Acceso a las aplicaciones: Services

Acceso a las aplicaciones: Services

Los servicios (Services) nos permiten acceder a las aplicaciones que hemos desplegado en el cluster.

- Un Service es una abstracción que **nos permite acceder a un conjunto de pods** (que se han creado a partir de un Deployment) que implementan una aplicación (Por ejemplo: acceder a un servidor web, a una servidor de base de datos, a un servicio que forma parte de una aplicación, ...).
- A cada Pod se le asigna una IP a la que no se puede acceder directamente, por lo tanto necesitamos un Service que nos ofrece **una dirección virtual (CLUSTER-IP) y un nombre** que identifica al conjunto de Pods que representa, al cual nos podemos conectar.
- La conexión al Service se puede realizar **desde otros Pods o desde el exterior** (mediante la generación aleatoria de un puerto). Por ejemplo, si tenemos una aplicación formada por dos Services: servidor web y servidor de base de datos, tendremos que acceder desde el exterior al servidor web, y acceder al servidor de base de datos desde el servidor web. En principio no será necesario acceder al servidor de base de datos desde el exterior.
- Si el Deployment que hemos creado tiene más de un Pod asociado, el Service que representa el acceso a esta aplicación **balanceará la carga** entre los Pods con una política Round Robin.
- En el cluster existirá un componente que nos ofrece un **servicio DNS**. Cada vez que creamos un Service se actualizará el DNS para resolver el nombre que hemos asignado al Service con la IP virtual (CLUSTER-IP) que se le ha asignado.
- **Nota** Cuando tenemos más de un Pod ofreciendo el mismo servicio, realmente tenemos un clúster y es importante distinguir entre servicios sin estado (*stateless*) o con estado (*stateful*)). En un servicio sin estado (por ejemplo, un servidor web que sirva contenido estático), las peticiones son independientes y se pueden servir por diferentes nodos sin problema, aunque en el caso de un servidor web, deberíamos asegurarnos previamente de que el directorio con los datos es el mismo. Un servicio de este tipo lo podemos escalar con un despliegue sin problema. Por otra parte, si el servicio tiene estado (por

ejemplo, un servidor de bases de datos), una petición puede depender de otra anterior, por lo que puede haber incoherencias si simplemente creamos un cluster de nodos iguales. En este tipo de servicios, es necesaria una configuración adicional que controle el estado y que haga que los datos que sirve cada Pod son coherentes entre sí. Veremos un ejemplo de este tipo de servicios en el módulo 9 del curso.

Tipos de Services

ClusterIP

Solo se permite el acceso interno a un Service de este tipo. Es decir, si tenemos un despliegue con una aplicación a la que no es necesario acceder desde el exterior, crearemos un Service de este tipo para que otras aplicaciones puedan acceder a ella (por ejemplo, una base de datos). Es el tipo por defecto. Si deseamos seguir accediendo desde el exterior, para hacer pruebas durante la fase de desarrollo podemos seguir utilizando la instrucción `kubectl port-forward`.

