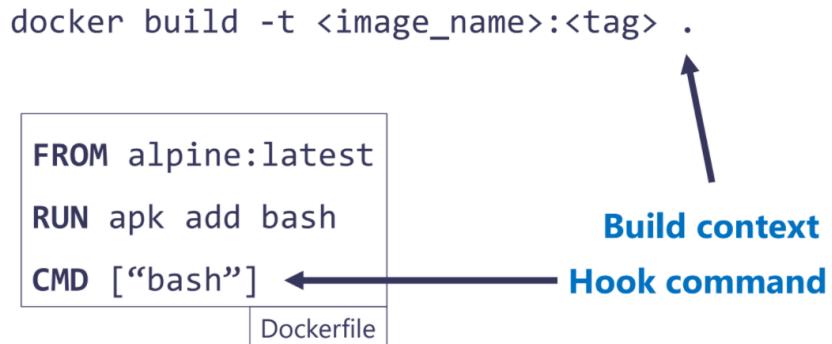
## 1.7.Cách tạo Docker Image

**Dockerfile:**

Dockerfile là một file template giúp chúng ta xây dựng Image riêng của riêng mình.

**Cú pháp:**

```
$ docker build -t <image_name>:<tag> .
```



### **Build Context :**

Build Context đề cập đến thư mục chứa Dockerfile. Toàn bộ nội dung bên trong thư mục chứa Dockerfile đó được gọi là “Build Context” bao gồm cả Dockerfile.

### **.dockerignore:**

.dockerignore sẽ bỏ qua những file và folder được chỉ định ở bên trong Build Context trước khi gửi nó đến Docker Daemon.

## **2. KUBERNETES**

### **2.1. Lý do Kubernetes ra đời?**

Khi số lượng các thành phần ứng dụng có thể triển khai trong hệ thống của chúng ta ngày càng tăng, việc quản lý tất cả chúng trở nên khó khăn hơn. Google có lẽ là công ty đầu tiên nhận ra rằng họ cần một cách tốt hơn nhiều để triển khai và quản lý các thành phần phần mềm cũng như cơ sở hạ tầng của họ để mở rộng quy mô toàn cầu. Đây là một trong số ít công ty trên thế giới vận hành hàng trăm nghìn máy chủ và đã phải đổi mới với việc quản lý triển khai trên quy mô lớn như vậy. Điều này đã buộc họ phải phát triển các giải pháp để làm cho việc phát triển và triển khai hàng nghìn thành phần phần mềm có thể quản lý được và tiết kiệm chi phí.

Qua nhiều năm, Google đã phát triển một hệ thống nội bộ gọi là Borg (và sau đó là một hệ thống mới có tên là Omega), giúp cả nhà phát triển ứng dụng và quản trị viên hệ thống quản lý hàng nghìn ứng dụng và dịch vụ đó. Ngoài việc đơn giản hóa việc phát triển và quản lý, nó cũng giúp họ đạt được hiệu quả sử dụng cơ sở hạ tầng cao hơn nhiều, điều này rất quan trọng khi tổ chức của họ lớn như vậy. Khi chúng ta chạy hàng trăm nghìn máy, ngay cả những cải tiến nhỏ trong việc sử dụng cũng đồng nghĩa với việc tiết kiệm hàng

triệu đô la, do đó, các động lực để phát triển một hệ thống như vậy là rất rõ ràng.

Sau khi giữ bí mật về Borg và Omega trong cả thập kỷ, vào năm 2014, Google đã giới thiệu Kubernetes, một hệ thống mã nguồn mở dựa trên kinh nghiệm thu được từ Borg, Omega và các hệ thống nội bộ khác của Google.

## 2.2. Kubernetes là gì?

Kubernetes là một hệ thống phần mềm cho phép chúng ta dễ dàng triển khai và quản lý các ứng dụng được đóng gói trên nó. Nó dựa vào các tính năng của Linux Container để chạy các ứng dụng không đồng nhất mà không cần biết bất kỳ chi tiết nội bộ nào của cùng một máy chủ này, điều này rất quan trọng khi chúng ta chạy các ứng dụng cho các tổ chức hoàn toàn khác nhau trên cùng một phần cứng. Đây là điều quan trọng đối với các nhà cung cấp dịch vụ đám mây, vì họ cố gắng sử dụng phần cứng của mình một cách tốt nhất có thể trong khi vẫn phải duy trì sự cô lập hoàn toàn đối với các ứng dụng được lưu trữ.

Kubernetes cho phép chúng ta chạy các ứng dụng phần mềm của mình trên hàng nghìn nút máy tính như thể tất cả các nút đó là một máy tính không lồ duy nhất. Nó loại bỏ cơ sở hạ tầng cơ bản và bằng cách đó, đơn giản hóa việc phát triển, triển khai và quản lý cho cả nhóm phát triển và hoạt động.

Việc triển khai các ứng dụng thông qua Kubernetes luôn giống nhau, cho dù cụm của chúng ta chỉ chứa một vài hoặc hàng nghìn nút trong số đó. Kích thước của cụm không có sự khác biệt nào cả. Các Cluster Node bổ sung chỉ đơn giản là đại diện cho một lượng tài nguyên bổ sung có sẵn cho các ứng dụng được triển khai.

Với việc ngày càng nhiều công ty lớn chấp nhận mô hình Kubernetes là cách tốt nhất để chạy ứng dụng, nó đang trở thành cách tiêu chuẩn để chạy các ứng dụng phân tán cả trên đám mây cũng như trên máy cục bộ.

## 2.3. Microservices

### 2.3.1. Chuyển từ Monolithic Application sang Microservices.

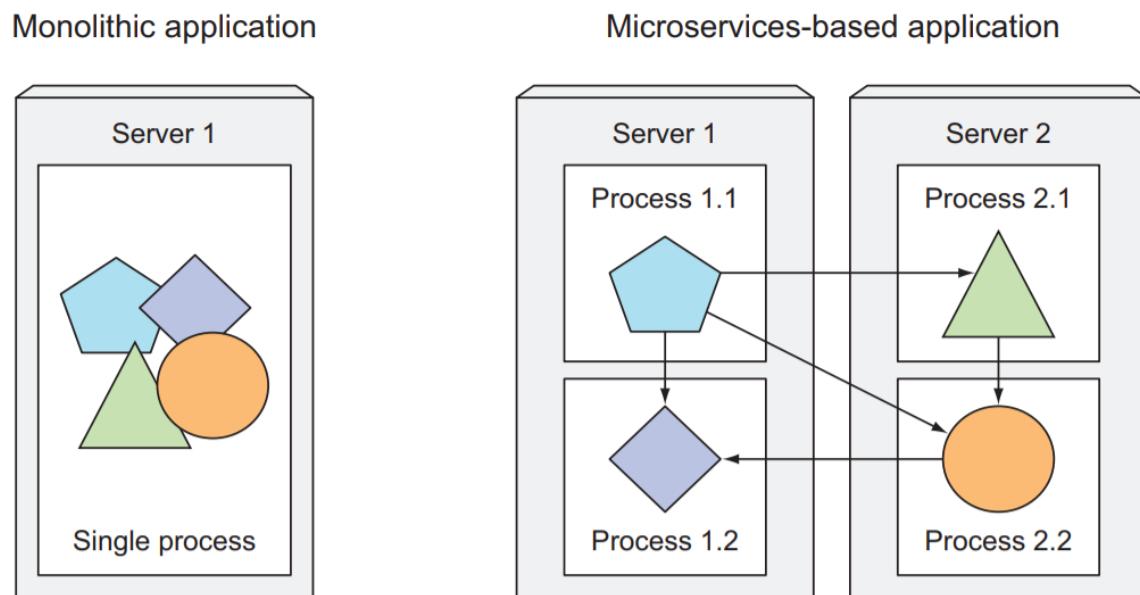
Các Monolithic Application bao gồm các thành phần được kết hợp chặt chẽ với nhau và phải được phát triển, triển khai và quản lý như một thực thể, bởi vì tất cả chúng đều chạy như một tiến trình hệ điều hành duy nhất. Khi cần thay đổi một phần của ứng dụng thì phải triển khai lại toàn bộ ứng dụng và theo thời gian, đã dẫn đến sự gia tăng độ phức tạp và hậu quả là chất lượng của toàn hệ thống bị giảm sút do sự phát triển không bị hạn chế. Việc

chạy một Monolithic Application thường yêu cầu một số lượng nhỏ các máy chủ mạnh có thể cung cấp đủ tài nguyên để chạy ứng dụng.

Để đối phó với việc lượt load hệ thống tăng, chúng ta phải mở rộng quy mô máy chủ theo chiều dọc (scaling up) bằng cách thêm nhiều CPU, bộ nhớ và các thành phần máy chủ khác hoặc mở rộng toàn bộ hệ thống theo chiều ngang, bằng cách thiết lập các máy chủ bổ sung và chạy nhiều bản sao (replicas) của một ứng dụng (scaling out). Mặc dù việc scaling up thường không yêu cầu bất kỳ thay đổi nào đối với ứng dụng, nhưng trên thực tế luôn có giới hạn trên. Mặt khác, mở rộng quy mô tương đối rẻ, nhưng có thể yêu cầu những thay đổi lớn trong code ứng dụng và không phải lúc nào cũng có thể thực hiện được, một số phần của ứng dụng cực kỳ khó hoặc gần như không thể mở rộng theo chiều ngang (quan hệ cơ sở dữ liệu chẳng hạn).

### 2.3.2. Chia nhỏ App thành các Microservice

Những vấn đề này đã buộc chúng ta phải bắt đầu chia các Monolithic Application phức tạp thành các thành phần nhỏ hơn có thể triển khai độc lập được gọi là Microservices. Mỗi Microservice chạy như một quá trình độc lập (xem hình dưới) và giao tiếp với các Microservice khác thông qua các giao diện (API) đơn giản, được xác định rõ ràng.



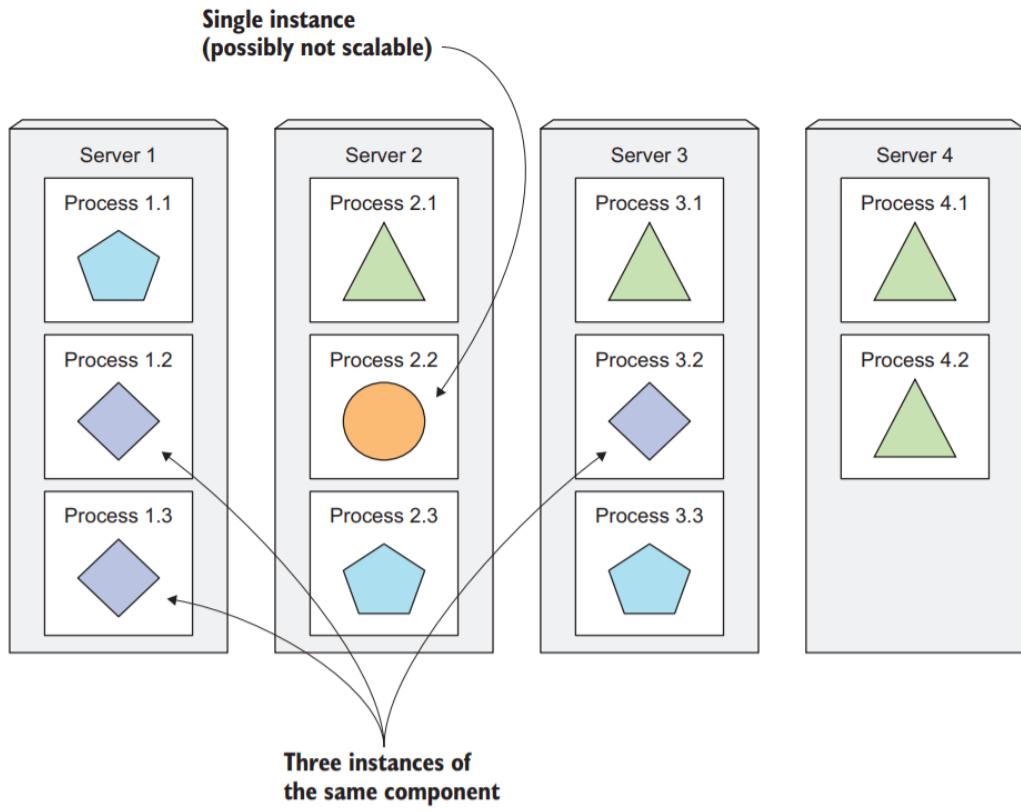
Hình 10: Các thành phần bên trong Monolithic Application và Microservices

Microservices giao tiếp thông qua các giao thức đồng bộ như HTTP, qua đó chúng thường hiển thị các API RESTful (REpresentational State Transfer) hoặc thông qua các giao thức không đồng bộ như AMQP (Giao thức xếp hàng tin nhắn nâng cao). Các giao thức này rất đơn giản, được hầu hết các nhà phát triển hiểu rõ và không bị ràng buộc với bất kỳ ngôn ngữ lập trình cụ thể nào. Mỗi Microservice có thể được viết bằng ngôn ngữ thích hợp nhất để triển khai Microservice cụ thể đó.

Vì mỗi Microservice là một tiến trình độc lập với “relatively static external API”, nên có thể phát triển và triển khai từng Microservice riêng biệt. Thay đổi đối với một trong số chúng không yêu cầu thay đổi hoặc triển khai lại bất kỳ Service nào khác, miễn là API không thay đổi hoặc chỉ thay đổi theo cách backward-compatible.

### 2.3.3. Scaling Microservices

Việc mở rộng quy mô microservices, không giống như các hệ thống nguyên khôi, nơi bạn cần mở rộng quy mô toàn hệ thống, được thực hiện trên cơ sở từng dịch vụ, có nghĩa là bạn có tùy chọn chỉ mở rộng những dịch vụ yêu cầu nhiều tài nguyên hơn, trong khi vẫn để những dịch vụ khác ở quy mô ban đầu. Hình 1.2 cho thấy một ví dụ. Một số thành phần nhất định được sao chép và chạy dưới dạng nhiều quy trình được triển khai trên các máy chủ khác nhau, trong khi các thành phần khác chạy như một quy trình ứng dụng duy nhất. Khi một ứng dụng nguyên khôi không thể thu nhỏ vì một trong các phần của nó không thể thay đổi tỷ lệ, việc chia ứng dụng thành các microservices cho phép bạn chia tỷ lệ các phần cho phép thu nhỏ theo chiều ngang và chia tỷ lệ các phần không có, theo chiều dọc thay vì chiều ngang.

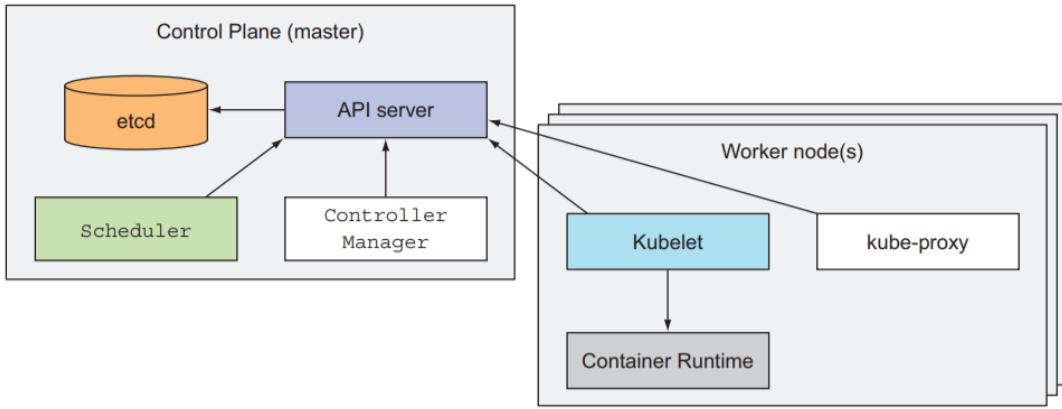


*Hình 11: Microservices*

## 2.4. Kiến trúc của một cụm Kubernetes

Ở cấp độ phần cứng, một cụm Kubernetes bao gồm nhiều node, có thể được chia thành hai loại:

- **Node Master:** máy chủ Kubernetes Control Plane điều khiển và quản lý toàn bộ hệ thống Kubernetes.
- **Các Node Worker:** chạy các ứng dụng thực tế mà chúng ta triển khai.



*Hình 12: Kiến trúc của một cụm Kubernetes*

#### 2.4.1. Control Plane (Master Node)

Control Plane là thứ điều khiển Cluster và làm cho nó hoạt động. Nó bao gồm nhiều thành phần có thể chạy trên một Node chính duy nhất hoặc được chia thành nhiều Node và được nhân rộng để đảm bảo tính sẵn sàng cao. Các thành phần này là:

- Kubernetes API Server: là thành phần giúp các thành phần khác liên lạc với nhau. Lập trình viên khi triển khai ứng dụng sẽ gọi API Kubernetes API Server này
- Scheduler: Thành phần này lập lịch triển khai cho các ứng dụng, ứng dụng được đặt vào Worker Node nào để chạy
- Controller Manager: Thành phần đảm nhiệm quản lý các Worker Nodes, kiểm tra trạng thái các Worker “sống hay chết”, sao chép các thành phần, theo dõi các Node Worker, xử lý lỗi Node, đảm nhận việc nhân bản ứng dụng, v.v..
- etcd: là cơ sở dữ liệu của Kubernetes, tất cả các thông tin của Kubernetes được lưu trữ cố định vào đây.

Các thành phần của Control Plane giữ và kiểm soát trạng thái của Cluster, nhưng chúng không chạy các ứng dụng. Điều này được thực hiện bởi các Node (Worker).

