Laboratorio Pods

Un pod es una colección de contenedores que comparten una red y un espacio de nombres de montaje y es la unidad básica de implementación en Kubernetes. Todos los contenedores en un pod están programados en el mismo nodo.

Para iniciar un pod utilizando la imagen del contenedor `mhausenblas/simpleservice:0.5.0` y exponiendo una API HTTP en el puerto `9876`, ejecute:

```
$ kubectl run sise --image=mhausenblas/simpleservice:0.5.0 --port=9876
```

Ahora podemos ver que el pod se está ejecutando:

```
$ kubectl get pods

NAME                        READY     STATUS    RESTARTS   AGE

sise-3210265840-k705b       1/1       Running   0          1m

$ kubectl describe pod sise-3210265840-k705b | grep IP:

IP:                     172.17.0.3
```

Desde el clúster (por ejemplo, a través de `minishift ssh`) se puede acceder a este pod a través de la IP del pod `172.17.0.3`, que hemos aprendido del comando `kubectl describe`:

```
[cluster] $ curl 172.17.0.3:9876/info

{"host": "172.17.0.3:9876", "version": "0.5.0", "from": "172.17.0.1"}
```

Tenga en cuenta que `kubectl run` crea un deployment , por lo que para deshacerse del pod debe ejecutar `kubectl delete deployment sise`.

Usando el archivo de configuración

También puede crear un pod desde un archivo de configuración. En este caso, el pod ejecuta la imagen `simpleservice` junto con un contenedor `CentOS`:

```
$ kubectl apply -f    kubernetes-labs2/pods/pod.yaml

$ kubectl get pods

NAME                        READY     STATUS    RESTARTS   AGE

twocontainers               2/2       Running   0          7s
```

Ahora podemos ejecutar en el contenedor `CentOS` y acceder al `simpleservice` localhost:

```
$ kubectl exec twocontainers -c shell -i -t -- bash

[root@twocontainers /]# curl -s localhost:9876/info

{"host": "localhost:9876", "version": "0.5.0", "from": "127.0.0.1"}
```

Especifique el campo `resources` en el pod para influir en la cantidad de CPU y/o RAM que puede usar un contenedor en un pod (aquí: `64MB` de RAM y `0.5` CPU):

```
$ kubectl create -f kubernetes-labs2/pods/constraint-pod.yaml

$ kubectl describe pod constraintpod

...

Containers:

  sise:

    ...

    Limits:

      cpu:      500m

      memory:   64Mi
```

```
    Requests:
      cpu:      500m
      memory:   64Mi
...
```

Para eliminar todos los pods creados, simplemente ejecute:

```
$ kubectl delete pod twocontainers
$ kubectl delete pod constraintpod
```

Laboratorio Labels

Las labels son el mecanismo que utiliza para organizar los objetos de Kubernetes. Una etiqueta es un par clave-valor con ciertas restricciones relacionadas con la longitud y los valores permitidos, pero sin ningún significado predefinido. Por lo tanto, puede elegir las etiquetas como mejor le parezca, por ejemplo, para expresar entornos como 'este pod se está ejecutando en producción' o propiedad, como 'el departamento X posee ese pod'.

Creemos un pod que inicialmente tenga una etiqueta (env=development):

```
$ kubectl apply -f  kubernetes-labs2/labels/pod.yaml

$ kubectl get pods --show-labels

NAME      READY    STATUS    RESTARTS   AGE    LABELS

labelex   1/1      Running   0          10m    env=development
```

En el comando anterior get pods , tenga en cuenta la opcion --show-labels que genera las etiquetas de un objeto en una columna adicional.

Puede agregar una etiqueta al pod como:

```
$ kubectl label pods labelex owner=miempresa

$ kubectl get pods --show-labels

NAME      READY    STATUS    RESTARTS   AGE    LABELS

labelex    1/1      Running   0          16m    env=development,owner=miempresa
```

Para usar una etiqueta para filtrar, por ejemplo, para enumerar solo pods que tengan un valor owner igual miempresa, use la opcion --selector:

```
$ kubectl get pods --selector owner=miempresa

NAME     READY    STATUS    RESTARTS   AGE

labelex   1/1      Running   0          27m
```

La opción --selector se puede abreviar -l, por lo que para seleccionar pods que estén etiquetados env=development, haga:

```
$ kubectl get pods -l env=development
NAME     READY    STATUS    RESTARTS   AGE
labelex  1/1      Running   0          27m
```

A menudo, los objetos de Kubernetes también admiten selectores basados en conjuntos.

Vamos a lanzar otro pod que tiene dos etiquetas (env=productiony owner=miempresa):

```
$ kubectl apply -f  kubernetes-labs2/labels/anotherpod.yaml
```

Ahora, enumeremos todos los pods que están etiquetados con env=development o con env=production:

```
$ kubectl get pods -l 'env in (production, development)'
NAME         READY    STATUS    RESTARTS   AGE
labelex      1/1      Running   0          43m
labelexother 1/1      Running   0          3m
```

Otros verbos también admiten la selección de etiquetas, por ejemplo, puede eliminar estos dos pods con:

```
$ kubectl delete pods -l 'env in (production, development)'
```

Tenga en cuenta que esto destruirá cualquier pod con esas etiquetas.

También puede eliminarlos directamente, a través de sus nombres, con:

```
$ kubectl delete pods labelex

$ kubectl delete pods labelexother
```

Tenga en cuenta que las etiquetas no están restringidas a las pods. De hecho, puede aplicarlos a todo tipo de objetos, como nodos o servicios.

Laboratorio Health Checks

Para verificar si un contenedor en un pod está sano y listo para servir el tráfico, Kubernetes proporciona una variedad de mecanismos de verificación de salud. Los controles de estado, o **sondas** como se les llama en Kubernetes, se llevan a cabo por el kubelet para determinar cuándo reiniciar un contenedor (para `livenessProbe`) y utilizado por los servicios y despliegues para determinar si una pod debe recibir tráfico (para `readinessProbe`).

Nos centraremos en las comprobaciones de estado de HTTP a continuación. Tenga en cuenta que es responsabilidad del desarrollador de la aplicación exponer una URL que el kubelet puede usar para determinar si el contenedor está en buen estado (y potencialmente listo).

Creemos un pod que exponga un punto final `/health`, respondiendo con un código `200` de estado HTTP :

```
$ kubectl apply -f /kubernetes-labs2/healthz/pod.yaml
```

En la especificación del pod hemos definido lo siguiente:

```
livenessProbe:
  initialDelaySeconds: 2
  periodSeconds: 5
  httpGet:
    path: /health
    port: 9876
```

Arriba significa que Kubernetes comenzará a verificar el punto final `/health`, después de esperar inicialmente 2 segundos, cada 5 segundos.

