Proyecto Kubernetes

Jorge Pastor Ruiz

- Kubernetes
  - Master
  - Nodo
  - Topología
  - Grandes clusters
  - Como funciona?
  - Pod
  - Instalación entorno pruebas
    - comandos basicos minikube
  - Instalación cluster real
    - Requisitos
    - Instalación
      - Preparación de nodos
      - Instalar servicios
      - Creación de master
      - Juntar nodos worker
      - Eliminar un nodo
  - Instalación cluster EKS
    - Requisitos
      - Instalar aws-clie
        - Crear usuario IAM
        - Instalar comando AWS
        - Configurar aws-clie
          - Comprobar
      - Instalar eksctl
      - Instalar kubectl
    - Desplegar cluster
    - Eliminar cluster
  - Gestionar pod manualmente
    - Comandos de gestion en pods
    - Manifiestos
      - Simple
      - Dos pods
      - Dos contenedores
      - Labels
      - Asignar pod a un nodo
  - Replicaset
    - Como identifica los pods
    - Gestión de replicaset
  - Deployment

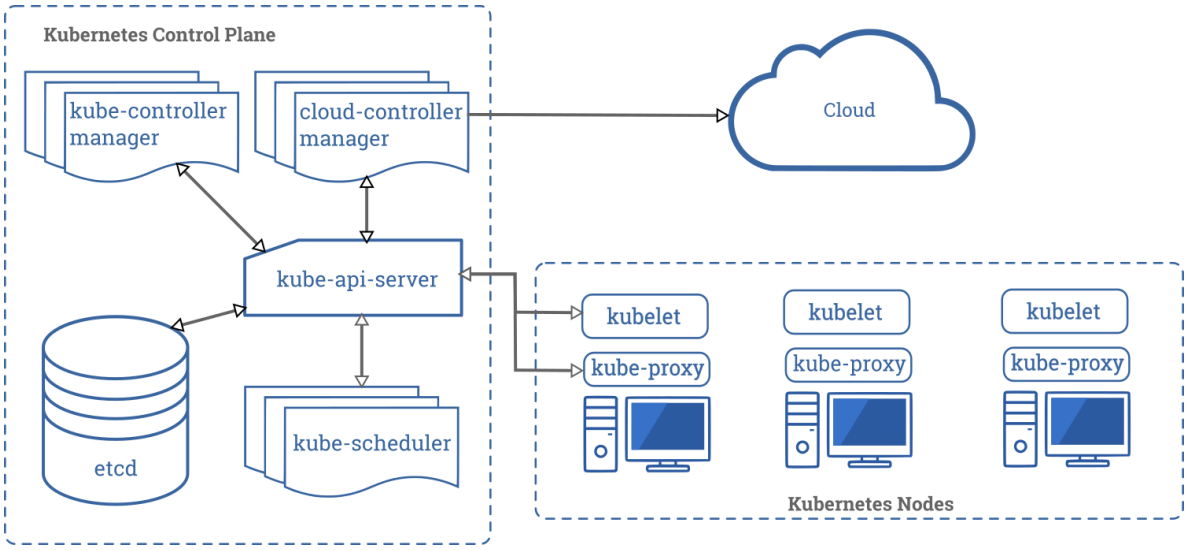
        - Como gestiona las actualizaciones
        - Gestion de manifiesto
            - Ejemplo manifiesto
            - Desplegar deployment
            - Comprobar que ha creado
            - Actualizar
                - History
                    - Metadata
                    - Cambio manual
                    - Record
                - Roll-back
    - Servicios & endpoints
        - Tipos de servicios
    - Namespaces
        - Gestión manual
            - Ejemplo espacios de trabajo
            - Template
            - Dns entre namespaces
        - Contexto
        - Limitaciones
            - Qos Classes
            - LimitRange
                - Limites por defecto
                - Mínimos y Máximos
            - ResourceQuota
                - Limitar pods
    - Probes
        - Tipos
            - liveness
                - cmd
                - tcp
                - http
    - ConfigMaps y Variables Entorno
        - Variables entorno
            - valores externos
        - ConfigMap
            - Comand-line
            - template
                - Especificar nuevo volumen
                - Asignar volumen
            - Environtment
    - Secrets
        - Consola

        - Template
        - Seguro
        - Utilizarlos
            - Volumen
            - items
        - Variables entorno
    - Volumenes
        - Tipos de Volúmenes
        - Emptydir
        - HostPath
        - Volumen AWS ebs
        - PV/PvC
            - Selectors
            - Asignar pvc a pod
            - StorageClass dinamico
        - NFS
        - Dinámico
            - Nfs
                - Configuración del nfs server
                - Despliegue requerido
                - StorageClass
                - Comprobar
    - RBAC
        - Permisos
        - Binding
        - Usuarios
            - x509 clients certs
        - Role
        - Rolebinding
        - ClusterRole
            - Cluster admin
        - Grupos roles
        - ServiceAccount
            - Asociar SA a Pod
    - Ingres
        - Nginx-controler
            - Instalación y comprobación
            - Tipo ruta
            - Tipo dominio
            - TLS

Kubernetes

Kubernetes es una plataforma portátil, extensible y de código abierto para gestionar cargas de trabajo y servicios en contenedores, que facilita tanto la configuración declarativa como la automatización. Tiene un ecosistema grande y de rápido crecimiento. Los servicios, el soporte y las herramientas de Kubernetes están ampliamente disponibles.

El nombre Kubernetes se origina del griego, que significa timonel o piloto. Google abrió el proyecto Kubernetes en 2014. Kubernetes combina más de 15 años de experiencia de Google ejecutando cargas de trabajo de producción a escala con las mejores ideas y prácticas de la comunidad.



Master

Master es el nodo configurado como *control-panel* encargado de controlar los diferentes nodos, para esto esta dividido en diferentes secciones.

kube-api-server: Esta será la parte encargada de interactuar con el usuario, el usuario le indica que hacer al master desde la api, y el gestionará donde crear o no los contenedores.

kube-scheduler: Se encarga de decidir en que nodo se aplicaran los contenedores

kube-controler-manager: controler está dividido en diferentes subsecciones.

- node controler: encargado de levantar nodos nuevos
- replication controler: encargado de gestionar las replicas.
- endpoint controler: servicios y pods en tema de redes
- service account: temas de tokens y autenticaciones.

Etcd: Base de datos del cluster, esta guarda datos de todo lo que sucede, versiones, etc...

Nodo

Un nodo puede ser una máquina virtual o física, esta se identifica por que corre un servicio llamado kubelet.

Como requisito tiene que tener instalado un gestor de contenedores, normalmente docker para poder gestionar contenedores.

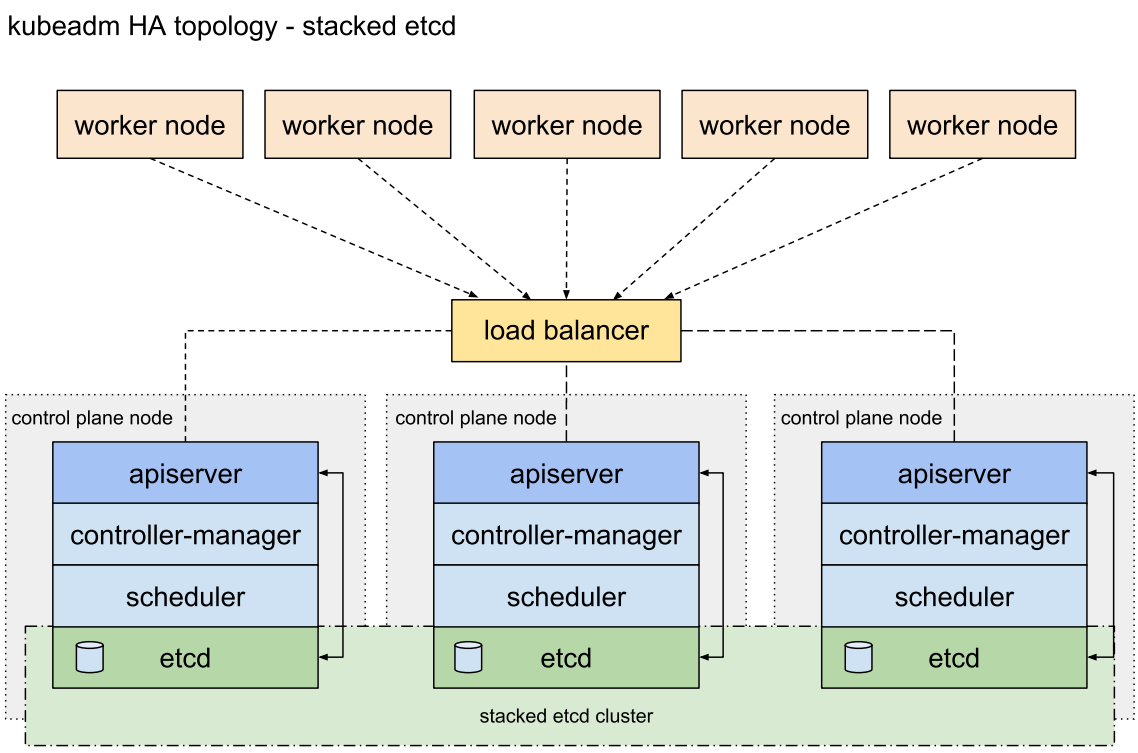
kubelet: este se encarga de recibir y enviar información con el master.

kube proxy: maneja todo lo relacionado con las redes dentro de los containers de cada nodo

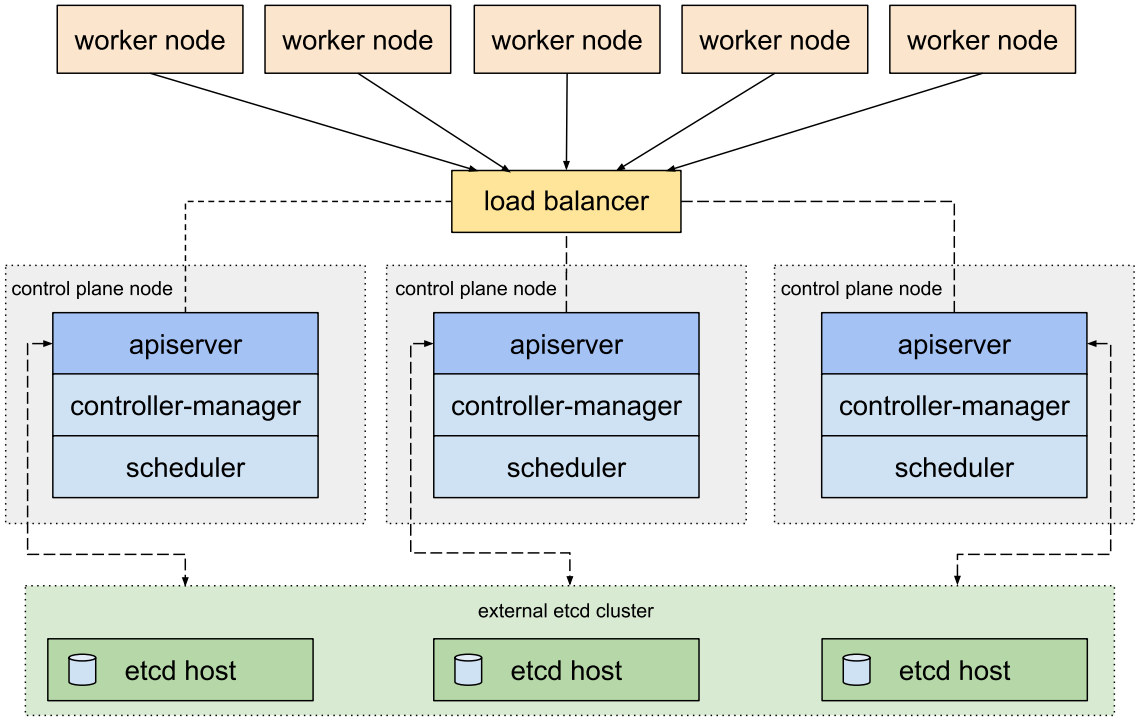
container runtime: es el tipo de gestor de contenedores que tendrá el nodo, docker, krio, ...

Topología

Kubernetes tiene dos formas de gestionar la base de datos en los nodos *control-panel* con la topolgía Stacked o External, entre ellas de diferencian en que Stacked crea una base de datos `etcd` en cada nodo *control-panel* y en el caso de tener diferentes nodos con esta función se sincroniza la base de datos. En cambio la topología external, la base de datos `etcd` esta en un nodo externo al *control-panel* .



kubeadm HA topology - external etcd



Grandes clusters

Kubernetes puede soportar una escalabilidad bastante amplia, pero como toda tecnología tiene sus limites, estos limites se establecen en:

- no mas de 5000 nodos
- no mas de 150000 pods totales
- no mas de 300000 containers
- no mas de 100 pods por nodo

Para el rendimiento optimo de kubernetes, este recomienda que los nodos master como mínimo tenga las siguientes características, respecto al numero de nodos a controlar.

Nodos	CPUs	RAM	Storage
1 - 5	2	4 GB	4 GB
6 - 10	2	8 GB	32 GB
11 - 100	4	16 GB	80 GB
101 - 250	8	30 GB	160 GB
251 - 500	16	30 GB	-
500+	36	61 GB	-

Como funciona?

En kubernetes siempre existirá como mínimo un nodo master (control-panel) y uno o mas nodos trabajadores. El nodo master se encarga de gestionar todo lo que ocurre en el cluster, donde desplegar los contenedores, servicios, ..., el master no corre contenedores de las aplicaciones del cluster, de esto ya se encargan los trabajadores.

En el cluster se crean redes virtuales internas para la comunicación entre contenedores, Pods y servicios. Los puertos de los servicios se pueden exportar al exterior de diferentes modos desde el nodo master para tener acceso desde el exterior a una aplicación del cluster.

Para el montaje de volumenes, siempre se necesitará un servicio de red externo al cluster, ya que cada aplicación corre en uno o varios nodos diferentes y no es recomendable el montaje local a no ser que sea cache o data sin importancia, para la data persistente se utiliza un servidor nfs externo o volumenes de aws, google cloud y similares.