#### 2.4.2. Worker Node

Các Worker Node là Server chạy các Pod chứa ứng dụng trên đó. Tác vụ chạy, giám sát và cung cấp dịch vụ cho các ứng dụng của chúng ta được thực hiện bởi các thành phần sau:

- Container Runtime: là thành phần giúp chạy các ứng dụng dưới dạng Container. Kubernetes hỗ trợ nhiều loại Container Runtime như Docker, containerd, CRI-O...
- Kubelet: đây là thành phần giao tiếp với Kubernetes API Server và quản lý các Container trên Node của nó.
- Kubernetes Service Proxy (kube-proxy): cân bằng tải lưu lượng mạng giữa các thành phần ứng dụng.

## 2.5. Chạy một ứng dụng trong Kubernetes

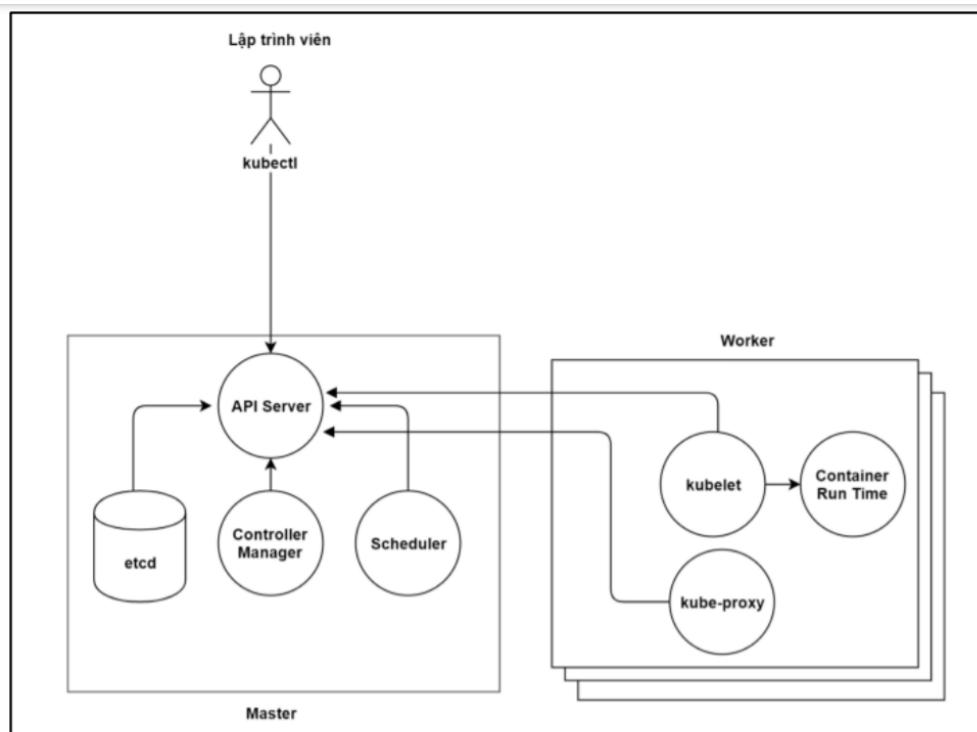
Để chạy một ứng dụng trong Kubernetes, trước tiên cần phải đóng gói nó thành một hoặc nhiều Image, push những Image đó vào Registry, sau đó đăng mô tả về ứng dụng của chúng ta lên máy chủ Kubernetes API.

Mô tả bao gồm thông tin như Images Container hoặc các Images chứa các thành phần ứng dụng của chúng ta, cách các thành phần đó liên quan với nhau và những thành phần nào cần được chạy cùng vị trí (cùng nhau trên cùng một Node) và những thành phần nào không. Đối với mỗi thành phần, chúng ta cũng có thể chỉ định số lượng bản sao (replica) chúng ta muốn chạy. Ngoài ra, mô tả cũng bao gồm thành phần nào trong số các thành phần đó cung cấp dịch vụ cho Client nội bộ hoặc bên ngoài và phải được hiển thị thông qua một địa chỉ IP duy nhất và có thể phát hiện được đối với các thành phần khác.

## 2.6. Các khái niệm cơ bản

### 2.6.1. Kubectl

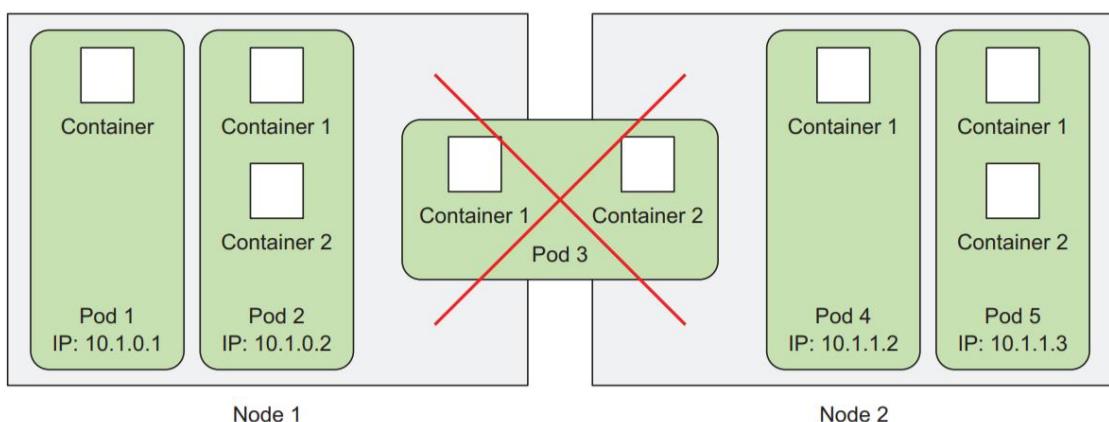
Là agent chạy trên từng worker node trong Kubernetes Cluster. Kubelet hoạt động dưới dạng PodSpec. Kubelet lấy một tập hợp các PodSpec được cung cấp thông qua các cơ chế khác nhau và đảm bảo rằng các Container được mô tả trong các PodSpec đó đang chạy và khỏe mạnh. Kubelet không quản lý các Container không được tạo bởi Kubernetes.



Hình 13: Kiến trúc của một cụm Kubernetes

## 2.6.2. Pod

Pod là đơn vị nhỏ nhất trong Kubernetes. Pod là một nhóm các Container đại diện cho khối xây dựng cơ bản trong Kubernetes. Thay vì triển khai các Container riêng lẻ, có thể triển khai một Pod gồm nhiều Container. Tuy nhiên điều đó không có nghĩa là một Pod luôn bao gồm nhiều hơn một Container — thông thường các Pod chỉ chứa một Container duy nhất. Điều quan trọng về Pod là khi một Pod chứa nhiều Container, tất cả chúng luôn được chạy trên một Worker Node duy nhất — nó không bao giờ chạy trên nhiều Worker Node.

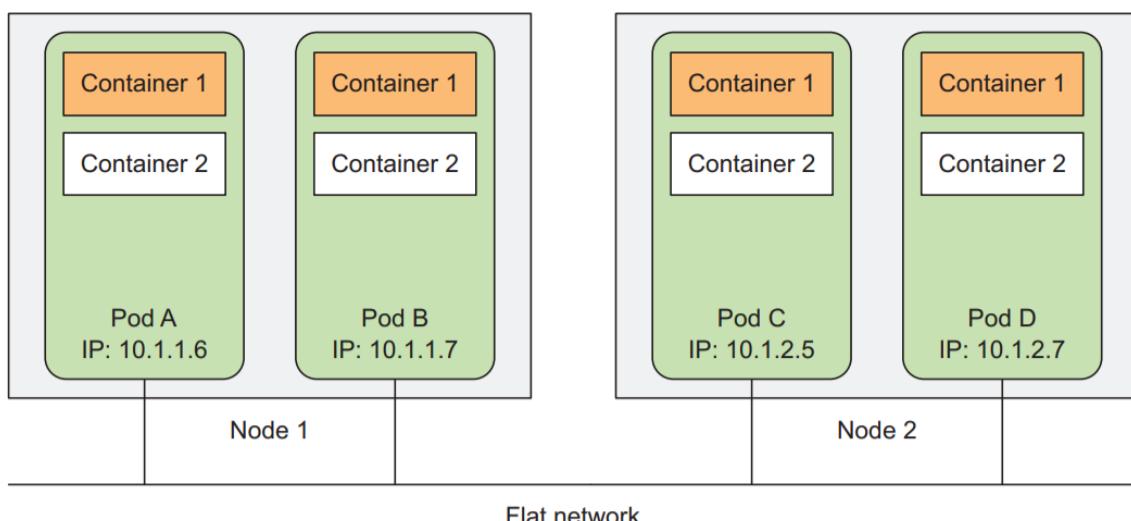


*Hình 14: Tất cả Container của một Pod chạy trên cùng một Node.*

Hãy tưởng tượng một ứng dụng bao gồm nhiều tiến trình giao tiếp thông qua IPC (Inter-Process Communication) hoặc thông qua các tệp được lưu trữ cục bộ, yêu cầu chúng chạy trên cùng một máy. Bởi vì trong Kubernetes, luôn chạy các tiến trình trong các Container và mỗi Container giống như một cỗ máy biệt lập, bạn có thể nghĩ rằng việc chạy nhiều tiến trình trong một Container duy nhất là hợp lý, nhưng không nên làm như vậy. Các Container được thiết kế để chỉ chạy một tiến trình duy nhất cho mỗi Container (trừ khi chính tiến trình này sinh ra các tiến trình con). Nếu chạy nhiều tiến trình không liên quan trong một Container, lập trình viên có trách nhiệm giữ cho tất cả các tiến trình đó chạy, quản lý nhật ký của chúng.

Vì không được hỗ trợ nhóm nhiều tiến trình trong cùng một Container, nên rõ ràng chúng ta cần một cấu trúc cấp cao hơn cho phép liên kết các Container với nhau và quản lý chúng như một đơn vị duy nhất. Đây là lý do cần đến Pod. Một Pod gồm các Container cho phép chúng ta chạy các tiến trình có liên quan chặt chẽ với nhau và cung cấp cho chúng (gần như) cùng một môi trường như thể tất cả chúng đang chạy trong một Container duy nhất.

Tất cả các Pod trong một Kubernetes Cluster nằm trong một không gian single flat, network-address (hình), có nghĩa là mọi Pod đều có thể truy cập vào mọi Pod khác tại địa chỉ IP của Pod kia. Không có cổng NAT (Network Address Translation) nào tồn tại giữa chúng. Khi hai Pod gửi các Network Packets giữa nhau, chúng sẽ thấy địa chỉ IP thực của Pod kia là IP nguồn trong gói.



*Hình 15: Mô Pod có một IP riêng. Các Pod giao tiếp với nhau bằng IP đó*

### **\* Tổ chức các Container trên các Pod đúng cách:**

Trước đây, chúng ta thường gom tất cả các loại ứng dụng vào cùng một máy chủ lưu trữ, tuy nhiên chúng ta không nên làm điều đó với các Pod. Bởi vì Pod tương đối nhẹ, chúng ta có thể tạo bao nhiêu tùy thích mà không tốn bất kì chi phí nào. Thay vì gom tất cả mọi thứ vào một Pod duy nhất, chúng ta nên tổ chức các ứng dụng thành nhiều Pod, trong đó mỗi Pod chỉ chứa các thành phần hoặc tiến trình liên quan chặt chẽ.

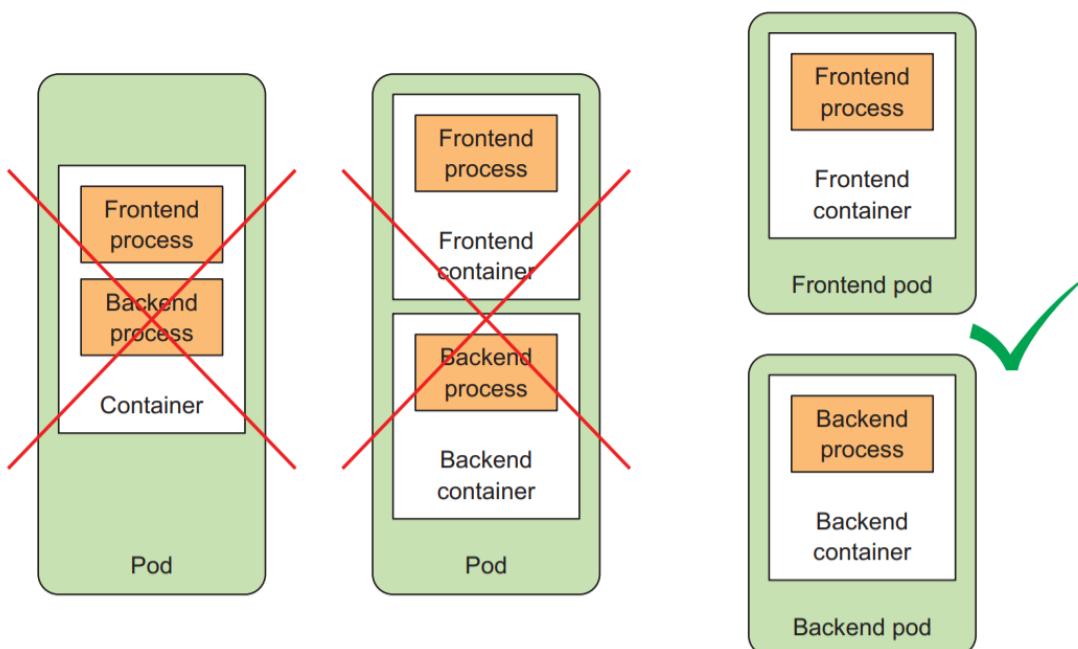
### **\* Khi nào thì sử dụng Multiple Containers trong một Pod:**

Lý do chính để đặt nhiều Container vào một Pod duy nhất là khi ứng dụng bao gồm một tiến trình chính và một hoặc nhiều tiến trình bổ sung.

Khi quyết định đặt hai Container vào một Pod duy nhất hay thành hai Pod riêng biệt, chúng ta luôn cần chú ý những câu hỏi sau:

- Chúng cần được chạy cùng nhau hay chúng có thể chạy trên các máy chủ khác nhau?
- Chúng đại diện cho một tổng thể duy nhất hay chúng là các thành phần độc lập?
- Chúng phải được scale cùng nhau hay riêng lẻ?

Về cơ bản, chúng ta nên luôn hướng tới việc chạy các Container trong các Pod riêng biệt, trừ khi một lý do cụ thể yêu cầu chúng phải là một phần của cùng một Pod.



*Hình 16: Một Container không được chạy nhiều tiến trình.*

**\* Tạo một file mô tả YAML đơn giản cho Pod:**

```
Descriptor conforms to version v1 of Kubernetes API
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual
spec:
  containers:
    - image: luksa/kubia
      name: kubia
      ports:
        - containerPort: 8080
          protocol: TCP
```

You're describing a pod.

The name of the pod

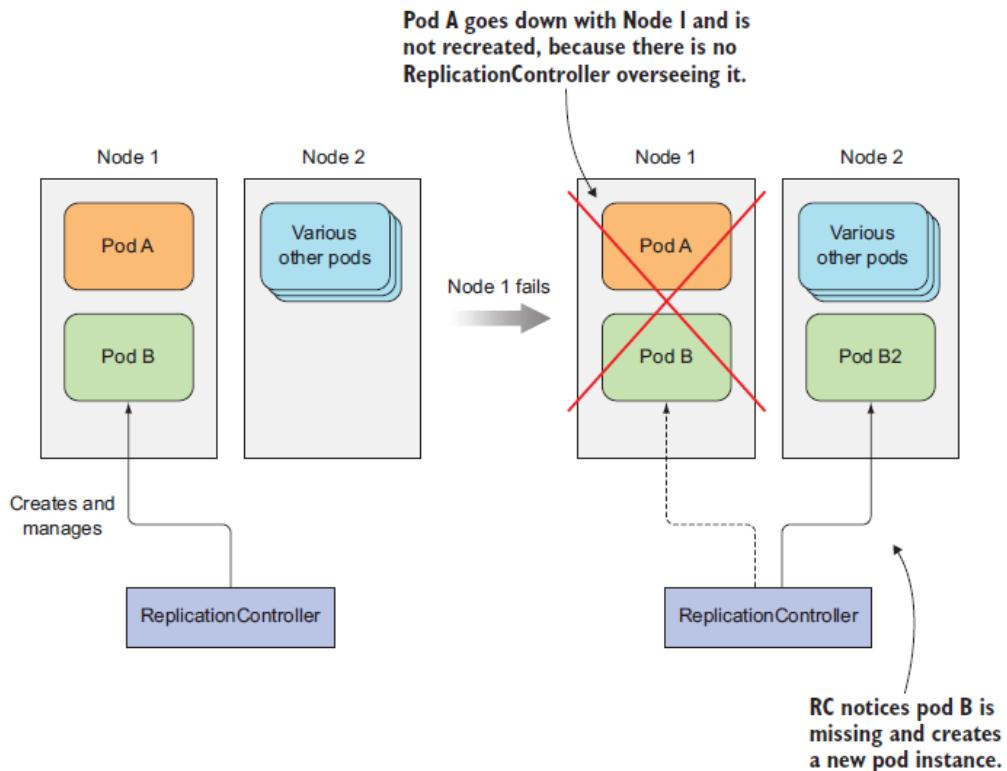
Container image to create the container from

Name of the container

The port the app is listening on

### 2.6.3. Replication và các Controller khác

ReplicationController là một tài nguyên Kubernetes đảm bảo các Pod của nó luôn chạy. Nếu Pod biến mất vì bất kỳ lý do gì, chẳng hạn như trong trường hợp một Node biến mất khỏi Cluster, ReplicationController sẽ thông báo Pod bị thiếu và tạo một Pod thay thế.



Hình 17: Mô tả ReplicationController.

