



# Kubernetes

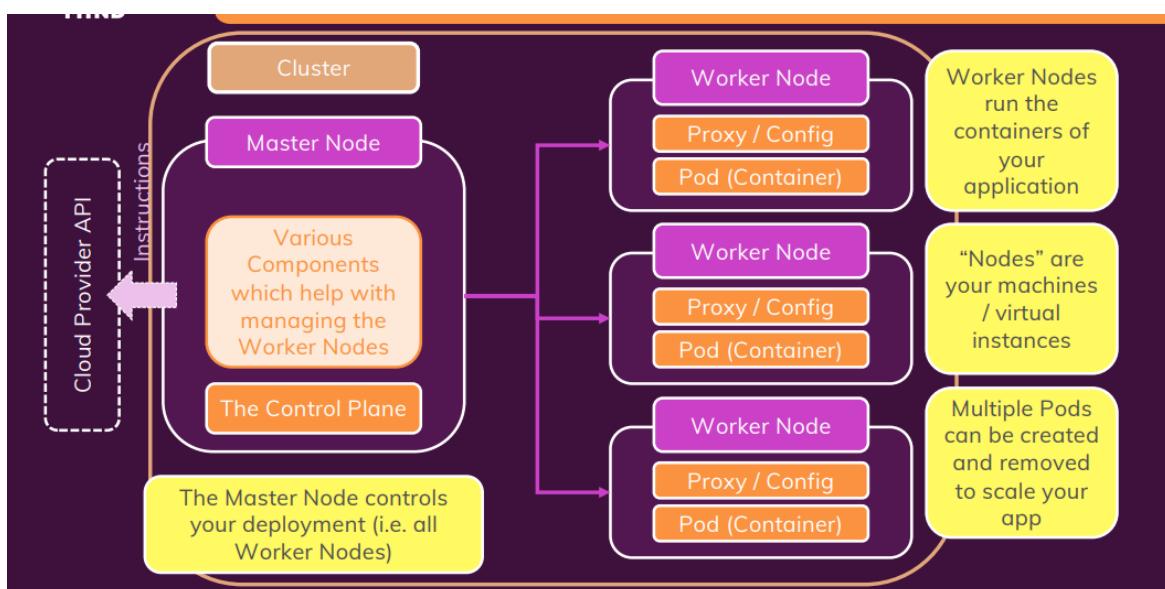
## ▼ Overall

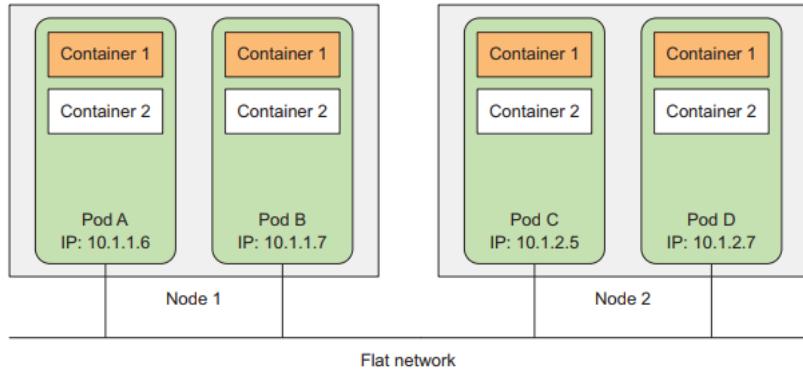
Một hệ thống open-source, giúp quản lý việc triển khai container, xây dựng hệ thống các containers. Hỗ trợ trong Automatic Deployment, Scaling & Load Balancing, Management container.

Cho phép viết Configuration file để định nghĩa cách triển khai container

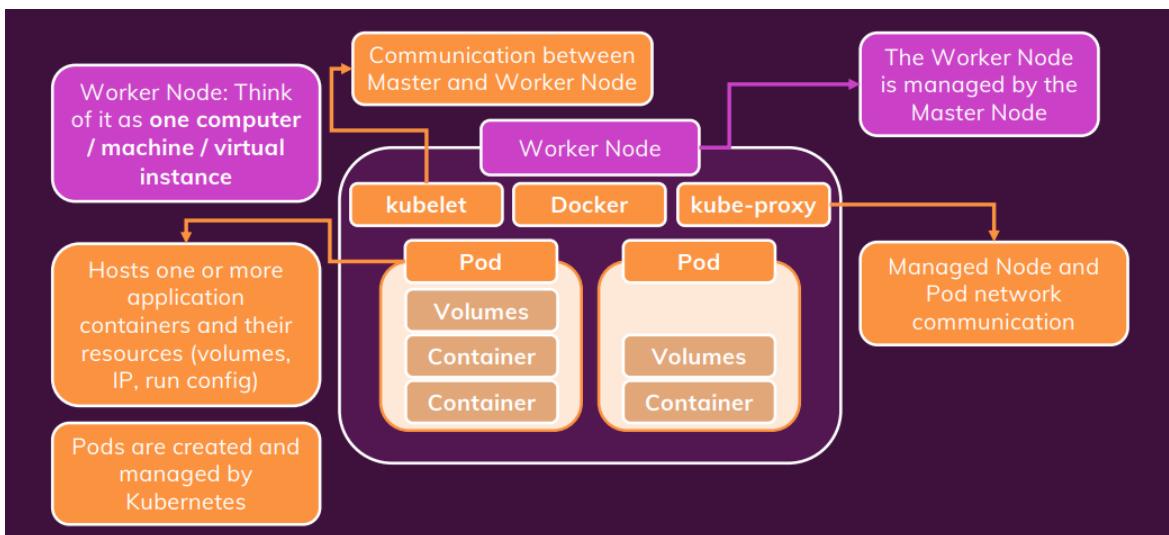
Có thể đưa file config đó cho bất kì Cloud Provider hoặc Remote Machines nào để triển khai.

Có thể kết hợp với các options cụ thể của một Cloud Provider vào trong config file trong trường hợp cần dùng đến.





**Worker node:** Một machine (virtual instance) chạy container. Có thể có nhiều pod chạy trên một Worker node



**Proxy:** một tool của K8s để điều khiển network traffic của các pod

**Master Node:** Trung tâm điều khiển, tương tác và điều khiển các worker node

**Control Plane:** một bộ các tools, các services chạy trên Master Node.

**kubelet:** communication device giữa Master Node và Worker Node, chạy trên Worker Node.

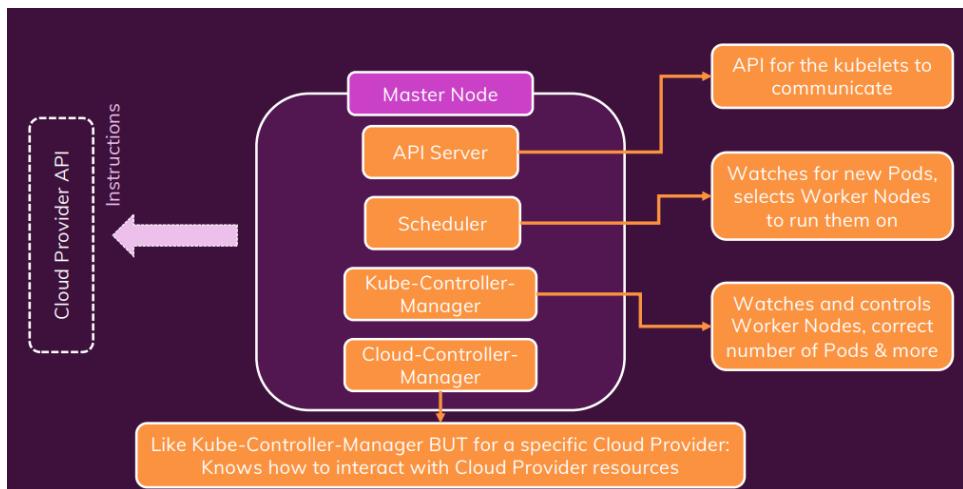
**kube-proxy:** Service kiểm soát traffic vào và ra

**API-Server:** Service chạy trên Master Node, là nơi giao tiếp với kubelet trên Worker Node.

**Scheduler:** Theo dõi các Worker node, trong trường hợp cần đặt các pod, tạo mới, mở rộng...  
(Giao tiếp thông qua API-Server: Scheduler → API-Server → Worker Node)

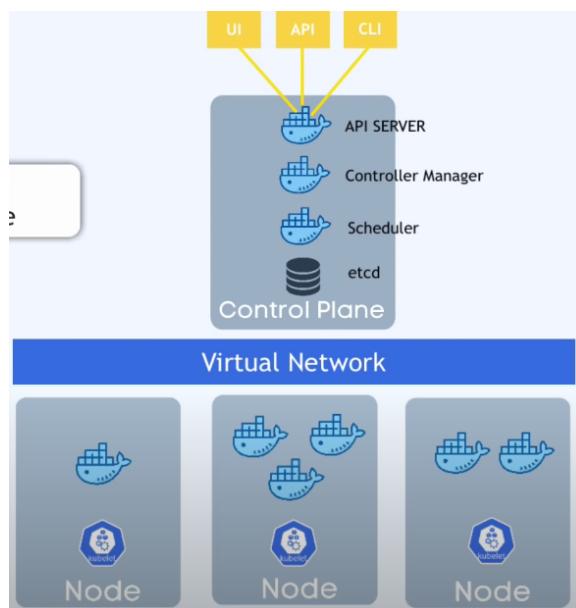
**Kube-Controller-Manager:** Kiểm soát tổng thể các Worker node, đảm bảo số lượng pods được up và running.

**Cloud-Controller-Manager:** Giống với Kube-Controller nhưng dành cho Cloud Provider. Về cơ bản, nó sẽ dịch instructions đến Cloud Provider.



