PORTAINER

José Manuel Fernández Atienza

2º ASIR

Índice

1.Objetivos que se quieren conseguir y se han conseguido

1.1¿Qué quiero conseguir?

Este proyecto tiene como objetivo implementar una solución simplificada para la gestión de clústeres de Kubernetes mediante el uso de Portainer.

El uso de Portainer, permitirá a los usuarios gestionar su infraestructura de Kubernetes de manera más sencilla e intuitiva. Además, se integrará Longhorn como solución de volúmenes persistentes, ofreciendo una forma eficiente y flexible de manejar el almacenamiento.

Por lo tanto, el objetivo principal del proyecto es proporcionar una solución integral que permita a los usuarios gestionar tantas aplicaciones como almacenamiento con mayor facilidad.

Objetivos

1. Configurar un clúster de Kubernetes con tres nodos, un nodo máster y dos nodos de computo
2. La Implementación de la herramienta Portainer para gestionar dicho clúster
3. Conseguir un despliegue de las aplicaciones en alta disponibilidad
4. Lograr el auto escalado de las aplicaciones

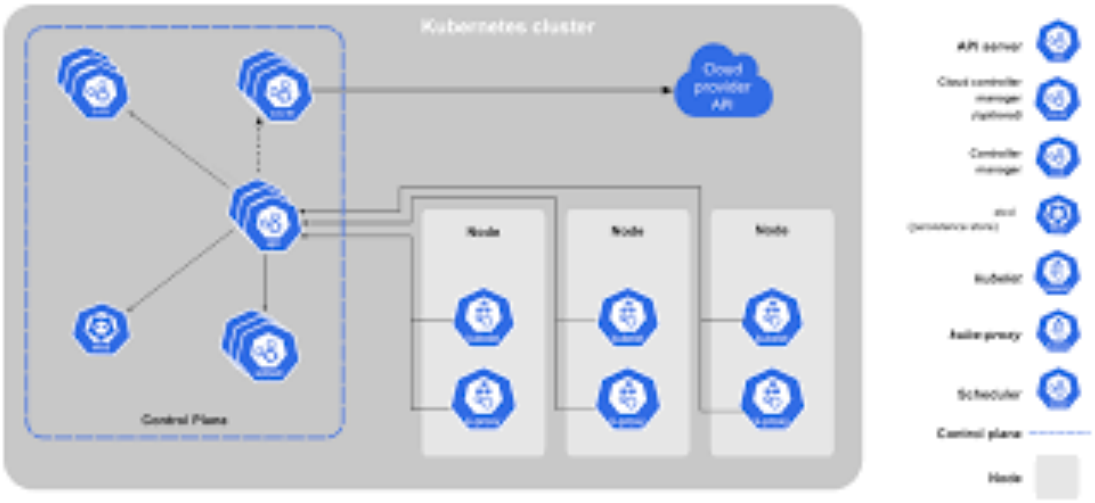
2.Fundamentos teóricos y conceptos

2.1 Kubernetes

Kubernetes es una plataforma de orquestación de contenedores de código abierto, diseñada para automatizar la implementación, escalabilidad y gestión de aplicaciones en contenedores.

Originalmente desarrollado por Google, Kubernetes permite a los usuarios gestiona aplicaciones distribuidas en múltiples nodos de manera eficiente.

Componentes de Kubernetes:



Control-plane:

Los componentes que forman el plano de control toman decisiones globales sobre el clúster y detectan y responden a eventos del clúster, como la creación de un nuevo pod cuando la propiedad réplicas de un controlador de replicación no se cumple.

- Kube-apiserver: Componente que expone la API de Kubernetes, recibe las peticiones y actualiza acordemente el estado en etcd.
- etcd: Almacén de datos persistente, consistente y distribuido de clave-valor utilizado para almacenar toda la información del clúster.
- kube-scheduler: Componente que está pendiente de los pods que no tienen ningún nodo asignado y selecciona un nodo donde ejecutarlo.
- Kube-controller-manager: Componente que ejecuta los controladores de Kubernetes. Estos controladores incluyen:
    - Controlador de nodos: responsable de detectar y responder cuando un nodo deja de funcionar.

    - Controlador de replicación: responsable de mantener el número correcto de pods.
    - Controlador de endpoints: construye el objeto endpoindts, es decir, hace la unión entre los services y los pods.
    - Controladores de Tokens y cuentas de servicio: crean cuentas y tokens de acceso a la API por defecto para los nuevos namespaces.
- Cloud-controller-manager: ejecuta controladores que interactúan con proveedores de la nube. Incluye:
    - Controlador de nodos: responsable de detectar y actuar cuando un nodo deja de responder.
    - Controlador de rutas: para configurar rutas en la infraestructura de nube subyacente.
    - Controlador de servicios: para crear, actualizar y eliminar balanceadores de carga en la nube.
    - Controlador de volúmenes: para crear, conectar y montar volúmenes e interactuar con el proveedor de la nube para orquestarlos.

Componentes de Nodo:

Los componentes de nodo corren en cada nodo, manteniendo a los pods en funcionamiento y proporcionando el entorno de ejecución de Kubernetes.

- Kubelet: Agente que se ejecuta en cada nodo de un clúster. Se asegura de que los contenedores estén corriendo en un pod.
- Kube-Proxy: permite abstraer un servicio en Kubernetes manteniendo las reglas de red en el anfitrión y haciendo reenvío de conexiones.
- Runtime de contenedores: software responsable de ejecutar los contenedores

Addons

Los addons son pods y servicios que implementan funcionalidades del clúster. Pueden ser administrados por deployments, ReplicationController y otros. Los addons asignados a un espacio de nombres se crean en el espacio kube-system.

2.2 Helm

Helm es un gestor de paquetes para Kubernetes, utilizado para automatizar la instalación, configuración y gestión de aplicaciones dentro de un clúster de Kubernentes.

Helm facilita la instalación y actualización de aplicaciones complejas a través de paquetes llamados Charts. Un chart es una colección de archivos que describen un conjunto de recursos de Kubernentes necesarios para ejecutar una aplicación, como servicios, configmaps, deployments, secrects.

Ventajas de usar Helm:

- Simplifica los despliegues.
- Configuraciones parametrizables
- Control de versiones
- Ecosistema de charts públicos

2.3 Longhorn

Longhorn es un sistema de almacenamiento en bloques distribuido liviano y fácil de usar para Kubernetes. Es un Software libre y de código abierto. Originalmente desarrollado por Rancher Labs, actualmente de desarrolla por la Cloud Native Computing Foundation.

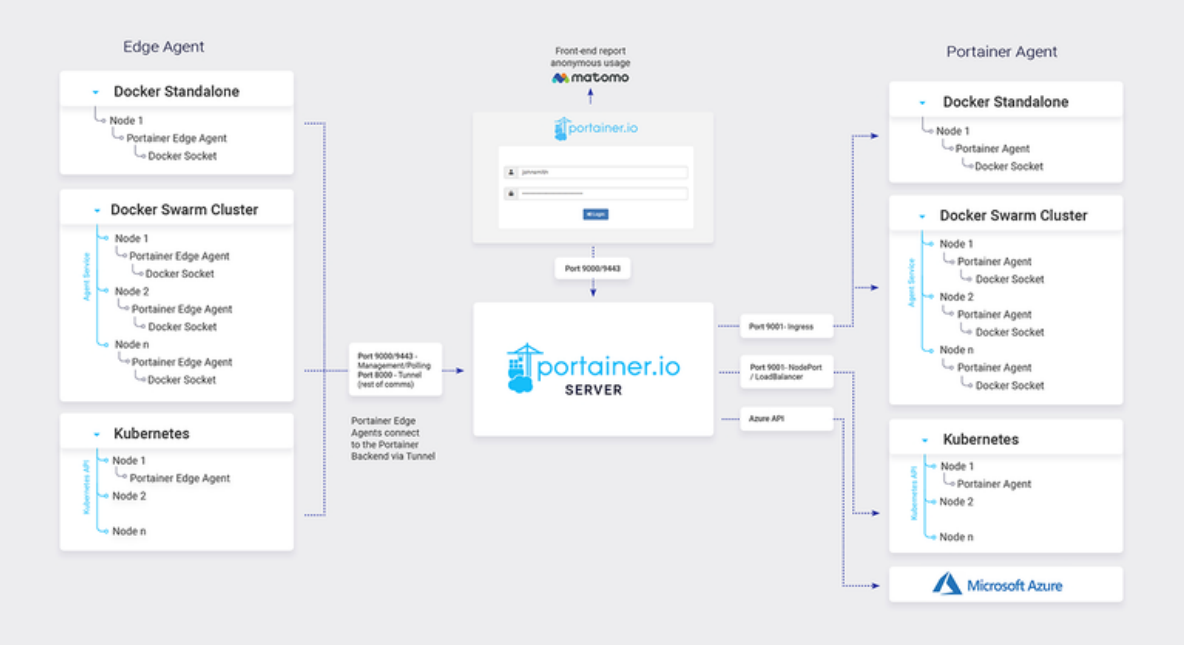
Características de Longhorn:

- Alto rendimiento: Longhorn es capaz de proporcionar un alto rendimiento al gestionar volúmenes distribuidos entre los nodos del clúster.
- Alta disponibilidad: Longhorn replica automáticamente los volúmenes en varios nodos del clúster, lo que significa que, si un nodo falla, los datos son accesibles desde otro nodo que contenga una réplica.
- Escalabilidad: El almacenamiento es fácilmente escalable, permitiendo añadir nodos de almacenamiento adicionales al clúster.
- Resiliencia a fallos: Longhorn ofrece una recuperación ante fallos rápida y automática.
- Interfaz de usuario: Proporciona una interfaz web intuitiva que permite gestionar y monitorear volúmenes de forma sencilla.
- Copias de seguridad y restauración: Permite crear copias de seguridad de los volúmenes.