Hình trên cho thấy điều gì sẽ xảy ra khi một Node gặp sự cố và mang theo hai Pod. Pod A được tạo trực tiếp và do đó là Pod không được quản lý, trong khi Pod B được quản lý bởi một ReplicationController. Sau khi Node bị lỗi, ReplicationController tạo một Pod mới (nhóm B2) để thay thế nhóm B bị thiếu, trong khi nhóm A bị mất hoàn toàn sẽ không bao giờ tạo lại được nó. ReplicationController trong hình chỉ quản lý một Pod duy nhất, nhưng nói chung, ReplicationControllers được dùng để tạo và quản lý nhiều bản sao (replicas) của một Pod.

Một ReplicationController có ba phần thiết yếu:

- Một Label Selector: xác định Pod nào nằm trong phạm vi của ReplicationController.
- Số lượng replica: chỉ định số lượng pod mong muốn sẽ được chạy.
- Pod Template: được sử dụng khi tạo các replica Pod mới.

#### 2.6.4. Service

Mỗi Service có địa chỉ IP và cổng không bao giờ thay đổi trong khi Service tồn tại. Client có thể mở các kết nối tới IP và cổng đó, và các kết nối đó sau đó được chuyển đến một trong các Pod hỗ trợ Service đó. Bằng cách

này, Client của một Service không cần biết vị trí của các Pod riêng lẻ cung cấp Service, cho phép các Pod đó được di chuyển xung quanh Cluster bất kỳ lúc nào.

#### **\* Tạo một file mô tả YAML đơn giản cho Service:**

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: kubia
```

Trong hình trên, tạo một Service có tên là kubia, Service này sẽ chấp nhận các kết nối trên cổng 80 và định tuyến từng kết nối đến cổng 8080 của một trong các Pod phù hợp với nhãn selector: app = kubia.

Tiếp tục và tạo dịch vụ bằng cách đăng tệp bằng kubectl create.

Sau khi apply file YAML, có thể liệt kê tất cả các Service và thấy rằng một cụm IP nội bộ đã được chỉ định cho Service:

```
$ kubectl get svc
NAME         CLUSTER-IP      EXTERNAL-IP     PORT(S)      AGE
kubernetes   10.111.240.1    <none>        443/TCP     30d
kubia        10.111.249.153  <none>        80/TCP      6m
```

Ở phần spec chúng ta có thể thêm 1 trường nữa là type biểu thị kiểu service cần dùng (mặc định là ClusterIP). Các loại Service bao gồm:

- **ClusterIP:** Service chỉ có địa chỉ IP cục bộ và chỉ có thể truy cập được từ các thành phần trong cluster Kubernetes.
- **NodePort:** Service có thể tương tác qua Port của các worker nodes trong cluster (sẽ giải thích kỹ hơn ở phần sau)
- **LoadBalancer:** Service có địa chỉ IP public, có thể tương tác ở bất cứ đâu.
- **ExternalName:** Ánh xạ service với 1 DNS Name

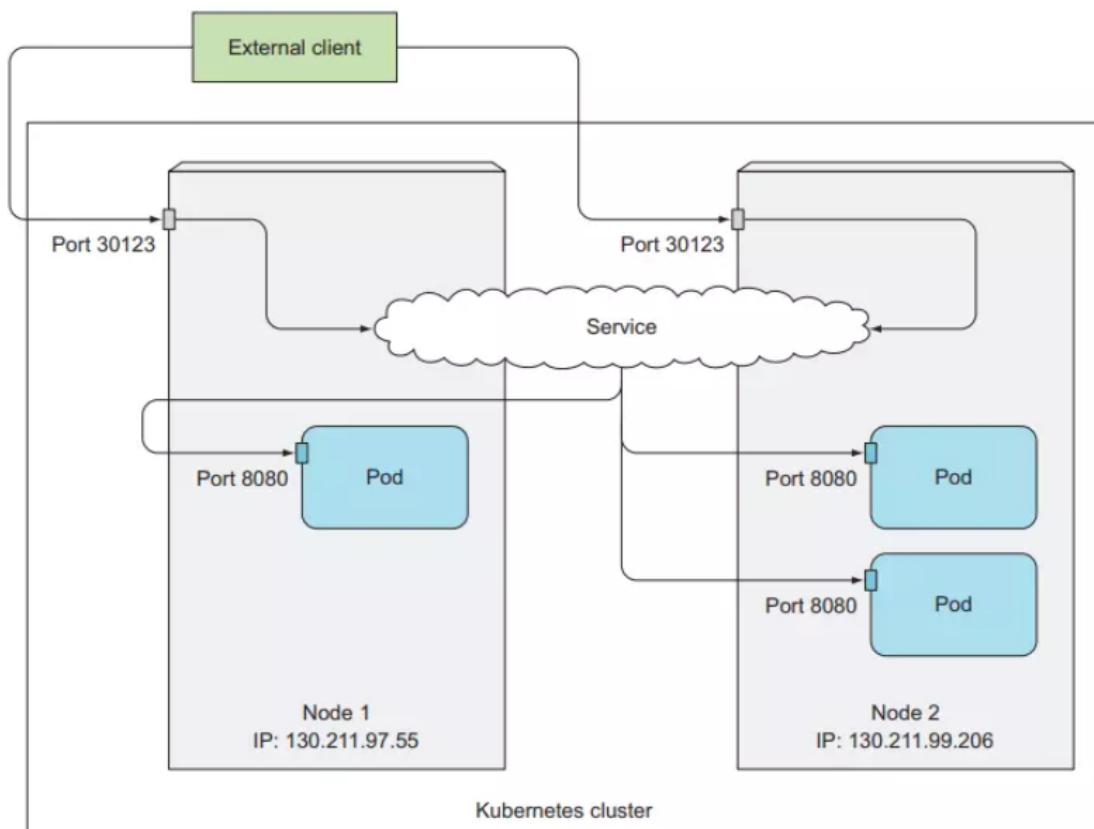
#### **\* Service NodePort:**

Sau khi tạo Service và dùng lệnh kubectl get svc để kiểm tra service vừa tạo thì chúng ta thấy rõ ràng có 2 thông số IP là CLUSTER\_IP và EXTERNAL\_IP có giá trị <none>.

- **CLUSTER\_IP**: Là địa chỉ IP cục bộ trong Cluster Kubernetes, với địa chỉ IP này thì các Pods hay Services có thể tương tác với nhau nhưng bên ngoài sẽ không thể tương tác với Service thông qua nó được.
- **EXTERNAL\_IP**: IP public, có thể dùng để client bên ngoài (hoặc bất cứ đâu) tương tác với Service.

Type NodePort giúp Service có thể tương tác được từ bên ngoài thông qua port của worker node.

Khi tương tác với Service, client sẽ truy cập qua <Địa chỉ Ip public của Node>:Port



Hình 18: Service NodePort

## 2.6.5. Volume

Các Kubernetes Volume là một thành phần của một Pod. Chúng không phải là một đối tượng Kubernetes độc lập và không thể tự tạo hoặc xóa chúng. Một Volume thì khả dụng cho tất cả các Container trong Pod, nhưng nó phải

được mount trong mỗi Container cần truy cập đến nó. Trong mỗi Container, có thể gắn ổ đĩa vào bất kỳ vị trí nào của hệ thống tệp của nó.

Có nhiều loại Volume. Một số là chung chung, trong khi một số khác dành riêng cho các công nghệ lưu trữ thực tế được sử dụng bên dưới. Một số loại Volume có sẵn:

- **voidDir**: Một thư mục trống đơn giản được sử dụng để lưu trữ dữ liệu tạm thời.
- **hostPath**: Được sử dụng để gắn các thư mục từ hệ thống tệp của Worker Node vào Pod.
- **gitRepo**: Ổ đĩa được khởi tạo bằng cách kiểm tra nội dung của kho lưu trữ Git.
- **nfs**: Một chia sẻ NFS được gắn vào Pod.
- **gcePersistentlyDisk (Google Compute Engine Persistent Disk), awsElastic-BlockStore (Amazon Web Services Elastic Block Store Volume), azureDisk (Microsoft Azure Disk Volume)**: Được sử dụng để gắn kết bộ nhớ cụ thể của nhà cung cấp dịch vụ đám mây.
- **cinder, cephfs, iscsi, floe, glusterfs, quobyte, rbd, flexVolume, vsphereVolume, photonPersistingDisk, scaleIO**: Được sử dụng để gắn các loại lưu trữ mạng khác.
- **configMap, secret, downwardAPI**: Các loại ổ đĩa đặc biệt được sử dụng để hiển thị một số tài nguyên Kubernetes và thông tin Cluster cho Pod.
- **PersistentVolumeClaim**: Một cách để sử dụng bộ lưu trữ liên tục được cung cấp trước hoặc động.

\* Sử dụng NFS Volume:

Nếu Cluster đang chạy trên tập hợp máy chủ riêng, có một loạt các tùy chọn được hỗ trợ khác để gắn bộ nhớ ngoài bên trong ổ đĩa của mình. Ví dụ, để gắn kết một chia sẻ NFS đơn giản, chỉ cần chỉ định máy chủ NFS và đường dẫn được xuất bởi máy chủ, như được hiển thị dưới đây:

```
volumes:  
- name: mongodb-data  
  nfs:  
    server: 1.2.3.4  
    path: /some/path
```

This volume is backed by an NFS share.  
The IP of the NFS server  
The path exported by the server

## 2.6.6. ConfigMap và Secret

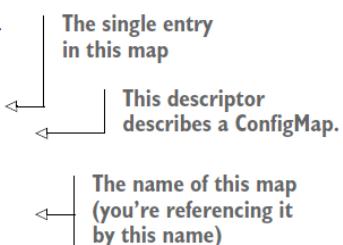
### 2.6.6.1.ConfigMap

Đây là một Resource giúp chúng ta tách Configuration ra riêng. Với giá trị sẽ được định nghĩa theo kiểu key/value pairs ở thuộc tính data.

Và giá trị này sẽ được truyền vào bên trong container như một env. Vì vì ConfigMap nó là một resource riêng lẻ, ta có thể sử dụng nó lại cho nhiều container khác nhau. Sử dụng ConfigMap là cách ta tách khỏi việc phải viết cấu hình env bên trong config container của Pod.

#### \* Tạo một ConfigMap:

```
$ kubectl get configmap fortune-config -o yaml
apiVersion: v1
data:
  sleep-interval: "25"
kind: ConfigMap
metadata:
  creationTimestamp: 2016-08-11T20:31:08Z
  name: fortune-config
  namespace: default
  resourceVersion: "910025"
  selfLink: /api/v1/namespaces/default/configmaps/fortune-config
  uid: 88c4167e-6002-11e6-a50d-42010af00237
```



### 2.6.6.2.Secret

Bí mật cũng giống như ConfigMap, chúng cũng là bản đồ chứa các cặp key-value. Chúng có thể được sử dụng giống như một ConfigMap. Secret giúp giữ bí mật của an toàn bằng cách đảm bảo mỗi Secret chỉ được phân phối cho các Node chạy các Pod cần truy cập vào Bí mật. Ngoài ra, trên chính các Pod, Secret luôn được lưu trữ trong bộ nhớ và không bao giờ được ghi vào bộ nhớ vật lý, điều này sẽ yêu cầu xóa sạch các đĩa sau khi xóa Secret khỏi chúng.

Trên Master Node (cụ thể hơn trong etcd), Secret thường được lưu trữ ở dạng không mã hóa, có nghĩa là Master Node cần được bảo mật để giữ an toàn cho dữ liệu nhạy cảm được lưu trữ trong Secret. Điều này không chỉ bao gồm việc giữ an toàn cho bộ lưu trữ etcd mà còn ngăn người dùng trái phép sử dụng máy chủ API vì bất kỳ ai có thể tạo Pod đều có thể gắn Secret vào Pod và có quyền truy cập vào dữ liệu nhạy cảm thông qua đó. Từ phiên bản Kubernetes 1.7, etcd lưu trữ Secret ở dạng mã hóa, giúp hệ thống an toàn hơn nhiều. Do đó, chúng ta bắt buộc phải chọn đúng thời điểm sử dụng Secret hoặc ConfigMap.

### 2.6.7. Deployment

Deployment là một Resource của Kubernetes giúp ta trong việc cập nhật một Version mới của ứng dụng một cách dễ dàng, nó cung cấp sẵn 2 strategy để deploy là Recreate và RollingUpdate, tất cả đều được thực hiện tự động bên dưới, và các Version được deploy sẽ có một histroy ở đằng sau, ta có thể Rollback và Rollout giữa các phiên bản bất cứ lúc nào mà không cần chạy lại CI/CD. Deployment dùng để triển khai những ứng dụng dạng Stateless (không trạng thái). Khi triển khai Deployment, các Pod được tạo với tên ngẫu nhiên.

Deployment là tài nguyên cấp cao hơn dành cho việc triển khai ứng dụng và cập nhật chúng, thay vì thực hiện nó thông qua ReplicationController hoặc ReplicaSet, cả hai đều được coi là các khái niệm cấp thấp hơn. Khi bạn tạo Deployment, tài nguyên ReplicaSet sẽ được tạo bên dưới (cuối cùng là nhiều tài nguyên trong số đó). ReplicaSets là một thế hệ mới của ReplicationControllers và nên được sử dụng thay thế chúng. ReplicaSets cũng sao chép và quản lý các Pods. Khi sử dụng Deployment, các Pod thực tế được tạo và quản lý bởi ReplicaSets của Deployment chứ không phải bởi trực tiếp Deployment.



Hình 19: Mô tả Deployment.

Khi ta tạo một Deployment, nó sẽ tạo ra một ReplicaSet bên dưới, và ReplicaSet sẽ tạo Pod. Luôn như sau **Deployment tạo và quản lý ReplicaSet -> ReplicaSet tạo và quản lý Pod -> Pod run container**.

**\* Tạo Deployment:**

Tạo một Deployment không khác gì tạo một ReplicationController. Triển khai cũng bao gồm nhãn selector, số lượng replica mong muốn và Pod Template. Ngoài ra, nó cũng chứa một trường chỉ định chiến lược triển khai xác định cách cập nhật sẽ được thực hiện khi tài nguyên Deployment được

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  template:
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:
        - image: luksa/kubia:v1
          name: nodejs
```

sửa đổi.

#### \* **Update Image trong Deployment:**

##### **Cú pháp:**

```
$ kubectl set image deployment <deployment-name> <container-name>=<new-image>
```

##### **Kiểm tra quá trình Update đã xong chưa:**

```
$ kubectl rollout status deploy <deployment-name>
```

##### **Rollback lại version trước khi version mới của ứng dụng bị lỗi**

```
$ kubectl rollout undo deployment <deployment-name> --to-revision=<old_revision>
```

Ngoài ra Deployment còn có một điểm mạnh rất tuyệt vời nữa là có thể giúp ta config để deploy ứng dụng zero downtime, tuy ta dùng RollingUpdate nhưng không cũng không chắc được là ứng dụng của chúng ta zero downtime

#### **2.6.8. StatefulSet**

Pod Replica được quản lý bởi ReplicaSet hoặc ReplicationController giống như “gia súc”. Bởi vì chúng hầu như không có trạng thái, chúng có thể

được thay thế bằng một bản sao Pod hoàn toàn mới bất kỳ lúc nào. Các Stateful Pod yêu cầu một cách tiếp cận khác. Khi một Stateful Pod chết (hoặc Node mà nó đang chạy không thành công), cần được phục hồi trên một Node khác, nhưng cần có cùng tên, danh tính mạng và trạng thái như Node mà nó đang thay thế. Đây là những gì sẽ xảy ra khi các Pod được quản lý thông qua StatefulSet.

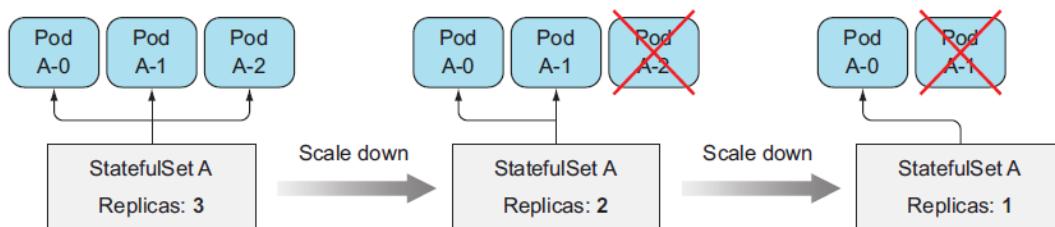
Ở trong Kubernetes ta có thể deploy một Stateful Application bằng cách tạo Pod và Config Volume cho Pod, hoặc dùng PersistentVolumeClaim. Nhưng ta chỉ có thể tạo một single instance của Pod mà kết nối tới PersistentVolumeClaim đó. Cái ta muốn là sẽ tạo ra nhiều replicas của Pod, và mỗi Pod ta sẽ dùng một PersistentVolumeClaim riêng, để chạy được một ứng dụng distributed data store.

Giống như ReplicaSet, StatefulSet là một resource giúp chúng ta chạy nhiều Pod mà cùng một Template bằng cách set thuộc tính Replicas, nhưng khác với ReplicaSet ở chỗ là Pod của StatefulSet sẽ được định danh chính xác và mỗi cái sẽ có một stable network identity của riêng nó.

Mỗi Pod được tạo ra bởi StatefulSet sẽ được gán với một index, index này sẽ được sử dụng để định danh cho mỗi Pod. Và tên của Pod sẽ được đặt theo kiểu <statefulset name>-<index>, chứ không phải random như của ReplicaSet.

