UNIDAD 4
DESPLIEGUE DE APLICACIONES EN KUBERNETES (II)

DNS

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

- Cada vez que se crea un nuevo servicio se crea un registro de tipo A con el nombre `servicio.namespace.svc.cluster.local.`

Ejemplo 1: DNS

```
kubectl create -f busybox.yaml
kubectl exec -it busybox -- nslookup nginx
kubectl exec -it busybox -- wget http://nginx
```

Comprobando el servidor DNS:

```
kubectl get pods --namespace=kube-system -o wide
kubectl get services --namespace=kube-system
kubectl exec -it busybox -- cat /etc/resolv.conf
```

EJEMPLO 2: Balanceo de carga

Hemos creado una imagen docker que nos permite crear un contenedor con una aplicación PHP que muestra el nombre del servidor donde se ejecuta, el fichero `index.php`:

```
<?php echo "Servidor:"; echo gethostname();echo "\n"; ?>
```

Si tenemos varios pod de esta aplicación, el objeto Service balancea la carga entre ellos:

```
kubectl create deployment pagweb --image=josedom24/infophp:v1
kubectl expose deploy pagweb --port=80 --type=NodePort
kubectl scale deploy pagweb --replicas=3
```

Al acceder hay que indicar el puerto asignado al servicio:

```
for i in `seq 1 100`; do curl http://192.168.99.100:32376; done
Servidor:pagweb-84f6d54fb7-56zj6
Servidor:pagweb-84f6d54fb7-mdvfn
Servidor:pagweb-84f6d54fb7-bhz4p
```

EJEMPLO 3: Servicio para acceder a servidor remoto

Vamos a crear un servicio de tipo ClusterIP que apunte (endpoint) a un servidor remoto, para ello:

service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: mariadb
spec:
 type: ClusterIP
 ports:
 - port: 3306
   TargetPort: 3306
```

endpoint.yaml

```yaml
apiVersion: v1
kind: Endpoints
metadata:
 name: mariadb
subsets:
 - addresses:
     - ip: 1.1.1.1
   ports:
     - port: 3306
```

- El nombre del servicio y del endpoint deben coincidir.
- Comprobamos que el servicio apunta al endpoint:

```
kubectl describe service mariadb
```

- Creamos un pod con el cliente de maridb y accedemos usando el nombre del servicio:

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

kubectl exec -it pod/mariadb -- mysql -u prueba -p -h mariadb.default.svc.cluster.local
```

EJEMPLO 4: Despliegue canary

- Creamos el servicio y el despliegue de la primera versión (5 réplicas):

```
kubectl create -f service.yaml
kubectl create -f deploy1.yaml
```

- En un terminal, vemos los pods:

```
watch kubectl get pod
```

- En otro terminal, accedemos a la aplicación:

```
service=$(minikube service my-app --url)
while sleep 0.1; do curl "$service"; done
```

- Desplegamos una réplica de la versión 2, para ver si funciona bien:

```
kubectl create -f deploy2.yaml
```

- Una vez que comprobamos que funciona bien, podemos escalar y eliminar la versión 1:

