Kubernetes offre le capacità di orchestrazione e gestione dei container, in modo scalabile, al fine di gestire i carichi di lavoro. L'orchestrazione di Kubernetes consente di creare applicativi su più container, programmare tali container in un cluster di macchine fisiche o virtuali, gestirne la scalabilità e l'integrità nel tempo. Kubernetes permette di migliorare notevolmente la sicurezza IT.

Con Kubernetes puoi:

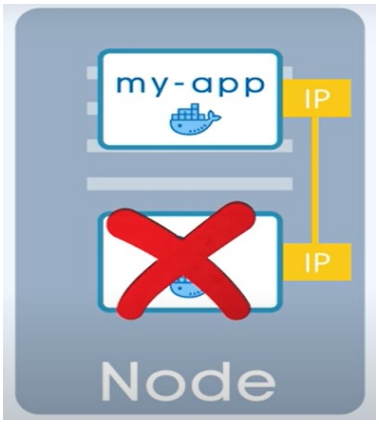
- Orchestrare i container su host multipli.
- Sfruttare meglio l'hardware per massimizzare le risorse necessarie al fine di gestire le app.
- Controllare e automatizzare i deployment e gli aggiornamenti delle applicazioni.
- Montare e aggiungere storage per eseguire app stateful.
- Gestire rapidamente la scalabilità delle applicazioni containerizzate e delle loro risorse.
- Gestire in maniera aperta i servizi, garantendo il deployment delle applicazioni secondo le modalità di deployment stabilite.
- Controllare lo stato di integrità delle applicazioni e gestire le correzioni con posizionamento, riavvio, replica e scalabilità automatici.

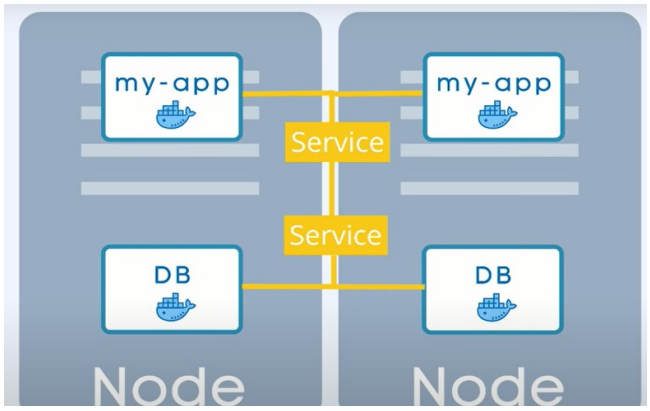
COMPONENTI KUBERNETES

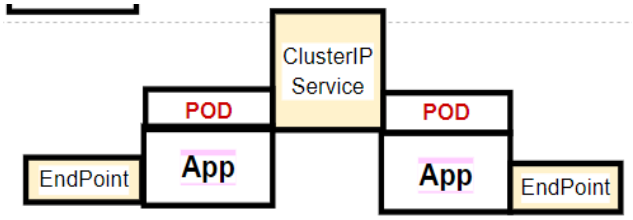
I **Pod** sono l'unità più piccola in kubernetes, è un ambiente di esecuzione o possiamo dire un livello di astrazione di un container, in cui sono eseguite applicazioni. Kubernetes fa questa astrazione dei container in maniera tale da rimpiazzarli facilmente.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

E' possibile eseguire più container, quindi più applicazioni, in un pod ma è buona pratica avere un applicazione per ogni pod. Perché ogni contenitore in un pod viene eseguito sullo stesso nodo e non si possono arrestare o riavviare in maniera indipendente. Oppure, nel caso avessimo due contenitori che comunicano sullo stesso pod, se solo il primo si arrestasse, bisogna riavviare/ripristinare il pod e quindi va riavviato anche il secondo contenitore nonostante sia funzionante. Kubernetes crea automaticamente ogni pod crashato e gli assegna ogni volta un **endpoint** e una numero **porta** diversi. In figura osserviamo due pod chiamati my-app e DB con le loro repliche, nel momento in cui la prima replica di my-app va down, viene usata la seconda replica mentre la prima viene ripristinata da kubernetes.







Un **Service** è un'astrazione che definisce un set logico di pod e una policy mediante la quale accedervi. Quindi un servizio può avere più pod associati. Un servizio funziona come proxy per i pod replicati e le richieste di servizio possono essere bilanciate tra i pod.

I FILE MANIFEST

Tutti i processi e i componenti citati precedentemente lavarono in modo autonomo grazie a kubernetes, il compito del programmatore inizialmente è determinarne lo stato desiderato del cluster kubernetes. Per definire uno stato desiderato, vengono utilizzati file **JSON o YAML** (chiamati manifest) per specificare il tipo di applicazione e il numero di repliche necessarie per eseguire il sistema. In questi file definiamo i pod, i service, i persistent volumes ecc. Gli sviluppatori utilizzano l'interfaccia della riga di comando (kubectl) o sfruttano l'API per interagire direttamente con il cluster per impostare manualmente lo stato desiderato. Il nodo master comunicherà quindi lo stato desiderato ai nodi worker tramite l'API. Kubernetes gestisce automaticamente i cluster per allinearli allo stato desiderato tramite il controller Kubernetes. Ad esempio, se distribuisci un'applicazione da eseguire con cinque repliche e una di queste si arresta in modo anomalo, il controller registrerà l'arresto anomalo e distribuirà una replica aggiuntiva in modo da mantenere lo stato desiderato di cinque repliche.

Se lo stato desiderato dichiarato (esempio; pod con 3 repliche) nel file yaml, non corrisponde allo stato effettivo, kubernetes provvede a ripristinare lo stato.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Esempi di file yaml per definire Service, Deployment, Config-Map e Secrets.

nginx-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:…
spec:
  replicas: 2
  selector:…
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16
        ports:
        - containerPort: 8080
```

Nel file **Service yaml** in generale abbiamo 3 attributi: **port** ossia il numero porta del servizio riconosciuta dagli altri componenti del cluster, **targetPort** è la porta del container (situato all'interno del pod) a cui si vuol far riferimento, possiamo avere anche **nodePort** che è la porta che viene esposta fuori cioè visibile esternamente al cluster. Osserviamo che il targetPort è lo stesso del containerPort del file deployment.

I **Deployment yaml** sono un astrazione dei pod, in cui è possibile definire il numero di replica del pod per gestire lo scale up e scale down automatico delle repliche. Quindi qui definiamo lo stato desiderato. Osserviamo replicas 2, in questo caso kube mantiene lo stato del cluster sempre con due pod nginx:1.16.

Importate precisare che il Deployment e il relativo servizio possono essere scritti nello stesso file yaml, separando le due configurazioni con tre trattini ---

Supponiamo di avere un pod my-app e un pod database, my-app comunica con il database tramite il servizio del database, configuriamo l'URL database o endpoint tramite il file my-app nello spazio delle variabili di ambiente (env), ma se il nome del servizio database cambia dovremmo regolarlo nuovamente nel file di my-app per poi rifare il rebuild, push app nel repository, pull image nel pod e restart il tutto. Siccome può essere un po' tedioso fare ciò, usiamo il file **config-map** che

è una configurazione esterna per l'applicazione my-app. Nel file configuriamo l'URL database e altri dati, in questo caso se cambiamo il nome del servizio non bisogna rifare il rebuild ecc.

Fare una configurazione esterna può essere definire usr e pwd del DB, i quali possono anche cambiare nell'applicazione, ma inserirli con un testo in chiaro potrebbe essere insicuro per questo motivo kube si affida ad un altro componente chiamato **Secrets** che usa un formato di codifica dei dati base64. Possiamo non definire configMap e Secret ma definire i dati da usare nel campo variabili di ambiente. In figura i 4 file yaml.

```yaml
image: mongo-express
ports:
- containerPort: 8081
env:
- name: ME_CONFIG_MONGODB_ADMINUSERNAME
  valueFrom:
    secretKeyRef:
      name: mongodb-secret
      key: mongo-root-username
- name: ME_CONFIG_MONGODB_ADMINPASSWORD
  valueFrom:
    secretKeyRef:
      name: mongodb-secret
      key: mongo-root-password
- name: ME_CONFIG_MONGODB_SERVER
  valueFrom:
    configMapKeyRef:
      name: mongodb-configmap
      key: database_url
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongodb-configmap
data:
  database_url: mongodb-service
```

```yaml
      - name: mongodb
        image: mongo
        ports:
        - containerPort: 27017
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongodb-secret
              key: mongo-root-username
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongodb-secret
              key: mongo-root-password
---
apiVersion: v1
kind: Service
metadata:
  name: mongodb-service
```

