

¿Qué son los orquestadores?

La misión principal de un orquestador es, dado un contenedor a ejecutar, seleccionar un servidor de entre un conjunto de máquinas donde ejecutarlo. A este conjunto de máquinas le denominaremos clúster, y a cada servidor le denominaremos *nodo*.

La lógica que selecciona el nodo donde el contenedor va a ser ejecutado es compleja, y tiene en cuenta cosas como recursos de memoria y cpu, tags, prioridades, afinidades y anti-afinidades.

Sin embargo, los orquestadores actuales suelen ofrecer funcionalidades añadidas. Algunas de ellas son:

- **Service Discovery:** se refiere al hecho de que mis contenedores van a tener *endpoints* de acceso reconocibles, de tal manera que pueda dirigir peticiones a los mismos. Los mecanismos de service discovery suelen utilizar el nombre de los contenedores creados para que sus *endpoints* sean reconocibles, y estar basados en DNS.
- **Balanceo de carga:** cuando corremos aplicaciones en producción lo normal es que tengamos muchas instancias del mismo contenedor corriendo mi aplicación, de tal manera que mi aplicación pueda escalar horizontalmente. Muchos orquestadores ofrecen puntos de entrada comunes a este conjunto de contenedores, de tal manera que cada petición se redirija a un contenedor diferente y la carga del sistema quede repartida entre todos los contenedores.
- **Configuración de red:** es importante que los distintos nodos del clúster sean accesible entre sí, pero no solo eso, si no que los distintos contenedores que el clúster está ejecutando tengan una manera única de accederse entre ellos. Esto significa que cada contenedor tiene que tener una IP única, y accesible desde cualquier nodo del clúster.
- **Escalabilidad:** muchos orquestadores permiten escalar arriba y abajo tanto el número de nodos del clúster, como el número de instancias de un contenedor que está corriendo mi aplicación, ofreciendo un uso óptimo de mis recursos en base a la carga del sistema.
- **Logging y monitoreo:** cuando tenemos cientos o miles de contenedores en un clúster es importante tener visibilidad de lo que está pasando en el mismo. Es por esto que los orquestadores suelen incorporar mecanismos de logging y monitoreo, para que en todo momento pueda consultar los logs de una aplicación o sus métricas de ejecución.
- **Respuesta a fallo:** en un sistema real la pregunta no es si un fallo se va a producir o no, si no cuándo se va a producir. Es por esto que una funcionalidad muy potente que ofrecen muchos orquestadores es la capacidad de detectar cuando un contenedor (o un nodo) está funcionando de una manera incorrecta, y en ese momento reemplazar dicho contenedor por uno nuevo.

- **Ecosistema:** aunque no se trata de una funcionalidad como tal, un aspecto fundamental de un orquestador es la comunidad que hay a su alrededor, y el ecosistema que tiene. Un ecosistema rico nos asegura que cuando tengamos problemas es muy posible que encontremos gente que ya ha pasado por el mismo problema, y también que dispongamos de numerosas integraciones con otras herramientas para poder ajustar el clúster a nuestras necesidades.

Ejemplos de orquestadores

El primer ejemplo de orquestador es aquél que podemos implementar nosotros mismos usando *bash scripting*, o lenguajes más avanzados como Ansible. Si bien esta puede ser la manera más antigua (y quizás común) de lanzar contenedores, en un aplicación que algo de escala, nos va a ser mucho mejor si utilizamos un orquestador ya implementado que no estar nosotros reinventando la rueda.

Otra alternativa es seguir un enfoque mixto, donde creamos una máquina virtual por cada contenedor que tenemos que crear, y de esta manera podemos usar funcionalidad nativa del cloud para tener balanceo de carga, auto-escalabilidad, o respuesta a fallo. En concreto, [i2kit](#) sigue este enfoque para AWS, utilizando Elastic Load Balancers, Auto Scalability Groups y otros recursos de AWS.

En cuanto a los orquestadores propios del mundo de los contenedores, **el primero que introducimos es Docker Swarm.** Es un orquestador muy potente y sencillo de usar, pero tardó mucho tiempo en asentarse y ser estable y no tiene un ecosistema potente a su alrededor. Es por esto que está quedando en desuso en favor de Kubernetes.

El siguiente que vamos a comentar es Mesosphere. Es posiblemente el orquestador más potente y que funciona a mayor escala de los últimos años, pero la industria parece que se ha puesto de acuerdo en adoptar Kubernetes como el orquestador estándar, y también está quedando en desuso. Como ejemplo, AWS, Azure, Google Compute o Digital Ocean, ofrecen ya servicios de Kubernetes auto-gestionados, pero ninguno ofrece Mesosphere o Docker Swarm.

Por último tendríamos soluciones como AWS Fargate. En este caso no necesitamos de tener un cluster pre-creado para lanzar contenedores, si no que nos ofrecen una API donde podemos lanzar contenedores bajo demanda. Es un enfoque muy prometedor porque nos evita el trabajo de gestionar un clúster, pero también es bastante más caro. Es una alternativa real a Kubernetes que puede acabar imponiéndose en el futuro, aunque es muy posible que estas soluciones acaben adoptando Kubernetes como *runtime*.

En base a esto es por lo que en este curso recomendamos lanzar nuestros aplicativos productivos usando Kubernetes, que se ha convertido en el estándar *de facto* para el despliegue de aplicaciones en el cloud.

Introducción a Kubernetes

Como hemos dicho en lecciones anteriores, **Kubernetes es un orquestador de contenedores, y que se ha convertido en el estándar *de facto* para desplegar aplicaciones cloud.**

Kubernetes es un proyecto *open source*, desarrollado en un principio por Google. Es una evolución de Borg, el orquestador que utiliza Google en producción, pero adaptado a usar contenedores Docker. Por tanto, es un orquestador que nace con muchos años de experiencia de Google ejecutando contenedores en producción, y que por tanto, es muy potente y escala muy bien.

La funcionalidad principal de Kubernetes es lanzar y gestionar contenedores en un clúster, permitiendo la ejecución de muchos contenedores en cada nodo. Y lo hace de un modo declarativo. Es decir, a Kubernetes le vamos a decir el número de instancias de un contenedor que queremos ejecutar, y Kubernetes se va a encargar de ejecutar ese número de instancias de la mejor manera posible. Si un contenedor empieza a funcionar mal, lo reemplazará. Y si en ese momento no se pueden ejecutar todos los contenedores deseados por falta de recursos, por ejemplo, esperará a que haya recursos disponibles y en ese momento seguirá creando contenedor hasta alcanzar el número deseado.

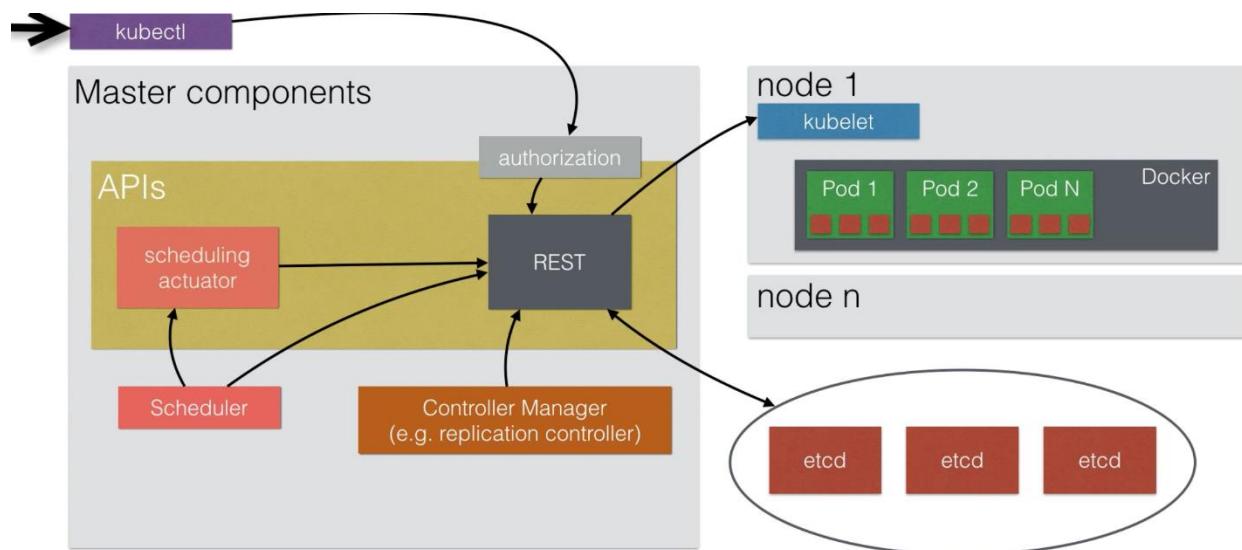
Probablemente **el aspecto más potente de Kubernetes es su ecosistema.** Parece que todo el mundo se ha puesto de acuerdo para convertirlo en el estándar de despliegue de aplicaciones en el cloud, y no hay herramienta que se precie que no tenga una versión para correr en Kubernetes. De hecho, Kubernetes ha conseguido hacer realidad el concepto de multi-cloud. Si empaquetamos nuestra aplicación para correr en Kubernetes, nos va a ser relativamente sencillo lanzar nuestra aplicación en un Kubernetes corriendo en AWS, o en Azure, o en Google Cloud, o en un clúster *on-premises*. Esta característica es de vital importancia, sobretodo para las empresas que hace software enterprise que ejecuta en la infraestructura del cliente.

Arquitectura y componentes de Kubernetes

Kubernetes divide los nodos de un clúster en **master nodes** y en **worker nodes**.

Los master nodes son la capa de control y en principio no ejecutan contenedores de usuario. Los worker nodes son los responsables de ejecutar los contenedores de usuario.

Aunque esta división es la más habitual, cuando corremos Kubernetes en local se suele crear un clúster de un solo nodo, que actúa a la vez como master y como worker node.



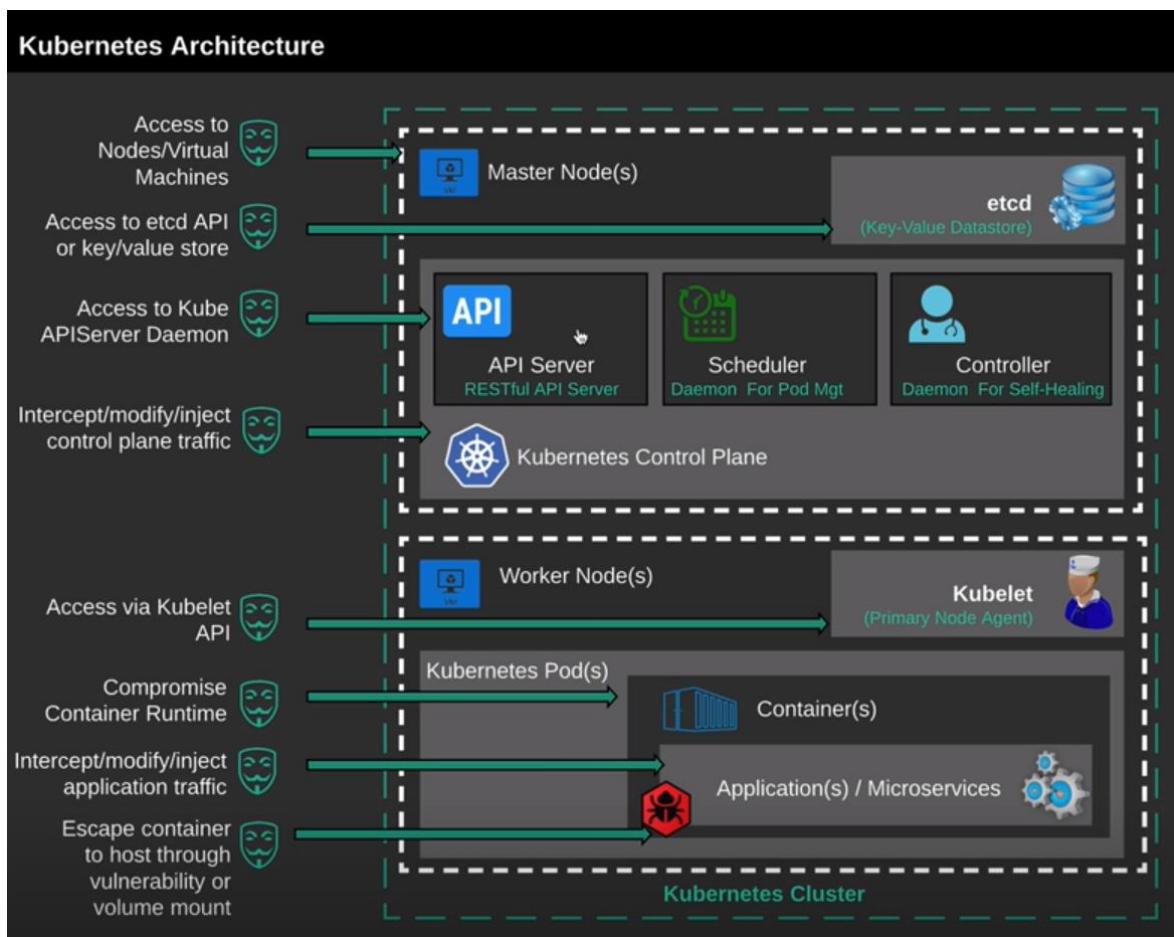
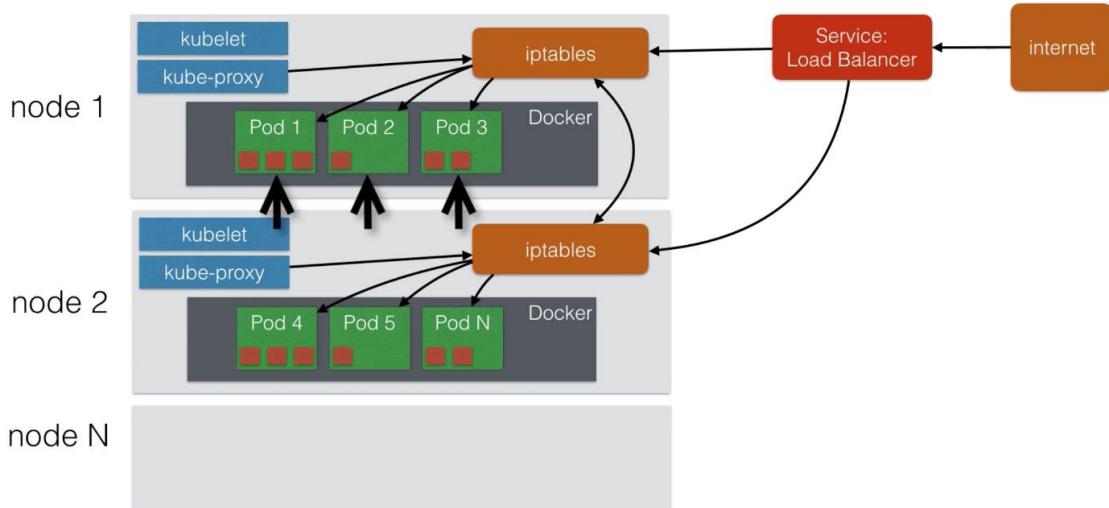
La lógica de la capa de control se fundamenta en `etcd`, una base de datos distribuida que nos asegura la consistencia de los datos usando el protocolo RAFT. Sobre `etcd` tenemos la capa REST, que es a través de la que pasan todos los accesos a la base de datos. Esa capa REST se expone vía un módulo de autenticación, para que podamos invocarla, por ejemplo, usando comandos `kubectl`.

El resto de la lógica que corre en los masters nodes son controladores, que son funciones que se ejecutan en bucle y que comprueban si el estado deseado para el clúster se corresponde con el estado actual, o si por el contrario hay que crear o eliminar contenedores.

Los masters siempre serán impares normalmente tres nodos, si el cluster es muy grande podríamos utilizar cinco.

Los worker nodes, además de los contenedores de usuario, tienen principalmente dos componentes, el `kubelet` y el `kube-proxy`.

El `kubelet` se encarga de chequear el estado de `etcd` vía la capa REST para ver qué contenedores tiene asignados para ejecutar, y en base a ello, crear o eliminar contenedores en el nodo que gestiona. Por su parte, el `kube-proxy` se encarga de gestionar la red y el *service discovery* entre contenedores creando reglas de *iptables*.



Distribuciones para la instalación de Kubernetes

Hay muchas maneras de tener un clúster de Kubernetes corriendo. Aunque no sea un objetivo de este curso, nombraremos alguna de ellas para su conocimiento por parte del alumno.

En primer lugar tenemos las instalaciones de Kubernetes en local, donde destacamos Minikube y Docker Desktop. Las prácticas de este curso las vamos a hacer con Minikube, pero en principio podríamos usar cualquier otra distribución.

Cuando queremos lanzar clústers de Kubernetes en el cloud, una opción interesante es Kops. Kops es un proyecto *open source* que nos permite lanzar clústers de Kubernetes en multitud de proveedores de cloud. Es una herramienta muy bien hecha, pero que según los proveedores de cloud van sacando sus propias instalaciones de Kubernetes, puede que vaya siendo menos utilizada.

Hoy en día, si necesitas lanzar un clúster de Kubernetes en el cloud, yo recomendaría las soluciones nativas de cada proveedor, como son AKS (de Azure), EKS (de AWS), KUBERNETES (de Google) o el servicio de Digital Ocean. Si estas soluciones no son lo suficientemente flexibles para nuestros casos de uso, Kops sería la opción a considerar.

Minikube

[Minikube](#) es una instalación local de Kubernetes muy útil para hacer desarrollo.

En el repositorio viene como hacer la instalación para los distintos sistemas operativos. En el caso de MacOS tenemos que ejecutar:

```
brew cask install minikube
```

A continuación necesitamos instalar la CLI de Kubernetes para ejecutar comandos contra el clúster de Minikube. Una guía completa para distintos sistemas operativos la tenemos [aquí](#). Para el caso de MacOS ejecutamos:

```
brew install kubernetes-cli
```

Lo siguiente que haremos es crear el clúster de Minikube con el comando:

```
minikube start
```

que creará una máquina virtual en nuestro ordenador con Kubernetes instalado.

Una vez que termine la ejecución de `minikube start` ya estaremos listos para trabajar.

Aplicacion Voting App Services (No desplegar)

kubectl funciona de una manera similar a docker, en el sentido que se puede configurar para hablar con distintos cluster de Kubernetes. Si ejecutamos `kubectl config get-contexts` veremos los clúster que tenemos configurados, y con un símbolo * nos indica el que tenemos activo en este momento, que debe ser `minikube`.

Para cambiar a otro cluster debemos ejecutar `kubectl config set-context nombre-del-cluster`.

Para empezar a realizar los laboratorios utilizaremos el repositorio suministrador por el formador:

[k8-for-devs](#)

y nos movemos al directorio de la Voting App.

```
cd voting-app
```

Voting App se compone de cinco microservicios: *vote*, *result*, *worker*, *redis* y *db*. Como usuario de Kubernetes, para desplegar la aplicación solo tengo que ejecutar:

```
kubectl apply -f .
```

Esta es la manera de trabajar que vamos a recomendar en este curso, teniendo siempre los ficheros YAML de mis componentes en ficheros. Ahora, tenemos que obtener la ip de Minikube:

```
minikube ip
```

Y cuando la aplicación termine de desplegarse podemos acceder a la misma en:

```
192.168.99.100:31000  
192.168.99.100:31001
```

Kubectl

En esta clase vamos a familiarizarnos con el uso de la línea de comandos **kubectl** haciendo uso de la aplicación de Voting App que ya tenemos corriendo.

Como en cualquier comando, podemos hacer:

```
kubectl --help
```

para tener una vista general de las cosas que hacer con `kubectl`.

Para ver lo que hay corriendo en nuestro clúster, podemos ejecutar:

```
kubectl get all --all-namespaces
```

donde veremos los servicios, deployments y pods que estamos ejecutando.

Para cualquiera de los objetos creados podemos recibir más información con el comando:

```
kubectl describe deployment.apps/vote
```

kubectl describe nos muestra los eventos asociados al objeto, que es muy útil para buscar errores en el despliegue. Sin embargo, si quiero ver los logs de un contenedor, tengo que ejecutar el comando:

```
kubectl logs -f pod-xxxxx-yyyy
```

Si queremos acceder a los puertos de un pod podemos usar el comando `kubectl port-forward` y exponerlos en la interfaz de `localhost`, y si queremos entrar dentro de un contenedor tenemos el comando `kubectl exec`.

Finalmente, para eliminar un objeto, ejecutamos el comando `kubectl delete`. Por ejemplo, para eliminar la aplicación entera ejecutamos:

```
kubectl delete -f .
```

Cuando crea definiciones de recursos YAML, necesita conocer los campos y sus significados. Para buscar esta información es en la referencia de la [API](#), que contiene las especificaciones completas de todos los recursos.

Sin embargo, cambiar a un navegador web cada vez que necesita buscar algo es tedioso. Por lo tanto, `kubectl` proporciona el comando **`kubectl explain`**, que puede mostrar las especificaciones de recursos de todos los recursos directamente desde nuestro terminal.

El uso de kubectl explain es el siguiente:

```
$kubectl explain resource[.field]...
```

El comando genera la especificación del recurso o campo solicitado. La información mostrada por `kubectl explain` es idéntica a la información en la referencia API.

Por defecto, kubectl explain muestra solo un único nivel de campos, podemos mostrar todo el árbol de campos con el parametro --recursive:

```
$kubectl explain pod.spec --recursive
```

En caso de que no esté seguro acerca de qué nombres de recursos puede usar kubectl explain, para mostrarlos todos con el siguiente comando:

```
$ kubectl api-resources
```

Este comando muestra los nombres de los recursos en su forma plural (por ejemplo, en deploymentslugar de deployment). También muestra el nombre corto (por ejemplo deploy) para aquellos recursos que tienen uno.

No te preocunes por estas diferencias. Todas estas variantes de nombre son equivalentes para kubectl. Es decir, puede usar cualquiera de ellos para kubectl explain.

Por ejemplo, todos los siguientes comandos son equivalentes:

```
kubectl explain deployments.spec  
# or  
kubectl explain deployment.spec  
# or  
kubectl explain deploy.spec
```

*Para más información acerca de la estructura de la definición de los objetos de Kubernetes:
[Understanding Kubernetes Objects](#).*

Use el formato de salida de columnas personalizadas:

```
-o custom-columns=<header>:<jsonpath>[,<header>:<jsonpath>]...
```

```
Select all elements of a list
kubectl get pods -o custom-columns='DATA:spec.containers[*].image'

# Select a specific element of a list
kubectl get pods -o custom-columns='DATA:spec.containers[0].image'

# Select those elements of a list that match a filter expression
kubectl get pods -o custom-columns='DATA:spec.containers[?(@.image!="nginx")].image'

# Select all fields under a specific location, regardless of their name
kubectl get pods -o custom-columns='DATA:metadata.*'

# Select all fields with a specific name, regardless of their location
kubectl get pods -o custom-columns='DATA:...image'

#Mostrar imágenes de contenedores de los pods
kubectl get pods \
-o custom-columns='NAME:metadata.name,IMAGES:spec.containers[*].image'
```

Configuring Monitoring for Kubernetes Cluster

Heapster

Heapster es un proyecto desarrollado por Kubernetes que permite el monitoreo y análisis de rendimiento en Kubernetes, actualmente Heapster se encuentra disponible para versiones de Kubernetes superior a las 1.0.6. Heapster recolecta y interpreta diversas señales como uso de computo, ciclo de vida y exporta las métricas via REST. Para visualización de esta información se puede utilizar Kubedash. El objetivo de Kubedash es permitir al usuario o administrador de un cluster de kubernetes verificar y entender el rendimiento de un cluster y sus trabajos corriendo a través de una interfaz visual. Para el uso de Kubedash es necesario:

Grafana e InfluxDB

Usar Grafana con InfluxDB es una solución popular para el monitoreo. InfluxDB expone una API que permite escribir y consultar información en series de tiempo (CPU, memory usage, llamada REST o errores) . Grafana entrega un interfaz que provee una configuración flexible y sencilla de paneles que consultan a la información recolectada en el InfluxDB.

Kubernetes proporciona configuraciones YAML listas para el stack de monitoreo. Estas configuraciones YAML no están destinadas a ser utilizadas directamente.

Podemos descargarlas de (`kubernetes/cluster/addons/cluster-monitoring/influxdb`)

```
# cd /laboratorios-kubernetes/Dhyaniarun-kubernetes
```

```
[root@master Dhyaniarun-kubernetes]# kubectl create -f cluster-monitoring/influxdb
service "monitoring-grafana" created
deployment "heapster-v1.2.0" created
service "heapster" created
replicationcontroller "monitoring-influxdb-grafana-v4" created
service "monitoring-influxdb" created
```

```
# kubectl cluster-info
```

Kubernetes master is running at <http://192.168.1.250:8080>

Heapster is running at <http://192.168.1.250:8080/api/v1/proxy/namespaces/kube-system/services/heapster>

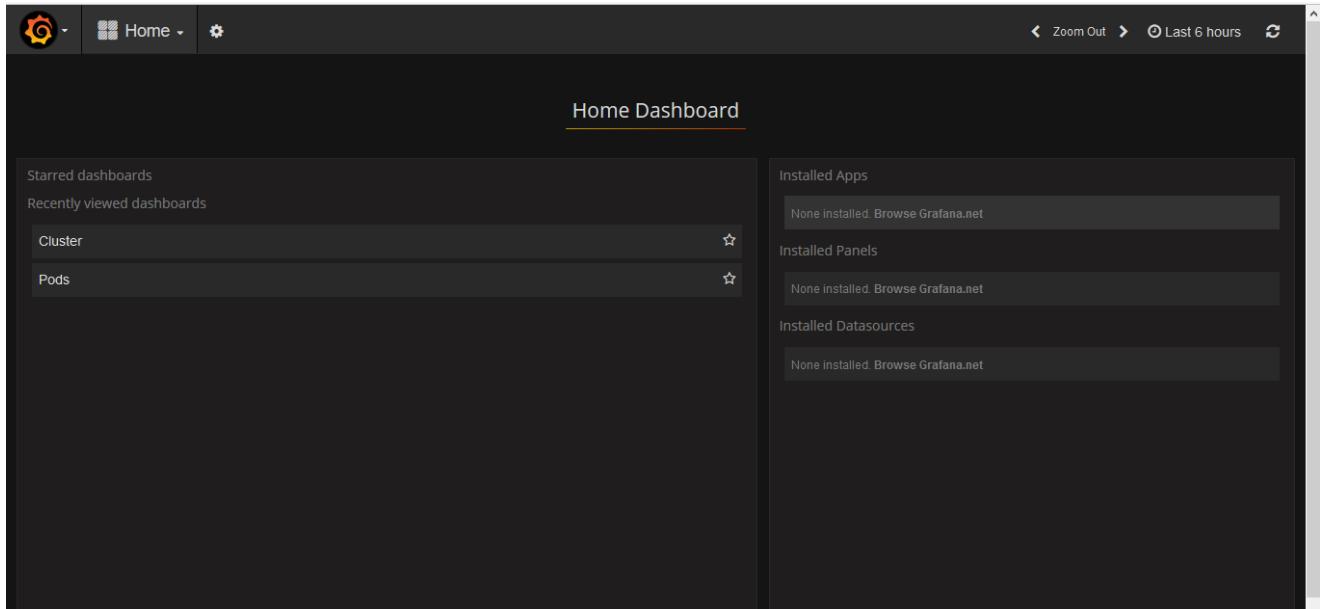
KubeDNS is running at <http://192.168.1.250:8080/api/v1/proxy/namespaces/kube-system/services/kube-dns>

kubernetes-dashboard is running at <http://192.168.1.250:8080/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard>

Grafana is running at <http://192.168.1.250:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana>

InfluxDB is running at <http://192.168.1.250:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb>

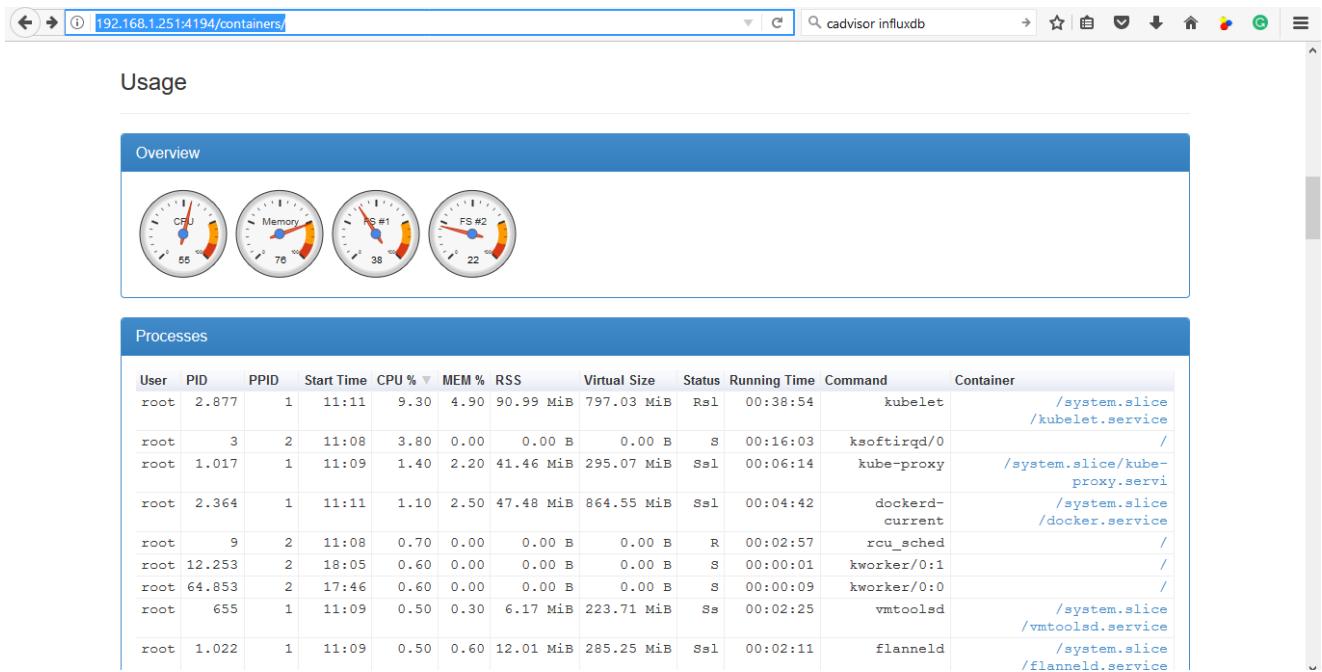
<http://192.168.1.250:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/>



Tambien podemos utilizar la herramienta **cAdvisor** es un sistema de monitorización de contenedores, desarrollado por Google y utilizado por kubernetes que tiene soporte nativo para Docker, el cual puede ser usado a través, del cualquier nodo del servidor en la URL:

<http://192.168.1.251:4194/containers/>

<http://192.168.1.252:4194/containers/>



Cockpit es una herramienta para la administración remota de servidores que nace dentro de Fedora.Next y Project Atomic como una opción gráfica para administradores de sistemas con características avanzadas para administradores de sistemas.

```
[root@minion1 ~]# yum install cockpit cockpit-docker cockpit-kubernetes cockpit-dashboard
```

```
[root@minion2 ~]# yum install cockpit cockpit-docker
```

```
[root@master ~]# yum install cockpit cockpit-docker
```

```
[root@minion1 ~]# systemctl start cockpit
```

```
[root@minion1 ~]# systemctl enable cockpit
```

```
[root@minion2 ~]# systemctl start cockpit
```

```
[root@minion2 ~]# systemctl enable cockpit
```

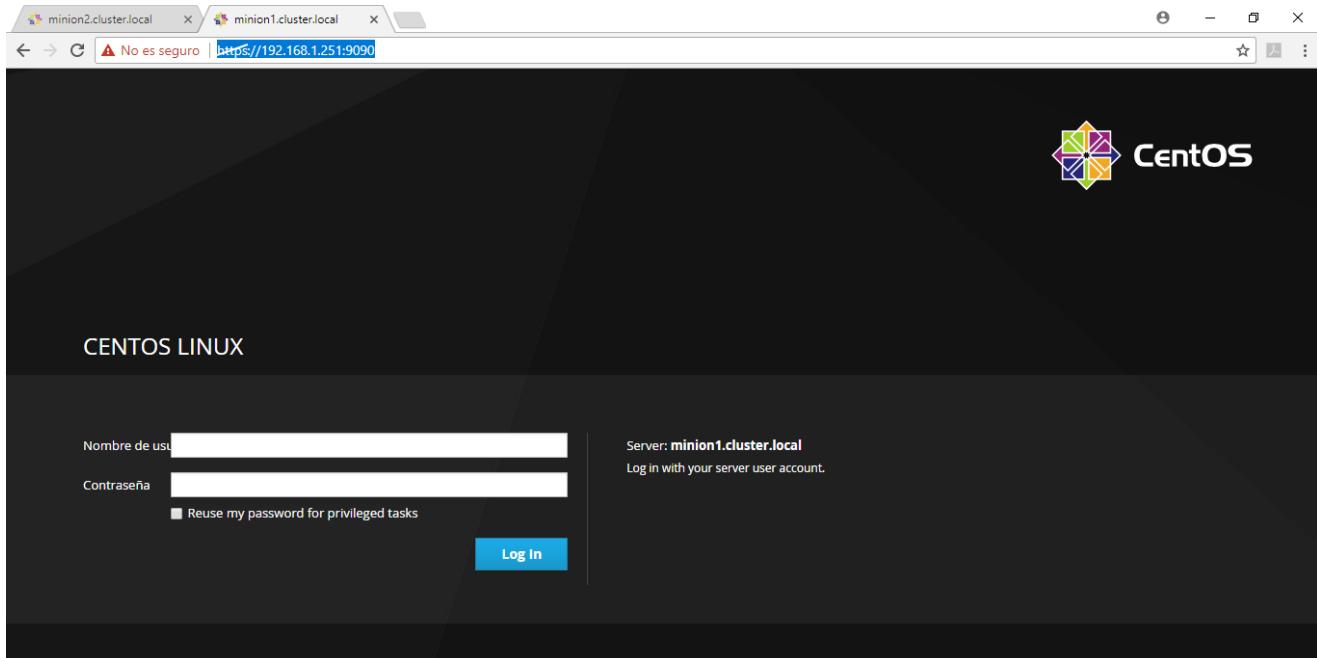
<https://192.168.1.250:9090/>

<https://192.168.1.251:9090/>

<https://192.168.1.252:9090/>

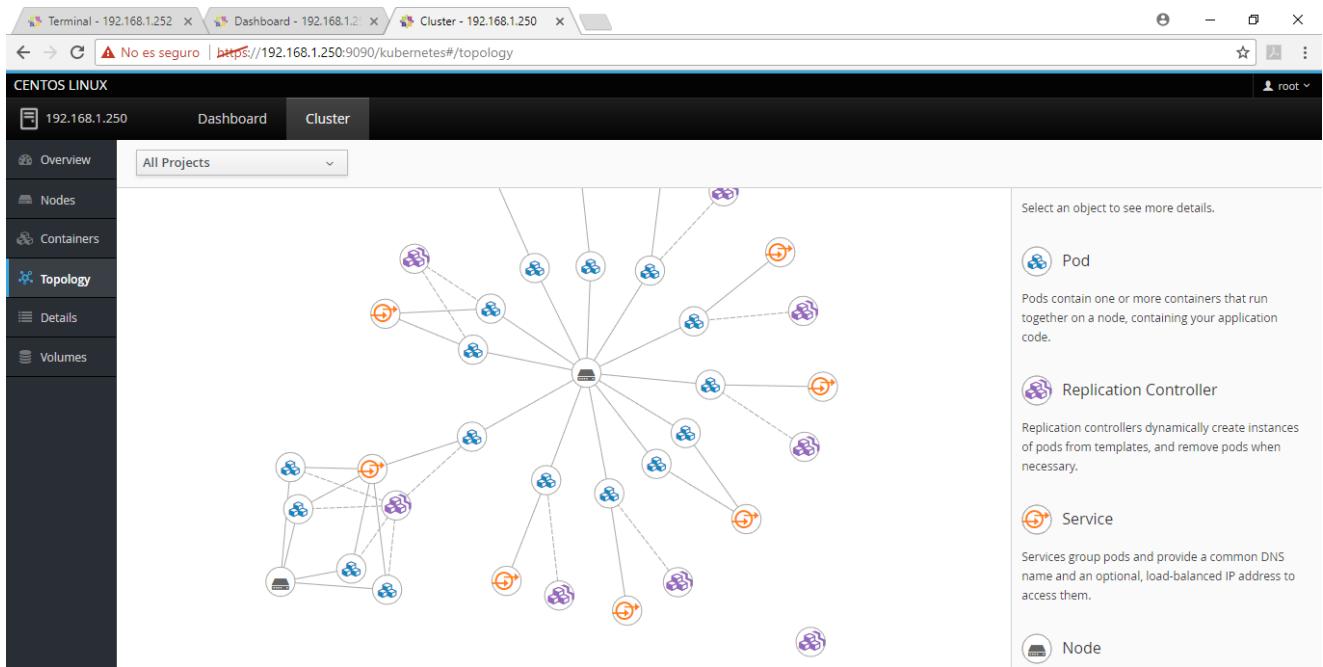
Nos validamos con las credenciales del usuario **root** del sistema, hacedemos al servidor master, que es donde podemos gestionar el cluster, y tenemos instalado el paquete cockpit-kubernetes:

<https://192.168.1.250:9090/>



The screenshot shows a web browser window with three tabs open: 'Terminal - 192.168.1.250', 'Dashboard - 192.168.1.250', and 'Cluster - 192.168.1.250'. The active tab is 'Cluster - 192.168.1.250'. A warning message 'No es seguro' is displayed above the address bar. The page is a Kubernetes Cluster dashboard. The left sidebar has navigation links for 'Overview', 'Nodes', 'Containers', 'Topology', 'Details', and 'Volumes'. The main content area has tabs for 'CPU' (selected), 'Memory', and 'Disk'. Below these tabs is a legend: red for > 90%, orange for 80-90%, yellow for 70-80%, light blue for < 70%, and grey for Unavailable. To the right of the CPU tab is a circular 'OS Versions' section showing the number '2' and a legend entry for 'CentOS Linux 7 (Core)'. Below these sections is a table titled 'Nodes' with columns 'Name', 'Operating System', 'Address', and 'Status'. It lists two nodes: 'minion1' (CentOS Linux 7 (Core), Address: minion1, Status: Ready) and 'minion2' (CentOS Linux 7 (Core), Address: minion2, Status: Ready). At the top right of the main content area is a link to 'Add Kubernetes Node'.

