Almacenamiento en Kubernetes

Almacenamiento en Kubernetes

Ya lo hemos comentando en anteriores módulos, pero es importante tener en cuenta que **los Pods son efímeros**, es decir, cuando un Pod se elimina se pierde toda la información que tenía. Evidentemente, cuando creamos un nuevo Pod no contendrá ninguna información adicional a la propia aplicación.

Podemos fijarnos en el Ejemplo: Despliegue y acceso a Wordpress + MariaDB (que vestudiamos en el módulo anterior), y responder las siguientes preguntas:

1. ¿Qué pasa si eliminamos el despliegue de mariadb?, o, ¿se elimina el Pod de mariadb y se crea uno nuevo?

   Evidentemente, toda la información guardada en la base de datos se perderá, por lo que al iniciar un nuevo despliegue, no tendremos información guardada, habremos perdido todo el contenido de nuestra aplicación y empezaría de nuevo el proceso de instalación.

2. ¿Qué pasa si escalamos el despliegue de la base de datos y tenemos dos Pods ofreciendo la base de datos?

   En este caso el Pod más antiguo tendría la información de la base de datos, pero el nuevo Pod creado al escalar el despliegue no tendría ninguna información. Como al acceder al Service de la base de datos se hace balanceo de carga, en unas ocasiones accederíamos al Pod antiguo, y todo funcionaría correctamente, pero cuando accederíamos al Pod nuevo, al no tener información, nos mostraría la pantalla de instalación de la aplicación. En definitiva, tendríamos dos bases de datos distintas a las que accederíamos indistintamente.

3. Si escribimos un post en el Wordpress y subimos una imagen, ¿qué pasa con esta información en el Pod?

   Está claro que cuando escribimos un post esa información se guarda en la base de datos. Pero la imagen que hemos subido al post se guardaría en un directorio del servidor web

(del Pod de Wordpress). Tendríamos los mismos problemas que con la base de datos, si eliminamos este Pod se perderá todo el contenido estático de nuestro Wordpress.

4. ¿Qué pasa si escalamos el despliegue de Wordpress a dos Pods?

   Pues la respuesta es similar a la anterior. En este caso, el Pod antiguo tendría almacenada el contenido estático (la imagen), pero el nuevo no tendría esa información. Como al acceder a la aplicación se balancea la carga, se mostraría la imagen diferente en función del Pod que estuviéramos accediendo.

Por lo tanto, es necesario usar un mecanismo que nos permita guardar la información con la que trabajan los Pods para que no se pierda en caso de que el Pod se elimine. Al sistema de almacenamiento persistente que nos ofrece Kubernetes lo llamamos **volúmenes**. Con el uso de dichos volúmenes vamos a conseguir varias cosas:

1. Si un Pod guarda su información en un volumen, está no se perderá. Por lo que podemos eliminar el Pod sin ningún problema y cuando volvamos a crearlo mantendrá la misma información. En definitiva, los volúmenes proporcionan almacenamiento adicional o secundario al disco que define la imagen.
2. Si usamos volúmenes, y tenemos varios Pods que están ofreciendo un servicio, estos Pods tendrán la información compartida y por tanto todos podrán leer y escribir la misma información.
3. También podemos usar los volúmenes dentro de un Pod, para que los contenedores que forman parte de él puedan compartir información.

Por último, indicar que vamos a tener a nuestra disposición distintos tipos de volúmenes para usar. Hay que tener en cuenta que si nuestro cluster tiene varios nodos y los Pods de una aplicación se reparten por estos nodos, necesitamos sistemas de almacenamiento que nos posibiliten compartir la información entre los nodos del cluster. Como en este curso estamos usando **minikube**, nuestro cluster tiene un solo nodo, por lo que vamos a usar un tipo de almacenamiento que permita compartir la información dentro de este nodo (por ejemplo un directorio en el sistema de archivos del nodo), por lo tanto este tema lo vamos a simplificar al no tener la posibilidad de tener un cluster con varios nodos.

Vídeo: Almacenamiento en Kubernetes

https://www.youtube.com/embed/30VOZd4U5VU

Vídeo: Almacenamiento en Kubernetes

Volúmenes en Kubernetes

Tipos de volúmenes

Los volúmenes nos permiten proporcional almacenamiento a los Pods, y podemos usar distintos tipos que nos ofrecen distintas características:

- Proporcionados por proveedores de cloud: AWS, Azure, GCE, OpenStack, etc
- Propios de Kubernetes:
  - configMap: Para usar un configMap como un directorio desde el Pod.
  - emptyDir: Volumen efímero con la misma vida que el Pod. Usado como almacenamiento secundario o para compartir entre contenedores del mismo Pod.
  - hostPath: Monta un directorio del host en el Pod (usado excepcionalmente, pero es el que nosotros vamos a usar con minikube).
  - ...
- Habituales en despliegues "on premises": glusterfs, cephfs, iscsi, nfs, etc.

