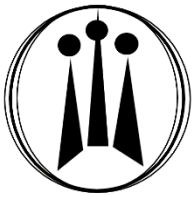
ELFHEIN



KubeMagic

Un producto original, adaptado a tus necesidades y sin límites.

Autor: Javier Ramírez Moral

Fecha de Publicación: 12/06/2023

Documentación Técnica & Manual de Uso Segunda Parte

Contenido

Resumen de este documento.

En este documento se pretende mostrar todo el apartado técnico con sus correspondientes explicaciones para complementarlo y poder usarlo posteriormente a modo de manual de uso de la segunda parte del proyecto.

Segunda Parte: Escenario de una App WEB y una Base de Datos.

Conforme vaya avanzando e implementando herramientas de configuración, iré justificando su uso para este proyecto. Vamos entonces en esta parte a montar el primer escenario que se nos encargó de la siguiente manera.

Lo que pretendemos conseguir es lo siguiente en esta parte. Voy a desplegar dos aplicaciones que serán la aplicación de Wordpress y su base de datos MariaDB.

Primero instalo minikube para crear el clúster y luego me sirvo de kubectl para interactuar y hacer todas las operaciones pertinentes en él.

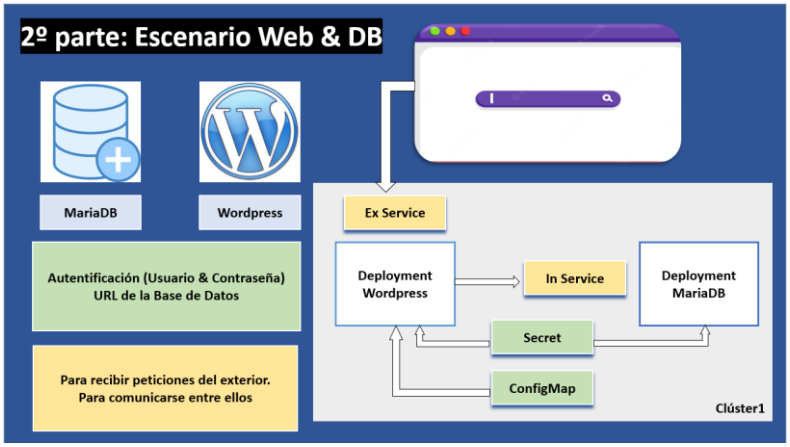
Voy a crear el pod para MariaDB con un Deployment y un Internal Service para comunicarme con él y que no reciba peticiones del exterior del clúster, solo los elementos que estén dentro del mismo clúster.

Después voy a crear el pod para Wordpress con un Deployment donde tendré la URL de MariaDB dentro de un ConfigMap para poder conectarme a ella. Además de que tendrá una autentificación para acceder a la base de datos, esto lo haré en el Deployment (archivo yaml) que tendrá dentro definido un Secret con las credenciales.

Una vez tenga todo esto, voy a necesitar que Wordpress sea accesible a través de un navegador, para ello voy a crear un External Service. Con esto voy a permitir peticiones desde el exterior para hablar con el pod. De esta forma la URL será el HTTP de la IP del nodo y el Service por.

Sería algo así:

1. La petición llega del navegador.
2. Llega al External Service del Wordpress.
3. Se envía así al pod de Wordpress.
4. El pod de Wordpress se conecta al Internal Service de MariaDB.
5. Y llega al pod de MariaDB donde se autentifica con las credenciales.



Escenario de la segunda parte

Instalación y preparación:

Vamos a necesitar instalar las siguientes cosas:

- Minikube y Kubectl para crear y gestionar nuestro clúster y demás.
- Instalar Docker como administrador de contenedores.

Para instalar Docker en nuestra máquina virtual, lo haremos de la siguiente manera:

```
javierramirez@javierramirez:~$ sudo apt update
[sudo] contraseña para javierramirez:
```

Paquetes necesarios para permitir que apt utilice repositorios a través de HTTPS y las dependencias necesarias para trabajar con Docker:

```
javierramirez@javierramirez:~$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Agrega el repositorio de Docker a las fuentes de apt:

```
javierramirez@javierramirez:~$ echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" |
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Docker Engine (motor de Docker) utilizando el siguiente comando:

```
javierramirez@javierramirez:~$ sudo apt install docker-ce docker-ce-cli containerd.io
```

Y vemos que esté correctamente instalado:

```
javierramirez@kubernetes:~$ sudo systemctl status docker
[sudo] contraseña para javierramirez:
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2023-05-20 15:16:14 CEST; 1h 45min ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 1383 (dockerd)
      Tasks: 51
     Memory: 104.7M
        CPU: 16.030s
     CGroup: /system.slice/docker.service
             ├─1383 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
             ├─4444 /usr/bin/docker-proxy -proto tcp -host-ip 127.0.0.1 -host-port 32768 -container-ip 192.168.49.2 -container-port 32443
             ├─4456 /usr/bin/docker-proxy -proto tcp -host-ip 127.0.0.1 -host-port 32769 -container-ip 192.168.49.2 -container-port 8443
             ├─4469 /usr/bin/docker-proxy -proto tcp -host-ip 127.0.0.1 -host-port 32770 -container-ip 192.168.49.2 -container-port 5000
             ├─4484 /usr/bin/docker-proxy -proto tcp -host-ip 127.0.0.1 -host-port 32771 -container-ip 192.168.49.2 -container-port 2376
             └─4497 /usr/bin/docker-proxy -proto tcp -host-ip 127.0.0.1 -host-port 32772 -container-ip 192.168.49.2 -container-port 22

May 20 15:16:13 kubernetes dockerd[1383]: time="2023-05-20T15:16:13.872292098+02:00" level=info msg="Removing stale sandbox 8ac5044b565f8d785b7629b683df950ba3c6e41b076c77028d01826
May 20 15:16:14 kubernetes dockerd[1383]: time="2023-05-20T15:16:14.212628495+02:00" level=warning msg="Error (Unable to complete atomic operation, key modified) deleting object [
May 20 15:16:14 kubernetes dockerd[1383]: time="2023-05-20T15:16:14.432501363+02:00" level=info msg="Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon
May 20 15:16:14 kubernetes dockerd[1383]: time="2023-05-20T15:16:14.504705651+02:00" level=info msg="Loading containers: done."
May 20 15:16:14 kubernetes dockerd[1383]: time="2023-05-20T15:16:14.677924418+02:00" level=info msg="Docker daemon" commit=9dbdbd4 graphdriver=overlay2 version=23.0.6
May 20 15:16:14 kubernetes dockerd[1383]: time="2023-05-20T15:16:14.680575607+02:00" level=info msg="Daemon has completed initialization"
May 20 15:16:14 kubernetes systemd[1]: Started Docker Application Container Engine.
May 20 15:16:14 kubernetes dockerd[1383]: time="2023-05-20T15:16:14.723406798+02:00" level=info msg="API listen on /run/docker.sock"
May 20 15:17:21 kubernetes dockerd[1383]: time="2023-05-20T15:17:21.212818588+02:00" level=info msg="No non-localhost DNS nameservers are left in resolv.conf. Using default extern
May 20 15:17:21 kubernetes dockerd[1383]: time="2023-05-20T15:17:21.213366199+02:00" level=info msg="IPv6 enabled; Adding default IPv6 external servers: [nameserver 2001:4860:4860
lines 1-27/27 (END)
```

Instalamos el binario kubectl con curl en Linux:

```
root@kubernetes:/home/javierramirez# curl -LO https://dl.k8s.io/release/v1.26.0/bin/linux/amd64/kubectl
```

Instalar kubectl:

```
root@kubernetes:/home/javierramirez# sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

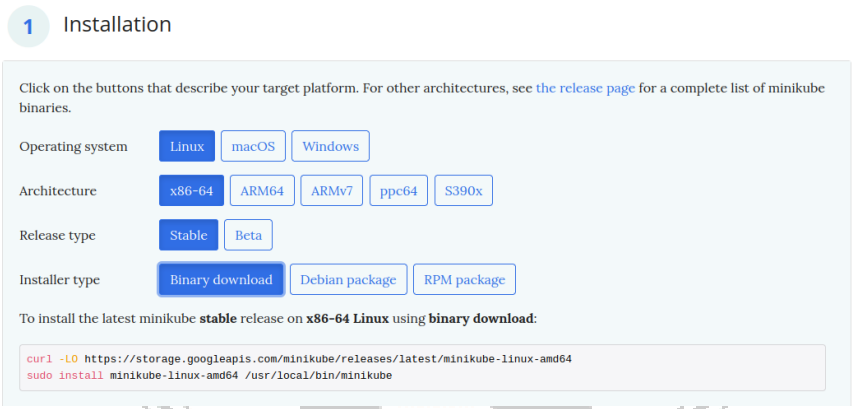
Para asegurarse de que la versión que instaló este actualizada ponemos:

```
root@kubernetes:/home/javierramirez# kubectl version --client=true
```

Y ya tendríamos kubectl instalado:

```
root@kubernetes:/home/javierramirez# curl -LO https://dl.k8s.io/release/v1.26.0/bin/linux/amd64/kubectl
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0    241      0 --:--:-- --:--:-- --:--:--   241
100 45.7M  100 45.7M    0     0  8407k      0  0:00:05  0:00:05 --:--:-- 9863k
root@kubernetes:/home/javierramirez# sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
root@kubernetes:/home/javierramirez# kubectl version --client=true
WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short.  Use --output=yaml|json to get the full version.
Client Version: version.Info{Major:"1", Minor:"26", GitVersion:"v1.26.0", GitCommit:"b46a3f887ca979b1a5d14fd39cb1af43e7e5d12d", GitTreeState:"clean", BuildDate:"2022-12-08T19:58:30Z
", GoVersion:"go1.19.4", Compiler:"gc", Platform:"linux/amd64"}
Kustomize Version: v4.5.7
root@kubernetes:/home/javierramirez#
```

Ahora instalamos Minikube:




```
root@kubernetes:/home/javierramirez# curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 80.0M  100 80.0M    0     0  6150k      0  0:00:13  0:00:13 --:--:-- 9374k
root@kubernetes:/home/javierramirez#
```

Iniciamos Minikube:

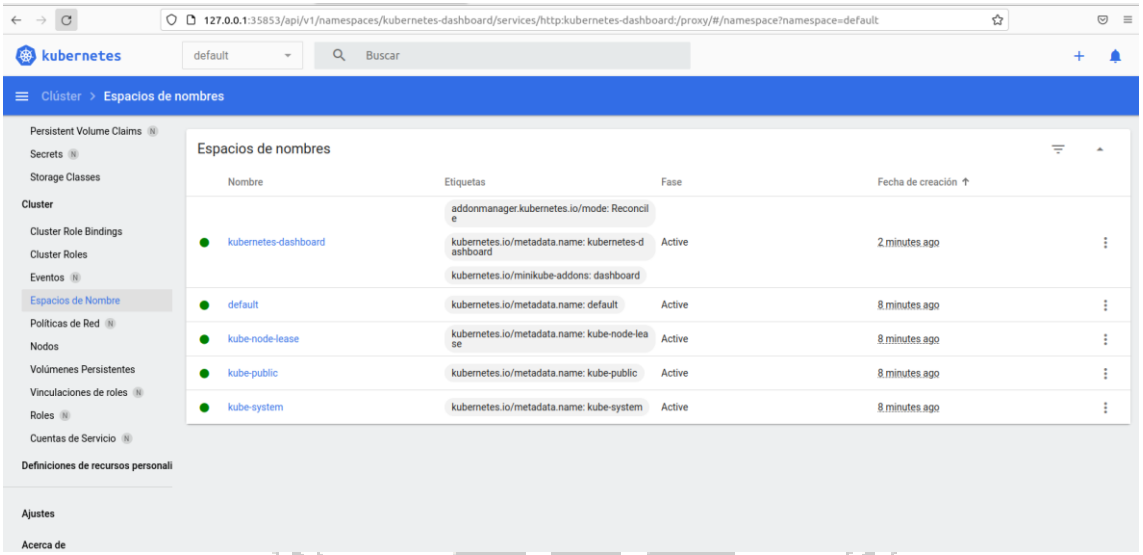
```
javierramirez@kubernetes:~$ minikube start --driver=docker
😄  minikube v1.30.1 en Ubuntu 22.04
✨  Using the docker driver based on user configuration
📌  Using Docker driver with root privileges
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
    > gcr.io/k8s-minikube/kicbase...:  373.53 MiB / 373.53 MiB  100.00% 7.28 Mi
🔥  Creando docker container (CPUs=2, Memory=2200MB) ...
🐳  Preparando Kubernetes v1.26.3 en Docker 23.0.2...
    ▪ Generando certificados y llaves
    ▪ Iniciando plano de control
    ▪ Configurando reglas RBAC...
🔗  Configurando CNI bridge CNI ...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🔎  Verifying Kubernetes components...
🌟  Complementos habilitados: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

```
javierramirez@kubernetes:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

```
javierramirez@kubernetes:~$ minikube dashboard
🔌  Habilitando dashboard
    ▪ Using image docker.io/kubernetesui/dashboard:v2.7.0
    ▪ Using image docker.io/kubernetesui/metrics-scraper:v1.0.8
💡  Some dashboard features require the metrics-server addon. To enable all features please run:

        minikube addons enable metrics-server


🤔  Verifying dashboard health ...
🚀  Launching proxy ...
🤔  Verifying proxy health ...
🎉  Opening http://127.0.0.1:35853/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in your default browser...
```



Vemos lo servicios que tememos de inicio:

```
javierramirez@kubernetes:~$
kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3m22s
javierramirez@kubernetes:~$
```

Vemos todos los pods y en que Namespaces están además de ver su estado.

```
javierramirez@kubernetes:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                 READY   STATUS    RESTARTS        AGE
kube-system   coredns-787d4945fb-rpqql             1/1     Running   0               3m38s
kube-system   etcd-minikube                        1/1     Running   0               3m50s
kube-system   kube-apiserver-minikube              1/1     Running   0               3m50s
kube-system   kube-controller-manager-minikube     1/1     Running   0               3m52s
kube-system   kube-proxy-6sds8                     1/1     Running   0               3m38s
kube-system   kube-scheduler-minikube              1/1     Running   0               3m50s
kube-system   storage-provisioner                  1/1     Running   1 (3m16s ago)   3m49s
javierramirez@kubernetes:~$
```

Vemos los nodos.

```
javierramirez@kubernetes:~$ kubectl get nodes
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   4m46s  v1.26.3
javierramirez@kubernetes:~$
```

Un poco de información del clúster:

```
javierramirez@kubernetes:~$ kubectl cluster-info
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
javierramirez@kubernetes:~$
```

```
javierramirez@kubernetes:~$
kubectl get events
LAST SEEN   TYPE     REASON                    OBJECT          MESSAGE
6m          Normal   NodeHasSufficientMemory   node/minikube   Node minikube status is now: NodeHasSufficientMemory
6m          Normal   NodeHasNoDiskPressure     node/minikube   Node minikube status is now: NodeHasNoDiskPressure
6m          Normal   NodeHasSufficientPID      node/minikube   Node minikube status is now: NodeHasSufficientPID
5m53s       Normal   Starting                  node/minikube   Starting kubelet.
5m53s       Normal   NodeHasSufficientMemory   node/minikube   Node minikube status is now: NodeHasSufficientMemory
5m53s       Normal   NodeHasNoDiskPressure     node/minikube   Node minikube status is now: NodeHasNoDiskPressure
5m53s       Normal   NodeHasSufficientPID      node/minikube   Node minikube status is now: NodeHasSufficientPID
5m53s       Normal   NodeNotReady              node/minikube   Node minikube status is now: NodeNotReady
5m53s       Normal   NodeAllocatableEnforced   node/minikube   Updated Node Allocatable limit across pods
5m53s       Normal   NodeReady                 node/minikube   Node minikube status is now: NodeReady
5m42s       Normal   RegisteredNode            node/minikube   Node minikube event: Registered Node minikube in Controller
5m40s       Normal   Starting                  node/minikube
javierramirez@kubernetes:~$
```

```
javierramirez@kubernetes:~$
kubectl get services
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3m22s
javierramirez@kubernetes:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                               READY   STATUS    RESTARTS        AGE
kube-system   coredns-787d4945fb-rpqql           1/1     Running   0               3m38s
kube-system   etcd-minikube                      1/1     Running   0               3m50s
kube-system   kube-apiserver-minikube            1/1     Running   0               3m50s
kube-system   kube-controller-manager-minikube   1/1     Running   0               3m52s
kube-system   kube-proxy-6sds8                   1/1     Running   0               3m38s
kube-system   kube-scheduler-minikube            1/1     Running   0               3m50s
kube-system   storage-provisioner                1/1     Running   1 (3m16s ago)   3m49s
javierramirez@kubernetes:~$ kubectl get nodes
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   4m46s  v1.26.3
javierramirez@kubernetes:~$ kubectl cluster-info
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
javierramirez@kubernetes:~$
kubectl get events
LAST SEEN   TYPE     REASON                    OBJECT          MESSAGE
6m          Normal   NodeHasSufficientMemory   node/minikube   Node minikube status is now: NodeHasSufficientMemory
6m          Normal   NodeHasNoDiskPressure     node/minikube   Node minikube status is now: NodeHasNoDiskPressure
6m          Normal   NodeHasSufficientPID      node/minikube   Node minikube status is now: NodeHasSufficientPID
5m53s       Normal   Starting                  node/minikube   Starting kubelet.
5m53s       Normal   NodeHasSufficientMemory   node/minikube   Node minikube status is now: NodeHasSufficientMemory
5m53s       Normal   NodeHasNoDiskPressure     node/minikube   Node minikube status is now: NodeHasNoDiskPressure
5m53s       Normal   NodeHasSufficientPID      node/minikube   Node minikube status is now: NodeHasSufficientPID
5m53s       Normal   NodeNotReady              node/minikube   Node minikube status is now: NodeNotReady
5m53s       Normal   NodeAllocatableEnforced   node/minikube   Updated Node Allocatable limit across pods
5m53s       Normal   NodeReady                 node/minikube   Node minikube status is now: NodeReady
5m42s       Normal   RegisteredNode            node/minikube   Node minikube event: Registered Node minikube in Controller
5m40s       Normal   Starting                  node/minikube
javierramirez@kubernetes:~$
```

Para ver todos los componentes que tenemos en nuestro clúster recién creado:

```
javierramirez@kubernetes:~$ kubectl get all
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   17h
javierramirez@kubernetes:~$
```

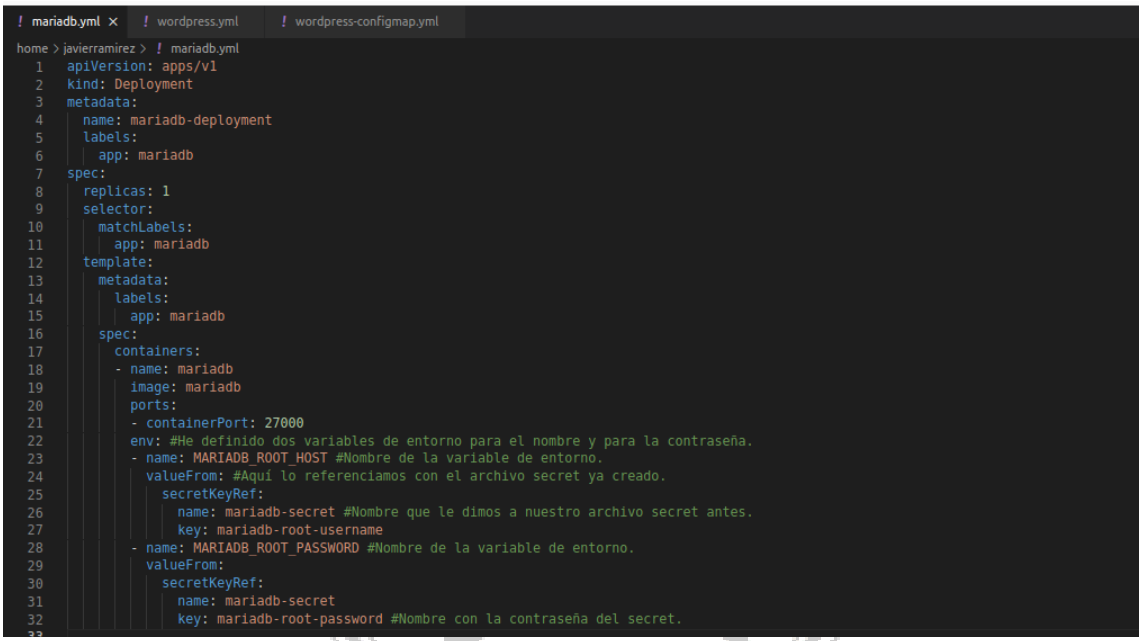
Puesta en marcha del escenario.

MariaDB Deployment:

Un Deployment en Kubernetes es la herramienta principal para crear y configurar componentes en nuestro clúster.

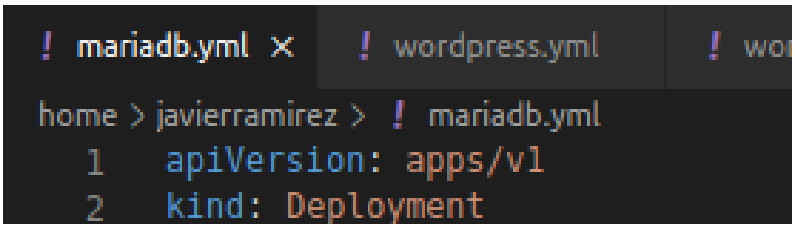
En términos simples, un Deployment es una descripción declarativa de cómo se debe ejecutar una aplicación en Kubernetes. Define qué contenedores se ejecutarán, qué imágenes se utilizarán, cómo se escalará el número de réplicas, cómo se actualizará la aplicación y cómo se gestionarán los errores.

Solo tengo el servicio de Kubernetes como hemos visto antes. Vamos a crear nuestro MariaDB Deployment de la siguiente manera. Creamos el siguiente archivo de tipo YAML en visual estudio:



Voy a explicar parte a parte el Deployment que hemos creado:

- En las primeras líneas tenemos que api Versión vamos a usar y que kind de recurso queremos crear, en nuestro caso un Deployment.



Esta parte de aquí es la ***metadata***, que contiene información sobre la implementación que se está definiendo, como su nombre y etiquetas.

Las etiquetas son útiles para identificar y agrupar recursos relacionados en Kubernetes. Las etiquetas también se pueden utilizar para organizar recursos de Kubernetes en función de su entorno. Además, mediante las etiquetas vamos a conectar los elementos de la parte

de la metadata. En name he puesto el nombre del Deployment. Este archivo ejecutará la imagen MariaDB desde el repositorio de imágenes Docker Hub.

```
3   metadata:
4     name: mariadb-deployment
5     labels:
6       app: mariadb
```

Si seguimos bajando tenemos esta parte que es la de ***especificación***. Es la sección donde se especifica el comportamiento deseado del Deployment.

- En este caso hemos puesto que solo queremos una réplica del pod en. spec. replicas. Kubernetes asegura que el número de Pods en ejecución coincida con el número de réplicas deseado. Una réplica es una copia exacta de un conjunto de Pods que están configurados y administrados de manera idéntica. El propósito de usar replicas es proporcionar alta disponibilidad y capacidad de escalado horizontal a las aplicaciones que se ejecutan en Kubernetes. Si por alguna razón un Pod falla o se elimina, Kubernetes automáticamente iniciará un nuevo Pod para mantener el número de réplicas deseado. Además, si se desea aumentar la capacidad de una aplicación, se puede aumentar el número de réplicas en el Deployment, y Kubernetes creará automáticamente los Pods adicionales.
- .spec.selector especifica qué pods se verán afectados por esta implementación. En este caso, se seleccionan todos los pods que tengan la etiqueta app: MariaDB.
- .spec. selector.matchLabels contiene un mapa de pares {clave, valor} que permite a la implementación buscar y administrar los pods creados. Dice a que pods del deployment se le debe de aplicar.
- Justo debajo.spec.selector.matchLabels.app tiene que coincidir con el atributo app del. spec. template.metadata.labels, como se ve en la imagen. La parte del template es

```
7   spec:
8     replicas: 1
9     selector:
10      matchLabels:
11        app: mariadb
12    template:
13      metadata:
14        labels:
15          app: mariadb
```

- En esta parte de especificación tenemos primero definido. spec.spec.containers donde se especifica la lista de contenedores que pertenecen al pod.
- Con spec.spec.containers.name se especifica el nombre del contenedor especificado como una etiqueta DNS.

- Con .spec.spec.containers.image_ se especifica el nombre de la imagen de contenedor.
- Con .spec.spec.containers.ports, se especifica la lista de puertos que se van a exponer desde el contenedor.
- En spec.spec.containers.ports.containerPort se especifica el número de puerto que se va a exponer en la dirección IP del pod.

Lo que sigue son las variables de entorno que las voy a explicar un poco más adelante en la sección del Secret.

```
16      spec:
17        containers:
18        - name: mariadb
19          image: mariadb
20          ports:
21          - containerPort: 27000
22          env: #He definido dos variables de entorno para el nombre y para la contraseña.
23          - name: MARIADB_ROOT_HOST #Nombre de la variable de entorno.
24            valueFrom: #Aquí lo referenciamos con el archivo secret ya creado.
25              secretKeyRef:
26                name: mariadb-secret #Nombre que le dimos a nuestro archivo secret antes.
27                key: mariadb-root-username
28          - name: MARIADB_ROOT_PASSWORD #Nombre de la variable de entorno.
29            valueFrom:
30              secretKeyRef:
31                name: mariadb-secret
32                key: mariadb-root-password #Nombre con la contraseña del secret.
```

Y la tercera y última parte de un deployment es el ***estado***. En esta sección se describen el estado actual de las réplicas del pod en comparación al estado que nosotros queremos. Podemos ver información sobre el número de réplicas que tenemos funcionando o la versión actual de este. Kubernetes lo estará actualizando y revisando constantemente. Toda esta información Kubernetes la coge del etcd, que como ya dijimos, es donde está almacenado toda la configuración y el estado de todos los componentes del clúster.

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl describe deployment mariadb-deployment
Name:                   mariadb-deployment
Namespace:              default
CreationTimestamp:      Sun, 09 Apr 2023 11:24:54 +0200
Labels:                 app=mariadb
Annotations:            deployment.kubernetes.io/revision: 10
Selector:               app=mariadb
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=mariadb
  Containers:
   mariadb:
    Image:      mariadb
    Port:       3306/TCP
    Host Port:  0/TCP
    Limits:
      cpu:     500m
      memory:  128Mi
    Requests:
      cpu:     250m
      memory:  64Mi
    Environment:
      MARIADB_USER:           <set to the key 'mariadb-root-username' in secret 'mariadb-secret'>  Optional: false
      MARIADB_ROOT_PASSWORD:  <set to the key 'mariadb-root-password' in secret 'mariadb-secret'>  Optional: false
```

MariaDB Secret:

En Kubernetes, un Secret es un objeto que permite almacenar información sensible, como contraseñas, tokens de autenticación u otros datos confidenciales que una aplicación necesita para funcionar correctamente, de forma segura en el clúster.

Los secretos en Kubernetes se almacenan en formato codificado en base64 y se pueden crear y administrar mediante la CLI de Kubernetes o a través de archivos YAML. Los secretos pueden ser utilizados por cualquier objeto dentro del clúster de Kubernetes, incluyendo Deployment, pods, Service y otros.

Por tanto, yo lo he usado para almacenar las credenciales de la base de datos y he codificado los registros. De tal forma que me permite tener más seguridad a la hora de controlar quien puede acceder y quien no mediante la autentificación.

Creo un Secret donde vamos a tener el usuario y la contraseña root.

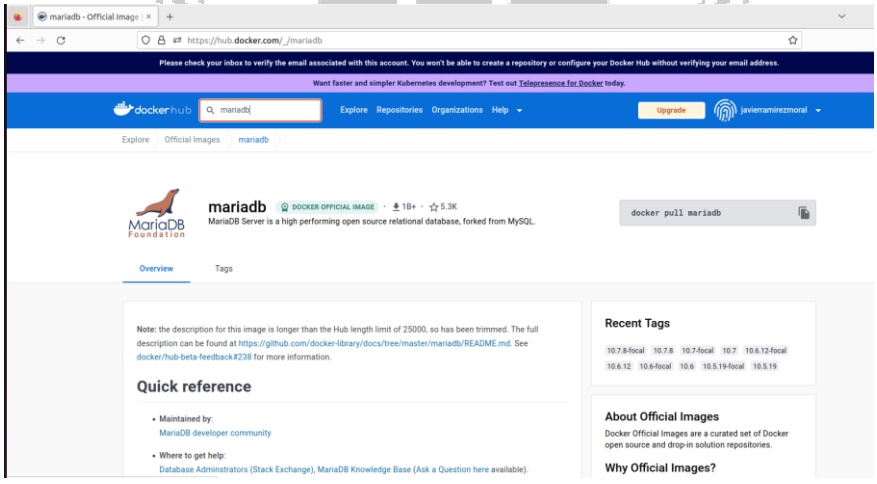
```yaml
! mariadb.yml   ! mariadb-secret.yml ×
home > javierramirez > ! mariadb-secret.yml
1  apiVersion: v1
2  kind: Secret #De tipo secreto
3  metadata: #La parte de metadata tenemos el nombre.
4      name: mariadb-secret
5  type: Opaque #El tipo es el por defecto y es el más básico para crear el par de llaves.
6  data: #Aquí están los valores de las dos llaves. No son de texto plano.
7      mariadb-root-username: dXNlcm5hbWU= #están cifradas con base64
8      mariadb-root-password: cGFzc3dvcmQ=
9
```