2.4 Portainer

Portainer es una plataforma de gestión de contenedores de código abierto que ofrece una interfaz de usuario gráfica para facilitar la administración de contenedores, aplicaciones y clústeres de Kubernetes. Originalmente fue diseñado para Docker, pero, en la actualizad ofrece soporte completo para Kubernetes, proporcionando a los usuarios una manera sencilla de gestionar y supervisar recursos de Kubernetes sin necesidad de interactuar con la línea de comandos.

Arquitectura de Portainer



Portainer consta de dos elementos: el servidor de Portainer y el agente de Portainer. Ambos se ejecutan como contenedores en nuestra estructura de Kubernetes. El agente debe implementarse en cada nodo del clúster y configurarse para que informe al contenedor del servidor Portainer.

Un solo servidor aceptará conexiones desde cualquier cantidad de agentes, lo que otorga la capacidad de administrar múltiples clústeres desde una interfaz centralizada. Para ello Portainer requiere persistencia de datos que obtenemos mediante la gestión de volúmenes de Longhorn.

3.Descripción detallada de lo que se ha realizado

En este apartado voy a explicar paso a paso todo el procedimiento que se ha realizado para tener el escenario por completo.

3.1 Instalación y configuración del clúster

Requisitos previos

Necesitaremos instalar estos paquetes en todos los nodos.

Actualizamos todos los paquetes

```
sudo apt update
sudo apt upgrade
```

Activar los módulos necesarios en el sistema:

```
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf overlay
br_netfilter EOF
sudo modprobe overlay sudo modprobe br_netfilter
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

Aplicamos los cambios

```
sudo sysctl –system
```

Instalamos containerd

```
sudo apt install containerd
```

Configuramos ContainerD para que funcione con Kubernetes

```
containerd config default | sudo tee
/etc/containerd/config.toml >/dev/null 2>&1
```

Luego editamos el fichero **/etc/containerd/config.toml** y cambiar el parámetro **SystemdCgroup** = **false** a **true**

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
  BinaryName = ""
  CriuImagePath = ""
  CriuPath = ""
  CriuWorkPath = ""
  IoGid = 0
  IoUid = 0
  NoNewKeyring = false
  NoPivotRoot = false
  Root = ""
  ShimCgroup = ""
  SystemdCgroup = true
```

Reiniciamos y ejecutamos al inicio containerd

```
sudo systemctl restart containerd
sudo systemctl enable container
```

Instalación de Kubeadm, Kubelet y Kubectl

He seguido los pasos de la documentación oficial:

Instalamos los paquetes necesarios:

```
sudo apt-get install -y apt-transport-https ca-certificates curl
gpg
```

Añadimos los repositorios de Kubernetes

```
curl -fsSL
https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key | sudo
gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-
keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.30/deb/ /' |
sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Actualizamos los repositorios

```
sudo apt-get update
```

Instalamos los paquetes

```
sudo apt-get install -y kubectl kubeadm kubelet
```

Marcamos los paquetes como hold para que no se actualicen y podamos tener algún problema con las versiones

```
sudo apt-mark hold kubelet kubeadm kubectl
```

Iniciamos el servicio de Kubelet antes de usar Kubeadm

```
sudo systemctl enable --now kubelet
```

Iniciando el clúster

En el nodo que vamos a usar como controlador del clúster ejecutamos la siguiente instrucción como superusuario

```
kubeadm init --pod-network-cidr=192.168.0.0/16 --apiserver-cert-extra-sans=172.22.201.232
```

Ya que estamos usando como infraestructura tres instancias de OpenStack tendremos que indicar el parámetro --apiserver-cert-extra-sans con la ip flotante del máster, para que el certificado sea válido para esa ip y poder controlar el clúster desde fuera.

Una vez aplicado este comando obtendremos una salida como esta:

```
Your Kubernetes control-plane has initialized successfully!
To start using your cluster, you need to run the following as a regular user:
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
Alternatively, if you are the root user, you can run: export KUBECONFIG=/etc/kubernetes/admin.conf
You should now deploy a pod network to the cluster. Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at: https://kubernetes.io/docs/concepts/cluster-administration/addons/
Then you can join any number of worker nodes by running the following on each as root:
kubeadm join 10.0.0.84:6443 --token b6697i.xsdc63sb9gd0t7hv \
 --discovery-token-ca-cert-hash sha256:0bcac499fd44015c818165ffac842cb6a8984820ae50e3792ef05fae5fff18ce
```

En ella podemos ver:

- Las instrucciones necesarias para configurar Kubectl para gestionar el clúster
- La necesidad de instalar un pod para la gestión de la red.
- Las instrucciones para añadir nodos al clúster

Configuración de Kubectl

Para poder controlar el clúster necesitaremos configurar Kubectl, para ello ejecutamos en el nodo máster:

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

Con estas instrucciones habremos creado un fichero de configuración para acceder al clúster en el directorio ~/.kube

Ahora para controlar el clúster desde fuera copiaremos el contenido del fichero **~/.kube/config** al fichero de configuración de la máquina que queremos usar para acceder.

Instalación del pod para gestionar la red

Tendremos que instalar un pod que nos permita la comunicación por red de los distintos pods que vamos a ejecutar en el clúster. kubeadm solo soporta plugins de red CNI (Container Network Interface), que es un proyecto que consiste en crear especificaciones y librerías para configure las redes que interconectan los contenedores.

De las distintas alternativas vamos a instalar Calico, para ello:

```
kubectl apply -f
https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml
```

Comprobamos que todos los pods del namespace kube-system están funcionando con normalidad

```
kubectl get pods -n kube-system
```

```
josema@Debian12:~$ kubectl get pods -n kube-system
NAME                                       READY   STATUS
RESTARTS        AGE
calico-kube-controllers-7dc5458bc6-69z82   1/1     Running   0
2d22h
calico-node-hcngj                          1/1     Running   0
2d22h
calico-node-vjsmj                          1/1     Running   0
2d22h
calico-node-vqsch                          1/1     Running   1
(2d22h ago)   2d22h
coredns-55cb58b774-m47lk                   1/1     Running   0
2d22h
coredns-55cb58b774-rx5bw                   1/1     Running   0
2d22h
etcd-master                                1/1     Running   0
2d22h
kube-apiserver-master                      1/1     Running   0
2d22h
```

```
kube-controller-manager-master              1/1     Running   0
2d22h
kube-proxy-ktjlk                            1/1     Running   0
2d22h
kube-proxy-qk5r4                            1/1     Running   0
2d22h
kube-proxy-xb2wk                            1/1     Running   0
2d22h
kube-scheduler-master                       1/1     Running   0
2d22h
metrics-server-7995dd4965-fcd9p             1/1     Running   0
2d22h
```

Uniendo nodos al clúster

En cada nodo que va a formar parte del clúster tenemos que ejecutar como superusuario el comando que nos ofreció la salida del comando Kubeadm al iniciar clúster en el master, en mi caso

```
kubeadm join 10.0.0.84:6443 --token b6697i.xsdc63sb9gd0t7hv \
--discovery-token-ca-cert-hash
sha256:0bcac499fd44015c818165ffac842cb6a8984820ae50e3792ef05fae5
fff18ce
```

Comprobamos que se han añadido los nodos

```
josema@Debian12:~$ kubectl get nodes
NAME     STATUS   ROLES           AGE     VERSION
master   Ready    control-plane   2d22h   v1.30.8
nodo1    Ready                    2d22h   v1.30.8
nodo2    Ready                    2d22h   v1.30.8
```

3.2 Instalación de Helm

Necesitaremos tener instalado Helm para realizar las instalaciones tanto de Portainer como de Longhorn

Para instalar Helm, añadimos los repositorios de Helm

```
curl https://baltocdn.com/helm/signing.asc | gpg --dearmor | sudo tee
/usr/share/keyrings/helm.gpg > /dev/null
sudo apt-get install apt-transport-https --yes
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/helm.gpg]
https://baltocdn.com/helm/stable/debian/ all main" | sudo tee
/etc/apt/sources.list.d/helm-stable-debian.list
```

Actualizamos los repositorios e instalamos Helm

```
sudo apt-get update
sudo apt-get install Helm
```

Con esto ya podremos instalar nuestras aplicaciones usando Helm

3.3 Instalación de Longhorn

Para instalar Longhorn debemos de satisfacer unos requisitos previos instalados en todos los nodos del clúster, para ello consultando la documentación oficial, los requisitos necesarios son:

- Open-iscsi instalado y el demonio iscsid corriendo en todos los nodos
- Tener un cliente NFS instalado en todos los nodos
- Tener instalado los siguientes paquetes: curl, findmnt, grep, awk, blkid, lsblk.

Ahora vamos a pasar a instalar los requisitos previos en todos los nodos:

Instalación de los paquetes necesarios

Instalación de los paquetes necesarios

```
sudo apt update
sudo apt install -y curl findmnt grep awk blkid lsblk
```

Instalamos el paquete open-iscsi

```
sudo apt install open-iscsi
```

Arrancamos los servicios iscsid

```
sudo systemctl enable iscsid.socket
sudo systemctl enable iscsid.service
sudo systemctl start iscsid.socket
sudo systemctl start iscsid.service
```

Activamos el módulo iscsi_tcp

```
sudo modprobe iscsi_tcp
```

Para permitir que este módulo se active solo al arranque de las máquinas tendremos que añadir una entrada en el fichero /etc/modules con el nombre del módulo, en este caso iscsi_tcp

Instalamos el paquete nfs-common

```
sudo apt install nfs-common
```

En la documentación oficial de Longhorn disponemos de un script para comprobar si nuestro clúster cumple con todos los requisitos

https://github.com/longhorn/longhorn/blob/v1.7.2/scripts/environment_check.sh

Compruebo con el script