Trabajando con volúmenes

Al trabajar con volúmenes en Kubernetes se realizan dos funciones claramente diferenciadas:

- **Desde el punto de vista del administrador del cluster de Kubernetes:**

  El administrador es el responsable de la gestión del almacenamiento en el clúster de k8s. Proporciona almacenamiento a las aplicaciones, entrando a detalle en configurar los diferentes mecanismo, bien proporcionados por el proveedor de cloud o configurados directamente. Para ello gestiona los recursos del cluster llamados PersistentVolume o StorageClasses. Ejemplos:

  - Configurar Azure Disk para que pueda usarlo k8s.
  - Configurar Cephfs o RBD en la red local para usarlo en k8s.

- **Desde el punto de vista del desarrollador de aplicaciones que van a ser ejecutadas en el cluster de Kubernetes:**

  A los desarrolladores de aplicaciones les interesa más la disponibilidad y las características del almacenamiento que los detalles sobre el mecanismo de

almacenamiento. Para solicitar almacenamiento se va a utilizar el recurso del cluster PersistentVolumeClaim. Ejemplos:

- Quiero 20 GiB de almacenamiento permanente que pueda compartir entre varios Pods de varios nodos en modo lectura.
- Quiero 10 GiB de almacenamiento provisional para usar desde un Pod en modo lectura y escritura.

Vídeo: Volúmenes en Kubernetes

https://www.youtube.com/embed/g1Elyt_OuqA

Vídeo: Volúmenes en Kubernetes

Aprovisionamiento de volúmenes

Para que el administrador de Kubernetes defina los volúmenes disponibles en nuestro cluster tenemos dos posibilidades:

Aprovisionamiento estático

En este caso, es el administrador del cluster el responsable de ir definiendo los distintos volúmenes disponibles en el cluster creando manualmente los distintos recursos PersistentVolumen (PV).

Un PersistentVolumen es un objeto que representa los volúmenes disponibles en el cluster. En él se van a definir los detalles del backend de almacenamiento que vamos a utilizar, el tamaño disponible, los modos de acceso, las políticas de reciclaje, etc.

Tenemos tres modos de acceso, que dependen del backend que vamos a utilizar:

- ReadWriteOnce: read-write solo para un nodo (RWO)
- ReadOnlyMany: read-only para muchos nodos (ROX)
- ReadWriteMany: read-write para muchos nodos (RWX)

Las políticas de reciclaje de volúmenes también dependen del backend y son:

- Retain: El PV no se elimina, aunque el PVC se elimine. El administrador debe borrar el contenido para la próxima asociación.
- Recycle: Reutilizar contenido. Se elimina el contenido y el volumen es de nuevo utilizable.
- Delete: Se borra después de su utilización.

A modo de resumen, ponemos en la siguiente tabla los modos de acceso de algunos de los sistemas de almacenamiento más usados:

Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS	✓	-	-
AzureFile	✓	✓	✓

Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
NFS	✓	✓	✓
RBD	✓	✓	-

Por último, vemos un ejemplo de un fichero yaml que nos permite la definición de un *PersitentVolumen*:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /data/pv1
```

- `storageClassName: manual`: Indica que este volumen se puede asignar de forma estática, sin utilizar ningún "aprovisonador" de almacenamiento.
- Se indica al tamaño del volumen, con `capacity`, `storage`.
- `accessModes`: El modo de acceso.
- `persistentVolumeReclaimPolicy`: La política de reciclaje.
- Y por útimo se indica el tipo (backend) de almacenamiento, en este caso es de tipo `hostPath` que creará un directorio (`/data/pv1`) en el nodo para guardar la información.

Aprovisionamiento dinámico

Cuando el desarrollador necesita almacenamiento para su aplicación, hace una petición de almacenamiento creando un recurso PersistentVolumenClaim (PVC) y de forma dinámica se

crea el recurso PersistentVolume que representa el volumen y se asocia con esa petición. De otra forma explicado, cada vez que se cree un PersistentVolumenClaim, se creará bajo demanda un PersistentVolumen que se ajuste a las características seleccionadas.

Para conseguir la gestión dinámica de volúmenes, necesitamos un "aprovisionador" de almacenamiento (tendremos distintos aprovisionadores para los distintos tipos de almacenamiento).

Para definir los "aprovisionadores" de almacenamiento, usaremos el objeto *StorageClass*. En Minikube, por defecto, ya tenemos un provisionador para almacenamiento del tipo *hostPath* (monta un directorio del host en el pod).

