Fuente: Bing Chat AI

Bloque 5. Kubernetes

MÁSTER EN INGENIERÍA WEB 2023 – 2024

Administración de Sistemas Operativos v1.2 *("L-62 Princesa de Asturias")*

© José Manuel Redondo López. Universidad de Oviedo

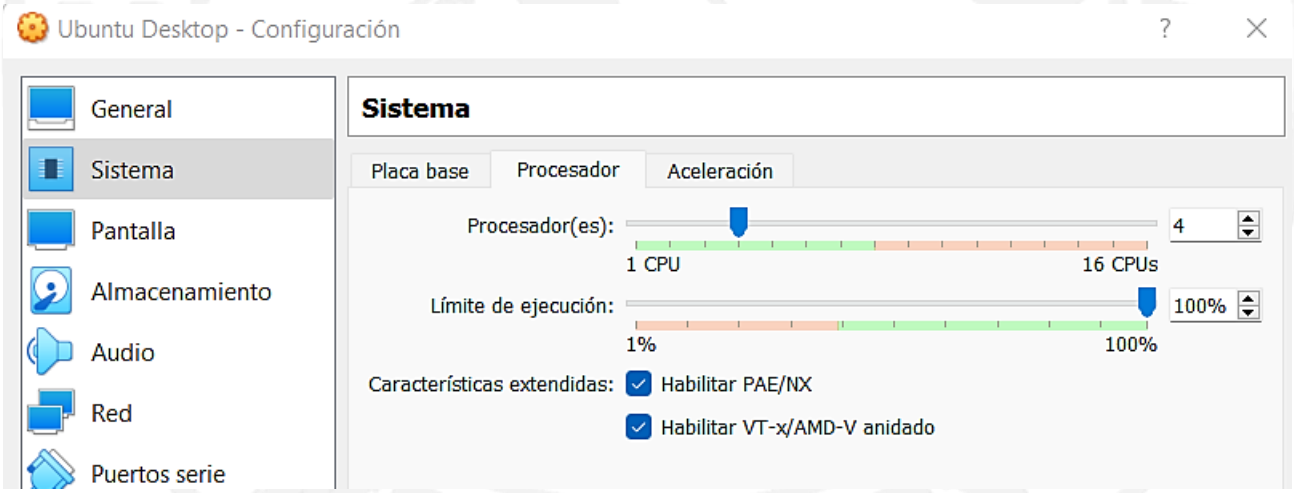
Contenido

Nivel 1 (Cabo): Instalación y uso básico

¿Qué necesito para hacer este bloque de actividades?

Para realizar este bloque de actividades necesitas una máquina física o virtual *Linux*, **como por ejemplo las que se puedan haber creado con el módulo de *Vagrant***. La máquina requiere lo siguiente:

- **Un *software* de virtualización instalado**: en principio sirve cualquiera que tenga la opción que se describirá en el último punto si la vamos a usar. Se recomienda *VirtualBox*, como en el caso de *Vagrant*.
- **Núcleos de CPU**: Con 2 se puede trabajar perfectamente, un extra siempre es bienvenido
- **RAM**: Mínimo 2, mejor 4 o más
- **SO**: Un *Linux*; se ha probado sobre *Ubuntu 18.04* con éxito.
- **(Opcional) Que la máquina tenga activo el soporte de virtualización *hardware*** (VT-X/AMD-V) anidada tal y como establezca tu programa de virtualización. En *VirtualBox* se trata de esta opción. **OJO**: No todas las CPUs soportan esta característica.



- **Software:** *Docker* y *Docker Compose* instalado (para usar el driver de `docker` de `minikube`) o bien el paquete `virtualbox` (sólo si la máquina soporta virtualización anidada, con lo cual podrás usar el driver `virtualbox` en `minikube`).

NOTA: Para realizar estas actividades adecuadamente se sugiere completar el boletín de actividades de contenedores de aplicaciones hasta el nivel 2, y además tener un control básico de la creación de *Dockerfiles*

N1KUB_INSTALAR: Instalación de un clúster *Kubernetes* local con *Minikube* *(0,2 puntos)*

Descripción de la actividad

Consiste en **crear una instalación operativa de *Minikube*** que funcione dentro de una MV, como implementación de K8s local y eficiente que no requiere el uso de recursos en nube y desembolsos adicionales.

Resultados esperados

Eres capaz de hacer funcionar una instalación de *Minikube* sin un proveedor de virtualización asociado para mejorar su rendimiento, o bien con un proveedor de virtualización *VirtualBox* y soporte de virtualización anidada activo.

Información necesaria para su realización

Para completar esta actividad es necesario hacer un despliegue local con *Minikube* **sin usar un driver de virtualización** (es decir, usando por debajo los componentes de una instalación previa de *Docker*) y comprobar que el comando funciona correctamente listando los nodos disponibles. **También es válido hacerla con un driver de virtualización `virtualbox` si la máquina virtual soporta virtualización anidada**. Otros proveedores de virtualización, como KVM/QEMU, también son válidos.

Antes de continuar, conviene decir que *Kubernetes* (y, por tanto, `minikube`) **decidió no soportar directamente el motor de contenedores *Docker CE* que usamos en esta asignatura a partir de la versión 1.24**. En las pruebas realizadas hemos obtenido algún problema de instalación, por lo que para evitarlos y concentrarnos en el uso puro de `minikube`, que es lo que nos importa, hay 3 opciones:

1. (**Recomendada**) **Descárgate la versión 1.23.X** (https://github.com/kubernetes/minikube/releases) y no la `latest`: `wget https://storage.googleapis.com/minikube/releases/v1.23.1/minikube-linux-amd64`. No es la última versión y, por tanto, no es lo recomendado para ir a producción, pero para hacer pruebas es más que suficiente, puesto que los comandos que vamos a usar son los mismos.
2. Descárgate la última versión, pero instala otro motor de contenedores como *CRI-O* (*https://computingforgeeks.com/install-cri-o-container-runtime-on-ubuntu-linux/*), **no instalando *Docker CE*** en la máquina virtual de pruebas, y cambia el *container runtime* en el arranque de *Minikube*: https://minikube.sigs.k8s.io/docs/commands/start/
3. Instala *Docker CE* y la última versión de *Minikube* en la máquina virtual de pruebas y además un **módulo de *Mirantis* de compatibilidad entre *Minikube* y este**: https://github.com/Mirantis/cri-dockerd#build-and-install. El proceso completo se explica aquí: https://www.fosstechnix.com/how-to-install-minikube-on-ubuntu-22-04-lts/