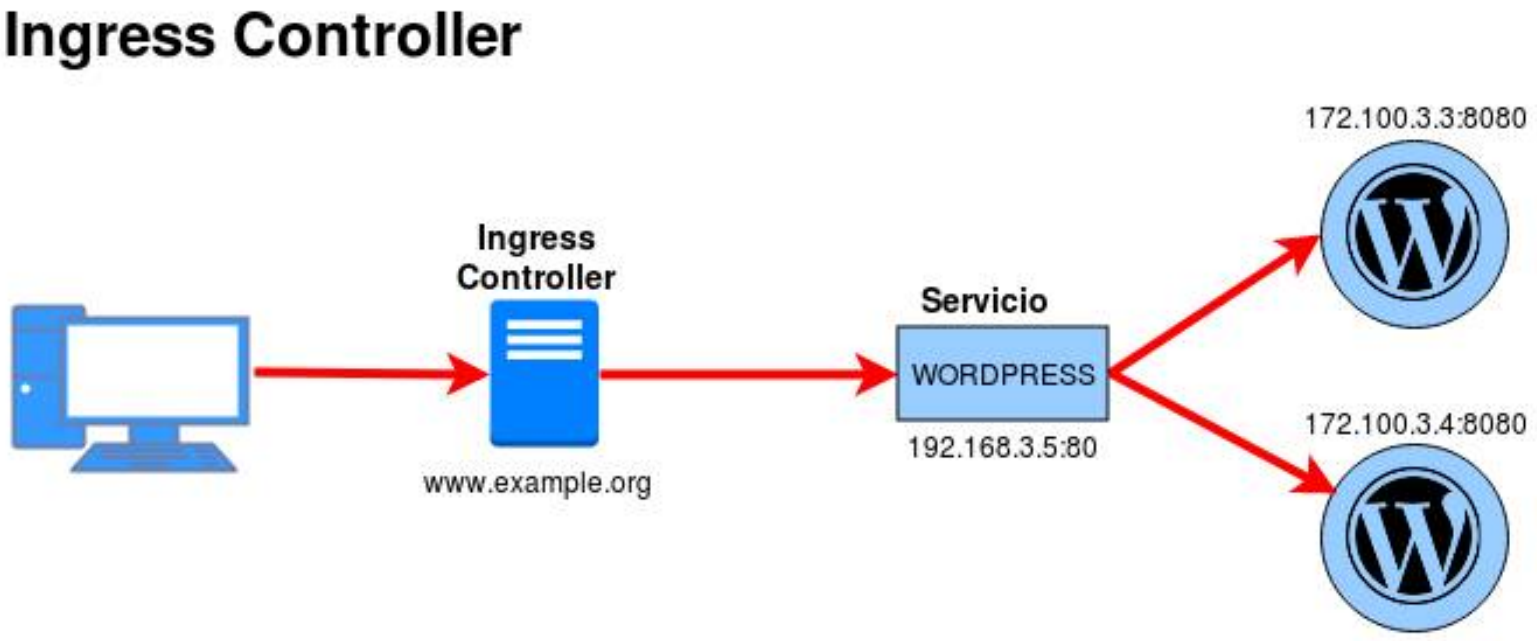
```
kubectl scale --replicas=5 deploy my-app-v2
kubectl delete deploy my-app-v1
```