**kubectl:** một tool dùng để gửi instructions đến cluster

#### ▼ Kubernetes Architecture



Các thành phần trong **Master Node**:

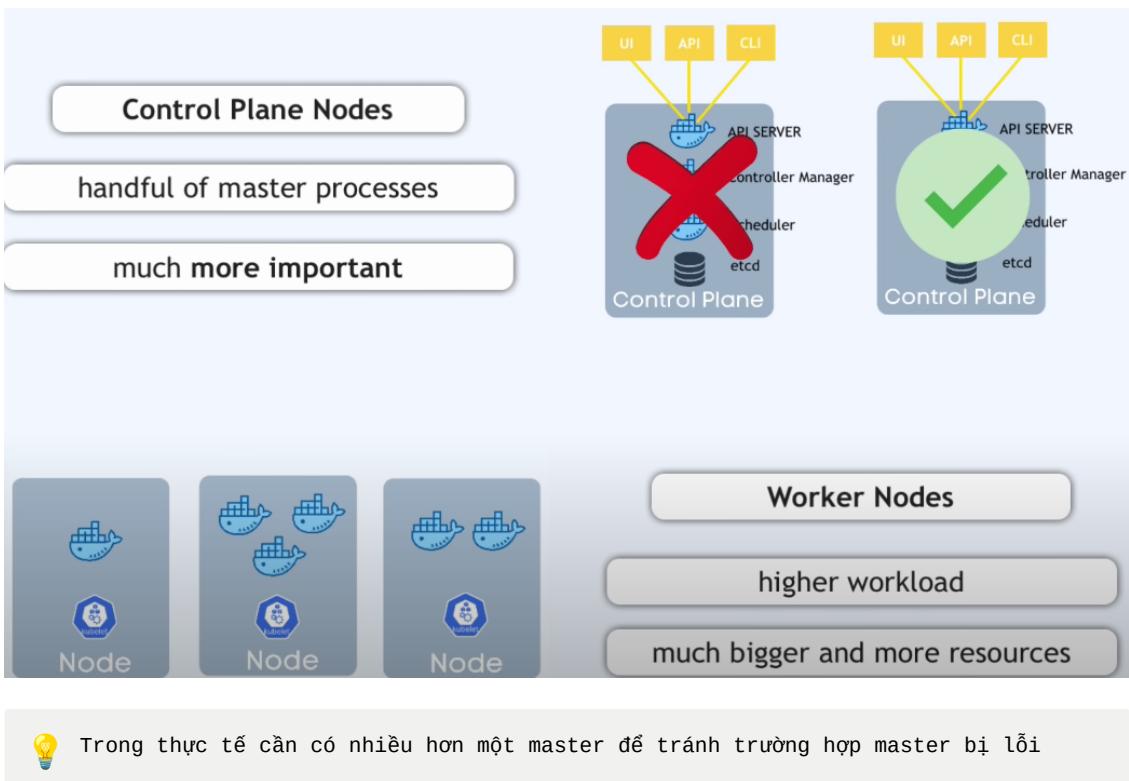
**API Server:** Một container, là entry point đến cluster.

**Controller Manager:** Theo dõi những gì xảy ra trong cluster

**Scheduler:** lập lịch các container ở các node khác nhau dựa trên workload và tài nguyên server. (Một cách để xem container mới được đặt ở đâu dựa trên tài nguyên hệ thống và lượng tải mà container cần)

**etcd:** Nơi lưu trữ cấu hình và trạng thái của mỗi node, mỗi container trong node

**Virtual network:** Một mạng ảo kết nối các thành phần của cluster



💡 Trong thực tế cần có nhiều hơn một master để tránh trường hợp master bị lỗi

#### ▼ Main Kubernetes Components

**Pod:** Thành phần nhỏ nhất trong một K8S cluster, là một thành phần trừu tượng bọc lấy một container (có thể hiểu là cái hộp của K8S bọc lấy một container). Tạo ra một trường chạy hoặc một lớp trên của container. K8S tạo ra pod để trừu tượng hóa hoạt động của container, để có thể thay thế khi cần. Và khiến K8S không cần quan tâm đến công nghệ container được sử dụng. Thường chỉ có 1 app / 1 pod. Mỗi pod có một IP riêng.

💡 namespace: các container trong cùng một Linux namespace (các containers chạy trong cùng một pod, based linux) chia sẻ chung resources

💡 Các container trong cùng một pod chia sẻ các tài nguyên như IP hoặc port → cần tránh tình trạng trùng lặp port

💡 Giữa các pod có thể truy cập trực tiếp đến nhau thông qua IP riêng của từng pod. Không có NAT gateway. Không có vấn đề gì ngay cả khi 2 pod nằm trên các worker node khác nhau

💡 K8S không thể scale container, thay vào đó, nó scale toàn bộ pod

💡 Pod không bền vững. Nếu nó lỗi và tạo ra một pod mới, pod mới sẽ mang IP mới → cần service

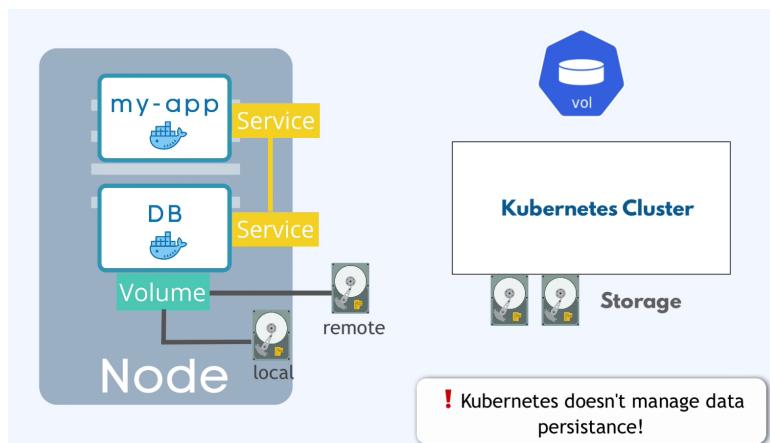
**Service:** Một static IP có thể cấp cho mỗi pod. Tuy nhiên Service tách biệt với Pod, nếu Pod chết, Service vẫn tồn tại. Gồm có External Service (dùng để kết nối từ bên ngoài) và Internal Service. Mặc định sẽ là Internal Service.

**Ingress:** Nơi Request tới đầu tiên rồi mới được chuyển tới service (Một cách tạo ra url)

**ConfigMap:** Là cấu hình bên ngoài của app. Thường chứa những dữ liệu cấu hình như là url của database hoặc một số service khác.. Và trong kubernetes chỉ cần kết nối vào Config Map (Như một dạng tên miền đại diện, tránh phải build lại app khi có thay đổi). Tuy nhiên với những cấu trúc dữ liệu quan trọng như tài khoản và mật khẩu thì không dùng ConfigMap vì lý do bảo mật. Thay vào đó dùng Secret.

**Secret:** Tương tự như ConfigMap nhưng các dữ liệu được lưu được mã hóa. Tuy nhiên, mã hóa không có sẵn trong K8S mà từ bên thứ 3

**Volumes:** Trong trường hợp Database reset → dữ liệu bị mất → giải pháp là dùng volumes. Dữ liệu được lưu trữ ở phần cứng trong local machine, hoặc remote (cloud, server khác...) mà không liên quan gì đến K8s cluster.



#### ▼ Kubernetes Configuration

Kubernetes Configuration được khởi tạo bằng khai báo. Đưa vào API Server dưới dạng file cấu hình yaml hoặc json. Dạng theo cấu hình được coi là desired state. Control Manager sẽ dựa vào đó và so sánh với actual state để duy trì trạng thái, đảm bảo desire state.

#### Các thành phần của K8s configuration file

Mỗi configuration file có 3 phần.

- 1) metadata: bao gồm tên của component
- 2) specification: nơi đưa vào các thiết lập cho component
- 3) status: được tự động tạo ra và chỉnh sửa bởi K8s

etcd giữ trạng thái hiện tại của tất cả các K8s component → thông tin status được lấy từ đây

#### ▼ Imperative

**kubectl create deployment <name> --image=<image-name>:** tạo một deployment object. (Câu lệnh này được gửi đến kubernetes cluster, nên nó phải pull image về → những image từ local sẽ không được đẩy lên)

**kubectl delete deployment <name>:** xóa deployment object

**kubectl get deployments:** trả về thông tin những deployment objects

**kubectl get pods:** trả về thông tin những pod objects.

**kubectl expose deployment <deployment-name> --type=<ClusterIP(default- only reachable from inside)/NodePort(expose deployment với Ip của worker node)/LoadBalancer( tạo một địa chỉ riêng cho Service, loadbalance với những traffic giữa các pods) --port=<port> :** expose pod trong deployment bằng cách tạo ra một service

**kubectl get services:** hiển thị thông tin các service

**kubectl delete service <service-name>**: Xóa service  
**minikube service <service-name>**: truy cập vào service thông qua port được map bởi minikube (bởi vì minikube là một máy ảo chạy trên local machine, thế nên nó chỉ được truy cập từ ngoài mà phải map một port vào một địa chỉ để truy cập).



K8s không tự động tạo hoặc giảm số lượng pods

**kubectl scale deployment/<deployment-name> --replicas=<number-of-replica>**: Scale số lượng pods

**kubectl set image deployment/<name-of-the-deployment> <old-image>=<new-image>**: update image trong deployment



image mới chỉ được download nếu nó có một tag khác

**kubectl rollout status deployment/<deployment-name>**: show trạng thái update

**kubectl rollout undo deployment/<deployment-name> --to-revision=<revision want to go back>**: undo lastest deployment

**kubectl rollout history deployment/<deployment-name>**: hiển thị deployment history --revision=<revision> (thêm revision để hiển thị chi tiết)

**kubectl logs <pod-name>** :show log

-c <container-name> : show log của container trong trường hợp có nhiều container trong pod

#### ▼ Declarative

**apiVersion:**

**kind:** định nghĩa loại Object muốn tạo

**metadata:** đưa ra các thông tin mô tả như tên

**spec:** đưa ra mô tả chi tiết của object

**replicas:** số lượng pod muốn được khởi tạo, mặc định giá trị là 1

**selector:** chỉ ra cho Deployments những pods nào nên được kiểm soát bởi nó

**matchLabels:** Đặt giá trị key:value của pods được kiểm soát

**matchExpression:** Giá trị match được viết theo dạng expression, có thể thêm các operator như IN, NotExist ...

**template:** mô tả về pod

**metadata:**

**labels:**

<key>:value: labels name ở dạng key:value



Deployment Object đã luôn có PodTemplate thế nên không cần define kind ở template.

**spec:** mô tả chi tiết về pod

**containers:** thông tin về các containers, có thể có nhiều containers được list

**-name:** tên của container

**images:** images dùng làm container

**livenessProbe:** health check cho container.

```
...
    preiodSeconds: mức độ check thường xuyên
kubectl apply -f <path-to-file>: apply file yaml
```

 Đối với Service, selector không cần phải define matchLabels.

**type**: cách expose các port ra ngoài

ClusterIP: port chỉ được expose internally

NodePort: Expose Ip và port của Worker Node

LoadBalancer

**kubectl delete -f <path-to-file>**: xóa file và resource được tạo ra bởi file đó

 Trường hợp muốn định nghĩa nhiều Object trong một file duy nhất, phân tách giữa các Object bằng 3 dấu --.