#### \* Scale một StatefulSet:

Scale StatefulSet tạo ra một Pod mới với index chưa dùng tới. Nếu Scale Up từ 2 lên 3 Pod, Pod mới sẽ có index là 2 (các Pod hiện tại có index 0 và 1). Điều hay ho khi Scale Down StatefulSet là chúng ta luôn biết Pod nào sẽ bị xóa. Một lần nữa, điều này cũng trái ngược với việc Scale Down ReplicaSet, vì chúng ta không biết Pod nào sẽ bị xóa và thậm chí không thể chỉ định Pod nào chúng ta muốn xóa trước (nhưng tính năng này có thể được giới thiệu trong tương lai). Scale Down StatefulSet sẽ loại bỏ Pod có index cao nhất trước tiên. Điều này làm cho việc Scale Down có thể dự đoán được.

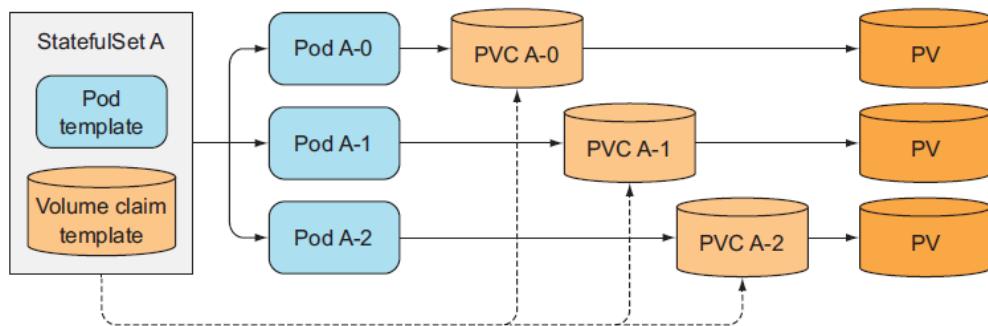


Hình 20: Scale StatefulSet.

### **\* Cung cấp storage riêng biệt cho mỗi Pod:**

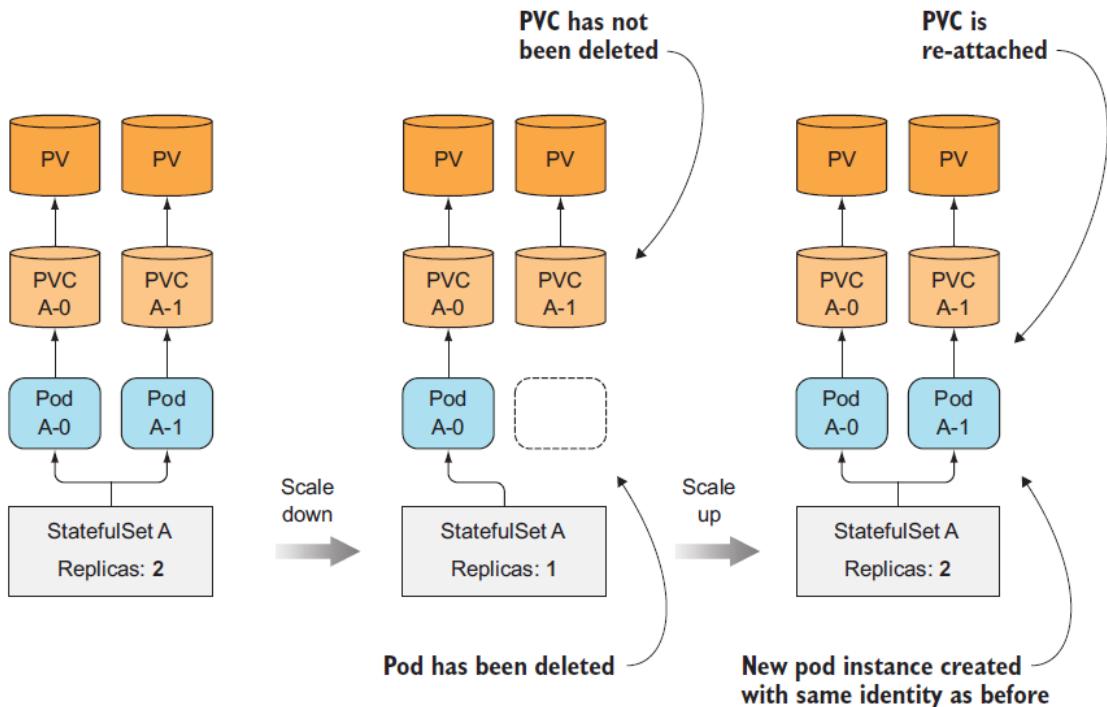
Mỗi Pod của chúng ta cần có một Storage của riêng nó, và khi ta Scale Down số lượng Pod và Scale Up lại thì Pod tạo ra mà có index giống với Pod cũ thì vẫn giữ nguyên Storage của nó chứ không tạo ra một Storage khác.

StatefulSets làm được việc đó bằng cách tách Storage ra khỏi Pod bằng cách sử dụng PersistentVolumeClaims. StatefulSets sẽ tạo ra PersistentVolumeClaims cho mỗi Pod và gắn nó vào cho từng Pod tương ứng.



*Hình 21: Persistent Volume.*

Khi ta Scale Up Pod trong StatefulSets, thì sẽ có một Pod và một PersistentVolumeClaims mới được tạo ra, nhưng khi ta Scale Down, thì chỉ có Pod bị xóa đi, PersistentVolumeClaims vẫn giữ ở đó và không bị xóa. Để khi ta Scale Up lại thì Pod vẫn được gắn đúng với PersistentVolumeClaims trước đó để dữ liệu của nó vẫn được giữ nguyên.



Hình 22: Persistent Volume Claims.

Trước khi triển khai StatefulSet, trước tiên bạn cần tạo một Headless Service, sẽ được sử dụng để cung cấp danh tính mạng cho các Pod.

Đối với Service ClusterIP bình thường, thì khi ta tạo Service đó, nó sẽ tạo ra một Virtual IP cho chính nó và một DNS tương ứng cho VIP đó, và VIP này sẽ mapping với những Pod phía sau Service. Còn đối với Headless Service, khi khai báo config ta sẽ chỉ định thuộc tính **clusterIP: None** cho nó, khi ta tạo một Headless Service thì nó sẽ không tạo ra một Virtual IP cho chính nó, mà chỉ tạo ra một DNS. Sau đó, nó sẽ tạo ra DNS cho chính xác từng Pod phía sau, và mapping DNS tới những DNS của Pod phía sau nó.

#### \* Tạo Headless Service:

```

apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  clusterIP: None
  selector:
    app: kubia
  ports:
    - name: http
      port: 80

```

**Name of the Service**

**The StatefulSet's governing Service must be headless.**

**All pods with the app=kubia label belong to this service.**

### \* Tao StatefulSet:

```

apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: kubia
spec:
  serviceName: kubia
  replicas: 2
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
        - name: kubia
          image: luksa/kubia-pet
          ports:
            - name: http
              containerPort: 8080
          volumeMounts:
            - name: data
              mountPath: /var/data
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        resources:
          requests:
            storage: 1Mi
        accessModes:
          - ReadWriteOnce

```

**Pods created by the StatefulSet will have the app=kubia label.**

**The container inside the pod will mount the pvc volume at this path.**

**The PersistentVolumeClaims will be created from this template.**

## 2.6.9. Tự động mở rộng Pod và Cluster Node

Nói về scale thì có 2 cách scale là horizontal scaling và vertical scaling:

- **Horizontal scaling** là cách scale mà ta sẽ tăng số lượng Worker (Application) đang xử lý công việc hiện tại ra nhiều hơn. Ví dụ ta đang có 2 Pod để xử lý tích điểm cho Client khi Client tạo Deal thành công, khi số lượng Client tăng đột biến, 2 Pod hiện tại không thể xử lý kịp, ta sẽ scale số lượng Pod lên thành 4 Pod chẳng hạn.
- **Vertical scaling** là cách scale thay vì tăng số lượng Worker lên, ta sẽ tăng số lượng tài nguyên có thể sử dụng của ứng dụng đó lên, như là tăng số lượng CPU và Memory của ứng dụng đó.

Trong Kubernetes, ta Horizontal Scale bằng cách tăng số lượng ở thuộc tính Replicas của ReplicationController, ReplicaSet, Deployment. Vertical scale bằng cách tăng Resource Requests và Limits của Pod. Ta có thể làm việc này bằng tay, nhưng sẽ gặp rất nhiều bất lợi, ta không thể ngồi cả ngày để kiểm tra lúc nào ứng dụng của ta có nhiều Client sử dụng nhất để ta kết nối lên Kubernetes Cluster rồi gõ câu lệnh để scale được, mà ta muốn công việc này có thể tự động được.

Kubernetes có cung cấp cho chúng ta cách Autoscaling dựa vào việc phát hiện CPU hoặc Memory ta chỉ định đã đạt tới ngưỡng scale. Nếu ta xài Cloud, nó còn có thể tự động tạo thêm Worker Node khi phát hiện không còn đủ Node cho Pod deploy.

#### \* Horizontal pod autoscaling:

Horizontal pod autoscaling là cách ta tăng giá trị replicas ở trong các scalable resource (Deployment, ReplicaSet, ReplicationController, hoặc StatefulSet) để scale số lượng Pod. Công việc này được thực hiện bởi Horizontal controller khi ta tạo một HorizontalPodAutoscaler (HPA) resource. Horizontal controller sẽ thường xuyên kiểm tra metric của Pod, và tính toán số lượng pod replicas phù hợp dựa vào metric kiểm tra của Pod hiện tại với giá trị metric mà ta đã chỉ định ở trong HPA resource, sau đó sẽ thay đổi trường replicas của các scalable resource (Deployment, ReplicaSet, ReplicationController, or StatefulSet) nếu nó thấy cần thiết.

#### \* Quá trình Autoscaling:

Quá trình autoscaling được chia thành 3 giao đoạn như sau:

- Thu thập metrics của tất cả các Pod được quản lý bởi scalable resource mà ta chỉ định trong HPA.
- Tính toán số lượng Pod cần thiết dựa vào metrics thu thập được.
- Cập nhật lại trường replicas của scalable resource.

#### \* **Thu thập Metrics:**

Horizontal controller sẽ không trực tiếp thu thập Metrics của Pod, mà nó sẽ lấy thông qua một cái khác, được gọi là Metrics Server. Ở trên từng Worker Node, sẽ có một cái được gọi là cAdvisor, đây là một Component của Kubelet, có nhiệm vụ thu thập metric của Pod và node, sau đó những metric này sẽ được tổng hợp ở Metrics Server, và cái Horizontal Controller sẽ lấy metric từ Metrics Server ra.



Hình 23: Horizontal controller

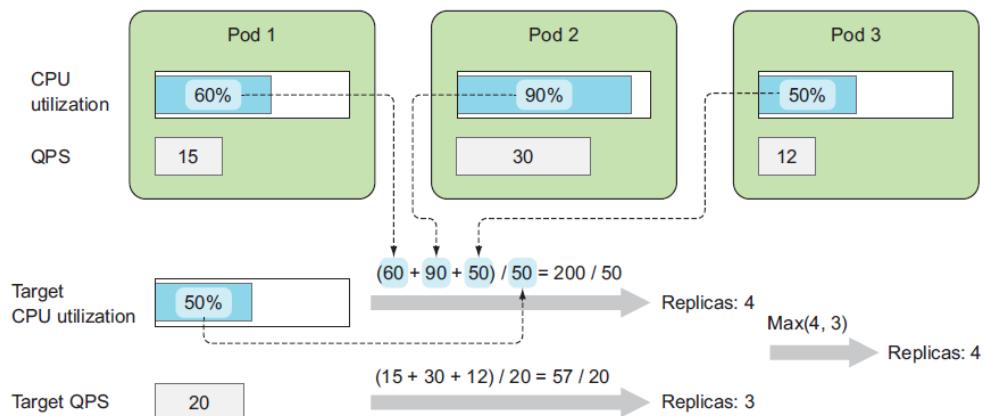
#### \* **Tính số Pod yêu cầu:**

Khi Autoscaler có số liệu cho tất cả các Pod thuộc tài nguyên mà Autoscaler đang mở rộng quy mô (tài nguyên Deployment, ReplicaSet, ReplicationController hoặc StatefulSet), nó có thể sử dụng các số liệu đó để tìm ra số lượng replica cần thiết. Nó cần tìm số sẽ mang lại giá trị trung bình của chỉ số trên tất cả các replica đó càng gần với giá trị mục tiêu đã định cấu hình càng tốt. Đầu vào cho phép tính này là một tập hợp các chỉ số nhóm (có thể nhiều chỉ số trên mỗi nhóm) và đầu ra là một số nguyên duy nhất (số lượng bản sao nhóm).

Khi Autoscaler được định cấu hình để chỉ xem xét một chỉ số duy nhất, việc tính toán số lượng bản sao được yêu cầu rất đơn giản. Tất cả những gì cần làm là tổng hợp các giá trị chỉ số của tất cả các Pod, chia giá trị đó cho giá trị mục tiêu được đặt trên tài nguyên HorizontalPodAutoscaler, sau đó làm tròn thành số nguyên lớn hơn tiếp theo.

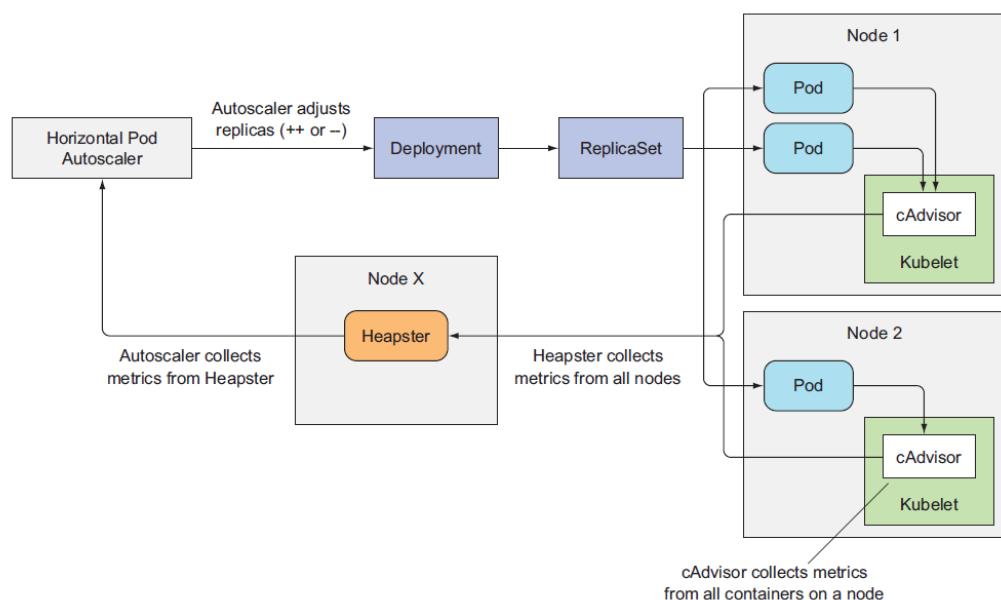
Tính toán thực tế có liên quan nhiều hơn một chút so với điều này, vì nó cũng đảm bảo Autoscaler không gặp sự cố khi giá trị chỉ số không ổn định và thay đổi nhanh chóng. Khi tính năng tự động chia tỷ lệ dựa trên nhiều chỉ số nhóm (ví dụ: cả mức sử dụng CPU và Truy vấn trên giây [QPS]), thì việc

tính toán không phức tạp hơn nhiều. Autoscaler tính toán số lượng bản sao cho từng chỉ số riêng lẻ và sau đó lấy giá trị cao nhất (ví dụ: nếu yêu cầu bốn nhóm để đạt được mức sử dụng CPU mục tiêu và ba nhóm được yêu cầu để đạt được QPS mục tiêu, thì Autoscaler sẽ mở rộng thành bốn nhóm). Hình sau cho thấy ví dụ này.



Hình 24: Minh họa AutoScaler

#### \* Quy trình tiến hành AutoScale:



Hình 25: Quy trình tiến hành AutoScale

Các mũi tên dẫn từ Pod đến cAdvisors, tiếp tục đến Heapster và cuối cùng là Horizontal Pod Autoscaler, cho biết hướng của luồng metrics data. Điều quan trọng cần lưu ý là mỗi thành phần sẽ nhận metrics từ các thành

phần khác theo định kỳ (nghĩa là, cCity lấy số liệu từ các nhóm trong một vòng lặp liên tục; điều này cũng đúng với Heapster và bộ điều khiển HPA). Kết quả cuối cùng là phải mất khá nhiều thời gian để truyền dữ liệu chỉ số và thực hiện hành động thay đổi tỷ lệ. Nó không phải là ngay lập tức.

### 3. CouchDB

#### 3.1. CouchDB là gì?

CouchDB là 1 cơ sở dữ liệu dạng NoSQL mã nguồn mở dùng để lưu trữ dữ liệu dạng document/JSON.

CouchDB được thiết kế nhằm tối tính dễ sử dụng và phục vụ cho môi trường web.

#### 3.2. Tại sao chúng ta lại cần CouchDB ?

CouchDB có API dạng RESTFul giúp cho việc giao tiếp với cơ sở dữ liệu được đơn giản.

Các RESTFul API rất trực quan và dễ thao tác.

Dữ liệu được lưu dưới cấu trúc document rất mềm dẻo, chúng ta không cần phải lo lắng về cấu trúc dữ liệu.

Map/reduce giúp việc lọc, tìm, tổng hợp dữ liệu dễ hơn bao giờ hết.

Nhân bản / đồng bộ là sức mạnh đặc biệt của CouchDB mà hiếm database nào có.

#### 3.3. Mô hình dữ liệu

Database là cấu trúc dữ liệu lớn nhất của CouchDB.

