Otras cargas de trabajo

Otras cargas de trabajo

¿Podemos usar un despliegue para todo?

Hasta ahora hemos visto con bastante detalle el uso de los despliegues (Deployments) en Kubernetes. Los despliegues son una herramienta enormemente potente ya que nos permiten adecuar el número de pods a la demanda y garantizan el funcionamiento continuo, tanto en el caso de que haya algún nodo con problemas, como en el caso de actualizaciones que se pueden realizar de forma continua. Sin embargo, no es posible utilizar despliegues en todos los casos, hay determinadas situaciones en las que hay cargas de trabajo (*workloads*) que no se ajustan adecuadamente a un despliegue de Kubernetes, por lo que se han desarrollado otros objetos para esas situaciones diferentes.

Sí es conveniente remarcar, que siempre que sea posible es mejor definir una carga de trabajo como un despliegue en Kubernetes y limitar el uso de las otras cargas de trabajo que vamos a ver a continuación para casos específicos. Luego la respuesta a la pregunta con la que empezamos este módulo es no, no podemos usar un despliegue para todo, pero sí debemos usarlo prioritariamente siempre que sea posible.

Aplicaciones con estado o sin estado

Una característica de una aplicación que es muy importante para Kubernetes es si se trata de una aplicación con estado (*stateful*) o sin estado (*stateless*). Una aplicación sin estado es aquella en la que las peticiones son totalmente independientes unas de otras y no necesita ninguna referencia de una petición anterior. Un ejemplo de una aplicación sin estado sería un servicio DNS, en el que cada vez que se realiza una petición es totalmente independiente de las anteriores o posteriores que se hagan. Las aplicaciones sin estado son perfectas para desplegarse el Kubernetes, ya que se ajustan perfectamente a un Deployment y se pueden escalar y balancear sin problemas, ya que cada pod responderá a las peticiones que reciba de forma independiente al resto.

Por contra, las aplicaciones con estado son aquellas en las que una petición puede verse afectada por el resultado de las anteriores y a su vez puede afectar a las posteriores (por eso se dice que tiene estado). Una base de datos sería el paradigma de una aplicación con estado, puesto que cada modificación que hagamos a la base de datos puede afectar a las consultas posteriores. Una aplicación con estado no se ajusta bien a un Deployment de Kubernetes, ya que de forma general, un cluster de pods independientes no puede tener en cuenta el estado de la aplicación correctamente.

Esto enlaza con el modelo de desarrollo de las aplicaciones, ya que si pensamos en la mayoría de las aplicaciones que utilizamos hoy en día, se trata de aplicaciones con estado, lo que inicialmente podría limitar su uso en Kubernetes. Sin embargo, si descomponemos estas aplicaciones en muchos y pequeños servicios que se intercomuniquen entre sí, bastantes de ellos se podrán gestionar como aplicaciones sin estado, mientras que otros tendrán que ser aplicaciones con estado. Éste es uno de los enfoques más utilizados hoy en día para desplegar aplicaciones en Kubernetes, hacer que la aplicación se ajuste al modelo de microservicios, para utilizar Deployments en todos los microservicios sin estado que se pueda y utilizar otras cargas de trabajo para el resto de componentes. En esta unidad veremos una pequeña introducción a estas otras cargas de trabajo.

Vídeo: Otras cargas de trabajo

https://www.youtube.com/embed/zZwlSERb6IA

Vídeo: Otras cargas de trabajo

StatefulSets

El objeto StatefulSet controla el despliegue de Pods con identidades únicas y persistentes, y nombres de host estables. El uso de StatefulSets es alternativo al de despliegues (Deployments) y su objetivo principal es poder utilizar en Kubernetes aplicaciones más restrictivas que no se ajusten bien a las características de los despliegues, principalmente aplicaciones con estado que necesiten algunas características fijas y estables en los Pods, algo que no puede ocurrir con los despliegues en los pods que son completamente indistinguibles unos de otros y la aplicación puede utilizar cualquiera de ellos en cada momento.

Veamos algunos ejemplos en los es adecuado utilizar StatefulSet en lugar de Deployment y por qué:

- Un despliegue de redis primario-secundario: necesita que **el primario esté operativo 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. Se crearán en orden ascendente y se eliminarán en orden descendente.
- 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 Service 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 Service no tendrá una ClusterIP.