El problema es que **las opciones 2 y 3 han sufrido numerosos fallos en *Ubuntu 18.04*** en las pruebas realizadas, por lo que o bien se instala en una versión de *Ubuntu* más moderna de la que estamos usando de base en el curso, como la 22.04 (https://linuxhint.com/install-minikube-ubuntu/#b5) **o se recomienda encarecidamente proceder con la opción 1** para no perder tiempo con esto, que no es el objetivo del curso. Como decíamos, aunque no sea la versión más moderna, para practicar con *Kubernetes* es perfectamente

válida. Para saber si tu instalación ha arrancado correctamente, se da esta imagen de la salida esperada como guía:

```
ssiuser@vagrant:~$ minikube version
minikube version: v1.23.1
commit: 84d52cd81015effbdd40c632d9de13db91d48d43
ssiuser@vagrant:~$ minikube start --vm-driver=docker start
😄  minikube v1.23.1 on Ubuntu 18.04 (vbox/amd64)
🔥  minikube 1.26.0 is available! Download it: https://github.com/kubernetes/min
ikube/releases/tag/v1.26.0
💡  To disable this notice, run: 'minikube config set WantUpdateNotification fal
se'

🤷  Using the docker driver based on user configuration
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Downloading Kubernetes v1.22.1 preload ...
    > preloaded-images-k8s-v13-v1...: 511.84 MiB / 511.84 MiB  100.00% 14.38 Mi
    > gcr.io/k8s-minikube/kicbase: 355.39 MiB / 355.40 MiB  100.00% 6.67 MiB p/
🔥  Creating docker container (CPUs=2, Memory=2200MB) ...
🐳  Preparing Kubernetes v1.22.1 on Docker 20.10.8 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass

❗  /usr/local/bin/kubectl is version 1.24.3, which may have incompatibilites wi
th Kubernetes 1.22.1.
    ▪ Want kubectl v1.22.1? Try 'minikube kubectl -- get pods -A'
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" name
space by default
```

N1KUB_WEBSIMPLE: Despliegue de un servidor web simple en *Kubernetes* *(0,1 puntos)*

Descripción de la actividad

Consiste en aprender a **hacer un *deployment* sencillo de un servidor web** sobre un clúster K8s.

Resultados esperados

Eres capaz de hacer un *deployment* de *Kubernetes* con una web simple estática

Información necesaria para su realización

Esta actividad consiste en crear un *deployment* **que contenga un contenedor de una imagen oficial de *Docker*** que despliegue un servidor web *Apache 2* o *Nginx*. Para completar la tarea, al contenedor de dicho servidor web se le debe introducir la web de ejemplo proporcionada para este curso de la forma que más fácil os resulte y, finalmente, acceder a la misma desde el *host* para comprobar que todo funciona correctamente. Ejemplo: `sudo kubectl create deployment server1 --image=ubuntu/apache2`

N1KUB_INFO: Extracción de información de *deployments* y *pods* *(0,1 puntos)*

Descripción de la actividad

Consiste en aprender a **leer la información acerca de los *deployments* y *pods*** que da K8s.

Resultados esperados

Eres capaz de obtener toda la información disponible de un *deployment* existente

Información necesaria para su realización

Siguiendo las indicaciones de la teoría, eres capaz de sacar la información que proporciona *Kubernetes* acerca del *deployment* y *pod* creado a partir de la actividad anterior.

N1KUB_PODMANAGE: Interacción con un *pod* *(0,1 puntos)*

Descripción de la actividad

Consiste en que compruebes que **entiendes y sabes interaccionar con un *pod* directamente** si te hiciera falta.

Resultados esperados

Eres capaz de interactuar con el contenedor de un *pod* directamente y ver información acerca del mismo. Puedes contestar a esta pregunta: *¿Entiendes por qué hacer esto en producción es inútil, ya que cualquier cosa que hagas se puede destruir en cualquier momento?*

Información necesaria para su realización

En esta tarea se pide, siguiendo las indicaciones de teoría y tras haber creado un servidor web en *Kubernetes* en la tarea anterior, que puedas interaccionar con el contenedor del *pod* creado de la siguiente forma:

- Puedes **entrar en sesión interactiva** dentro de él
- Puedes **instalar paquetes** dentro del contenedor como en un *Linux* normal
- Puedes **ver sus *logs***
- Puedes **ver sus variables de entorno**

Nivel 2 (Sargento): Usos comunes de Kubernetes

N2KUB_CREARSERV: Creación de servicios *(0,1 puntos)*

Descripción de la actividad

Consiste en **aprender a crear correctamente un servicio de K8s** a partir de un *deployment* que esté en funcionamiento.

Resultados esperados

Eres capaz de crear un servicio a partir de un *deployment* ya creado y permitir acceder al contenido de este a clientes externos

Información necesaria para su realización

En esta actividad se trata de crear un servicio a partir del *deployment* anterior, usando, de acuerdo con lo visto en teoría, un modo en el que dicho servicio pueda ser exportado para que lo usen clientes externos al clúster *Kubernetes*. Hecho esto, debe comprobarse que el servidor web anterior es accesible usando dicho servicio. Ten en cuenta en que rango se IPs se le da una IP al servicio y mira cómo coincide con el interfaz *bridge* que se crea en la instalación de `minikube`, que debería tener este aspecto:

```
ssiuser@vagrant:~$ ifconfig
br-457d155ad38f: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.49.1  netmask 255.255.255.0  broadcast 192.168.49.255
        inet6 fe80::42:6ff:fe27:cb42  prefixlen 64  scopeid 0x20<link>
        ether 02:42:06:27:cb:42  txqueuelen 0  (Ethernet)
        RX packets 5401  bytes 529442 (529.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8083  bytes 65786958 (65.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Observa además como tu servicio realmente está accesible a través del puerto que especifiques al crearlo, pero usando la IP del clúster `minikube` que has creado

```
ssiuser@vagrant:~$ sudo minikube ip
192.168.49.2
```

N2KUB_LABELS: Interacción con etiquetas *(0,1 puntos)*

Descripción de la actividad

Consiste en **familiarizarte con el uso de etiquetas** para designar a distintos elementos que forman parte de la infraestructura de K8s

Resultados esperados

Eres capaz de gestionar las etiquetas de un servicio y usarlas para encontrar servicios en cualquier momento

Información necesaria para su realización

Esta actividad consiste en **mostrar las etiquetas de los servicios** activos en un despliegue de *Kubernetes* en un momento dado, **añadir** a dicho servicio etiquetas nuevas y **probar a buscar** dicho servicio por sus etiquetas.

Para ello, en esta actividad se pide **asignar una etiqueta nueva** al servicio anterior y comprobar que se puede localizar usándola posteriormente.

N2KUB_CODEDEPLOY: Creación de *deployments* por código *(0,2 puntos)*

Descripción de la actividad

Consiste en **aprender a desplegar un *deployment* por código** en un clúster K8s funcional.

Resultados esperados

Eres capaz de crear *deployments* con código en un fichero YAML adecuado

Información necesaria para su realización

A partir de los contenidos vistos en teoría, en este ejercicio se pide **desplegar un contenedor del *Docker Hub*** que tenga un servidor web (*Apache 2* o *Nginx*) vía código, creando un *deployment* distinto al que creamos en las actividades anteriores. Dicho contenedor debe tener una sola réplica y exponer su puerto 80 para poder acceder al servicio.

Finalmente, **se pide también acceder a él** para comprobar que funciona, y probar a actualizar el código de alguna manera que te resulte más sencilla para ver que los cambios que has hecho hacen efecto cuando vuelves a desplegarlo.

N2KUB_CODESERVICE: Creación de servicios por código *(0,1 puntos)*

Descripción de la actividad

Consiste en **aprender a desplegar servicios por código** en un clúster K8s operativo.

Resultados esperados

Eres capaz de crear un servicio vía código vinculado con un fichero YAML a un *deployment* existente

Información necesaria para su realización

Usando los conocimientos de teoría, se pide **crear un servicio de tipo** `NodePort` que esté vinculado al *deployment* anterior vía código, y que el servidor web de sus contenedores se exponga al exterior a través del puerto 25000, comprobando que es visible.

N2KUB_WEBDEPLOY: Despliegue de una aplicación web simple sobre una imagen personalizada *(0,3 puntos)*

Descripción de la actividad

Consiste en **aprender a desplegar una web dinámica simple** en un clúster de K8s operativo.

Resultados esperados

Eres capaz de desplegar una aplicación simple que use una tecnología de procesamiento de peticiones en el servidor como *Node.js*

Información necesaria para su realización

En esta actividad se va a probar el despliegue en *Kubernetes* de una aplicación web muy simple **hecha con *Node.js***. Para ello **la instalaremos en un contenedor personalizado** que crearemos a partir de uno disponible en el *Docker Hub* que soporta esta clase de aplicaciones. Para ello:

- Copiamos esta aplicación simple de *Node.js* en un fichero llamado `server.js`