La codificación en base 64 se ha hecho así:

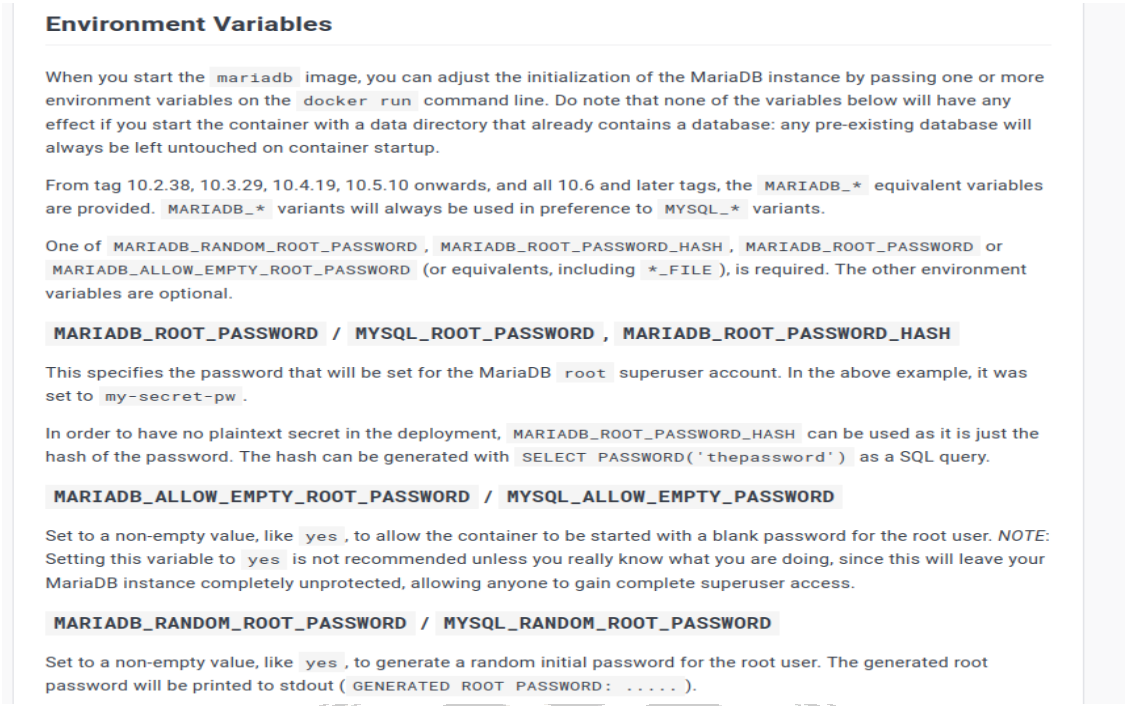
```
javierramirez@kubernetes:~$ echo -n 'username' | base64
dXNlcm5hbWU=
javierramirez@kubernetes:~$ echo -n 'password' | base64
cGFzc3dvcmQ=
javierramirez@kubernetes:~$
```

Las variables de entorno e imagen comprobamos que todo esté bien desde Docker hub.

Las variables de entorno en Kubernetes permiten configurar la aplicación de manera dinámica y personalizada en función del entorno en el que se esté ejecutando la aplicación.



Las estamos usando para pasar información sensible, como contraseñas o claves de acceso, de manera segura a los contenedores de la aplicación. En lugar de incluir esta información en el código de la aplicación, se puede configurar como una variable de entorno en el Deployment o pod correspondiente.



Environment Variables

When you start the `mariadb` image, you can adjust the initialization of the MariaDB instance by passing one or more environment variables on the `docker run` command line. Do note that none of the variables below will have any effect if you start the container with a data directory that already contains a database: any pre-existing database will always be left untouched on container startup.

From tag 10.2.38, 10.3.29, 10.4.19, 10.5.10 onwards, and all 10.6 and later tags, the `MARIADB_*` equivalent variables are provided. `MARIADB_*` variants will always be used in preference to `MYSQL_*` variants.

One of `MARIADB_RANDOM_ROOT_PASSWORD`, `MARIADB_ROOT_PASSWORD_HASH`, `MARIADB_ROOT_PASSWORD` or `MARIADB_ALLOW_EMPTY_ROOT_PASSWORD` (or equivalents, including `*_FILE`), is required. The other environment variables are optional.

`MARIADB_ROOT_PASSWORD` / `MYSQL_ROOT_PASSWORD`, `MARIADB_ROOT_PASSWORD_HASH`

This specifies the password that will be set for the MariaDB `root` superuser account. In the above example, it was set to `my-secret-pw`.

In order to have no plaintext secret in the deployment, `MARIADB_ROOT_PASSWORD_HASH` can be used as it is just the hash of the password. The hash can be generated with `SELECT PASSWORD('thepassword')` as a SQL query.

`MARIADB_ALLOW_EMPTY_ROOT_PASSWORD` / `MYSQL_ALLOW_EMPTY_PASSWORD`

Set to a non-empty value, like `yes`, to allow the container to be started with a blank password for the root user. *NOTE*: Setting this variable to `yes` is not recommended unless you really know what you are doing, since this will leave your MariaDB instance completely unprotected, allowing anyone to gain complete superuser access.

`MARIADB_RANDOM_ROOT_PASSWORD` / `MYSQL_RANDOM_ROOT_PASSWORD`

Set to a non-empty value, like `yes`, to generate a random initial password for the root user. The generated root password will be printed to stdout (`GENERATED ROOT PASSWORD: .....`).

Hemos hecho la configuración de los archivos, pero no tenemos creado aún nada en nuestro clúster. Creamos el Secret con el siguiente comando:

```
javierramirez@kubernetes:~$ kubectl apply -f mariadb-secret.yml
secret/mariadb-secret created
javierramirez@kubernetes:~$ kubectl get secret
NAME             TYPE     DATA   AGE
mariadb-secret   Opaque   2      46s
javierramirez@kubernetes:~$
```

Ya podemos referenciarlo en nuestro archivo Deployment que es esta parte. Tenemos que crear antes el Secret para ello y luego en el Deployment referenciarlo porque si no dará error.

```yaml
env: #He definido dos variables de entorno para el nombre y para la contraseña.
- name: MARIADB_ROOT_HOST #Nombre de la variable de entorno.
  valueFrom: #Aquí lo referenciamos con el archivo secret ya creado.
    secretKeyRef:
      name: mariadb-secret #Nombre que le dimos a nuestro archivo secret antes.
      key: mariadb-root-username
- name: MARIADB_ROOT_PASSWORD #Nombre de la variable de entorno.
  valueFrom:
    secretKeyRef:
      name: mariadb-secret
      key: mariadb-root-password #Nombre con la contraseña del secret.
```

Y lo aplicamos todo

```
javierramirez@kubernetes:~$ kubectl apply -f mariadb.yml
deployment.apps/mariadb-deployment created
```

Vemos que ya tenemos el pod activado, el Deployment y la réplica.

```
javierramirez@kubernetes:~$ kubectl get all
NAME                                        READY   STATUS    RESTARTS   AGE
pod/mariadb-deployment-6ffdbcfc68-l5f7t     1/1     Running   0          40s

NAME                        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
service/kubernetes          ClusterIP   10.96.0.1       <none>        443/TCP     18h
service/mariadb-service     ClusterIP   10.105.59.240   <none>        27017/TCP   40s

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mariadb-deployment      1/1     1            1           40s

NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/mariadb-deployment-6ffdbcfc68      1         1         1       40s
javierramirez@kubernetes:~$
```

Podemos ver toda la información del pod de MariaDB.

```
javierramirez@kubernetes:~$ kubectl describe pod mariadb-deployment-6ffdbcfc68-l5f7t
Name:             mariadb-deployment-6ffdbcfc68-l5f7t
Namespace:        default
Priority:         0
Service Account:  default
Node:             minikube/192.168.49.2
Start Time:       Thu, 06 Apr 2023 13:32:08 +0200
Labels:           app=mariadb
                  pod-template-hash=6ffdbcfc68
Annotations:      <none>
Status:           Running
IP:               10.244.0.4
IPs:
  IP:           10.244.0.4
Controlled By:  ReplicaSet/mariadb-deployment-6ffdbcfc68
Containers:
  mariadb:
    Container ID:   docker://ce74d84b476c067047ece8e1620acf29491f95a9d5d6fc87132676cbf66fceab
    Image:          mariadb
    Image ID:       docker-pullable://mariadb@sha256:9ff479f244cc596aed9794d035a9f352662f2caed933238c533024df64569853
    Port:           27000/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Thu, 06 Apr 2023 13:32:40 +0200
    Ready:          True
    Restart Count:  0
    Environment:
      MARIADB_ROOT_HOST:      <set to the key 'mariadb-root-username' in secret 'mariadb-secret'>  Optional: false
      MARIADB_ROOT_PASSWORD:  <set to the key 'mariadb-root-password' in secret 'mariadb-secret'>  Optional: false
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-cjvx6 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-cjvx6:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
```

```
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  4m50s  default-scheduler  Successfully assigned default/mariadb-deployment-6ffdbcfc68-l5f7t to minikube
  Normal  Pulling    4m49s  kubelet            Pulling image "mariadb"
  Normal  Pulled     4m18s  kubelet            Successfully pulled image "mariadb" in 30.228973237s (30.228982882s including waiting)
  Normal  Created    4m18s  kubelet            Created container mariadb
  Normal  Started    4m18s  kubelet            Started container mariadb
javierramirez@kubernetes:~$
```

Ya tenemos el pod de MariaDB corriendo y su réplica. Ahora vamos a crear el Internal Service para que otros componentes puedan hablar con MariaDB. Se puede hacer en el mismo fichero de MariaDB o en otro aparte como he hecho con el Secret. Lo haré en el mismo fichero esta vez, que es lo normal a la hora de definir services.