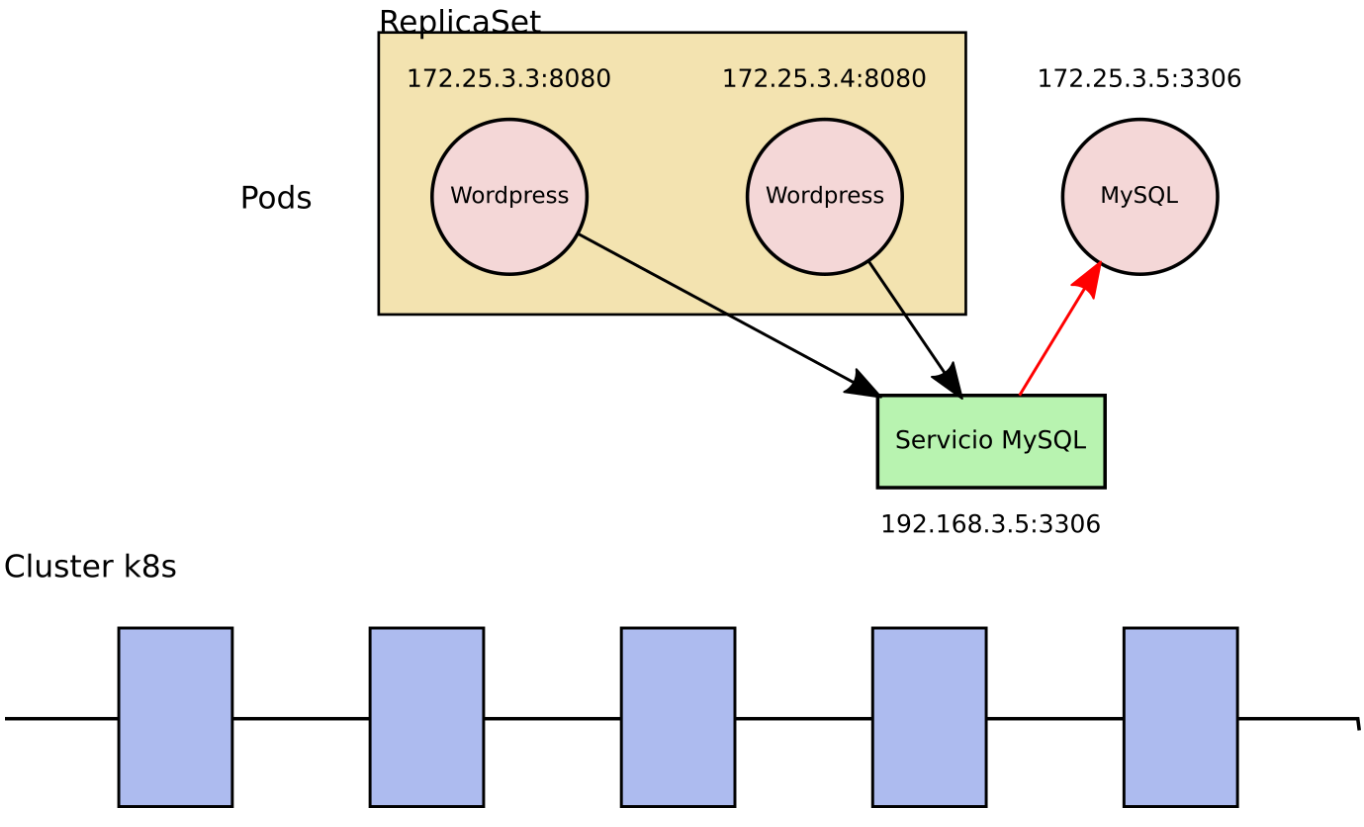


Imagen de elaboración propia. *Servicio ClusterIP* (CC BY-NC-SA)

Veamos el ejemplo:

1. Necesitamos que los Pods de Wordpress accedan al Pod del MySQL.

2. La IP que ha tomado el Pod de MySQL (`172.25.3.5`) es inaccesible desde los Pods de Wordpress.
3. Por lo tanto hemos creado un Service de tipo ClusterIP, que ha obtenido una ip virtual (`192.168.3.5`) y expone el puerto de MySQL 3306.
4. Esta IP sí es accesible desde los Pods de Wordpress.
5. Al acceder a esta IP se balanceará la carga entre los Pods de MySQL (en el ejemplo sólo tenemos uno).
6. Además en el Wordpress no necesitamos configurar la IP virtual del Service que hemos creado, ya que disponemos de un servidor DNS que resuelve el nombre del Service `mysql` en la dirección virtual del Service (`192.168.3.5`). Por lo tanto en la configuración de Wordpress pondremos el nombre `mysql` como host del servidor de base de datos al que debe acceder.

NodePort

Abre un puerto, para que el Service sea accesible desde el exterior. Por defecto el puerto generado está en el rango de 30000:40000. Para acceder usamos la ip del servidor master del cluster y el puerto asignado.