```
var http = require('http');

var handleRequest = function(request, response) {
  console.log('He recibido una petición de: ' + request.url);
  response.writeHead(100);
  response.end('¡Funciona!');
};
var www = http.createServer(handleRequest);
www.listen(8080);
```

- Ahora construimos la imagen de los contenedores a desplegar así, y le damos un nombre a la imagen a nuestro gusto. Nótese que estamos usando una versión de la imagen `node` del *Docker Hub*:

```
FROM node:6.14.2
EXPOSE 8080
COPY server.js .
CMD [ "node", "server.js" ]
```

- Hecho esto, debemos seguir los pasos anteriores para:
  - **Crear un *deployment*** con el nombre que queramos, que despliegue la imagen recién construida.
  - **Asociarle un servicio de tipo** `NodePort` y un puerto a través del cual queramos acceder a la aplicación.
  - **Acceder a la aplicación desplegada** para comprobar que funciona.

Nivel 3 (Teniente de navío): Resolución de situaciones más avanzadas con Kubernetes

N3KUB_DASHBOARD: Desplegar el *Dashboard* *(0,2 puntos)*

Descripción de la actividad

Consiste en **desplegar un *Dashboard* gráfico** que nos da el control y un acceso sencillo a cómo se está comportando nuestro clúster K8s en lo relativo a pods, servicios, consumo de recursos...

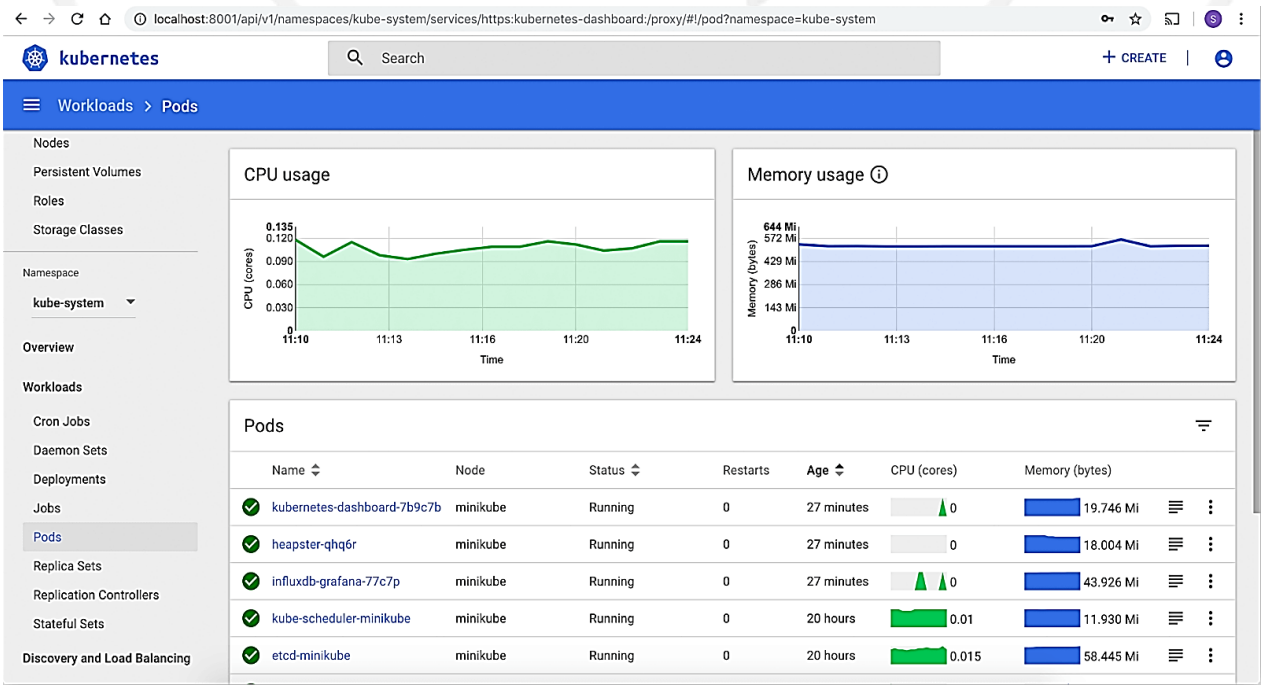
Resultados esperados

Eres capaz de arrancar el *dashboard* de *Minikube* y manejarte con él para ver la información de la instalación de *Minikube* en la máquina actual. Puedes contestar a esta pregunta: *¿Entiendes por qué este GUI es muy importante para facilitar la gestión de tu clúster K8s?*

Información necesaria para su realización

Ahora que tenemos un par de *deployments* y servicios en funcionamiento, es buen momento para **desplegar el *dashboard* de *Minikube*** y ver de forma gráfica los elementos que están desplegados y los recursos que consumen.

En esta tarea simplemente hay que arrancar dicho *dashboard* (`minikube dashboard`) y navegar por el mismo para ver el uso de recursos a lo largo del tiempo, elementos creados, etc.



N3KUB_ESCMANUAL: Escalado manual de clústeres *(0,1 puntos)*

Descripción de la actividad

Consiste en que **aprendas a escalar manualmente un servicio** de K8s especificando el nº de réplicas que quieres que tenga para atender peticiones

Resultados esperados

Eres capaz de escalar manualmente servicios de *Kubernetes* para poner el nº de réplicas que estimes oportunas para dar un servicio en un momento dado. Puedes contestar a esta pregunta: *¿Cuál crees que es el principal error que puedes cometer en un escalado manual?*

Información necesaria para su realización

En esta actividad se trata de **escalar manualmente los dos servicios que hemos arrancado hasta el momento** (el hecho vía comandos y el hecho vía código) para que tengan **2 réplicas cada uno**, actualizando su configuración y consultando la información de estos para comprobar que realmente dichas réplicas están en marcha.

N3KUB_STRESS1: Respuesta de un escalado manual a cargas de trabajo elevadas *(0,2 puntos)*

Descripción de la actividad

Consiste en que puedas **someter a tu clúster K8s a pruebas de *stress* manuales** para ver cómo responde.

Resultados esperados

Eres capaz de hacer mediciones de rendimiento y de carga de peticiones contra un clúster *Kubernetes* y comprobar si aumentar el nº de réplicas mejora el rendimiento o no en el entorno de pruebas. Puedes contestar a estas preguntas:

- *¿Encuentras coherente el resultado de tu clúster ante una carga de peticiones cuando lo escalas manualmente?*
- *Si no es así, ¿A qué crees que se debe el comportamiento que observas?*

Información necesaria para su realización

Instalar y usar la herramienta ***Apache Benchmark*** (ab) para lanzar un nº grande de peticiones contra el clúster *Kubernetes* con dos réplicas (cualquiera de los dos escalados en la actividad anterior) y guardar la información de rendimiento devuelta. Puedes usar este tutorial como guía: https://juantrucupei.wordpress.com/2015/11/10/pruebas-con-ab-apache-benchmark/.

Hecho esto, debes **aumentar el nº de replicas a 4** y repetir el proceso. **Auméntalas por último a 6** y vuelve a hacer la medición. *¿Qué comportamiento observas? ¿Puedes explicarlo?*

N3KUB_SELFHEAL: *Self-healing* de clústeres *(0,1 puntos)*

Descripción de la actividad

Consiste en que compruebes por ti mismo la **capacidad de un clúster de K8s de resistir a la "muerte" de algunos de sus *pods*** por cualquier motivo.

Resultados esperados

Eres capaz de destruir réplicas de un *pod* y comprobar que el servicio asociado al mismo vuelve al cabo de un tiempo a relanzar más réplicas hasta alcanzar el estado que tenía inicialmente. Puedes responder esta pregunta *¿Qué beneficios crees que tiene esta capacidad?*

Información necesaria para su realización

Para hacer este ejercicio fácilmente, puedes usar cualquiera de los servicios anteriores y **destruir la mitad de sus réplicas activas**, comprobando luego que efectivamente el sistema es capaz de volver a su estado inicial por sí solo.

N3KUB_VOLUMEN: Despliegue de un volumen persistente *(0,3 puntos)*

Descripción de la actividad

Consiste en **aprender a asociar un volumen a *deployments*** de K8s.

Resultados esperados

Eres capaz de definir un volumen persistente, asociarlo a un *deployment* y comprobar su funcionamiento. Puedes responder a esta pregunta: *¿qué parte de una aplicación web situarías tú en un volumen como este?*

Información necesaria para su realización

En esta actividad se trata de que apliques los conocimientos de teoría y **despliegues un volumen persistente asociado a un *deployment***, nuevo o que ya tengas de un ejercicio anterior, y que puedas acceder a él desde el mismo, probando que los datos en el volumen no son efímeros y sobreviven a reinicios de los *pods* del *deployment*. Para ello se debe seguir este procedimiento:

- **Definir un volumen** llamado `shareddata` que se monte en `/mnt/shared` en el *deployment* que lo use, con capacidad de 100Mb y modo de acceso `ReadWriteOnce`.