Ingress Controller

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

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

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



EJEMPLO 5: Trabajando con Ingress

Kubernetes <1.19

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
 name: nginx
spec:
 rules:
 - host: nginx.192.168.99.100.nip.io
   http:
     paths:
     - path: /
       backend:
         serviceName: nginx
         servicePort: 80
```

Kubernetes 1.19

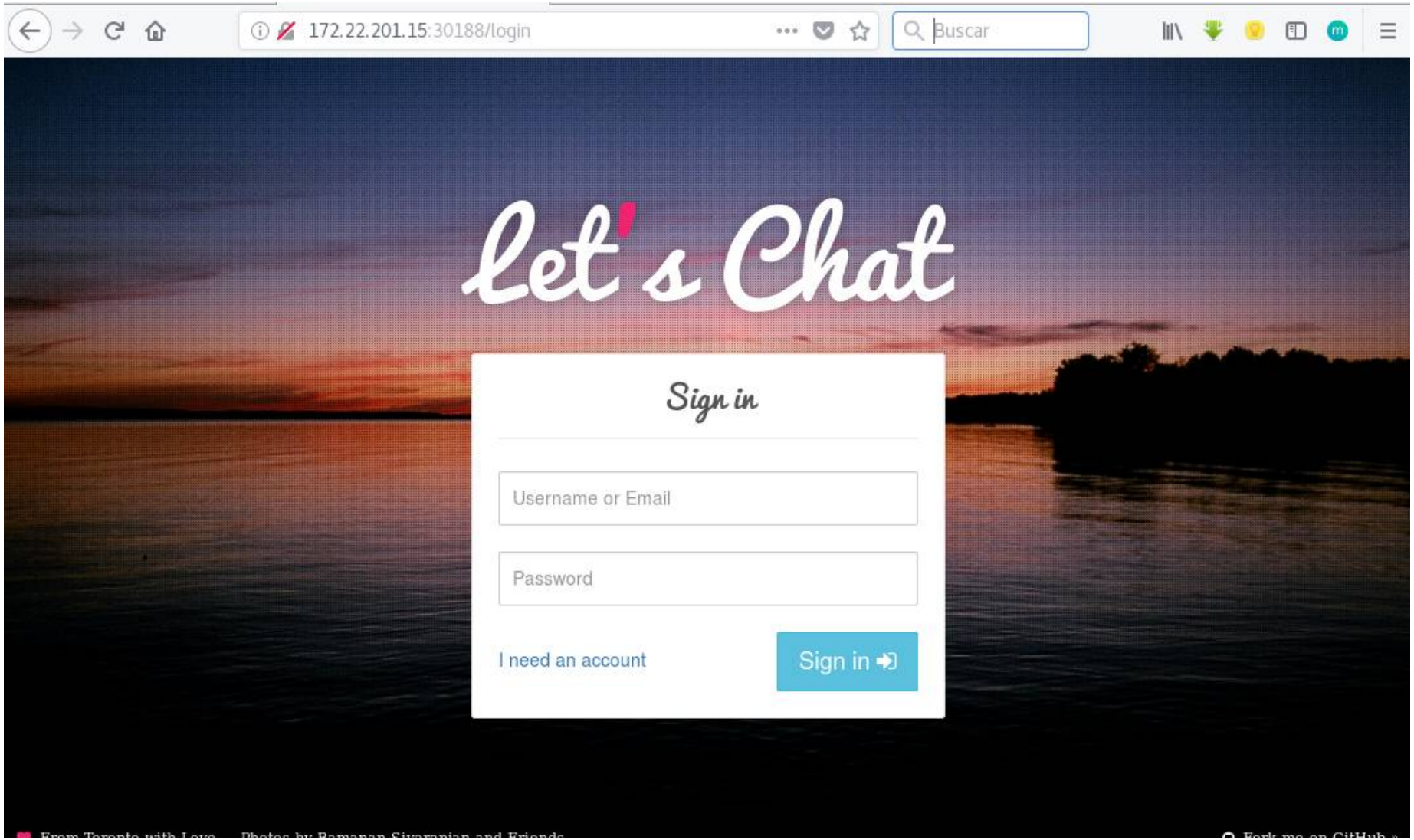
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
  annotations:
     nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: nginx.192.168.99.100.nip.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nginx
            port:
              number: 80
```

Utilizamos anotaciones para configurar algunas opciones dependiendo del ingress controller

```
kubectl create -f nginx-ingress.yaml

kubectl get ingress
```

EJEMPLO 6: Desplegando la aplicación LetsChat



Variables de Entorno

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mariadb-deployment
 labels:
   app: mariadb
   type: database
spec:
 replicas: 1
 template:
   metadata:
     labels:
       app: mariadb
       type: database
   spec:
     containers:
       - name: mariadb
         image: mariadb
         ports:
           - containerPort: 3306
             name: db-port
         env:
           - name: MYSQL_ROOT_PASSWORD
             value: my-password
```

EJEMPLO 7

Podemos definir un Deployment que defina un contenedor configurado por medio de variables de entorno.

Creamos el despliegue:

```
kubectl create -f mariadb-deployment.yaml
```

O directamente ejecutando:

```
kubectl run mariadb --image=mariadb --env MYSQL_ROOT_PASSWORD=my-password
```

Veamos el pod creado:

```
kubectl get pods -l app=mariadb
```

Y probamos si podemos acceder, introduciendo la contraseña configurada:

```
kubectl exec -it mariadb-deployment-fc75f956-f5zlt -- mysql -u root -p
```

ConfigMap

```
...
     containers:
       - name: mariadb
         image: mariadb
         ports:
           - containerPort: 3306
             name: db-port
         env:
           - name: MYSQL_ROOT_PASSWORD
             valueFrom:
               configMapKeyRef:
                 name: mariadb
                 key: root_password
           - name: MYSQL_USER
             valueFrom:
               configMapKeyRef:
                 name: mariadb
                 key: mysql_usuario
           - name: MYSQL_PASSWORD
             valueFrom:
               configMapKeyRef:
                 name: mariadb
                 key: mysql_password
           - name: MYSQL_DATABASE
             valueFrom:
               configMapKeyRef:
                 name: mariadb
                 key: basededatos
```

EJEMPLO 8

ConfigMap te permite definir un diccionario (clave,valor) para guardar información que puedes utilizar para configurar una aplicación.
Al crear un ConfigMap los valores se pueden indicar desde un directorio, un fichero o un literal.