```yaml
apiVersion: v1
kind: Secret
metadata:
    name: mongodb-secret
type: Opaque
data:
    mongo-root-username: dXNlcm5hbWU=
    mongo-root-password: cGFzc3dvcmQ=
```

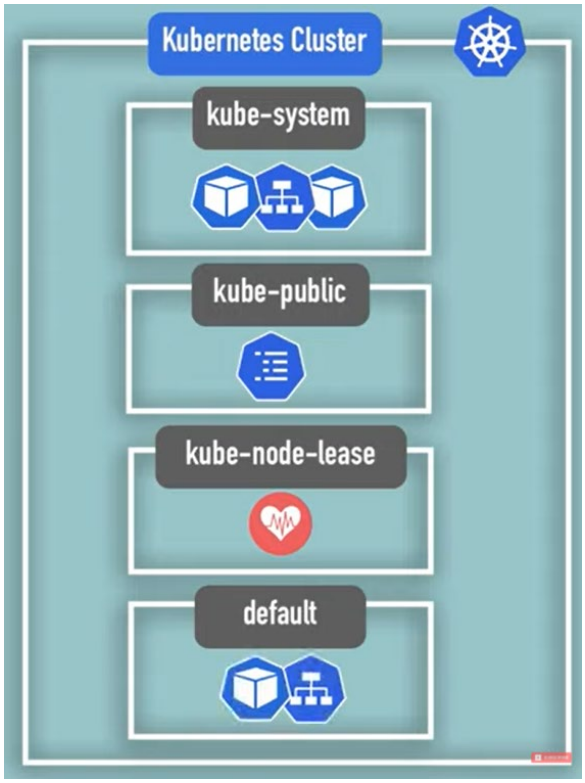
Un file **ConfigMap** è un oggetto API utilizzato per archiviare dati non riservati in coppie chiave-valore. Il pod può consumare ConfigMaps come variabili di ambiente, argomenti della riga di comando o come file di configurazione in a volume. Un ConfigMap ti consente di separare la configurazione specifica dell'ambiente dalla tua immagini del contenitore, in modo che le applicazioni siano facilmente trasportabili.

Il file **Secrets** è un oggetto che contiene una piccola quantità di dati sensibili come una password, un token o una chiave altrimenti essere inserite in un Sotto specifica o in a immagine del contenitore. L'utilizzo di un secrets significa che non è necessario includere dati riservati nel codice dell'applicazione. Poiché i secrets possono essere creati indipendentemente dai pod che li utilizzano, c'è meno rischio che i secrets(e i relativi dati) vengano esposti durante il flusso di lavoro di creazione, visualizzazione e modifica dei pod.

ARCHITETTURA DI UN CLUSTER KUBERNETES

Un cluster Kubernetes (K8s) è un insieme di nodi che eseguono applicazioni containerizzate e consente l'esecuzione dei container su più macchine e ambienti: virtuali, fisici, basati su cloud e on-premises.

Un **namespace** è un modo per un utente Kubernetes di organizzare molti cluster diversi all'interno di un solo cluster fisico. Quando creo il cluster di default kube crea 4 ns:

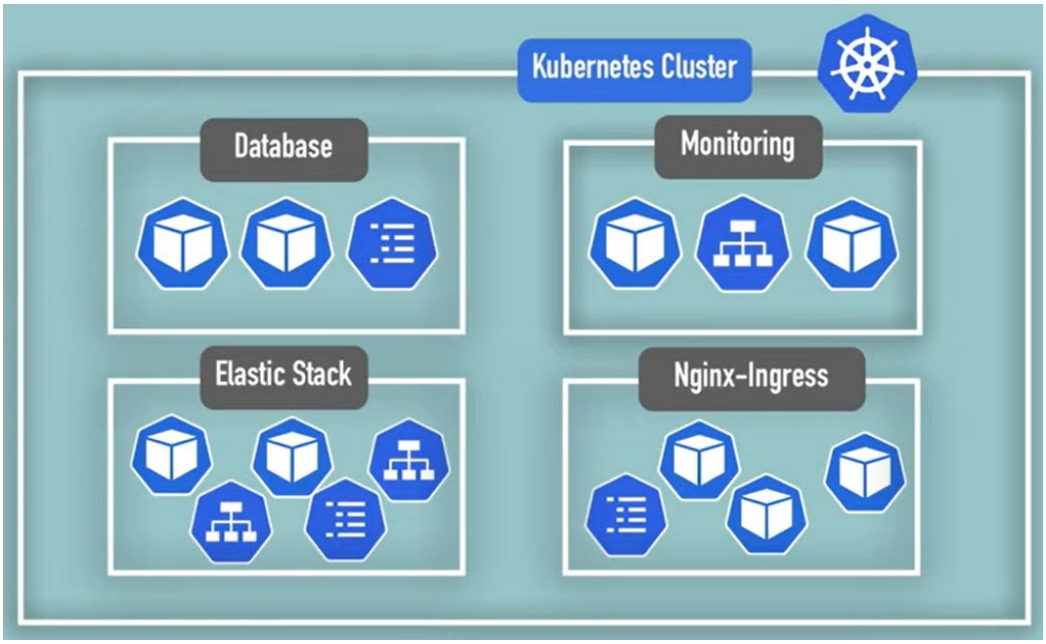


- kube-system: Lo spazio dei nomi per gli oggetti creati dal sistema Kubernetes
- kube-public: Questo spazio dei nomi viene creato automaticamente ed è leggibile da tutti gli utenti (compresi quelli non autenticati). Questo spazio dei nomi è principalmente riservato all'utilizzo del cluster, nel caso in cui alcune risorse debbano essere visibili e leggibili pubblicamente nell'intero cluster. L'aspetto pubblico di questo spazio dei nomi è solo una convenzione, non un requisito.
- kube-node-leaseQuesto spazio dei nomi contiene oggetti Lease associati a ciascun nodo. I lease dei nodi consentono al kubelet di inviare heartbeat in modo che il piano di controllo possa rilevare l'errore del nodo.

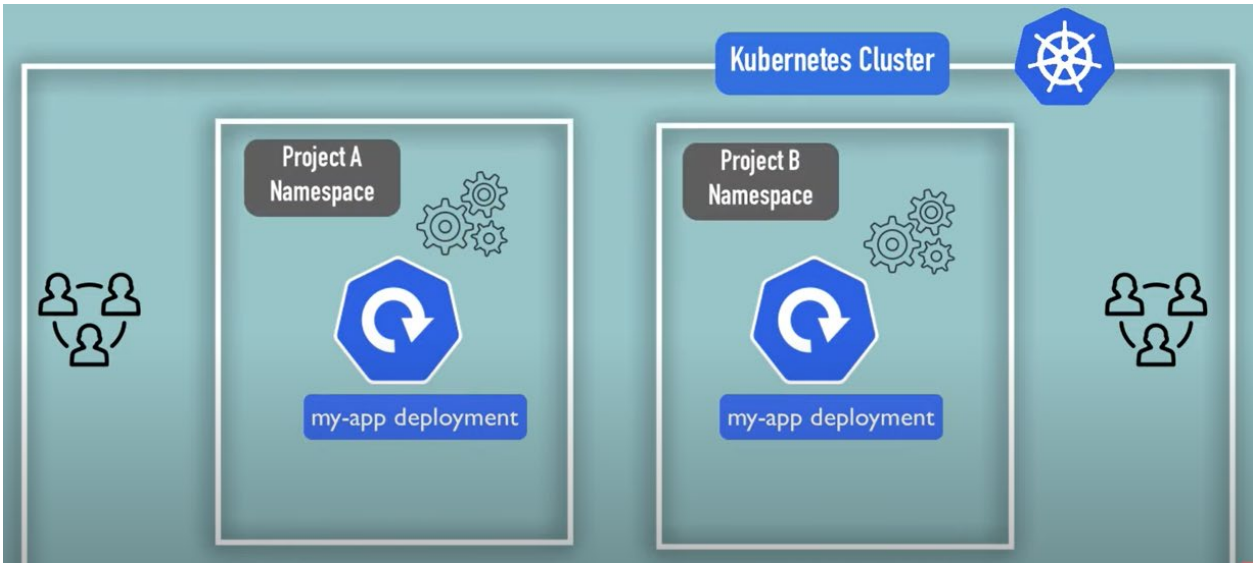
  kubectl get ns
  kubectl cluster-info
  kubectl create namespace “nomeNamespace”

Quando conviene usare più namespace ?

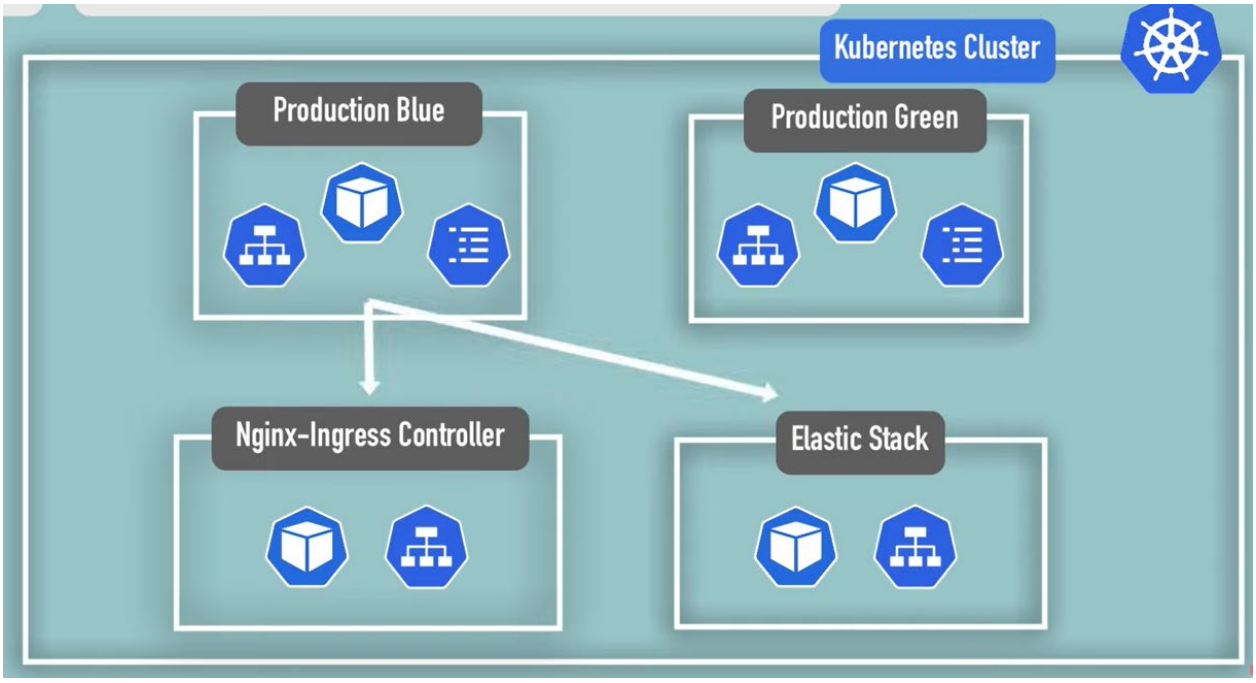
1. Nel momento in cui si hanno molti componenti nel ns, soprattutto se ho diversi dev che creano cose nel ns, in questo caso la cosa migliore è raggruppare le risorse/componenti in un unico ns separato. Ad esempio usiamo un ns differente se abbiamo un database con molti componenti/risorse associati. Oppure creamo un ns Monitoring con Prometheus app e i vari componenti di cui ha bisogno.



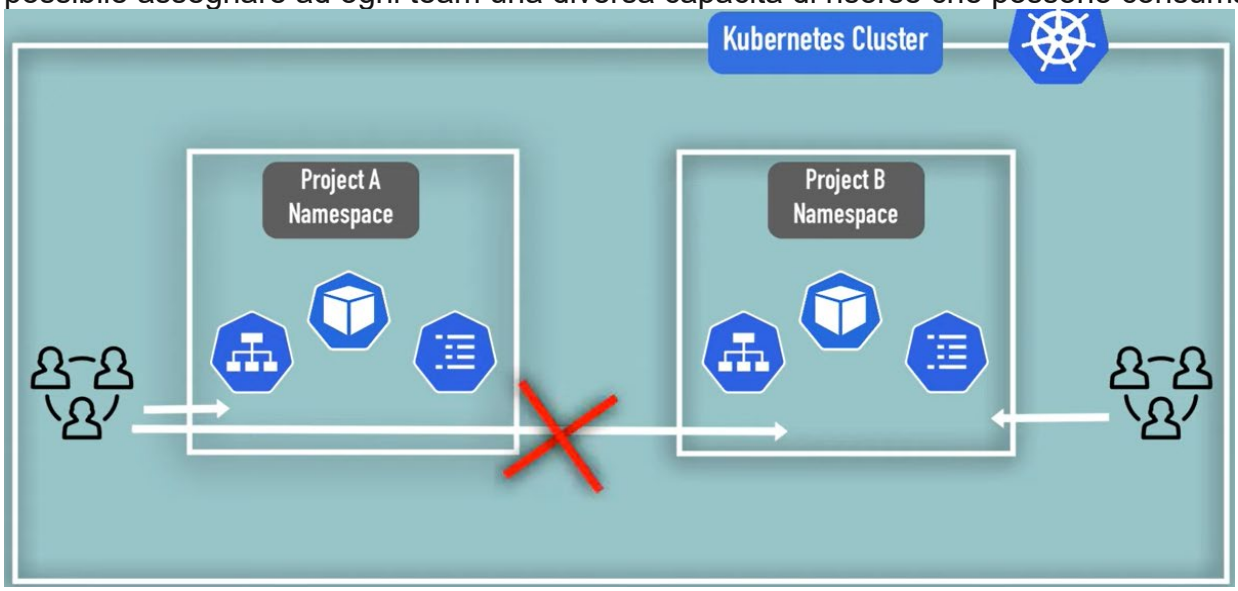
2. Usato quando si ha un multiple teams, supponiamo di avere 2 teams che lavorano sullo stesso cluster. Con 2 ns evitiamo Override dei nomi delle applicazioni.



3. Se vuoi avere 2 versioni diverse di produzione in cui solo una è in esecuzione (in questo possono condividere il ns Ingress e Elastic Stack)

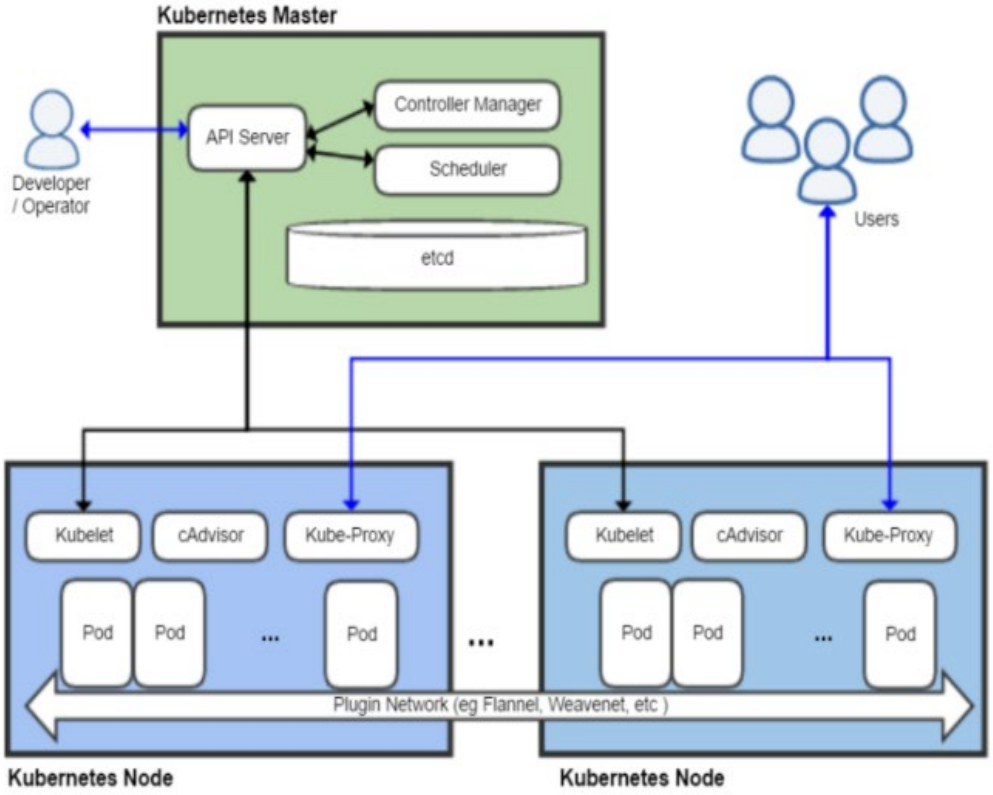


4. Se abbiamo 2 teams che lavorano sullo stesso namespace, ogni team ha il suo ambiente ns ciò per evitare di interferirsi. Inoltre se si hanno risorse limitate (ram,cpu) per il cluster è possibile assegnare ad ogni team una diversa capacità di risorse che possono consumare.



Cluster Kubernetes

Il cluster Kubernetes è composto da un nodo master(anche se consigliabile averne due) e più nodi worker, i nodi possono essere computer fisici o macchine virtuali, a seconda del cluster.



Nel **nodo master** o control plane ci sono dei processi che controllano lo stato del cluster e i nodi worker, tali processi sono.

- **Api-Server**: lo possiamo vedere come un gateway cluster, il quale riceve le richieste di update o di query. Fa da guardia per l'autenticazione per assicurarsi che solo chi è autenticato e autorizzato può fare richieste di creare un nuovo pod, servizio o altro.
- **Scheduler**: dopo che l'api-server riceve e poi valida la richiesta di creare un nuovo pod, lo schedule avvia il nuovo pod su uno dei worker node con una determinata logica. Infatti guarda la richiesta e capisce quante risorse sono necessarie per girare l'app (cpu, ram, ecc), dopo di che schedula il pod sul worker node che ha più risorse disponibili. (non schedula solo pod, ma anche servizi, deployment ecc).
- **Controller-Manager**: rileva ciclicamente lo stato del cluster, non appena un pod, va down il controller confronta lo stato desiderato dichiarato precedentemente, con lo stato attuale. Se lo stato attuale è diverso dallo stato desiderato, invia una richiesta allo Scheduler in modo tale da rischedulare il componente crashato. (Non è detto che lo scheduler ripristina il componente sullo stesso nodo in cui è crashato, dato che calcola sempre le risorse disponibili su ogni nodo)
- **Etcd**: key/value store dello stato del cluster, ogni cambiamento o aggiornamento che avviene nel cluster, come ad esempio la creazione di un nuovo pod, viene memorizzato nel etcd. Capiamo che gli altri processi del master prendono info dall'etcd per effettuare le loro operazioni dato che tiene traccia, ad esempio, di quali risorse sono disponibili o lo stato del cluster. Importante sottolineare che i dati delle applicazioni non sono memorizzati nel'etcd.

Inoltre possiamo avere anche 2 o più nodi master, in tal caso il processo api-server del primo master è sincronizzato con un loadbalancer con l'api-server del secondo master. Anche gli etcd sono sincronizzati.

I **worker node** sono i componenti che eseguono queste applicazioni. I nodi di lavoro eseguono le attività assegnate dal nodo master. Per la produzione e la gestione temporanea, il cluster è

distribuito su più nodi di lavoro. Per i test, i componenti possono essere eseguiti tutti sullo stesso nodo fisico o virtuale. Hanno due processi:

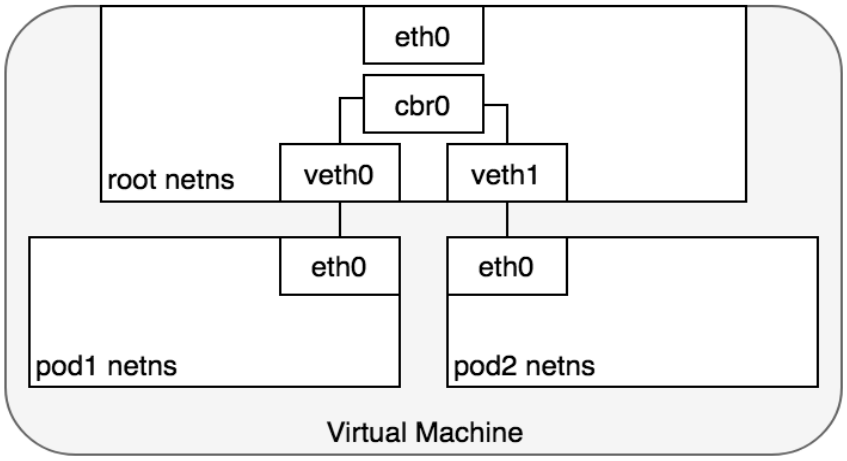
- **kubelet** il quale assicura che i contenitori siano in esecuzione in un pod interagendo con il motore Docker e garantisce che i contenitori corrispondenti siano pienamente operativi.
- **Kube-proxy** gestisce la connettività di rete e mantiene le regole di rete tra i nodi. Implementa il concetto di servizio Kubernetes su ogni nodo in un determinato cluster

NETWORK KUBERNETES

- Comunicazione tra contenitori nello stesso pod
- Comunicazione tra pod sullo stesso nodo
- Comunicazione tra pod su nodi diversi
- Comunicazione tra pod e servizi
- Come funziona il DNS? Come facciamo a scoprire gli indirizzi IP?

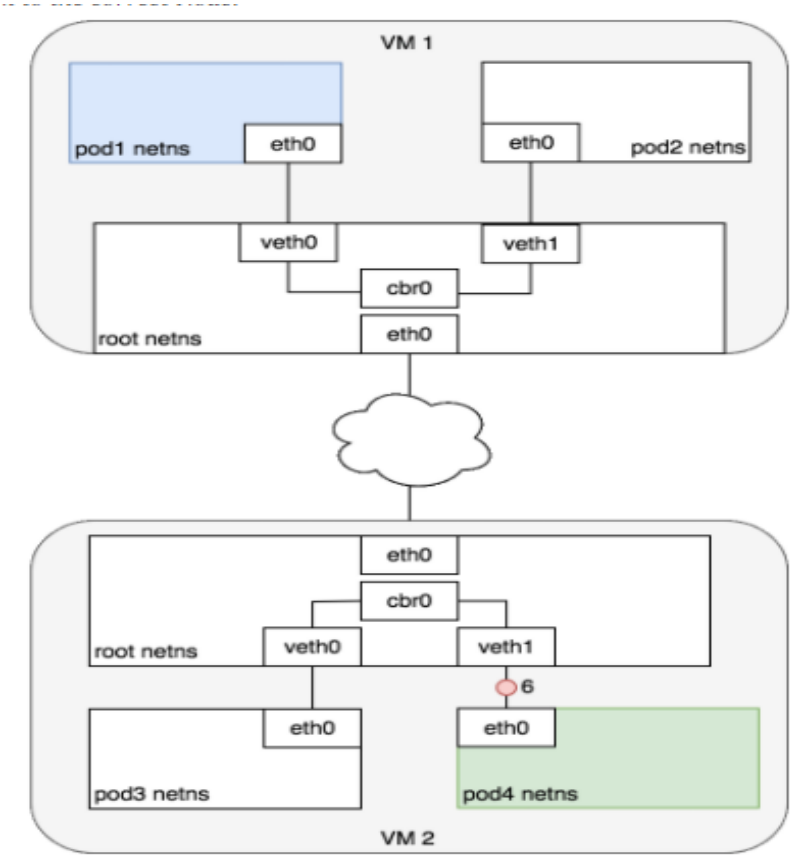
La **comunicazione tra container nello stesso pod** avviene tramite localhost e numero di porta. Ciò è possibile perché i contenitori nello stesso pod si trovano nello stesso spazio dei nomi di rete: condividono le risorse di rete. Esiste un contenitore che viene eseguito su ogni pod in Kubernetes il cui compito è quello di mantenere aperto lo spazio dei nomi nel caso in cui tutti gli altri contenitori sul pod muoiano. Si chiama pause container. Si deve attenzione ai conflitti di porte quando si hanno più container nello stesso pod.

