PBS El Salvador
Final Boulevard Santa Elena y Boulevard
Orden de Malta, Edificio PBS
San Salvador, EL SALVADOR
Tel. + 503 2239 3000 | fax + 503 2239 3095
www.grouppbs.com

Kubernetes y OKE

PBS El Salvador
Final Boulevard Santa Elena y Boulevard
Orden de Malta, Edificio PBS
San Salvador, EL SALVADOR
Tel. + 503 2239 3000 | fax + 503 2239 3095
www.grouppbs.com

Contents

PBS El Salvador
Final Boulevard Santa Elena y Boulevard
Orden de Malta, Edificio PBS
San Salvador, EL SALVADOR
Tel. + 503 2239 3000 | fax + 503 2239 3095
www.grouppbs.com

Pasos iniciales

En la máquina virtual proporcionada ejecute esta práctica de laboratorio, veremos cómo administrar una especificación de minikube.

Minikube implementa un clúster local de Kubernetes en macOS, Linux y Windows. Los objetivos principales de minikube son ser la mejor herramienta para el desarrollo de aplicaciones locales de Kubernetes y admitir todas las características de Kubernetes que se ajusten.

1. Validación de minikube

```
$ minikube versión
 minikube version: v0.25.0
```

2. Start minicube

```
$ minikube start  --driver=docker
```

Cree una carpeta con la primera letra de su nombre y su apellido y nos ubicamos en ella

```
$ mkdir jperez
```

Pods

Un pod es un grupo de uno o más contenedores (como contenedores Docker), el almacenamiento compartido para esos contenedores y opciones sobre cómo ejecutar los contenedores. Los pods siempre están coubicados y coprogramados, y se ejecutan en un contexto compartido. Un pod modela un "host lógico" específico de la aplicación, contiene uno o más contenedores de aplicaciones que están relativamente acoplados: en un mundo previo al contenedor, se habrían ejecutado en la misma máquina física o virtual.

Hello World Pod

Cree el archivo hello-world-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: "alpine:3.5"
    command: ["echo", "Hello", "World"]
```

- Usamos Imagen alpine:3.5
- Ejecutamos el comando echo Hello World
- Le decimos a Pod que no reinicie si está terminado

Vamos a crear el Pod con kubectl CLI. Para crearlo:

```
$ kubectl create -f hello-world-pod.yaml
```

Veamos los Pods (-a es ver también los Completados):

```
$ kubectl get pods -a
```

Para ver los logs:

```
$ kubectl logs hello-world
```

Eliminemos el Pod usando la referencia de archivo:

```
$ kubectl delete -f hello-world-pod.yaml
```

También puede eliminar un Pod usando la referencia de nombre:

```
$ kubectl delete pod hello-world
```

Pod de un solo contenedor de larga duración

Un ejemplo más útil es un contenedor de larga duración. Ahora crearemos un pod con un contenedor que ejecutará nginx.

Cree la carpeta ejemplo02:

```
mkdir ejemplo02
cat > single-container-pod.yaml
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.13-alpine
    ports:
     - containerPort: 80
```

Vamos a implementarlo (puede hacer referencia a un directorio para el comando create:

```
$ kubectl create -f {tucarpeta}/ejemplo02/
```

Verá que el Pod una vez que terminade descargar la imagen, está en estado de ejecución y se mantiene en funcionamiento.

Si haces clic en el Pod no verás ningún registro, pero eso está bien porque Nginx no da ningún registro. Veremos más adelante cómo podemos usar el Nginx, pero por ahora vamos a eliminar el Pod.

```
$ kubectl delete pod nginx
```

Replica Sets

Con Pods logramos declararla definición de una unidad de despliegue, por ejemplo, una instancia de Nginx.
En un entorno de clúster, probablemente queramos tener más de una instancia de nuestras aplicaciones, por ejemplo, en aras de la redundancia y el rendimiento.
Para gestionar esta necesidad utilizaremos conjuntos de réplicas. Un conjunto de réplicasdefinelos Pods que necesitan ejecutarse y el número de sus réplicas.
Usaremos el mismo ejemplo de Nginx antes, pero ahora ejecutaremos varias instancias.