```
kubectl create cm mariadb --from-literal=root_password=my-password \
                          --from-literal=mysql_usuario=usuario     \
                          --from-literal=mysql_password=password-user \
                          --from-literal=basededatos=test
```

```
kubectl get cm
kubectl describe cm mariadb
```

Creamos un deployment indicando los valores guardados en el ConfigMap:

```
kubectl create -f mariadb-deployment-configmap.yaml
kubectl exec -it mariadb-deploy-cm-57f7b9c7d7-ll6pv -- mysql -u usuario -p
```

Secrets

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mariadb-deploy-secret
 labels:
   app: mariadb
   type: database
spec:
 replicas: 1
 template:
   metadata:
     labels:
       app: mariadb
       type: database
   spec:
     containers:
       - name: mariadb
         image: mariadb
         ports:
           - containerPort: 3306
             name: db-port
         env:
           - name: MYSQL_ROOT_PASSWORD
             valueFrom:
               secretKeyRef:
                 name: mariadb
                 key: password
```

EJEMPLO 9

Los Secrets nos permiten guardar información sensible que será codificada. Por ejemplo,nos permite guarda contraseñas, claves ssh, ...
Al crear un Secret los valores se pueden indicar desde un directorio, un fichero o un literal.

```
kubectl create secret generic mariadb --from-literal=password=root
kubectl get secret
kubectl describe secret mariadb
```

Creamos el despliegue y probamos el acceso:

```
kubectl create -f mariadb-deployment-secret.yaml
kubectl exec -it mariadb-deploy-secret-f946dddfd-kkmlb -- mysql -u root -p
```

EJEMPLO 10: Desplegando WordPress con MariaDB

mariadb

```
kubectl create secret generic mariadb-secret \
                            --from-literal=dbuser=user_wordpress \
                            --from-literal=dbname=wordpress \
                            --from-literal=dbpassword=password1234 \
                            --from-literal=dbrootpassword=root1234 \
                            -o yaml --dry-run=client > mariadb-secret.yaml

kubectl create -f mariadb-secret.yaml
```

Creamos el servicio, que será de tipo *ClusterIP*:

```
kubectl create -f mariadb-srv.yaml
```

Y desplegamos la aplicación:

```
kubectl create -f mariadb-deployment.yaml
```

wordpress

Lo primero creamos el servicio:

```
kubectl create -f wordpress-srv.yaml
```

Y realizamos el despliegue:

```
kubectl create -f wordpress-deployment.yaml
```

Por último creamos el recurso ingress que nos va a permitir el acceso a la aplicación utilizando un nombre:

```
kubectl create -f wordpress-ingress.yaml
```

Los pods son efímeros

Cuando se elimina un pod su información se pierde. Por lo tanto nos podemos encontrar con algunas circunstancias:

1. **¿Qué pasa si eliminamos el despliegue de mariadb?, o, ¿se elimina el pod de mariadb y se crea uno nuevo?.**
2. **¿Qué pasa si escalamos el despliegue de la base de datos y tenemos dos pods ofreciendo la base de datos?.**
3. **Si escribimos un post en el wordpress y subimos una imagen, ¿qué pasa con esta información en el pod?**
4. **En el caso que tengamos un pod con contenido estático (por ejemplo imágenes), ¿qué pasa si escalamos el despliegue de wordpress a dos pods?**

StatefulSet

El objeto StatefulSet controla el despliegue de pods con identidades únicas y persistentes, y nombres de host estables. Veamos algunos ejemplos en los que podemos usarlo:

- Un despliegue de redis master-slave: necesita que **el master esté corriendo antes de que podamos configurar las réplicas**.
- Un cluster mongodb: Los diferentes nodos deben **tener una identidad de red persistente** (ya que el DNS es estático), para que se produzca la sincronización después de reinicios o fallos.
- Zookeeper: cada nodo necesita **almacenamiento único y estable**, ya que el identificador de cada nodo se guarda en un fichero.