Name	Operating System	Address	Status
minion1	CentOS Linux 7 (Core)	minion1	Ready
minion2	CentOS Linux 7 (Core)	minion2	Ready



Tambien podemos observar el **Cockpit**, del minion1, que tiene todos los paquetes instalados de cockpit:

```
[root@minion1 ~]# rpm -qa cockpit*
```

```
[root@minion1 ~]# systemctl start cockpit
```

Tras comprobación de la aplicación Cockpit, paramos el servicio en el minion1, porque sino nos cometa la cpu de la maquina virtual, ya que tenemos muy pocos recursos de memoria y cpu para los labs.

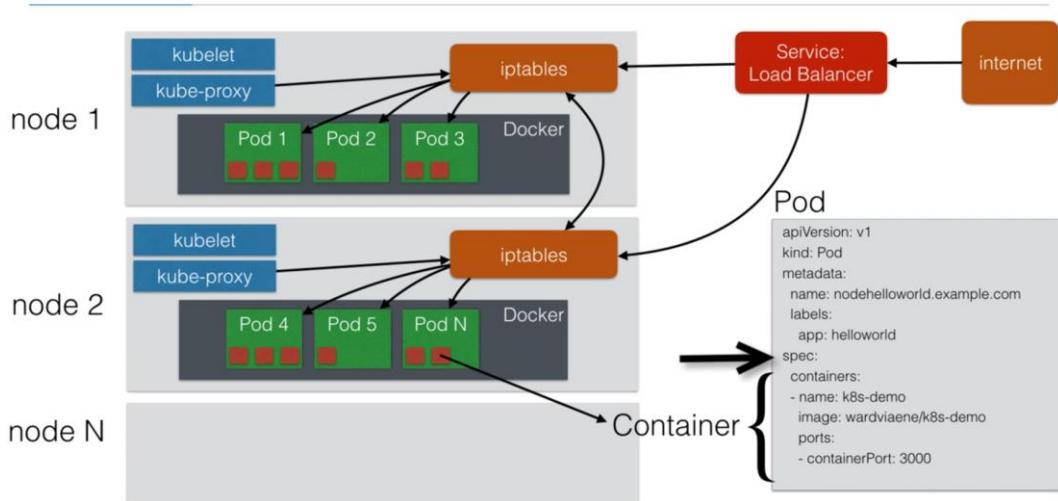
Laboratorios kubernetes Básico

En este laboratorio se resumen los conceptos básicos de Kubernetes.

Ahora, vamos a profundizar un poco más en cómo ejecutar diferentes tipos de cargas de trabajo en Kubernetes.

Primero vamos a comenzar con un diagrama de arquitectura.

Architecture overview



En verde podemos ver las pods.

Entonces, tenemos Pod 1, 2 y 3 en Nodo 1 y Pod 4, 5 y N en Nodo 2.

Estos son los pods como empezamos en la introducción, dentro de esos Pods tienes los contenedores.

Por lo tanto, los contenedores aquí se denotan en rojo.

Puede tener varios contenedores, si observa el Pod 3, ese pod tiene 2 contenedores y el Pod 1 tiene 3 contenedores.

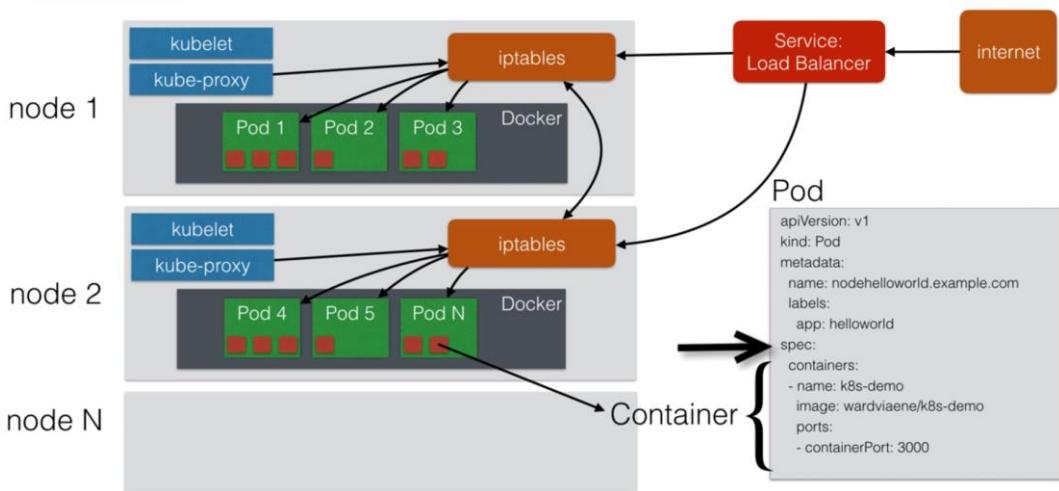
Esos contenedores pueden comunicarse fácilmente entre sí, a través de localhost

Sólo pueden utilizar el número de puerto.

Por lo tanto, un host local y un número de puerto para comunicarse entre sí si tiene dos servicios pequeños.

Digamos que uno es el backend y el otro es el servicio de autenticación.

Architecture overview



Los contenedores que están en los pods se están ejecutando usando Docker.

Entonces, en nuestros nodos tenemos Docker instalado.

También puede utilizar un motor de contenedor alternativo, rkt, cri-o.

En esos nodos también hay kubelets y un servidor kube-proxy en ejecución.

Así, los kubelets son los encargados de lanzar las pods.

Por lo tanto, se conectará con el nodo maestro para obtener esta información.

El nodo maestro no se muestra en este diagrama, solo los nodos se muestran aquí, son minions.

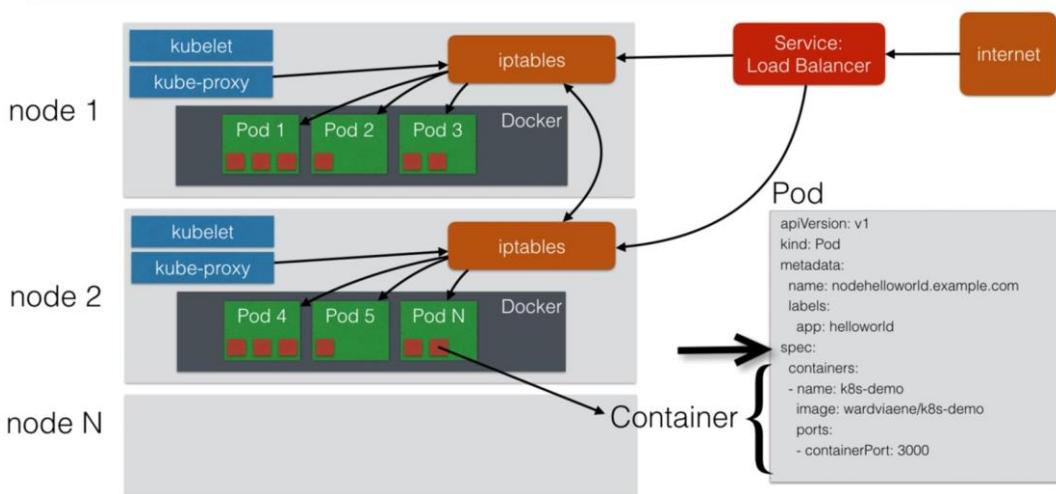
El kube-proxy va a enviar su información sobre qué pods hay en estos nodos a "iptables".

"Iptables" es el servidor de seguridad en Linux y también puede enrutar el tráfico.

Por lo tanto, cada vez que se lance un nuevo pod, kube-proxy cambiará las reglas de "iptables" para asegurarse de que el pod sea enrutable dentro del clúster.

Eso nos lleva al servicio.

Architecture overview



Y la forma más fácil de recordar cómo funciona un servicio es imaginarse un balanceador de carga.

"iptables" tiene entonces las reglas para reenviar el tráfico a otro nodo, por ejemplo, si el pod se está ejecutando en otro nodo, entonces tiene que ir a este destino.

Si el pod está en ese nodo, simplemente reenviará el tráfico al pod y al contenedor específico en este pod.

Solo tiene un puerto configurado, todos los nodos escuchan ese puerto y "iptables" enrutaría el tráfico que viene de ese puerto al módulo correcto. Para traducir esto a nuestro archivo YAML, veamos cómo se ve esto.

Todo el pod verde es en realidad la definición completa que se ve aquí a la derecha.

El contenedor es simplemente todo lo que ve en la especificación. Si tuviera dos contenedores, tendría una especificación con dos contenedores.

Laboratorio1 Ejecutando la primera aplicación en Kubernetes

En este laboratorio veremos como ejecutar nuestra primera aplicación.

Lanzaremos un contenedor basado en una imagen, necesitamos crear la "definición de pod". Un pod describe una aplicación que se ejecuta en Kubernetes.

Un pod puede contener uno o más contenedores estrechamente acoplados que conforman la aplicación. Esas aplicaciones pueden comunicarse fácilmente entre sí utilizando sus números de puerto locales.

Por lo tanto, si tuviera otra aplicación, puede conectarse a esa aplicación en ese puerto. Nuestra aplicación solo tiene un contenedor para hacer que esta aplicación se ejecute en nuestro clúster Kubernetes, crearemos un archivo pod, por ejemplo, "pod-helloworld.yml" con la definición de pod.

Create a pod

- Create a file pod-helloworld.yml with the pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
    - name: k8s-demo
      image: wardviaene/k8s-demo
      ports:
        - containerPort: 3000
```

- Use kubectl to create the pod on the kubernetes cluster:

```
$ kubectl create -f k8s-demo/pod-helloworld.yml
$
```

Comienza con la versión API v1.

Puede proporcionar metadatos, puede darle un nombre a su pod. Este pod se llama nodehelloworld.example.com

También podemos asignarle una etiqueta. Más adelante, podemos filtrar fácilmente las etiquetas.

Este pod tiene la etiqueta "app" como clave, "helloworld" como valor "espec." Donde se colocarán las definiciones del contenedor.

Solo tenemos un contenedor.

Vamos a darle un nombre, el nombre de este contenedor es "k8s-demo" y la imagen se refiere a la imagen que cargó en Docker Hub.

El contenedor expone el puerto 3000.

Puede ejecutar comandos en el pod, por lo que esto se hace simplemente ejecute un comando en el contenedor.

Si tiene varios contenedores, puede especificar contenedores específicos utilizando "**-C**".

Useful Commands

Command	Description
kubectl get pod	Get information about all running pods
kubectl describe pod <pod>	Describe one pod
kubectl expose pod <pod> --port=444 --name=frontend	Expose the port of a pod (creates a new service)
kubectl port-forward <pod> 8080	Port forward the exposed pod port to your local machine
kubectl attach <podname> -i	Attach to the pod
kubectl exec <pod> -- command	Execute a command on the pod
kubectl label pods <pod> mylabel=awesome	Add a new label to a pod
kubectl run -i --tty busybox --image=busybox --restart=Never -- sh	Run a shell in a pod - very useful for debugging

Comenzamos el laboratorio observando nuestra primera aplicación pod a desplegar en el archivo:

```
[root@docker /]# cat /kubernetes-curso/first-app/helloworld.yml
[root@docker /]# kubectl create -f /kubernetes-curso/first-app/helloworld.yml
# kubectl get pods -o wide
```

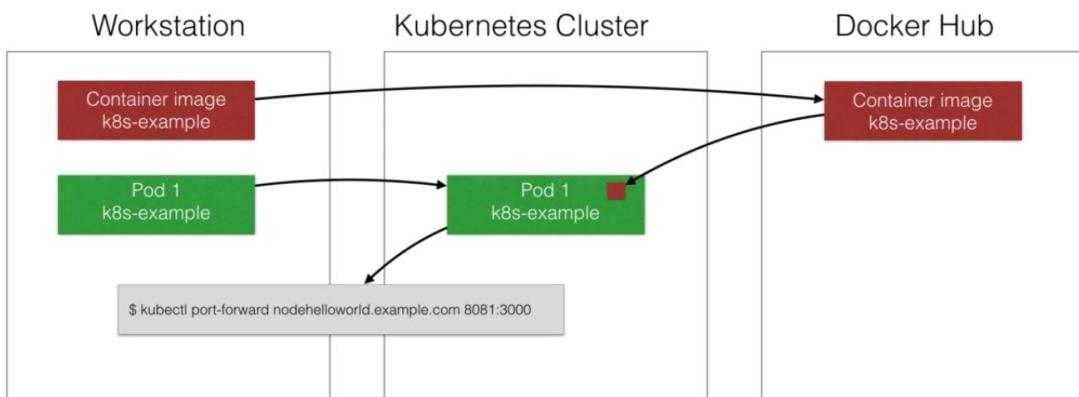
NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
NOMINATED NODE	READINESS	GATES				

```
nodehelloworld.example.com 1/1 Running 0 37s 10.36.0.22 orion3.curso.local <none>
<none>
```

Ahora veremos la descripción de nuestro pod:

```
[root@docker /]# kubectl describe pod nodehelloworld.example.com
```

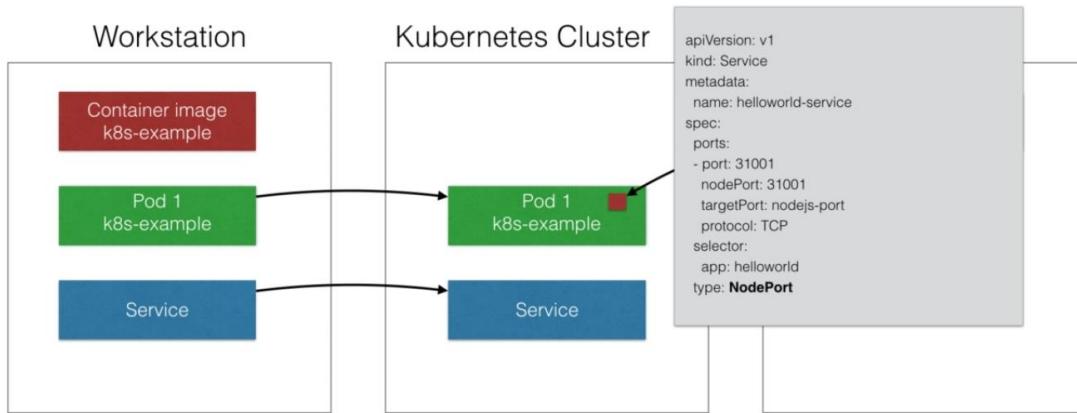
Podemos acceder a nuestro pod de forma interna a través del localhost:



```
[root@docker /]# kubectl port-forward nodehelloworld.example.com 8081:3000
```

```
[root@docker ~]# curl localhost:8081
```

Ahora si queremos acceder desde el exterior a nuestro pod, creamos un service de tipo NodePort:



```
[root@docker /]# kubectl expose pod nodehelloworld.example.com --type=NodePort --name nodehelloworld
```

```
[root@docker /]# kubectl get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	21d
nodehelloworld	NodePort	10.104.223.21	<none>	3000:32728/TCP	4s

Ahora describimos el service y vemos el puerto que mapeara en todos los nodos del cluster para acceder a nuestro contenedor:

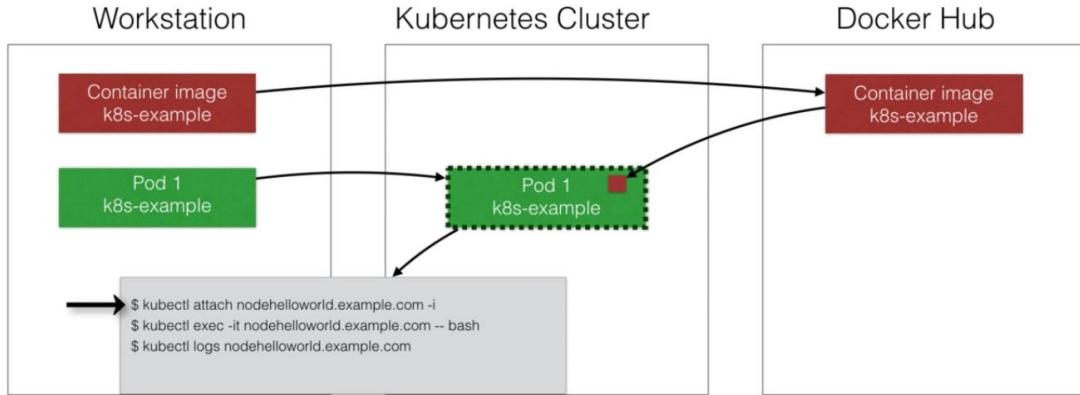
```
[root@docker /]# kubectl describe service nodehelloworld
```

```
Name:           nodehelloworld
Namespace:      default
Labels:         app=helloworld
Annotations:    <none>
Selector:       app=helloworld
Type:          NodePort
IP:            10.104.223.21
Port:          <unset> 3000/TCP
TargetPort:     3000/TCP
NodePort:       <unset> 32728/TCP
Endpoints:     10.36.0.22:3000
Session Affinity: None
External Traffic Policy: Cluster
Events:        <none>
```

<http://192.168.1.150:32728/>

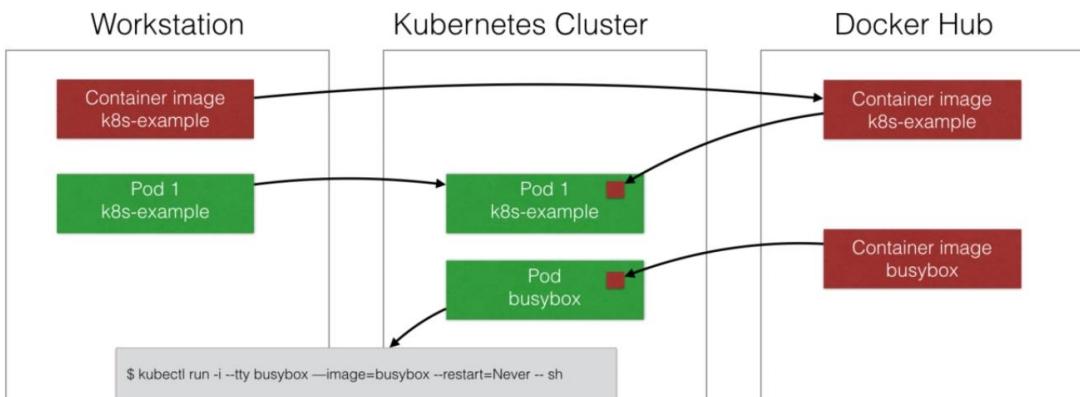


Podemos lanzar comandos dentro del contenedor:



```
#kubectl exec nodehelloworld.example.com -- ls /
#kubectl exec nodehelloworld.example.com – touch /app/hola.txt
#kubectl exec nodehelloworld.example.com – ls -l /app/
# kubectl exec -ti nodehelloworld.example.com -- bash
```

Este seria el endpoint interno para llegar al contenedor, es decir desde dentro del cluster:



```
[root@docker /]# kubectl describe service nodehelloworld
```

```
Endpoints: 10.36.0.11:3000
```

```
[root@docker /]# curl 10.36.0.11:3000
```

Lanzamos un contenedor por ejemplo de busybox y no conectamos al Endpoints:10.36.0.11:3000, que es el del servicio nodehelloworld, con lo que conectaremos a la ip y puerto del pod directamente:

```
[root@docker /]# kubectl run -i --tty busybox --image=busybox --restart=Never – sh  
# telnet 10.36.0.22 3000  
GET /  
HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 12  
ETag: W/"c-7Qdih1MuhjZehB6Sv8UNjA"  
Date: Wed, 06 Feb 2019 17:30:56 GMT  
Connection: close  
  
Hello World!Connection closed by foreign host  
/ #
```

Podemos ver los eventos en nuestro cluster:

```
[root@docker /]# kubectl get events
```

Recursos de kubernetes

Lab 2 Pods:

<https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>

<https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.15/#pod-v1-core>

La unidad de despliegue de Kubernetes no es el contenedor sino el Pod. Un Pod se compone de uno o más contenedores, que deben estar estrechamente relacionados, y es una abstracción de una máquina virtual donde corren esos contenedores. Quiere decir esto, por ejemplo, que los contenedores corriendo en el mismo Pod comparten el mismo stack de red (son accesibles entre ellos usando *localhost*). También tienen acceso a los mismos volúmenes e incluso a memoria compartida.

Es importante destacar que el Pod es la unidad de despliegue de Kubernetes. **Un Pod solo se considera healthy si todos los contenedores que lo componen están funcionando correctamente.** Aunque el caso más habitual es que un Pod solo tenga un contenedor, existen escenarios para tener más de un contenedor en el mismo Pod. Algunos ejemplos son el patrón *sidecar*, servicios de logging, o autenticación embebida.

Por ejemplo, en un pod podemos tener un contenedor que genere el contenido estático de la aplicación, y un nginx que sirve este contenido, a través de un volumen compartido entre los dos contenedores dentro del mismo pod.

Pods Practica

Si estamos utilizando vagrant para el curso podemos utilizar la ruta con los ejemplos de los archivos:

C:\kubernetes-vagrant-cluster\k8-for-devs-master

En la maquina virtual master estará en la ruta:

\$ /vagrant/k8-for-devs-master

Vamos a hacer algunos ejemplos con Pods, para lo que nos vamos al directorio `pods` del repo.

En el fichero **pod.yaml** tenemos un ejemplo muy sencillo de un Pod. En él definimos el tipo del objeto, el nombre del Pod, y los contenedores que contiene. En este caso es solo un contenedor `nginx` que expone el puerto 80, en el campo `spec`: definimos los recursos dentro del pod, que principalmente son contenedores, volúmenes, labels, etc...

Para lanzar el pod ejecutamos:

```
kubectl apply -f pod.yaml
```

Y si hacemos:

```
kubectl get all
```

veremos el Pod creándose. Con `kubectl describe pod/pod` vemos los eventos asociados a este pod.

Ahora, podemos acceder a este Pod con el comando:

```
kubectl port-forward 8081:80
```

y lo tendríamos accesible en `localhost:8081`.

Para ver los logs del Pod ejecutamos:

```
kubectl logs pod/pod
```

Ahora vamos a eliminar este Pod con el comando:

```
kubectl delete -f pod.yaml
```

Pasemos a ver ahora en el fichero `pod-volumes.yaml`. Este ejemplo muestra cómo compartir volúmenes entre contenedores dentro del mismo pod.

Para ello definimos una sección de `volumes` en la `Spec` del pod, y lo montamos con `volumeMounts` en cada uno de los contenedores.

Para conectarnos al pod (pod-volumes), al contenedor:nginx

```
$ kubectl exec -ti --namespace=lab pod-volumes -c nginx -- bash
```

Para conectarnos al pod (pod-volumes), al contenedor: content-alpine

```
$kubectl exec -ti --namespace=lab pod-volumes -c content-alpine – ash
```

Podemos ver el directorio `/pod-data/` que esta compartiendo el `index.html`, contra el contenedor `nginx` el el directorio `/usr/share/nginx/html`

Laboratorio1 Scaling Pods RC

En este laboratorio veremos como podemos scalar pods en kubernetes, trabajando con el objeto ReplicationController

Scaling

- To replicate our example app 2 times

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: helloworld-controller
spec:
  replicas: 2
  selector:
    app: helloworld
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
        - name: k8s-demo
          image: wardviaene/k8s-demo
          ports:
            - containerPort: 3000
```

El escalado en Kubernetes se puede hacer usando el "Controlador de Replicación". El replication controller garantizará que se ejecutará un número específico de réplicas de pod en todo momento.

Un pod creado con un replication controller se reemplazará automáticamente si fallan, se eliminan o se terminan.

Si le dice a Kubernetes que ejecute 5 pods y solo hay 4 en ejecución, porque 1 nodo se bloqueó, por ejemplo, Kubernetes lanzará otra instancia de ese pod en otro nodo. También se recomienda usar replicatio controller si solo quiere asegurarse de que su 1 pod siempre esté en ejecución, incluso después de reiniciar.

Nos aseguramos de eliminar en nuestro clueter, todos los elementos de kubernetes del laboratorio anterior.

Para este laboratorio utilizaremos el siguiente archivo yml:

```
[root@docker ~]# cat /kubernetes-curso/replication-controller/helloworld-repl-controller.yml
```

```
# kubectl create -f /kubernetes-curso/replication-controller/helloworld-repl-controller.yml
```

```
# kubectl get rc -o wide
```

```
# kubectl describe pod helloworld-controller-895r8
```

```
# kubectl delete pod helloworld-controller-895r8
```

Si elimino un pod el rc, levantara otro pod:

```
# kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
NOMINATED NODE	READINESS	GATES				
helloworld-controller-ckzvb	1/1	Running	0	13m	10.36.0.12	orion3.curso.local <none>
<none>						
helloworld-controller-lmhr9	1/1	Running	0	5m32s	10.44.0.22	docker2 <none>
<none>						

```
# kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
helloworld-controller-wvkbq	1/1	Running	0	23s	app=helloworld
helloworld-controller-zk4nv	1/1	Running	0	23s	app=helloworld

Para escalar nuestro rc, podemos realizarlo mediante el comando:

```
# kubectl scale --replicas=4 -f /kubernetes-curso/replication-controller/helloworld-repl-controller.yml
```

```
# kubectl get pod -o wide
```

```
# kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
helloworld-controller	4	4	4	18m

O también podemos escalar nuestro rc con el comando:

```
# kubectl scale rc helloworld-controller --replicas=1
```

```
# kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
helloworld-controller-ckzvb	1/1	Running	0	19m

Solo se pude escalar horizontalmente cuando el pod no tiene estado.

Para finalizar el laboratorio eliminamos el rc, y veremos que se eliminan los pods, asiciados a este rc:

```
#kubectl get rc
```

```
# kubectl delete rc helloworld-controller
```

```
# kubectl get pod,rc
```

nginx-rc.yaml

```

# Número de versión del api que se quiere utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: ReplicationController
# Datos propios del replication controller
metadata:
    # Nombre del Replication Controller
    name: my-nginx-rc
# La especificación del estado deseado que queremos que tenga el pod.
spec:
    # Número de réplicas que queremos que se encargue de mantener el rc. (Esto creará un pod)
    replicas: 1
    # En esta propiedad se indican todos los pods que se va a encargar de gestionar este replication controller. En este caso, se va a encargar de todos los que tengan el valor "nginx" en el label "app"
    selector:
        app: nginx
    # Esta propiedad tiene exactamente el mismo esquema interno que un pod , excepto que como está anidado no necesita ni un "apiVersion" ni un "kind"
    template:
        metadata:
            name: nginx
            labels:
                app: nginx
        spec:
            containers:
                - name: nginx
                  image: nginx
                  ports:
                      - containerPort: 80

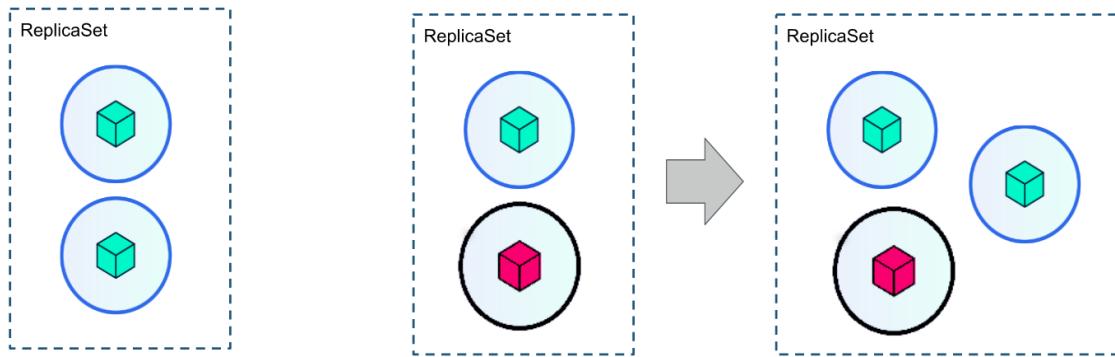
```

Lab 2 ReplicaSet

<https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>

Un **Replica Set** es un recurso de Kubernetes que asegura que siempre se ejecute un número de réplicas de un pod determinado. Por lo tanto, nos asegura que un conjunto de pods siempre están funcionando y disponibles. Nos proporciona las siguientes características:

- Que no haya caída del servicio
- Tolerancia a errores
- Escalabilidad dinámica



Definición yaml de un ReplicaSet

Vamos a ver un ejemplo de definición de ReplicaSet en el fichero `nginx-rs.yaml`:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: nginx-rs
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          name: nginx
```

- **replicas**: Indicamos el número de pos que siempre se van a estar ejecutando.
- **selector**: Indicamos el pods que vamos a replicar y vamos a controlar con el

- **ReplicaSet.** En este caso va a controlar pods que tenga un *label* `app` cuyo valor sea `nginx`. Si no se indica el campo `selector` se seleccionarán los pods cuyos labels sean iguales a los que hemos declarado en la sección siguiente.
- **template:** El recurso `ReplicaSet` contiene la definición de un `pod`.

Al crear el `ReplicaSet` se crearán los pods que hemos indicado como número de replicas:

```
$kubectl create -f nginx-rs.yaml
replicaset.extensions "nginx" created
```

Veamos el `ReplicaSet` creado y los pods que ha levantado.

```
$kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx	2	2	2	44s

```
kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
nginx-5b2rn   1/1     Running   0          1m
nginx-6kfzg   1/1     Running   0          1m
```

¿Qué pasaría si borro uno de los pods que se han creado? Inmediatamente se creará uno nuevo para que siempre estén ejecutándose los pods deseados, en este caso 2:

```
$kubectl delete pod nginx-5b2rn
```

```
pod "nginx-5b2rn" deleted
```

```
$kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
nginx-6kfzg   1/1     Running   0          2m
nginx-lkvzj   0/1     ContainerCreating   0          4s
```

Podemos escalar el número de pods que queremos que se ejecuten:

```
$kubectl scale rs nginx --replicas=5
replicaset.extensions "nginx" scaled

kubectl get pods --watch
NAME           READY   STATUS    RESTARTS   AGE
nginx-6kfzg   1/1     Running   0          5m
nginx-bz2gs   1/1     Running   0          46s
nginx-lkvzj   1/1     Running   0          3m
nginx-ssblp   1/1     Running   0          46s
nginx-xxg4j   1/1     Running   0          46s
```

Como anteriormente vimos podemos modificar las características de un `ReplicaSet` con la siguiente instrucción:

```
$kubectl edit rs nginx
```

Por último si borramos un `ReplicaSet` se borraran todos los pods asociados:

```
$kubectl delete rs nginx
replicaset.extensions "nginx" deleted
```

```
$kubectl get rs
No resources found.
```

```
$kubectl get pods
No resources found.
```

El uso del recurso `ReplicaSet` sustituye al uso del recurso [ReplicaController](#), más concretamente el uso de **Deployment** que define un `ReplicaSet`.

Replica Set Lab 2:

Vamos a ver ahora un ejemplo con Replica Sets, para lo que nos vamos al directorio `replica-sets` desde el raíz del repo.

C:\kubernetes-vagrant-cluster\k8-for-devs-master\replica-sets

En el fichero `replica-set.yaml` tenemos un ejemplo muy sencillo de un Replica Set. Estos ficheros YAML son un poco complejos de crear desde cero, lo más sencillo es hacer un *copy/paste* de uno ya creado y editarlos con los valores que deseemos.

En `replica-set.yaml` definimos el tipo del objeto, el nombre del Replica Set, el número de réplicas en el campo `replicas` y un campo `selector`, que es un conjunto de `matchLabels` y `matchExpressions` de los pods que queremos controlar desde el Replica Set.

Para lanzar el Replica Set ejecutamos:

```
kubectl apply -f replica-set.yaml
```

Y si hacemos:

```
kubectl get all
kubectl get all --namespace lab
kubectl get pod -l app --namespace lab
```

veremos el Replica Set creado y 3 pods creándose.

Si eliminamos uno de los pods y volvemos a hacer `kubectl get all` veremos que el Replica Set ha creado un nuevo pod para suplantar el pod que acabamos de eliminar.

Por último, para eliminar el Replica Set y sus pods asociados ejecutamos:

```
kubectl delete -f replica-set.yaml
```

Laboratorio Deployments en kubernetes

Los Deployments son una abstracción muy útil sobre el concepto de Replica Controllers, y es el objeto que se utiliza como norma general para desplegar nuestras aplicaciones.

Sobre la abstracción del Replica Controller ofrece **rolling updates**. De esta manera, si tenemos 4 instancias de un Pod y queremos desplegar una nueva versión de nuestra aplicación, el Deployment se encarga de ir sustituyendo uno a uno los Pods antiguos por los nuevos, de tal manera que no nos veamos afectados por un *downtime*.

Además, el Deployment mantiene un histórico de versiones de los Pods que ha ejecutado, permitiendo hacer *rollbacks* a versiones pasadas si detectamos un problema en producción.

Replication Set

- **Replica Set** is the next-generation Replication Controller
- It supports a new selector that can do selection based on **filtering** according a **set of values**
 - e.g. “environment” either “dev” or “qa”
 - not only based on equality, like the Replication Controller
 - e.g. “environment” == “dev”
- This **Replica Set**, rather than the Replication Controller, is used by the Deployment object

Deployments

- A deployment declaration in Kubernetes allows you to do app **deployments** and **updates**
- When using the deployment object, you define the **state** of your application
 - Kubernetes will then make sure the clusters matches your **desired** state
- Just using the **replication controller** or **replication set** might be **cumbersome** to deploy apps
 - The **Deployment Object** is easier to use and gives you more possibilities

Deployments

- With a deployment object you can:
 - Create** a deployment (e.g. deploying an app)
 - Update** a deployment (e.g. deploying a new version)
 - Do **rolling updates** (zero downtime deployments)
 - Roll back** to a previous version
 - Pause / Resume** a deployment (e.g. to roll-out to only a certain percentage)

Deployments

- This is an example of a deployment:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
        - name: k8s-demo
          image: wardviaene/k8s-demo
      ports:
        - containerPort: 3000
```

Useful Commands

Command	Description
kubectl get deployments	Get information on current deployments
kubectl get rs	Get information about the replica sets
kubectl get pods --show-labels	get pods, and also show labels attached to those pods
kubectl rollout status deployment/helloworld-deployment	Get deployment status
kubectl set image deployment/helloworld-deployment k8s-demo=k8s-demo:2	Run k8s-demo with the image label version 2
kubectl edit deployment/helloworld-deployment	Edit the deployment object
kubectl rollout status deployment/helloworld-deployment	Get the status of the rollout
kubectl rollout history deployment/helloworld-deployment	Get the rollout history
kubectl rollout undo deployment/helloworld-deployment	Rollback to previous version
kubectl rollout undo deployment/helloworld-deployment --to-revision=n	Rollback to any version version

Comenzamos el laboratorio visualizando el archivo de nuestro deployment y comentándolo:

```
# cat /kubernetes-curso/deployment/helloworld.yml

# kubectl create -f /kubernetes-curso/deployment/helloworld.yml --record

#kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE
helloworld-deployment 3/3 3 3 30s
```

Buscamos nuestro replica set:

```
# kubectl get rs

NAME DESIRED CURRENT READY AGE
helloworld-deployment-969d5cbd5 3 3 3 5m36s

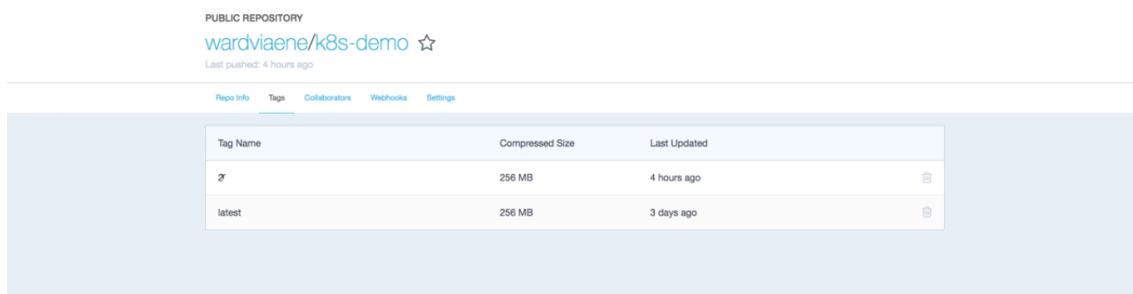
# kubectl get pod --show-labels

NAME READY STATUS RESTARTS AGE LABELS
helloworld-deployment-969d5cbd5-g42nz 1/1 Running 0 6m15s app=helloworld,pod-template-hash=969d5cbd5
helloworld-deployment-969d5cbd5-hcpqm 1/1 Running 0 6m15s app=helloworld,pod-template-hash=969d5cbd5
helloworld-deployment-969d5cbd5-mg6tk 1/1 Running 0 6m15s app=helloworld,pod-template-hash=969d5cbd5
```

Visualizamos el estado de nuestro deployment:

```
# kubectl rollout status deployment helloworld-deployment
```

En el repositorio de la imagen con la que estamos realizando este laboratorio vemos que tenemos dos versiones:



Ahora exponemos nuestro deployment para poder llegar desde el exterior:

```
# kubectl expose deployment helloworld-deployment --type=NodePort
```

```
#kubectl get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	ClusterIP	10.100.177.133	<none>	4444/TCP	11h
helloworld-deployment	NodePort	10.96.37.189	<none>	3000:30911/TCP	25s

Describimos nuestro service creado con el comando anterior:

```
[root@docker ~]# kubectl describe service helloworld-deployment
```

Name: helloworld-deployment

Namespace: default

Labels: app=helloworld

Annotations: <none>

Selector: app=helloworld

Type: NodePort

IP: 10.104.167.203

Port: <unset> 3000/TCP

TargetPort: 3000/TCP

NodePort: <unset> 31518/TCP

Endpoints: 10.36.0.23:3000,10.36.0.24:3000,10.36.0.25:3000 + 2 more...

Session Affinity: None

External Traffic Policy: Cluster

Events: <none>

Accedemos desde el navegador de nuestro equipo a los pods, a través del puerto de nuestro servicio:

<http://192.168.1.150:31518/>

Ahora actualizamos nuestro deployment con la imagen k8s-demo:2, que sería la nueva versión de nuestra aplicación:

```
# kubectl set image deployment/helloworld-deployment k8s-demo=wardviaene/k8s-demo:2 --record
```

```
# curl 192.168.1.155:32584
```

Hello World v2!

Ahora podemos ver el historial de las versiones de nuestro deployment:

```
# kubectl rollout history deployment helloworld-deployment
```

Ahora podemos volver atras en nuestras aplicaciones, es decir a la versión anterior:

```
# kubectl rollout undo deployment helloworld-deployment
```

```
# kubectl rollout status deployment helloworld-deployment
```

```
# curl 192.168.1.155:31518
```

Hello World!

Ahora podemos comprobar el historial de los deployments:

```
# kubectl rollout history deployment helloworld-deployment
```

Por defecto siempre mantiene dos revisiones

Con este comando cambiamos el límite de revisiones por ejemplo a 100:

```
# kubectl edit deployments helloworld-deployment
```

```
spec:
  replicas: 3
  revisionHistoryLimit: 100
```

```
# kubectl set image deployment/helloworld-deployment k8s-demo=wardviaene/k8s-demo:2
```

```
# kubectl rollout history deployment helloworld-deployment
```

```
deployment.extensions/helloworld-deployment
```

```
REVISION CHANGE-CAUSE
```

```
3 <none>
```

```
4 <none>
```

Ahora podemos volver a la versión 1 de nuestra imagen

```
# kubectl set image deployment/helloworld-deployment k8s-demo=wardviaene/k8s-demo:1
```

```
REVISION CHANGE-CAUSE
```

```
3 <none>
```

```
4 <none>
```

```
5 <none>
```

Ahora podemos comprobar que tenemos 3 revisiones

```
# kubectl rollout history deployment helloworld-deployment
```

Ahora volvemos a la revisión 3, que en realidad es una nueva revisión porque se ha convertido en la 6.