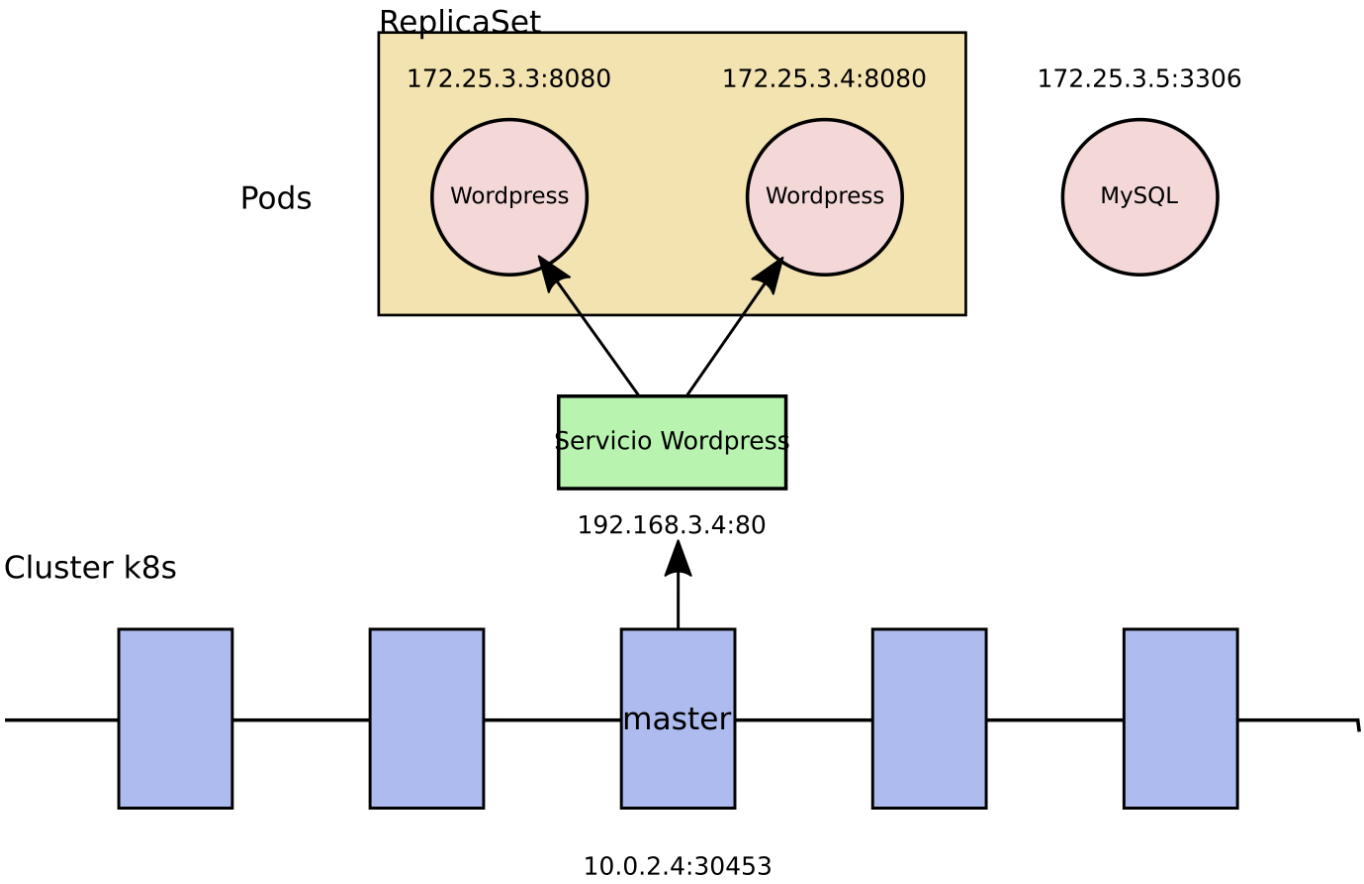


Imagen de elaboración propia. *Servicio NodePort* (CC BY-NC-SA)

Veamos el ejemplo:

1. Necesitamos que los Pods de Wordpress sean accesibles desde el exterior, para que podamos acceder a la aplicación.
2. La IP que han tomado los Pods de Wordpress (`172.25.3.3`, ...) no son accesibles desde el exterior. Además comprobamos que estos Pods están ofreciendo el servicio en el puerto 8080.
3. Por lo tanto, hemos creado un Service de tipo NodePort que ha obtenido una IP virtual (`192.168.3.4`) y expone el puerto 80.
4. Al acceder a esta IP al puerto 80 se balanceará la carga entre los Pods de Wordpress, accediendo a las IPs de los Pods de Wordpress al puerto 8080.
5. El Service NodePort ha asignado un puerto de acceso aleatorio (entre el 30000 - 40000) que nos permite acceder a la aplicación mediante la IP del nodo master. En el ejemplo si accedemos a `10.0.2.4:30453` estaremos accediendo al Service que nos permitirá acceder a la aplicación.

LoadBalancer

Este tipo sólo está soportado en servicios de cloud público (GKE, AKS o AWS). El proveedor asignará un recurso de balanceo de carga para el acceso a los Services. Si usamos un cloud privado como OpenStack, necesitaremos un plugin para configurar el funcionamiento. Este tipo de Service no lo vamos a utilizar en el presente curso.

