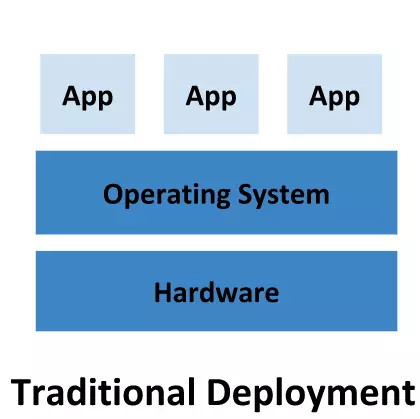
Báo cáo Kubernetes

**Kubernetes là gì?**

Kubernetes là một nền tảng open-source được dùng để quản lý các container.

Các cách deploy ứng dụng phổ biến:

Chạy thẳng ứng dụng trên server(Traditional Deployment)

Ở cách này ta chạy ứng dụng trên server vật lý. Vấn đề khi sử dụng cách deploy này là ta không có cách nào để phân bổ các tài nguyên cho các ứng dụng. Ví dụ khi ta có nhiều ứng dụng chạy trên cùng 1 server vật lý, nếu có một ứng dụng sử dụng tài nguyên hơn thì những ứng dụng còn lại sẽ có ít tài nguyên hơn, gây ảnh hưởng đến hiệu suất ứng dụng -> để khắc phục vấn đề này, ta chỉ có cách triển khai ứng dụng qua một server vật lý khác.

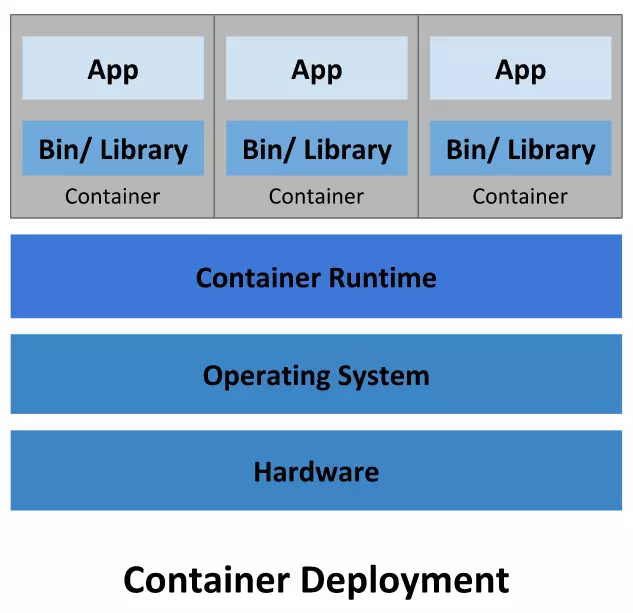
Chia server ra thành các máy ảo và chạy ứng dụng trên các máy ảo đó

A screenshot of a device

Description automatically generated

Mỗi máy ảo sẽ có hệ điều hành(OS), share CPU riêng, các ứng dụng sẽ được chạy trong các máy ảo này. VM (Virutal Machine) được định nghĩa có giới hạn tài nguyên, do đó sẽ không xảy ra vấn đề một ứng dụng sẽ chay tốn tài nguyên giới hạn VM và ảnh hưởng đến những VM khác. Điểm yếu của cách deploy này là do VM copy cả OS và hardware nên server chỉ tạo một số lượng nhỏ VM.

Chạy các ứng dụng bằng cách sử dụng container



Giống như VM, container cũng là cách để ảo hóa ứng dụng, nó cũng có file system, OS nhưng khác với VM, container chỉ copy OS mà không copy hardware, do đó cho phép chúng ta chạy nhiều ứng dụng trên cùng server vật lý mà không lo chiếm tài nguyên của nhau vì mỗi ứng dụng sẽ có môi trường riêng của nó.