MariaDB Internal Service:

Un Internal Service en Kubernetes es un tipo de servicio que se utiliza para exponer los pods dentro del clúster de Kubernetes. A diferencia de los servicios externos, los servicios internos solo son accesibles desde dentro del clúster. Los servicios internos se crean con

una dirección IP virtual (ClusterIP), que se asigna a un conjunto de pods que tienen las mismas etiquetas. Cuando se crea un servicio interno, se puede acceder a los pods correspondientes mediante su dirección IP virtual.

Los servicios internos se utilizan comúnmente para permitir que las aplicaciones dentro del clúster se comuniquen entre sí de manera eficiente y segura, sin tener que exponerlos directamente a Internet.

Entonces, yo he usado un servicio debido a que debido a que como ya mencionamos los pods son efímeros y fácilmente mueren, esto haría que Kubernetes al reemplazarlo por otro nuevo también le asignara una nueva dirección IP, lo que haría que tuviésemos que estar reconfigurando cada vez que suceda. En cambio, al usar un service, este asigna una dirección IP permanente a nuestro pod a pesar de que este muera. Esto es debido a que el ciclo de vida del service no está relacionado con el del pod. Además de una dirección IP estática tiene un Loadbalancer que se utiliza para distribuir el tráfico de red entre los diferentes pods que forman parte del servicio, se adapta automáticamente y distribuye el tráfico de manera equitativa entre los pods disponibles.

El que las IP cambien sería un problema a la hora de comunicar nuestras aplicaciones por ello el motivo de usar un service, en este caso configurado como internal service.

Y nuestro archivo de configuración quedaría así. El tipo de servicio es ClusterIP que es el que pone Kubernetes por defecto por lo que no hace falta especificarlo si queremos. El servicio se asigna a una dirección IP interna única dentro del clúster. Solo los recursos internos del clúster pueden acceder al servicio a través de esta dirección IP. Este tipo de puerto no expone el servicio externamente.

```
34  apiVersion: v1
35  kind: Service #De tipo servicio.
36  metadata:
37    name: mariadb-service
38  spec:
39    selector: #Este servicio se va a conectar a través de una label al pod.
40      app: mariadb
41    ports: #Aquí exponemos el servicio
42      - protocol: TCP
43        port: 27000 #El puerto del servicio.# Puede ser igual o diferente al del targetport.
44        targetPort: 27000 #El puerto del contenedor o pod. Coincide con el definido arriba.
45
```

Vemos como se referencian el Deployment y el Service:

```
    spec:
      containers:
      - name: mariadb
        image: mariadb
        ports:
        - containerPort: 27000
        env: #He definido dos variables de entorno para el nombre y para la contraseña
        - name: MARIADB_ROOT_HOST #Nombre de la variable de entorno.
          valueFrom: #Aquí lo referenciamos con el archivo secret ya creado.
            secretKeyRef:
              name: mariadb-secret #Nombre que le dimos a nuestro archivo secret antes
              key: mariadb-root-username
        - name: MARIADB_ROOT_PASSWORD #Nombre de la variable de entorno.
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-password #Nombre con la contraseña del secret.
---
apiVersion: v1
kind: Service #De tipo servicio.
metadata:
  name: mariadb-service
spec:
  selector: #Este servicio se va a conectar a través de una label al pod.
    app: mariadb
  ports: #Aquí exponemos el servicio
    - protocol: TCP
      port: 27000 #El puerto del servicio.
      targetPort: 27000 #El puerto del contenedor. Coincide con el definido arriba.
```

Vamos a crearlo. El unchanged significa que la parte de Deployment no la he cambiado

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f mariadb.yml
deployment.apps/mariadb-deployment unchanged
service/mariadb-service created
javierramirez@kubernetes:~/k8s-configuration$
```

Vamos a comprobarlo:

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl get service
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes        ClusterIP   10.96.0.1       <none>        443/TCP    38h
mariadb-service   ClusterIP   10.96.77.154    <none>        3306/TCP   45s
javierramirez@kubernetes:~/k8s-configuration$
```

Ahora vamos a comprobar que el servicio está en el pod correcto. El endpoint es la IP de un pod y el puerto donde la aplicación dentro del pod está escuchando.

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl describe service mariadb-service
Name:              mariadb-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=mariadb
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.96.77.154
IPs:               10.96.77.154
Port:              <unset>  3306/TCP
TargetPort:        3306/TCP
Endpoints:         10.244.0.10:3306
Session Affinity:  None
Events:            <none>
```

Vamos a comprobar que sea el pod correcto que creamos antes 3306 es el puerto por dónde la aplicación estará escuchando dentro del pod.

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl describe service mariadb-service
Name:              mariadb-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=mariadb
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.96.77.154
IPs:               10.96.77.154
Port:              <unset>  3306/TCP
TargetPort:        3306/TCP
Endpoints:         10.244.0.10:3306
Session Affinity:  None
Events:            <none>
javierramirez@kubernetes:~/k8s-configuration$ kubectl get pod -o wide
NAME                                  READY   STATUS    RESTARTS   AGE   IP            NODE       NOMINATED NODE   READINESS GATES
mariadb-deployment-f678bdf97-87688   1/1     Running   0          55s   10.244.0.10   minikube   <none>           <none>
javierramirez@kubernetes:~/k8s-configuration$
```

Para ver todo lo creado hasta ahora en MariaDB:

```
javierramirez@kubernetes:~$ kubectl get all | grep  mariadb
pod/mariadb-deployment-6ffdbcfc68-l5f7t   1/1     Running   0          34m
service/mariadb-service   ClusterIP   10.105.59.240   <none>        27000/TCP   34m
deployment.apps/mariadb-deployment   1/1     1            1           34m
replicaset.apps/mariadb-deployment-6ffdbcfc68   1         1         1       34m
javierramirez@kubernetes:~$
```

Ahora vamos a crear el Deployment de WordPress y en su Service y vamos a poner la URL en un ConfigMap de la base de datos y conectarlos.

Así nos quedaría todo el archivo de MariaDB:

mariadb.yml | wordpress.yml | wordpress-configmap.yml

home > javierramirez > ! mariadb.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-deployment
  labels:
    app: mariadb
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mariadb
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb
        image: mariadb
        ports:
        - containerPort: 27000
        env: #He definido dos variables de entorno para el nombre y para la contraseña.
        - name: MARIADB_ROOT_HOST #Nombre de la variable de entorno.
          valueFrom: #Aquí lo referenciamos con el archivo secret ya creado.
            secretKeyRef:
              name: mariadb-secret #Nombre que le dimos a nuestro archivo secret antes.
              key: mariadb-root-username
        - name: MARIADB_ROOT_PASSWORD #Nombre de la variable de entorno.
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-password #Nombre con la contraseña del secret.
---
apiVersion: v1
kind: Service #De tipo servicio.
metadata:
  name: mariadb-service
spec:
  selector: #Este servicio se va a conectar a través de una label al pod.
    app: mariadb
  ports: #Aquí exponemos el servicio
    - protocol: TCP
      port: 27000 #El puerto del servicio.# Puede ser igual o diferente al del targetport.
      targetPort: 27000 #El puerto del contenedor o pod. Coincide con el definido arriba.
```

Para ver nuestra base de datos y crearla:

```
javierramirez@kubernetes:~$ mariadb -u dXNlcm5hbWU= -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 31
Server version: 10.11.3-MariaDB-1:10.11.3+maria~ubu2204 mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

```
MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mariadb            |
+--------------------+
2 rows in set (0,015 sec)
```

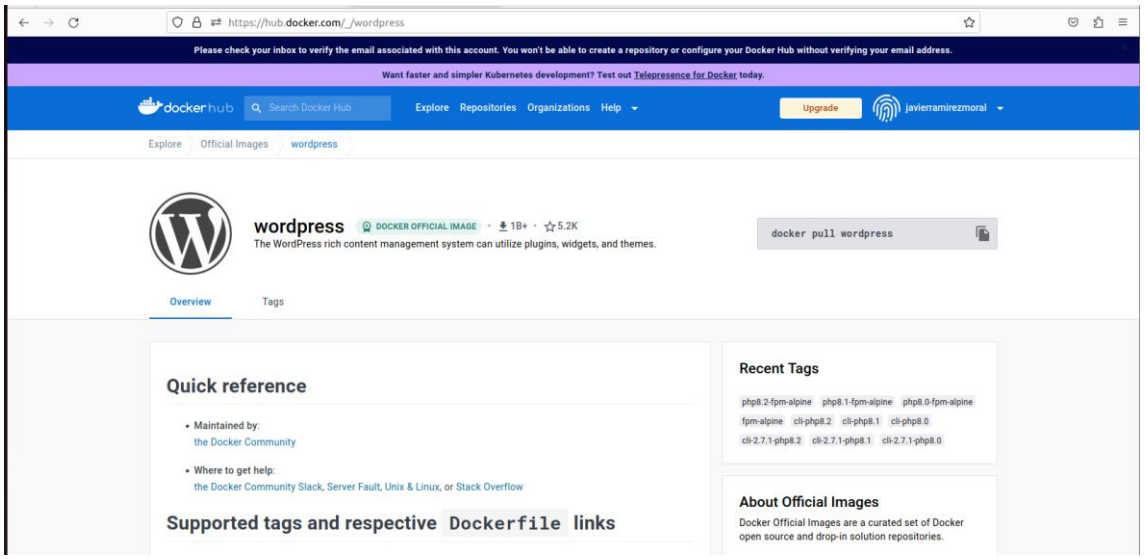
Creamos una para el WordPress:

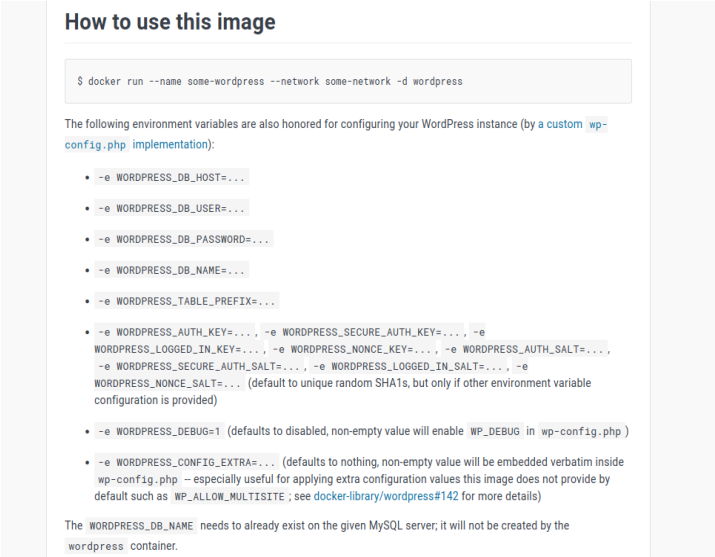
```
MariaDB [(none)]> CREATE DATABASE wordpress;
Query OK, 1 row affected (0,040 sec)

MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mariadb            |
| mysql              |
| performance_schema |
| sys                |
| wordpress          |
+--------------------+
6 rows in set (0,042 sec)
```

Deployment Wordpress:

Creamos el archivo yaml para el deployment. Al igual que antes comprobamos que la imagen este correcto y las variables de entorno cuales son:





Necesitamos para Wordpress decirle tres cosas con las variables de entorno:

- A que base de datos se debe de conectar, para ello necesitamos la IP de MariaDB para conectarse al Internal Service. Para ello usaremos la variable de entorno WORDPRESS_DB_HOST.
- Vamos a necesitar credenciales para que MariaDB pueda autentificar la conexión, en nuestro caso WORDPRESS_DB_USER, WORDPRESS_DB_PASSWORD.