```
# kubectl rollout undo deployment helloworld-deployment --to-revision=3
```

```
# kubectl rollout history deployment helloworld-deployment
```

REVISION CHANGE-CAUSE

```
4    <none>
5    <none>
6    <none>
```

Lab 2 Deployment

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

Los Deployments son una abstracción muy útil sobre el concepto de Replica Controllers, y es el objeto que se utiliza como norma general para desplegar nuestras aplicaciones.

Sobre la abstracción del Replica Controller ofrece *rolling updates*. De esta manera, si tenemos 4 instancias de un Pod y queremos desplegar una nueva versión de nuestra aplicación, el Deployment se encarga de ir sustituyendo uno a uno los Pods antiguos por los nuevos, de tal manera que no nos veamos afectados por un *downtime*. Además, el Deployment mantiene un histórico de versiones de los Pods que ha ejecutado, permitiendo hacer *rollbacks* a versiones pasadas si detectamos un problema en producción.

Deployment Práctica

Vamos a ver ahora un ejemplo con Deployments, para lo que nos vamos al directorio `deployments` desde el raíz del repo.

C:\kubernetes-vagrant-cluster\k8-for-devs-master\replica-sets

Podemos eliminar todo lo realizado en la práctica anterior con, ojo al posicionamiento del directorio actual:

```
vagrant@master:/vagrant/k8-for-devs-master/replica-sets$ kubectl delete -f .
```

En el fichero `deployment.yaml` tenemos un ejemplo muy sencillo de un Deployment. En `deployment.yaml` definimos el tipo del objeto, el nombre del Deployment, y la definición en `Spec` de un Replica Set.

Para lanzar el Deployment ejecutamos:

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

Y si hacemos:

```
kubectl get all
kubectl get deployments
```

Editamos nuestro deployment, esto es una forma de modificar en caliente objetos en kubernetes:

```
$ kubectl edit deployments deployment
```

veremos el Deployment, el Replica Set creado y 3 pods creándose. El YAML de Deployment nos permite también definir el tipo de estrategia para *upgrades* en el campo *strategy* y cuántas versiones del Deployment vamos a almacenar de cara a hacer *rollbacks*.

##Salida del deployment que estaría corriendo en nuestro cluster de kubernetes:

```
# Please edit the object below. Lines beginning with a '#' will be
ignored,
# and an empty file will abort the edit. If an error occurs while
saving this file will be
# reopened with the relevant failures.
#
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
    creationTimestamp: "2019-08-12T08:24:25Z"
    generation: 1
  labels:
    app: nginx
    name: deployment
    namespace: default
    resourceVersion: "422515"
    selfLink:
      /apis/extensions/v1beta1/namespaces/default/deployments/deployment
      uid: 98440a25-bcda-11e9-8186-080027c5bc64
spec:
  progressDeadlineSeconds: 600
  replicas: 3
```

```

##Estamos guardando 10 versiones en el historial.
revisionHistoryLimit: 10
selector:
  matchLabels:
    app: nginx
strategy:
  rollingUpdate:
    ##Como mucho un 25% de las instancias extras se pueden estar creando
    dentro del ru rollingupdate
    maxSurge: 25%
    ##Como mucho un 25% de las instancias no pueden estar
    ##disponibles cuando estemos haciendo el rollingupdate
    maxUnavailable: 25%
##Se puede realizar una estrategia Recreate, que hace de uno en uno
  type: RollingUpdate
template:
  metadata:
    creationTimestamp: null
  labels:
    app: nginx

```

**Con el comando `rollout` gestionamos la historia de un deployment.
Por ejemplo:**

```
kubectl rollout history deployment.apps/deployment
```

Si cambiamos el *manifest* para cambiar la imagen de `nginx` y aplicamos los cambios, por ejemplo `image: nginx:latest`

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

Y hacemos un:

```
kubectl rollout status deployment.apps/deployment
```

nos muestra el estado del despliegue. Si volvemos a hacer:

```
kubectl rollout history deployment.apps/deployment
```

vemos que ahora tenemos dos versiones en el historial. Si ahora necesitamos hacer un *rollback* a la versión anterior, podemos ejecutar:

```
kubectl rollout undo deployment.apps/deployment
```

que nos devuelve el deployment a la versión inicial.

Laboratorio kubernetes Services

En este laboratorio veremos como trabajar con servicios dentro del un clúster de kubernetes:

Services

- **Pods** are very **dynamic**, they come and go on the Kubernetes cluster
 - When using a **Replication Controller**, pods are **terminated** and created during scaling operations
 - When using **Deployments**, when **updating** the image version, pods are **terminated** and new pods take the place of older pods
- That's why Pods should never be accessed directly, but always through a **Service**
- A service is the **logical bridge** between the “mortal” pods and other **services** or **end-users**

Services

- When using the “kubectl expose” command earlier, you created a new Service for your pod, so it could be accessed externally
- Creating a service will create an endpoint for your pod(s):
 - a **ClusterIP**: a virtual IP address only reachable from within the cluster (*this is the default*)
 - a **NodePort**: a port that is the same on each node that is also reachable externally
 - a **LoadBalancer**: a LoadBalancer created by the **cloud provider** that will route external traffic to every node on the NodePort (ELB on AWS)

Services

- The options just shown only allow you to create **virtual IPs** or **ports**
- There is also a possibility to use **DNS names**
 - **ExternalName** can provide a DNS name for the service
 - e.g. for service discovery using DNS
 - This only works when the **DNS add-on** is enabled
- I will discuss this later in a **separate** lecture
- This is an example of a Service definition (also created using kubectl expose):

```
apiVersion: v1
kind: Service
metadata:
  name: helloworld-service
spec:
  ports:
    - port: 31001
      nodePort: 31001
      targetPort: nodejs-port
      protocol: TCP
  selector:
    app: helloworld
  type: NodePort
```

Comenzamos el laboratorio, eliminando todos los laboratorios anteriores de kubernetes

Desplegaremos nuestro pod:

```
# cat /kubernetes-curso/first-app/helloworld.yml
# kubectl create -f /kubernetes-curso/first-app/helloworld.yml
# kubectl get pods -o wide
# kubectl describe pod nodehelloworld.example.com
```

Ahora observamos el archivo de configuración de nuestro servicio, es importante fijarse en el selector **app: helloworld, que se relaciona con el pod desplegado anteriormente**

```
# cat /kubernetes-curso/first-app/helloworld-nodeport-service.yml
# kubectl create -f /kubernetes-course/first-app/helloworld-nodeport-service.yml
# kubectl get service -o wide
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
helloworld-service	NodePort	10.98.238.201	<none>	31001:31001/TCP	24s	app=helloworld

Comprobamos que llegamos a nuestro pod a través del servicio:

<http://192.168.1.150:31001/>

```
# curl 192.168.1.152:31001
Hello World!
# kubectl describe service helloworld-service
# kubectl delete service helloworld-service
```

Lab 2 Services

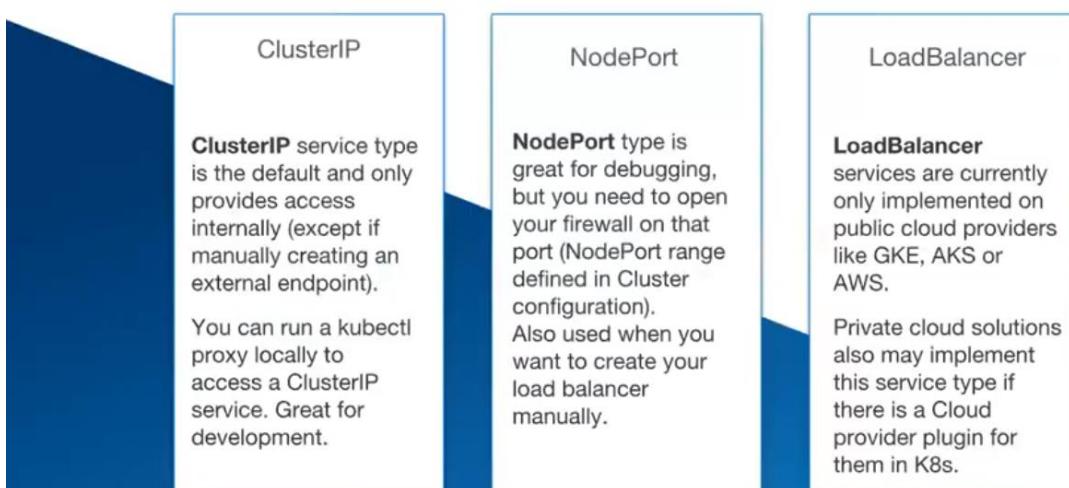
<https://kubernetes.io/docs/concepts/services-networking/service/>

Si algo hemos visto hasta este momento es que la vida de un Pod es muy dinámica. **Los Pods se crean y destruyen constantemente en un clúster de Kubernetes**, por lo que el acceso entre aplicaciones no se puede basar en la IP de los Pods. De alguna manera, es necesario tener un *endpoint* permanente que de acceso a un conjunto de Pods.

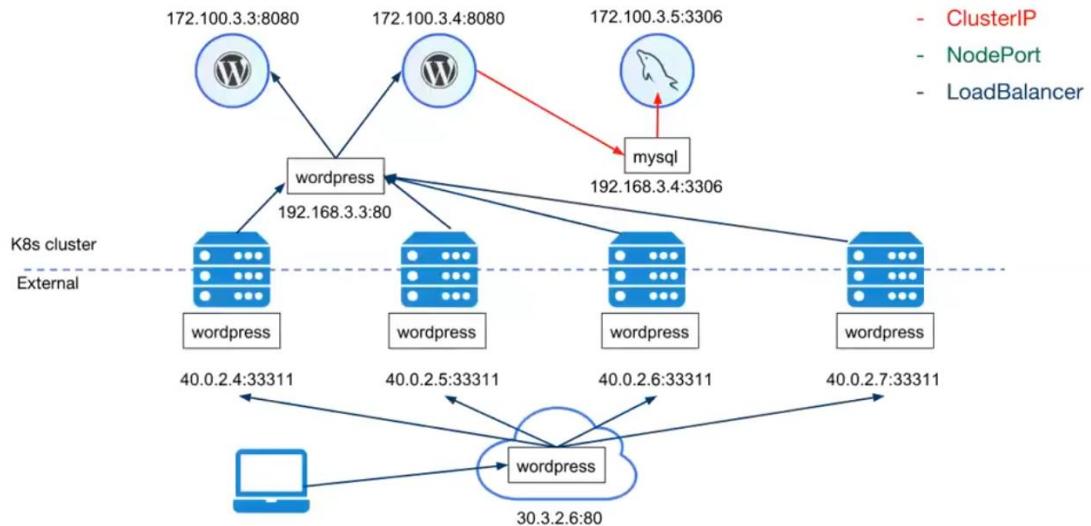
Esta es la función de un Service. **Un Service se asocia a un conjunto de Pods**. Cada Service tiene un endpoint de acceso fijo que se corresponde con su nombre (aunque también podríamos acceder a él vía un *Load Balancer* externo o un *ingress*). *Por tanto, cada vez que accedamos por DNS a nombre de un servicio, el servicio redirigirá la petición a uno de los Pods a los que está asociado, ofreciendo un mecanismo eficaz de service discovery y de balanceo de carga.*

Los diferentes tipos de servicio que tenemos son ClusterIP, NodePort (30000:32767) y LoadBalancer.

Services Types



Basic Objects: Services



Services

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    app: nginx
```

You can also expose a deployment on the fly. Let's try it:

```
$ kubectl expose deployment/nginx --port=80
--type=NodePort
$ kubectl get svc
NAME      TYPE      CLUSTER-IP
EXTERNAL-IP  PORT(S)   AGE
nginx     NodePort   10.100.56.229
<none>    80:30306/TCP  8s
```

DNS

```
$ kubectl create -f busybox.yaml
$ kubectl exec -it busybox -- nslookup nginx
Server: 10.0.0.10
Address 1: 10.0.0.10

Name: nginx
Address 1: 10.0.0.112

$ kubectl get svc
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes 10.0.0.1 <none> 443/TCP 19h
nginx 10.0.0.112 nodes 80/TCP 36m

$ kubectl exec -ti busybox -- wget http://nginx
Connecting to nginx (10.0.0.112:80)
index.html 100% [*****] 612 0:00:00 ETA
```

DNS

A [DNS service](#) is provided as a Kubernetes add-on in clusters. On GKE, Minikube and K8s Sandbox this DNS service is provided by default.

A service gets registered in DNS and DNS lookup will further direct traffic to one of the matching Pods via the ClusterIP of the service.

Services (Práctica)

C:\kubernetes-vagrant-cluster\k8-for-devs-master\services

Vamos a ver ahora un ejemplo con Services, para lo que nos vamos al directorio services desde el raíz del repo.

En el fichero `service.yaml` tenemos un ejemplo muy sencillo de un Service. En él definimos el tipo del objeto, el nombre del Service, un campo `selector` para seleccionar los pods a los que aplica el servicio, y el mapeo de puertos entre en servicio y los pods. En este caso usamos un servicio `NodePort` que nos balancea a el puerto 80 de los pods desde el puerto 30080 de la máquina de Minikube.

Para lanzar el Service ejecutamos:

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

Y si hacemos:

```
kubectl get all
```

veremos el Service ya creado. Desde este momento podemos acceder a los pods usando la ip de Minikube:

```
minikube ip
```

y accediendo desde el navegador:

```
192.168.99.100:30080
```

Además, si accedemos con `kubectl exec` a uno de los pods, vemos que podemos acceder a los pods a través de la ip del servicio, y usando directamente `service`, que es el nombre del servicio, lo que resulta muy útil para *service discovery*.

Por último, para eliminar el Service ejecutamos:

```
kubectl delete -f service.yaml
```

Y para eliminar el Deployment, su replica set y pods asociados ejecutamos:

```
kubectl delete deployment.apps/deployment
```

Laboratorio kubernetes Deployment

En este laboratorio trabajaremos con las operaciones básicas de kubernetes, pods, rc, services, deployments....

Using kubectl to Create a Deployment

```
[root@master ~]# kubectl get nodes -help
```

```
[root@master ~]# kubectl version
```

```
Client Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.2",
GitCommit:"269f928217957e7126dc87e6adfa82242bfe5b1e", GitTreeState:"clean", BuildDate:"2017-07-03T15:31:10Z", GoVersion:"go1.7.4", Compiler:"gc", Platform:"linux/amd64"}
```

```
Server Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.2",
GitCommit:"269f928217957e7126dc87e6adfa82242bfe5b1e", GitTreeState:"clean", BuildDate:"2017-07-03T15:31:10Z", GoVersion:"go1.7.4", Compiler:"gc", Platform:"linux/amd64"}
```

```
[root@master ~]# kubectl get nodes
```

NAME	STATUS	AGE
minion1	Ready	19h
minion2	Ready	19h

Obtenemos información de los pods que tenemos corriendo:

```
[root@master ~]# kubectl get pods
```

```
[root@master ~]# kubectl describe pods
```

```
[root@master ~]# export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}\n{{end}}')
```

```
[root@master ~]# echo Name of the Pod: $POD_NAME
```

Vamos a ejecutar nuestra primera aplicación en Kubernetes con el comando kubectl run. El comando run crea un nuevo deployment. Tenemos que proporcionar el nombre del deployment y la ubicación de la imagen de la aplicación (incluya la url completa del repositorio para las imágenes alojadas fuera del hub de Docker). Queremos ejecutar la aplicación en un puerto específico, por lo que agregamos el parámetro --port

```
[root@master ~]# kubectl run kubernetes-bootcamp --image=docker.io/jocatalin/kubernetes-bootcamp:v1 --port=8080
```

deployment "kubernetes-bootcamp" created

```
[root@master ~]# kubectl get deployments
```

```
[root@master ~]# kubectl describe deployment kubernetes-bootcamp
```

Name:	kubernetes-bootcamp					
Namespace:	default					
CreationTimestamp:	Thu, 17 Aug 2017 11:11:43 +0200					
Labels:	run=kubernetes-bootcamp					
Selector:	run=kubernetes-bootcamp					
Replicas:	1 updated 1 total 1 available 0 unavailable					
StrategyType:	RollingUpdate					
MinReadySeconds:	0					
RollingUpdateStrategy: 1 max unavailable, 1 max surge						
Conditions:						
Type	Status Reason					
---	-----					
Available	True MinimumReplicasAvailable					
OldReplicaSets: <none>						
NewReplicaSet: kubernetes-bootcamp-390780338 (1/1 replicas created)						
Events:						
FirstSeen	LastSeen	Count From	SubObjectPath	Type	Reason	Message
-----	-----	----	-----	-----	-----	-----
3m	3m	1 {deployment-controller }				

Podemos haceder al contedor con los comandos:

```
[root@master ~]# kubectl get pod,rc,services
```

NAME	READY	STATUS	RESTARTS	AGE
po/busybox	1/1	Running	7	16h
po/kubernetes-bootcamp-390780338-v2p88	1/1	Running	0	5m

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

svc/kubernetes	10.254.0.1	<none>	443/TCP	20h
----------------	------------	--------	---------	-----

```
[root@master ~]# kubectl describe po/kubernetes-bootcamp-390780338-v2p88
```

Name: kubernetes-bootcamp-390780338-v2p88

Namespace: default

Node: minion1/192.168.1.251

Start Time: Thu, 17 Aug 2017 11:11:44 +0200

Labels: pod-template-hash=390780338

run=kubernetes-bootcamp

Status: Running

IP: 172.17.10.5

Controllers: ReplicaSet/kubernetes-bootcamp-390780338

Containers:

kubernetes-bootcamp:

Container ID: docker://f4fe00e8c7615d4009dcf98bef713f0494ca65972cdbdd46a11536b51cc60618

Image: docker.io/jocatalin/kubernetes-bootcamp:v1

Image ID: docker-pullable://docker.io/jocatalin/kubernetes-bootcamp@sha256:0d6b8ee63bb57c5f5b6156f446b3bc3b3c143d233037f3a2f00e279c8fcc64af

Port: 8080/TCP

State: Running

Started: Thu, 17 Aug 2017 11:12:44 +0200

Ready: True

Restart Count: 0

Para haceder al pod:

```
[root@master ~]# curl 172.17.10.5:8080
```

Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1

Para poder ver los logs de los pods:

```
[root@master ~]# kubectl logs $POD_NAME
[root@master ~]# kubectl logs kubernetes-bootcamp-390780338-v2p88
```

Para ejecutar comando en el contenedor:

```
[root@master ~]# kubectl exec $POD_NAME env
[root@master ~]# kubectl exec kubernetes-bootcamp-390780338-v2p88 env
```

Para abrir una consola en el contenedor:

```
[root@master ~]# kubectl exec -ti kubernetes-bootcamp-390780338-v2p88 bash
root@kubernetes-bootcamp-390780338-v2p88:/# cat server.js
root@kubernetes-bootcamp-390780338-v2p88:/# exit
```

Crear un servicio para exponer el pod al exterior de tipo NodePort:

```
[root@master ~]# kubectl get pod,svc
[root@master ~]# kubectl expose deployment/kubernetes-bootcamp --
type="NodePort" --port 8080
[root@master ~]# kubectl get services
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.254.0.1	<none>	443/TCP	20h
kubernetes-bootcamp	10.254.215.176	<nodes>	8080:31503/TCP	2m

```
[root@master ~]# kubectl describe service kubernetes-bootcamp
```

```
Name:           kubernetes-bootcamp
Namespace:      default
Labels:         run=kubernetes-bootcamp
Selector:       run=kubernetes-bootcamp
Type:          NodePort
IP:            10.254.215.176
Port:          <unset> 8080/TCP
NodePort:    <unset> 31503/TCP
Endpoints:   172.17.10.5:8080
Session Affinity: None
```

Con este comando podemos haceder al pod desde el exterior a través del Endpoints, o lo que hace NodePort, es exponer en todos los minions el puerto 31503, a través de los cuales podemos hacer al contenedor desde el exterior.

```
[root@minion1 ~]# netstat -tan |grep 31503
```

```
tcp6      0      0 :::31503          :::*          LISTEN
```

```
[root@minion2 ~]# netstat -tan |grep 31503
```

```
tcp6      0      0 :::31503          :::*          LISTEN
```

```
http://192.168.1.251:31503/
```



```
http://192.168.1.252:31503/
```



Lo normal seria pasarlo a través de un balanceador (lo realizaremos a través de Haproxy).

También podemos crear una variable de entorno en la que podamos ver los puertos que tengamos expuestos:

```
[root@master ~]# export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')
```

```
[root@master ~]# echo NODE_PORT=$NODE_PORT
```

Para correr multiples instancias de nuestro contenedor, es decir escalar el contenedor, en nuestro laboratorio solo tenemos un contenedor o una sola instancia corriendo de kubernetes-bootcamp:

```
[root@master ~]# kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
kubernetes-bootcamp	1	1	1	1	31m

Como podemos observar solo tenemos corriendo 1 instancia.

A continuación, vamos a escalar el despliegue a 4 réplicas. Usaremos el comando kubectl scale, seguido por el tipo de despliegue, el nombre y el número de instancias deseado:

```
[root@master ~]# kubectl scale deployments/kubernetes-bootcamp --replicas=4
```

```
deployment "kubernetes-bootcamp" scaled
```

```
[root@master ~]# kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
kubernetes-bootcamp	4	4	4	4	36m

Podemos observar los contenedores corriendo y que ip están listando en que nodo están corriendo, para poder llegar desde el exterior a los contenedores, podemos pasar a través de un balanceador (haproxy), o a través del NodePort de un minion y tendremos que ver como van cambiando los nombres de los contenedores:

```
[root@master ~]# kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
busybox	1/1	Running	7	17h	172.17.10.2	minion1
kubernetes-bootcamp-390780338-1sx7m	1/1	Running	0	2m	172.17.10.6	minion1
kubernetes-bootcamp-390780338-hqgv3	1/1	Running	0	2m	172.17.22.4	minion2
kubernetes-bootcamp-390780338-mtc3h	1/1	Running	0	2m	172.17.22.5	minion2
kubernetes-bootcamp-390780338-v2p88	1/1	Running	0	36m	172.17.10.5	minion1

```
[root@master ~]# kubectl describe deployments/kubernetes-bootcamp
```

```
Name: kubernetes-bootcamp
```

```
Namespace: default
```

CreationTimestamp: Thu, 17 Aug 2017 11:11:43 +0200

Labels: run=kubernetes-bootcamp

Selector: run=kubernetes-bootcamp

Replicas: 4 updated | 4 total | 4 available | 0 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge

Conditions:

Type	Status	Reason
---	---	---
Available	True	MinimumReplicasAvailable

OldReplicaSets: <none>

NewReplicaSet: kubernetes-bootcamp-390780338 (4/4 replicas created)

Events:

FirstSeen	LastSeen	Count	From	SubObjectPath	Type	Reason
Message	-----	-----	-----	-----	-----	-----
40m	40m	1	{deployment-controller}		Normal	ScalingReplicaSet
			Scaled up replica set kubernetes-bootcamp-390780338 to 1			
6m	6m	1	{deployment-controller}		Normal	ScalingReplicaSet
			Scaled up replica set kubernetes-bootcamp-390780338 to 4			

[root@master ~]# kubectl get pod,rc,services,deployments

NAME	READY	STATUS	RESTARTS	AGE
po/busybox	1/1	Running	7	17h
po/kubernetes-bootcamp-390780338-1sx7m	1/1	Running	0	7m
po/kubernetes-bootcamp-390780338-hqgv3	1/1	Running	0	7m
po/kubernetes-bootcamp-390780338-mtc3h	1/1	Running	0	7m
po/kubernetes-bootcamp-390780338-v2p88	1/1	Running	0	41m

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/kubernetes	10.254.0.1	<none>	443/TCP	21h
svc/kubernetes-bootcamp	10.254.215.176	<nodes>	8080:31503/TCP	23m

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/kubernetes-bootcamp	4	4	4	4	41m

```
[root@master ~]# export NODE_PORT=$(kubectl get services/kubernetes-bootcamp -o yaml | grep 'nodePort: ' | awk '{print $2}' | sed 's/\r//')
```

```
[root@master ~]# echo NODE_PORT=$NODE_PORT
```

```
NODE_PORT=31503
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-mtc3h | v=1
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-mtc3h | v=1
```

```
[root@master ~]# curl minion1:31503
```

Para reducir el servicio a 2 réplicas, vuelva a ejecutar el comando scale:

```
[root@master ~]# kubectl scale deployments/kubernetes-bootcamp --replicas=2
```

```
deployment "kubernetes-bootcamp" scaled
```

```
[root@master ~]# kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
kubernetes-bootcamp	2	2	2	2	46m

Ahora podemos observar que solo tenemos dos pods:

```
[root@master ~]# kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
minion1	1/1	Running	0	46m	10.0.2.15	minion1

```

busybox           1/1    Running   7      17h    172.17.10.2  minion1
kubernetes-bootcamp-390780338-1sx7m 1/1    Running   0      13m    172.17.10.6  minion1
kubernetes-bootcamp-390780338-v2p88 1/1    Running   0      47m    172.17.10.5  minion1

```

Comprobamos el balanceo del NodePort:

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-1sx7m | v=1
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1
```

Rolling Upgrade

Un Rolling Upgrade permite actualizar un sistema distribuido mientras los miembros del grupo son actualizados gradualmente, el objetivo es que el usuario no observe caídas, ni perdida de servicio.

```
[root@master ~]# kubectl get pods,deployments
```

NAME	READY	STATUS	RESTARTS	AGE
po/busybox	1/1	Running	8	17h
po/kubernetes-bootcamp-390780338-1sx7m	1/1	Running	0	24m
po/kubernetes-bootcamp-390780338-v2p88	1/1	Running	0	58m

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/kubernetes-bootcamp	2	2	2	58m	

```
[root@master ~]# kubectl describe pods
```

Si hacedemos a un pod de la aplicación bootcamp podemos observar que estamos corriendo la **v=1**.

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-390780338-v2p88 | v=1
```

Para actualizar la imagen de la aplicación a la versión 2, utilice el comando set image, seguido por el nombre de la implementación y la nueva versión de la imagen:

```
[root@master ~]# kubectl set image deployments/kubernetes-bootcamp
kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2
deployment "kubernetes-bootcamp" image updated
```

El comando notificó a la implementación que utilizaba una imagen diferente para su aplicación e inició una actualización continua, es decir crea un nuevo pod de la v2 y para el de la v1, sin perdida de servicio para el usuario. Compruebe el estado de los nuevos Pods y vea el antiguo que termina con el comando get pods:

```
[root@master ~]# kubectl get pods
```

Comprobamos la nueva versión:

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-2100875782-vlp9q |
v=2
```

```
[root@master ~]# curl minion1:31503
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-2100875782-0r7n0 |
v=2
```

La actualización también se puede confirmar ejecutando un comando de estado de despliegue:

```
[root@master ~]# kubectl rollout status deployments/kubernetes-bootcamp
```

```
deployment "kubernetes-bootcamp" successfully rolled out
```

Ahora vamos a realizar otra actualización e implementar la imagen etiquetada como v1:

```
[root@master ~]# kubectl set image deployments/kubernetes-bootcamp kubernetes-
bootcamp=jocatalin/kubernetes-bootcamp:v10
```

```
deployment "kubernetes-bootcamp" image updated
```

```
[root@master ~]# kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
kubernetes-bootcamp	2	3	2	1	5h

```
[root@master ~]# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
busybox	1/1	Running	13	22h
kubernetes-bootcamp-1951388213-0hhtc	0/1	ImagePullBackOff	0	26s
kubernetes-bootcamp-1951388213-7zv8x	0/1	ImagePullBackOff	0	26s
kubernetes-bootcamp-2100875782-0r7n0	1/1	Running	0	4h
kubernetes-bootcamp-2100875782-vlp9q	1/1	Terminating	0	4h

```
[root@master ~]# kubectl describe pods
```

Si queremos revertir el deployment en el estado conocido anterior (v2 de la imagen). Las actualizaciones se versionan y se puede volver a cualquier estado previamente conocido de un Despliegue. Lista nuevamente los Pods:

```
[root@master ~]# kubectl rollout undo deployments/kubernetes-bootcamp
deployment "kubernetes-bootcamp" rolled back
```

Podemos comprobar las versiones:

```
[root@master ~]# kubectl rollout history deployments/kubernetes-bootcamp
deployments "kubernetes-bootcamp"
REVISION      CHANGE-CAUSE
1            <none>
3            <none>
4            <none>
```

Ahora podemos verificar el estado de los pod y hacerlos pasar por un balanceador, en este caso, utilizaremos Haproxy, con la configuración indicada por el formador:

Vamos a pasar a través de haproxy que esta instalado en el servidor master,(192.168.1.250), que es donde esta nuestra red de empresa, el balanceo para llegar a estos pods, a través de una URL
aplicaciones.miempresa.com:

```
[root@master ~]# kubectl describe service kubernetes-bootcamp
```

```
Name:           kubernetes-bootcamp
Namespace:     default
```

```

Labels:          run=kubernetes-bootcamp
Selector:        run=kubernetes-bootcamp
Type:            NodePort
IP:              10.254.215.176
Port:            <unset> 8080/TCP
NodePort:        <unset> 31503/TCP
Endpoints:    172.17.10.5:8080,172.17.22.4:8080

```

[root@master ~]# vi /etc/haproxy/haproxy.cfg

frontend http-in

```

bind 192.168.1.250:80
acl is_site1 hdr_end(host) -i aplicaciones.miempresa.com
acl is_site2 hdr_end(host) -i kubia.miempresa.local
acl is_site3 hdr_end(host) -i bd.miempresa.local

```

use_backend site1 if is_site1

```

use_backend site2 if is_site2
use_backend site3 if is_site3

```

backend site1

```

balance roundrobin
option httpclose
option forwardfor
server s2 172.17.10.5:8080 maxconn 32
server s3 172.17.22.4:8080 maxconn 32

```

[root@master ~]# systemctl start haproxy.service

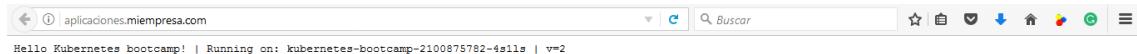
[root@master ~]# systemctl status haproxy.service

Resolvemos aplicaciones.miempresa.com, a través del fichero hosts de nuestro Windows:

C:\Windows\System32\drivers\etc

192.168.1.250 aplicaciones.miempresa.com

<http://aplicaciones.miempresa.com/>



En este laboratorio estoy utilizando el navegador Firefox, ojo con la cache de los navegadores al comprobar el balanceo de las peticiones.

Laboratorio kubernetes Labels

En este laboratorio veremos como trabajar con labels.

Las labels son pares de valores clave que se pueden adjuntar a los objetos.

En nuestros laboratorios anteriores, ya hemos estado usando etiquetas para etiquetar pods agregamos etiquetas, por ejemplo, "app: helloworld".

Las etiquetas no son únicas y se pueden agregar varias etiquetas a un objeto.

Una vez que las etiquetas se adjuntan a un objeto, puede utilizar filtros para limitar los resultados, esto se llama los selectores de etiquetas. Usando los selectores de etiquetas, puede usar expresiones coincidentes para hacer coincidir las etiquetas.

También puede utilizar etiquetas para etiquetar nodos. Una vez que los nodos están etiquetados, puede usar el selector de etiquetas para permitir que los pods solo se ejecuten en nodos específicos.

Se requieren dos pasos para ejecutar un pod en un conjunto específico de nodos.

Primero etiqueta el nodo, luego agrega nodeSelector a la confirmación de su pod.

Puede ser para un solo pod o una plantilla de pod dentro de, por ejemplo, un "deploy".

Labels

- Labels are key/value pairs that can be attached to objects
 - Labels are like **tags** in AWS or other cloud providers, used to tag resources
 - You can **label** your **objects**, for instance your pod, following an organizational structure
 - **Key:** environment - **Value:** dev / staging / qa / prod
 - **Key:** department - **Value:** engineering / finance / marketing
 - In our previous examples I already have been using labels to tag pods:

```
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
```

Labels

- Labels are **not unique** and **multiple labels** can be added to one object
- Once labels are attached to an object, you can use filters to narrow down results
 - This is called **Label Selectors**
- Using Label Selectors, you can use **matching expressions** to match labels
 - For instance, a particular pod can only run on a node labeled with “environment” equals “development”
 - More complex matching: “environment” in “development” or “qa”

Node Labels

- You can also use labels to tag **nodes**
- Once nodes are tagged, you can use **label selectors** to let pods only run on **specific nodes**
- There are **2 steps** required to run a pod on a specific set of nodes:
 - First you **tag** the node
 - Then you add a **nodeSelector** to your pod configuration

Node Labels

- First step, add a label or multiple labels to your nodes:

```
$ kubectl label nodes node1 hardware=high-spec
$ kubectl label nodes node2 hardware=low-spec
```

- Secondly, add a pod that uses those labels:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
    - name: k8s-demo
      image: wardviaene/k8s-demo
      ports:
        - containerPort: 3000
  nodeSelector:
    hardware: high-spec
```

Comenzamos el laboratorio obteniendo nuestro nodos y eliminaremos todos los laboratorios anteriores:

```
# kubectl get nodes

NAME      STATUS ROLES AGE VERSION
docker    Ready  master 18d v1.13.2
docker2   Ready  <none> 18d v1.13.2
orion3.curso.local Ready  <none> 17d v1.13.2
```

Ahora visualizamos el fichero yml, con el que vamos a realizar este laboratorio, no fijamos en **nodeSelector:hardware: high-spec**

affinity:

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
      - matchExpressions:
          - key: env
            operator: In
            values:
              - dev
  preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      preference:
        matchExpressions:
          - key: team
            operator: In
            values:
              - engineering-project1
```

Obtenemos todas las etiquetas de nuestros nodos:

```
# kubectl get nodes --show-labels
```

Desplegamos nuestro archivo yml:

```
# kubectl create -f /kubernetes-curso/deployment/helloworld-nodeselector.yml
# kubectl get deployments
```

Ahora podemos observar que los pods, no se crearan, si describo un pod, vere la label Node-Selectors: hardware=high-spec

Y como no están etiquetados los nodos con este node-selector, no despliegan los pods:

```
# kubectl describe pod helloworld-deployment-794c748d5b-njvkr
```

Etiquetamos con un hardware=high-spec al nodo docker2:

```
# kubectl label nodes docker2 hardware=high-spec
```

```
# kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE
READINESS GATES							
helloworld-deployment-794c748d5b-njvkr	1/1	Running	0	8m11s	10.44.0.31	docker2	<none>
	<none>						
helloworld-deployment-794c748d5b-srrjh	1/1	Running	0	8m11s	10.44.0.29	docker2	<none>
	<none>						
helloworld-deployment-794c748d5b-xnhjf	1/1	Running	0	8m11s	10.44.0.30	docker2	<none>
	<none>						

Ahora comprobamos las labels de nuestros nodos:

```
# kubectl get nodes --show-labels
```

Si ahora describimos un pod veremos el warning, de que no teníamos el node selector etiquetado correctamente.

