02/20223

KUBERNETES AL COMPLETO

DANILO RIVERO PÉREZ

Curso en Udemy realizado por Apasoft Training

ÍNDICE

INTRODUCCIÓN

Este documento se ha escrito a medida que iba avanzando en el curso de Apasoft Training expuesto en la plataforma online *Udemy*. He decidido comprar y realizar este curso de Kubernetes al completo debido a que una de mis pasiones dentro del mundo de la informática es la administración de sistemas, y esta tecnología de Kubernetes está en auge a día de hoy, por lo que he me adentrado a la misma gracias a este curso realizado por Apasoft Training.

El objetivo principal de este documento es plasmar por escrito los conocimientos aprendidos durante el curso y apuntarlos para poder consultarlos en un futuro si hiciera falta.

CURSO KUBERNETES UDEMY APASOFT TRAINING PAGADO

- Kubernettes: Cluster de contenedores que facilita mucho la vida cuando se tiene muchos contenedores docker y para automatizar el despliegue de las apps en contenedores. Utiliza nodos, namespaces, pods, etcd (bd de kubernettes), una API, servicios, agentes, etc. Permite crear un cluster de nodos que implementan determinadas funcionalidades sobre los contenedores para mejorarlos como: alta disponibilidad, tolerancia a fallos, permite escalar cuando se queda corto de recursos, trabaja de forma eficiente, se pueden realizar modificaciones en caliente, etc. Cuando teníamos solo Docker los contenedores eran como individualidades, pues con Kubernetes tenemos una alta disponibilidad que es automática de Kubernetes, por ejemplo. Los contenedores por sí mismo no son capaces de hacer nada, pero con un orquestador como Kubernetes si tiene esas funcionalidades adicionales.

ESTÁNDARES DE CONTENEDORES

- OCI: Su misión es promover la estandarización de contenedores. Los contenedores pueden ejecutarse en cualquier HW y sistema y componentes relacionados. Implementa 2 estándares: Runtime Specification (como ejecutar

contenedores) y el Image Specification (como deben ser las imágenes). Hay un tercer elemento en OCI llamado OCI. runC, el cuál es un Runtime Container muy ligero, que cumple con la especificación OCI, es utilizado por ejemplo por Docker

- CNCF (Cloud Native Computing Foundation): Organización que ordena y aglutina varios proyectos e iniciativas de código abierto que tratan de crear herramientas que apoyen a los desarrolladores de SW a correr sus apps en la nube. Pertenece a Linux Foundation. Gestiona proyectos como: Kubernetes, Prometeus, etc.

ARQUITECTURA DE KUBERNETES

- Tenemos el Maestro o control plane y el Esclavo (Nodos) o Data Plane.

- Control plane o maestro: En los cluster de producción suelen haber múltiples maestros o control planes. Se dispone de varios nodos que son los que van a trabajar, los que van a hacer el trabajo de gestionar los contenedores y las apps desplegadas dentro de esos contenedores.

- El cliente se conecta al API Server, que es un core de Kubernetes, el cuál expone todo el contenido con el cuál yo me puedo conectar a Kubernetes. Luego, lo que hace un cliente es enviar vía POST a un fichero YAML para decirle a Kubernetes lo que quiere desplegar, y eso le llega a la API server.

- El controller manager es otro componente que esta compuesto por múltiples componentes que están escuchando y que son: Node Controller, Replication Controller, Endpoints Controller y el Service Account & Token Controllers.

- El scheduler: Componente muy importante que determina cuando algo se pide, donde va a ir a parar. Donde va a poner las cosas dentro del clúster, teniendo en cuenta distintas cosas y características que le hacen determinar en que sitio pone las cosas.

- Etcd: Base de datos, es un almacén de tipo clave-valor, que todo lo que vayamos haciendo a nivel de características, componentes, Contenedores, deployments, pods ... se va a almacenar en esta Base de Datos para que todo se perviva entre arranques. Componente que es muy buena idea tenerlo replicado para tener mayor seguridad con la información de nuestro Cluster y demás.

- Kubernetes DNS: DNS interno que todos los nodos conocen, y que es imprescindible porque es donde vamos a pedir todo lo referente al nombre de la red de los servicios que quiero ofrecer en mi clúster de K8's.

- La API-SERVER, el controller Manager, el scheduler y la BD etcd, forma parte del Maestro o CONTROL PLANE.

- Cuando se trabaja en modo CLOUD hay un nuevo componente en el MAESTRO además, que se llama "Cloud controller manager", se encarga de relacionarse con los clouds (AWS, Digital Ocean, Azure, Oracle ...)

- Acto seguido, yéndonos a la parte del esclavo, tenemos el Container Runtime, que se encarga de lanzar los contenedores y hay varios tipos de Container runtime como Docker, CRIO, containerd, Kubernetes CRI ... Una vez que hayamos elegido el container runtime, tenemos los "pods.

- Pods: Componente mas pequeño que se utiliza dentro de Kubernetes. Un pod, en su forma más básica, podremos decir que es un contenedor, pero puede tener de 1 a más contenedores si esos contenedores forman una unidad de trabajo. Cuando yo despliego un container o una aplicación lo que estoy haciendo es desplegar pods.

- Kubelet: Otro componente que tenemos en los esclavos, y Kubelet es un agente que se despliega en cada uno de los nodos del clúster y que está trabajando cooperativamente con el Maestro, es el corazón de Kubernetes dentro de un Nodo o Worker.

- Kube-proxy: Es un network proxy que permite la conectividad de todos los componentes dentro del nodo con el resto del clúster.

- namespace: Carpeta en la que trabaja el clúster y voy a poder tener mis objetos.

TIPOS DE INSTALACIÓN DE KUBERNETES

- Hay varios tipos de instalación por eso genera dudas para la organización.

Las formas más habituales son:

- Local o de 1 nodo solo. (Para hacer pruebas, prácticas, autoformarnos, etc. Por ejemplo: Minikube, kind, K3s, Microk8s, Kubernetes con Docker Desktop. Con Minikube ahora si se pueden añadir más de 1 nodo al cluster).

- Instalación manual con alguna herramienta como kubeadm (Permite implementar un cluster de forma rápida y simple)

- Instalación automática con alguna herramienta como kubespray (Para desplegar un cluster de manera autónoma, completo. Hay otras herramientas para este tipo de instalación como Kops, RKE, KubeOne, KubeSphere)

- Utilizar algún proveedor como AWS o Azure. (A base de clic me creo un clúster de manera de sencilla. Solución muy potente)

- Sin herramientas (No recomendada)

DISTRIBUCIONES DE KUBERNETES

- Red Hat Openshift
- Suse Rancher
- Canonical Kubernetes

- VMware Tanzu
- Platform9 Managed Kubernetes
- Giant Swarm
- Portainer
- Etc

PLAYGROUNDS

- En la página Killercoda tenemos varios simuladores playground para probar por ejemplo: Ubuntu, CKA, Kubernetes, etc.

KUBECTL

- Herramienta para trabajar contra los clusters de Kubernetes (Windows, Linux y Mac).
- https://kubernetes.io/docs/tasks/tools/

MINIKUBE

- https://minikube.sigs.k8s.io/docs/start/ (INSTALACIÓN)

- Hace falta un virtualizador porque al iniciar Minikube crea una MV. Por lo que hay que instalar virtualBox, VMWare, Docker, Parallels, etc. Depende en que S.O estés instalas un virtualizador u otro. Docker se puede usar pero cuando se utiliza con muchas características Minikube y demás, es mejor usar un virtualizador de máquinas virtuales como VirtualBox, KVM y demás antes que usar Docker, ya que docker son contenedores.

COMANDOS INTERESANTES MINIKUBE

1) minikube status (Para ver si nuestro Minikube está funcionando. Ver si están funcionando y configurados: el host, el control plane, el kubelet, la API Server, el kubeconfig, etc.)

2) minikube logs (Para ver todos los logs. Es funcional por si falla algo y ver lo que pasa y demás)

3) minikube ip (Para saber a que IP tenemos que conectarnos para hacer algún tipo de acceso. Esta es la IP del servidor donde se están ejecutando todos los pods, contenedores, servicios, deployments, etc. Ya que recordemos que minikube es una máquina y debajo de esa máquina se ejecutan todos los recursos)

4) minikube start (Arrancar un cluster local de kubernetes)

5) minikube stop (Parar un cluster local de kubernetes)

6) minikube pause (Pausar kubernetes)

7) minikube unpause (Reanudar kubernetes)

8) minikube delete (Eliminar un clúster local de kubernetes)

9) minikube dashboard (Acceder al panel gráfico de kubernetes que corre dentro del clúster minikube. Para monitorear)

10) minikube mount (Montar el directorio especificado dentro de minikube)

11) minikube ssh (Entrar mediante ssh al entrono de minikube para debugear)

12) minikube kubectl (Correr un binario de kubectl)

13) minikube node (Usa add, remove, list para agregar, eliminar o listar nodos adicionales)

14) minikube cp (Copie el fichero dentro de minikube)

15) minikube profile list (Nos lista los cluster de minikube y la información acerca de ellos)

16) minikube profile (Devuelve el perfil en el que estoy, el cluster con el que estoy trabajando)

17) minikube delete -p ClusterDesarrollo (Borrar el clúster con nombre: ClusterDesarrollo)

18) minikube start -p ClusterDesarrollo (Arrancar o crear un cluster llamado ClusterDesarrollo)

19) minikube service list (A veces el servicio minikube falla, y al hacer aveces minikube ip, nos da una IP que no es la buena, así ue comprobar siempre que tengamos duda con minikube service list, que nos muestra los servicios nodePort, Loadbalancer y ClusterIP)

FICHEROS DE MINIKUBE

- Directorio /root/.kube: Directorio que se utiliza en todos los Kubernetes y tiene varias configuraciones. Contiene un fichero de configuración que contiene la información de conexión a los clústers de Kubernetes.

- Directorio /root/.minikube: Exclusivo de Minikube y tiene ficheros relativos a minikubes. Fichero de logs, de certificados, etc. Hay un directorio llamado "machines" que muestra todos los cluster de minikube creados

- Script /usr/local/bin/minikube: Script de minikube

CLUSTER DE MINIKUBE CON MÚLTIPLES NODOS

- minikube start --driver=docker -p clusterDesarrollo --nodes=2 (Se crea un cluster de 2 nodos)

CAMBIAR LA CONFIGURACIÓN DE UN CLÚSTER

- https://minikube.sigs.k8s.io/docs/commands/config/

- minikube config set memory 4G -p ClusterDesarrollo (Le cambia la memoria al clúster ClusterDesarrollo. Esto solo se activa cuando se borra y se vuelva a crear el container)

- minikube config get disk-size (Nos da el Valero que tiene. Solo lo muestra si antes lo hemos modificado, por defecto si no lo cambias con "set" no lo muestra. En el fichero .minikube/config/config.json están los cambios que se han hecho con "set")

CAMBIAR DE CLÚSTER

- En el fichero .kube/config hay secciones llamadas "clusters" donde están los cluster de Kubernetes. En la línea "current-context" nos dice que clúster se está dando actualmente.

- minikube profile ClusterDesarrollo (Me cambio de perfil y empiezo a utilizar e cluster clusterDesarrollo). También podría cambiar la línea "current-context" del fichero .kube/config y poner ClusterDesarrollo y se cambiaría igual.

MINIKUBE DASHBOARD

- minikube dashboard (Interfaz gráfica de Minikube)

TRABAJAR CON OTRO CONTAINER RUNTIME

- minikube start --container-runtime=cri-o -p cluster2 (Usa el container runtime de cri-o, pero de driver de VM sigue usando docker)

OTRA OPCIÓN SI NO TENEMOS MINIKUBE

- Docker Desktop (Hyper-V o WSL2 [Windows SubSystem for Linux])

VISUAL STUDIO CODE PARA TRABAJAR CON KUBERNETES

- En Visual Studio Code descargamos las extensiones Docker y Kubernetes. Con esta herramientas nos facilitaría trabajar en la vida real.

CICLO DE VIDA DE UNA APLICACIÓN CONTENERIZADA

1) Convertimos nuestro app en una imagen que luego pueda convertirse en un contenedor cuando se despliegue dentro de nuestro clúster de Kubernetes.

2) Luego cuando yo quiera hacer el deploy, esa imagen la subo a mi contenedor.

3) Tras ello, hay 2 formas de desplegar o pasar esta imagen y crear estos contenedores basados en esta imagen a nuestro clúster: Modo imperativo y modo declarativo. El modo imperativo lo que hace es, tengo esta imagen y le digo créame un contenedor con varias réplicas, por ejemplo. Hay un comando "kubectl run" para hacer esto (Para pruebas y demás), pero lo mejor es hacer la versión declarativa. El modo declarativo usa un fichero de manifest que puede ser o bien YAML (más adecuado) o JSON donde se declara como quiero desplegar mi aplicación y pedirle al clúster que lo cree.

4) El estado deseado de la aplicación, lo que hace Kubernetes es preguntar si lo que venía en el manifest coincide con la realidad, es decir cual es el estado

deseado y cual el estado real que tengo ahora. Por ejemplo, si yo le he dicho que necesito 5 replicas y de repente se me caen 2 y me quedo con 3, pues entonces Kubernetes a través de los controladores se encarga de darme los recursos que deseo tener.

PODS

- La unidad mínima de trabajo, el runtime más pequeño de Kubernetes. Internamente están compuestos por contenedores. Por ejemplo en docker el componente mínimo es el contenedor y en Kubernetes son los pods. Los pods le da unas funcionalidades a un contenedor que NO tiene por sí mismo. Envoltorio que tiene una serie de propiedades comunes con las que gobierna sus contenedores.

- Imaginamos que tengo una app (que estará en forma de imagen en algún sito, por ejemplo Docker Hub), un apache web, y la quiero desplegar dentro de Kubernetes, y eso lo estoy haciendo o bien a través de un fichero YAML (modo declarativo) o bien a través de un comando (modo imperativo). Y esta app queda embebida dentro de un POD, ya que esa imagen cuando la subo a un POD la estoy convirtiendo en un contenedor (el cual se encierra dentro de un POD).

- La mayoría de los PODS solo tienen un contenedor. Los PODS le dan direcciones, puertos, hostnames, sockets, memoria, volúmenes, etc a los contenedores y todo englobado con una API IP.

- Los PODS se despliegan dentro de los nodos/workers del clúster. Los nodos manejan los PODS, yo asocio un POD a un contenedor y ese POD se despliega dentro de un nodo del clúster.

- Los PODS no tienen estado, no se debe guardar información en los PODS.

- En definitiva, cuado yo utilizo una imagen de Docker y la quiero desplegar dentro de Kubernetes no se hace directamente con un contenedor pelado ... sino

que se hace en un POD y dentro de ese POD vamos a tener una serie de características y funcionalidades que son las que le van a dar sentido a nuestro proyecto o a nuestro componente.

MÚLTIPLES CONTENEDORES EN PODS

- *¡AVISO!:* Lo ideal es usar tan solo 1 contenedor en un POD.

- Aveces, no tiene sentido poner muchos servicios en varios contenedores en tan solo un POD. Mejor idea es separarlos en varios PODS y poner 1 contenedor en cada POD diferente.

- Por ejemplo, si tengo una app donde tengo 2 módulos que están tan unidos lógicamente, pues entonces si es buena idea tenerlo en dos contenedores en un solo POD.

- Imagínate que tengo un servidor web Apache y una BD MySQL, en dos contenedores que están en 1 solo POD. Si yo todos los días tengo que reiniciar el apache, al tenerlo todo en 1 solo POD también estoy reiniciándose el MySQL sin necesidad, ya que MySQL es un servicio autónomo e individual que no tiene que ver con el servidor web Apache. Es más, un POD tiene una IP, puerto ... común. Por eso, lo mejor es poner en 1 pod un contenedor con el servidor web Apache y en otro pod otro contenedor con el servicio MySQL.

CREAR UN POD MODO IMPERATIVO, FORMAS DE BORRAR PODS Y VER PROPIEDADES

MODO IMPERATIVO:

- kubectl run nginx1 --image nginx (Crear un POD que va a contener un contenedor que está basado en una imagen de nginx)

- kubectl run apache1 --image=httpd --port=8080 (Otra manera de crear un POD/contenedor y especificarle un puerto)

- kubectl delete pod/apache1 (Borrar el pod/contenedor llamado "apache1")

- kubectl delete pod apache1 (Borrar el pod/contenedor llamado "apache1")

- kubectl delete pod apache1 --grace-period=5 (Borrar el pod/contenedor llamado "apache1" pero antes de borrarlo del todo se espera 5 segundos)

- kubectl delete pod apache1 --now (Borrar el pod/contenedor llamado "apache1" inmediatamente)

- kubectl delete pods --all (Borrar todos los pods que tengamos. ¡CUIDADO CON ESTE COMANDO!)

- kubectl get pods (Nos da los pods creados)

- kubectl get pod apache1 (Nos da la info solo del pod/contenedor llamado "apache1")

- kubectl get pods -o wide (Igual que el anterior pero nos da más info interesante como la IP del pod, el nodo donde se está ejecutando minikube, etc.)

- kubectl get all (Visualiza todo lo que esté dentro del namespace, deployments, pods, sis, rs, etc.)

VER PROPIEDADES DE UN POD

- kubectl describe pod/nginx1 (Nos da características muy específicas del recurso que se le indique, por eso hay que especificar el tipo de recursos del que queremos ver sus características)

EJECUTAR COMANDOS CONTRA UN POD: EXEC

- kubectl exec nginx1 -- ls (Hacemos un ls del contenedor/pod nginx1)

- kubectl exec nginx1 -- uname -a (Hacemeos un uname -a del contenedor/pod nginx1)

- kubectl exec nginx1 -it -- bash (Ejecutamos el comando bash de forma interactiva dentro del contenedor/pod nginx1)

VER LOGS DE UN POD Y MODIFICAR SU CONTENIDO

- kubectl logs apache1 (Ver los logs del pod/contenedor apache1)

- kubectl logs -f nginx1 (Ver los logs del pod/contenedor nginx1 pero de manera que se quede ahí la pantalla)

- kubectl logs apache1 --tail=30 (Vemos las ultimas 30 líneas de los logs del contenedor/pod apache1)

- kubectl exec apache1 -it -- bash (Accedo al contenedor/pod de apache1 y puedo ver que apache funciona haciendo un wget localhost, por ejemplo. Además, los contenedores/pods están basados en Debian)

PROBAR EL POD CON KUBECTL PROXY

- Si yo quisiera comprobar que mis pods/contenedores nginx1 y apache1 funcionan bien, y comprobar esto desde un navegador de internet externo (ya que desde los contenedores no tengo navegadores gráficos, como mucho puedo usar curl, wget, etc), no puedo hacerlo porque eso lo puedo hacer, sí, pero con deployments y servicios y así poder ver mi aplicación desde fuera del clúster, pero de momento, con tan solo los pods/contenedores que se han creado con el comando kubectl run no puedo hacer eso. Sin embargo, con la herramienta "kubectl proxy" si que podría hacerlo, ya que kubectl proxy me permite a través

de un navegador web acceder a un recurso del clúster y por lo tanto poder probar la aplicación.

- kubectl proxy (Me abre en el servidor localhost un puerto por el que puedo preguntar por las características del clúster y de las aplicaciones. Tras ejecutar kubectl proxy me puedo ir a mi navegador y poner "localhost:8001" [8001 --> puerto que me abre kubectl proxy]. Tras esto, puedo ver diferentes recursos y demás, pero si pongo "http://localhost:8001/api/v1/namespaces/default/pods/apache1/proxy/" puedo ver de manera gráfica la página de Apache y ver si funciona bien o no. Pero no preocuparse mucho por esto, porque lo normal es ver esto con un servicio)

PROBAR EL POD CON UN SERVICIO

- kubectl expose pod nginx1 --port=80 --name=nginx1-service --type=LoadBalancer (Expongo el recurso (pod) llamado nginx1 al exterior y le especifico el puerto por el que escucha el pod [puerto 80 en este caso]. Además, le ponemos nombre al servicio que estamos creando y para que desde fuera del clúster se pueda acceder al pod hay que indicarle como tipo "LoadBalancer". De esta manera, se ha creado un servicio para poder acceder al pod llamado nginx1 desde fuera del clúster)

- kubectl get svc (Vemos los servicios que hay ejecutándose y sus características)

- kubectl get services (Vemos los servicios que hay ejecutándose y sus características. Otra forma, más larga de escribir y de hacer lo mismo de arriba)

- kubectl get svc nginx1-service (Vemos solo el servicio nginx1-service)

- kubectl get svc -o wide (Vemos además el selector del servicio y demás datos si se da el caso)

- kubectl describe svc/nginx1-service (Vemos muchas más características del servicio)

- De esta manera, al haber creado este servicio, me da un puerto (Por ejemplo el 32159) desde el que yo pueda acceder desde fuera del clúster, desde el exterior, pero a su vez ese servicio apunta al puerto interno del pod, que en este caso es el puerto 80 (puerto del pod/contenedor nginx1)

- Ahora, con este servicio, yo puedo acceder desde un navegador externo, a través de la URL: "http://192.168.59.100:32159/" [192.168.59.100 es la IP del cluster de kubernetes, se puede averiguar haciendo minikube ip y el 32159 es el puerto que se asignó al servicio nginx1-service] al POD llamado nginx1 y ver si funciona bien. Con esto, ya hemos conseguido, al igual que con kubectl proxy, acceder al pod nginx1 desde fuera del clúster de Kubernetes, desde el exterior.

PROBAR EL POD CON UN PORT-FORWARDING

- Consiste en mapear un puerto local con un puerto remoto, similar al servicio pero este recurso de Port-Forwarding es un recurso general que se suele utilizar en redes, y el recurso de servicio es interno de Kubernetes.

- kubectl port-forward nginx1 9999:80 (Indicándole que voy a entrar por el puerto local de la máquina "9999" y voy a acceder por el puerto "80" del entrono remoto. Así se puede probar el funcionamiento de un pod usando Port-Forwarding. Se accedería desde un navegador externo con la siguiente URL: localhost:9999)

- Es una técnica que se puede utilizar de manera rápida en entornos que NO son de producción, sobre todo porque es más rápida que estar creando un servicio para ver si funciona un recurso de manera correcta y demás.

VER LOS PODS DESDE UN NODO DEL CLÚSTER

- Haciendo "minikube ssh" accedemos a la MV de minikube, y desde ahí podemos hacer "docker ps" y ver los contenedores que hay ejecutándose demás.

INTRODUCCIÓN A YAML

- YAML es un lenguaje, una manera de construir ficheros, que trabaja con serialización de datos.

CREAR UN POD MEDIANTE UN MANIFEST

- Vamos a hacer un ejemplo real partiendo desde 0. Vamos a crear nuestra propia imagen de Docker (podríamos haber usado una de dockerHub), y vamos a subir esa imagen creada con el DockerFile a nuestro DockerHub. Por lo tanto, vamos a crear un DockerFile que contendrá un nginx pero con un mensaje personalizado en su página principal (index.html).

1) Creamos el Dockerfile que tiene el nginx personalizado. De esta manera creamos una imagen, que luego la convertiremos en un Pod a través de un fichero YAML:

```
##Descargamos UBUNTU
FROM ubuntu

##Actualizamos el sistema
RUN apt-get update

##En algunas versiones de Linux es necesario configurar una variable para el TIMEZONE
ENV TZ=Europe/Madrid
```

```
##Luego creamos un fichero llamado /etc/timezone para configurar
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ >
/etc/timezone

##Instalamos NGINX
RUN apt-get install -y nginx

##Creamos un fichero index.html en el directorio por defecto de
nginx
RUN echo 'Ejemplo de POD con KUBERNETES y YAML' >
/var/www/html/index.html

##Arrancamos NGINX a través de ENTRYPOINT para que no pueda ser
modificado en la creación del contenedor
ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]

##Exponemos el Puerto 80
EXPOSE 80
```

2) docker build -t danilooo99/nginx:v1 . (Construimos la imagen)

3) docker login (Nos logueamos con nuestra cuenta de DockerHub)

4) docker push danilooo99/nginx:v1 (Subimos la imagen a DockerHub)

5) docker run -d -p 80:80 --name nginx danilooo99/nginx:v1 (Creamos un contenedor llamado nginx que usa la imagen que hemos subido a DockerHub)

6) Abro un navegador y pongo "localhost" y veo el funcionamiento del contenedor, del nginx personalizado.

7) docker stop nginx (Paro el contenedor ya que solo lo cree de prueba para ver que todo funcionaba bien y que la imagen creada estaba bien y funcionaba correctamente)

8) Ahora, para subir todo esto a Kubernetes de manera declarativa, usamos un fichero YAML llamado nginx.yaml, por ejemplo. El fichero nginx.yaml contiene lo siguiente:

```
apiVersion: v1 (Versión de la API de Kubernetes)
kind: Pod (Nombre del recurso a crear)
metadata: (Los metadatos, poniéndole el nombre del pod, y algunas etiquetas como zone y version que veremos más adelante, son como los tags)
  name: nginx
  labels:
    zone: prod
    version: v1
spec: (Las especificaciones del pod, las características deseadas que quiero que tenga el pod. En este caso, se ha usado la especificación del contenedor "containers:", donde se ha especificado el nombre del contenedor que estará dentro del pod a crear y la imagen en la que estará basado ese contenedor, que es la imagen que se ha subido a docker hub anteriormente)
  containers:
   - name: nginx
     image: danilooo99/nginx:v1
```

9) Creamos un recurso de Kubernetes (en este caso, un pod) haciendo uso de un fichero YAML:

- kubectl create -f nginx.yaml

10) Podemos comprobar que nuestro pod funciona bien con kubectl proxy, creando un servicio o con port-forwarding, como hemos hecho anteriormente ya.

GENERAR LA CONFIGURACIÓN DE UN POD EN FORMATO JSON O YAML

- kubectl get pod/nginx -o yaml (Nos saca mucha información de la configuración del pod llamado "nginx" en formato YAML. Con > Fichero.yaml, puedo exportar toda esta info a un fichero)

- kubectl get pod/nginx -o json (Nos saca mucha información de la configuración del pod llamado "nginx" en formato JSON. Con > Fichero.json, puedo exportar toda esta info a un fichero)

PODS DASHBOARD

- Desde el dashboard se pueden modificar, ver, comprobar, crear y eliminar PODS.

PODS MULTICONTENEDORES

- Voy a crear un POD que va a tener dos contenedores; por un lado un contenedor con el servicio nginx; y otro contenedor con un servicio de monitorización que estará continuamente haciendo pings al servidor web nginx (Como sabemos dos contenedores en un mismo pod NO ES RECOMENDABLE, pero esto es un ejemplo).