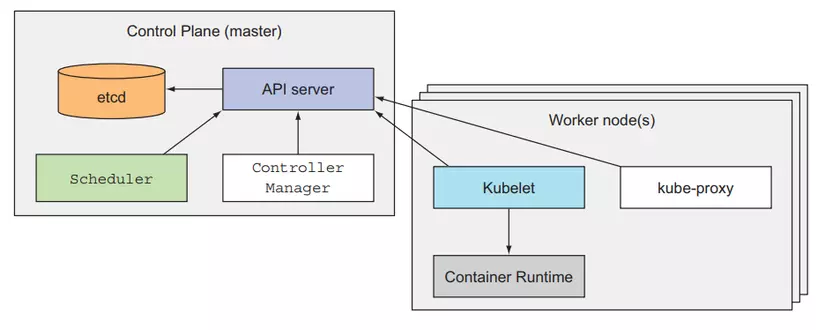
**Kubernetes giải quyết vấn đề gì?**

Nếu số lượng container của chúng ta lên tới vài nghìn thì làm cách nào ta biết được một container nào đó sẽ thuộc về ứng dụng nào? Và nếu ta muốn tang hiệu suất của ứng dụng bằng cách sử dụng nhiều container chạy ứng dụng đó, thì làm cách nào mà ta có thể dẫn request của người dung tới ứng dụng mà có nhiều container? Nếu server vật lý của ta gặp sự cố thì ứng dụng sẽ chết?

Kubernetes sẽ giúp ta giải quyết các vấn đề trên.

**Kiến trúc của Kubernetes**

Kubernetes Cluster gồm 1 master Node và 1 hoặc nhiều worker Node



Master Node gồm 4 thành phần chính là:

+API server: API server chịu trách nhiệm nhận các yêu cầu từ người dung, các thành phần của hệ thống và thực hiện các hành động tương ứng trên cluter. API server cho phép người dung tạo, xóa, cập nhật và truy vấn các tài nguyên như Pod, Service, Deployment, ….

+etcd

+Scheduler

+Controller Manager

Worker Node gồm 3 thành phần chính là:

+Container runtime(docker, containerd, rkt, …) : run các container

+Kubelet: giao tiếp với api-server và quản lý các container

+Kube-proxy: quản lý network và traffic của các ứng dụng trong worker node(metrics)

**POD**

Pod là thành phần cơ bản nhất để deploy và chạy 1 ứng dụng, được tạo và quản lý bởi Kubernetes.

Pod dung để chạy 1 hoặc nhiều container,

A couple of containers with text

Description automatically generated with medium confidence

## ReplicaSet

ReplicaSet cũng là một tài nguyên trên K8S có mục đích là để duy trì một trạng thái ổn định của một bộ các Pod ở một thời điểm nhất định. ReplicaSet có 2 thành phần quan trọng là "thông tin cấu hình của Pod" (tương ứng với thông tin trong mục **template**) và "số lượng Pod" mong muốn (tương ứng với tham số **replicas**). ReplicaSet Controller sẽ làm nhiệm vụ đảm bảo số lượng Pod được khai báo trong ReplicaSet này luôn đảm bảo đúng bằng số lượng Pod mong muốn.

Nếu ta chạy worker node RS sẽ giải quyết vấn đề trên. RS sẽ chắc chắn số pod mà nó tạo ra là không đổi, ví dụ khi ta tạo RS với replicas=1, RS sẽ tạo ra 1 pod và giám sát nó, khi pod đó die, số lương pod bây giờ = 0, RS sẽ tạo ra 1 pod khác để đạt được số pod = replicas đã khai báo là 1.

**Service**

Service là một resource sẽ tạo ra một single, constant point của một nhóm pod phía sau nó. Mỗi service sẽ có một địa chỉ IP và port không đổi. Client sẽ mở kết nối đến service, và connect đến 1 trong những pod phía sau nó.A diagram of a computer system

Description automatically generated

Service giải quyết những vấn đề gì?

Pod có địa chỉ IP riêng, Tại sao ta gọi thẳng pod mà lại dung service.