```
# kubectl describe pod helloworld-deployment-794c748d5b-njvkr
```

Lab 2 Labels

C:\kubernetes-vagrant-cluster\k8-for-devs-master\labels

Las Labels son un mecanismo de consulta y filtrado muy potente que nos ofrece Kubernetes, pero con el que hay que tener especial atención porque es una fuente común de errores en la configuración de mis aplicaciones.

Los Labels son pares clave/valor que podemos asociar a cualquier objeto de Kubernetes.

Por ejemplo, podemos añadir la clave `environment` en todos mis objetos, y darle el valor `dev`, `staging` o `prod` según el entorno en el que estemos ejecutando.

Esto nos va a permitir hacer consultas filtrando por el valor de estos labels, y así refinar los resultados que recibo.

Pero muy importante, las Labels sirven para muchas más cosas. Ya hemos visto como usarlas para mapear los Pods a los que afecta un Replica Controller, y también para decidir entre qué Pods hace balanceo de carga un Service. Otro uso común es para seleccionar los nodos en los que un Pod puede ser ejecutado.

Otro tipo de labels son de tipo node Selector, para que etiquete mis nodos y despliegue los objetos de kubernetes en estos nodos.

Filtrando pods, por label:

```
~$ kubectl get pods -l app=nginx
```

Para el laboratorio tenemos dos archivos `labels1.yaml` (app: `nginx1`) y `labels2.yaml`, que tiene de label (app: `nginx2`), para todos los objetos creados en el Deployment:

Nos posicionamos en el directorio donde tenemos los yamel y los desplegamos:

```
$ cd /vagrant/k8-for-devs-master/labels/
$ vagrant@master:/vagrant/k8-for-devs-master/labels$ kubectl apply -f .
deployment.apps/labels1 created
deployment.apps/labels2 created
```

Ahora podemos ver que tenemos los dos deployments, cada uno con tres pods:

```
$kubectl get all
```

Ahora podemos filtrar todos los objetos que tengan la label app=nginx1 y app=nginx2, ojo este comando solo no dará los objetos del namespace default:

```
$ kubectl get all -l app=nginx1
```

```
$ kubectl get all -l app=nginx2
```

Ahora podemos eliminar el laboratorio, nos posicionamos en el directorio donde tenemos los yaml y ejecutamos:

```
$ cd /vagrant/k8-for-devs-master/labels
```

```
$ kubectl delete -f .
```

```
deployment.apps "labels1" deleted
```

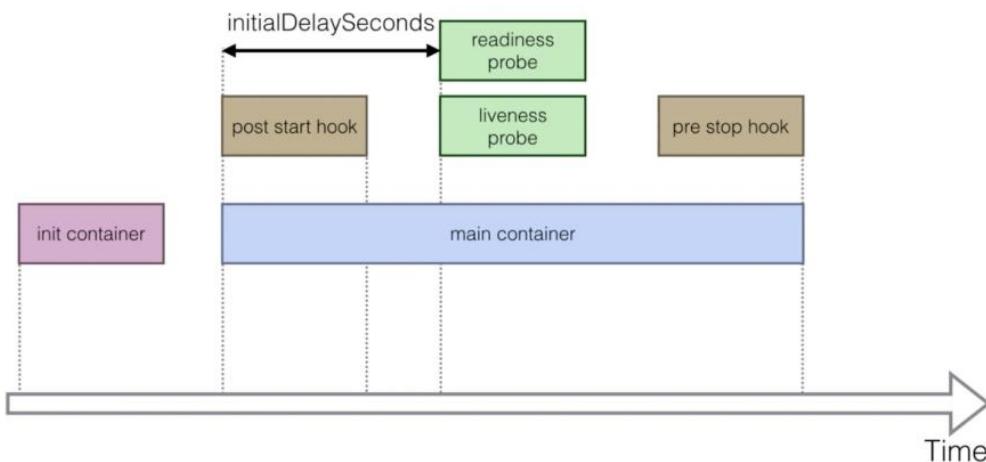
```
deployment.apps "labels2" deleted
```

Pod Life Cycle

<https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy>

Un Pod se puede encontrar en uno de estos estados:

- **Running:** al menos uno de sus contenedores que están en el pod está siendo ejecutado, es decir si tenemos un pod con varios contenedores pero solo uno esta corriendo aunque el resto no el pod estará en estado running.
- **Pending:** significa que estamos descargando la imagen de alguno de los contenedores, o que no hay recursos disponibles para ejecutar el Pod.
- **Succeeded:** significa que todos los contenedores del Pod acabaron su ejecución correctamente (exit code igual a cero).
- **Failed:** significa que al menos uno de los contenedores del Pod falló con error (exit code distinto de cero).
- **Unknown:** significa que no se puede acceder al estado del Pod, probablemente por problemas de red.



Además, es importante conocer el orden de ejecución de los contenedores de un Pod, y cuando se activan los *hooks* y los healchecks. Lo primero es que podemos definir un conjunto de `init containers`, que se ejecutan de manera secuencial cuando un Pod arranca. Cuando termina el último de los `init containers`, pasan a ejecutarse en paralelo todos los `main containers`, los `main containers` son como los que hemos estado viendo como por ejemplo con nginx, es decir el contenedor esta corriendo el servicio de nginx.

Imaginemos que necesitamos primero migrar la base de datos, pues esto lo ponemos como un init container y cuando termina lanza mi aplicación web como main container que accede a la base de datos, **este concepto es muy importante**, ya que muchas veces necesitamos hacer inicializaciones antes de ejecutar nuestro contenedores principales

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-initialization/>

Cada uno de estos contenedores puede definir un hook de post start hook y de pre stop hook.

Ambos son scripts que se ejecutan en el contexto del contenedor, uno al arrancar el contenedor, otro cuando recibe la señal de que pare su ejecución.

Los ReadinessProbe y los LivenessProbe se ejecutan contra los main containers, pero se puede configurar una cantidad de initialDelaySeconds para darle tiempo al contenedor de arrancar correctamente y no entrar en un bucle infinito donde Kubernetes mata el Pod antes de que al Pod le dé tiempo de arrancar correctamente.

Lab Configurar la inicialización del pod

Crear un pod que tenga un contenedor de inicio

En este labaratorio, creará un Pod que tiene un contenedor de aplicaciones y un Contenedor Init. El contenedor init se ejecuta hasta su finalización antes de que se inicie el contenedor de la aplicación.

Aquí está el archivo de configuración para el Pod:

En el archivo de configuración, puede ver que el Pod tiene un Volumen que comparten el contenedor de inicio y el contenedor de la aplicación.

El **contenedor init** monta el volumen compartido en /work-dir, y el contenedor de la aplicación monta el volumen compartido en /usr/share/nginx/html. El contenedor init ejecuta el siguiente comando y luego termina:

```
wget -O /work-dir/index.html http://kubernetes.io
```

Observe que el contenedor init escribe el index.html archivo en el directorio raíz del servidor nginx.

Crea el Pod init-containers.yaml:

```

apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: workdir
          mountPath: /usr/share/nginx/html
    # These containers are run during pod initialization
    initContainers:
      - name: install
        image: busybox
        command:
          - wget
          - "-O"
          - "/work-dir/index.html"
          - http://kubernetes.io
        volumeMounts:
          - name: workdir
            mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
    - name: workdir
      emptyDir: {}

```

```
kubectl apply -f init-containers.yaml
```

Verifique que el contenedor nginx se esté ejecutando:

```
kubectl get pod init-demo
```

El resultado muestra que el contenedor nginx se está ejecutando:

NAME	READY	STATUS	RESTARTS	AGE
init-demo	1/1	Running	0	1m

Obtenga un shell en el contenedor nginx que se ejecuta en el Pod init-demo:

```
kubectl exec -it init-demo -- /bin/bash
```

En su shell, envíe una solicitud GET al servidor nginx:

```
root@nginx:~# apt-get update
root@nginx:~# apt-get install curl
root@nginx:~# curl localhost
```

El resultado muestra que nginx está sirviendo la página web que fue escrita por el contenedor init:

```
<!Doctype html>
<html id="home">

<head>
...
"url": "http://kubernetes.io/" }</script>
</head>
<body>
...
<p>Kubernetes is open source giving you the freedom to take
advantage ...</p>
...
```

Si queremos entrar en el pod de nuestro nginx y visualizar el index.html

```
$ kubectl exec -it init-demo -c nginx -- /bin/bash
root@init-demo:/# cat /usr/share/nginx/html /index.html
```

Laboratorio Repaso conceptos kubernetes

Conceptos básicos para este laboratorio

Kubernetes es una plataforma opensource para el automatizar el despliegue, escalado de las aplicaciones, así como las operaciones con los contenedores de aplicaciones:

- **Cluster:** Conjunto de másquinas físicas o virtuales y otros recursos utilizados por kubernetes.
- **Nodo:** Una máquina física o virtual ejecutándose en kubernetes donde *pods* pueden ser programados.
- **Pod:** Son la unidad más pequeña desplegable que puede ser creada, programada y manejada por kubernetes.
- **Replication Controller:** Se asegura de que el número especificado de réplicas del pod estén ejecutándose. Permite escalar de forma facil los sistemas y maneja la re-creación de un pod cuando ocurre un fallo.
- **Service:** Es una abstracción que define un conjunto de pods y la lógica para acceder a los mismos.

Operaciones básicas

A continuación se indican las operaciones básicas que se pueden realizar con Kubernetes como crear y eliminar pods, crear y eliminar replication controllers y gestionar estos con los servicios, exponiéndolos fuera del cluster.

Pods

Como hemos definido antes, **un pod es la unidad mínima que es manejada por kubernetes**. Este pod es un grupo de uno o más contenedores (normalmente de Docker), con almacenamiento compartido entre ellos y las opciones específicas de cada uno para ejecutarlos. Un modelo de pods específico de una aplicación contiene uno o más contenedores que normalmente irían en la misma máquina.

Varios contenedores que pertenezcan al mismo pod **son visibles unos de otros vía localhost**. Los contenedores que se encuentran en distintos pods no pueden comunicarse de esta manera.

En términos de Docker, un pod es un conjunto de contenedores de Docker con namespace y volúmenes compartidos.

Hay que tener en cuenta que los pods son **entidades efímeras**. En el ciclo de vida de un pod estos se crean y se les asigna un UID hasta que terminen o se borren. Si un nodo que contiene un pod es eliminado, todos los pods que contenía ese nodo se pierden. Este pod puede ser reemplazado en otro nodo, aunque el UID será diferente. Esto es

importante porque un pod no debería de tener información almacenada que pueda ser utilizada después por otro pod en caso de que a este le pasara algo.

Usos de un pod

Los pods pueden utilizarse para realizar escalado horizontal, aunque fomentan el trabajo con microservicios puestos en contenedores diferentes para crear un sistema distribuido mucho más robusto.

Creación de un pod

Los pods se pueden crear de dos maneras: directamente por línea de comandos o a través de un fichero de tipo YAM, **en este laboratorio se muestra de las dos formas nosotros lo realizaremos sobre ficheros YAM.**

(No realizar)

Para crearlo directamente por línea de comandos ejecutamos lo siguiente:

Creación de replication controller por línea de comandos:

```
[root@master ~]# kubectl run my-nginx --image=nginx --port=80
```

Kubectl es el programa que vamos a utilizar para interactuar con el api de kubernetes.

- El primer parámetro indica la acción, que sirve para arrancar un pod.
- Después el nombre que va a recibir, en este caso my-nginx.
- Luego la imagen a partir de la que se va a construir el pod (la imagens e llama nginx).
- Por último el puerto en el que escucha.

Creación de pod a través de un fichero YAML (Comenzamos laboratorio)

En nuestro servidor master, (/laboratorios-kubernetes/pods) están creados todos los archivos de este laboratorio, para nuestra práctica, podemos crear un directorio en esta ruta, para realizarla desde un principio:

```
[root@master /]# mkdir /laboratorios-kubernetes/pods/lab1
```

```
[root@master /]# vi /laboratorios-kubernetes/pods/lab1/nginx-pod.yaml
```

```
# Número de versión del api que se quiere utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: Pod
# Aquí van los datos propios del pod como el nombre y los labels que tiene asociados para seleccionarlo
metadata:
  name: my-nginx
  # Especificamos que el pod tenga un label con clave "app" y valor "nginx"
  labels:
    app: nginx
# Contiene la especificación del pod
spec:
  # Aquí se nombran los contenedores que forman parte de este pod. Todos estos contenedores serían
  # visibles por localhost
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
  # Aquí se define la política de restauración en caso de que el pod se detenga o deje de ejecutarse debido
  # a un fallo interno.
  restartPolicy: Always
```

Con este comando creamos el pod directamente, vemos como ahora no se crea un replication controller, solo se creara cuando ejecutamos kubectl run: .

```
[root@master lab1]# kubectl create -f nginx-pod.yaml
pod "my-nginx" created
```

```
[root@master lab1]# kubectl get pod,rc,svc
NAME                  READY   STATUS    RESTARTS   AGE
po/busybox            1/1     Running   25        2d
po/kubernetes-bootcamp-2100875782-0r7n0 1/1     Running   1         1d
po/kubernetes-bootcamp-2100875782-4s1ls 1/1     Running   1         1d
po/my-nginx           1/1     Running   0         2m
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/kubernetes	10.254.0.1	<none>	443/TCP	2d

```
svc/kubernetes-bootcamp 10.254.215.176 <nodes> 8080:31503/TCP 1d
```

Replication Controllers

Un **replication controller** se asegura de que grupo de uno o más pods esté siempre **disponible**. Si hay muchos pods, eliminará algunos. Si hay pocos, creará nuevos. Por este motivo, se recomienda siempre crear un replication controller aunque solo tengas un único pod (este es el motivo por el cual cuando creamos un pod por comando automáticamente se crea un replication controller para el pod que acabamos de crear). Un replication controller es al fin y al cabo un supervisor de un grupo de uno o más pods a través de un conjunto de nodos.

Creación de un replication controller

A continuación crearemos un replication controller llamado **nginx-rc.yaml** en el directorio **/laboratorios-kubernetes/pods/lab1** a partir de la siguiente plantilla, encargado de levantar un servidor nginx:

```
# Número de versión del api que se quiere utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: ReplicationController
# Datos propios del replication controller
metadata:
  # Nombre del Replication Controller
  name: my-nginx
# La especificación del estado deseado que queremos que tenga el pod.
spec:
  # Número de réplicas que queremos que se encargue de mantener el rc. (Esto creará un pod)
  replicas: 1
  # En esta propiedad se indican todos los pods que se va a encargar de gestionar este replication controller. En este caso, se va a encargar de todos los que tengan el valor "nginx" en el label "app"
  selector:
    app: nginx
  # Esta propiedad tiene exactamente el mismo esquema interno que un pod , excepto que como está anidado no necesita ni un "apiVersion" ni un "kind"
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Creamos el replication controller:

```
[root@master lab1]# kubectl create -f nginx-rc.yaml
replicationcontroller "my-nginx" created
```

```
[root@master lab1]# kubectl get rc
NAME      DESIRED   CURRENT   READY   AGE
my-nginx  1         1         1       9s
```

Trabajando con replication controllers

Una vez creado un rc te permite:

- **Escalarlo:** puedes escoger el número de réplicas que tiene un pod de forma dinámica.
- **Borrar el replication controller:** puedes borrar solo el replication controller o borrarlo junto a todos los pods de los que se encarga
- **Aislard al Pod del replication controller:** Los pods pueden no pertenecer a un replication controller cambiando los labels. El pod que ha sido removido de esta manera será reemplazado por un pod nuevo, que será creado por el replication controller.

Para añadir un pod que sea controlado por el replication controller ejecutamos:

```
# kubectl scale rc my-nginx --replicas=2
replicationcontroller "my-nginx" scaled
```

Tras listar el número de pods comprobaríamos como se ha añadido uno nuevo:

```
# kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
my-nginx      1/1     Running   0          10m
my-nginx-4jjn0 0/1     ContainerCreating   0          55s
```

Para borrar un replication controller ejecutaríamos el comando (**No realizar en este lab, comprobarlo al finalizar el laboratorio**):

```
[root@master lab1]# kubectl delete rc my-nginx
```

Si después listamos los pods comprobaremos que han desaparecido todos.

Services

Como ya sabemos, los pods son volátiles. Son creados y destruidos, de forma que no pueden recuperarse. De hecho, los replication controllers son los encargados de manejar su ciclo de vida, y de definir sus políticas de restauración. Como decíamos anteriormente, cada pod tiene su propia dirección IP (que podría incluso no ser constante en el mismo pod a lo largo del tiempo). Esto nos supone un problema en caso de que un pod necesite comunicarse con otro pod. ¿Qué manera tienen de comunicarse ambos, si las ip de cada pod son variables, o si uno de los dos se cae y lo sustituye otro? De esto justo se encargan los services.

Un service es una **abstracción que define un grupo lógico de pods y una política de acceso a los mismos**. Los pods apuntan a un servicio normalmente por la propiedad label. Pongamos como ejemplo nuestro caso anterior, dónde tenemos un replication controller encargado de ejecutar un pod con un contenedor nginx. Si algo causara la destrucción de este pod, el replication controller crearía uno nuevo con una ip diferente, de forma que el resto de la infraestructura que dependiera de ese pod por esa ip fija dejaría de funcionar. **El servicio lo que hace es que ese pod siempre sea accesible de la misma manera**, de forma que aunque el pod se destruya o se modifique siempre sea accesible por la abstracción. A continuación crearemos un servicio y también comprobaremos como podemos exponerlo desde fuera del cluster.

Acceder a un servicio desde fuera del cluster

Por último vamos a acceder al servicio desde fuera de cluster, de modo que accederemos al servidor nginx de nuestra máquina virtual desde el navegador. Para ello necesitamos modificar el servicio que estamos utilizando al que le añadimos la propiedad **type: NodePort**, que lo que hace es exponer el servicio en cada nodo del cluster de forma que serás capaz de contactar con el servicio desde cualquier ip de los nodos. Nuestro servicio quedaría de la siguiente manera:

A continuación, vamos a crear un service:

```
[root@master lab1]# vi /laboratorios-kubernetes/pods/lab1/nginx-svc.yaml
```

```
# Número de versión del api que se quiere utilizar
apiVersion: v1
# Tipo de fichero que se va a crear.
kind: Service
# Aquí van los datos propios del pod como el nombre y los labels que tiene asociados para seleccionarlo
metadata:
  name: my-nginx-service
# Contiene la especificación del pod
spec:
  type: NodePort
  # En esta propiedad se indican todos los pods que apuntan a este servicio. En este caso, se va a
  # encargar de todos los que tengan el valor "nginx" en el label "app"
  selector:
    app: nginx
  ports:
    # Indica el puerto en el que se debería de servir este servicio
    - port: 80
```

Creamos el servicio y verificamos su funcionamiento:

```
# kubectl create -f nginx-svc.yaml
service "my-nginx-service" created
```

```
# kubectl get service
NAME      CLUSTER-IP   EXTERNAL-IP PORT(S)      AGE
kubernetes  10.254.0.1 <none>     443/TCP      2d
kubernetes-bootcamp 10.254.215.176 <nodes>    8080:31503/TCP 1d
my-nginx-service  10.254.109.132 <nodes>    80:30992/TCP 14s
```

```
# kubectl describe service my-nginx-service
Name:           my-nginx-service
Namespace:      default
Labels:         <none>
Selector:       app=nginx
Type:          NodePort
IP:            10.254.109.132
Port:          <unset> 80/TCP
NodePort:       <unset> 30992/TCP
Endpoints:     172.17.63.6:80,172.17.93.5:80
```

Ahora podríamos hacer referirnos a los pods, bien a través del puerto expuesto en cada minion que conforma el cluster, que es el valor del campo NodePort:

```
# curl 192.168.1.251:30992
# curl 192.168.1.252:30992
```

Tambien los podemos hacer pasar a través de un balanceador con HAProxy con los Endpoints, el formador realizará la demostración de la configuración correcta.

Para finalizar el laboratorio procedemos a borrar todos los pods,rc y services de este laboratorio.

```
# kubectl get pod,rc -l app=nginx
```

```
# kubectl get pods,rc --selector=app="nginx"
```

NAME	READY	STATUS	RESTARTS	AGE
po/my-nginx	1/1	Running	0	39m
po/my-nginx-4jjn0	1/1	Running	0	29m

NAME	DESIRED	CURRENT	READY	AGE
------	---------	---------	-------	-----

rc/my-nginx	2	2	2	32m
-------------	---	---	---	-----

```
# kubectl get svc
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-nginx-service	10.254.109.132	<nodes>	80:30992/TCP	22m

Healthchecks

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/>

C:\kubernetes-vagrant-cluster\k8-for-devs-master\healthchecks

Los *Healthchecks* son un mecanismo fundamental para cargas productivas. Es el principal mecanismo por el cual Kubernetes va a saber si nuestros Pods están funcionando correctamente o no.

Hay dos tipos de healthchecks

Por comandos:

Podemos configurar la ejecución de comando periódico que se ejecuta en contexto de uno de los contenedores de mi pod, u otro muy común es que podemos ejecutar llamadas HTTP *endpoint* que nos responda a una url y un path que le digamos y dependiendo de la respuesta nos dirá si nuestro contenedor está funcionando correctamente.

Por último, Kubernetes distingue entre dos tipos de healthchecks: ReadinessProbe y LivenessProbe.

LivenessProbe indica si el contenedor está funcionando incorrectamente y tiene que ser recreado.

ReadinessProbe indica si está listo para recibir tráfico, por ejemplo, imaginemos que tenemos un microservicio con una cache de redis, si la cache de redis no está lista aunque yo como contenedor estoy listo si no tengo la cache de redis no podré servir peticiones, para este tipo de situaciones utilizamos ReadinessProbe.

Nótese que no significan lo mismo, un contenedor podría estar pasando el LivenessProbe, pero no pasar el ReadinessProbe porque, por ejemplo, necesita acceder a una base de datos que en este momento no está disponible.

Resumen:

- **Readiness:** ¿El contenedor está listo para recibir tráfico?
- **Liveness:** ¿El contenedor sigue vivo?

El flujo es el siguiente:

- Si **Readiness** falla
 - Kubernetes detiene el tráfico hacia el “pod” que falla de la aplicación
- Si **Liveness** falla
 - Kubernetes reinicia el pod de la aplicación
- Si **Readiness** funciona
 - Kubernetes restablece el tráfico hacia el pod de la aplicación nuevamente

Realizamos el despliegue:

```
$ kubectl apply -f healcheck.yaml
```

```
$ kubectl get all
```

Visualizamos el deployment en formato yaml

```
$ kubectl get deployment.apps/deployment -oyaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: "2019-08-12T17:27:45Z"
  generation: 1
  labels:
    app: nginx
    name: deployment
    namespace: default
  resourceVersion: "442675"
  selfLink: /apis/apps/v1/namespaces/default/deployments/deployment
  uid: 7f7c73e0-bd26-11e9-8186-080027c5bc64
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          imagePullPolicy: Always
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /
              port: 80
          #Vemos que la request es http
            scheme: HTTP
            initialDelaySeconds: 10
            periodSeconds: 10
        #Cuantos seguidos helchehs tenemos que tener para considerar que el servicio este healthy
        # Un Pod solo se considera healthy si todos los contenedores que lo componen están funcionando correctamente.
          successThreshold: 1
          timeoutSeconds: 10
```

Si ahora realizamos un describe del deployment, nos dará información de los **Healthchecks, buscaremos Liveness** :

```
$ kubectl describe deployment.apps/deployment
```

```
Liveness: http-get http://:80/ delay=10s timeout=10s period=10s
#success=1 #failure=3
```

Lo que esta haciendo un get al 80, cada 10 segundos, con un delay inicial de 10 segundos para que se inicie la aplicación dentro del pod, y que el número de situaciones es 1 para conseguir el **healthy** y que para considerarlo fallido tendrían que ser tres seguidos

Un Pod solo se considera healthy si todos los contenedores que lo componen están funcionando correctamente.

Con esta configuración, si por cualquier situación uno de estos contenedores es incapaz de devolver, esta petición tres veces seguidas, pues kubernetes lo detectará y nos restablecerá la salud en la aplicación, matando ese contenedor y creando uno nuevo que si que será capaz de atender las peticiones.

Laboratorio 2 Health Checks

En este laboratorio veremos como trabajar en nuestros pods con chequeos de salud, para nuestras aplicaciones en producción.

Cuando ponga su aplicación en producción, definitivamente querrá configurar sus controles de estado. Si la aplicación no funciona correctamente, el pod en el contenedor aún puede estar ejecutándose, pero es posible que la aplicación ya no funcione.

La aplicación da un error debido a algo interno que salió mal, para detectar y resolver problemas con su aplicación, puede ejecutar controles de estado.

Puede ejecutar dos tipos diferentes de controles de salud:

- El primero es ejecutar un comando en el contenedor periódicamente.
 - Solo se va a ejecutar un comando dentro del contenedor para verificar si sus contenedores siguen funcionando.
- El otro y el que se utiliza la mayor parte del tiempo son las verificaciones periódicas de una URL que utiliza HTTP.

Su aplicación probablemente expondrá una interfaz utilizando controles periódicos en esa interfaz.

Puede comprobar si su aplicación sigue siendo saludable.

La aplicación de producción típica detrás de un Load Balancer siempre debe tener controles de estado implementados de alguna manera para asegurar la disponibilidad y flexibilidad de la aplicación.

Así es como se ve un chequeo en nuestro contenedor de ejemplo.

Health checks

- This is how a health check looks like on our example container:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
    - name: k8s-demo
      image: wardviaene/k8s-demo
      ports:
        - containerPort: 3000
      livenessProbe:
        httpGet:
          path: /
          port: 3000
        initialDelaySeconds: 15
        timeoutSeconds: 30
```

Aquí nuevamente tengo el pod, el pod de ejemplo, y aquí está en negrita el healt checks.

Es un "livenessProbe" que usa un "httpget" en una ruta que significa solo el sitio web principal.

En la aplicación de producción probablemente tendrás algo como "/salud".

Pero aquí no tengo una página de salud separada, así que solo uso "/".

El puerto será el puerto 3000, que es el puerto de contenedor, o si le da un nombre a su puerto de contenedor, entonces puede usar ese nombre.

Hay un retraso inicial en segundos, por lo que "livenessProbe" va a esperar 15 segundos y hay un tiempo de espera de 30 segundos.

Para comprender el funcionamiento comenzamos el laboratorio:

```
# cat /kubernetes-curso/deployment/helloworld-healthcheck.yml
# kubectl create -f /kubernetes-curso/deployment/helloworld-healthcheck.yml
# kubectl get pod -o wide
```

Ahora describimos un pod y nos fijamos en la línea Liveness que hay un retraso de 15s un tiempo de espera de 30s con un retraso de verificación de 10s para que el pod se considere que está en buen estado, cuando hay 1 éxito y se considera erroneo cuando tenemos 3 fracasos:

```
Liveness: http-get http://:nodejs-port/ delay=15s timeout=30s period=10s #success=1  
#failure=3
```

```
# kubectl describe pod helloworld-deployment-5d5b64f8b9-zdtm9
```

Cuando hay un error kubernetes termina el pod y crea un nuevo pod y esto es muy importante porque si el pod puede recibir un error hay otros que están funcionando correctamente, podemos estar enviando a los usuarios información errónea, entonces lo que queremos es que no se rediriga el tráfico a los pod no saludables, esta implementación sería interesante para implementar aplicaciones en producción.

Podemos modificar en caliente en el deployment las características del livenessProbe:

```
# kubectl edit deployment helloworld-deployment
```

Laboratorio1 Secrets en kubernetes

<https://kubernetes.io/docs/concepts/configuration/secret/>

En este laboratorio veremos como trabaja con "Secretos". Los secretos proporcionan una forma en Kubernetes de distribuir credenciales, claves, contraseñas o datos secretos a los pods, incluso Kubernetes utiliza este mecanismo de Secretos para proporcionar las credenciales para acceder a la API interna.

Secrets

- Secrets provides a way in Kubernetes to distribute **credentials, keys, passwords** or "**secret**" **data** to the pods
- Kubernetes itself uses this Secrets mechanism to provide the credentials to access the internal API
- You can also use the **same mechanism** to provide secrets to your application
- Secrets is one way to provide secrets, native to Kubernetes
 - There are still **other ways** your container can get its secrets if you don't want to use Secrets (e.g. using an **external vault services** in your app)

Eso significa que los pods tienen acceso a secretos y esos secretos deben distribuirse usando este mecanismo de Secretos.

También puede utilizar el mismo mecanismo para proporcionar secretos a su aplicación.

Si tiene una aplicación que necesita credenciales, puede usar este mismo mecanismo.

Los secretos podrían ser, por ejemplo, una base de datos que busque contraseñas o cualquier cosa que deba mantenerse en secreto.

Los secretos se pueden usar de las siguientes maneras en que puede usar los secretos como variables de entorno entonces, en su aplicación, solo tendrá que leer las variables de entorno y ahí es donde la aplicación encontrará los secretos.

Secrets

- Secrets can be used in the following ways:
 - Use secrets as **environment variables**
 - Use secrets **as a file** in a pod
 - This setup uses **volumes** to be mounted in a container
 - In this volume you have **files**
 - Can be used for instance for **dotenv** files or your app can just read this file
 - Use an **external image** to pull secrets (from a **private image registry**)

También puede utilizar secretos como un archivo en un pod.

Esta configuración utiliza volúmenes para ser montados en un contenedor. En este volumen tienes archivos entonces, en tu contenedor vas a tener un volumen y significa que es solo un directorio al que puedes acceder, que contiene todos los secretos.

Por lo tanto, en su aplicación, simplemente puede escribir un código que diga que debe ir y buscar en el directorio específico los secretos que necesita, por ejemplo, para conectarse a una base de datos.

También puede utilizar una imagen externa para extraer los secretos. Obviamente, desde un registro de imágenes privadas, no puede poner su secreto en un registro de imágenes públicas.

En ese caso, significaría que tendrá una segunda imagen acoplable y que esa imagen se extraerá y su aplicación leerá los datos que contienen los secretos de esa imagen.

Secrets

- To generate secrets using files:

```
$ echo -n "root" > ./username.txt
$ echo -n "password" > ./password.txt
$ kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
secret "db-user-pass" created
$
```

- A secret can also be an SSH key or an SSL certificate

```
→ $ kubectl create secret generic ssl-certificate --from-file=ssh-privatekey=~/.ssh/id_rsa --ssl-cert=ssl-cert=mysslcert.crt
```

Secrets

- To generate secrets using yaml definitions:

secrets-db-secret.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  password: cm9vdA==
  username: cGFzc3dvcmQ=
```

```
$ echo -n "root" | base64
cm9vdA==
$ echo -n "password" | base64
cGFzc3dvcmQ=
```

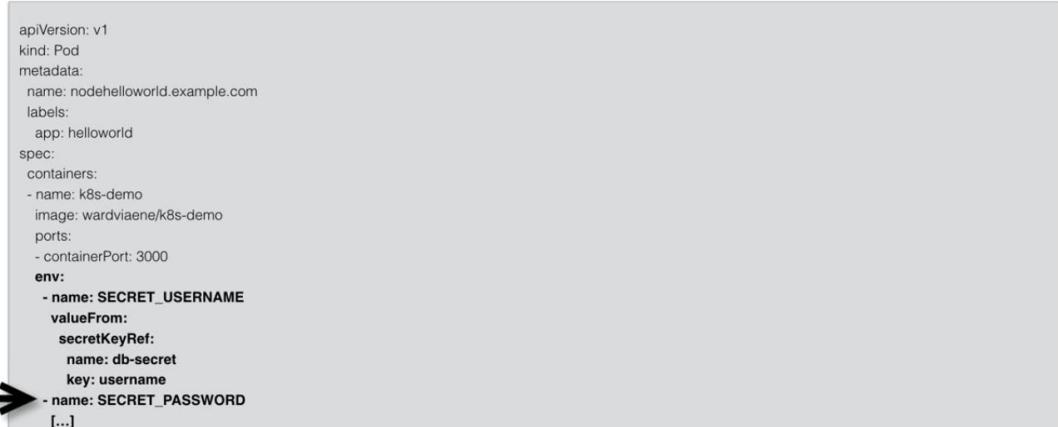
- After creating the yml file, you can use kubectl create:

```
$ kubectl create -f secrets-db-secret.yml
secret "db-secret" created
$
```

Using secrets

- You can create a pod that exposes the secrets as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
labels:
  app: helloworld
spec:
  containers:
    - name: k8s-demo
      image: wardviaene/k8s-demo
    ports:
      - containerPort: 3000
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-secret
            key: username
      - name: SECRET_PASSWORD
        [...]
```



Using secrets

- Alternatively, you can provide the secrets in a file:

```
apiVersion: v1
kind: Pod
metadata:
  name: nodehelloworld.example.com
  labels:
    app: helloworld
spec:
  containers:
    - name: k8s-demo
      image: wardviaene/k8s-demo
      ports:
        - containerPort: 3000
      volumeMounts:
        - name: credvolume
          mountPath: /etc/creds
          readOnly: true
  volumes:
    - name: credvolume
      secret:
        secretName: db-secrets
```

The secrets will be stored in:
 /etc/creds/db-secrets/username
 /etc/creds/db-secrets/password

Los Secrets nos permiten guardar información sensible que será codificada. Por ejemplo, nos permite guardar contraseñas, claves ssh, ...

Al crear un **Secret los valores se pueden indicar desde un directorio, un fichero o un literal, en esta primera parte del laboratorio utilizaremos secret a través de un **literal**:**

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

Los Secrets no son seguros, no están encriptados, ahora sacamos nuestro secreto en formato yaml y vemos como podemos desencriptar el hash del password:

```
#kubectl get secret mariadb -o yaml
apiVersion: v1
data:
  password: cm9vdA==
kind: Secret
metadata:
  creationTimestamp: 2018-05-23T18:22:27Z
  name: mariadb
  namespace: default
  resourceVersion: "162405"
  selfLink: /api/v1/namespaces/default/secrets/mariadb
  uid: 3fa5elad-5eb6-11e8-ab66-fa163e99cb75
type: Opaque
```

```
#echo 'cm9vdA==' | base64 --decode
root
```

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

mariadb-deployment-secret.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mariadb-deploy-secret
  labels:
    app: mariadb
    type: database
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb
        type: database
    spec:
      containers:
        - name: mariadb
          image: mariadb
          ports:
            - containerPort: 3306
              name: db-port
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mariadb
                  key: password
```

Creamos el despliegue y probamos el acceso:

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

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

NAME	READY	STATUS	RESTARTS	AGE
mariadb-deploy-secret-f946dddf-dkkmlb	1/1	Running	0	15s

```
#kubectl exec -it mariadb-deploy-secret-f946dddf-dkkmlb -- mysql -u root -p
```

Ahora en este laboratorio, eliminaremos todos los elementos de kubernetes de laboratorios anteriores, echaremos un vistazo al archivo que contiene nuestros secrets:

```
# cat /kubernetes-curso/deployment/helloworld-secrets.yml
```

Si queremos codificar nuestros secretos podemos realizarlo:

```
# echo -n "000000" | base64
```

Desplegamos nuestros secretos en kubernetes:

```
[root@docker ~]# kubectl create -f /kubernetes-curso/deployment/helloworld-secrets.yml
```

Ahora veremos que tenemos un deployment que le pasamos los secretos, tendremos tres replicas, un contenedor de "volumeMount" y este es el volumen con el nombre "db-secrets", el que acabo de crear en Kubernetes y este será montado en "/etc/creds"

```
[root@docker ~]# cat /kubernetes-curso/deployment/helloworld-secrets-volumes.yml
```

```
# kubectl create -f /kubernetes-curso/deployment/helloworld-secrets-volumes.yml
```

```
# [root@docker ~]# kubectl get pods -o wide
```

Describimos un pod y observaremos el **podMount /etc/creds from cred-volume (ro):**

```
[root@docker ~]# kubectl describe pod helloworld-deployment-78457f7dfc-zn2bf
```

Ahora podemos conectarnos al pod:

```
# kubectl exec -ti helloworld-deployment-78457f7dfc-zn2bf -- /bin/bash
```

Nuestras credenciales estaran disponibles en esta ruta dentro del contenedor, una aplicacion solo tendria que leer estos archivos para obtener las credenciales, en /etc/creds:

```
#root@helloworld-deployment-78457f7dfc-zn2bf:/app# cat /etc/creds/password
```

```
#root@helloworld-deployment-78457f7dfc-zn2bf:/app# cat /etc/creds/username
```

Laboratorio Crear e interrogar secretos desde la línea de comandos con kubectl.

Objetivos de aprendizaje

Desde el nodo maestro,

Ejemplo: nombre de usuario y contraseña

Primero, almacene los datos secretos en un archivo. En este ejemplo, colocaremos un nombre de usuario y contraseña en dos archivos codificados con base64.

```
echo -n 'admin' > username.txt
echo -n 'Linux01' > password.txt
```

El kubectl puede empaquetar estos archivos en un objeto 'Secreto' en el servidor API.

```
kubectl create secret generic ks-user-pass --from-file=username.txt --
from-file=password.txt
```

Puede buscar secretos con get y describe de la siguiente manera:

```
kubectl get secrets
kubectl describe secrets/ks-user-pass
```

Los secretos están enmascarados por defecto. Si necesita obtener el valor de un secreto almacenado, puede usar los siguientes comandos:

```
kubectl get secret ks-user-pass -o yaml
```

Luego decodifique los valores con:

```
echo '[stored value here]' | base64 -d
```

Crea secretos usando YAML.

También puede crear secretos con un archivo YAML. Lo siguiente es un ejemplo:

Ejemplo YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: ks-lab-secret
type: Opaque
data:
  username: "admin"
  password: "Linux01"
```

Los campos adicionales también se pueden almacenar en un archivo YAML.

Usa un editor para crear `ks-secret-config.yaml`.

```
#vi ks-secret-config.yaml
apiVersion: v1
kind: Secret
metadata:
  name: ks-secret-config
type: Opaque
stringData:
  config.yaml: |-
    apiUrl: https://ks.api.com/api/v1
    username: admin
    password: Linux01
    branchid: branch21
```

Luego crea el secreto con:

```
kubectl create -f ks-secret-config.yaml
```

Puede mirar los campos obteniendo el secreto en YAML y luego pasando el campo config.yaml a través del decodificador.

```
kubectl get secret ks-secret-config -o yaml
echo '[stored value here]' | base64 -d
```

Pasar secretos a un pod a través de un volumen montado

Los secretos se pueden pasar a los pods a través de volúmenes montados o mediante variables de entorno.

El siguiente es un ejemplo de cómo se pueden usar volumeMounts especificados en el archivo YAML de un pod:

```
vi ks-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  name: ks-pod
  namespace: default
spec:
  containers:
    - name: ks-pod
      image: busybox
      command:
        - sleep
        - "10000"
      volumeMounts:
        - name: ks-path
          mountPath: "/etc/ks-path"
          readOnly: true
  restartPolicy: Never
  volumes:
    - name: ks-path
      secret:
        secretName: ks-secret-config
        items:
          - key: config.yaml
            path: config.yaml
            mode: 400
```

Luego crea el pod.

```
kubectl create -f ks-pod.yaml
```

Después de crear el pod, verifique que esté listo.

```
kubectl get pods
```

Una vez que el pod esta listo:

```
kubectl exec -it ks-pod -- sh
```

Una vez que esté dentro del contenedor busybox, echemos un vistazo a nuestros secretos.

```
cd /etc/ks-path  
ls -l  
cat config.yaml
```

Pase secretos a un pod a través de una variable de entorno.

Ahora hagamos un ejemplo donde podemos obtener estos secretos a través de una variable de entorno.

```
vi ks-pod-env.yaml
apiVersion: v1
kind: Pod
metadata:
  name: ks-pod-env
spec:
  containers:
  - name: ks-pod-env
    image: busybox
    command:
    - sleep
    - "10000"
  env:
  - name: SECRET_CONFIG
    valueFrom:
      secretKeyRef:
        name: ks-secret-config
        key: config.yaml
  restartPolicy: Never
```

Ahora vamos a crear el pod.

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

Vamos a echar un vistazo.

```
kubectl exec -it ks-pod-env -- sh
```

Y revisa nuestra variable.

```
echo $SECRET_CONFIG
```

Laboratorio 2 Secrets en kubernetes

En este laboratorio veremos cómo configurar un WordPress usando los secretos. Todavía no es un Wordpress completamente funcional, ya que no estaremos persistiendo datos.

Aún no sabemos cómo usar contenedores con estado, por eso siempre que pueda poner datos en este WordPress, no va a ser persistente, independientemente, es decir si se cae o reinicia el pod los datos no estarán persistidos.

Este "wordpress-secrets" tiene un secreto, llamado db-password, la contraseña de la base de datos, podemos cambiar la contraseña de la base de datos, como lo vimos en el laboratorio anterior.