Mỗi database là 1 danh sách các document độc lập.

Document bao gồm dữ liệu người dùng thao tác lẫn thông tin về phiên bản của dữ liệu để tiện việc merge dữ liệu.

CouchDB sử dụng cơ chế phiên bản hoá dữ liệu để tránh tình trạng khoá dữ liệu khi đang ghi.

#### 3.4. Các tính năng chính

##### \* Lưu trữ dạng document:

CouchDB là một NoSQL database dạng document. Document là một đơn vị dữ liệu (giống như 1 object của Javascript), mỗi field có một tên riêng không trùng nhau, chứa các loại dữ liệu như chữ, số, Boolean, danh sách... Không có bất kì giới hạn nào về dung lượng text hay số field trong 1 document.

CouchDB cung cấp 1 RESTFul API cho việc đọc và ghi (thêm, sửa, xoá) document.

##### \* Các thuộc tính ACID:

Khi dữ liệu được ghi xuống ổ cứng thì nó sẽ không bị ghi đè. Bất kì thay đổi nào (thêm, sửa, xoá) đều theo chuẩn Atomic, có nghĩa là dữ liệu sẽ

được lưu lại toàn diện hoặc không được lưu lại. Database không bao giờ thêm hay sửa một phần dữ liệu.

Hầu hết các cập nhật đều được serialized để đảm bảo tất cả người dùng có thể đọc document mà không bị chờ đợi hoặc gián đoạn.

\* ***Khả năng nén (compaction):***

Nén là 1 hành động giúp giải phóng dung lượng ổ cứng được sử dụng bằng cách xoá đi các dữ liệu không còn được sử dụng. Khi tiến hành nén dữ liệu ở 1 file thì 1 file mới với định dạng .compaction sẽ được tạo ra và dữ liệu sẽ được sao chép vào file mới này. Khi quá trình copy hoàn thành thì file cũ sẽ được xoá bỏ. Database vẫn online trong quá trình nén và các thao tác thay đổi / đọc dữ liệu vẫn diễn ra bình thường.

\* ***Views:***

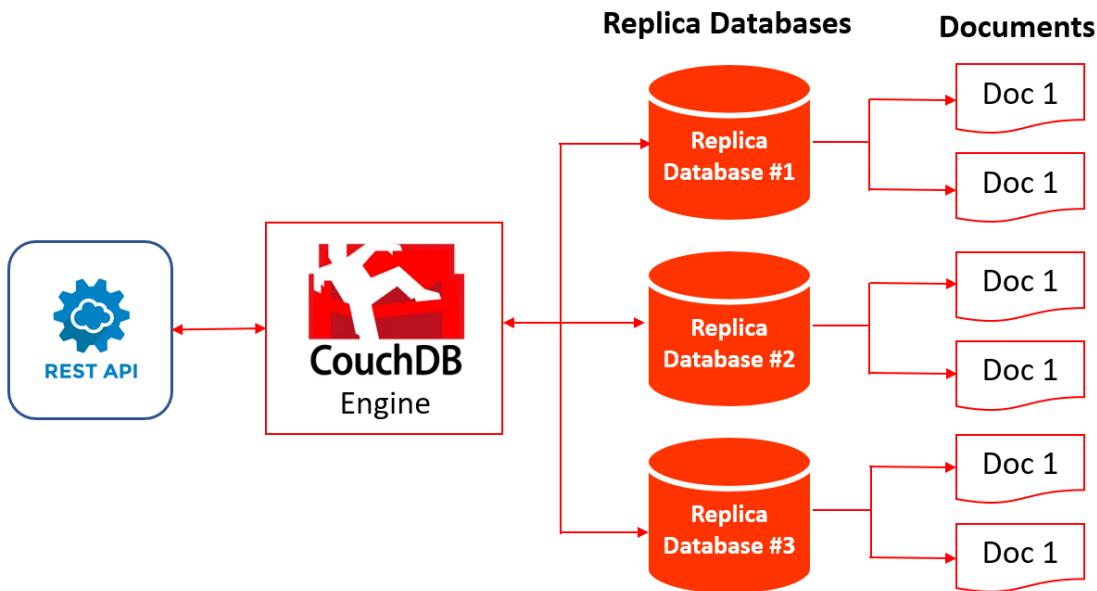
Dữ liệu trong CouchDB được lưu trữ trong các document. Bạn có thể tưởng tượng tương như 1 database là 1 table và 1 document là 1 row. Khi chúng ta muốn trình bày dữ liệu bằng nhiều góc nhìn khác nhau thì chúng ta cần 1 phương pháp để filter, tổ chức để hiển thị kết quả cuối cùng.

### 3.5. Công nghệ Cluster

\* **Cluster là gì?**

Clustering là một kiến trúc nhằm đảm bảo nâng cao khả năng sẵn sàng cho các hệ thống mạng máy tính. Clustering cho phép sử dụng nhiều máy chủ kết hợp với nhau tạo thành một cụm có khả năng chịu đựng hay chấp nhận sai sót (fault-tolerant) nhằm nâng cao độ sẵn sàng của hệ thống mạng.

Cluster là một hệ thống bao gồm nhiều máy chủ được kết nối với nhau theo dạng song song hay phân tán và được sử dụng như một tài nguyên thống nhất. Nếu một máy chủ ngừng hoạt động do bị sự cố hoặc để nâng cấp, bảo trì, thì toàn bộ công việc mà máy chủ này đảm nhận sẽ được tự động chuyển sang cho một máy chủ khác (trong cùng một cluster) mà không làm cho hoạt động của hệ thống bị ngắt hay gián đoạn. Quá trình này gọi là “fail-over” và việc phục hồi tài nguyên của một máy chủ trong hệ thống (cluster) được gọi là “fail-back”.



*Hình 26: Cluster là gì?*

**\* Cơ chế hoạt động của Cluster:**

Trong một Cluster có nhiều node có thể kết hợp cả node chủ động và node thụ động. Trong những mô hình loại này việc quyết định một node được cấu hình là chủ động hay thụ động rất quan trọng. Để hiểu lý do tại sao, hãy xem xét các tình huống sau:

- Nếu một node chủ động bị sự cố và có một node thụ động đang sẵn sàng, các ứng dụng và dịch vụ đang chạy trên node hỏng có thể lập tức được chuyển sang node thụ động. Vì máy chủ đóng vai trò node thụ động hiện tại chưa chạy ứng dụng hay dịch vụ gì cả nên nó có thể gánh toàn bộ công việc của máy chủ hỏng mà không ảnh hưởng gì đến các ứng dụng và dịch vụ cung cấp cho người dùng cuối (Ngầm định rằng các máy chủ trong cluster có cấu trúc phần cứng giống nhau).

- Nếu tất cả các máy chủ trong cluster là chủ động và có một node bị sự cố, các ứng dụng và dịch vụ đang chạy trên máy chủ hỏng sẽ phải chuyển sang một máy chủ khác cũng đóng vai trò node chủ động. Vì là node chủ động nên bình thường máy chủ này cũng phải đảm nhận một số ứng dụng hay dịch vụ gì đó, khi có sự cố xảy ra thì nó sẽ phải gánh thêm công việc của máy chủ hỏng. Do vậy để đảm bảo hệ thống hoạt động bình thường kể cả khi có sự cố

thì máy chủ trong Cluster cần phải có cấu hình dư ra đủ để có thể gánh thêm khối lượng công việc của máy chủ khác khi cần.

Trong cấu trúc Cluster mà số node chủ động nhiều hơn số node bị động, các máy chủ cần có cấu hình tài nguyên CPU và bộ nhớ mạnh hơn nữa để có thể xử lý được khối lượng công việc cần thiết khi một node nào đó bị hỏng.

Các node trong một cluster thường là một bộ phận của cùng một vùng (domain) và có thể được cấu hình là máy điều khiển vùng (domain controllers) hay máy chủ thành viên. Lý tưởng nhất là mỗi Cluster nhiều node có ít nhất hai node làm máy điều khiển vùng và đảm nhiệm việc failover đối với những dịch vụ vùng thiết yếu. Nếu không như vậy thì khả năng sẵn sàng của các tài nguyên trên cluster sẽ bị phụ thuộc vào khả năng sẵn sàng của các máy điều khiển trong domain.

Nhờ những tính năng vượt trội như vậy, công nghệ Cluster đã được áp dụng vào trong các máy chủ, giúp cho việc quản trị dữ liệu được tự động, an toàn và có thể dễ dàng phục hồi khi xảy ra sự cố.

## 4. NFS

### 4.1. NFS là gì?

NFS (Network File System) là một hệ thống giao thức chia sẻ file phát triển bởi Sun Microsystems từ năm 1984, cho phép một người dùng trên một máy tính khách truy cập tới hệ thống file chia sẻ thông qua một mạng máy tính giống như truy cập trực tiếp trên ổ cứng. Hiện tại có 3 phiên bản NFS là NFSv2, NFSv3, NFSv4.

#### \* NFSv2:

Phiên bản 2 của giao thức (được xác định trong RFC 1094, tháng 3 năm 1989) ban đầu chỉ hoạt động trên Giao thức gói dữ liệu người dùng (UDP). Các nhà thiết kế của nó có nghĩa là giữ cho phía máy chủ không trạng thái, với việc khóa (ví dụ) được triển khai bên ngoài giao thức cốt lõi. Những người liên quan đến việc tạo ra NFS phiên bản 2 bao gồm Russel Sandberg, Bob Lyon, Bill Joy, Steve Kleiman và những người khác. Các hệ thống tập tin ảo giao diện cho phép thực hiện mô đun, phản ánh trong một giao thức đơn giản. Đến tháng 2 năm 1986, việc triển khai đã được chứng minh cho các hệ điều hành như System V phát hành 2, DOS và VAX / VMS sử dụng Eunice. NFSv2 chỉ cho phép đọc 2 GB đầu tiên của tệp do giới hạn 32 bit.

#### \* NFSv3:

Phiên bản 3 (RFC 1813, tháng 6 năm 1995) đã thêm:

Hỗ trợ kích thước và độ lệch tệp 64 bit, để xử lý các tệp lớn hơn 2 gigabyte (GB);

Hỗ trợ ghi không đồng bộ trên máy chủ, để cải thiện hiệu suất ghi;

Thuộc tính tệp bổ sung trong nhiều phản hồi, để tránh phải tìm nạp lại chúng;

Một thao tác REaddirplus, để xử lý tệp và các thuộc tính cùng với tên tệp khi quét thư mục;

Các loại cải tiến khác.

Đề xuất NFS Phiên bản 3 đầu tiên trong Sun microsystems đã được tạo ra không lâu sau khi phát hành NFS Phiên bản 2. Động lực chính là nỗ lực giảm thiểu vấn đề hiệu năng của hoạt động ghi đồng bộ trong NFS Phiên bản 2. Đến tháng 7 năm 1992, việc thực hiện thực tế đã giải quyết được nhiều thiếu sót của NFS Phiên bản 2, chỉ thiếu hỗ trợ tệp lớn (kích thước tệp 64 bit và độ lệch) là một vấn đề cấp bách. Điều này đã trở thành một điểm đau cấp tính đối với Tập đoàn Thiết bị Kỹ thuật số với việc giới thiệu phiên bản Ultrix 64 bit để hỗ trợ bộ xử lý RISC 64 bit mới phát hành của họ , Alpha 21064 . Tại thời điểm giới thiệu Phiên bản 3, nhà cung cấp hỗ trợ cho TCP dưới dạng giao thức lớp vận chuyển bắt đầu tăng. Mặc dù một số nhà cung cấp đã thêm hỗ trợ cho NFS Phiên bản 2 với TCP làm phương tiện vận chuyển, Sun microsystems đã thêm hỗ trợ cho TCP dưới dạng vận chuyển cho NFS đồng thời bổ sung hỗ trợ cho Phiên bản 3. Sử dụng TCP làm phương tiện vận chuyển được thực hiện bằng NFS qua mạng WAN khả thi hơn và cho phép sử dụng các kích thước truyền và đọc lớn hơn vượt quá giới hạn 8 KB do Giao thức gói dữ liệu người dùng áp đặt .

#### **\* NFSv4 [chỉnh sửa]:**

Phiên bản 4 ( RFC 3010 , tháng 12 năm 2000; được sửa đổi trong RFC 3530, tháng 4 năm 2003 và một lần nữa trong RFC 7530, tháng 3 năm 2015), chịu ảnh hưởng của Andrew File System (AFS) và Server Message Block (SMB, cũng được gọi là CIFS), bao gồm cải tiến hiệu suất, bắt buộc bảo mật mạnh mẽ, và giới thiệu một giao thức trạng thái. Phiên bản 4 trở thành phiên bản đầu tiên được phát triển với Lực lượng đặc nhiệm kỹ thuật Internet (IETF) sau khi Sun microsystems bàn giao việc phát triển các giao thức NFS.

Phiên bản NFS 4.1 ( RFC 5661, tháng 1 năm 2010) nhằm mục đích cung cấp hỗ trợ giao thức để tận dụng các triển khai máy chủ phân cụm bao gồm khả năng cung cấp quyền truy cập song song có thể mở rộng vào các tệp được phân phối giữa nhiều máy chủ (tiện ích mở rộng pNFS). Phiên bản 4.1 bao gồm cơ chế trung kế phiến (Còn được gọi là NFS Multipathing) và có sẵn trong một số giải pháp doanh nghiệp như VMware ESXi.

Phiên bản NFS 4.2 ( RFC 7862 ) đã được xuất bản vào tháng 11 năm 2016 [8] với các tính năng mới bao gồm: sao chép và sao chép phía máy chủ, tư vấn I / O ứng dụng, tệp thưa, đặt chỗ không gian, khối dữ liệu ứng dụng (ADB), được gắn nhãn NFS với sec\_label có thể chứa bất kỳ hệ thống bảo mật MAC nào và hai hoạt động mới cho pNFS (LAYOUTERROR và LAYOUTSTATS).

Một lợi thế lớn của NFSv4 so với các phiên bản trước đó là chỉ có một cổng UDP hoặc TCP, 2049, được sử dụng để chạy dịch vụ, giúp đơn giản hóa việc sử dụng giao thức trên tường lửa.

#### **4.2. Những tính năng của NFS là gì?**

NFS cho phép truy cập cục bộ đến các tệp từ xa, cho phép nhiều máy tính sử dụng cùng một tệp để mọi người trên mạng có thể truy cập vào cùng một dữ liệu

Với sự trợ giúp của NFS, chúng ta có thể cấu hình các giải pháp lưu trữ tập trung.

Giảm chi phí lưu trữ bằng cách để các máy tính chia sẻ ứng dụng thay vì cần dung lượng ổ đĩa cục bộ cho mỗi ứng dụng của người dùng

Giảm chi phí quản lý hệ thống và minh bạch hệ thống tập tin

Cung cấp tính nhất quán và độ tin cậy của dữ liệu vì tất cả người dùng đều có thể đọc cùng một bộ tệp

Có thể bảo mật với Firewalls và Kerberos

#### **4.3. Một vài tùy chọn của đặc quyền cần biết.**

**ro:** Các máy khách chỉ có quyền đọc các tập tin chia sẻ.

**rw:** Tùy chọn cho phép quyền đọc và ghi tập tin.

**sync:** Đồng bộ xác nhận các yêu cầu tới thư mục được chia sẻ chỉ khi các thay đổi đã được cam kết.

**no\_subtree\_check:** Tùy chọn này ngăn việc kiểm tra cây con. Khi một thư mục dùng chung là thư mục con của một hệ thống tệp lớn hơn, nfs thực hiện quét mọi thư mục bên trên nó, để xác minh các quyền và chi tiết của nó. Việc tắt kiểm tra cây con có thể làm tăng độ tin cậy của NFS, nhưng làm giảm tính bảo mật..

**no\_root\_squash:** Cụm từ này cho phép root kết nối với thư mục được chỉ định.

## CHƯƠNG 3: PHƯƠNG PHÁP THỰC HIỆN

### 1. TỔNG QUAN BÀI TOÁN

Như đã nói ở trên, xây dựng một ứng dụng Web mua sắm online trong tình hình hiện nay là rất cần thiết. Và việc xây dựng một ứng dụng Web thì thời gian hoạt động liên tục của máy chủ là yếu tố vô cùng quan trọng đối với người sử dụng, đặc biệt là đối với một doanh nghiệp. Khi xảy ra thời gian chết hệ thống hoặc lưu lượng truy cập quá mức khiến máy chủ bị quá tải thì việc có một máy chủ dự phòng là cần thiết. Tuy nhiên việc đó khiến chúng ta phải đổi mới với vấn đề về chi phí. Có thể chúng ta sẽ phải tăng gấp đôi chi phí cho một máy chủ vật lý chính và một máy chủ vật lý dự phòng, ngoài ra còn có không gian đặt máy chủ vật lý và các hệ thống khác kèm theo. Đó chính là lý do em chọn Kubernetes.

Kubernetes cho phép chúng ta chạy các ứng dụng phần mềm của mình trên hàng nghìn Node máy tính như thể tất cả các Node đó là một máy tính khổng lồ duy nhất. Nó loại bỏ cơ sở hạ tầng cơ bản và bằng cách đó, đơn giản hóa việc phát triển, triển khai và quản lý cho cả nhóm phát triển và hoạt động.

Kubernetes có cung cấp cho chúng ta cách Autoscaling dựa vào việc phát hiện CPU hoặc Memory ta chỉ định đã đạt tới ngưỡng scale. Nếu ta xài Cloud, nó còn có thể tự động tạo thêm Worker Node khi phát hiện không còn đủ Node cho Pod deploy. Trong Luận Văn này, em sẽ sử dụng Horizontal pod autoscaling để tiến hành Scale tự động CouchDB Cluster và ứng dụng Web.