 Các object tích hợp liên tục với nhau. Ví dụ, một service được tạo ra với selector chỉ đến một giá trị cụ thể. Nó không chỉ monitor những object match với giá trị đó đã được tạo ra từ trước. Mà ngay cả những object match được tạo ra sau này. → khi tạo nhiều object trong một file, tốt hơn là nên tạo Service đầu tiên.

 Có thể thêm:

labels:

<key>: <values>

vào phần metadata của object, sau đó có thể xóa object thông qua labels bằng:  
**kubectl delete -l <key>=<values>**

 Với selector, chỉ có thể được chọn bởi labels, chứ không thể được chọn bởi tên, tên (name) không thể được sử dụng.

 Khi thiết lập lại deployment, nội dung app không thay đổi trừ khi đổi tag, bởi vì k8s mặc định chỉ pull tag lastest. Thêm **imagePullPolicy: Always** vào dưới containers để k8s luôn luôn pull mới image thuộc tag đó

- *Metadata* includes the name, namespace, labels, and other information about the pod.
  - *Spec* contains the actual description of the pod's contents, such as the pod's containers, volumes, and other data.
- 

## Creating pods from YAML or JSON descriptors

63

- *Status* contains the current information about the running pod, such as what condition the pod is in, the description and status of each container, and the pod's internal IP and other basic info.

## ▼ Deployment & StatefulSet

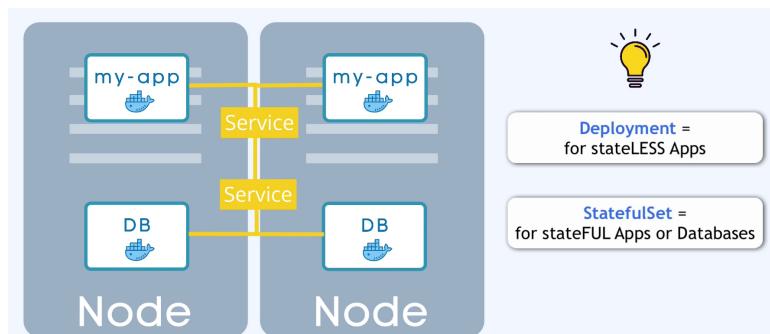


Khi triển khai, ta làm việc với Deployment chứ không trực tiếp với các Pod.

Database không thể nhân lên thông qua Deployment. Bởi vì Database và các bản sao vẫn sẽ phải dùng chung một Data Storage, điều này dẫn đến việc cần một thuật toán xem cái nào đang đọc và cái nào đang ghi Data Storage để tránh dữ liệu không nhất quán.



K8s có một component là StatefulSet đáp ứng thuật toán này. Dùng cho việc tạo bản sao các app đặc thù như database → các database nên được tạo bằng StatefulSet chứ không phải Deployment.



Triển khai trên StatefulSet rất khó, thế nên Database thường được đặt ở bên ngoài K8s cluster

## ▼ Deployment

Khi sử dụng Deployment, các pod được tạo ra được quản lý bởi Deployment's ReplicaSets, không phải trực tiếp bởi Deployment

**Table 9.1 Modifying an existing resource in Kubernetes**

Method	What it does
kubectl edit	Opens the object's manifest in your default editor. After making changes, saving the file, and exiting the editor, the object is updated. Example: kubectl edit deployment kubia
kubectl patch	Modifies individual properties of an object. Example: kubectl patch deployment kubia -p '{"spec": {"template": {"spec": {"containers": [{"name": "nodejs", "image": "luksa/kubia:v2"}]} }}}'
kubectl apply	Modifies the object by applying property values from a full YAML or JSON file. If the object specified in the YAML/JSON doesn't exist yet, it's created. The file needs to contain the full definition of the resource (it can't include only the fields you want to update, as is the case with kubectl patch). Example: kubectl apply -f kubia-deployment-v2.yaml
kubectl replace	Replaces the object with a new one from a YAML/JSON file. In contrast to the apply command, this command requires the object to exist; otherwise it prints an error. Example: kubectl replace -f kubia-deployment-v2.yaml
kubectl set image	Changes the container image defined in a Pod, ReplicationController's template, Deployment, DaemonSet, Job, or ReplicaSet. Example: kubectl set image deployment kubia nodeis=luksa/kubia:v2



Modify ConfigMap sẽ không trigger update. Để trigger update khi chỉnh sửa config là tạo một ConfigMap mới và chỉnh sửa pod template trỏ đến configMap

#### ▼ AWS deployment



K8s biết đang giao tiếp với cluster nào nhờ vào file config bên trong thư mục .kube



Để giao tiếp với EKS cluster, dùng lệnh:

```
aws eks --region ap-southeast-1 update-kubeconfig --name demo
```



Sử dụng instance ở mức micro trong EKS có thể dẫn đến tình trạng pending → mức thấp nhất nên dùng là small

```
kubectl describle <object> <object-name>: hiển thị thông tin cấu hình của một object
```

#### ▼ Update application



Nếu update các phiên bản dùng chung 1 tag, trong đó dùng các tag khác ngoài latest (eg: v1) → cần phải chuyển ImagePullPolicy thành Always để cập nhật phiên bản mới nhất của image. Nếu không sẽ gặp tình trạng các pod dùng phiên bản khác nhau của container.

 Trường hợp dùng latest, hoặc không đề cập tag → imagePullPolicy mặc định là Always (nó sẽ tự động cập nhật bản mới nhất của container). Nhưng trong trường hợp dùng tag khác default sẽ là IfNotPresent → dẫn đến tình trạng nếu container đã tồn tại rồi thì nó sẽ không pull bản mới nhất về mà dùng luôn bản có sẵn → Để tránh thì đổi thành Always

#### Các cách update app:

Xóa các pod cũ → thay bằng các pod mới → có một khoảng downtime

Tạo các pod mới, xóa dần các pod cũ → tăng gấp đôi memory cần.

Vì các pod nằm dưới Service → tạo các pod phiên bản mới → đổi label của service sang bản mới này.

*Rolling update:* scale down dần các bản cũ, scale up dần các bản mới → K8s cho phép thực hiện trong 1 lệnh. → Không thường được dùng, bởi vì k8s sẽ tự thêm label deployment vào các pod được tạo ra, ngoài ra, quá trình được thực hiện bởi kubectl client, dễ xảy ra lỗi nếu mất kết nối. → Thay thế = Deployment

#### ▼ StatefulSet

Cũng có khả năng tạo và quản lý các pod. Nhưng không giống như ReplicaSet, tạo lại pod mới một cách ngẫu nhiên với thông tin hoàn toàn mới. Stateful set có thể tạo lại pod mới với thông tin y hệt pod cũ. Ngoài ra, khi scale, pod mới được tạo ra có thể được đặt trước các thông tin như Ip, name,... Và có thể tạo ra Pod mới với mỗi pod được thiết lập volume lưu trữ riêng. (Stable)

Mỗi pod được tạo ra bởi StatefulSet được thiết lập một index tuần tự. Index được dùng để thiết lập pod's name, hostname, attach stable storage của pod. → những thứ này có thể được thiết lập trước.



Bởi vì trong StatefulSet, mỗi pod không giống nhau → trong một số trường hợp có thể tạo cho chúng một service để có thể truy cập lẫn nhau hoặc có thể truy cập từ client

#### Scaling:

Scale up StatefulSet tạo ra một pod mới với index tiếp theo.

Trong trường hợp scale down, pod với index lớn nhất sẽ luôn bị xóa trước tiên.



StatefulSet scale down chỉ một pod trong một lần. Một data store riêng lẻ có thể mất data nếu nó và các bản copy của nó nằm trên các node down → nếu quá trình scale down là tuần tự, data store có thời gian để tạo thêm các bản replica.



StatefulSet không cho phép scale down nếu có các instance unhealthy

#### Volume:

Khi StatefulSet scale up, nó sẽ tạo thêm 2 hoặc nhiều API object mới là PersistentVolumeClaims và Pod. Với trường hợp Scale down, chỉ xóa mỗi pod, không xóa Claim → để giữ lại PersistentVolume → trường hợp muốn xóa PersistentVolume thì xóa thủ công.

Khi scale up lại, PersistentVolumeClaim có thể được reattach vào pod mới.

```
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.**

 SRV record: một DNS record trong K8s, được dùng để các pod trong StatefulSet có thể discover peer. Để các pod có thể list các pod trong StatefulSet, chỉ cần tạo một SRV DNS lookup

 ResfulSet giống với Replica hơn là Deployment. Cần phải xóa pod cũ đi để có thể update.

 Trường hợp có một node bị tách khỏi Master. Các pod trong node đó sẽ được đánh dấu là unknown, sau một khoảng thời gian (configable) Master sẽ terminate các pod = cách xóa resource pod. Tuy nhiên vì đã bị tách ra nên các pod vẫn chạy bình thường ở node bị tách ra. Nếu nó được kết nối lại sau đó, các pod sẽ trở lại. Tuy nhiên nếu node ko trở lại nhưng vẫn muốn pod hoạt động bình thường → xóa thủ công các pod cũ.

 Xóa thủ công bình thường sẽ không có tác dụng. Bởi vì khi một node bị tách ra, các pod đã được đánh dấu là xóa trước đó. Chính vì vậy mà lệnh xóa tiếp theo sẽ ko có tác dụng. Tuy nhiên lệnh xóa của API server sẽ không hoàn thành do phải đợi thông tin các pod từ kubelet → các pod sẽ bị treo. Để xóa pod, cần thêm option --force --grace-period 0. (Không nên sử dụng cách này trừ khi biết rằng node bị hỏng sẽ không bao giờ quay lại)

```
kubectl delete po <pod> --force --grace-period 0
```

## ▼ Volumes

 **State:** dữ liệu được tạo ra và được sử dụng bởi ứng dụng mà không thể bị mất

K8s có thể mount volumes vào trong containers.

Rất nhiều loại Volume và drives được hỗ trợ: Local volume (một directory trên worker node), Cloud provider specific Volumes,...

Volume lifetime phụ thuộc vào Pod lifetime. Volume có thể tồn tại trong trường hợp Containers gấp lõi hoặc restart, tuy nhiên, bởi vì Volume nằm ngoài container nhưng lại nằm trong pod, thế nên nếu pod bị xóa, volume cũng sẽ bị xóa theo.

 **K8s Volumes:** Hỗ trợ nhiều drivers và Volume Types. Volume không hoàn toàn tồn tại ổn định, khi pod có khả năng bị xóa

**Docker Volumes:** Không hỗ trợ Drive/Types, chỉ đơn giản là một directory trên local machine. Tồn tại ổn định cho đến khi bị xóa đi thủ công.

 Khi định nghĩa volumes trong config file, phải cùng định nghĩa ở nơi định nghĩa pod.

**volumes:** ở cùng cấp với containers, định nghĩa tất cả các containers mà pod sử dụng. Tất cả containers trong pod có thể sử dụng những volumes này.

**emptyDir:[]** - Tạo một directory trống khi pod chạy, directory sẽ được giữ tồn tại và ghi dữ liệu chung nào pod còn tồn tại.

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  containers:
```

```

- image: luksa/fortune
  name: html-generator
  volumeMounts:
    - name: html
      mountPath: /var/htdocs
- image: nginx:alpine
  name: web-server
  volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
  ports:
    - containerPort: 80
      protocol: TCP
  volumes:
    - name: html
      emptyDir: {}

```

The first container is called html-generator and runs the luksa/fortune image.