```
josema@Debian12:~/Cluster/Longhorn$ ./requirements.sh
[INFO]  Required dependencies 'kubectl jq mktemp sort printf' are
installed.
[INFO]  All nodes have unique hostnames.
[INFO]  Waiting for longhorn-environment-check pods to become ready
(0/2)...
[INFO]  All longhorn-environment-check pods are ready (2/2).
[INFO]  MountPropagation is enabled
[INFO]  Checking kernel release...
[INFO]  Checking iscsid...
[INFO]  Checking multipathd...
[INFO]  Checking packages...
[INFO]  Checking nfs client...
[INFO]  Cleaning up longhorn-environment-check pods...
[INFO]  Cleanup completed.
```

Para instalar Longhorn, añadimos el repositorio Helm de Longhorn

```
helm repo add longhorn https://charts.longhorn.io
```

Actualizamos los repositorios de Helm

```
helm repo update
```

Instalamos Longhorn

```
helm install longhorn longhorn/longhorn --namespace longhorn-
system --create-namespace --version 1.7.2
```

Este comando nos creará un nuevo namespace llamado Longhorn-system donde instalará Longhorn

Comprobamos que Longhorn está corriendo correctamente

```
kubectl -n longhorn-system get pod
```

```
josema@Debian12:~/Cluster/Longhorn$ kubectl -n longhorn-system get pod
NAME                                                READY   STATUS    RESTARTS   AGE
csi-attacher-698944d5b-2fz56                        1/1     Running   0          2d1h
csi-attacher-698944d5b-2g4qg                        1/1     Running   0          2d1h
csi-attacher-698944d5b-6cg9d                        1/1     Running   0          2d1h
csi-provisioner-b98c99578-27vfp                     1/1     Running   0          2d1h
csi-provisioner-b98c99578-pfqgp                     1/1     Running   0          2d1h
csi-provisioner-b98c99578-q225d                     1/1     Running   0          2d1h
csi-resizer-7474b7b598-9xg9w                        1/1     Running   0          2d1h
csi-resizer-7474b7b598-gwh48                        1/1     Running   0          2d1h
csi-resizer-7474b7b598-vwkrd                        1/1     Running   0          2d1h
csi-snapshotter-774467fdc7-jn45b                    1/1     Running   0          2d1h
csi-snapshotter-774467fdc7-jr4zb                    1/1     Running   0          2d1h
csi-snapshotter-774467fdc7-trt9c                    1/1     Running   0          2d1h
engine-image-ei-51cc7b9c-tx6nq                      1/1     Running   0          2d1h
engine-image-ei-51cc7b9c-vh2r4                      1/1     Running   0          2d1h
instance-manager-35fcd1a915d0e1f4276548b11e3a1df3   1/1     Running   0          2d1h
instance-manager-4cfc91fff903489289419a25a1128943   1/1     Running   0          2d1h
longhorn-csi-plugin-9n6v4                           3/3     Running   0          2d1h
longhorn-csi-plugin-jwpgf                           3/3     Running   0          2d1h
longhorn-driver-deployer-7d77d779dd-vkcxj           1/1     Running   0          2d1h
longhorn-manager-dcghk                              2/2     Running   0          2d1h
longhorn-manager-xzrqz                              2/2     Running   0          2d1h
longhorn-ui-7c784d8587-n5tv6                        1/1     Running   0          2d1h
longhorn-ui-7c784d8587-wfd5f                        1/1     Running   0          2d1h
josema@Debian12:~/Cluster/Longhorn$
```

3.4 Instalación Portainer

Al igual que con Longhorn instalaré portainer usando Helm. En mi caso voy a usar la versión Busisness de Portainer, ya que he obtenido una prueba gratuita para poder usar con tres nodos. Para poder instalar Portainer necesitaremos persistencia de datos que lograremos gracias a Longhorn. Para comprobar los storageclass que disponemos en nuestro clúster

```
kubectl get sc
```

```
josema@Debian12:~/Cluster/Longhorn$ kubectl get sc
NAME                 PROVISIONER          RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
longhorn (default)   driver.longhorn.io   Delete          Immediate           true                   2d1h
longhorn-static      driver.longhorn.io   Delete          Immediate           true                   2d1h
josema@Debian12:~/Cluster/Longhorn$
```

Tendremos que asegurarnos de que Longhorn es el storageclass por defecto, en caso de que no lo sea, podemos hacerlo usando

```
kubectl patch storageclass longhorn -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-
class":"true"}}}'
```

Para instalar Portainer usando Helm, añadimos los repositorios de Portainer para Helm