Pod

Un pod es uno o mas contenedores que comparten namespaces entre si.

Los contenedores internos de un pod comparen NCT, IPC, UTS. es decir comparten ip, hostname, y procesos.

Para crear los contenedores del pod, primero crea un contenedor de configuración a este le extrae el id, y a partir de este id crea los contenedores necesarios, de esta forma pueden compartir la red, hostname y procesos entre ellos. para finalizar la creación del pod elimina el container de configuración.

comparten: IPC inter process comunication, Network, UTS Unix timesharing system

no comparten: mount, PID, USER, Cgroup.

El no compartir el mount, permite decidir que montar y no montar en cada contenedor, y el no compartir Cgroup da control sobre la cpu que consume cada container.

Instalación entorno pruebas

Para el entorno de pruebas es necesario instalar docker, kubectl y minikube.

kubectl es la aplicación con la que se gestiona kuberetes.

minikube es un entorno de pruebas muy útil para aprender a utilizar kubernetes, este genera un cluster de nodos donde se pueden hacer todo tipo de pruebas.

https://docs.docker.com/install/linux/docker-ce/debian/

https://kubernetes.io/docs/tasks/tools/install-kubectl/

https://kubernetes.io/es/docs/tasks/tools/install-minikube/

comandos basicos minikube

```
sudo minikube status
sudo minikube start
sudo minikube stop
sudo minikube delete
sudo minikube  -h
```

Instalación cluster real

Requisitos

- Sistema operativo: Ubuntu 16.04+, Debian9+, centOS7, Red Had Enterprise Lihnux (RHEL)7, Fedora25+, HypriotOS v1.0.1+
- 2GB como mínimo de RAM
- 2 CPUs como mínimo
- Conectividad entre todas las máquinas del cluster
- Nombre de host único, dirección MAC y product_uuid para cada nodo.
- desactivar Swap para el correcto funcionamiento de kubelet
- Ciertos puertos abiertos en las máquinas

Control-plane node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services†	All

CNI ports on both control-plane and worker nodes

Según la red de pods que se configure en el cluster también es necesario abrir los puertos necesarios para el modelo de red

Protocol	Port Number	Description
TCP	179	Calico BGP network
TCP	9099	Calico felix (health check)
UDP	8285	Flannel
UDP	8472	Flannel
TCP	6781-6784	Weave Net
UDP	6783-6784	Weave Net

Instalación

En este ejemplo de instalación se muestra como instalar un cluster de tres máquinas, un master (control-plane) y dos workers, esta instalación se realiza en máquinas virtuales fedora27 y la tipología del master es *stacked*.

Preparación de nodos

Asignar hostname a cada nodo

```
hostnamectl set-hostname master
hostnamectl set-hostname node1
hostnamectl set-hostname node2
```

Deshabilitar swap

```
swapoff -a
sed -i '/ swap / s/^/#/' /etc/fstab
```

Resolución de nombres

```
[jorge@master ~]$ cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.122.2 master
192.168.122.3 node1
192.168.122.4 node2
```

Ip fijas

```
[jorge@node1 ~]$ cat /etc/sysconfig/network-scripts/ifcfg-enp1s0
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
```

```
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=enp1s0
UUID=ed734a8b-a63b-3d9e-8016-b1a7a731f08a
ONBOOT=yes
AUTOCONNECT_PRIORITY=-999

BOOTPROTO=none
DEVICE=enp1s0
IPADDR=192.168.122.3
NETMASK=255.255.255.0
GATEWAY=192.168.122.1
DNS1=192.168.122.1
DNS2=1.1.1.1
```

Abrir puertos en el master y habilitar forwarding

```
firewall-cmd --permanent --add-port=6443/tcp
firewall-cmd --permanent --add-port=2379-2380/tcp
firewall-cmd --permanent --add-port=10250/tcp
firewall-cmd --permanent --add-port=10251/tcp
firewall-cmd --permanent --add-port=10252/tcp
firewall-cmd --permanent --add-port=10255/tcp
firewall-cmd --permanent --add-port=8472/udp
firewall-cmd --add-masquerade --permanent
# only if you want NodePorts exposed on control plane IP as well
firewall-cmd --permanent --add-port=30000-32767/tcp

systemctl restart firewalld


# Enable IP Forwarding
echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

Abrir puertos en los worker

```
firewall-cmd --permanent --add-port=10250/tcp
firewall-cmd --permanent --add-port=10255/tcp
firewall-cmd --permanent --add-port=8472/udp
firewall-cmd --permanent --add-port=30000-32767/tcp
firewall-cmd --add-masquerade --permanent

systemctl restart firewalld
```

Instalar servicios

En todos los nodos es necesario instalar docker, kubelet, kubectl, kubeadm.

Instalación de docker

https://docs.docker.com/engine/install/fedora/

Instalación de kubeam, kubectl, kubelet

```
# añadir repositorio de kubernetes
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-\$basearch
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kubelet kubeadm kubectl
EOF

# Set SELinux in permissive mode (effectively disabling it)
setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config

# instalar servicios y abilitar kubelet
dnf install -y kubelet kubeadm kubectl --disableexcludes=kubernetes
systemctl enable --now kubelet
```

- Esta instalación es para distribuciones fedora para otras distribuciones consultar aquí

Verificar instalación de kubeadm

```
[root@master jorge]# kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.2",
GitCommit:"52c56ce7a8272c798dbc29846288d7cd9fbae032", GitTreeState:"clean",
BuildDate:"2020-04-16T11:54:15Z", GoVersion:"go1.13.9", Compiler:"gc",
Platform:"linux/amd64"}
```

Creación de master

En la creación del master hay que tener en que red se crearan los Pods, esta opción kubernetes lo deja en addons externos, puedes elegir entre diferentes opciones que encontrarás en este documento. En esta instalación se asignara el addon calico.

> **Nota**: El dns del cluster CoreDNS no se iniciará si no hay antes una red de Pods instalada.

Iniciar configuración del master con red de pod

```
kubeadm init --pod-network-cidr=192.168.0.0/16
```

> Este comando descarga imágenes que necesita el cluster para funcionar y tarda un rato en completarse.

El comando anterior acaba mostrando las siguientes instrucciones a realizar, para la finalización de la instalación del master.

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as
root:

kubeadm join 192.168.122.2:6443 --token 13whh6.k1mj5fu316n7kjfy \
    --discovery-token-ca-cert-hash
sha256:6a0e0da8c83e9136a099f2ae11d17e39a9f6697fcc910c60c7634aef30a0dc1f
```

> Es muy importante guardarse bien la línea de `kubeadm join` ya que con esta juntaremos los nodos al master.

En caso de no querer gestionar el cluster como root, y quererlo gestionar como usuario.

```
[jorge@master ~]$ mkdir -p $HOME/.kube
[jorge@master ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[jorge@master ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Aplico el addon para gestionar las redes de Pods

```
kubectl apply -f https://docs.projectcalico.org/v3.11/manifests/calico.yaml
```

Esta opción no es recomendable: Por razones de seguridad por defecto el nodo master no ejecuta Pods del cluster, en caso de querer que haga de master y nodo simultáneamente, por ejemplo para entornos de pruebas de un cluster de un solo nodo, aplicar la siguiente opción.

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

Juntar nodos worker

Para juntar un worker al nodo master simplemente hay que ejecutar `kubeadm join` con el token del master.

```
kubeadm join 192.168.122.2:6443 --token namzwa.gri0onsuwamo19gg \
    --discovery-token-ca-cert-hash
sha256:3bd21a07fdbaf780d5bf8103a5dff2a46945219393fda74897579d43524c931a
```

- Si esta opción muestra algún error, es posible que el firewall del master este rechazando la conexión, comprobar los puertos accesibles por el firewall.
- En caso de no disponer del token, siempre se puede crear uno nuevo desde el master con `kubeadm token create --print-join-command`

Ahora desde el master se puede ver que el nodo se a añadido al master, es una buena practica asignarle algún label al nodo, para identificar que modo de nodo es y por si se quieren asignar pods específicos al nodo.

```
# nodo recien añadido
[jorge@master ~]$ kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
master   Ready    master   22m   v1.18.2
node1    Ready    <none>   19m   v1.18.2

# asignar label de tipo de nodo
[jorge@master ~]$ kubectl label node node1 node-role.kubernetes.io/worker=worker
node/node1 labeled

[jorge@master ~]$ kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
master   Ready    master   27m   v1.18.2
node1    Ready    worker   23m   v1.18.2

# asignar label extra de identificación de nodo
[jorge@master ~]$ kubectl label nodes node1 node=worker1
node/node1 labeled

# visualizar labels de nodos
[jorge@master ~]$ kubectl get nodes --show-labels
```

Eliminar un nodo

Desde el nodo master drenar el nodo que queremos eliminar, una vez drenado de todas sus tareas eliminarlo.

```
kubectl drain <node name> --delete-local-data --force --ignore-daemonsets
kubectl delete node <node name>
```

En el nodo que se a eliminado del cluster restablecer la configuración inicial

```
kubeadm reset
```

El proceso de reinicio no reinicia ni limpia las reglas de iptables o las tablas de IPVS. Si desea restablecer iptables, debe hacerlo manualmente:

```
iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X
```

Si desea restablecer las tablas IPVS, debe ejecutar el siguiente comando:

```
ipvsadm -C
```

Instalación cluster EKS

La instalación de un cluster mediante la herramienta `eksctl` de aws permite desplegar un cluster rápidamente en la plataforma aws.

Requisitos

Instalar aws-clie

Para poder manejar la api de aws desde la linea de comandos es necesario crear un usuario IAM con los permisos apropiados a las acciones que se quieran ejecutar.

En los pasos que se muestran a continuación se crea un usuario con permisos de administración.

documentación oficial

Crear usuario IAM

https://console.aws.amazon.com/iam/

1. Iniciar sesión con usuario raíz de AWS
2. En la barra superior ir a `mi cuenta` y bajar hasta `acceso de los roles y usuarios de IAM`
    - (editar) y Activar el acceso de usuarios IAM
    - Volvemos a la barra superior `servicios` --> sección `IAM`
3. Crear un usuario IAM administrador nuevo
    - Buscamos a la izquierda la opción `usuarios` --> nuevo --> dar permisos de acceso a la consola
4. Crear grupo de administradores
    - poner nombre al grupo (administradores por ejemplo) y en las políticas señalar `administratorAccess` (que es permiso a todo)
5. Cerrar sesión con el usuario raíz y entrar con el usuario administrador a la cuenta IAM
    - clicar usuario -- credenciales de seguridad
        - crear clave de acceso
        - se genera automáticamente unas claves que se tendrán que guardar, estas claves son las que indicaras en cada nodo para manipular aws desde la terminal.

Instalar comando AWS

```
➜  ~ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
➜  ~ unzip awscliv2.zip

➜  ~  sudo ./aws/install
➜  ~  aws --version
```

Configurar aws-clie

Primero de todo se a de configurar aws para conectar con la cuenta deseada.

Las claves de usuario proporcionadas por `IAM` son las que se tienen que indicar a `aws-cli` para conectar y la región donde quieres trabajar.

```
➜ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

Comprobar

Crear un volumen de prueba en aws desde consola del nodo.

```
➜ aws ec2 create-volume --availability-zone=eu-west-2a --size=5 --volume-
type=gp2
```

```
{
    "AvailabilityZone": "eu-west-2a",
    "CreateTime": "2020-04-11T19:58:18+00:00",
    "Encrypted": false,
    "Size": 5,
    "SnapshotId": "",
    "State": "creating",
    "VolumeId": "vol-0fb30c332ecff512d",
    "Iops": 100,
    "Tags": [],
    "VolumeType": "gp2"
}
```

Instalar eksctl

`eksctl` es el herramienta que proporciona aws para la creación de clusteres kubernetes en la nube aws desde una terminal

```
curl --silent --location
"https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -
s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/local/bin
eksctl version
```

Instalar kubectl

`kubectl` es necesario para acceder a la api del cluster y gestionarlo.

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF

dnf install -y kubectl
kubectl version
```

- Instalación otras distribuciones aqui

Desplegar cluster

Al desplegar un cluster con `eksctl` se han de tener en cuenta opciones como: la región donde desplegar, el número de nodos, el tipo ne nodo, entre otras opciones adicionales mostradas con `eksctl create cluster -h`.

En el siguiente ejemplo se despliega un cluster en la región de londres de dos nodos con 2 CPU y 2GB RAM cada uno, con acceso mediante ssh desde el host que se crea.

> La instalación del cluster tarda entre 10 y 15 minutos.