Nella **comunicazione tra pod sullo stesso nodo** ogni pod ha un normale dispositivo ethernet virtuale chiamato eth0 per effettuare richieste di rete. Il eth di ciascun pod è connesso al corrispettivo dispositivo Ethernet virtuale vethX, situati nello spazio dei nomi root. Ora, vogliamo che i Pod parlino tra loro attraverso lo spazio dei nomi di root, e per questo usiamo un bridge di rete. I bridge implementano il protocollo ARP per scoprire l'indirizzo MAC a livello di collegamento associato a un determinato indirizzo IP. Quando un frame di dati viene ricevuto sul bridge, il bridge trasmette il frame a tutti i dispositivi collegati (tranne il mittente originale) e il dispositivo che risponde al frame viene archiviato in una tabella di ricerca. Il traffico futuro con lo stesso indirizzo IP utilizza la tabella di ricerca per scoprire l'indirizzo MAC corretto a cui inoltrare il pacchetto.



Il modello di rete di Kubernetes impone che i pod debbano essere raggiungibili dal loro indirizzo IP *attraverso i* nodi. Cioè, l'indirizzo IP di un pod è sempre visibile agli altri pod nella rete e ciascun pod visualizza il proprio indirizzo IP come lo vedono gli altri pod.

Nella **comunicazione tra pod su nodi diversi** il pacchetto finisce nel bridge di rete dello spazio dei nomi radice. ARP fallirà sul bridge perché nessun dispositivo è connesso al bridge con l'indirizzo MAC corretto per il pacchetto. In caso di errore, il bridge invia il pacchetto al percorso predefinito: lo spazio dei nomi di root eth0 dispositivo. A questo punto il percorso lascia il Nodo1 ed entra nella rete. Il pacchetto entra nello spazio dei nomi radice del Nodo2 di destinazione, dove viene instradato attraverso il bridge al dispositivo Ethernet virtuale corretto.



Comunicazione da Pod a Servizio

Abbiamo mostrato come instradare il traffico tra i Pod e i loro indirizzi IP associati ma gli indirizzi IP del pod appariranno e scompariranno in risposta a ridimensionamento in aumento o diminuzione, arresti anomali dell'applicazione o riavvii del nodo. Ciascuno di questi eventi può far cambiare l'indirizzo IP del Pod senza preavviso. I servizi sono stati integrati in Kubernetes per risolvere questo problema.

Un *servizio* Kubernetes gestisce lo stato di un insieme di Pod, consentendo di tenere traccia di un insieme di indirizzi IP di Pod che cambiano dinamicamente nel tempo. I servizi agiscono come un'astrazione sui Pod e assegnano un singolo indirizzo IP virtuale a un gruppo di indirizzi IP Pod. L'eventuale traffico indirizzato all'IP virtuale del Servizio verrà instradato all'insieme di Pod associati all'IP virtuale. Ciò consente all'insieme di Pod associati a un Servizio, di cambiare in qualsiasi momento: i client devono solo conoscere l'IP virtuale del Servizio, che non cambia.

Quando si crea un nuovo servizio Kubernetes, viene creato un nuovo IP virtuale (noto anche come ClusterIP). Ovunque all'interno del cluster, il traffico indirizzato all'IP virtuale sarà bilanciato dal carico al set di Pod di supporto associati al Servizio. In effetti, Kubernetes crea e mantiene automaticamente un sistema di bilanciamento del carico nel cluster che distribuisce il traffico ai Pod sani associati a un servizio.

Kubernetes può facoltativamente utilizzare DNS per evitare di dover codificare l'indirizzo IP del cluster di un servizio nella applicazione. **Kubernetes DNS** viene eseguito come un normale servizio Kubernetes pianificato nel cluster. Configura kubelets su ciascun nodo in modo che i contenitori utilizzino l'IP del servizio DNS per risolvere i nomi DNS. Ad ogni Servizio definito nel cluster (incluso il server DNS stesso) viene assegnato un nome DNS. I record DNS risolvono i nomi DNS nell'IP del cluster del Servizio o nell'IP di un POD, a seconda delle esigenze.

Un Pod DNS è costituito da tre contenitori separati:

- kubedns: controlla il master Kubernetes per le modifiche ai servizi e agli endpoint e mantiene le strutture di ricerca in memoria per soddisfare le richieste DNS.
- dnsmasq: aggiunge la cache DNS per migliorare le prestazioni.
- sidecar: fornisce un singolo endpoint di controllo dell'integrità per eseguire controlli dell'integrità per dnsmasqe kubedns.

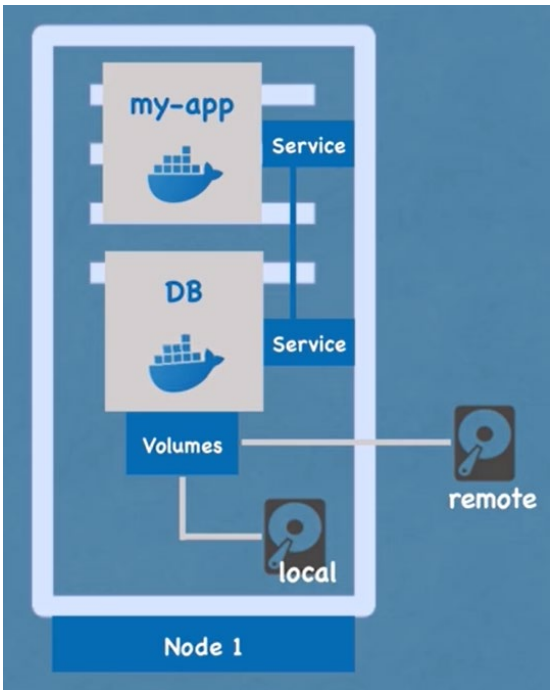
Il pod DNS stesso viene esposto come servizio Kubernetes con un IP cluster statico che viene passato a ciascun container in esecuzione all'avvio in modo che ogni container possa risolvere le voci DNS. Le voci DNS vengono risolte tramite il kubedns che mantiene le rappresentazioni DNS in memoria. etcd è il sistema di archiviazione back-end per lo stato del cluster e kubedns usa una libreria che converte etcd, gli archivi chiave-valore, in interi DNS per ricostruire lo stato della struttura di ricerca DNS in memoria quando è necessario.

CoreDNS funziona in modo simile kubednsma è costruito con un'architettura a plugin che lo rende più flessibile. A partire da Kubernetes 1.11, CoreDNS è l'implementazione DNS predefinita per Kubernetes.

PERSISTENT VOLUME e PERSISTENT VOLUME CLAIM

I file su disco in un contenitore sono effimeri, il che presenta alcuni problemi per le applicazioni durante l'esecuzione dei contenitori. Un problema è la perdita di file quando un contenitore si blocca. Il kubelet riavvia il contenitore ma con uno stato pulito. Un secondo problema si verifica quando si condividono file tra contenitori in esecuzione su uno stesso Pod. I **Volumi** in Kubernetes risolvono entrambi questi problemi. Lo storage può essere sullo stesso server su cui è in esecuzione il pod o un remote storage (ntf server) ovvero al di fuori del cluster kubernetes oppure può essere un cloud-storage. Tutti i persistent volume citati non sono gestiti da kubernetes ma

permette di configurarli inizialmente tramite file yaml. I persistent volume (PV) sono situati fuori dal namespace.



Possiamo avere 3 tipi di storage:

- Local storage è visibile solo dal nodo in cui è situato, se un pod sul nodoA crasha e viene ripristinato sul nodoB, sul nodoB non riesce recuperare i dati che gli servono per continuare le operazioni che l'app richiede. Inoltre se il cluster crasha lo storage non riesce a sopravvivere. Siccome è interno al cluster, se quest'ultimo crasha perdiamo i dati.
- Remote storage è visibile da tutti i nodi del cluster, se un pod sul nodoA crasha e viene ripristinato sul nodoB, anche il nodoB è collegato allo stesso storage in maniera tale da poter recuperare i dati. Siccome è esterno al cluster non perdiamo i dati se il cluster crasha.
- Cloud storage ha le stesse funzionalità del Remote storage, ma affidiamo i nostri dati a un cloud.

Local storage

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

NFS Storage

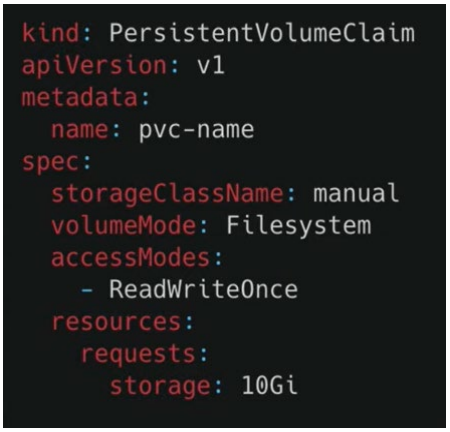
```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-name
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.0
  nfs:
    path: /dir/path/on/nfs/server
    server: nfs-server-ip-address
```

Google Cloud

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
  labels:
    failure-domain.beta.kubernetes.io/zone: us-central1-a__us-central1-b
spec:
  capacity:
    storage: 400Gi
  accessModes:
  - ReadWriteOnce
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

Il **Persistet Volume Claim (PVC)** è una richiesta di archiviazione da parte di un utente. E' configurato tramite il file yaml e reclama un PV con le caratteristiche di storage dichiarate nel file yaml pvc. Supponiamo di avere nel cluster 3 PV diversi con capacità diverse (ad esempio 5 Gi, 10 Gi e 15 Gi). Se configuriamo un PVC con 10 Gi stiamo reclamando il PV con capacità 10 Gi. Il PVC dovrebbe stare sullo stesso namespace del pod a cui è connesso, mentre i PV non sono nel namespace.

Sebbene PersistentVolumeClaims consenta a un utente di utilizzare risorse di archiviazione astratte, è comune che gli utenti necessitano di PersistentVolumes con proprietà variabili, come le prestazioni, per problemi diversi. Gli amministratori del cluster devono essere in grado di offrire una varietà di volumi persistenti che differiscono in più modi rispetto alle dimensioni e alle modalità di accesso, senza esporre gli utenti ai dettagli di come tali volumi vengono implementati. Per queste esigenze, esiste la risorsa *StorageClass*