```
[root@docker ~]# cat /kubernetes-curso/wordpress/wordpress-secrets.yml
```

Despelgamos el secreto en kubernetes:

```
[root@docker ~]# kubectl create -f /kubernetes-curso/wordpress/wordpress-secrets.yml
```

Ahora realizaremos un deployment, en el cual estamos desplegando un pod, con un contenedor de Wordpress y un contenedor de mysql, a través de las variables de entorno de estas imágenes pasamos el password del secreto db-password, a nuestros contenedores.

Tenemos un pod, dos contenedores y solo una réplica porque si lo incrementara a dos, se ejecutarán dos bases de datos y esta imagen de docker no esta preparada para cluster:

```
[root@docker ~]# cat /kubernetes-curso/wordpress/wordpress-single-deployment-no-volumes.yml
```

```
# kubectl create -f /kubernetes-curso/wordpress/wordpress-single-deployment-no-volumes.yml
```

```
# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
wordpress-deployment-6958b7b48f-zr2p6	2/2	Running	0	26s

```
# kubectl describe pod wordpress-deployment-6958b7b48f-zr2p6
```

Ahora para llegar a este WordPress, necesito desplegar un service, por lo tanto, también tengo un "servicio de wordpress" que apuntará a mi puerto HTTP para mi pod:

```
[root@docker ~]# cat /kubernetes-curso/wordpress/wordpress-service.yml
```

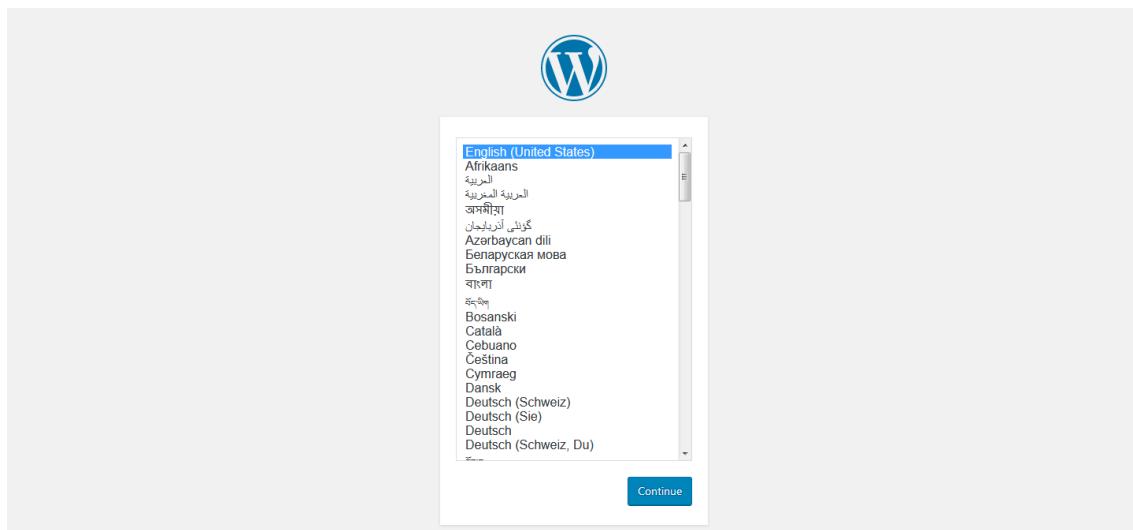
```
[root@docker ~]# kubectl create -f /kubernetes-course/wordpress/wordpress-service.yml
```

Ahora descibimos el servicio:

```
[root@docker ~]# kubectl describe service wordpress-service
```

Y entonces, debería poder acceder a nuestra wordpress.

<http://192.168.1.150:31001>



Si ahora tras realizar la instalacion eliminamos el deployment y lo volvemos a desplegar veremos como no tenemos persistidos los datos y tendremos que realizar de nuevo la instalacion de nuestro wordpress.

Lab 3 Secrets

<https://kubernetes.io/docs/concepts/configuration/secret/>

Un Secret es la manera que tenemos en Kubernetes de insertar secretos, como pueda ser un *password*, en el entorno de ejecución de un Pod de la manera más segura posible. La realidad es que a día de hoy, los secretos de Kubernetes no son del todo seguros, por lo que muchas empresas utilizan servicios externos como Vault.

Existen dos maneras de pasar un secreto a un Pod. Una opción es pasarlos como variables de entorno.

Otra opción más segura es pasarlos como un volumen que se monta en un path concreto dentro del contenedor, de tal manera que podemos eliminar el fichero con los secretos una vez que han sido leídos por la aplicación principal.

Vamos a ver una demo del uso de Secrets, para lo que nos vamos al directorio secrets desde el raíz del repo.

C:\kubernetes-vagrant-cluster\k8-for-devs-master\secrets

Recordar de eliminar los laboratorios anteriores:

En este caso vamos a crear el Secret desde un fichero. Creamos un fichero con el comando:

```
echo XXXXX > password
```

Ahora creamos el Secret desde ese fichero con el comando:

```
kubectl create secret generic secret --from-file=password
```

que nos crea un Secret con el nombre `secret` de tipo genérico. Podemos ver que lo ha creado con el comando:

```
kubectl get secrets
```

he incluso podemos ver su valor asociado (en base64) con el comando:

```
kubectl get secret secret -oyaml
```

En el fichero `secret.yaml` tenemos un ejemplo muy sencillo de un Deployment que consume un Secret.

En este fichero hemos definido un campo `env` dentro del contenedor `nginx`. Y en él tenemos una variable de entorno llamada `USERNAME` que toma el valor de `usuario1`, y una variable de entorno llamada `SECRET` que toma su valor del secreto que acabamos de crear.

Para lanzar el Deployment ejecutamos:

```
kubectl apply -f secret.yaml
```

Y si hacemos:

```
kubectl get all
```

veremos el Deployment creado y 3 pods creándose. Si hacemos `kubectl exec` a uno de los pods veremos que las variables de entorno han sido correctamente creadas, pero en YAML del Deployment no tenemos acceso al valor del secreto.

```
$ kubectl get pod
```

Entramos dentro del Pod:

```
$ kubectl exec -it secret-fcc867946-9mfxd -- bash
```

Ahora dentro del Pod, podemos ver las variables de entorno:

```
root@secret-fcc867946-9mfxd: # env |grep -i usuario && env |grep -i secret
USERNAME=usuario1
SECRET=XXXXXX
```

Por último, para eliminar el Deployment y sus pods asociados ejecutamos:

```
kubectl delete -f secret.yaml
```

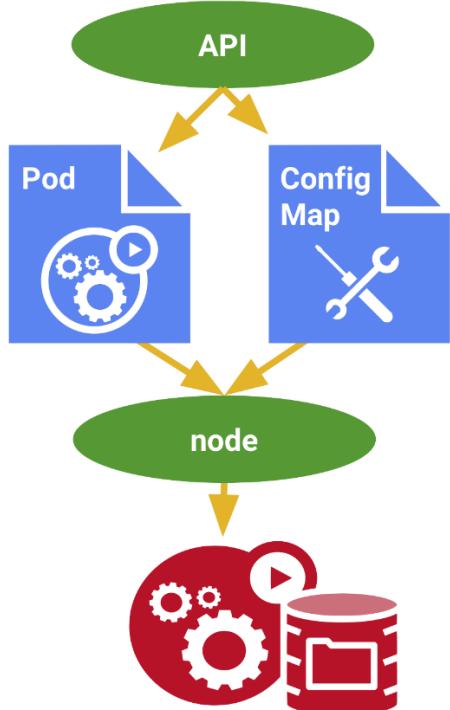
y el secreto lo eliminamos con:

```
kubectl delete secret/secret
```

Laboratorio ConfigMap kubernetes

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>

¿Qué es un ConfigMap?



ConfigMaps vincula los archivos de configuración, los argumentos de la línea de comandos, las variables de entorno, los números de puerto y otros artefactos de configuración a los contenedores y componentes del sistema de tus pods en entorno de ejecución.

ConfigMaps te permite separar tus configuraciones de tus pods y componentes, lo que ayuda a mantener tus cargas de trabajo portátiles, hace que sus configuraciones sean más fáciles de cambiar y administrar, y evita codificar los datos de configuración según las especificaciones del pod.

Los ConfigMaps son útiles para almacenar y compartir información de configuración *no confidencial* y sin cifrar. Para usar información sensible en tus clústeres, debes usar Secretos.

Cómo crear un ConfigMap

Crea un ConfigMap con el siguiente comando:

```
kubectl create configmap [NAME] [DATA]
```

[DATA] puede ser:

- Una ruta de acceso a un directorio que contiene uno o más archivos de configuración, indicada mediante el marcador `--from-file`.
- Pares clave-valor, cada uno especificado con marcadores `--from-literal`.

Para obtener más información sobre `kubectl create`, consulta la [documentación de referencia](#).

También puedes crear un ConfigMap cuando defines un [objeto ConfigMap](#) en un archivo de manifiesto YAML y luego implementas el objeto con `kubectl create -f [FILE]`.

Desde archivos

Para crear un ConfigMap a partir de uno o más archivos, usa `--from-file`. Especifica los archivos en cualquier texto sin formato, como `.properties`, `.txt`, o `.env`, siempre que los archivos contengan pares clave-valor.

Puedes pasar un solo archivo o varios archivos:

```
kubectl create configmap [NAME] --from-file [/PATH/TO/FILE.PROPERTIES]
--from-file [/PATH/TO/FILE2.PROPERTIES]
```

También puedes pasar un directorio que contenga varios archivos:

```
kubectl create configmap [NAME] --from-file [/PATH/TO/DIRECTORY]
```

Cuando creas un ConfigMap basado en archivos, la clave se establece de forma predeterminada en el nombre base del archivo y el valor predeterminado en el contenido del archivo. También puedes [especificar una clave alterna](#).

Para un ConfigMaps basado en directorios, cada archivo cuyo nombre base es una clave válida en el directorio está empaquetado en el ConfigMap. `kubectl` ignora los archivos no regulares, como enlaces simbólicos, dispositivos y barras verticales. Los subdirectorios también se ignoran; `kubectl create configmap` no se clasifica en subdirectorios.

ConfigMap

- Configuration parameters that are not secret, can be put in a **ConfigMap**
- The input is **again** key-value pairs
- The ConfigMap **key-value pairs** can then be read by the app using:
 - **Environment** variables
 - **Container commandline arguments** in the Pod configuration
 - Using **volumes**

ConfigMap

- A ConfigMap can also contain **full configuration** files
 - e.g. an webserver config file
- This file can then be **mounted** using volumes where the application expects its config file
- This way you can “**inject**” configuration settings into containers without changing the container itself

Para configurar nuestras aplicaciones que vamos a desplegar usamos variables de entorno, por ejemplo podemos ver las variables de entorno que podemos definir para configurar la imagen docker de [MariaDB](#).

Podemos definir un Deployment que defina un contenedor configurado por medio de variables de entorno, **mariadb-deployment.yaml**:

NO REALIZAR SOLO LECTURA

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mariadb-deployment
  labels:
    app: mariadb
    type: database
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb
        type: database
    spec:
      containers:
        - name: mariadb
          image: mariadb
          ports:
            - containerPort: 3306
              name: db-port
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: my-password
```

Y creamos el despliegue:

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

O directamente ejecutando:

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

Veamos el pod creado:

```
#kubectl get pods -l app=mariadb
NAME                      READY   STATUS    RESTARTS   AGE
mariadb-deployment-fc75f956-f5zlt   1/1     Running   0          15s
```

Y probamos si podemos acceder, introduciendo la contraseña configurada:

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

Enter password:
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 10.2.15-MariaDB-10.2.15+maria~jessie mariadb.org binary
distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

COMENZAMOS EL LAB

Configurando nuestras aplicaciones: ConfigMaps

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#define-container-environment-variables-using-configmap-data>

ConfigMap nos permite definir un diccionario (clave,valor) para guardar información que puedes utilizar para configurar una aplicación.

Al crear un ConfigMap los valores se pueden indicar desde un directorio, un fichero o un literal, en nuestro laboratorio veremos como hacerlo a través de literales y pasandole un fichero de configuración a nuestro pod.

Al crear un ConfigMap los valores se pueden indicar desde un directorio, un fichero o un literal:

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

#kubectl get cm
NAME      DATA      AGE
mariadb   4          15s

#kubectl describe cm mariadb
```

Ahora podemos configurar el fichero yaml que define el despliegue:

mariadb-deployment-configmap.yaml

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mariadb-deploy-cm
  labels:
    app: mariadb
    type: database
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb
        type: database
    spec:
      containers:
        - name: mariadb
          image: mariadb
          ports:
            - containerPort: 3306
              name: db-port
      env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: mariadb
              key: root_password
        - name: MYSQL_USER
          valueFrom:
            configMapKeyRef:
              name: mariadb
              key: mysql_usuario
        - name: MYSQL_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: mariadb
              key: mysql_password
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
              name: mariadb
              key: basededatos

```

Creamos el despliegue y probamos el acceso:

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

# kubectl get pods -l app=mariadb

# kubectl exec -it mariadb-deploy-cm-57f7b9c7d7-ll6pv -- mysql -u usuario -p

Enter password:
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 10.2.15-MariaDB-10.2.15+maria~jessie mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| test          |
+-----+
2 rows in set (0.00 sec)
```

Ahora trabajaremos en este laboratorio con ConfigMap, para pasar un archivo de configuración a nuestra imagen de nginx:

En esta configuración de nginx, todo lo que llegue al puerto 80 lo reenviara al localhost al puerto 3000, que será nuestro contenedor helloworld

```
#cat /kubernetes-curso/configmap/reverseproxy.conf
```

```
server {
    listen      80;
    server_name localhost;

    location / {
        proxy_bind 127.0.0.1;
        proxy_pass http://127.0.0.1:3000;
    }

    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
}
```

Ahora creamos nuestro cm con el fichero de configuración:

```
# kubectl create configmap nginx-config --from-file=kubernetes-
curso/configmap/reverseproxy.conf

# kubectl get cm
# kubectl describe cm nginx-config
# kubectl get configmap nginx-config -o yaml
```

Ahora vemos que en este ficherero de configuración vamos a desplegar dos conetenedores, y al **contenedor nginx le montamos un volumen en /etc/nginx/conf.d, para pasarle el config map reverseproxy con el fichero de configuración reverseproxy.conf:**

```
# cat kubernetes-curso/configmap/nginx.yml

# kubectl create -f /kubernetes-curso/configmap/nginx.yml

# kubectl create -f /kubernetes-curso/configmap/nginx-service.yml

# kubectl get service
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)      AGE
helloworld-nginx-service  NodePort  10.97.251.124 <none>     80:30590/TCP  19s

# kubectl get pods -o wide
NAME                  READY  STATUS    RESTARTS  AGE   IP          NODE
helloworld-nginx     2/2    Running   0         3m30s  10.36.0.33  orion3.curso.local
<none>               <none>

# curl http://192.168.1.150:30590 -vvv
* About to connect() to 192.168.1.150 port 30590 (#0)
* Trying 192.168.1.150...
* Connected to 192.168.1.150 (192.168.1.150) port 30590 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 192.168.1.150:30590
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx/1.11.13
< Date: Fri, 08 Feb 2019 11:29:50 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 12
< Connection: keep-alive
< X-Powered-By: Express
< ETag: W/"c-7Qdih1MuhjZehB6Sv8UNjA"
<
* Connection #0 to host 192.168.1.150 left intact
Hello World!
```

Ahora podemos visualizar los logs que están corriendo en el pod, por cada contenedor, k8s-demo, nginx:

```
# kubectl logs -f helloworld-nginx -c k8s-demo  
# kubectl logs -f helloworld-nginx -c nginx
```

Ejecutamos un bash en el contenedor nginx y visualizamos el archivo de configuración de nginx que le estamos pasando a través de configMap:

```
# kubectl exec -ti helloworld-nginx -c nginx – bash  
  
root@helloworld-nginx:/# cat /etc/nginx/conf.d/reverseproxy.conf  
server {  
    listen 80;  
    server_name localhost;  
  
    location / {  
        proxy_bind 127.0.0.1;  
        proxy_pass http://127.0.0.1:3000;  
    }  
  
    error_page 500 502 503 504 /50x.html;  
    location =/50x.html {  
        root /usr/share/nginx/html;  
    }  
}
```

Como hemos visto en este laboratorio ConfiMap, permite injectar archivos de configuración a nuestras aplicaciones que tenemos corriendo en contenedores.

Laboratorio kubernetes (**NO REALIZAR**)

En este laboratorio de kubernetes+docker, instalaremos en los servidores minion la herramienta **Docker Registry**, la cual permite crear repositorios de imágenes de contenedores de forma local, el cual escuchara en el puerto 5000.

A continuación crearemos dos imágenes, una imagen de base de datos llamada (dbforweb) y una imagen de un servidor web llamada (webwithdb), las cuales comunicaremos a través de kubernetes.

Activamos Docker Registry, en los dos servidores minion1 y minion2:

```
# systemctl start docker-distribution
# systemctl enable docker-distribution
# systemctl is-active docker-distribution
```

Ahora que tenemos Docker Registry activo, en los servidores minios, comenzamos a crear las imágenes de los contenedores (dbforoweb y webwitdb).

Install and Deploy a MariaDB Container

Este procedimiento lo tendremos que realizar en los servidores minion1 y minion2:

```
# mkdir /laboratorio/mydbcontainer
# cd /laboratorio/mydbcontainer
```

Comenzamos descargándonos una imagen de Centos7 a nuestros servidores minion:

```
[root@minion1 mydbcontainer]# docker pull docker.io/0702/centos7
```

Nos aseguramos que tenemos la imagen descargada en los dos servidores minion:

```
[root@minion1 mydbcontainer]# docker images
REPOSITORY          TAG      IMAGE ID   CREATED       SIZE
docker.io/0702/centos7    latest   3a5f278e66df  8 weeks ago  368.4 MB
```

Descomprimimos el archivo (mariadb_cont_2.tgz):

```
# root@minion1 mydbcontainer]# tar xvf mariadb_cont*.tgz
[root@minion1 mydbcontainer]# pwd
/laboratorio/mydbcontainer
[root@minion1 mydbcontainer]# ls
Dockerfile  gss_db.sql  mariadb_cont_2.tgz
```

Creamos el archivo Dockerfile y lo modificamos a nuestras necesidades, en este laboratorio utilizaremos el que nos da el formador para el laboratorio:

```
[root@minion1 mydbcontainer]# cat Dockerfile

# Database container with simple data for a Web application
# Using RHEL 7 base image and MariaDB database
# Version 1

# Pull the rhel image from the local repository
FROM docker.io/0702/centos7
USER root

MAINTAINER Maintainer_Name

# Update image
RUN yum update -y --disablerepo=*-eus-* --disablerepo=*-htb-* --
disablerepo=*sjis* \
--disablerepo=*-ha-* --disablerepo=*-rt-* --disablerepo=*-lb-* \
--disablerepo=*-rs-* --disablerepo=*-sap-* \
--disablerepo=*-sjis-* > /dev/null

RUN yum-config-manager --disable *-eus-* *-htb-* *-ha-* *-rt-* *-lb-* \
*-rs-* *-sap-* *-sjis-* > /dev/null

# Add Mariahdb software
RUN yum -y install net-tools mariadb-server

# Set up Mariahdb database
ADD gss_db.sql /tmp/gss_db.sql
RUN /usr/libexec/mariadb-prepare-db-dir
RUN test -d /var/run/mariadb || mkdir /var/run/mariadb; \
chmod 0777 /var/run/mariadb; \
/usr/bin/mysqld_safe --basedir=/usr & \
sleep 10s && \
/usr/bin/mysqladmin -u root password 'redhat' && \
mysql --user=root --password=redhat < /tmp/gss_db.sql && \
mysqladmin shutdown --password=redhat

# Expose Mysql port 3306
EXPOSE 3306

# Start the service
CMD test -d /var/run/mariadb || mkdir /var/run/mariadb; chmod 0777 \
/var/run/mariadb;/usr/bin/mysqld_safe --basedir=/usr
```

Generamos el contenedore de la base de datos, asegurare de esta en el directorio que corresponde:

```
[root@minion1 mydbcontainer]# pwd
/laboratorio/mydbcontainer
[root@minion1 mydbcontainer]# docker build -t dbforweb .
```

```
[root@minion1 mydbcontainer]# docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
dbforweb            latest   13f1851998d7   About a minute ago  750.1 MB
```

Arrancamos la imagen dbforoweb:

```
[root@minion1 mydbcontainer]# docker run -d -p 3306:3306 --name=mydbforweb
dbforweb
```

Para probar el contenedor del servidor de bases de datos: Asumiendo que la interfaz docker0 en el host es 172.17.63.1 (la suya puede ser diferente), compruebe que el contenedor de la base de datos esté operativo ejecutando el comando:

```
[root@minion1 mydbcontainer]# nc -v 172.17.63.1 3306
Ncat: Version 6.40 ( http://nmap.org/ncat )
Ncat: Connected to 172.17.63.1:3306.

R
5.5.52-MariaDB*@Cp,)%■~p4Px+jVNd.Wmysql_native_password
```

Tras comprobar el correcto funcionamiento paramos el contenedor y lo eliminamos:

```
[root@minion1 mydbcontainer]# docker stop mydbforweb
mydbforweb
[root@minion1 mydbcontainer]# docker rm mydbforweb
mydbforweb
```

Install and Deploy Apache Web Server Container

NO REALIZAR

Los archivos de configuración los suministrara el formador para los laboratorios.

Nos posicionamos en el directorio /laboratorio y creamos el directorio:

```
mkdir ~/mywebcontainer
[root@minion1 mywebcontainer]# # cd ~/mywebcontainer
[root@minion1 mywebcontainer]# # touch action Dockerfile
```

Crear un script CGI: cree el archivo de acción en el directorio ~ / mywebcontainer, que se utilizará para obtener datos del contenedor del servidor de base de datos backend. Este script asume que la interfaz docker0 en el sistema host está en la dirección IP 172.17.63.1, puede iniciar sesión en la base de datos con la cuenta de **usuario dbuser1 y redhat** como contraseña y usar la base de datos denominada gss. Si esa es la dirección IP y utiliza el contenedor de la base de datos descrito más adelante, no es necesario modificar esta secuencia de comandos. (También puede ignorar este script y utilizar el servidor Web para obtener contenido HTML).

Asegurarse de modificar este valor en el script CGI action, para cada servidor minion, comprobando el interface de docker0, en cada miniom:

```
[root@minion1 mywebcontainer]# cat action
#!/usr/bin/python
# -*- coding: utf-8 -*-
import MySQLdb as mdb
import os

con = mdb.connect(os.getenv('DB_SERVICE_SERVICE_HOST','172.17.63.1'), 'dbuser1', 'redhat', 'gss')
```

with con:

```
cur = con.cursor()
cur.execute("SELECT MESSAGE FROM atomic_training")
```

```
rows = cur.fetchall()

print 'Content-type:text/html\r\n\r\n'

print '<html>'

print '<head>'

print '<title>My Application</title>'

print '</head>'

print '<body>'

for row in rows:

    print '<h2>' + row[0] + '</h2>'

    print '</body>'

print '</html>'

con.close()
```

Creamos el archivo Dockerfile, en el directorio /laboratorio/mywebcontainer con el contenido:

```
[root@minion1 mywebcontainer]# cat Dockerfile
# Webserver container with CGI python script
# Using RHEL 7 base image and Apache Web server
# Version 1
# Pull the rhel image from the local registry
FROM docker.io/0702/centos7
USER root
MAINTAINER Maintainer_Name

# Fix per https://bugzilla.redhat.com/show_bug.cgi?id=1192200
RUN yum -y install deltarpm yum-utils --disablerepo=-eus-* --disablerepo=-htb-* *-sjis-* \
--disablerepo=-ha-* --disablerepo=-rt-* --disablerepo=-lb-* --disablerepo=-rs-* --disablerepo=-sap-* \
sap-* 

RUN yum-config-manager --disable *-eus-* *-htb-* *-ha-* *-rt-* *-lb-* *-rs-* *-sap-* *-sjis* > /dev/null

# Update image
RUN yum update -y
RUN yum install httpd procps-ng MySQL-python -y

# Add configuration file
ADD action /var/www/cgi-bin/action
RUN echo "PassEnv DB_SERVICE_SERVICE_HOST" >> /etc/httpd/conf/httpd.conf
RUN chown root:apache /var/www/cgi-bin/action
RUN chmod 755 /var/www/cgi-bin/action
RUN echo "The Web Server is Running" > /var/www/html/index.html
EXPOSE 80

# Start the service
CMD mkdir /run/httpd ; /usr/sbin/httpd -D FOREGROUND
```

Generamos el contenedor del servidor Web, asegurare de esta en el directorio que corresponde:

```
[root@minion1 mywebcontainer]# pwd
/laboratorio/mywebcontainer
[root@minion1 mywebcontainer]# docker build -t webwithdb .
```

Start the Web server container:

```
[root@minion1 mywebcontainer]# docker run -d -p 80:80 --name=mywebwithdb
webwithdb
f8a86bda85839e767539a635871f467a6d03831a1a19da5ee2b155fd2aa58577
```

Testteamos el corecto funcionamiento del contendor:

```
[root@minion1 mywebcontainer]# curl http://localhost/index.html
```

The Web Server is Running

Para ejecutar este comando, tendremos que tener el contenedor de la base de datos arrancado:

```
[root@minion1 mydbcontainer]# docker run -d -p 3306:3306 --name=mydbforweb
dbforweb
```

```
[root@minion1 mywebcontainer]# curl http://localhost/cgi-bin/action

<html>
<head>
<title>My Application</title>
</head>
<body>
<h2>RedHat rocks</h2>
<h2>Success</h2>
</body>
</html>
```

Paramos y elminimamos los dos contenedores:

```
[root@minion1 mywebcontainer]# docker stop mydbforweb mywebwithdb
mydbforweb
mywebwithdb
```

```
[root@minion1 mywebcontainer]# docker rm mydbforweb mywebwithdb
mydbforweb
mywebwithdb
```

Podemos añadir nuestro propio contenido al servidor web, mapeando un volumen local:

```
# docker run -d -p 80:80 -v /var/www/html:/var/www/html \
--name=mywebwithdb webwithdb
```

Asegurarse de realizar este procedimiento en los dos servidores minion.

Ahora realizamos el Tag de las imágenes en los dos servidores minion, verificar en el lab el ID IMAGE, en cada minion:

En el servidor minion1:

```
[root@minion1 /]# docker images
REPOSITORY          TAG      IMAGE ID
CREATED             SIZE
webwithdb           latest   f13777f0c45a   About an
hour ago            917.4 MB
dbforweb            latest   13f1851998d7   About an hour
ago                750.1 MB
```

```
[root@minion1 /]# docker tag 13f1851998d7 localhost:5000/dbforweb
```

```
[root@minion1 /]# docker tag f13777f0c45a localhost:5000/webwithdb
```

```
[root@minion1 /]# docker push localhost:5000/dbforweb
```

```
[root@minion1 /]# docker push localhost:5000/webwithdb
```

En el servidor minion2:

```
[root@minion2 /]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
webwithdb	latest	89f3c67dc993	55 minutes ago	917.5 MB
dbforweb	latest	87aa296540fb	About an hour ago	750.1 MB

```
[root@minion2 /]# docker tag 87aa296540fb localhost:5000/dbforweb
```

```
[root@minion2 /]# docker tag 89f3c67dc993 localhost:5000/webwithdb
```

```
[root@minion2 /]# docker push localhost:5000/dbforweb
```

```
[root@minion2 /]# docker push localhost:5000/webwithdb
```

Las dos imágenes ya están disponibles en local Docker Registry.

Con las dos imágenes de los contenedores en su lugar, ahora puede lanzar los contenedores utilizando pods Kubernetes.

Pods separados: Aunque puede lanzar varios contenedores en un solo pod, al tenerlos los pods separados, cada contenedor puede replicar varias instancias según lo requieran, sin tener que iniciar el otro contenedor.

Servicio de Kubernetes: Este procedimiento define los servicios de Kubernetes para la base de datos y los pods del servidor web para que los contenedores puedan pasar por Kubernetes para encontrar esos servicios. De esta forma, la base de datos y el servidor web pueden encontrarse sin conocer la dirección IP, el número de puerto ni el nodo en el que se ejecuta el servicio.

Creamos un servicio database en kubernetes: Database Kubernetes service **db-service.yaml**:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: db
  name: db-service
  namespace: default
spec:
  ports:
  - port: 3306
  selector:
    app: db
```

Creamos un archivo replication controller del servidor de base de datos **db-rc.yaml** que utilizaremos para implementar el pod de servidor de base de datos

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: db-controller
spec:
  replicas: 1
  selector:
    app: "db"
  template:
    metadata:
      name: "db"
      labels:
        app: "db"
  spec:
    containers:
      - name: "db"
        image: "localhost:5000/dbforweb"
        ports:
          - containerPort: 3306
```

Create a Web server Kubernetes Service file: Create a webserver-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webserver
  name: webserver-service
  namespace: default
spec:
  ports:
    - port: 80
  selector:
    app: webserver
```

Create a Web server replication controller file: **webserver-rc.yaml**

```

kind: "ReplicationController"
apiVersion: "v1"
metadata:
  name: "webserver-controller"
spec:
  replicas: 1
  selector:
    app: "webserver"
  template:
    spec:
      containers:
        - name: "apache-frontend"
          image: "localhost:5000/webwithdb"
          ports:
            - containerPort: 80
  metadata:
    labels:
      app: "webserver"
      uses: db

```

Orquestar los contenedores con kubectl: Con los dos archivos YAML en el directorio actual, ejecute los siguientes comandos para iniciar los pods para comenzar a ejecutar los contenedores:

```
[root@master laboratorio]# kubectl create -f db-service.yaml
service "db-service" created
```

```
[root@master laboratorio]# kubectl create -f db-rc-yaml
replicationcontroller "db-controller" created
```

```
[root@master laboratorio]# kubectl create -f webserver-service.yaml
service "webserver-service" created
```

```
[root@master laboratorio]# kubectl create -f webserver-rc.yaml
replicationcontroller "webserver-controller" created
```

Chequeamos los pods,rc y services:

```
[root@master laboratorio]# kubectl get pod,rc,services --all-namespaces=true
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	po/busybox	1/1	Running	37	3d
default	po/db-controller-4bm52	1/1	Running	0	2m
default	po/kubernetes-bootcamp-2100875782-0r7n0	1/1	Running	2	2d
default	po/kubernetes-bootcamp-2100875782-4slls	1/1	Running	2	2d
default	po/my-nginx	1/1	Running	1	1d
default	po/my-nginx-4jjn0	1/1	Running	1	1d
default	po/webserver-controller-2sphp	1/1	Running	0	1m
kube-system	po/heapster-v1.2.0-4001981223-t3c27	4/4	Running	12	2d
kube-system	po/kube-dns-v11-bh5xl	4/4	Running	12	3d
kube-system	po/kubernetes-dashboard-3543765157-10ph7	1/1	Running	3	3d
kube-system	po/monitoring-influxdb-grafana-v4-qnbcl	2/2	Running	6	2d

NAMESPACE	NAME	DESIRED	CURRENT	READY	AGE
default	rc/db-controller	1	1	1	2m
default	rc/my-nginx	2	2	2	1d
default	rc/webserver-controller	1	1	1	1m
kube-system	rc/kube-dns-v11	1	1	1	3d
kube-system	rc/monitoring-influxdb-grafana-v4	1	1	1	2d

```
[root@master laboratorio]# kubectl get services --all-namespaces=true
```

NAMESPACE	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default	db-service	10.254.67.185	<none>	3306/TCP	4m
default	kubernetes	10.254.0.1	<none>	443/TCP	3d
default	kubernetes-bootcamp	10.254.215.176	<nodes>	8080:31503/TCP	2d
default	my-nginx-service	10.254.109.132	<nodes>	80:30992/TCP	23h
default	webserver-service	10.254.14.120	<none>	80/TCP	3m

Comprobamos los contenedores: Si ambos contenedores están en ejecución y el contenedor del servidor Web puede ver el servidor de base de datos, debería poder ejecutar el comando curl para ver que todo funciona, como se indica a continuación (tenga en cuenta que la dirección IP coincide con la dirección de servicio web):

```
[root@master laboratorio]# kubectl describe service webserver-service
```

Name:	webserver-service
Namespace:	default
Labels:	app=webserver
Selector:	app=webserver
Type:	ClusterIP
IP:	10.254.14.120
Port:	<unset> 80/TCP

Endpoints: **172.17.63.8:80**

Session Affinity: None

No events.

```
[root@master laboratorio]# curl 172.17.63.8
```

The Web Server is Running

```
[root@master laboratorio]# curl http://172.17.63.8/cgi-bin/action
```

```
<html>
<head>
<title>My Application</title>
</head>
<body>
<h2>RedHat rocks</h2>
<h2>Success</h2>
</body>
</html>
```

Ahora podríamos posar el acceso al service **webserver-service** a través de nuestro balanceador haproxy, el formador explicara como realizarlo para tener acceso desde la red interna de la empresa, en este laboratorio esta exponiendo, comprobaremos que dirección tenemos en nuestro lab:

Endpoints: 172.17.63.8:80

```
[root@master haproxy]# vi /etc/haproxy/haproxy.cfg
```

```
frontend http-in
```

```
    bind 192.168.1.250:80
```

```
    acl is_site1 hdr_end(host) -i aplicaciones.miempresa.com
```

```
acl is_site2 hdr_end(host) -i www.miempresa.local
```

```
    acl is_site3 hdr_end(host) -i db.miempresa.local
```

```
backend site2
```

```
    balance roundrobin
```

```
    option httpclose
```

```
    option forwardfor
```

```
server s3 172.17.63.8:80 maxconn 32
```

```
[root@master haproxy]# systemctl restart haproxy.service
```

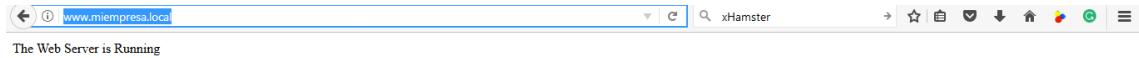
```
[root@master haproxy]# systemctl status haproxy.service
```

Resolvemos www.miempresa.com en:

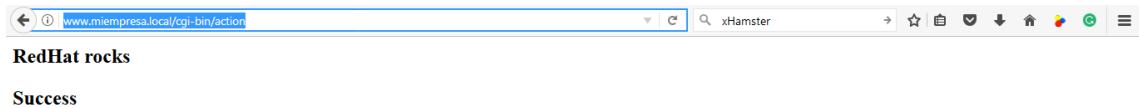
C:\Windows\System32\drivers\etc

192.168.1.250 www.miempresa.local

<http://www.miempresa.local/>



<http://www.miempresa.local/cgi-bin/action>



Tambien podemos observar los logs del pod:

```
[root@master haproxy]# kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
webserver-controller-2sphp     1/1     Running   0          24m
```

```
[root@master haproxy]# kubectl logs webserver-controller-2sphp
mkdir: cannot create directory '/run/httpd': File exists
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using
172.17.63.8. Set the 'ServerName' directive globally to suppress this message
```

Si cometimos un error al crear su aplicación en los pods, puede eliminar los rc y los servicios. (Los pods desaparecerán cuando se eliminan los replication controller). Después de eso, puede arreglar los archivos YAML y crearlos de nuevo.

A continuación, le indicamos cómo eliminar los controladores y servicios de replicación:

```
# kubectl delete rc webserver-controller
replicationcontrollers/webserver-controller

# kubectl delete rc db-controller
replicationcontrollers/db-controller

# kubectl delete service webserver-service
services/webserver-service

# kubectl delete service db-service
```

Laboratorio Kubernetes Service Discovery

En este laboratorio veremos como descubrir servicios, en kubernetes a usando el DNS, interno que tenemos en kubernetes.

DNS

- As of Kubernetes 1.3, DNS is a **built-in** service launched automatically using the addon manager
 - The addons are in the /etc/kubernetes/addons **directory** on **master node**
- The DNS service can be used within pods to **find other services** running on the same cluster
- Multiple containers **within 1 pod** don't need this service, as they can **contact** each other **directly**
 - A container in the same pod can connect the port of the other container directly using **localhost:port**
- To make DNS work, a pod will need a **Service definition**

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

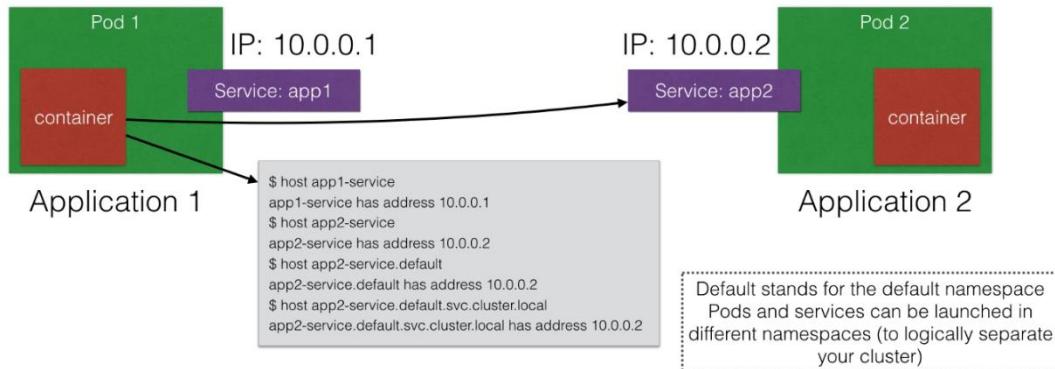
El servicio *KubeDNS* se comunica con el servidor de API y comprueba los servicios y pods creado para gestionar los diferentes registros de sus zonas de DNS.

El servicio DNS se puede usar dentro de los pods para encontrar otros servicios que se ejecutan en el mismo clúster. Los contenedores múltiples dentro de un pod no necesitan este servicio, ya que pueden contactarse entre sí directamente. Un contenedor en el mismo pod puede conectarse al puerto del otro contenedor directamente usando "localhost: port".

Si desea conectarse desde un servicio web en un pod a una base de datos en otro pod, entonces necesita Service Discovery porque no puede acceder a este otro pod, no conoce la dirección IP o el puerto. Para hacer de la red, un puerto siempre necesitará una definición de servicio, por lo que solo cuando cree un servicio para un pod, el pod se vuelve realmente accesible y el descubrimiento de servicios funcionará.

DNS

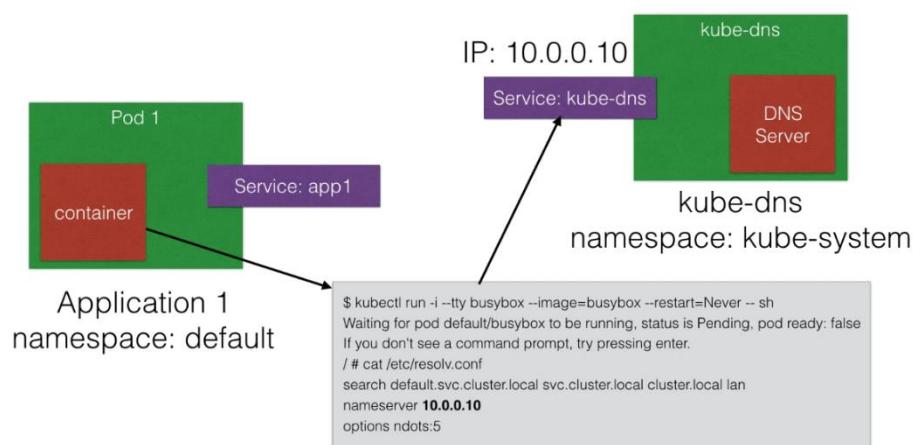
- An example of how app 1 could reach app 2 using DNS:



¿Qué se puede resolver?

- Cada vez que se crea un nuevo servicio se crea un registro de tipo A con el nombre `servicio.namespace.svc.cluster.local`.
- Para cada puerto nombrado se crea un registro SRV del tipo `_nombre-puerto._nombre-protocolo.my-svc.my-namespace.svc.cluster.local` que resuelve el número del puerto y al CNAME: `servicio.namespace.svc.cluster.local`.
- Para cada pod creado con dirección IP 1.2.3.4, se crea un registro A de la forma `1-2-3-4.default.pod.cluster.local`.

DNS - How does it work?



En este laboratorio desplegaremos un contenedor que contendrá una aplicación, la cual conectaremos a un pod en el que estará corriendo una base de datos de mysql, desplegaremos un archivo con los secretos con el nombre de la base de datos, el password, el usuario...