Pods are ephemeral

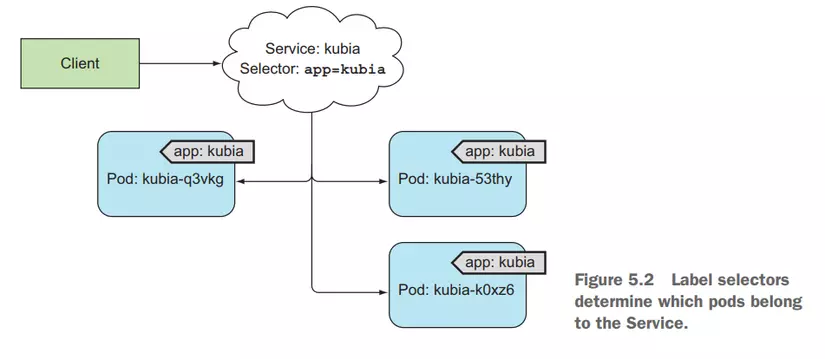
Có nghĩa là khi một pod được tạo ra, bị xóa, bị thay thế bằng 1 pod khác bất cứ lúc nào. Khi tạo ra 1 pod mới, nó sẽ có địa chỉ IP khác với pod cũ. Nếu ta dung IP của pod để kết nối tới client thì khi pod được thay thế thì ta phải update lại code.

Multiple Pod run same application

Có nghĩa là ta sẽ có nhiều pod đang chạy 1 ứng dụng để tang performance. Ví dụ ta dung ReplicaSet với replicas = 3, ta có 3 pod. Vậy làm sao ta biết được nên gửi request tới pod nào?

Để giải quyết những vấn đề trên thì k8s cung cấp cho chúng ta services resource, service sẽ tạo ra 1 endpoint không đổi cho các pod phía sau, client chỉ tương tác với endpoint này.

Service quản lý connection như thế nào?

Làm sao service biết được nó sẽ chọn pod nào để quản lý connection tới những pod đó? Service sử dụng label selectors để chọn pod mà nó quản lý connection

Service có 4 loại cơ bản:

+ClusterIP

+NodePort

+ExternalName

+LoadBalancer

ClusterIP

Đây là loại service sẽ tạo ra 1 IP và local DNS mà sẽ truy cập ở bên trong cluster, không thể truy cập từ bên ngoài, được dung chủ yếu để các pod bên trong cluster giao tiếp với nhau.

NodePort

Đây là cách đầu tiên để expose Pod cho client bên ngoài có thể truy cập được pod bên trong cluster.

LoadBalancer

Ingress resource

**Deployment**

A diagram of a service

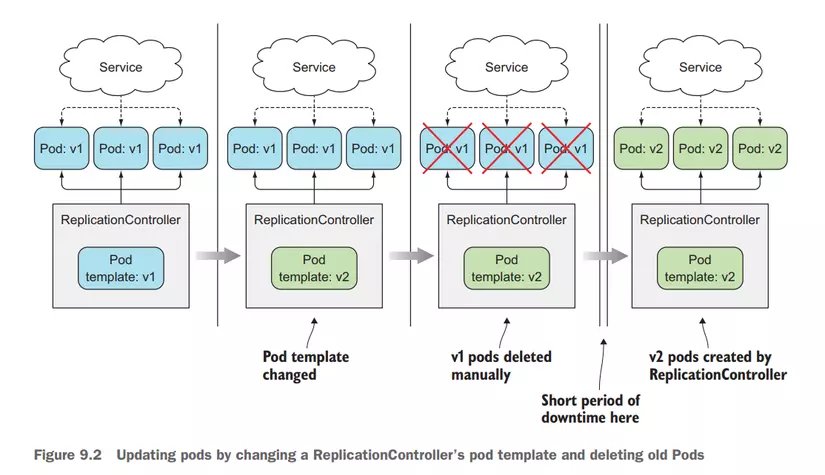
Description automatically generated

Ban đầu chúng ta có 1 ứng dụng đang chạy với số pod = 3, ta deploy 1 service để expose nó ra ngoài cho client.