### 2. CÁC BƯỚC THỰC HIỆN

#### *Bước 1: Cài đặt Docker và Kubernetes:*

Ở bước này, tiến hành tạo 3 máy ảo Ubuntu Version 20.04 sử dụng Virtual Box. Sau đó cấu hình cho 3 máy như sau:

- Máy Master: 192.168.1.100
- Máy Worker 1: 192.168.1.101
- Máy Worker 2: 192.168.1.102

**Chú ý:** Máy Master phải có ít nhất 2 CPU để chạy được Kubernetes.

#### Chi tiết máy:

- Máy Master: 2 CPU, RAM 4GB, Memory 30 GB
- Máy Worker 1: 2 CPU, RAM 4GB, Memory 30 GB
- Máy Worker 2: 2 CPU, RAM 4GB, Memory 30 GB

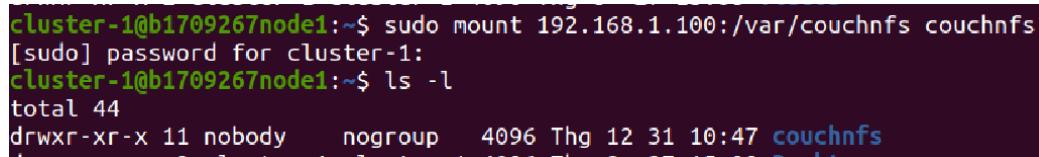
## **Bước 2: Triển khai CouchDB trên Kubernetes, cấu hình Cluster cho CouchDB.**

Ở bước này, đầu tiên cần tạo chia sẻ thư mục “/var/couchnfs” trên máy Master Node với NFS

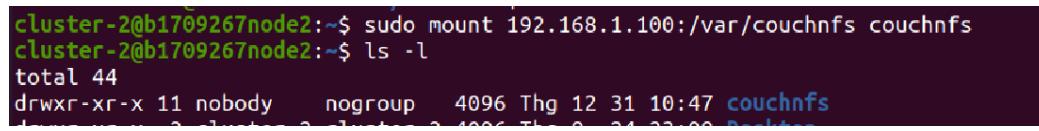


```
Open Save etc exports
1 # /etc/exports: the access control list for filesystems which may be exported
2 # to NFS clients. See exports(5).
3 #
4 # Example for NFSv2 and NFSv3:
5 # /srv/homes      hostname1(rw,sync,no_subtree_check)
6 #           hostname2(ro,sync,no_subtree_check)
7 #
8 # Example for NFSv4:
9 # /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
10 # /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
11 #
12 /var/couchnfs 192.168.1.0/24(rw,sync,no_subtree_check,no_root_squash)
```

Sau đó yêu cầu truy cập vào nội dung đã chia sẻ trên máy Master Node từ các máy Worker Node 1, Worker Node 2 bằng lệnh “mount”

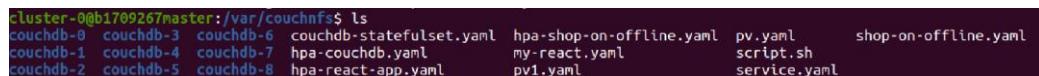


```
cluster-1@b1709267node1:~$ sudo mount 192.168.1.100:/var/couchnfs couchnfs
[sudo] password for cluster-1:
cluster-1@b1709267node1:~$ ls -l
total 44
drwxr-xr-x 11 nobody nogroup 4096 Thg 12 31 10:47 couchnfs
```



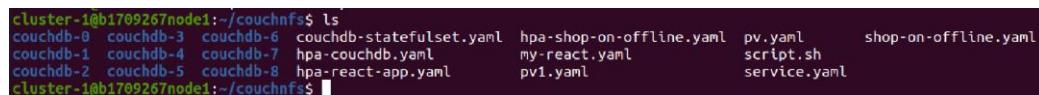
```
cluster-2@b1709267node2:~$ sudo mount 192.168.1.100:/var/couchnfs couchnfs
cluster-2@b1709267node2:~$ ls -l
total 44
drwxr-xr-x 11 nobody nogroup 4096 Thg 12 31 10:47 couchnfs
```

Trên máy Master Node:



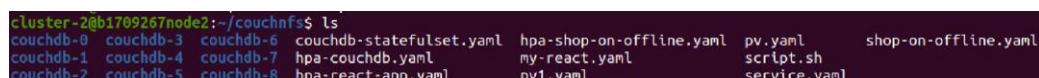
```
cluster-0@b1709267master:/var/couchnfs$ ls
couchdb-0 couchdb-3 couchdb-6 couchdb-statefulset.yaml hpa-shop-on-offline.yaml pv.yaml shop-on-offline.yaml
couchdb-1 couchdb-4 couchdb-7 hpa-couchdb.yaml my-react.yaml script.sh
couchdb-2 couchdb-5 couchdb-8 hpa-react-app.yaml pv1.yaml service.yaml
```

Trên máy Worker Node 1:



```
cluster-1@b1709267node1:~/couchnfs$ ls
couchdb-0 couchdb-3 couchdb-6 couchdb-statefulset.yaml hpa-shop-on-offline.yaml pv.yaml shop-on-offline.yaml
couchdb-1 couchdb-4 couchdb-7 hpa-couchdb.yaml my-react.yaml script.sh
couchdb-2 couchdb-5 couchdb-8 hpa-react-app.yaml pv1.yaml service.yaml
cluster-1@b1709267node1:~/couchnfs$
```

Trên máy Worker Node 2:



```
cluster-2@b1709267node2:~/couchnfs$ ls
couchdb-0 couchdb-3 couchdb-6 couchdb-statefulset.yaml hpa-shop-on-offline.yaml pv.yaml shop-on-offline.yaml
couchdb-1 couchdb-4 couchdb-7 hpa-couchdb.yaml my-react.yaml script.sh
couchdb-2 couchdb-5 couchdb-8 hpa-react-app.yaml pv1.yaml service.yaml
```

Sau đó viết file mô tả YAML để tạo Persistent Volume để lưu trữ dữ liệu, vì CouchDB là Database cần được lưu trữ dữ liệu.



```
1 ---
2 apiVersion: v1
3 kind: PersistentVolume
4 metadata:
5   name: couch-vol-0
6   labels:
7     volume: couch-volume
8 spec:
9   capacity:
10    storage: 10Gi
11   accessModes:
12     - ReadWriteOnce
13   nfs:
14     server: 192.168.1.100
15     path: "/var/couchnfs/couchdb-0"
16 ---
17 apiVersion: v1
18 kind: PersistentVolume
19 metadata:
20   name: couch-vol-1
```

Apply file mô tả để tạo Persistent Volume

```
cluster-0@b1709267master:/var/couchnfs$ kubectl apply -f pv.yaml
persistentvolume/couch-vol-0 created
persistentvolume/couch-vol-1 created
persistentvolume/couch-vol-2 created
cluster-0@b1709267master:/var/couchnfs$ kubectl apply -f pv1.yaml
persistentvolume/couch-vol-3 created
persistentvolume/couch-vol-4 created
persistentvolume/couch-vol-5 created
persistentvolume/couch-vol-6 created
persistentvolume/couch-vol-7 created
persistentvolume/couch-vol-8 created
```

Tiếp theo, tạo file mô tả YAML để triển khai CouchDB. Loại ở đây là StatefulSet vì StatefulSet đúng như cái tên của nó, nó thích hợp dùng để triển khai những ứng dụng dạng Stateful. Và CouchDB thuộc loại đó. Database của chúng ta cần phải được lưu trữ lại dữ liệu.

```

1 ---
2 apiVersion: apps/v1
3 kind: StatefulSet
4 metadata:
5   name: couchdb
6   labels:
7     app: couch
8 spec:
9   replicas: 3
10  serviceName: "couch-service"
11  selector:
12    matchLabels:
13      app: couch
14  template:
15    metadata:
16      labels:
17        app: couch # pod label
18    spec:
19      containers:
20        - name: couchdb

```

The screenshot shows a code editor window with the title "couchdb-statefulset.yaml" and the path "/var/couchnfs". The code is a YAML configuration for a StatefulSet named "couchdb". It specifies 3 replicas, a service name "couch-service", and a pod template with a label "app: couch".

Apply file mô tả “couchdb-statefulset.yaml” để tạo pod

```

cluster-0@b1709267master:/var/couchnfs$ kubectl apply -f couchdb-statefulset.yaml
Warning: spec.template.spec.containers[0].env[6].name: duplicate name "ERL_FLAGS"
statefulset.apps/couchdb created

```

Kiểm tra các pod đã được tạo và hoạt động chưa

```

cluster-0@b1709267master:/var/couchnfs$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
couchdb-0  1/1     Running   0          23s
couchdb-1  1/1     Running   0          15s
couchdb-2  1/1     Running   0          8s

```

Cuối cùng cần tạo Service cho các Pod để các Pod có thể giao tiếp với nhau và kết nối ra bên ngoài.

Viết file mô tả “service.yaml”

```

1 ---
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: couch-service
6   namespace: default
7   labels:
8     app: couch
9 spec:
10   type: ClusterIP
11   clusterIP: None
12   ports:
13     - port: 5984
14       protocol: TCP
15       targetPort: 5984
16   selector:
17     app: couch      # label selector
18 ---
19 kind: Service
20 apiVersion: v1

```

Apply và kiểm tra các Service vừa tạo

```

cluster-0@b1709267master:/var/couchnfs$ kubectl apply -f service.yaml
service/couch-service created
service/couch-nodep-svc created

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
couch-nodep-svc	NodePort	10.108.90.154	<none>	5984:30984/TCP	34s
couch-service	ClusterIP	None	<none>	5984/TCP	35s
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	109d

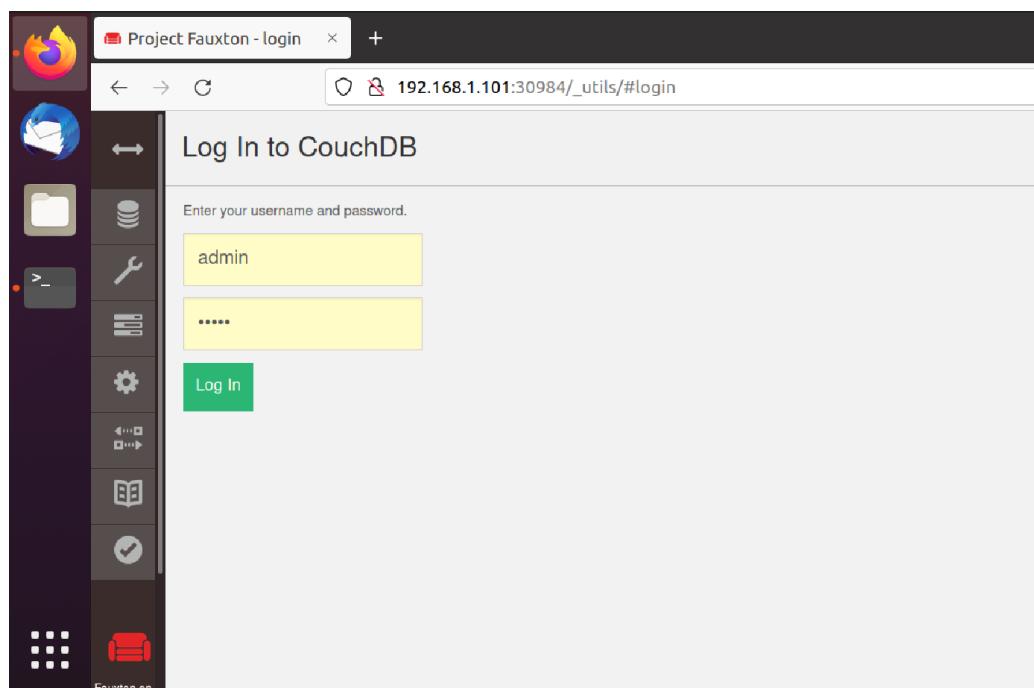
SSH vào pod couchdb-0 để tiến hành cấu hình Cluster

```

cluster-0@b1709267master:/var/couchnfs$ kubectl exec couchdb-0 -i -t -- bash
root@couchdb-0:# curl http://admin:admin@127.0.0.1/_membership
curl: (7) Failed to connect to 127.0.0.1 port 80: Connection refused
root@couchdb-0:# curl http://admin:admin@127.0.0.1:5984/_membership
{"all_nodes":["couchdb@couchdb-0.couch-service"],"cluster_nodes":["couchdb@couchdb-0.couch-service"]}
root@couchdb-0:# curl -X POST -H "Content-Type: application/json" http://admin:admin@127.0.0.1:5984/_cluster_setup -d '{"action": "add_node", "host": "couchdb-1.couch-service", "port": 5984, "username": "admin", "password": "admin"}'
{"ok":true}
root@couchdb-0:# curl -X POST -H "Content-Type: application/json" http://admin:admin@127.0.0.1:5984/_cluster_setup -d '{"action": "add_node", "host": "couchdb-2.couch-service", "port": 5984, "username": "admin", "password": "admin"}'
{"ok":true}
root@couchdb-0:# curl http://admin:admin@127.0.0.1:5984/_membership
{"all_nodes":["couchdb@couchdb-0.couch-service","couchdb@couchdb-1.couch-service","couchdb@couchdb-2.couch-service"],"cluster_nodes":["couchdb@couchdb-0.couch-service","couchdb@couchdb-1.couch-service","couchdb@couchdb-2.couch-service"]}
root@couchdb-0:# exit

```

Kiểm tra Database trên trình duyệt với Port là 30984



### *Bước 3: Build ứng dụng Web, triển khai ứng dụng Web trên Kubernetes.*

Dùng kiến thức tìm hiểu được về Docker để tiến hành build ứng dụng Web thành Image. Trước đó cần tạo tài khoản Docker Hub và tạo repo mới trên Docker Hub

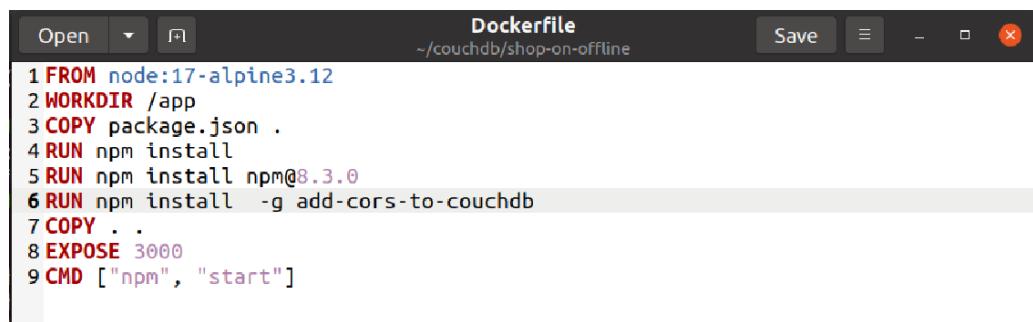
Sau khi đã tạo Repo mới, tiến hành login vào tài khoản Docker Hub

```
cluster-0@b1709267master:~/couchnfs/shop-on-offline$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: b1709267
Password:
WARNING! Your password will be stored unencrypted in /home/cluster-0/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
Login Succeeded
```

Tiến hành Clone ứng dụng Web từ Gib Hub về để thực hiện bước tiếp theo của quá trình build Image

```
cluster-0@b1709267master:~/couchnfs$ git clone https://github.com/b1709267/shop-on-offline.git
Cloning into 'shop-on-offline'...
remote: Enumerating objects: 136, done.
remote: Counting objects: 100% (136/136), done.
remote: Compressing objects: 100% (109/109), done.
remote: Total 136 (delta 19), reused 127 (delta 18), pack-reused 0
Receiving objects: 100% (136/136), 3.57 MiB | 5.07 MiB/s, done.
Resolving deltas: 100% (19/19), done.
cluster-0@b1709267master:~/couchnfs$ ls
shop-on-offline
```

Tạo 1 file Dockerfile trong thư mục “shop-on-offline” vừa Clone từ Git Hub về để mô tả những công việc cần làm khi build Image



Tạo thêm file “.dockerignore” để bỏ qua thư mục “node\_module” vì đã thực hiện copy file “package.json”, nếu trong thư mục “shop-on-offline” có thư mục “.git” do Clone về từ Git Hub thì thêm dòng “.git” vào file này để bỏ qua thư mục “.git”, hoặc loại bỏ thư mục “.git” bằng lệnh “rm –r name” vì chúng ta không cần đến thư mục này cho việc build Image.



Tiến hành build ứng dụng Web thành Image, quá trình build Image có thể mất từ 5 đến 10 phút.

```

cluster-0@b1709267master:~/couchnfs/shop-on-offline$ docker build -t b1709267/shop-on-offline:13.2.1 .
Sending build context to Docker daemon 4.631MB
Step 1/9 : FROM alpine:3.12
--> 9e73a3e8baaf
Step 2/9 : WORKDIR /app
--> Using cache
--> 9e28fb63ab5
Step 3/9 : COPY package.json .
--> Using cache
--> 19982d618e21
Step 4/9 : RUN npm install
--> Running in 44b97e7e25b9

```

Sau khi thấy lệnh build Image thành công, ta kiểm tra Image bằng lệnh “docker images”

```

Successfully built 8c561245eb0a
Successfully tagged b1709267/shop-on-offline:13.2.1
cluster-0@b1709267master:~/couchnfs/shop-on-offline$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
b1709267/shop-on-offline   13.2.1    8c561245eb0a  14 seconds ago  633MB
node                17-alpine3.12  9e73a3e8baaf  9 days ago   170MB
k8s.gcr.io/kube-apiserver  v1.23.1   b6d7abedde39  2 weeks ago   135MB
k8s.gcr.io/kube-proxy     v1.23.1   b46c42588d51  2 weeks ago   112MB
k8s.gcr.io/kube-controller-manager  v1.23.1   f51846a4fd28  2 weeks ago   125MB
k8s.gcr.io/kube-scheduler   v1.23.1   71d575efe628  2 weeks ago   53.5MB
quay.io/coreos/flannel    v0.15.1   e6ea68648f0c  6 weeks ago   69.5MB
k8s.gcr.io/etcd           3.5.1-0   25f8c7f3da61  8 weeks ago   293MB
rancher/mirrored-flannelcni-flannel-cni-plugin  v1.0.0    cd5235cd7dc2  2 months ago  9.03MB
k8s.gcr.io/coredns/coredns  v1.8.6    a4ca41631cc7  2 months ago  46.8MB
k8s.gcr.io/pause          3.6      6270bb605e12  4 months ago  683kB

```

Tiến hành đẩy lên Registry Docker Hub.