The volume called html is mounted at /var/htdocs in the container.

The second container is called web-server and runs the nginx:alpine image.

The same volume as above is mounted at /usr/share/nginx/html as read-only.

A single emptyDir volume called html that's mounted in the two containers above

emptyDir được tạo trên ổ đĩa trên worker node hosting pod → hiệu năng phụ thuộc vào loại của ổ đĩa đó. Tuy nhiên có thể tạo emptyDir trên tmpfs filesystem (in memory) thay vì trên đĩa) bằng cách thêm medium.

```

volumes:
  - name: html
    emptyDir:
      medium: Memory

```

This emptyDir's files should be stored in memory.

**volumesMounts:** mount volumes vào container

**-mountPath:** đường dẫn để mount volumes vào

Bởi vì volumes gắn liền với pod, nên trong trường hợp có nhiều replicas, nếu pod chứa volume bị hỏng và traffic được route tới pod khác, dữ liệu sẽ không truy cập được

**hostPath:** tất cả các node chia sẻ chung một directory trên host machine (worker node).

path: path trên host machine

type: đưa ra phương pháp để k8s handle volume

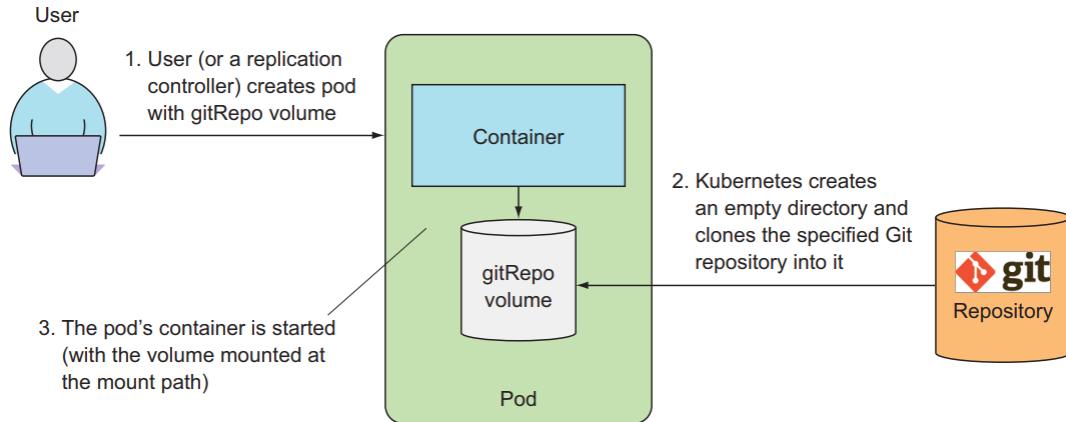
DirectoryOrCreate: tự động tạo khi thư mục chưa tồn tại.

hostPath volumes chỉ được sử dụng khi dùng để đọc/viết systemfile trên node (như là metadata, CA certificate, log...) không dùng để lưu trữ data giữa các pod.

**gitRepo:** một volume được khởi tạo từ nội dung của một git repository

Là một emptyDir volumes nhưng clone một git repository.

volume được tạo trước khi container khởi chạy.



**💡** Volume sau khi clone sẽ không update nội dung trên git Repository nữa. Tuy nhiên nếu pod bị tạo lại → pod mới sẽ clone phiên bản mới nhất trên git repository

```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepo-volume-pod
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
      protocol: TCP
  volumes:
  - name: html
    gitRepo:
      repository: https://github.com/luksa/kubia-website-example.git
      revision: master
      directory: .
```

You're creating a gitRepo volume.

The volume will clone this Git repository.

You want the repo to be cloned into the root dir of the volume.

The master branch will be checked out.

**💡** gitRepo volume không thể clone private git repository. Nếu muốn, cần sử dụng side car git sync hoặc các phương pháp tương tự để thay thế.

nfs: network file system

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

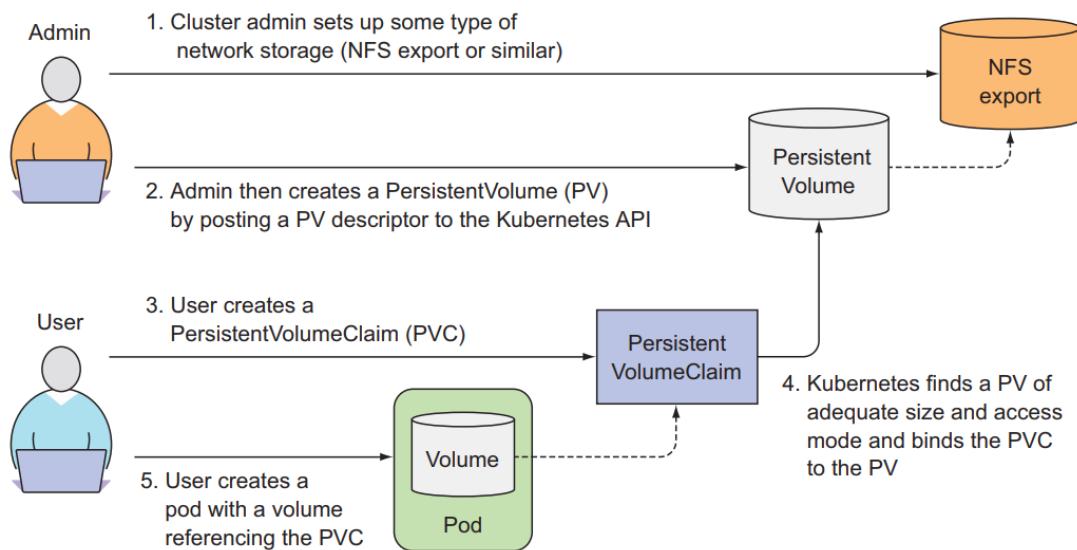
This volume is backed by an NFS share.

The IP of the NFS server

The path exported by the server

**CSI**: cho phép tích hợp với nhiều loại storage nếu chúng có cơ chế tích hợp

**Persistent Volumes**: Dữ liệu được lưu trên một Node tách biệt so với node lưu trữ các pod



**Claim**: dùng để kết nối từ các pods đến persistent volumes

**capacity**: dung lượng mà volume có thể cho pods dùng

**storage**:

**volumeMode**: (Filesystem/Block)

**accessModes**:

- (ReadWriteOnce(tất cả các pod có quyền read, write, nhưng phải trong cùng một node)/ReadOnlyMany/ReadWriteMany)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

**Defining the PersistentVolume's size**  
It can either be mounted by a single client for reading and writing or by multiple clients for reading only.

**The PersistentVolume is backed by the GCE Persistent Disk you created earlier.**

After the claim is released, the PersistentVolume should be retained (not erased or deleted).

Các policy khác là Recycle và Delete.



Đối với Retain, sau khi release khỏi claim, PV sẽ ở trạng thái realeased và không thể claim bởi một claim mới. Cách duy nhất để claim lại là tạo lại PV.

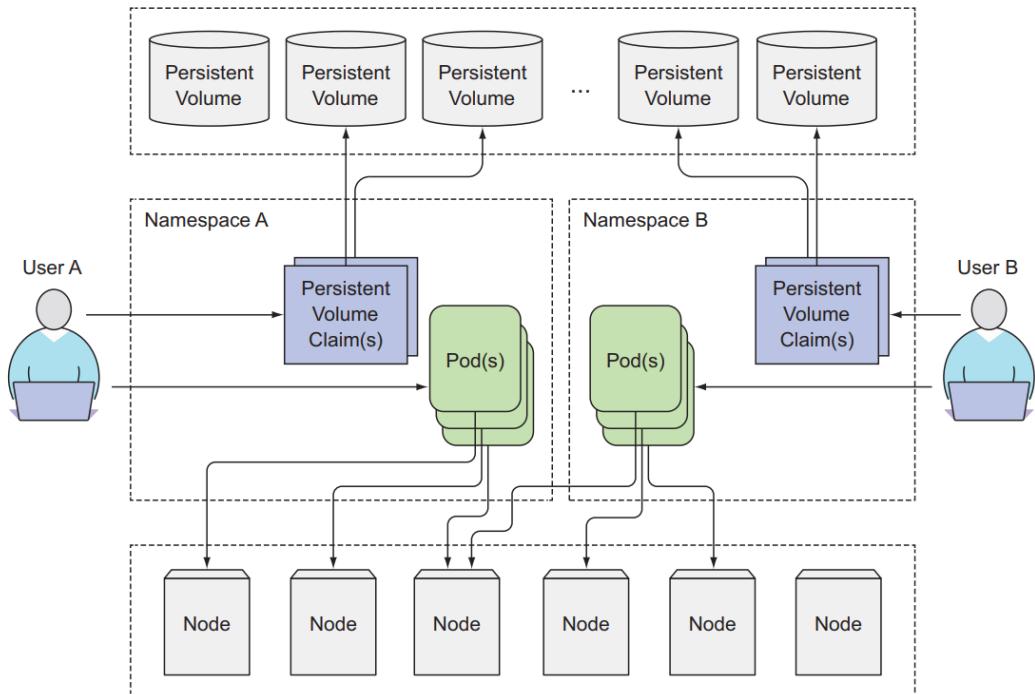


Retain và Delete tồn tại sẵn ở PV, tuy nhiên Recycle thì còn tùy thuộc vào phiên bản và storage

 Trong cấu hình Claim có thể cấu hình claim theo nhiều cách, như claim theo resource để match với nhiều volume.

 Storage Class: định nghĩa trùu tượng để mô tả các lớp lưu trữ khác nhau có thể sử dụng để lưu trữ dữ liệu trong các ứng dụng chạy trên k8s. Cho phép người quản trị có các quyền về cách lưu trữ và thiết lập volume, nó làm việc cùng với Persistent volume

 PersistentVolume không thuộc về một namespace nào, nó ở cluster-level, ngang với node



#### PersistentVolumeClaim:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
```

The name of your claim—you'll need this later when using the claim as the pod's volume.