Bây giờ ta có tính năng mới, build image với code mới, ta muốn update lại các Pod đang chạy với image mới này. Ta sẽ làm như thế nào? Có 2 cách để update là recreate và rollingUpdate

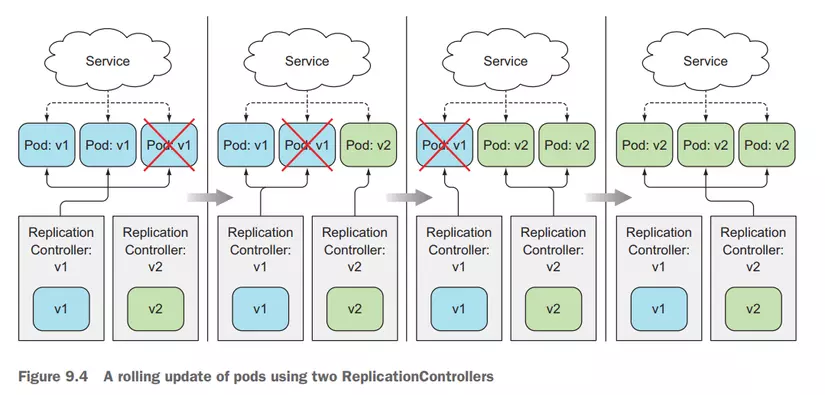
Recreate

Ở cách deploy này, đầu tiên là sẽ xóa toàn bộ version cũ, sau đó ta sẽ deploy version mới. Đối với Kubernetes thì đầu tiên ta sẽ cập nhật Pod template của ReplicaSet, sau đó ta sẽ xóa toàn bộ Pod hiện tại, để ReplicaSet tạo ra Pod với image mới.



Nhưng với cách deploy này, ta sẽ gặp vấn đề rất lớn đó là ứng dụng sẽ downtime với client, client không thể request tới ứng dụng của chúng ta trong quá trình update.

RollingUpdate

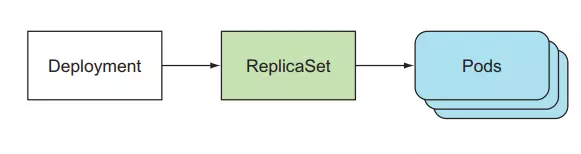
Ở cách này, ta sẽ deploy từng phần của ứng dụng lên, chắc chắn rằng nó đã chạy, ta dẫn request tới version mới của ứng dụng này, lặp đi lặp lại cho đến khi toàn bộ version mới của ứng dụng đã được deploy và toàn bộ version cũ đã bị xóa. Đối với k8s, ta sẽ lần lượt từng pod và ReplicaSet sẽ tạo Pod mới.

Cách deploy này sẽ giảm thời gian downtitme của ứng dụng, còn điểm yếu là version mới và version cũ sẽ chạy cùng một lúc.

Lùi lại phiên bản trước của ứng dụng

Deployment là 1 resource của k8s giúp ta dễ dàng cập nhật version mới của ứng dụng, nó cung cấp sẵn 2 phương thức deploy là recreate và rollingUpdate và giúp ta có thể rollback giữa các phiên bản của ứng dụng.

Khi tạo ra 1 Deployment, nó sẽ tạo ra 1 ReplicaSet bên dưới, và ReplicaSet này sẽ tạo ra Pod, Pod sẽ run container



Để update lại ứng dụng trong Pod, ta chạy câu lệnh sau: kubectl set image deployment<deployment-name> <container-name>=<new-image>

Cách Deployment update Pod

Khi ta thay đổi image của Deployment, nó sẽ tạo ra 1 ReplicaSet khác, ReplicaSet này sẽ giữ template Pod mới, và các Pod mới sẽ được tạo ra bởi ReplicaSet. Và ReplicaSet cũ sẽ không bị xóa đi, mà thuộc tính replicas của nó sẽ được cập nhật lại = 0. Vì trong quá trình update version mới, nếu version mới bị lỗi, ta muốn rollback lại version trước đó, thì ReplicaSet cũ vẫn ở đó, và ta sẽ rollback lại dễ dàng hơn.

Rollback lại version trước khi version mới bị lỗi ta dung lệnh: Kubectl rollout undo deployment <deployment-name> --to-version=<reversion-number>

**Volume**

Volume là một mount point từ hệ thống file của serve vào bên trong container.

Khi ta làm việc với container, những thứ ta ghi vào filesystem của nó chỉ tồn tại khi container còn chạy. Khi một Pod bị xóa và tạo lại, container mới sẽ được tạo ra, lúc này những thứ ta ghi ở container trước sẽ bị mất đi. Nếu ta muốn giữ lại những dữ liệu đó thì ta phải sử dụng volume.