Si ahora miramos la cápsula, podemos ver que se considera saludable:

```
$ kubectl describe pod hc

Name:                   hc

Namespace:              default

Security Policy:        anyuid

Node:                   192.168.99.100/192.168.99.100

Start Time:             Tue, 25 Apr 2017 16:21:11 +0100

Labels:                 <none>

Status:                 Running

...

Events:

  FirstSeen     LastSeen        Count   From                            Subob
jectPath            Type            Reason          Message

  ---------     --------        -----   ----                            -----
--------            --------        ------          -------

  3s            3s              1       {default-scheduler }
Normal          Scheduled       Successfully assigned hc to 192.168.99.100

  3s            3s              1       {kubelet 192.168.99.100}        spec.
containers{sise}    Normal          Pulled          Container image "mhausenbl
as/simpleservice:0.5.0"

already present on machine

  3s            3s              1       {kubelet 192.168.99.100}        spec.
containers{sise}    Normal          Created         Created container with doc
ker id 8a628578d6ad; Security:[seccomp=unconfined]

  2s            2s              1       {kubelet 192.168.99.100}        spec.
containers{sise}    Normal          Started         Started container with doc
ker id 8a628578d6ad
```

Ahora lanzamos un pod malo , es decir, un pod que tiene un contenedor que aleatoriamente (en el rango de tiempo de 1 a 4 segundos) no devuelve un código 200:

```
$ kubectl apply -f /kubernetes-labs2/healthz/badpod.yaml
```

Al observar los eventos del pod malo, podemos ver que la comprobación de estado falló:

```
$ kubectl describe pod badpod

...

Events:

  FirstSeen     LastSeen        Count   From                            Subob
jectPath            Type            Reason          Message

  ---------     --------        -----   ----                            -----
--------            --------        ------          -------

  1m            1m              1       {default-scheduler }
Normal          Scheduled       Successfully assigned badpod to 192.168.99.10
0

  1m            1m              1       {kubelet 192.168.99.100}        spec.
containers{sise}   Normal          Created         Created container with doc
ker id 7dd660f04945; Security:[seccomp=unconfined]

  1m            1m              1       {kubelet 192.168.99.100}        spec.
containers{sise}   Normal          Started         Started container with doc
ker id 7dd660f04945

  1m            23s             2       {kubelet 192.168.99.100}        spec.
containers{sise}   Normal          Pulled          Container image "mhausenbl
as/simpleservice:0.5.0" already present on machine

  23s           23s             1       {kubelet 192.168.99.100}        spec.
containers{sise}   Normal          Killing         Killing container with doc
ker id 7dd660f04945: pod "badpod_default(53e5c06a-29cb-11e7-b44f-be3e8f4350ff
)" container "sise" is unhealthy, it will be killed and re-created.
```

```
  23s           23s             1       {kubelet 192.168.99.100}        spec.
containers{sise}   Normal          Created         Created container with doc
ker id ec63dc3edfaa; Security:[seccomp=unconfined]

  23s           23s             1       {kubelet 192.168.99.100}        spec.
containers{sise}   Normal          Started         Started container with doc
ker id ec63dc3edfaa

  1m            18s             4       {kubelet 192.168.99.100}        spec.
containers{sise}   Warning         Unhealthy       Liveness probe failed: Get
http://172.17.0.4:9876/health: net/http: request canceled (Client.Timeout exc
eeded while awaiting headers)
```

Esto también se puede verificar de la siguiente manera:

```
$ kubectl get pods

NAME                      READY     STATUS    RESTARTS   AGE

badpod                    1/1       Running   4          2m

hc                        1/1       Running   0          6m
```

Desde arriba, puede ver que `badpod` ya se había reiniciado 4 veces, ya que la comprobación de estado falló.

Además de `livenessProbe`, también puede especificar a `readinessProbe`, que se puede configurar de la misma manera pero tiene un caso de uso y una semántica diferentes: se utiliza para verificar la fase de inicio de un contenedor en el pod.

Imagine un contenedor que carga algunos datos del almacenamiento externo, como S3 o una base de datos que necesita inicializar algunas tablas. En este caso, desea indicar cuándo el contenedor está listo para servir el tráfico.

Creemos un pod con un `readinessProbe` que se active después de 10 segundos:

```
$ kubectl apply -f /kubernetes-labs2/healthz/ready.yaml
```

Al observar los eventos del pod, podemos ver que, eventualmente, el pod está listo para servir el tráfico:

```
$ kubectl describe pod ready

...

Conditions:
[0/1888]

  Type           Status

  Initialized    True

  Ready          True

  PodScheduled   True

...
```

Puede eliminar todos los pods creados con:

```
$ kubectl delete pod/hc pod/ready pod/badpod
```

Lab Deployments

Un deployment es un supervisor para los pods , que le brinda un control detallado sobre cómo y cuándo se implementa una nueva versión de pod, así como de nuevo a un estado anterior.

Creemos un deployment llamado `sise-deploy` que supervise dos réplicas de un pod, así como un conjunto de réplicas:

```
$ kubectl apply -f kubernetes-labs2/deployments/d09.yaml
```

Puede echar un vistazo a la implementación, así como al conjunto de réplicas y los pods que la implementación cuida de esta manera:

```
$ kubectl get deploy
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
sise-deploy   2         2         2            2           10s
$ kubectl get rs
NAME                     DESIRED   CURRENT   READY     AGE
sise-deploy-3513442901   2         2         2         19s
$ kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
sise-deploy-3513442901-cndsx   1/1       Running   0          25s
sise-deploy-3513442901-sn74v   1/1       Running   0          25s
```

Tenga en cuenta los nombres de los conjuntos de pods y réplicas, derivados del nombre del deployment.

En este momento, los contenedores `sise` que se ejecutan en los pods están configurados para devolver la versión `0.9`. Verifiquemos eso desde dentro del clúster (usando `kubectl describe` primero para obtener la IP de uno de los pods):

```
[cluster] $ curl 172.17.0.3:9876/info

{"host": "172.17.0.3:9876", "version": "0.9", "from": "172.17.0.1"}
```

Veamos ahora qué sucede si cambiamos esa versión a `1.0` un deployment actualizado :

```
$ kubectl apply -f kubernetes-labs2/deployments/d10.yaml

deployment "sise-deploy" configured
```

Tenga en cuenta que podría haber utilizado alternativamente `kubectl edit deploy/sise-deploy` para lograr lo mismo editando manualmente el deployment.

Lo que ahora vemos es el lanzamiento de dos nuevos pods con la versión actualizada `1.0`, así como los dos pods antiguos con la versión que se `0.9` está terminando:

```
$ kubectl get pods

NAME                           READY     STATUS        RESTARTS   AGE

sise-deploy-2958877261-nfv28   1/1       Running       0          25s

sise-deploy-2958877261-w024b   1/1       Running       0          25s

sise-deploy-3513442901-cndsx   1/1       Terminating   0          16m

sise-deploy-3513442901-sn74v   1/1       Terminating   0          16m
```

Además, la implementación ha creado un nuevo conjunto de réplicas:

```
$ kubectl get rs

NAME                     DESIRED   CURRENT   READY     AGE

sise-deploy-2958877261   2         2         2         4s

sise-deploy-3513442901   0         0         0         24m
```

Tenga en cuenta que durante la implementación puede verificar el progreso utilizando `kubectl rollout status deploy/sise-deploy`.