1) Creamos un pod llamado "multi" que contendrá los 2 contenedores mencionados anteriormente. Para ello creamos el fichero multi.yaml con lo siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: multi
spec:
  containers:
  - name: web
    image: nginx
    ports:
    - containerPort: 80   (El contender nginx escuchará por el puerto 80)
  - name: frontal
    image: alpine (El segundo contenedor del POD estará basado en alpine, distribución de linux muy ligera)
    command: ["watch", "-n5", "ping",  "localhost"] (Ejecuta el comando watch -n5 ping localhost. De esta manera se ejecuta el comando ping localhost cada 5 segundos [-n5]. Al hacer un ping a localhost, estoy haciendo un ping a la IP del pod, ya que este comando se ejecuta dentro del POD llamado "multi")
```

2) kubectl apply -f multi.yaml (Creo el POD con 2 contenedores haciendo uso del fichero multi.yaml y además uso "apply" en lugar de "create")

3) kubectl logs multi -c frontal (Nos muestra los logs solo del container llamado "frontal" del POD multi)

4) kubectl exec multi -c frontal -- date (Ejecuto el comando date en el contenedor frontal del pod multi)

COMANDO APPLY. TRABAJANDO DE FORMA DECLARATIVA

- Con este comando puedo modificar recursos, añadirle propiedades, etc. Anteriormente, para crear los pods hacíamos "kubectl create -f nginx.yaml", es decir usábamos create para crear el pod. Pero si yo por ejemplo al fichero nginx.yaml quiero añadirle una nueva label ... si yo vuelvo a crear el pod con create "kubectl create -f nginx.yaml", entonces no me va a dejar, ya que diría que ese pod con ese nombre ya existe. Por esta misma razón, es mejor usar apply ya que con él si puedo modificar el pod y sus propiedades.

- kubectl apply -f nginx.yaml (Creo el pod haciendo uso del fichero "nginx.yaml", pero esta vez con "apply" en lugar de "create". De esta manera, ahora si yo modifico el fichero "nginx.yaml" y le añado otra label y vuelvo a ejecutar el "kubectl apply -f nginx.yaml" funcionaría todo correctamente y se aplicarían las modificaciones realizadas.)

COMO SE DEBE REBOTAR/REINICIAR UN POD (restartPolicy)

- Hay 3 políticas de restart (restartPolicy) que se usan para cuando un POD termina:

* Always: Siempre que el pod tenga algún problema o se caiga se reiniciará siempre. Por defecto si no se pone restartPolicy, su valor será Always.

* OnFailure: El pod solo se reinicia si hay algún problema o falla el pod. Es decir el POD solo se reiniciar si falla el contenedor. Si por ejemplo yo tengo un servidor de aplicaciones en un contenedor de ese pod y yo lo paro, pues al parar el servidor también se va a parar el POD, ya que es el proceso padre que se está ejecutando dentro de ese contenedor, pero yo no quiero que se reinicie ese pod, porque no ha sido un error sino tan solo una parada del servidor de aplicaciones. Esto con OnFailure me lo evito y no reiniciaría el pod, en ese caso.

* Never: Quiero que el POD nunca haga un reinicio, es decir, si el pod falla, se cae o tiene algún problema, no se reiniciará salvo que lo haga yo mismo a mano.

1) Crea un POD llamado tomcat que en su interior tiene un contenedor llamado tomcat que está basado en el mismo servidor Apache Tomcat. RestartPolicy ALWAYS:

```
apiVersion: v1
kind: Pod
metadata:
  name: tomcat
  labels:
    app: tomcat
spec:
  containers:
   - name: tomcat
     image: tomcat
  restartPolicy: Always
```

- kubectl apply -f restart-always.yaml (Creo el pod tomcat con la política always usando el fichero YAML especificado anteriormente)

- Para ver cuántos veces ha rebotado/reiniciado, podemos hacer "kubectl describe pod tomcat" y mirar la variable RestartCount.

- Si yo ahora hago "kubectl exec -it tomcat -- bash" y ejecuto dentro de la consola del contenedor tomcat "ps -ef" veremos el proceso de Tomcat ejecutándose (/usr/local/openjdk-8/bin/java). Si yo paro tomcat: "catalina.sh stop", se me sale de la consola del contenedor y haciendo un "kubectl get pods" veremos que el pod de tomcat sigue running pero ya ha tenido 1 reinicio debido a que el servicio tomcat se cayó, ya que lo apagué.

- Ahora bien, si yo vuelvo a realizar todos los pasos anteriores pero tan solo cambiando la política restartPolicy a On-Failure, entonces el pod tendrá 0 reinicios, ya que el contenedor no ha fallado, solamente he parado el servicio tomcat en el contenedor, por lo que con la política On-Failure no se reiniciaría el pod.

2) Crea un POD llamado on-failure que en su interior tiene un contenedor llamado on-failure que está basado en una imagen busybox (Herramienta para ejecutar utilidades Linux (ls, cat, ps ...) en distros que no tengan esas utilidades, como Android, windows, otros linux, etc) y que va a ejecutar el comando "echo Ejemplo ... && exit 1" dentro del contenedor (con el exit 1 sale con algún tipo de fallo). RestartPolicy ONFAILURE:

```
apiVersion: v1
kind: Pod
metadata:
  name: on-failure
  labels:
    app: app1
spec:
  containers:
  - name: on-failure
    image: busybox
    command: ['sh', '-c', 'echo Ejemplo de pod fallado  && exit
1']
  restartPolicy: OnFailure
```

- kubectl apply -f on-failure.yaml (Creo el pod on-failure con la política OnFailure usando el fichero YAML especificado anteriormente)

- Si ahora hago un "kubectl get pods", veré que el pod ha fallado, es decir tendrá un ERROR de status, debido a que en el contenedor on-failure se ejecuto "echo Ejemplo error && exit 1" y el exit 1 lo que hace es informar que ha habido un error. Debido a que el pod ha fallado entonces la política On Failure ahora sí entra en juego y se reinicia el pod (Restarts 1). Además, cada vex que Kubernetes intenta arrancar el pod, y se ejecuta el echo error && exit 1 en el contenedor, se reiniciar el pod, debido a que seguirá fallando.

- Si hacemos kubectl logs pod on-failure, veremos los logs de ese pod y veremos el echo ejecutado en el contenedor on-failure.

3) Crea un POD llamado never que en su interior tiene un contenedor llamado never que está basado en una imagen busybox (Herramienta para ejecutar utilidades Linux (ls, cat, ps ...) en distros que no tengan esas utilidades, como Android, windows, otros linux, etc) y que va a ejecutar el comando "echo Ejemplo ... && exit 1" dentro del contenedor (con el exit 1 sale con algún tipo de fallo). RestartPolicy NEVER:

```
apiVersion: v1
kind: Pod
metadata:
  name: never
  labels:
    app: app1
spec:
  containers:
  - name: never
    image: busybox
    command: ['sh', '-c', 'echo Ejemplo de pod fallado  && exit
1']
  restartPolicy: Never
```

- kubectl apply -f never.yaml (Creo el pod never con la política Never usando el fichero YAML especificado anteriormente)

- Como se ve, si hago un "kubectl get pods" vemos que el STATUS es Error, debido a que dentro del contenedor never se ha ejecutado "echo error en el pod && exit 1" y el exit 1 provoca un fallo con ese código de error, y por eso el pod falla. Pero a diferencia de On-Failure, ahora el pod no se reinicia nunca, debido a que le he dicho que no reinicie nunca el pod aunque hayan fallos, se caiga o lo que sea, debido a que la restartPolicy ahora es Never.

CREAR UN POD y TRABAJAR CON ELLOS EN VSCODE

- Me creo mi carpeta y ahí puedo crearme ficheros YAML y haciendo clic derecho-abrir terminal integradas, ya me posiciona en el directorio y tan solo le doy a kubectl apply -f fichero.yaml y desde esa terminal ya trabajo. Más cómodo.

- Además, si voy a kubernetes en el VSCode y voy a PODS y hago click-derecho en un pod, puedo hacer GET, DELETE, port-forwarding, exec, etc y se me abre una terminal ejecutando la operación en ese pod.

LABELS

- A través de las labels (etiquetas) se realizan varias relaciones entre los objetos de Kubernetes. Las label son un conjunto de propiedad: "valor". Los nombres de los labels son inventados, no tengo que llamarlas de una determinada forma. Hay otras etiquetas de sistemas que no se deben utilizar porque ya están "cogidas" para otras cosas y demás. Por ejemplo:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: tomcat
  labels:
    estado: "desarrollo"
spec:
  containers:
   - name: tomcat
     image: tomcat
```

- En el YAML anterior que crea un pod con un contenedor que tiene un tomcat, existe una etiqueta llamada estado y su valor es desarrollo.

- Si yo creo ese pod, ejecutando: "kubectl apply -f tomcat2.yaml" y ejecuto el "kubectl get pod tomcat --show-labels", puedo ver las etiquetas que tiene ese pod.

- Además, si ejecuto kubectl get pod tomcat --show-labels -L estado, me sale por pantalla una nueva columna llamada estado con el valor que tiene la etiqueta estado, que en este caso es desarrollo.

TRABAJAR CON LABELS

- Las labels se pueden modificar, tanto de manera imperativa como de manera declarativa. Además, las etiquetas tienen una nomenclatura, por ejemplo no pueden pasarse de 63 caracteres de largo, admiten puntos y guiones, no admiten espacios en blanco, etc.

FORMA IMPERATIVA:

- kubectl label pod tomcat responsable=danilo (Con esto, se ha añadido una nueva etiqueta llamada responsable con el valor danilo al pod tomcat)

- kubectl label pod tomcat --overwrite estado=test (Con esto, estoy modificando una etiqueta del pod tomcat. He sobreescrito la etiqueta estado que antes valía "desarrollo" y ahora pasa a tener el valor de "test", ya que la he sobreescrito)

- kubectl label pod tomcat responsable- (Con esto, elimino la etiqueta responsable del pod tomcat)

FORMA DECLARATIVA (FORMA MEJOR DE CREAR, MODIFICAR Y BORRAR ETIQUETAS):

- Edito el fichero YAML. Añado, modifico, o elimino las labels correspondientes.

- Vuelvo a ejecutar el apply del fichero YAML: kubectl apply -f tomcat.yaml

NOTA: Con kubectl describe pod/tomcat, podemos ver las labels del pod.

SELECTORES

- Sirven para encontrar objetos. Son el objeto primario con los que se relacionan algunos de los componentes, como los deployments, los replicaSets, los servicios, etc.

- kubectl get pods --show-labels -l estado=desarrollo (Me salen todos los pods que tengan la etiqueta estado=desarrollo. En lugar de -l, también vale --selector)

- kubectl get pods --show-labels -l estado=desarrollo,responsable=danilo (Me salen todos los pods que tengan la etiqueta estado=desarrollo y además la etiqueta responsable=danilo. También se puede poner ==)

- kubectl get pods --show-labels -l responsable!=danilo (Me salen todos los pods cuya etiqueta responsable no sea danilo)

- kubectl get pods --show-labels -l 'estado in(desarrollo)' (Me salen todos los pods que tengan como etiqueta estado=desarrollo. Usando conjuntos)

- kubectl get pods --show-labels -l 'estado in(desarrollo,testing)' (Me salen todos los pods que tengan como etiqueta estado=desarrollo o estado=testing. Usando conjuntos)

- kubectl get pods --show-labels -l 'estado notin(desarrollo,testing)' (Me salen todos los pods que NO tengan como etiqueta estado=desarrollo o estado=testing. Usando conjuntos)

- kubectl delete pods -l estado=desarrollo (Borra los pods que tengan como label estado=desarrollo)

ANOTACIONES

- Similares a los labels, pero son más bien descriptivas. Poner documentación, comentarios, etc.

```
apiVersion: v1
kind: Pod
metadata:
  name: tomcat7
  labels:
    estado: "produccion"
    responsable: "danilo"
  annotations:
    doc: "Se debe compilar con gcc"
    adjunto: "ejemplo de anotacion"
spec:
  containers:
```

```
  - name: tomcat7
    image: tomcat
```

- kubectl apply -f tomcat7.yaml (Creo el pod a través del fichero YAML)

- No hay un get específico para ver las anotaciones, pero sí puedo hacer esto: kubectl describe pod tomcat7 y podemos ver las anotaciones desde ahí.

- Otra forma es buscar por el path de la metadata: kubectl get pod tomcat7 -o jsonpath={.metadata.annotations y me lo devuelve en forma de mapa.

DASHBOARD Y LABELS

- En el dashboard de minikube también se pueden ver las labels y anotaciones.

WORKLOADS Y CONTROLLERS

- En Kubernetes teníamos nodos maestros y nodos esclavos.

- Un workload es algo que estoy utilizando para poder desplegar contenedores, son los que se encargan de hacer el trabajo pesado, de hacer los procesos. Los pods son el workload más básico. Envolturas que van a acoger uno o varios pods y van a hacer que se ejecuten de una determinada manera, siempre teniendo en cuenta el estado ideal que se ha decidido para ese conjunto de pods.

- Los controllers, tienen como trabajo estar continuamente controlando que el clúster está correcto y que los PODS y los workloads que se han definido se adaptan a lo que yo quiero, es decir, intentan mantener siempre en forma el clúster de acuerdo a los instrucciones que nosotros le damos.

DEPLOYMENTS

- Componente que va a orquestar el despliegue de los pods, las réplicas, las recuperaciones ante catástrofes, etc. En definitiva, gestiona u orquesta los despliegues de nuestras aplicaciones. Son los componentes que van a envolver a los pods, y que además les da una serie de características, como por ejemplo: hacer updates y rollbacks de manera sencilla y además comprobar o mantener que funciona correctamente.

REPLICASETS

- Son los encargados de hacer el escalado de los pods cuando es necesario, es decir, subir o bajar los pods, de acuerdo a lo que el clúster tenga indicado. Es decir, cuando yo determino que deben haber 5 pods, los replicasets van a intentar a toda costa, mantener esos 5 pods funcionando puesto que es el mandato que se le ha dado.

STATEFUL SET

- Objeto que gestiona el despliegue y el escalado de los pods y garantiza el orden y la unicidad de esos pods.

DAEMON SET

- Un componente que asegura que todos los nodos del clúster van a tener una copia de un Pod. Si yo por ejemplo, añado un nuevo nodo al clúster, el daemon set intentará a toda costa que haya un pod dentro del mismo. Es decir se encarga que los pods estén presentes en todos los nodos de clúster.

JOB

- Componente que crea uno o varios pods y se van a asegurar que un número determinado de ellos se terminen satisfactoriamente. Es decir que se encarga que los pods funcionen correctamente y cuando el número que yo le he indicado de terminaciones correctas se alcance, el job se habrá terminado.

CRON JOB

- Igual que los JOBSs pero de forma planificada con scheduler.

INTRODUCCIÓN A LOS DEPLOYMENTS

- Por ejemplo, yo tengo un pod y ese POD engloba en su interior 1 o mas contenedores, pero claro, los pods tienen varios problemas que no podemos solucionar trabajando tan solo con PODS, por ejemplo; Los pods no escalan, no se recuperan ante caídas y además no son nada buenos para hacer updates y rollbacks. Pues para solucionar todos estos problemas, aparecen los deployments, que es otro componente que es como una especie de burbuja que engloba unas propiedades, unas características, entre ellas, los pods que debe tener y que me permite tener updates y rollbacks adecuados, voy a poder recuperarme ante caídas y ademas puede escalar (añadir o quitar pods).

- Por ejemplo, yo quiero desplegar una app que es un Apache, y yo podría empezar con 1 solo pod en el deployment, pero si yo tengo problemas, puedo crear mas pods que son réplicas entre sí, pudiendo tener varios pods idénticos pero con diferente IP y que están siendo gestionados por un deployment. De esta manera, si algún pod se cae, tendría otro que podría dar el servicio. De todas formas, si mi estado deseado eran 5 pods y se me cae 1, entonces kubernetes intenta a toda costa crear un nuevo pod para que ya hayan 5 pods, ya que es el estado deseado.

- Cuando yo creo un deployments, automáticamente también me crea un componente llamado replica set, para que las replicas de los pods se hagan y gestionar la recuperación ante caídas de pods. Cuando yo hago un deployemnts, tendré: el propio deployment, replicas set y pods.

- En definitiva, los deployments son los componentes más adecuados y más habituales para desplegar mis aplicaciones en Kubernetes.

CREAR UN DEPLOYMENT DE MANERA IMPERATIVA

- kubectl create deployment apache-deployment --image=httpd (Me crea un deployment llamado apache-deployment usando la imagen de Apache. Recordar que se ha creado 1 deployment, 1 replica set y los pods)

- kubectl get deploy (Me permite ver el deployment creado)

- kubectl get rs (Me permite ver los replica sets. Puedo ver los pods deseados cuantos están funcionando, los que están ready, etc.)

- kubectl get pods (Veo el pod creado con el deployments)

VER INFORMACION DE LOS DEPLOYMENTS

- kubectl describe deploy apache-deployment (Nos da information acerca del deployment)

- kubectl get deploy apache-deployment -o wide

- kubectl get deploy apache-deployment -o yaml

DEPLOYMENTS Y REPLICASETS EN DASHBOARD

- Desde el dashboard de minikube también se pueden ver los deployments y replicases y ver sus características y demás.

CREAR UN DEPLOYMENT DE MANERA DECLARATIVA

- Creamos un deployment llamado nginx-d, que tendrá un selector (como una query) con una etiqueta para buscar objetos la cuál es app: nginx, que tendrá 2 pods (replicas) idénticos, ofreciendo alta disponibilidad si uno de los pods se cae, y esos pods ejecutaran un contenedor llamado nginx que estará basado en la imagen de nginx en su version 1.7.9 y escuchará por el puerto 80. El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione. Se usa para buscar por ejemplo objetos que tenga la etiqueta app=nginx
    matchLabels:
      app: nginx
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
```

```
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

- kubectl apply -f deploy_nginx.yaml (Creo el deploy haciendo uso del fichero YAML)

- Tras la creación veremos el deploy creado, los dos pods y el replica set.

- Además, si hacemos kubectl get deploy -o wide ó kubectl get rs -o wide, vemos el selector especificado app=nginx.

- kubectl get pods -l app=nginx (Puedo buscar por una etiqueta los pods que hayan)

- kubectl get pods -L app (Me muestra una columna con el valor de la etiqueta app)

- kubectl get deploy, pods,rs (Me muestran los deploys, pods y replica sets)

COMANDO EDIT

- Si yo me voy al YAML y en vez de 2 replicas le digo que quiero 3 y hago kubectl apply -f deploy_nginx.yaml, entonces ahora ya tengo 3 replicas en lugar de 3, he escalado.

- Hay otro forma, de editarlo y es: kubectl edit deploy nginx-d, y desde ahí podemos editar de manera dinámica el deploy añadiendo más pods, por ejemplo.

ESCALAR UN DEPLOYMENT

- kubectl scale deploy nginx-d --replicas=3 (Escalo el número de replicas de pods del deployment nginx-d. Otra forma de escalar sin usar edit ni apply)

- Ahora si yo le añado una label al fichero YAML del deploy que se llame estado=1 y ejecuto kubectl apply -f deploy_nginx.yaml, ahora puedo buscar deploys por labels. Por ejemplo: kubectl scale deploy -l estado=1 --replicas=10 (Y escalo el deploy con etiqueta estado=1 y le pongo que tenga 10 replicas de sus pods)

CONSIDERACIONES SOBRE EL ESCALADO

- Yo por ejemplo puedo hacer replicas sin miedo cuando tengo pods que ejecutan, por ejemplo, un apache web, un nginx, etc. Pero si yo tengo un pod dentro de un deployment que contiene una BD MySQL y hago 2 replicas de ese pod, serán 2 BD distintas, ya que se crean dos BD idénticas pero una vez que se repliquen y se modifiquen datos en alguna BD o se guarden nuevas cosas, habrá una BD con esos cambios y la otra no tendrá esos cambios. Para que los cambios que se realicen en una BD y se repliquen inmediatamente a su replica de manera sincronía, habría que crear un cluster de BD MySQL, por ejemplo con Galera Cluster. En este caso, si yo quiero tener pods dentro del deployment con BD y hacer replicas exactas y sincronías y que funcione todo bien, entonces los pod en vez de ejecutar MySQL secas, deberían ejecutar MySQL Cluster o Galera Cluster. Por eso hay que tener cuidado con lo que se escala.

VSCODE Y DEPLOYMENTS

- Con VSCODE podemos escalar, crear, eliminar, editar deployments sin problemas.

INTRODUCCIÓN A LOS SERVICIOS

- Imaginamos que tengo un deployment y dentro de el tengo 4 replicas de mis PODS (Pods que ejecutan Apache). Cada pod tiene una IP con la que me puedo conectar a sus contenedores, por lo que manejar un deployment demandar directa es complicado, ya que cada pod tiene na IP, puerto distintos. Por tanto si un cliente se quiere conectar al deployment, no puede usar una IP fija, ni un puerto fijo ni nada, porque cada POD tiene su IP su puerto, etc. Por lo que es complicado manejar un deployment de manera directa. Entonces, es aquí cuando aparecen los servicios, que actuan entre el cliente y el deployment, de forma que el servicio ofrece una IP fija, un nombre fijo y un puerto fijo, y el cliente se conecta a este servicio y ofrece los pods al cliente. Por tanto, con el servicio, el cliente solo debe conocer el nombre del servicio y su dirección IP para poder acceder al deployment o a la aplicación.

TIPOS DE SERVICIOS:

* ClusterIP: El servicio predefinido, el por defecto. Esto hace que un deployment o nuestros pods sean accesibles solo desde dentro del clúster. Por ejemplo sirve para BD internas y demás. Sirven para que los pods interactúen entre ellos de manera privada.

* NodePort: Accesible desde fuera del clúster. Hace que los deployments o sus pods sean accesibles mediante una IP externa desde fuera. Además, le añade un puerto nuevo, que será el puerto con el que se podrá acceder de forma externa a la aplicación. El más habitual para exponer servicios web.

* LoadBalancer: Accesible desde fuera del clúster. Hace que los deployments o sus pods sean accesibles desde el mundo exterior. Es muy similar al NodePort y al ClusterIP, pero solo es accesible desde fuera del clúster y está integrado con otros productos de terceros como Amazon, Azure, Google, etc. Se comporta igual que el NodePort solo que integra también el Cloud controller. Solo tiene

sentido crearlo si estoy en AWS, Azure, Google, etc trabajando con Kubernetes y trabajando en modo cloud nos dan una IP externa.

- Los servicios reconocen a los pods gracias a un selector que busca las labels de los pods.

CREAR UN SERVICIO DE TIPO NODEPORT

- kubectl create deployment apache1-d --image=httpd (Nos creamos un deploy que tendrá un pod de Apache de manera imperativa)

- kubectl expose deploy apache1-d --port=80 --type=NodePort (Me creo un servicio de tipo NodePort para poder acceder al deploy apache1-d desde el exterior. Con --name le hubiera puesto nombre al servicio)

- Al poner en el navegador externo la URL: http://192.168.59.100:32763, nos saldrá la página de Apache y de esta manera nuestro deployment es accesible desde le exterior a través del puerto 32763.

ESCALAR UN DEPLOY COMO REACCIONA EL SERVICIO

- Con "kubectl describe svc apache1-d" puedo ver toda la información del servicio y en la etiqueta Endpoints se ve la IP del pod del deploy apache1-d que está asociado a este servio Nodeport apache1-d.

- Ahora, si yo escalo el deploy apache1-d y le añado más replicas con: kubectl scale deploy apache1-d --replicas=3, entonces si yo hago kubectl describe svc apache1-d, en los EndPoints veré ahora 3IP's, correspondientes a las IP's de los pods del deploy que está exponiendo el servicio apache1-d.

- Si ahora yo por ejemplo borro uno de los 3 pods: kubectl delete pod apache1-d-b4f7d78c5-94m7q, automáticamente el replicaSet no recupera el pod

eliminado, sino que crea uno nuevo con otra IP. Así funcionan los deployments, cosa que no hacían los pods por si solos, que cuando un pod se elimina se vuelve a crear otro porque yo he indicado que deben haber un cierto numero de pods, que es mi estado ideal y kubernetes se intenta ajustar a eso intentando a toda costa que si le dije que hubieran 3 pods, y se elimina 1 pues inmediatamente crea otro pod para que si hayan 3 pods. Además, los endpoints del servicio de ese deploy que tiene los pods, también actualiza las IPS y asigna la nueva IP del pod que se creó tras la eliminación de uno de ellos.

CREAR UN SERVICIO DE TIPO LOADBALANCER

- Como sabemos, el LoadBalancer se usa en Cloud, pero en minikube se puede usar de igual manera un LoadBalancer, lo malo esquí se comporta igual que un servicio del tipo NodePort, ya que en minikube al trabajar el local no tiene posibilidad de trabajar en modo Cloud.

- kubectl create deployment apache2-d --image=httpd (Creamos un deploy de manera imperativa usando un pod de apache)

- kubectl expose deploy apache2-d --port=80 --type=LoadBalancer (Expongo el deploy apache2-d al exterior por medio del puerto 80, creando un servicio de tipo LoadBalancer)

- Como se podrá ver funciona igual que el NodePort ya que no estamos en cloud, y si hacemos kubectl get svc, veremos que el servicio apache2-d tiene en EXTERNAL-IP pending mientras que el svc Nodeport tendrá NONE. Por lo tanto, si hubiéramos estado en un entorno cloud s habría diferencias entre NodePort y LoadBlancer, de hecho, en cloud se usaría Loadbalancer en lugar de NodePort.

- Podemos acceder a través de un navegador externo con la URL: http://192.168.59.100:30628 y se mostrará la página principal de Apache.

CREAR UN SERVICIO DE TIPO CLUSTERIP

- Nuestros pods de nuestros deployments solo son accesibles desde dentro del clúster.

- Ejemplo, tengo 2 deployments, uno de ellos es un cliente Redis y el otro es el servidor Redis, y ambos se van a conectar o comunicar a través de un servicio de tipo ClusterIP.

- kubectl create deployment redis-master --image=redis (Deploy que tendrá un pod con el servidor Redis. Donde vamos a tener los datos)

- kubectl create deployment redis-cli --image=redis (Deploy que tendrá un pod con el cliente redis con el cuál nos vamos a conectar al servidor redis maestro (redes-master))

- kubectl expose deploy redis-master --port=6379 --type=ClusterIP (Servicio que sirve para conectarnos al maestro. Como queremos conectarnos al redis-master, tenemos que exponer el deploy redes-master solamente, ya que será al que queremos acceder desde el redes-cli)

- kubectl exec redis-cli-9cf5fbc5-m5wnt -it -- bash (Entro al pod del cliente de redis, es decir, entro a través de exec al contender redis-cli para conectarme desde ahí al Servicio clusterIP llamado redis-master que expone el servidor redis (redis-master))

- Desde la consola del pod del cliente redis: redis-cli-9cf5fbc5-m5wnt hago: "redis-cli -h redis-master", y me conecto al deploy maestro, al servidor redis, desde el cliente redis (redis-cli).

- Como vemos, puedo utilizar el servicio ClusterIP de manera privada de manera interna en el clúster y además el servicio me vale como nombre, ya que yo desde el cliente redis me he conectado al servidor redis a través del nombre redis-master. Aunque podría haber utilizado la IP en lugar del nombre también.

DASHBOARD SERVICIOS

- Desde el dashboard de minikube puedo ver los servicios, los endpoints, sus IPS, etc.

SERVICIO DE FORMA DECLARATIVA

- Vamos a hacer un ejemplo real, donde vamos a desplegar una aplicación escrita manualmente en HTML, CSS Y JS y que está basada en Apache y vamos a crear un servicio para poder acceder a la misma. Todo de forma declarativa. Para ello, partiremos de un DockerFile que contendrá una imagen APACHE y que añadirá la aplicación hecha a mano al directorio correspondiente de APACHE (/var/www/html), y luego, la subiremos a DockerHUB. Una vez la imagen esté en DockerHub, crearemos un deployment haciendo uso de esa imagen subida a DockerHub. Por último, se creará un servicio de tipo NodePort para poder acceder a la aplicación basada en Apache desde el exterior. Tanto el servicio como el Deployment, se crearán a través de un mismo fichero YAML.

1) Ejemplo DockerFile:

```
##Descargamos una versión concreta de UBUNTU, a través del tag
FROM ubuntu
MAINTAINER Apasoft Formacion "apasoft.formacion@gmail.com"
##Actualizamos el sistema
RUN apt-get update
##Instalamos HTTPD Apache 2
RUN apt-get install -y apache2
##Creamos un fichero index.html en el directorio por defecto de
apache
ADD web /var/www/html (Añadimos la carpeta web al directorio
/var/www/html)
##Exponemos el Puerto 80
```

```
EXPOSE 80
##Arrancamos Apache a través de CMD en segundo plano
CMD /usr/sbin/apachectl -D FOREGROUND
```

2) docker build -t danilooo99/web . (Construye esta imagen de Docker a partir del DockerFie que está en el directorio .)

3) docker login (Nos loqueamos en DockerHub)

4) docker push danilooo99/web (Subimos la imagen a DockerHub)

5) docker run --name=web1 -d -p 9090:80 danilooo99/web (Arrancamos el contenedor con la imagen creada y le ponemos de nombre web1, mapeando el puerto externo 9090 con el puerto 80 interno del contenedor)

6) http://localhost:9090/ (Comrpobamos que el contenedor Docker recien creado utilizándotelo la imagen Apache del DockerFIle funciona correctamente. Abrimos desde un navegador la siguiente URL y si todo va bien nos saldrá la web de prueba realizada)

7) Una vez comprobado que el contenedor funciona correctamente, paramos el contenedor: docker stop web1

8) EJEMPLO fichero con el deployment y el servicio en el mismo fichero YAML:

```
##############
# DEPLOYMENT
##############
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-d
```

```
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: web
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: apache
        image: danilooo99/web:latest
        ports:
        - containerPort: 80

--- # (GRACIAS A ESTO SE PUEDEN PONER DOS COMPONENTES EN EL MISMO
FICHERO YAML, GRACIAS a los ---)

#############
# SERVICIO
#############
apiVersion: v1
kind: Service
metadata:
  name: web-svc
  labels:
     app: web
spec:
  type: NodePort
  ports:
  - port: 80
```

```
    nodePort: 30002
    protocol: TCP
  selector:
     app: web # (Para que sea capaz de encontrar los pods o componentes que tengas como label: app: web. Como el deploy de arriba)
```

9) kubectl apply -f completo.yaml (Creamos el deployment con 2 replicas de pods de la app y el servicio NodePort para robar la misma haciendo uso del fichero completo.yaml. TODO JUNTO)

10) Accedemos a través de un navegador externo a la URL: http://192.168.59.100:30002/ y se mostrará la página realizada en HTML, JS y CSS, y además está ofrecida por un servicio NODEPORT y el puerto externo 3002 y en alta disponibilidad, ya que si se cae 1 pod, dará el servicio el otro POD y los RS se encargarán de que hayan 2 pods replicados siempre.