Volume được chia thành 3 dạng chính:

+Volume dung để chia sẻ dữ liệu giữa các container trong Pod

+Volume đính kèm vào trong filesystem một node

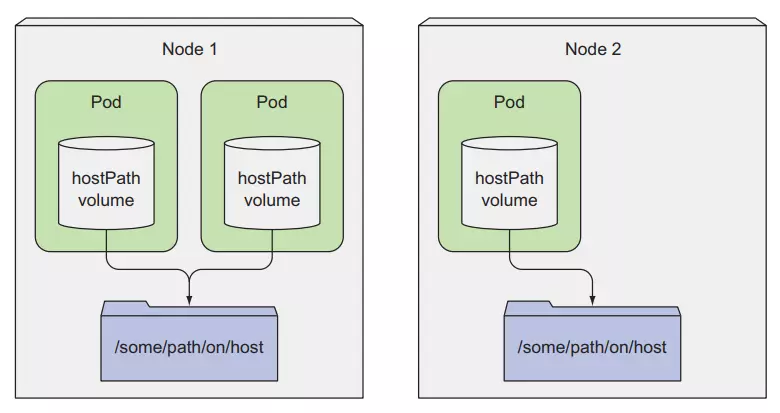
+Volume đính kèm vào cluster và các node khác nhau có thể truy cập

Sử dụng emptyDir volume để chia sẻ dữ liệu giữa các container

emptyDir volume là loại volume đơn giản nhất, nó sẽ tạo ra một empty directory bên trong Pod, các container trong một Pod có thể ghi dữ liệu vào bên trong nó. Empty volume chỉ tồn tại trong một vòng đời của Pod, dữ liệu trong empty volume chỉ được lưu trữ tạm thời và sẽ mất đi khi Pod bị xóa. Ta dung loại volume để các container chia sẽ dữ liệu lẫn nhau và không cần lưu trữ lại dữ liệu.

Sử dụng hostPath để truy cập filesystem của worker node

HostPath volume là loại volume sẽ tạo ra một mount point từ Pod ra ngoài filesystem của node. Đây là loại volume đầu tiên mà có thể dung để lưu trữ persistent data. Dữ liệu lưu trong volume này chỉ tồn tại trên một worker node và sẽ không xóa khi Pod bị xóa.



**PersistentVolumeClaims**

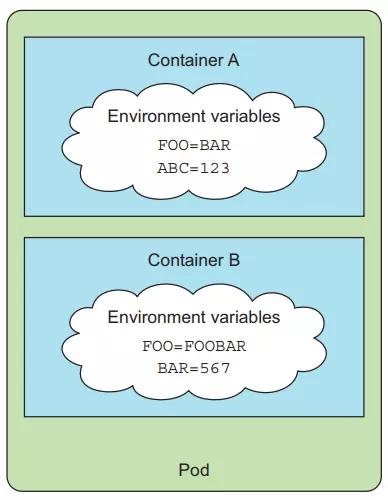
Những loại volume khác đòi hỏi chúng ta phải biết hostPath dẫn tới folder nào của worker node, còn đối với PVC, ta không cần quan tâm đến điều đó, ta chỉ cần quan tâm đến size của volume thôi.

**PersistenVolumeClaims và PesistentVolumes**

PV là resource sẽ tương tác với kiến trúc storage bên dưới và PVC sẽ request storage từ PV.

**ConfigMap và Secret: truyền cấu hình vào container**

Cấu hình của ứng dụng trong một container thường được truyền thông qua biến môi trường(env)



Ta có thể truyền cấu hình Port của ứng dụng vào trong container thông qua env PORT như sau:

apiVersion: v1

kind: Pod

metadata:

name: hello-env

spec:

containers:

- image: 080196/hello-env

name: hello-env

ports:

- containerPort: 3000

env: # pass env to container

- name: PORT # env name

value: "3000" # env value

Dùng env để truyền configuration vào bên trong container rất dễ dàng nhưng khi ứng dụng của ta càng lớn, cần càng nhiều env hơn, thì cách truyền này sẽ có nhiều hạn chế. Khi đó file conrfig của ta sẽ dài dòng, một số env ta sẽ sử dụng đi sử dụng lại nhiều lần, ta sẽ cần khai báo ở nhiều nơi. Ví dụ:

apiVersion: v1

kind: Pod

metadata:

name: hello-env

spec:

containers:

- image: 080196/hello-env

name: hello-env

ports:

- containerPort: 3000

env:

- name: PORT

value: "3000"

- name: POSTGRES\_DB

value: postgres

- name: POSTGRES\_USER

value: postgres

- name: POSTGRES\_PASSWORD

value: postgres

---

apiVersion: v1

kind: Pod

metadata:

name: postgres

spec:

containers:

- image: postgres

name: postgres

ports:

- containerPort: 5432

env:

- name: POSTGRES\_DB

value: postgres

- name: POSTGRES\_USER

value: postgres

- name: POSTGRES\_PASSWORD

value: postgres

Để khắc phục tình trạng đó, k8s cung cấp 1 resource là configMap

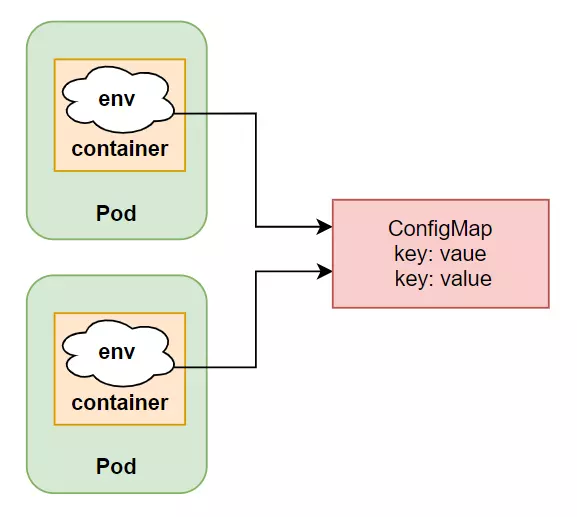
ConfigMap là 1 resource giúp ta tách configuration ra riêng, không cần khai báo trong Pod. Giá trị sẽ được định nghĩa kiểu key/values như sau:

data:

<key-1>: <value-1>

<key-2>: <value-2>

Giá trị này sẽ được truyền vào bên trong container như 1 env. Ta có thể sử dụng configMap cho nhiều container khác nhau.



Ví dụ tạo 1 configMap:

apiVersion: v1

kind: ConfigMap

metadata:

name: postgres-config

data:

DB: postgres

USER: postgres

PASSWORD: postgres

Ở đây ta tạo 1 configMap với 3 key là DB, USER, PASSWORD.

Ví dụ ta truyền configMap vào bên trong container như sau:

apiVersion: v1

kind: Pod

metadata:

name: hello-cm

labels:

app: application

spec:

containers:

- image: 080196/hello-cm

name: hello-cm

ports:

- containerPort: 3000

envFrom: # using envFrom instead of env

- configMapRef: # referencing the ConfigMap

name: postgres-config # name of the ConfigMap

prefix: POSTGRES\_ # All environment variables will be prefixed with POSTGRES\_

env:

- name: PORT

value: "3000"

- name: DB\_HOST

value: postgres

---

apiVersion: v1

kind: Pod

metadata:

name: postgres

labels:

app: db

spec:

containers:

- image: postgres

name: postgres

ports:

- containerPort: 5432

envFrom:

- configMapRef:

name: postgres-config

prefix: POSTGRES\_

---

apiVersion: v1

kind: Service

metadata:

name: postgres

labels:

app: db

spec:

selector:

app: db

ports:

- port: 5432

targetPort: 5432

Ta thấy thay vì ta phải viết lại env ở cả 2 Pod thì ta chỉ cần tạo 1 configMap rồi sử dụng nó nhiều lần.

**Secret**

Secret cũng giống như ConfigMap, dữ liệu được lưu dưới dạng key/value pairs, cách ta sử dụng Secret tương tự như ConfigMap. Secret khác với ConfigMap ở chỗ là nó được dùng để chứa sensitive data, đối với ConfigMap, developer truy cập được vào kubernetes cluster của ta cũng có thể đọc được, còn đối với Secret thì dữ liệu này không phải ai cũng đọc được, phải có quyền mà administrator cho phép, ta mới có thể đọc.