Obviamente vamos a llamar a todo igual que en el Deployment de MariaDB.

Wordpress ConfigMap:

Un ConfigMap en Kubernetes es un objeto que permite separar la configuración del contenedor del propio contenedor. Es decir, es una forma de almacenar datos de configuración en un objeto de Kubernetes para que los contenedores puedan acceder a ellos como variables de entorno, argumentos de línea de comandos o archivos de configuración montados.

Lo uso para que nuestra aplicación web donde ya defino la información de conexión en su archivo de configuración, se conecta a la base de datos utilizando un ConfigMap para

almacenar esa información de conexión y hacer que la aplicación la lea desde allí en lugar de tenerla directamente en su archivo de configuración. Esto hace que la aplicación sea más flexible y fácil de mantener, ya que no hay necesidad de actualizar el archivo de configuración de la aplicación cada vez que se cambia la información de conexión a la base de datos.

El ConfigMap debe de estar ya creado en el clúster antes de referenciarlo. Primero el ConfigMap y luego lo ponemos en el Deployment.

mariadb.yml | wordpress.yml | wordpress-configmap.yml

home > javierramirez > ! wordpress-configmap.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: wordpress-configmap
data:
  database_url: mariadb-service #Ponemos el nombre del service que hicimos antes.
```

Así quedaría nuestro Deployment con el ConfigMap y Secret referenciado:

home > javierramirez > k8s-configuration > ! wordpress.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:latest
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-username
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-password
        - name: WORDPRESS_DB_HOST
          valueFrom:
            configMapKeyRef:
              name: mariadb-configmap
              key: database_url
---
```

Ahora vamos a ejecutarlo y comprobar.

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f mariadb-configmap.yml
configmap/mariadb-configmap created
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f wordpress.yml
deployment.apps/wordpress created
javierramirez@kubernetes:~/k8s-configuration$
```

Vemos los dos pods creados en nuestro clúster.

```
javierramirez@kubernetes:~$ kubectl get pod
NAME                                READY   STATUS    RESTARTS      AGE
mariadb-deployment-6ffdbcfc68-l5f7t   1/1     Running   1 (55m ago)   5h59m
wordpress-b7dccb9bb-rlfvt             1/1     Running   0             88s
javierramirez@kubernetes:~$
```

WordPress Service External:

El paso final es acceder a WordPress desde un navegador y para ello vamos a necesitar un Service External. Para ello, en la sección de especificación pongo que sea de tipo Loadbalancer que lo que hará será aceptar las peticiones externas asignando al External Service una dirección IP. Además, vamos a poner un tercer puerto que será el nodeport donde la IP externa será abierta. Este es el puerto que tendré que poner en el navegador para acceder a este Service.

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: wordpress-service
spec:
  selector:
    app: wordpress
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30000
---
```

El tema de los puertos es el siguiente:

- ContainerPort: Es el puerto de del contenedor dónde está Wordpress.
- Port: Es el puerto del Service que se expone internamente dentro del clúster, es decir, otros recursos de Kubernetes dentro del mismo clúster lo usarán para acceder al servicio
- TargetPort: Es el puerto de los pods donde están escuchando para recibir el tráfico. Conviene que sea el mismo que el del container port.
- NodePort: Es el puerto por donde vamos a buscar junto con la IP de nuestra aplicación en el navegador y por donde van a llegar las peticiones y el tráfico externo. Es el puerto por el que exponemos el servicio y redirige este tráfico al target port.

Así se vería todo:

home > javierramirez > k8s-configuration > ! wordpress.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:latest
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-username
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-password
        - name: WORDPRESS_DB_HOST
          valueFrom:
            configMapKeyRef:
              name: mariadb-configmap
              key: database_url
        - name: WORDPRESS_DB_NAME
          value: mariadb-deployment
---
apiVersion: v1
kind: Service
metadata:
  name: wordpress-service
spec:
  selector:
    app: wordpress
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30000
```

Aplicamos y testeamos todo.

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f wordpress.yml
deployment.apps/wordpress unchanged
service/wordpress-service created
javierramirez@kubernetes:~/k8s-configuration$ kubectl get service
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes          ClusterIP      10.96.0.1       <none>        443/TCP        39h
mariadb-service     ClusterIP      10.96.77.154    <none>        3306/TCP       42m
wordpress-service   LoadBalancer   10.99.220.250   <pending>     80:30000/TCP   10s
javierramirez@kubernetes:~/k8s-configuration$
```

En el fichero mariadb.yml al definir el Internal Service no definimos ningún tipo por que ClusterIP es como un tipo de servicio interno por defecto así que no hace falta definirlo.

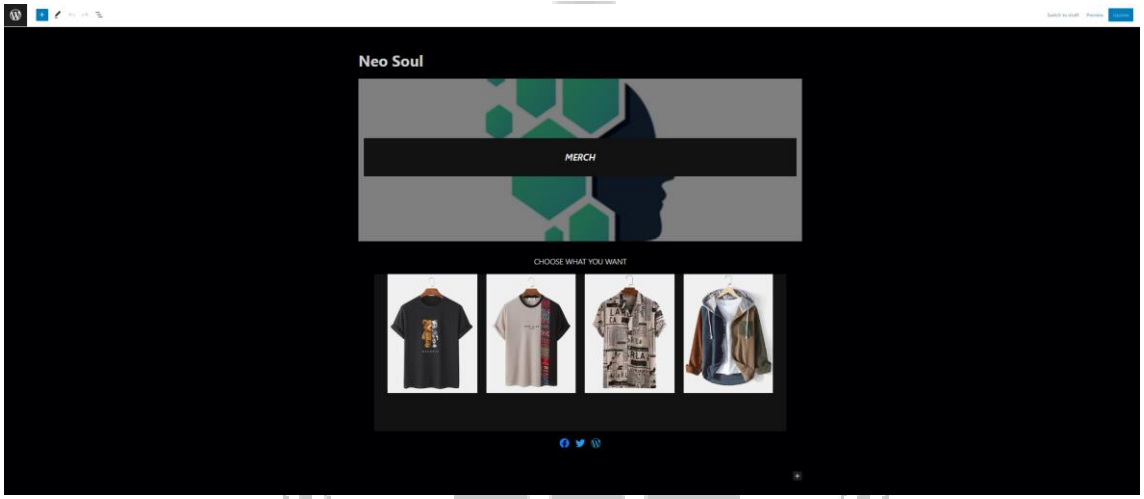
La diferencia aquí es que Clúster IP nos da la dirección IP del servicio interno y el Loadbalancer nos da la dirección IP del servicio interno y además de eso nos da la dirección IP externa donde se producen las peticiones desde el exterior (pending). Pending significa que no tiene aún la IP externa. He decido que sea de tipo Loadbalancer ya que como vamos luego a integrar el clúster en la nube de Azure, nos vamos a servir de

una de las ventajas que tiene que es el balanceador de carga. *Así si nuestro sitio web tiene miles de clientes diferentes por minuto, es fácil para un solo servidor mantenerse al día con la demanda de solicitudes. Esto crea una mala experiencia de usuario con tiempos de inactividad impredecibles. Un balanceador de carga nos ayuda a distribuir las solicitudes a los diferentes servidores en el grupo de recursos. Esto garantiza que ningún servidor se sobrecargue en ningún momento.* El *tráfico entrante se distribuye uniformemente entre varios pods para un mejor rendimiento y una alta disponibilidad.*

Para darle la dirección externa. Con este comando vamos a asignarle la dirección IP automáticamente y se nos abre en nuestro navegador WordPress y ya lo tendríamos todo.

```
javierramirez@kubernetes:~/k8s-configuration$ minikube service wordpress-service
```

Y nos registramos con nuestras credenciales y accedemos Wordpress donde ya podremos crear nuestra página web.