ENDPOINTS

- Los servicios apuntan a edpoints (que son los pods con los que trabajamos). Si yo hago kubectl describe sv web-svc, me salen los enpoints que serán los pods del deployments que está expuesto.

- El endpoint es el objeto que se asocia a una determinada IP y por lo tanto a un determinado POD y por lo tanto se asocia a un servicio y es capaz de mantener la relación entre el servicio y el POD.

- kubectl get endpoints (Me muestra todos los endpoints de los servicios que tengo)

- kubectl get endpoints web-svc -o wide (Me muestra mas información acerca del endpoint del servicio web-sec)

- kubectl describe endpoints web-svc (Me describe el endpoint del servicio web-svc)

- kubectl get endpoints web-svc -o yaml (Me da la informacion en formato YAML del endpoint del servicio web-svc)

VARIABLES DE ENTORNO DEL SERVICIO EN LOS PODS

- Cuando se crea un servicio, dentro de cada pod se activan o se crean una serie de variables de entorno que se integran directamente con Kubernetes.

- Entramos dentro de un pod del deploy nginx-d, por ejemplo así: kubectl exec -it nginx-d-7759cfdc55-km949 -- bash y una vez dentro del pod ejecuto: env, Y me aparecerán las variables de entorno del pod nginx-d-7759cfdc55-km949. Y podemos verlas y analizarlas.

VSCODE Y LOS SERVICIOS

- Como siempre, con VSCODE podemos manejar los servicios, deploys, endpoints de manera gráfica sin problemas.

ENUNCIADO DEL EJEMPLO DE APLICACIÓN PHP-REDIS CON SERVICIOS

- La aplicación es un frontend PHP y que va a trabajar contra una BD Redis. Vamos a tener un servidor Redis maestro (redis-master2) y va a tener un servicio del tipo ClusterIP. Además, vamos a tener una serie de pods (redis-slave) que se van a encargar de la parte de la lectura, van a devolver la información de la aplicación PHP y les van a devolver el dato, estos pods tendrán otro servicio del tipo ClusterIP. Luego tendremos frontales/frontend PHP que accederán en modo lectura al servicio redis-slave. Estos frontales van a leer de los slaves y van a

escribir en el redis-master y tendrán un servicio del tipo NodePort para que los clientes externos puedan acceder a la aplicación.

1) Creación del servidor maestro (redis-master). El redis-master, será un deployment llamado redis-master que tendrá tan solo 1 replica de sus pods, tendrá varios selectores para encontrar sus pods y usará una imagen redis para sus contenedores. Para ello usamos el siguiente fichero YAML:

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
spec:
  selector:
    matchLabels: # Busca todos los pods que tengan estas tres
LABELS
      app: redis
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: k8s.gcr.io/redis:e2e  # or just image: redis
```

```
        ports:
        - containerPort: 6379
```

- kubectl apply -f redis-master.yaml (Creamos el deployment haciendo uso del fichero YAML)

- Además, como he puesto selectores, si hago kubectl get all -l app=redis me saca los pods, deploys, rs, servicios, etc. que tengan la etiqueta app=redis

2) Creación del servicio de tipo ClusterIP llamado redis-master-svc del servidor redis maestro (redis-master), con algunas labels y selectors. Para ello usamos el siguiente fichero YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master-svc
  labels:
    app: redis
    role: master
    tier: backend
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

- kubectl apply -f redis-master-service.yaml (Creamos el servicio ClusterIP haciendo uso del fichero YAML)

- Con kubectl describe svc redis-master-svc, podemos ver el endpoint/pod del redis-master-svc. NOTA: Como no estamos poniendo a quien va dirigido el servicio, es decir, a que deploy, pod .. lo encuentra por las labels y selector.

- Podemos comprobar que este servicio ClusterIP funciona correctamente accediendo al contenedor del pod del deploy que está siendo ofrecido por el servicio redis-master-svc, de esta manera: kubectl exec -it redis-master-567f9dcd4d-x55mb -- bash. Una vez dentro podemos comprobar que llega al pod haciendo: ping redis-master-567f9dcd4d-x55mb y podemos comprobar que gracias al DNS interno de kubernetes podemos hacer un ping al servicio de redis-master: ping redis-master-svc y no contesta porque el servicio no tiene ping pero sí que resuelve el nombre y muestra su IP. De hecho, si instalamos nslookup en el pod, podemos hacer: nslookup redis-master-svc y nos mostrará la IP, el nombre y demás del servicio redis-master-svc.

3) Creación del redis Esclavo (redis-slave), el cuál es un deployment llamado redis-slave, con una serie de selectores, con 2 replicas de sus pods que contendrán un contenedor llamado slave que ejecuta la imagen redisslave de Google (que supongo que será una especie de API o algo así). Además, este deploy tendrá una variable entorno (GET_HOSTS_FROM) y tendrá el valor de "dns". Para ello usamos el siguiente fichero YAML:

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: redis-slave
  labels:
    app: redis
spec:
  selector:
    matchLabels:
```

```
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v3
        env:
        - name: GET_HOSTS_FROM
          value: dns
```

- kubectl apply -f redis-slave.yaml (Creamos el deployment haciendo uso del fichero YAML)

4) Creación del servicio de tipo ClusterIP llamado redis-slave-svc del servicio redis esclavo (redis-slave), con algunas labels y selectors. Para ello usamos el siguiente fichero YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave-svc
  labels:
```

```
    app: redis
    role: slave
    tier: backend
spec:
  type: ClusterIP
  ports:
  - port: 6379
  selector:
    app: redis
    role: slave
    tier: backend
```

- kubectl apply -f redis-slave-service.yaml (Creamos el servicio ClusterIP haciendo uso del fichero YAML)

- Podemos comprobar que este servicio ClusterIP funciona correctamente accediendo al contenedor del pod del deploy que está siendo ofrecido por el servicio redis-slave-svc, de esta manera: kubectl exec -it redis-master-567f9dcd4d-x55mb -- bash. Una vez dentro podemos comprobar que llega al pod haciendo (instalar dnsutils): nslookup redis-slave-svc y nos mostrará la IP, el nombre y demás del servicio redis-slave-svc.

5) Creación del frontend PHP, el cuál es un deoloyment llamado frontend, con selectores y etiquetas, que usa 3 replicas de sus pods y esos pods ejecutan una imagen frontend de Google que es un guestbook. Además, se crea una variable de entrono llamada GET_HOSTS_FROM (variable que existe) y que su valor es "dns". Para ello usamos el siguiente fichero YAML:

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: frontend
```

```
  labels:
    app: guestbook
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        env:
        - name: GET_HOSTS_FROM
          value: dns
```

- kubectl apply -f frontend.yaml (Creamos el deployment haciendo uso del fichero YAML anterior)

6) Creación del servicio frontend-svc de tipo NodePort para poder acceder desde el exterior a lam aplicación. El fichero YAML del servicio será el siguiente:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-svc
  labels:
```

```
    app: guestbook
    tier: frontend
spec:
  type: NodePort
  #type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: guestbook
    tier: frontend
```

- kubectl apply -f frontend-service.yaml (Creamos el servicio frontend-sec de tip NodePort haciendo uso del fichero YAML)

- Para probar el servicio puedo acceder desde un navegador externo a la siguiente URL: http://192.168.59.100:30766/ y se me verá el frontend de Guestbook que está siendo ofrecido por el servicio frontend-svc. Este frontend escribe en la BD que esta en el redis-master deploy y además coge los datos del redis-slave deploy.

- Otra comprobación: Accedemos al pod del redis-master con: kubectl exec -it redis-master-567f9dcd4d-x55mb -- bash y hacemos un: nslookup frontend-svc y nos contestará con la IP del servicio frontend-svc.

NAMESPACES. AGRUPAR NUESTROS OBJETOS EN KUBERNETES

- Son como particiones/un trozo de kubernetes que voy a hacer dentro de mi clúster para poner objetos. Los podría utilizar para albergar los objetos exclusivos de una aplicación o de un proyecto. Es una división lógica del clúster de kubernetes para poder dividir los objetos en distintas formas. Yo puedo tener varios namespaces y dentro de ellos crear objetos y trabajar con ellos. Se suelen utilizar cuando en nuestra infraestructura tenemos varias áreas como la de

desarrollo, despliegue, etc. y así puedo crear objetos que se llamen igual pero en NS diferentes.

- kubectl get ns (Nos permite ver todos los namespaces que hayan)

- kubectl get namespace default (Nos da info solo del ns default)

- kubectl describe ns default (Nos permite ver más información detallada del namespace default)

- kubectl get pods -n kube-system (Nos permite ver los objetos de tipo pod que tiene el namespace kube-system)

- Hay varios namespaces que se crean, y por defecto si no se le indica lo contrario todos los objetos se crean en el namespace default. También esta el namespace kube-system que se crea tras arrancar el clúster y lo utiliza el propio kubernetes. El namespace kube-public se crea de forma automática al instalar kubernetes y se utiliza por todos los usuarios. El namespace kuberentes-dashboard se activa cuando se activa el dashboard de minikube.

CREAR Y BORRAR NAMESPACES

- Vamos a crear un namespace para poder crear objetos dentro de él. Se pueden crear tanto de forma imperativa como de forma declarativa.

* *FORMA IMPERATIVA:*

- kubectl create namespace n1 (Creo de forma imperativa un namespace llamado n1)

** FORMA DECLARATIVA:*

- Fichero YAML:

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev1
  labels:
     tipo: desarrollo
```

- kubectl apply -f namespace.yaml

* *BORRAR UN NAMESPACE:*

- kubectl delete ns/n1

CREAR OBJETOS EN UN NAMESPACE

- Para no trabar por defecto en el namespace default, vamos a crear objetos en otro namespace, dev1 en este caso. En este caso vamos a crear un deployment llamado "elastic" con 2 replicas de sus pods y esos pods ejecutarán contenedores basados en la imagen elasticsearch. Además como estrategia de modificación se usará un Roll Update.

- El fichero YAML del deploy elástica es:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones
anteriores a 1.9.0
kind: Deployment
metadata:
```

```
  name: elastic
  labels:
    tipo: "desarrollo"
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: elastic
  replicas: 2 # indica al controlador que ejecute 2 pods
  strategy:
     type: RollingUpdate
  minReadySeconds: 2
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: elastic
    spec:
      containers:
      - name: elastic
        image: elasticsearch:7.6.0
        ports:
        - containerPort: 9200
```

- kubectl apply -f deploy_elastic.yaml --namespace=dev1 (Creo el deploy elastic en el namespace dev1 creado anteriormente y haciendo uso del fichero YAML)

- kubectl get deploy elastic -n dev1 (Vemos el deploy recién creado)

- kubectl get rs -n dev1 (Vemos los replicases del namespace dev1)

- kubectl get pods -n dev1 (Vemos los pods que hay en el namespace dev1)

- kubectl describe pod elastic-5c49c49458-7t8xs -n dev1 (Vemos la información de uno de los pods del deploy elastic y así puedo ver si hay errores y demás en la sección Events)

ESTABLECER UN NAMESPACE POR DEFECTO

- En el fichero .kube/config se encuentra la configuración de los clusters de kubernetes. Pues e este fichero, en la parte de los contextos, yo podrá modificar la línea namespace: default por otro nombre de namespace, rearrancar el clúster, y ya tendría otro namespace por defecto.

- Otra forma de ver ese fichero .kube/config es usando el comando: kubectl config view

- kubectl config set-context --current --namespace=dev1 (Cambia inmediatamente el namespace por defecto default por el ns dev1)

PONER CPU Y MEMORIA A LOS NAMESPACES

- Vamos a poner limites de memoria y CPU a nuestros pods pero dentro de un namespace. Para ello haremos uso de un nuevo objeto de kubernetes, los cuáles son los "LimitRange". Para ello crearemos un LimitRange llamado "recursos" que tendrá varios límites de memoria y de CPU. El fichero YAML del objeto LimitRange es:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: recursos
spec:
  limits:
```

```
  - default: # Límite por defecto
      memory: 512Mi
      cpu: 1
    defaultRequest: # Lo que voy a tener por defecto cuando se
crean los objetos
      memory: 256Mi
      cpu: 0.5
    max: # El máximo de CPU y memoria
      memory: 1Gi
      cpu: 4
    min: # El mínimo de CPU y memoria
      memory: 128Mi
      cpu: 0.5
    type: Container # Se puede poner a nivel de container o a
nivel de Pod
```

- kubectl create namespace n1 (Creamos un nuevo namespace llamado n1)

- kubectl apply -f limitex.yaml -n n1 (Creo el objeto LimitRange haciendo uso del fichero YAML para poner límites a los namespaces, ya que antes no tenían)

- kubectl delete -f limitex.yaml -n n1 (Eliminar el LimitRange del namespace n1)

DASHBOARD Y NAMESPACES

- Como siempre, desde dashboard de minikube se pueden hacer operaciones con los namespaces.

CONTROLAR LOS EVENTOS DENTRO DE UN NAMESPACE

- kubectl create namespace desarrollo (Creo un nuevo namespace llamado desarrollo)

- kubectl config set-context --current --namespace=desarrollo (Pongo el namespace desarrollo como namespace por defecto)

- kubectl get events --namespace desarrollo (Nos da los eventos del namespace desarrollo)

- kubectl get events --field-selector type="Warning" (Filtro los eventos que en columna type tengan de valor Warning)

- kubectl get events --field-selector reason="Pulled" (Filtro los eventos que en columna reason tengan de valor Pulled)

- kubectl get events --namespace desarrollo -w (Me lanza los eventos pero se queda esperando a que ocurran eventos en tiempo real. Es como el tail -f. -w == watch)

NAMESPACES Y VSCODE

- Como siempre en escode de manera gráfica se trabaja con los namespaces y hacer templates y demás.

ROLLING UPDATES

- Hacer updates a nuestras aplicaciones para mantener a nuestro cliente sin necesidad de que se pare. Cambiar una imagen, cambiar las características de un container ... todo eso son rolling updates. Estrategia que va modificando los

pods de manera evolutiva, es decir, poco a poco, para que siempre haya algún pod que esté dando el servicio y no haya pérdida de servicio.

MODIFICAR UNA APLICACIÓN - ROLLING UPDATES

NOTA: EL NAMESPACE POR DEFECTO EN ESTE MOMENTO ES EL NAMESPACE DESARROLLO.

** PARTE 1:*

- Tengo un deployment llamado nginx-d con 10 replicas de sus pods ejecutando cada pod un nginx, Se usará en primer lugar una estrategia de actualización "RollingUpdate". El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
  labels:
    estado: "1"
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: nginx
  replicas: 10 # indica al controlador que ejecute 10 pods
  strategy:
     type: RollingUpdate # Si yo le pongo de estrategia "Recreate" sería una estrategia que directamente borra todos los pods de la version antigua y los sustituye por la nueva, y en este caso el servicio se pierde temporalmente
```

```
  minReadySeconds: 2
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.17.8
        ports:
        - containerPort: 80
```

- kubectl apply -f deploy_nginx2.yaml (Creo el deploy haciendo uso del fichero YAML)

- kubectl rollout history deploy nginx-d (Me da el histórico del despliegue del deployment nginx-d. Y tendrá tan solo 1 revision)

- kubectl rollout history deploy nginx-d --revision=1 (Me da mucho mas detalle acerca de la revision 2 del histórico del deploy nginx-d)

- Ahora, obligo a hacer el update, ya que edito el fichero deploy_nginx2.yaml y le cambio el tag de la imagen y le pongo el tag, 1.7.9.

- kubectl apply -f deploy_nginx2.yaml (Modifico el deploy haciendo uso del fichero YAML, ya que he cambiado la version de nginx a ejecutar por los containers de los pods)

- kubectl rollout history deploy nginx-d (Me da el histórico del despliegue del deployment nginx-d. Y tendrá ahora 2 revisiones)

- kubectl rollout history deploy nginx-d --revision=2 (Me da mucho mas detalle acerca de la revision 2 del histórico del deploy nginx-d)

** PARTE 2:*

- El fichero deploy_nginx2.yaml lo modifico un poco y le añado 2 properties en la declaración "rollingUpdate", que son: maxUnavailable (Suma 1. Nos dice nunca seas superior a 11 pods) y maxSurge (Resto 1. Nunca seas inferior a 9 pods). Y luego la otra propiedad que hemos añadido es la de "minReadySecond : 10", que son el número de segundos mínimo que transcurren para crear un pod. Es decir que cada 10 segundos se crea un pod para no hacer tan agresivo el update:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones
anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
  labels:
    estado: "1"
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: nginx
  replicas: 10 # indica al controlador que ejecute 10 pods
  strategy:
     type: RollingUpdate
     rollingUpdate:
       maxUnavailable: 1
       maxSurge: 1
  minReadySeconds: 10
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

- Una vez hecho estas modificaciones en el YAML, también cambio el tag de la imagen de nginx a la 1.16.1, para que se produzca el update.

- Antes de hacer el apply, vamos a ver como estos updates surten efecto también en los servicios, ya que los endpoints de los servicios, si se producen updates en sus pods, también se actualizan.

- Para ello, creamos un servicio llamado nginx-svc de tipo NodePort. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  labels:
     app: nginx
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 31000
    protocol: TCP
  selector:
     app: nginx
```

- kubectl apply -f nginx-svc.yaml (Creamos el servicio haciendo uso del fichero YAML)

- kubectl describe svc nginx-svc (Vemos los endpoints del servicio nginx-svc)

- Ahora, cambio el tag de la imagen de nginx a la 1.16.1, como se he comentado anteriormente.

- kubectl apply -f deploy_nginx2.yaml (Modificamos el deploy con la nueva version de nginx)

- kubectl describe svc nginx-svc (Volvemos a ver los endpoints del servicio nginx-svc y vemos como ahora han cambiado, ya que se han actualizado)

- kubectl rollout history deploy nginx-d (Ahora vemos que hay una version 3)

- kubectl rollout history deploy nginx-d --revision=3 (Me da mucho mas detalle acerca de la revision 3 del histórico del deploy nginx-d)

DESHACER CAMBIOS - ROLLING BACK

- Volver atrás. Por ejemplo, si un update de un deploy no me ha funcionado bien o algo del estilo, y quiero volver al estado anterior en el que estaba ese deploy.

- Vamos a modificar otra vez el fichero deploy_nginx2.yaml y vamos a poner una imagen que no existe, por ejemplo un nginx 1.16.hh1:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones
anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
  labels:
```

```
    estado: "1"
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: nginx
  replicas: 10 # indica al controlador que ejecute 2 pods
  strategy:
     type: RollingUpdate
     rollingUpdate:
       maxUnavailable: 1
       maxSurge: 1
  minReadySeconds: 10
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.hh1
        ports:
        - containerPort: 80
```

- kubectl apply -f deploy_nginx2.yaml (Modificamos el deploy con la nueva version de nginx, que no existe, dará fallo)

- kubectl rollout status deploy nginx-d (Nos da info acerca de lo que se va haciendo en los rollouts)

- Viendo los pods, los deploys, los rs, los revisions y demás, me doy cuenta de que algo ha fallado, y si yo quiero volver al estado anterior, hago un roll-back.

- kubectl rollout undo deployment nginx-d --to-revision=2 (Hacemos un rollback a la revision 2, es decir, volvemos al estado del deploy en la revision2, que sabemos nosotros que en ese estado todo iba bien)

- kubectl rollout history deploy nginx-d (Ahora vemos que hay una nueva revision, la 5, pero que ha movido la revision 2, la ha eliminado, y ha puesto la 5)

ESTRATEGIA RECREATE

- Estrategia que directamente borra todos los pods de la version antigua y los sustituye por la nueva, y en este caso el servicio se pierde temporalmente

- Ahora, cambiamos la estrategia RollingUpdate del fichero deploy_nginx2.yaml y sus propiedades, y la cambiamos por una estrategia Recreate. El fichero YAML quedaría así:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
  labels:
    estado: "1"
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: nginx
  replicas: 10 # indica al controlador que ejecute 2 pods
  strategy:
     type: Recreate # Estrategia Recreate
  minReadySeconds: 10
  template:   # Plantilla que define los containers
```

```
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:1.16.1
      ports:
      - containerPort: 80
```

- kubectl apply -f deploy_nginx2.yaml (Modificamos el deploy con la nueva estrategia, Recreate)

- kubectl rollout history deploy nginx-d (Ahora vemos que hay una nueva revision, la 6)

- Ahora, para que se aplique el recreate, vamos a cambiar de nuevo el tag de la version de nginx del fichero deploy_nginx2.yaml a la 1.17.8.

- kubectl apply -f deploy_nginx2.yaml (Modificamos el deployment con la nueva version de nginx)

- Si hacemos un kubectl get pods, vemos como borra todos los pods y los recrea de nuevo con sus modificaciones. Esto es un poco brusco, pero podría venir bien cuando tengo un problema muy grave y no quiero esperar a que se vayan apagando uno y encendiendo otros y no me importa que la aplicación este temporalmente caída.

- kubectl rollout history deploy nginx-d (Ahora vemos que hay una nueva revision, la 7)

- kubectl rollout history deploy nginx-d --revision 7 (Vemos con detalle la nueva revision 7)

VARIABLES

- Objeto mas sencillo que podemos utilizar para conectar nuestro exterior o inyectarle datos y propiedades a nuestros pods.

- Tenemos el siguiente Pod llamado var-ejemplo que ejecuta una imagen node-hello. Este pod tiene una serie de variables de entorno, que van a ir a parar dentro del contenedor y voy a poder acceder a ellas dentro de la aplicación. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: var-ejemplo
  labels:
    app: variables
spec:
  containers:
  - name: contenedor-variables
    image: gcr.io/google-samples/node-hello:1.0
    env:
    - name: NOMBRE # Tiene dos variables de entorno: NOMBRE=CURSO
DE KUBERNETES y PROPIETARIO=Apasoft Training
      value: "CURSO DE KUBERNETES"
    - name: PROPIETARIO
      value: "Apasoft Training"
```

- kubectl apply -f var1.yaml (creamos el pod haciendo uso del fichero YAML)

- kubectl exec -it pod/var-ejemplo -- bash (Entramos al contenedor del pod var-ejemplo) y ejecutamos en él: printenv para ver las variables de entorno que hay, y podremos ver las variables de entorno definidas en el YAML anterior: CURSO DE KUBERNETES y Apasoft Training.

EJEMPLO CON MYSQL

- Creamos un deployment llamado mysql-deploy con 1 replica de sus pods y que ejecuta una imagen mysql, el contenedor de su pod. Además, tiene una serie de variables de entorno que configuran la contraseña del usuario root, la contraseña del usuario de MySQL, el usuario de MySQL y la BD a crear. El fichero YAML es el siguiente:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deploy
  labels:
    app: mysql
    type: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
      type: db
  template:
    metadata:
      labels:
        app: mysql
        type: db
    spec:
      containers:
        - name: mysql57
          image: mysql:5.7
          ports:
            - containerPort: 3306
              name: db-port
```

```
env:
  - name: MYSQL_ROOT_PASSWORD
    value: kubernetes
  - name: MYSQL_USER
    value: usudb
  - name: MYSQL_PASSWORD
    value: usupass
  - name: MYSQL_DATABASE
    value: kubernetes
```

- kubectl apply -f mysql.yaml (creamos el deployment haciendo uso del fichero YAML)

- kubectl exec -it deploy/mysql-deploy -- bash (Entro al contenedor de MySQL) y ejecuto dentro de él: printenv (Y me salen todas las variables de entorno de MySQL que se han configurado)

- Una vez dentro del contenedor MySQL del pod del deploy mysql-deploy, ejecuto: "mysql -u usudb -pusupass kubernetes" y accedo al cliente de Mysql con el usuario, contraseña y BD especificados en las variables de entorno del fichero YAML.

CONFIGMAPS

- Cuando el número de variables de entorno es muy grande, se puede hacer complicado meter todas las variables de entorno en el fichero YAML y lo ideal es tener un fichero adicional del cual cargar los datos y que se pudieran incorporar dentro del environment de un contenedor de un pod. Este objeto existe y se denomina ConfigMap, que son ficheros que nos permiten incorporar los conjuntos de propiedad: valor de una forma muy sencilla y así automáticamente se incorporan al contenedor.

** FORMA IMPERATIVA:*

- kubectl create configmap cf1 --from-literal=usuario=usu1 --from-literal=password=pass1 (Estoy creando un fichero llamado cf1 que contiene las propiedades USUARIO=usu1 y PASSWORD=pass1)

- kubectl get cm (Nos muestra los configmaps existentes)

- kubectl describe cm cf1 (Nos muestra mas datos, como las variables configuradas)

- kubectl get cm cf1 -o yaml (Nos muestra el fichero YAML del ConfigMap)

CONFIGMAPS DESDE FICHEROS CON FROM-FILE

- Es un poco engorroso pasar las variables o crear confirmas de forma imperativa, por eso es mejor hacerlo de forma declarativa. Con from-file podemos cargar multiple contenido, pero no sirve para cargar variables de entorno.

** FORMA IMPERATIVA:*

- *FICHERO datos_mysql.properties:*

```
MYSQL_ROOT_PASSWORD=kubernetes
MYSQL_USER=usudb
MYSQL_PASSWORD=usupass
MYSQL_DATABASE=kubernetes
```

- kubectl create cm datos-mysql --from-file=datos_mysql.properties (Creo un configmap llamado datos-mysql y le importo toda la información de las variables que contiene el fichero datos_mysql.properties. Al usar from-file lo lees todo como un único valor. Eso es lo malo de usar from-file)

- Para ver un ejemplo, creamos un pod que tiene un pequeño Linux, llamado pod1. Y además se ha definido una variable de entorno llamada "DATOS_MYSQL" y su valor viene del configmap "datos-mysql" que recoge los datos del fichero "datos_mysql.properties". El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
spec:
  containers:
    - name: test-container
      image: gcr.io/google-samples/node-hello:1.0
      env:
        # Define the environment variable
        - name: DATOS_MYSQL
          valueFrom:
            configMapKeyRef:
              name: datos-mysql
              key: datos_mysql.properties
  restartPolicy: Never
```

- kubectl apply -f pod1.yaml (Creamos el pod haciendo uso del fichero YAML)

- kubectl exec -it pod1 -- bash (Entramos al contenedor del pod pod1) y ejecutamos "printenv". Se verá la variable configurada llamada "DATOS_MYSQL" y su valor es las 4 variables mysql_root, mysql_user, mysql_password, mysql_database. No tiene valor único, sino un conjunto de valores y esos se debe por haber creado el configmap con --from-file

CARGAR VARIABLES CON CONFIGMAP CON FROM-ENV-FILE

- Con from-file teniamos un inconveniente, ya que lo metía todo en un único valor. Ahora veremos como cargar todo esa información como variables de entorno individuales.

** FORMA IMPERATIVA:*

- FICHERO *datos_mysql.properties:*

```
MYSQL_ROOT_PASSWORD=kubernetes
MYSQL_USER=usudb
MYSQL_PASSWORD=usupass
MYSQL_DATABASE=kubernetes
```

- kubectl create cm datos-mysql-env --from-env-file datos_mysql.properties (Crea un configmap llamado datos-mysql-env pero ahora al usar --from-env-file los carga como variables de entorno)

- Para ver otro ejemplo, creamos un pod llamado pod2. Además, ahora se crearán varias variables de entorno en el contenedor del pod pod2, que son las especificadas en el fichero datos_mysql.properties. Para ello se le especifica el configmap que tiene esas variables configuradas, en este caso datos-mysql-env. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod2
spec:
  containers:
    - name: test-container
      image: gcr.io/google-samples/node-hello:1.0
      envFrom:
```

```
    - configMapRef:
        name: datos-mysql-env
  restartPolicy: Never
```

- kubectl apply -f pod2.yaml (Creamos el pod haciendo uso del fichero YAML)

- kubectl exec -it pod2 -- bash (Entramos al contenedor del pod pod2) y ejecutamos printenv y vemos las 4 variables de forma separada, como variables de entorno.