Ta có thể dung CLI để tạo Secret

$ kubectl create secret generic postgres-config --from-literal=DB=postgres --from-literal=USER=postgres --from-literal=PASSWORD=postgres

Khi ta coi thông tin secret ta sẽ thấy nó được lưu ở dạng base64 encode.\

$ kubectl get secret postgres-config -o yaml

apiVersion: v1

data: // base64 encode.

DB: cG9zdGdyZXM=

PASSWORD: cG9zdGdyZXM=

USER: cG9zdGdyZXM=

kind: Secret

metadata:

creationTimestamp: "2021-09-24T10:40:26Z"

name: postgres-config

namespace: default

resourceVersion: "1027803"

uid: 9c7ed846-2ca4-4c58-946a-26cdccce2b1f

type: Opaque

* Sử dụng ConfigMap cho thông tin không nhạy cảm, plain configuration.
* Sử dụng Secret cho thông tin nhạy cảm, dùng CLI chứ không dùng config file.

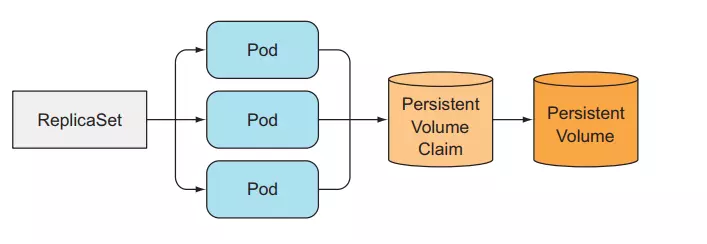
# **StatefulSets: deploying replicated stateful applications**

Stateless application là một ứng dụng mà không có lưu trữ trạng thái của chính nó, hoặc là không có lưu trữ dữ liệu mà cần persistent storage. Ví dụ là một web server API mà không có lưu trữ hình ảnh, hoặc session login của user, thì đó là một stateless apps, bởi vì cho dù ta có xóa ứng dụng của ta và tạo lại nó bao nhiêu lần thì cũng không ảnh hưởng tới dữ liệu của người dùng. Bởi vì dữ liệu của ta được lưu trữ thông qua database, web server API chỉ kết nối với database và lưu trữ dữ liệu, chứ nó không có dữ liệu của chính nó. Một ví dụ nữa là command line app, nó không cần phải lưu trữ dữ liệu gì cả, tất cả những gì nó cần là xuất ra kết quả và không cần phải lưu lại kết quả đó.

Stateful application thì yêu cầu có trạng thái (state) của chính nó, và cần lưu lại state đó, hoặc yêu cầu phải có lưu trữ dữ liệu mà cần persistent storage, dữ liệu này sẽ được sử dụng bởi client và các app khác. Ví dụ như là database, nó sẽ có dữ liệu riêng của nó.

## Hạn chế của việc sử dụng ReplicaSet để tạo replicated stateful app

Vì ReplicaSet tạo nhiều pod replicas từ một Pod template, nên những Pod được replicated đó không khác với những Pod khác ngoại trừ tên và IP. Nếu ta config volume trong Pod template thì tất cả các Pod được replicated đều lưu trữ dữ liệu chung một storage.



### Tạo nhiều ReplicaSet chỉ có một Pod mỗi ReplicaSet

Ta tạo nhiều ReplicaSet và mỗi ReplicaSet đó sẽ có một template Pod khác nhau.