```
# cat /kubernetes-curso/service-discovery/secrets.yml
```

Podemos decodificar las encriptaciones del archivo secrets.yml

```
[root@docker /]# echo 'aGVsbG93b3JsZA==' | base64 --decode
```

Helloworld

```
[root@docker /]# echo 'cGFzc3dvcmQ=' | base64 --decode
```

Password

```
[root@docker /]# echo 'cm9vdHBhc3N3b3Jk' | base64 --decode
```

Rootpassword

```
[root@docker /]# echo 'aGVsbG93b3JsZA==' | base64 --decode
```

Helloworld

Desplegamos el archivo secrets:

```
# kubectl create -f /kubernetes-curso/service-discovery/secrets.yml
```

Desplegamos la base de datos de mysql, le pasamos al fichero de configuración, a través de las variables de entorno de la imagen de mysql, los secretos que definimos anteriormente de username, rootPassword, database, password:

```
# cat /kubernetes-curso/service-discovery/database.yml
```

Desplegamos la aplicación database y su servicio correspondiente:

```
# kubectl create -f /kubernetes-curso/service-discovery/database.yml
```

```
# kubectl create -f /kubernetes-curso/service-discovery/database-service.yml
```

```
#cat /kubernetes-curso/service-discovery/helloworld-db-service.yml
```

Ahora desplegamos la aplicación helloworld-db que conectara contra nuestra database y su servicio correspondiente:

```
# kubectl create -f /kubernetes-curso/service-discovery/helloworld-db.yml
```

```
# kubectl create -f /kubernetes-curso/service-discovery/helloworld-db-service.yml
```

Ahora podemos ver como relaciono la conexión a base de datos a través del nombre del servicio (database-service), que hemos desplegado y que lo descubrirá nuestro dns interno de kubernetes a través de service discovery:

helloworld-db.yml

```
command: ["node", "index-db.js"]
ports:
- name: nodejs-port
  containerPort: 3000
env:
- name: MYSQL_HOST
  value: database-service
- name: MYSQL_USER
  value: root
- name: MYSQL_PASSWORD
  valueFrom:
    secretKeyRef:
      name: helloworld-secrets
      key: rootPassword
- name: MYSQL_DATABASE
  valueFrom:
    secretKeyRef:
      name: helloworld-secrets
      key: database
```

helloworld-db-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  ports:
  - port: 3306
    protocol: TCP
  selector:
    app: database
  type: NodePort
```

```
# kubectl get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
database-service	NodePort	10.97.21.14	<none>	3306:32472/TCP	25m

```
#kubectl describe service helloworld-db-service
```

```
Name:           helloworld-db-service
Namespace:      default
Labels:          <none>
Annotations:    <none>
Selector:        app=helloworld-db
Type:            NodePort
IP:              10.110.160.155
Port:            <unset> 3000/TCP
TargetPort:      3000/TCP
NodePort:      <unset> 30713/TCP
Endpoints:      10.36.0.25:3000,10.36.0.26:3000,10.44.0.41:3000
Session Affinity: None
External Traffic Policy: Cluster
Events:          <none>
```

kubectl get pods -o wide

Ahora podemos ver como los pods de la aplicaciones están conectados a la base de datos, con visualizar los logs de uno de los pods:

```
[root@docker /]# kubectl logs helloworld-deployment-dd876ccc6-vhdqg
```

Example app listening at http://:::3000

Connection to db established

La conexión se establece gracias a la declaración en el archivo helloworld-db.yml:

```
env:
  - name: MYSQL_HOST
    value: database-service
  - name: MYSQL_USER
    value: root
  - name: MYSQL_PASSWORD
    valueFrom:
      secretKeyRef:
        name: helloworld-secrets
        key: rootPassword
  - name: MYSQL_DATABASE
    valueFrom:
      secretKeyRef:
        name: helloworld-secrets
        key: database
```

Si nos conectamos a los nodos a través del NodePort, veremos como va aumentado de valor nuestra aplicación por cada conexión que realizamos:

<http://192.168.1.152:30713/>



<http://192.168.1.152:30713/>

Hello World! You are visitor number 6

Ahora nos conectamos a nuestro pod de mysql , con el password de la cuenta roor (rootpassword**), sin los paréntesis, es la clave que codificamos anteriormente en nuestro secreto**

```
# kubectl exec database -ti -- mysql -u root -p
```

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| helloworld    |
| mysql          |
| performance_schema |
| sys            |
+-----+
mysql> use helloworld;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
```

Database changed

```
mysql> show tables;
+-----+
| Tables_in_helloworld |
+-----+
| visits      |
+-----+
1 row in set (0.00 sec)
```

```
mysql> select * from visits;
+---+-----+
| id | ts      |
+---+-----+
| 1 | 1549567071238 |
| 2 | 1549567073996 |
| 3 | 1549567175995 |
| 4 | 1549567185506 |
| 5 | 1549567198987 |
| 6 | 1549567203151 |
```

```
| 7 | 1549567485521 |
```

```
+-----+
```

7 rows in set (0.01 sec)

Ahora creamos y nos conectamos a un contenedor, dentro del clusrter, para ver como resuelve a través del dns interno de kubernetes nuestro servicios:

```
# kubectl run -i --tty buxybox --image=busybox --restart=Never – sh
```

```
/ # nslookup helloworld-db-service
```

```
Server: 10.96.0.10
```

```
Address: 10.96.0.10:53
```

```
Name: helloworld-db-service.default.svc.cluster.local
```

```
Address: 10.110.160.155
```

Desde el interior del contenedor busybox, realizamos el telnet y ejecutamos el get:

```
/# telnet helloworld-db-service 3000
```

GET /

```
HTTP/1.1 200 OK
```

```
X-Powered-By: Express
```

```
Content-Type: text/html; charset=utf-8
```

```
Content-Length: 37
```

```
ETag: W/"25-PISG+fvxVYkXqSb8BRQIbQ"
```

```
Date: Thu, 07 Feb 2019 19:42:53 GMT
```

```
Connection: close
```

Hello World! You are visitor number 8Connection closed by foreign host

Finalmente eliminamos el pod buxybox

```
[root@docker /]# kubectl delete pod buxybox
```

Laboratorio kubernetes Affinity and Anti-Affinity

Affinity and anti-affinity

- The affinity/anti-affinity feature allows you to do **more complex scheduling** than the nodeSelector and also **works on Pods**
 - The language is **more expressive**
 - You can create **rules that are not hard requirements**, but rather a **preferred rule**, meaning that the scheduler will still be able to schedule your pod, even if the rules cannot be met
 - You can create rules that take other pod labels into account
 - For example, a rule that makes sure 2 different pods will never be on the same node

La función de afinidad/antiafinidad le permite realizar una programación más compleja que nodeSelector y también funciona en pods.

No es solo para nodos, también se utiliza para pods.

El lenguaje que puedes usar es más expresivo, podemos crear reglas que no son requisitos estrictos, sino una regla preferida, lo que significa que el programador aún podrá programar su pod, incluso si las reglas no se pueden cumplir.

Entonces, con un nodeSelector, si un nodo no tiene su etiqueta, nunca se programará, mientras que, con afinidad y antiafinidad, puede crear un tipo diferente de regla o establecer una preferencia.

Por lo tanto, preferirá los nodos con una etiqueta específica, pero si no hay un nodo con esa etiqueta programable, aún se programará en otro nodo.

Puede crear reglas que tengan en cuenta otras etiquetas de pod.

Por ejemplo, puede crear una regla que asegure que dos pods diferentes nunca estarán en los mismos nodos.

Affinity and anti-affinity

- Kubernetes can do **node affinity** and **pod affinity/anti-affinity**
 - Node affinity is similar to the nodeSelector
 - Pod affinity/anti-affinity allows you to create rules **how pods should be scheduled taking into account other running pods**
 - Affinity/anti-affinity mechanism is **only relevant during scheduling**, once a pod is running, it'll need to be recreated to apply the rules again
- I'll first cover **node affinity** and will then cover pod affinity/anti-affinity

Pod affinity/anti-affinity le permite crear reglas sobre cómo deben programarse los pods teniendo en cuenta otros pods que se ejecutan.

Affinity/anti-affinity El mecanismo solo es relevante durante la programación, una vez que se ejecuta un pod, deberá volver a crearse para aplicar las reglas nuevamente.

Primero cubrirémos la afinidad del nodo y luego cubrirémo de del pod.

Affinity and anti-affinity

- There are currently 2 types you can use for node affinity:
 - 1) requiredDuringSchedulingIgnoredDuringExecution
 - 2) preferredDuringSchedulingIgnoredDuringExecution
- The **first one** sets a **hard requirement** (like the nodeSelector)
 - The rules must be met before the pod can be scheduled
- The **second type** will try to enforce the rule, but it will not guarantee it
 - Even if the rule is not met, the pod can still be scheduled, it's a soft requirement, a preference
- El primero establece un requerimiento hard, con el nodeSelector.
 - las reglas deben cumplirse antes de que el pod pueda ser reprogramado.
- El segundo tipo se ejecutará para hacer cumplir la regla, pero no lo garantizará
 - Incluso si la regla no se cumple, el pod puede programarse, es un requisito indispensable

Affinity and anti-affinity

```
→ spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: env
                operator: In
                values:
                  - dev
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: team
                operator: In
                values:
                  - engineering-project1
    containers:
      [...]
```

```
# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
docker	Ready	master	24d	v1.13.2
docker2	Ready	<none>	24d	v1.13.2
orion3.curso.local	Ready	<none>	24d	v1.13.2

Ahora podemos describir los nodos y fijarnos en las labels, que tienen estos nodos:

```
# kubectl describe nodes docker
# kubectl describe nodes docker2
# kubectl describe nodes orion3.curso.local
```

Podemos observar que necesitamos env=dev y que de forma opcional, team=engineering-project1

```
# cat /kubernetes-curso/affinity/node-affinity.yaml
```

```
# kubectl create -f /kubernetes-curso/affinity/node-affinity.yaml
```

Ahora veremos que los pods, están en estado pendiente, porque no hemos creado las labels:

```
# kubectl get pods
```

Etiquetamos con la label obligatoria a nodo docker2:

```
# kubectl label node docker2 env=dev
```

Y ahora veremos como se crean los PODS:

```
#kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED
NODE	READINESS	GATES					
node-affinity-59c9767f55-9f6tt	1/1	Running	0	3m29s	10.44.0.54	docker2	<none> <none>
node-affinity-59c9767f55-9gx6b	1/1	Running	0	3m29s	10.44.0.52	docker2	<none>
<none>							
node-affinity-59c9767f55-sn7w	1/1	Running	0	3m29s	10.44.0.53	docker2	<none>
<none>							
webtest	2/2	Running	6	26h	10.44.0.50	docker2	<none> <none>

Si ahora describimos un pod, podemos ver en campo Events:

```
# kubectl describe pod node-affinity-59c9767f55-sn7w
```

```
# kubectl label node orion3.curso.local team=engineering-project1
```

Laboratorio kubernetes Volumes

<https://kubernetes.io/docs/concepts/storage/volumes/>

Kubernetes soporta multiples tipos de Volúmenes:

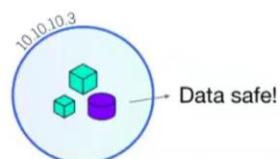
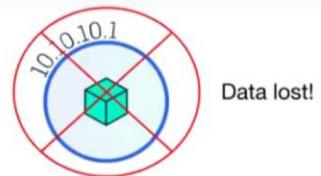
Enlace de doc: <https://kubernetes.io/docs/concepts/storage/volumes/>

- **emptyDir:** Este tipo de volumen es creado cuando un pod es asignado a un nodo, y existirá tanto tiempo como el mismo pod. Inicialmente este volumen esta vacío y permite a contenedores compartir ficheros. Todos estos puede leer y escribir del volumen. El punto de montaje puede diferir en los contenedores que usan este volumen. Con este tipo de volumen es posible usar como dispositivo de almacenamiento la memoria. Si usamos la propiedad emptyDir.medium con valor Memory este se montará usando tmpfs (RAM filesystem). Este tipo de sistema de almacenamiento es bastante rápido pero: 1) consume memoria de la maquina; y 2) no mantiene la información.
- **hostPath:** Nada recomendable ya que se añade la persistencia en el hosts que este aloja.
- **awsElasticBlockStore:** Este tipo de volúmenes usan el sistema EBS de AWS. Los volúmenes tienen que estar en instancias de tipo AWS, en la misma región y availability zone para poder ser utilizados entre contenedores. Solo un contenedor puede ser montado en una maquina. EBS snapshots es una funcionalidad disponible en AWS volumes.
- **nfs:** nfs volumen permite montar un nfs share en un pod. A diferencia de emptyDir el cual es borrado una vez el pod es borrado. El contenido del volumen persiste y el volumen es simplemente desmontado, umount. Este tipo de volúmenes permite tener múltiple puntos de escritura, la información puede ser compartido entre pods múltiples.
- **iscsi:** Permite montar un volumen existente de tipo iSCSI. Este tipo persiste una vez que el pod es borrado y la información puede ser reutilizada. Una característica de iSCSI es que puede ser montado como solo lectura y accedido por múltiples consumidores simultáneamente. Sin embargo, este tipo solo permite tener un consumidor de tipo read-write, así que no hay varios escritores simultáneamente como ocurre con nfs.
- **Flocker:** Proporciona una herramienta para gestionar y orquestar la gestión de nuestros volúmenes con soporte para diversos tipos de sistemas de almacenamiento. En otras palabras, ofrece un sistema agnóstico para la gestión de volúmenes en diferentes plataformas. Un volumen Flocker permite montar un dataset en un pod. Si este volumen no existe en Flocker control, se creará uno nuevo.
- **Glusterfs:** Un volumen glusterfs permite usar el sistema de ficheros en red de glusterfs. El contenido de este volumen permanece después de borrar el pod. Al igual que nfs, puede ser montado con múltiples escrituras accediendo simultáneamente.
- **rbd:** Es un volumen para Rados Block Device que puede ser montado en tus pods. Al igual que la mayoría, la información persiste incluso después de borrar el pod. Como iSCSI, solo puede haber un consumidor con derecho a lectura mientras que no hay límite con consumidores de tipo lectura.

- **gitRepo:** Es un ejemplo de lo que se puede conseguir con un plugin para volúmenes. Este monta un directorio vacío y clona un repositorio git en el pod para su uso. En el futuro, este tipo de volúmenes deberán ser más independientes y no depender de la API de kubernetes.
- **Secret:** Es un volumen que se utiliza para compartir credenciales en un pod. A través de la API de kubernetes podemos almacenar credenciales que luego pueden montarse como volúmenes en los pods. Los volúmenes de tipo secret usan tmpfs (RAM filesystem) por lo que nunca son almacenados en discos persistentes.
- **downwardAPI:** Este volumen se usa para hacer accesible información downward API a las aplicaciones. Este monta un directorio y escribe las peticiones en forma de ficheros de texto.

Volumes

- Some applications require persistent data
- Pods are ephemeral
- We need a way to share content between containers in the same pod
- Solution: use volumes to provide persistent storage



Una de las características que tenemos en kubernetes es que los pods, son efímeros, pueden morir o kubernetes decide moverlo de nodo, no todos los volúmenes son persistentes, en este primer laboratorio utilizaremos un volumen entre contenedores para compartir datos.

En este ejemplo, estaremos compartiendo datos entre distintos contenedores dentro de un pod, y tenemos un volumen llamado test, de tipo emptyDir, un emptydir, es básicamente un directorio que está vacío y que se genera cuando kubernetes lo lanza y los contenedores que generan en este pod, lo pueden montar en su sistemas de ficheros.

Tipos de Volumes

Adding persistence to a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
    - image: busybox
      name: busy
      volumeMounts:
        - mountPath: /busy
          name: test
  volumes:
    - name: test
      <????????>
```

This depends on the type of volume

hostPath: local node filesystem hostPath: path: /data	gcePersistentDisk: Google Compute Engine storage gcePersistentDisk: pdName: my-data-disk fsType: ext4
awsElasticBlockStore: AWS EBS storage awsElasticBlockStore: volumeID: vol-8a07f3e37b fsType: ext4	gitRepo: clone a git repository gitRepo: repository: "git@github.com:my/repo.git" revision: "22f1d8406d464b0c087407253775"
configMap: kubernetes config map configMap: name: log-config items: - key: log_level path: log_level	secret: kubernetes secret secret: secretName: mysecret

En la definición, podemos ver que cada contenedor lo monta en un sistema distinto, en este ejemplo el primer contenedor lo monta en /busy y hace referencia al volumen declarado llamado test y el segundo contenedor lo monta en /box y hacer referencia a nuestro volumen llamado test.

Example: sharing volumes between containers

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
    - image: busybox
      volumeMounts:
        - mountPath: /busy
          name: test
        - name: busy
    - image: busybox
      volumeMounts:
        - mountPath: /box
          name: test
        - name: box
  volumes:
    - name: test
      emptyDir: {}
```

emptyDir: persistent storage only until pod is deleted

Creamos nuestro fichero yaml, para desplegar un pod con dos contenedores y un volumen comparitdo entre los contenedores de tipo emptyDir:

```
# vi webtest-volum-emptydir.yml

apiVersion: v1
kind: Pod
metadata:
  name: webtest
  labels:
    app: webtest
spec:
  containers:
    - name: contenedor1
      image: busybox
      command: ["sleep", "3600"]
      volumeMounts:
        - name: test
          mountPath: /web1
    - name: contenedor2
      image: wardviaene/k8s-demo
      ports:
        - containerPort: 3000
      volumeMounts:
        - name: test
          mountPath: /web2
  volumes:
    - name: test
      emptyDir: {}
```

```
# kubectl create -f webtest-volum-emptydir.yml
pod/webtest created
```

Nos aseguramos de que los contenedores estén creados en el pod:
kubectl get pod

Si ahora entramos en el contenedor2 y creamos un archivo en /web2, se tendría que ver y modificar en el contenedor1:

```
# kubectl exec -ti webtest -c contenedor2 -- bash
```

Si ahora entramos en el contenedor1 y creamos un archivo en /web1, se tendría que ver y modificar en el contenedor2, utilizamos el shell ash para la imagen busybox:

```
# kubectl exec -ti webtest -c contenedor1 – ash
```

Este tipo de volumen, emptyDir, puede ser interesante, por ejemplo, si tenemos una aplicación dejaremos los logs en este emptyDir, y metemos otro contenedor que lea estos logs, y nos valla lanzando alertas, etc.

hostPath

<https://kubernetes.io/docs/concepts/storage/volumes/#hostpath>

Un volumen de tipo **hostPath** monta un archivo o directorio desde el sistema de archivos del nodo host a su Pod.

En este laboratorio configuraremos una base de datos mysql con un volumen persistente contra el nodo orion3.curso.local, persistiendo el directorio de la base de datos /var/lib/mysql en el directorio del nodo /basedatos-mysql, recordar que si no existe el directorio se crea automáticamente, utilizaremos affinity contra el nodo orion3.curso.local, para que siempre se encuentren los datos salvados de la base de datos:

Etiquetamos nuestro nodo con la variable env con el valor bd:

```
#kubectl label nodes orion3.curso.local env=bd
```

Describimos el nodo para visualizar sus labels:

```
# kubectl describe nodes orion3.curso.local
```

Creamos el archivo con la configuración, visualizamos la configuración nodeAffinity y volumens:

mysql-affinity.yml

```

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: env
                    operator: In
                    values:
                      - bd
      containers:
        - image: mysql:5.7
          name: mysql
          env:
            # Use secret in real usage
            - name: MYSQL_ROOT_PASSWORD
              value: password
          ports:
            - containerPort: 3306
              name: mysql
      # nodeSelector:
      #   bd: mysql
      volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
        - name: mysql-persistent-storage
          hostPath:
            path: /basedatos-mysql

```

Desplegamos nuestro Deployment:

```
[root@docker kubernetes-curso]# kubectl create -f mysql-affinity.yml
```

Ahora visulizamos que el pod se ha creado en el nodo orion3 y que tenemos persistido la base de datos en el directorio /basedatos-mysql, en el nodo orion3:

```
# kubectl get pods -o wide
mysql-5485bccd89-5xczp      1/1   Running   0      16s   10.36.0.44   orion3.curso.local <none>
<none>
```

En orion3, comprobamos que se ha creado el volumen:

```
[root@orion3 /]# cd /basedatos-mysql/
[root@orion3 basedatos-mysql]# ls
auto.cnf      client-key.pem ib_logfile1      private_key.pem sys
ca-key.pem    ib_buffer_pool ibtmp1          public_key.pem
ca.pem        ibdata1       mysql           server-cert.pem
client-cert.pem ib_logfile0  performance_schema server-key.pem
```

Ahora podemos entrar en nuestro pod y ejecutar mysql, crearemos una base de datos llamana kubernetes, a continuación, eliminamos el deployment, volveremos a desplegarlo y tendremos que comprobar que la base de datos de kubernetes creada anteriormente, tiene que estar persistida:

```
# kubectl exec -it mysql-5485bccd89-5xczp -- mysql -u root -p
mysql> create database kubernetes;
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| kubernetes     |
| mysql          |
| performance_schema |
| sys            |
+-----+
5 rows in set (0.09 sec)
```

Laboratorio kubernetes Volúmenes NFS

<https://kubernetes.io/docs/concepts/storage/volumes/#nfs>

En este laboratorio configuraremos un volumen NFS desde el servidor master (192.168.1.150), que exportara el directorio /bd-nfs, donde persistiremos nuestro contenedor de mysql:

```
# cat /etc/exports
/bd-nfs *(rw,no_root_squash)
# systemctl restart nfs-server
```

En los nodos clientes tener el servicio iniciado:

```
#systemctl start rpcbind
```

Almacenamiento para Kubernetes

Para dar almacenamiento persistente en Kubernetes, podremos utilizar:

- NFS
- iSCSI
- RBD (Ceph Block Device)
- Glusterfs
- HostPath
- GCEPersistentDisk
- AWSElasticBlockStore

Para cualquiera de ellas en Kubernetes tendremos que crear:

- **PersistentVolume** -- Donde especificamos el volumen persistente
- **PersistentVolumeClaim** -- Donde reclamamos espacio en el volumen

Modos de acceso:

- ReadWriteOnce -- read-write solo para un nodo (RWO)
- ReadOnlyMany -- read-only para muchos nodos (ROX)
- ReadWriteMany -- read-write para muchos nodos (RWX)

Políticas de reciclaje de volúmenes son son:

- Retain - Reclamación manual
- Recycle - Reutilizar contenido
- Delete - Borrar contenido

Estados de un volumen:

- Available - disponible para reclamación
- Bound - No disponible, se está utilizando por una reclamación.
- Released - La reclamación del volumen se ha eliminado y está esperando otra petición del cluster.
- Failed - En estado de fallo.

Una vez verificado, crearemos un yaml de pv y pvc, desde nuestro servidor de Kubernetes y verificaremos su correcto funcionamiento:

```
# vi nfs-pv-pvc.yaml
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kube-nfs-pv
spec:
  storageClassName: storage-nfs
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.150
    path: "/bd-nfs"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: kube-nfs-pvc
spec:
  storageClassName: storage-nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

```
#kubectl create -f nfs-pv-pvc.yaml
```

```
# kubectl get pv,pvc
```

Ahora, solamente nos queda crear un deployment para mysql enlazados con nuestros PV y PVC:

```
# vi mysql-storage-nfs.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-storage-nfs
spec:
  selector:
    matchLabels:
      app: mysql-nfs
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql-nfs
    spec:
      containers:
        - image: mysql:5.7
          name: mysql
          env:
            # Use secret in real usage
            - name: MYSQL_ROOT_PASSWORD
              value: password
          ports:
            - containerPort: 3306
              name: mysql
        volumeMounts:
          - name: kube-nfs-pvc
            mountPath: /var/lib/mysql
      volumes:
        - name: kube-nfs-pvc
          persistentVolumeClaim:
            claimName: kube-nfs-pvc
```

```
# kubectl create -f mysql-storage-nfs.yml
```

```
# kubectl get pod -o wide
```

```
# kubectl exec -ti mysql-storage-nfs-cd667d85f-7qtk7 -- bash
```

```
root@mysql-storage-nfs-cd667d85f-7qtk7:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay         17G   4.7G   13G  28% /
tmpfs           64M     0   64M  0% /dev
tmpfs           916M     0   916M  0% /sys/fs/cgroup
/dev/mapper/cl-root  17G   4.7G   13G  28% /etc/hosts
shm              64M     0   64M  0% /dev/shm
192.168.1.150:/bd-nfs  26G   6.6G   20G  26% /var/lib/mysql
```

```
# kubectl exec -it mysql-storage-nfs-cd667d85f-7qtk7 -- mysql -u root -p

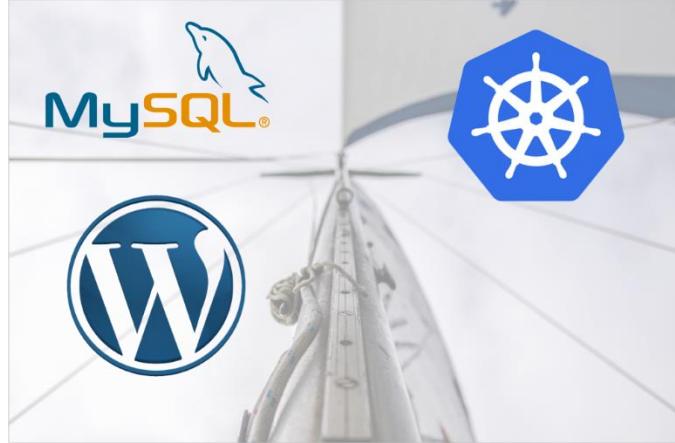
mysql> create database intranet;
Query OK, 1 row affected (0.10 sec)

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| intranet      |
| mysql          |
| performance_schema |
| sys            |
+-----+
5 rows in set (0.13 sec)
```

Si ahora realizamos un ls al directorio del servidor NFS, /bd-nfs vermos que tenemos creada la base de datos intranet.

A continuación, eliminar el deployment mysql-storage-nfs, si lo volvemos a desplegar comprobar que llegamos al punto de montaje nfs, y tenemos persistida la base de datos intranet, en en nuevo contenedor.

Laboratorio Wordpress kubernetes volúmenes NFS



En este laboratorio desplegaremos la aplicación Wordpress sobre kubernetes, persitiendo la base de datos de mysql y lo ficheros estáticos de Wordpress, con volúmenes en un servidor NFS:

En el directorio /kubernetes-curso/WordPress-kubernetes, estarán los ficheros de configuración para realizar el despliegue:

**Utilizaremos como servidor NFS, nuestro master de kubernetes
192.168.1.150:**

```
[root@docker WordPress-kubernetes]# cat /etc/exports
```

```
/bd-nfs *(rw,no_root_squash)
/mysql *(rw,no_root_squash)
/html *(rw,no_root_squash)
```

```
[root@docker /]# systemctl restart nfs-server
```

En el / de nuestro servidor master creamos los directorio mysql y html donde persistiremos nuestra bd de mysql y nuestro wordpress:

```
# mkdir/{mysql,html}
# chmod -R 755/{mysql,html}
# chown nfsnobody:nfsnobody/{mysql,html}
```

Crear un secreto para la contraseña de MySQL

Un **secreto** es un objeto que almacena una parte de datos confidenciales como una contraseña o clave. Cada elemento en un secreto debe estar codificado en base64. Vamos a crear un secreto para uso de la cuenta root.

Codifique la contraseña (en nuestro caso - admin):

```
#echo -n 'admin' | base64
```

Creamos un secret-mysql-pass.yml, para desplegar nuestro secreto de mysql:

```
# cat secret-mysql-pass.yml
```

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-pass
type: Opaque
data:
  password: YWRtaW4=
```

Desplegamos el secreto en kubernetes

```
# kubectl create -f secret-mysql-pass.yml
```

Deploy Persistent Volume for WordPress y MySQL

```
# cat pv-wordpress-mysql.yml

# Create PersistentVolume
# change the ip of NFS server
apiVersion: v1
kind: PersistentVolume
metadata:
  name: wordpress-persistent-storage2
  labels:
    app: wordpress
    tier: frontend
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.150
    # Exported path of your NFS server
    path: "/html"

---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-persistent-storage2
  labels:
    app: wordpress
    tier: mysql
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.1.150
    # Exported path of your NFS server
    path: "/mysql"
```

Desplegamos el PV, para wordpress y mysql:

```
# kubectl create -f pv-wordpress-mysql.yml
```

Deploy PersistentVolumeClaim(PVC)

El PVC es una solicitud de almacenamiento que en algún momento puede estar disponible, vinculada a algún PV real.

```
# cat pvc-mysql.yml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-persistent-storage
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 6Gi
```

```
# cat pvc-wordpress.yml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wordpress-persistent-storage
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 6Gi
```

Desplegamos PersistentVolumeClaim:

```
# kubectl create -f pvc-wordpress.yml
# kubectl create -f pvc-mysql.yml
```

Deploy MySQL

En este deploy observamos que estamos desplegando también el service, para mysql llamado wordpress-mysql.

```
# cat mysql-deploy.yml
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql    # will be used as a value in
  labels:                   # WORDPRESS_DB_HOST in wordpress-deploy.yml
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
    tier: mysql
  clusterIP: None
---
apiVersion: apps/v1beta2 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
        - image: mysql:5.6
          name: mysql
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-pass           # the one generated before in secret.yml
                  key: password
          ports:
            - containerPort: 3306
              name: mysql
          volumeMounts:
            - name: mysql-persistent-storage  # which data will be stored
              mountPath: "/var/lib/mysql"
          volumes:
            - name: mysql-persistent-storage    # PVC
              persistentVolumeClaim:
                claimName: mysql-persistent-storage
```

Realizamos el deploy, y nos aseguramos de que esta todo correcto:

```
#kubectl create -f mysql-deploy.yml
```

Deploy WordPress

Observamos que el el deploy estamos desplegando un service de tipo NodePort (30000), que será el puerto donde estemos publicando nuestro wordpress en los minion

```
# cat wordpress-deploy.yml
```

```
# create a service for wordpress
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
      nodePort: 30000
  selector:
    app: wordpress
    tier: frontend
  type: NodePort
---
apiVersion: apps/v1beta2 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
        - image: wordpress:4.8-apache
          name: wordpress
          env:
            - name: WORDPRESS_DB_HOST
              value: wordpress-mysql
            - name: WORDPRESS_DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-pass           # generated before in secret.yml
                  key: password
          ports:
            - containerPort: 80
              name: wordpress
          volumeMounts:
            - name: wordpress-persistent-storage
              mountPath: "/var/www/html"      # which data will be stored
      volumes:
        - name: wordpress-persistent-storage
          persistentVolumeClaim:
            claimName: wordpress-persistent-storage
```

Desplegamos el deployment y el service y nos aseguramos de que todo este correcto:

```
#kubectl create -f wordpress-deploy.yml
```

Ahora en cualquier nodo minion, en el puerto 30000, que es donde tenemos le servicio wordpress, realizamos la instalación y vemos que todo este correcto, a continuación eliminamos los deploymets y services de wordpress y mysql, los volvemos a desplegar y tedremos que ver que tenemos persistido la web de Wordpress.

<http://192.168.1.152:30000/>

<http://192.168.1.155:30000/>

Laboratorio kubernetes Volúmenes NFS

En este laboratorio de kubernetes trabajaremos con volúmenes NFS, recordar que los arvhivos en disco en contenedores son efímeros, lo que representa algunos problemas para las aplicaciones cuando corren en contenedores. En primer lugar, cuando un contenedor se rompe, kubernetes lo reiniciara pero los archivos se perderan y el contenedor se iniciara limpio. En segundo lugar cuando dos contenedores corren en un pod, a veces es necesario que comparten archivos entre ellos. Los volúmenes de kubernetes resuelven dicho problema.

En el servidor master (192.168.1.250), creamos una ruta de directorios:

/shared/kubernetes/wordpress

/shared/kubernetes/mysql

El laboratorio se basa en que instalamos en el servidor de forma local (192.168.1.250) un CMS llamado joomla que lo sirve apache y una base de datos local (mariadb), las rutas de la instalación en local son:

Apache → /var/www/html

MariaDB → /var/lib/mysql

Lo que realizaremos será sincronizar estas dos rutas con los directorios en /shares/kubernetes, de tal forma que a traves del servicio NFS, se exporten a los servidores minions, los cuales a través de kubernetes crearemos un pod para la base de datos mariadb y montaremos un volumenes en /shared/kubernetes/mysql, y crearemos un pod para wordpress y montaremos un volumen para este pod, de estea forma están sirviendo la web de la empresa, el pod de wordpress, lo escalaremos con 3 replicas para balancearlo.

Tras la instalación de joomla, realizada por el formador.

Comando para sincronizar el contenido:

```
[root@master /]# rsync -avP /var/lib/mysql/ /shared/kubernetes/mysql/
```

```
[root@master /]# rsync -avP /var/www/html/ /shared/kubernetes/wordpress/
```

Almacenamiento para Kubernetes

Para dar almacenamiento persistente en Kubernetes, podremos utilizar:

- NFS
- iSCSI
- RBD (Ceph Block Device)
- Glusterfs
- HostPath
- GCEPersistentDisk
- AWSElasticBlockStore

Para cualquiera de ellas en Kubernetes tendremos que crear:

- PersistentVolume -- Donde especificamos el volumen persistente
- PersistentVolumeClaim -- Donde reclamamos espacio en el volumen

Modos de acceso:

- ReadWriteOnce -- read-write solo para un nodo (RWO)
- ReadOnlyMany -- read-only para muchos nodos (ROX)
- ReadWriteMany -- read-write para muchos nodos (RWX)

Políticas de reciclaje de volúmenes son son:

- Retain - Reclamación manual
- Recycle - Reutilizar contenido
- Delete - Borrar contenido

Estados de un volumen:

- Available - disponible para reclamación
- Bound - No disponible, se esta utilizando por una reclamación.
- Released - La reclamación del volumen se a eliminado y esta esperando otra petición del cluster.
- Failed - En estado de fallo.

Utilizaremos el servidor Master (192.168.1.250), para compartir los recursos, a través de NFS /shared/kubernetes:

```
[root@master ~]# mkdir -p /shared/kubernetes/wordpress
```

```
[root@master ~]# mkdir -p /shared/kubernetes/mysql
```

Exportamos a través de NFS en el servidor master los recursos:

```
[root@master kubernetes]# vi /etc/exports  
/shared 172.17.0.0/16(rw,sync,no_root_squash,no_all_squash)  
/shared 192.168.1.0/24(rw,sync,no_root_squash,no_all_squash)
```

En el servidor master y minions arrancamos todos los servicios de NFS:

```
[root@master ~]# systemctl enable rpcbind  
[root@master ~]# systemctl start rpcbind  
[root@master ~]# systemctl enable nfs-server  
Created symlink from /etc/systemd/system/multi-user.target.wants/nfs-server.service to  
/usr/lib/systemd/system/nfs-server.service.  
[root@master ~]# systemctl start nfs-server
```

Desde los servidores minion, podemos comprobar si vemos el servidor NFS:

```
[root@minion1 ~]# showmount -e 192.168.1.250  
Export list for 192.168.1.250:  
/shared 192.168.1.0/24,172.17.0.0/16
```

Deploy Aplicación en Kubernetes. (NO REALIZAR)

Luego de seguir correctamente los pasos y tener nuestro Cluster en funcionamiento es hora de levantar las aplicaciones.

En nuestro caso vamos utilizar de ejemplo de sitio creado por el formador, el mismo esta compuesto por una base de datos MYSQL y WORDPRESS y Joomla.