```

spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: ""

```

**Requesting 1 GiB of storage**

You want the storage to support a single client (performing both reads and writes).

You'll learn about this in the section about dynamic provisioning.

Để claim thì cần match access mode và storage.

access mode:

- RWO - ReadWriteOnce: chỉ một node có thể mount volume để read và write
- ROX - ReadOnlyMany: nhiều node có thể mount volume để read
- RWX - ReadWriteMany: nhiều node có thể mount volume để cả read và write

Use PV in a pod:

```

apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
    - image: mongo
      name: mongodb
      volumeMounts:
        - name: mongodb-data
          mountPath: /data/db
    ports:
      - containerPort: 27017
        protocol: TCP
  volumes:
    - name: mongodb-data
      persistentVolumeClaim:
        claimName: mongodb-pvc

```

**Referencing the PersistentVolumeClaim by name in the pod volume**

Note: Nếu không set storageClassName là rỗng → Provisioner sẽ tạo ra một PV thay vì sử dụng PV được provision thủ công được tạo từ trước, ngay cả khi nó là phù hợp.

```

kind: PersistentVolumeClaim
spec:
  storageClassName: ""

```

Specifying an empty string as the storage class name ensures the PVC binds to a pre-provisioned PV instead of dynamically provisioning a new one.

**kubectl get sc:** get all storage class  
**kubectl get pv:** get all persistent volumes

**env:** biến môi trường.

-name:

```

valueFrom:
  configMapKeyRef:
    name: tên của configmap
    key: keyname
object ConfigMap: map configuration
kubectl get configmap: get all configmap.

```

 Container không thể access files chứa trong volumes, ngay cả khi volume và container là một phần của một pod chung. Như vậy vẫn không đủ để định nghĩa một volumes trong pod. Để làm vậy cần định nghĩa VolumeMount bên trong container spec.

#### ▼ Storage Class:

 Storage Class không có namespace

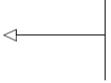
 K8s thường bao gồm provisioners trên hầu hết cloud provider → không cần deploy provisioner. Tuy nhiên nếu K8s là on-premises → cần deploy provisioner.

Thay vì phải provision các Persistent Volumes → chỉ cần define một hoặc hai StorageClasses và để hệ thống tạo PersistentVolume mỗi khi có Request từ PersistentVolumeClaim.

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zone: europe-west1-b

```



The volume plugin to use for provisioning the PersistentVolume

The parameters passed to the provisioner

Tạo PVC để request đến Storage Class:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc

```

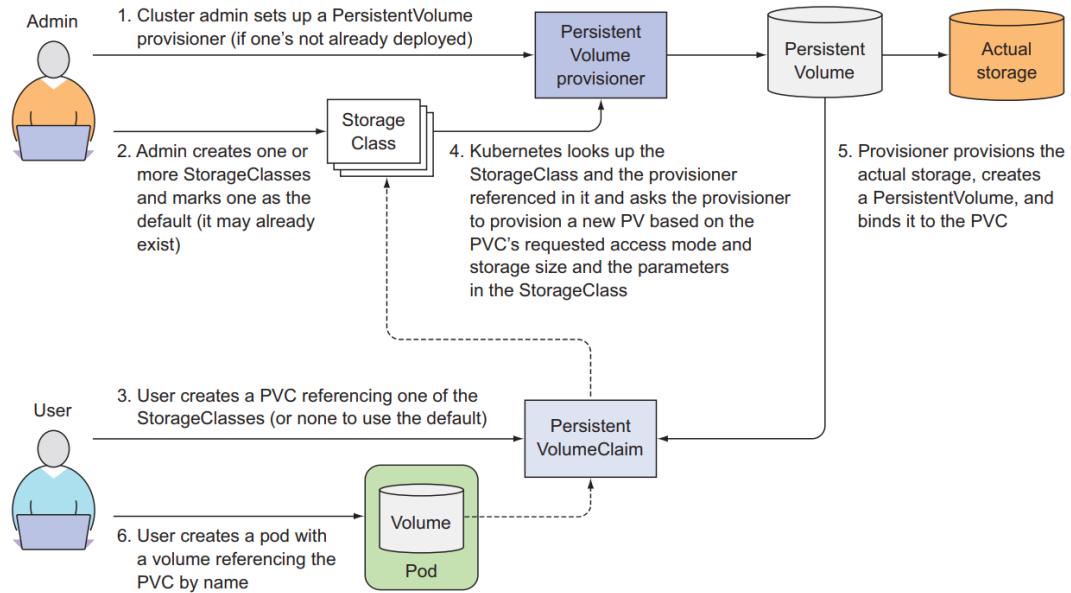
```

spec:
  storageClassName: fast
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce

```



This PVC requests the custom storage class.



**💡** `volumeBindingMode` là một thuộc tính được sử dụng trong `StorageClass` của Kubernetes (K8s) để xác định cách thức ràng buộc (binding) các PersistentVolumes (PVs) cho PersistentVolumeClaims (PVCs).

Có hai giá trị chính cho `volumeBindingMode`:

1. `Immediate`: Khi `volumeBindingMode` được đặt thành `Immediate`, PVCs sẽ được ràng buộc (bind) ngay lập tức với PVs. Điều này có nghĩa là khi một PVC được tạo, Kubernetes sẽ ngay lập tức tìm kiếm một PV phù hợp và ràng buộc PVC với nó. Nếu không có PV phù hợp ngay lập tức, PVC sẽ không được ràng buộc và sẽ chờ cho đến khi một PV phù hợp xuất hiện.
2. `WaitForFirstConsumer`: Khi `volumeBindingMode` được đặt thành `WaitForFirstConsumer`, PVCs sẽ không được ràng buộc với PVs ngay lập tức. Thay vào đó, PV sẽ chỉ được ràng buộc khi PVC đầu tiên được sử dụng (consumer). Khi PVC được sử dụng cho một Pod, Kubernetes sẽ tìm kiếm một PV phù hợp và ràng buộc PVC với nó. Nếu không có PV phù hợp ngay lập tức, Kubernetes sẽ tạo một PV mới và ràng buộc PVC với nó.

## ▼ Network

**💡** Để chứa 2 container trong cùng một pod, không định nghĩa thêm deployment mới. Thay vào đó, thêm định nghĩa container trong cùng deployment.

**💡** Khi có tag: latest, K8s sẽ tự động phát hiện những thay đổi trong container và pull về phiên bản đã được cập nhật.

**💡** Sử dụng `localhost` để thiết lập giao tiếp giữa các container trong cùng một pod.  
(Trường hợp này pod chính là localhost)

Kết nối giữa các pod có thể thực hiện bằng cách thông qua biến môi trường là địa chỉ internal IP giữa các Pod, biến môi trường được khởi tạo tự động bởi k8s, hoặc là DNS được tạo ra.

 K8S có thể tự động tạo các biến môi trường. có dạng (SERVICE\_NAME\_SERVICE\_HOST)(- được thay bằng \_)

**CoreDNS:** DNS được sử dụng internal trong K8s cluster.

**Namespace:** Assign Service, Deployment,... trong cùng một cluster thành một nhóm cụ thể.

Trong environment variable, thay vì thiết lập values là internal Ip, có thể thay bằng internal DNS, được cấu thành theo cấu trúc <service-name>/namespace.

 Add frontend, có thể fetch trực tiếp API từ địa chỉ Service được expose.

 Trong trường hợp frontend nằm trong cùng cluster, vẫn có thể fetch trực tiếp từ Service khác.

 reverse proxy:

Code front-end được thực thi trên browser, trong khi nginx được thực thi bên trong server. Vì vậy, khi cần thực hiện reverse proxy, không thể trả bằng địa chỉ được đưa ra bởi minikube, thay vào đó, có thể dùng internal ip hoặc, internal dns... (ex. trả vào task-service: traffic vào nginx → task service → front-end) gọi từ backend ra → reverse

## ▼ Label

**kubectl label pod <pod-name> <label-key>=<label-value>:** thêm label cho pod đang chạy  
--overwrite: thay đổi label

 Resource phải match toàn bộ label trong selector thì mới được selector nhận diện

 Label không chỉ được gán vào pod, nó có thể được gán vào nhiều nơi, ví dụ là node...  
→ lệnh kubectl label có thể được dùng để gán/sửa label

## ▼ Namespace

Chia hệ thống thành các thành phần tách biệt (đối tượng có thể có nhiều label → chia = label bị chồng chéo không rõ = namespace). Ngoài ra cũng có thể chia theo môi trường (QA, Test, DEV ...)

Tên của tài nguyên chỉ cần độc nhất trong namespace, khác namespace vẫn có thể trùng tên.

```
kubectl get ns : hiển thị các namespace
```

 Khi dùng lệnh kubectl mà không có namespace, k8s sẽ luôn hiển thị các tài nguyên thuộc namespace default

```
kubectl ... --namespace <namespace-name>: hiển thị tài nguyên thuộc namespace cụt thế
```

```
kubectl create -f <file.yaml>: tạo namespace  
kubectl create namespace <namespace's name>
```

Để tạo resource trong namespace:

thêm namespace: <namespace's name> vào phần metadata  
hoặc thêm -n <namespace's name> vào câu lệnh kubectl



current context: namespace mà kubectl làm việc, có thể thay đổi trong kubeconfig

```
kubectl delete ns <namespace's name>: xóa toàn bộ namespace
```



ReplicationController: tự động khởi tạo lại pod

```
kubectl delete all -all: all đầu tiên thế hiện xóa resource ở mọi type (tuy nhiên, một số Certain resource như Secret thì vẫn phải
```

#### ▼ Healthcheck

*liveness probes*: K8s dùng để check container vẫn alive hay không. Có thể chỉ định liveness probes cho mỗi container trong pod's specification. K8s sẽ định kì chạy kiểm tra và restart container nếu fail.

K8s thăm dò container dựa trên 3 phương pháp:

- HTTP GET: tạo một GET request dựa trên Ip, port, path.. của container. Nếu nhận được response (mà không phải error response) → thành công
- TCP socket: tạo một kết nối TCP đến port cụ thể của container. Nếu thành công → probe thành công.
- Exec: thực thi một lệnh bất kì trong container và check status code. Nếu status = 0 → thành công