```
cat > nginx-replicaset.yaml
```

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
      template: replicaset
  template:
    metadata:
      labels:
        app: nginx
        template: replicaset
    spec:
      containers:
      - name: nginx
        image: nginx:1.13-alpine
        ports:
        - containerPort: 80
```

Vamos a implementarlo:

```
$ kubectl create -f nginx-replicaset/
```

Veamos los Pods:

```
$ kubectl get pods
```

Veamos el ReplicaSet (puede usar tanto singular como plural en la CLI):

```
$ kubectl get replicaset
```

También puedes usar alias para escribir más rápido:

```
$ kubectl get rs
```

Para ver los recursos disponibles (y sus alias):

```
      $ kubectl get
```

Podemos cambiar el número de réplicas deseadas y el controlador se asegurará de actualizar el clúster:

Podemos ampliar:

```
$ kubectl scale replicaset nginx --replicas=10
```

o reducir:

```
$ kubectl scale replicaset nginx --replicas=2
```

Si se elimina un Pod, el Replica Set Controller creará un nuevo Pod para cumplir con el número deseado de réplicas.

Entonces, si pierdo un nodo, los Pods allí se reprogramarán a otros nodos au-

tomáticamente. Vamos a eliminar el ReplicaSet:

```
$ kubectl delete replicaset nginx
```

Al eliminar el ReplicaSet se propaga la eliminación de los Pods que administra

Corriendo instancias de redis

```
        cat > redis-replicaset.yaml
```

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11-alpine
        ports:
        - containerPort: 80
```

- Cambiar el nombre del ReplicaSet y el número de réplicas
- Configurarel selector de etiquetas y las etiquetas de los Pods
- Nombre del contenedor y la imagen del contenedor (redis:3.2.8-alpine)
- Configurar el puerto del contenedor (6379)

Deployment

Con Replica Sets ahora tenemos la definición de múltiples Pods bajo una versión específica.

Para poder administrar una actualización o degradación de un servicio sin tener tiempo de inactividad durante la actualización, tenemos que manejar múltiples instancias de ambas versiones durante la actualización.

El objetivo de la implementación es controlar este proceso

Deployment Process

Usemos una definición de implementación para implementar 2 instancias Nginx en nuestro clúster.

```
$ cat nginx-deployment.yaml
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.12-alpine
        ports:
        - containerPort: 80
```

Vamos a implementarlo:

```
$ kubectl create -f nginx-deployment/
```

Verá que la implementación ha generado un conjunto de réplicas y que a su vez ge-

neró 2 pods. Veamos los Pods:

```
$ kubectl get pods
```

Veamos los ReplicaSets:

```
$ kubectl get replicasets
```

Veamos el despliegue:

```
$ kubectl get deployment
```

El despliegue se completa con 2 réplicas, pero podemos actualizar la definición del despliegue para aumentar o disminuir el número de pods y esa información se delega al conjunto de réplicas, que a su vez es responsable del número de pods en ejecución.

Como ejemplo, aumentaremos el número de Pods a 8.

```
$ kubectl scale deployment nginx --replicas=8
```

Actualización

Actualmente usamos Nginx v1.12, pero hay una nueva versión de Nginx v1.13 que queremos implementar en el clúster.

Digamos que tenemos 8 instancias de Nginx ejecutándose, el objetivo es cambiar todas ellas de v1.12 a v1.13 sin perder la disponibilidad del servicio.

```
$ kubectl set image deployment nginx *=nginx:1.13-alpine
```

Podemos ver el estado de la implementación:

```
$ kubectl rollout status deployment nginx
```

Si algo sale mal con la nueva implementación, simplemente podemos revertir la implementación a la versión anterior:

```
$ kubectl rollout undo deployment nginx
```

Vamos a limpiar la implementación:

```
$ kubectl delete deployment nginx
```

Networking Basics

Como mencionamos antes, los pods son efímeras y pueden nacer y morir durante el día.

Para poder agrupar un conjunto de Pods y hacerlos detectables sin tener que tomar en cuenta que pueden morir y regenerarse (por ejemplo, porque un nodo era defectuoso) necesitamos algo más abstracto. Aquí es donde entran los servicios.

Un servicio es básicamente un grupo de pods que consisten en un servicio (por ejemplo, un grupo de instancias de Nginx).

Cuando alguien quiere acceder a este grupo de Pods lo hace a través del servicio, que redirigirá la petición a uno de esos Pods.

En esta sección implementaremos una implementación con 3 réplicas nginx (pods) y un servicio que las conecta.

Esta es la forma más común de implementar una aplicación sin estado.

El despliegue es el mismo que antes, a continuación, se muestra la definición del servicio:

Ejecute

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.41.2/deploy/static/provider/cloud/deploy.yaml