- **Definir un *claim*** adecuado para este volumen
- **Asociar el volumen a un *deployment*** que crees o tengas de otro ejercicio y desplegar el *deployment* con él.
- **Comprobar que se han creado los *pods* del *deployment*** y describir (`kubectl describe pod/<id de un pod>`) uno de ellos para ver si sale la información del volumen añadido en esa información.

- **Acceder a la consola de uno de los *pods*** con `kubectl exec -it <identificador de un pod> bash`
- **Escribir un fichero** con algún contenido en la ruta del volumen añadido y salir de la consola
- **Matar el *pod*** al que acabamos de acceder y dejar que se regenere solo.
- **Acceder al *pod* recién generado** y comprobar que el fichero escrito sigue en el volumen añadido al *deployment*.

Nivel 4 (Capitán de navío): Usos de Kubernetes para aplicaciones exigentes

N4KUB_AUTOESC: Auto-escalando un clúster *(0,3 puntos)*

Descripción de la actividad

Consiste en aprender a **activar y usar el auto-escalado de servicios** en un clúster K8s

Resultados esperados

Eres capaz de habilitar satisfactoriamente y probar el auto-escalado de servicios *Kubernetes*, asegurándose de que se comporta como es esperable.

Información necesaria para su realización

Sobre cualquiera de los servicios que venimos manejando hasta ahora, hacer los cambios necesarios de acuerdo con lo visto en teoría **para que uno de ellos auto-escale su nº de *pods* desde un mínimo de 2 a un máximo de 6**.

N4KUB_STRESS2: Comportamiento del auto-escalado frente a un nº creciente de peticiones *(0,2 puntos)*

Descripción de la actividad

Consiste en aprender a **someter a un clúster K8s que tenga la capacidad de auto-escalar a cargas de trabajo intensivas**, observando y comprobando que se comporta como es esperable

Resultados esperados

Eres capaz de comprobar el auto-escalado de un clúster Kubernetes "en vivo", verificando que el clúster se comporta de la forma debida tanto en el incremento de peticiones como cuando no hay peticiones en un tiempo determinado.

Información necesaria para su realización

Usar de nuevo la herramienta `ab` para probar el rendimiento de un clúster auto-escalado, detectando tanto cuando **se aumentan las réplicas** cuando el clúster se ve sometido a gran nº de peticiones como cuando **estas disminuyen** al bajar la carga de trabajo.

N4KUB_CONFIGSECRET: Uso de *ConfigMaps* y *Secrets* para la confidencialidad de nuestros datos *(0,3 puntos)*

Descripción de la actividad

Consiste en **aprender a usar los elementos de un clúster K8s *ConfigMaps* y *Secrets*** para sacarle partido a los mismo a la hora de desplegar aplicaciones en un clúster K8s.

Resultados esperados

Eres capaz de hacer que un *deployment* pueda incorporar secretos y ficheros de configuración `clave: valor` a sus *pods*. Puedes contestar a estas preguntas:

- *¿Qué valores de una aplicación que diseñes pondrías en un ConfigMap?*
- *¿Qué valores de una aplicación que diseñes pondrías en un Secret?*
- *¿Entiendes que el valor real de un secret es que no se vea en el fichero de código que define tu servicio? ¿Qué escenario se te ocurre en el que dejarse un secreto ahí puede ser especialmente peligroso?*

Información necesaria para su realización

Esta actividad consiste en que practiques con lo explicado en teoría para *ConfigMaps* y *Secrets* y seas capaz de usarlos en un *deployment* existente de ejercicios anteriores o que se cree nuevo. El procedimiento a seguir es:

- **Crea un fichero de *secrets*** que guarde como dato una dirección URL que quieres mantener secreta (la que tú quieras)
- **Haz que un *deployment*** sea capaz de usar este secreto asociándolo con él como se vio en teoría, asociando su valor a una variable de entorno con el nombre que tú quieras.
- **Despliega ese *deployment* y accede** a la consola de uno de sus *pods* con `kubectl exec -it <identificador de un pod> bash`
- **Comprueba que existe una variable de entorno con el nombre que has creado** y el valor del secreto listo para usar con `env`.
- **Hecho esto, define un fichero *ConfigMap*** con unos pares `<clave>:<valor>` a tu gusto y añádelo al clúster *Kubernetes* con el nombre `systemConfig`.
- **Actualiza el fichero del *deployment* anterior** para que además del secreto incorpore el fichero del *ConfigMap* en un volumen que se cree en la ruta `/etc/config`.
- **Despliega de nuevo el *deployment*** y accede ahora a la consola de uno de sus *pods* para comprobar que el fichero de configuración está en la ruta esperada.

Nivel 5 (Almirante): Operaciones avanzadas con Kubernetes

N5KUB_WEBDBDEPLOY: Despliegue de una aplicación con base de datos en *Kubernetes* *(0,6 puntos)*

Descripción de la actividad

Consiste en **desplegar una aplicación web en dos capas**, incluyendo lógica de negocio y base de datos, sobre un clúster K8s.

Resultados esperados

Eres capaz de instalar una aplicación web en dos capas **respondiendo a las preguntas** que se hacen durante todo el proceso. Puedes contestar a esta pregunta: *¿Dónde guardarías la base de datos en circunstancias normales?*

Información necesaria para su realización

Esta actividad desplegará una aplicación web ya existente (el servicio web `url-shortener` de Xabier Coulon) en tu clúster de `minikube`, para practicar todo el proceso de despliegue de una aplicación que tenga varias capas. Por suerte, la aplicación mencionada ya se encuentra subida a un contenedor en *Docker Hub*, por lo que obtenerla será muy sencillo. Para ello es necesario seguir este procedimiento:

- **Desplegar primero la base de datos *PostgreSQL*** que usa la aplicación usando este fichero de *deployment*. *¿Qué defectos le ves de cara a pasarlo a producción? ¿Cómo lo arreglarías (no hace falta hacerlo, sólo decir cómo)?*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:9.6.5
        ports:
        - containerPort: 5432
```

```
        env:
        - name: POSTGRES_DB
          value: url_shortener_db
        - name: POSTGRES_USER
          value: user
        - name: POSTGRES_PASSWORD
          value: mysecretpassword
```