```
helm repo add portainer  https://portainer.github.io/k8s/
helm repo update
```

Instalamos Portainer

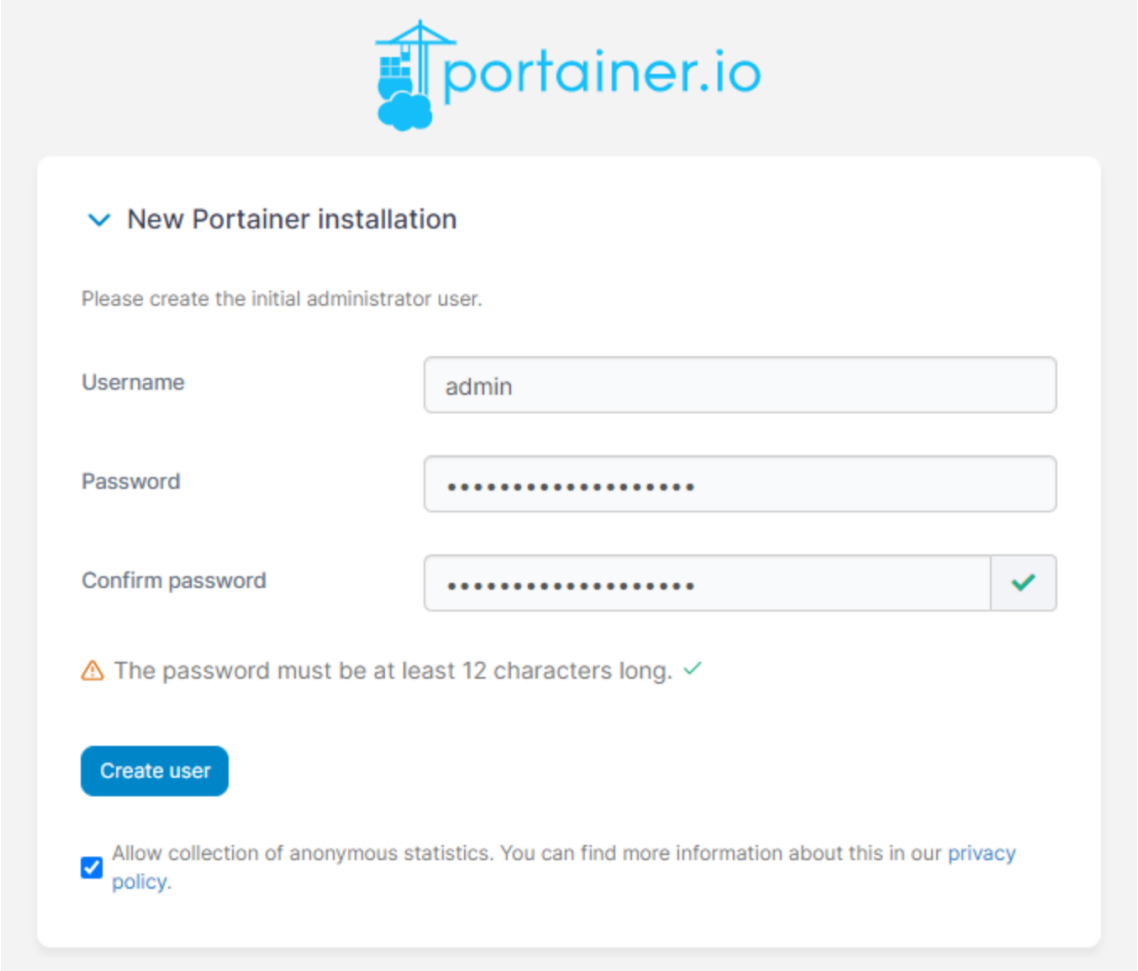
```
helm upgrade --install --create-namespace -n portainer portainer
portainer/portainer \\
    --set enterpriseEdition.enabled=true \\
    --set enterpriseEdition.image.tag=2.21.4 \\
    --set tls.force=true
```

Al igual que antes Helm creará un nuevo namespace donde instalará portainer.

Con este comando estamos exponiendo la web de portainer al exterior mediante un servicio NodePort, por lo tanto, podemos acceder mediante la ip pública del nodo master y usando el puerto 30779

https://172.22.201.232:30779

Una vez dentro nos pedirá crear una nueva contraseña para el usuario Admin



4.Uso de Portainer

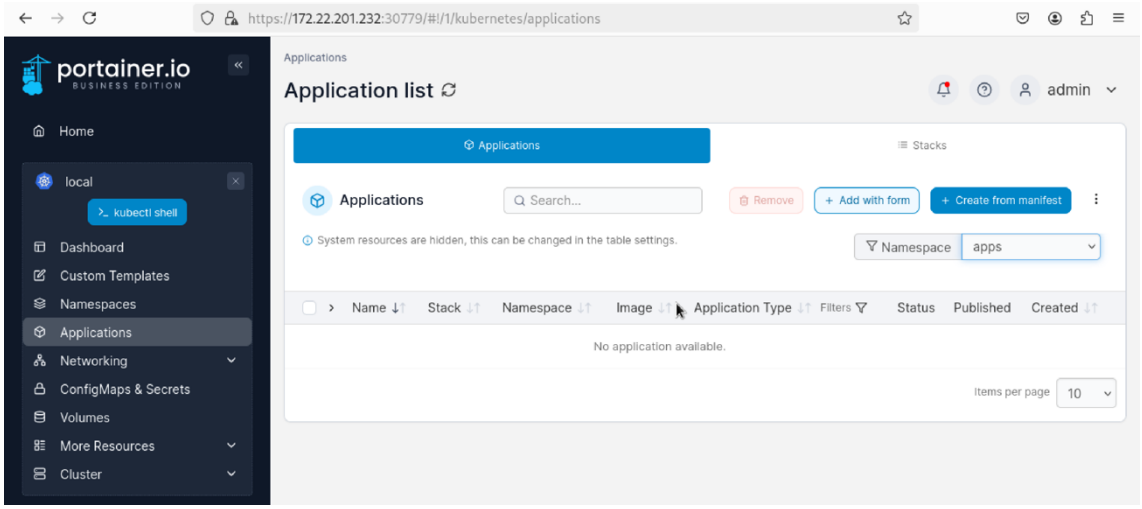
En este apartado explicaré brevemente algunos casos prácticos del uso de portainer para desplegar aplicaciones.

4.1Creación de aplicaciones con Portainer.

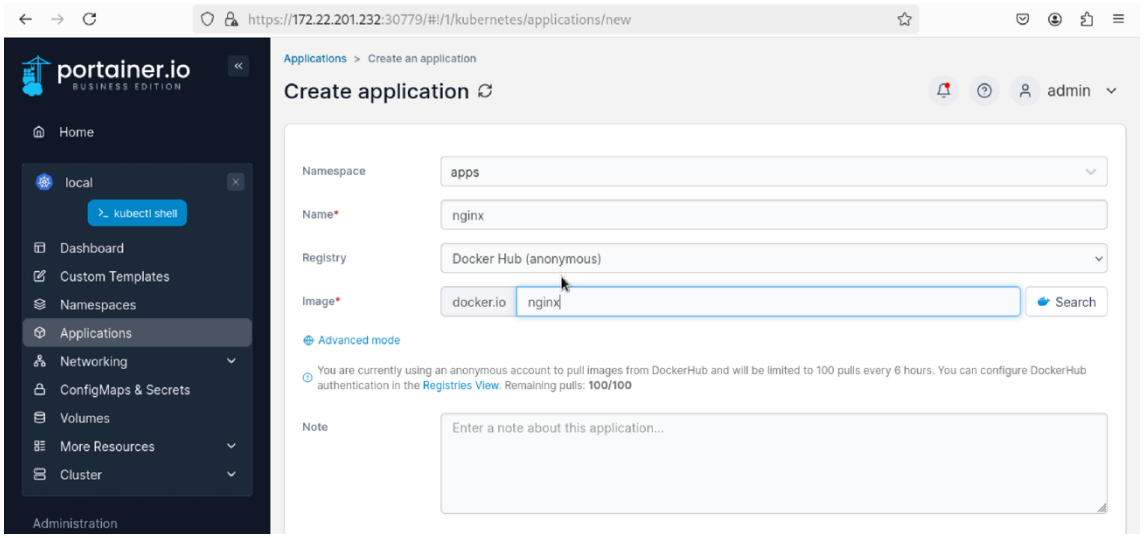
En este apartado mostraremos rápidamente el despliegue de una aplicación sencilla mediante Portainer.

En este caso voy a desplegar un despliegue de Nginx y lo mostraremos mediante un servicio NodePort

Para desplegar las aplicaciones tendremos distintos métodos, mediante una imagen Docker (add with form) o mediante charts de Helm o repositorios (add with manifest

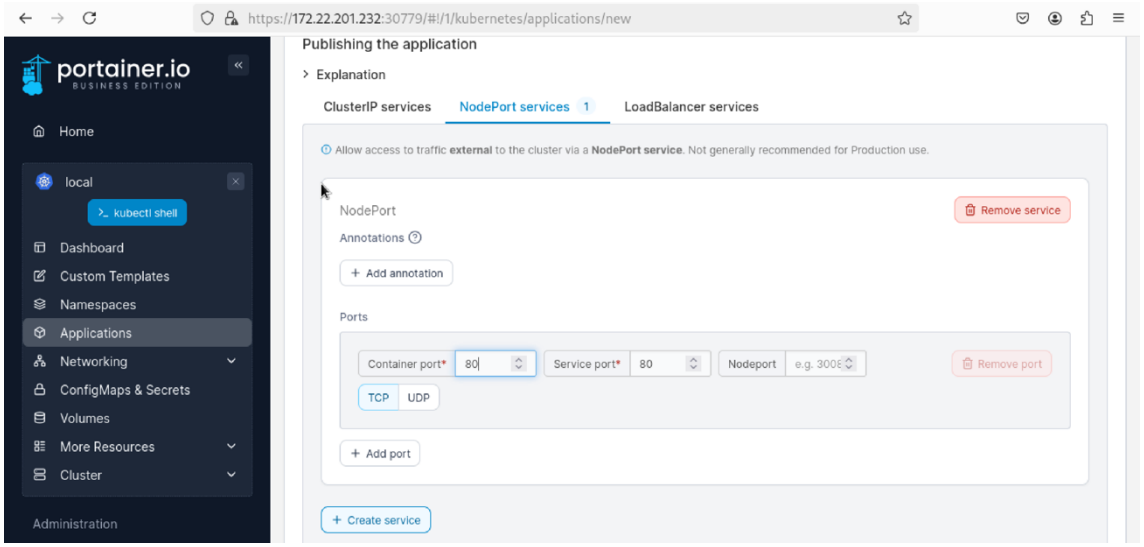


Si seleccionamos add with form, tendremos que especificar el nombre de la imagen Docker que vamos a desplegar, un namespace y el servicio por el cuál ofrecemos la aplicación.



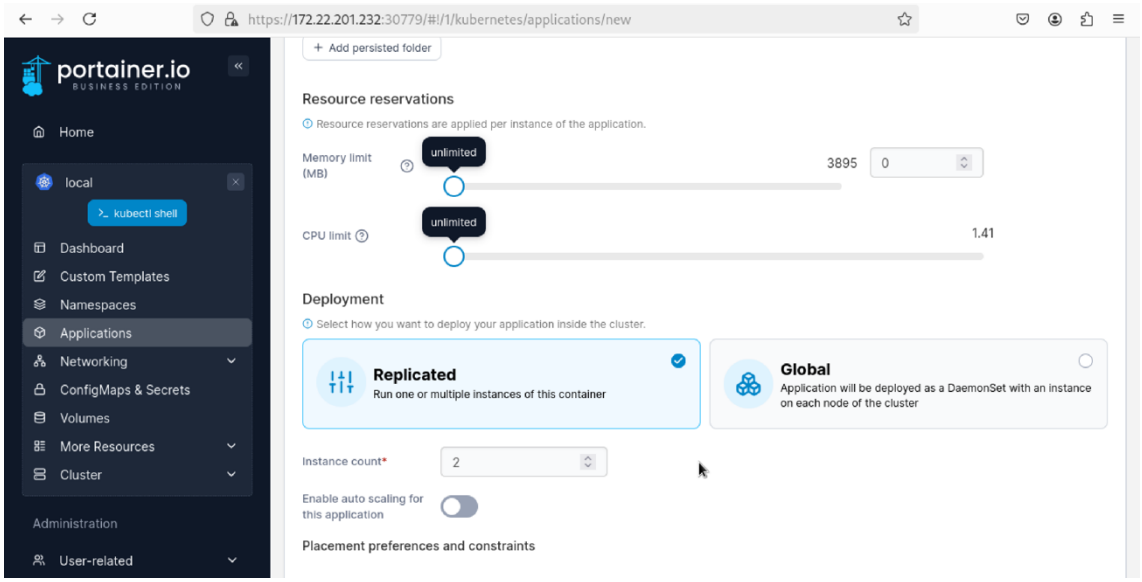
En mi caso he seleccionado la imagen nginx en el namespace apps.

Si bajamos más abajo encontraremos la sección services



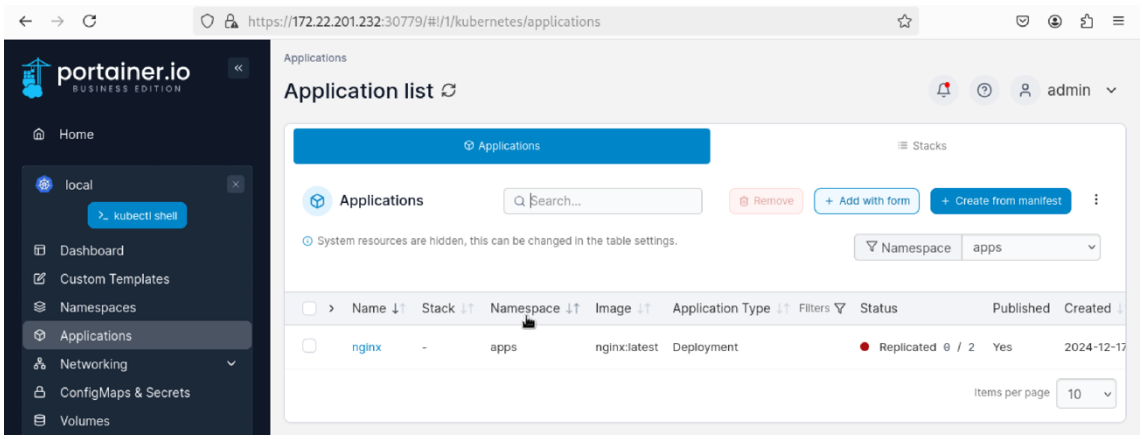
Aquí tendremos que especificar el puerto usado por el contenedor, y el puerto usado por NodePort para exponer la aplicación, si dejamos el campo NodePort en blanco automáticamente Portainer asignará un puerto aleatorio.