```

cluster-0@b1709267master:~/couchnfs/shop-on-offline$ docker push b1709267/shop-on-offline:13.2.1
The push refers to repository [docker.io/b1709267/shop-on-offline]
c26b59aeb157: Pushed
72eead89e4fa: Pushed
2c706f10068f: Pushing [=====] 15.44MB
59dffbd157b: Pushing [====>] 38.67MB/443.6MB
ccc5a75674ff: Pushed
6b3311ab7711: Pushed
ac21e925463c: Layer already exists
9b403f53f62d: Layer already exists
5924aa91f2bb: Layer already exists
eb4bde6b29a6: Waiting

```

Sau khi quá trình hoàn tất, kiểm tra trên Docker Hub

The screenshot shows the Docker Hub repository page for the user 'b1709267'. The repository name is 'shop-on-offline'. The page displays the following information:

- Tags and Scans:** This repository contains 10 tags. The tags listed are: 13.2.1, 11.2.4, 11.2.3, 11.2.2, 11.2.1. There is a note indicating that Vulnerability scanning is disabled.
- Docker commands:** A button labeled 'Public View' is present. Below it is a command input field containing 'docker push b1709267/shop-on-offline:tagname'.
- Automated Builds:** A section explaining that manually pushing images to Hub? is possible. It also mentions connecting to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated.

Khi đã có Image, tiến hành viết file mô tả YAML để triển khai. Loại ở đây sẽ là Deployment. Vì Deployment dùng để tạo những ứng dụng Stateless. Và React App thuộc loại đó. Tương tự như CouchDB, cần tạo Service để có thể kết nối ra bên ngoài.

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: shop-on-offline
5 spec:
6   selector:
7     matchLabels:
8       app: shop-on-offline
9   replicas: 1
10 template:
11   metadata:
12     labels:
13       app: shop-on-offline
14   spec:
15     containers:
16       - name: shop-on-offline
17         image: b1709267/shop-on-offline:13.2.1
18         imagePullPolicy: IfNotPresent
19         resources:
20           requests:
21
YAML ▾ Tab Width: 8 ▾ Ln 23, Col 19 ▾ INS

```

Sau khi đã apply, kiểm tra xem Pod đã được tạo và hoạt động chưa

```

cluster-0@b1709267master:/var/couchnfs$ kubectl get pod -o wide
NAME                           READY   STATUS    RESTARTS   AGE
ATES
couchdb-0                      1/1     Running   0          3m8s
couchdb-1                      1/1     Running   0          3m
couchdb-2                      1/1     Running   0          2m53s
shop-on-offline-58b5f6f75c-926r2  1/1     Running   0          22s

```

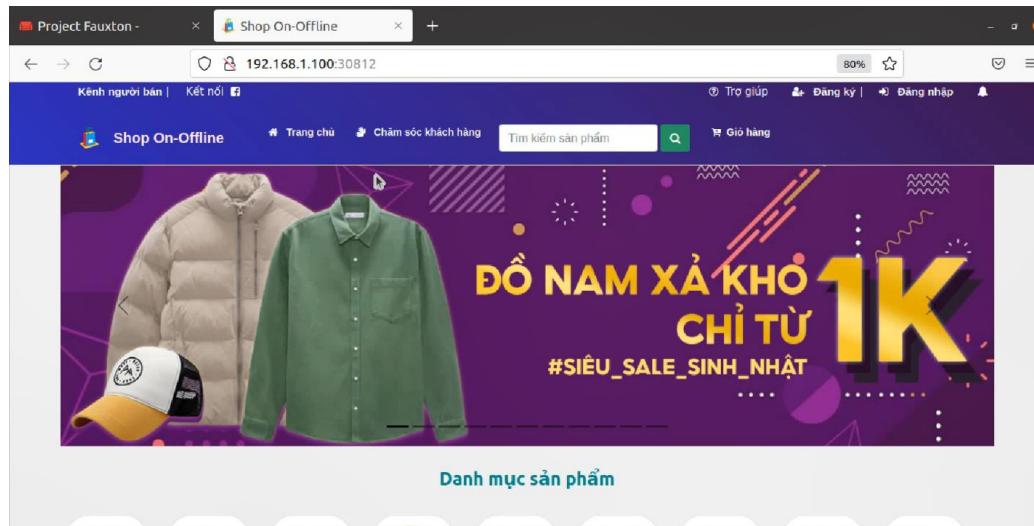
Kiểm tra Service đã tạo và Port của ứng dụng Web sẽ hoạt động trên trình duyệt.

```

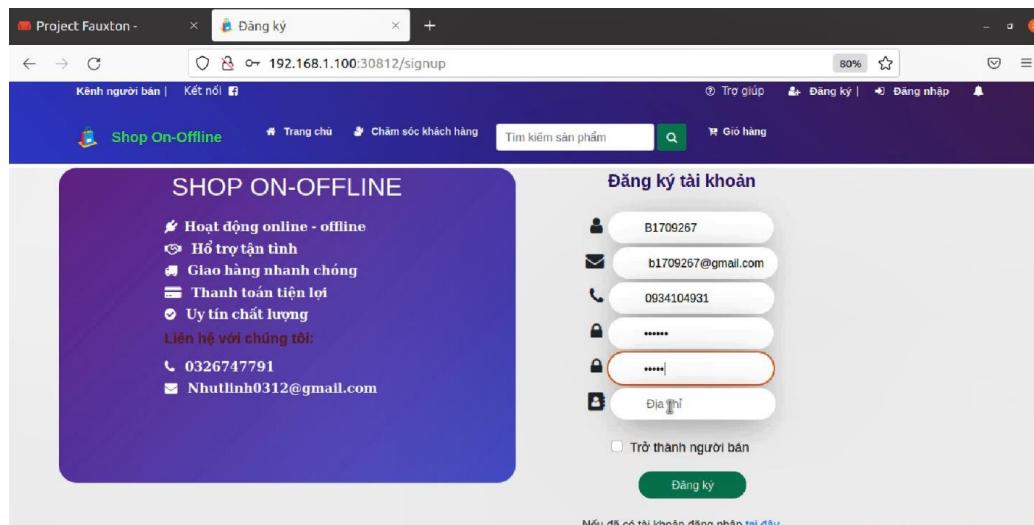
cluster-0@b1709267master:/var/couchnfs$ kubectl get svc
NAME        TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
couch-nodep-svc  NodePort  10.108.90.154 <none>      5984:30984/TCP  3m8s
couch-service  ClusterIP None        <none>      5984/TCP   3m9s
kubernetes    ClusterIP 10.96.0.1    <none>      443/TCP   109d
shop-on-offline NodePort  10.111.93.113 <none>      3000:30812/TCP  29s

```

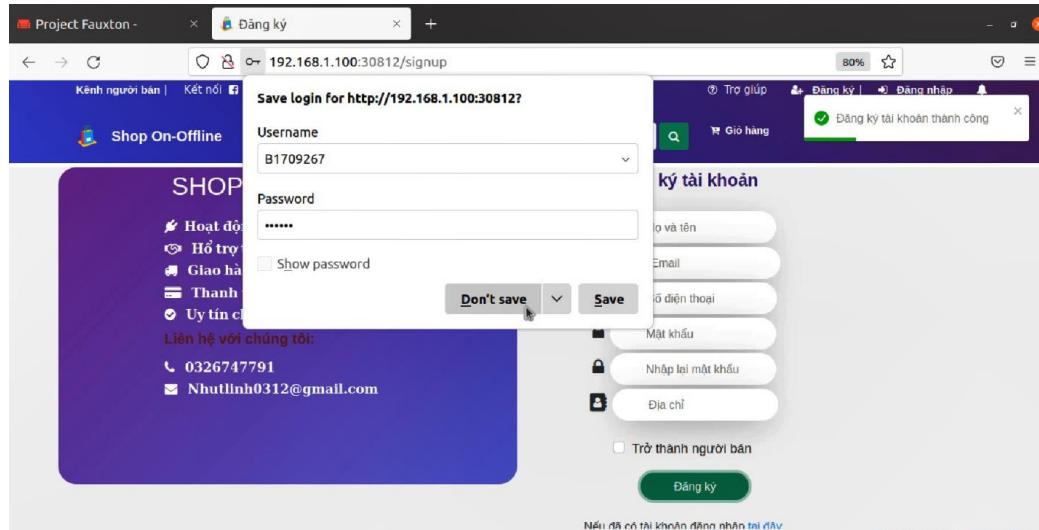
Trên trình duyệt Web, nhập địa chỉ IP của bất kỳ máy nào trong cụm Kubernetes và Port 30812 để kiểm tra.



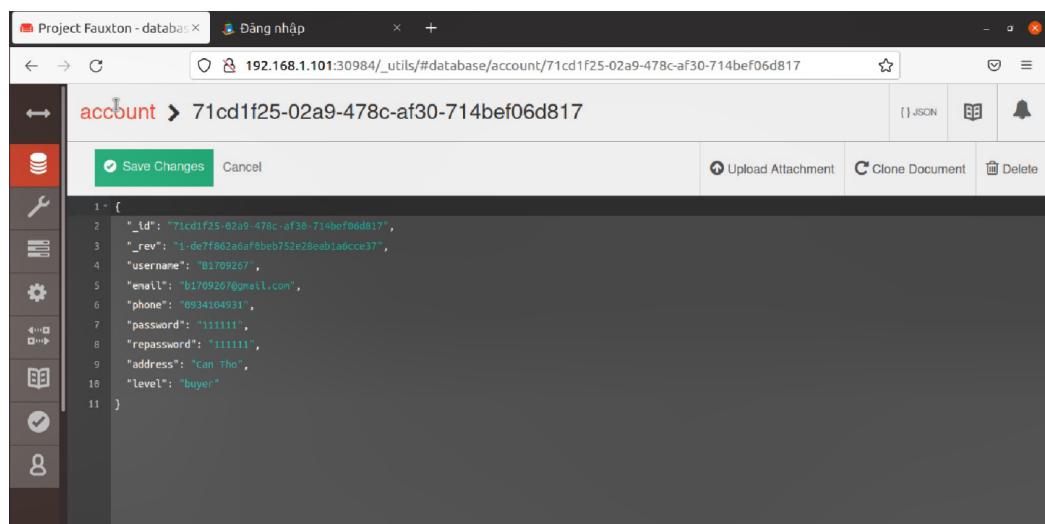
Tiến hành tạo thử một tài khoản người dùng



Sau khi đã có thông báo tạo thành công tài khoản thì kiểm tra dữ liệu trong Database.



Kiểm tra xem dữ liệu đã được lưu vào Database chưa.



Trong trường hợp dữ liệu chưa được lưu vào Database, ta cần SSH vào pod shop-on-offline để addcors

```

cluster-0@b1709267master:/var/couchnfs$ kubectl exec shop-on-offline-58b5f6f75c-926r2 -i -t -- sh
/app # add-cors-to-couchdb http://admin:admin@192.168.1.100:30984/
success
/app # exit

```

**Bước 4: Apply Mertric Server để theo dõi CPU và Memory**  
Vì Horizontal Pod Autoscaler đã được cài đặt sẵn khi cài đặt Kubernetes nên chúng ta cần thêm Metrics Server để theo dõi CPU Download và thêm nội dung sau vào file vừa Download

command:

- /metrics-server
- --kubelet-insecure-tls
  - --kubelet-preferred-address-types=InternalIP

```

127     labels:
128       k8s-app: metrics-server
129   spec:
130     containers:
131       - args:
132           - --cert-dir=/tmp
133           - --secure-port=4443
134           - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
135           - --kubelet-use-node-status-port
136           - --metric-resolution=15s
137       command:
138         - /metrics-server
139         - --kubelet-insecure-tls
140         - --kubelet-preferred-address-types=InternalIP
141       image: k8s.gcr.io/metrics-server/metrics-server:v0.5.2
142       imagePullPolicy: IfNotPresent
143       livenessProbe:
144         failureThreshold: 3
145         httpGet:
146           path: /livez
147           port: https
148           scheme: HTTPS
149           periodSeconds: 10
150       name: metrics-server
151       ports:
152         - containerPort: 4443
153           name: https
154           protocol: TCP

```

Sau khi apply, kiểm tra xem Metrics Server có hoạt động không

```

cluster-0@b1709267master:/var/couchnfs$ kubectl top node
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
b1709267master   686m        34%    2363Mi        61%
b1709267node1    614m        30%    1611Mi        55%
b1709267node2    598m        29%    1860Mi        64%

```

#### *Bước 5: Cài đặt Scale tự động cho CouchDB và ứng dụng Web theo dõi CPU*

Tạo file mô tả YAML để tiến hành cài đặt Scale Pod, cần khai báo MINPODS và MAXPODS hợp lý cũng như name của StatefulSet, Deployment cần Scale và thiết lập Scale khi Pod vượt quá X% CPU.

Kiểm tra Horizontal Pod Autoscaler vừa tạo, số % CPU mà Pod đang sử dụng cũng như số lượng bản sao hiện tại.

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get hpa
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS
hpa-couchdb    StatefulSet/couchdb <unknown>/80%  4          9          0
hpa-shop-on-offline Deployment/shop-on-offline <unknown>/60%  3          20         0
```

Lúc đầu, khi mới tạo ra thì Horizontal Pod Autoscaler sẽ tính toán % CPU mà Pod đang sử dụng và tính toán số lượng replica (bản sao) hiện có, nếu số replica (bản sao) hiện có ít hơn giá trị MINPODS đã thiết lập, Horizontal Pod Autoscaler sẽ tiến hành tạo thêm replica (bản sao) để bằng với giá trị MINPODS đã thiết lập.

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get hpa
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
hpa-couchdb    StatefulSet/couchdb <6%/80%>  4          9          4          40s
hpa-shop-on-offline Deployment/shop-on-offline <0%/60%>  3          20         3          35s
cluster-0@b1709267master:/var/couchnfs$ kubectl get pod
NAME            READY   STATUS    RESTARTS   AGE
couchdb-0       1/1     Running   0          10m
couchdb-1       1/1     Running   0          10m
couchdb-2       1/1     Running   0          10m
couchdb-3       1/1     Running   0          29s
shop-on-offline-58b5f6f75c-926r2 1/1     Running   0          8m
shop-on-offline-58b5f6f75c-cd4cb  1/1     Running   0          24s
shop-on-offline-58b5f6f75c-v6qks  1/1     Running   0          24s
```

Số phần trăm phía trước là số % CPU mà Pod hiện đang sử dụng, số % phía sau là số % đã thiết lập, Horizontal Pod Autoscaler sẽ thực hiện Scale Up khi Pod sử dụng CPU vượt mức này, số lượng replica (bản sao) được tạo thêm là tùy thuộc vào mức sử dụng.