```
apiVersion: v1  
kind: pod  
metadata:  
  name: kubia-liveness  
spec:  
  containers:  
    - image: luksa/kubia-unhealthy  
      name: kubia  
      livenessProbe:  
        httpGet:  
          path: /  
          port: 8080
```

This is the image containing the (somewhat) broken app.

A liveness probe that will perform an HTTP GET

The path to request in the HTTP request

The network port the probe should connect to



Xem log của container trước khi bị restart

```
kubectl logs <pod> --previous
```

 Luôn phải set initial delay để app có thời gian start, nếu không probe sẽ được chạy quá sớm → fail

 error code 137/ 143 cho thấy container bị ngắt từ bên ngoài

 Nên thiết lập probe để tạo request trên một URL path cụ thể (/health). Đàm bảo /health HTTP endpoint không yêu cầu authentication nếu không probe sẽ luôn fail.

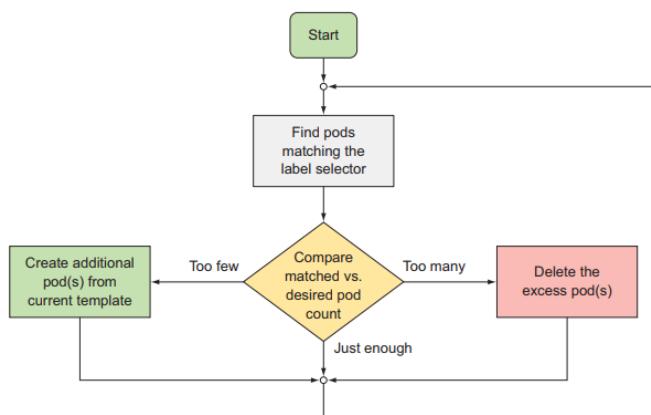
 Chỉ nên đặt probe cho internal check, không đặt với các yếu tố external.

 Trường hợp chạy Java app, dùng HTTP GET liveness probe thay vì Exec. Bởi vì nếu dùng Exec sẽ thành ra tạo một JVM hoàn toàn mới. Điều tương tự với các app sử dụng JVM based hay những ứng dụng tương tự, những cái cần nhiều tài nguyên tính toán để khởi động.

## ▼ ReplicationControllers

ReplicationController là một k8s resource đảm bảo pods luôn được giữ chạy. Nếu một pod biến mất, ReplicationController sẽ nhận ra và tạo lại một pod thay thế.

ReplicationController liên tục monitor danh sách các pod đang chạy và đảm bảo một số lượng pod cùng loại (cùng match label selector) luôn luôn match với số lượng yêu cầu. Nếu quá ít, nó sẽ tạo thêm từ pod templates, nếu quá nhiều, nó sẽ loại bỏ.

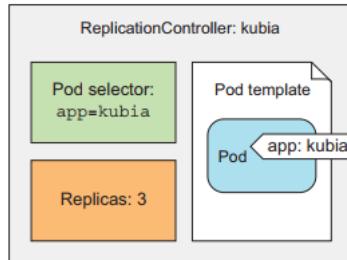


ReplicationController có 3 thành phần chính:

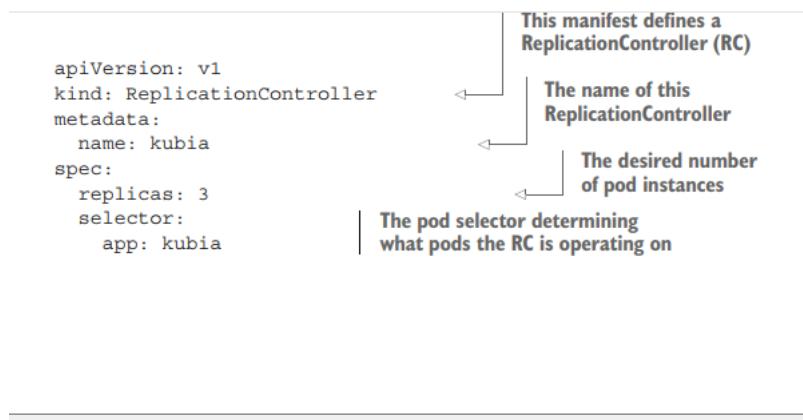
*label selector*: xem xét những pods thuộc quản lý của ReplicationController

*replica count*: đưa ra số lượng pods cần chạy

*pod template*: được sử dụng để tạo pod replicas mới



Các thành phần này có thể thay đổi mọi lúc, nhưng chỉ có thay đổi về replica count có ảnh hưởng đến các pod đang chạy



#### CHAPTER 4 Replication and other controllers: deploying managed pods

```

template:
  metadata:
    labels:
      app: kubia
  spec:
    containers:
      - name: kubia
        image: luksa/kubia
        ports:
          - containerPort: 8080

```

The pod template for creating new pods

Có thể không dùng selector, khi đó ReplicationController sẽ nhận label từ template.

chỉnh sửa scaling của ReplicationController

```
kubectl scale rc <rc-name> --replicas=<number-to-scale>
```

```
kubectl edit rc <rc-name>
```

 Khi xóa ReplicationController các pod sẽ bị xóa theo, tuy nhiên, vì ReplicationController chỉ quản lý các pod, nên có thể xóa mà không ảnh hưởng đến pod

```
kubectl delete rc <rc-name> --cascade=false
```

## ▼ ReplicaSets

Phiên bản mới của ReplicationController, đã thay thế RC.

Chức năng tương tự nhau, tuy nhiên với ReplicationController chỉ cho phép match một label cụ thể. ReplicaSet cho phép match dễ dàng hơn, chỉ cần key, không cần quan tâm đến value.  
→ Khả năng match đa dạng, linh hoạt hơn Replication Controllers

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubia
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
        - name: kubia
          image: luksa/kubia
```

 ReplicaSets aren't part of the v1 API, but belong to the apps API group and version v1beta2.

 You're using the simpler matchLabels selector here, which is much like a ReplicationController's selector.

 The template is the same as in the ReplicationController.

**Listing 4.9 A matchExpressions selector: kubia-replicaset-matchexpressions.yaml**

```
selector:
  matchExpressions:
    - key: app
      operator: In
      values:
        - kubia
```

 This selector requires the pod to contain a label with the "app" key.

 The label's value must be "kubia".

Expression khi sử dụng selector trong ReplicaSet.

- In–Label's value must match one of the specified values.
- NotIn–Label's value must not match any of the specified values.
- Exists–Pod must include a label with the specified key (the value isn't important). When using this operator, you shouldn't specify the values field.
- DoesNotExist–Pod must not include a label with the specified key. The values property must not be specified.

Nếu đặt nhiều expression, tất cả chúng phải là true để có thể match vào pod

## ▼ DaemonSets

Chạy một pod duy nhất trên mỗi node.



Có thể sử dụng Node selector để chỉ định các Node chạy pod.

```
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
        - name: main
          image: luksa/ssd-monitor
```

DaemonSets are in the apps API group, version v1beta2.

The pod template includes a node selector, which selects nodes with the disk=ssd label.

```
kubectl label <node-name> <label>: label node
```

```
kubectl label node minikube <label (key:value)> --overwrite: sửa lại label
```

## ▼ Job

Dùng trong việc chạy các tác vụ ngắn. Sau khi hoàn thành sẽ bỏ pod



Trường hợp Node fail → pod sẽ được reschedule sang node khác.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: batch-job
spec:
  template:
    metadata:
      labels:
        app: batch-job
  spec:
    restartPolicy: OnFailure
    containers:
      - name: main
        image: luksa/batch-job
```

Jobs are in the batch API group, version v1.

You're not specifying a pod selector (it will be created based on the labels in the pod template).

← Jobs can't use the default restart policy, which is Always.

Job có thể tạo nhiều pod và chạy chúng theo cách tuần tự hoặc song song.

```

apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  template:
    <template is the same as in listing 4.11>

```

Setting completions to 5 makes this Job run five pods sequentially.

Với cách chạy tuần tự, mỗi pod sẽ chạy ngay sau khi một cái được hoàn thành. Trường hợp có một pod fail, Job sẽ tạo một pod mới → có thể số lượng pod tạo ra lớn hơn chỉ định.

```

apiVersion: batch/v1
kind: Job
metadata:
  name: multi-completion-batch-job
spec:
  completions: 5
  parallelism: 2
  template:
    <same as in listing 4.11>

```

This job must ensure five pods complete successfully.

Up to two pods can run in parallel.

Cách chạy song song cho phép số lượng pod được khởi tạo và chạy cùng lúc. Khi chúng hoàn thành, pod tiếp theo sẽ được tạo ra.

 Có thể thay đổi số lượng các pod chạy song song khi Job đang chạy (như một cách scale số lượng pod), hoặc dùng lệnh scale:

```
kubectl scale job <job's name> --replicas <number of replicas>
```

 Có thể sử dụng trường activeDeadlineSeconds trong spec của pod để giới hạn thời gian pod được chạy → chạy lâu hơn → Job failed.

 Có thể sử dụng trường backoffLimit trong spec của Job để đưa ra số lần retry trước khi Job bị đánh fail.

## ▼ CronJob

Chạy tác vụ ở thời điểm cụ thể / định kỳ.

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
          restartPolicy: OnFailure
          containers:
            - name: main
              image: luksa/batch-job

```

**API group is batch, version is v1beta1**

**This job should run at the 0, 15, 30 and 45 minutes of every hour, every day.**

**The template for the Job resources that will be created by this CronJob**

**💡** Vì khoảng thời gian là tương đối, có thể có trường hợp pod chạy muộn hơn. Trong trường hợp đó, có thể thêm startingDeadlineSeconds vào spec của CronJob để giới hạn thời gian khởi chạy pod. Nếu quá hạn này, Cronjob sẽ được tính fail

#### ▼ Service

sessionAffinity ở spec của Service, cho phép kết nối đến một pod duy nhất thay vì load balance giữa các pod.

```

apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...

```

Tất cả request từ ClientIp sẽ được tới 1 pod. sessionAffinity chỉ có 2 loại là None và ClientIP.

**Các cách để Client có thể biết được IP được tạo ra bởi Service:**

- Thông qua Environment Variable:

Khi một pod được khởi chạy, k8s khởi tạo một tập các biến môi trường trả đến các service đang tồn tại. Trường hợp tạo Service trước pod, processes trong các pod này có thể lấy ip và port từ service bằng cách kiểm tra biến môi trường.

```

$ kubectl exec kubia-3inly env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=kubia-3inly
KUBERNETES_SERVICE_HOST=10.111.240.1
KUBERNETES_SERVICE_PORT=443
...
KUBIA_SERVICE_HOST=10.111.249.153
KUBIA_SERVICE_PORT=80
...