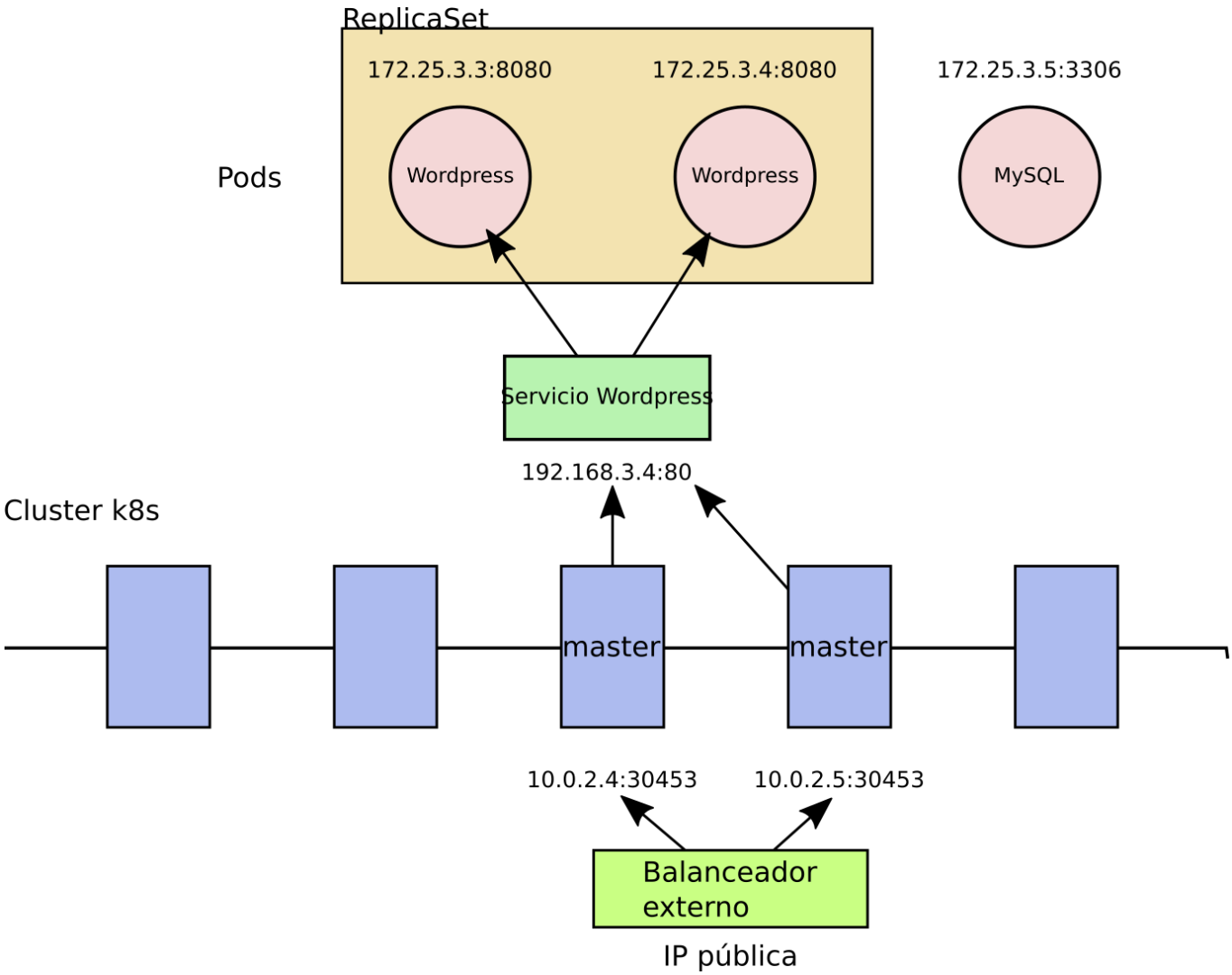


Imagen de elaboración propia. *Servicio LoadBalancer* (CC BY-NC-SA)

Como vemos en el ejemplo, el cloud de infraestructura donde tengamos instalado el cluster nos ofrecerá un recurso *balanceador de carga* con una IP accesible desde el exterior que nos permitirá acceder a la aplicación directamente.

Vídeo: Acceso a las aplicaciones: Services

https://www.youtube.com/embed/kI2rZmqA7TI

Vídeo: Acceso a las aplicaciones: Services

Describiendo Services

Services NodePort

Suponemos que tenemos desplegado nginx usando el fichero yaml: `nginx-deployment.yaml`:

```
kubectl apply -f nginx-deployment.yaml
```

Por lo tanto tenemos dos Pods ofreciendo el servidor web nginx, a los que queremos acceder desde el exterior y que se balancee la carga entre ellos.

Aunque podríamos crear un recurso Service desde la línea de comandos:

```
kubectl expose deployment/nginx --port=80 --type=NodePort
```

Normalmente lo que hacemos es describir las características del Service en un fichero yaml `nginx-srv.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: NodePort
  ports:
  - name: service-http
    port: 80
    targetPort: http
  selector:
    app: nginx
```

Veamos la descripción:

- Vamos a crear un recurso Service (parámetro `kind`) y lo nombramos como `nginx` (parámetro `name`). Este nombre será importante para la resolución dns.
- En la especificación del recurso indicamos el tipo de Service (parámetro `type`).
- A continuación, definimos el puerto por el que va a ofrecer el Service y lo nombramos (dentro del apartado `port`: el parámetro `port` y el parámetro `name`). Además, debemos indicar el puerto en el que los Pods están ofreciendo el Service (parámetro `targetPort`),

en este caso, hemos usado el nombre del puerto (`http`) que indicamos en el recurso Deployment:

```
...
ports:
 - name: http
   containerPort: 80
...
```

- Por ultimo, seleccionamos los Pods a los que vamos acceder y vamos a balancear la carga seleccionando los Pods por medio de sus etiquetas (parámetro `selector`).

Nota

Nota: La definición de un Service de tipo ClusterIP sería exactamente igual, pero cambiando el parámetro `type`.

Para seguir aprendiendo

- Para más información acerca de los Services puedes leer: la documentación de la API y la guía de usuario.

Vídeo: Describiendo Services

https://www.youtube.com/embed/Qr9YWGTL_W8

Vídeo: Describiendo Services

Gestionando los Services

Service de tipo NodePort

Para aprender cómo gestionamos los Services, vamos a trabajar con el Deployment de nginx (`nginx-deployment.yaml`) y el Service NodePort (`nginx-srv.yaml`) para acceder a los Pods de este despliegue desde el exterior.

Creamos el Deployment

El primer paso sería crear el Deployment de nginx:

```
kubectl apply -f nginx-deployment.yaml
```

Creamos el Service

A continuación vamos a crear el Service de tipo NodePort que nos permitirá acceder al servidor nginx.

```
kubectl apply -f nginx-srv.yaml
```

Para ver los Services que tenemos creado:

```
kubectl get services
```

Recuerda que si usamos `kubectl get all` también se mostrarán los Services.

Antes de acceder a la aplicación podemos ver la información más detallada del Service que acabamos de crear:

```
kubectl describe service/nginx
Name:                     nginx
...
Selector:                 app=nginx
Type:                     NodePort
...
IP:                       10.110.81.74
Port:                     service-http  80/TCP
TargetPort:               http/TCP
```

```
NodePort:                 service-http  32717/TCP
Endpoints:                172.17.0.3:80,172.17.0.4:80
...
```

Podemos ver la etiqueta de los Pods a los que accede (`Selector`). El tipo de Service (`Type`). La IP virtual que ha tomado (CLUSTER-IP) y que es accesible desde el cluster (`IP`). El puerto por el que ofrece el Service (`Port`). El puerto de los Pods a los que redirige el tráfico (`TargetPort`). Al ser un service de tipo NodePort nos da información del puerto que se asignado para acceder a la aplicación (`NodePort`). Y por último, podemos ver las IPs de los Pods que ha seleccionado y sobre los que balanceará la carga (`Endpoints`).

Accediendo a la aplicación

Vemos el Service que hemos creado:

```
kubectl get services
...
nginx        NodePort    10.110.81.74   <none>        80:32717/TCP   32s
```

Observamos que se ha asignado el puerto 32717 para el acceso, por lo tanto si desde un navegador accedemos a la IP del nodo master y a este puerto podremos ver la aplicación.

¿Cómo sé la dirección ip del nodo master del cluster minikube? Podemos ejecutar:

```
minikube ip
192.168.39.222
```

Y ya podemos acceder desde un navegador web:

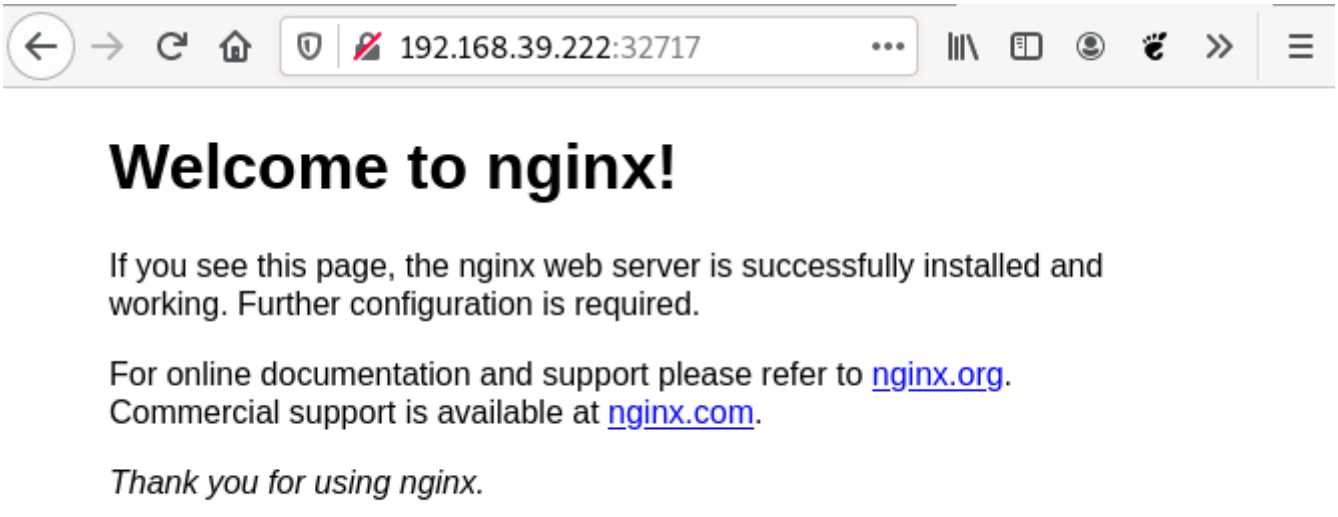


Imagen de elaboración propia. *Service NodePort* (CC BY-NC-SA)

Service ClusterIP

En esta ocasión vamos a desplegar una base de datos MariaDB. En este caso no vamos a necesitar acceder a la base de datos desde el exterior, pero necesitamos que los Pods de otro despliegue puedan acceder a ella. Por lo tanto vamos a crear un Service de tipo ClusterIP.

Para el despliegue de MariaDB vamos a usar el fichero `mariadb-deployment.yaml`. Puedes comprobar que en la definición del contenedor hemos añadido la sección `env` que nos permite establecer variables de entorno para configurar el contenedor (los estudiaremos en el siguiente módulo).

Para la creación del Service utilizamos el fichero `mariadb-srv.yaml`.

Para la creación del Deployment y el Service vamos ejecutando las siguientes instrucciones:

```
kubectl apply -f mariadb-deployment.yaml
kubectl apply -f mariadb-srv.yaml
```

Comprobamos el Service creado:

```
kubectl get services
mariadb      ClusterIP   10.106.60.233   <none>        3306/TCP       2m22s

kubectl describe service/mariadb
Name:              mariadb
...
Selector:          app=mariadb
Type:              ClusterIP
...
IP:                10.106.60.233
Port:              service-bd  3306/TCP
TargetPort:        db-port/TCP
Endpoints:         172.17.0.5:3306
...
```

Podemos comprobar que no se ha mapeado un puerto aleatorio para que accedamos usando la IP del nodo master. Los Pods que accedan a la IP 10.106.60.233 o al nombre `mariadb` y al puerto 3306 estarán accediendo al Pod (172.17.0.5:3306) del despliegue de mariadb.

Eliminando los servicios

Por ejemplo para borrar el servicio `mariadb`, ejecutaríamos:

```
kubectl delete service mariadb
```

Vídeo: Gestionando los Services

https://www.youtube.com/embed/UAaBzXG13XU

Vídeo: Gestionando los Services

Servicio DNS en Kubernetes

Existe un componente de Kubernetes llamado CoreDNS, que ofrece un servidor DNS interno para que los Pods puedan resolver diferentes nombres de recursos (Services, Pods, ...) a direcciones IP.

Cada vez que se crea un nuevo recurso Service se crea un registro de tipo A con el nombre:

`<nombre_servicio>.<nombre_namespace>.svc.cluster.local.`

Comprobemos el servidor DNS

Partimos del punto anterior donde tenemos creados los dos Services:

```
kubectl get services
mariadb      ClusterIP   10.106.60.233   <none>        3306/TCP
nginx        NodePort    10.110.81.74    <none>        80:32717/TCP
```

Para comprobar el servidor DNS de nuestro cluster y que podemos resolver los nombres de los distintos Services, vamos a usar un Pod (busybox.yaml) creado desde una imagen `busybox`. Es una imagen muy pequeña pero con algunas utilidades que nos vienen muy bien:

```
kubectl apply -f busybox.yaml
```

¿Qué servidor DNS está configurado en los Pods que estamos creando? Podemos ejecutar la siguiente instrucción para comprobarlo:

```
kubectl exec -it busybox -- cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
```

- El servidor DNS (componente coreDNS) tiene asignado la IP del cluster 10.96.0.10.
- Podemos utilizar el nombre corto del Service, porque buscará el nombre del host totalmente cualificado usando los dominios indicados en el parámetro `search`. Como vemos el primer nombre de dominio es el que se crea con los Services: `default.svc.cluster.local` (recuerda que el *namespace* que estamos usando es `default`).

Vamos a comprobar que realmente se han creado dos registros A para cada uno de los Service, haciendo consultas DNS:

```
kubectl exec -it busybox -- nslookup nginx
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:   nginx.default.svc.cluster.local
Address: 10.110.81.74
```

Vemos que ha hecho la resolución del nombre `nginx` con la IP correspondiente a su servicio. Y con el Service mariadb también lo podemos hacer:

```
kubectl exec -it busybox -- nslookup mariadb
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:   mariadb.default.svc.cluster.local
Address: 10.106.60.233
```

También podemos comprobar que usando el nombre podemos acceder al servicio:

```
kubectl exec -it busybox -- wget http://nginx
Connecting to nginx (10.110.81.74:80)
saving to 'index.html'
...
```

Podemos concluir que, cuando necesitemos acceder desde alguna aplicación desplegada en nuestro cluster a otro servicio ofrecido por otro despliegue, **utilizaremos el nombre que hemos asignado a su Service de acceso**. Por ejemplo, si desplegamos un Wordpress y un servidor de base de datos mariadb, y creamos dos Services: uno de tipo NodePort para acceder desde el exterior al CMS, y otro, que llamamos `mariadb` de tipo ClusterIP para acceder ala base de datos, cuando tengamos que configurar el Wordpress para indicar la dirección de la base de datos, pondremos `mariadb`.

Vídeo: Servicio DNS en Kubernetes

https://www.youtube.com/embed/nxnyRvdHpsI

Vídeo: Servicio DNS en Kubernetes

Ingress Controller

Hasta ahora tenemos dos opciones principales para acceder a nuestras aplicaciones desde el exterior:

1. Utilizando Services del tipo NodePort: Esta opción no es muy viable para entornos de producción ya que tenemos que utilizar puertos aleatorios desde 30000-40000.
2. Utilizando Services del tipo LoadBalancer: Esta opción sólo es válida si trabajamos en un proveedor Cloud que nos ofrece un balanceador de carga para cada una de las aplicaciones, en cloud público puede ser una opción muy cara.

La solución puede ser utilizar un Ingress controller que nos permite utilizar un proxy inverso (HAproxy, nginx, traefik,...) que por medio de reglas de encaminamiento que obtiene de la API de Kubernetes, nos permite el acceso a nuestras aplicaciones por medio de nombres.

Instalación de Ingress Controller en minikube

Cuando hacemos una instalación de minikube el componente de Ingress Controller no viene instalada por defecto. minikube nos ofrece un conjunto de *addons* que al activarlos nos instalan un determinado componente que nos ofrece una funcionalidad adicional. Para ver los *addons* que nos ofrece minikube podemos ejecutar:

```
minikube addons list
```

Para activar el Ingress Controller ejecutamos:

```
minikube addons enable ingress
```

Para comprobar si tenemos instalado el componente, podemos visualizar los Pods creados en el *namespace* `ingress-nginx`. Este espacio de nombre se ha creado para desplegar el controlador de ingress Por lo tanto al ejecutar:

```
kubectl get pods -n ingress-nginx
...
ingress-nginx-controller-558664778f-shjzp   1/1     Running     0
...
```

Debe aparece un Pod que se llama `ingress-nginx-controller-...`, si es así, significa que se ha instalado un Ingress Controller basado en el proxy inverso nginx.

Describiendo el recurso Ingress

Una vez instalado el componente Ingress Controller, ya podemos definir un recurso Ingress en un fichero yaml. Para ello vamos a trabajar con el despliegue y el Service que hemos creado de nginx.

El recurso Ingress para acceder a nuestro despliegue de nginx lo tenemos en el fichero `ingress.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  rules:
  - host: www.example.org
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nginx
            port:
              number: 80
```

Hemos indicado el tipo de recurso Ingress (`kind`) y le hemos puesto un nombre (`name`). A continuación en la especificación del recurso vamos a poner una regla que relaciona un nombre de host con un Service que me permita el acceso a una aplicación:

- `host`: Indicamos el nombre de host que vamos a usar para el acceso. Este nombre debe apuntar a la ip del nodo master.
- `path`: Indicamos el path de la url que vamos a usar, en este caso sería la ruta raíz: `/`. Esto nos sirve por si queremos servir la aplicación en una ruta determinada, por ejemplo: `www.example.org/app1`.
- `pathType`: No es importante, nos permite indicar cómo se van a trabajar con las URL.
- `backend`: Indicamos el Service al que vamos a acceder. En este caso indicamos el nombre del Service (`service/name`) y el puerto del Service (`service/port/number`).