```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-name
spec:
  storageClassName: manual
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Nel caso in cui si aggiungono nuove app stateful bisogna aggiungere nuovo PV manualmente, per questo motivo implemento le storage class; creano PV dinamicamente quando il PVC lo reclama

Descrizione attributi dei file manifest PV e PVC:

Le modalità di accesso sono:

ReadWriteOnce: il volume può essere montato in lettura-scrittura da un singolo nodo. La modalità di accesso ReadWriteOnce può comunque consentire a più pod di accedere al volume quando i pod sono in esecuzione sullo stesso nodo.

ReadOnlyMany: il volume può essere montato in sola lettura da molti nodi.

ReadWriteMany: il volume può essere montato in lettura-scrittura da molti nodi.

NodeAffinity definisce i vincoli che limitano i nodi da cui è possibile accedere a questo volume. Questo campo influenza la pianificazione dei pod che utilizzano questo volume.

persistenteVolumeReclaimPolicy Cosa succede a un volume persistente quando viene rilasciato dalla sua richiesta. Le opzioni valide sono Mantieni (impostazione predefinita per volumi persistenti creati manualmente), Elimina (impostazione predefinita per volumi persistenti con provisioning dinamico) e Ricicla (obsoleto).

storageClassName Nome di StorageClass a cui appartiene questo volume persistente. Il valore vuoto significa che questo volume non appartiene a nessuna StorageClass.

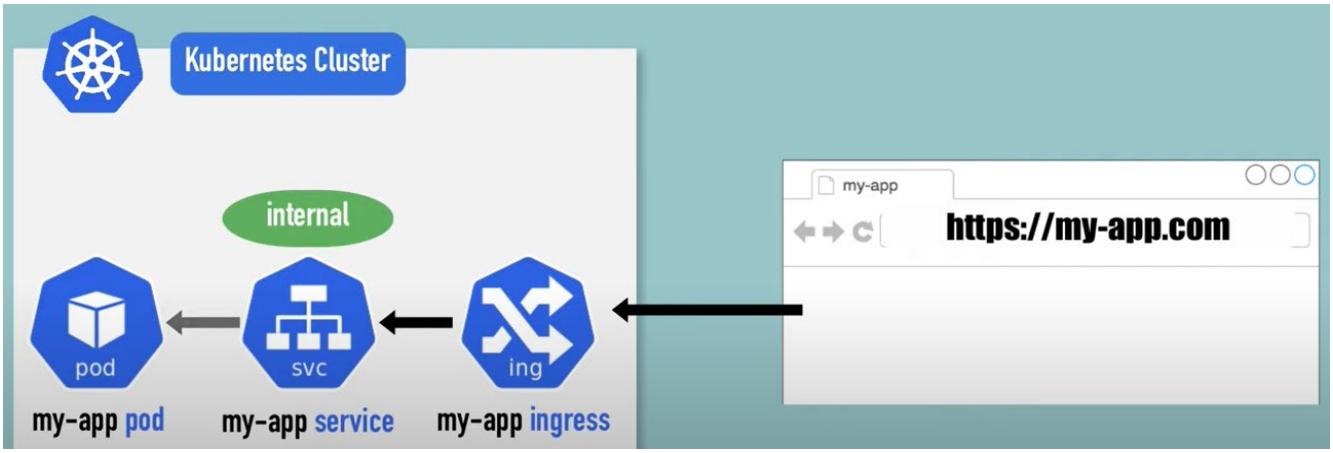
volumeMode Filesystem viene montato in Pods in una directory. Quindi volumeMode definisce se un volume deve essere utilizzato con un filesystem formattato o se deve rimanere nello stato di blocco raw. Il valore di Filesystem è implicito quando non è incluso nelle specifiche.

resources le risorse rappresentano le risorse minime che il volume dovrebbe avere.

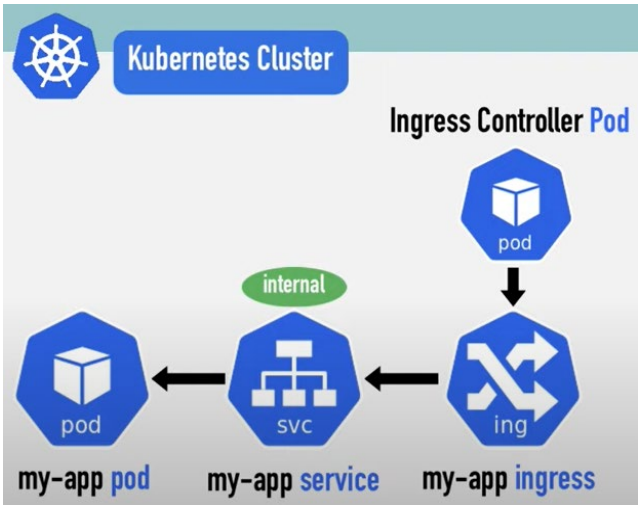
- **resources.limits** descrive la quantità massima di risorse di calcolo consentita.
- **resources.requests**: descrive la quantità minima di risorse di calcolo richieste. Se Requests viene omesso per un contenitore, per impostazione predefinita viene impostato Limits se specificato in modo esplicito, altrimenti su un valore definito dall'implementazione

Ingress - Instradamento del traffico da Internet a Kubernetes

Possiamo usare il servizio in maniera esterna per collegare il browser direttamente ad esso, ma è necessario che il browser conosca l'indirizzo IP e il numero port. Per avere un nome di dominio dell'application e una connessione sicura HTTPS usiamo il componente Ingress, in questo caso il servizio esterno diventa un internal service.



Ingress viene gestito da Ingress controller, che si occupa di processare le regole stabilite, gestisce il reindirizzamento delle richieste e fa da entrypoint per il cluster



Ora consideriamo gli ambienti nel quale il cluster Kubernetes può essere in esecuzione:

In un cloud provider ho out-of-the-box K8s solutions oppure hanno il loro cloud loadbalancer virtuale in cui le richieste del browser colpiscono il cloud loadbalancer che le reindirizza all'Ingress controller. Non è l'unico modo per farlo, ma è quello più comune.

Se deployamo il kube cluster in un ambiente bare metal abbiamo bisogno di identificare il tipo di entrypoint del cluster o all'interno del cluster o all'esterno in un server separato. Quest'ultima soluzione può essere configurata con un server proxy che adotta le regole del loadbalancer, fungendo da entrypoint per il cluster. Ciò significa che bisogna avere un IP publico del server e una porta esposta. Quindi abbiamo la richiesta browser -> server proxy -> Ingress controller -> servizi.