EJEMPLO - CONFIGURAR UN MYSQL CON CONFIGMAPS

* *FORMA DECLARATIVA*:

1) Tenemos un fichero llamado datos-mysql-env.yaml, que crea un ConfigMap llamado datos-mysql-env2 en el namespace default con 4 variables de entorno. El fichero YAML es el siguiente:

```
apiVersion: v1
data:
  MYSQL_DATABASE: kubernetes
  MYSQL_PASSWORD: usupass
  MYSQL_ROOT_PASSWORD: kubernetes
  MYSQL_USER: usudb
kind: ConfigMap
metadata:
  name: datos-mysql-env2
  namespace: default
```

- kubectl apply -f datos-mysql-env.yaml (Creamos el configmap haciendo uso del fichero YAML)

2) Ahora, tenemos un fichero YAML que crea un deployment llamado mysql-deploy2, con 1 replica de sus pods y que carga 4 variables de entorno del confirma datos-mysql-env2 del paso 1. El fichero YAML es el siguiente:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deploy2
  labels:
    app: mysql
    type: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
      type: db
  template:
    metadata:
      labels:
        app: mysql
        type: db
    spec:
      containers:
        - name: mysql57
          image: mysql:5.7
          ports:
            - containerPort: 3306
              name: db-port
          env: # Es mejor usar el envFrom como en la sección anterior. Lo que así se puede usar para si las variables del
```

```
fichero properties o del configmap se llaman distintas a las
originales de MySQL
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                configMapKeyRef:
                  name: datos-mysql-env
                  key: MYSQL_ROOT_PASSWORD

            - name: MYSQL_USER
              valueFrom:
                configMapKeyRef:
                  name: datos-mysql-env
                  key: MYSQL_USER

            - name: MYSQL_DATABASE
              valueFrom:
                configMapKeyRef:
                  name: datos-mysql-env
                  key: MYSQL_DATABASE

            - name: MYSQL_PASSWORD
              valueFrom:
                configMapKeyRef:
                  name: datos-mysql-env
                  key: MYSQL_PASSWORD
```

- kubectl apply -f dmysql.yaml (Creamos el deployment haciendo uso del fichero YAML)

- kubectl exec -it mysql-deploy2-68f5f96677-4hp69 -- bash (Entramos al contenedor del pod del deploy mysql-deploy2) y ejecutamos printenv y vemos las 4 variables de forma separada, como variables de entorno.

- Si hacemos mysql -u usudb -p kubernetes, entro a la BD con el cliente de MySQL.

CONFIGMAPS Y VOLUMENES

1) Creamos un configmap llamado config-volumen, con 2 variables: ENTORNO y VERSION. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-volumen
  namespace: default
data:
  ENTORNO: "desarrollo"
  VERSION: "1.0"
```

- kubectl apply -f configmap.yaml (Creo el configmap haciendo uso del fichero YAML)

2) Ahora, creamos un pod llamado pod3 con una imagen de busybox y ejecutamos el comando sleep 1000000 para hacer una espera en el contenedor. Además, tiene los puntos de montaje dentro del contenedor del volumen con el nombre de volumen-config-map y defino los volúmenes con el configmap confié-volumen configurado anteriormente. El fichero YAML es:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod3
```

```
spec:
  containers:
    - name: contenedor1
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "sleep 1000000" ]
      volumeMounts:
      - name: volumen-config-map
        mountPath: /etc/config-map
  volumes:
    - name: volumen-config-map
      configMap:
        name: config-volumen
  restartPolicy: Never
```

- kubectl apply -f pod3.yaml (Creo el pod haciendo uso del fichero YAML)

3) kubectl exec -it pod3 -- sh (Entro al contenedor del pod pod3) y ejecuto env: no veremos las variables, porque las hemos configurado como volúmenes. Entonces, habría que ir a /etc/config-map y las veríamos ahí y podríamos hacer un "cat" de ellas para ver su valor. Nos hemos traído las variables en forma de fichero, gracias a los volúmenes.

SECRETS

- Muy parecidos a los configmaps, pero la información la guarda protegida o con cierta codificación. Es un recurso de kuberentes, para guardar información confidencial, como contraseñas, tokens de autentificaron, claves SSH, etc. Los secrets normalmente van asociado a un pod.

Tipos de Secrets:

* Opaques: Tipo por defecto, contiene cualquier información que queramos incluir y proteger. Muy parecidos a los confirmas.

* Service account token: Almacena un token que identifica un service account. Es una identidad que nos permite ejecutar dentro de un pod un determinado proceso.

* Docker config: Almacena las credenciales para acceder a un registro privado de Docker, como un repo, una imagen, etc.

* Basic authentication: Se utilizan para la autenticación básica, y contiene dos propiedades: usuario y password.

* SSH: Para almacenar las claves SSH.

* TLS: Guarda un certificado y la clave asociada que se utilizan para TLS.

* Bootstrap: Tokens especiales y se utilizan para el proceso de bootstrap del nodo.

SECRETS DE TIPO OPACO (POR DEFECTO)

** FORMA IMPERATIVA:*

- kubectl create secret generic passwords --from-literal=pass-root=kubernetes --from-literal=pass-usu=kubernetes (Creo de forma imperativa un secreto de tipo genérico (opaco), llamado passwords con 2 variables: pass-root y pass-usu)

- kubectl get secrets (Me devuelve los secrets que existen)

- Además, si hacemos un kubectl get secret passwords -o yaml (Veremos en el YAML que las variables esta encriptadas).

- Creamos un configmap llamado datos-mysql-env2 que tiene dos variables MYSQL_DATABASE y MYSQL_USER. El fichero datos_mysql_env2.yaml contiene lo siguiente:

```
apiVersion: v1
data:
  MYSQL_DATABASE: kubernetes
  MYSQL_USER: usudb
kind: ConfigMap
metadata:
  name: datos-mysql-env2
  namespace: default
```

- kubectl apply -f datos_mysql_env2.yaml (Creamos el configmap datos_mysql_env2 haciendo uso de un fichero YAML)

- kubectl get secret (Me devuelven los secrets)

- kubectl get cm (Me devuelven los config maps)

- Creo un deploy llamado mysql-deploy3 que tiene 1 replica y tiene 4 variables de entorno, donde le usuario root y contraseña son cogidas del secret passwords creado anteriormente y el usuario normal y la base de datos son cogidas del configmap datos-mysql-env2 creado con anterioridad. El fichero mysql2.yaml es el siguiente:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-deploy3
  labels:
    app: mysql
    type: db
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
      type: db
  template:
    metadata:
      labels:
        app: mysql
        type: db
    spec:
      containers:
        - name: mysql57
          image: mysql:5.7
          ports:
            - containerPort: 3306
              name: db-port
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: passwords
                  key: pass-root

            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: passwords
                  key: pass-usu

            - name: MYSQL_USER
              valueFrom:
```

```
      configMapKeyRef:
        name: datos-mysql-env2
        key: MYSQL_USER

  - name: MYSQL_DATABASE
    valueFrom:
      configMapKeyRef:
        name: datos-mysql-env2
        key: MYSQL_DATABASE
```

- kubectl apply -f mysql2.yaml (Creamos el deployment mysql2 haciendo uso de un fichero YAML)

- kubectl exec -it mysql-deploy3-5bf44bdf89-qx7bh -- bash (Entro dentro del pod del deploy mysql-deploy3) y ejecuto printenv: y veo las 4 variables de entorno configuradas anteriormente.

- De esta manera he conseguido inyectar de manera codificada el user root y su contraseña en el contenedor.

SECRETS Y FICHEROS

- Ahora, vamos a inyectar un fichero TXT con unas frases dentro del contenedor, pero de manera codificada. El fichero TXT contiene lo siguiente:

```
Esto es un ejemplo de secrets
incorporaos desde fichero
dentro de un contenedor
```

- kubectl create secret generic datos --from-file=datos.txt (Creo un secret llamado datos de tipo Opaco, que codificará las frases del fichero datos.txt)

- kubectl get secret datos -o yaml (Vemos como el fichero datos.txt esta codificado)

- Creamos un pod de forma declarativa llamado pod4, que ejecuta el comando sleep 1000000 de ubuntu además hace uso de una Variable DATOS que valdrá el fichero datos.txt. El YAML del pod pod4 es:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod4
spec:
  containers:
    - name: test-container
      image: ubuntu
      command: [ "/bin/sh", "-c", "sleep 1000000" ]
      env:
        - name: DATOS
          valueFrom:
            secretKeyRef:
              name: datos
              key: datos.txt
  restartPolicy: Never
```

- kubectl apply -f pod1.yaml (Creo el pod haciendo uso del fichero YAML)

- kubectl exec -it pod4 -- bash (Entro dentro del contenedor del pod pod4) y ejecuto: printenv: y me saldrá la variable DATOS con el contenido del fichero datos.txt y desde dentro del contenedor si lo puedo ver de forma clara.

SECRETS DECLARATIVOS

** EJEMPLO 1:*

- Vamos a crear un secret de forma declarativa llamado secreto1, que tiene dos variables: usuario1 y usu1-pass. El fichero YAML contiene lo siguiente:

```
apiVersion: v1
kind: Secret
metadata:
  name: secreto1
type: Opaque
data:
  usuario1: dXN1MQ== # Esto se ha generado así: echo -n "usu1" | base64
  usu1-pass: cGFzc3dvcmQtdXN1MQ== # Esto se ha generado así: echo -n "password-usu1" | base64
```

- kubectl apply -f secreto1.yaml (Creo el secreto haciendo uso del fichero YAML)

** EJEMPLO 2:*

- Vamos a crear un secret de forma declarativa llamado secreto2, que tiene dos variables: usuario2 y usu2-pass. El fichero YAML contiene lo siguiente:

```
apiVersion: v1
kind: Secret
metadata:
  name: secreto2
type: Opaque
stringData: # Al poner StringData le estoy pidiendo a Kubernetes que me encripte esta información y se la pase al contenedor
```

```
  usuario2: 'usu2'
  usu2-pass: 'password-usu2'
```

- kubectl apply -f secreto2.yaml (Creo el secreto haciendo uso del fichero YAML)

** CREACIÓN DEL POD:*

- Creamos un pod llamado pod5 que ejecuta un ubuntu y el comando sleep 10000000. Además usa los dos secretos, el secreto 1 y el secreto 2. El fichero YAML del pod es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod5
spec:
  containers:
    - name: test-container
      image: ubuntu
      command: [ "/bin/sh", "-c", "sleep 1000000" ]
      envFrom:
        - secretRef:
            name: secreto1
        - secretRef:
            name: secreto2
  restartPolicy: Never
```

- kubectl apply -f pod5.yaml (Creo el pod haciendo uso del fichero YAML)

- kubectl exec -it pod5 -- bash (Entro al contenedor del pod pod5) y ejecuto printenv: y veré las 4 variables configuradas con los 2 secretos

SECRETS Y VOLÚMENES

- Asignar a un secret un directorio, donde cada componente del secret será un directorio.

- Me creo un secreto llamado secreto-volumen con dos propiedades. Me creo un fichero YAML con lo siguiente:

```
apiVersion: v1
kind: Secret
metadata:
  name: secreto-volumen
type: Opaque
stringData: # Al poner StringData le estoy pidiendo a Kubernetes
que me encripte esta información y se la pase al contenedor
  DATO1: "dato1"
  DATO2: "dato2"
```

- kubectl apply -f secreto-volumen.yaml (Creo el secret haciendo uso del fichero YAML)

- Ahora me creo un Pod llamado pod6 que usa una imagen ubuntu. Además, usa volúmenes y se indica el punto de montaje y el volumen, además de que ese volumen haga uso del secreto. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod6
spec:
  containers:
    - name: contenedor1
      image: ubuntu
      command: [ "/bin/sh", "-c", "sleep 1000000" ]
```

```
    volumeMounts:
    - name: volumen-secretos
      mountPath: /tmp/datos
  volumes:
    - name: volumen-secretos
      secret:
        secretName: secreto-volumen
  restartPolicy: Never
```

- kubectl apply -f pod6.yaml (Creo el pod haciendo uso del fichero YAML)

- kubectl exec -it pod6 -- bash (Entro al contenedor del pod6) y ejecuto printenv: y no veo nada, pero en el directorio /tmp/datos, tengo 2 ficheros que son las dos variables DATO1 Y DATO2 de los secrets. Puedo hacer un cat /tmp/datos/DATO1 y del DATO2 y veré su valor.

SECRETS DE TIPO DOCKER PARA ACCEDER A UN REGISTRO DE IMÁGENES PRIVADO

- Creamos un pod llamado ejemplo que usa una imagen de un repo privado de DockerHUb. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: ejemplo
spec:
  containers:
    - name: ejemplo
      image: danilooo99/web
```

- Si yo ahora un kubectl apply -f pod-secret-docker.yaml, me crea el pod, pero al hacer kubectl get pods me saldrá ErrImagePull debido a que la imagen proviene de un repo privado.

- Para evitar esto, tengo que crearme un secret de tipo docker-registry, que lo llamaré midocker y le legré que especificar: usuario, contraseña, email y server de docker. Para todo ello ejecuto:

* kubectl create secret docker-registry midocker --docker-server=docker.io --docker-username=danilooo99 --docker-password=gomamilan --docker-email=danilo.informatica.99@gmail.com

- Una vez generada las credenciales con el secret que acabo de crear, borro elmpod que cree anteriormente: kubectl delete pod ejemplo

- Ahora, para indicarle al pod que use las credenciales del secret, le añado la cláusula ImagePullSecrets con el nombre del secret de las credenciales, al fichero YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: ejemplo
spec:
  containers:
    - name: ejemplo
      image: danilooo99/web
  imagePullSecrets:  # Clausula para usar el secret creado anteriormente
    - name : midocker
```

- kubectl apply -f pod-secret-docker.yaml (Vuelvo a crear el pod ejemplo, pero ahora con la claúsla ImagePullSecrets)

- Ahora si que está corriendo el pod, y se ha descargado la imagen, ya que gracias al secret se ha logueado y se ha podido descargar la imagen desde el repo privado.

VSCODE - CONFIGMAPS Y SECRETS

- Como siempre, en VSCODE se puede trabajar con confirmas y cons secrets.

KUBECONFIG

- Fichero de configuración del clúster. Kubeconfig es un fichero en formato YAML que necesitamos para determinar como me conecto a un determinado cluster de kubernetes, con que credenciales, donde se encuentra el servidor, etc. Este fichero lo usa la herramienta kubectl. Puedo tener varios ficheros kubeconfig. La ubicación de este fichero se puede configurar mediante la opción --kubeconfig del comando kubectl o bien mediante una variable de entrono como $KUBECONFIG. Además, si no usa ninguno de los 2 métodos anteriores, por defecto este fichero se encuentra en ~/.kube/config en Linux y en MAC, y en Windows en %USERPROFILE%\.kube\config.

- Este fichero kubeconfig contiene: certificados, el servidor al que conectarse, el nombre, el usuario con el que me conecto, un token de usuario, etc.

- El fichero está estructurado en bloques o secciones: clusters (contiene los endpoints, la URL para el servidor api-server, etc. COMANDO: kubectl config set-cluster), users (las credenciales para autentificarse en el cluster de kuberentes, sus certificados y demás. COMANDO: kubectl config set-credentials), contexts (una fila que contiene el cluster, el usuario y el namespace con el que quiero trabajar COMANDO: kubectl config set-context), etc.

- kubectl config view (Ver la configuration actual)

EJEMPLO DE FICHERO KUBECONFIG EN MINIKUBE

- El fichero ~/.kube/config se ha configurado mediante minikube, ya que es el cluster de kubernetes que se está usando. Y en el aparecen los cluster de minikube, sus contextos, etc. Podemos ver este fichero también con el comando: kubectl config view.

VER LA CONFIGURACION Y CAMBIAR EL CONTEXTO ACTUAL

- kubectl config view (Nos muster la configuration del cluster. Nos muestra el contenido del fichero ~/.kube/config. Con el flag -o puedo poner formato YAML, json ...).

- kubectl config current-context (Nos muster que cluster se está usando actualmente. Es igual que minikube profile)

- kubectl config use-context ClusterDesarrollo (Cambio el contexto, de manera que ahora se usará el clúster "ClusterDesarrollo". Es igual que hacer minikube profile ClusterDesarrollo)

AÑADIR LOS DATOS DE UN CLUSTER

- kubectl config set-cluster clusterDesarrollo --server=https://1.2.3.4 (Crea un cluster llamado clusterDesarrollo y modifica el servidor del cluster clusterDesarrollo)

- kubectl --kubeconfig=config_alternativo config set-cluster clusterDesarrollo --server=https://1.2.3.4 (Crea un nuevo fichero llamado config_alternativo en el directorio ~/.kube e incorpora el cluster clusterDesarrollo que se ha creado. Esto puede servir, porque se puede dar el caso que yo quiera un fichero de configuración para el cliente A y un fichero de configuración para el cliente B, y que estén totalmente separados.)

- kubectl config set-cluster clusterDesarrollo --certificate-authority=/home/dev/cert.crt (Añado una autoridad de certification ak cluster clusterDesarrollo recién creado)

AÑADIR USUARIOS Y CREDENCIALES

- kubectl config set-credentials usu1 --username=usu1 --password=c2VjcmV0Cg== (Añado un usuario y su contraseña a un cluster)

- kubectl config --kubeconfig=config_alternativo set-credentials usu1 --username=usu1 --password=c2VjcmV0Cg== (Añado un usuario y su contraseña a un cluster, pero en este caso se lo añado al fichero de configuración config_alternativo)

- kubectl config set-credentials usu2 --client-certificate=/home/dev1/fichero.crt (Añadimos el certificado al usuario usu2)

- kubectl config set-credentials usu2 --client-key=/home/dev1/fichero.key (Añadimos la clave al usuario usu2)

AÑADIR CONTEXTOS

- kubectl config set-context context-clusterDesarrollo --cluster=clusterDesarrollo --user=usu1 --namespace=desarrollo (Añade un contexto llamado context-clusterDesarrollo al fichero de configuración config y nos dice que esta formado por el namespace desarrollo, por el usuario usu1 y se usará el cluster clusterDesarrollo)

- kubectl config --kubeconfig=config_alternativo set-context context-clusterDesarrollo --cluster=clusterDesarrollo --user=usu1 --namespace=desarrollo (Añade un contexto llamado context-clusterDesarrollo al fichero de configuración llamado config_alternativo y nos dice que esta formado

por el namespace desarrollo, por el usuario usu1 y se usará el cluster clusterDesarrollo)

CREAR UN CLUSTER REAL CON KUBEADM - INSTALACION DE LAS HERRAMIENTAS EN LOS SERVER VIRTUALES Y PREPARACIÓN DE LAS MVS

* *REQUISITOS:*

- Para ello dispondremos de 3 MV de VMware Workstation/VirtualBox/KVM con 2,5 GB de memoria, 2 CPUS, 20 GiB de espacio en disco, y una interfaz de red en modo adaptador puente. (TODOS LOS NODOS)

- Cada MV ejecutará un S.O Red Hat 9.1/CentOS7. (TODOS LOS NODOS)

- Tener instalado en cada MV kubectl y algún container run time (docker, podían, etc). (TODOS LOS NODOS)

** PARA HACER LOS 3 PASOS SIGUIENTES (kubectl, kubeadm y kubelet) MIRAR:* "https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/"

- Instalar kubeadm. (TODOS LOS NODOS)

- Instalar kubelet. (TODOS LOS NODOS)

- Instala kubectl. (TODOS LOS NODOS)

- Trabajar como usuario normal pero que esté registrado en el sudores. (TODOS LOS NODOS)

- Ejecutar swapoff -a. (TODOS LOS NODOS)

- Desactivar el firewall. (TODOS LOS NODOS)

- Registrar la IP de cada MV en el fichero /etc/hosts apuntando al hostname de la MV. (TODOS LOS NODOS)

- Tener el SWAP desactivado, ya que kubernetes se lleva mal con SWAP. Para ello: swapoff -a o de forma permanente comentándola en el fichero fstab. (TODOS LOS NODOS)

- Borrar todo el contenido del fichero /etc/containerd/config.toml y añadir (https://serverfault.com/questions/1074008/containerd-1-4-9-unimplemented-desc-unknown-service-runtime-v1alpha2-runtimese):. Después de añadir eso: systemctl restart containerd en todos los nodos. (TODOS LOS NODOS)

```
[plugins."io.containerd.grpc.v1.cri"]
  systemd_cgroup = true
```

- TODOS LOS PASOS ANTERIORES HACERLO CON 1 SOLA MAQUINA, TRAS ELLO CLONARLA 2 VECES PARA LOS DEMAS NODOS.

- Tras ello, verificar la IP y MAC de cada MV y ver que sean diferentes y que se comuniquen entre ellas y tengas acceso a internet.

- Modificar el /etc/hosts y HOSTNAME de las MV clonadas.

MODELO DE RED DE KUBERNETES

- El modelo de red de Kubernetes es ambiguo, kubernetes no intenta encargarse de la red realmente, sino intenta que el container run time que está por debajo se encargue junto con la infraestructura. Para ello se usará varios plugins. Esto permite indicar cómo se relacionan los contenedores, los pods y los servicios entre ellos. Entre las características básicas de este modelo de red encontramos que:

* cada POD tiene su IP

* los contenedores que están dentro de un pod comparten la misma IP y se pueden comunicar entre ellos sin problemas

* un pod se puede comunicar con cualquier otro pod dentro del cluster utilizando el direccionamiento IP de los PODS sin necesidad de usar NAT.

* Podemos mantener un filtro para que los pods solo se comuniquen con lo que nosotros deseemos.

- Todo esto, permite que cada POD pueda ser visto como si fuera una mV o maquina física, lo que hace fácil la migración de aplicaciones a un entorno de contenedores.

- Para implementar la red dentro de kubernetes se utilizan plugins de distintos proveedores. Estos plugins se implementan mediante CNI (container network interface) y se encarga el container runtime de cada nodo.

- Hay varios tipos de plugins CNI:

* Network: Permiten conectar los pods a la red.

* IPAM: Responsables de direccionar las direcciones IP.

- Web acerca de CNI, su documentación y sus plugins: https://cni.dev/docs

- Para hacer la prueba se usará el plugin Calico. Aunque podríamos utilizar Flannel, Weave o Cilium.

- Como ya se ha mencionado, kubernetes tiene un DNS, donde cada cluster de kubernetes proporciona un servicio DNS, ademas, cualquier pod o servicio es

localizable a través de este DNS. El servidor DNS interno de kubernetes es: nameserver 10.96.0.10

BOOTSTRAPING DEL CLUSTER CON KUBEADM

- Para todo ello nos guiaremos de la página: https://projectcalico.docs.tigera.io/getting-started/kubernetes/quickstart

- IMPORTANTE: ELEGIREMOS COMO NODO MAESTRO/CONTROL PLANE (MASTER) EL NODO1 Y COMO NODOS ESCLAVOS (SLAVES) LOS NODOS 2 Y 3.

- sudo kubeadm init --pod-network-cidr=192.168.0.0/16 (Inicilaiza el master usando este comando. De esta manera se crea un cluster, se descarga todas las imágenes, se crea el directorio /etc/kubernetes, etc.) (SOLO EN EL NODO MASTER)

- Para empezar a usar el cluster, ejecutar como usuario normal:

  * mkdir -p $HOME/.kube (SOLO EN EL NODO MASTER)

  * sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config (SOLO EN EL NODO MASTER) (SI PIDE SOBREESCRIBIR, DARLE QUE SI)

  * sudo chown $(id -u):$(id -g) $HOME/.kube/config (SOLO EN EL NODO MASTER)

- Ahora, en el directorio ~/.kube tenemos lo mismo que en minikube, un fichero config.

- Ya puedo hacer un: kubectl get nodes y veremos que ya tendremos un nodo pero Not ready. (SOLO EN EL NODO MASTER)

INSTALAR PLUGIN NETWORK TIGERA CALICO

- Un operador es un componente de kubernetes que permite desplegar aplicaciones complejas. Es algo parecido a una plantilla, que me facilita las operaciones a la hora de instalar el network Calico.

- kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.25.0/manifests/tigera-operator.yaml (Instalamos el operador de Tigera Calico y las definiciones de recursos personalizados) (SOLO EN EL NODO MASTER)

- kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.25.0/manifests/custom-resources.yaml (Instalamos el plugin Calico) (SOLO EN EL NODO MASTER)

- watch kubectl get pods -n calico-system (Vemos los pods creados) (SOLO EN EL NODO MASTER)

- kubectl get ns (Vemos los NS de calico y tiger) (SOLO EN EL NODO MASTER)

- kubectl get all -n tigera-operator (Vemos todos los operadores de tigera) (SOLO EN EL NODO MASTER)

PERMITIR QUE EL MASTER EJECUTE PODS

- kubectl taint nodes --all node-role.kubernetes.io/control-plane-node-role.kubernetes.io/master- (Hacemos un untainted del nodo para que pueda realizar scheduling de los workloads) (SOLO EN EL NODO MASTER)

- Si el anterior comando falla, hacer: (SOLO EN EL NODO MASTER)

kubectl taint nodes --all node-role.kubernetes.io/control-plane-

AÑADIR NODOS

- Si no he copiado el join del kubeadm init, puedo ejecutar: kubeadm token create --print-join y me da un nuevo token.

- Con kubeadm token list puedo ver un alista de mis tokens.

- Ahora, desde los nodos esclavos, Nodos 2 y 3, ejecutamos lo siguiente para unir los nodos:

- sudo kubeadm join 192.168.147.131:6443 --token g08hda.2ppt4iqlws45f13w \ --discovery-token-ca-cert-hash sha256:e1ab36314a1c61774368fa7c4ff4df19d2437352a3ae913b2f038da6b0b79984 (SOLO EN LOS ESCLAVOS)

- Desde el nodo MASTER (CONTROL PLANE), hacemos: kubectl get nodes y veremos los otros dos nodos que acabamos de unir, nodo 2 y nodo 3. (SOLO EN EL NODO MASTER)

PROBAR EL CLUSTER CREANDO DEPLOYMENTS

- kubectl create deployment apache6 --replicas=3 --image=httpd (Creamos el deploy apache6 desde uno de los nodos, en este caso en el Nodo1)

- kubectl get pods -o wide (Nos muestra los 3 pods (3 replicas) y veríamos que alomejor 2 esta ejecutados en el NODO2 y el otro en el nodo 3, por ejemplo)

- kubectl scale deploy apache6 --replicas=5 (Añado dos pods mas al deploy apache6)

- kubectl get pods -o wide (Veremos que los pods están distribuidos entre los nodos workers, los nodos esclavos)

- kubectl get nodes (Podemos ver los nodos del cluster. Nodo 1, 2 y 3. Los que se han unido al mismo)

PROCESOS Y FICHEROS GENERADOS POR EL CLUSTER

- Hay más procesos en el nodo master que en los nodos esclavos. Algunos de los procesos son kube-controller-manager, sube-proxy, kubelet, etc. Para ver los procesos de kubernetes: ps -ef | grep kube

- En el directorio /etc/kubernetes, hay varios ficheros.

INTRODUCCION AL SCHEDULER

- El scheduler es el responsable de saber a que nodos van a parar cada uno de los pods que se despliegan en un cluster de kubernetes. El scheduler tiene un filtrado en el que se evitan aquellos nodos que no cumplan determinadas condiciones, que pueden ser condiciones de tipo: Taints (para no permitir que le lleguen a los nodos determinados PODS), de tipo recursos y de tipo selectors. Luego, el scheduler tiene un segundo paso después del filtrado, el cuál es el Scoring, para determinar que nodo tiene la puntuación mas alta de los que quedan sin filtrar, y se tienen también en cuenta distintos mecanismos: la afinidad de los nodos (se le puede marcar a un pod que prefiera a unos nodos antes que a otros), la existencia imagen (se tiene en cuenta si la imagen que necesita para desplegar el pod ya está en un nodo, así evita un pull) y por carga (es decir si 1 nodo tiene 50 pods y otro nodo tiene 1 pod solo, entonces el scheduler decidirá llevarlo al que tenga menos carga). Una vez seleccionado el nodo, se pone en contacto con el kubelet (agente de kuberentes) y le lanza el pod al container runtime para que lo ejecute.

- Se puede sustituir el scheduler predefinido por otro.

ASIGNAR UN NODO DE FORMA MANUAL A UN POD

- *NOTA:* TODOS LOS COMANDOS DE VER PODS, CREARLOS, VER DEPLOYS/SERVICIOS, TODO LO QUE TENGA QUE VER CON KUBECTL, EJECUTARLO DEESDE EL NODO MASTER - EL CONTROL PLANE. DESDE LOS SLAVES NO FUNCIONARÁ.

- Creamos un pod llamado nginx, basado en la imagen de nginx. Además, le indicamos que ese pod se ejecute en el nodo curso-kubernetes2 (Nodo2-Slave). El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    zone: prod
    version: v1
spec:
  containers:
   - name: nginx
     image: nginx
  nodeName: curso-kubernetes2 # Si el nodo que se indica aquí, no
llega a existir, tarde o temprano el pod se rechaza y desaparece
```

- kubectl apply -f nginx.yaml (Creamos el pod nginx, que irá a parar al nodo curso-kubernetes2)

- De esta manera, se ha asignado el pod nginx al nodo curso-kubernetes2. Cabe recordar que esta manera NO es eficiente.