Por lo tanto el objeto StatefulSet nos ofrece las siguientes **características**:

- Estable y único identificador de red (Ejemplo mongodb)
- Almacenamiento estable (Ejemplo Zookeeper)
- Despliegues y escalado ordenado (Ejemplo redis)
- Eliminación y actualizaciones ordenadas

Por lo tanto cada pod es distinto (tiene una identidad única), y este hecho tiene algunas consecuencias:

- El nombre de cada pod tendrá un número (1,2,...) que lo identifica y que nos proporciona la posibilidad de que la creación actualización y eliminación sea ordenada.
- Si un nuevo pod es recreado, obtendrá el mismo nombre (hostname), los mismos nombres DNS (aunque la IP pueda cambiar) y el mismo volumen que tenía asociado.
- Necesitamos crear un servicio especial, llamado **Headless Service**, que nos permite acceder a los pods de forma independiente, pero que no balancea la carga entre ellos, por lo tanto este servicio no tendrá una ClusterIP.

StatefulSet us Deployment

- A diferencia de un Deployment, un StatefulSet mantiene una identidad fija para cada uno de sus Pods.
- Eliminar y / o escalar un StatefulSet no eliminará los volúmenes asociados con StatefulSet.
- StatefulSets actualmente requiere que un Headless Service sea responsable de la identidad de red de los Pods.
- Cuando use StatefulSets, cada Pod recibirá un PersistentVolume independiente.
- StatefulSet actualmente no admite el escalado automático

Componentes StatefulSet: headless service

```yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 labels:
   app: nginx
spec:
 ports:
 - port: 80
   name: web
 clusterIP: None
 selector:
   app: nginx
```

- El headless service nos proporciona acceso a los pods creados.
- No va a tener una IP (`clusterIP: None`)
- Será referencia por el objeto StatefulSet (`name: nginx`)

Headless Service

Componentes StatefulSet: StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: web
spec:
 serviceName: "nginx"
 replicas: 2
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
     - name: nginx
       image: k8s.gcr.io/nginx-slim:0.8
       ports:
       - containerPort: 80
         name: web
       volumeMounts:
       - name: www
         mountPath: /usr/share/nginx/html
   ...
```

- Se indica los pods que vamos a controlar que vamos a utilizar:

```yaml
selector:
   matchLabels:
     app: nginx
```

- Vamos a montar un volumen persistente

```yaml
volumeMounts:
     - name: www
         mountPath: /usr/share/nginx/html
```

Componentes StatefulSet: volumenClainTemplate

```
...
volumeClaimTemplates:
 - metadata:
     name: www
   spec:
     accessModes: [ "ReadWriteOnce" ]
     resources:
       requests:
         storage: 1Gi
```

- Nos ofrece almacenamiento estable usando PersistentVolumes
- La definición es similar a un `PersistantVolumenClaim`.

StatefulSet: EJEMPLO 11

Vamos a crear los distintos objetos de la API:

```
kubectl create -f service.yaml
```

Creación ordenada de pods: En un terminal observamos la creación de pods y en otro terminal creamos los pods

```
watch kubectl get pod
kubectl create -f statefulset.yaml
```

Comprobamos la identidad de red estable: Vemos los hostnames y los nombres DNS asociados:

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1

kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
/ # nslookup web-0.nginx
...
Address 1: 172.17.0.4 web-0.nginx.default.svc.cluster.local
/ # nslookup web-1.nginx
...
Address 1: 172.17.0.5 web-1.nginx.default.svc.cluster.local
```

StatefulSet: EJEMPLO 11

Eliminación de pods: En un terminal observamos la creación de pods y en otro terminal eliminamos los pods

```
watch kubectl get pod
kubectl delete pod -l app=nginx
```

Comprobamos la identidad de red estable: Vemos los hostnames y los nombres DNS asociados **(Las IP pueden cambiar)**:

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
/ # nslookup web-0.nginx
...
/ # nslookup web-1.nginx
```

StatefulSet: EJEMPLO 11

Escribiendo en los volúmenes persistentes: Comprobamos que se han creado volúmenes para los pods:

```
kubectl get pv,pvc
```

Escribimos en los documentroot y accedemos al servidor:

```
for i in 0 1; do kubectl exec "web-$i" -- sh -c 'echo "$(hostname)" > /usr/share/nginx/html/index.html'; done
for i in 0 1; do kubectl exec -i -t "web-$i" -- sh -c 'curl http://localhost/'; done
web-0
web-1
```

Volvemos a eliminar los pods, y comprobamos que la información es persistente al estar guardadas en los volúmenes:

```
kubectl delete pod -l app=nginx
for i in 0 1; do kubectl exec -i -t "web-$i" -- sh -c 'curl http://localhost/'; done
...
```

DaemonSet

El objeto DaemonSet (DS) nos asegura que en todos (o en algunos) nodos de nuestro cluster vamos a tener un pod ejecutándose. Si añadimos nuevos nodos al cluster se crearán nuevo pods. Para que podemos necesitar esta característica:

- Monitorización del cluster (Prometheus)
- Recolección y gestión de logs (fluentd)
- Cluster de almacenamiento (glusterd o ceph)

Ejemplo 12

Vemos el **ejemplo**en un cluster de 3 nodos:

```
kubectl get nodes
NAMESTATUS   ROLES   AGE   VERSION
k3s-1   Ready    <none>   17d   v1.14.1-k3s.4
k3s-2   Ready    <none>   17d   v1.14.1-k3s.4
k3s-3   Ready    <none>   17d   v1.14.1-k3s.4

kubectl create -f ds.yaml

kubectl get pods -o wide
NAME         READY   STATUS   RESTARTS   AGE   IP         NODE
logging-5v4dh   1/1   Running   0          9s   10.42.2.26   k3s-2
logging-gqfbb   1/1   Running   0          9s   10.42.1.55   k3s-3
logging-wbdjj   1/1   Running   0          9s   10.42.0.25   k3s-1
```

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: logging
spec:
 template:
   metadata:
     labels:
       app: logging-app
   spec:
     containers:
       - name: webserver
         image: nginx
         ports:
         - containerPort: 80
```

DaemonSet

Podemos seleccionar los nodos en los que queremos que se ejecuten los pod por medio de un selector.

```
kubectl create -f ds2.yaml

kubectl get pods -o wide
No resources found.

 kubectl get ds
NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR      AGE
logging   0         0         0       0            0           app=logging-node   17s

kubectl label node k3s-3 app=logging-node --overwrite

kubectl get ds
NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR      AGE
logging   1         1         0       1            0           app=logging-node   41s

kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS   AGE   IP           NODE    NOMINATED NODE   READINESS GATES
logging-556r9   1/1     Running   0          7s    10.42.1.56   k3s-3   <none>           <none>
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: logging
spec:
 template:
   metadata:
     labels:
       app: logging-app
   spec:
     nodeSelector:
       app: logging-node
     containers:
       - name: webserver
         image: nginx
         ports:
         - containerPort: 80
```

Otros recursos: Jobs, cronJobs,...

Jobs

- Deseamos ejecutar una acción y asegurarse que se finaliza correctamente (Rellenar una base de datos, descargar datos,...)
- Un Job crea uno o más pods y se asegura que un número determinado de ellos ha terminado de forma adecuada.

Si necesita que un Job se repita periódicamente usamos un cronJob:

- Por ejemplo si quieres hacer backup de base de datos
- Se puede especificar una momento determinado, o indicar una repetición periodica.

Horizontal Pod AutoScaler

El HPA de Kubernetes nos permite variar el número de pods desplegados mediante un *deployment* en función de diferentes métricas: de forma estable usando el porcentaje de **CPU** utilizado y de forma experimental de utilización de la **memoria.**

Necesitamos tener instalado en nuestro cluster un software que permita monitorizar el uso de recursos: **metrics-server.**
En minikube es muy fácil instalarlo: `minikube addons enable metrics-server`

Ejemplo 13

Creamos un despliegue de una aplicación php y modificamos lo que va a reservar del CPU el pod (0,2 cores de CPU):

```
kubectl create deploy php-apache --image=k8s.gcr.io/hpa-example
kubectl expose deploy php-apache --port=80 --type=NodePort
kubectl set resources deploy php-apache --requests=cpu=200m
```