```
kubectl get storageclass
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate
```

En este caso la configuración del objeto `storageclass` se definió con las siguientes características:

- La política de reciclaje tiene el valor `Delete`.
- Y el modo de asociación (`VOLUMEBINDINGMODE`) tiene el valor `Immediate`, es decir, cuando se cree el objeto PersistenVolumenClaim se asociará de forma dinámica un volumen (objeto PersistenVolumen) inmediatamente. Otro valor podría ser `WaitForFirstConsumer`, en ese caso la asociación se haría cuando se utilizará el volumen.

Vídeo: Aprovisionamiento de volúmenes

https://www.youtube.com/embed/7D9R0_f60-Q

Vídeo: Aprovisionamiento de volúmenes

Solicitud de volúmenes

Independientemente de cómo haya aprovisionado el almacenamiento el administrador del cluster (de forma dinámica o de forma estática), el desarrollador debe hacer una **solicitud de almacenamiento**, indicando las características del volumen que necesita.

Un desarrollador no necesita conocer los distintos tipos de volúmenes disponibles en el cluster. ¡Son detalles muy específicos!

Un desarrollador se centra en indicar los requerimientos que debe tener el volumen que necesita:

- Tamaño.
- Tipo de acceso (sólo lectura o lectura / escritura).
- Tipo de volumen (sólo si es importante).

Para hacer la solicitud de un volumen, el desarrollador debe crear un recurso en el cluster llamado *PersitentVolumenClaim*, veamos un ejemplo:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: pvc1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

En esta solicitud el desarrollador indica los requisitos que necesita para su almacenamiento:

- `storageClassName: manual`: Con esto indicamos que no se use ningún aprovisionador dinámico. Si no pongo esta línea se intentaran asociar un volumen de forma dinámica.
- `accessModes`: El tipo de acceso que necesita.
- Y el tamaño que necesita en `resources`, `requests`, `storage`.

Una vez que se crea este recurso, el cluster intentará asignar un volumen (ya sea de forma estática o dinámica) que cumpla con los requisitos indicados.

Vídeo: Solicitud de volúmenes

https://www.youtube.com/embed/YV21W_hjo0Q

Vídeo: Solicitud de volúmenes

Uso de volúmenes

Una vez que hemos solicitado el almacenamiento, y se ha asignado un volumen (ya sea de forma dinámica o estática), vamos a definir el uso que se va a hacer de este volumen.

Como ejemplo, vamos a definir un Pod que utilice dicho volumen, para ello vamos a crear un fichero yaml con la siguiente definición:

```
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: pvc1
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

- En la especificación de este Pod, además de indicar el contenedor, hemos indicado que va a tener un volumen (campo `volumes`).
- En realidad definimos una lista de volúmenes (en este caso solo definimos uno) indicando su nombre (`name`) y la solicitud del volumen (`persistentVolumeClaim`, `claimName`).
- Además en la definición del contenedor tendremos que indicar el punto de montaje del volumen (`volumeMounts`) señalando el directorio del contenedor (`mountPath`) y el nombre (`name`).

Cuando el Pod termina, el pvc mantiene el volumen reservado (bound). Es necesario que se borre explícitamente el pvc para liberarlo.

La recuperación del volumen dependerá de la política de reciclaje que tuviera asignada.

Ejemplo 1: Gestión estática de volúmenes

En este ejemplo vamos a desplegar un servidor web que va a servir una página html que tendrá almacenada en un volumen. En este primer ejemplo, la asignación del volumen se va a realizar de forma estática.

Aprovisonamiento del volumen

En este caso, será el administrador del cluster el responsable de dar de alta en el cluster los volúmenes disponibles. Como hemos estudiado anteriormente, indicaremos algunas características del volumen: la capacidad, el modo de acceso, la política de reciclaje, el tipo de volumen,...

Para ello vamos a describir el objeto PersistentVolume en el fichero `pv-ejemplo1.yaml`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-ejemplo1
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /data/pv-ejemplo1
```

Nota

Nota: como estamos utilizando minikube, y nuestro cluster está formado por un sólo nodo, el tipo de almacenamiento más simple que podemos usar es `hostPath`, que creará un directorio en el nodo (`/data/pv-ejemplo1`) que será el que se monte en el Pod para guardar la información.

El administrador crea el volumen:

```
kubectl apply -f pv-ejemplo1.yaml
```

Podemos ver los volúmenes que tenemos disponibles en el cluster:

```
kubectl get pv
NAME                          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
persistentvolume/pv-ejemplo1   5Gi        RWX            Recycle          Available
```

Nos fijamos que el estado del volumen es `Available`, todavía no se ha asociado con ninguna solicitud de volumen.

Y podemos obtener los detalle de este recurso:

```
kubectl describe pv pv-ejemplo1
```

Solicitud del volumen

A continuación, nosotros como desarrolladores necesitamos solicitar un volumen con ciertas características para nuestra aplicación, para ello vamos a definir un objeto PersistentVolumeClaim, que definiremos en el fichero `pvc-ejemplo1.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: pvc-ejemplo1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Como vemos, desde el punto de vista del desarrollador no se necesita saber los tipos de volúmenes que tenemos disponibles, simplemente indicamos que queremos un 1Gb de almacenamiento, el tipo de acceso y que se haga la asignación de forma estática (`storageClassName: manual`).

Cuando creemos el objeto PersistentVolumeClaim podremos comprobar si hay algún volumen (PersistentVolume) disponible en el cluster que cumpla con los requisitos:

```
kubectl apply -f pvc-ejemplo1.yaml
```

```
kubectl get pv,pvc
NAME                            CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
persistentvolume/pv-ejemplo1    5Gi        RWX            Recycle          Bound