NODE SELECTOR CON POD

- Una manera mas eficiente de saber a que nodos pueden ir a parar un pod es utilizar el node-selector. El node-selector busca en una serie de nodos que tengan una determinada condición para ser candidatos a que ese pod pueda parar en ese nodo. Esas condiciones se ponen a través de las etiquetas/labels.

- Para ello se usa la cláusula nodeSelector, que en este caso busca los nodos que tengan como etiqueta "entorno=desarrollo", y esos nodos que tengan esa etiqueta/label son candidatos. El resto de nodos que no posean esa label, directamente se van a rechazar (Fase de filtrado). El fichero nginx.yaml es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    zone: prod
    version: v1
spec:
  containers:
   - name: nginx
     image: nginx
  nodeSelector: # NodeSelector. Si no hay ningún nodo que cumpla la condición del nodeSelector, que no tenga esa etiqueta, el pod se quedará pending.
     entorno: desarrollo
```

- kubectl get nodes --show-labels (Para ver las etiquetas de los nodos que poseo)

- Ahora, como ejemplo, voy a asignarle la etiqueta "entorno: desarrollo" al nodo curso-kubernetes2: kubectl label node curso-kubernetes2 entorno=desarrollo

- kubectl apply -f nginx.yaml (Recreamos el pod nginx o lo eliminamos y lo volvemos a crear, e irá a parar al nodo curso-kubernetes2, ya que se lo hemos indicado en el nodeSelector y ese nodo tiene la etiqueta entorno: desarrollo)

- De esta manera, también se puede seleccionar un nodo al que puede parar un pod. Pero esta manera es mas eficiente que la anterior.

EJEMPLO NODE SELECTOR CON DEPLOYMENT

- Creamos un deployment llamado nginx-d, eue ejecutará 6 replicas de sus pods basados en la imagen nginx, escuchará por el puerto 80 e irá a parar a los nodos cuyas labels coincidan con aplicacion=web. El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: nginx
  replicas: 6 # indica al controlador que ejecute 6 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
```

```
    ports:
    - containerPort: 80
  nodeSelector:
    aplicacion: web
```

- Por ejemplo, etiqueto los nodos curso-kubernetes2 y curso-kubernetes3 (NODOS SLAVES), añadiendo a ambos la etiqueta aplicacion=web para que así ambos sean candidatos para ejecutar el deployment nginx-d. Para ello: kubectl label node curso-kubernetes2 aplicacion=web y kubectl label node curso-kubernetes3 aplicacion=web

- kubectl apply -f deploy_nginx.yaml (Creo el deploy haciendo uso del fichero YAML)

- Y ahora vemos que los pods del deploy nginx-d solo se están ejecutando en los nodos curso-kubernetes2 y curso-kubernetes3, ya que son los únicos nodos que poseen la etiqueta aplicacion=web. El nodo curso-kubernetes NO ejecuta ningún pod del deploy nginx-d debido a que no posee la etiqueta/label aplicacion=web, por lo que no es candidato y se rechaza que ese nodo ejecute esos pods.

ETIQUETAS BIEN DEFINIDAS - LABELS AUTOMÁTICAS DE KUBERNETES

- Hay etiquetas y anotaciones predefinidas que se le asignan a los objetos de kubernetes por defecto, de manera automática, y suelen llevar la extensión kubernetes.io

- Si yo quiero desplegar una app en varios nodos, y esa app solo funciona en maquinas linux, y yo en mi infraestructura tengo varias maquinas donde hay maquinas linux y maquinas windows, entonces podríamos usar el nodeSelector para coger la label/etiqueta por defecto de los nodos como: kubernetes.io/os=linux, y que esa app se ejecute solo e nodos que ejecuten un S.O Linux.

AFINIDAD DE NODOS

- Hay 2 tipos de afinidad en Kubernetes: la afinidad de nodos y la afinidad interpods.

- La afinidad de nodos nos permite que algunos pods prefieran unos nodos antes que otros.

- La afinidad de nodos es muy parecida al node selector, pero mucho mas detallada, más granular, donde se pueden incluir varias reglas y se pueden decidir si estas reglas se aplican en la fase planificación o en la fase de ejecución.

Además, se puede indicar si es preferido u obligatorio:

* requiredDuringSchedulingIgnoredDuringExecution

* requiredDuringSchedulingRequiredDuringExecution

* preferredDuringSchedulingIgnoredDuringExecution

* prefereredDuringSchedulingRequiredDuringExecution

- El Node Affinity también utiliza etiquetas/labels para identificar los nodos correctos. Se pueden utilizar diferentes operadores y opciones para poner estas reglas.

- Operadores: In, NotIn, exists, gt, lt, etc.

- Podemos usar: nodeSelectorTerms y matchExpression.

AFINIDAD DE NODOS - EJEMPLO PRÁCTICO

- Creamos un pod llamado apache1, que ejecutará una imagen basada en Apache y que además tiene una afinidad de nodo (nodeAffinity) que es requerido durante la fase de planificación e ignorado en la fase de ejecución. También, expresamos la condición de comparación (matchExpressions) con la cuál detectamos este nodo, donde hay una label llamada "equipo" que tenga entre (IN) sus valores desarrollo-web (equipo=desarrollo-web) y/o desarrollo-python (equipo=desarrollo-python). El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: apache1
spec:
  affinity:
    nodeAffinity: # También se podría podAffinity, pero en este caso estamos trabajando contra nodos
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: equipo
            operator: In
            values:
            - desarrollo-web
            - desarrollo-python
  containers:
   - name: apache1
     image: httpd
```

- Si ningún nodo NO tuviera como labels equipo=desarrollo-web o equipo=desarrollo-python, entonces el pod se quedaría en estado pending, ya que no encuentra ningún nodo que tenga esas características.

- Para que con este ejemplo el pod apache1 funcione, vamos a añadir una sola label más al nodo curso-kubernetes3: kubectl label node curso-kubernetes3 equipo=desarrollo-web (Nótese que sólo se añadió equipo=desarrollo-web y no equipo=desarrollo-python)

- kubectl apply -f apache1.yaml (Creamos el pod apache1 haciendo uso del fichero YAML)

- Ahora, el pod apache1 está corriendo en el nodo curso-kubernetes3. Usar Afinidad de nodos (nodeAffinity) es mucho mejor que usar el nodeSelector, ya que permite más detalles que nodeSelector.

TAINTS Y TOLERATIONS

- Un taint permite que un nodo NO acepte 1 o varios PODS. Es como una regla que si un pod no la cumple, no se asocia a un nodo. Es lo contrario al Node Affinity, es decir estoy diciendo que nodos deberíamos rechazar siempre. Los taints se aplican a los nodos y las tolerations a los pods

- Por ejemplo, tengo 1 nodo con un taint, y tiene una label entorno=produccion:NoSchedule. Luego tengo 1 nodo sin ninguna toleration, entonces ese pod no se puede desplegar en ese nodo. Luego tengo otro nodo con una toleration que tiene una label entorno=produccion y con un efecto NoSchedule, entonces este nodo sí sería tolerante y sí se podría desplegar en ese nodo.

TAINTS Y TOLERATIONS - LABORATORIO PRÁCTICO - EJEMPLO

- kubectl taint node curso-kubernetes3 memoria=grande:NoSchedule (Le pongo un taint al nodo curso-kubernetes3 con la etiqueta meoria=grande:NoSchedule)

- kubectl describe nodes | grep Taint (Ver los Taints de los nodos)

- Ahora si yo despliego algún pod o algun deployment o algo, ninguno debería ir a parar al nodo curso-kubernetes3, debido a que ningún deploy, pod ... que tengo ahora mismo tiene un tolerancia que coincida con el taint del nodo curso-kuberentes3.

- kubectl create deploy apache2 --replicas=4 --image=httpd (Creo un deploy llamado apache 2 con 4 replicas de sus pods basados en la imagen Apache httpd)

- Ahora, si hacemos un: kubectl get pods -o wide, veremos que ninguna de las 4 replicas de los pods del deploy apache2 está corriendo en el nodo curso-kuberentes3, debido a que no tiene una tolerancia que permitan ejecutarse en ese nodo.

- Ahora, vamos a poner una tolerancia. Para ello, creamos un deploy llamado nginx2-d que tiene 5 replicas de sus pods y que además tiene como tolerancia que el efecto sea NoSchedule y que algún taint tenga como etiqueta de taint memoria=grande, cosas que coinciden al 100% con el taint del nodo curso-kubernetes3, por lo que gracias a esta tolerancia, el despliegue de este deploy nginx2-d puede ir a parar al nodo curso-kubernetes3, entre otros, ya que la tolerancia del deploy coincide con el taint del nodo curso-kubernetes3. El fichero YAML del deploy es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones
anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx2-d
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
```

```
    matchLabels:
      app: nginx2
  replicas: 5 # indica al controlador que ejecute 5 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx2
    spec:
      containers:
      - name: nginx2
        image: nginx:1.7.9
        ports:
        - containerPort: 80
      tolerations:
      - effect: NoSchedule
        key: memoria
        operator: Equal
        value: grande
```

- kubectl apply -f deploy_nginx2.yaml (Creamos el deploy nginx2-d con sus tolerancias haciendo uso del fichero YAML anterior)

- Ahora, si ejecutamos: kubectl get pods -o wide, veremos que hay algunos pods que sí están corriendo en el nodo curso-kubernetes3, ya que el deploy nginx2-d es tolerante a ese nodo. Además, este deploy también puede correr en el resto d nodos: curso-kubernetes y curso-kubernetes2 y. De hecho, están corriendo: 2 en el nodo3, 2 en el nodo2 y 1 en el nodo1, en mi MV.

INTRODUCCIÓN A LA GESTIÓN DE RECURSOS

- En kubernetes, es importante determinar correctamente recursos como la CPU y la memoria, es decir que todo el SW que se vaya a ejecutar dentro de los

nodos, se ejecute con unas prioridades y unos recursos en concreto. Estos recursos se pueden configurar de 2 formas:

* Manualmente: podemos configurar los recursos a nivel de Pod, Namespace, Nodos, etc.

* Automáticamente: Se pueden usar distintas alternativas, para que sea el propio Cluster el que determine el escalado correcto en caso de cargar de trabajo más pesadas.

- Usaremos ResourceQuotas y LimitRange.

CONFIGURAR LA MEMORIA DE UN DEPLOY- POD

- Creamos un deploy llamado nginx3-d, que ejecuta 5 replicas de sus pods de nginx y ademas los pods tendrán como limite de memoria 200Mi (y nunca se podrá pasar de ese límite y si se pasa ese pod se elimina) y la memoria inicial con la que se arrancan será de 100M. El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx3-d
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: nginx3
  replicas: 5 # indica al controlador que ejecute 5 pods
  template:   # Plantilla que define los containers
    metadata:
```

```
      labels:
        app: nginx3
    spec:
      containers:
      - name: nginx3
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        resources:
           limits:
             memory: "200Mi"
           requests:
             memory: "100M"
```

- kubectl apply -f deploy_nginx.yaml (Creo del deploy nginx3-d con los recursos de memoria especificados en el fichero YAML anterior)

CONFIGURAR LA CPU DE UN DEPLOY-POD

- Creamos un deploy llamado nginx4-d que ejecuta 3 replicas de sus pods de nginx y además tendrá como límites máximos de memoria y CPU: 200Mi y 2 CPU, y además el deploy y todos sus pods se arrancaran (peticiones que se hacen iniciales de esos contenedores) inicialmente conn 100Mi de memoria y con 0.5 CPU. El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx4-d
  labels:
    estado: "1"
spec:
```

```
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: nginx4
  replicas: 3 # indica al controlador que ejecute 3 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx4
    spec:
      containers:
      - name: nginx4
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        resources: # Esta sección de recursos, vale para el pod
en global, es decir, si el pod tiene 2 contenedores, el limite de
cada container no sera de 2 CPU y 200Mi, sino que se repartirá,
por ejemplo 1 CPU de limite y 100Mi pa 1 container y lo mismo para
el segundo container del pod.
          limits:
              memory: "200Mi"
              cpu: "2"
          requests:
              memory: "100Mi"
              cpu: "0.5"
```

- kubectl apply -f deploy_nginx_cpu.yaml (Creo del deploy nginx4-d con los recursos de memoria y CPU especificados en el fichero YAML anterior)

INFORMACIÓN DE LOS PODS - INSTALAR METRICS SERVER EN MINIKUBE

- MAS INFO ACERCA DE KUBERNETES METRIC SERVER: https://github.com/kubernetes-sigs/metrics-server

- kubectl top (Nos permite ver el consumo de los nodos y pods).

- Las métricas se usan para determinar el uso de la CPU y la memoria. Esto es un recurso de kubernetes, la API Metrics, pero en clusters que están recién creados normalmente no está configurada.

- El kubernetes Metrics Server recolecta las métricas y las expone al API Server de Kubernetes a través de la Metrics API. Además, el Kubernetes Metrics Server es usado por Horizontal Pod Autoscale y el Vertical Pod Autoscale.

- El kubernetes Metrics Server solo gestiona memoria y CPU, pero si yo quiero gestionar mas recursos, tengo que usar un producto externo, como Prometeus, que es un monitorizador.

- Para instalar Kubernetes Metric Server en Minikube:

  * minikube addons list (Nos lista los productos/servicios/plugins que se instalan automáticamente con Minikube. Con esta lista, nos fijamos que uno de los productos de la lista es "metrics-server" que está desactivado)

  * minikube addons enable metrics-server (Activamos el producto metrics-server en Minikube)

  * minikube addons list (Veremos que ahora el producto metrics-server está activado)

- kubectl top pod nginx-d-7759cfdc55-8mqrx (Ahora, de esta manera podemos ver cuanto de memoria y CPU consume un pod, gracias al metrics-server)

INSTALAR METRICS SERVER EN UN CLUSTER REAL CON KUBEADM

- Para ello, se usará el clúster conformado por los nodos 1 (maestro/control-plane), 2 (slave) y 3 (slave) creado con Kubeadm y MV's en secciones anteriores.

- Primero, antes de instalar metrics-server con kubeadm hay que tener en cuenta una serie de requerimientos.

- Normalmente las metrics-server se ejecutan en el namespace kube-system.

- Para instalar metrics-server en un cluster real como kubeadm hacer (VERSIÓN QUE FALLA CON KUBEADM):

* kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

* kubectl get svc metrics-server -n kube-system (Vemos el servicio creado por metrics-server en namespace kube-system)

* kubectl get deploy metrics-server -n kube-system (Veremos que nos sale un error 0/1 debido a que algunos de los requisitos para instalar metrics-server con kubeadm no están instalados o configurados)

- Para instalar metrics-server en un cluster real como kubeadmn hacer (VERSIÓN QUE FUNCIONA CON KUBEADM):

* mkdir metrics && cd metrics

* wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml (Nos traemos/descargamos el YAML de instalación de metrics-server a la carpeta metrics recién creada para modificar algunas cosas de este fichero)

* vi componentes.yaml (Modificar la parte del "deployment - args", para que metrics-server funcione correctamente. Para ello, añadir a la lista de argumentos (args): - --kubelet-insecure-tls. Además, en el argumento (args) --kubelet-preferred-address-types añadir: InternalDNS y ExternalDNS. Luego, subir el argumento (args) --metric.resolution a 30s. Por último, añadir la línea hostNetwork: true al deployment del fichero componentes.yaml, justo después del nodeSelector)

* kubectl delete deploy metrics-server -n kube-system (Borro el deployment que se creó anteriormente con la instalación fallida de metrics-server. Si no se ha realizado anteriormente la instalación fallida o no existe este deploy en ese namespace, no hace falta hacer este paso (delete))

* kubectl apply -f components.yaml (Creamos todo lo referente al metrics-server haciendo uso del fichero YAML recién modificado y descargado desde GitHub)

* kubectl get pods -n kube-system (Veremos el pod metrics-server running 1/1 ahora)

* kubectl top nodes (Vemos las estadísticas de CPU y memoria de los nodos)

* kubectl top pods (Vemos las estadísticas de CPU y memoria de los pods)

- *NOTA:* RECORDAMOS QUE SI NO QUEREMOS USAR METRICS-SERVER, TENEMOS EL SOFTWARE PROMETEUS PARA TENER METRICAS, LO QUE ESO SÍ, PROMETEUS ES UN SOFTWARE DE TERCEROS, PERO SE PODRIA USAR SIN PROBLEMA PROMETEUS EN LUGAR DE METRICS-SERVER.

DETERMINAR LOS RECURSOS DE MEMORIA Y CPU EN UN NAMESPACE

- Hay 2 objetos para determinar los recursos de memoria y CPU en un namespace: LimitRange y ResourceQuotas.

- LimitRange: Permite especificar los límites y el uso para los PODS y los contenedores que tenemos dentro de un namespace. Esto impide que los POD o contenedores NO superen determinados límites dentro de un entorno. Se aplica de manera individuala cada pod/contenedor.

- ResourceQuotas: Permite limitar los recursos totales utilizados dentro de un namespace, Esto permite que haya múltiples proyectos en nuestra infraestructura que consuman distintos límites de recursos. Se aplica a todo un namespace.

- Se pueden combinar los LimitRanges y los ResourceQuotas.

LIMITRANGE - LIMITAR LOS RECURSOS DE LOS CONTENEDORES EN UNA NAMESPACE

- Vamos a poner límites de memoria y CPU a nuestros pods pero dentro de un namespace. Para ello haremos uso de un nuevo objeto de kubernetes, los cuáles son los "LimitRange". Para ello crearemos un LimitRange llamado "recursos2" que tendrá varios límites de memoria y de CPU. El LimitRange solo funciona con los objetos que se crean cuando esos límites ya están establecidos en el NS, no modifica los límites de los recursos que estén anteriormente en funcionamiento. El fichero YAML del objeto LimitRange es:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: recursos2
```

```
spec:
  limits:
  - default:
      memory: 512Mi # Límites por defecto
      cpu: 1
    defaultRequest: # Lo que voy a tener por defecto cuando se
crean los objetos ()
      memory: 256Mi
      cpu: 0.5
    max: # El máximo de CPU y memoria
      memory: 1Gi
      cpu: 4
    min: # El mínimo de CPU y memoria
      memory: 128Mi
      cpu: 0.5
    type: Container # Se puede poner a nivel de container o a
nivel de Pod
```

- kubectl create namespace n2 (Creamos un nuevo namespace llamado n2)

- kubectl apply -f limitex2.yaml -n n2 (Creo el objeto LimitRange haciendo uso del fichero YAML para poner límites a los namespaces, ya que antes no tenían)

- kubectl describe ns n2 (Vemos los límites establecidos por el LimitRange)

- kubectl exec nginx -n n2 -it -- bash (Entramos al contenedor del pod nginx del namespace n2)

- Dentro del contenedor hacemos: apt-get update y apt-get install stress (Para instalar stress para comprobar el estrés de la memoria)

- Dentro del contenedor hacer stress -m 10 y después ejecutar dmesg (Veremos el fallo en la memoria)

- Si yo lanzara un recurso que supere el máximo de esos límites, ese recurso no me lo dejaría lanzar y me saldría que supera el límite.

RESOURCEQUOTA - LIMITAR LOS RECURSOS DE UN NAMESPACE

- Creamos un resourcequota llamada pods-grandes y que tendrá como máximo una CPU de 100 cores, una memoria de 500Mi y 5 pods dentro del namespace. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-grandes
spec:
  hard:
    cpu: "100"
    memory: 500Mi
    pods: "5"
```

- kubectl apply -f resource1.yaml (Creamos la quota de recursos haciendo uso del fichero YAML anterior)

- kubectl get quota (Nos muestra las quotas y sus estadísticas)

- Si yo ahora edito el fichero resource1.yaml de esta manera:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-grandes
spec:
  hard:
    requests.cpu: "100"
    requests.memory: 500Mi
    limits.cpu: "500"
    limits.memory: 1Gi
    pods: "5"
```

- kubectl apply -f resource1.yaml (Recreamos la quota de recursos haciendo uso del fichero YAML anterior)

- kubectl get quota (Nos muestra las quotas y sus estadísticas con los nuevos valores)

- kubectl run apache2 --image=httpd (Al crear este pod no me dejaría porque tengo que especificar los valores establecidos en la quota)

- Creamos un nuevo pod, llamado nginx3, donde se le especifican los limites y las peticiones por defecto de memoria y CPU. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx3
  labels:
    zone: prod
    version: v1
spec:
  containers:
```

```
  - name: nginx  3
    image: apasoft/nginx:v1
    resources:
     limits:
       memory: "100Mi"
       cpu: "1"
     requests:
       memory: "10Mi"
       cpu: "0.5"
```

- kubectl apply -f nginx3.yaml (Creamos el pods haciendo uso del fichero YAML)

- kubectl get quota (Vemos que las estadísticas de quota han cambiado)

- Si yo creara un deploy con 2 réplicas de sus pods, que sumando su memoria de los 2 pods superen los límites de la quota, entonces 1 de los 2 pods no se creará.

DAR PRIORIDADES A NUESTROS PODS - PRIORITYCLASSENAME

- Se usan para dar prioridad a algunos pods frente a otros.

- Me creo dos Priorityclass llamadas producción y desarrollo, donde se van a expulsar a los pods que tengan menos prioridad (value). En este caso, la priorityClass desarrollo tiene menos prioridad que la de producción, ya que 50 < 100 (value). El fichero YAML es el siguiente:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: produccion
value: 100 # El valor de la prioridad
```

```
preemptionPolicy: PreemptLowerPriority # Si ponemos None, no sustituye a los pods con mas baja prioridad, sino se queda esperando, eso si, esa espera hace que se pongan los primeros en la lista, en la cola para hacer el deploy, por ejemplo.
globalDefault: false # Si pongo true es utilizada por los pods que no usen la prioridad, Y solo se puede poner 1 a true
description: "Pods para entornos de Producción."

---

apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: desarrollo
value: 50
preemptionPolicy: PreemptLowerPriority
globalDefault: false
description: "Pods para entornos de desarrollo."
```

- kubectl apply -f prioridades.yaml (Creamos las dos prioridades haciendo uso del fichero YAML anterior)

- kubectl get pc (Nos muestra las priorityclass)

- Creo un deploy llamado apache-deploy con 60 replicas de sus pods. Además, los pods tendrán por defecto 256Mi de memoria y 0.5 CPu y como límites máximos 1 CPU y 400Mi de memoria y dandole prioridad a la priorityclass desarrollo. El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
```

```
metadata:
  name: apache-deploy
  labels:
    estado: "1"
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: apache
  replicas: 60 # indica al controlador que ejecute 60 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: apache
    spec:
      priorityClassName: "desarrollo" # La priorityclassName será
la de desarrollo, la de value 50
      containers:
      - name: apache
        image: httpd
        ports:
        - containerPort: 80
        resources:
          limits:
              memory: "400Mi"
              cpu: "1"
          requests:
              memory: "256Mi"
              cpu: "0.5"
```

- kubectl apply -f apache-deploy.yaml (Creamos el deploy haciendo uso del fichero YAML)

- Tras esperar un rato, veremos cuantos pods ha sido capaz de crear de los 60 especificados.

- kubectl describe pod apache-deploy-8456845446-tk94h (Al describir alguno de los 60 pods que si están running, podemos ver su información detallada y ver que su priority class name es desarrollo, como se especificó en el fichero YAML)

- Acto seguido, creo otro deploy llamado apache-prod que creará 5 replicas de sus pods, tendrán esos pods unos determinados recursos y además esos pods tendrás como priorityClassNmae producción (value=100). El fichero YAML es el siguiente:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: apache-prod
  labels:
    estado: "1"
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: apache
  replicas: 5  # indica al controlador que ejecute 2 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: apache
    spec:
      priorityClassName: "produccion"
      containers:
```

```
    - name: apache
      image: httpd
      ports:
      - containerPort: 80
      resources:
        limits:
            memory: "400Mi"
            cpu: "1"
        requests:
            memory: "256Mi"
            cpu: "0.5"
```

- kubectl apply -f apache-deploy2.yaml (Creamos el deploy haciendo uso del fichero YAML)

- Tras lanzar estos pods, algunos de los pods anteriores (con prioridad 50, de desarrollo), al no encontrar espacio para crear los pods de producción (ya que tienen mayor prioridad), echa del cluster algunos pods creados anteriormente (con prioridad desarrollo). De esta manera, la prioridad se impone y si no hay espacio para crear los pods de producción, se eliminan algunos de la prioridad desarrollo para que estos se creen.

QUOTAS - TRABAJAR CON MÚLTIPLES QUOTAS

- Tenemos dos resourcequotas, pods-grandes y podas-peques, con unos límites de CPU y memoria. Además, una quota es para producción y otra para desarrollo. Por último, se le pone la priority class de cada quota, para que los pods vayan a parar una cuota u otra. De esta manera, os pods que pertenezcan a la class priorityclass produccion tendrán unos limites de quota y los de desarrollo tendrán otras quotas. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: ResourceQuota
```

```
metadata:
  name: pods-grandes
spec:
  hard:
    requests.cpu: "100"
    requests.memory: 500Mi
    limits.cpu: "500"
    limits.memory: 1Gi
    pods: "5"
  scopeSelector:
    matchExpressions:
    - operator : In
      scopeName: PriorityClass
      values: ["produccion"] # Esta ProrityClass debe existir
obviamente
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-peques
spec:
  hard:
    requests.cpu: "50"
    requests.memory: 100Mi
    limits.cpu: "100"
    limits.memory: 200Mi
    pods: "10"
  scopeSelector:
    matchExpressions:
    - operator : In
      scopeName: PriorityClass
      values: ["desarrollo"]
```

- kubectl apply -f resources2.yaml (Creamos las quotas haciendo uso del fichero YAML)

- Ahora, creo un pod llamado nginx6, con unos límites y una priorty class producción. El fichero YAML es:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx6
  labels:
    zone: prod
    version: v1
spec:
  containers:
   - name: nginx6
     image: apasoft/nginx:v1
     resources:
      limits:
        memory: "100Mi"
        cpu: "1"
      requests:
        memory: "10Mi"
        cpu: "0.5"
  priorityClassName: produccion
```

- kubectl apply -f nginx6.yaml (Creamos el pod haciendo uso del fichero YAML)

- kubectl get quota (Vremeos que el pod nginx6 se ha asociado a la quota pods-grandes ya que tenia el priority class de producción)

- Ahora, creo un pod llamado nginx7, con unos límites y una priorty class desarrollo. El fichero YAML es:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx7
  labels:
    zone: prod
    version: v1
spec:
  containers:
   - name: nginx7
     image: apasoft/nginx:v1
     resources:
      limits:
        memory: "100Mi"
        cpu: "1"
      requests:
        memory: "10Mi"
        cpu: "0.5"
  priorityClassName: desarrollo
```

- kubectl apply -f nginx7.yaml (Creamos el pod haciendo uso del fichero YAML)

- kubectl get quota (Veremos que el pod nginx7 se ha asociado a la quota pods-peques ya que tenia el priority class de desarrollo)

VER LOS RECURSOS DE UN NODO

- kubectl get nodes (Vemos los nodos que tenemos)

- kubectl top nodes (Vemos el porcentaje de memoria y CPU de los nodos)

- kubectl get nodes -o wide (Vemos los nodos que tenemos, aparte la IP y demás)

- kubectl describe node curso-kubernetes2 (Nos muestra información acerca del nodo 2. Aquí podemos ver la capacidad de ese nodo y la que tiene disponible)

HUGE PAGES

- Páginas grandes. Se utilizan para las zonas de memoria de kubernetes. Es un recurso que se utiliza dentro de plataformas Linux, y nos permiten manejar páginas de memoria mucho más grandes de lo habitual. Por ejemplo, el kernel de linux normalmente soporta un tamaño de página de 4k de tamaño. Con este recurso de Huge pages podemos tener páginas de memoria de 2Gigas o de 1 Giga. Esto es beneficioso y óptimo para aplicaciones que necesitan mucha memoria contigua, por ejemplo: ELK, Cassandra, Oracle, etc.

