CONTENIDOS

1 DESPLIEGUE DE APLICACIONES EN CONTENEDORES (2 HORAS)

- Introducción a los contenedores
- Arquitectura de microservicios
- Tecnologías subyacentes y diferencias entre ellas: docker, cri-o, LXC, ...
- Ciclo de vida en el despliegue de aplicaciones con docker

2 INTRO A KUBERNETES (2 HORAS)

- Características, historia, estado actual del proyecto kubernetes (k8s)
- Arquitectura básica de k8s
- Alternativas para instalación simple de k8s: minikube, kubeadm, k3s
- Instalación con minikube
- Instalación y uso de kubectl
- Despliegue de aplicaciones con k8s

3 DESPLIEGUE DE APLICACIONES CON K8S (1:30 HORAS)

- Pods
- ReplicaSet: Tolerancia y escalabilidad
- Deployment: Actualizaciones y despliegues automáticos

4 COMUNICACIÓN ENTRE SERVICIOS Y ACCESO DESDE EL EXTERIOR (1:30 HORAS)

- Services
- DNS
- Ingress
- Ejemplos de uso y despliegues

5 CONFIGURACIÓN DE APLICACIONES (1 HORA)

- Variables de entorno
- ConfigMaps
- Secrets
- Ejemplo de despliegue parametrizado

6 ALMACENAMIENTO (1:30 HORAS)

- Consideraciones sobre el almacenamiento
- PersistentVolume
- PersistentVolumeClaim
- Ejemplo de despliegue con volúmenes

CONTENIDOS

7 OTROS TIPOS DE DESPLIEGUES (1:30 HORAS)

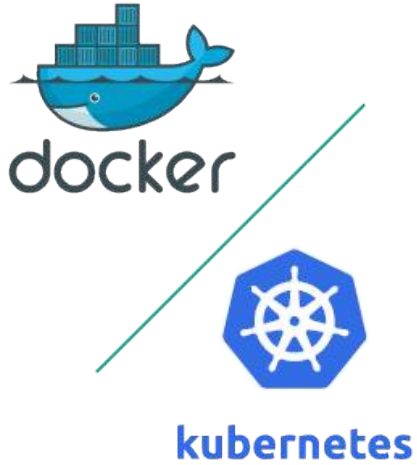
- StatefulSet
- DaemonSet
- AutoScale
- Helm

8 ADMINISTRACIÓN BÁSICA (1 HORA)

- Namespaces
- Usuarios
- RBAC
- Cuotas y límites

9 INSTALACIÓN PASO A PASO (4 HORAS)

- Consideraciones previas: Requisitos hardware, arquitectura física y lógica, entornos y herramientas para el despliegue
- Instalación completa componente a componente en múltiples nodos



MÓDULO 7. OTROS TIPOS DE DESPLIEGUES

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 a**lmacenamiento 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 1

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 1

Creación ordenada 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
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
```

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)

Vemos el ejemplo 2 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 /tmp/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: extensions/v1beta1
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 /tmp/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: extensions/v1beta1
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`

Veamos un ejemplo: 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
```

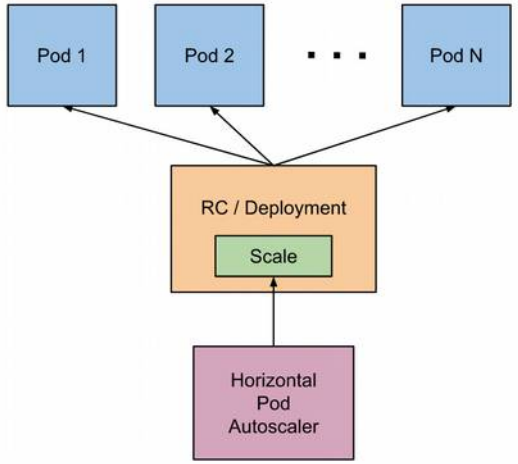
Creamos el recurso hpa, indicando el mínimo y máximos de pods que va a tener el despliegue, y el límite de uso de CPU que va a tener en cuenta para crear nuevos pods:

```
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:

```
$ while true; do wget -q -O- http://192.168.99.100:30372/; done

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



"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

Descargamos el binario de helm de la página de descarga

Inicializamos helm
```
helm init
```

Actualizamos el repositorio
```
helm repo update
```

Buscamos un chart
```
helm search wordpress
```

Instalamos el chart
```
helm install --set serviceType=NodePort --name wp-k8s stable/wordpress
```

Listamos las aplicaciones instaladas
```
helm ls
```

Borramos la aplicación
```
helm delete wp-k8s
```

Vemos los recursos creados
```
helm status wp-k8s
```

¡¡¡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