```
eksctl create cluster \
--name first-eks \
--region eu-west-2 \
--nodegroup-name standard-nodes \
--node-type t3.small \
--nodes 2 \
--nodes-min 2 \
```

```
--nodes-max 2 \
--ssh-access \
--ssh-public-key .ssh/id_rsa.pub \
--managed

[ℹ]  eksctl version 0.18.0
[ℹ]  using region eu-west-2
[ℹ]  setting availability zones to [eu-west-2a eu-west-2b eu-west-2c]
[ℹ]  subnets for eu-west-2a - public:192.168.0.0/19 private:192.168.96.0/19
[ℹ]  subnets for eu-west-2b - public:192.168.32.0/19 private:192.168.128.0/19
[ℹ]  subnets for eu-west-2c - public:192.168.64.0/19 private:192.168.160.0/19
[ℹ]  using SSH public key ".ssh/id_rsa.pub" as "eksctl-first-eks-nodegroup-
standard-nodes-9a:84:e3:29:f3:d4:5d:4b:23:18:da:b6:b5:bd:fb:68"
[ℹ]  using Kubernetes version 1.15
[ℹ]  creating EKS cluster "first-eks" in "eu-west-2" region with managed nodes
[ℹ]  will create 2 separate CloudFormation stacks for cluster itself and the
initial managed nodegroup
[ℹ]  if you encounter any issues, check CloudFormation console or try 'eksctl
utils describe-stacks --region=eu-west-2 --cluster=first-eks'
[ℹ]  CloudWatch logging will not be enabled for cluster "first-eks" in "eu-west-
2"
[ℹ]  you can enable it with 'eksctl utils update-cluster-logging --region=eu-
west-2 --cluster=first-eks'
[ℹ]  Kubernetes API endpoint access will use default of {publicAccess=true,
privateAccess=false} for cluster "first-eks" in "eu-west-2"
[ℹ]  2 sequential tasks: { create cluster control plane "first-eks", create
managed nodegroup "standard-nodes" }
[ℹ]  building cluster stack "eksctl-first-eks-cluster"
[ℹ]  deploying stack "eksctl-first-eks-cluster"
[ℹ]  building managed nodegroup stack "eksctl-first-eks-nodegroup-standard-nodes"
[ℹ]  deploying stack "eksctl-first-eks-nodegroup-standard-nodes"
[✔]  all EKS cluster resources for "first-eks" have been created
[✔]  saved kubeconfig as "/home/debian/.kube/config"
[ℹ]  nodegroup "standard-nodes" has 2 node(s)
[ℹ]  node "ip-192-168-62-60.eu-west-2.compute.internal" is ready
[ℹ]  node "ip-192-168-67-118.eu-west-2.compute.internal" is ready
[ℹ]  waiting for at least 2 node(s) to become ready in "standard-nodes"
[ℹ]  nodegroup "standard-nodes" has 2 node(s)
[ℹ]  node "ip-192-168-62-60.eu-west-2.compute.internal" is ready
[ℹ]  node "ip-192-168-67-118.eu-west-2.compute.internal" is ready
[ℹ]  kubectl command should work with "/home/debian/.kube/config", try 'kubectl
get nodes'
[✔]  EKS cluster "first-eks" in "eu-west-2" region is ready
```

La instalación ya configura el acceso al cluster mediante `kubectl` automáticamente.

```
➜ kubectl get nodes
NAME                                           STATUS   ROLES    AGE     VERSION
ip-192-168-62-60.eu-west-2.compute.internal    Ready    <none>   3m18s
v1.15.10-eks-bac369
ip-192-168-67-118.eu-west-2.compute.internal   Ready    <none>   2m26s
v1.15.10-eks-bac369
```

Eliminar cluster

```
# ver clusteres desplegados
➜ eksctl get cluster --region eu-west-2
NAME        REGION
first-eks   eu-west-2

# eliminar cluster
➜ eksctl delete cluster --name first-eks --region eu-west-2
```

Gestionar pod manualmente

https://kubernetes.io/docs/reference/kubectl/conventions/

Al crear un pod hay que tener en cuenta la versión de la api que estamos utilizando y que la imagen desde la que creará el contenedor la cogerá de docker.

También hay que tener en cuenta, que crear un pod desde linea de comandos es una mala practica, lo recomendable es utilizar manifiestos o un objeto de mas alto nivel, ya que pod por si mismo no puede auto ejecutarse, por lo tanto si muere se tendrá que arrancar manualmente otra vez.

Importante: En el caso de que un container de un pod fallara, el pod se crea, pero no da datos de que a fallado, en ese caso se tendrá que inspeccionar el pod con `kubectl describe pod <nombre-pod> o kubectl logs <nombre-pod>`

```
➜  sudo kubectl api-versions # ver versiónnde kubectl
➜  ~ sudo kubectl run --generator=run-pod/v1 podtest --image=nginx:alpine
pod/podtest created

➜  ~ sudo kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
podtest   1/1     Running   0          12s
```

- con `get pods` se puede ver como a creado el pod y su estado es corriendo, en la columna `READY` se ve que esta corriendo un contenedor de uno, ya que en un pod puede haber diferentes contenedores.

Comandos de gestion en pods

```
# crear pod
➜  sudo kubectl run --generator=run-pod/v1 podtest --image=nginx:alpine
➜  sudo kubectl delete pod podtest1 # eliminar pod

# informativos
➜  sudo kubectl describe  pod podtest # informacion del pod
➜  sudo kubectl get pod podtest -o yaml # extraer info de pot en yaml
```

```
➜  sudo kubectl api-resources # diferentes recursos de la api

# entrada
➜  sudo kubectl exec -it podtest -- sh # entrar pod con 1 contenedor
➜  sudo kubectl exec -it doscont -c contenedor2 -- sh # entrar contenedor2 del
pod doscont

# logs
➜  sudo kubectl logs podtest    # ver logs
➜  sudo kubectl logs podtest -f # vre logs interactivos
➜  sudo kubectl logs doscont -c contenedor1 # ver logs del contendor1

# manifiestos
➜  sudo kubectl apply -f pod.yaml  # aplicar manifiesto
➜  sudo kubectl delete -f pod.yaml # eliminar manifiesto
```

Manifiestos

Un manifiesto es un archivo en yaml , yml o json que define el recurso que queremos crear o actualizar en kubernetes, la ventaja que proporciona esta practica, es que, se puede desplegar uno o mas pods a la vez y estos teniendo uno o mas contenedores en su interior.

https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

Simple

Ejemplo de manifiesto muy simple, con un pod podtest2 y un contenedor de una imagen nginx.

pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: podtest2
spec:
  containers:
  - name: contenedor1
    image: nginx:alpine
```

Desplegar manifiesto.

```
➜  pods sudo kubectl apply -f pod.yaml
pod/podtest2 created
➜  pods sudo kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
podtest2   1/1     Running   0          15s

➜  pods sudo kubectl delete -f pod.yaml
```

Dos pods

En este ejemplo de manifiesto se crean dos pods, para identificar donde acaba un pod y donde empieza el siguiente se define con `---` .

```
apiVersion: v1
kind: Pod
metadata:
  name: podtest2
spec:
  containers:
  - name: contenedor1
    image: nginx:alpine
---
apiVersion: v1
kind: Pod
metadata:
    name: podtest3
spec:
    containers:
    - name: contenedor1
      image: nginx:alpine
```

Se puede ver como al cargarlo se crean dos pods a la vez.

```
➜  pods sudo kubectl apply -f pod.yaml
pod/podtest2 unchanged
pod/podtest3 created
➜  pods sudo kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
podtest2   1/1     Running   0          45s
podtest3   1/1     Running   0          19s

➜  pods sudo kubectl delete -f pod.yaml
pod "podtest2" deleted
pod "podtest3" deleted
```

Dos contenedores

Al crear dos contenedores en el mismo pod se tiene que tener en cuenta, que estos dos contenedores están compartiendo ip, pero nunca se puede compartir puerto.

```
apiVersion: v1
kind: Pod
metadata:
  name: doscont
spec:
  containers:
  - name: contenedor1
    image: python:3.6-alpine
    command: ['sh', '-c', 'echo "contenedor1" > index.html && python -m
http.server 8082']
  - name: contenedor2
    image: python:3.6-alpine
    command: ['sh', '-c', 'echo "contenedor2" > index.html && python -m
http.server 8083']
```

Cargar manifiesto.

```
➜  pods sudo kubectl apply -f doscont.yaml
pod/doscont created

➜  pods sudo kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
doscont   2/2     Running   0          8s
```

n el caso que un contenedor fallara, se vería de esta manera, `READY 1/2`

```
➜  pods sudo kubectl get pods
NAME      READY   STATUS             RESTARTS   AGE
doscont   1/2     CrashLoopBackOff   6          7m58s
```

Labels

Los labels juegan un papel importante en los pods, esto es por dos razones:

- Los objetos de mas alto nivel se pueden referir a los pods mediante labels, es decir puedo indicar aumenta las replicas de los backend, siendo backend un label.
- Se puede filtrar el output de pods mediante labels.

La sección de label se define dentro de metadata, y dentro de label, las variables que indiques son a tu gusto, aun que una buena practica es definir app como mínimo.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: podtest2
  labels:
    app: front
    env: dev
spec:
  containers:
  - name: contenedor1
    image: nginx:alpine
```

Ejemplo de filtro.

```
➜  sudo kubectl get pods -l app=backend
NAME       READY   STATUS    RESTARTS   AGE
podtest3   1/1     Running   0          46s
➜  sudo kubectl get pods -l env=dev
NAME       READY   STATUS    RESTARTS   AGE
podtest2   1/1     Running   0          5m54s
podtest3   1/1     Running   0          5m54s
```

Asignar pod a un nodo

En ocasiones se quiere asignar un Pod a un nodo especifico, esto se puede hacer gracias a que el nodo tiene un label que le hemos asignado anteriormente `node: worker1` y desde el template del pod indicamos que ese pod se asignara en el label indicado en `nodeSelector`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    node: worker1
```

Replicaset

https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

Replicaset es un objeto superior a los pods, este se encarga de tener pods replicados, es decir si yo indico quiero dos pods siempre activos, en el caso que uno se muera levantará otro automáticamente.

Importante: replicaset solo se encarga de mantener el numero replicas activas, si muere un pod levantará otro, para actualizar los pods a nuevas versiones se debe utilizar deployment. Puedes actualizar el replicaset a nivel de replicas pero no es recomendable actualizar los pods, desde este nivel.

Como identifica los pods

Esta gestión la hace a través de que el replicaset inserta en sus pods la metadata `owner reference` (referencia al propietario), replicaset busca por un label indicado, si encuentra pods con ese label, lo hereda e introduce la metadata owner si no encuentra ninguno lo crea con su marca owner.

Gestión de replicaset

Características de replicaset son que es un `kind` (objeto) replicaset, este tendrá su propio nombre y label, en las especificaciones se indica el numero de replicas y el label de identificación para sus pods. A partir del template se especifica los datos del pod, este pod no tendrá name ya que se lo añade replicaset.

rs.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-test
  labels:
    app: rs-test
spec:
  replicas: 5
  selector:
    matchLabels:
      app: pod-label
# aqui comienza la especificacion del pod
  template:
    metadata:
      labels:
        app: pod-label
    spec:
      containers:
      - name: contenedor1
        image: python:3.6-alpine
        command: ['sh', '-c', 'echo "contenedor1" > index.html && python -m
http.server 8082']
      - name: contenedor2
        image: python:3.6-alpine
        command: ['sh', '-c', 'echo "contenedor2" > index.html && python -m
http.server 8083']
```

Se puede ver como al lanzar lo se crean las replicas automáticamente.

```
➜ sudo kubectl apply -f rs.yaml
replicaset.apps/rs-test created

➜ sudo kubectl get replicaset
NAME      DESIRED   CURRENT   READY   AGE
rs-test   5         5         5       2m13s

➜ sudo kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
rs-test-8gbzm   2/2     Running   0          14s
rs-test-db9kz   2/2     Running   0          14s
rs-test-nfnsw   2/2     Running   0          14s
rs-test-vrghj   2/2     Running   0          14s
rs-test-z85fq   2/2     Running   0          14s
```

Si elimino un pod automáticamente se crea uno nuevo.

```
➜ sudo kubectl delete pod rs-test-8gbzm
pod "rs-test-8gbzm" deleted

➜ sudo kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
rs-test-54qcj   2/2     Running   0          62s
rs-test-db9kz   2/2     Running   0          5m13s
rs-test-nfnsw   2/2     Running   0          5m13s
rs-test-vrghj   2/2     Running   0          5m13s
rs-test-z85fq   2/2     Running   0          5m13s
```

Replicaset permite actualizar su contenido, en este ejemplo e modificado en el archivo `rs.yaml` el dato `replicas: 2` y automáticamente al aplicarlo el gestiona los cambios.

```
➜  replicaset git:(master) ✗ sudo kubectl apply -f rs.yaml
replicaset.apps/rs-test configured

➜  replicaset git:(master) ✗ sudo kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
rs-test-db9kz   2/2     Running   0          6m32s
rs-test-z85fq   2/2     Running   0          6m32s
```

Deployment

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Deployment es el objeto encargado de gestionar las actualizaciones de versiones de los pods y replicasets, por lo tanto es un objeto superior de replicaset y pod, los engloba. Es decir un deployment tiene en su interior un replicaset que este en su interior tiene un pod.

Como gestiona las actualizaciones

Deployment tiene el replicaset versión 1 en su interior, cuando tu le añades una actualización, deployment crea un nuevo replicaset versión 2 y va encendiendo pods del replicaset versión 2 y apagando los de la versión 1, para así asegurar una transición sin interrupciones. Además se guarda por defecto hasta 10 versiones por si quieres hacer un rollback (ir a una versión anterior por si a habido problemas).

Gestion de manifiesto

Ejemplo manifiesto

En el siguiente manifiesto se especifica, crear un deployment que contenga un replicaset que creara tres replicas de una pod con un container nginx.

Ejemplo deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: "version inicial nginx"
  name: deployment-test
  labels:
    app: front-nginx
spec:
#   revisionHistoryLimit: 3
  replicas: 3
  selector:
    matchLabels:
      app: front-nginx
  template:
    metadata:
      labels:
        app: front-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
```

Desplegar deployment

```
sudo kubectl apply -f dep.yaml
deployment.apps/deployment-test created

sudo kubectl get deployment --show-labels
NAME              READY   UP-TO-DATE   AVAILABLE   AGE   LABELS
deployment-test   3/3     3            3           40s   app=front-nginx

sudo kubectl rollout status deployment deployment-test
deployment "deployment-test" successfully rolled out
```

Comprobar que ha creado

Efectivamente a creado un resource con tres pods

```
sudo kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
deployment-test-694b78cb4c    3         3         3       6m32s

sudo kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deployment-test-694b78cb4c-mcm27    1/1     Running   0          6m51s
deployment-test-694b78cb4c-r62tx    1/1     Running   0          6m51s
deployment-test-694b78cb4c-w8t2p    1/1     Running   0          6m51s
```

Actualizar

Actualmente los pods están con la imagen `nginx:alpine` y voy a cambiar el manifiesto para hacer una actualización con la imagen `nginx:latest` y así ver como se comporta.

```
sudo kubectl get pods deployment-test-694b78cb4c-mcm27 -o yaml | grep 'image:'

  - image: nginx:alpine
    image: nginx:alpine
```

hago el cambio en el manifiesto, lo despliego y veo el resultado.

```
sudo kubectl apply -f dep.yaml
deployment.apps/deployment-test configured

sudo kubectl get pods deployment-test-78895f88b7-2z525 -o yaml | grep 'image:'
  - image: nginx:latest
    image: nginx:latest
```