Cuando se crea el recurso, y accedamos al nombre indicado, un proxy inverso redirigirá las peticiones HTTP a la IP y al puerto del Service correspondiente. **Nota: Utilizando Ingress no es**

necesario que los Services sean de tipo NodePort para acceder a la aplicación desde el exterior.

Gestionando el recurso Ingress

Para crear el recurso Ingress:

```
kubectl apply -f ingress.yaml
```

Y podemos ver el recurso Ingress que hemos creado:

```
kubectl get ingress
```

Y obtener información detallada del recurso con:

```
kubectl describe ingress/nginx
```

Accediendo a la aplicación

Como no tenemos un servidor DNS que nos permita gestionar los nombres que vamos a utilizar para el acceso a las aplicaciones, vamos a usar resolución estática. Para ello como `root` añadimos una nueva línea en el fichero `/etc/hosts`, indicando el nombre (`www.example.org`) y la ip a la que corresponde, la ip del nodo master:

```
192.168.39.222  www.example.org
```

Y ya podemos acceder a la aplicación usando el nombre:

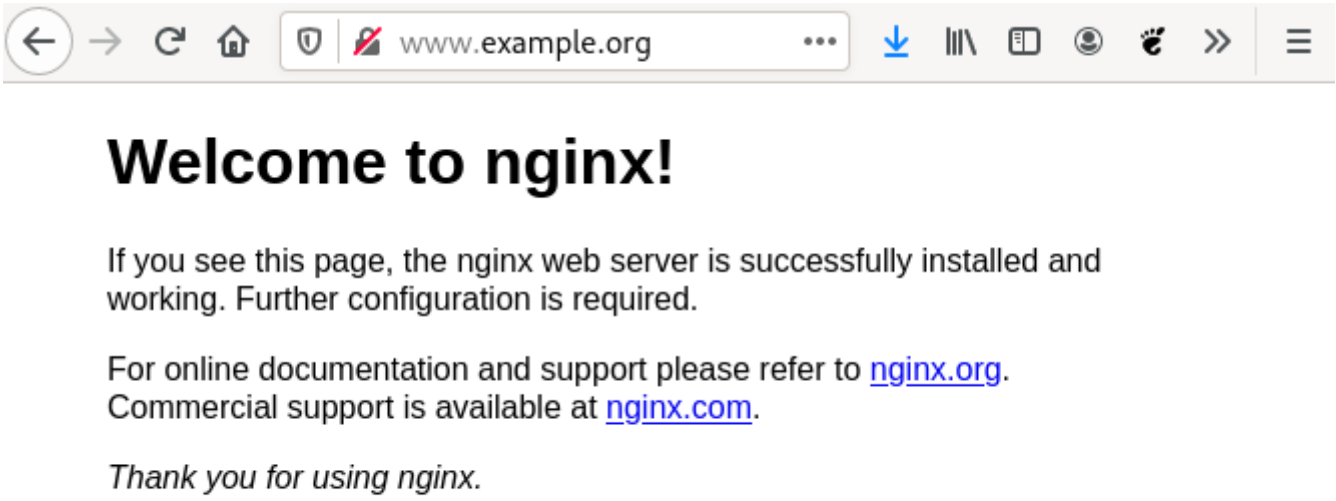


Imagen de elaboración propia. *Ingress Controller* (CC BY-NC-SA)

Vídeo: Ingress Controller

https://www.youtube.com/embed/X2dW9UbfU88

Vídeo: Ingress Controller

Ejemplo completo: Desplegando y accediendo a la aplicación Temperaturas

Vamos a hacer un despliegue completo de una aplicación llamada **Temperaturas**. Esta aplicación nos permite consultar la temperatura mínima y máxima de todos los municipios de España y estará formada por dos microservicios:

- `frontend`: Es una aplicación escrita en Python que nos ofrece una página web para hacer las búsquedas y visualizar los resultados. Este microservicio hará peticiones HTTP al segundo microservicio para obtener la información.
- `backend`: Es el segundo microservicio que nos ofrece un servicio web de tipo API Restful. A esta API Web podemos hacerles consultas sobre los municipios y sobre las temperaturas.

Algunas consideraciones sobre el despliegue que vamos a realizar:

- Como la aplicación está formada por dos microservicios, tendremos que crear dos recursos Deployment para desplegar y controlar los Pods de cada despliegue por separado.
- Necesitaremos acceder desde el exterior al microservicio `frontend` por lo que crearemos un recurso Service de tipo NodePort.
- Al componente `backend` no es necesario acceder desde el exterior, por lo tanto crearemos un recurso Service de tipo ClusterIP para permitir que se acceda desde `frontend`.
- Para terminar usaremos un recurso Ingress para acceder al componente `frontend` por lo tanto, en ese momento, no es necesario que el Service para acceder a ese componente sea de tipo NodePort, bastaría con que fuera ClusterIP.