Si continuamos podemos observar más opciones como la replicación de los pods, o la ubicación de estos, además de la reserva de los recursos usados por el pod.

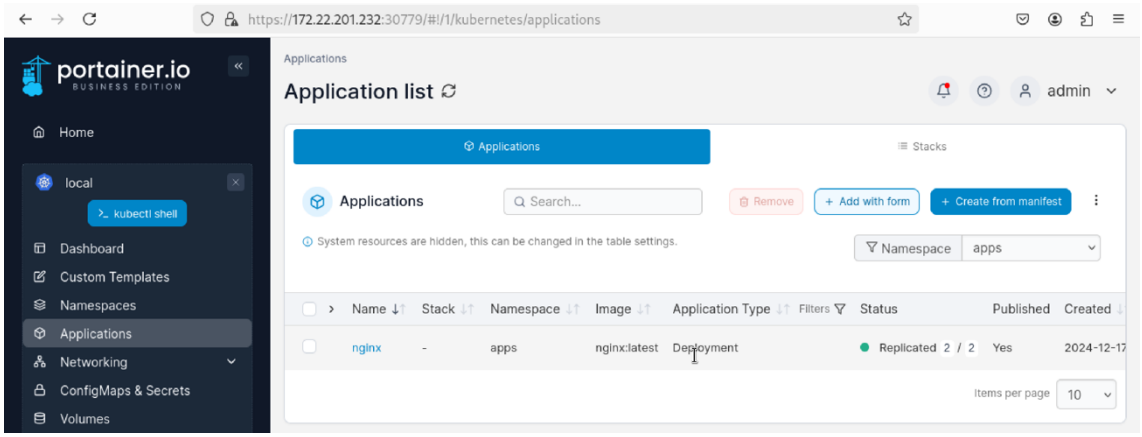


He seleccionado 2 replicas para este pod

Para desplegar la aplicación pulsaremos en el botón Deploy Application

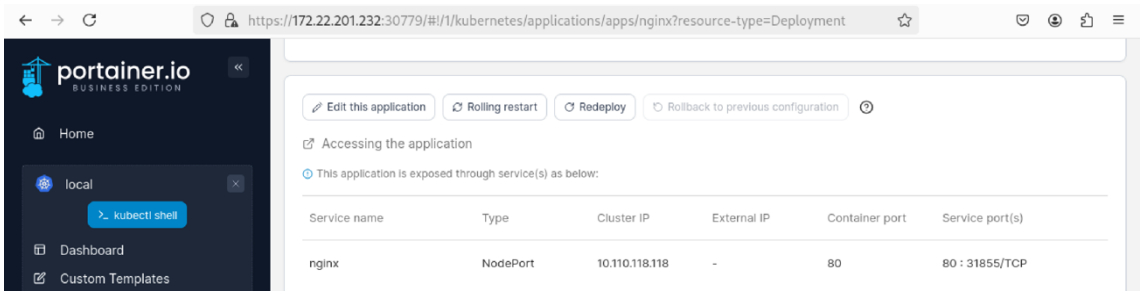


Esperamos que se realice el deploy

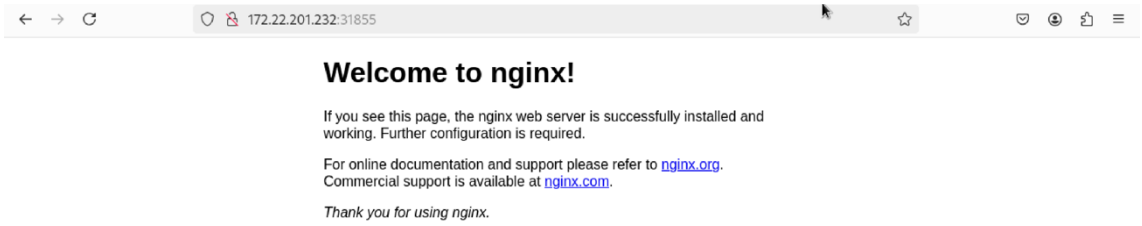


Cuando el deploy haya finalizado podremos ver nuestra aplicación desplegada en nuestro namespace apps y como vemos disponemos de dos replicas.

Para acceder a la aplicación necesitaremos conocer el puerto del servicio NodePort donde está expuesto, para ello entramos a los detalles de la aplicación

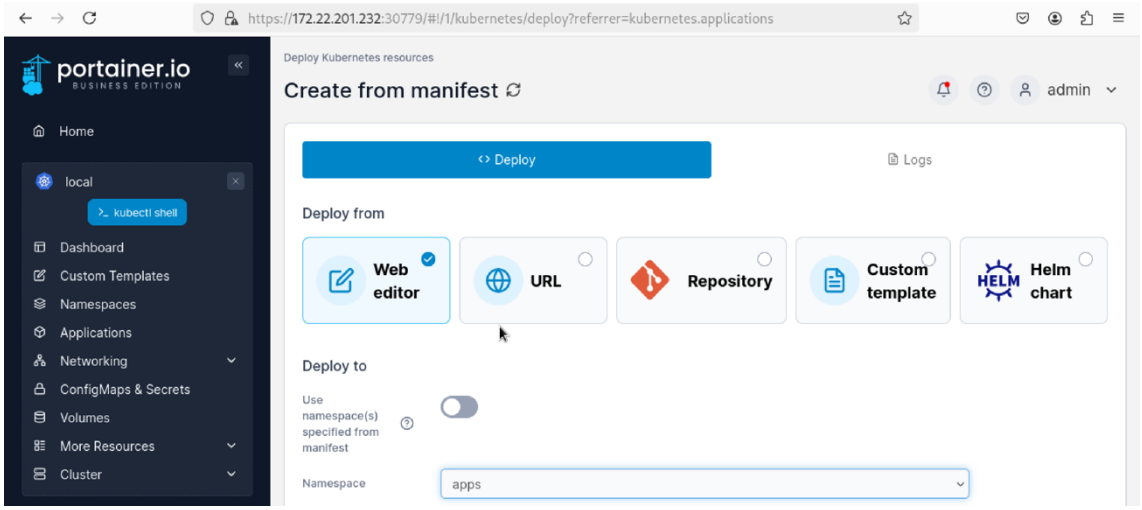


Como vemos nuestra aplicación esta expuesta en el puerto 31855 por lo tanto, accederemos mediante la ip pública del nodo master y el puerto



Como vemos hemos accedido a nuestra aplicación de Nginx

Si seleccionamos5 la opción add with form veremos las distintas formas que tenemos de añadir nuestra aplicación



Observamos que podemos añadir aplicaciones mediante:

- Editor web: escribiendo directamente en la web nuestra definición de app en Yaml.
- URL: especificando la URL de una imagen Docker
- Repositorio: especificando la url de nuestro repositorio GitHub donde tenemos nuestros ficheros Yaml

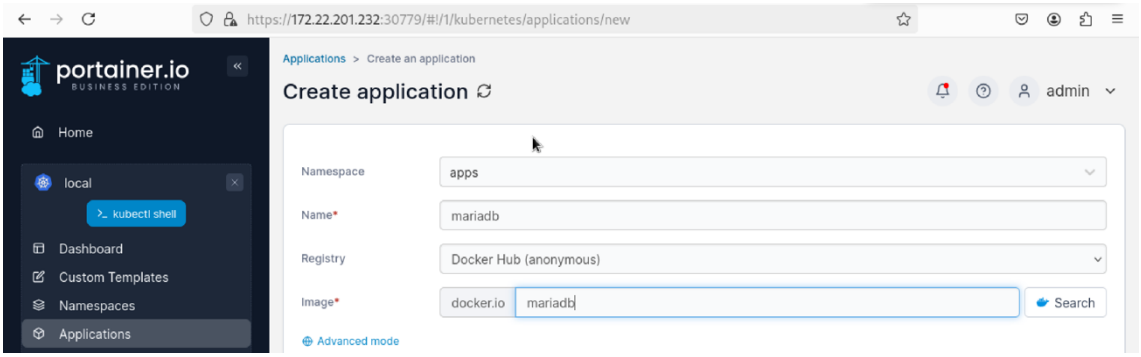
- Ficheros personalizados: Subiendo nuestros ficheros Yaml
- Helm: usando charts provisionados por Helm

4.2Despliegue Wordpress con Mariadb con volumen persistente

En este apartado voy a realizar un despliegue completo de una aplicación wordpres conectado a una base de datos MariaDB usando volúmenes persistentes

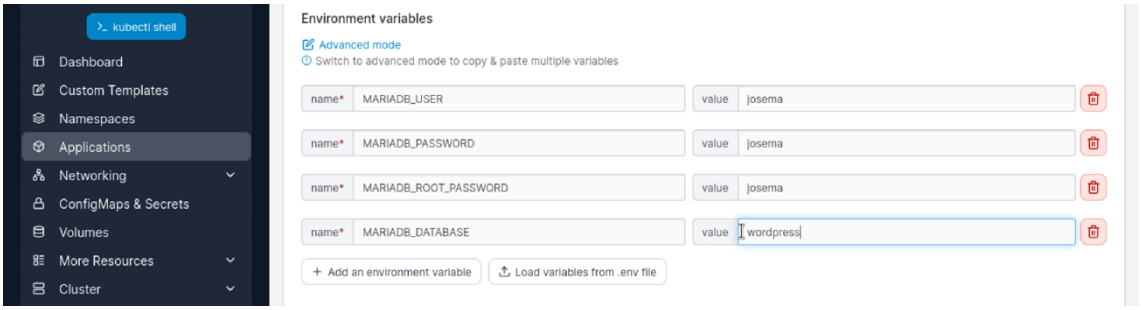
Para comenzar empezaremos desplegando la base datos mariaDB, para ello añadiremos nuestra aplicación mediante una imagen Docker especificando las variables necesarias para la creación de la base datos.

Empezaremos definiendo el namespace, el nombre y la imagen Docker



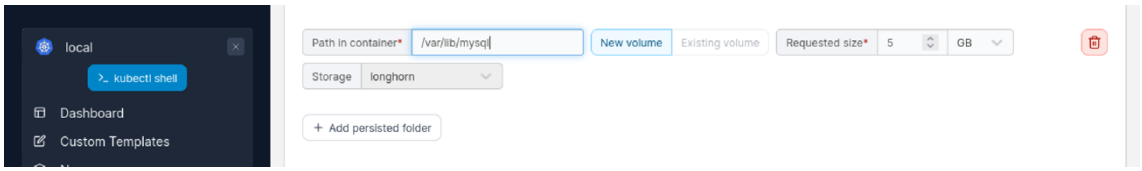