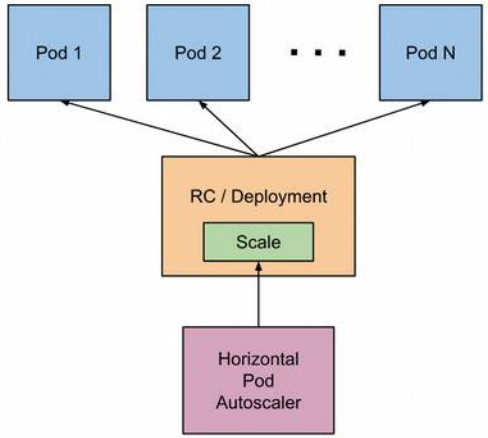
Creamos el recurso hpa, indicando el mínimo y máximos de pods que va a tener el despliegue, y el límite de uso de CPU que va a tener en cuenta para crear nuevos pods:

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=5
```

Vamos a hacer una prueba de estrés a nuestra aplicación y observamos cómo se comporta:

```
$ service=$(minikube service php-apache --url)
  while true; do wget -q -O- "$service"; done

kubectl get pod -w
kubectl get hpa -w
```



```
Type: Deployment
name: wp-dep2

Type: Deployment
name: mysql-dep2

Type: Service
name: wp-svc2

Type: Service
name: mysql-svc2

Type: PVC
name: wp-pvc2

Type: Secret
name: wp-secret2

Type: Secret
name: wp-secret2
```

"Package" WordPress??

Kubernetes Packaging

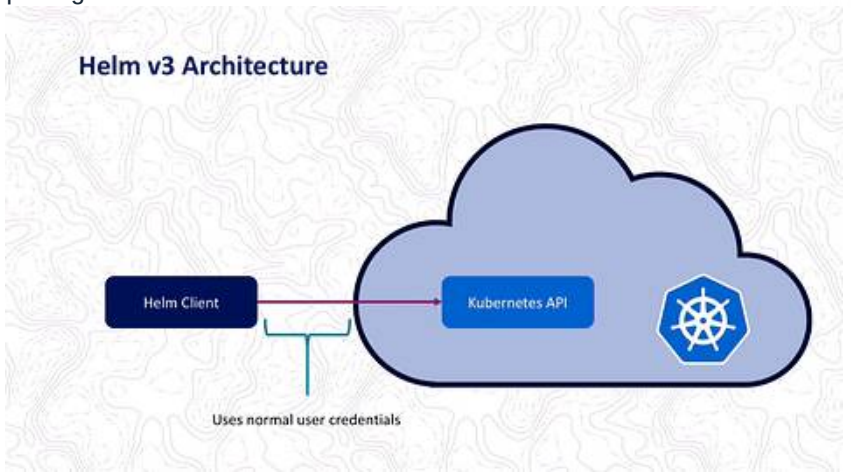
Necesitamos una herramienta para gestionar un conjunto de objetos como una unidad.

Helm Chart: Paquete Kubernetes, que define un conjunto de recursos.
Tenemos un catálogo de Chart disponible.

HELM: Ejemplo 14

Instalación:

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```



Inicializamos repositorios de Chart

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Actualizamos el repositorio

```
helm repo update
```

Buscamos un chart

```
helm search repo nginx
```

Instalamos el chart

```
helm install server_web bitnami/nginx --set service.type=NodePort
```

Listamos las aplicaciones instaladas

```
helm ls
```

Borramos la aplicación

```
helm delete server_web
```

Vemos los recursos creados

```
helm status server_web
```

Para saber los parámetros que podemos configurar en la instalación del chart:

https://hub.helm.sh/charts/bitnami/nginx

HELM: Ejemplo 14

Tenemos un repositorio público de charts:

https://hub.helm.sh

¡¡¡Nosotros podemos crear nuestros propios charts!!!

```
helm create my-wordpress
helm install my-wordpress
```

- Se utilizan templates para parametrizar la instalación (puedo instalar varios wp)
- The Chart Template Developer's Guide