StatefulSet vs 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.
- Al utilizar StatefulSet, cada Pod recibe un PersistentVolume independiente.
- StatefulSet actualmente no admite el escalado automático.

Creando el *Headless Service* para acceder a los Pods del StatefulSet

Una de las características de los Pods controlados por un StatefulSet es que son únicos (todos los Pods son distintos), por lo tanto al acceder a ellos por medio de la definición de un Service no necesitamos el balanceo de carga entre ellos.

Para acceder a los Pods de un StatefulSet vamos a crear un Service Headless que se caracteriza por no tener IP (ClusterIP) y por lo tanto no va a balancear la carga entre los distintos Pods. Este tipo de Service va a crear una entrada DNS por cada Pod, que nos permitirá acceder a cada Pod de forma independiente. El nombre DNS que se creará será `<nombre del Pod>.<dominio del StatefulSet>`. El dominio del StatefulSet se indicará en la definición del recurso usando el parámetro `serviceName`.

Veamos un ejemplo de definición de un Headless Service (fichero `service.yaml`):

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

```
  selector:
    app: nginx
```

En esta definición podemos observar que al indicar `clusterIP: None` estamos creando un Headless Service, que no tendrá ClusterIP (por lo que no balanceará la carga entre los pods). Este Service será el responsable de crear, por cada Pod seleccionado con el `selector`, una entrada DNS.

Creando el recurso StatefulSet

Vamos a definir nuestro recurso StatefulSet en un fichero yaml `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
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Vamos a estudiar las características de la definición de este recurso:

- Con el parámetro `serviceName` indicaremos el nombre de dominio que va a formar parte del nombre DNS que el Headless Service va a crear para cada Pod.
- Con el parámetro `selector` se indica los Pods que vamos a controlar con StatefulSet.
- Una de las características que hemos indicado del StatefulSet es que cada Pod va a tener un almacenamiento estable. El tipo de almacenamiento se indica con el parámetro `volumeClaimTemplates` que se define de forma similar a un `PersistentVolumenClaim`.
- Además observamos en la definición del contenedor que el almacenamiento que hemos definido se va a montar en cada Pod (en este ejemplo el punto de montaje es el DocumentRoot de nginx), con el parámetro `volumeMounts`.

Ejemplo: Creación de un StatefulSet

Vamos a crear los recursos estudiados en este apartado: el Service Headless y el StatefulSet, y vamos a comprobar sus características.

Lo primero es crear el Headless Service:

```
kubectl apply -f service.yaml
```

Creación ordenada de Pods

El statefulSet que hemos definido va a crear dos Pods (`replicas: 2`). Para observar cómo se crean de forma ordenada podemos usar dos terminales, en la primera ejecutamos:

```
watch kubectl get pod
```

Con esta instrucción vamos a ver en "vivo" cómo se van creando los Pods, que vamos a crear al ejecutar en otra terminal la instrucción:

```
kubectl apply -f statefulset.yaml
```

Comprobamos la identidad de red estable

En este caso vamos a comprobar que los hostname y los nombres DNS son estables para cada Pod. Las ips de los Pods pueden cambiar si eliminamos el recurso StatefulSet y lo volvemos a crear, pero los nombres van a permanecer.

Para ver los nombres de los Pods podemos ejecutar lo siguiente:

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

Veamos los nombres DNS. En este ejemplo el Headless Service ha creado una entrada en el DNS para cada Pod. El nombre DNS que se creará será `<nombre del pod>.<dominio del StatefulSet>`. El dominio del StatefulSet se indicará en la definición del recurso usando el parámetro `serviceName`. En este caso el nombre del primer Pod será `web-0.nginx`. Vamos a comprobarlo, haciendo una consulta DNS desde otro pod:

```
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
```

Eliminación de pods

Podemos usar las dos terminales para observar cómo la eliminación también se hace de forma ordenada. En la primera terminal ejecutamos:

```
watch kubectl get pod
```

Y en la segunda:

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

Al eliminar los Pods, el statefulSet ha creado nuevos Pods que serán idénticos a los anteriores y por lo tanto mantendrán la identidad de red, es decir tendrán los mismos hostname y los mismos nombres DNS (aunque es posible que cambien las ip):

```
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
...
/ # nslookup web-1.nginx
...
```

Escribiendo en los volúmenes persistente

Podemos comprobar que se han creado los distintos volúmenes para cada pod:

```
kubectl get pv,pvc
```

Y podemos comprobar que realmente la información que guardemos en el directorio que hemos montado en cada Pod es persistente:

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

Ahora si eliminamos los Pods, los nuevos Pods creados mantendrán la información:

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

Vídeo: StatefulSets

https://www.youtube.com/embed/ULaNgtJ8NHQ

Vídeo: StatefulSets

Ejemplo: Despliegue de un cluster de MySQL

Una base de datos relacional es un ejemplo perfecto de aplicación con estado, cualquier petición puede depender del estado resultante de una petición anterior, y todas las peticiones deben consultar o modificar la base de datos que incluya la última modificación realizada. Las implicaciones que tiene esto si queremos desplegar la base de datos sobre Kubernetes son importantes, ya que será necesario que la base de datos se pueda desplegar en un cluster para proporcionar disponibilidad, se pueda adaptar a variaciones de demanda y no haya interrupciones, mientras que hay que garantizar que todos los pods accedan a la base de datos de forma coherente.

- ¿Podemos utilizar un Deployment con réplicas indistinguibles que se crean o destruyen a demanda?
- ¿Necesitamos un volumen adicional para almacenar localmente la base de datos en cada pod?
- ¿Cómo garantizamos la replicación de la base de datos entre los nodos?

Hay diferentes formas de afrontar esto y dependen mucho de las características de la base de datos en cuestión. En este ejemplo vamos a desplegar un cluster de MySQL con un nodo primario (también denominado *master*) en modo lectura y escritura (se podrán hacer consultas y modificaciones de la base de datos) y varios nodos secundarios en modo lectura (sólo se utilizarán para realizar consultas). Utilizaremos para ello un StatefulSet, en el que los diferentes pods son distinguibles entre sí, tienen siempre el mismo nombre DNS interno y el mismo volumen se conecta siempre al mismo pod. El primer nodo que se desplegará será el primario, con un volumen asociado para la base de datos, el resto de nodos se arrancarán después del primario y al iniciarse sincronizarán la base de datos con la del primario y la almacenarán también en un volumen diferente para cada Pod.

Este ejemplo es probablemente el más avanzado de todo el curso y utilizaremos además otros recursos como ConfigMap, Services o volúmenes. Este ejemplo está extraído directamente de la documentación de k8s: Run a Replicated Stateful Application.

Nota

Nota Las características de este cluster no son para poner en producción, ya que se ha simplificado la configuración de MySQL, para centrarnos en los aspectos relacionados con Kubernetes.

Nota

Nota En caso de no tener recursos suficientes para realizar este ejemplo, se puede reducir el número de réplicas a dos, o bien eliminar el cluster y volverlo a crear con suficientes recursos (en el ejemplo siguiente creamos un nuevo cluster de k8s con 6 GiB de RAM y 4 cores virtuales:

```
minikube stop
minikube delete
minikube start --driver ... --memory 6144 --cpus 4
```

Vamos pues con la creación de este cluster, para lo que utilizaremos diferentes objetos de Kubernetes que hemos visto durante todo el curso.

Creamos un ConfigMap para modificar el fichero de configuración de MySQL, de manera que el primario genere los registros para la sincronización y los secundarios actúen en modo lectura: configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  primary.cnf: |
    # Modificación del primario
    [mysqld]
    log-bin
  replica.cnf: |
    # Modificación de los secundarios
    [mysqld]
    super-read-only
```

```
kubectl apply -f configmap.yaml
```

Creamos dos servicios, uno de tipo Headless asociado con el StatefulSet para gestionar los nombres internos de los pods y otro para balancear entre los diferentes pods secundarios las peticiones de lectura: servicios.yaml

```
# Servicio para usar los nombres DNS internamente
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
# Servicio para balancear los clientes entre los nodos secundarios
# en modo lectura
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

```
kubectl apply -f servicios.yaml
```