```

**Here's the cluster IP of the service.**

**And here's the port the service is available on.**

- Thông qua DNS:

<service-name>.<namespace-service-is-defined-in>.svc.cluster.local

**💡** Client vẫn cần phải biết service's port number. Nếu là các standard port như 80,... thì không có vấn đề gì. Nếu không phải thì client có thể lấy port number từ biến môi trường

Có thể bỏ sót phần hậu tố svc.cluster.local hoặc cả namespace nếu các pod ở trong cùng một namespace

**💡** Service không nối trực tiếp đến pod mà thông qua một resource khác là endpoint. endpoint thường là danh sách các IP mà Service nối đến.

Có thể get chúng thông qua kubectl get

Có thể tạo các endpoint thủ công (không tạo selector trong khi define Service).

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  ports:
    - port: 80
```

The name of the service must match the name of the Endpoints object (see next listing).

This service has no selector defined.

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service
subsets:
  - addresses:
      - ip: 11.11.11.11
      - ip: 22.22.22.22
    ports:
      - port: 80
```

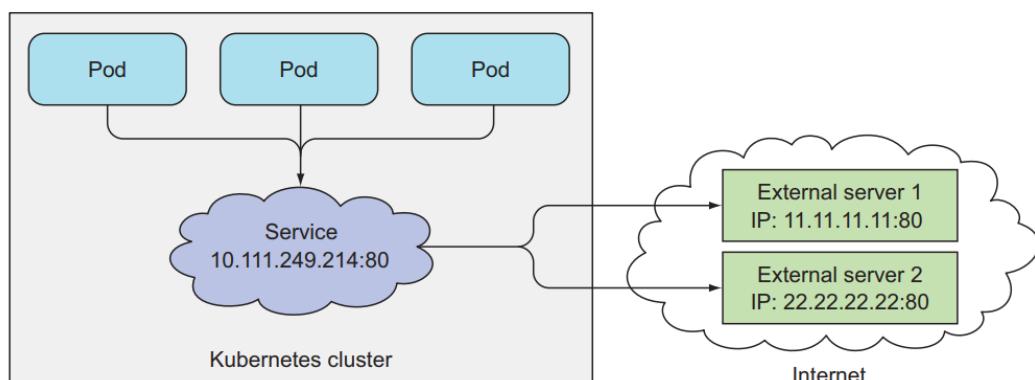
The name of the Endpoints object must match the name of the service (see previous listing).

The IPs of the endpoints that the service will forward connections to

The target port of the endpoints

(Trường hợp này dùng để kết nối đến các external endpoint)

Tên của Service phải giống với tên External Endpoint



Kết nối đến External Service thông qua Service cũng sẽ được loadbalance

### Tạo ExternalName Service:

Tạo một external name làm alias cho Domain. Kết nối đến domain này thông qua service → khi có thay đổi có thể dễ dàng thay thế domain.

External Name hoạt động ở DNS level → kết nối đến service sẽ kết nối trực tiếp đến external service, bỏ qua service proxy. Do đó các loại service này không nhận được cluster IP.

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ExternalName
  externalName: someapi.somecompany.com
  ports:
  - port: 80
```

Service type is set to ExternalName  
The fully qualified domain name of the actual service

### Expose Service:

NodePort: sử dụng NodePort K8s dự trù một port trên tất cả các nodes (tất cả cùng dùng một port cụ thể) và forward kết nối đến các port thuộc service

Để ý rule trên firewall để allow port trong NodePort

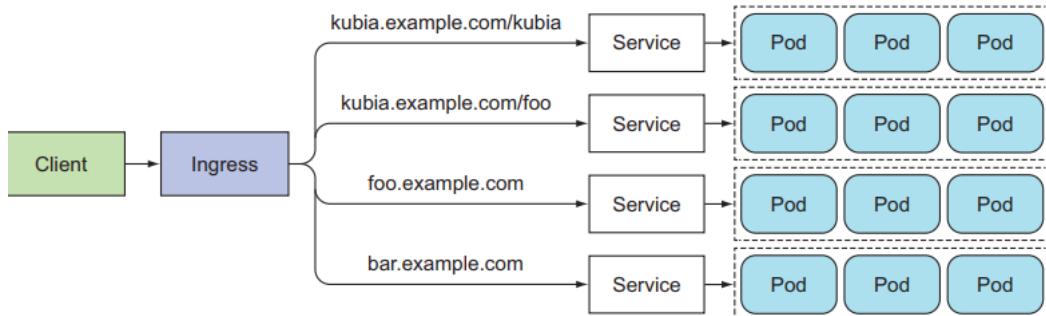
### LoadBalancer

Khi external client kết nối đến service thông qua NodePort hoặc LoadBalancer, có thể pod được kết nối đến lại không nằm trong Node được chọn → sinh ra thêm một hop. Để bỏ qua trường hợp này, ta có thể thêm externalTrafficPolicy: Local trong spec của Service. Trường hợp node đó không có pod nào → sẽ bị hang. Ngoài ra việc dùng annotation có thể khiến LoadBalance không đều.

Trường hợp kết nối từ bên ngoài vào → phải đi qua Node → Source trong Ip sẽ bị thay đổi bởi SNAT. Nhưng nếu dùng annotation externalTrafficPolicy, do không có thêm một hop đứng giữa → SNAT không hoạt động → pod có thể biết client IP.

### Ingress resource:

Ingress có thể cung cấp access cho rất nhiều service. Khi client gửi HTTP request đến Ingress, host và path trong request quyết định service nào được hướng đến.



Ingress hoạt động ở application layer và có thể cung cấp các tính năng như cookie-based session affinity, cái mà service không có.

Để cho Ingress resource hoạt động cần phải có Ingress controller hoạt động bên trong controller,

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  rules:
    - host: kubia.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: kubia-nodeport
              servicePort: 80
```

This Ingress maps the **kubia.example.com** domain name to your service.

All requests will be sent to port 80 of the **kubia-nodeport** service.

 Application chạy trong pod không cần thiết phải hỗ trợ TLS, chỉ cần để cho Ingress controller xử lý các vấn đề liên quan đến TLS. Để kích hoạt khả năng này, cần phải gán certificate và private key vào Ingress → 2 cái này cần phải lưu ở Secret.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  tls:
    - hosts:
        - kubia.example.com
        secretName: tls-secret
  rules:
    - host: kubia.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: kubia-nodeport
              servicePort: 80
```

The whole TLS configuration is under this attribute.

TLS connections will be accepted for the **kubia.example.com** hostname.

The private key and the certificate should be obtained from the **tls-secret** you created previously.

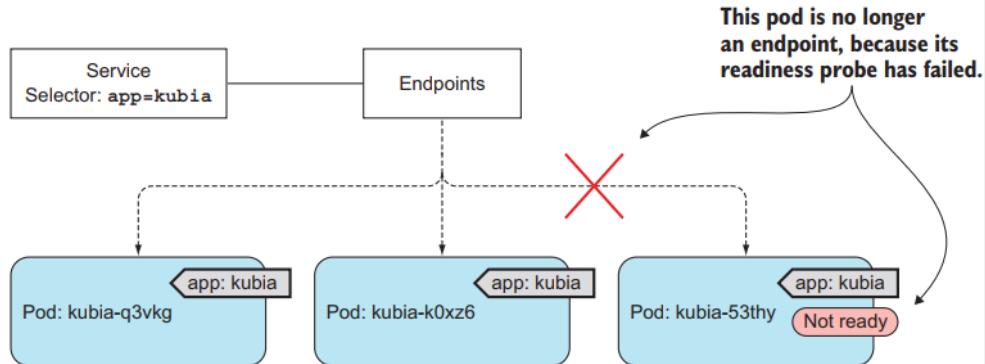
#### readiness probe:

Được chạy định kì để kiểm tra một pod cụ thể có nên nhận request từ client hay không.

Các type của readiness probe tương tự với liveness probe

Khi một container được khởi chạy, K8s có thể được thiết lập để đợi một khoảng thời gian trước khi chạy readiness probe lần đầu tiên. Tiếp đó nó sẽ chạy định kì. Nếu một container được đánh giá là không sẵn sàng, nó sẽ bị xóa khỏi service, nếu đã sẵn sàng, nó sẽ được add lại.

 Khác với liveness probe, nếu một container fail readiness check, nó không bị restart hoặc kill.



**Figure 5.11** A pod whose readiness probe fails is removed as an endpoint of a service.

 Trường hợp muốn xóa thủ công pod khỏi service. Add label enabled=true cho pod và service selector. Nếu muốn xóa pod khỏi service thì xóa label

Trường hợp client muốn connect vào toàn bộ các pod

→ cần IP của toàn bộ pod. K8s cho phép client discover tất cả IP của pod thuộc service thông qua DNS lookup. Thông thường, khi dùng DNS lookup sẽ trả về cluster IP. tuy nhiên nếu set clusterIP field = None trong service spec, DNS sẽ trả về IP của toàn bộ các pod thuộc service. → headless

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-headless
spec:
  clusterIP: None
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

This makes the service headless.

Với headless, do DNS trả về IP của pod, thế nên truy cập từ client sẽ tới trực tiếp pod chứ không thông qua Service proxy.

 Headless vẫn cung cấp khả năng loadbalance giữa các pod, tuy nhiên thông qua DNS roundrobin chứ không phải qua Service proxy

**Discovering all pod - cả sẵn sàng và không sẵn sàng**

Để có thể tìm thấy tất cả các pod match với service selector, ngay cả khi chúng đã sẵn sàng hay chưa thì thêm annotations: service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"