```
minikube addons enable ingress
```

```
$ cat >  nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - name: web
    port: 80
  selector:
     app: nginx
```

Hay tres tipos de Servicios:

- **ClusterIP**: expone el servicio en una IP interna del clúster. Si elige este valor, solo se puede acceder al servicio desde el clúster (valor predeterminado).
- **NodePort**: Expone el servicio en la IP de cada nodo en un puerto estático (NodePort).
- **LoadBalancer**: expone el servicio externamente utilizando el equilibrador de carga de un proveedor de nube.
- **ExternalName**: Asigna el servicio al contenido del archivo externalName (por ejemplo, foo.bar.example.com), devolviendo un registro CNAME con su valor.

Usamos LoadBalancer , pero como estamos en minikube y no tenemos un proveedor de nube, es equivalente a NodePort.

Vamos a implementarlo:

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

Obtenga el servicio:

```
$ kubectl get service nginx
```

Obtenga el NodePort del servicio

```
$ kubectl get service nginx -o jsonpath={.spec.ports[*].nodePort}
```

Obtén el IP de minikube

```
$ minikube ip
```

Obtenga la salida de NODE_IP:NODE_PORT

```
$ curl -sL $(minikube ip):$(kubectl get service nginx -o jsonpath={.spec.ports[*].node
Port}
```

En Docker el lanzamiento de múltiples contenedores en un nodo

- A cada contenedor se le asignó una IP
- Solo los servicios expuestos podían ser vistos al Mundo exterior

En Kubernetes tenemos más de un nodo

- Queremos dos Pods que estén en diferentes máquinas para poder hablar juntos
- Pero no queremos exponer todo al mundo exterior

Esto se resuelve mediante el modelo de red.

En Kubernetes hay dos rangos de IP:

- Los rangos de Pods
- La rangos Services

A cada Pod se le asigna una IP (modelo IP por pod) de la rama Pods.
Cada Pod puede hablar con otros Pods directamente con su IP (en el mismo o diferentes Nodos)
A cada Servicio se le asigna una IP en el rango de Servicios. Esa IP no cambia durante la vida útil de los Pods conectados al Servicio
Hay varias implementaciones de la red

- `Contiv`
- `Flannel GCE`
- `L2 networks and linux bridging`
- `Nuage Networks VCS`
- `Weave Net`

Vamos a crear dos Pods y hablar entre nosotros Primero crear un pod nginx:

```
$ kubectl create -f single-container-pod.yaml
```

Obtenga la IP del Pod

```
$ kubectl get pod nginx --template={{.status.podIP}}
```

Luego cree un Pod interactivo temporal para hacer ping al Pod de arriba

```
$ kubectl run --rm ping -it --image alpine:3.5 sh wget -qO- 172.17.0.5
```

Ahora limpieza

```
$ kubectl delete -f single-container-pod
```

Pero de esta manera necesitamos saber la IP del otro Pod. ¿Qué pasa si el Pod muere y la IP cambia? Necesitamos una manera de descubrir otros Servicios. Hay dos formas:

1. Variables de entorno: Cada pod cuando se ejecuta tiene variables de entorno para cada servicio que apuntan a la IP virtual del servicio.
2. DNS: Un complemento de clúster opcional (aunque muy recomendable) es un servidor DNS. Para cada servicio crea un conjunto de registros DNS.

Usando DNS podemos simplemente hacer:

```
$ kubectl run --rm ping -it --image alpine:3.5 sh wget -qO- nginx
```

- DNS responderá para nginx con la IP del servicio nginx
- La IP virtual básicamente equilibrará la carga entre las diferentes instancias de nginx
- Si un Pod muere, otro tomará su lugar, la IP del servicio permanecerá igual y el servicio continuará funcionando como se esperaba.

Vamos a limpiar:

```
$ kubectl delete -f nginx-service/
```

Configuration Management

Para mejorar nuestro medio ambiente hay dos maneras de hacerlo:

- Variables de entorno
- Mounted files

¿Por qué contenedores configurables, por qué no almacenarlo dentro de la imagen?

- La configuración es diferente en cada entorno
- La misma imagen debe poder reutilizarse en diferentes entornos (compilar una vez, ejecutar en cualquier lugar)
- La configuración no debe grabarse dentro de la imagen, sino implementarse en tiempo de ejecución.

Vamos a crear un Pod con una variable de entorno en un archivo env-pod.yaml