Nos desplazamos hasta la parte de Variables de entorno para añadir las variables de entorno

- MARIADB_USER
- MARIADB_PASSWORD
- MARIADB_ROOT_PASSWORD
- MARIADB_DATABASE

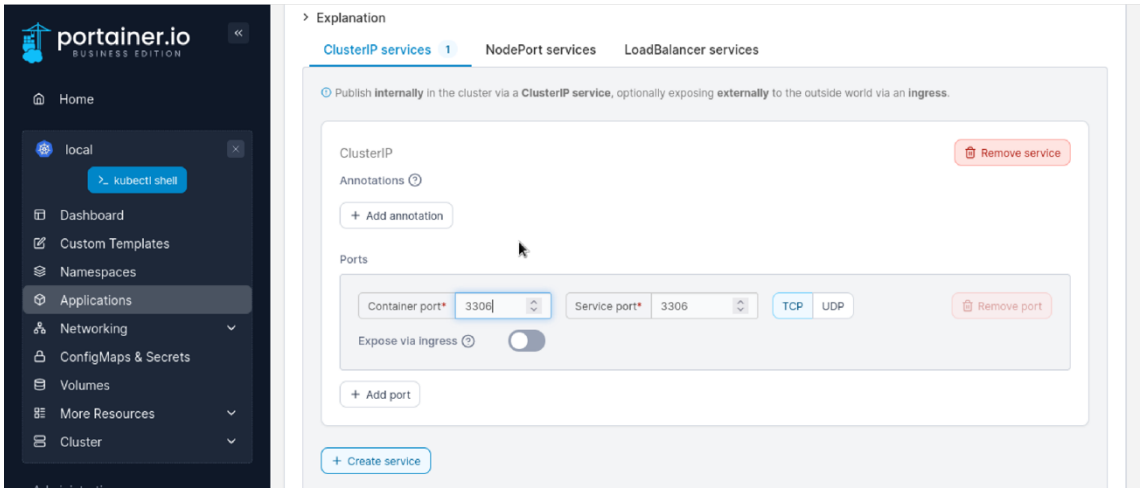


Para crear un nuevo volumen persistente nos desplazamos hasta carpetas persistentes y tendremos que especificar el punto de montaje del volumen, si deseamos crear un nuevo volumen y el tamaño. Ya que uso Longhorn creare el

volumen en Longhorn

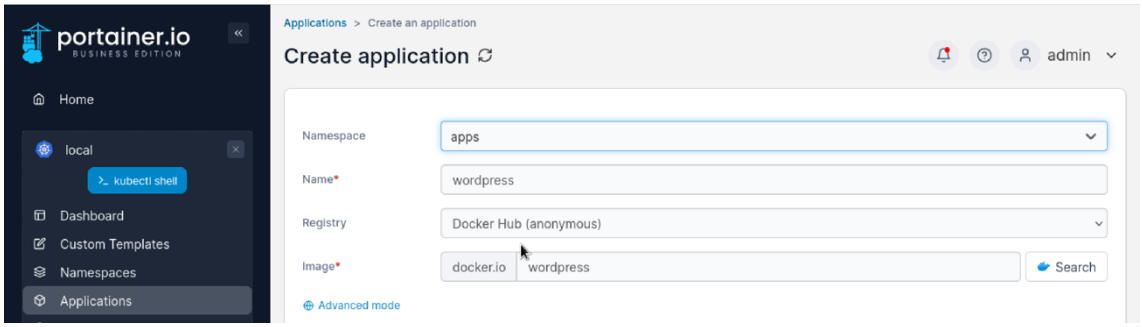


Para que la base de datos sea accesible por WordPress necesitamos exponer la aplicación mediante un servicio ClusterIp, especificando el puerto 3306 que es el usado por defecto por MariaDB



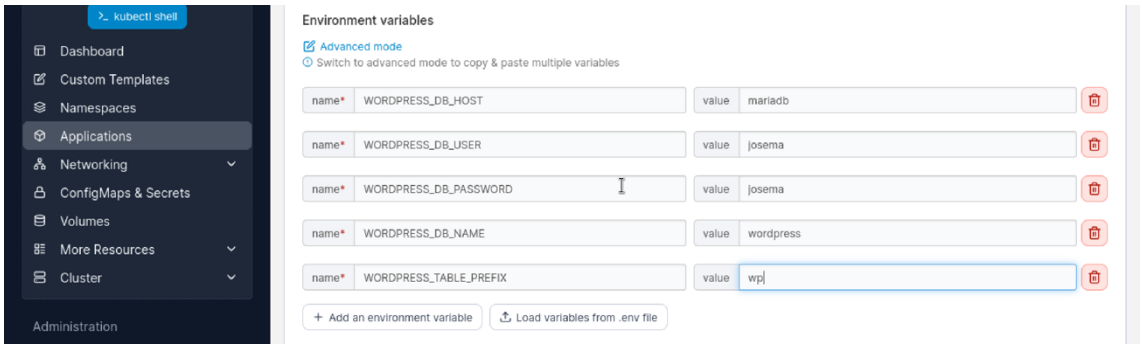
Con esto ya tendríamos la base de datos accesible desde otro pod del Clúster, Ahora desplegaremos Wordpress.

Tendremos que desplegar nuestra app WordPress en el mismo namespace en el que hemos desplegado la base de datos

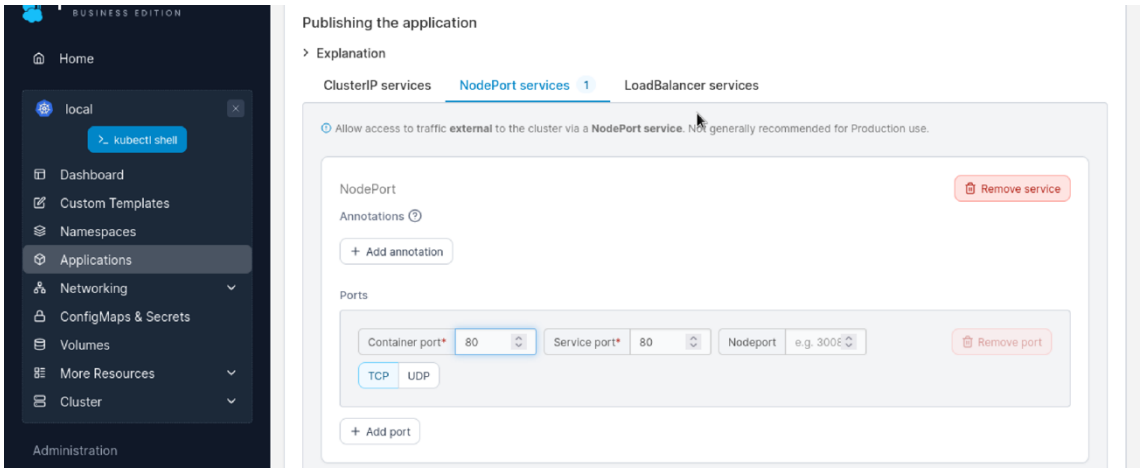


Ahora tendremos que definir las variables de entorno para que Wordpress pueda acceder a la base de datos

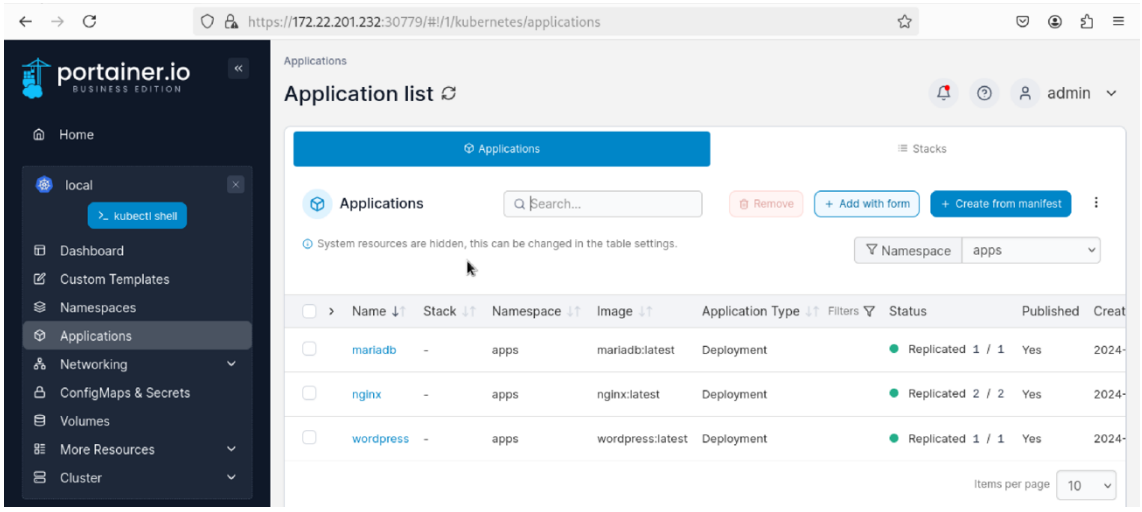
- WORDPRESS_DB_HOST
- WORDPRESS_DB_USER
- WORDPRESS_DB_NAME
- WORDPRESS_TABLE_PREFIX



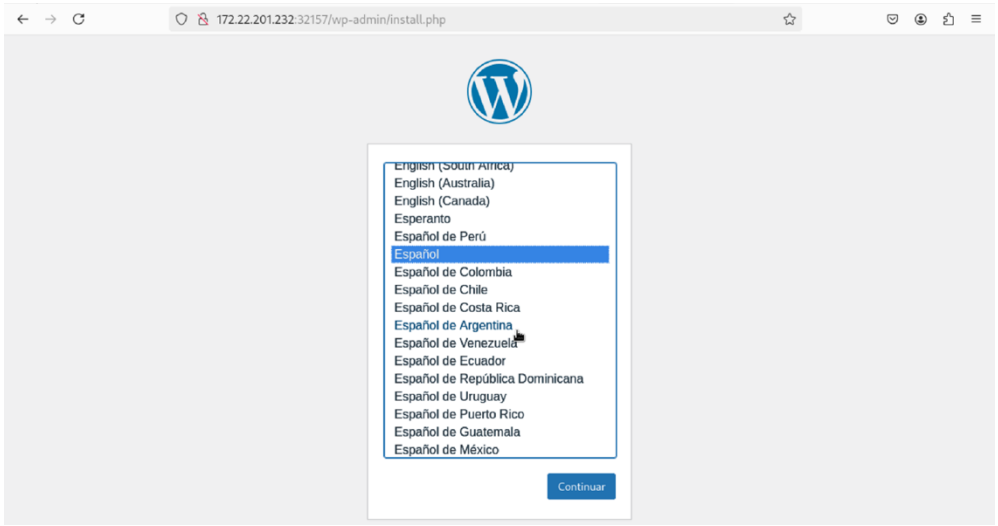
Ahora tendemos que exponer nuestra aplicación mediante un servicio NodePort para poder acceder a ella desde fuera

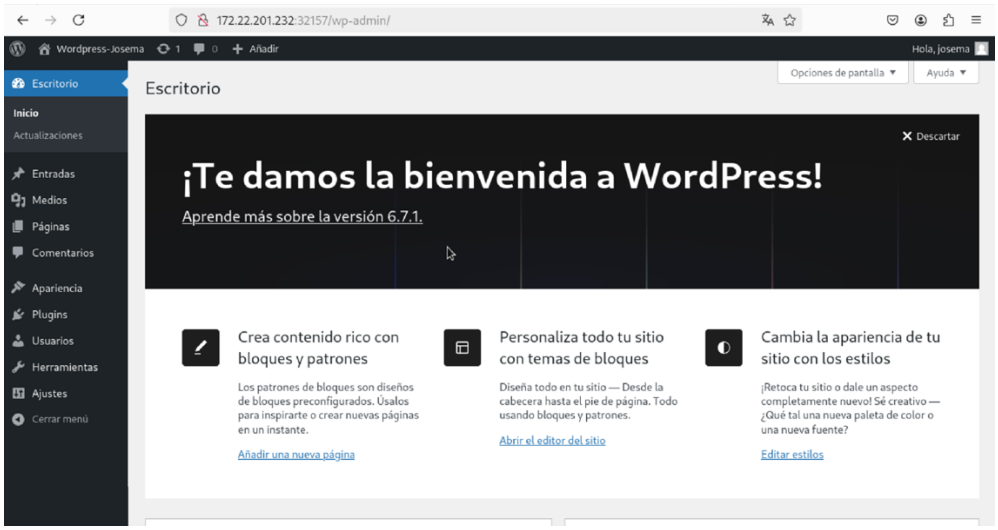


Desplegamos nuestra aplicación y esperamos a que se creen los pods



Accedemos a nuestra app Wordpress y realizar la instalación de Wordpress





4.3.Auto escalado en Portainer

Para poder realizar el auto escalado en un clúster de Kubernetes deberemos tener instalado la función de métricas. En mi caso he instalado metric-server.

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

Con esto desplegaremos el servicio de métricas en nuestro clúster, pero obtendremos un error de certificados, para solucionar este error, tendremos que editar el despliegue y añadir los siguientes argumentos:

```
args:
  - --kubelet-insecure-tls
  - --kubelet-preferred-address-types=InternalIP
```

```
kubectl edit pod -n kube-system metrics-server-7995dd4965-fcd9p
```

```
spec:
  containers:
  - args:
    - --cert-dir=/tmp
    - --secure-port=10250
    - --kubelet-preferred-address-types=InternalIP
    - --kubelet-use-node-status-port
    - --metric-resolution=15s
    - --kubelet-insecure-tls
    image: registry.k8s.io/metrics-server/metrics-server:v0.7.2
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 3
      httpGet:
        path: /livez
        port: https
        scheme: HTTPS
      periodSeconds: 10
      successThreshold: 1
      timeoutSeconds: 1
    name: metrics-server
"/tmp/kubectl-edit-563259678.yaml" 171L, 4767B
```

Comprobamos que se está ejecutando correctamente el servicio de métricas ejecutando el siguiente comando

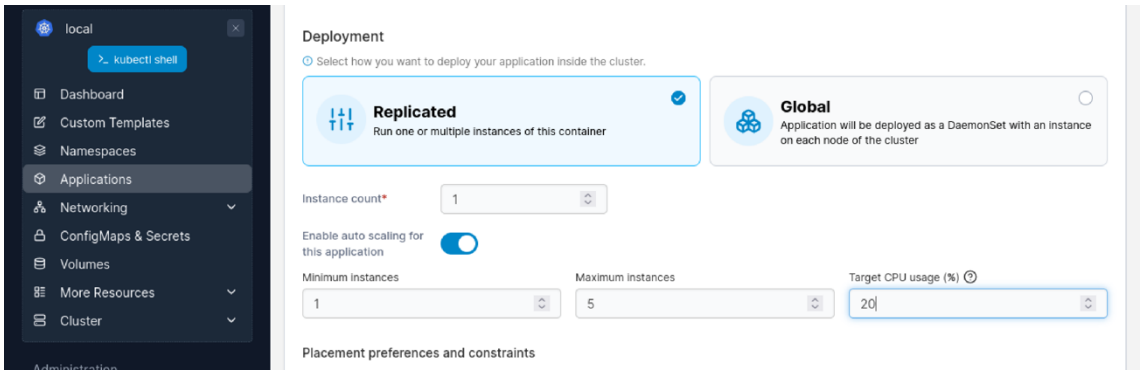
```
josema@Debian12:~/Cluster/Longhorn$ kubectl top nodes
NAME     CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
master   251m         12%    1949Mi          51%
nodo1    188m         9%     2038Mi          53%
nodo2    205m         10%    2552Mi          66%
```

Con este servicio ya podemos configurar el auto escalado en Portainer, como sabemos Portainer es una herramienta muy sencilla de usar para todo tipo de usuarios.

Para usar el auto escalado simplemente nos bastaría con seleccionar la opción de auto escalado y seleccionar el número mínimo y máximo de replicas, y el tanto por ciento de uso CPU al que debe activarse el auto escalado.

En mi caso he configurado para que se desplieguen como máximo 5 replicas y que se repliquen cuando se use un 20% de CPU.

Para ello voy a realizar un nuevo despliegue de una nueva aplicación nginx llamada nginx-escalable para probar



Desplegamos la aplicación .

Para realizar las pruebas de carga a un servidor web he usado la utilidad apache2-utils para realizar peticiones simultaneas al servidor.

Para realizar las peticiones:

```
ab -n 1000000 -c 500 http://172.22.201.232:30743/
```

Ahora veremos en directo la salida del comando `kubectl get pods` para ver cómo se crean las réplicas.