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get pod
NAME            READY   STATUS    RESTARTS   AGE
couchdb-0       1/1     Running   0          12m
couchdb-1       1/1     Running   0          12m
couchdb-2       1/1     Running   0          12m
couchdb-3       1/1     Running   0          2m13s
shop-on-offline-58b5f6f75c-926r2 1/1     Running   0          9m44s
shop-on-offline-58b5f6f75c-9vk2r  1/1     Running   0          8s
shop-on-offline-58b5f6f75c-cd4cb  1/1     Running   0          2m8s
shop-on-offline-58b5f6f75c-v6qks  1/1     Running   0          2m8s
shop-on-offline-58b5f6f75c-vs2lv  1/1     Running   0          53s
cluster-0@b1709267master:/var/couchnfs$ kubectl get hpa
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
hpa-couchdb    StatefulSet/couchdb <6%/80%>  4          9          4          2m36s
hpa-shop-on-offline Deployment/shop-on-offline <75%/60%>  3          20         5          2m31s
```

Khi mức sử dụng CPU giảm dưới mức quy định, lúc này Horizontal Pod Autoscaler sẽ tiến hành tính toán và thực hiện Scale Down

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get hpa
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
hpa-couchdb    StatefulSet/couchdb <6%/80%>  4          9          4          5m49s
hpa-shop-on-offline Deployment/shop-on-offline <75%/60%>  3          20         13         5m44s
```

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get hpa
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
hpa-couchdb    StatefulSet/couchdb <4%/80%>  4          9          4          13m
hpa-shop-on-offline Deployment/shop-on-offline <0%/60%>  3          20         9          13m
```

```

cluster-0@b1709267master:/var/couchnfs$ kubectl get hpa
NAME             REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
hpa-couchdb     StatefulSet/couchdb  4%/80%      4         9          4          14m
hpa-shop-on-offline Deployment/shop-on-offline 0%/60%      3         20         3          14m
cluster-0@b1709267master:/var/couchnfs$ kubectl get pod
NAME                READY   STATUS    RESTARTS   AGE
couchdb-0           1/1    Running   0          24m
couchdb-1           1/1    Running   0          24m
couchdb-2           1/1    Running   0          24m
couchdb-3           1/1    Running   0          14m
shop-on-offline-58b5f6f75c-926r2  1/1    Running   0          21m
shop-on-offline-58b5f6f75c-qpfrog  1/1    Running   0          10m
shop-on-offline-58b5f6f75c-vs2lv   1/1    Running   0          13m

```

Quá trình Scale Down sẽ diễn ra chậm hơn quá trình Scale Up, quá trình Scale Down sẽ hoàn tất trong khoảng từ 10 đến 15 phút. Và số lượng replica (bản sao) sẽ giảm về bằng với giá trị MINPODS đã thiết lập.

#### \* Rollout và Rollback:

Như đã đề cập ở phần cơ sở lý thuyết, Deployment cho phép chúng ta cập nhật Image và cho phép rollback lại phiên bản cũ khi việc update lên phiên bản mới dẫn đến việc ứng dụng Web của chúng ta bị lỗi.

Để update Image ta dùng lệnh “set image”. Trước tiên, kiểm tra tag của Image hiện tại

```

cluster-0@b1709267master:/var/couchnfs$ kubectl describe pod shop-on-offline-58b5f6f75c-926r2
Name:           shop-on-offline-58b5f6f75c-926r2
Namespace:      default
Priority:      0
Node:          b1709267node2/192.168.1.102
Start Time:    Fri, 31 Dec 2021 10:50:56 +0700
Labels:        app=shop-on-offline
               pod-template-hash=58b5f6f75c
Annotations:   <none>
Status:        Running
IP:            10.244.2.89
IPs:
  IP:          10.244.2.89
Controlled By: ReplicaSet/shop-on-offline-58b5f6f75c
Containers:
  shop-on-offline:
    Container ID: docker://a7ec14fc9a0ff07a0b3e922793f6fe99e1f73ce5874a0ee651d29577350e24fcf
    Image:          b1709267/shop-on-offline:2.1.2
    Image ID:      docker://pullable://v1/b1709267/shop-on-offline@sha256:30030358b3e734120389d29832932406751002d55a05e7cacdba388073a3
    Port:          <none>
    Host Port:    <none>
    State:        Running
      Started:   Fri, 31 Dec 2021 10:50:59 +0700
    Ready:        True
    Restart Count: 0
    Limits:
      cpu: 250m
      memory: 400Mi
    Requests:
      cpu: 250m

```

Thực hiện Set Image với Tag 13.2.1

```

cluster-0@b1709267master:/var/couchnfs$ kubectl set image deployment shop-on-offline shop-on-offline=b1709267/shop-on-offline:13.2.1
deployment.apps/shop-on-offline image updated

```

Kiểm tra quá trình Rollout đã hoàn tất chưa

```

cluster-0@b1709267master:/var/couchnfs$ kubectl rollout status deploy shop-on-offline
Waiting for deployment "shop-on-offline" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "shop-on-offline" rollout to finish: 1 old replicas are pending termination...
deployment "shop-on-offline" successfully rolled out

```

Kiểm tra lại Tag của Image bằng cách Describe Pod

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
couchdb-0      1/1     Running   0          49m
couchdb-1      1/1     Running   0          49m
couchdb-2      1/1     Running   0          49m
couchdb-3      1/1     Running   0          39m
shop-on-offline-749867d45f-69zq8  1/1     Running   0          2m15s
Shop-on-offline-749867d45f-gxcqn  1/1     Running   0          12s
shop-on-offline-749867d45f-m8nff  1/1     Running   0          74s
cluster-0@b1709267master:/var/couchnfs$ kubectl describe pod shop-on-offline-749867d45f-69zq8
```

```
IP:           10.244.2.95
IPs:
  IP:          10.244.2.95
Controlled By: ReplicaSet/shop-on-offline-749867d45f
Containers:
  shop-on-offline:
    Container ID:  docker://62fe148b3ea6bc0f6f4e0eb91e33d91a1fc9c7e0a33086b36eaa809c89129d3
    Image:         b1709267/shop-on-offline:13.2.1
    Image ID:      docker-pullable://b1709267/shop-on-offline@sha256:3880f566737372d441f54447d58b17acd500867ca41c38c195cf82b7282
    cc96
      Port:          <none>
      Host Port:    <none>
      State:        Running
      Started:     Fri, 31 Dec 2021 11:36:29 +0700
      Ready:        True
      Restart Count: 0
      Limits:
        cpu: 250m
        memory: 400Mi
      Requests:
        cpu: 250m
        memory: 50Mi
      Environment:  <none>
      Mounts:
        /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-4fskg (ro)
    Conditions:
```

Để quay lại phiên bản cũ, tiến hành kiểm tra lịch sử Rollout

```
cluster-0@b1709267master:/var/couchnfs$ kubectl rollout history deploy shop-on-offline
deployment.apps/shop-on-offline
REVISION  CHANGE-CAUSE
1          <none>
2          <none>
```

Thực hiện việc quay lại phiên bản trước và kiểm tra quá trình

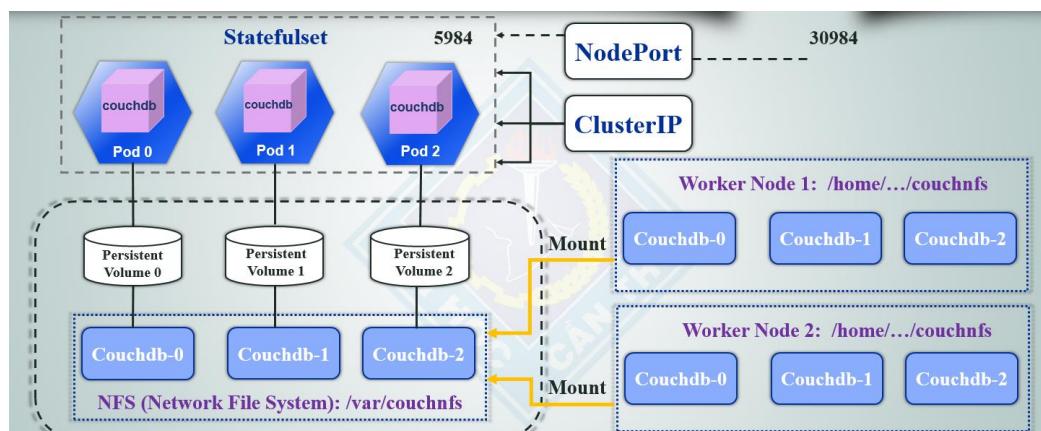
```
cluster-0@b1709267master:/var/couchnfs$ kubectl rollout undo deployment shop-on-offline --to-revision=1
deployment.apps/shop-on-offline rolled back
cluster-0@b1709267master:/var/couchnfs$ kubectl rollout status deploy shop-on-offline
Waiting for deployment "shop-on-offline" rollout to finish: 6 out of 7 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 6 out of 7 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 6 out of 7 new replicas have been updated...
Waiting for deployment "shop-on-offline" rollout to finish: 2 old replicas are pending termination...
Waiting for deployment "shop-on-offline" rollout to finish: 2 old replicas are pending termination...
Waiting for deployment "shop-on-offline" rollout to finish: 2 old replicas are pending termination...
Waiting for deployment "shop-on-offline" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "shop-on-offline" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "shop-on-offline" rollout to finish: 6 of 7 updated replicas are available...
deployment "shop-on-offline" successfully rolled out
cluster-0@b1709267master:/var/couchnfs$
```

Tiến hành Describe một Pod để kiểm tra

```
cluster-0@b1709267master:/var/couchnfs$ kubectl get pod
NAME                      READY   STATUS    RESTARTS   AGE
couchdb-0                  1/1    Running   0          51m
couchdb-1                  1/1    Running   0          50m
couchdb-2                  1/1    Running   0          50m
couchdb-3                  1/1    Running   0          40m
shop-on-offline-58b5f6f75c-bb44z 1/1    Running   0          21s
shop-on-offline-58b5f6f75c-jk2gj  1/1    Running   0          11s
shop-on-offline-58b5f6f75c-nhhwd  1/1    Running   0          22s
shop-on-offline-58b5f6f75c-ptdwp  1/1    Running   0          24s
shop-on-offline-58b5f6f75c-s4s2f  1/1    Running   0          27s
shop-on-offline-58b5f6f75c-tz74g  1/1    Running   0          27s
shop-on-offline-58b5f6f75c-vnwc2  1/1    Running   0          27s
cluster-0@b1709267master:/var/couchnfs$ kubectl describe pod shop-on-offline-58b5f6f75c-bb44z
```

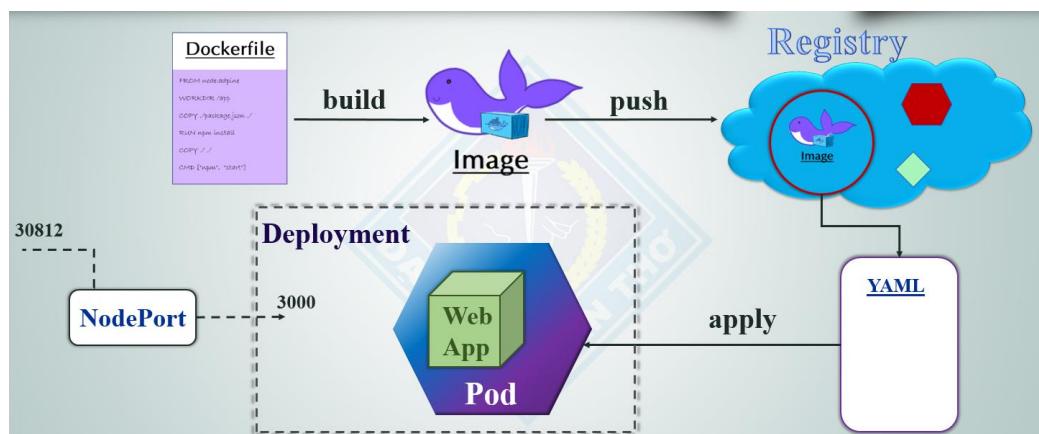
```
Controlled By: ReplicaSet/shop-on-offline-58b5f6f75c
Containers:
  shop-on-offline:
    Container ID: docker://d386a0b8ffe0392ce08f868a009c5d4401b2a12194cc126ad22e7dc482964415
    Image:          b1709267/shop-on-offline:2.1.2
    Image ID:       docker-pullable://b1709267/shop-on-offline@sha256:30030358b3e734120389d29832932406751002d55a05e7cacdba388073a3
    e04f
      Port:           <none>
      Host Port:      <none>
      State:          Running
      Started:        Fri, 31 Dec 2021 11:39:02 +0700
      Ready:          True
      Restart Count:  0
      Limits:
        cpu:          250m
        memory:       400Mi
      Requests:
        cpu:          250m
        memory:       50Mi
      Environment:    <none>
      Mounts:
        /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-p99s5 (ro)
    Conditions:
      Type     Status
      Initialized  True
      Ready      True
      ContainersReady  True
      PodScheduled  True
    Volumes:
      kube-api-access-p99s5:  Type:  Projected (a volume that contains injected data from multiple sources)
```

### \* Sơ đồ tổng quát quá trình triển khai CouchDB trên các Node:



Hình 27: Sơ đồ tổng quát quá trình triển khai CouchDB

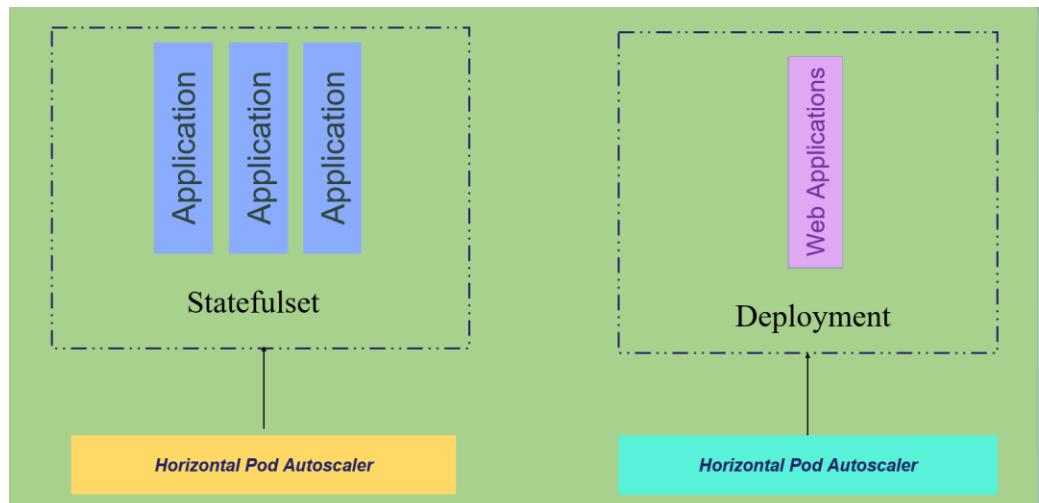
### \* Sơ đồ tổng quát quá trình triển khai ứng dụng Web trên các Node:



Hình 28: Sơ đồ tổng quát quá trình triển khai ứng dụng Web

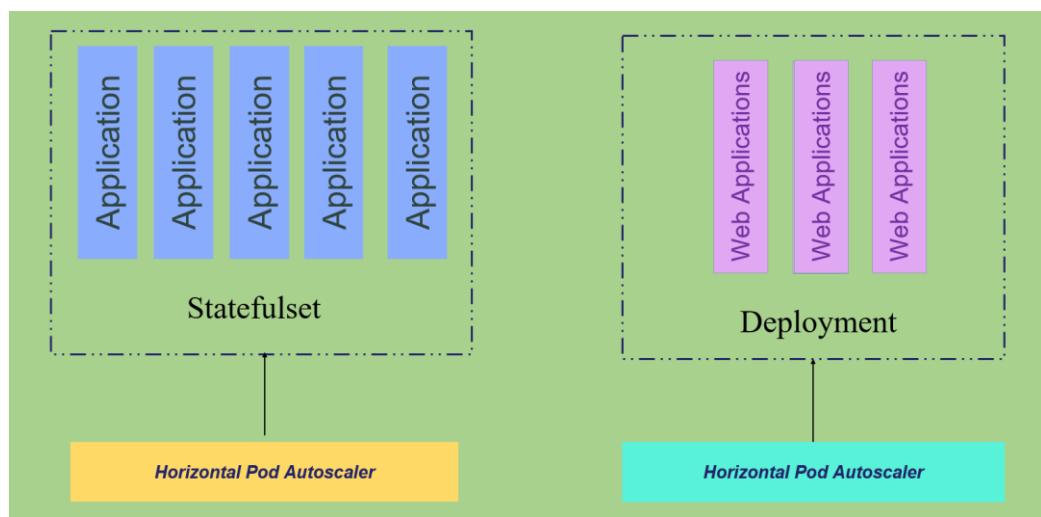
#### \* Minh họa Scale tự động:

Trong đề tài này em đã triển khai một cụm CouchDB gồm 3 node với loại Statefulset và một ứng dụng Web với loại Deployment như hình bên dưới.



Hình 29: Minh họa Scale tự động 1

Thiết lập AutoScaler sẽ hoạt động khi Pod sử dụng CPU vượt mức quy định. Khi Pod sử dụng CPU vượt mức quy định thì Horizontal Pod Autoscaler sẽ tạo thêm Pod đủ để phục vụ nhu cầu sử dụng. Minh họa như hình dưới.



Hình 30: Minh họa Scale tự động 2

## CHƯƠNG 4: PHÂN TÍCH VÀ ĐÁNH GIÁ

### 1. KẾT QUẢ ĐẠT ĐƯỢC

Có thêm những kiến thức hữu ích về Docker và Kubernetes. Tiếp cận gần hơn với công nghệ Container.

Thành công triển khai CouchDB Cluster lên Kubernetes và cấu hình Cluster cho CouchDB.

Biết cách sử dụng Registry Docker Hub để lưu trữ Image và có thể pull lại khi cần sử dụng.

Biết cách sử dụng Git Hub để upload code giúp thuận tiện hơn cho quá trình triển khai.

Thành công triển khai ứng dụng Web lên Kubernetes.

Scale tự động ứng dụng Web khi bị quá tải.

### 2. HẠN CHẾ

Vì triển khai CouchDB dưới dạng StatefulSet nên mỗi Pod được tạo ra sẽ gắn với một Persistent Volume, chúng không thể dùng chung Persistent Volume và phải tạo Persistent Volume bằng với số Pod CouchDB muốn tạo ra trước đó. Việc này dẫn đến một vấn đề khi phải triển khai với số lượng replica lên đến hàng ngàn. Không thể dùng cách thủ công để tạo Persistent Volume vì số lượng quá lớn.

Chưa áp dụng được trong thực tế để kiểm tra tính hiệu quả việc Scale tự động.

### 3. HƯỚNG PHÁT TRIỂN

Như đã đề cập ở trên, trong trường hợp phải triển khai số lượng replica lên đến hàng ngàn thì việc tạo Persistent Volume bằng cách thủ công là bất khả thi. Để giải quyết vấn đề này, giải pháp chính là cấp phát động sử dụng Dynamic Provisioning Of Persistent Volumes.Thêm vào đó, hướng phát triển khác của đề tài là triển khai trên hệ thống đám mây.

## TÀI LIỆU THAM KHẢO

1. Marko Lukša. Kubernetes in Action. (2018).
2. <https://docs.couchdb.org/en/stable/install/index.html>
3. <https://kubernetes.io/>
4. <https://docs.docker.com/get-started/overview/>
5. <https://faun.pub/deploying-a-couchdb-cluster-on-kubernetes-d4eb50a08b34>
6. <https://linuxize.com/post/how-to-install-and-configure-an-nfs-server-on-ubuntu-18-04/>
7. <https://github.com/shivaniarbat/metrics-server-setup-kubernetes>
8. <https://artifacthub.io/packages/helm/couchdb/couchdb>