- **Aplicar el contenido del fichero** con `kubectl create -f <nombre del fichero>` y espera con `kubectl get all` a que esté desplegado completamente.
- **Mira el identificador del *pod* que sale en la orden anterior** y comprueba que *PostgreSQL* se ha instalado correctamente con `kubectl get pod/<id del pod> -o go-template={{.metadata.labels}}`.
    - Debería salir algo como: `map[app:postgres pod-template-hash:1831265952]`
- **Comprueba que *PostgreSQL* funciona adecuadamente** entrando en sesión en el *pod* con `kubectl exec -it <id del pod> bash` y, una vez dentro de la consola de este, hacer `psql -U user -d url_shortener_db` para entrar en la consola del motor de base de datos.
    - Dentro de esa consola teclear `\d`: Debería devolver "*No relations found*".
    - Para salir de la consola de *PostgreSQL*, teclear `\q`.
- Ahora que hemos comprobado que el *deployment* está operativo, podemos **exponerlo como servicio creando** y aplicando un fichero como este:

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  ports:
  - port: 5432
  selector:
    app: postgres
```

- **Listamos ahora los servicios de nuestro clúster** hasta que se muestre como operativo antes de pasar a desplegar la web.
- Como decíamos anteriormente, la aplicación web de pruebas **está en el *Docker Hub* en el contenedor** `xcoulon/go-url-shortener:0.1.0`. Podemos desplegarla creando y aplicando un fichero como este, que como ves tiene los mismos problemas que el anterior. No obstante, suponiendo que algunas variables de entorno no fueran de carácter secreto, *¿En qué entidad de Kubernetes las pondrías?*:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
```

```
        app: webapp
    spec:
      containers:
      - image: xcoulon/go-url-shortener:0.1.0
        name: go-url-shortener
        env:
        - name: POSTGRES_HOST
          value: postgres
        - name: POSTGRES_PORT
          value: "5432"
        - name: POSTGRES_DATABASE
          value: url_shortener_db
        - name: POSTGRES_USER
          value: user
        - name: POSTGRES_PASSWORD
          value: mysecretpassword
        ports:
        - containerPort: 8080
```

- **Comprobamos ahora que la aplicación funciona** haciendo `kubectl get all -l app=webapp` y esperando hasta que se despliegue completamente.
- Otra cosa que podemos hacer es **inspeccionar sus *logs*** para ver que se muestra una conexión con la base de datos con éxito con `kubectl logs pod/<id del pod de la aplicación web>`
- De nuevo ahora **exponemos la aplicación web como un servicio** creando y aplicando un fichero como este:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
    - nodePort: 31317
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: webapp
```

- **Comprobamos que el servicio** está operativo con: `kubectl get services -o wide`
- Y llegados a este punto ya podemos usar la aplicación web desplegada usando sus tres funcionalidades, finalizándose la actividad si pasa todas las pruebas:
    - `curl -v http://$(minikube ip):31317/ping` debería devolver datos de la conexión HTTP y `pong!` al final
    - `curl -X POST http://$(minikube ip):31317/ -d "full_url=<pon una URL aquí>"` nos debería devolver una cadena de texto que es el token acortado equivalente a la URL pasada.
    - `curl -X GET http://$(minikube ip):31317/<el token que obtuvimos anteriormente> -v` nos devolverá una serie de datos de la conexión HTTP y la URL real que usamos en la comprobación anterior dentro de la cabecera HTTP `Location`.

N5KUB_NETPOL: Uso de *Network Policies* para prevenir accesos no autorizados *(0,6 puntos)*

Descripción de la actividad

Consiste en **aprender a usar *Network Policies*** en un clúster K8s para proteger nuestros *deployments* de accesos no autorizados.

Resultados esperados

Eres capaz de definir una *Network Policy* sencilla basada en que los *pods* tengan o no definida una determinada etiqueta. Puedes responder a estas preguntas:

- *¿En qué momento del ejercicio crees que se activó la posibilidad de usar `nginx` para referirnos al servicio creado?*
- *¿Qué partes de una aplicación web separarías así?*

Información necesaria para su realización

El uso de *Network Policies* tiene mucho más sentido **en clústeres grandes** compuestos de *deployments* que representen **varias capas de una aplicación**, o distintas aplicaciones, cuyas comunicaciones deban estar muy restringidas para asegurarse de que sólo capas o aplicaciones autorizados puedan acceder a su API o interfaz de servicios.

En nuestras pruebas no vamos a crear una infraestructura grande para ello, pero si podemos probar cómo funcionan usando muy pocas entidades en el clúster. Este es un ejercicio que requiere una preparación previa, ya que **no funcionará sobre el `minikube` que venimos usando hasta ahora**.

Parte 1: Preparando un minikube que pueda usar NetworkPolicies

Nuestra instalación por defecto de `minikube` puede definir *Network Policies* y aplicarlas sin dar error, pero el problema es que **usa un *driver* de red que no las soporta**. Esto quiere decir que, aunque aparentemente todo se ejecuta correctamente, luego las restricciones de red que hayamos establecido en nuestra política de red **no funcionarán**: todos los *pods* podrán acceder a otros a los que tengan acceso vía IP.

Para evitar esto, tenemos que reinstalar `minikube` con soporte para *Network Policies*, y para eso vamos a usar el plugin de red ***Cilium***. Existen otros *plugins* de red que permiten hacer esto, como *Calico* (https://projectcalico.docs.tigera.io/getting-started/kubernetes/minikube), pero *Cilium* es muy fácil de usar. Para instalarlo debemos partir de **una máquina que no haya hecho todavía `minikube start`**, ya que **tendremos que añadir al `start` que vimos en la teoría el parámetro `--network-plugin=cni`**. Hecho esto, debemos descargar el fichero de instalación de *Cilium*, descomprimirlo y ejecutar `cilium install`:

```
curl -LO https://github.com/cilium/cilium-cli/releases/latest/download/cilium-linux-amd64.tar.gz
sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin
rm cilium-linux-amd64.tar.gz
cilium install
```

Una vez terminada la instalación (*Cilium* detectará `minikube` y añadirá a nuestra instalación todo lo necesario para funcionar), ejecutar `cilium status` hasta que en la información aparezca que todos los *pods* están "*managed by Cilium*".