```
apiVersion: v1
kind: Pod metadata:
name: env-pod spec:
restartPolicy: Never containers:
name: app
image: "alpine:3.5" command: ["env"] env:
name: HELLO
            value: "Hello from the environment"
```

Vamos a crear el Pod:

```
$ kubectl create -f env-pod.yaml
```

Veamos los registros:

```
$ kubectl logs env-pod
```

Vamos a limpiar:

```
$ kubectl delete pod env-pod
```

Vamos a crear un Pod con un archivo montado.

El contenido del archivo se puede almacenar en un ConfigMap:

```
$ cat kubernetes/08.volumemount-configmap/volumemount-configmap.yaml
apiVersion: v1
kind: ConfigMap metadata:
name: volumemount-configmap data:
key-a: bob key-b: alice key-c: |
This is a multiline
          data value
```

Y en un Pod podemos hacer referencia a ese ConfigMap para enlazarlo montarlo en un directorio:

```
apiVersion: v1
kind: Pod metadata:
name: volumemount spec:
restartPolicy: Never containers:
name: app
image: "alpine:3.5"
command: ["cat", "/config/app.conf"] volumeMounts:
mountPath: /config name: config
volumes:
name: config configMap:
name: volumemount-configmap items:
key: key-c
              path: app.conf
```

Vamos a crear el Pod:

```
$ kubectl create -f volumemount-configmap/
```

Veamos los registros:

```
$ kubectl logs volumemount
```

Vamos a limpiar:

```
$ kubectl delete -f volumemount-configmap
```

Persistent Volumes

El subsistema PersistentVolume proporciona una API para usuarios y administradores que abstrae detalles de cómo se proporciona el almacenamiento de cómo se consume.

Para ello introducimos dos nuevos recursos API

- **PersistentVolumeClaim** (PVC): es una solicitud de almacenamiento por parte de un usuario.
- **PersistentVolume** (PV): es una parte del networked storage del clúster que ha sido aprovisionada por un administrador.
- **StorageClass** Proporciona una manera para que los administradores describan las "clases" de almacenamiento que ofrecen.

Persistent Volume Claim

This is a PersistentVolumeClaim:

```
$ cat kubernetes/09.myclaim-pvc/myclaim-pvc.yaml kind: PersistentVolumeClaim
apiVersion: v1 metadata:
name: myclaim-1 labels:
temporal: "true" spec:
accessModes:
- ReadWriteOnce resources:
requests:
storage: 2Gi
```

Vamos a crearlo:

```
$ kubectl create -f kubernetes/09.myclaim-pvc/
```

Persistent Volume

Si revisa sus volúmenes persistentes ahora, verá que ya hay un PV generado y vinculado al PVC.

Así es como se ve (-o yaml significa salida en formato yaml):

```
$ kubectl get persistentvolume -o yaml kind: PersistentVolume
apiVersion: v1 metadata:
name: pvc-ded278e1-08f3-11e7-8842-0800276b3654 annotations:
volume.beta.kubernetes.io/storage-class: standard spec:
capacity: storage: 2Gi
accessModes:
- ReadWriteOnce hostPath:
path: /tmp/hostpath-provisioner/pvc-ded278e1-08f3-11e7-8842-0800276b3654 persistentVolumeRe-
claimPolicy: "Delete"
```

Storage Class

El PV fue generado por una clase de almacenamiento existente en minikube.

```
$ kubectl get storageclass
NAME   PROVISIONER  AGE
standard (default) k8s.io/minikube-hostpath 4d
```

En este caso, minikube utiliza rutas de host dentro del directorio / tmp/ y las proporciona como almacenamiento para las notificaciones.

Existen otras implementaciones:

- AWS uses EBS volumes as storageGCE
- Uses PD volumes as storage
- etc

Pero el PVC es el mismo que se extrae del proveedor de almacenamiento.

Eliminemos PersistentVolumeClaim:

```
$ kubectl delete persistentvolumeclaim myclaim-1
```

Ghost

Como ejemplo, implementaremos Ghost:

```
...
spec:
replicas: 1 template:
metadata:
...
spec:
containers:
- image: ghost:0.10 name: ghost
...
volumeMounts:
- name: ghost
mountPath: /var/lib/ghost subPath: "ghost"
volumes:
- name: ghost persistentVolumeClaim:
claimName: ghost-claim
```

Vamos a implementarlo:

```
$ kubectl create -f kubernetes/10.ghost/
```

Objetos de Ghost

```
$ kubectl get pvc,pv,pod,svc
```

Vamos a eliminarlo:

```
$ kubectl delete -f kubernetes/10.ghost/
```

Memory and CPU Quotas

Kubernetes programa un Pod para que se ejecute en un nodo solo si el nodo tiene suficiente CPU y RAM disponibles para satisfacer el total de CPU y RAM solicitadas por todos los contenedores del Pod.

- **recursos:requests** field: Al crear un Pod, puede solicitar recursos de CPU y RAM para los contenedores que se ejecutan en el Pod.

Además, como un contenedor se ejecuta en un nodo, Kubernetes no permite que la CPU y la RAM consumidas por el contenedor excedan los límites especificados para el contenedor.

Si un contenedor excede su límite de RAM, se termina. Si un contenedor excede su límite de CPU, se convierte en un candidato para que se limite el uso de su CPU.

- **recursos:limits** field: También puede establecer límites para los recursos de CPU y RAM.

Este es un ejemplo de asignación de recursos de CPU y RAM:

```
$ cat kubernetes/11.cpu-ram-limited-pod/cpu-ram-limited-pod.yaml apiVersion: v1
kind: Pod metadata:
name: cpu-ram-limited spec:
containers:
name: nginx
image: nginx:1.13-alpine resources:
requests: memory: "64Mi" cpu: "250m"
limits:
memory: "128Mi"
cpu: "1"
```

El archivo de configuración para el Pod solicita 250 miliCPU y 64 mebibytes de RAM. También establece límites superiores de 1 CPU y 128 mebibytes de RAM.

(Como referencia: megabyte = 1000 bytes, mebibyte = 1024 bytes)

Vamos a implementarlo:

```
$ kubectl create -f kubernetes/11.cpu-ram-limited-pod/
```

Puede comprobar el uso de CPU y memoria por nodo describiendo los nodos:

```
$ kubectl describe node
...
Namespace    Name   CPU Requests CPU Limits Memory
```

```
...
default      cpu-ram-limited    250m (12%) 1 (50%) 64Mi (3%)
...
```

Vamos a eliminarlo:

```
$ kubectl delete pod cpu-ram-limited
```

Health Checks

Liveness Probe: El kubelet utiliza sondas de vivaz para saber cuándo reiniciar un contenedor.

Por ejemplo, las sondas de vida podrían detectar un punto muerto, donde una aplicación se está ejecutando, pero no puede progresar. Reiniciar un contenedor en tal estado puede ayudar a que la aplicación esté más disponible a pesar de los errores.

Readiness Probe: El kubelet utiliza sondas de preparación para saber cuándo un contenedor está listo para comenzar a aceptar tráfico.

Un Pod se considera listo cuando todos sus contenedores están listos. Un uso de esta señal es controlar qué Pods se utilizan como backends para los Servicios. Cuando un Pod no está listo, se elimina de los equilibradores de carga de servicio.

Liveness Probe

```
$ cat kubernetes/12.liveness-probe-pod/liveness-probe-pod.yaml apiVersion: v1
kind: Pod metadata:
name: liveness-probe spec:
containers:
name: liveness-probe args:
/server
image: gcr.io/google_containers/liveness livenessProbe:
httpGet:
path: /healthz port: 8080
failureThreshold: 2
initialDelaySeconds: 3
periodSeconds: 2
```

La imagen liveness devuelve OK (200) en /healthz durante 10 segundos y luego devuelve error (500).
Liveness Probe está comprobando si /healthz en el puerto 8080 está bien, comenzando después de 3 segundos de retraso, repitiendo cada 2 segundos. Debe fallar al menos 2 veces consecutivas.

Vamos a implementarlo:

```
$ kubectl create -f kubernetes/12.liveness-probe-pod/
```

Veamos el Pod:

```
$ kubectl get pod
```

Vamos a eliminarlo:

```
$ kubectl delete pod liveness-probe
```

Readiness Probe

```
$ cat kubernetes/13.readiness-probe-pod/readiness-probe-pod.yaml apiVersion: v1
kind: Pod metadata: labels:
app: readiness-probe name: readiness-probe
spec:
containers:
name: readiness-probe args:
/server
image: gcr.io/google_containers/liveness readinessProbe:
httpGet:
path: /healthz port: 8080
failureThreshold: 2
initialDelaySeconds: 3
periodSeconds: 2
```

Igual que la vida, pero ahora en lugar de reiniciar el Pod, lo marca como insalubre y lo desconecta de los Servicios.

Vamos a implementarlo:

```
$ kubectl create -f kubernetes/13.readiness-probe-pod/
```

Veamos el Pod:

```
$ kubectl get pod
```

Veamos el Servicio:

```
$ kubectl describe svc readiness-probe
```

Vamos a eliminarlo:

```
$ kubectl delete -f kubernetes/13.readiness-probe-pod/
```

Kubernetes Addons

En esta sección hablaremos sobre los complementos de Kubernetes:

- Dashboard
- StorageClass
- DNS
- Heapster
- Ingress

Minikube Addons

Minikube viene con algunos complementos incluidos que se pueden habilitar / deshabilitar:

```
$ minikube addons list
```

Dashboard

Dashboard es una interfaz de usuario web de uso general para Clústeres de Kubernetes
Para abrirlo:

```
$ minikube dashboard
```

StorageClass

Minikube viene con una clase de almacenamiento predeterminada que crea Persistent-Volumes.

Lo hemos visto en acción en la sección Volúmenes persistentes, cuando creamos un PersistentVolumeClaim y generó el Volumen persistente.

DNS

kube-dns es el servicio DNS dentro del clúster. Esto responde cuando solicitamos Servicios por nombres.

Lo hemos visto en acción en la sección Conceptos básicos de redes cuando hicimos wget -qO- nginx .

Heapster

Heapster es responsable de supervisar el consumo de recursos en el clúster.

```
$ minikube addons enable heapster
```

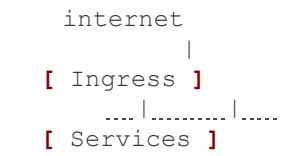
El addon Heapster también incluye Grafana frontend al que se puede acceder con:

```
$ minikube addons open heapster
```

Ingress Controller

Normalmente, los servicios y los pods solo tienen direcciones IP enrutables por la red del clúster. Todo el tráfico que termina en un enrutador de borde se deja caer o se reenvía a otro lugar. Conceptualmente, esto podría verse así:

Un Ingress es una colección de reglas que permiten que las conexiones entrantes lleguen a los servicios de clúster.



Sepuede configurarpara proporcionar a los servicios URL accesibles externamente, equilibrar la carga de las transacciones, terminar SSL, ofrecer alojamiento virtual basado en nombres, etc.

Primero habilite Ingress:

```
$ minikube addons enable ingress
```

Aquí hay un ejemplo de un Ingress:

```
$ cat kubernetes/14.coffee-ingress/coffee-ingress.yaml apiVersion: extensions/v1beta1
kind: Ingress metadata:
name: coffee-ingress spec:
rules:
host: coffee.example.com http:
paths:
backend:
serviceName: tea servicePort: 80
path: /tea
backend:
serviceName: coffee servicePort: 80
path: /coffee
```

Vamos a implementarlo:

```
$ kubectl create -f kubernetes/14.coffee-ingress/
```

Creará dos despliegues, coffee con 2 réplicas, tea con 3 réplicas.

Para cada uno hay un servicio de coffee y tea respectivamente (Los servicios son tipo

ClusterIP , solo visible dentro del clúster)

Creará una regla de entrada que hace lo siguiente:

http://coffee.example.com/coffee > `coffee` Servicehttp://coffee.example.com/tea > `tea` Service

Ingress Controller escucha en el puerto 80 y redirige todos los traffic dependiendo de las reglas.

Configure /etc/hosts con la siguiente línea (reemplazar 192.168.99.100 real con minikube ip)

192.168.99.100 coffee.example.com

Vamos a eliminarlo:

```
$ kubectl delete -f kubernetes/14.coffee-ingress/
```

Autoscaling

Hay dos tipos de escalado automático:

- Pod Autoscaling
- Cluster Autoscaling

Y hay dos formas de escalar algo:

- Vertically (Grow bigger)
- Horizontally (Grow in number)

Horizontal Pod Autoscaler

Vamos a implementar nginx

Chart de nuevo. Ahora vamos a

crear HPA:

```
kubectl autoscale deployment nginx --min 1 --max 10 --cpu-percent=50
```

Y exec dentro del pod nginx y

hacer algo de consumo de CPU

```
dd if=/dev/zero of=/dev/null
```

Esto provocará que HPA agre-

gue más instancias.