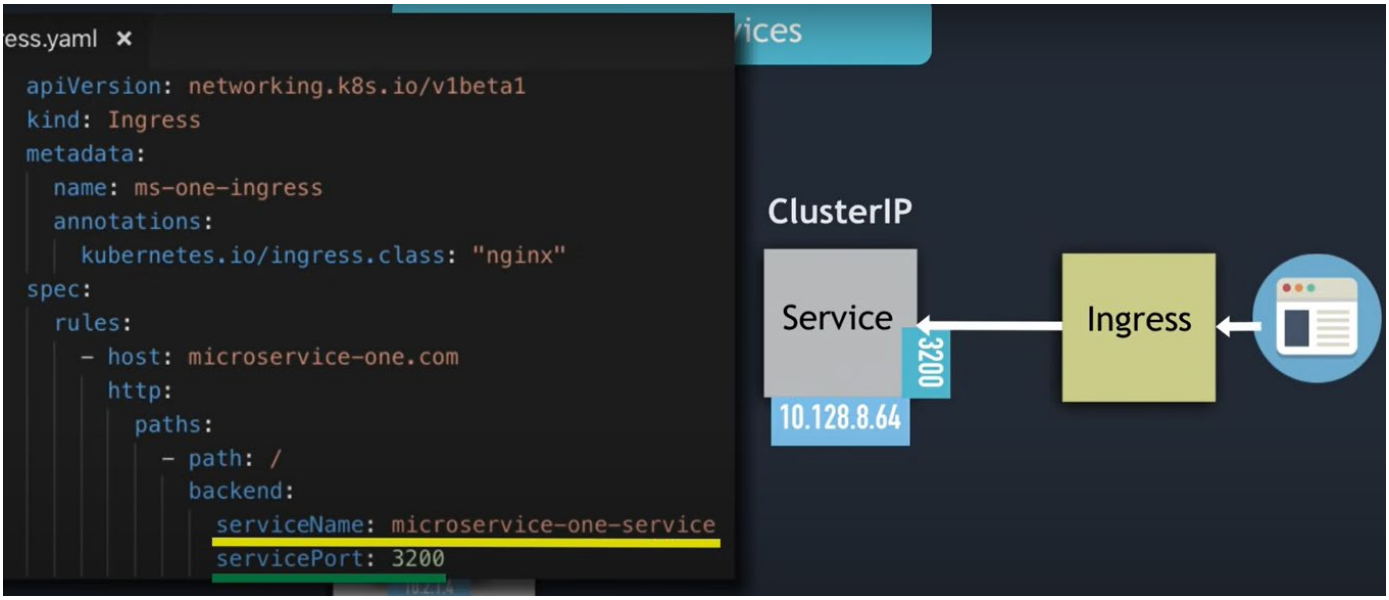
Esempio di Implementazione file yaml Ingress con il Servizio a cui fa riferimento

service-clusterIP.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: microservice-one-service
spec:
  selector:
    app: microservice-one
  ports:
    - protocol: TCP
      port: 3200
      targetPort: 3000
```

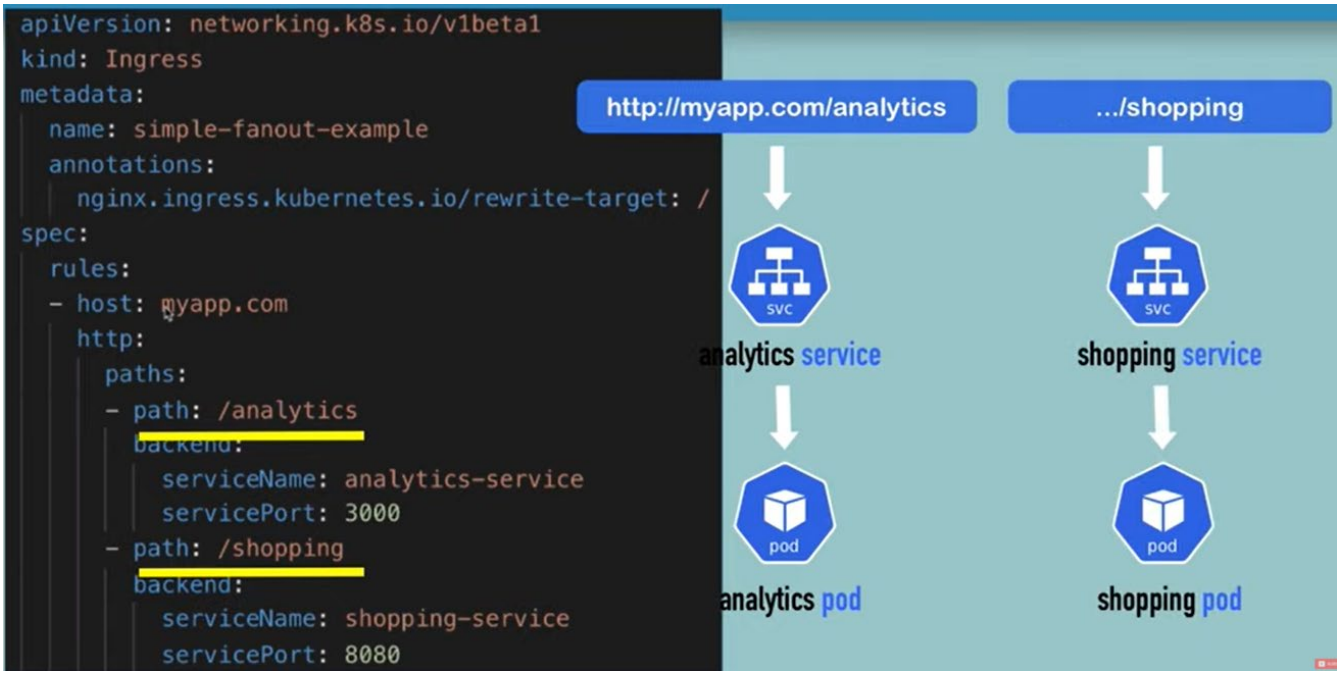
ess.yaml

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ms-one-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
    - host: microservice-one.com
      http:
        paths:
          - path: /
            backend:
              serviceName: microservice-one-service
              servicePort: 3200
```



Diversi casi d'uso:

Multiple paths per lo stesso Host

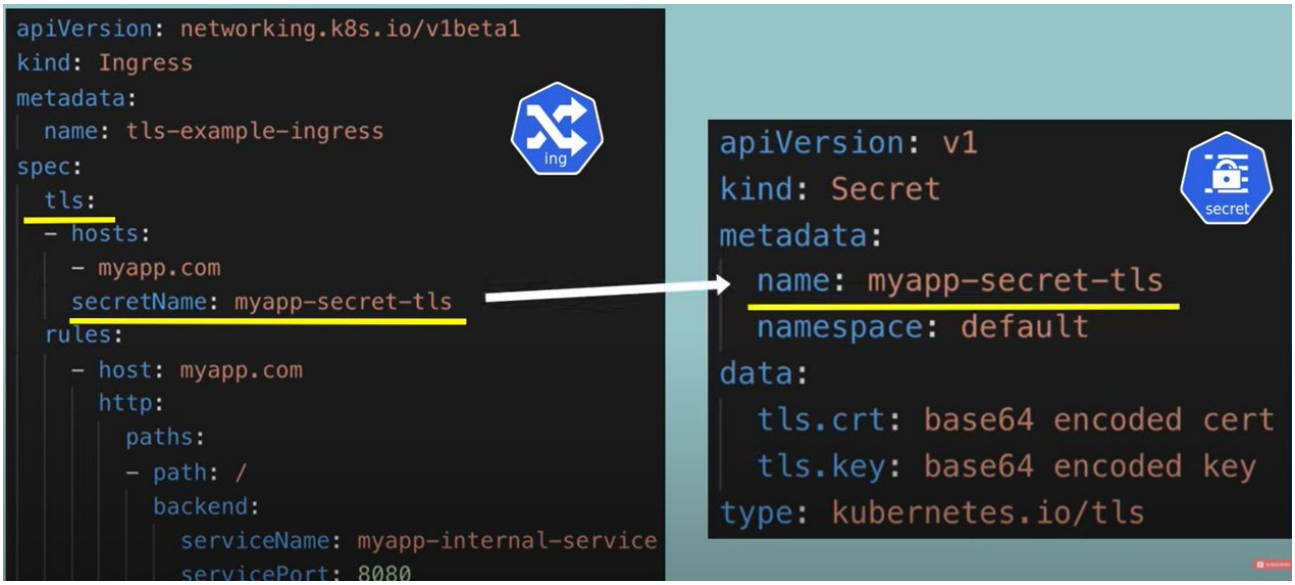


Multiple sub-domain or domain

Ho più host con un unico percorso. Ogni host rappresenta un sub-domain

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: analytics.myapp.com
    http:
      paths:
        backend:
          serviceName: analytics-service
          servicePort: 3000
  - host: shopping.myapp.com
    http:
      paths:
        backend:
          serviceName: shopping-service
          servicePort: 8080
```

TLS certificate

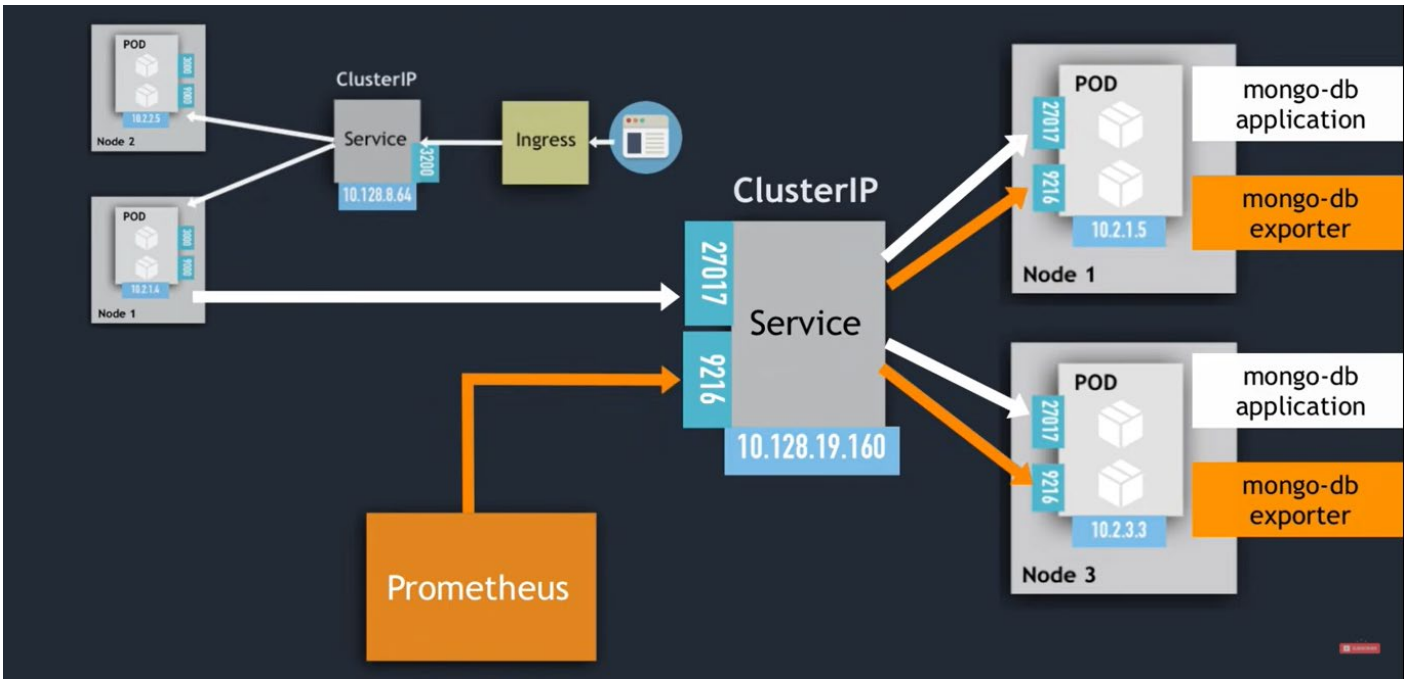


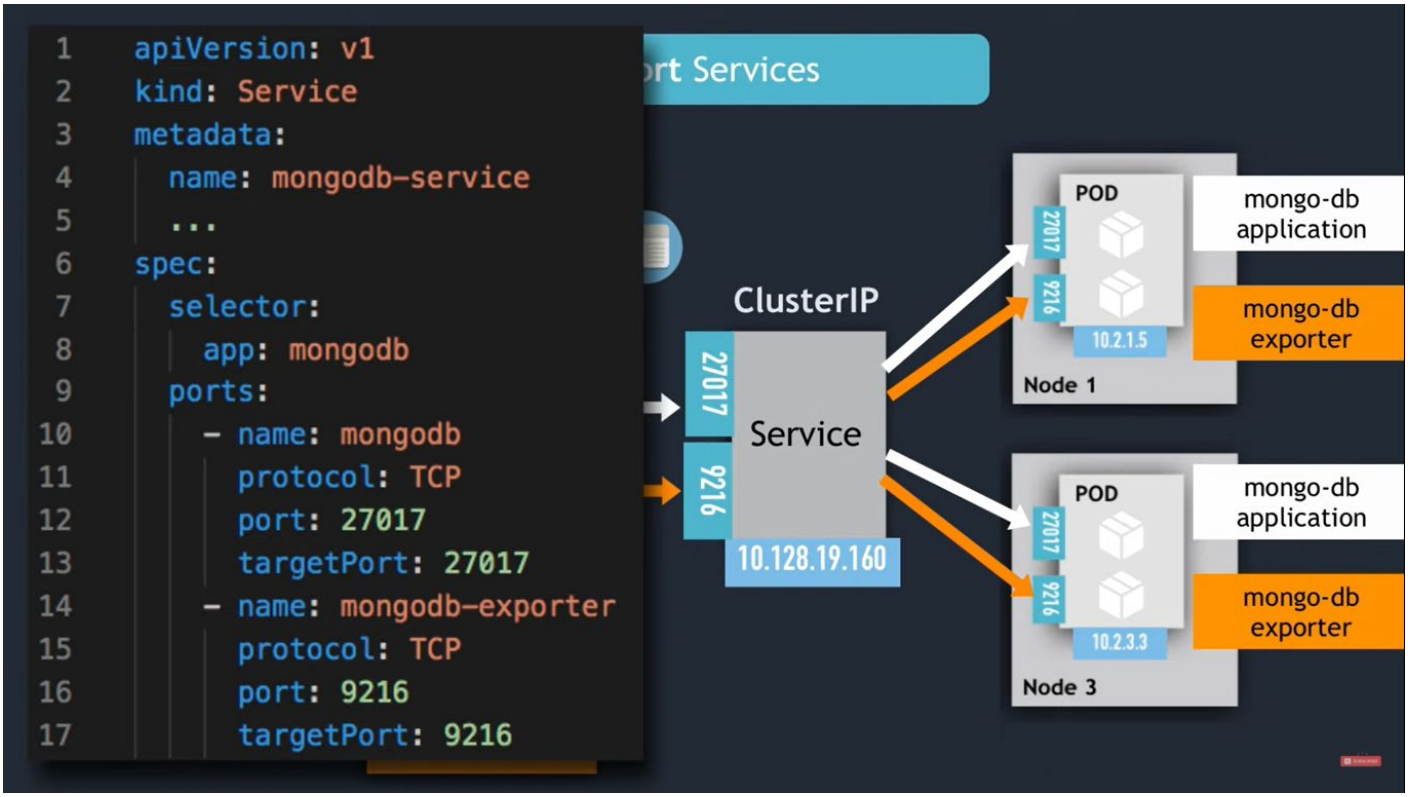
Kubernetes fornisce un tipo Secret l'archiviazione di un certificato e la relativa chiave associata che vengono in genere utilizzati per TLS.

VARI TIPI DI SERVICE

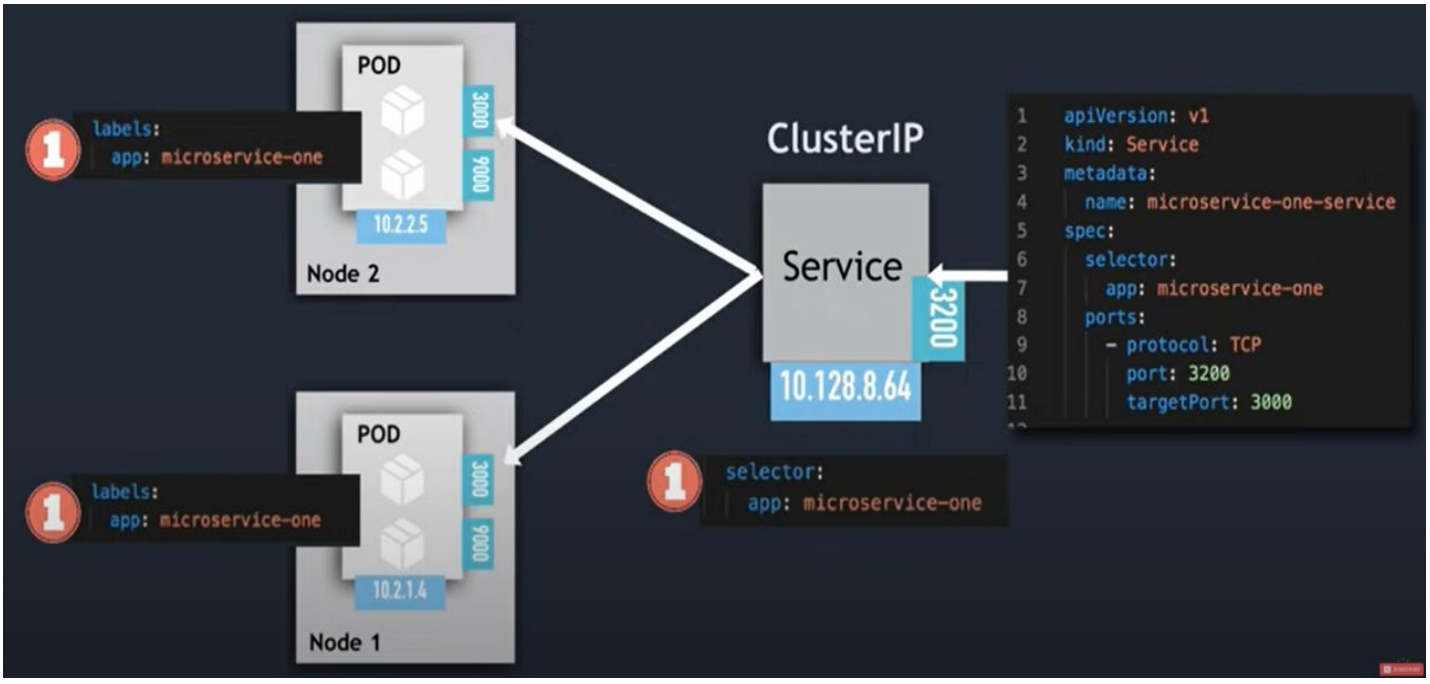
ClusterIP

Un servizio può avere più pod associati, Il ciclo di vita del pod e del servizio associato è diverso, ciò vuol dire che quando un pod crasha, il servizio ad esso collegato rimane su e non appena viene ripristinato il pod, gli viene associato automaticamente quel servizio. Il servzio contiene un IP virtuale permanente ClusterIP a cui un servizio di un altro pod fa riferimento nel momento in cui i due pod, con servizi diversi, vogliono comunicare. Il servizio gestisce gli endpoint dei Pod a esso collegati, sotto il ClusterIP, cioè tutti gli endpoind quindi i pod, associati a quel servizio, sono collegati al ClusterIP. Gli altri componenti del cluster, che vogliono collegarsi con quei pod, devono far riferimento al servizio. Tutto ciò è gestito automaticamente da Kubernetes.





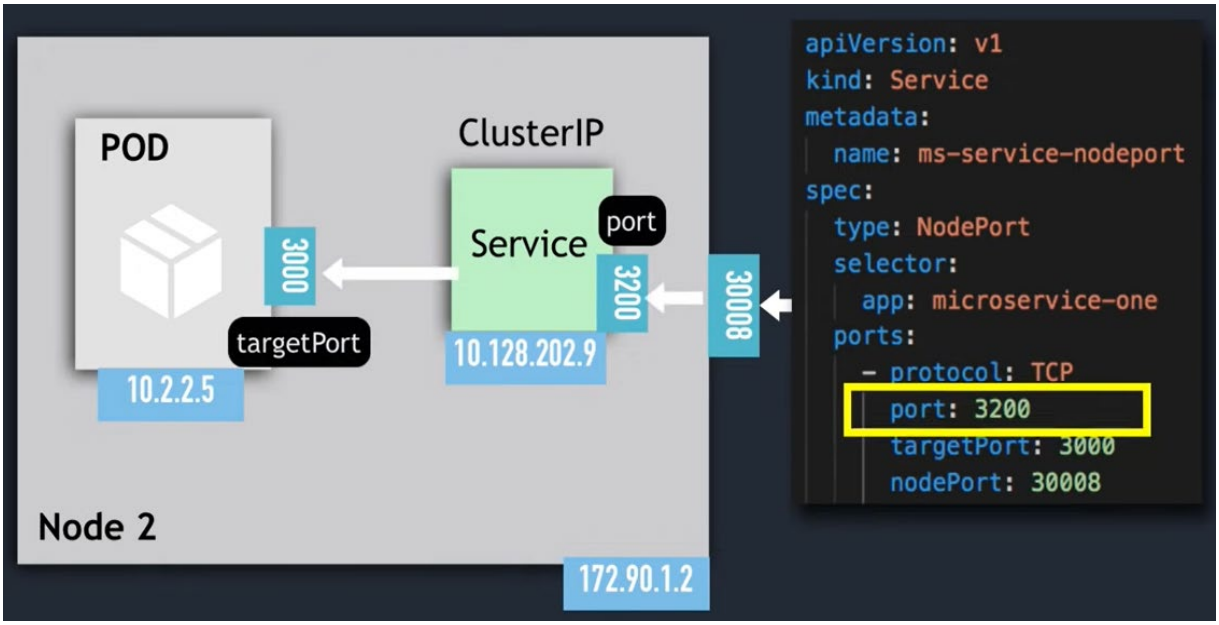
Per definire i pod che devono far parte dello stesso servizio, si dà un nome nel campo selector.app del file yaml del Servizio. Mentre nel file yaml dei pod, in questo caso Deployment, nei campi labels.app e selector.matchLabels.app devono essere definiti con lo stesso nome (come mostra la figura sotto).



nodePort

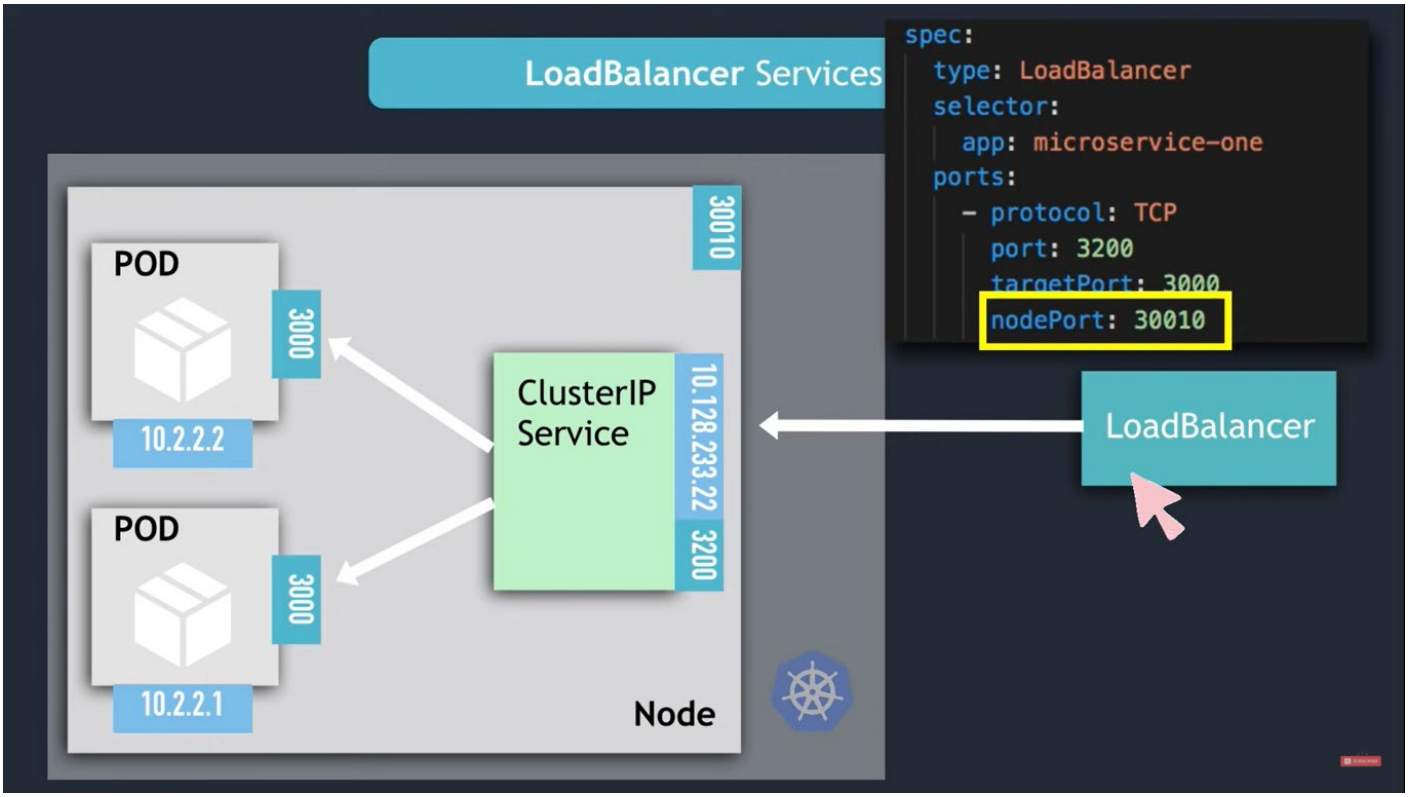
Il servizio è accessibile su una porta statica del worker node del cluster, il traffico esterno può raggiungere direttamente il servizio. Mentre nel caso precedente ClusterIP è raggiungibile solo dal traffico interno al cluster, nessun traffico esterno può raggiungere il servizio, che viene raggiunto solo tramite Ingress. Quindi nel caso del tipo nodePort Service le richieste del browser

raggiungono direttamente il worker node e la porta che è stata specificata nel servizio. Le porte possibili dell'attributo nodePort hanno un range 30000 – 32767. Questo tipo di Servizio non è sicuro in produzione, dato che esponiamo la porta direttamente al traffico esterno, ma può essere usato fare test più velocemente.

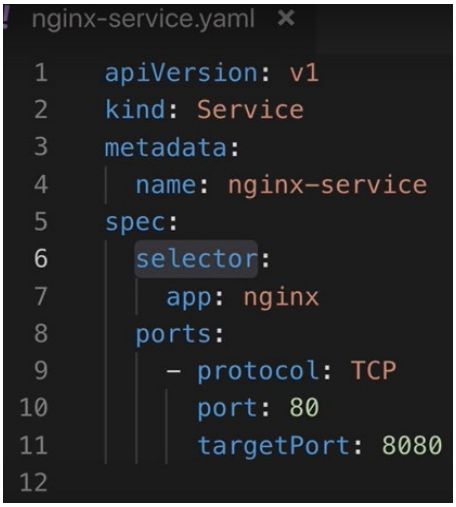


LoadBalancer

Con il type Service loadbalancer il servizio ClusterIP è accessibile esternamente tramite un cloud provider avente funzionalità LoadBalancer, dato che ogni provider ha la sua implementazione nativa loadbalancer. Quando creo un service loadbalancer, gli attributi nodePort e clusterIP sono automaticamente creati da Kubernetes, e il loadblancer esterno della piattaforma cloud reindirizza il traffico automaticamente. In questo caso l'attributo nodePort non renderà accessibile il worker node direttamente dal traffico esterno (browser) ma sarà accesibile solo attraverso il loadbalancer stesso.



HEADLESS SERVICE