CONFIGURAR HUGE PAGES EN LINUX

- minikube ssh -n curso-kubernetes2 -p minikube (Entro via ssh al nodo2, si trabajamos con minikube) ó ssh curso-kubernetes2 (entro via ssh normal, si trabajo con kubeadm)

- cat /proc/meminfo (Podemos comprobar si las HugePages está configurada) ó usar sudo sysctl -a | grep huge* (otra forma de comprobar si tenemos hugepages)

- sudo vi /etc/sysctl.conf (Añadimos al final del fichero: vm.nr_hugepages=100. Y ahora, se tendría un total de 100 Páginas de memoria Huge, cuando antes se tenia 0)

INTRODUCCIÓN AL AUTOESCALADO

- Esto posibilita aumentar o reducir los recursos de forma automática dependiendo de la carga de trabajo que tengamos.

- Dentro de kubernetes se pueden utilizar 3 tipos de autoescalado:

  * HPA- (Horizontal pod scaler. El utilizado a día de hoy)

  * VPA- (Vertical pod autoscaler. Esta en Beta)

  * CA- (Cluster Autoscaler. Para determinadas plataformas)

- HPA: Si me quedo corto de recursos, se aumentan el numero de pods que tengo en el despliegue. Es gestionado por el controller manager. En cada bucle el controller manager compara el uso actual de los recursos con las métricas definidas para cada HP, estas métricas pueden ser utilizadas por un metric server o bien se pueden obtener directamente desde los Pods. Este es el más funcional a día de hoy.

- VPA: Aumento de la memoria y CPU de los pods existentes. Hasta el día de hoy esta en Beta.

- CA: Para aumentar el número de nodos en el clúster si los nodos existentes no son capaces de afrontar el numero de pods que se le soliciten. Solo funciona en entornos Cloud.

LABORATORIO HPA AUTOESCALADO

- Creamos un deployment llamado apache6, con tan solo 1 replica de sus pods y con unos límites de CPU. El fichero YAML es:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache6
spec:
  selector:
    matchLabels:
      run: apache6
  replicas: 1
  template:
    metadata:
      labels:
        run: apache6
    spec:
      containers:
      - name: apache6
        image: registry.k8s.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
              cpu: 500m
          requests:
              cpu: 200m
```

- kubectl apply -f apache6.yaml (creamos el deploy haciendo uso del fichero YAML anterior)

- Ahora, creamos un servicio de tipo ClusterIP llamado apache6-svc para probar los pods. El fichero YAML es:

```
apiVersion: v1
kind: Service
metadata:
  name: apache6-svc
  labels:
    run: apache6
spec:
  ports:
  - port: 80
  selector:
    run: apache6
```

- kubectl apply -f apache6-svc.yaml (creamos el servicio haciendo uso del fichero YAML anterior)

- kubectl autoscale deployment apache6 --cpu-percent=40 --min=1 --max=8 (Indicamos que este recurso (deploy) vaya haciendo una autoescalada a partir de un uso de la cpu de un 40% de media, que tenga como mínimo 1 pod y como máximo 8 pods)

- kubectl get hpa (Nos muestra el autoescalado de tipo HPA con las configuraciones que hemos indicado en el comando anterior)

- kubectl run -i --tty carga --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://apache6-svc; done" (Creo un pod llamado carga con la imagen busybox dede la linea de comnados que hace una consulta al servicio creado con anterioridad cada segundo)

- Ahora, ejecutamos "kubectl get hpa -n n4" en otra pestaña y vemos que el 40% de CPU se ha superado, por ello, para que este porcentaje vaya bajando se van creando replicas de los pods (cuando de manera inicial solo había 1 replica). Cuando el porcentaje baje hasta menos del 40%, se paran de crear replicas

ALMACENAMIENTO EN KUBERNETES

- Esta sección trata de crear espacio en disco (volúmenes) a los que se pueden conectar mis pods.

- Todo el almacenamiento dentro de un contenedor de kubernetes es efímero, es decir, es temporal. Se crea de forma temporal de un directorio de la máquina. Es decir, cuando el pod se destruye el almacenamiento también se elimina, debido a la inmutabilidad de Kuberentes.

- Volúmenes: Es un objeto que permite persistir el almacenamiento para trabajar, de forma permanente. Es decir, cuando el contenedor se elimine el almacenamiento seguirá existiendo y podía ser reutilizado. Kubernetes soporta volúmenes de diferentes tipos:

  * Locales
  * Externos
  * NFS, iSCSI, etc.

- CSI (Container Storage Interface): Extension/Driver de kubernetes que nos permite simplificar la gestión del almacenamiento. Estándar que expone el almacenamiento a los diferentes workloads. CSI es externo a kubernetes. CSI implementa una arquitectura de plugin extensible, de forma que podemos añadir de forma sencilla dispositivos de almacenamiento. Hay una lista de drivers CSI soportados en github.io.

COMO CREAR VOLÚMENES

- Utilizamos 2 cláusulas para implementar el almacenamiento dentro de un POD:

  * Una para indicarle dónde montarlos dentro de el POD (volumeMounts).

  * Una para indicar los volúmenes que vamos a utilizar (volumes).

CREAR VOLÚMENES EN UN POD

- Creamos un pod llamado volúmenes que ejecutará una imagen nginx y que además usará 3 volúmenes llamados home, git y temp. El volumen home se montará en el directorio /home del contenedor del pod volúmenes, usará el driver hostPath y su ruta de almacenamiento en la máquina física será /home/kubernetes/datos, ya que este volumen va a estar asociado a un directorio del sistema de ficheros. Luego, el volumen git se montará en el directorio /git del contenedor del pod volúmenes, será de sólo lectura, usará el driver gitRepo y su repositorio será https://github.com/apasoftTraining/cursoKubernetes.git, de esta manera se clonará todo los ue haya en el repo de Github en el volumen temp. Por último, el volumen temp se montará en el directorio /temp del contenedor del pod volúmenes y de esta manera es emptydir, de manera que cuando el pod se acabe este directorio también, ya que es temporal. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: volumenes
spec:
  containers:
  - name: nginx
    image: nginx
```

```
    volumeMounts: # Donde montar los volumenes en los contenedores
del pod
    - mountPath: /home
      name: home
    - mountPath: /git
      name: git
      readOnly: true
    - mountPath: /temp
      name: temp
  volumes: # Listado de los volúmenes a utilizar
  - name: home
    hostPath:
      path:   /home/kubernetes/datos   #   Debe   estar   creado
previamente en la máquina. En este caso si usamos minikube habría
que entra con minikube ssh al nodo para ver este directorio, ya
que nuestro cluster minikube es una MV.
  - name: git
    gitRepo:
      repository:
https://github.com/apasoftTraining/cursoKubernetes.git
  - name: temp
    emptyDir: {}
```

- kubectl apply -f volumenes.yaml (Creamos el pod con los volúmenes haciendo uso del fichero YAML anterior)

- kubectl describe pod volumenes (Podemos ver los montajes que tiene el pod y sus volúmenes)

- kubectl exec -it volumenes -- bash (Si ahora entro al pod volúmenes, podemos irnos al directorio /git y veremos que en su contenido ha clonado el repo de GitHub https://github.com/apasoftTraining/cursoKubernetes.git. Luego en el directorio /home tendremos lo mapeado en /home/kubernetes/datos, de esta

manera lo que haya en ese directorio de la maquina física (nodo de minikube que está ejecutando el POD) estará en el directorio del contenedor y viceversa y por ultimo también tenemos el directorio /temp)

ARQUITECTURA DE VOLÚMENES PERSISTENTES

- PV: Persitent volume. Es un recurso que apunta al almacenamiento real. Deben ser creados por los administradores. Es como un disco lógico/virtual que apunta al disco real que hay por detrás.

- PVC: PERSISTENT VOLUME CLAIM. Es una petición de recursos para reclamar un determinado PV. Localiza PV´s. Los PVC se utilizan en el POD.

- Si yo por ejemplo tengo 10 PV´s y los meto en una clase llamada "discos lentos". Entonces, en el PVC digo yo quiero un PV de la clase discos lentos.

- Tipos de acceso a los PV:

  * ReadWriteOnce (RWO): read-write solo para un nodo.

  * ReadOnlyMany (ROX): read-only para muchos nodos.

  * ReadWriteMany (RWX): read-write para muchos nodos.

  * ReadWriteOncePod (RWOP): read-write para un solo POD. Solo SCSI y versión 1.22+.

- Tipos de aprovisionamiento de un PV:

  * Estático: se asocia un PV de forma estática.

  * Dinámico: se usan las StorageClasses para encontrar un PV adecuado.

- Tipos de reciclaje de un PV:

  * Retain: Reclamación manual. Política por defecto.

  * Recycle: Reutilizar contenido (deprecated).

  * Delete: Borrar contenido.

REPASO DE COMO CREAR UN CLUSTER DE VARIOS NODOS CON MINIKUBE

- minikube start -p cluster1 -n 3 (Arranco un cluster de 3 nodos con minikube)

- minikube profile cluster1 (Cambio de perfil de clúster)

- kubectl get nodes (Y veremos los tres nodos)

CREAR UN PV - EJEMPLO CON HOSTPATH

- En primer lugar, crearemos un PV, luego un PVC y por último un POD que consuma esos recursos.

- Creo un PV llamado pv-volume, con una etiqueta/label type=local, con la cláusula storageClassName con el nombre sistemaficheros (este nombre debe coincidir con el del PVC) para que el PV sea asociado a un PVC, el PV tendrá un tamaño de 10GB, con un modo de acceso readwriteOnce (read-write solo para un nodo) y además el tipo de driver a utilizar será el hostPath (que crea el directorio /mnt/data en la máquina local/física). El fichero YAML es el siguiente:

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: pv-volume
  labels:
    type: local
spec:
  storageClassName: sistemaficheros
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

- kubectl apply -f pv.yaml (Creo el PV haciendo uso del fichero YAML)

- kubectl get pv (Vemos el PV llamado pv-volume recién creado)

- kubectl describe pv pv-volume (Nos muestra mucha información acerca del PV)

CREAR UN PVC

- Creamos un PVC llamado pv-claim, con la cláusula storageClassName con el nombre sistemaficheros (mismo nombre que el del PV anterior), con un modo de acceso readwriteonce y con una capacidad de almacenamiento de 3GB para el PVC (Aunque el PV tenga más tamaño que la capacidad del PVC, el PVC busca entonces el pv de la clase sistemaficheros que sea de menos capacidad por arriba de los 3 GB de su capacidad, en este caso como solo hay 1 PV con la clase sistemaficheros, pues coge el PV pv-volume y la capacidad del PVC ahora será de 10GB). El fichero YAML es el siguiente:

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

```
spec:
  storageClassName: sistemaficheros
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

- kubectl apply -f pvclaim.yaml (Creamos el PVC haciendo uso del fichero YAML)

- kubectl get pvc (Me muestras los PVC´s existentes)

- kubectl describe pvc pv-claim (Nos muestra información mas detallada del PVC)

- kubectl get pv (Ahora veremos el pv-claim en la columna CLAIM)

CREAR EL POD ASOCIADO

- Creo un Pod llamado pv-pod con una imagen nginx. Este pod tendrá un volumen llamado pv-storage que será el PVC pv-claim creado anteriormente. Además, este volumen asociado al PVC, se montará en el directorio /var/www/html del contenedor del POD pv-pod. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod
spec:
  volumes:
    - name: pv-storage
      persistentVolumeClaim:
        claimName: pv-claim
  containers:
```

```
  - name: task-pv-container
    image: nginx
    ports:
      - containerPort: 80
        name: "http-server"
    volumeMounts:
      - mountPath: "/var/www/html/"
        name: pv-storage
```

- kubectl apply -f pod-pvc.yaml (Creo el pod pv-pod que se asocia al PVC haciendo uso del fichero YAML)

- kubectl get pods -o wide (Podemos ver en que nodo de los 3 de minikube se está ejecutando el pod. En este caso se está ejecutando en el nodo esclavo llamado cluster1-m03)

- kubectl describe pod pv-pod (Vemos los detalles del POD, sus mounts y demas)

- minikube ssh -p cluster1 -n cluster1-m03 (Accedo vía SSH al nodo 3. Una vez dentro veré que en esa máquina local tendrá el directorio /mnt/data)

- De esta manera, hemos asociado el PVC al pod creado y ahora en el directorio /var/www/html de ese pod pv-pod se guardará de manera síncrona el volumen PVC de tipo hostPath del directorio /mnt/data del nodo de minikube que ofrezca el pod pv-pod. De esta manera el pod consume el PVC que tiene el PV.

EJEMPLO CON NFS - MONTAR EL ENTORNO

- Para que todos los nodos tengan un espacio de almacenamiento común, se usará un servidor NFS que compartirá un espacio de almacenamiento. Para ello se instalará el servidor NFS en el nodo master y el cliente NFS que usara ese almacenamiento compartido NFS exportado desde el servidor en los nodos esclavos. Para este ejemplo se usará el cluster de kubeadm con 3 nodos que

son 3 MV. Podemos seguir este tutorial para CentOS7: https://dev.to/prajwalmithun/setup-nfs-server-client-in-linux-and-unix-27id

- En el nodo maestro tendremos el servidor NFS, para instalarlo y configurarlo (¡OJO ESTO ES PARA CENTOS7!):

* sudo yum -y install nfs-utils

* mkdir /var/datos (Directorio de almacenamiento compartido)

* chmod -R 777 /var/datos

* systemctl enable rpcbind

* systemctl enable nfs-server

* systemctl enable nfs-lock

* systemctl enable nfs-idmap

* systemctl start rpcbind

* systemctl start nfs-server

* systemctl start nfs-lock

* systemctl start nfs-idmap

* vi /etc/exports y añadir:

    - /var/datos *(rw,sync,no_root_squash)

* exportfs -arv

* systemctl restart nfs-server

* showmount -e 192.168.1.85

- En los nodos esclavos instalar y configurar el cliente NFS:

* sudo yum install nfs-utils -y

* showmount -e 192.168.1.85

* mkdir /var/datos

* mount -t nfs 192.168.1.85:/var/datos /var/datos

* Ahora, en cualquiera de los esclavos creo un fichero en /var/datos y ese fichero creado desde un esclavo es visible en el maestro.

EJEMPLO CON NFS - APLICACIÓN NGINX

- Para esta sección se trabajará con el cluster de 3 nodos con kubeadm formado por MV.

- Creamos un PV llamado nfs-pv con una capacidad de almacenamiento de 5GB, con un modo de acceso ReadWriteMany, con la cláusula storageClassName volumen-nfs (que debe coincidir con el PVC) y que usará el directorio compartido /var/datos del servidor NFS del nodo master. El fichero YAML es:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
```

```
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: volumen-nfs
  nfs:
    path: /var/datos
    server: 192.168.1.85
```

- kubectl apply -f pv-nfs.yaml (Creo el PV nfs haciendo uso del fichero YAML)

- kubectl get pv (Veremos el PV)

- Luego, tras crear el PV, creamos el PVC llamado nfs-pc, con una sotrgaeclasname volumen-nfs (que coincide con class del PV, para reclamar el pv anterior), con un modo de acceso readwritemany y con una capacidad de 1GB (pero que tendrá 5Gb en realidad debido al peso del PV). El fichero YAML es:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  storageClassName: volumen-nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

- kubectl apply -f pvc-nfs.yaml (Creo el PVC nfs haciendo uso del fichero YAML)

- kubectl get pvc (Veremos el PVC)

- Por último, creamos el pod llamado pod-nfs que ejecuta una imagen de nginx, tendrá 1 volumen llamado nfs-vol que se montará en el directorio /usr/share/nginx/html del contenedor del POD y además usará el PVC creado anteriormente llamado nfs-pc. El fichero YAML del pod es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nfs
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
      - mountPath: /usr/share/nginx/html # Directorio html de
nginx por defecto
        name: nfs-vol
  volumes:
    - name: nfs-vol
      persistentVolumeClaim:
        claimName: nfs-pvc
```

- kubectl apply -f pod-nfs.yaml (Creo el Pod nfs haciendo uso del fichero YAML)

- kubectl get pods -o wide (Veremos el pod creado y en que nodo se esta ejecutando)

- kubectl proxy (Para probar el pod)

- Acceder a localhost:8001/api/v1/namespaces/default/pods/pod-nfs/proxy y vemos la página por defecto de nginx.

- Ahora, si por ejemplo desde el directorio /var/datos del nodo2, creamos en él un fichero index.html que diga pruebita, ese fichero aparecerá en /usr/sahre/nginx/html (gracias a NFS y al PVC) y accediendo ahora a la pagina de Nginx nos saldrá pruebita en lugar de la página por defecto. Ademas, en el directorio /var/datos del Nodo 3 y en el directorio /var/datos del Nodo1, también aparecerá ese index.html con Pruebita en su interior. También aparecerá por supuesto en el directorio /usr/share/nginx/html del contenedor del POD.

- Además, si yo eliminara el pod: kubectl delete pod pod-nfs y ahora voy al /var/datos de alguno de los nodos, sigo teniendo los mismos ficheros index y demás. De esta manera si yo hago el apply otra vez del pod, sigue funcionando sin problema con los mismos datos de HTML de antes.

EJEMPLO WORDPRESS CON MYSQL - PREPARACION DEL ENTORNO - PARTE 1

- Para esta sección se trabajará con el cluster de 3 nodos con kubeadm formado por MV. Se usará el CMS Wordpress que usará una BD MySQL.

- En los 3 nodos crear: mkdir -p /home/kubernetes/datos y dentro de ella crear 2 capetas: mkdir mysql y mkdir wordpress

- En el nodo master hacer: sudo chmod -R 777 /home/kubernetes/datos/mysql y sudo chmod -R 777 /home/kubernetes/datos/wordpress

- Desde el nodo 1 master y como root hacer: sudo vi /etc/exports y añadir:

```
/home/kubernetes/datos/mysql *(rw,sync,no_root_squash,no_all_squash)
/home/kubernetes/datos/wordpress *(rw,sync,no_root_squash,no_all_squash)
```

- Luego, en el nodo master ejecutar:

* sudo exportfs -arv

* sudo systemctl restart nfs-server

* showmount -e 192.168.1.85

- Tras esto, en los 2 esclavos (Nodo 2 y 3) hay que montar ambos directorios compartidos. Para ello ejecutar:

* sudo mount -t nfs 192.168.1.85:/home/kubernetes/datos/mysql /home/kubernetes/datos/mysql

* sudo mount -t nfs 192.168.1.85:/home/kubernetes/datos/wordpress /home/kubernetes/datos/wordpress

- df -h (Puedo ver los directorios montados)

- Una vez hecho esto a tenemos preparado el entorno.

EJEMPLO WORDPRESS CON MYSQL - CREACION DE LOS PV Y PVC DE MYSQL Y WORDPRESS - PARTE 2

- Creamos el PV para wordpress llamado pv-wordpress, con una capacidad de 25GB, con la cláusula storageClassname wordpress para que el PVC pueda usar este PV, con un modo de acceso ReadWriteMany y que usará el directorio compartido /home/kubernetes/datos/wordpress del servidor NFS del nodo Master (Nodo1), gracias al driver nfs. El fichero YAML del PV de wordpress es el siguiente:

```
##########################
## PV para WORDPRESS  ###
```

```
########################
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-wordpress
spec:
  capacity:
    storage: 25Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: wordpress
  nfs:
    path: /home/kubernetes/datos/wordpress
    server: 192.168.1.85
```

- kubectl apply -f pv-wordpress.yaml (Creamos el PV para wordpress)

- kubectl get pv (Vemos el pv creado)

- Creamos el PV para mysql llamado pv-mysql, con una capacidad de 25GB, con la cláusula storageClassname mysql para que el PVC pueda usar este PV, con un modo de acceso ReadWriteMany y que usará el directorio compartido /home/kubernetes/datos/mysql del servidor NFS del nodo Master (Nodo1), gracias al driver nfs. El fichero YAML del PV de MySQL es el siguiente:

```
######################
## PV para MySQL  ###
######################
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
  name: pv-mysql
spec:
  capacity:
    storage: 25Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: mysql
  nfs:
    path: /home/kubernetes/datos/mysql
    server: 192.168.1.85
```

- kubectl apply -f pv-mysql.yaml (Creamos el PV para MySQL)

- kubectl get pv (Vemos el pv creado)

- Tras haber creado los 2 PVS para Wordpress y MySQL, se crearán los 2 PVC para los mismos.

- En primer lugar se crea el PVC para wordpress, por ejemplo, que se llamará pvc-wordpress, con una capacidad de almacenamiento de 20GB (que serán 25GB debido al tamaño del PV que se asociará al mismo), con un modo de acceso ReadWriteMany y que tendrá la cláusula storageClassNAme wordpress para asociar el pv-wordpress a este PVC. El fichero YAML del PVC para Wordpress es:

```
############################
## CLAIM PARA  WORDPRESS ##
############################
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: pvc-wordpress
spec:
  storageClassName: wordpress
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 20Gi
```

- kubectl apply -f pvc-wordpress.yaml (Creamos el PVC para Wordpress)

- kubectl get pvc (Vemos el pvc creado)

- En segundo lugar se crea el PVC para mysql, por ejemplo, que se llamará pvc-mysql, con una capacidad de almacenamiento de 20GB (que serán 25GB debido al tamaño del PV que se asociará al mismo), con un modo de acceso ReadWriteMany y que tendrá la cláusula storageClassNAme mysql para asociar el pv-mysql a este PVC. El fichero YAML del PVC para MySQL es:

```
######################
## CLAIM PARA MYSQL ##
######################
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-mysql
spec:
  storageClassName: mysql
  accessModes:
    - ReadWriteMany
```

```
  resources:
    requests:
      storage: 20Gi
```

- kubectl apply -f pvc-mysql.yaml (Creamos el PVC para MySQL)

- kubectl get pvc (Vemos el pvc creado)

EJEMPLO WORDPRESS CON MYSQL - CREACION DEL DEPLOYMENT Y SERVICIO PARA SU PRUEBA DE WORDPRESS - PARTE 3

- Creamos el deployment para Wordpress llamado wordpress, con tan solo 1 replica de sus pods, con una serie de etiquetas y selectores, dónde sus pods usarán una imagen basada en Wordpress, tendrá 2 variables de entorno cada uno de los contenedores de sus pods para establecer la BD a usar por parte wordpress (que será la bd del deploy que se creará en la parte 4) y la contraseña de la misma: WORDPRESS_DB_HOST: mysql y WORDPRESS_DB_PASSWORD: password, se usará un tipo de estrategia Recreate para que si en el caso que se modifique algo en el deploy wordpress todos sus pods antiguos se eliminen y se creen los nuevos perdiendo el servicio temporalmente y además este deploy usará un volumen llamado wordpress-persistent-storage que se montará dentro de los contenedores de cada pod en el directorio /var/www/html y donde este volumen usará el pvc-wordpress creado anteriormente. Por último, en el mismo fichero se creará un servicio de tipo Nodeport llamado wordpress-svc para probar este wordpress desde un navegador exterior. El fichero YAML para la creación del deploy y el servicio para Wordpress es el siguiente:

```
##############################
## SERVICIO PARA WORDPRESS ##
##############################
```

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-svc
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  #type: LoadBalancer
  type: NodePort
---
################################
## DEPLOYMENT PARA WORDPRESS ##
################################
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
```

```
    labels:
      app: wordpress
      tier: frontend
  spec:
    containers:
    - image: wordpress:4.8-apache
      name: wordpress
      env:
      - name: WORDPRESS_DB_HOST
        value: mysql
      - name: WORDPRESS_DB_PASSWORD
        value: password
      ports:
      - containerPort: 80
        name: wordpress
      volumeMounts:
      - name: wordpress-persistent-storage
        mountPath: /var/www/html
    volumes:
    - name: wordpress-persistent-storage
      persistentVolumeClaim:
        claimName: pvc-wordpress
```

- kubectl apply -f wordpress-deployment.yaml (Creamos el deploy para Wordpress)

- kubectl get pods (Vemos los pods del deploy)

- kubectl get deploy (Vemos el deploy)

- kubectl get rs (Vemos los rs)

EJEMPLO WORDPRESS CON MYSQL - CREACION DEL DEPLOYMENT Y SERVICIO PARA SU PRUEBA DE MYSQL - PARTE 4

- Creamos el deployment para MySQL llamado mysql, con tan solo 1 replica de sus pods, con una serie de etiquetas y selectores, dónde sus pods usarán una imagen basada en MySQL, tendrá 1 variable de entorno cada uno de los contenedores de sus pods para establecer la contraseña del usuario ROOT de MySQL: MYSQL_ROOT_PASSWORD: password, se usará un tipo de estrategia Recreate para que si en el caso que se modifique algo en el deploy mysql todos sus pods antiguos se eliminen y se creen los nuevos perdiendo el servicio temporalmente y además este deploy usará un volumen llamado mysql-persistent-storage que se montará dentro de los contenedores de cada pod en el directorio /var/lib/mysql y donde este volumen usará el pvc-mysql creado anteriormente. Por último, en el mismo fichero se creará un servicio de tipo ClusterIP llamado mysql-svc para probar este MySQL de manera privada dentro del cluster de kubernetes. El fichero YAML para la creación del deploy y el servicio para MySQL es el siguiente:

```
#####################
## SERVICIO MYSQL  ##
#####################
apiVersion: v1
kind: Service
metadata:
  name: mysql-svc
  labels:
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
```

```
    tier: mysql
  clusterIP: None
---
########################
## DEPLOYMENT  MYSQL  ##
########################
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
```

```
        name: mysql
      volumeMounts:
      - name: mysql-persistent-storage
        mountPath: /var/lib/mysql
    volumes:
    - name: mysql-persistent-storage
      persistentVolumeClaim:
        claimName: pvc-mysql
```

- kubectl apply -f mysql-deployment.yaml (Creamos el deploy para MySQL)

- kubectl get pods (Vemos los pods del deploy)

- kubectl get deploy (Vemos el deploy)

- kubectl get rs (Vemos los rs)

- Para probar el servicio de Wordpress, a través del navegador del nodo1 Master acceder a : http://192.168.1.85:30651 y veremos la pagina de Wordpress, que tendrá en /home/kuberentes/datos/mysql y en /home/kubernetes/datos/wordpress los directorios NFS compartidos que equivale a wordpress y la BD MySQL de los directorios de los contenedores de los pods /var/lib/mysql y /var/www/html.

EJEMPLO WORDPRESS CON MYSQL - ESCALAR WORDPRESS - PARTE 5

- Hasta ahora solo tengo 1 replica del deploy wordpress.

- kubectl scale --replicas=4 deploy/wordpress (Escalo el deploy Wordpress y le digo que quiero 4 replicas de sus pods)

- De esta manera podemos escalar wordpress sin problema. Recordar que MySQL no se debería replicar.

ALMACENAMIENTO DINÁMICO - STORAGE CLASS - CREAR UNA SC

- Almacenamiento que va a estar solicitado de manera dinámica por los PODS.

- Las storage class definen qué aprovisionador se debe usar y qué parámetros se deben pasar a ese aprovisionador cuando se invoca el aprovisionamiento dinámico.

- A continuación, voy a crear un storage class llamada bbdd, que tendrá una anotación, usará un provisioner para trabajar de manera local (kubernetes.io/no-provisioner), el tipo de reclamación será Delete y el modo de vinculación de volumen será WaitForFirstsConsumer (es decir, no se va a relacionar un PV con un PVC hasta que no tenga un POD). El fichero YAML del storage class es:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    description: Esto es un ejemplo de Storage Class
  name: bbdd
provisioner: kubernetes.io/no-provisioner
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

- kubectl apply -f storage-class.yaml (Creo el SC hacienda uso del fichero YAML)

- kubectl get sc (Vemos las storage class existentes)

CREAR PV Y PVCs asociados a Storage Class

- Creo un PV llamado pv-bbdd-vol1, que tendrá de stroageClassName bbdd (misma sc que la creada anteriormente), con una capacidad de 10GB y usará un driver de tipo hostPatch que tendrá como directorio el /tmp/bbdd-vol1 de la máquina física/local. El fichero YAML es:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-bbdd-vol1
spec:
  storageClassName: bbdd
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/bbdd-vol1"
```

- kubectl apply -f pv.yaml (Creo el PV haciendo uso del fichero YAML)

- kubectl get pv (Veremos el PV)

- Luego, crearemos otro PV llamado pv-bbdd-vol2 que será igual que el PV anterior pero con menos capacidad (2GB), que pertenecerá a la misma SC (bbdd) pero su hostPath apuntará a otro directorio de la máquina física /tmp/bbdd-vol2. El fichero YAML es:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-bbdd-vol2
spec:
```

```
  storageClassName: bbdd
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/bbdd-vol2"
```

- kubectl apply -f pv1.yaml (Creo el PV haciendo uso del fichero YAML)

- kubectl get pv (Veremos el PV)

- Tras crear los 2 PVs, vamos a crear un PVC que usará los dos PVS anteriores, ya que este PVC llamado pvc-bbdd usará la SC bbdd, tendrá 2 GB de capacidad y un modo de acceso ReadWriteOnce. En este caso, el PV que se va a asociar al PVC será el PV pv-bbdd-vol2, ya que se ajusta mejor al espacio disponible en el PVC. El fichero YAML será:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-bbdd
spec:
  storageClassName: bbdd
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

- kubectl apply -f pvc.yaml (Creo el PVC haciendo uso del fichero YAML)

- kubectl get pvc (Veremos el PVC y veremos que estará pendiente debido a que aun no se ha creado un Pod ya que la SC bbdd tiene el modo WaitForFirstConsumer)

CREAR UN POD USANDO UN Storage Class

- Creo un pod llamado postores-db1 que usará una imagen de Postgres 11, tendrá configuradas 3 variables de entorno para la password de Postgres, el usuario de postgres y la BD de postgress (básicamente crea una BD, un usuario y sus contraseña), usará un volumen llamado postgres-db-volume1 que se montará en el directorio /var/lib/postgresql/data del contenedor del Pod y ese volumen usará el PVC pvc-bbdd creado anteriormente. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  name: postgres-db1
spec:
      containers:
      - image: postgres:11
        name: postgres-db1
        ports:
        - containerPort: 5432
          protocol: TCP
        volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgres-db-volume1
        env:
         - name: POSTGRES_PASSWORD
           value: "secret"
         - name: POSTGRES_USER
           value: "odoo"
```

```
        - name: POSTGRES_DB
          value: "postgres"
      volumes:
      - name: postgres-db-volume1
        persistentVolumeClaim:
          claimName: pvc-bbdd
```

- kubectl apply -f postgres.yaml (Creo el Pod haciendo uso del fichero YAML)

- kubectl get pods (Verems el nuevo pod)

- kubectl get pvc y kubectl get pv (Ahora veremos que el PVC ya esta asociado al pv pv-bbddd-vol2)

- Tras ver que el nodo que ejecuta el pod postgres-db1 es el nodo curso-kubernetes2, podemos ver el directorio /tmp/bbdd-vol2 con toda la info de Postgres. Además, en el directorio /var/lib/postgresql/data del contenedor del pod habrá el mismo contenido.