NAME                                    STATUS   VOLUME        CAPACITY   ACCESS MODES
persistentvolumeclaim/pvc-ejemplo1      Bound    pv-ejemplo1   5Gi        RWX
```

Podemos apreciar que el el estado del volumen ha cambiado a `Bound` que significa que ya está asociado al PersistentVolumeClaim que hemos creado.

Nota

Nota: El desarrollador quería 1 Gb de disco, demanda que se cumple de sobra con los 5 Gb del volumen que se ha asociado.

Uso del volumen

Una vez que tenemos un volumen a nuestra disposición, vamos a crear un despliegue de un servidor web, indicando en la especificación del Pod, que estará formado por el volumen y el directorio donde vamos a montarlo. Para ello vamos a usar el fichero `deploy-ejemplo1.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ejemplo1
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
        - name: volumen-ejemplo1
          persistentVolumeClaim:
```

```
            claimName: pvc-ejemplo1
      containers:
        - name: contenedor-nginx
          image: nginx
          ports:
            - name: http-server
              containerPort: 80
          volumeMounts:
            - mountPath: "/usr/share/nginx/html"
              name: volumen-ejemplo1
```

Podemos observar que en la especificación del Pod hemos indicado que estará formado por un volumen correspondiente al asignado al PersistentVolumeClaim `pvc-ejemplo1` y que el contenedor tendrá en el volumen un punto de montaje en el directorio *DocumentRoot* de nginx (`/usr/share/nginx/html`) .

Creamos el Deployment:

```
kubectl apply -f deploy-ejemplo1.yaml
```

Y a continuación, cuando el contenedor esté funcionando:

```
kubectl get all
...
NAME                                  READY   STATUS    RESTARTS   AGE
pod/nginx-ejemplo1-86864d84b5-s62dq   1/1     Running   0          6s
...
```

Vamos a ejecutar un comando en el Pod para que se cree un fichero `index.html` en el directorio `/usr/share/nginx/html` (evidentemente estaremos guardando ese fichero en el volumen).

```
kubectl exec pod/nginx-ejemplo1-86864d84b5-s62dq -- bash -c "echo '<h1>Almacenamier
```

Finalmente creamos el Service de acceso al despliegue, usando el fichero srv-ejemplo1.yaml.

```
kubectl apply -f srv-ejemplo1.yaml

kubectl get all
...
service/nginx-ejemplo1   NodePort    10.106.238.146   <none>        80:32581/TCP
...
```

Y accedemos a la aplicación, accediendo a la ip del nodo controlador del cluster y al puerto asignado al Service NodePort:

```
minikube ip
192.168.39.222
```

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

Comprobemos la persistencia de la información

En primer lugar podemos acceder al nodo del cluster y comprobar que en el directorio que indicamos en la creación del volumen, efectivamente existe el fichero `index.html`:

```
minikube ssh
ls /data/pv-ejemplo1
index.html
```

En segundo lugar podemos hacer la prueba de eliminar el despliegue, volver a crearlo y volver a acceder a la aplicación para comprobar que el servidor web sigue sirviendo el mismo fichero `index.html`:

```
kubectl delete -f deploy-ejemplo1.yaml
kubectl apply -f deploy-ejemplo1.yaml
```

Y volvemos acceder al mismo puerto:

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

Eliminación del volumen

Si finalmente queremos eliminar los volúmenes creados, tendremos que eliminar la solicitud, el objeto PersistentVolumeClaim, y dependiendo de la política de reciclaje con la que creamos el objeto PersistentVolume tendremos distintos comportamientos.

En este caso, como la política de reciclaje con la que creamos el volumen es `Recycle`, no se eliminará pero se borrará su contenido y el volumen se podrá reutilizar, es decir su estado volverá a `Available`:

```
kubectl delete persistentvolumeclaim/pvc-ejemplo1
```

```
kubectl get pv,pvc
NAME                           CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
persistentvolume/pv-ejemplo1   5Gi        RWX            Recycle          Available
```

Si queremos eliminar el objeto PersistentVolume, ejecutamos:

```
kubectl delete persistentvolume/pv-ejemplo1
```

Vídeo: Ejemplo 1: Gestión estática de volúmenes

https://www.youtube.com/embed/z3DOCCjRnSY

Vídeo: Ejemplo 1: Gestión estática de volúmenes

Ejemplo 2: Gestión dinámica de volúmenes

En este ejemplo vamos a desplegar un servidor web que va a servir una página html que tendrá almacenada en un volumen. En esta ocasión, la asignación del volumen se va a realizar de forma dinámica.

Aprovisonamiento del volumen

Para que la asignación del volumen (objeto *PersitentVolume*) se haga de forma dinámica al crear la solicitud (objeto *PersitentVolumeClaim*) es necesario tener configurado un aprovisonador de almacenamiento que se define en un objeto `StorageClass`. Como vimos en minikube tenemos configurado un aprovisonador para volúmenes de tipo *hostPath*:

```
kubectl get storageclass
NAME                 PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE
standard (default)   k8s.io/minikube-hostpath   Delete          Immediate
```

De tal manera que cuando creemos una solicitud de volumen (objeto *PersitentVolumeClaim*) se creara de forma dinámica un objeto *PersitentVolume*, que se asociaría a la solicitud.