Para verificar que si la nueva version `1.0` está realmente disponible, ejecutamos desde dentro del clúster (nuevamente usando `kubectl describe` para obtener la IP de uno de los pods):

```
[cluster] $ curl 172.17.0.5:9876/info

{"host": "172.17.0.5:9876", "version": "1.0", "from": "172.17.0.1"}
```

Un historial de todas las implementaciones está disponible a través de:

```
$ kubectl rollout history deploy/sise-deploy

deployments "sise-deploy"

REVISION        CHANGE-CAUSE

1               <none>

2               <none>
```

Si hay problemas en la implementación, Kubernetes retrocederá automáticamente a la versión anterior, sin embargo, también puede retroceder explícitamente a una revisión específica, como en nuestro caso a la revisión 1 (la versión original del pod):

```
$ kubectl rollout undo deploy/sise-deploy --to-revision=1

deployment "sise-deploy" rolled back

$ kubectl rollout history deploy/sise-deploy

deployments "sise-deploy"

REVISION        CHANGE-CAUSE

2               <none>

3               <none>



$ kubectl get pods

NAME                           READY     STATUS    RESTARTS   AGE

sise-deploy-3513442901-ng8fz   1/1       Running   0          1m

sise-deploy-3513442901-s8q4s   1/1       Running   0          1m
```

En este momento, volvemos a donde comenzamos, con dos nuevos pods que vuelven a servir la versión `0.9`.

Finalmente, para limpiar, eliminamos la implementación y con ella los conjuntos de réplicas y pods que supervisa:

```
$ kubectl delete deploy sise-deploy

deployment "sise-deploy" deleted
```

Laboratorio Services

Un servicio es una abstracción para pods, que proporciona una dirección IP virtual (VIP) estable. Si bien los pods pueden ir y venir y con él sus direcciones IP, un servicio permite a los clientes conectarse de manera confiable a los contenedores que se ejecutan en el pod utilizando el VIP.

En VIP virtual significa que no es una dirección IP real conectada a una interfaz de red, pero su propósito es simplemente reenviar el tráfico a uno o más pods. Mantener actualizada la asignación entre el VIP y los pods es el trabajo de kube-proxy , un proceso que se ejecuta en cada nodo, que consulta al servidor API para conocer los nuevos servicios en el clúster.

Creemos un pod supervisado por un RC y un servicio junto con él:

```
$ kubectl apply -f    kubernetes-labs2/services/rc.yaml

$ kubectl apply -f    kubernetes-labs2/services/svc.yaml
```

Ahora tenemos el pod supervisado funcionando:

```
$ kubectl get pods -l app=sise

NAME         READY    STATUS    RESTARTS   AGE

rcsise-6nq3k   1/1      Running   0          57s


$ kubectl describe pod rcsise-6nq3k

Name:                rcsise-6nq3k

Namespace:           default

Security Policy:     restricted

Node:                localhost/192.168.99.100

Start Time:          Tue, 25 Apr 2017 14:47:45 +0100

Labels:              app=sise

Status:              Running
```

```
IP:                 172.17.0.3
Controllers:        ReplicationController/rcsise
Containers:
...
```

Puede, desde dentro del clúster, acceder al pod directamente a través de su IP asignada 172.17.0.3:

```
[cluster] $ curl 172.17.0.3:9876/info
{"host": "172.17.0.3:9876", "version": "0.5.0", "from": "172.17.0.1"}
```

Sin embargo, esto no es aconsejable, como se mencionó anteriormente, ya que las IP asignadas a los pods pueden cambiar. Por lo tanto, ingrese simpleservice que hemos creado:

```
$ kubectl get svc
NAME            CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
simpleservice   172.30.228.255  <none>        80/TCP              5m


$ kubectl describe svc simpleservice
Name:               simpleservice
Namespace:          default
Labels:             <none>
Selector:           app=sise
Type:               ClusterIP
IP:                 172.30.228.255
Port:               <unset> 80/TCP
Endpoints:          172.17.0.3:9876
Session Affinity:   None
```

```
No events.
```

El servicio realiza un seguimiento de los pods a los que reenvía el tráfico a través de la etiqueta, en nuestro caso app=sise.

Desde dentro del clúster ahora podemos acceder así simpleservice:

```
[cluster] $ curl 172.30.228.255:80/info

{"host": "172.30.228.255", "version": "0.5.0", "from": "10.0.2.15"}
```

¿Qué hace que el VIP 172.30.228.255 reenvíe el tráfico al pod?

La respuesta es: IPtables , que es esencialmente una larga lista de reglas que le dice al kernel de Linux qué hacer con un determinado paquete de IP.

Al observar las reglas que conciernen a nuestro servicio (ejecutado en un nodo de clúster) se obtienen:

```
[cluster] $ sudo iptables-save | grep simpleservice

-A KUBE-SEP-4SQFZS32ZVMTQEZV -s 172.17.0.3/32 -m comment --comment "default/simpleservice:" -j KUBE-MARK-MASQ

-A KUBE-SEP-4SQFZS32ZVMTQEZV -p tcp -m comment --comment "default/simpleservice:" -m tcp -j DNAT --to-destination 172.17.0.3:9876

-A KUBE-SERVICES -d 172.30.228.255/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -j KUBE-SEP-4SQFZS32ZVMTQEZV
```

Arriba puede ver las cuatro reglas que kube-proxy afortunadamente se han agregado a la tabla de enrutamiento, esencialmente indicando que el tráfico
TCP 172.30.228.255:80 debe reenviarse 172.17.0.3:9876, que es nuestro pod.

Ahora agreguemos un segundo pod ampliando el RC que lo supervisa:

```
$ kubectl scale --replicas=2 rc/rcsise

replicationcontroller "rcsise" scaled

$ kubectl get pods -l app=sise

NAME          READY     STATUS    RESTARTS   AGE

rcsise-6nq3k   1/1       Running   0          15m

rcsise-nv8zm   1/1       Running   0          5s
```

Cuando ahora verificamos nuevamente las partes relevantes de la tabla de enrutamiento, notamos la adición de un montón de reglas de IPtables:

```
[cluster] $ sudo iptables-save | grep simpleservice

-A KUBE-SEP-4SQFZS32ZVMTQEZV -s 172.17.0.3/32 -m comment --comment "default/simpleservice:" -j KUBE-MARK-MASQ

-A KUBE-SEP-4SQFZS32ZVMTQEZV -p tcp -m comment --comment "default/simpleservice:" -m tcp -j DNAT --to-destination 172.17.0.3:9876

-A KUBE-SEP-PXYYII6AHMUWKLYX -s 172.17.0.4/32 -m comment --comment "default/simpleservice:" -j KUBE-MARK-MASQ

-A KUBE-SEP-PXYYII6AHMUWKLYX -p tcp -m comment --comment "default/simpleservice:" -m tcp -j DNAT --to-destination 172.17.0.4:9876

-A KUBE-SERVICES -d 172.30.228.255/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-4SQFZS32ZVMTQEZV

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -j KUBE-SEP-PXYYII6AHMUWKLYX
```

En la lista de la tabla de enrutamiento anterior, vemos reglas para el servicio de pod recién creado 172.17.0.4:9876 y una regla adicional:

```
-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleserv
ice:" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-4SQFZS3
2ZVMTQEZV
```

Esto hace que el tráfico al servicio se divida por igual entre nuestros dos pods al invocar el modulo statistics de IPtables.