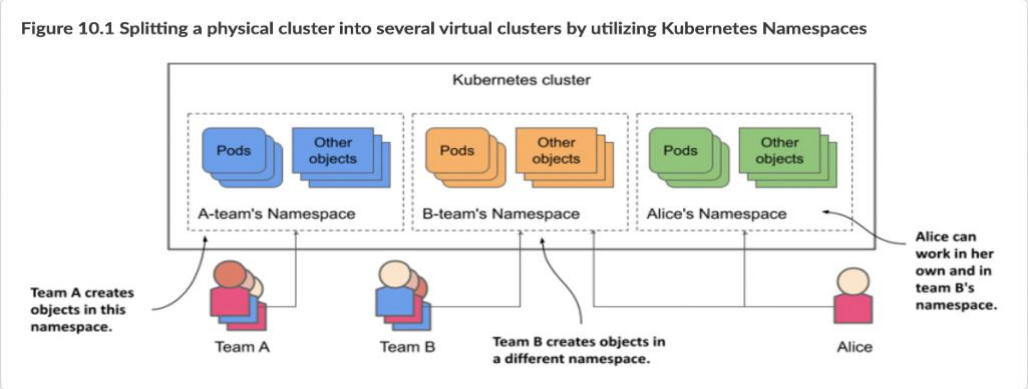


Namespaces:

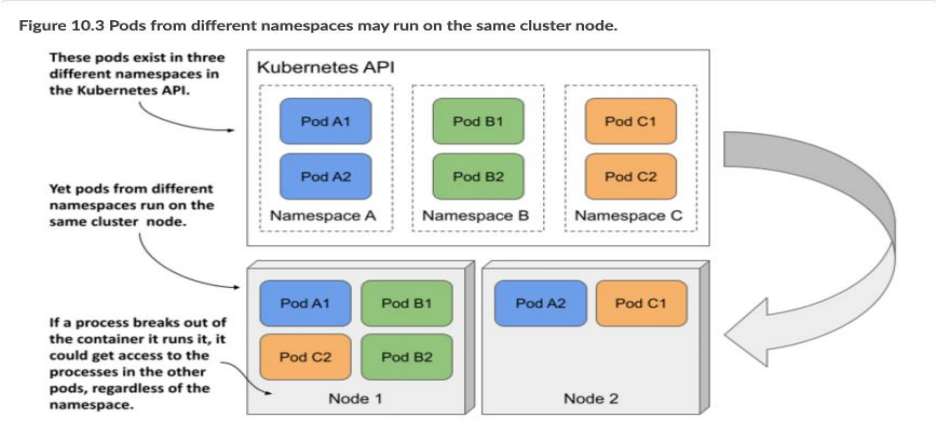
Los Namespaces en Kubernetes son una forma de dividir un clúster en Kubernetes de manera lógica en múltiples clústeres virtuales más pequeños y aislados lógicamente. Cada namespace actúa como entorno de trabajo separado en el que los objetos de Kubernetes, como los pods, los servicios y los volúmenes, pueden ser creados y gestionados.

Algunas de las razones por las que se he decidió usar los Namespaces en Kubernetes son los siguientes:

- Aislamiento y seguridad: los Namespaces nos van a permitir aislar los recursos de Kubernetes en grupos lógicos separados, lo que nos dará una gran mejorar en la seguridad y en la reducción del riesgo de interferencia entre diferentes aplicaciones. Así como facilitar la gestión y el mantenimiento del clúster.
- Escalabilidad: los Namespaces nos permiten escalar los recursos de Kubernetes de forma independiente para diferentes aplicaciones, lo que nos ayuda a mejorar el rendimiento y la capacidad de respuesta.
- Multitenancy: los namespace permiten que varios usuarios o equipos compartan un clúster de Kubernetes de forma segura y aislada, lo que puede ayudar a reducir los costos de infraestructura y simplificar la gestión.



14. Funcionamiento de los espacios de nombre



15. Funcionamiento de los espacios de nombre visto de otra forma

Estos son el espacio de nombres predeterminados:

- Default: el espacio de nombres para los objetos que no tienen otro espacio de nombres.
- Kube-node-lease: este espacio de nombres contiene objetos Lease para cada nodo. Los arrendamientos de nodos permiten que el kubelet envíe latidos al plano de control, lo que le permite detectar fallas en los nodos.
- Kube-public: este espacio de nombres se genera automáticamente y todos los usuarios (incluidos los que no están autenticados) pueden acceder a él. Este espacio de nombres se reserva principalmente para el uso del clúster, en caso de que algunos recursos deban ser visibles públicamente y legibles en todo el clúster. El aspecto público de este espacio de nombres es simplemente una convención, no un requisito.
- Kube-system: el espacio de nombres para los objetos creados por Kubernetes.

```
javierramirez@kubernetes:~$ kubectl get namespaces
NAME                   STATUS   AGE
default                Active   2d19h
kube-node-lease        Active   2d19h
kube-public            Active   2d19h
kube-system            Active   2d19h
kubernetes-dashboard   Active   2d19h
javierramirez@kubernetes:~$
```

Voy a crear un espacio de nombres por cada aplicación que he desplegado.

```
javierramirez@kubernetes:~$ kubectl create namespace mariadb
namespace/mariadb created
javierramirez@kubernetes:~$ kubectl create namespace wordpress
namespace/wordpress created
javierramirez@kubernetes:~$ kubectl get namespaces
NAME                   STATUS   AGE
default                Active   2d19h
kube-node-lease        Active   2d19h
kube-public            Active   2d19h
kube-system            Active   2d19h
kubernetes-dashboard   Active   2d19h
mariadb                Active   14s
wordpress              Active   9s
javierramirez@kubernetes:~$
```

Podemos ver las características de nuestro espacio de nombres y su definición yalm:

```
javierramirez@kubernetes:~$ kubectl describe ns mariadb
Name:         mariadb
Labels:       kubernetes.io/metadata.name=mariadb
Annotations:  <none>
Status:       Active

No resource quota.

No LimitRange resource.
javierramirez@kubernetes:~$ kubectl get ns mariadb -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2023-04-10T14:37:01Z"
  labels:
    kubernetes.io/metadata.name: mariadb
  name: mariadb
  resourceVersion: "26838"
  uid: 047de6a5-bfbb-4cd6-b663-cc55ac3e306d
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
javierramirez@kubernetes:~$
```

Para crear o meter recursos dentro de este espacio de nombres lo haré poniendo su nombre en la etiqueta namespace en la definición de los recursos que quiera.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-deployment
  namespace: mariadb
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mariadb-service
  namespace: mariadb
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mariadb-secret
  namespace: mariadb
```

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f mariadb.yml
deployment.apps/mariadb-deployment created
service/mariadb-service created
persistentvolumeclaim/pvc-mariadb unchanged
javierramirez@kubernetes:~/k8s-configuration$
```

Y ahora vemos el contenido de nuestro namespace:

```
javierramirez@kubernetes:~$ kubectl get deploy -n mariadb
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
mariadb-deployment   1/1     1            1           2m4s
javierramirez@kubernetes:~$ kubectl get secret -n mariadb
NAME             TYPE     DATA   AGE
mariadb-secret   Opaque   2      105s
javierramirez@kubernetes:~$
```

Por último, vamos a limitar el acceso al espacio de nombres de MariaDB mediante RBAC. *El control de acceso basado en roles es un método que nos sirve para regular el acceso a los recursos de nuestro clúster en función de las funciones de los usuarios.*

Un rol establece permisos en un determinado espacio de nombres y debemos especificar a cuál cuando lo creamos. Si queremos establecer un rol en todo nuestro clúster usaríamos ClusterRole, pero nosotros solo vamos a usar rol porque lo queremos para los espacios de nombres.

Vamos con ello. Vamos a agregar un rol a nuestro espacio de nombres de MariaDB que hemos creado anteriormente. Con este nuevo archivo damos permisos de crear, obtener, actualizar, borrar etc. Sobre recursos como los pod, servicios o los secrets.

home > javierramirez > k8s-configuration > ! role.yml

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: mariadb
  name: mariadb-role
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps", "secrets"]
  verbs: ["create", "get", "update", "delete", "list", "watch"]
```

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f role.yml
role.rbac.authorization.k8s.io/mariadb-role created
javierramirez@kubernetes:~/k8s-configuration$
```

Ahora vamos a establecer un RoleBinding, lo que vamos a hacer con ello es asociar un rol a uno o más usuarios o grupos dentro del espacio de nombres. Definiendo quien tiene acceso a que recursos dentro del espacio de nombres. Con este archivo lo que hacemos

es vincularlo al rol que creamos antes, es decir al usuario javierramirez en el namespace MariaDB. Decimos a quien se aplican esos permisos.

home > javierramirez > k8s-configuration > ! rolebinding.yml

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: mariadb-rolebinding
  namespace: mariadb
subjects:
- kind: User
  name: javierramirez
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: mariadb-role
  apiGroup: rbac.authorization.k8s.io
```

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f rolebinding.yml
rolebinding.rbac.authorization.k8s.io/mariadb-rolebinding created
javierramirez@kubernetes:~/k8s-configuration$
```

De esta manera el usuario javierramirez tendrá permisos definidos.

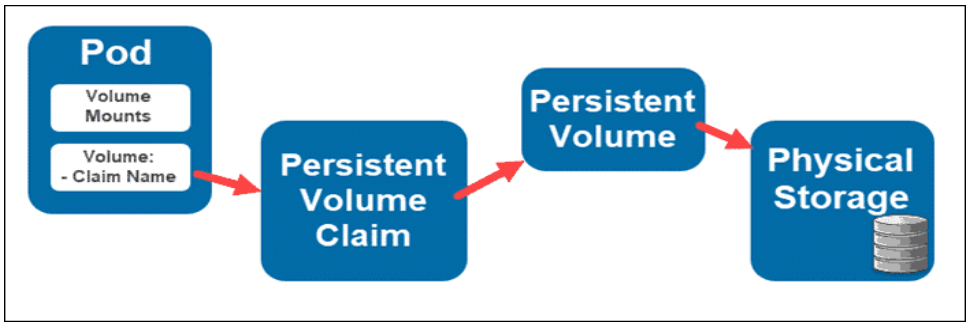
La diferencia entre RoleBinding y Role es que este último en lugar de asociar usuarios se usa para definir los permisos y restricciones de acceso. Roles como el qué y RoleBinding a quién.