Solicitud del volumen

Vamos a realizar la solicitud de volumen, en este caso usaremos el fichero `pvc-ejemplo2.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: pvc-ejemplo2
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Nota

Nota: Fíjate que esta definición hemos quitado la declaración `storageClassName: manual`. Al no ponerla se elegirá el `storageclass` por defecto, cuya definición hemos visto anteriormente en minikube y que en este caso se llama `standard`.

Cuando creemos el objeto PersistentVolumeClaim, veremos que de forma dinámica se creará un *PersitentVolumen* que se asociará a nuestra solicitud::

```
kubectl apply -f pvc-ejemplo2.yaml

kubectl get pv,pvc
NAME                                                        CAPACITY   ACCESS MODES
persistentvolume/pvc-6a09c69a-4344-447c-b23d-d85c7edd7f36   1Gi        RWX

NAME                                    STATUS   VOLUME
persistentvolumeclaim/pvc-ejemplo2   Bound    pvc-6a09c69a-4344-447c-b23d-d85c7edd7
```

Nota

Nota: En este caso, como el volumen se ha generado dinámicamente, su capacidad es igual a la solicitada, 1 Gb.

Como el volumen ha sido generado de forma dinámica por el aprovisonador, éste habrá escogido una carpeta del host que corresponda al volumen.

```
kubectl describe persistentvolume/pvc-6a09c69a-4344-447c-b23d-d85c7edd7f36
...
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /tmp/hostpath-provisioner/default/pvc-ejemplo2
...
```

Uso del volumen

A partir de este punto el ejercicio es muy parecido al que vimos en el ejemplo1.

Creamos el Deployment usando el fichero `deploy-ejemplo2.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ejemplo2
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
        - name: volumen-ejemplo2
          persistentVolumeClaim:
            claimName: pvc-ejemplo2
      containers:
        - name: contenedor-nginx
          image: nginx
          ports:
            - name: http-server
              containerPort: 80
          volumeMounts:
            - mountPath: "/usr/share/nginx/html"
              name: volumen-ejemplo2
```

Creamos el Deployment:

```
kubectl apply -f deploy-ejemplo2.yaml
```

Y a continuación, cuando el contenedor esté funcionando, creamos el fichero `index.html`:

```
kubectl get all
...
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-ejemplo2-7b79b5966-zbdqh   1/1     Running   0          5s
...
kubectl exec pod/nginx-ejemplo2-7b79b5966-zbdqh -- bash -c "echo '<h1>Almacenamient
```

Finalmente creamos el Service de acceso al despliegue, usando el fichero srv-ejemplo2.yaml.

```
kubectl apply -f srv-ejemplo2.yaml

kubectl get all
```

```
...
service/nginx-ejemplo2   NodePort    10.99.48.24   <none>        80:31053/TCP   3s
...
```

Y accedemos a la aplicación accediendo a la ip del nodo controlador del cluster y al puerto asignado al Service NodePort:

```
minikube ip
192.168.39.222
```

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

Finalmente puedes volver a comprobar que la información de la aplicación no se pierde borrando el Deployment y volviéndolo a crear, comprobando que se sigue sirviendo el fichero `index.html`.

Eliminación del volumen

En este caso, los volúmenes que crea de forma dinámica el `storageclass` que tenemos creado en minikube, tienen como política de reciclaje el valor de `Delete`. Esto significa que cuando eliminemos la solicitud, el objeto PersistentVolumeClaim, también se borrará el volumen, el objeto PersistentVolume.

```
kubectl delete persistentvolumeclaim/pvc-ejemplo2
```

```
kubectl get pv,pvc
No resources found
```

Vídeo: Ejemplo 2: Gestión dinámica de volúmenes

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

Ejemplo 2: Gestión dinámica de volúmenes

Ejemplo 3: Wordpress con almacenamiento persistente

En este ejemplo vamos a volver e realizar el Ejemplo completo: Despliegue y acceso a Wordpress + MariaDB del módulo anterior, pero añadiendo el almacenamiento necesario para que la aplicación sea persistente.

Para llevar a cabo esta tarea necesitaremos tener a nuestra disposición dos volúmenes:

- Uno para guardar la información de Wordpress.
- Otro para guardar la información de MariaDB.

Para este ejercicio utilizaremos asignación dinámica de volúmenes.

Creación de los volúmenes necesarios

Como hemos comentado vamos a usar la asignación dinámica de volúmenes, por lo tanto tendremos que crear dos objetos PersistentVolumenClaim para solicitar los dos volúmenes.

Para solicitar el volumen para la aplicación Wordpress usaremos el fichero `wordpress-pvc.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: wordpress-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Y para solicitar el volumen para la base de datos usaremos un fichero similar: `mariadb-pvc.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: mariadb-pvc
spec:
```

```
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Creamos las solicitudes y comprobamos que se ha asociado un volumen a cada una de ellas:

```
kubectl apply -f wordpress-pvc.yaml
kubectl apply -f mariadb-pvc.yaml

kubectl get pv,pvc
NAME                                                        CAPACITY   ACCESS MODES
persistentvolume/pvc-01ed3c4c-a542-4161-93a9-b9d5ea2bf6d1   5Gi        RWX
persistentvolume/pvc-78acc14b-71da-4cf0-861d-0ab7780bca4f   5Gi        RWX

NAME                                     STATUS   VOLUME
persistentvolumeclaim/mariadb-pvc        Bound    pvc-78acc14b-71da-4cf0-861d-0ab7780b
persistentvolumeclaim/wordpress-pvc      Bound    pvc-01ed3c4c-a542-4161-93a9-b9d5ea2b
```

Modificación de los Deployments para el uso de los volúmenes

A continuación vamos a modificar el fichero `wordpress-deployment.yaml` para añadir el volumen al Pod y el punto de montaje:

```
...
    spec:
      containers:
      ...
          volumeMounts:
            - name: wordpress-vol
              mountPath: /var/www/html
      volumes:
        - name: wordpress-vol
          persistentVolumeClaim:
            claimName: wordpress-pvc
```

Como observamos vamos a usar el volumen asociado al PersistentVolumenClaim `wordpress-pvc` y que lo vamos a montar en el directorio *DocumentRoot* del servidor web: `/var/www/html`.

De forma similar, modificamos el fichero `mariadb-deployment.yaml`:

```
...
    spec:
```

```
      containers:
    ...
          volumeMounts:
            - name: mariadb-vol
              mountPath: /var/lib/mysql
      volumes:
        - name: mariadb-vol
          persistentVolumeClaim:
            claimName: mariadb-pvc
```

En esta ocasión usaremos el volumen asociado a `mariadb-pvc` y el punto de montaje se hará sobre el directorio donde se guarda la información de la base de datos: `/var/lib/mysql`.

Evidentemente, no es necesario modificar la definición de los otros recursos: Services e Ingress.

Creamos el Deployment, los Services y el Ingress:

- mariadb-srv.yaml
- wordpress-srv.yaml
- wordpress-ingress.yaml

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

Acedemos a la aplicación y la configuramos:

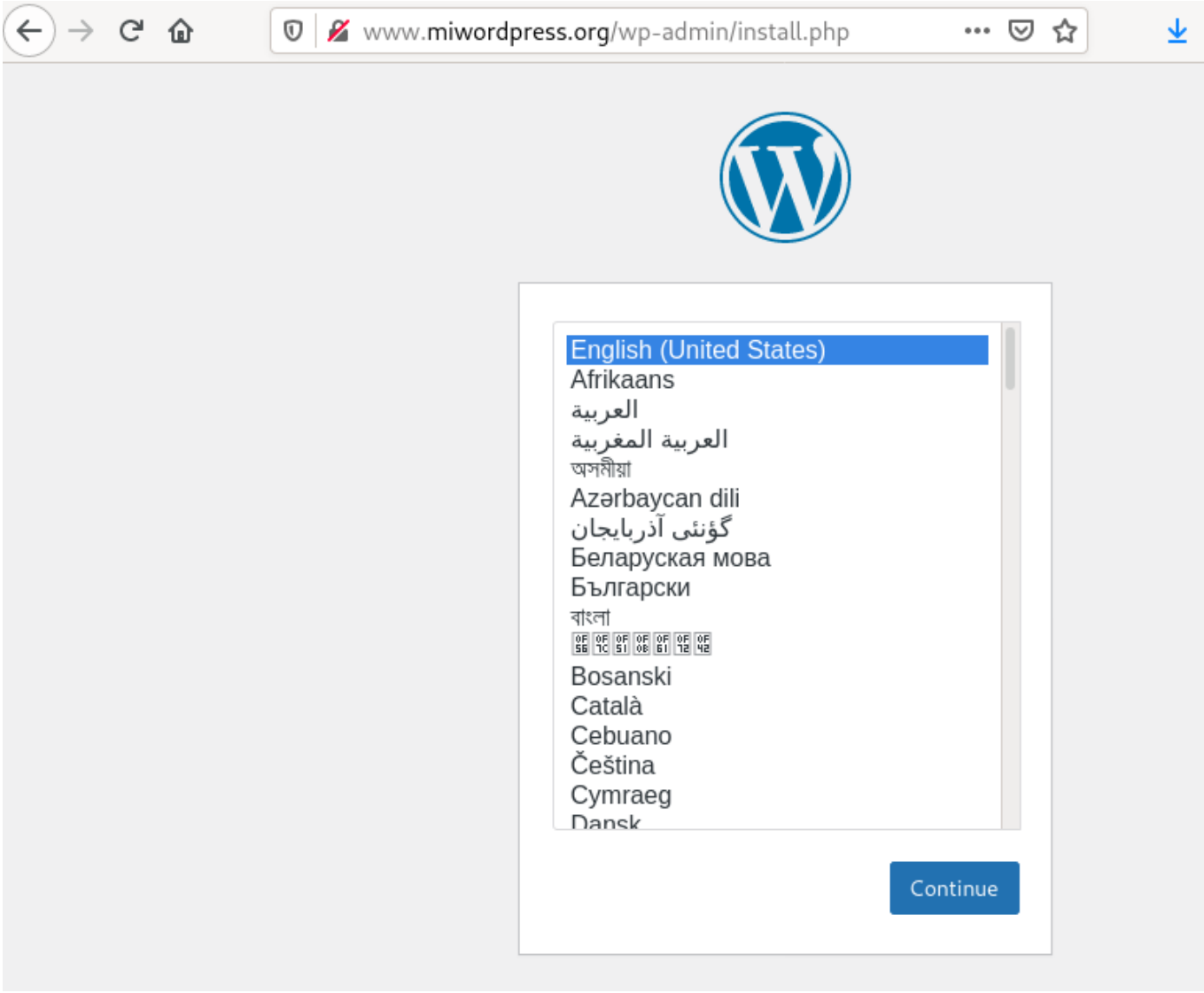


Imagen de elaboración propia (CC BY-NC-SA)

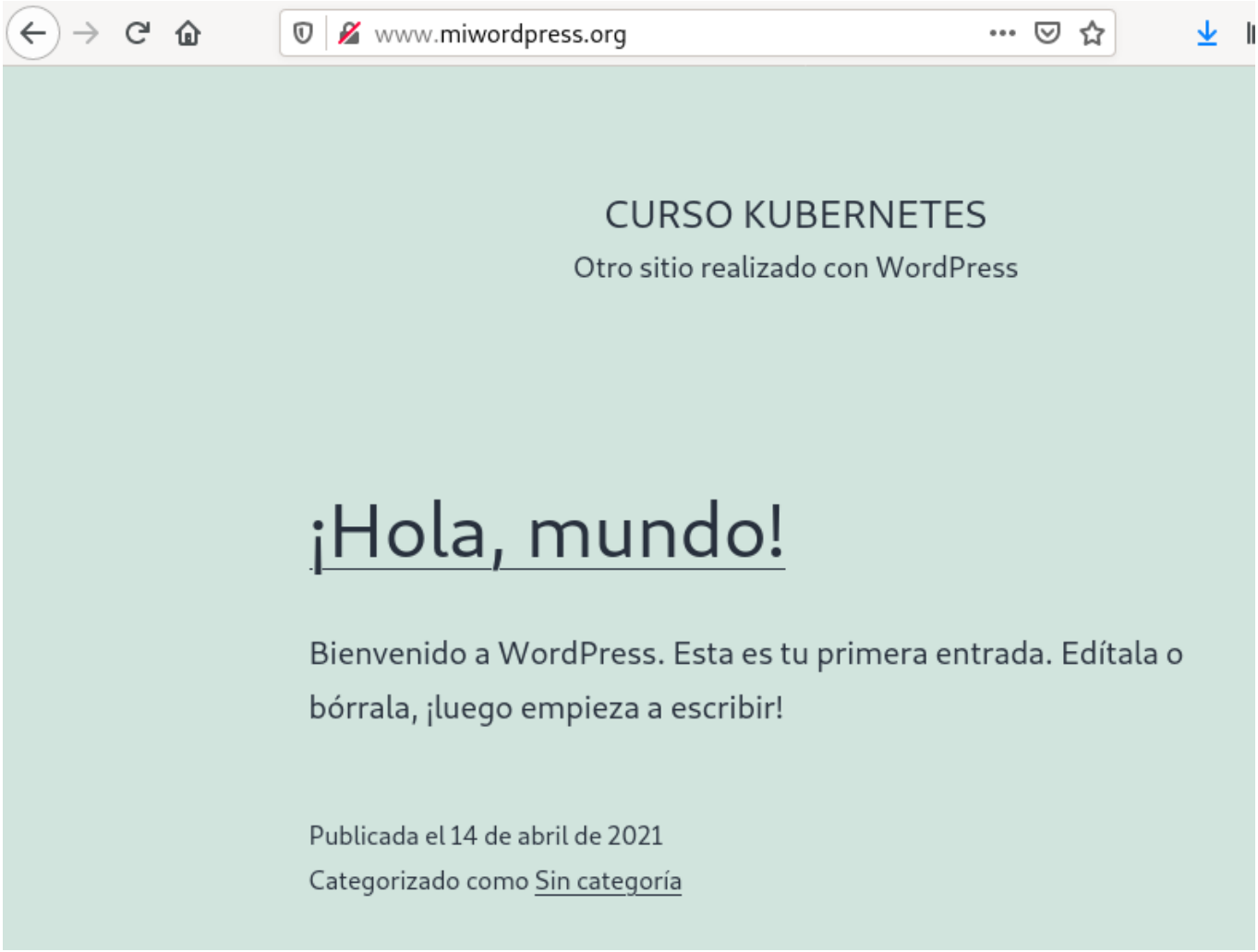


Imagen de elaboración propia (CC BY-NC-SA)

Comprobando la persistencia de la información

Si en cualquier momento tenemos que eliminar o actualizar uno de los despliegues, podemos comprobar que la información sigue existiendo después de volver a crear los Deployments:

```
kubectl delete -f mariadb-deployment.yaml
kubectl delete -f wordpress-deployment.yaml
kubectl apply -f mariadb-deployment.yaml
kubectl apply -f wordpress-deployment.yaml
```

Si volvemos acceder, comprobamos que la aplicación sigue funcionando con toda la información:

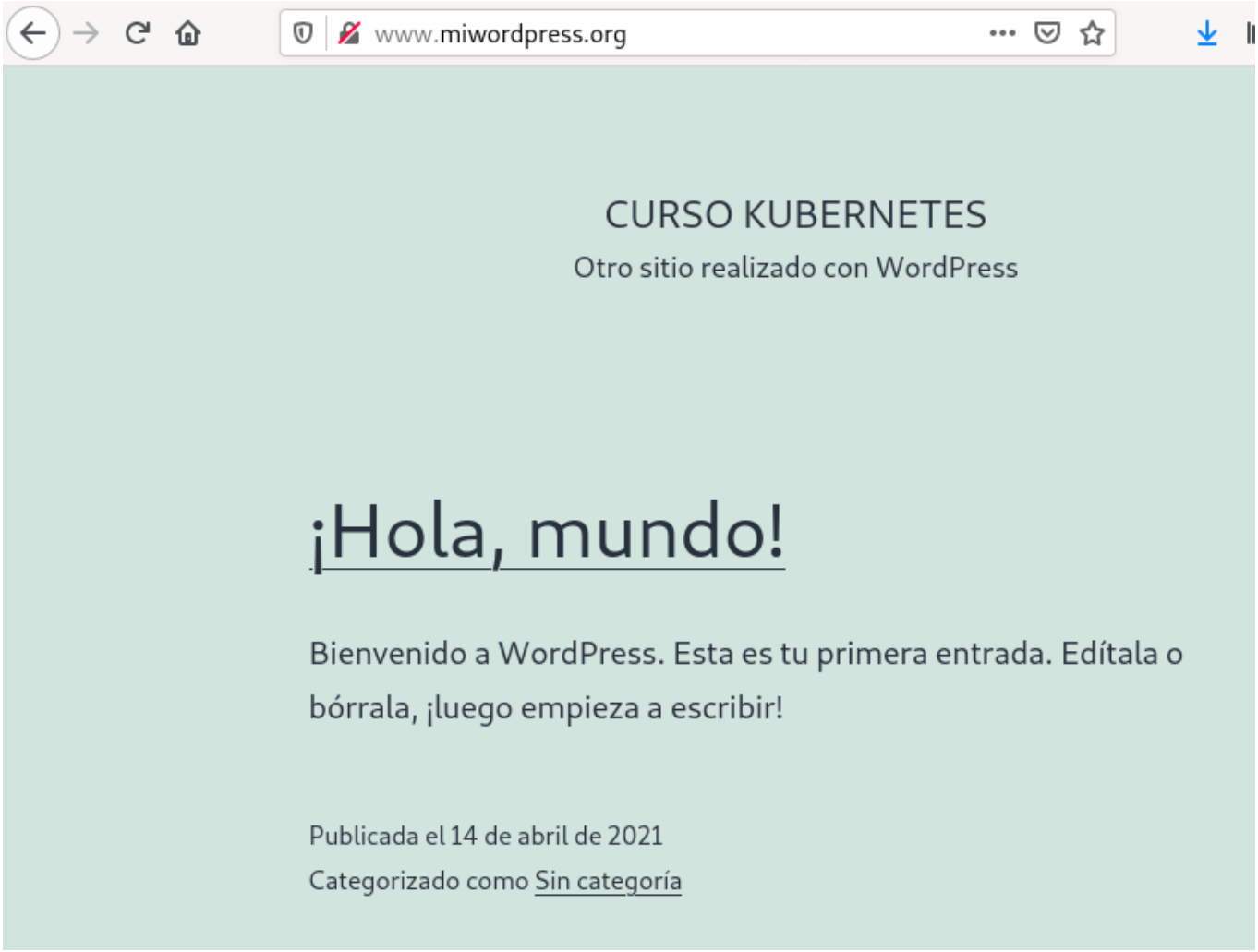
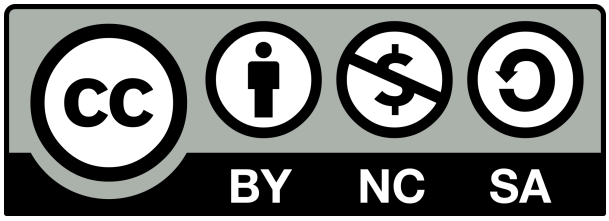


Imagen de elaboración propia (CC BY-NC-SA)

Créditos

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



2021