Se puede ver que a echo exactamente con `describe` , se que crea un nuevo replicaset y va encendiendo pods a la vez que los apaga en el antiguo.

```
sudo kubectl describe deployment deployment-test
Events:
  Type    Reason             Age                    From
Message
  ----    ------             ----                   ----                   -----
--
  Normal  ScalingReplicaSet  31m                    deployment-controller
Scaled up replica set deployment-test-694b78cb4c to 3
  Normal  ScalingReplicaSet  10m                    deployment-controller
Scaled up replica set deployment-test-78895f88b7 to 1
  Normal  ScalingReplicaSet  10m                    deployment-controller
Scaled down replica set deployment-test-694b78cb4c to 2
  Normal  ScalingReplicaSet  10m                    deployment-controller
Scaled up replica set deployment-test-78895f88b7 to 2
  Normal  ScalingReplicaSet  10m                    deployment-controller
Scaled down replica set deployment-test-694b78cb4c to 1
  Normal  ScalingReplicaSet  10m                    deployment-controller
Scaled up replica set deployment-test-78895f88b7 to 3
  Normal  ScalingReplicaSet  10m                    deployment-controller
Scaled down replica set deployment-test-694b78cb4c to 0
```

Ahora ahí dos replicasets, el nuevo y el antiguo se guarda por si quiero volver a una versión anterior a la actual.

```
sudo kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
deployment-test-694b78cb4c    3         3         3       35m
deployment-test-78895f88b7    0         0         0       15m

sudo kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

History

El historial de versiones por defecto no añade ninguna anotación y en el caso de tener diferentes versiones, es difícil saber cual es cual. para solucionar esto existen diferentes maneras de añadir un comentario.

- `--record` al lanzar la actualización, se añade la misma linea del comando.
- con metadata `annotations` en el archivo yaml (es la mas practica)
- Manualmente desde linea de commando (muy engorrosa)

Este es un ejemplo por defecto:

```
sudo kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
1         <none>
```

Metadata

Esta es la manera más agradable de insertar los comentarios del historial, desde la metadata del archivo yaml en la sección deploymet.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: "cambio puerto a 110"
  name: deployment-test
  labels:
    app: front-nginx
[ texto recortado ]
```

Al desplegar se ve de la siguiente manera.

```
sudo kubectl apply -f deployment/dep.yaml
deployment.apps/deployment-test configured

sudo kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
1         <none>
2         cambio puerto a 110
```

Cambio manual

El cambio manual de anotación de versión es muy engorroso, pero útil si te has olvidado cambiarlo anteriormente o era erróneo.

```
sudo kubectl annotate deployment.v1.apps/deployment-test kubernetes.io/change-cause="cambio manual de anotacion"

sudo kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
1         <none>
2         cambio manual de anotacion
```

Record

El modo record, no es del todo útil ya que no de una descripción como tal.

```
sudo kubectl apply -f deployment/dep.yaml --record
deployment.apps/deployment-test configured

sudo kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
1         <none>
2         cambio puerto a 110
3         kubectl apply -f deployment/dep.yaml --record=True
```

Roll-back

Roll-back permite regresar a una versión anterior con la siguiente sintaxis: `kubectl rollout undo deployment deployment-test --to-revision=2` . Esto es muy útil si al desplegar una nueva versión a habido algún tipo de error y se quiere volver a una versión estable.

En el siguiente ejemplo muestro como añado una versión con una imagen inexistente, dará errores y volveré a la anterior estable.

```
# lanzo nueva versión
sudo kubectl apply -f deployment/dep.yaml
deployment.apps/deployment-test configured

# veo como se añade la nueva versión
sudo kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
1         <none>
2         cambio manual de anotacion
3         cambio version imagen fake

# al ver el estado del proceso de cambio, veo que no avanza
sudo kubectl rollout status deployment deployment-test
Waiting for deployment "deployment-test" rollout to finish: 1 out of 3 new
replicas have been updated...

# y veo que no puede hacer la transicion de pods, porque no puede descargar la
imagen
sudo kubectl get pods
NAME                               READY   STATUS             RESTARTS   AGE
deployment-test-64bcc5dd9c-4b9rk   1/1     Running            0          21m
deployment-test-64bcc5dd9c-bkp5v   1/1     Running            0          21m
deployment-test-64bcc5dd9c-rlc6l   1/1     Running            0          21m
deployment-test-7b799b8bcf-gw9nl   0/1     ImagePullBackOff   0          3m49s

# vuelvo a la versión 2 que era estable
sudo kubectl rollout undo deployment deployment-test --to-revision=2
deployment.apps/deployment-test rolled back

# parece haber ido bien
sudo kubectl rollout status deployment deployment-test
```

```
deployment "deployment-test" successfully rolled out

# los pods vuleven a estar correctos
sudo kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
deployment-test-64bcc5dd9c-4b9rk   1/1     Running   0          24m
deployment-test-64bcc5dd9c-bkp5v   1/1     Running   0          24m
deployment-test-64bcc5dd9c-rlc6l   1/1     Running   0          24m

# compruevo que su versión de imagen es la correcta
sudo kubectl describe pod deployment-test-64bcc5dd9c-bkp5v | grep -i image:
    Image:          nginx:alpine
```

Servicios & endpoints

El **objeto services** es el encargado de balancear la carga entre los diferentes pods, esto lo hace a trabes de los labels para identificar a que pods de estar observando, no importa si esos pods están en un replicaset u otro, solo mira labels de pods y hay que tener en cuenta eso.

El balanceo de carga sirve (en el caso de una web) para aumentar las peticiones que puede llegar a recibir al mismo tiempo, ya que se distribuirán entre los múltiples pods en vez de uno solo. es decir un cliente hace la petición a una ip y el objeto services se encarga a redirigir esa petición al pod indicado.

El **endpoint** de un servicio es el encargado de guardar la lista de ip's de los pods a consultar, en el caso de que un pod muera y se arranque otro, borrara la ip del pod muerto y añadirá la del pod nuevo.

Un servicio siempre esta asociado a una serie de pods, por lo tanto creo un deployment con pods nginx y lo asocio a el servicio, en el que expongo el puerto 80 de los pods al 8080 de la ip virtual del servicio.

Ejemplo de servicio:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
  labels:
    app: front-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: front-nginx
  template:
    metadata:
      labels:
        app: front-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
```

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

Desplegar servicio.

```
sudo kubectl apply -f service/service.yaml
deployment.apps/deployment-test created
service/my-service created

sudo kubectl get svc -l app=front-nginx
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
my-service   ClusterIP   10.99.212.24   <none>        8080/TCP   93s
```

Descripción del servicio, donde se ve que a creado su endpoint, con las ips de los pods que balanceará la carga.

```
# descripción del servicio creado
sudo kubectl describe service my-service
Name:              my-service
Namespace:         default
Labels:            app=front-nginx
Annotations:       kubectl.kubernetes.io/last-applied-configuration:
                     {"apiVersion":"v1","kind":"Service","metadata":
{"annotations":{},"labels":{"app":"front-nginx"},"name":"my-
service","namespace":"default"}...
Selector:          app=front-nginx
Type:              ClusterIP
IP:                10.99.212.24
Port:              <unset>  8080/TCP
TargetPort:        80/TCP
Endpoints:         172.17.0.4:80,172.17.0.5:80,172.17.0.6:80
Session Affinity:  None
Events:            <none>

# crea automaticamente su endpoint
sudo kubectl get endpoints
NAME         ENDPOINTS                                      AGE
kubernetes   192.168.88.2:8443                              9d
my-service   172.17.0.4:80,172.17.0.5:80,172.17.0.6:80   7m31s
```

```
# pods con ip asociadas al deployment
sudo kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE     IP
    NODE   NOMINATED NODE   READINESS GATES
deployment-test-5699757d6-77ps4   1/1     Running   0          8m18s
172.17.0.5   pc02   <none>           <none>
deployment-test-5699757d6-kk96q   1/1     Running   0          8m18s
172.17.0.6   pc02   <none>           <none>
deployment-test-5699757d6-xb2rc   1/1     Running   0          8m18s
172.17.0.4   pc02   <none>           <none>
```

Tipos de servicios

clusterip es la opción por defecto y consiste en una ip virtual que se le asigna al servicio, esta ip es interna, es decir desde mi ip local no podre ver el servicio, solo podre ver mi servicio indicando localmente la ip virtual que se le a añadido a dicho cluster.

nodeport es una capa superior a clusterip, y proporciona una redirección a la ip local del host (a localhost no, solo la local). por defecto asigna un puerto aleatorio entre el rango 30000-32767 , a no ser que se le indique manualmente en que puerto salir, eso si, entre el rango permitido de 30000-32767.

En el siguiente ejemplo se ve como a redireccionado automáticamente al puerto local `30425` .

```
sudo kubectl get svc
NAME               TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes         ClusterIP   10.96.0.1      <none>        443/TCP          9d
my-service         ClusterIP   10.99.212.24   <none>        8080/TCP         57m
my-service-nginx   NodePort    10.99.15.134   <none>        8080:30425/TCP   20m
```

Ejemplo de nodeport con redirección manual de puerto local

```
apiVersion: v1
kind: Service
metadata:
  name: my-service-nginx
  labels:
    app: backend-nginx
spec:
  type: NodePort
  selector:
    app: backend-nginx
  ports:
    - protocol: TCP
      nodePort: 30001
      port: 8080
      targetPort: 80
```

LoadBalancer Ee un tipo de balanceador de carga desarrollado para el cloud, este servicio estaría funcionando en la nube y balancea la carga a diferentes nodos, donde cada nodo se crea automaticamente su NodePort y ClusterIp

Namespaces

Namespaces son espacios de trabajo dentro de un cluster, donde cada namespace es independiente del otro, es decir los services, deployments, replicasets y pods no se ven entre cada espacio de trabajo. esto es especialmente bueno para trabajar con diferentes departamentos de trabajo, como desarrolladores, sysadmin, ... Otra ventaja es que se puede limitar los espacios de trabajo en los recursos que podrán consumir cada uno de ellos y también en acceso.

Por defecto kubernetes tiene los espacios de trabajo que no debemos tocar `kube-public, kube-system y kube-node-lease` estos espacios de trabajo los utiliza kubernetes y no es buena practica trastear en ellos. El usuario o administrador trabaja en el namespace default o uno creado para un grupo especifico.

```
sudo kubectl get namespaces
NAME              STATUS   AGE
default           Active   15d
kube-node-lease   Active   15d
kube-public       Active   15d
kube-system       Active   15d
```

Comandos

```
sudo kubectl get namespaces    # visualizar
sudo kubectl create namespace test-ns # crear
sudo kubectl delete namespace test-ns # eliminar
sudo kubectl describe namespaces test-ns # inspeccionar
# ver objeto de namespace -n abreviacion
sudo kubectl get pods|replicasets|deployments|services --namespace test-ns
sudo kubectl get pods|replicasets|deployments|services -n test-ns
# ver de todos los namespaces -A abreviacion de all-namespaces
sudo kubectl get pods|replicasets|deployments|services -A
```

Gestión manual

```
# Crear namespace simple
sudo kubectl create namespace test-ns
namespace/test-ns created

sudo kubectl get namespaces
NAME              STATUS   AGE
default           Active   15d
kube-node-lease   Active   15d
kube-public       Active   15d
kube-system       Active   15d
```

```
test-ns           Active   5s

# ver contenido namspace
sudo kubectl describe namespaces test-ns
Name:         test-ns
Labels:       <none>
Annotations:  <none>
Status:       Active

No resource quota.
No LimitRange resource.
```

resource quota: Es la limitación de cierta cantidad de recursos, que el namespace no va a ser sobrepasar.

limitRange resource: Es una manera de controlar recursos a nivel de objetos individuales. Es decir, pods, deployments, replicasets, ...

Ejemplo espacios de trabajo

```
# creo un pod en el namespace test-ns
sudo kubectl run   --generator=run-pod/v1 podtest --image=nginx:alpine --
namespace test-ns
pod/podtest created

# desde el namespace default no hay nada
sudo kubectl get pods
No resources found in default namespace.

# pero en el namespace test-ns se a creado
sudo kubectl get pods -n test-ns
NAME      READY   STATUS    RESTARTS   AGE
podtest   1/1     Running   0          26s

sudo kubectl delete pod podtest -n test-ns
pod "podtest" deleted

sudo kubectl delete namespaces test-ns
namespace "test-ns" deleted
```

Template

En el siguiente templte se crean dos namespaces, con un deployment en cada uno. Para indicar que un deployment pertenece a un namespace, se le añade en el deployment la metadata `namespace: nombre-del-namespace` .

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
  labels:
    name: dev
---
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    name: prod
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: "version inicial"
  name: deployment-test
  namespace: dev
  labels:
    app: front-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: front-nginx
  template:
    metadata:
      labels:
        app: front-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: "version inicial"
  name: deployment-prod
  namespace: prod
  labels:
    app: front-nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: front-nginx
  template:
    metadata:
      labels:
        app: front-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
```

Se pude ver como se crea en cada namespace su deployment con sus pods independientes del cada espacio de trabajo.

```
sudo kubectl apply -f  namespaces/envs-ns.yml
namespace/dev unchanged
namespace/prod unchanged
deployment.apps/deployment-test created
deployment.apps/deployment-prod created

# en el namespace default no hay nada
sudo kubectl get deploy
No resources found in default namespace.

# el namespace dev esta corriendo el deployment con sus pods
sudo kubectl get deploy -n dev
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test   2/2     2            2           81s

sudo kubectl get pods -n dev
NAME                               READY   STATUS    RESTARTS   AGE
deployment-test-694b78cb4c-2lfp9   1/1     Running   0          2m7s
deployment-test-694b78cb4c-ddcv4   1/1     Running   0          2m7s

# en el namespace prod corre su deploy y sus pods
sudo kubectl get deploy -n prod
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment-prod   4/4     4            4           99s

sudo kubectl get pods -n prod
NAME                               READY   STATUS    RESTARTS   AGE
deployment-prod-694b78cb4c-c8w4f   1/1     Running   0          2m
deployment-prod-694b78cb4c-f5bx6   1/1     Running   0          2m
deployment-prod-694b78cb4c-w57ch   1/1     Running   0          2m
deployment-prod-694b78cb4c-z66ln   1/1     Running   0          2m

# la opcion all-namespaces muestra todo
sudo kubectl get deploy -A
NAMESPACE     NAME              READY   UP-TO-DATE   AVAILABLE   AGE
dev           deployment-test   2/2     2            2           6m59s
kube-system   coredns           2/2     2            2           15d
prod          deployment-prod   4/4     4            4           6m59s
```

Dns entre namespaces

El dns en un namespace se crea automáticamente y los pods de un mismo espacio de trabajo se ven entre ellos, pero entre espacios de trabajo solo se podran ver a nivel de servicios y especificando el FQDN de la siguiente manera.