```
NAME                                 READY   STATUS    RESTARTS   AGE
nginx-escalable-b7fcdb5d5-5lwsf      1/1     Running   0          84s
nginx-escalable-b7fcdb5d5-8wftr      1/1     Running   0          37s
nginx-escalable-b7fcdb5d5-9qcmz      1/1     Running   0          52s
nginx-escalable-b7fcdb5d5-mjclx      1/1     Running   0          52s
nginx-escalable-b7fcdb5d5-pnr4x      1/1     Running   0          52s
traefik-deployment-667459fcf8-cdkb7  1/1     Running   0          27h
whoami-57b48994d9-x5sf7              1/1     Running   0          27h
```

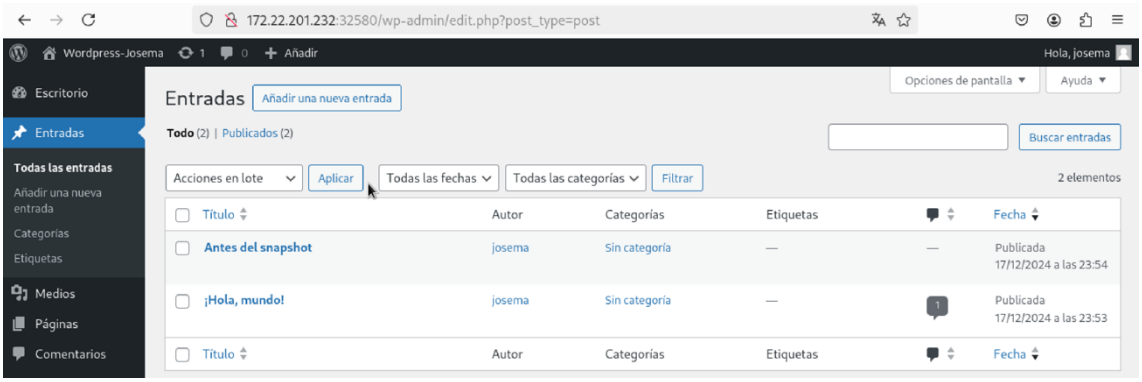
```
  66%    180
  75%    193
  80%    204
  90%    494
  95%   1175
  98%   1206
  99%   1240
 100%  15395 (longest request)
josema@Debian12:~/Cluster/traefik$ ab -n 1000000 -c 500 http://172.22.201.232:30743/
This is ApacheBench, Version 2.3 <$Revision: 1913912 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 172.22.201.232 (be patient)
Completed 100000 requests
```

Como vemos se han replicado los pods hasta llegar al máximo de replicas. Como vemos tenemos 4 pods creados más tarde que el primer pod.

4.4.Backups con Longhorn

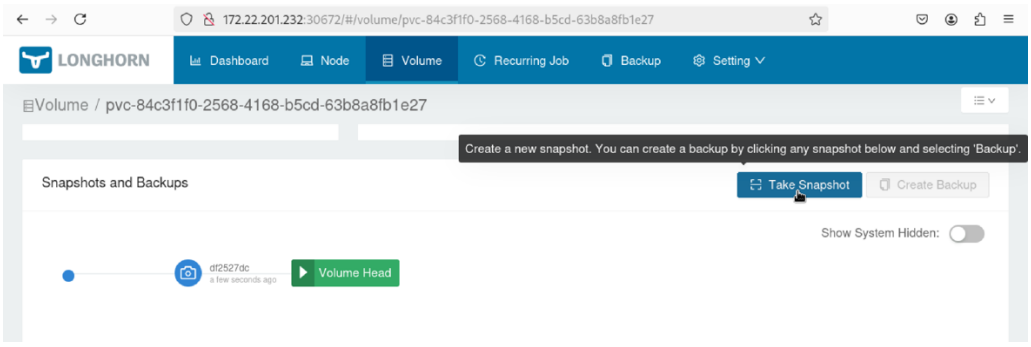
En este apartado comprobaremos la facilidad que nos ofrece Longhorn para restaurar backups de los volúmenes.

Para ello usaremos el despliegue de Wordpress con la base de datos, en el cual vamos a crear dos entradas. Cuando creemos la primera entrada realizaremos el snapshot y luego crearemos la segunda entrada para comprobar que al restaurar la snapshot no existe la segunda entrada.

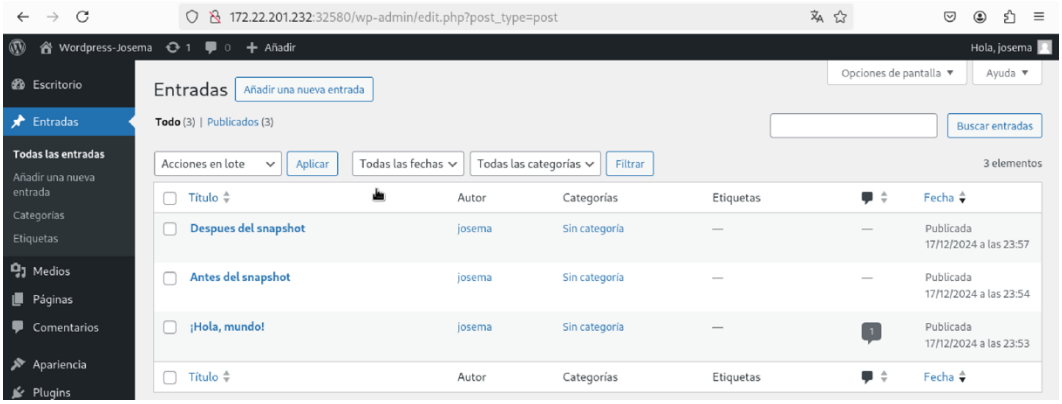


Tenemos la primera entrada creada, me dispongo a crear el snapshot del volumen de mariadb

Desde la interfaz web de Longhorn navegamos hasta el volumen asociado a mariadb y seleccionamos la opción crear snapshot



Creamos la segunda entrada en el blog

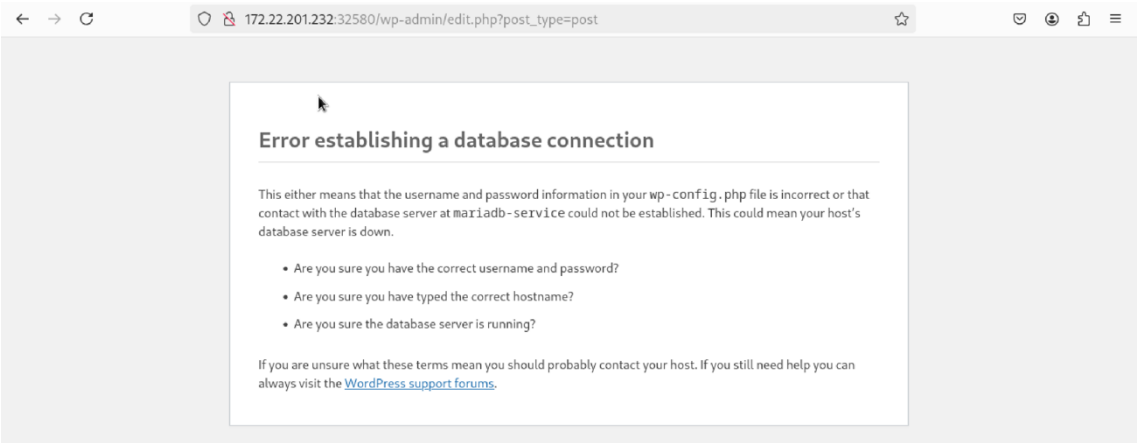


Con la segunda entrada creada nos disponemos a eliminar el deployment de mariadb

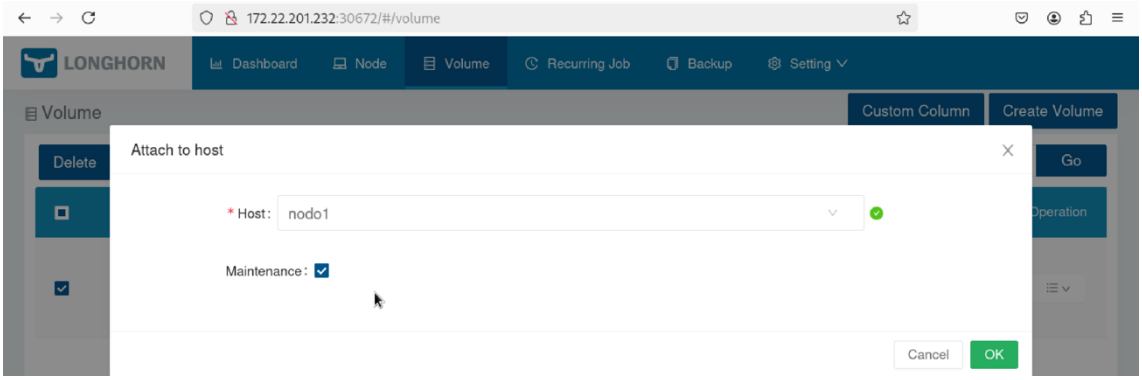
```
josema@Debian12:~/Cluster/wordpress-kubernetes$ kubectl delete deployments.apps mariadb-deployment
```

```
deployment.apps "mariadb-deployment" deleted
```

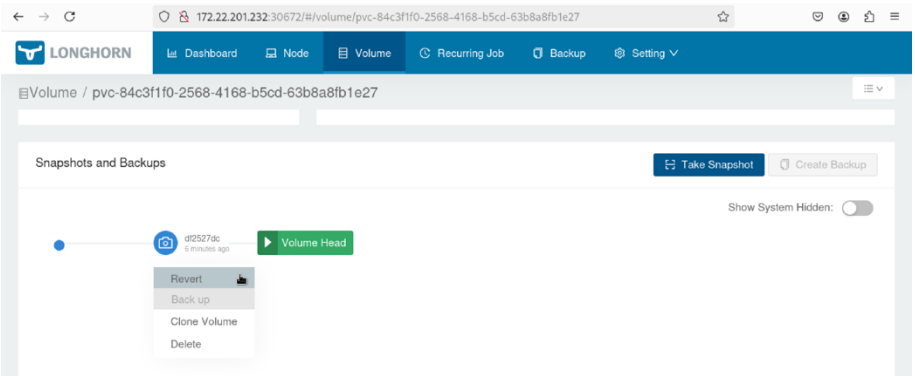
Si intentamos entrar a WordPress, nos mostrara que no tiene conexión con la base de datos.



Para restaurar la snapshot, nos dirigimos al volumen asociado a mariadb, asociamos dicho volumen en modo mantenimiento a cualquier nodo del clúster

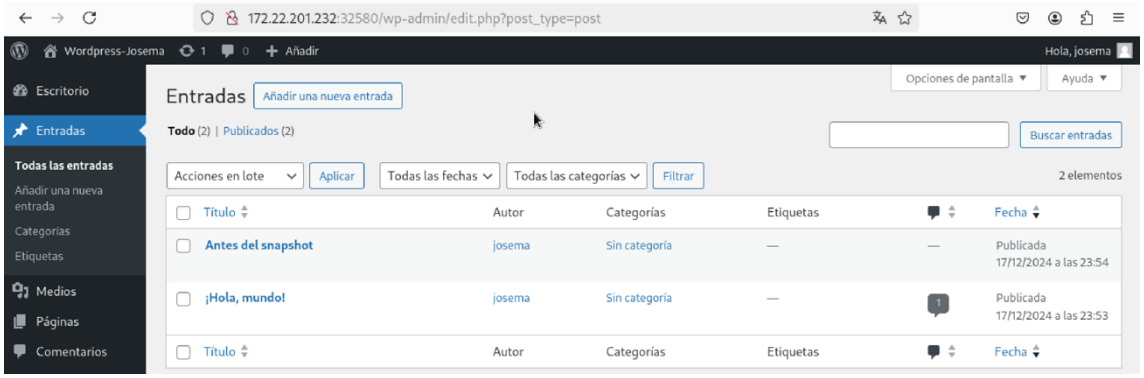


Nos dirigimos al snapshot y seleccionamos la opción recuperar.



Con el volumen recuperado, desasociamos el volumen y volveremos a realizar el deployment de mariadb.

Volvemos a acceder a WordPress y comprobamos que solo debería de existir una entrada.



Como vemos se ha restaurado el snapshot correctamente

5.Dificultades que se han encontrado

En este proyecto he encontrado las siguientes dificultades:

- Dificultad a la hora de instalar el servicio de métricas en el clúster, ya que tenía problemas con los certificados.
- Dificultad a la hora de realizar el auto escalado debido a que no se activa el servicio de métricas.
- Dificultad de implementar un controlador ingress para exponer los servicios.

6.Bibliografía

https://kubernetes.io/es/docs/reference/setup-tools/kubeadm/

https://longhorn.io/docs/1.7.2/

https://docs.portainer.io