```yaml
apiVersion: v1
kind: Service
metadata:
  name: headless-svc
spec:
  clusterIP: None
  selector:
    app: web
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Usato quando il client vuole comunicare direttamente con uno specifico pod. In questo caso, se il servizio è associato a più pod, perdiamo la scelta automatica random che fa di solito il servizio, per associare la richiesta a un pod anziché all'altro pod. Utilizziamo questo tipo di servizio quando le repliche dei pod non sono identiche quindi nel caso di applicazioni stateful, come database. Kubernetes permette ai client di scoprire l'IP pod e collegarsi direttamente ad esso attraverso il DNS LookUp. Il DNS LookUp per Service, ritorna un singolo IP (ClusterIP), ma se settiamo il campo clusterIP a None, invece ritorna l'indirizzo IP del pod. Nel caso di un app mysql poiché il servizio headless è denominato mysql, i <pod-name>.mysqlpod sono accessibili risolvendoli da qualsiasi altro pod nello stesso cluster e spazio dei nomi Kubernetes.

Differenze tra DEPLOYMENT e STATEFULSET

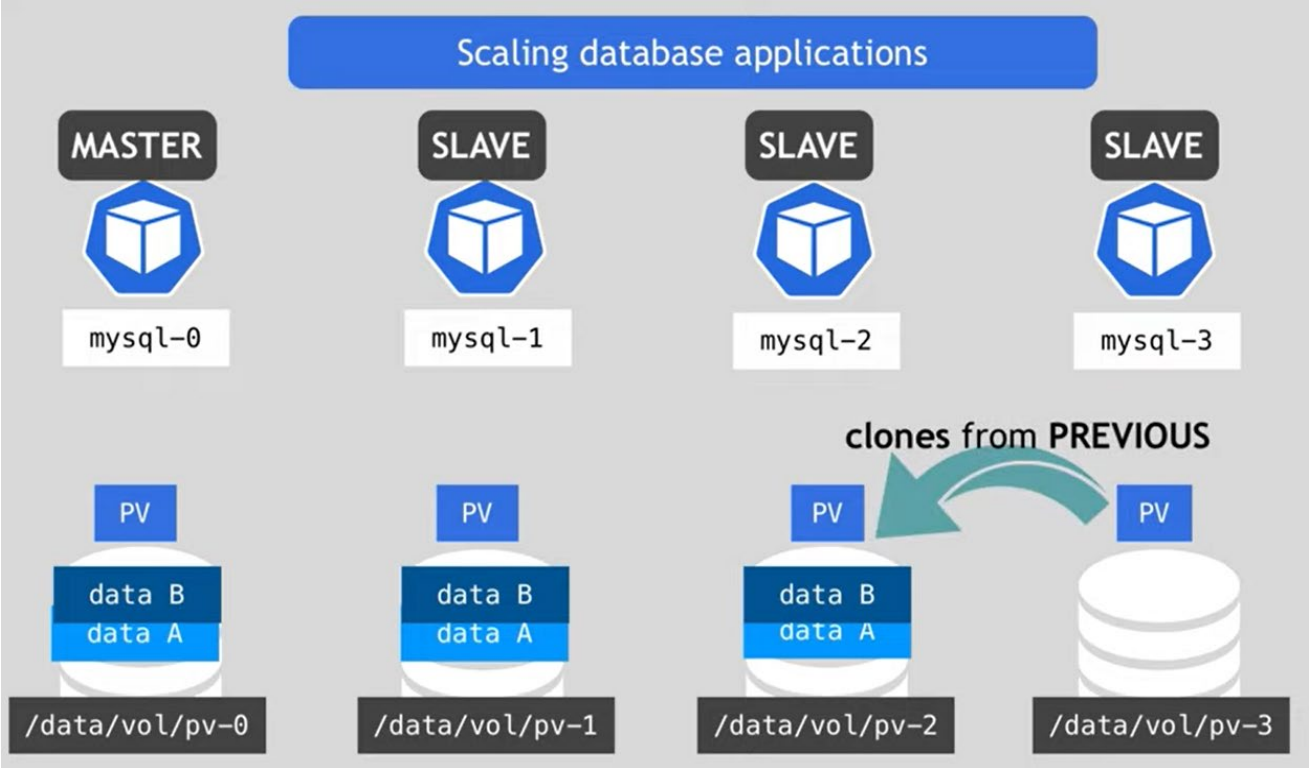
Le applicazioni stateless sono deployate usando **Deployment** component, sono app che non mantengono lo stato dell'appicazione e ogni richiesta è completamente nuova, cioè non dipende dalle precedenti. Deployment è un astrazione del pod e ci permette di replicare il pod in più repliche. In questo caso le repliche dei pod hanno le seguenti caratteristiche:

- identiche e interscambiabili; create con ordine random e con hash random (es nome della prima replica pod su 3 repliche: nginx-deployment-1564180365-khku8)
- creo un solo servizio che gestisce tutte le repliche, il servizio stesso fungerà da loadbalancer per distribuire le richieste sui vari pod. Il servizio sceglie la replica in maniera randomica.
- Nel caso cancelliamo qualche replica, ovvero scaliamo il num replica da 4 a 2 (ad esempio), la scelta di quale replica viene cancellata è randomica.

Le applicazioni stateful sono quelle in cui deve essere mantenuto lo stato della applicazione, come i database, e si utilizza il component **StatefulSet** per crearle. Caratteristiche delle repliche dei pod:

- Ogni pod viene identificato univocamente. Ad esempio, se creo uno StatefulSet con nome counter, creerà un pod con nome counter-0 e per repliche multiple di uno statefulset, i loro nomi aumenteranno come counter-0, counter-1, counter-2, ecc. Quindi capiamo che le repliche non sono identiche e interscambiabili.
- Archiviazione stabile e persistente; ogni replica di un set con stato avrà il proprio stato e ciascuno dei pod creerà il proprio PVC (Persistent Volume Claim). Quindi uno statefulset con 3 repliche creerà 3 pod, ognuno con il proprio volume, quindi un totale di 3 PVC (PVC-0, PVC-1 e PVC-2).
- Distribuzione e ridimensionamento ordinati; se il pod counter-2 muore, viene ripristinato un altro pod con la stessa identità, quindi stesso nome counter-2, e viene collegato allo stesso PVC-2
- Cancellazione e risoluzione ordinata; cioè, in questo caso, viene cancellata prima la replica 2 poi la replica 1 e infine la replica 0.

Perché è importate mantenere un identità stretta dei pod; mysql-0 ha il suo pv-0 e cosi via. I dati vengono sincronizzati tra i pod continuamente. Se si aggiunge un'altra replica mysql-3 vengono clonati i dati. Il master legge e scrive mentre gli slave leggono solo, per evitare l'inconsistenza dei dati.



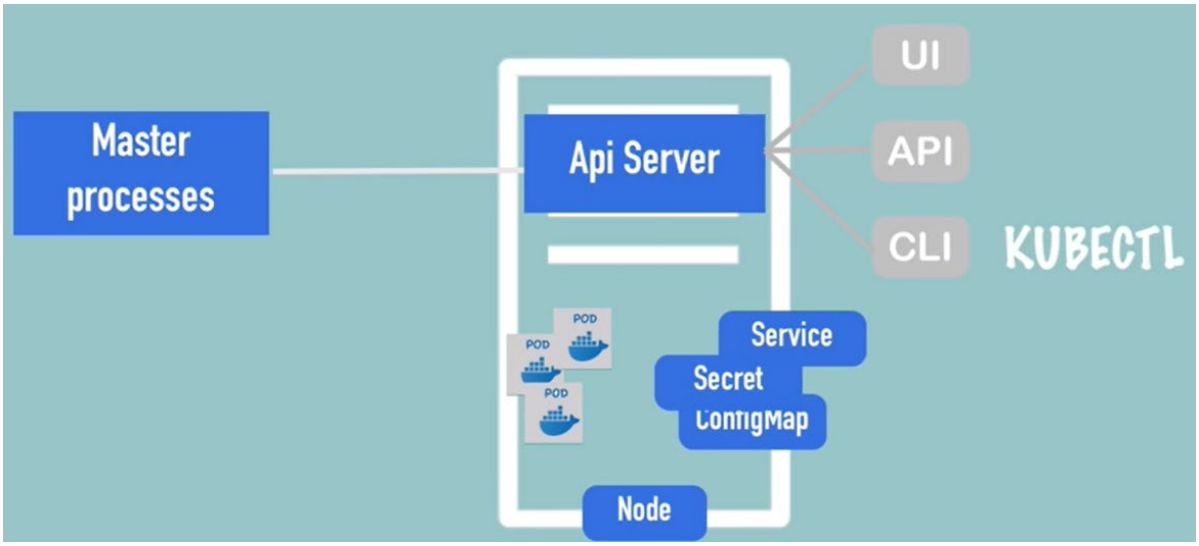
Nel caso tutti i pod mysql stateful muoiono, i dati non vengono persi perche i PV locali hanno un ciclo di vita differente dai pod stateful ad essi collegati, quindi i pod nuovi ricreati recuperano i dati dai PV locali. Mentre se il cluster kubernetes va in crash possiamo avere la perdita dei dati anche dei PV local, quindi è importante usare un remote storage PV siccome è situato all'esterno del cluster.

MINIKUBE

Perché si usa minikube ? Se ho master e worker node su macchine fisiche o virtuali che rappresentano nodi e si vuole fare test su una macchina locale o fare qualcosa di veloce come deployment di app o provare nuovi componenti, sarà un po' difficile fare ciò se l'intero cluster è distribuito su più macchine.

Cos'è minikube ? Un unico nodo in cui c'è l'intero cluster, il master e i worker sono processi che eseguono entrambi su un unico nodo. Ha un docker container preinstallato.

Dove e come si esegue ? Possiamo eseguire minikube su una virtualbox
Interagiamo con minikube per configurare i componenti che ci servono con KUBECTL è una command line tool. Api server del master process è il principale entrypoint del cluster, quindi qualsiasi componente voglio creare è necessario parlare con l'api. Possiamo usare acnhe UI e api.