`service-name + namespace-name + svc.cluster.local`

Ejemplo de dns entre namespaces.

```
sudo kubectl get services -n dev
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
my-service   ClusterIP   10.98.97.69   <none>        8080/TCP   15m

sudo kubectl run -ti --generator=run-pod/v1 podtest --image=nginx:alpine --
namespace default -- sh
# no esta en el mismo namespace y no lo ve
/ # curl my-service:8080
curl: (6) Could not resolve host: my-service

# pero con el FQDN llegan a verse.
/ # curl my-service.dev.svc.cluster.local:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

Contexto

El contexto en kubernetes es el espacio de trabajo en el que trabaja un usuario.

Un usuario por defecto trabaja en el namespace default y si es un desarrollador y tiene su namespace desarrollo, tiene que estar indicando siempre `-n desarrollo` o `--namespace desarrollo` y esto es engorroso.

Para esto mismo existen los contextos, para asignarte un namespace por defecto en un usuario.

```
# contexto actual
sudo kubectl config current-context
minikube

# añadier nuevo contexto context-dev con namespace por defecto dev
sudo kubectl config set-context context-dev --namespace=dev \
  --cluster=minikube \
  --user=minikube

# usar contexto
sudo kubectl config use-context context-dev
Switched to context "context-dev".

# verificar
sudo kubectl config current-context
context-dev

# ahora por defecto muestra los objetos del namespace dev
sudo kubectl get svc
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
my-service   ClusterIP   10.98.97.69   <none>        8080/TCP   35m
```

Limitaciones

Las limitaciones de recursos se especifican como `limis` y `request` , estas se indican en ram como `Mi, Gi` referiendose a Mebibyte, y a cpu por unidades o Milicors, que 1 core es 1000 milicors, por lo tanto se especifican porcentajes como `0.1` o `100m` en un 10% de un core.

requests: Cantidad de recursos de las que el pod siempre va a disponer, se le reserva esa cantidad de ram o cpu a ese pod, es suya.

limits: Es la cantidad de ram o cpu hasta la que podrá llegar el pod en el caso de que el nodo lo permita, es decir una vez alcanzada la cantidad total de request tiene un margen para superar siempre que el nodo tenga recursos suficientes. Una vez se alcance el limite, kubernetes reinicia o elimina el pod, según se especifique.

Por defecto si un pod excede el limite de ram kubernetes lo reinicia, por otra parte si intentas crear un pod con el request y limit superior al de un nodo, kubernetes lo deja pendiente hasta encontrar un nodo en el que cumpla los requisitos.

En el caso de la cpu: cuando llega el pod al máximo del limite no pasa nada, kubernetes no reinicia el pod ni lo destruye, simplemente no le deja superar ese limite. Si al crear un pod su request y limit es superior al del nodo se queda en pendiente hasta haber un nodo que cumpla los requisitos.

Ejemplo de limites en un pod a nivel de container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "500Mi"
        cpu: "1200m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

Qos Classes

Kubernetes usa clases de QoS para tomar decisiones sobre la programación y el desalojo de Pods.

Guaranteed: es el pod en el que el limite es igual al request.

Burstable: es el pod en el que el limite es superior al request, por lo tanto puede aumentar sus capacidades si están disponibles en el nodo.

BestEffort: es el pod en el que no se han establecido ningún tipo de limites, este tipo de pod puede ser peligroso ya que puede llegar a utilizar el total de recursos del nodo..

LimitRange

Permite controlar limites a nivel de objetos , los `Containes, Pod, PersistentVolumeClaim` dentro de un namespace estarán limitados a los limites definidos.

Limites por defecto

Al poner un valor de limites por defecto, los objetos de un espacio de trabajo, en el caso de crearlos sin limites se aplican el de defecto.

En el siguiente manifiesto, se especifica un valor por defecto a los contenedores del namespace dev.

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
  labels:
    name: dev
---
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
  namespace: dev
spec:
  limits:
  - default:
      memory: 512Mi
      cpu: 1
    defaultRequest:
      memory: 256Mi
      cpu: 0.5
    type: Container
```

```
sudo kubectl describe limitrange -n dev
Name:       mem-limit-range
Namespace:  dev
Type        Resource  Min  Max  Default Request  Default Limit  Max
Limit/Request Ratio
----        --------  ---  ---  ---------------  -------------  ----------------
-------
Container   cpu       -    -    500m             1              -
Container   memory    -    -    256Mi            512Mi          -
```

Cuando creas un pod dentro del namespace dev sin limites definidos, se añaden los de defecto.

```
sudo kubectl run   --generator=run-pod/v1 podtest --image=nginx:alpine --
namespace dev
sudo kubectl describe pod podtest -n dev
    Limits:
      cpu:     1
      memory:  512Mi
    Requests:
      cpu:        500m
      memory:     256Mi
```

Mínimos y Máximos

Los valores mínimos y máximos en limitrange son los recursos mínimos y máximos que puede especificarse a un objeto, en caso que exceda el máximo o no llegue al mínimo, al crear el objeto dará un error y no lo creara, en el caso de no especificar limites en el objeto se añadirán los máximos permitidos.

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    name: prod
---
apiVersion: v1
kind: LimitRange
metadata:
  name: min-max
  namespace: prod
spec:
  limits:
  - max:
      memory: 1Gi
      cpu: 1
    min:
      memory: 100Mi
      cpu: 0.1
    type: Container
---
apiVersion: v1
kind: Pod
metadata:
  name: podtest
  namespace: prod
  labels:
    app: backend
spec:
  containers:
  - name: contenedor1
    image: nginx:alpine
    resources:
      limits:
        memory: "500Mi"
        cpu: 1
      requests:
```

```
        memory: "300Mi"
        cpu: 0.5
```

```
sudo kubectl describe limitrange min-max -n prod
Name:       min-max
Namespace:  prod
Type        Resource  Min    Max  Default Request  Default Limit  Max
Limit/Request Ratio
----        --------  ---    ---  ---------------  -------------  -------------
---------
Container   cpu       100m   1    1                1              -
Container   memory    100Mi  1Gi  1Gi              1Gi            -
```

En el caso de exceder los limites dará un error similar a este.

```
sudo kubectl apply -f limit-range/limits-min-max.yml
namespace/prod unchanged
limitrange/min-max configured
Error from server (Forbidden): error when creating "limit-range/limits-min-
max.yml": pods "podtest" is forbidden: maximum cpu usage per Container is 1, but
limit is 2
```

ResourceQuota

Limita recursos a nivel de namespace, establece un limite de recursos que pueden consumir la sumatoria de todos los objetos dentro de el namespace indicado.

Una vez establecido el resourcequota, al crear contenedores tienen que tener obligatoriamente el request y el limit, en caso contrario no dejará crearlos.

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    name: prod
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
  namespace: prod
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Se puede ver que el namespace tiene unos limites y aún no tiene nada ocupado.

```
sudo kubectl describe ns prod
...
Resource Quotas
 Name:            mem-cpu-demo
 Resource         Used  Hard
 --------         ---   ---
 limits.cpu       0     2
 limits.memory    0     2Gi
 requests.cpu     0     1
 requests.memory  0     1Gi
```

En este caso ya tiene objetos en su interior y a llegado al limite en request.

```
sudo kubectl describe resourceQuota -n prod
Name:            mem-cpu-demo
Namespace:       prod
Resource         Used    Hard
--------         ----    ----
limits.cpu       1       2
limits.memory    1000Mi  2Gi
requests.cpu     1       1
requests.memory  1000Mi  1Gi
```

Limitar pods

limitar el número de pods en un espacio de trabajo es otra opción que tiene resourceQuota.

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    name: prod
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
  namespace: prod
spec:
  hard:
    pods: "3"
```

Probes

Probe es una test que ejecuta kubelet cada cierto tiempo sobre un contenedor para comprobar su estado, en caso de no dar lo acordado ejecuta una acción configurada.

Las tres formas de ejecutar test que tiene kubelet son:

- Mediante un **comando**, si retorna 0 todo correcto.
- Por **puerto**, verifica que cierto puerto este abierto.

- Desde **HTTP**, si la respuesta esta entre `200-399` todo correcto, si es superior a 400 error.

Tipos

Los tipos de pruebas que tiene kubelet para asegurar el funcionamiento de una aplicación son:

`liveness` Comprueba que esta funcionando el contenedor como debe cada x tiempo, si la prueba falla, liveness reinicia el contenedor.

`readines` hace un test cada x tiempo par ver que el contenedor funciona correctamente, si falla la prueba, readiness desregistra la ip del contenedor del endpoint para no recibir mas carga, hasta que el contenedor este en buen estado.

`startup` se utiliza para aplicaciones que tardan bastante en configurarse y arrancar. Mientras una aplicación está en modo startup, liveness y readiness permanecen desactivadas hasta que startup la da por buena.

> **Nota:** `readiness` y `liveness` es bueno que trabajen juntos, ya que si se envía una petición a un pod mientras este se esta reiniciando, la petición fallara.

liveness

cmd

Ejemplo **comando** tipo liveness, en este yaml el contenedor ejecuta un comando de crear un archivo y borrarlo en 30 segundos para provocar un fallo.

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 10
      periodSeconds: 5
```

El livenessProbe una vez arrancado el contenedor espera 10 segundos, y ejecuta un comando cada 5 segundos para verificar que el archivo `/tmp/healthy`existe, en caso contrario reinicia el contenedor.

```
➜ sudo kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
liveness-exec   1/1     Running   3          3m52s

➜ sudo kubectl describe pod liveness-exec
...
  Normal   Started    62s                kubelet, pc02      Started container
liveness
  Warning  Unhealthy  20s (x3 over 30s)  kubelet, pc02      Liveness probe
failed: cat: can't open '/tmp/healthy': No such file or directory
  Normal   Killing    20s                kubelet, pc02      Container liveness
failed liveness probe, will be restarted
```

tcp

Este es un ejemplo de probe en readness y liveness que comprueban que el puerto 8080 del contenedor este abierto.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

http

En este ejemplo liveness hace una consulta a una ruta de un servidor web y según el estado de la respuesta lo da por bueno o no.

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
```

```
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Cualquier código mayor o igual a 200 y menor a 400 indica éxito. Cualquier otro código indica falla.

Durante los primeros 10 segundos que el contenedor está vivo, el controlador / healthz devuelve un estado de 200. Después de eso, el controlador devuelve un estado de 500.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

```
➜ sudo kubectl get pod liveness-http
NAME            READY   STATUS    RESTARTS   AGE
liveness-http   1/1     Running   1          28s

➜ sudo kubectl describe pod liveness-http
...
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Warning  Unhealthy  11s (x3 over 17s)  kubelet, pc02      Liveness probe
failed: HTTP probe failed with statuscode: 500
  Normal   Killing    11s                kubelet, pc02      Container liveness
failed liveness probe, will be restarted
  Normal   Pulled     10s (x2 over 29s)  kubelet, pc02      Successfully pulled
image "k8s.gcr.io/liveness"
  Normal   Created    10s (x2 over 29s)  kubelet, pc02      Created container
liveness
  Normal   Started    10s (x2 over 29s)  kubelet, pc02      Started container
liveness
```

ConfigMaps y Variables Entorno

ConfigMaps es un objeto que permite guardar una configuración especifica para un tipo de containers y así cambiar la configuración de un container sin tener que rehacer el Dockerfile ni la imagen, esto lo hace mediante puntos de montaje o variables de entorno.

Esto puede servir para terner diferentes configuraciones de un mismo contenedor y utilizar la mas adecuada para el momento deseado.

Variables entorno

A un pod se le pueden asignar variables de entorno directamente desde el manifiesto indicándolas dentro de la sección contnedor variable `env`.

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envars
spec:
  containers:
  - name: envar-demo-container
    image: nginx:alpine
    env:
    - name: DEMO_GREETING
      value: "Hello from the environment"
    - name: DEMO_FAREWELL
      value: "Such a sweet sorrow"
```

Dentro del contenedor se asignan correctamente

```
➜ sudo kubectl get pods
NAME         READY   STATUS    RESTARTS   AGE
envar-demo   1/1     Running   0          14s

➜ sudo kubectl exec -it envar-demo -- sh
/ # env | grep DEMO
DEMO_FAREWELL=Such a sweet sorrow
DEMO_GREETING=Hello from the environment
```

valores externos

Cada pod tiene una serie de valores perteneciente a el, pero que interiormente no están definidos, es decir desde dentro del contenedor no se ven, esto puede ser metadata, specificaciones, etc... Estos valores se pueden ver con el siguiente comando:

```
➜  sudo kubectl get pod envar-demo -o yaml
```

En ocasiones es posible que se necesiten en el interior y se pueden definir como valores de entorno.

- `name` nombre de la variable dentro del container
- `fieldPath` es la ruta que sigue descrita en el comando anterior en formato yaml.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: nginx:alpine
      env:
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
```

ConfigMap

Para utilizar ConfigMap primero se tiene que crear el objeto, para después asignarlo a los containers que queramos.

Comand-line

En est ejemplo se esta asignando solo un archivo en el configmap, en el caso de querer asignar un directorio es lo mismo pero indicando el directorio.