LABORATORIO PRÁCTICO CON UNA SC DINÁMICA

- Creamos una nueva SC llamada sc-discos-locales, con un volumeBindingMode inmediato y con una política de reclamación Delete. El fichero YAML es:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: sc-discos-locales
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: Immediate
reclaimPolicy: Delete
```

- kubectl apply -f sc2.yaml (Creamos la SC)

- kubectl get sc (vemos la sc)

- Tras esto, creamos tres PVS que tendrán como storageClass sc-discos-locales (creada anteriormente), distinta capacidad de almacenamiento (4GB, 1GB y 6GB), y usarán un driver de tipo hostpath haciendo uso de directorio /mnt/datos de la maquina física. El fichero YAML para la creación de estros 3 PVs ES:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: volume1
  labels:
    type: local
    curso: kubernetes
spec:
  storageClassName: sc-discos-locales
  capacity:
    storage: 4Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/datos"

---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: volume2
  labels:
    type: local
    curso: kubernetes
```

```
spec:
  storageClassName: sc-discos-locales
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/datos"

---

apiVersion: v1
kind: PersistentVolume
metadata:
  name: volume3
  labels:
    type: local
    curso: kubernetes
spec:
  storageClassName: sc-discos-locales
  capacity:
    storage: 6Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/datos"
```

- kubectl apply -f pvs.yaml (Creamos los 3 PVC haciendo uso del fichero YAML anterior)

- kubectl get pv (vemos los pvs)

- Tras ello, creamos el PVC llamado pvc-dinamico de 3GB de capacidad, con una clausula SC "sc-discos-locales" y que se asociará con el PV volume1 ya que es el PV con una capacidad más cercana a la capacidad del PVC pvc-dinamico. El fichero YAML del PVC es:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-dinamico
spec:
  storageClassName: sc-discos-locales
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

- kubectl apply -f pvcs.yaml (Creamos el PVC haciendo uso del fichero YAML anterior)

- kubectl get pvc (vemos los pvcs)

GENERAR PV AUTOMÁTICOS

- Creamos una nueva SC llamada sc-dinamica, con un volumeBindingMode inmediato y con una política de reclamación Delete. Además, el provisioner k8s.io/minikube-hostpath permite crear un directorio local en alguno de los nodos, donde se creará un PV de forma automática. El fichero YAML es:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
```

```
   name: sc-dinamica
provisioner: k8s.io/minikube-hostpath
volumeBindingMode: Immediate
reclaimPolicy: Delete
```

- kubectl apply -f sc3.yaml (Creamos la SC)

- kubectl get sc (vemos la sc)

- Al usar storage class dinámica con un provisioner como es k8s.io/minikube-hostpath no hace falta crear un PV antes del PVC porque directamente ese provisioner del SC crea un PV en alguno de los nodos.

- Es por ello, que directamente crearemos el PVC, que se llamará pvc-dinamico, con una capacidad de 3GB, con un modo de acceso ReadWriteOnce y que se asociara a un PV que sea de la clase sc-dinamica. El fichero YAML del PVC es:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-dinamico
spec:
  storageClassName: sc-dinamica
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

- kubectl apply -f pvc-dianmico.yaml (Creamos el PVC)

- kubectl get pvc (Vemos los pvc y veremos que esta asociado a un PV con un nombre raro, ya que ese PV se ha creado automáticamente gracias al provisioner

de la SC. Pero vemos como de forma automática, gracias al provisioner minikube-hostpath se ha creado un PV dinámico y automático y se ha asociado al PVC que tiene la clase SC del provisioner dinámico)

LABORATORIO AWS Y DISCOS ESB - INSTALAR DRIVER CSI - CRAER VOLUMEN EBS, ROLES Y POLÍTICAS - CREAR PV Y PVC

- Para más info acerca de esto ver los vídeos 202, 203 y 204 debido a que este laboratorio usa AWS y AWS es de pago.

WORKLOADS

- El pod es el workload más básico de kubernetes.

- Otros workloads son, los Stateffulsets, deployments, DaemonSets, Jobs, CronJobs, etc.

- DEPLOYMENTS: Componente que va a orquestar el despliegue de los pods, las réplicas, las recuperaciones ante catástrofes, etc. En definitiva, gestiona u orquesta los despliegues de nuestras aplicaciones. Son los componentes que van a envolver a los pods, y que además les da una serie de características, como por ejemplo: hacer updates y rollbacks de manera sencilla y además comprobar o mantener que funciona correctamente.

- REPLICASETS: Son los encargados de hacer el escalado de los pods cuando es necesario, es decir, subir o bajar los pods, de acuerdo a lo que el clúster tenga indicado. Es decir, cuando yo determino que deben haber 5 pods, los replicasets van a intentar a toda costa, mantener esos 5 pods funcionando puesto que es el mandato que se le ha dado.

- STATEFUL SET: Objeto que gestiona el despliegue y el escalado de los pods y garantiza el orden y la unicidad de esos pods. Soporte para aplicaciones con

estado y también para aplicaciones con alta disponibilidad. Estas apps deben almacenar datos y ocuparse de que se guarden y se gestionen correctamente. Por ejemplo, las BD, que son claros ejemplos de apps con estados. Los statefullsets son muy similares a los deployments, pero los statefulset asigna un nombre único a cada pod (ya que los pods de un deploy reciben un nombre raro y aleatorio). Los statefullsets no son intercambiables, si se borrar uno de sus pods se vuelve a recrear ese pod, si se baja el numero de replicas de los pods statefulsets siempre se comienza por el último. Además, los pods de tipo Statefullsets no comparten datos, cada uno de ellos maneja su propio volumen persistente.

- DAEMON SET: Un componente que asegura que todos los nodos del clúster van a tener una copia de un Pod. Si yo por ejemplo, añado un nuevo nodo al clúster, el daemon set intentará a toda costa que haya un pod dentro del mismo. Es decir se encarga que los pods estén presentes en todos los nodos de clúster. Para entornos de tipo log o monitorización.

- JOB: Componente que crea uno o varios pods y se van a asegurar que un número determinado de ellos se terminen satisfactoriamente. Es decir que se encarga que los pods funcionen correctamente y cuando el número que yo le he indicado de terminaciones correctas se alcance, el job se habrá terminado. Utilizado para backups y demás.

- CRON JOB: Igual que los JOBSs pero de forma planificada con scheduler.

- CUSTOM RESOURCE: Creados de forma personalizada. Crearnos nuestros propios workloads.

STATEFULLSETS

- STATEFUL SET: Objeto que gestiona el despliegue y el escalado de los pods y garantiza el orden y la unicidad de esos pods. Soporte para aplicaciones con estado y también para aplicaciones con alta disponibilidad. Estas apps deben

almacenar datos y ocuparse de que se guarden y se gestionen correctamente. Por ejemplo, las BD, que son claros ejemplos de apps con estados. Los statefullsets son muy similares a los deployments, pero los statefulset asigna un nombre único a cada pod (ya que los pods de un deploy reciben un nombre raro y aleatorio). Los statefullsets no son intercambiables, si se borra uno de sus pods se vuelve a recrear ese pod, si se baja el numero de replicas de los pods statefulsets siempre se comienza por el último. Además, los pods de tipo Statefullsets no comparten datos, cada uno de ellos maneja su propio volumen persistente. Es decir, cada replica de los Pods de un StateFulSet tendrá su propio PV y su propio PVC. Si se añade una nueva repica de un pod, siempre se copia del pod anterior. Cada POD statefulset tiene su propio nombre DNS dentro del servicio (POD0.svc, POD1,svc, etc.).

EJEMPLO CON MONGO - CREAR SC Y SERVICIO

- Creamos una SC llamada estado, con el provisioner minikube-hostpath que creará un PV de forma automática y que tendrá como parámetros type = pd-ssd. El fichero YAML de la SC será el siguiente:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: estado
provisioner:  k8s.io/minikube-hostpath  #  solo  funciona  en
minikube, en kubeadm no funciona
parameters:
  type: pd-ssd
```

- kubectl apply -f sc-mongo.yaml (Creamos la SC haciendo uso del fichero YAML)

- kubectl get sc (Vemos la SC)

- Una vez creada la SC anterior, crearemos un servicio llamado mongodb-svc de tipo ClusterIP. El fichero YAML es el siguiente:

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-svc
  labels:
    app: db
    name: mongodb
spec:
  ports:
  - port: 27017
  clusterIP: None
  selector:
    app: db
    name: mongodb
```

- kubectl apply mongo-svc.yam (Creamos el servicio haciendo uso del fichero YAML)

- kubectl get svc (Vemos el servicio recién creado de Mongo)

EJEMPLO CON MONGO - CREAR EL STATEFULSET

- Una vez creado la SC y el SVC de mongo, creamos el statefulset que se llamará mongo, tendrá unos selectores y etiquetas, tendrá 3 replicas de sus pods donde cada uno de ellos ejecuta una imagen MongoDB, que será arrancada con el comando "mongod" y los argumentos --bind_ip=0.0.0.0 (para que cualquier IP sea valida para entra a la BD), --replSet=rs (Nombre del replica set con los que van a reconocer la replica todos los nodos del clúster, en este caso rs0) y --dbpath=/data/db (Directorio donde va a ir a parar la BD). Además, se va a crear un volumen llamado mongo-storage que va a estar asociado al directorio /data/db

de los contenedores de cada pod. También, se va a usar un PVC que usará el volumen mongo-storage, se hará uso en este PVC de la SC estado creada anteriormente (SC que crea un PV automáticamente debido a su provisioner minikub-hostpath) y que tendrá 1 GB de capacidad. Por último, el nombre del servicio que estará asociado al Statefulset (serviceNAme) será el servicio mongodb-svc creado anteriormente. El YAML es:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  selector:
    matchLabels:
      app: db
      name: mongodb
  serviceName: mongodb-svc #Servicio headless
  replicas: 3
  template:
    metadata:
      labels:
        app: db
        name: mongodb
    spec:
      terminationGracePeriodSeconds: 10 # La BD de mongo tendrá
10 segundos para parar
      containers:
      - name: mongo
        image: mongo:3.6
        command:
          - mongod
        args:
          - --bind_ip=0.0.0.0
```

```
          - --replSet=rs0 #Nombre del replica set. Todos los
miembros del cluster usan este nombre
          - --dbpath=/data/db
        livenessProbe: # Sondas
            exec:
              command:
                - mongo
                - --eval
                - "db.adminCommand('ping')"
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-storage
          mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: mongo-storage
      spec:
        storageClassName: estado
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

- kubectl apply -f mongo-statefullset.yaml (Creamos el StateFullSet haciendo uso del fichero YAML)

- kubectl get statefulset (Vemos el StateFullset)

- kubectl get pods (Veremos las 3 replicas de los pods de StateFulSet creados, y veremos que cada pod se va creando de forma individual, primero se crea el pod mongo-0, cuando se crea, se crea el mago-1 y luego el mongo-2)

- kubectl get pvc (Veremos que se han creado 3 PVCS, uno por cada réplica de cada pod)

- kubectl get pv (Veremos que se han creado 3 PVS, uno por cada réplica de cada pod)

EJEMPLO CON MONGO - ARRANCAR LA RÉPLICA

- En este punto, se deberá entrar en 1 de las 3 replicas de los pods de mongo y ejecutar un comando para iniciar las 3 religas del mongo. El comando será:

```
rs.initiate({_id: "rs0", version: 1, members: [{ _id: 0, host :"mongo-0.mongodb-svc:27017" },{ _id: 1, host: "mongo-1.mongodb-svc:27017" }, { _id: 2, host: "mongo-2.mongodb-svc:27017" } ]});
```

- Entramos por ejemplo en el pod1 de mOngo: kubectl exec -it mongo-0 -- bash y una vez dentro ejecutamos el cliente de mongoDB: mongo. Una vez dentro del contenedor del pod y tras haber ejecutado el cliente mongo, se ejecutaría el comando:

```
rs.initiate({_id: "rs0", version: 1, members: [{ _id: 0, host :"mongo-0.mongodb-svc:27017" },{ _id: 1, host: "mongo-1.mongodb-svc:27017" }, { _id: 2, host: "mongo-2.mongodb-svc:27017" } ]});
```

- Una vez que se han iniciado las replicas haciendo lo anterior, lanzamos un deployment que ejecuta 1 replica de sus pods y ejecuta una imagen de mongo-express. Mongo exprés es un mongo con una interfaz web. Además, este deploy se conecta al primer pod a través de us nombre de servicio: mongo-0.mongodb.svc. Además, en este mismo fichero se crea un servicio de tipo

NodePort para probar este deploy (NO HE COPIADO AQUÍ EL FICHERO YAML).

- kubectl apply -f client-mongo-express.yaml (Creamos el deploy y servicio para Mongo Express haciendo uso del fichero YAML que no he copiado aquí)

- Si ahora accedo a través del servicio que he creado al pod mongo-express de esta manera desde una navegador: 192.168.2.129:30570 y veré las diferentes BD.

ESCALAR, DESECALAR Y BORRAR PODS EN UN STATEFULSET

- kubectl scale statefulset mongo --replicas=4 (Escalo añadiendo una replica mas al Statefulset mongo)

- kubectl get pods (El nuevo pod se llamará mongo-3, ya que a cada replica de pods se le asigna un numero único para identificar cada réplica con un orden)

- kubectl scale statefulset mongo --replicas=3 (Si yo ahora desescalo el numero de replicas, quitando un pod, el que se elimina será el pod mongo-3, que fue el ultimo en crearse)

- kubectl delete pod mongo-1 (Pasa igual que con los deployments, al intentar borrar el pod, se borra, pero se recrea un nuevo pod automáticamente debido a que el estado deseado es de 3 religas. La diferencia con los deployments es que con los Statefulsets cuando se recrea el pod, se recrea con el mismo nombre, es decir, si se borro el pod mongo-1 se recrea de nuevo el pod mongo-1, con el mismo nombre. Además al borrarlo, la BD mongo sigue activa ya que el PV sigue existiendo de manera persistente)

SERVICIOS EN LOS STATEFULSETS

- kubectl exec -it mongo-0 -- bash (Entro al primer pode mongo, y si vemos el contenido del fichero /etc/resolv.conf veremos el dominio curso.svc.cluster.local)

- Además, una vez dentro del contenedor si ejecutamos: nslookup mongodb-svc veremos las 3 IPs de los 3 pods del statefulset de mongo. Y además. Si ejecuto: nslookup mngo-1.mongodb-svc me da solo la IP del pod mongo-1.

DAEMONSETS

- Cuando yo lanzo un Demonsets, genera un pod en cada nodo que tenga en el clúster. De hecho si escalo el cluster y añado mas nodos, automáticamente se tendrá una copia del pod en los nuevos nodos.

- Los daemonsets se utilizan para ejecutar procesos que se deben lanzar en cada nodo, por ejemplo para un backup. También se utiliza para monitoreo.

- Con los DemonSets no se pueden lanzar servicios.

- Acto seguido, creamos un DaemonSet llamado log-daemonset con una serie de selectores y etiquetas y que ejecutará una imagen de logstash (herramienta para la administración de logs. Se utiliza para analizar y guardar los logs para futuras búsquedas). En este caso, como yo dispongo de 3 nodos en mi cluster de kubernetes, se van a generar 3 pods, 1 en cada nodo. El fichero YAML es el siguiente:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: log-daemonset
```

```
spec:
  selector:
    matchLabels:
      name: log
  template:
    metadata:
      labels:
        name: log
    spec:
      containers:
      - name: log
        image: logstash:8.4.1
```

- kubectl apply -f logstash.yaml (Creams el daemonset haciendo uso del fichero YAML)

- kubectl get daemonset (Vemos el daemonset)

- kubectl get pods -o wide (Vemos los3 pods ejecutándose en cada nodo)

- minikube node add -p cluster1 (Añado un nuevo nodo a mi cluster cluster1)

- kubectl get pods -o wide (Vemos que hay un nuevo pod, debido a que se ha añadido un nuevo nodo. Ademas este nuevo pod se estará ejecutando en e nuevo nodo creado, cluster1-m04)

JOBS

- Componente que crea uno o varios pods y se van a asegurar que un número determinado de ellos se terminen satisfactoriamente. Es decir que se encarga que los pods funcionen correctamente y cuando el número que yo le he indicado de terminaciones correctas se alcance, el job se habrá terminado. Utilizado para backups y demás.

- Creo un job llamado job1, con 1 replica de sus pods que ejecutarán una imagen de alpine (linux muy ligero) y se ejecutará dentro del contenedor del pod el comando: echo Estas en la maquina con el nombre $(uname -n). El fichero YAML es:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job1
spec:
  template:
    spec:
      containers:
      - name: job1
        image: alpine
        command: ["sh","-c"]
        args: ["echo Estas en la maquina con el nombre $(uname -
n)"]
      restartPolicy: Never
```

- kubectl apply -f job.yaml (Creo el job haciendo uso del fichero YAML)

- kubectl get job (Vemos los jobs)

- kubectl get pods (Vemos el job finalizado y completado)

- kubectl logs job1-22lpv (Veremos el nombre de la maquina, salida del comando echo del container alpine)

- Si yo quiero modificar el fichero YAML del Job, no podré hacer un apply, sino tengo que hacer un delete y volver a ejecutar el apply. Esto es debido a que el job ya ha terminado, por lo tanto, no puedo volver a lanzar un job.

JOBS CON VARIAS RÉPLICAS

- Creo un job llamado job2, con 3 replicaS de sus pods (COMPLETIONS: 3, se lanzarán 3 jobs) que ejecutarán una imagen de alpine (linux muy ligero) y se ejecutará dentro del contenedor del pod el comando: echo Estas en la maquina con el nombre $(uname -n). El fichero YAML es:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job2
spec:
  completions: 3
  template:
    spec:
      containers:
      - name: job1
        image: alpine
        command: ["sh","-c"]
        args: ["echo Estas en la maquina con el nombre $(uname -
n)"]
      restartPolicy: Never
```

- kubectl apply -f job2.yaml (Creo los jobs haciendo uso del fichero YAML)

- kubectl get job (Vemos los jobs)

- kubectl get pods (Vemos los job finalizados y completados. Se lanzan de forma secuencial, se lanza el job1, se termina, se lanza el job2 se termina ...)

- Como vemos, se han completado los 3 jobs de forma secuencial.

- Ahora, creo un job llamado job3, con 3 replicaS de sus pods (COMPLETIONS: 3, se lanzarán 3 jobs) que ejecutarán una imagen de alpine (linux muy ligero),

además 2 de ellos se lanzarán en paralelo (parallelism: 2) y el job restante se lanzará de forma secuencial y se ejecutará dentro del contenedor del pod el comando: echo Estas en la maquina con el nombre $(uname -n). El fichero YAML es:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job3
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
      containers:
      - name: job1
        image: alpine
        command: ["sh","-c"]
        args: ["echo Estas en la maquina con el nombre $(uname -
n)"]
      restartPolicy: Never
```

- kubectl apply -f job3.yaml (Creo los jobs haciendo uso del fichero YAML)

- kubectl get job (Vemos los jobs)

- kubectl get pods (Vemos los job finalizados y completados. Vemos que 2 de ellos se han completado y lanzado al mismo tiempo y el tercer job se ha lanzado y completado una vez que se hayan completado los 2 anteriores, lanzándose de forma secuencial el último job.)

JOBS CON PROBLEMAS

- Si un job falla va a intentar volver a ejecutarse.

- Para ello, creamos un job llamado job4, con 1 replica de sus pods que ejecutarán una imagen alpine y que el contenedor en su interior ejecutará el comando: cat /etc/puff.txt (dará error porque el fichero puff.txt no existe). El fichero YAML es:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job4
spec:
  template:
    spec:
      containers:
      - name: job1
        image: alpine
        command: ["sh","-c"]
        args: ["cat/etc/puff.txt"]
      restartPolicy: Never
```

- kubectl apply -f job4.yaml (Creamos el job4)

- kubectl get job (Vemos los jobs)

- kubectl get pods (Vemos los pods y veremos que como da error, se crea un job termina con error, se crea otro job, termina con erro y así sucesivamente)

- Para evitar que se creen jobs infinitos, se usa la cláusula backoffLimit:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job4
spec:
  template:
    spec:
      backoffLimit: 2 # Si falla el Job que se creen 2 JOBS y que
pare si sigue fallando
      containers:
      - name: job1
        image: alpine
        command: ["sh","-c"]
        args: ["cat/etc/puff.txt"]
      restartPolicy: Never
```

- kubectl delete job job4 (Borro el job)

- kubectl apply -f job4.yaml (Recreamos el job4 y veremos que se lanzar ale primer Job fallara, se lanzara el segundo fallará y ya no se ejecutan mas JOBS debido a que he puesto la cláusula backoflimit: 2)

- Hay otra propiedad llamada activeDeadlineSeconds, que sirve para si el job no se completa correctamente y pasan 15 segundos, pues entonces se para y no se siguen creando más Jobs. Un ejemplo de esta cláusula podría ser:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job4
spec:
  template:
```

```
    spec:
      activeDeadlineSeconds: 15 # Establecida a 15 segundos en
este caso.
      containers:
      - name: job1
        image: alpine
        command: ["sh","-c"]
        args: ["cat/etc/puff.txt"]
      restartPolicy: Never
```

CRONJOBS - TRABAJOS PLANIFICADOS

- Los cornJobs son JOBS pero se ejecutan de manera planificada, es decir, cada cierto tiempo. Para ello se utiliza la sintaxis Cron (Minute(0-59) Hour(0-23) DayMonth(1-31) Month(1-12) DayWeek(0-6)).

- El "*" significa "cualquiera".

CRONJOB - EJEMPLO

- Creamos un CronJob llamado cron-job1 que ejecutará un job cada mes, cada día del mes, cada día de la semana, cada hora y cada minuto (básicamente se lanzará cada minuto un Job, schedule: * * * * *). El job ejecutará 1 replica de sus pods basado en la imagen alpine y en él ejecutará el comando "echo Me ejecuto en este momento -- $(date)", que imprimirá la fecha exacta. El fichero YAML del cronjob será el siguiente:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cron-job1
```

```
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: job1
            image: alpine
            command: ["sh","-c"]
            args: ["echo Me ejecuto en este momento -- $(date)"]
          restartPolicy: Never
```

- kubectl apply -f cron-job.yaml (Creamos el cron-job)

- kubectl get cronjob (Vemos los jobs)

- kubectl get pods (Vemos los pods del CronJob, que se lanzará cada minuto. Es decir que si pasan 45 minutos habrá 45 jobs/pods completados)

INTRODUCCIÓN A LAS SONDAS

- Es un componente que puedo asociar a un pod y con el cual puedo comprobar que está funcionando correctamente. Hay 2 tipos de sondas (liveness y readiness). También hay sondas de tipo command, de tipo socket y de tipo HTTP.

- Las sondas permiten realizar diagnósticos sobre los contenedores de Kubernetes.

- Existen 3 tipos de sondas por el tipo de prueba que lanzan:

  * COMANDO (Para ejecutar comandos en un contenedor)

* HTTP GET (Para aplicaciones frontales)

* SOCKET (Para escuchar por un puerto)

- Existen 3 tipos de respuestas:

* SUCCESS

* FAILURE

* UNKNOWN

- Además, tenemos 3 tipos de comportamientos de las sondas:

* LIVENESS (si la sonda no funciona se va a matar el contenedor y dependiendo de su política de arranque pues arrancará, se quedará parado, etc.)

* READINESS (se intenta comprobar si el servicio está funcionando, no se mira si el contenedor está funcionando sino si está funcionando la aplicación/servicio que está dentro de ese contenedor)

* STARTUP

SONDAS LIVENESS DE TIPO COMMAND - COMPROBAR SI NUESTRO POD FUNCIONA

- Creamos un pod llamado sonda-liveness que ejecuta una imagen de ubuntu y en el contenedor del pod se ejecuta el comando mkdir /tmp/prueba; sleep 30; rm -rf /tmp/prueba; sleep 600 (Es decir, se crea un directorio, espera 30 segundos y la sonda funcionará porque el directorio existe, borra el directorio creado y espera otros 60 segundos y ahora ya la sonda no funcionará debido a que el directorio /tmp/prueba no existirá). Este pod tendrá una sonda de tipo COMMAND y se

comporta de manera Liveness (es decir, si la sonda no funciona se va a matar el contenedor y dependiendo de su política de arranque pues arrancará, se quedará parado, etc.), esta sonda de tipo COMMAND ejecutará el comando ls /tmp/prueba y habrá un periodo de 30 segundos en que sí funcionará (RESPUESTA SUCCESS) y después de 30 segundos la sonda dejará de funcionar y se eliminará el contenedor debido al comportamiento Lívenes de la sonda y como la restarPolicy es always (por defecto si no se indica) el contenedor intentará arrancar de nuevo. Por último tenemos 2 propiedades: initialDelaySecond (tiempo que tarda la sonda en ponerse en marcha. ESTO ES MUY IMPORTANTE SIEMPRE PONERLE UN DELAY) y periodSeconds (cada cuanto tiempo está la sonda preguntando). El fichero YAML del pod con la sonda es el siguiente:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: sonda-liveness
spec:
  containers:
  - name: pod-liveness
    image: ubuntu
    args:
    - /bin/sh
    - -c
    - mkdir /tmp/prueba; sleep 30; rm -rf /tmp/prueba; sleep 600
    livenessProbe: # SONDA LIVENESS DE TIPO COMMAND
      exec:
        command:
        - ls
        - /tmp/prueba
      initialDelaySeconds: 5
```

```
      periodSeconds: 5
```

- kubectl apply -f sonda1.yaml (Creamos el pod con la sonda Liveness command haciendo uso del fichero YAML)

- kubectl get pods (Veremos que cada cierto tiempo el pod se reincida debido a que habrá un periodo que la sonda funciona y otro periodo en el que no, y como es Liveness se mata al contenedor del pod, pero clareo, al tener el pod la política de restart a Always se recrea el contenedor de nuevo, por eso se reinicia)