```
ssiuser@vagrant:~$ cilium status
    /¯¯\
 /¯¯\__/¯¯\    Cilium:         OK
 \__/¯¯\__/    Operator:       OK
 /¯¯\__/¯¯\    Hubble:         disabled
 \__/¯¯\__/    ClusterMesh:    disabled
    \__/

DaemonSet         cilium             Desired: 1, Ready: 1/1, Available: 1/1
Deployment        cilium-operator    Desired: 1, Ready: 1/1, Available: 1/1
Containers:       cilium-operator    Running: 1
                  cilium             Running: 1
Cluster Pods:     2/2 managed by Cilium
Image versions    cilium             quay.io/cilium/cilium:v1.12.1@sha256:ea2db1ee21b88
127b5c18a96ad155c25485d0815a667ef77c2b7c7f31cab601b: 1
                  cilium-operator    quay.io/cilium/operator-generic:v1.12.1@sha256:93d
5aaeda37d59e6c4325ff05030d7b48fabde6576478e3fdbfb9bb4a68ec4a1: 1
```

Con esto ya tendremos `minikube` listo para hacer las pruebas

Parte 2: Probando Network Policies

Para probar las políticas de red sin necesidad de hacer un gran despliegue en nuestro clúster podemos hacer lo siguiente:

- **Crear o reutilizar un fichero de *deployment*** que ya tengamos hecho de ejercicios anteriores y desplegarlo en `minikube`. En nuestras pruebas supondremos que los *pods* del *deployment* creado **tienen la etiqueta `app: nginx` definida** para poder localizarlo fácilmente, pero puedes usar la que quieras.
- **Exponer el *deployment* como servicio**, si no lo está ya. Se recomienda hacer un *deployment* que use un servidor web y exponer el puerto 80 para hacer las pruebas más sencillas. Por ejemplo esto expondría un servicio llamado "`nginx`": `kubectl expose deployment nginx --port=80`
- **Comprobar que el servicio se ha terminado de exponer** con `kubectl get svc,pod`
- **Probar desde otro *pod* que podemos acceder al servicio expuesto**.
  - Para crear un *pod* rápidamente con un contenedor interactivo que ocupe muy poco y que se destruya automáticamente cuando salgamos de sesión podemos usar: `kubectl run busybox --rm -ti --image=busybox:1.28 -- /bin/sh`
  - Una vez dentro del mismo, hacer `wget nginx` para comprobar que podemos acceder al *deployment* anterior (asumiendo que le has llamado `nginx`, cambia el nombre en función de lo que hayas hecho). Sal de la consola con `exit` para que el *pod* se destruya.
- **Crea una *Network Policy* que limite el tráfico entrante al *pod* definido por la etiqueta `app: nginx`** (o cualquiera que hayas usado en tus pruebas) a todo *pod* que no tenga definida la etiqueta `trustable: "true"`. Al fichero lo llamaremos `network.yml` en estas pruebas y debería ser algo similar a esto.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
```

```
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          trustable: "true"
```

- **Aplica la *Network Policy*** con `kubectl apply -f network.yml`
- **Vuelve a crear un *pod*** con `busybox` auto-destruible como antes, y comprueba que ya no tienes acceso al servicio, ya que este *pod* no tiene esta etiqueta definida.
- **Sal del *pod* anterior y vuelve a crearlo con la etiqueta definida de esta forma**: `kubectl run busybox --rm -ti --labels="trustable=true" --image=busybox:1.28 -- /bin/sh`.
- Comprueba finalmente que ahora **sí tienes acceso al servicio.**

N5KUB_HELM: Prueba de *Helm* instalando un servicio *Memcached*

(1 punto)

Descripción de la actividad

Consiste en **aprender a desplegar *Helm Charts*** sobre un clúster K8s, aunque no tengamos un conocimiento profundo de esta tecnología.

Resultados esperados

Eres capaz de desplegar un *Helm Chart* público disponible públicamente en un clúster `minikube` y crear una aplicación básica que sea capaz de contactar con él y usarlo. Puedes contestar a estas preguntas:

- *¿Has tenido que preocuparte en algún momento de alguna particularidad de la estructura y puesta en marcha de los distintos componentes de Memcached?*
- *¿Entiendes por tanto el trabajo que te ha ahorrado tener un Helm Chart?*

Información necesaria para su realización

Como se dijo en la teoría, ***Helm* es un administrador de paquetes para *Kubernetes***, es decir, una herramienta para instalar y administrar aplicaciones que se ejecutan en él. En este ejercicio vamos a usar *Helm* para desplegar una aplicación compleja, **el sistema de cachés distribuidas** `memcached`.

El objetivo de este ejercicio, además de tener un poco de experiencia con el manejo básico de *Helm*, es que veáis como gracias a él la puesta en marcha de una aplicación compleja en un clúster *Kubernetes* es bastante sencilla, aunque inicialmente no sepamos cómo funciona lo que vamos a instalar.

Introducción: Qué es Memcached

Memcached es un **sistema de almacenamiento en caché** de código abierto y multifuncional de uso muy extendido. Funciona como un almacén temporal de datos para acelerar el acceso a bases de datos o para

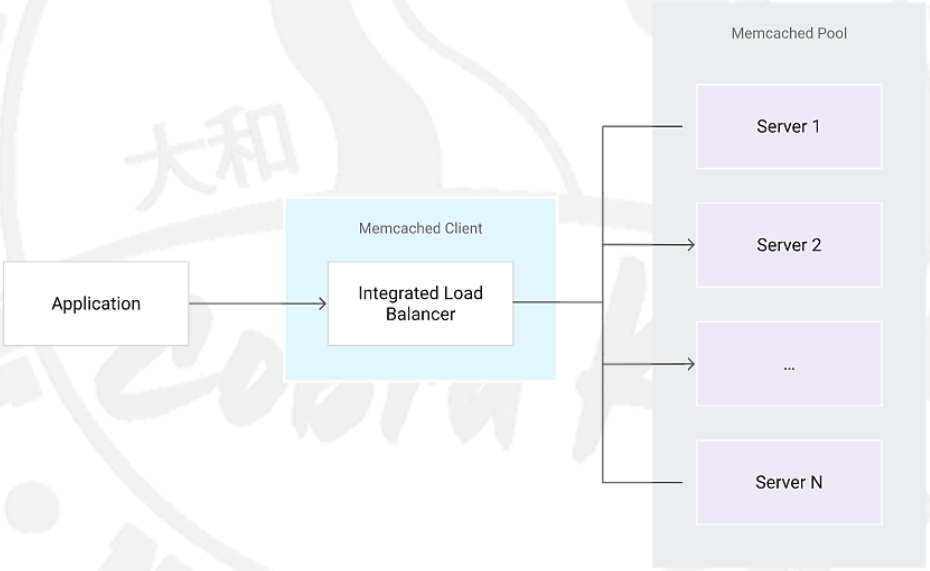
guardar datos compartidos, como por ejemplo IDs de sesión en clústeres *Tomcat* que repliquen el estado de las aplicaciones web que alojan. Tiene dos objetivos de diseño principales:

- **Simplicidad**: funciona como una gran tabla *hash* y ofrece una API simple con la que almacenar y recuperar objetos de cualquier tipo y estructura mediante claves.
- **Rapidez**: almacena los datos exclusivamente en RAM.