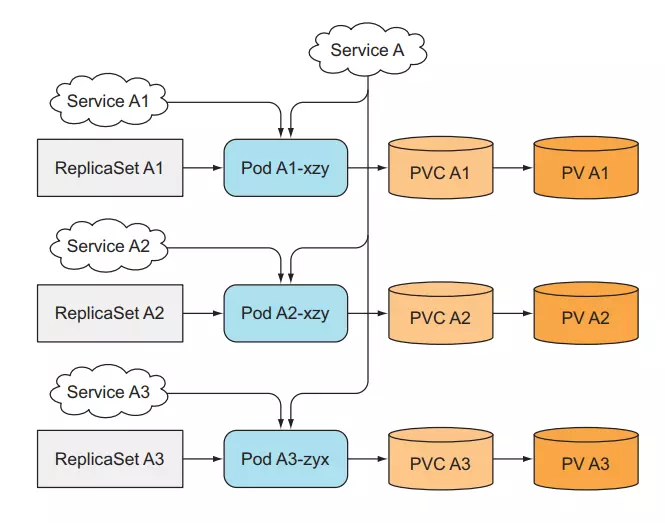
A diagram of several cylinders

Description automatically generated

Ta có thể dụng cách này để deploy một ứng dụng distributed data store. Nhưng đây không phải là cách tốt, ví dụ khí ta muốn scale ứng dụng của ta lên thì ta sẽ làm thế nào? Ta chỉ có cách là tạo thêm một ReplicaSet bằng tay, công việc này không tự động chút nào. Ta chọn kubernetes để chạy ứng dụng là vì muốn mọi thứ như scale đều được tự động và dễ dàng nhất.

### Cung cấp stable identity cho mỗi Pod

Đối với statefull application, ta cần định danh cho mỗi Pod, vì Pod có thể bị xóa và tạo lại bất cứ lúc nào, khi ReplicaSet thay thế một thằng Pod cũ bằng Pod mới thì Pod mới tạo ra sẽ có tên khác và IP khác. Cho dù dữ liệu ta vẫn còn đó và giống như Pod cũ, nhưng đối với một số ứng dụng, khi ta tạo Pod mới mà nó có một network identity mới (như địa chỉ IP) thì sẽ sinh ra nhiều vấn đề. Nên ta cần phải dùng Service để identity IP cho Pod, ta có bao nhiêu ReplicaSet thì ta sẽ cần tạo bấy nhiêu Service tương ứng.



Khi ta có thêm Service, lúc này ta muốn scale lên, bên cạnh việc phải tạo một ReplicaSet mới, bây giờ ta cần phải tạo thêm một Service mới cho thằng ReplicaSet tương ứng nữa, gấp đôi công việc phải làm bằng tay.

Để giải quyết vấn đề trên, k8s cung cấp 1 resource là StatefulSet

## StatefulSets

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 thằng 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.

## 

## Khi một Pod mà được quản lý bởi một StatefulSets bị mất (do bị ai đó xóa đi), thằng StatefulSets sẽ tạo ra một Pod mới để thay thế thằng cũ tương tự như cách làm của ReplicaSet, nhưng thằng Pod được tạo mới này sẽ có tên và hostname giống y như thằng cũ.

## A diagram of a system Description automatically generated

### Cách StatefulSets scale Pod

Khi ta scale up Pod trong StatefulSets, nó sẽ tạo ra một Pod mới được đánh index là số tiếp theo của index hiện tại. Ví dụ StatefulSets đang có replicas bằng 2, sẽ có 2 Pod là <pod-name>-0,<pod-name>-1, khi ta scale up Pod lên bằng 3, Pod mới được tạo ra sẽ có tên là <pod-name>-2.

Tương tự như vậy với scale down, nó sẽ xóa Pod với index lớn nhất. Đối với StatefulSets thì khi ta scale up và scale down thì ta có thể biết chính xác tên của Pod sẽ được tạo ra hoặc xóa đi.

## A diagram of a diagram Description automatically generated

### 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 thằng Pod tạo ra mà có index giống với thằng cũ thì vẫn giữ nguyên storage của nó như không phải tạo ra một thằng storage khác.

## StatefulSets làm được việc đó bằng cách tách storage ra khỏi Pod bằng cách sử dụng [PersistentVolumeClaims](https://viblo.asia/p/kubernetes-series-bai-7-persistentvolumeclaims-tach-pod-ra-khoi-kien-truc-storage-ben-duoi-6J3Zgyeq5mB" \t "_blank). StatefulSets sẽ tạo ra PersistentVolumeClaims cho mỗi Pod và gắn nó vào cho từng Pod tương ứng.

## A diagram of a flowchart Description automatically generated

## 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 thằng Pod bị xóa đi, thằng PersistentVolumeClaims vẫn giữ ở đó và không bị xóa. Để khi ta scale up lại thì thằng Pod vẫn được gắn đúng với PersistentVolumeClaims trước đó để dữ liệu của nó vẫn được giữ nguyên.

## 