Puede eliminar todos los recursos creados haciendo:

```
$ kubectl delete svc simpleservice

$ kubectl delete rc rcsise
```

Laboratorio Service Discovery

El descubrimiento de servicios es el proceso de descubrir cómo conectarse a un servicio . Si bien existe una opción de descubrimiento de servicios basada en variables de entorno disponibles, es preferible el descubrimiento de servicios basado en DNS. Tenga en cuenta que DNS es un complemento de clúster, así que asegúrese de que su distribución de Kubernetes proporcione uno o instálelo usted mismo.

Creemos un servicio llamado `thesvc` y un RC que supervise algunos pods junto con él:

```
$ kubectl apply -f kubernetes-labs2/sd/rc.yaml

$ kubectl apply -f kubernetes-labs2/sd/svc.yaml
```

Ahora queremos conectarnos al servicio `thesvc` desde dentro del clúster, por ejemplo, desde otro servicio. Para simular esto, creamos un pod de salto en el mismo espacio de nombres (`default` ya que no especificamos nada más):

```
$ kubectl apply -f kubernetes-labs2/sd/jumpod.yaml
```

El complemento DNS se asegurará de que nuestro servicio este `thesvc` disponible a través del FQDN `thesvc.default.svc.cluster.local` desde otros pods en el clúster:

```
$ kubectl exec -it jumpod -c shell -- ping thesvc.default.svc.cl
uster.local

PING thesvc.default.svc.cluster.local (172.30.251.137) 56(84) by
tes of data.

...
```

La respuesta al `ping` nos dice que el servicio está disponible a través de la IP del clúster `172.30.251.137`. Podemos conectarnos directamente y consumir el servicio (en el mismo espacio de nombres) así:

```
 $ kubectl exec -it jumpod -c shell -- curl http://thesvc/info

{"host": "thesvc", "version": "0.5.0", "from": "172.17.0.5"}
```

Tenga en cuenta que la dirección IP `172.17.0.5` anterior es la dirección IP interna del clúster del pod de salto.

Para acceder a un servicio que se implementa en un espacio de nombres diferente del que está accediendo a él, use un FQDN en:

`$SVC.$NAMESPACE.svc.cluster.local`.

Veamos cómo funciona eso creando:

1. un espacio de nombres `other`
2. un servicio `thesvc` en espacio de nombres`other`
3. un RC que supervisa los pods, también en el espacio de nombres `other`

```
$ kubectl apply -f kubernetes-labs2/sd/other-ns.yaml

$ kubectl apply -f kubernetes-labs2/sd/other-rc.yaml

$ kubectl apply -f kubernetes-labs2/sd/other-svc.yaml
```

Ahora estamos en condiciones de consumir el servicio `thesvc` en el espacio de nombres `other` desde el espacio de nombres `default` (nuevamente a través del pod de salto):

```
$ kubectl exec -it jumpod -c shell -- curl http://thesvc.other/i
nfo

{"host": "thesvc.other", "version": "0.5.0", "from": "172.17.0.5
"}
```

En resumen, el descubrimiento de servicios basado en DNS proporciona una forma flexible y genérica de conectarse a los servicios en todo el clúster.

Puedes destruir todos los recursos creados con:

```
$ kubectl delete pods jumpod

$ kubectl delete svc thesvc

$ kubectl delete rc rcsise

$ kubectl delete ns other
```

Tenga en cuenta que eliminar un espacio de nombres destruirá todos los recursos en su interior.

Laboratorio Environment Variables

Puede establecer variables de entorno para contenedores que se ejecutan en un pod y, además, Kubernetes expone ciertas informaciones de tiempo de ejecución mediante variables de entorno automáticamente.

Vamos a lanzar un pod en el que pasamos una variable de entorno

`SIMPLE_SERVICE_VERSION` con el valor `1.0`:

```
$ kubectl apply -f kubernetes-labs2/envs/pod.yaml

$ kubectl describe pod envs | grep IP:

IP:                     172.17.0.3
```

Ahora, verifiquemos desde dentro del clúster si la aplicación que se ejecuta en el pod ha recogido la variable de entorno `SIMPLE_SERVICE_VERSION`:

```
[cluster] $ curl 172.17.0.3:9876/info

{"host": "172.17.0.3:9876", "version": "1.0", "from": "172.17.0.1"}
```

Y de hecho, ha recogido la variable de entorno proporcionada por el usuario, ya que la respuesta predeterminada sería `"version": "0.5.0"`.

Puede verificar qué variables de entorno proporciona Kubernetes automáticamente (desde dentro del clúster, utilizando un punto final dedicado que la aplicación expone):

```
[cluster] $ curl 172.17.0.3:9876/env

{"version": "1.0", "env": "{'HOSTNAME': 'envs', 'DOCKER_REGISTRY_SERVICE_PORT': '5000', 'KUBERNETES_PORT_443_TCP_ADDR': '172.30.0.1', 'ROUTER_PORT_80_TCP_PROTO': 'tcp', 'KUBERNETES_PORT_53_UDP_PROTO': 'udp', 'ROUTER_SERVICE_HOST': '172.30.246.127', 'ROUTER_PORT_1936_TCP_PROTO': 'tcp', 'KUBERNETES_SERVICE_PORT_DNS': '53', 'DOCKER_REGISTRY_PORT_5000_TCP_PORT': '5000', 'PATH': '/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'ROUTER_SERVICE_PORT_443
```

```
_TCP': '443', 'KUBERNETES_PORT_53_TCP': 'tcp://172.30.0.1:53', 'KUBERNETES_SERVICE_PORT': '443',
'ROUTER_PORT_80_TCP_ADDR': '172.30.246.127', 'LANG': 'C.UTF-8', 'KUBERNETES_PORT_53_TCP_ADDR': '
172.30.0.1', 'PYTHON_VERSION': '2.7.13', 'KUBERNETES_SERVICE_HOST': '172.30.0.1', 'PYTHON_PIP_VE
RSION': '9.0.1', 'DOCKER_REGISTRY_PORT_5000_TCP_PROTO': 'tcp', 'REFRESHED_AT': '2017-04-24T13:50
', 'ROUTER_PORT_1936_TCP': 'tcp://172.30.246.127:1936', 'KUBERNETES_PORT_53_TCP_PROTO': 'tcp', '
KUBERNETES_PORT_53_TCP_PORT': '53', 'HOME': '/root', 'DOCKER_REGISTRY_SERVICE_HOST': '172.30.1.1
', 'GPG_KEY': 'C01E1CAD5EA2C4F0B8E3571504C367C218ADD4FF', 'ROUTER_SERVICE_PORT_80_TCP': '80', 'R
OUTER_PORT_443_TCP_ADDR': '172.30.246.127', 'ROUTER_PORT_1936_TCP_ADDR': '172.30.246.127', 'ROUT
ER_SERVICE_PORT': '80', 'ROUTER_PORT_443_TCP_PORT': '443', 'KUBERNETES_SERVICE_PORT_DNS_TCP': '5
3', 'KUBERNETES_PORT_53_UDP_ADDR': '172.30.0.1', 'KUBERNETES_PORT_53_UDP': 'udp://172.30.0.1:53'
, 'KUBERNETES_PORT': 'tcp://172.30.0.1:443', 'ROUTER_PORT_1936_TCP_PORT': '1936', 'ROUTER_PORT_8
0_TCP': 'tcp://172.30.246.127:80', 'KUBERNETES_SERVICE_PORT_HTTPS': '443', 'KUBERNETES_PORT_53_U
DP_PORT': '53', 'ROUTER_PORT_80_TCP_PORT': '80', 'ROUTER_PORT': 'tcp://172.30.246.127:80', 'ROUT
ER_PORT_443_TCP': 'tcp://172.30.246.127:443', 'SIMPLE_SERVICE_VERSION': '1.0', 'ROUTER_PORT_443_
TCP_PROTO': 'tcp', 'KUBERNETES_PORT_443_TCP': 'tcp://172.30.0.1:443', 'DOCKER_REGISTRY_PORT_5000
_TCP': 'tcp://172.30.1.1:5000', 'DOCKER_REGISTRY_PORT': 'tcp://172.30.1.1:5000', 'KUBERNETES_POR
T_443_TCP_PORT': '443', 'ROUTER_SERVICE_PORT_1936_TCP': '1936', 'DOCKER_REGISTRY_PORT_5000_TCP_A
DDR': '172.30.1.1', 'DOCKER_REGISTRY_SERVICE_PORT_5000_TCP': '5000', 'KUBERNETES_PORT_443_TCP_PR
OTO': 'tcp'}"}
```