Memcached puede escalar su tabla *hash* horizontalmente a través de un grupo de servidores. Cada servidor de *Memcached* opera de forma independiente de los otros del grupo. Por lo tanto, **el enrutamiento y el balanceo de carga entre los servidores debe realizarse en el cliente**. Los clientes de *Memcached* aplican un esquema de *hashing* coherente para seleccionar el servidor al que hacer la petición, garantizando lo siguiente:

- Siempre se selecciona el mismo servidor para una misma clave.
- El uso de memoria se balancea de forma uniforme entre los servidores.
- Se reubica un número mínimo de claves cuando el grupo de servidores se reduce o amplía.

Este dibujo (**Fuente**: https://cloud.google.com/architecture/deploying-memcached-on-kubernetes-engine?hl=es-419) representa la interacción entre un cliente *Memcached* y un grupo distribuido de servidores de *Memcached*.



Como puedes verse, este servicio es complejo y su instalación manual puede llevarnos tiempo. Por suerte, para estos casos tenemos *Helm* y sus ***Helm Charts***: **aplicaciones listas para desplegar en *Kubernetes***. Existe un *Helm Chart* con un *Memcached* preconfigurado y listo para funcionar, con metadatos que describen la aplicación y la infraestructura necesaria para operarla con primitivas de *Kubernetes*, incluyendo todos los contenedores necesarios y el código de la aplicación a ejecutar. Veamos cómo usarlos en la siguiente sección.

Puesta en marcha de Memcached con Helm Charts

Helm trabaja con clústeres *Kubernetes* pero no forma parte de él, por lo que lo primero que debemos hacer es **descargar el binario de *Helm* así**:

```
HELM_VERSION=3.7.1
cd ~
```

```
wget https://get.helm.sh/helm-v${HELM_VERSION}-linux-amd64.tar.gz
```

Ahora tenemos que **descomprimir el archivo descargado** en el sistema en una carpeta adecuada:

```
mkdir helm-v${HELM_VERSION}
tar zxfv helm-v${HELM_VERSION}-linux-amd64.tar.gz -C helm-v${HELM_VERSION}
```

Y para poder usarlo directamente, **agregar el directorio anterior al** `PATH` con este comando, o bien añadiéndolo al `~/.bashrc` para que se mantenga entre reinicios: `export PATH="$(echo ~)/helm-v${HELM_VERSION}/linux-amd64:$PATH"`

Con *Helm* instalado ya estamos en disposición de descargarnos y usar el *Helm Chart* que controla la instalación de *Memcached* sobre nuestro clúster `minikube`. El *Chart* se encuentra en https://charts.bitnami.com/bitnami y es muy complejo, pero entenderlo está fuera del alcance de esta asignatura: **sólo lo vamos a usar**. Para ello hacemos:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install mycache bitnami/memcached --set architecture="high-availability" --set autoscaling.enabled="true"
```

Este *Helm Chart* para *Memcached* usa un controlador *StatefulSet* que hace que los nombres de los *pods* generados para `memcached` sigan una secuencia ordenada predecible (Ej.: `mycache-memcached-{0..2}`), lo que facilita que los clientes de *Memcached* hagan referencia a ellos. Esto se debería ver si consultamos los *pods* creados una vez instalado el *Chart*, esperando a que se creen todos los *pods* necesarios:

```
CHART NAME: memcached
CHART VERSION: 6.2.3
APP VERSION: 1.6.17

** Please be patient while the chart is being deployed **

Memcached can be accessed via port 11211 on the following DNS name from within y
our cluster:

    mycache-memcached.default.svc.cluster.local

Please see https://github.com/memcached/memcached/wiki/ConfiguringClient to unde
rstand the Memcached model and need for client-based consistent hashing.
You might also want to consider more advanced routing/replication approaches wit
h mcrouter: https://github.com/facebook/mcrouter/wiki/Replicated-pools-setup
ssiuser@vagrant:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
mycache-memcached-0           1/1     Running   0          67s
mycache-memcached-1           1/1     Running   0          52s
mycache-memcached-2           1/1     Running   0          52s
nginx-6799fc88d8-62sfd        1/1     Running   0          18h
postgres-5b99f887b7-blqnr     1/1     Running   0          60m
webapp-748d597874-8fggn       1/1     Running   0          48m
ssiuser@vagrant:~$
```

Usando los servicios de Memcached

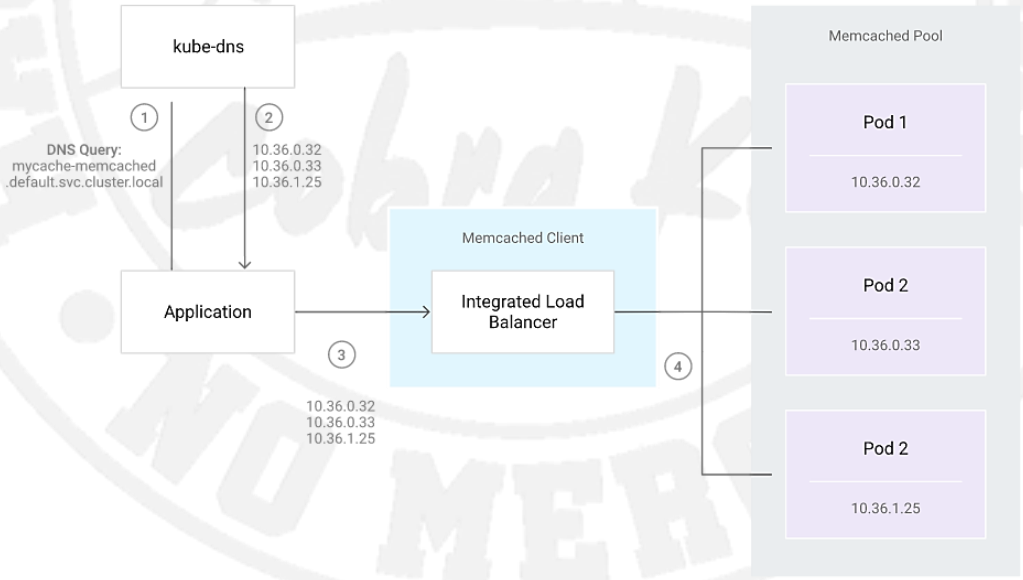
El *Helm Chart* que hemos usado usa un ***headless service***, lo que expone las direcciones IP para todos sus *pods* de modo que puedan detectarse individualmente, algo que es necesario para los clientes de *Memcached* de acuerdo con el diagrama anterior. Para que un cliente sepa estas IP debe hacer `kubectl get endpoints mycache-memcached`, obteniendo este resultado, cuyas IPs en vuestra instalación serán muy probablemente distintas:

```
ssiuser@vagrant:~$ kubectl get endpoints mycache-memcached
NAME                ENDPOINTS                                         AGE
mycache-memcached   10.0.0.235:11211,10.0.0.44:11211,10.0.0.94:11211   2m54s
ssiuser@vagrant:~$
```

Como puede verse, cada *pod* queda a la escucha del puerto `11211`, que es el **puerto predeterminado de *Memcached***. Ahora, para probar la implementación abrimos una sesión de `telnet` interactiva con uno de los servidores de *Memcached* en ejecución en el puerto `11211`, usando un contenedor mínimo que se borre cuando salgamos de consola, así: `kubectl run -it --rm busybox --image=busybox:1.33 --restart=Never telnet mycache-memcached-0.mycache-memcached.default.svc.cluster.local 11211`. Hecho esto, en el símbolo del sistema de `telnet`, ejecutamos estos comandos mediante el protocolo ASCII de *Memcached*, que deberían tener la siguiente salida:

```
set mykey 0 0 5
hello
STORED
get mykey
VALUE mykey 0 5
hello
END
```

Si todas estas pruebas son satisfactorias, ya podemos implementar la lógica básica de descubrimiento de servicios de este diagrama (**Fuente**: https://cloud.google.com/architecture/deploying-memcached-on-kubernetes-engine?hl=es-419):



Para ello **arrancaremos un contenedor interactivo mínimo con *Python*** instalado que se borre cuando salgamos de su consola así: `kubectl run -it --rm python --image=python:3.10-alpine --restart=Never sh`. Una vez dentro, instalamos la librería `pymemcache` con `pip install pymemcache` y luego arrancamos *Python* y ejecutamos este programa, cuyo resultado debería ser el de la siguiente captura.

```
import socket
```

```
from pymemcache.client.hash import HashClient
_, _, ips = socket.gethostbyname_ex('mycache-memcached.default.svc.cluster.local')
servers = [(ip, 11211) for ip in ips]
client = HashClient(servers, use_pooling=True)
client.set('mykey', 'hello')
client.get('mykey')
```

```
/ # python
Python 3.10.6 (main, Aug 10 2022, 00:19:55) [GCC 11.2.1 20220219] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> from pymemcache.client.hash import HashClient
>>> _, _, ips = socket.gethostbyname_ex('mycache-memcached.default.svc.cluster.l
ocal')
>>> servers = [(ip, 11211) for ip in ips]
>>> client = HashClient(servers, use_pooling=True)
>>> client.set('mykey', 'hello')
True
>>> client.get('mykey')
b'hello'
>>>
```

Lo que está haciendo el programa es lo siguiente:

- **Consulta el servicio** `kube-dns` de nuestro clúster `minikube` para obtener las IPs del registro DNS de `mycache-memcached.default.svc.cluster.local`.
    - Este nombre se creó así porque en el *Helm Chart* el servicio se llama `mycache-memcached` y al no especificar un *namespace* estamos usando el *namespace* `default` de *Kubernetes*.
- **Obtiene las direcciones IP asociadas** con ese registro DNS, las tres de los *pods* que vimos antes.
- **Crea un nuevo cliente de *Memcached*** (`HashClient`) y le pasa esas direcciones IP.
- **El balanceador de carga del cliente de *Memcached* se conecta** con los servidores de *Memcached* en las direcciones IP proporcionadas y ya podemos guardar y leer valores desde el programa.

Ahora ya podemos cerrar la consola de *Python* con `exit()` y salir de la sesión de shell del *pod* con `Control + D`.

Insignias y Autoevaluación

Esta asignatura hace uso de insignias. Esto quiere decir que, cuando vayas desbloqueando niveles de maestría en la tecnología, podrás obtener **una insignia pública** que luego podrás usar en tu currículum gracias al soporte de estas del campus virtual. Este enlace te enseña a usar esta característica para poder unirlas a tu currículum: https://docs.moodle.org/all/es/Insignias. Cada insignia pública **lleva adjuntos los textos de las habilidades que desbloquea** que puedes consultar en la siguiente tabla, para que cualquiera los vea si decides usarlas y sepa lo que has conseguido hacer, validado por mí. De esta manera, mi intención es ayudarte a aumentar tus posibilidades de contratación en un futuro, **exponiendo claramente lo que has podido hacer en esta asignatura** referente a cada tecnología sobre la que has decidido evaluarte.

Rango de Maestría en la Tecnología y Habilidades Demostradas		Insignia Pública Obtenida
Nivel 1: Cabo de *Kubernetes*		
	Puede hacer una instalación operativa de *Minikube*	
	Puede desplegar una página web simple en un *deployment* de un clúster *Kubernetes*	
	Es capaz de obtener información de un *deployment* cualquiera	
	Puede interaccionar con un *pod* de un clúster *Kubernetes*	
Nivel 2: Sargento de *Kubernetes*		
	Puede crear servicios asociados a *deployments Kubernetes*	
	Puede manejar etiquetas de un servicio y usarlas para buscarlos	
	Puede crear *deployments* y servicios con ficheros de código YAML ***(2 ejercicios)***	
	Puede desplegar una aplicación web simple en *Kubernetes*	
Nivel 3: Teniente de Navío de *Kubernetes*		
	Puede gestionar de forma gráfica un clúster *Kubernetes* con su *dashboard*	
	Puede escalar manualmente un servicio de *Kubernetes* y comprobar como vuelve a su estado deseado si algunas réplicas se pierden ***(2 ejercicios)***	
	Puede hacer pruebas de carga sobre un servicio	
	Puede desplegar un volumen persistente y asociarlo a un *deployment* en un clúster *Kubernetes*	
Nivel 4: Capitán de Navío *Kubernetes*		
	Es capaz de activar, probar y usar el auto-escalado de servicios en *Kubernetes* ***(2 ejercicios)***	
	Es capaz de usar *ConfigMaps* y *Secrets* para la gestión adecuada de información secreta en una aplicación desplegada en *Kubernetes*	

Nivel 5: Almirante *Kubernetes*		
	Puede desplegar una aplicación web en capas en un clúster *Kubernetes*	
	Puede controlar los accesos vía red entre servicios gracias a las políticas de red	
	Sabe cómo hacer un despliegue de un *Helm Chart* simple en un clúster *Kubernetes* existente y entiendes la utilidad del servicio [redacted]	
[redacted]	[redacted]	[redacted]
	**Cruz con el distintivo blanco en *Kubernetes***: Ha completado con éxito **4 puntos** (aproximadamente el 80%) de las tareas correspondientes de los 5 niveles de la parte de *Kubernetes*, demostrando así su maestría en esta tecnología y estar preparado para usarla en tu futuro trabajo de forma efectiva.