Y ya por último creamos el StatefulSet, que en este caso es el objeto más complicado que vamos a ver en este curso. Veamos los elementos que utiliza:

- Está formado inicialmente por tres pods, de los que en el primero se ejecutará el servidor MySQL primario y en los otros dos los servidores MySQL secundarios.
- Se define un volumen de 10 GiB para cada pod a través de un volumeClaim.
- Utiliza dos contenedores en cada pod, el principal que se encarga de ejecutar el proceso mysql y uno adicional que se encarga de la sincronización de la base de datos mediante

XtraBackup. Ambos contenedores deben poder acceder al mismo volumen en el que se encuentra la base de datos, en el punto de montaje `/var/lib/mysql`.

- Utiliza InitContainers que no hemos visto en el curso; estos contenedores se ejecutan dentro del pod antes del contenedor normal y se utilizan para realizar configuraciones o modificaciones previas a la utilización del contenedor que va a ejecutar la aplicación. Una vez que el InitContainer ha finalizado se lanza el contenedor normal. En este caso se utilizan para la configuración inicial del contenedor mysql mediante un script que se lanza como un comando (si el índice es 0 configura el contenedor como primario y si es otro número, lo hace como secundario). El otro InitContainer se utiliza para clonar inicialmente la base de datos en los secundarios con `xtrabackup`.

- Se establece límite de consumo de recursos y se definen las pruebas de disponibilidad para que Kubernetes pueda comprobar si se está ofreciendo el servicio de forma adecuada.

- Se define el acceso a la base de datos sin contraseña, lo que hace que el sistema no sea válido para un despliegue real.

statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Numera los servidores en función del índice del pod
```

```
      [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
      ordinal=${BASH_REMATCH[1]}
      echo [mysqld] > /mnt/conf.d/server-id.cnf
      # Establece el número del servidor a partir de 100
      echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf
      # Modifica MySQL con el ConfigMap en función de si es el primario (0) o r
      if [[ $ordinal -eq 0 ]]; then
        cp /mnt/config-map/primary.cnf /mnt/conf.d/
      else
        cp /mnt/config-map/replica.cnf /mnt/conf.d/
      fi
    volumeMounts:
    - name: conf
      mountPath: /mnt/conf.d
    - name: config-map
      mountPath: /mnt/config-map
  - name: clone-mysql
    image: gcr.io/google-samples/xtrabackup:1.0
    command:
    - bash
    - "-c"
    - |
      set -ex
      # No clona si ya existen datos.
      [[ -d /var/lib/mysql/mysql ]] && exit 0
      # No clona si se trata del primario.
      [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
      ordinal=${BASH_REMATCH[1]}
      [[ $ordinal -eq 0 ]] && exit 0
      # Clona los datos del pod inmediatamente anterior
      ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream -x -C /var/l
      # Prepara la copia de seguridad
      xtrabackup --prepare --target-dir=/var/lib/mysql
    volumeMounts:
    - name: data
      mountPath: /var/lib/mysql
      subPath: mysql
    - name: conf
      mountPath: /etc/mysql/conf.d
  containers:
  - name: mysql
    image: mysql:5.7
    env:
    - name: MYSQL_ALLOW_EMPTY_PASSWORD
      value: "1"
    ports:
    - name: mysql
```

```
        containerPort: 3306
      volumeMounts:
      - name: data
        mountPath: /var/lib/mysql
        subPath: mysql
      - name: conf
        mountPath: /etc/mysql/conf.d
      resources:
        requests:
          cpu: 200m
          memory: 1Gi
      livenessProbe:
        exec:
          command: ["mysqladmin", "ping"]
        initialDelaySeconds: 30
        periodSeconds: 10
        timeoutSeconds: 5
      readinessProbe:
        exec:
          # Comprueba si se pueden hacer consultas sobre TCP
          command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
        initialDelaySeconds: 5
        periodSeconds: 2
        timeoutSeconds: 1
    - name: xtrabackup
      image: gcr.io/google-samples/xtrabackup:1.0
      ports:
      - name: xtrabackup
        containerPort: 3307
      command:
      - bash
      - "-c"
      - |
        set -ex
        cd /var/lib/mysql