Despliegue y acceso al microservicio frontend

En primer lugar vamos a desplegar el primer microservicio. Esta aplicación está usando el puerto 3000/tcp para ofrecer la aplicación. Para describir el despliegue utilizaremos el fichero `frontend-deployment.yaml` y para crear el despliegue ejecutaremos la siguiente instrucción:

```
kubectl apply -f frontend-deployment.yaml
```

A continuación usamos el fichero `frontend-srv.yaml` para crear el Service NodePort:

```
kubectl apply -f frontend-srv.yaml
```

Obtenemos los recursos que hemos creado. Nos fijamos en el puerto que nos ha asignado por el Service NodePort (en mi caso el 30053). Vamos a acceder desde un navegador web usando la ip del nodo master y el puerto que nos han asignado:

Imagen de elaboración propia (CC BY-NC-SA)

Como podemos observar la aplicación nos muestra un mensaje de error: **"No puedo conectar con el servidor de temperaturas...**. Evidentemente el componente `frontend` está intentando conectar con el componente `backend` y, evidentemente, no puede, ya que ni la hemos desplegado, ni hemos creado el Service correspondiente.

Despliegue y acceso al microservicio backend

Es el momento de desplegar el segundo microservicio. Este microservicio ofrece un servicio API Restful en el puerto 5000/tcp. Para ello utilizaremos el fichero `backend-deployment.yaml` para realizar el despliegue y el fichero `backend-srv.yaml` para crear el Service.

```
kubectl apply -f backend-deployment.yaml
kubectl apply -f backend-srv.yaml
```

Si volvemos acceder al navegador y refrescamos la página, veremos que ya no nos sale el mensaje de error y podemos buscar la temperatura de nuestra ciudad:

Imagen de elaboración propia (CC BY-NC-SA)

Nota: Por defecto el componente frontend hace peticiones al componente backend utilizando el nombre `temperaturas-backend`, que es el nombre que hemos asignado al Service ClusterIP para el acceso al backend. En el próximo módulo veremos como podemos cambiar la configuración de frontend para cambiar el nombre del servicio web al que conecta.

Algunas consideraciones acerca del despliegue que hemos realizado

Esta manera de trabajar donde cada microservicio que forma parte de la aplicación (o si tenemos una aplicación que necesite varios servicios (servidor web, servidor de base de datos,...)) se despliega de forma separada usando distintos recursos Deployment nos proporciona las siguientes características:

1. Cada conjunto de Pods creado en cada despliegue ejecutará un solo proceso para ofrecer un servicio o microservicio.
2. Cada conjunto de Pods se puede escalar de manera independiente. Esto es importante ya que si identificamos que al acceder a alguno de los servicios se crea un cuello de botella, podemos escalarlo para tener más Pods ejecutando el servicio. En el ejemplo que

hemos desarrollado, se crearon 3 Pods del frontend y un Pod del backend, pero se pueden escalar independientemente los dos despliegues. Te invito a escalar los dos despliegues y comprobar que sigue funcionando la aplicación.
3. Las actualizaciones de los distintos servicios / microservicios no interfieren en el resto.
4. Lo estudiaremos en un módulo posterior, pero podremos gestionar el almacenamiento de cada servicio de forma independiente.
5. Ya lo hemos comentado, pero con esta aplicación podemos observar el balanceo de carga que realiza el Service al acceder al `frontend`: la aplicación visualiza el nombre del servidor que está ofreciendo la página. Por lo tanto si vamos refrescando la página con F5 observaremos cómo se va realizando el balanceo de carga y va cambiando el nombre del Pod al que está accediendo.

Acceso a la aplicación usando el Ingress Controller

Para terminar vamos a crear un recurso Ingress que nos posibilite acceder a la aplicación utilizando un nombre. Como hemos indicado al utilizar Ingress no es necesario que el Service al que accede sea de tipo NodePort, por lo tanto lo primero que vamos a hacer es cambiar el tipo de Service que accede a `frontend` y lo vamos a poner ClusterIP, para ello vamos a modificar el fichero `frontend-srv.yaml` cambiando el tipo de Service de NodePort a ClusterIP, y posteriormente aplicamos los cambios:

```
kubectl apply -f frontend-srv.yaml
```

Comprobamos que realmente ha cambiado el tipo de Service, y que ya no tenemos un puerto para acceder usando la ip del nodo master.

A continuación usamos el fichero `ingress.yaml` para crear el recurso Ingress, que definirá el nombre del host `www.temperaturas.org` que tendremos que introducir en la resolución estática tay como hemos visto anteriormente. Por lo tanto modificamos el fichero `/etc/hosts` de nuestro ordenador con la siguiente línea:

```
192.168.39.222  www.temperaturas.org
```

Creamos el recurso Ingress:

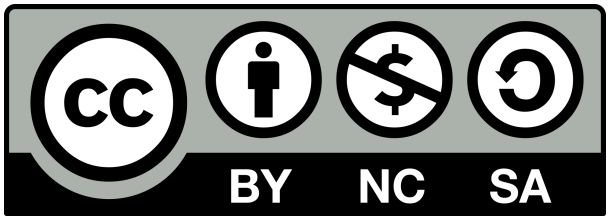
```
kubectl apply -f ingress.yaml
```

Y accedemos a la aplicación usando el nombre:

Imagen de elaboración propia (CC BY-NC-SA)

Créditos

Materiales desarrollados por:
Alberto Molina Coballes
José Domingo Muñoz Rodríguez
Propiedad de la Consejería de Educación y Deporte de la Junta de Andalucía



2021