```
# creo l objeto con la configuración de nginx
➜ sudo kubectl create configmap nginx-config --from-file=configmap/cf-
examples/nginx.conf
configmap/nginx-config created

# visualizo
➜ sudo kubectl get cm
NAME           DATA   AGE
nginx-config   1      12s

➜ sudo kubectl describe cm nginx-config
Name:         nginx-config
Namespace:    default
Labels:       <none>
Annotations:  <none>
```

```
Data
====
nginx.conf:
----
server {
    listen       8080;
    server_name  localhost;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }

}

Events:  <none>
```

ahora solo quedaría asignarlo a un contenedor, pero como en kubernetes es mejor trabajar desde manifiestos, lo haré en la siguiente sección

template

Ejemplo de manifiesto de ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  labels:
    app: front
data:
  test: hola
  nginx: |
    server {
        listen       8080;
        server_name  localhost;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
```

Para poder utilizar un configMap dentro de un contenedor se tiene que montar como volumen, por lo tanto se necesita especificar el nuevo volumen y después montarlo en el contenedor.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
  labels:
    app: front-nginx
  spec:
    containers:
    - name: nginx
      image: nginx:alpine
      volumeMounts:
        - name: nginx-vol
          mountPath: /etc/nginx/conf.d
   volumes:
     - name: nginx-vol
       configMap:
         name: nginx-config
         items:
         - key: nginx
           path: default.conf
```

Especificar nuevo volumen

dentro de `volumes` indicas el volumen configmap a montar, la sección `key` es la sección dentro de configmap y la `path` el nombre del archivo final dentro del contenedor.

Asignar volumen

`volumeMounts` esta indicando que volumen montar y donde.

Environtment

kubernetes también permite trabajar con variables de entorno dentro de los objetos configmap e incluso, combinar variables con archivos.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vars
  labels:
    app: front
data:
  db_host: dev.host.local
  db_user: dev_user
  script: |
    echo DB host es $DB_HOST y DB user es $DB_USER >
/usr/share/nginx/html/test.html
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-test
```

```
  labels:
    app: front-nginx
  spec:
    containers:
    - name: nginx
      image: nginx:alpine
      env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: vars
              key: db_host
        - name: DB_USER
          valueFrom:
            configMapKeyRef:
              name: vars
              key: db_user
      volumeMounts:
         - name: script-vol
           mountPath: /opt
    volumes:
      - name: script-vol
        configMap:
          name: vars
          items:
          - key: script
            path: script.sh
```

Comprobaciones de que todo a salido bien.

```
# aplico el yaml y creo el config map y el pod
➜ sudo kubectl apply -f configmap/cm-nginx-env.yml
configmap/vars created
pod.apps/pod-test created

➜ sudo kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
pod-test   1/1     Running   0          37s

# entro dentro del contenedor para ver que todo esta bien
➜ sudo kubectl exec -it pod-test -- sh

# compruebo que las variables de entorno se han asignado y el archivo script
existe.
/ # env
...
DB_HOST=dev.host.local
DB_USER=dev_user

/ # ls /opt/
script.sh
/ # sh /opt/script.sh

# despues de ejcutar el script desde un navegador compruevo, que todo es
correcto.
```

```
➜ curl 172.17.0.5/test.html
DB host es dev.host.local y DB user es dev_user
```

Secrets

Es un objeto muy parecido a ConfigMap, con la diferencia que secrets se utiliza para pasar datos sensibles, como tockens, usuarios, contraseñás, etc... La forma de pasar los datos es la misma que con configMap a trabes de volumenes o variables de entorno.

Secrets no encripta solo codifica los datos pasados a base64.

Consola

He creado un archivo que contiene datos sensibles y voy a crear un secret con ellos dentro.

```
sudo kubectl create secret generic mysecret --from-file=./secrets/secrets-
files/secret.txt
secret/mysecret created

➜  sudo kubectl get secret mysecret
NAME       TYPE     DATA   AGE
mysecret   Opaque   1      61s
```

Como se puede ver a continuación el secret se crea con el contenido codificado a base64, que no es nada seguro, pero es lo que ofrece kubernetes.

```
➜  sudo kubectl describe secret mysecret
Name:         mysecret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
secret.txt:  29 bytes

➜ sudo kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  secret.txt: c2VjcmV0bzE9aG9sYQpzZWNyZXRvMj1hZGlvcwo=
kind: Secret
metadata:
  creationTimestamp: "2020-04-10T08:23:30Z"
  name: mysecret
  namespace: default
  resourceVersion: "235029"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: d28711a2-8954-44f3-a50b-0df3282762e4
type: Opaque
```

```
# se puede decodificar  revirtiendo el base64 y ver l contenido del archivo.
➜ echo c2VjcmV0bzE9aG9sYQpzZWNyZXRvMj1hZGlvcwo= | base64 --decode
secreto1=hola
secreto2=adios
```

Template

desde un template yaml se puede pasar los datos de diferentes modos con:

- `data` : en esta opción se pasaran en base64 directamente
- `stringData`: con esta opción se pasan los datos literales para luego kubernetes haga el codificado a base64.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret2
type: Opaque
stringData:
    username: admin
    password: "12345"
---
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

En los dos templates anteriores el resultado es el mismo.

```
➜ sudo kubectl apply -f secrets/secret-stringdata.yml
secret/mysecret2 created

➜ sudo kubectl get secret mysecret2 -o yaml
[sudo] password for debian:
apiVersion: v1
data:
  password: MTIzNDU=
  username: YWRtaW4=
kind: Secret
...
```

Seguro

Como podemos ver esta opción que proporciona kubernetes muy segura no es, pero se puede jugar con ella para mantener versiones en el git, utilizando variables de entorno.

Suponemos que tenemos un git y no queremos exponer las contraseñas, puedes asignar variables de entorno a la versión de git, para después remplazarlas con `envsubst, sed` u otro, para finalmente desplegar el secret.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret3
type: Opaque
stringData:
  username: $USER
  password: $PASSWORD
```

remplazar variables y desplegar secret.

```
# creo variables
➜  export PASSWORD=contrasena
➜  export USER=jorge

# remplazo variables
➜  envsubst < secrets/secure.yml > tmp.yml

# jemplo final
➜  cat tmp.yml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret3
type: Opaque
stringData:
  username: jorge
  password: contrasena

# y desplego con el nuevo yaml
➜ sudo kubectl apply -f tmp.yml
secret/mysecret3 created

➜ sudo kubectl get secret mysecret3 -o yaml
apiVersion: v1
data:
  password: Y29udHJhc2VuYQ==
  username: am9yZ2U=
kind: Secret
```

Utilizarlos

La manera de poder utilizar los secrets en un contenedor o pod es mediante volumenes o variables de entorno o mediante ambos.

Volumen

En el siguiente yaml estoy incorporando tanto el secret como el pod, no es necesario que estén juntos solo es por visualización.

se muestra como a `volumes` se le asigna el valor del secret `secretName` mysecret y este sera montado en la ruta `/opt/` con permisos read only.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  username: admin
  password: "12345"
---
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:alpine
    volumeMounts:
    - name: test
      mountPath: "/opt"
      readOnly: true
  volumes:
  - name: test
    secret:
      secretName: mysecret
```

una vez desplegado se ve como efectivamente en `/opt/` se han montado dos archivos con los valores del secret.

```
➜ sudo kubectl apply -f secrets/pod-vol-secret.yml
secret/mysecret created
pod/mypod created

➜ sudo kubectl exec -it mypod -- sh
/ # ls /opt/
password  username
```

items

En los volumenes, si añadimos el valor de `items` se puede especificar que nombre tendrá el archivo una vez dentro del contenedor.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:alpine
    volumeMounts:
    - name: test
      mountPath: "/opt"
      readOnly: true
  volumes:
  - name: test
    secret:
      secretName: mysecret
      items:
      - key: username
        path: user.txt
      - key: password
        path: pass.txt
```

```
➜ sudo kubectl apply -f secrets/pod-vol-secret.yml
secret/mysecret configured
pod/mypod created

➜ sudo kubectl exec -it mypod -- sh
/ # ls /opt/
pass.txt  user.txt
```

Variables entorno

otra manera de asignar los datos de secret dentro de un pod son con las variables de entorno, donde se asigna cada valor del secret a una variable de entorno. No es necesario asignar todos los valores, si se quiere se puede asignar solo uno o mas.

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  username: admin
  password: "12345"
---
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:alpine
    env:
      - name: SECRET_USER
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
```

```
➜ sudo kubectl apply -f secrets/pod-env-secret.yml
secret/mysecret configured
pod/mypod created

➜ sudo kubectl exec -it mypod -- sh
/ # echo $SECRET_USER
admin
/ # echo $SECRET_PASSWORD
12345
/ #
```

Volumenes

Los volúmenes son puntos de montaje que se utilizan para compartir datos entre contenedores o almacenaje persistente de datos.

Tipos de Volúmenes

emptydir: es un directorio vacio que se crea como volumen y va ligado a la vida del pod, mientras el pod viva ese volumen seguirá existiendo y manteniendo los datos, los containers dentro del pod pueden morir y revivir las veces que quieran que el volumen seguirá existiendo, pero si muere el pod el volumen desaparece.

hostPath: es un directorio que en forma de volumen y va ligado a la vida del nodo, ese directorio existirá en el nodo, si un pod muere y se despliega en otro nodo no podrá acceder a la información del volumen donde murió, este volumen existirá hasta que se elimine o el nodo muera.

Volumenes cloud: kubernetes permite crear puntos de montaje en diferentes plataformas como aws, openstack, google cloud, donde previamente a de haber una configuración en el nodo para conectar con la plataforma.

PV/PvC: mas que un tipo de montaje es una metodología que utiliza kubernetes para separar el trabajo del desarrollador al de administrador.

Funciona de la siguiente manera, un pod solicita el punto de montaje al objeto PvC (Persistent Volume claim) en este objeto se especifica la capacidad que se quiere, el PvC pide ese espacio al objeto PV (Persistent volume) que este sabe donde esta cada disco externo al nodo y su capacidad, si lo solicitado por el PvC esta disponible en algún PV el punto de montaje se crea.

De esta manera los desarrolladores no tienen que preocuparse d donde almacenar los datos, solo solicitan el espacio deseado, y los administradores ponen espacio disponible a disposición de recogida.

Diferentes tipos de funcionamientos:

- **retain**: en caso de eliminar el pvc, retener el pv y los datos de disco, ademas no podrá ser usado por otro pvc
- **recycle**: en caso de eliminar el pvc el pv se mantiene pero los datos del disco se borran para que otro pvc lo reclame.
- **delete**: en este caso si eliminas el pvc también eliminas el pv y los datos del disco.

Emptydir

Este tipo de montaje al ser muy volátil es utilizado normalmente para guardar cache o archivos que pueden perderse. Una vez muera el POD los datos del punto de montaje se perderán.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: nginx:alpine
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

Ruta del directorio local

```
/var/lib/kubelet/pods/b60897a7-f689-4bff-bef6-
0545d1727f9b/volumes/kubernetes.io~empty-dir/cache-volume
```

HostPath

hostPath es un punto de montaje a nivel de nodo, en el nodo persistirán los datos pero si el pod por la razón que sea cambia de nodo, no podrá llegar a los datos a del anterior nodo.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: nginx.alpine
    name: test-container
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data/front-nginx
```

Volumen AWS ebs

Este es un ejemplo de punto de montaje con un volumen creado en amazon ebs, se indica el id del volumen a montar. La ventaja de estos puntos de montaje, son que los datos están disponibles en todos los nodos y no hay perdidas ni conflictos, la desventaja que se utiliza una entidad externa.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: vol-0fb30c332ecff512d
      fsType: ext4
```

> **Nota**: Este proceso de montaje requiere una configuración previa, tanto de amazon ebs con usuarios especiales IAM, como de cada nodo establecer la conexión con amazon correctamente con `aws-clie`

PV/PvC

Esta metodología qu a creado kubernetes, en un principio se hace pesada de utilizar y un tanto engorrosa, pero una vez te acostumbras, acaba siendo muy útil y rápida de utilizar, sobre todo en su opción dinámica.

PV y PvC son dos objetos diferentes donde PvC solicita un espacio determinado y PV lo proporciona si esta disponible.

En el siguiente ejemplo se ve un PV con 5Gi de espacio disponible, permisos sobre el espacio de `ReadWriteOnce` escritura y lectura solo por un PvC y que ese espacio está en `/mnt/data`

El PvC que se observa solicita a cualquier PV que exista: 5Gi de espacio `readWriteOnce` y que en ese espacio solo pueda escribir el.

```yaml
# pv
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
# pvc
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Una vez desplegado se observa que el PvC se a asignado al PV que había.

```
➜ sudo kubectl get pv
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
   STORAGECLASS   REASON   AGE
pv-volume   5Gi        RWO            Retain           Bound    default/pv-claim
  manual

➜ sudo kubectl get pvc
NAME       STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pv-claim   Bound    pv-volume   5Gi        RWO            manual         18s
```

Selectors

Los selectores es una manera de especificar a que PV quieres que se asigne un PvC. En el siguiente ejemplo se muestra un PvC que se asigna a un PV que tenga el label `type:local`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      type: local
```

Se puede observar que teniendo dos PV disponibles se a asociado al que coincide con el label indicado.

```
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
           STORAGECLASS   REASON   AGE
pv-volume        5Gi        RWO            Retain           Bound
default/pv-claim
pv-volume-mysql  5Gi        RWO            Retain           Available
```

Asignar pvc a pod

Teniendo el siguiente PvC corriendo, asigno un POD a este para mostrar un ejemplo de funcionamiento.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-claim
```

```
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      mysql: local
```

Para asociar un PvC a un POD solo se tiene que indicar el nombre del PvC en la sección `volumes` con la opción `persistentVolimeClaim y claimName`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-mysql
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:latest
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: "12345678"
        volumeMounts:
        - mountPath: "/var/lib/mysql"
          name: vol-mysql
      volumes:
        - name: vol-mysql
          persistentVolumeClaim:
            claimName: pv-claim
```

Resultado del montaje:

```
# se a creado el POD del deployment
➜ sudo kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
deploy-mysql-6c5cdb988b-7w9cp   1/1     Running   0          26s

# el pvc pc-claim se a asociado al PV pv-volume
➜ sudo kubectl get pvc
NAME       STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pv-claim   Bound    pv-volume   5Gi        RWO            manual         35s
```

```
➜ sudo kubectl get pv
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
   STORAGECLASS   REASON   AGE
pv-volume   5Gi        RWO            Retain           Bound    default/pv-claim
  manual

# Y desde el host podemos ver el directorio que se a montado
➜ ls /mnt/data
 auto.cnf        binlog.000002   ca-key.pem   client-cert.pem   ib_buffer_pool
...
```

StorageClass dinamico

Esta es una opción para facilitar rápidamente a los desarrolladores puntos de montaje, con esta opción solo tienes que crear un PvC y StorageClass te crea un PV con unos valores por defecto.