        # Determina la posición de log a clonar.
        if [[ -f xtrabackup_slave_info && "x$(<xtrabackup_slave_info)" != "x" ]];
          # Modificaciones previas
          cat xtrabackup_slave_info | sed -E 's/;$//g' > change_master_to.sql.in
          rm -f xtrabackup_slave_info xtrabackup_binlog_info
        elif [[ -f xtrabackup_binlog_info ]]; then
          # Si existe xtrabackup_binlog_info, estamos clonando desde el primario
          [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+(.*?)$ ]] || exit
          rm -f xtrabackup_binlog_info xtrabackup_slave_info
          echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}',\
                MASTER_LOG_POS=${BASH_REMATCH[2]}" > change_master_to.sql.in
```

```
            fi

            # Se comprueba si es necesaria completar un clon iniciando la replicación
            if [[ -f change_master_to.sql.in ]]; then
              echo "Esperando a que mysqld esté disponible"
              until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

              echo "Inicializando la réplica desde la última modificación"
              mysql -h 127.0.0.1 \
                    -e "$(<change_master_to.sql.in), \
                            MASTER_HOST='mysql-0.mysql', \
                            MASTER_USER='root', \
                            MASTER_PASSWORD='', \
                            MASTER_CONNECT_RETRY=10; \
                          START SLAVE;" || exit 1
              # En caso de que el contenedor se reinicie, se intenta de nuevo.
              mv change_master_to.sql.in change_master_to.sql.orig
            fi

            # Lanza un servidor que pueda mandar copias solicitadas por otros.
            exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \
              "xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1 --
          volumeMounts:
          - name: data
            mountPath: /var/lib/mysql
            subPath: mysql
          - name: conf
            mountPath: /etc/mysql/conf.d
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
        volumes:
        - name: conf
          emptyDir: {}
        - name: config-map
          configMap:
            name: mysql
    volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi
```

```
kubectl apply -f statefulset.yaml
```

Podemos ir comprobando con `kubectl` cómo se van creando los diferentes pods y contenedores, tanto los InitContainers como los contenedores de cada pod y al tratarse de un StatefulSet, los pods no se crean en paralelo, lo hacen de manera secuencial (algo fundamental en este caso, ya que hasta que no ha terminado el primer pod que contiene el contendor primario, no deben lanzarse los secundarios).

Prueba de funcionamiento de la base de datos

Creamos un pod efímero con un cliente de MySQL para crear una tabla con un registro en el pod mysql-0 (con nombre DNS mysql-0.mysql):

```
kubectl run mysql-client --image=mysql:5.7 -i --rm --restart=Never --\
  mysql -h mysql-0.mysql <<EOF
CREATE DATABASE prueba;
CREATE TABLE prueba.saludos (mensaje VARCHAR(250));
INSERT INTO prueba.saludos VALUES ('Bienvenidos al curso del CEP de k8s');
EOF
```

Una vez realizada la modificación en la base de datos, se eliminará el pod con el cliente MySQL. Creamos a continuación otro pod que realizará una consulta a la base de datos, pero lo hará a `mysql-read` con lo que comprobaremos que se ha realizado la sincronización a los secundarios:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never --\
  mysql -h mysql-read -e "SELECT * FROM prueba.saludos"
```

Podemos comprobar los servidores que responden a una consulta a `mysql-read` solicitando el identificador del servidor en unas cuantas iteraciones, para lo que ejecutamos de forma interactiva una consulta repetidas veces para que se muestre el id del servidor que está respondiendo y así se vea el balanceo sobre todos los nodos:

```
kubectl run mysql-client-loop --image=mysql:5.7 bash
```

```
for i in `seq 1 10`; do mysql -h mysql-read -e 'SELECT @@server_id,NOW()'; sleep 1;
```

Vídeo: Ejemplo: Despliegue de un cluster de MySQL

https://www.youtube.com/embed/aUH2x9WtJZM

Vídeo: Ejemplo: Despliegue de un cluster de MySQL

DaemonSets

El objeto DaemonSet (DS) se utiliza cuando queremos ejecutar un pod en todos los nodos del cluster o al menos en un conjunto de ellos que tienen una serie de características en común. Un DaemonSet se utiliza en algunas circunstancias muy concretas, por ejemplo:

- Ejecutar un pod en cada nodo para la monitorización del cluster: Prometheus, Sysdig, collectd, datadog, etc.
- Ejecutar un pod en cada nodo para la recolección y gestión de logs: fluentd, logstash
- Ejecutar un pod en cada nodo para el almacenamiento del cluster: ceph o glusterfs

Un ejemplo de DaemonSet tendría el siguiente aspecto:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: daemonset1
spec:
  selector:
      matchLabels:
        name: daemonset-pod
  template:               # Plantilla con las características del Pod
    metadata:
      labels:
        name: daemonset-pod
    spec:
      nodeSelector:
        type: worker-prod # Etiqueta del nodo en el que se ejecuta (opcional)
      containers:
      - name: daemon-pod
        image: ...
```

Los parámetros tienen los valores habituales anteriormente descritos y en este caso, se incluye una plantilla con la descripción del pod que se ejecutará en cada nodo (en este caso en cada nodo con etiqueta worker-prod).

Para seguir aprendiendo

- ara más información acerca de los DaemonSets puedes leer la documentación de la API.

Jobs y CronJobs

Job

El objeto Job se utiliza cuando queremos ejecutar una tarea puntual, para lo que se define el objeto y se crean todos los objetos necesarios para realizarla, principalmente creando uno o varios Pods hasta que se finaliza la tarea. Una vez se termina la tarea y de forma general, los pods permanecerán creados y no se borrarán hasta que se elimine el Job que los creó.

Un ejemplo de Job tendría el siguiente aspecto:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

En el ejemplo anterior se lanza un contenedor de la imagen perl y realiza el cálculo de Pi con una precisión de 2000 decimales utilizando este lenguaje.

Una vez lanzada la tarea, podremos ver que se crea tanto un objeto Job como un Pod, el primero aparece sin finalizar y el Pod aparece ejecutándose:

```
NAME           READY   STATUS    RESTARTS   AGE
pod/pi-jbt4r   1/1     Running   0          4s

...

NAME           COMPLETIONS   DURATION   AGE
job.batch/pi   0/1           4s         4s
```

Sin embargo, una vez que la tarea del contenedor finaliza, en este caso cuando se consigue el número Pi con dos mil decimales de precisión, el Pod se para y la tarea se marca como completada:

```
NAME           READY   STATUS      RESTARTS   AGE
pod/pi-jbt4r   0/1     Completed   0          10s

....

NAME           COMPLETIONS   DURATION   AGE
job.batch/pi   1/1           9s         10s
```

Podemos ver que no se borra el Pod, ya que lo necesitamos en muchas ocasiones para ver el resultado de la tarea. En este caso para ver el número Pi con la precisión solicitada, veríamos los logs del pod:

```
kubectl logs pi-jbt4r
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899
```

Para seguir aprendiendo

- Para más información acerca de los Jobs puedes leer la documentación de la API.

CronJob

En el caso de que la tarea que tengamos que realizar no sea puntual, sino que se tenga que repetir cada cierto tiempo conforme a un patrón, k8s ofrece el objeto CronJob, que creará tareas conforme a la periodicidad que se indique.

En el siguiente ejemplo de CronJob, ejecutamos una tarea cada minuto, en la que se muestra la fecha y hora junto al texto "Curso del CEP":

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
```

```
        template:
          spec:
            containers:
            - name: hello
              image: busybox
              imagePullPolicy: IfNotPresent
              command:
              - /bin/sh
              - -c
              - date; echo Curso del CEP
            restartPolicy: OnFailure
```

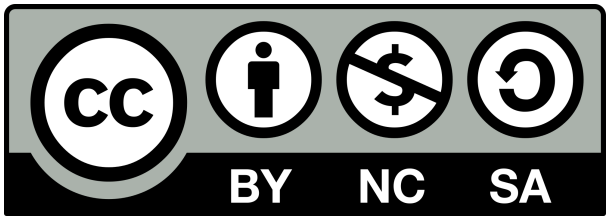
Para la definición del objeto CronJob debe especificarse un nombre y el patrón de repetición conforme al cron de UNIX, además de incluir en jobTemplate la definición del objeto Job que se desea ejecutar.

Para seguir aprendiendo

- Para más información acerca de los CronJobs puedes leer la documentación de la API.

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