SONDAS LIVENESS DE TIPO HTTP - COMPROBAR SI NUESTRO POD FUNCIONA

- Creamos un DOCKERFILE para crear una imagen a utilizar por un contenedor. Para ello se descarga una version de APACHE, se copia el contenido del directorio "web" de la maquina física en el directorio /usr/local/apache2/htdocs del contenedor de Docker y expone el puerto 80. El dockerfile es:

```
##Descargamos una versión concreta de APACHE, a través del tag
FROM httpd:2.4
COPY ./web/ /usr/local/apache2/htdocs/
MAINTAINER Apasoft Formacion "apasoft.formacion@gmail.com"
##Exponemos el Puerto 80
EXPOSE 80
```

- docker build -t danilooo99/sonda-web . (Construimos la imagen)

- docker login (Iniciamos sesión en Docker HUB)

- docker push danilooo99/sonda-web (Subimos la imagen a Docker Hub)

- Una vez que hayamos creado la imagen de Docker y la hayamos subido a Docker Hub, ahora crearemos un deploy llamado web-d, que ejecutará 1 replica de sus pods que ejecutará la imagen recién creada y subida a DockerHUB. Este deploy usará una sonda Liveness de tipo HTTP GET y comprobará si el PATH /sonda.html es correcto, ósea que mientras exista /sonda.html el resultado será OK.. La sonda tardará 3 segundos en ponerse en marcha y además cada 3 segundos la sonda estará preguntando. El fichero YAML es

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: web-d
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: web
  replicas: 1 # indica al controlador que ejecute 2 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web-d-777b977ff9
        image: danilooo99/sonda-web:latest
        ports:
        - containerPort: 80
        livenessProbe:
            httpGet:
                path: /sonda.html
                port: 80
```

```
            initialDelaySeconds: 3
            periodSeconds: 3
```

- kubectl apply -f sonda2.yaml (Creamos el deploy con la sonda Liveness http get haciendo uso del fichero YAML)

- Para probar el deploy anterior se ha creado un servicio de tipo Nodeport llamado web-sec. El fichero YAML del servicio es:

```
apiVersion: v1
kind: Service
metadata:
  name: web-svc
  labels:
     app: web
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30002
    protocol: TCP
  selector:
     app: web
```

- kubectl apply -f web-svc.yaml (Creamos el servicio haciendo uso del fichero YAML)

- Si ahora accedemos a la URL: http://192.168.2.129:32133 veremos la página index HTML.

- kubectl exec -it web-d-857c579bf9-9dd6m -- bash (Entramos al pod y veremos que en el directorio /usr/local/apache2/htdocs estarán los ficheros index y sonda HTML. Si yo borrara el fichero sonda.html entonces la sonda fallaría y al ser Liveness el contenedor se eliminar pero se reinicia al tener una política de restart Always)

SONDAS READINESS DE TIPO SOCKET - COMPROBAR SI NUESTRA APP FUNCIONA

- CREAMOS UNA IMAGEN TOMCAT con una app que se desplegará en él (app1.war) PARA USARLA POSTERIORMENTE. El fichero DockerFile es el siguiente:

```
FROM centos

MAINTAINER apasoft.training@gmail.com

RUN mkdir /opt/tomcat/

WORKDIR /opt/tomcat # Es como hacer un cd /opt/tomcat dentro del container (directorio de trabajo)
RUN curl -O https://apache.rediris.es/tomcat/tomcat-8/v8.5.51/bin/apache-tomcat-8.5.51.tar.gz
RUN tar xvfz apache*.tar.gz
RUN mv apache-tomcat-8.5.51/* /opt/tomcat/.
RUN yum -y install java
RUN java -version

WORKDIR /opt/tomcat/webapps
COPY app1.war .
EXPOSE 8080
```

- docker build -t danilooo99/tomcat . (Construimos la imagen)

- docker login (Iniciamos sesión en Docker Hub)

- docker push danilooo99/tomcat (Subimos la imagen a DockerHub)

- Luego, creamos un deploy con 3 replicas de sus pods que ejecutaran la imagen creado anteriormente. Este deploy usa una sonda de tipo readiness con nun socket que escucha en el puerto de la aplicación tomcat el 8080. Esta sonda readiness comprueba por el TCP socket el puerto 8080. Si el puerto 8080 no esta disponible dará un warning pero no se cae pero si lo quita del endpoint del servicio. El fichero YAML es:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: tomcat-d
  labels:
     app: tomcat
spec:
  selector:    #permite seleccionar un conjunto de objetos que cumplan las condicione
    matchLabels:
      app: tomcat
  replicas: 3 # indica al controlador que ejecute 2 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
      - name: tomcat
```

```
        image: danilooo99/tomcat
        command:   ['sh',   '-c',   '          sleep   40   &&
/opt/tomcat/bin/catalina.sh run'] # Ejecuto el comando sleep 40
(para que se duerma 40 segundos) y ejecuto el script Catalina de
tomcat para iniciar Tomcat (ahora si funcionará sonda)
        ports:
        - containerPort: 8080
        readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 0
            periodSeconds: 5
```

- kubectl apply -f tomcat.yaml (Creamos el deploy con el YAML)

- kubectl get pods (Veremos un 0/1 en los pods, pero tras pasar los 40 segundos del sleep se arrancará el tomcat y si funcionará la sonda y los pods)

- Además, también creamos un servicio de tipo NodePort para probar la app de Tomcat. El fichero YAML del servicio es:

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-svc
  labels:
     app: tomcat
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 30005
    protocol: TCP
```

```
  selector:
     app: tomcat
```

- kubectl apply -f tomcat-service.yaml (Creamos el servicio con el YAML)

RBAC - IMPLEMENTANDO SEGURIDAD EN NUESTRO CLÚSTER

- RBAC - Role-Based Access Control. No existe una API específica para la gestión de usuarios en kubernetes. La gestión de usuarios se realiza de forma externa. Algunas opciones son: certificados, tokens, basic authentication y Oauth2.

- Para entender RBAC necesito entender: Clientes (users and service accounts), recursos (pods, secrets, nodes, pv, services, confirmaos, ingress, etc.) y verbos-operaciones (List, create, watch, get, delete, patch).

- RBAC implementa un concepto que son los roles, y los roles son un conjunto de permisos que identifican lo que se puede hacer con un objeto. El rol identifica recursos y verbos, y siempre tienen que estar asociado a un namespace.

- Un clusterRole es como un rol, también podemos dar los mismos permisos sobre distintos recursos, pero lo vamos a hacer a nivel de clúster, por lo que está en un ámbito superior al del rol. Los clusterole NO son a nivel de namespace, SON a nivel del clúster.

- Para asociar un rol o un cluster Role a un cliente se usan los RoleBindings o los ClusterRoleBinding.

VER ROLES A NIVEL DE NAMESPACE

- Hay algunos roles que están ya presentes en algunos namespaces por defecto.

- kubectl get roles -n kube-system (Vemos los roles del NS kube-system)

- kubectl describe role kube-proxy -n kube-system (Vemos muchas mas info acerca del rol kube-proxy del namespace kube-system)

CREAR ROLES EN UN NAMESPACE

- kubectl create namespace ventas (Creamos una nueva namespace llamada ventas)

- kubectl api-resources (Nos devuelven mis objetos de kuberentes, sus shortnames, etc.)

- Creamos un rol llamado operador y se va a crear dentro del namespace ventas recién creado. Este rol tendrá como reglas, que los recursos que se van a gestionar con este rol serán los pods y los verbos/operaciones que se usarán en ellos serán get, watch y list. Por tanto a los usuarios que se les asigne este rol solo podrán hacer gets, watchs y lists de los objetos pods. El fichero YAML es el siguiente:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ventas # Es obligatorio indicar esto
  name: operador
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

- kubectl apply -f rol.yaml (Creamos el rol operador en el NS ventas haciendo uso del fichero YAML anterior)

- kubectl get roles -n ventas (Vemos los roles de NS ventas)

- kubectl describe role operador -n ventas (Vemos más info acerca del role operador del NS ventas)

VER CLUSTER ROLES - PERMISOS A NIVEL DE CLÚSTER

- kubectl get clusterroles (Vemos los cluster roles)

CREAR CLUSTER ROLES

- Creamos un clusterrole llamado desarrollo, que tendrá como reglas 2 apigroups. El primer apigroup, se podrán hacer get, watch y list de los recursos secrets y confirmas. Mientras que con el segundo apigroup se podrán realizar create, watch, list, get y edits de los recursos pods. El fichero YAML del clusterrole es:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: desarrollo
rules:
- apiGroups: [""]
  resources: ["secrets","configmaps"]
  verbs: ["get", "watch", "list"]
- apiGroups: [""]
```

```
  resources: ["pods"]
  verbs: ["create", "watch", "list","get","edit"]
```

- kubectl apply -f cluster-rol.yaml (Creamos el clusterrol desarrollo haciendo uso del fichero YAML anterior)

- kubectl get clusterrole (Vemos los clusterroles)

- kubectl get clusterrole desarrollo (Vemos el clusterrole desarrollo)

- kubectl describe clusterrole desarrollo (Vemos mas info sobre el clusterrole desarrollo)

CREAR CERTIFICADOS PARA EL ACCESO DE UN USUARIO

- mkdir certs && cd certs (creamos el directorio para los certificados y acedemos a él)

- openssl genrsa -out desa1.key 4096 (Generamos una clave privada para el usuario con openssl y el algoritmo de 4096 bytes rsa. Esta clave se almacenará en el fichero desa1.key)

- Creamos un archivo de configuración para gestionar la petición de solicitud de firma. Le llamamos “desa1.csr.cnf”. ¡ESTO SE PUEDE GENERAR DE FORMA INTERACTIVA!. El fichero es:

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn
[ dn ]
```

```
CN = desa1 # Usuario
O = desarrollo # Grupo
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
```

- openssl req -config desa1.crs.cnf -new -key desa1.key -out desar1.csr (A partir del fichero de configuration desa1.crs.cnf haciendo uso de la clave privada desa1.key me generas un certificate request y los guaranás en el archivo desar1.csr. solicitud de firma)

- Acto seguido, generamos una llamada al cluster de Kubernetes para aprobar o rechazar la petición/solicitud de firma. En este caso se crea un objeto de tipo CertificateSigningRequest y le pasamos desar1.csr codificado en base 64. Para ello, ejecutamos:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: desa1-req-auth
spec:
  groups:
  - system:authenticated
  request: $(cat desar1.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
  - client auth
EOF
```

- kubectl get csr (Comprobamos que el certificado está, el desa1-req-auth)

- kubectl certificate approve desa1-req-auth (Aprobamos el certificado)

- Luego, ejecutamos: kubectl get csr desa1-req-auth -o jsonpath='{.status.certificate}' | base64 --decode > desa1.crt (Descargamos el certificado en el fichero desa1.crt en el directorio certs.)

- Tras esto, dentro del directorio certs hacemos: mkdir .kube

- cd .kube (Accedemos a .kube)

- cp /home/kubernetes/.kube/config . (Copiamos el fichero config de kubernetes en el directorio .kube)

- Modificamos el fichero config que acabamos de copiar en el directorio /certs/.kube y modificamos en el context: el user (desa1), el name (desa1-context) y el current-context (desa1-context). Luego en la sección users ponemos: client-certificate (La ruta absoluta de nuestro desa1.crt) y el client-key (la ruta absoluta de nuestro desa1.key)

- kubectl --kubeconfig=/home/kubernetes/certs/.kube/config get pods (Nos devuelve los pods pero cogiendo la configuración del fichero config del directorio /certs/.kube)

ASOCIAR UN ROL A UN USUARIO

- kubectl get roles -n ventas (Veremos nuestro rol operador creado en secciones anteriores)

- kubectl run nginx --image=nginx (Me creo un pod llamado nginx)

- Para usar la nueva configuración que está en el directorio /certs/.kube sin necesidad de estar todo el rato añadiendo el argumento --kubeconfig=... hacemos: export KUBECONFIG=/home/kubernetes/certs/.kube/config

- Ahora podemos hacer: kubectl get pods (Y ahora se mostrarán los pods con la configuración del fichero /certs/.kube/config, PERO EN ESTE CASO DARA UN ERROR DEBIDO A QUE NO ESTA ASOCIADO UN ROL AL USUARIO)

- Acto seguido, para asociar el rol operador al usuario desa1, creamos un RoleBinding. El fichero del RoleBinding es:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
   name: role-operador
   namespace: ventas
subjects:
- kind: User
  name: desa1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: operador
  apiGroup: rbac.authorization.k8s.io
```

- kubectl apply -f role-binding.yaml (Creamos el rolebinding)

- kubectl get rolebindings -n ventas (Vemos los rolebindings del namespace ventas)

- Ahora si hacemos: kubectl get pods, sí nos funcionará y ya no habrán errores, debido a que el usuario desa1 ahora está asociado a un rol.

- Además, si hacemos: kubectl run nginx1 --imagen=nginx, no creará el pod, ya que el rol operador que tiene el usuario desa1 no permite crear pods, solo listarlos, verlos y mostrarlos.

ASOCIAR UN CLUSTERROLE A UN USUARIO

- Asociamos el clusterRole desarrollo mediante un clusterrolebinding al usuario desa1. El fichero YAML del clusterRolebinding es:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
   name: cluster-role-ejemplo
subjects:
- kind: User
  name: desa1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: desarrollo
  apiGroup: rbac.authorization.k8s.io
```

- kubectl apply -f clusterrole-binding.yaml (Creamos el clusterrolebinding)

- kubectl get clusterrolebindings (Vemos los clusterrolebindings)

- Ahora, si hacemos: kubectl run nginx1 --imagen=nginx, sí creará el pod, ya que el clusterrol le ha dado permisos al usuario desa1 de crear pods, ya que el cluster roles sobre los pods tenia reglas de create, wacth, list, edit y get.

SERVICE ACCOUNTS

- Toda esta parte de los service accounts no la seguí al pie de la letra, por lo que si en algún momento necesito usarlos, mejor es ver los vídeos de nuevo antes Neus seguir este TXT acerca de la sección de service accounts.

- Este recurso provee de una identidad a los procesos que se están ejecutando dentro de un POD. Los SA están asociados a un conjunto de credenciales que se almacenan como un secret. Sirven para autenticarme en la API Server de kubernetes.

- La SA por defecto se llama "default".

SERVICE ACCOUNTS - INTEGRACION CON EL POD

- kubectl get sa (Nos da las services accounts).

- Cuando se crea un POD se monta un volumen con una SA con datos para la autenticación.

COMPROBAR QUE EL SA DEAFULT NO TIENE PERMISOS

- Si yo hago un: kubectl exec -it apache1 -- bash y dentro del container hago un curl -X GET https://kubernetes/api no me dejará acceder a la API

CREAR UN SERVICE ACCOUNT

- Fichero YAML de un SA:

```
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: sa1
  namespace: default
```

- Creamos un secret con este YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: sa1-token
  annotations:
    kubernetes.io/service-account.name: sa1
type: kubernetes.io/service-account-token
```

- Creamos un deploy que utiliza el SA anterior. El fichero YAML del deploy es:

```
apiVersion: apps/v1 # i se Usa apps/v1beta2 para versiones
anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-d
spec:
  selector:    #permite seleccionar un conjunto de objetos que
cumplan las condicione
    matchLabels:
      app: nginx
  replicas: 1 # indica al controlador que ejecute 2 pods
  template:   # Plantilla que define los containers
    metadata:
      labels:
        app: nginx
    spec:
      serviceAccount: sa1 # SERVICE ACCOUNT
      containers:
```

```
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

- Creamos un role con este YAML:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: sa-role
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

- Asociamos el rol con el SA sa1. Este es el YAML:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
   name: binding-sa
   namespace: default
subjects:
- kind: ServiceAccount
  name: sa1
roleRef:
  kind: Role
  name: sa-role
  apiGroup: rbac.authorization.k8s.io
```

- Creamos un token con este YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: sa1-secret
  annotations:
    kubernetes.io/service-account.name: sa1
type: kubernetes.io/service-account-token
```

INGRESS

- Ingress Controller. Es un recurso que nos permite acceder de manera externa a los servicios que hay dentro de kubernetes. Los ingress nos permiten tener un recurso externo, al que yo puedo acceder o bien por IP o bien por nombre, y que me van a llevar a los servicios que tengo dentro de kubernetes. El servicio ingress hay que activarlo y hay varios tipos de Ingress. Ingress no forma parte del core de kubernetes, no forma parte del clúster de kubernetes, pero se puede implementar instalando un ingress controller de un tercero e integrarlo en nuestro clúster de kubernetes para poder crear objetos de tipo Ingress.

- Con Ingress, podemos gestionar los accesos desde el exterior al clúster de kubernetes. También balancea el tráfico entre los endpoints. Implementa encriptación a través de HTTPS.

- Un cliente se relaciona con el objeto Ingress y el Ingress se conecta con un servicio del clúster de kubernetes.

- Algunas soluciones de Ingress son: Google cloud, Amazon, F5, nginx (muy popular, potente y es el que utilizaremos), traefik, countour, etc.

ACTIVAR EL CONTROLADOR INGRESS NGINX EN MINIKUBE

- minikube addons enable ingress (Añadimos/activamos el plugin ingress en Minikube)

- minikube addons list (Listamos los plugins de minikube y podemos ver si el componente ingress está activado)

- kubectl get ns (Veremos un nuevo namespace llamado ingress-nginx que estará active)

- kubectl get all -n ingress-nginx (Veremos que hay 3 nuevos pods en el namespace ingress-nginx, 2 servicios, 1 deployment, 1 replicaste y 2 jobs)

OPCIONES DE LOS INGRESS

- EJEMEPLO FICHERO YAML Y SUS OPCIONES (VER PDF DE LA CLASE 244 QUE ESTA TODO BIEN EXPLICADO AHÍ):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: ingress-apache # Nombre del Ingress
 annotations:
  nginx.ingress.kubernetes.io/rewrite-target: / # Sirven para
pasar opciones al Ingress
spec:
 ingressClassName: nginx-example # Clase de Ingress
 rules: # Reglas del Ingress
 - http:
    paths:
    - path: /testpath # Paths usados por el Ingress
      pathType: Prefix # Su tipo
```

```
      backend: # Servicio y puerto asociado al PATH
       service:
        name: test
        port:
         number: 80
```

- Hay 3 tipos de PATH:

  * ImplementationSpecific

  * Exact

  * Prefix

EJEMPLO CON UN SERVICIO

- kubectl run apache2 --image=httpd (Creamos un pod de apache llamado apache2)

- kubectl expose pod apache2 --port=80 --name=apache2-service --type=NodePort (Creamos un servicio llamado apache2-service de tipo Nodeport para probar el pod apache2)

- Al servicio para probar el pod apache2 se accede a través de la URL: http://192.168.2.129:30703/

- Tras crear el pod y el servicio, creamos un ingress llamado ingress-apache, que apunte al servicio apache2-service, su anotación será de tipo rewrite-target (una de las más habituales), como reglas se tiene el nombre del host (curso.prueba.com) que corresponde a un nombre/dominio DNS que apunte a mi cluster de minikube (Para hacer esta prueba, como no tenemos servidor DNS para registrar el dominio curso.prueba.com, modificamos el fichero /etc/hosts y añadimos la IP de minikube apuntando a curso.prueba.com), como path tenemos

/apache de tipo exact y voy a ir al servicio apache2-service por el puerto 80. El fichero YAML del ingress es el siguiente:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
   name: ingress-apache
   annotations:
     nginx.ingress.kubernetes.io/rewrite-target: /
spec:
   rules:
   - host: curso.prueba.com
     http:
       paths:
       - path: /apache
         pathType: Exact
         backend:
           service:
             name: apache2-service
             port:
               number: 80
```

- kubectl apply -f ingress.yaml (Creamos el ingress haciendo uso del fichero YAML anterior)

- kubectl get ingress (Vemos el ingress)

- Ahora, si yo accedo a: http://curso.prueba.com/ me saldrá un 404 de nginx, ya que en path "/" no hay nada, pero si accedo a http://curso.prueba.com/apache me saldrá la página apache del pod apache2. Por lo tanto, se ha accedido para comprobar el pod apache2 mediante un ingress que usa el servicio apache2-service, pero con el ingress no hace falta usar el puerto 30703 ni nada de eso.

EJEMPLO CON DOS SERVICIOS

- kubectl run nginx2 --image=nginx (Creamos un pod de nginx llamado nginx2)

- kubectl expose pod nginx2 --port=80 --name=nginx2-service --type=NodePort (Creamos un servicio llamado nginx2-service de tipo Nodeport para probar el pod nginx2)

- Al servicio para probar el pod nginx2 se accede a través de la URL: http://192.168.2.129:32167/

- Creamos un ingress llamado ingress-apache-nginx, que apunte al servicio apache2-service y al servicio nginx2-service, su anotación será de tipo rewrite-target (una de las más habituales), como reglas se tiene el nombre del host (curso.prueba.com) que corresponde a un nombre/dominio DNS que apunte a mi cluster de minikube (Para hacer esta prueba, como no tenemos servidor DNS para registrar el dominio curso.prueba.com, modificamos el fichero /etc/hosts y añadimos la IP de minikube apuntando a curso.prueba.com), como path tenemos /apache de tipo exact que estará apuntando al servicio del pod apache2 y el path /nginx de tipo exact que estará apuntando al servicio del pod nginx2, donde el servicio apache2-service por el puerto 80 probará el pod apache2 y el servicio nginx2-service probará el pod nginx2, mediante el ingress sin necesidad de usar puertos externos como en los servicios de tipo NodePort o LoadBalancer. El fichero YAML del ingress es el siguiente:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
   name: ingress-apache-nginx
   annotations:
     nginx.ingress.kubernetes.io/rewrite-target: /
spec:
   rules:
```

```
  - host: curso.prueba.com
    http:
      paths:
      - path: /apache
        pathType: Exact
        backend:
          service:
            name: apache2-service
            port:
              number: 80
      - path: /nginx
        pathType: Exact
        backend:
          service:
            name: nginx2-service
            port:
              number: 80
```

- kubectl apply -f ingress2.yaml (Creamos el ingress haciendo uso del fichero YAML anterior)

- kubectl get ingress (Vemos el nuevo ingress)

- Ahora, si yo accedo a: http://curso.prueba.com/ me saldrá un 404 de nginx, ya que en path "/" no hay nada, pero si accedo a http://curso.prueba.com/apache ó a http://curso.prueba.com/nginx me saldrá la página apache del pod apache2 y la página de nginx del pod nginx2. Por lo tanto, se ha accedido para comprobar el pod apache2 y el pod nginx2 mediante un ingress que usa el servicio apache2-service y el servicio nginx2-service, pero con el ingress no hace falta usar el puerto 30703 o 32167 ni nada de eso.

- De esta manera, a través del ingress se han probado a la vez y usando el mismo host (curso.prueba.com) 2 pods, donde a cada pod se accede a través de sus paths correspondientes.

EJEMPLO CON VARIOS HOST VIRTUALES

- kubectl run blog --image=apasoft/blog (Creo un pod llamado blog basado en una imagen del dockerhub de apasoft)

- kubectl run web --image=apasoft/web (Creo un pod llamado web basado en una imagen del dockerhub de apasoft)

- kubectl expose pod blog --port=8080 --name=blog-service --type=NodePort (Creamos un servicio llamado blog-service de tipo Nodeport para probar el pod blog)

- kubectl expose pod web --port=80 --name=web-service --type=NodePort (Creamos un servicio llamado web-service de tipo Nodeport para probar el pod web)

- Al servicio para probar el pod blog se accede a través de la URL: http://192.168.2.129:31873/ y para probar el pod web se accede a la URL: http://192.168.2.129:31298/

- Creamos un ingress llamado ingress-multi-hosts, que tendrá 2 hosts (desarrollo.curso.com y produccion.curso.com, ambos registrados en el /etc/hosts apuntando a la IP de minikube), su anotación será de tipo rewrite-target (una de las más habituales). Con el host desarrollo.curso.com se podrán acceder a los paths /apache y /nginx de tipo Exact para probar los pods apache2 y nginx2, ya que estos paths están apuntando a los servicios apache2-service y nginx2-service, respectivamente, como en los ejemplos anteriores. Luego, con el host produccion.curso.com se podrán acceder a los paths /blog y /web de tipo

Prefix para probar los pods blog y web, ya que estos paths están apuntando a los servicios blog-service y web-service, respectivamente. El fichero YAML es:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
   name: ingress-multi-hosts
   annotations:
     nginx.ingress.kubernetes.io/rewrite-target: /
spec:
   rules:
   - host: desarrollo.curso.com
     http:
       paths:
       - path: /apache
         pathType: Exact
         backend:
           service:
             name: apache2-service
             port:
               number: 80
       - path: /nginx
         pathType: Exact
         backend:
           service:
             name: nginx2-service
             port:
               number: 80
   - host: produccion.curso.com
     http:
       paths:
       - path: /blog
         pathType: Prefix
```

```
      backend:
        service:
          name: blog-service
          port:
            number: 8080
    - path: /web
      pathType: Prefix
      backend:
        service:
          name: web-service
          port:
            number: 80
```

- kubectl apply -f ingress3.yaml (Creamos el ingress haciendo uso del fichero YAML anterior)

- kubectl get ingress (Vemos el nuevo ingress)

- Ahora, si yo accedo a: http://desarrollo.curso.com/ me saldrá un 404 de nginx, ya que en path "/" no hay nada, pero si accedo a http://desarrollo.curso.com/apache ó a http://desarrollo.curso.com/nginx me saldrá la página de apache del pod apache2 y la página de nginx del pod nginx2. Por lo tanto, se ha accedido para comprobar el pod apache2 y el pod nginx2 mediante un ingress que usa el servicio apache2-service y el servicio nginx2-service, pero con el ingress no hace falta usar el puerto 30703 o 32167 ni nada de eso. Además, si accedo a http://produccion.curso.com me saldrá un 404 de nginx, ya que en path "/" no hay nada, pero si accedo a http://produccion.curso.com/blog ó a http://produccion.curso.com/web me saldrá la página del blog del pod blog y la página de web del pod web. De esta manera accedo a varios paths que prueban diferentes pods, haciendo uso de varios hosts donde sus paths apuntan a diferentes servicios de distintos pods.

NGINX CONTROLLER

- Hay varios controllers como: HAProxy Ingress Controller, Kusk Gateway, Istio Ingress, Nginx Controller, etc. Más info sobre los controllers y el listado de todos ellos aquí: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

- Como vamos a utilizar e instalar el Nginx controllers, accedemos a su página oficial encontrada a través de GitHub para instalarlo en nuestro cluster kubeadm de 3 nodos formado por MV's: https://kubernetes.github.io/ingress-nginx/

INSTALAR NGINX EN UN CLÚSTER BARE METAL (KUBEADM)

- kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.6.4/deploy/static/provider/baremetal/deploy.yaml (Esto crea varios recursoskubectl)

- kubectl get pods -n ingress-nginx (Veremos 3 pods)

- kubectl get svc -n ingress-nginx (Veremos 2 servicios)

- A partir de aquí, tendremos instalado ingress en un host bare metal en nuestro clúster de kubernetes con kubeadm y por tanto podremos crear objetos ingress y realizar los ejemplos anteriores, pero ahora en el cluster con kubeadm en lugar de en minikube.

CREACIÓN DE UN CLUSTER DE KUBERNETES EN AWS CON AMAZON EKS

- Para hacer todo esto cuesta dinero por lo que si en algún momento hace falta volver a ver los vídeos de la sección 24 del curso.

- Antes de crear un cluster con EKS se crea un stack de red (VPC).

- Una vez creado el cluster EKS en AWS e instalado y configurado el cliente de AWS (necesario para trabajar con el cluster), podré hacer todas las operaciones y secciones que están este fichero TXT pero ahora con un cluster que está en la nube (cloud).

CREACIÓN DE UN CLUSTER DE KUBERNETES EN AZURE CON AMAZON AKS

- Para hacer todo esto cuesta dinero por lo que si en algún momento hace falta volver a ver los vídeos de la sección 25 del curso.

- Para conectarnos al cluster se usa una terminal integrada en Azure, y la interfaz de AKS es muy parecida a la de minikube dashboard.

CONCLUSIONES ENTORNOS CLOUD (AWS Y AZURE)

- En AWS para trabajar con el clúster se necesita instalar y configurar un cliente AWS y en Azure se utiliza una terminal integrada para trabajar con el clúster.

- Tanto en AWS y en Azure, se trabajar con la herramienta kubectl (se necesita instalarla en algunos casos).

CONCLUSIONES

Tras finalizar el curso y este documento, he aprendido bastante acerca de Kubernetes, de todos sus objetos, distintas herramientas y configuraciones, como instalar un clúster de Kubernetes de varias maneras, como trabajar en local o en remoto con este orquestador de contenedores, etc. Por todo ello, estos conocimientos me han servido de base para seguir aprendiendo y avanzando en esta tecnología muy utilizada a día de hoy por varias empresas.