Por defecto minikube ya tiene creado un storageclass como HostPath, por lo contrario con minikube no puede configurar un StorageClass dinamico para la nube.

```
# sc abreviación de storageClass
➜ sudo kubectl get sc
NAME                 PROVISIONER                RECLAIMPOLICY
VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate
    false
```

Solo creando un PvC storageClass ya nos proporciona el punto de montaje.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sc-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Comprobación de que se a creado el pv automáticamente:

> **Atención**: el PV lo crea en modo delete si borras el PvC se borraran tanto los datos como el PV, esto solo es un entorno de pruebas, se puede cambiar utilizando `sudo kubectl edit storageClass standard`

```
# despliego el PvC
➜ sudo kubectl apply -f volumenes/storage-class.yml
persistentvolumeclaim/sc-pvc created

# ompruevo que se a asignado a un PV
➜ sudo kubectl get pvc
```

```
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS   AGE
sc-pvc   Bound    pvc-a1ac3b34-d19c-4bd8-8fd5-26cc4ae1123b   5Gi        RWO
       standard       16s

➜ sudo kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM
POLICY   STATUS   CLAIM            STORAGECLASS   REASON   AGE
pvc-a1ac3b34-d19c-4bd8-8fd5-26cc4ae1123b   5Gi        RWO            Delete
      Bound    default/sc-pvc   standard               24s

# Observo que el pv crea un punto de montaje en tmp.
➜ sudo kubectl describe pv pvc-a1ac3b34-d19c-4bd8-8fd5-26cc4ae1123b
Name:            pvc-a1ac3b34-d19c-4bd8-8fd5-26cc4ae1123b
StorageClass:    standard
Status:          Bound
Claim:           default/sc-pvc
Reclaim Policy:  Delete
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        5Gi
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/hostpath-provisioner/pvc-a1ac3b34-d19c-4bd8-8fd5-
26cc4ae1123b
```

NFS

nfs junto con kubernetes es una manera de tener puntos de montaje en el cluster sin utilizar una plataforma externa.

El funcionamiento de esto es Montar un servidor NFS en el que todos los nodos del cluster tengan acceso y desde los nodos utilizar PV/PvC.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
  labels:
    datos: nfs
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: my-nfs-server
    path: "/var/shared"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
```

```
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: manual
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      datos: nfs
```

Dinámico

El aprovisionamiento dinámico es un recurso normalmente en la nube, donde se crean volumenes automáticamente al solicitarlos.

Los puntos de aprovisionamiento dinámico están diseñados para dar puntos de montaje rápidamente y sin tener que pedir permiso al administrador. Hasta ahora el diseñador cada vez que quería pedir un punto de montaje con un PVC, el administrador tenia que crear un PV manualmente, con el aprovisionamiento dinámico esto se automatiza.

Nfs

https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client/deploy

Por defecto kubernetes deja esta metodología para terceros y proporciona el siguiente git para configurar un servidor nfs como aprovisionamiento dinámico.

> Previamente se tiene que disponer de un servidor nfs configurado

Los archivos del repositorio proporcionan un deployment para poder gestionar las solicitudes de storage en el namespace en el que se despliege, ademas un archivo `rbac.yaml` con todos los permisos necesarios para hacer posible esta gestión.

Configuración del nfs server

```
pi@raspberrypi:~ $ cat /etc/exports
/srv/dinamic
192.168.88.0/24(rw,sync,no_subtree_check,no_root_squash,no_all_squash,insecure)

pi@raspberrypi:~ $ ls -l /srv/
total 20
drwxr-xr-x  2 nobody   root            4096 May  6 19:14 dinamic
```

Despliegue requerido

Despliegue de los archivos requeridos para el funcionamiento de nfs dinámico proporcionados por el git de esta sección.

```
➜ kubectl apply -f rbac.yaml
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
```

```
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner
created

➜ kubectl get clusterrole,role,rolebinding,clusterrolebinding,svc | grep nfs
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner
                            10m
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner   10m
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner
10m
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner
                    10m

➜ kubectl apply -f deployment.yaml
deployment.apps/nfs-client-provisioner created
```

StorageClass

StorageClass es el objeto que kubernetes proporciona para el almacenamiento dinámico

class.yml

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: managed-nfs-storage
provisioner: my-nfs-provisioner # or choose another name, must match
deployment's env PROVISIONER_NAME'
parameters:
  archiveOnDelete: "false"
reclaimPolicy: Delete # Delete | Retain | recycle
```

Aplicar StorageClass y visualizar

```
➜ kubectl apply -f class.yaml
storageclass.storage.k8s.io/managed-nfs-storage created

➜ kubectl get storageclass,deploy
NAME                                            PROVISIONER
RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
storageclass.storage.k8s.io/managed-nfs-storage   my-nfs-provisioner
Delete          Immediate           false                  57s
storageclass.storage.k8s.io/standard (default)    k8s.io/minikube-hostpath
Delete          Immediate           false                  81d

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nfs-client-provisioner   1/1     1            1           49s
```

Comprobar

Para comprobar creamos un PVC y automáticamente se tiene que crear un PV asociado y un directorio en el servidor nfs.

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
  annotations:
    volume.beta.kubernetes.io/storage-class: "managed-nfs-storage"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Mi
```

Despliegue:

```
# no hay pv, pvc exisentes
➜ kubectl get pv,pvc
No resources found in default namespace.

# crear PVC
➜ kubectl apply -f test-claim.yml

# automaticamente se a creado un PV asociado al PVC
➜ kubectl get pv,pvc
NAME                                                        CAPACITY   ACCESS
MODES   RECLAIM POLICY   STATUS   CLAIM                STORAGECLASS
REASON   AGE
persistentvolume/pvc-1355ca1f-7e3a-4b67-a0aa-b9bdc229baae   10Mi       RWX
     Delete           Bound    default/test-claim   managed-nfs-storage
   5s

NAME                                 STATUS   VOLUME
      CAPACITY   ACCESS MODES   STORAGECLASS          AGE
persistentvolumeclaim/test-claim   Bound    pvc-1355ca1f-7e3a-4b67-a0aa-
b9bdc229baae   10Mi       RWX            managed-nfs-storage   5s

# En el servidor nfs se a creado el recurso automaticamente
pi@raspberrypi:~ $ ls -l /srv/dinamic/
total 4
drwxrwxrwx 2 root root 4096 May  6 19:12 default-test-claim-pvc-1355ca1f-7e3a-
4b67-a0aa-b9bdc229baae
pi@raspberrypi:~ $
```

RBAC

Control de Acceso Basado en Rol es una forma de controlar que puede o no puede hacer cierto usuario en el cluster, esto se controla especificando diferentes roles con sus reglas para después asignar usuarios a ese rol. De esta manera se evita que todos los usuarios tengan un control total del cluster.

Por defecto RBAC en minikube viene habilitado, aún que es bueno comprobarlo por si acaso.

```
# comprobar que esta habilitado
➜ kubectl cluster-info dump | grep -i authorization-mode
                            "--authorization-mode=Node,RBAC",

# habilitar
➜ minikube start --vm-driver=none --extra-config=apiserver.authorization-
mode=RBAC
```

Permisos

Kubernetes diferencia dos tipos de permisos, `Role y ClusterRole` donde los dos son muy similares, son una lista de `resources` y `verbs` donde *resources* especifica a que tipo de objeto tienes permiso acceder `pods, deployments ...` y *verbs* se refiere a acciones que puedes hacer `listar, crear, eliminar ...` la diferencia es que `Role` se refiere a los permisos que tienes dentro de un namespace y `ClusterRole` son los permisos que tiene dentro del cluster.

Binding

El binding en kubernetes es la asociación de un usuario, grupo o serviceAccound a un Role o ClusterRole. Binding es un objeto con dos denominaciones diferentes `RoleBinding` y `ClusterRoleBinding` según se utilice para asociar `Roles` o `ClusterRoles`.

Usuarios

kubernetes no tiene ningun objeto donde puedas crear usuarios, de esta manera todos los usuarios seran externos a la aplicación de kubernetes

kubernetes solo aporta el método de autenticación y soporta diferentes métodos `x509 clients certs, statik token file, bootstrap tokens, static Password File, service Account Tokens`.

Un nuevo usuario necesita instalar y configurar `kubectl` para que la api le deje autentificar en alguno de los métodos anteriormente nombrados.

x509 clients certs

Uso de certificados es el método mas utilizado y el que por defecto utiliza minikube, en los siguientes pasos se muestra como añadir un usuario nuevo.

1. El **nuevo cliente** genera su llave y el certificado csr para que lo firme el administrador del cluster. Es importante que `CN/=jorge` se refiere al usuario y `/O=dev` es el grupo del usuario.

```
➜ openssl genrsa  -out jorgekey.pem 2048
➜ openssl req -new -key jorgekey.pem -out jorgecsr.pem -subj "/CN=jorge/O=dev"
```

2. El administrador confirma el `csr` del nuevo cliente/usuario y lo firma con la CA del cluster

```
# identificar donde esta la CA del cluster
➜ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/debian/.minikube/ca.crt
    server: https://192.168.88.2:8443

# firmar el csr del cliente y generar el certificado
➜ openssl x509 -req -in jorgecsr.pem -CAkey /home/debian/.minikube/ca.key -CA
/home/debian/.minikube/ca.crt -days 365 -CAcreateserial   -out jorgecrt.pem


Signature ok
subject=CN = jorge, O = dev
Getting CA Private Key
```

3. En este paso el administrador del cluster nos a proporcionado dos archivos `jorgecrt.pem`, `ca.crt` el nuevo certificado y la clave pública del cluster. Ahora el cliente tiene que configurar su kubectl para poder acceder al cluster

```
# añadir cluster a la configuración de kubectl
jorge@pc02:~$ kubectl config set-cluster minikube --
server=https://192.168.88.2:8443 --certificate-authority=ca.crt

# añadir credenciales del usuario
jorge@pc02:~$ kubectl config set-credentials jorge --client-
certificate=jorgecrt.pem --client-key=jorgekey.pem

# crear un espacio de trabajo del usuario
jorge@pc02:~$ kubectl config set-context jorge --cluster=minikube --user=jorge

# utilizar ese espacio
jorge@pc02:~$ kubectl config use-context jorge
```

Comprobar que todo se a aplicado correctamente:

```
jorge@pc02:~$ kubectl config view
apiVersion: v1
clusters:
```

```
- cluster:
    certificate-authority: /home/jorge/ca.crt
    server: https://192.168.88.2:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: jorge
  name: jorge
current-context: jorge
kind: Config
preferences: {}
users:
- name: jorge
  user:
    client-certificate: /home/jorge/jorgecrt.pem
    client-key: /home/jorge/jorgekey.pem
```

> **Nota:** Una vez configurado `kubectl`, si el administrador no nos asigna ningún tipo de Role no tendremos permisos para hacer nada. Una vez asignado un Role podremos hacer lo que ese Role nos permita.

Role

En los Roles hay que tener en cuenta el `namespace` en el que se asignan los permisos, los `resources` (objetos) a los que se podrá acceder y los `verbs` permisos que se tendrá sobre los objetos.

verbs

Los verbs indica que puedes hacer sobre algún objeto, esta es la tabla de los diferentes verbs que existen.

HTTP verb	request verb
POST	create
GET, HEAD	get (for individual resources), list (for collections, including full object content), watch (for watching an individual resource or collection of resources)
PUT	update
PATCH	patch
DELETE	delete (for individual resources), deletecollection (for collections)

Resources

Los resources son los diferentes objetos que encontramos en kubernetes, `Pods, Replicasets, Deployments, ...`

```
➜  ~ kubectl api-resources
NAME                              SHORTNAMES   APIGROUP
pods                              po
secrets
services                          svc
deployments                       deploy       apps
replicasets                       rs           apps
....
```

Template de **ejemplo de Role** en el namespace default donde puede ver, listar y observar (describir) Pods. El apartado `apiGroups` se tiene que indicar solo cuando el objeto pertenece a un APIGROUP, como podemos ver en el bloque de arriba que deployments pertenece al apigroup apps.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Una vez desplegado el rol se verá de la siguiente forma. Para asignar este rol a un usuario o grupo se tiene que utilizar `Rolebinding`

```
➜ kubectl get roles
NAME         AGE
pod-reader   13s

➜ kubectl describe  role pod-reader
Name:         pod-reader
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 []              [get watch list]
```

Rolebinding

Rolebinding es el enlace entre roles y Usuarios/Grupos. En este ejemplo muestro como enlazar el usuario `jorge` con el role `pod-reader` .

Las secciones importantes del objeto `RoleBinding` son:

- especificar el namespace donde trabajar
- `subjects` el Usuer o Group que enlazar
- `roleRef` la referencia al rol que pertenecerá

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: users-read-pods
  namespace: default
subjects:
- kind: User
  name: jorge
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Comprobación de que se a asignado los permisos correctamente.