Alternativamente, también puede usar `kubectl exec` para conectarse al contenedor y enumerar las variables de entorno directamente, allí:

```
$ kubectl exec envs -- printenv

PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=envs

SIMPLE_SERVICE_VERSION=1.0

KUBERNETES_PORT_53_UDP_ADDR=172.30.0.1

KUBERNETES_PORT_53_TCP_PORT=53

ROUTER_PORT_443_TCP_PROTO=tcp

DOCKER_REGISTRY_PORT_5000_TCP_ADDR=172.30.1.1

...
```

Puedes destruir el pod creado con:

```
$ kubectl delete pod/envs
```

Laboratorio Namespaces

Los espacios de nombres proporcionan un alcance de recursos de Kubernetes, dividiendo su clúster en unidades más pequeñas. Puede considerarlo como un espacio de trabajo que está compartiendo con otros usuarios. Muchos recursos, como pods y servicios, tienen espacios de nombres, mientras que algunos, por ejemplo, los nodos no tienen espacios de nombres (sino todo el clúster). Como desarrollador, generalmente usaría un espacio de nombres asignado, sin embargo, los administradores pueden querer administrarlos, por ejemplo, para configurar el control de acceso o las cuotas de recursos.

Hagamos una lista de todos los espacios de nombres (tenga en cuenta que la salida dependerá del entorno que esté utilizando):

```
$ kubectl get ns

NAME               STATUS    AGE

default            Active    13d

kube-system        Active    13d

namingthings       Active    12d

openshift          Active    13d

openshift-infra    Active    13d
```

Puedes aprender más sobre un espacio de nombres usando el verbo `describe`, por ejemplo:

```
$ kubectl describe ns default

Name:   default

Labels: <none>

Status: Active



No resource quota.
```

```
No resource limits.
```

Ahora creemos un nuevo espacio de nombres llamado `test`:

```
$ kubectl apply -f kubernetes-labs2/ns/ns.yaml

namespace "test" created



$ kubectl get ns

NAME               STATUS    AGE

default            Active    13d

kube-system        Active    13d

namingthings       Active    12d

openshift          Active    13d

openshift-infra    Active    13d

test               Active    3s
```

Alternativamente, podríamos haber creado el espacio de nombres usando el comando `kubectl create namespace test`.

Para iniciar un pod en el espacio de nombres recién creado `test`, haga:

```
$ kubectl apply --namespace=test -f kubernetes-labs2/ns/pod.yaml
```

Tenga en cuenta que al usar el método anterior, el espacio de nombres se convierte en una propiedad de tiempo de ejecución, es decir, puede implementar el mismo pod o servicio, etc. en múltiples espacios de nombres (por ejemplo: `dev` y `prod`). Es posible codificar el espacio de nombres directamente en la seccion `metadata` como se muestra a continuación, pero causa menos flexibilidad al implementar sus aplicaciones:

```
apiVersion: v1

kind: Pod

metadata:

  name: podintest

  namespace: test
```

Para enumerar objetos con espacios de nombres como nuestro pod `podintest`, ejecute el siguiente comando:

```
$ kubectl get pods --namespace=test

NAME         READY     STATUS    RESTARTS   AGE

podintest    1/1       Running   0          16s
```

Puede eliminar el espacio de nombres (y todo lo que contiene) con:

```
$ kubectl delete ns test
```

Si es administrador, puede consultar los documentos para obtener más información sobre cómo manejar los espacios de nombres.

Aplicando quotas a los namespaces

quota-object.yaml

```
kind: ResourceQuota
apiVersion: v1
metadata:
  name: preproduccion
  namespace: preproduccion
spec:
  hard:
    limits.cpu: '2000'
    limits.memory: 4Gi
    pods: '10'
    requests.cpu: '2000'
    requests.memory: 4Gi
    requests.storage: 10G
```

Para crear la cuota, aplica el fichero YAML

```
# kubectl create -f quota-object.yaml --namespace preproduccion
resourcequota "object-counts" created

# kubectl describe ns preproducción
```

Laboratorio Volumes

Un volumen de Kubernetes es esencialmente un directorio accesible para todos los contenedores que se ejecutan en un pod. A diferencia del sistema de archivos local del contenedor, los datos en volúmenes se conservan en los reinicios del contenedor. El medio que respalda un volumen y su contenido están determinados por el tipo de volumen:

- tipos locales de nodo como `emptyDir` o `hostPath`
- tipos de intercambio de archivos como `nfs`
- nube tipos proveedor-específico como `awsElasticBlockStore`, `azureDisk` o `gcePersistentDisk`
- tipos de sistemas de archivos distribuidos, por ejemplo `glusterfs` o `cephfs`
- tipos de propósito especial como `secret`, `gitRepo`

Un tipo especial de volumen es el `PersistentVolume` que cubriremos en otro lugar.