```

kind: Service
metadata:
  annotations:
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"

```

 Một service spec mới là publishNotReadyAddress được dùng để thay thế ở các phiên bản mới

#### Quy trình debug nếu không kết nối được vào service:

Đảm bảo rằng đang kết nối vào service cluster IP bên trong cluster, không phải từ bên ngoài.

Không ping đến Service IP để kiểm tra khả năng kết nối. (nó chỉ là virtual IP)

Nếu có readiness probe đảm bảo rằng nó chạy ổn → pod là một phần của service.

Để kiểm tra pod có phải là một phần của service hay ko, kiểm tra pod endpoint.

Nếu kết nối đến service bằng FQDN hoặc một phần của nó mà không được, thử chuyển sang kết nối bằng cluster IP.

Kiểm tra port kết nối là port được expose bởi service (không phải target port)

Kết nối trực tiếp vào pod bằng pod IP để xác nhận khả năng kết nối

Nếu không được, đảm bảo rằng app đang chạy không chỉ hoạt động trên localhost.

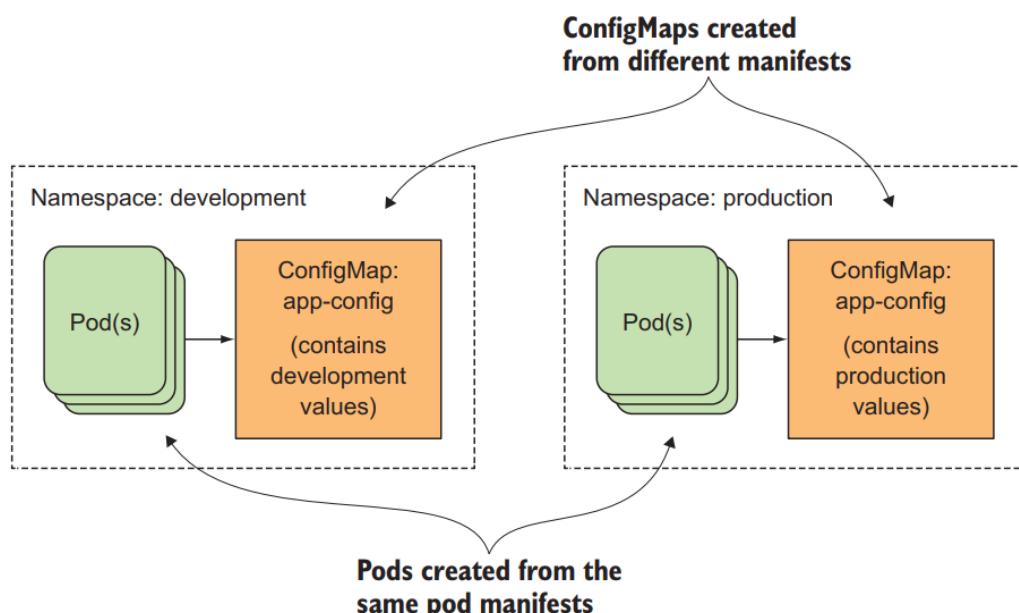
#### ▼ ConfigMap and Secrets

 Trường command với args trong containers tương đương với ENTRYPPOINT và CMD trong docker

Lưu trữ configuration một cách riêng biệt.

Nội dung config được đưa vào container như các biến môi trường hoặc files trong một volume.

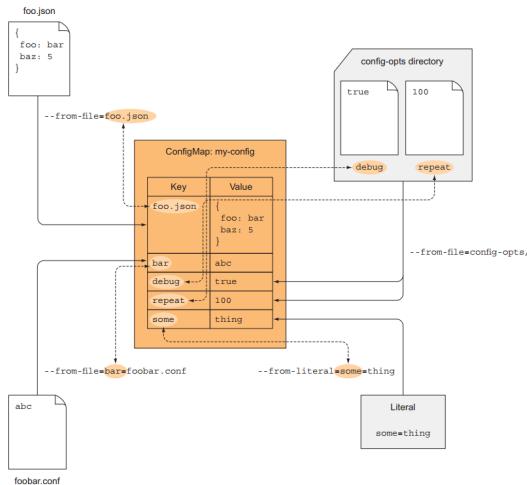
App có thể đọc được nội dung ConfigMap trực tiếp thông qua K8s API, tuy nhiên không nên dùng trừ trường hợp thật sự cần.



Config tách biệt so với pod → có thể thiết lập nhiều phiên bản config cho các môi trường khác nhau. Bởi vì pod tham chiếu đến ConfigMap bằng tên → chỉ cần giữ tên, thay nội dung cho các môi trường khác nhau.

ConfigMap có thể được tạo thành từ:

```
key=values config  
config from file  
multiple config files from a directory  
combine options
```



#### ▼ Accessing pod metadata

Dùng Downward API để expose metadata của pod/container cho process chạy bên trong.

Có 2 cách là expose theo dạng Biến môi trường / hoặc theo dạng file và mount vào một volume downward.

**💡** Với các metadata là label và annotations của pod. Chỉ có thể dùng theo dạng file. Bởi vì label và annotations có thể được thay đổi khi pod đang chạy → K8s sẽ update → những biến môi trường sẽ không update được.

**💡** Ngoài ra, số biến môi trường có thể được sử dụng trong Downward API tương đối hạn chế, một số trường hợp phải truy cập đến API server của K8s (thông tin về các pod khác / tài nguyên khác được defined)

**💡** K8s sử dụng Https để giao tiếp → có authentication → sử dụng kubectl proxy để chạy một proxy → không cần phải thêm token mỗi lần truy cập đến API server.

**💡** Từ pod, muốn truy cập đến API server phải có thêm những bước như tìm API server, Dùng certificate và Authorized. Tuy nhiên, một cách khác là chạy một container có chức năng tương tự như kubectl proxy trong cùng một pod. Container này sẽ lo liệu authentication.

#### ▼ Install

## ▼ Helm

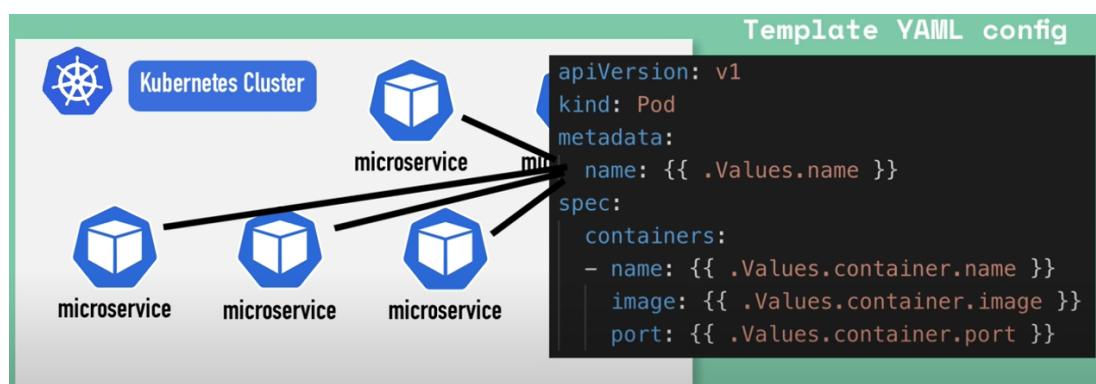
Helm có hai tính năng chính:

**Packet Manager cho Kubernetes:** Đóng gói các file yaml và phân phối đến public/private repository

Tập hợp những yaml files được gói lại với nhau được gọi là Helm Chart



**Template Engine:** tạo ra một template để thiết lập các service giống nhau. Những phần khác nhau sẽ được thay thế bởi placeholders

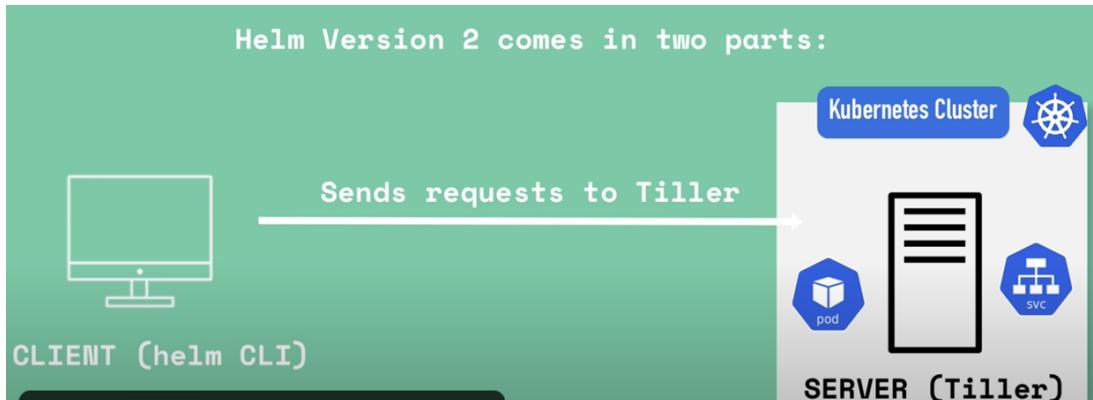


Giá trị của những placeholders đến từ file bên ngoài (ex: values.yaml)



```
helm show values [chart name]: show all parameters
```

**Release Management:** Helm version 2 được chia thành 2 phần, Client (Helm CLI) và Server (Tiller). Mỗi khi Client gửi request (helm install..) cho Tiller, nó sẽ lưu lại bản cấu hình của lần gửi đó. Sự thay đổi sẽ được apply đến Deployment đã tồn tại thay vì tạo một cái mới, chính vì thế có thể biết được phiên bản, ngoài ra có thể roll back



**💡** Bởi vì Tiller có quá nhiều quyền bên trong K8s cluster → gây ra vấn đề bảo mật, ở Helm version 3 đã loại bỏ Tiller → giải quyết vấn đề bảo mật nhưng gây khó khăn trong quản lý

#### ▼ Helm Chart Structure:

Chart được tạo thành từ cấu trúc Directory

#### Directory structure:

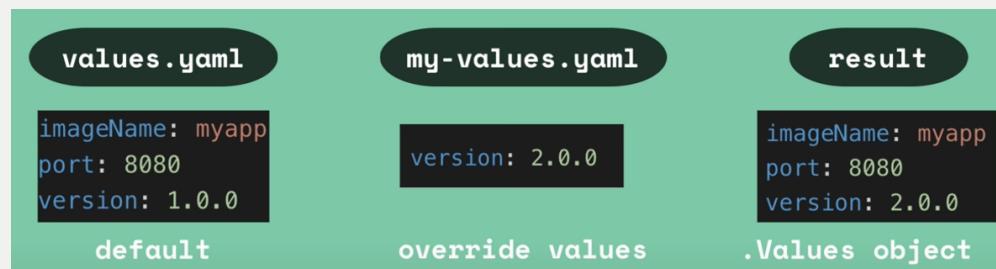
```
mychart/  
  Chart.yaml  
  values.yaml  
  charts/  
  templates/  
  ...
```

Top level mychart folder ➔ name of chart  
Chart.yaml ➔ meta info about chart  
values.yaml ➔ values for the template files  
charts folder ➔ chart dependencies  
templates folder ➔ the actual template files

**💡** Template sẽ được áp những giá trị trong values.yaml khi dùng lệnh **helm install <chartname>** để deploy lên kubernetes



Có thể dùng lệnh `helm install --values=<values-file.yaml> <chartname>` để override giá trị trong file values.yaml mặc định. Hoặc cũng có thể dùng flag để thay đổi một giá trị trong values `helm install --set <value>=<>`



[Install K8s Cluster](#)

[rancher/local-path provisioner](#)

[Full Lab](#)