Volúmenes persistentes:

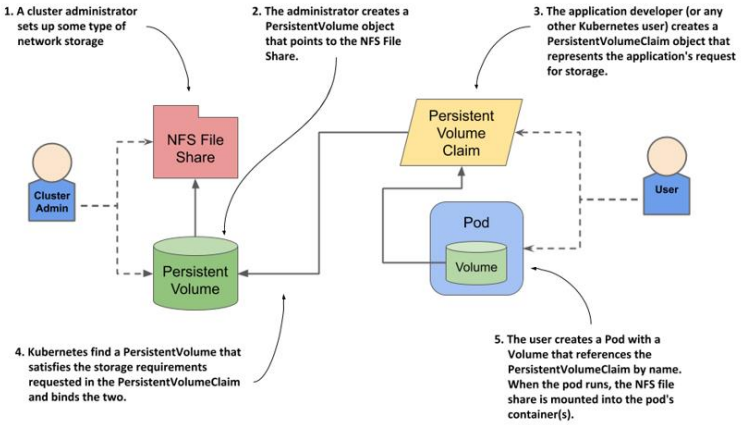
En Kubernetes, los volúmenes persistentes son una forma de almacenar datos de manera persistente en un clúster, de modo que los datos persistan a pesar de que los pods asociados se reinicien o se eliminen. Permiten que los datos sean compartidos entre diferentes pods y también permiten la migración de pods entre nodos en el clúster sin perder datos. También pueden ser utilizados para realizar copias de seguridad de los datos almacenados.

Estos se definen en los archivos YAML que describen los pods y se pueden configurar de varias maneras.

Cuando el pod solicita un volumen persistente, Kubernetes monta el volumen en el contenedor del pod como un sistema de archivos normal, y el pod puede leer y escribir datos en el volumen como si fuera un disco local.



16.Funcionamiento de almacenamiento



17.Funcionamiento de almacenamiento

El objeto de volumen persistente representa una pieza de almacenamiento de datos en un clúster. Puede ser utilizado por uno o varios pods y su ciclo de vida es independiente del ciclo de vida de los pods. *He decidido por tanto usar Volúmenes persistentes para mis aplicaciones ya que, por ejemplo, nosotros tenemos una base de datos que está siendo usada por una aplicación, si el pod se reinicia perderíamos esos datos almacenados. Con un volumen, al no estar relacionado su ciclo de vida con el del pod. El almacenamiento debe de estar disponible para todos los nodos.*

En el modo de acceso elegí es ReadWriteMany (RWX) que permite que el volumen sea montado tanto para lectura como para escritura en múltiples nodos al mismo tiempo. Varios Pods en diferentes nodos pueden acceder y escribir en el volumen de manera concurrente.

En la política de reclamación se define qué hacer con el volumen después de que se rompa el límite de la reclamación de volumen persistente. He elegido que se retenga para que cuando se elimina un PVC asociado a un PV con política de retención, el volumen persistente no se elimina automáticamente y se queda en el clúster. De esta manera, se puede reutilizar el mismo volumen por otro PVC en el futuro, sin perder los datos almacenados en él.

El tipo de volumen le he puesto que sea HostPath. un volumen HostPath permite a los pods acceder a los archivos del sistema de archivos local del nodo en el que se ejecutan. Esto es útil en situaciones en las que se necesita acceso a datos que ya existen en el nodo, como configuraciones o datos de aplicaciones que se mantienen en el nodo.

Con una capacidad de almacenamiento de 5Gi.

home > javierramirez > k8s-configuration > ! pv-mariadb.yml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-mariadb
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  hostPath:
    path: /home/prueba
```

home > javierramirez > k8s-configuration > ! pv-wordpress.yml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-wordpress
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  hostPath:
    path: /home/prueba
```

Un PersistentVolumeClaim (PVC) es un objeto que solicita un volumen persistente y define sus requisitos de almacenamiento, como el tamaño y el tipo de acceso. Un PVC es creado por un usuario o por una aplicación y luego se utiliza por los pods para acceder al almacenamiento persistente. Kubernetes gestiona la asignación de un volumen persistente a un PVC y asegura que un PVC no pueda ser asignado a más de un volumen persistente a la vez. Este lo he añadido al archivo yaml de MariaDB y al de Wordpress.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-mariadb
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: local-storage
```

```yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-wordpress
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: local-storage
```

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f pv-mariadb.yml
persistentvolume/pv-mariadb created
```

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f mariadb.yml
deployment.apps/mariadb-deployment unchanged
service/mariadb-service unchanged
persistentvolumeclaim/pvc-mariadb unchanged
```

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f pv-wordpress.yml
persistentvolume/pv-wordpress created
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f wordpress.yml
deployment.apps/wordpress unchanged
service/wordpress-service unchanged
persistentvolumeclaim/pvc-wordpress created
javierramirez@kubernetes:~/k8s-configuration$
```

Vemos que los Volúmenes persistentes estén creados correctamente.

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                     STORAGECLASS    REASON   AGE
pv-mariadb                                 5Gi        RWO            Retain           Bound    default/pvc-mariadb       local-storage            31h
pv-wordpress                               5Gi        RWO            Retain           Bound    default/pvc-wordpress     local-storage            3m31s
```

Vemos que los volúmenes persistentes claims estén creados correctamente:

```
javierramirez@kubernetes:~/k8s-configuration$ kubectl get pvc
NAME            STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS    AGE
pvc-mariadb     Bound    pv-mariadb                                 5Gi        RWO            local-storage   31h
pvc-wordpress   Bound    pv-wordpress                               5Gi        RWO            local-storage   4m40s
```

Creamos el pod tanto para MariaDB como para Wordpress para que los usen. VolumeMounts hace referencia a donde se van a montar, y en volumes le indicamos el path a través del nombre que hemos definido anteriormente.

pv-mariadb.yml | pv-wordpress.yml | volumenpod.yml | mariadb.yml

home > javierramirez > k8s-configuration > volumenpod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mariadb-pod
spec:
  containers:
  - name: mariadb
    image: mariadb
    volumeMounts:
    - mountPath: /home/prueba
      name: pv-mariadb
  volumes:
  - name: pv-mariadb
    persistentVolumeClaim:
      claimName: pvc-mariadb
```

Gestión de Recursos:

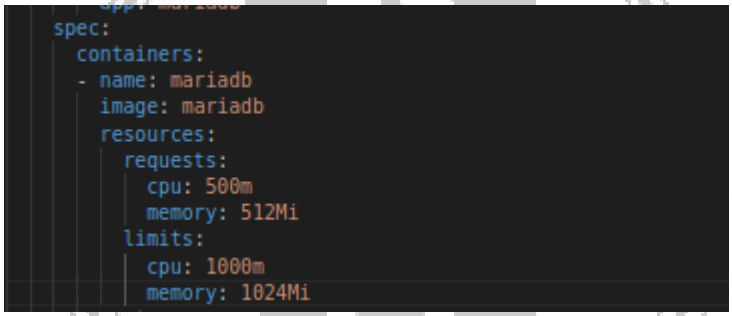
Podemos limitar y establecer los recursos que un contenedor puede utilizar a través de los recursos delimite donde especificamos el límite máximo de recursos que un contenedor puede utilizar. Los recursos que se pueden limitar incluyen la CPU o la memoria.

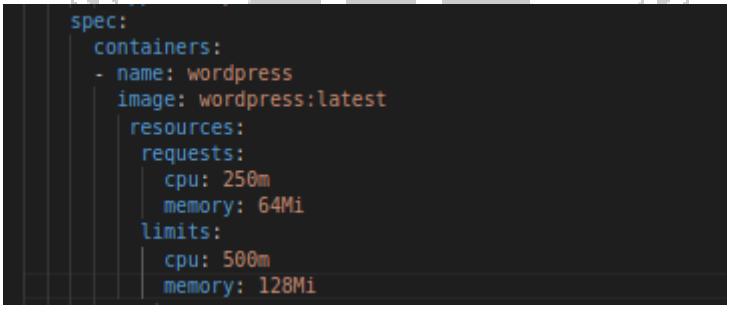
Las solicitudes de recursos las usamos para indicar la cantidad de recursos que se necesitan para que un contenedor se ejecute de manera efectiva. Con ellas planificamos la distribución de los contenedores en los nodos del clúster, si esta solicitud no se puede satisfacer en un nodo, Kubernetes programara el contenedor en ese nodo.

Por otro lado, los límites de recursos se utilizan para garantizar que un contenedor no utilice más recursos de los que se han asignado. Si un contenedor supera su límite de recurso, Kubernetes tomará medidas para reducir su consumo de recursos.

Es importante tener en cuenta que las solicitudes y límites de recursos pueden afectar el rendimiento y la disponibilidad de las aplicaciones. Es recomendable establecerlos adecuadamente en función de las necesidades de la aplicación y del clúster.

- .spec.spec.resources se especifica los recursos de proceso necesarios para el contenedor.
- .spec.spec.resources.requests se especifica la cantidad mínima de recursos de proceso necesarios.
- .spec.spec.resources.requests.cpu se especifica la cantidad mínima de CPU necesaria.
- spec.spec.resources.requests.memory se especifica la cantidad mínima de memoria necesaria.
- .spec.spec.resources.limits se especifica la cantidad máxima de recursos de proceso permitidos. **El kubelet aplica este límite.**
- **.**spec.spec.resources.limits.cpu se especifica la cantidad máxima de CPU permitida. **El kubelet aplica este límite.**
- .spec.spec.resources.limits.memory se especifica la cantidad máxima de memoria permitida. **El kubelet aplica este límite**






```
javierramirez@kubernetes:~/k8s-configuration$ kubectl apply -f mariadb.yml
deployment.apps/mariadb-deployment configured
service/mariadb-service unchanged
```

Dashboard:

Es una interfaz gráfica de usuario (GUI) que permite a los usuarios gestionar y monitorear los clústeres de Kubernetes. Proporciona una visión general de alto nivel del estado del clúster y sus recursos, como nodos, servicios, volúmenes y pods.

Nos sirve para ver de manera general todo lo que hemos hecho hasta el momento en comparación con el principio que apenas teníamos lo más básico en nuestro clúster.

```
javierramirez@kubernetes:~$ minikube dashboard
    Executing "docker container inspect minikube --format={{.State.Status}}" took an unusually long time: 2.266798956s
    Restarting the docker service may improve performance.
    Verifying dashboard health ...
    Launching proxy ...
    Verifying proxy health ...
    Opening http://127.0.0.1:44223/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in your default browser...
```