Creemos un pod con dos contenedores que usen un volumen `emptyDir` para intercambiar datos:

```
$ kubectl apply -f kubernetes-labs2/volumes/pod.yaml

$ kubectl describe pod sharevol

Name:                   sharevol

Namespace:              default

...

Volumes:

  xchange:

    Type:       EmptyDir (a temporary directory that shares a pod's lifetime)

    Medium:
```

Primero ejecutamos en uno de los contenedores en el pod `c1`, verificamos el montaje del volumen y generamos algunos datos:

```
$ kubectl exec -it sharevol -c c1 -- bash

[root@sharevol /]# mount | grep xchange

/dev/sda1 on /tmp/xchange type ext4 (rw,relatime,data=ordered)

[root@sharevol /]# echo 'some data' > /tmp/xchange/data
```

Cuando ahora ejecutamos en `c2` el segundo contenedor que se ejecuta en el pod, podemos ver el volumen montado en `/tmp/data` y podemos leer los datos creados en el paso anterior:

```
$ kubectl exec -it sharevol -c c2 -- bash

[root@sharevol /]# mount | grep /tmp/data

/dev/sda1 on /tmp/data type ext4 (rw,relatime,data=ordered)

[root@sharevol /]# cat /tmp/data/data

some data
```

Tenga en cuenta que en cada contenedor debe decidir dónde montar el volumen y que `emptyDir` actualmente no puede especificar límites de consumo de recursos.

Puede eliminar la pod con:

```
$ kubectl delete pod/sharevol
```

Como ya se describió, esto destruirá el volumen compartido y todo su contenido.

Laboratorio Persistent Volumes

Este lab no funcionara, tendremos que crear primero un pv sobre nfs

Un volumen persistente (PV) es un recurso de todo el clúster que puede usar para almacenar datos de una manera que persista más allá de la vida útil de un pod. El PV no está respaldado por el almacenamiento conectado localmente en un nodo de trabajo sino por un sistema de almacenamiento en red como EBS o NFS o un sistema de archivos distribuido como Ceph.

Para utilizar un PV, primero debe reclamarlo, utilizando un reclamo de volumen persistente (PVC). El PVC solicita un PV con su especificación deseada (tamaño, velocidad, etc.) de Kubernetes y luego lo une a un pod donde puede montarlo como un volumen. Así que creemos un PVC de este tipo , pidiéndole a Kubernetes 1 GB de almacenamiento usando la clase de almacenamiento predeterminada :

```
$ kubectl apply -f kubernetes-labs2/pv/pvc.yaml

$ kubectl get pvc

NAME      STATUS   VOLUME                                     CAPACITY   ACCE
SS MODES   STORAGECLASS    AGE

myclaim   Bound    pvc-27fed6b6-3047-11e9-84bb-12b5519f9b58   1Gi        RWO
gp2-encrypted   18m
```

Para comprender cómo se desarrolla la persistencia, creemos un deployment que use el PVC anterior para montarlo como un volumen en `/tmp/persistent`:

```
$ kubectl apply -f kubernetes-labs2/pv/deploy.yaml
```

Ahora queremos probar si los datos en el volumen realmente persisten. Para esto, encontramos el pod gestionado por el deployment anterior, ejecute en su contenedor principal y cree un archivo llamado `data` en el directorio `/tmp/persistent` (donde decidimos montar el PV):

```
$ kubectl get po

NAME                          READY   STATUS    RESTARTS   AGE

pv-deploy-69959dccb5-jhxx     1/1     Running   0          16m


$ kubectl exec -it pv-deploy-69959dccb5-jhxxw -- bash

bash-4.2$ touch /tmp/persistent/data

bash-4.2$ ls /tmp/persistent/

data  lost+found
```

Es hora de destruir el pod y dejar que la implementación lance un nuevo pod. La expectativa es que el PV esté disponible nuevamente en el nuevo pod y los datos `/tmp/persistent` aún estén presentes. Vamos a ver eso:

```
$ kubectl delete po pv-deploy-69959dccb5-jhxxw

pod pv-deploy-69959dccb5-jhxxw deleted

$ kubectl get po

NAME                          READY   STATUS    RESTARTS   AGE

pv-deploy-69959dccb5-kwrrv    1/1     Running   0          16m

$ kubectl exec -it pv-deploy-69959dccb5-kwrrv -- bash

bash-4.2$ ls /tmp/persistent/

data  lost+found
```

Y, de hecho, el archivo `data` y su contenido todavía está donde se espera que esté.

Tenga en cuenta que el comportamiento predeterminado es que incluso cuando se elimina la implementación, el PVC (y el PV) continúa existiendo. Esta función de protección de almacenamiento ayuda a evitar la pérdida de datos. Una vez que esté seguro de que ya no necesita los datos, puede continuar y eliminar el PVC y con él eventualmente destruir el PV:

```
$ kubectl delete pvc myclaim

persistentvolumeclaim "myclaim" deleted
```

Los tipos de PV disponibles en su clúster de Kubernetes dependen del entorno (en la nube pública o local). Consulte el sitio de referencia Stateful Kubernetes.

Laboratorio Secrets

No desea que se guarde información confidencial como una contraseña de base de datos o una clave API en texto claro. Los secretos le proporcionan un mecanismo para utilizar dicha información de manera segura y confiable con las siguientes propiedades:

- Los secretos son objetos con espacio de nombres, es decir, existen en el contexto de un espacio de nombres
- Puede acceder a ellos a través de un volumen o una variable de entorno desde un contenedor que se ejecuta en un pod
- Los datos secretos en los nodos se almacenan en volúmenes tmpfs
- Existe un límite de tamaño por secreto de 1 MB
- El servidor API almacena secretos como texto sin formato en etcd

Creemos un secreto `apikey` que contenga una clave API (inventada):

```
$ echo -n "A19fh68B001j" > ./apikey.txt

$ kubectl create secret generic apikey --from-file=./apikey.txt

secret "apikey" created

$ kubectl describe secrets/apikey

Name:           apikey

Namespace:      default

Labels:         <none>

Annotations:    <none>

Type:   Opaque

Data

====

apikey.txt:     12 bytes
```

Ahora usemos el secreto en un pod a través de un volumen :

```
$ kubectl apply -f kubernetes-labs2/secrets/pod.yaml
```

Si ahora ejecutamos en el contenedor, vemos el secreto montado en `/tmp/apikey`:

```
$ kubectl exec -it consumesec -c shell -- bash

[root@consumesec /]# mount | grep apikey

tmpfs on /tmp/apikey type tmpfs (ro,relatime)

[root@consumesec /]# cat /tmp/apikey/apikey.txt

A19fh68B001j
```

Tenga en cuenta que para las cuentas de servicio, Kubernetes crea automáticamente secretos que contienen credenciales para acceder a la API y modifica sus pods para usar este tipo de secreto.

Puede eliminar tanto el pod como el secreto con:

```
$ kubectl delete pod/consumesec secret/apikey
```

Laboratorio Logging

El registro es una opción para comprender lo que sucede dentro de sus aplicaciones y el clúster en general. El registro básico en Kubernetes hace que la salida que produce un contenedor esté disponible, lo cual es un buen caso de uso para la depuración.