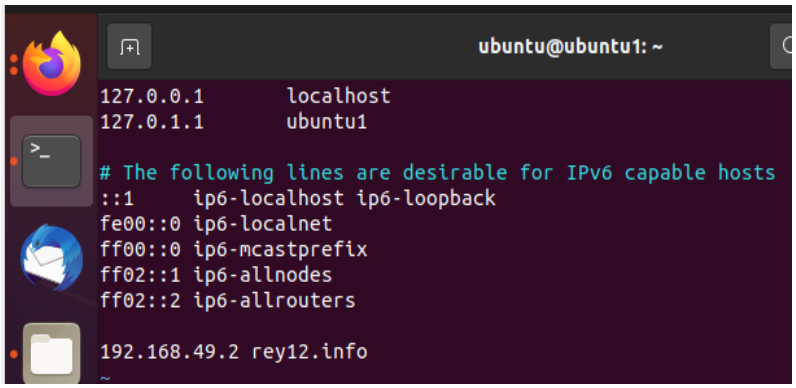


Installo il controller-ingress nginx:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.0/deploy/static/provider/cloud/deploy.yaml
```

Il controller di ingresso può essere installato tramite il sistema di componenti aggiuntivi di minikube:
`minikube addons enable ingress`

Una volta configurato il file Ingress-service.yaml, impostiamo come host il nome rey12.info mappandolo con l'ip di minikube. Per farlo usiamo il comando:
`sudo vim /etc/hosts`



Per scoprire l'ip di minikube:
`minikube ip`

Apriamo di nuovo il terminale, andiamo nel percorso in cui ci sono i file yaml creati e digitiamo i seguiti comandi per avviare i deployment creati:

```
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl apply -f ingress-service.ya
ml
ingress.networking.k8s.io/ingress-service configured
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl apply -f central-deployment
.yaml
deployment.apps/central-deployment created
service/central-service created
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl apply -f mysql-central-depl
oyment.yaml
error: error validating "mysql-central-deployment.yaml": error validating data:
 ValidationError(Deployment.spec.template.spec.containers[0].env[0].valueFrom):
 unknown field "secretkeyRef" in io.k8s.api.core.v1.EnvVarSource; if you choose
 to ignore these errors, turn validation off with --validate=false
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl apply -f mysql-central-depl
oyment.yaml
deployment.apps/mysql-central-deployment created
service/mysql-central-service created
persistentvolumeclaim/mysql-persistent-volume-claim created
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl apply -f redis-central-depl
oyment.yaml
deployment.apps/redis-central-deployment created
service/redis-central-service created
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl get pod
NAME                                          READY   STATUS    RESTARTS   AGE
central-deployment-7588487c4d-h6jbw           1/1     Running   0          6m50s
central-deployment-7588487c4d-nthnf           1/1     Running   0          6m50s
mysql-central-deployment-6468448ccf-bs45b     1/1     Running   0          3m55s
redis-central-deployment-6f58576d49-p2sjm     1/1     Running   0          107s
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl get deployment
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
central-deployment         2/2     2            2           7m4s
mysql-central-deployment   1/1     1            1           4m8s
redis-central-deployment   1/1     1            1           2m
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl get service
NAME                     TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AG
E
central-service          ClusterIP   10.111.1.203     <none>        80/TCP      7m
16s
kubernetes               ClusterIP   10.96.0.1        <none>        443/TCP     84
m
mysql-central-service    ClusterIP   10.96.5.134      <none>        3306/TCP    4m
21s
redis-central-service    ClusterIP   10.103.142.21    <none>        6379/TCP    2m
13s
```

```
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl describe deployment central
-deployment
Name:                   central-deployment
Namespace:              default
CreationTimestamp:      Mon, 10 Jan 2022 01:42:59 +0100
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               component=central
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unava
ilable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  component=central
  Containers:
   central:
    Image:      nginx
    Port:       80/TCP
    Host Port:  0/TCP
    Environment:
      MYSQL_HOST:            mysql-central-service
      MYSQL_TCP_PORT:        3306
      MYSQL_ROOT_PASSWORD:   pippo
      REDIS_HOST:            redis-central-service
      REDIS_PORT:            6379
    Mounts:                  <none>
  Volumes:                   <none>
Conditions:
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   central-deployment-7588487c4d (2/2 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  11m   deployment-controller  Scaled up replica set
 central-deployment-7588487c4d to 2
```

```
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl describe pod central-deploy
ment-7588487c4d-h6jbw
Name:         central-deployment-7588487c4d-h6jbw
Namespace:    default
Priority:     0
Node:         minikube/192.168.49.2
Start Time:   Mon, 10 Jan 2022 01:43:01 +0100
Labels:       component=central
              pod-template-hash=7588487c4d
Annotations:  <none>
Status:       Running
IP:           172.17.0.2
IPs:
  IP:           172.17.0.2
Controlled By:  ReplicaSet/central-deployment-7588487c4d
Containers:
  central:
    Container ID:   docker://c8c714e566dbf1e801e27bf651305fe11a2c29fc229eee6d33
3fd529a0f5e0d8
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:0d17b565c37bcbd895e9d92315a0
5c1c3c9a29f762b011a10c54a66cd53c9b31
    Port:           80/TCP
    Host Port:      0/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Mon, 10 Jan 2022 01:43:15 +0100
    Ready:          True
    Restart Count:  0
    Environment:
      MYSQL_HOST:            mysql-central-service
      MYSQL_TCP_PORT:        3306
      MYSQL_ROOT_PASSWORD:   pippo
      REDIS_HOST:            redis-central-service
      REDIS_PORT:            6379
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-sh8f8
 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-sh8f8:
```

```
  kube-api-access-sh8f8:
    Type:                    Projected (a volume that contains injected data fr
om multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists f
or 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists
 for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  12m   default-scheduler  Successfully assigned default/cen
tral-deployment-7588487c4d-h6jbw to minikube
  Normal  Pulling    12m   kubelet            Pulling image "nginx"
  Normal  Pulled     12m   kubelet            Successfully pulled image "nginx"
 in 8.956244433s
  Normal  Created    12m   kubelet            Created container central
  Normal  Started    12m   kubelet            Started container central
```

```
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl describe service central-se
rvice
Name:              central-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          component=central
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.111.1.203
IPs:               10.111.1.203
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         172.17.0.2:80,172.17.0.3:80
Session Affinity:  None
Events:            <none>
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$
```

CASO MYSQL TIPO DEPLOYMENT CON PIU REPLICHE

```
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl get pod
NAME                                          READY   STATUS
GE
mysql-central-deployment-6468448ccf-8t425     1/1     Running
m39s
mysql-central-deployment-6468448ccf-ng4vc     1/1     Running
m39s
mysql-central-deployment-6468448ccf-rm2tp     1/1     Running
m39s
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$ kubectl get pvc
NAME                           STATUS   VOLUME
     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mysql-persistent-volume-claim   Bound    pvc-e87f523b-7633-4
57   10Mi       RWO            standard       7m9s
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/k8s$
```

MYSQL STATEFULSET MASTER-SLAVE

Configuriamo il file config-map in modo tale che il server primario sia in grado di fornire i log di replica alle repliche e che le repliche rifiutino le scritture.

Configuriamo due servizi in un unico file: il servizio Headless fornisce un nome DNS che il controller StatefulSet crea per ogni Pod che fa parte del set. Poiché il servizio headless è denominato `mysql`, i `<pod-name>.mysql`pod sono accessibili risolvendoli da qualsiasi altro pod nello stesso cluster e spazio dei nomi Kubernetes.
Il servizio client, chiamato mysql-read, è un normale servizio con il proprio IP cluster che distribuisce le connessioni tra tutti i pod MySQL che segnalano di essere pronti. L'insieme di potenziali endpoint include il server MySQL primario e tutte le repliche.Si noti che solo le query di

lettura possono utilizzare il servizio client con bilanciamento del carico. Poiché esiste un solo server MySQL primario, i client devono connettersi direttamente al Pod MySQL primario (tramite la sua voce DNS all'interno del servizio Headless) per eseguire le scritture.

Clonazione di dati esistenti

In generale, quando un nuovo Pod si unisce al set come replica, deve presumere che il server MySQL primario possa già contenere dati. Un Init Container cioè uno script di installazione richiamato prima di avviare tutti gli altri pod, denominato clone-mysql, esegue un'operazione di clonazione su un Pod di replica la prima volta che viene avviato su un PersistentVolume vuoto. Ciò significa che copia tutti i dati esistenti da un altro Pod in esecuzione, quindi il suo stato locale è sufficientemente coerente per iniziare la replica dal server primario.

MySQL stesso non fornisce un meccanismo per farlo, quindi si utilizza uno strumento open source chiamato Percona XtraBackup. Durante il clone, il server MySQL di origine potrebbe subire una riduzione delle prestazioni. Per ridurre al minimo l'impatto sul server MySQL primario, lo script indica a ciascun Pod di clonare dal Pod il cui indice ordinale è inferiore di uno. Funziona perché il controller StatefulSet assicura sempre che Pod N sia pronto prima di avviare Pod N+1.

Inizio replica

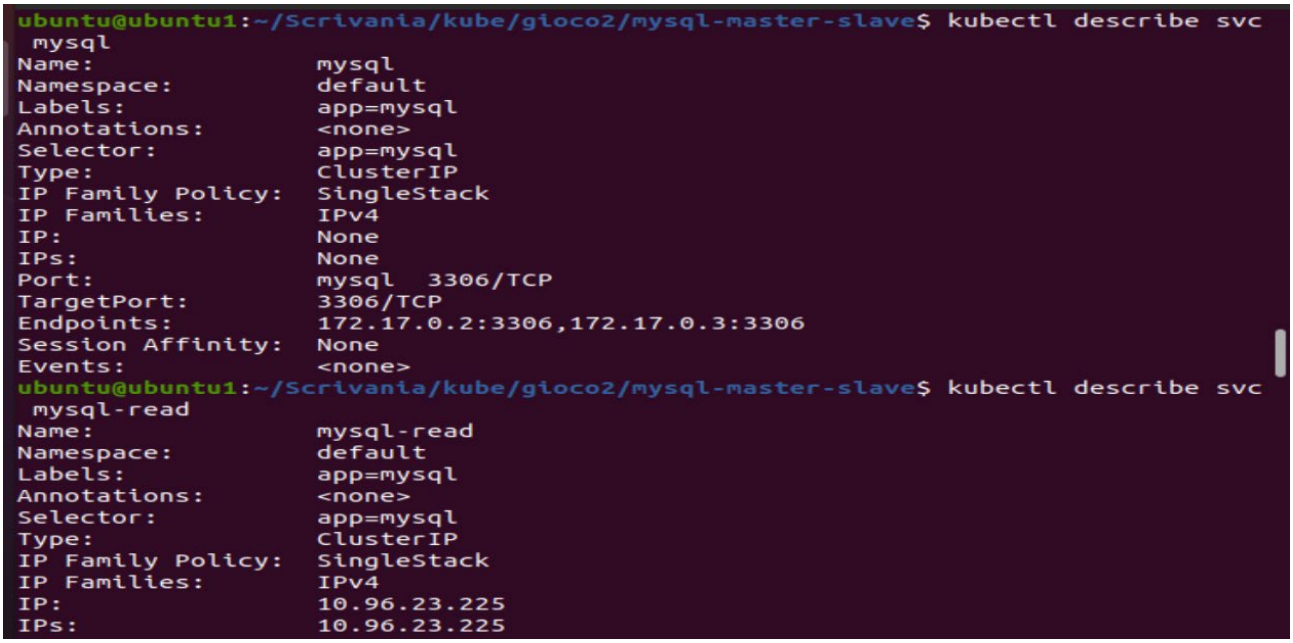
Dopo che gli Init Containers sono stati completati correttamente, i normali container vengono eseguiti. I Pod MySQL sono costituiti da un mysql contenitore che esegue il mysqld server effettivo e da un xtrabackup contenitore che funge da sidecar .

Il xtrabackupsidecar esamina i file di dati clonati e determina se è necessario inizializzare la replica di MySQL sulla replica. In tal caso, attende mysql di essere pronto e quindi esegue i comandi CHANGE MASTER TO e START SLAVE con i parametri di replica estratti dai file clone di XtraBackup.

Una volta che una replica inizia la replica, ricorda il suo server MySQL primario e si riconnette automaticamente se il server si riavvia o la connessione si interrompe. Inoltre, poiché le repliche cercano il server primario con il suo nome DNS stabile (mysql-0.mysql), trovano automaticamente il server primario anche se riceve un nuovo IP Pod a causa della riprogrammazione.

Infine, dopo aver avviato la replica, il xtrabackup container resta in ascolto delle connessioni da altri Pod che richiedono un clone di dati. Questo server rimane attivo a tempo indeterminato nel caso in cui StatefulSet aumenti o nel caso in cui il pod successivo perda il suo PersistentVolumeClaim e debba ripetere il clone.

```
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/mysql-master-slave$ kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
mysql-0   2/2     Running   0          7m52s
mysql-1   2/2     Running   0          5m23s
mysql-2   0/2     Pending   0          4m44s
```


ubuntu@ubuntu1:~/Scrivania/kube/gioco2/mysql-master-slave$ kubectl describe svc mysql
Name: mysql
Namespace: default
Labels: app=mysql
Annotations: <none>
Selector: app=mysql
Type: ClusterIP
IP Family Policy: SingleStack
IP Families: IPv4
IP: None
IPs: None
Port: mysql 3306/TCP
TargetPort: 3306/TCP
Endpoints: 172.17.0.2:3306,172.17.0.3:3306
Session Affinity: None
Events: <none>
ubuntu@ubuntu1:~/Scrivania/kube/gioco2/mysql-master-slave$ kubectl describe svc mysql-read
Name: mysql-read
Namespace: default
Labels: app=mysql
Annotations: <none>
Selector: app=mysql
Type: ClusterIP
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.96.23.225
IPs: 10.96.23.225