Copiando los directorios de nuestra app de la DB y WWW (Realizado por el formador).

```
[root@master /]#rsync -avP /var/lib/mysql/ /shared/kubernetes/mysql/
```

```
[root@master /]#rsync -avP /var/www/html/ /shared/kubernetes/wordpress/
```

Luego vamos a otorgar permisos de escritura/lectura chmod 755
`/shared/kubernetes/mysql`

Vamos a realizar el mismo paso con los archivos de WordPress/WWW, copiamos los archivos de nuestro sitio en `/shared/kubernetes/wordpress`

Ahora vamos a otorgar permisos de escritura/lectura

`chmod 755 /shared/kubernetes/ wordpress`

Vamos a proceder a crear nuestros volúmenes de almacenamiento persistente:

```
[root@master storage]# vi mariadb1.yaml
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mariadb1-proyecto
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /shared/kubernetes/mysql
    server: 192.168.1.250
```

En este paso estamos indicando que tenemos un volumen de almacenamiento persistente que tiene 2GB, alojado en un storage nfs, Luego de realizar este paso debemos reclamar espacio de este volumen para poder montarlo en nuestra imagen:

```
[root@master almacenamiento]# vi claim-mariadb1.yaml
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: claim-mariadb1-proyecto
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
```

Ahora ejecutamos los yaml de almacenamiento MYSQL.

```
[root@master almacenamiento]# kubectl create -f claim-mariadb1.yaml
```

```
[root@master almacenamiento]# kubectl create -f claim-mariadb1.yaml
persistentvolumeclaim "claim-mariadb1-proyecto" created
```

Revisamos que nuestro almacenamiento levanto correctamente

```
[root@master almacenamiento]# kubectl get pv,pvc
NAME          CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
CLAIM
pv/mariadb1-proyecto   2Gi        RWX           Retain       Bound
default/claim-mariadb1-proyecto
                           2m

NAME          STATUS     VOLUME          CAPACITY
ACCESSMODES   AGE
pvc/claim-mariadb1-proyecto   Bound   mariadb1-proyecto   2Gi
RWX           1m
```

Creando almacenamiento persistente y lo Reclamamos, para wordpress:

```
[root@master storage]# vi proyecto-var.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
  name: itshell-var
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /shared/kubernetes/wordpress
    server: 192.168.1.250
```

Reclamos el espacio de nuestro volumen creado.

```
[root@master storage]# vi claim-var-proyecto.yaml
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: claim-var-proyecto
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
```

Ahora ejecutamos los yaml de almacenamiento wordpress:

```
[root@master almacenamiento]# kubectl create -f proyecto-var.yaml
```

```
persistentvolume "itshell-var" created
```

```
[root@master almacenamiento]# kubectl create -f claim-var-proyecto.yaml
persistentvolumeclaim "claim-var-proyecto" created
```

Revisamos que nuestros almacenamientos levantaron correctamente:

```
[root@master almacenamiento]# kubectl get pv,pvc
```

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS
CLAIM		REASON	AGE	
pv/itshell-var	2Gi	RWX	Retain	Bound
default/claim-var-proyecto			50s	
pv/mariadb1-proyecto	2Gi	RWX	Retain	Bound
default/claim-mariadb1-proyecto			8m	

NAME	ACCESSMODES	AGE	STATUS	VOLUME	CAPACITY
pvc/claim-mariadb1-proyecto	RWX	7m	Bound	mariadb1-proyecto	2Gi
pvc/claim-var-proyecto	RWX	37s	Bound	itshell-var	2Gi

Replication Controller para MYSQL/MARIADB: En este caso voy a crear solamente una replica de MYSQL ya que si queremos más deberíamos crear un cluster.

```
[root@master almacenamiento]# vi mariadb01-rc.yaml
```

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mariadb01-it
  labels:
    name: mariadb01-it
spec:
  replicas: 1
  selector:
    name: mariadb01-it
  template:
    metadata:
      labels:
        name: mariadb01-it
    spec:
      containers:
        - name: mariadb01-it
          image: mysql:5.6
          ports:
            - containerPort: 3306
              name: mariadb01-it
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "root"
        - name: MYSQL_DATABASE
          value: "000000"
        - name: MYSQL_ROOT_PASSWORD
          value: root
        - name: MYSQL_USER
          value: root
      volumeMounts:
        - name: claim-mariadb1-proyecto
          mountPath: /var/lib/mysql
  volumes:
    - name: claim-mariadb1-proyecto
      persistentVolumeClaim:
        claimName: claim-mariadb1-proyecto
```

```
[root@master almacenamiento]# kubectl create -f mariadb01-rc.yaml
```

```
replicationcontroller "mariadb01-it" created
```

Microservicio de MARIADB/MYSQL – Lo vamos a llamar mariadb01-it:

```
[root@master almacenamiento]# vi mariadb01-svc.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: mariadb01-it
    name: mariadb01-it
spec:
  clusterIP: 10.254.1.70
  ports:
    - port: 3306
  selector:
    name: mariadb01-it
```

```
[root@master almacenamiento]# kubectl create -f mariadb01-svc.yaml
```

Revisamos que levanto correctamente:

```
[root@master almacenamiento]# kubectl get svc
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
db-service	10.254.67.185	<none>	3306/TCP	2d
kubernetes	10.254.0.1	<none>	443/TCP	5d
kubernetes-bootcamp	10.254.215.176	<nodes>	8080:31503/TCP	4d
mariadb01-it	10.254.1.70	<none>	3306/TCP	22h
my-nginx-service	10.254.109.132	<nodes>	80:30992/TCP	3d
webserver-service	10.254.14.120	<none>	80/TCP	2d
wordpress-service	10.254.195.167	<nodes>	80:31046/TCP	21h

Replication Controller de WORDPRESS

NOTA: Cuando llamo al volumen por el claim siempre se debe respetar el nombre de la variable, pero el nombre de configuración de volumen lo podemos cambiar como se representa en este otro yaml.

En este caso vamos levantar 3 replicas de WordPress que van a terminar en diferentes servidores y vamos a publicar el servicio el cual va a tener los ENDPOINTS balanceados

Importante es la variable

value: mariadb01-it

Ya que es la que a través del servicio de DNS de mariadb, va a poder resolver, la dirección ip del cluster, para que el pod de wordpres se pueda conectar a la base de datos:

[root@master almacenamiento]# vi wordpress-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: wordpress
  labels:
    name: wordpress
spec:
  replicas: 1
  selector:
    name: wordpress
  template:
    metadata:
      labels:
        name: wordpress
    spec:
      containers:
        - name: wordpress
          image: localhost:5000/centos7-wordpress
          ports:
            - containerPort: 80
              name: wordpress
      env:
        - name: WORDPRESS_DB_USER
          value: wordpressuser
        - name: WORDPRESS_DB_PASSWORD
          value: "000000"
        - name: WORDPRESS_DB_NAME
          value: wordpress
        - name: WORDPRESS_DB_HOST
          value: mariadb01-it
      volumeMounts:
        - name: wordpress-1
          mountPath: /var/www/html
    volumes:
      - name: wordpress-1
```

```
persistentVolumeClaim:
  claimName: claim-var-proyecto
```

[root@master almacenamiento]# **kubectl create -f wordpress-rc.yaml**

Publicando nuestro servicio de tipo NodePort

[root@master almacenamiento]# **vi wordpress-service.yaml**

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-service
  labels:
    app: wordpress
spec:
  type: NodePort
  ports:
  - port: 80
nodePort: 31903
  protocol: TCP
  name: http
  selector:
    name: wordpress
```

Importante la variable ***nodePort*** del servicio, de esta forma fijamos el puerto que estará abierto en todos los minions, para acceder a los pods, de este servicio y poder balancearlos a través de nuestro balanceador haproxy

[root@master almacenamiento]# **kubectl create -f wordpress-service.yaml**

kubectl get service				
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
db-service	10.254.67.185	<none>	3306/TCP	2d
kubernetes	10.254.0.1	<none>	443/TCP	5d
kubernetes-bootcamp	10.254.215.176	<nodes>	8080:31503/TCP	4d
mariadb01-it	10.254.1.70	<none>	3306/TCP	22h
my-nginx-service	10.254.109.132	<nodes>	80:30992/TCP	3d
webserver-service	10.254.14.120	<none>	80/TCP	2d
wordpress-service	10.254.195.167	<nodes>	80:31046/TCP	21h

Vemos la configuración del service wordpress y podemos acceder a los pod o bien a través de los NodePort de los minions o balanceando a través de haproxy con los Endpoints:

```
[root@master almacenamiento]# kubectl describe service wordpress-service
Name:           wordpress-service
Namespace:      default
Labels:         app=wordpress
Selector:       name=wordpress
Type:          NodePort
IP:            10.254.250.184
Port:          http  80/TCP
NodePort:       http  31903/TCP
Endpoints:     172.17.51.7:80,172.17.60.10:80,172.17.60.8:80
Session Affinity: None
```

Los comandos para realizar troubleshooting son:

kubectl describe po “ID del contenedor” – Lo utilizamos en caso de un contenedor que no se creó correctamente.

kubectl logs “ID del contenedor” – Lo utilizamos en un contenedor creado correctamente y vemos el Log.

```
[root@master /]# kubectl exec -ti busybox nslookup mariadb01-it
```

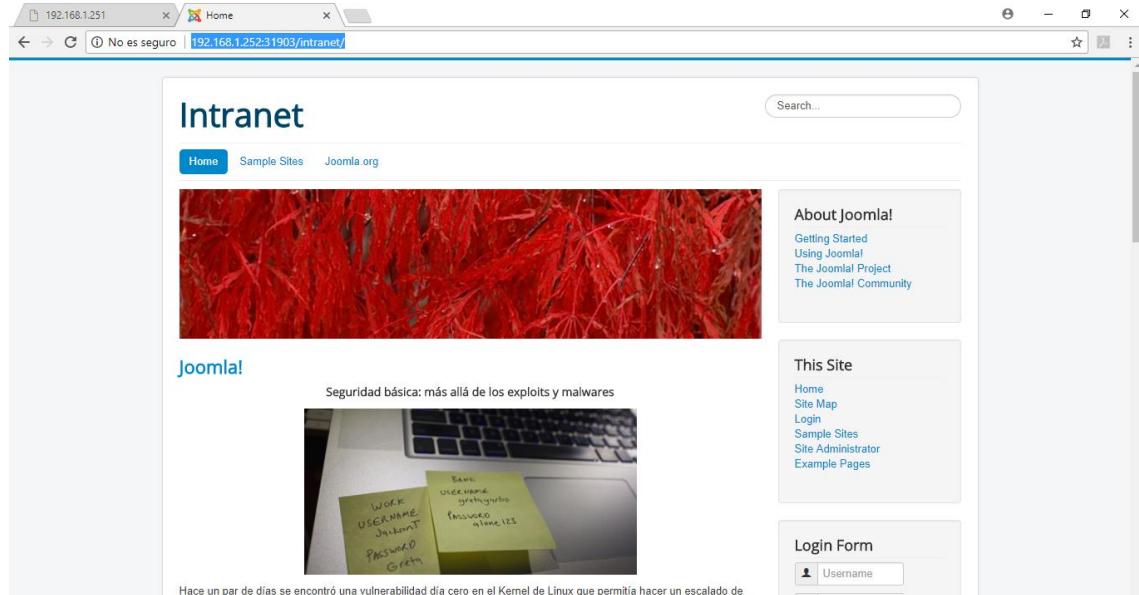
```
Server: 10.254.254.254
Address 1: 10.254.254.254
```

```
Name: mariadb01-it
Address 1: 10.254.1.70
```

Comprobamos el acceso:

<http://192.168.1.251:31903/intranet/>

<http://192.168.1.252:31903/intranet/>



Ahora configuraremos el acceso a través de los endpoints, a través de haproxy:

Endpoints: **172.17.51.7:80,172.17.60.10:80,172.17.60.8:80**

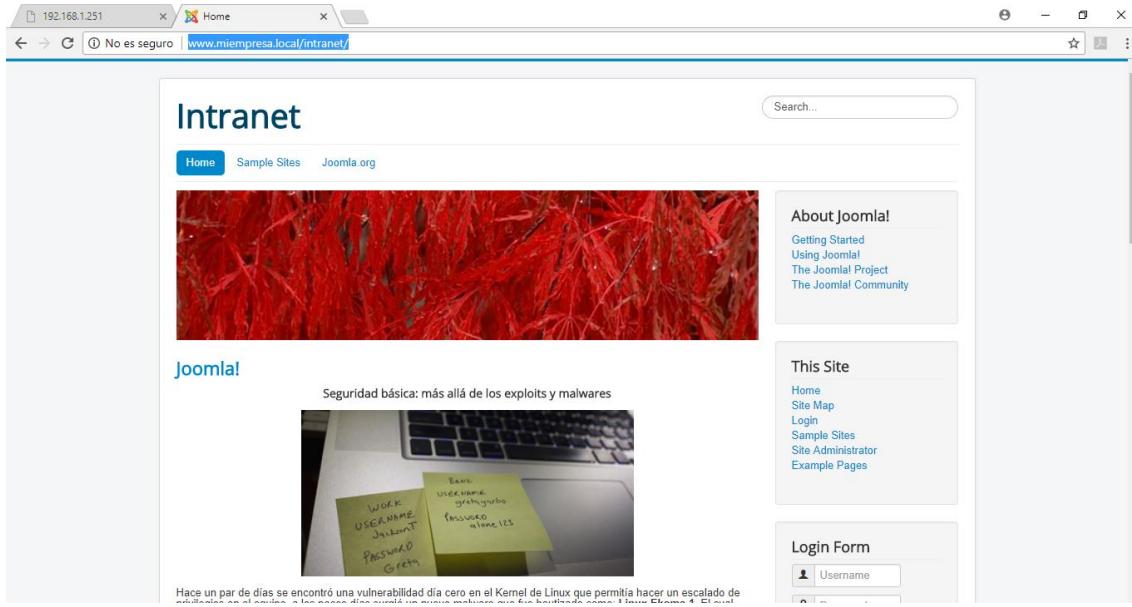
[root@master /]# vi /etc/haproxy/haproxy.cfg

```
backend site2
    balance roundrobin
    option httpclose
    option forwardfor
    server s1 172.17.51.7:80 maxconn 32
    server s2 172.17.60.10:80 maxconn 32
    server s3 172.17.60.8:80 maxconn 32
```

[root@master /]# systemctl restart haproxy.service

[root@master /]# systemctl status haproxy.service

<http://www.miempresa.local/intranet/>



Si quisieramos eliminar todo el proyecto, tendríamos que elminimar los services y rc tanto de mariabd como de wordpress.

Ahora si queremos escalar el replication controles wordpress

```
[root@master /]# kubectl scale rc wordpress --replicas=1
```

```
replicationcontroller "wordpress" scaled
```

```
root@master /]# kubectl describe service wordpress-service
```

```
Name:           wordpress-service
```

```
Namespace:      default
```

```
Labels:         app=wordpress
```

```
Selector:       name=wordpress
```

```
Type:          NodePort
```

```
IP:            10.254.250.184
```

```
Port:          http  80/TCP
```

```
NodePort:       http  31903/TCP
```

```
Endpoints:     172.17.51.7:80
```

```
Session Affinity: None
```

```
[root@master /]# kubectl scale rc wordpress --replicas=4
```

```
[root@master /]# kubectl describe service wordpress-service
```

```
Name:           wordpress-service
Namespace:     default
Labels:         app=wordpress
Selector:       name=wordpress
Type:          NodePort
IP:            10.254.250.184
Port:          http  80/TCP
NodePort:      http  31903/TCP
Endpoints:    172.17.51.7:80,172.17.51.9:80,172.17.60.10:80 + 1 more...
Session Affinity: None
```

Tambien podemos modificar el replication controller de la siguiente forma:

```
[root@master ~]# kubectl edit rc wordpress
```

```
spec:
```

```
replicas: 5
```

Modificamos el número de replicas y automáticamente se crearan el número de replicas que hemos configurado, se aplica los cambios en caliente:

```
[root@master ~]# kubectl get pod -o wide |grep -i wordpress*
```

wordpress-583f1	1/1	Running	1	13h	172.17.11.3	minion1
wordpress-jnxq8	1/1	Running	0	4m	172.17.11.5	minion1
wordpress-lsg0n	1/1	Running	3	2d	172.17.58.3	minion2
wordpress-rl4n4	1/1	Running	0	4m	172.17.11.4	minion1
wordpress-tcr16	1/1	Running	1	13h	172.17.11.2	minion1

Laboratorio kubernetes Namespaces

En este laboratorio configuraremos en kubernetes espacios de nombres y configuraremos asignaciones de CPU y Memoria a contenedores a través de kubernetes.

Los ***namespaces*** (espacios de nombres) en Kubernetes permiten establecer un nivel adicional de separación entre los contenedores que comparten los recursos de un clúster.

Esto es especialmente útil cuando diferentes grupos de trabajo usan el mismo clúster y existe el riesgo potencial de colisión de nombres de los *pods*, etc usados por los diferentes equipos.

Los espacios de nombres también facilitan la creación de cuotas para limitar los recursos disponibles para cada *namespace*. Puedes considerar los espacios de nombres como clústers *virtuales* sobre el clúster físico de Kubernetes. De esta forma, proporcionan separación lógica entre los entornos de diferentes equipos.

Kubernetes proporciona dos *namespaces* por defecto: **kube-system** y **default**.

Los objetos “de usuario” se crean en el espacio de nombres `default`, mientras que los de “sistema” se encuentran en `kube-system`.

Para ver los espacios de nombres en el clúster, ejecuta:

```
[root@master laboratorio5]# kubectl get namespaces
NAME      STATUS  AGE
default   Active  8d
kube-system   Active  8d
```

Para comprobar la separación lógica entre los objetos de diferentes *namespaces*, lista los pods mediante `kubectl get pods`:

```
[root@master laboratorio5]# kubectl get pods
NAME                  READY  STATUS    RESTARTS  AGE
busybox               1/1    Unknown  83       8d
db-controller-4bm52   1/1    Unknown  4        5d
db-controller-kshmj   1/1    Running  0        4m
kubernetes-bootcamp-2100875782-0r7n0 1/1    Running  7        7d
kubernetes-bootcamp-2100875782-17h8g  1/1    Running  0        4m
kubernetes-bootcamp-2100875782-4s1ls  1/1    Unknown  6        7d
```

Analizando al detalle el *pod* mediante kubectl describe pod kubernetes-bootcamp-2100875782-0r7n0, observa como se encuentra en el espacio de nombres default:

```
[root@master laboratorio5]# kubectl describe pod kubernetes-bootcamp-2100875782-0r7n0
```

```
Name:      kubernetes-bootcamp-2100875782-0r7n0
```

```
Namespace:  default
```

Tambien podemos utilizar el comando:

```
[root@master laboratorio5]# kubectl get pods --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	busybox	1/1	Unknown	83	8d
default	db-controller-4bm52	1/1	Unknown	4	5d
default	db-controller-kshmj	1/1	Running	0	7m
default	kubernetes-bootcamp-2100875782-0r7n0	1/1	Running	7	7d
default	kubernetes-bootcamp-2100875782-17h8g	1/1	Running	0	7m
default	kubernetes-bootcamp-2100875782-4s1ls	1/1	Unknown	6	7d
default	mariadb01-it-ln8md	1/1	Running	4	2d
default	my-nginx	1/1	Unknown	5	6d
default	my-nginx-4jjn0	1/1	Running	6	6d
default	my-nginx-6v0bs	1/1	Running	0	7m
default	webserver-controller-2sphp	1/1	Unknown	4	5d
default	webserver-controller-dt26f	1/1	Running	0	7m
default	wordpress-lsg0n	1/1	Running	0	7m
default	wordpress-q0z2v	1/1	Running	3	2d
default	wordpress-q32nj	1/1	Unknown	2	2d
default	wordpress-sgm9b	1/1	Unknown	2	2d
default	wordpress-t2jhr	1/1	Running	3	2d
default	wordpress-x8vbb	1/1	Running	0	7m
kube-system	elastickube-mongo-njztt	1/1	Running	3	2d
kube-system	elastickube-server-rkk8l	4/4	Running	12	2d
kube-system	heapster-v1.2.0-4001981223-bd40k	0/4	Pending	0	7m

```

kube-system  heapster-v1.2.0-4001981223-t3c27    4/4    Unknown  28     8d
kube-system  kube-dns-v11-bh5xl        4/4    Running  33     8d
kube-system  kubernetes-dashboard-3543765157-10ph7  1/1    Unknown  7      8d
kube-system  kubernetes-dashboard-3543765157-kz64v  1/1    Running  0      7m
kube-system  monitoring-influxdb-grafana-v4-qnbcl  2/2    Running  16     8d

```

Crea un nuevo espacio de nombres

Para crear un *namespace*, crea un fichero YAML llamado **ns-preproduccion.yaml** como el siguiente:

```

apiVersion: v1
kind: Namespace
metadata:
  name: preproduccion

```

```
[root@master laboratorio5]# kubectl create -f ns-preproduccion.yaml
```

```
namespace "preproduccion" created
```

Al obtener la lista de espacios de nombres disponibles, observa que ahora el nuevo *namespace* aparece:

```
[root@master laboratorio5]# kubectl get ns
NAME      STATUS  AGE
default   Active  8d
kube-system  Active  8d
preproduccion  Active  49s
```

Observa con detalle el *namespace* creado:

```
[root@master laboratorio5]# kubectl describe ns preproduccion
```

Name: preproduccion

Labels: <none>

Status: Active

No resource quota.

No resource limits.

El particionamiento del clúster en espacios de nombres permite repartir los recursos del clúster imponiendo cuotas, de manera que los objetos de un determinado *namespace* no acaparen todos los recursos disponibles. ([Apply Resource Quotas and Limits](#)).

Aplicando quotas al número de objetos en el *namespace*

Para aplicar una cuota, creamos un fichero YAML del tipo ResourceQuota:

```
[root@master laboratorio5]# vi quota-object-counts.yaml
```

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    persistentvolumeclaims: "2"
    services.loadbalancers: "2"
    services.nodeports: "0"
```

Esta cuota limita el número de:

- volúmenes persistentes (2)
- balanceadores de carga (2)
- *node ports* (0)

Para crear la cuota, aplica el fichero YAML

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

```
resourcequota "object-counts" created
```

```
# kubectl describe ns preproduccion
```

Name: preproduccion

Labels: <none>

Status: Active

Resource Quotas

Name:	object-counts
-------	---------------

Resource	Used	Hard
----------	------	------

-----	---	--
persistentvolumeclaims	0	2
services.loadbalancers	0	2
services.nodeports	0	0

Esta cuota impide la creación de más objetos de cada tipo de los especificados en la cuota (es decir, como máximo, puede haber dos *load balancers* en el espacio de nombres preproducción).

Aplicando cuotas a los recursos del *namespace*

Habitualmente los límites que se suelen establecer para cada espacio de nombres están enfocados a limitar los recursos de CPU y memoria del *namespace*.

El siguiente fichero YAML especifica un límite de 2 CPUs y 2GB de memoria. Además, especifica una limitación en cuanto a las peticiones que debe realizar un *pod* en este espacio de nombres. Finalmente, también se establece una limitación de como máximo, 4 *pods*.

vi quota-compute-resources.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.cpu: "1"
    requests.memory: 1Gi
    pods: "4"
```

Tendremos que especificar el *namespace* al que aplicar la cuota.

Aplicamos la nueva cuota mediante:

```
[root@master laboratorio5]# kubectl create -f quota-compute-resources.yaml --namespace preproduccion
```

```
resourcequota "compute-resources" created
```

El espacio de nombres está limitado ahora de la siguiente manera:

kubectl describe ns preproduccion

```
Name: preproduccion
```

```
Labels: <none>
```

```
Status: Active
```

Resource Quotas

Name:	compute-resources
-------	-------------------

Resource	Used	Hard
----------	------	------

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

limits.cpu	0	2
-------------------	----------	----------

```

limits.memory      0    2Gi
pods              0    4
requests.cpu      0    1
requests.memory   0    1Gi

```

Name:	object-counts	
Resource	Used	Hard
---	--	--
persistentvolumeclaims	0	2
services.loadbalancers	0	2
services.nodeports	0	0

La limitación impuesta en las peticiones (`requests`) de memoria y CPU **obligan a que se especifiquen límites en la definición de los recursos asignados a cada pod**. En general, al crear la definición de un *deployment* no se especifican estos límites, lo que puede provocar algo de desconcierto.

Vamos a crear un *Deployment* en el *namespace* `preproduccion`. Aunque asignamos el *deployment* al *namespace* desde la línea de comando, en un fichero `YAML` usaríamos:

```

apiVersion: v1
kind: Service
metadata:
  name: ejemplo
  namespace: preproduccion
spec:
  ...

```

Creamos un *deployment*:

```
# kubectl run nginx --image=nginx --replicas=1 --namespace=preproduccion
deployment "nginx" created
```

Todo parece ok hasta que buscamos el *pod* que debería crearse:

```
# kubectl get pods --namespace preproduccion
```

No resources found.

Analizamos el detalle del *deployment*:

```
# kubectl get pods --namespace preproduccion
```

No resources found.

```
# kubectl describe deployment nginx --namespace preproduccion
```

Name: nginx

Namespace: preproduccion

CreationTimestamp: Fri, 25 Aug 2017 11:03:02 +0200

Labels: run=nginx

Selector: run=nginx

Replicas: 0 updated | 1 total | 0 available | 1 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge

Conditions:

Type	Status	Reason
------	--------	--------

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

Available	True	MinimumReplicasAvailable
-----------	------	--------------------------

ReplicaFailure	True	FailedCreate
----------------	------	--------------

OldReplicaSets: <none>

NewReplicaSet: nginx-701339712 (0/1 replicas created)

Events:

FirstSeen	LastSeen	Count	From	SubObjectPath	Type	Reason
-----------	----------	-------	------	---------------	------	--------

Message	-----	-----	-----	-----	-----	-----
---------	-------	-------	-------	-------	-------	-------

1m	1m	1	{deployment-controller }	Normal	ScalingReplicaSet
----	----	---	--------------------------	--------	-------------------

Scaled up replica set nginx-701339712 to 1

No se ha creado el *ReplicaSet*. Vamos a ver porqué:

```
# kubectl describe rs nginx-701339712 --namespace preproduccion

Name:           nginx-701339712
Namespace:      preproduccion
Image(s):       nginx
Selector:       pod-template-hash=701339712,run=nginx
Labels:         pod-template-hash=701339712
                run=nginx
Replicas:      0 current / 1 desired
Pods Status:   0 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
FirstSeen     LastSeen      Count  From            Message
SubObjectPath Type          Reason           Message
-----        -----        ----  ----
-----        -----        ----  -----
3m           41s           16    {replicaset-controller} 
Warning       FailedCreate   Error creating: pods "nginx-701339712-
" is forbidden: failed quota: compute-resources: must specify
limits.cpu,limits.memory,requests.cpu,requests.memory
```

El *deployment* crea un *ReplicaSet*, que a su vez intenta crear uno o más *pods*. Como en el *Deployment* no se ha especificado un límite para la CPU y memoria del *pod* y lo hemos exigido en las cuotas impuestas al *namespace*, la creación del *pod* falla. El mensaje de error es claro:

```
Error creating: pods "nginx-701339712-" is forbidden: failed quota:
compute-resources: specify
limits.cpu,limits.memory,requests.cpu,requests.memory
```

Si creamos el *pod* especificando los límites:

```
# kubectl delete deployment nginx --namespace preproduccion
deployment "nginx" deleted
```

```
[root@master laboratorio5]# kubectl run nginx \
--image=nginx \
--replicas=1 \
--requests(cpu=100m,memory=256Mi) \
--limits(cpu=200m,memory=512Mi) \
--namespace=preproducción
```

Ahora, al revisar el *namespace*:

```
# kubectl describe ns preproduccion
```

Name: preproduccion

Labels: <none>

Status: Active

Resource Quotas

Name: compute-resources

Resource	Used	Hard
----------	------	------

limits.cpu	200m	2
------------	------	---

limits.memory	512Mi	2Gi
---------------	-------	-----

pods	1	4
------	---	---

requests.cpu	100m	1
--------------	------	---

requests.memory	256Mi	1Gi
-----------------	-------	-----

Name: object-counts

Resource	Used	Hard
----------	------	------

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

persistentvolumeclaims	0	2
------------------------	---	---

services.loadbalancers	0	2
------------------------	---	---

services.nodeports	0	0
--------------------	---	---

```
# kubectl get pod,rc,ns,deployment,service --namespace preproduccion
```

NAME	READY	STATUS	RESTARTS	AGE
po/nginx-2953480023-wbv0k	0/1	Pending	0	3m

NAME	STATUS	AGE
ns/default	Active	8d
ns/kube-system	Active	8d
ns/preproduccion	Active	38m

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/nginx	1	1	0	3m	

Lab 2 Namespaces

<https://kubernetes.io/es/docs/concepts/overview/working-with-objects/namespaces/>

Los Namespaces son un concepto muy útil para crear particiones lógicas dentro de mi clúster, o lo que también podemos llamar clúster virtuales dentro de mi clúster físico de Kubernetes.

Es importante recordar que para cada tipo de objeto de Kubernetes, por ejemplo un Service o un Deployment, el nombre del objeto debe ser único. Pero dentro de un clúster podemos crear distintos Namespaces, y esta restricción de unicidad aplica solo dentro de cada Namespace. Así, por ejemplo, puedo tener un Namespace *staging* y otro *production*, y seré capaz de crear un Deployment *web* tanto en uno como en otro Namespace.

También podríamos crear un namespace por cada desarrollador, para que trabajen de forma aislada en cada namespace

Otro uso muy útil de los Namespaces es para gestionar equipos, ya que cada desarrollador podría tener acceso a un Namespace distinto, y de esta manera compartir los recursos del clúster sin que el trabajo de un desarrollador afecte al de otro miembro del equipo porque tienen aislamiento a nivel de Namespace.

Por defecto, este aislamiento es solo a nivel de los nombres de los objetos que vamos a crear (y por ejemplo se comparte la misma red), pueden comunicarse entre distintos namespaces, pero podemos hacer uso de Pod Policies y de Network Policies para limitar estos accesos.

Por último, podemos limitar los recursos máximos que se pueden consumir dentro de un Namespace. De esta forma podemos asegurarnos de que ningún desarrollador consume más de una cierta CPU o memoria en su Namespace de desarrollo.

Para ver todos los namespaces:

```
$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	17m
kube-public	Active	17m
kube-system	Active	17m

Para crear un namespace

```
$ kubectl create namespace test
```

```
namespace/test created
```

Para trabajar con un namespace utilizamos el -n nombre del namespace con kubectl

```
$ kubectl -n test create deployment.yaml
```

```
$ kubectl -n test get all
```

Para ver el contexto por defecto podemos ver que el namespace por defecto es default:

```
$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	kubernetes-admin@kubernetes	kubernetes	kubernetes-admin	default

Cambiamos el namespace por defecto a test y lo comprobamos:

```
vagrant@master:/$ kubectl config set-context kubernetes-admin@kubernetes --namespace=test
```

```
Context "kubernetes-admin@kubernetes" modified.
```

```
vagrant@master:/$ kubectl get all
```

Dejamos el contexto nuevamente a default

```
$ kubectl config set-context kubernetes-admin@kubernetes --namespace=default
$ kubectl config get-contexts
CURRENT NAME          CLUSTER AUTHINFO      NAMESPACE
*   kubernetes-admin@kubernetes  kubernetes  kubernetes-admin  default
```

Namespace y DNS

Cuando se crea un *service*, se crea la correspondiente entrada en el DNS. Esta entrada es de la forma <nombre-servicio>.<espacio-de-nombres>.svc.cluster.local, lo que significa que si un contenedor usa únicamente <nombre-de-servicio>, la resolución del nombre se realizará de forma local en el espacio de nombres en el que se encuentre.

Esta configuración permite usar la misma configuración entre diferentes espacios de nombres (por ejemplo *Desarrollo*, *Integración* y *Producción*).

Para que un contenedor pueda resolver el nombre de otro contenedor en otro *namespace*, debes usar el FQDN.

Borrando un *namespace*

IMPORTANTE:

Al borrar un *namespace* se borran **todos los objetos** del espacio de nombres.

Para borrar un *namespace*, usa el comando **delete**:

```
# kubectl delete ns preproduccion
```

```
namespace "preproduccion" deleted
```

```
# kubectl get ns
```

NAME	STATUS	AGE
default	Active	8d
kube-system	Active	8d

Asignar recursos de CPU y RAM a un contenedor POD

Cuando se crea un *pod* se pueden reservar recursos de CPU y RAM para los contenedores que corren en el *pod*. Para reservar recursos, usa el campo `resources: requests` en el fichero de configuración. Para establecer límites, usa el campo `resources: limits`.

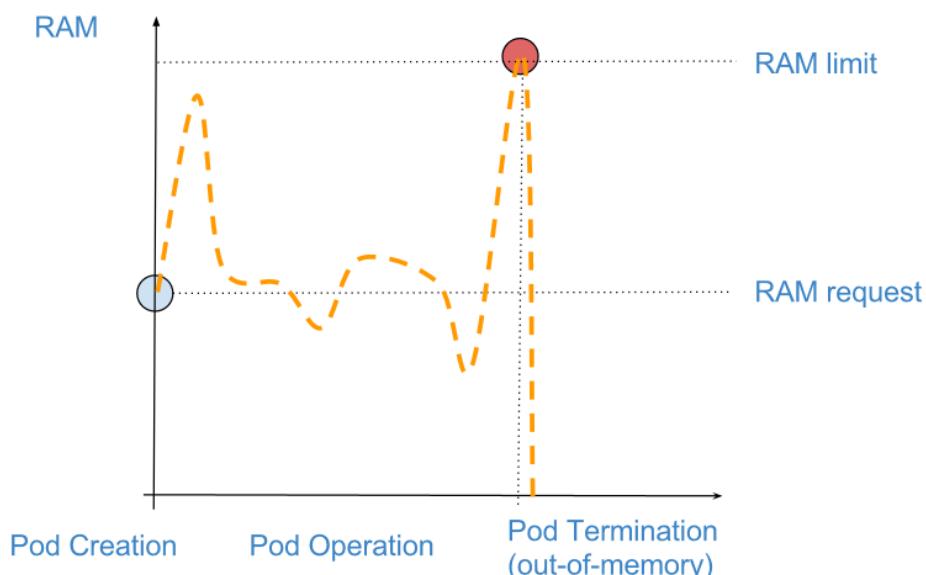
Kubernetes planifica un *pod* en un nodo sólo si el nodo tiene suficientes recursos de CPU y RAM disponibles para satisfacer la demanda de CPU y RAM total de todos los contenedores en el *pod*. Es decir, la *request* es la cantidad que necesita el *pod* para arrancar y ponerse en funcionamiento.

En función de las tareas que ejecute el *pod*, los recursos que consume pueden aumentar.

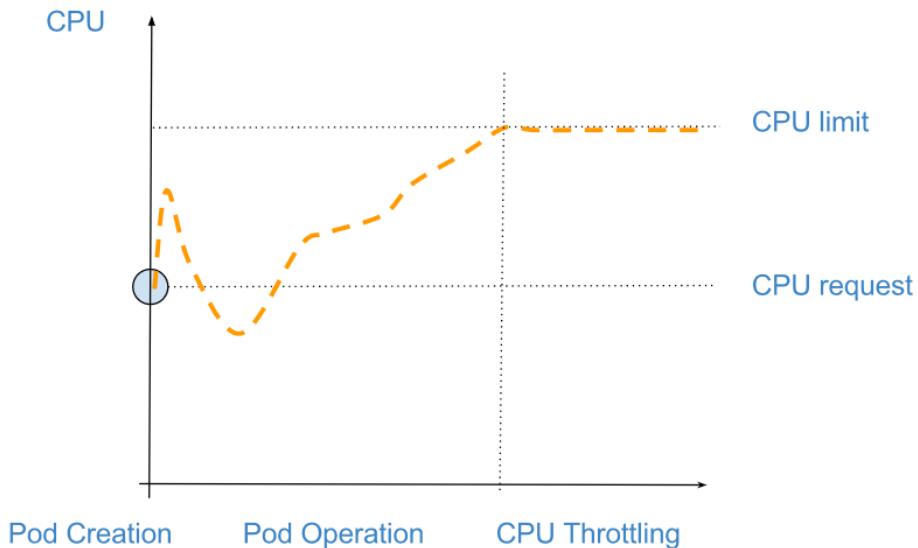
Mediante el establecimiento de los *limits* podemos acotar el uso máximo de recursos disponible para el *pod*.

Kubernetes no permite que el *pod* consuma más recursos de CPU y RAM de los límites especificados para en el fichero de configuración.

Si un contenedor excede el límite de RAM, es eliminado.



Si un contenedor excede el límite de CPU, se convierte en un candidato para que su uso de CPU se vea restringido (*throttling*) .



Unidades de CPU y RAM

Los recursos de CPU se miden en **cpus**. Se admiten valores fraccionados. Puedes usar el sufijo *m* para indicar “mili”; por ejemplo, 100m cpu son 100 milicpu o 0.1 cpu.

Los recursos de RAM se miden en **bytes**. Puedes indicar la RAM como un entero usando alguno de los siguientes sufijos: E, P, T, G, M, Ei, Pi,Ti, Gi, Mi y Ki. Por ejemplo, las siguientes cantidades representan aproximadamente el mismo valor:

128974848, 129e6, 129M , 123Mi

Si no conoces por adelantado los recursos que reservar para un *pod* puedes lanzar la aplicación sin especificar límites, usar el monitor de uso de recursos y determinar los valores apropiados.

Si un contenedor excede los límites establecidos de RAM, se elimina al quedarse sin memoria disponible: *out-of-memory*. Debes especificar un valor ligeramente superior al valor esperado para dar un poco de margen al *pod*.

Si especificas una reserva (*request*), el *pod* tendrá garantizado disponer de la cantidad reservada del recurso. El *pod* puede usar más recursos que los reservados, pero nunca más del límite establecido.

En el siguiente laboratorio especificamos tanto una reserva como el límite de recursos de los que puede disponer un *pod*:

```
[root@master laboratorio5]# vi cpu-ram-demo-container.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-ram-demo
  labels:
    name: cpu-ram-demo
spec:
  containers:
  - name: cpu-ram-demo-container
    image: gcr.io/google-samples/node-hello:1.0
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "1"
```

```
[root@master laboratorio5]# kubectl create -f cpu-ram-demo-container.yaml
```

```
pod "cpu-ram-demo" created
```

El *pod* reserva 64Mi de RAM y 0.25 cpus, pero puede llegar a usar hasta el doble de RAM y toda una CPU.

Comprobamos que sucede en nuestro entorno:

```
[root@master laboratorio5]# kubectl describe pod cpu-ram-demo
```

```
Name:      cpu-ram-demo
Namespace: default
Node:      /
Labels:    <none>
Status:   Pending
IP:
Controllers: <none>
Containers:
  cpu-ram-demo-container:
    Image:  gcr.io/google-samples/node-hello:1.0
    Port:
```

Limits:

```
cpu:    1
memory: 128Mi
```

Requests:

```
cpu:    250m
memory: 64Mi
```

Volume Mounts:

```
/var/run/secrets/kubernetes.io/serviceaccount from default-token-f95v2 (ro)
```

Environment Variables: <none>

Conditions:

Type	Status
------	--------

PodScheduled	False
--------------	-------

Volumes:

default-token-f95v2:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-f95v2

QoS Class: Burstable

Tolerations: <none>

Events:

FirstSeen	LastSeen	Count	From	SubObjectPath	Type	Reason
Message						
-----	-----	-----	-----	-----	-----	-----
2m	18s	14	{default-scheduler }		Warning	FailedScheduling

ram-demo) failed to fit in any node

fit failure summary on nodes : Insufficient cpu (1)

Para solucionar el problema y exponer el contenedor al exterior:

```
# vi cpu-ram-demo-container.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-ram-demo
labels:
  name: cpu-ram-demo
  app: cpu-ram-demo
spec:
  containers:
    - name: cpu-ram-demo-container
      image: gcr.io/google-samples/node-hello:1.0
      resources:
        requests:
          memory: "64Mi"
        limits:
          memory: "128Mi"
```

```
# kubectl create -f cpu-ram-demo-container.yaml
```

Ahora exponemos el servicio:

```
# vi cpu-ram-demo-service.yaml

apiVersion: v1
kind: Service
metadata:
  name: cpu-ram-demo-service
labels:
  app: cpu-ram-demo
spec:
  type: NodePort
  ports:
    - port: 8080
      protocol: TCP
      name: http
  selector:
    name: cpu-ram-demo
```

kubectl create -f cpu-ram-demo-service.yaml

kubectl describe service cpu-ram-demo-service