Las configuraciones más avanzadas consideran registros a través de nodos y los almacenan en un lugar central, ya sea dentro del clúster o mediante un servicio dedicado (basado en la nube).

Creemos un pod llamado `logme` que ejecute un contenedor escribiendo `stdout`y para `stderr`:

```
$ kubectl apply -f kubernetes-labs2/logging/pod.yaml
```

Para ver las cinco líneas de registro más recientes del contenedor `gen` en el pod `logme`, ejecute:

```
$ kubectl logs --tail=5 logme -c gen

Thu Apr 27 11:34:40 UTC 2017

Thu Apr 27 11:34:41 UTC 2017

Thu Apr 27 11:34:41 UTC 2017

Thu Apr 27 11:34:42 UTC 2017

Thu Apr 27 11:34:42 UTC 2017
```

Para transmitir el registro del contenedor `gen` en el pod `logme`(como `tail -f`), haga:

```
$ kubectl logs -f --since=10s logme -c gen

Thu Apr 27 11:43:11 UTC 2017

Thu Apr 27 11:43:11 UTC 2017

Thu Apr 27 11:43:12 UTC 2017
```

```
Thu Apr 27 11:43:12 UTC 2017

Thu Apr 27 11:43:13 UTC 2017

...
```

Tenga en cuenta que si no hubiera especificado `--since=10s` en el comando anterior, habría obtenido todas las líneas de registro desde el inicio del contenedor.

También puede ver registros de pods que ya han completado su ciclo de vida. Para esto creamos un pod llamado `oneshot` que cuenta regresivamente de 9 a 1 y luego sale. Con la opción `-p` puede imprimir los registros de instancias anteriores del contenedor en un pod:

```
$ kubectl apply -f kubernetes-labs2/logging/oneshotpod.yaml

$ kubectl logs -p oneshot -c gen

9

8

7

6

5

4

3

2

1
```

Puede eliminar las pods creadas con:

```
$ kubectl delete pod/logme pod/oneshot
```

Laboratorio Jobs

Un job en Kubernetes es un supervisor para pods que llevan a cabo procesos por lotes, es decir, un proceso que se ejecuta durante un cierto tiempo hasta su finalización, por ejemplo, un cálculo o una operación de respaldo.

Creemos un job llamado `countdown` que supervise un conteo de pods de 9 a 1:

```
$ kubectl apply -f kubernetes-labs2/jobs/job.yaml
```

Puede ver el job y el pod que cuida de esta manera:

```
$ kubectl get jobs

NAME                      DESIRED   SUCCESSFUL   AGE

countdown                 1         1            5s


$ kubectl get pods --show-all

NAME                            READY     STATUS      RESTARTS   AGE

countdown-lc80g                 0/1       Completed   0          16s
```

Para obtener más información sobre el estado del trabajo, haga lo siguiente:

```
$ kubectl describe jobs/countdown

Name:           countdown

Namespace:      default

Image(s):       centos:7

Selector:       controller-uid=ff585b92-2b43-11e7-b44f-be3e8f4350ff

Parallelism:    1

Completions:    1
```

```
Start Time:     Thu, 27 Apr 2017 13:21:10 +0100
Labels:         controller-uid=ff585b92-2b43-11e7-b44f-be3e8f4350ff
                job-name=countdown
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath
Type            Reason                  Message
  ---------     --------        -----   ----                    -------------
--------        ------                  -------
  2m            2m              1       {job-controller }
Normal          SuccessfulCreate        Created pod: countdown-lc80g
```

Y para ver la salida del trabajo a través del pod que supervisó, ejecute:

```
kubectl logs countdown-lc80g
9
8
7
6
5
4
3
2
1
```

Para limpiar, use el `delete` en el objeto de trabajo que eliminará todos los pods supervisados:

```
$ kubectl delete job countdown

job "countdown" deleted
```

Tenga en cuenta que también hay formas más avanzadas de usar jobs, por ejemplo, utilizando una cola de trabajo o programando la ejecución en un momento determinado a través de trabajos cron .

Laboratorio StatefulSet

Si tiene una aplicación sin estado, desea utilizar un deployment. Sin embargo, para una aplicación con estado, es posible que desee utilizar un StatefulSet .

A diferencia de un deployment, `StatefulSet` proporciona ciertas garantías sobre la identidad de los pods que está administrando (es decir, nombres predecibles) y sobre el orden de inicio.

Dos cosas más que son diferentes en comparación con un deployment: para la comunicación de red, debe crear Headless Services y para la persistencia, `StatefulSet` gestiona un volumen persistente por pod.

Comencemos con la creación de la aplicación con estado , es decir, `StatefulSet` junto con los volúmenes persistentes y el servicio sin cabeza:

```
$ kubectl apply -f https://raw.githubusercontent.com/mhausenblas/mehdb/master
/app.yaml
```

Después de un minuto más o menos, puede echar un vistazo a todos los recursos que se han creado:

```
$ kubectl get sts,po,pvc,svc

NAME                       DESIRED   CURRENT   AGE

statefulset.apps/mehdb     2         2         1m



NAME           READY   STATUS    RESTARTS   AGE

pod/mehdb-0    1/1     Running   0          1m

pod/mehdb-1    1/1     Running   0          56s



NAME                                  STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
```

```
persistentvolumeclaim/data-mehdb-0   Bound    pvc-bc2d9b3b-310d-11e9-aeff-123
713f594ec   1Gi        RWO            ebs            1m

persistentvolumeclaim/data-mehdb-1   Bound    pvc-d4b7620f-310d-11e9-aeff-123
713f594ec   1Gi        RWO            ebs            56s



NAME             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE

service/mehdb    ClusterIP   None         <none>        9876/TCP   1m
```

Ahora podemos verificar si la aplicación con estado funciona correctamente. Para hacer esto, utilizamos el punto final `/status` del servicio sin cabeza `mehdb:9876` y, dado que todavía no hemos puesto ningún dato en el almacén de datos, esperamos que `0` se informen las claves:

```
$ kubectl run -it --rm jumpod --restart=Never --image=quay.io/mh
ausenblas/jump:0.2 -- curl mehdb:9876/status?level=full

If you don't see a command prompt, try pressing enter.

0

pod "jumpod" deleted
```

Y, de hecho, vemos que hay `0` llaves disponibles, como se informó anteriormente.

Tenga en cuenta que a veces a `StatefulSet` no es la mejor opción para su aplicación con estado. Puede que sea mejor definir un recurso personalizado junto con escribir un controlador personalizado para tener un control más preciso sobre su carga de trabajo.

Laboratorio Init Containers

A veces es necesario preparar un contenedor que se ejecuta en un pod. Por ejemplo, es posible que desee esperar a que un servicio esté disponible, configurar cosas en tiempo de ejecución o iniciar algunos datos en una base de datos.

En todos estos casos, los contenedores init son útiles. Tenga en cuenta que Kubernetes ejecutará todos los contenedores init (y todos deben salir con éxito) antes de que se ejecuten los contenedores principales.