```
➜ kubectl get rolebinding
NAME              AGE
users-read-pods   49s

# miro la descripción completa del enlace
➜ kubectl describe rolebinding users-read-pods
Name:         users-read-pods
Role:
  Kind:  Role
  Name:  pod-reader
Subjects:
  Kind  Name   Namespace
  ----  ----   ---------
  User  jorge
```

Actualmente estoy trabajando como administrador minikube, cambio a jorge para comprobar permisos

```
➜ kubectl config current-context
minikube

# cambiar a usuario jorge
➜ kubectl config use-context jorge
Switched to context "jorge".

# puedo ver pods de namespace default
➜ kubectl get pods
No resources found in default namespace.

# y no puedo verbni hacer nada mas.
➜ kubectl get pods -n dev
```

```
Error from server (Forbidden): pods is forbidden: User "jorge" cannot list
resource "pods" in API group "" in the namespace "dev"

➜ kubectl get svc
Error from server (Forbidden): services is forbidden: User "jorge" cannot list
resource "services" in API group "" in the namespace "default"
```

ClusterRole

ClusterRole es exactamente lo mismo que role pero en vez de dar permisos sobre un namespace, da permisos sobre todo un cluster.

En este ejemplo e juntado el `ClusterRol` con el `ClusterRolebinding` porque es exactamente igual a role, simplemente se cambia el `kind` de referencia al objeto.

El siguiente template otorga al usuario jorge permisos de lectura de Pods y deployments sobre todo el cluster.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-pod-deploy-reader
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "watch", "list"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: users-read-pods-deploy
subjects:
- kind: User
  name: jorge
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-pod-deploy-reader
  apiGroup: rbac.authorization.k8s.io
```

Cluster admin

Por defecto kubernetes ya tiene roles definidos, como se ve en el siguiente bloque, en el caso de querer asignar ese rol ya definido como por ejemplo el `cluster-admin` simplemente se tendría que hacer la asignación al usuario.

```
➜  k8s git:(proceso) ✗ kubectl get clusterroles
NAME                                                                   AGE
admin                                                                  66d
cluster-admin                                                          66d
edit                                                                   66d
kubernetes-dashboard                                                   9d
system:aggregate-to-admin                                              66d
system:aggregate-to-edit                                               66d
...
```

Asignación de role cluster-admin a usuario jorge.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: users-cluster-admin
subjects:
- kind: User
  name: jorge
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Grupos roles

Los grupos de roles son muy útiles en la gestión de permisos, ya que te permite agrupar a un grupo a tener ciertos permisos.

El grupo de un usuario se establece en el método de crear el usuario, utilizando certificados `➜ openssl req -new -key jorgekey.pem -out jorgecsr.pem -subj "/CN=jorge/O=dev"` , en este caso el usuario `jorge` pertenece al grupo `dev`

El siguiente ejemplo de template se ve como se crea un *clusterRoleBinding* de el grupo de usuarios `dev` asignando los permisos del *ClusterRole* con nombre `svc-clusterrole`.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: svc-clusterrole
rules:
- apiGroups: ["apps"]
  resources: ["services","Pods","replicasets","deployments"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-svc
subjects:
- kind: Group
  name: dev
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
```

```
  name: svc-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

ServiceAccount

ServiceAccount proporciona una identidad para los procesos que se ejecutan en un Pod.

Cuando alguien accede a un cluster utilizando por ejemplo `kubectl`, el api server lo autentica como usuario y puede ver que ocurre en el cluster, según los permisos. Los procesos de los contenedores también pueden consultar datos al api server, pero para eso necesitan autenticarse y para eso utilizan serviceAccount.

ServiceAccount utiliza un tocken para la autenticación del pod con la api, este tocken se monta automáticamente en el pod al crearlo, si no se especifica ninguno se monta el de por default, y según los permisos que se otorgan a ese tocken puede acceder a distintas consultas de la api.

Ejemplo de serviceAccount simple.

sa.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-sa
```

Un serviceAccount solo implica crear un tocken, para que este tenga utilidad se tiene que aplicarle un role y asociarlo con un binding a algún pod.

```
# desplegar serviceaccount
➜ kubectl apply -f k8s/RBAC/serviceacount/sa.yaml
serviceaccount/my-sa created

# por defecto kubernetes ya tiene uno
➜ kubectl get sa
NAME      SECRETS   AGE
default   1         67d
my-sa     1         20s

# vemos que se a creado el tocken my-sa-token-7z2ms
➜ kubectl describe sa my-sa
Name:                my-sa
Namespace:           default
...
Mountable secrets:   my-sa-token-7z2ms
Tokens:              my-sa-token-7z2ms

# comprovamos que exise ese tocken
➜ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-k8kw2   kubernetes.io/service-account-token   3      67d
my-sa-token-7z2ms     kubernetes.io/service-account-token   3      57s
```

Asociar SA a Pod

En el siguiente template se muestra como se asigna un rol a un serviceAccount y este se asigna a un Pod

```yaml
# creo el service account
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-sa
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
  name: deployment-test
  labels:
    app: front-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: front-nginx
  template:
    metadata:
      labels:
        app: front-nginx
    spec:
     # assigno el SA my-sa al pod
      serviceAccountName: my-sa
      containers:
      - name: nginx
        image: nginx:alpine
---
# creo el role de permisos que tendra el SA
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: sa-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
# assigno el Role al SA
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sa-read-pods
  namespace: default
 # subjects.kind  especifica el objeto SA, y el name del SA
subjects:
- kind: ServiceAccount
  name: my-sa
  apiGroup: ""
roleRef:
```

```
  kind: Role
  name: sa-reader
  apiGroup: rbac.authorization.k8s.io
```

Verificar la asignación y utilizar el token

```
# vemos el pod desplegado
➜ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deployment-test-79bb7486b-cnx7s   1/1     Running   0          8m16s

# verificamos que se a asignado bien el SA al Pod
➜ kubectl get pod deployment-test-79bb7486b-cnx7s -o yaml
...
  serviceAccount: my-sa
  serviceAccountName: my-sa
...

# entramos en el pod y utilizar el token haciendo una consulta a la api.
➜ kubectl exec -it deployment-test-79bb7486b-cnx7s -- sh
/ # apk add curl
/ # TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
/ # curl -H "Authorization: Bearer ${TOKEN}"
https://kubernetes/api/v1/namespaces/default/pods --insecure
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "430633"
  },
  "items": [
    {
      "metadata": {
        "name": "deployment-test-79bb7486b-cnx7s",
        "generateName": "deployment-test-79bb7486b-",
...
```

Ingres

Ingres ayuda a exponer un servicio de una manera mas sencilla para el cliente, es decir hasta ahora el cliente para acceder a un servicio del cluster, tenia que acceder a un puerto expuesto por `nodeport` que son del `30000-32767` o desde un `Loadbalancer` en aws o google cloud, lo malo de este ultimo que por cada puerto se abre un loadbalancer.

Ingres proporciona un método de con un solo puerto abierto exponer todas aplicaciones del cluster (hace como de proxy). configurando unas reglas tipo ruta o subdominio te redirige a una aplicación u otra.

Ejemplo de tipo reglas:

- ruta: `miapp.org/, miapp.org/nextcloud, miapp.org/ldap`
- subdominio: `miapp.org, nextcloud.miapp.org, ldap.miapp.org`
- También admite ambas a la vez.

Ingres controller es el controlador que proporciona kubernetes para hacer su magia, da dos opciones nginx-controler y balanceadores de google cloud. Este es un deployment en el cluster que se encarga de leer las reglas de `Ingres` y exponerlas en el cluster o en el cloud según definamos.

instación del controlador

Nginx-controler

Si no queremos exponer el cluster a google cloud, aws o alguna plataforma externa nginx-controler es la opción perfecta para gestionar las apps de nuestro cluster desde un solo puerto. (bueno dos puertos http y https)

nginx ya proporciona un yaml para la instalación del controler en nuestro cluster kubernetes, donde simplemente desplegando-lo se genera todo lo necesario.

https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/baremetal/deploy.yaml

De todo lo que despliega, lo que realmente nos interesa es el objeto service `ingress-nxinx-controller`. este es el servicio que expone en nuestro cluster los puertos en modo `nodePort` desde donde conectaremos al interior del cluster.

```yaml
# Source: ingress-nginx/templates/controller-service.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    helm.sh/chart: ingress-nginx-2.0.0
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/version: 0.30.0
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/component: controller
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  type: NodePort
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: http
    - name: https
      port: 443
      protocol: TCP
      targetPort: https
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/component: controller
```

Instalación y comprobación

La instalación genera diferentes objetos que se despliegan en el namespace `ingress-nxinx`.

```
➜ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/master/deploy/static/provider/baremetal/deploy.yaml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-
admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
serviceaccount/ingress-nginx-admission created

# compruevo que nxinx-controles se a puesto en marcha
➜ kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create-zrdlr        0/1     Completed   0          2m31s
ingress-nginx-admission-patch-gcj9f         0/1     Completed   0          2m31s
ingress-nginx-controller-76679bcc4b-qgg4d   1/1     Running     0          2m41s

# tengo un nuevo servicio nginx-controler
➜ kubectl get svc -n ingress-nginx
NAME                                 TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)                      AGE
ingress-nginx-controller             NodePort    10.110.18.92   <none>
 80:32698/TCP,443:31579/TCP   6m55s
ingress-nginx-controller-admission   ClusterIP   10.111.14.17   <none>
 443/TCP                      6m55s
```

Para hacer los ejemplos de tipos de reglas de ingres utilizo el siguiente deployment y servicio, es un deployment que despliega pods nginx que en su index muestran el hostname del pod, el servicio balancea las ip de los pods a la ip del servicio en el puerto 8080.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-nginx
  labels:
    app: front
spec:
  replicas: 3
```

```
  selector:
    matchLabels:
      app: front
  template:
    metadata:
      labels:
        app: front
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        command: ["sh", "-c", "echo VERSION 1.0 desde $HOSTNAME >
/usr/share/nginx/html/index.html && nginx -g 'daemon off;'"]
---
apiVersion: v1
kind: Service
metadata:
  name: app-nginx-v1
  labels:
    app: front
spec:
  type: ClusterIP
  selector:
    app: front
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

Tipo ruta

Los ingres de tipo ruta, se especifica una ruta de entrada tipo `192.168.88.2/appv1` y el servicio donde se redirige dicha petición.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /appv1
        backend:
          serviceName: app-nginx-v1
          servicePort: 8080
```

Si despliego la app nginx solo tengo conexion dentro del cluster gracias al servicio `app-nxinx-v1` que esta en modo clusterIP

```
# deplegar app nginx con su servicio
➜ kubectl apply -f ingres/app-ingres-example.yml
deployment.apps/app-nginx created
service/app-nginx-v1 created
```

```
# veo la ip que a creado dentro del cluster y que expone el puerto 8080
➜ kubectl get svc
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
app-nginx-v1   ClusterIP   10.98.154.19   <none>        8080/TCP   7m53s
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP    67d

# los pods funcionan correctamente
➜ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
app-nginx-5fddd97c47-kgk2b   1/1     Running   0          2m44s
app-nginx-5fddd97c47-tc8g2   1/1     Running   0          2m44s
app-nginx-5fddd97c47-zbrkl   1/1     Running   0          2m44s

# y desde dentro del cluster puedo hacer peticiones
➜ curl 10.98.154.19:8080
VERSION 1.0 desde app-nginx-5fddd97c47-kgk2b
```

Ahora toca aplicar las reglas del ingres, ya teniendo corriendo el `nginx-controler` simplemente añadiendo el objeto ingres, nginx-controler ya lo detecta y esta vlisto para aplicar sus reglas.

```
# miro que el controles esta en marcha y exponiendo puertos
80:32698/TCP,443:31579
➜ kubectl get svc -n ingress-nginx
NAME                                 TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)                      AGE
ingress-nginx-controller             NodePort    10.110.18.92   <none>
 80:32698/TCP,443:31579/TCP   38m
ingress-nginx-controller-admission   ClusterIP   10.111.14.17   <none>
 443/TCP                      38m

# espliego reglas de ingres
➜ kubectl apply -f ingres/ingres-rules.yml
ingress.networking.k8s.io/test-ingress configured

# desde la ip del cluster compruebo
➜ curl 192.168.88.2:32698/appv1
VERSION 1.0 desde app-nginx-5fddd97c47-kgk2b
```

> **nota**: Hay que tener en cuenta que el servicio `nginx-controler` esta en modo `nodePort` y solo exporta a puertos de `30000-32767` de forma dinamica, si queremos cambiar esto, se tendra que modificar el servicio `nginx-controler`.

Tipo dominio

Igual que podemos indicar una ruta para redirigir a una app, también podemos especificar un dominio o subdominio tipo `nginx.mydomain.org, nextcloud.mydomain.org, ...`

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
```

```
  - http:
      paths:
      - path: /appv1
        backend:
          serviceName: app-nginx-v1
          servicePort: 8080
  - host: mydomain.org
    http:
      paths:
      - path: /
        backend:
          serviceName: app-nginx-v1
          servicePort: 8080
```

Una vez desplegado en el cluster la regla, desde un host externo compruebo el resultado.

```
# añado la resolución del dominio manualmente al no utilizar dns
[jorge@pc03 ~]$ sudo vim /etc/hosts

# desde un host externo accedo al cluster por el dominio indicado.
[jorge@pc03 ~]$ curl mydomain.org:32698
VERSION 1.0 desde app-nginx-5fddd97c47-tc8g2
```

TLS

Tls junto a ingres permite que la entrada al cluster por el proxy de ingres se hagan cifradas, en el siguiente ejemplo se muestra como añadir unas claves auto-firmadas a un objeto ingres.

Crear claves y crear secret:

```
➜ openssl genrsa -out clusterkey.pem
➜ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout clusterkey.pem -
out clustercrt.pem -subj "/CN=mydomain.org/O=mydomain.org"

# crear secret
➜ kubectl create secret tls cert-cluster --key clusterkey.pem --cert
clustercrt.pem
```

También se puede añadir con un template

```
apiVersion: v1
kind: Secret
metadata:
  name: cert-cluster
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

En el objeto ingres se añade la sección tls con la entrada y el certificado que utilizar

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
    - hosts:
      - nextcloud.mydomain.org
      secretName: cert-cluster
  rules:
  - host: nginx.mydomain.org
    http:
      paths:
      - path: /
        backend:
          serviceName: app-nginx
          servicePort: 8080
  - host: nextcloud.mydomain.org
    http:
      paths:
      - path: /
        backend:
          serviceName: nextcloud
          servicePort: 8080
```