Entonces, creemos un deployment que consista en un contenedor de inicio que escriba un mensaje en un archivo `/ic/this` y el contenedor principal (de larga ejecución) comprobamos la configuración del deploy.yaml:

```
$ kubectl apply -f kubernetes-labs2/ic/deploy.yaml
```

Ahora podemos verificar la salida del contenedor principal:

```
$ kubectl get deploy,po
NAME                                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
deployment.extensions/ic-deploy     1         1         1            1
11m


NAME                                READY   STATUS    RESTARTS   AGE
pod/ic-deploy-bf75cbf87-8zmrb       1/1     Running   0          59s


$ kubectl logs ic-deploy-bf75cbf87-8zmrb -f
INIT_DONE
INIT_DONE
INIT_DONE
```

```
INIT_DONE

INIT_DONE

^C
```

Si desea obtener más información sobre los contenedores init y temas relacionados, consulte la publicación del blog Kubernetes: A Pod's Life .

Laboratorio Nodes

En Kubernetes, los nodos son las máquinas (virtuales) donde se ejecutan sus cargas de trabajo en forma de pods. Como desarrollador, normalmente no se ocupa directamente de los nodos, sin embargo, como administrador, es posible que desee familiarizarse con las operaciones de los nodos .

Para enumerar los nodos disponibles en su clúster (tenga en cuenta que la salida dependerá del entorno que esté usando:

```
$ kubectl get nodes

NAME             STATUS    AGE

192.168.99.100   Ready     14d
```

Una tarea interesante, desde el punto de vista del desarrollador, es hacer que Kubernetes programe un pod en un determinado nodo. Para esto, primero debemos etiquetar el nodo al que queremos apuntar:

```
$ kubectl label nodes 192.168.99.100 shouldrun=here

node "192.168.99.100" labeled
```

Ahora podemos crear un pod que se programa en el nodo con la etiqueta `shouldrun=here`:

```
$ kubectl apply -f kubernetes-labs2/nodes/pod.yaml


$ kubectl get pods --output=wide

NAME                       READY     STATUS    RESTARTS   AGE       IP
NODE

onspecificnode             1/1       Running   0          8s        172.17.0.3
192.168.99.100
```

Para obtener más información sobre un nodo específico, `192.168.99.100` en nuestro caso, haga lo siguiente:

```
$ kubectl describe node 192.168.99.100

Name:                   192.168.99.100

Labels:                 beta.kubernetes.io/arch=amd64

                        beta.kubernetes.io/os=linux

                        kubernetes.io/hostname=192.168.99.100

                        shouldrun=here

Taints:                 <none>

CreationTimestamp:      Wed, 12 Apr 2017 17:17:13 +0100

Phase:

Conditions:

  Type                  Status  LastHeartbeatTime                       La
stTransitionTime                Reason                          Message

  ----                  ------  -----------------                       --
----------------                ------                          -------

  OutOfDisk             False   Thu, 27 Apr 2017 14:55:49 +0100  Thu, 27 Ap
r 2017 09:18:13 +0100   KubeletHasSufficientDisk         kubelet has suffici
ent disk space available

  MemoryPressure        False   Thu, 27 Apr 2017 14:55:49 +0100  Wed, 12 Ap
r 2017 17:17:13 +0100   KubeletHasSufficientMemory       kubelet has suffici
ent memory available

  DiskPressure          False   Thu, 27 Apr 2017 14:55:49 +0100  Wed, 12 Ap
r 2017 17:17:13 +0100   KubeletHasNoDiskPressure         kubelet has no disk
pressure

  Ready         True    Thu, 27 Apr 2017 14:55:49 +0100  Thu, 27 Apr 2017 09
:18:24 +0100    KubeletReady                    kubelet is posting ready st
atus

Addresses:              192.168.99.100,192.168.99.100,192.168.99.100

Capacity:
```

```
 alpha.kubernetes.io/nvidia-gpu:  0
 cpu:                             2
 memory:                          2050168Ki
 pods:                            20
Allocatable:
 alpha.kubernetes.io/nvidia-gpu:  0
 cpu:                             2
 memory:                          2050168Ki
 pods:                            20
System Info:
 Machine ID:                      896b6d970cd14d158be1fd1c31ff1a8a
 System UUID:                     F7771C31-30B0-44EC-8364-B3517DBC8767
 Boot ID:                         1d589b36-3413-4e82-af80-b2756342eed4
 Kernel Version:          4.4.27-boot2docker
 OS Image:                        CentOS Linux 7 (Core)
 Operating System:                linux
 Architecture:                    amd64
 Container Runtime Version:       docker://1.12.3
 Kubelet Version:                 v1.5.2+43a9be4
 Kube-Proxy Version:              v1.5.2+43a9be4
ExternalID:                       192.168.99.100
Non-terminated Pods:              (3 in total)
  Namespace                       Name                              CPU Reques
ts       CPU Limits       Memory Requests  Memory Limits
  ---------                       ----                              ----------
--       ----------       ---------------  -------------
  default                         docker-registry-1-hfpzp           100m (5%)
         0 (0%)           256Mi (12%)      0 (0%)
```

```
  default                         onspecificnode                    0 (0%)
        0 (0%)          0 (0%)          0 (0%)

  default                         router-1-cdglk                    100m (5%)
        0 (0%)          256Mi (12%)     0 (0%)

Allocated resources:

  (Total limits may be over 100 percent, i.e., overcommitted.

  CPU Requests   CPU Limits       Memory Requests  Memory Limits

  ------------   ----------       ---------------  -------------

  200m (10%)     0 (0%)           512Mi (25%)      0 (0%)

No events.
```

Tenga en cuenta que existen métodos más sofisticados que los mostrados anteriormente, como el uso de la afinidad, para asignar pods a los nodos y, según su caso de uso, es posible que también desee verificarlos.

Laboratorio API Server access

A veces es útil o necesario acceder directamente al servidor API de Kubernetes , con fines de exploración o prueba.

Para hacer esto, una opción es proxy de la API a su entorno local, utilizando:

```
$ kubectl proxy --port=8080

Starting to serve on 127.0.0.1:8080
```

Ahora puede consultar la API (en una sesión de terminal separada) así:

```
$ curl http://localhost:8080/api/v1

{

  "kind": "APIResourceList",

  "groupVersion": "v1",

  "resources": [

    {

...

    {

      "name": "services/status",

      "singularName": "",

      "namespaced": true,

      "kind": "Service",

      "verbs": [

        "get",

        "patch",
```

```
        "update"
      ]
    }
  ]
}
```

Alternativamente, sin proxy, puede usarlo `kubectl` directamente de la siguiente manera para lograr lo mismo:

```
$ kubectl get --raw=/api/v1
```

Además, si desea explorar las versiones y / o recursos de API compatibles, puede usar los siguientes comandos:

```
$ kubectl api-versions
admissionregistration.k8s.io/v1beta1
...
v1


$ kubectl api-resources
NAME                                SHORTNAMES     APIGROUP
NAMESPACED   KIND
bindings
true         Binding
...
storageclasses                      sc             storage.k8s.io
```