```
Name:           cpu-ram-demo-service
Namespace:     default
Labels:        app=cpu-ram-demo
Selector:      name=cpu-ram-demo
Type:          NodePort
IP:            10.254.254.205
Port:          http  8080/TCP
NodePort:      http  31485/TCP
Endpoints:    172.17.14.18:8080
Session Affinity: None
```

```
# curl 172.17.14.18:8080
```

Hello Kubernetes!

En este caso solo tenemos un pod y esta en el nodo

<http://192.168.1.252:31485/>



Si no especificas reservas o límites

Si no especificas un límite para la RAM, Kubernetes no restinge la cantidad de RAM que puede usar el contenedor. En esta situación un contenedor puede usar toda la memoria disponible en el nodo donde se está ejecutando. Del mismo modo, si no se especifica un límite máximo de CPU, un contenedor puede usar toda la capacidad de CPU del nodo.

Los límites por defecto se aplican en función de la disponibilidad de recursos aplicados al espacio de nombres en el que se ejecutan los *pods*.

Puedes consultar los límites mediante:

```
kubectl describe limitrange limits.
```

Es importante tener en cuenta que si se especifican límites a nivel de *namespace*, la creación de objetos en el *namespace* debe incluir también los límites o se producirán errores al crear objetos (a no ser que se hayan especificado límites por defecto).

En [Set Pod CPU and Memory Limits](#) se indica cómo establecer límites superiores e inferiores para los recursos de un *pod*. También se pueden especificar límites por defecto para los *pods* aunque el usuario no los haya especificado en el fichero de configuración.

Los límites establecidos en el *namespace* se aplican durante la creación o modificación de los *pods*. Si cambias el rango de recursos permitidos, no afecta a los *pods* creados previamente en el espacio de nombres.

Se pueden establecer límites en los recursos consumidos por diferentes motivos, pero normalmente se limitan para evitar problemas *a posteriori*. Por ejemplo, si un nodo tiene 2GB de RAM, evitando la creación de *pods* que requieran más memoria previene que el *pod* no pueda desplegarse nunca (al no disponer de memoria suficiente disponible), por lo que es mejor evitar directamente su creación.

El otro motivo habitual para imponer límites es para distribuir los recursos del nodo entre los diferentes equipos/entornos; por ejemplo, asignando un 25% de la capacidad al equipo de desarrollo y el resto a los servicios en producción.

Actualización

En el caso de que se establezca sólo una de las dos opciones (es decir, sólo *request* o sólo límites), Kubernetes actúa de la siguiente manera:

- Si sólo se establecen límites, Kubernetes establece una reserva (*request*) **igual** al límite.
- Si sólo se establece una reserva, no hay un límite definido, por lo que el *pod* puede llegar a consumir el total de la memoria/CPU disponible en el nodo.

Ingress

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

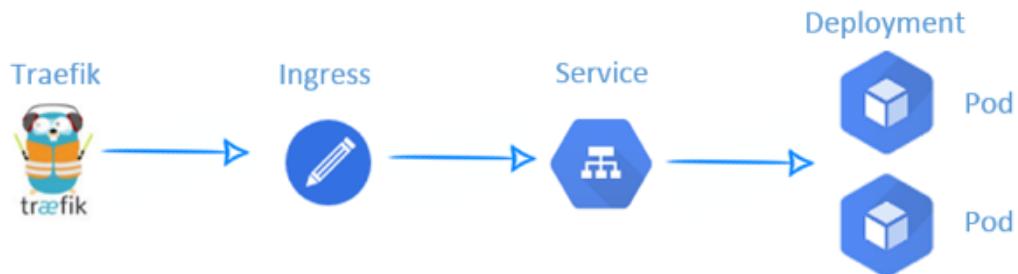
Cuando tenemos que exponer nuestros servicios a tráfico externo hemos visto que podemos crear un servicio de tipo Load Balancer, pero esta solución puede resultar demasiado rígida y costosa. La alternativa es utilizar un Ingress.

Un Ingress nos permite definir rutas de entrada a nuestro servicio de una manera programática. Existen numerosos Ingress Controllers, como el de Nginx, HAProxy o traefik, cada uno de ellos permitiendo distintas configuraciones.

Por ejemplo, con el Nginx Ingress Controller, podemos definir a dónde redirigir una petición en base al dominio solicitado, o al `path` solicitado dentro de esa petición.

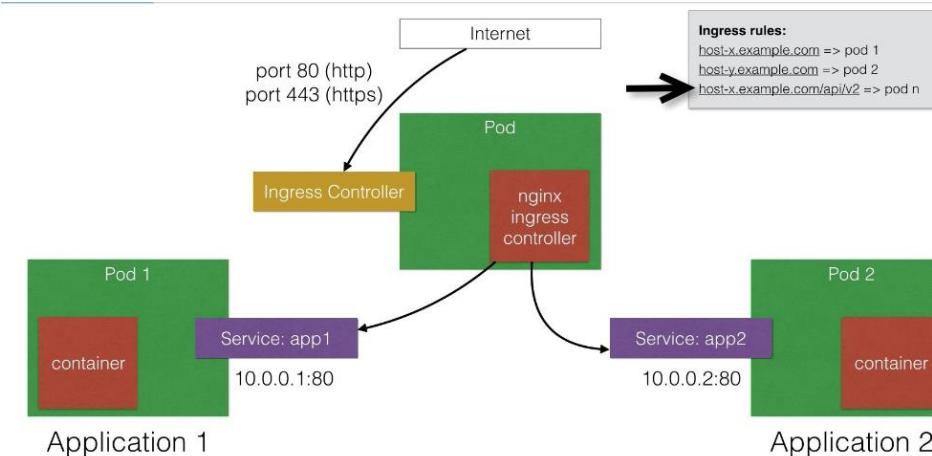
En el laboratorio veremos como redireccionar el tráfico de entrada a mi clúster en función de la cabecera `Host` de la petición entrante.

Hay aplicaciones que dependen mucho de la ruta donde se expongan, entonces como programadores o administradores tenemos que saber en qué rutas o dominios se expone su aplicación.



INGRESS

- ▶ Permite conexiones entrantes al cluster.
- ▶ Funciona con servicios tipo ClusterIP.
- ▶ Ingress Controller.



En el esquema vemos que tenemos nuestro ingres controller con nginx, y que se expone a través del puerto 80 y 443 y en este ingres controller, le vamos a crear a través de la API de kubernetes Ingress rules, en este caso le estamos diciendo el host-x vaya al pod1, el host-y vaya al pod 2 y el host-x pero con esa ruta vaya a el pod n, estas reglas se puedan definir de forma programática para que balance ente unos pods y otros.

El ingres es otra alternativa, para acceder a mis servicios y es muy interesante porque es muy programático, es una forma nativa de kubernetes definir las rutas, para hacer balanceo a la ruta y a la URL, y luego es mas barato en principio nuestro ingres controller es un servicio que desplegamos en el cluster de kubernetes.

Hay muchos tipos, nginx, traefik, que se expone con un servicio de LoadBalancer pero este LoadBalancer se va a compartir entre todas las aplicaciones a través de nuestro ingress controller (en este caso traefick), con esto tambien conseguimos ahorrar dinero de LoadBalancer de nuestro proveedor de cloud.

Para realizar los laboratorios tenemos los archivos en, tendremos que explicar que es lo que hacer estos archivos.:

C:\kubernetes-vagrant-cluster\k8-for-devs-master\ingress

Lo que buscamos es realizar dos deployments, hola y adios, y hacerlo pasar a través del ingress controller, con las url, que estarán descritas en el archivo hosts de nuestro Windows, o en nuestro servidor dns, para poder resolver.

El archivo **nginx-ingress-controller.yml**, es el ingress controller de nginx, **no instalar** en caso de tener instalado traefick

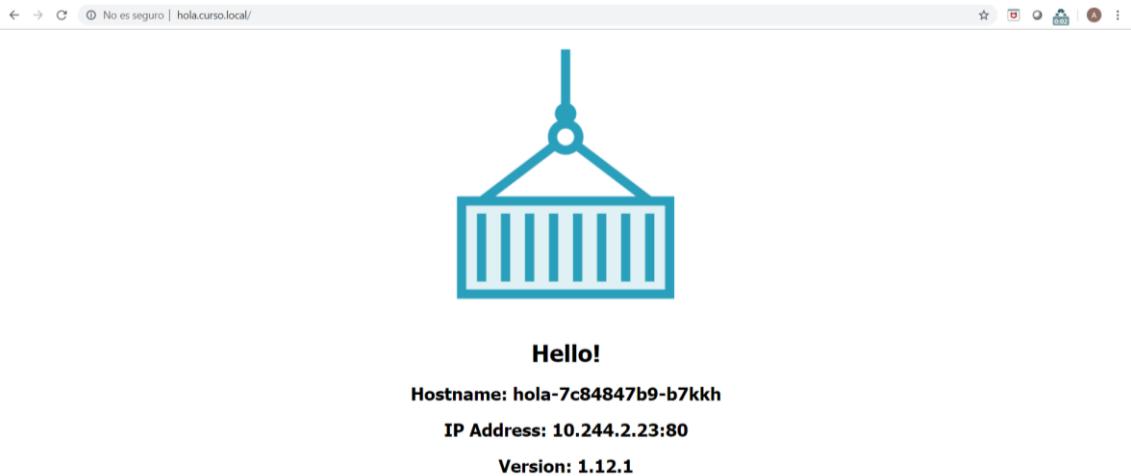
Ahora damos de alta en el fichero hosts de Windows:

C:\Windows\System32\drivers\etc\hosts

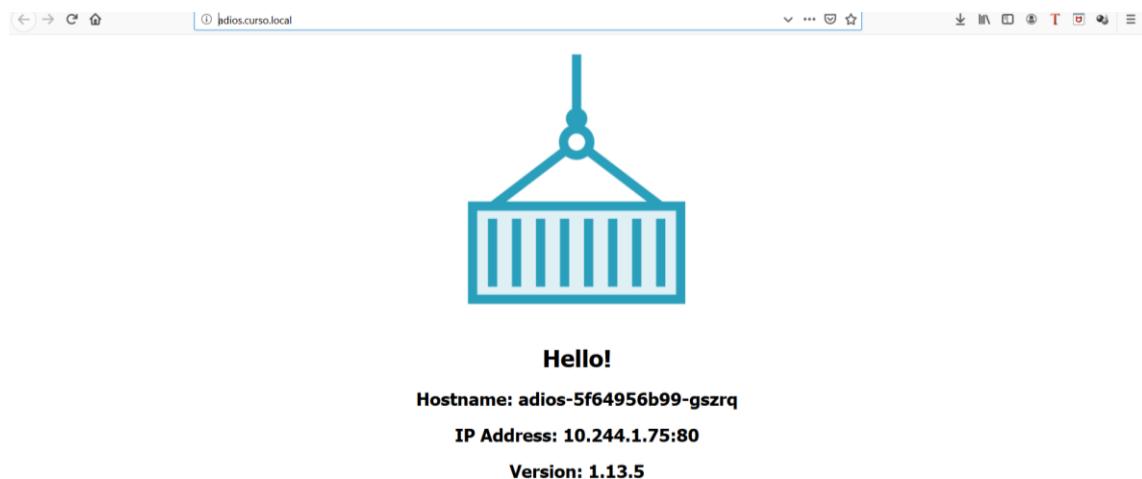
10.0.0.11 hola.curso.local

10.0.0.12 adios.curso.local

hola.curso.local



adios.curso.local



<http://10.0.0.11:8080/dashboard/>

<http://10.0.0.12:8080/dashboard/>

The screenshot shows the Kubernetes dashboard interface. At the top, there's a navigation bar with icons for back, forward, search, and other dashboard functions. The URL in the address bar is 10.0.0.11:8080/dashboard/. Below the navigation is a header with 'PROVIDERS' and 'HEALTH' tabs, and a version indicator 'V1.7.13 / MAROILLES DOCUMENTATION'.

The main content area is titled 'kubernetes'. It displays two sections: 'FRONTENDS' and 'BACKENDS'.

FRONTENDS:

- adios.curso.local/**:
 - Main tab (selected)
 - Details tab
 - Route Rule:
 - PathPrefix: /
 - Host: adios.curso.local
 - Entry Points:
 - http (selected)
 - https
 - Backend: adios.curso.local/
- hola.curso.local/**:
 - Main tab (selected)
 - Details tab
 - Route Rule:
 - PathPrefix: /
 - Host: hola.curso.local
 - Entry Points:
 - http (selected)
 - https
 - Backend: hola.curso.local/

BACKENDS:

- adios.curso.local/**:

Server	Weight
http://10.244.1.75:80	1
http://10.244.2.21:80	1
http://10.244.2.22:80	1
- hola.curso.local/**:

Server	Weight
http://10.244.1.77:80	1
http://10.244.2.23:80	1
http://10.244.1.76:80	1

Laboratorio repaso conceptual kubernetes

Despliegue de una aplicación de dos niveles

En este laboratorio realizaremos un práctica completa para desplegar y ofrecer dos aplicaciones que interactúen entre ellas.

Antes de comenzar a explorar las redes en Kubernetes, debe implementar una aplicación, aún mejor, dos.

Primero, implementará [Redis](#), un proyecto de estructura de datos en memoria de código abierto que a menudo se usa como un almacén de valor-clave.

A continuación, desplegará [Redis Commander](#), una interfaz de usuario para interactuar con Redis.

La aplicación de dos niveles es una excelente excusa para discutir:

- Cómo diseñar aplicaciones que abarcan múltiples implementaciones
- Cómo aprovechar el descubrimiento de servicios en Kubernetes
- Cómo se diseña la red en Kubernetes

Tenga en cuenta que `kubectl` solo está disponible en el nodo maestro.

Puede conectarse al nodo maestro con:

```
bash
vagrant ssh master
```

Despliegue de Redis

Debe crear un Deployment y un Servicio para Redis, **deben de compartir el mismo manifiesto la declaración del deployment y del service**.

Creación de un Deployment

- Crea un Deployment para Redis usando la imagen. `redis:4-alpine`
- El contenedor expone el puerto 6379.
- El despliegue debe tener una réplica.
- Debes asignar la etiqueta `app: redis` a los Pods.

Creando un servicio

- El Servicio expone el puerto 6379 como un ClusterIP
- El selector del Servicio apunta a `app: redis`
- El nombre del servicio debe ser `redis`

Probando el despliegue

Debes verificar que hay un pod corriendo con:

```
vagarnt@maester  
kubectl get pods
```

Debes conectarte al pod de correr con:

```
kubectl exec -ti <replace with Pod name> sh
```

Y verifica que puedas conectarte a la instancia de Redis:

```
sh  
redis-cli -h redis
```

Puedes listar inspeccionar el estado de Redis con:

```
sh  
info
```

Despliegue de la interfaz de administración

Debe crear una Implementación para [joeferner/redis-commander](#) y conectarla a Redis.

Creando el Despliegue

Crea un archivo `deployment-redis-commander.yaml` con el siguiente contenido:

```
deploy-redis-commander.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-commander
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: redis-commander
    spec:
      containers:
        - name: web
          image: rediscommander/redis-commander
          imagePullPolicy: IfNotPresent
        ports:
          - containerPort: 8081
        env:
          - name: REDIS_HOSTS
            value: local:redis:6379
```

Puedes crear el recurso con:

```
vagrant@master
kubectl create -f deployment-redis-commander.yaml
```

Creando el servicio

Debe crear un servicio para exponer el deployment.

Crea un archivo `service-redis-commander.yaml` con el siguiente contenido:

```
service-redis-commander.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    name: redis-commander
    name: redis-commander
spec:
  ports:
  - port: 8080
    targetPort: 8081
  selector:
    name: redis-commander
```

Puedes crear el recurso con:

```
vagrant@master
kubectl create -f service-redis-commander.yaml
```

Probando el despliegue

Debes verificar que Kubernetes haya creado los Pods correctamente con:

```
vagrant@master
kubectl get pods
```

Debe revisar ambos servicios con:

```
vagrant@master
kubectl get service
```

Puede reenviar el tráfico del servicio a un puerto local dentro del nodo maestro con:

```
vagrant@master
kubectl port-forward service/redis-commander 8080:8080 &
```

Puedes verificar que la interfaz de administración sirve una página HTML con:

```
vagrant@master
curl localhost:8080
```

No te olvides de matar el proceso en background:

```
vagrant@master
killall kubectl
```

Exploring the Endpoints

En Kubernetes hay un tipo de objeto llamado Endpoint.

Un objeto Endpoint es una colección de endpoints.

Los Single endpoints (no los objetos) están formados por dos partes: una dirección IP y un puerto.

Un endpoint se crea (o actualiza) cada vez que se agrega o elimina un Pod en un servicio.

Listing Endpoints

Puede recuperar todos los objetos de endpoint en su cluster con:

```
vagrant@master
kubectl get endpoints
```

El endpoint controller, que forma parte del controller manager, se encarga de monitorear los endpoints.

Cuando creas un servicio, el selector del servicio se evalúa continuamente.

Cuando se produce un cambio, los resultados se realiza un POST a un objeto de endpoint.

Puede pensar en los objetos de Endpoint como un enlace entre un Servicio y un conjunto de Pods.

El servicio describe el conjunto de Pods a los que se debe enrutar el tráfico.

El endpoint realiza el trabajo de evaluar el selector y recopilar una lista de direcciones IP y puertos de los Pods.

Puede inspeccionar el punto extremo asociado con el servicio `redis-commander` con:

```
vagrant@master
kubectl describe service redis-commander
```

En la línea que comienza con `Endpoints:` usted debe notar una sola dirección IP.

Endpoints as a dynamic list

¿Qué pasa cuando cambias el número de Pods?

Escale el deployment `redis-commander` a tres instancias:

```
vagrant@master
kubectl edit deployment redis-commander
```

Y cambiar las réplicas a `replicas: 3`.

Debe recuperar los endpoints de nuevo con:

```
vagrant@master
kubectl get endpoints
```

Tienes el mismo número de endpoints que antes.

Solo que esta vez el Endpoint `redis-commander` tiene tres elementos en lugar de uno.

Puedes notar el mismo cambio en el Servicio con:

```
vagrant@master
kubectl describe service redis-commander
```

Tenga en cuenta que la línea que comienza `Endpoints:` tiene tres elementos.

De hecho, hay tres Pods con tres direcciones IP diferentes.

Puede verificar que las direcciones IP son las direcciones IP asignadas a los Pods con:

```
vagrant@master
kubectl get pod -o wide
```

Enrutamiento de tráfico a la red interna.

Puede llegar al pod cuando esté en el clúster con:

```
vagrant@master
curl <pod ip>:8081
```

Y puede hacerlo también para el Servicio:

```
vagrant@master
kubectl get svc
curl <service ip>:8080
```

Para buscar direcciones IP desde dentro de la red.

Si necesita depurar un Servicio o un Pod, debe considerar usar `kubectl port-forward`.

Enrutar el tráfico a un Pod

Puede enrutar el tráfico desde un Pod dentro del clúster a su computadora con:

```
vagrant@master
kubectl port-forward <pod_id> 8080:8081
```

Donde puerto `8081` es el puerto utilizado por el Pod y `8080` es el puerto abierto en su servidor.

Puedes conectarte a uno de los tres Pod `redis-commander` con:

```
vagrant@master
kubectl port-forward <pod_id> 8080:8081 &
```

Puedes verificar que la conexión funciona con:

```
vagrant@master
curl localhost:8080
```

Cuando hayas terminado, cierra la conexión con:

```
vagrant@master
killall kubectl
```

Enrutar el tráfico a un servicio

También puede enrutar el tráfico de un Servicio dentro del clúster a su computadora sin exponerlo al mundo exterior.

El comando es similar a cuando enrutó el tráfico a un Pod:

```
vagrant@master
kubectl port-forward service/redis-commander 3000:8080 &
```

Puedes verificar que la conexión funciona con:

```
vagrant@master
curl localhost:3000
```

Tenga en cuenta que en este escenario particular, el tráfico se distribuye entre las tres réplicas.

No se garantiza que la misma réplica atenderá todas sus solicitudes.

kube-proxy

El plano de control almacena una lista de endpoints en etcd.

También es consciente de qué Nodo están desplegados los Pods y su dirección IP.

`kube-proxy` es un agente que regularmente consume esa información y crea rutas para distribuir el tráfico en el clúster.

Existe kube-proxy?

¿kube-proxy Un proceso está escuchando conexiones entrantes?

¿Está instalado en cada nodo?

Puede enumerar todos los procesos que escuchan las conexiones entrantes en worker1 con:

```
bash  
vagrant ssh worker1 -c "netstat -ntlp"
```

Puedes hacer lo mismo en worker2:

```
bash  
vagrant ssh worker2 -c "netstat -ntlp"
```

No hay proceso escuchando las conexiones entrantes.

¿Cómo encamina el tráfico kube-proxy?

Routing table (No Realizar)

`kube-proxy` escribe reglas de iptables en cada nodo.

Cuando una solicitud se reenvía a un nodo en el clúster, iptables la inspecciona y muta en el espacio del núcleo.

`kube-proxy` crea reglas de iptables basadas en los endpoints almacenados en el plano de control.

Tenga en cuenta que desde Kubernetes 1.11 IPVS es predeterminado.

Debes inspeccionar las reglas de iptables creadas por `kube-proxy`:

```
bash
vagrant ssh worker1 -c "sudo iptables-save"
```

Las mismas reglas se escriben a worker2:

```
bash
vagrant ssh worker2 -c "sudo iptables-save"
```

El tráfico se enruta al pod correcto usando las reglas de iptables en el espacio del kernel.

DaemonSet

`kube-proxy` se implementa como un Pod en su clúster.

Pero como desea una única instancia `kube-proxy` para cada nodo, no tiene sentido implementarla con una implementación regular.

`kube-proxy` se implementa como un DaemonSet, un tipo diferente de implementación donde solo hay un Pod desplegado para cada nodo.

Puede enumerar las implementaciones de DaemonSets en su clúster con:

```
vagrant@master
kubectl get daemonset --all-namespaces
```

Su clúster es un clúster de tres nodos, por lo que debe tener tres `kube-proxy`.

Tenga en cuenta que **flannel** también se implementa como un DaemonSet.

Los pods creados por un DaemonSet se reaparecen cuando se bloquean.

Debes intentar eliminar un pod `kube-proxy`

Puedes listar todos los Pods con:

```
vagrant@master
kubectl get pods -n=kube-system
vagrant@master
kubectl delete pod <name> -n=kube-system
```

Kubernetes volverá a crear el Pod y `kube-proxy` continuará actualizando las reglas de iptables desde donde lo dejó.

Exponiendo la aplicación

Un Pod que se ejecuta en el clúster tiene una dirección IP dinámica.

Si enruta el tráfico directamente a él utilizando la dirección IP, es posible que deba actualizar la tabla de enrutamiento cada vez que vuelva a implementar el Pod.

De hecho, en cada deployment, se asigna una nueva dirección IP al Pod.

Para evitar la administración manual de direcciones IP, puede utilizar un Servicio.

El Servicio actúa como un balanceador de carga para un grupo de Pods.

Entonces, incluso si la dirección IP de un Pod cambia, el servicio siempre lo apunta.

Servicio ClusterIP

El Servicio ClusterIp es el Servicio predeterminado en Kubernetes.

Si no especifica un tipo de Servicio, Kubernetes crea un Servicio ClusterIp.

Debe inspeccionar el servicio redis y redis-commander con:

```
vagrant@master
kubectl get services
```

La columna *TIPO* debe leer ClusterIp para ambos.

En ClusterIP, el servicio tiene una dirección IP asignada.

Servicio NodePort

Los servicios son平衡adores de carga internos y no son accesibles desde fuera del clúster.

Sin embargo, a veces es necesario exponer el Servicio al público.

Cuando ese sea el caso, debe usar un servicio NodePort.

Un servicio NodePort expone un puerto fijo en cada Nodo del clúster.

El puerto es accesible desde el exterior y enruta el tráfico a su servicio.

Puede hacer que el servicio redis-commander sea un NodePort editándolo con:

```
vagrant@master
kubectl edit service redis-commander
```

y cambiando el tipo a `type: NodePort`.

Puede recuperar el puerto expuesto al público (NodePort) con:

```
vagrant@master
kubectl describe service redis-commander
```

El mismo puerto está disponible cuando enumera el servicio:

```
vagrant@master
kubectl get services
```

Debe verificar que puede acceder al Servicio desde fuera del clúster.

Visita las siguientes URLs:

- `http://worker1.local:<replace with NodePort>` o
- `http://worker2.local:<replace with NodePort>`

Tenga en cuenta que el Servicio NodePort tiene un puerto interno y una dirección similar a la del Servicio ClusterIP.

Using an Ingress

En nuestro laboratorio utilizaremos las ingress-controller que ya tenemos desplegado en el cluster que es **traefik-ingress.**

http://10.0.0.12:8080/dashboard/

http://10.0.0.11:8080/dashboard/

http://10.0.0.12

http://10.0.0.11

No realizaremos ninguna practica con nginx Ingres, es decir todo lo que pida el lab de realizar con nginx *no lo realizamos*.

El Ingress se encarga de enrutar el tráfico desde fuera del cluster a los Pods.

Cuando creas un manifiesto de Ingress y configuras un Servicio como backend, el controlador de Ingress toma prestada la lista de endpoints del Servicio y configura un proxy inverso para enrutar el tráfico a los Pods.

El tráfico no fluye a través del Servicio; en su lugar, se enruta directamente a los Pods.

Dado que los Servicios tienen su lista de endpoints actualizada gracias al controlador de endpoints, el controlador de Ingress siempre está al tanto de los Pods que se agregan o eliminan.

Pero antes de que puedas usar el controlador Ingress, debes instalarlo primero.

Installing the nginx Ingress (**No Realizar**)

Puede instalar el ingress de nginx creando el siguiente recurso:

```
vagrant@master
kubectl apply -f \
https://academy.learnk8s.io/networking/nginx-ingress-0.24.1.yaml
```

Debes verificar que el controlador Ingress fue creado correctamente con:

```
vagrant@master
kubectl get pods --all-namespaces
```

Observe cómo hay dos Pods nuevos: el controlador Ingress y el backend predeterminado de http.

Backend HTTP predeterminado

Puede inspeccionar la Implementación para el backend predeterminado de http con:

```
vagrant@master
kubectl get deployments default-http-backend -n=ingress-nginx
```

El backend predeterminado es simplemente otra aplicación.

Y probablemente tampoco sea emocionante.

Siempre responde con *el backend predeterminado* - mensaje 404 .

Lo utiliza el controlador nginx cuando no se configura una ruta.

Debe verificarlo visitando el Servicio.

Primero, conéctate al servicio con:

```
vagrant@master
kubectl port-forward service/default-http-backend -n=ingress-nginx 8080:80 &
```

Puede acceder al Servicio en su localhost con:

```
bash
curl localhost:8080
```

Una vez que hayas terminado, no olvides cerrar la conexión.

```
vagrant@master
killall kubectl
```

Dado que el predeterminado-http-backend es una implementación regular, puede escalarlo para ejecutar tres Pods con:

```
vagrant@master
kubectl edit deployments default-http-backend -n=ingress-nginx
```

Y cambiando las réplicas a `replicas: 3`.

Puedes verificar que hay dos pods más con:

```
vagrant@master
kubectl get pods -n=ingress-nginx
```

Ahora puedes manejar tres veces más 404 páginas.

Exposing the Ingress Deployment

La implementación de Ingress nginx se expone con un Servicio con un NodePort.

Puedes verificar que con:

```
vagrant@master
kubectl get services --all-namespaces
```

Debe visitar a uno de los trabajadores de su navegador utilizando el NodePort que se encuentra en el Servicio.

```
bash
curl http://worker1.local:<replace with Ingress NodePort>
```

Debería ser recibido por el mismo *backend predeterminado*: mensaje 404 .

Puede ver el mensaje porque no hay una configuración de ruta y la entrada vuelve al backend predeterminado.

Exposing a Deployment with an Ingress manifest

Debe crear un manifiesto de Ingress para exponer la aplicación **redis-commander** a nuestros usuarios publicos, este manifest lo tendremos que modificar para poder llegar a través de <http://redis.curso.local>

Crea un archivo `ingress-redis-commander.yaml` con el siguiente contenido:

```
ingress-redis-commander.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: redis-commander
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - http:
    paths:
    - backend:
        serviceName: redis-commander
        servicePort: 8080
    path: /
```

Puedes crear el recurso con:

```
vagrant@master
kubectl create -f ingress-redis-commander.yaml
```

Si creó el manifiesto de Ingress correctamente, debería poder acceder a la interfaz web para Redis Commander desde `http://worker1.local:<replace with Ingress NodePort>`

Si hemos modificado el manifiesto, tendríamos que llegar:

<http://redis.curso.local/>

The screenshot shows the Redis Commander web application. At the top, there's a navigation bar with links for Refresh, Commands, More, and Disconnect. Below the navigation is a sidebar on the left containing a tree view with three nodes: 'local (localhost null 0)', 'local (redis:6379 0)', and 'local (localhost null 0)'. The main area is a table titled 'Redis' with columns 'Name' and 'Value'. The table lists various Redis configuration parameters. The 'Value' column for most parameters is truncated at the end of the screenshot.

Name	Value
# server	
Redis version	4.0.14
Redis git sha1	00000000
Redis git dirty	0
Redis build	788bbe4983c54eaa
Redis mode	standalone
Os	Linux 4.15.0-51-generic x86_64
Arch bits	64
Multiplexing api	epoll
Atomicvar api	atomic-builtin
Gcc version	8.3.0
Process	1
Run	e1c1e349c9287325ec9054bc9fd3726b435225a
Tcp port	6379

Ingress está enrutando el tráfico a la aplicación redis-commander.

Recordar que **redis.curso.local** lo resolvemos a través de ficheros hosts de nuestro puesto de trabajo:

10.0.0.12 redis.curso.local

10.0.0.11 redis.curso.local

Inspecting the Ingress controller (**No Realizar**)

El controlador nginx está diseñado para:

- Escucha los cambios para los manifiestos de ingress. Cuando se agrega, edita o elimina un recurso de Ingress, el controlador reacciona al cambio y reconfigura los bloques del servidor nginx (hosts virtuales) en consecuencia
- enrutar el tráfico como un proxy inverso utilizando la información almacenada en los bloques del servidor (hosts virtuales)

Dado que el controlador nginx ejecuta nginx (el servidor web), debería poder inspeccionar los bloques de servidor generados automáticamente (hosts virtuales).

Vamos a intentar eso.

Listar el controlador nginx con:

```
vagrant@master
kubectl get pods --all-namespaces
```

Adjuntar al controlador nginx en ejecución con:

```
vagrant@master
kubectl exec -ti <pod id> -n=ingress-nginx bash
```

Localice la configuración de nginx:

```
bash
cd /etc/nginx/
```

Puede inspeccionar la configuración completa con:

```
bash
cat nginx.conf
```

Es abrumador porque es autogenerado.

Puedes saltar a las partes relevantes buscando `redis-commander`.

```
bash
cat nginx.conf | grep -B 10 -A 5 redis-commander
```

Debería poder reconocer el bloque del servidor (host virtual).

Service discovery

El controlador de Ingress no es el único componente que toma prestada la lista de endpoints vinculados al servicio.

El servidor DNS observa la API de Kubernetes en busca de nuevos servicios y endpoints y crea un conjunto de registros DNS.

En particular, el servidor DNS crea un registro DNS para cada servicio en el clúster.

Usando registros DNS

Debería estar en la misma red que sus Pods para probar la resolución del nombre de dominio.

La forma más fácil de hacerlo es lanzar un Pod dentro de Kubernetes.

Un Pod en Kubernetes puede acceder a cualquier otro Pod o Servicio (a menos que haya instalado políticas de red o un service mesh).

Puedes implementar un solo Pod de busybox y conectarte a él con:

```
vagrant@master
kubectl run -ti --rm=true busybox --image=busybox
```

Recupere la dirección IP del servicio redis-commander con:

```
busybox
kubectl get service redis-commander
```

Desde ese shell, puede consultar el servicio redis-commander:

```
busybox
wget <ip_redis-commander_service>:8080
```

Pero como cada servicio tiene una entrada en el servidor DNS, puede usar un nombre de dominio completo:

```
busybox
wget redis-commander:8080
# nslookup redis-commander.default.svc.cluster.local
```

Variables de entorno

Si prefiere no utilizar un servidor DNS, puede usar las variables de entorno que se llenan automáticamente con `kubelet`:

Puede enumerar todas las variables de entorno en el contenedor con:

```
busybox  
printenv
```

Observe cómo hay un `REDIS_COMMANDER_SERVICE_HOST` apuntar al servicio redis-commander.

Puedes usar eso para solicitar la aplicación:

```
busybox  
wget $REDIS_COMMANDER_SERVICE_HOST:8080
```

Tenga en cuenta que cada vez que crea un servicio, Kubernetes crea una entrada de DNS similar a esta:

`my-service-name.the-namespace.svc.cluster.local`

Puede desglosar la cadena de la siguiente manera:

- `my-service-name` es el nombre que le asignaste al Servicio
- `the-namespace` es el nombre del espacio de nombres actual (`default` si no ha creado uno)
- `svc.cluster.local` siempre se añade al final

Si está buscando un servicio dentro de su espacio de nombres, no necesita especificar nada más que el nombre del servicio.

Laboratorio kubernetes Autoscaling POD'S

<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

Con "Horizontal Pod Autoscaling", Kubernetes automáticamente escala el numero de pods, dependiendo del uso de CPU (o cualquier otra métrica de las soportadas) que hayamos definido en nuestro Deployment. Este escalado no está soportado para los de tipo DaemonSet.

Es soportado por el comando kubectl para crear y visualizar nuestros "HPA",

Desplegar una imagen de un sitio con HPA

Crear un autoescalado para nuestra aplicación.

```
# kubectl run php-apache --image=gcr.io/google_containers/hpa-example \
--requests=cpu=200m --expose --port=80

# kubectl autoscale deployment php-apache --cpu-percent=5 --min=2 --max=20
```

Visualizar nuestros hpa:

```
# kubectl get hpa

NAME          REFERENCE          TARGETS      MINPODS     MAXPODS
REPLICAS     AGE                0% / 5%    2           20          2
php-apache   Deployment/php-apache 1m
```

Generamos tráfico para probar nuestro HPA

```
# kubectl run -i --tty load-generator --image=busybox /bin/sh
while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!
```

Verificamos que el HAP que hemos creado esté autoescalando nuestra aplicación

```
# kubectl get hpa php-apache

NAME          REFERENCE          TARGETS      MINPODS     MAXPODS
REPLICAS     AGE                242% / 5%   2           20          4
php-apache   Deployment/php-apache 5m
```

Pasados unos minutos veremos que los pods siguen creciendo hasta un límite de 20

```
# kubectl get hpa php-apache
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS
REPLICAS	AGE			
php-apache	Deployment/php-apache	28% / 5%	2	20
	18m			16

Ver en detalle uno de nuestros hpa

```
# kubectl describe hpa php-apache
```

Borrar uno de nuestros hpa

```
# kubectl delete hpa php-apache
```

Laboratorio HPA con tomcat a través de archivo declarativo yaml

En este laboratorio trabajaremos con una imagen de tomcat, limitando la carga en cpu y provocando el autoescalado:

```
# vi deployment-tomcat-hpa.yaml
```

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: tomcat-deployment
spec:
  selector:
    matchLabels:
      app: tomcat
  replicas: 1
  template:
    metadata:
      labels:
        app: tomcat
  spec:
    containers:
    - name: tomcat
      image: tomcat:9.0
      ports:
      - containerPort: 8080
      resources:
        requests:
          cpu: "100m"
        limits:
          cpu: "200m"
```

Desplegamos el deployment:

```
#kubectl apply -f deployment-tomcat-hpa.yaml
```

Exponemos el deployment:

```
#kubectl expose deployment/tomcat-deployment --type="NodePort" --port 8080
```

Le decimos que el deployment

Le decimos que escala nuestros pod del deployment:

```
#kubectl autoscale deployment tomcat-deployment --cpu-percent=15 --min=1 --max=10
```

Lanzamos un contenedor basado en busybox y ejecutamos el bucle contra el servicio de nuestro tomcat, para meter carga de CPU a los contenedores:

```
#kubectl run -i --tty load-tomcat --image=busybox /bin/sh
while true;do wget -q -O- http://tomcat-deployment.default.svc.cluster.local:8080 ;done
```

Ahora tendríamos que ver como escalan, tendremos que esperar un poco de tiempo:

```
#watch kubectl get hpa tomcat-deployment
#watch kubectl get pod -o wide -l app=tomcat
```

jobs

Descripción general

En kubernetes un **jobs** es un objeto de controlador que representa una tarea finita. Los jobss difieren de otros objetos de controlador porque los jobss administran la tarea a medida que se ejecuta hasta su finalización, en lugar de administrar un estado deseado en curso (como la cantidad total de Pods en ejecución).

Los jobss son útiles para tareas de procesamiento y tareas por lotes grandes. Los jobss se pueden usar para admitir la ejecución paralela de Pods. Puedes usar un jobs para ejecutar en paralelo elementos de jobs independientes que se relacionan, como enviar correos electrónicos, procesar marcos, transcodificar archivos, analizar claves de base de datos, etcétera. Sin embargo, los jobss no están diseñados para procesos paralelos de comunicación estrecha, como flujos continuos de procesos en segundo plano.

En KUBERNETES, se encuentran los siguientes tipos de jobs:

- **Jobs no paralelo:** un jobs que crea solo un Pod (que se vuelve a crear si el Pod finaliza sin éxito) y que se completa cuando el Pod finaliza de forma correcta.
- **Jobss paralelos con un recuento de finalización:** un jobs que se completa cuando una cierta cantidad de Pods finaliza de forma correcta. Especifica la cantidad deseada de finalizaciones mediante el campo de `completions`.

Los [objetos_jobs](#) de Kubernetes representan los jobss. Cuando se crea un jobs, el controlador del jobs crea uno o más Pods y garantiza que sus Pods terminen de forma correcta. A medida que los Pods finalizan, un jobs realiza un seguimiento de la cantidad de Pods que completaron sus tareas de forma correcta. Cuando se alcanza la cantidad deseada de finalizaciones exitosas, el jobs está completo.

Al igual que otros controladores, un controlador de jobs crea un Pod nuevo si uno de sus Pods falla o se borra.

Crea un Jobs

<https://cloud.google.com/kubernetes-engine/docs/how-to/jobs?hl=es-419>

<https://raw.githubusercontent.com/openshift-evangelists/kbe/master/specs/jobs/job.yaml>

Puedes crear un jobs mediante `kubectl apply` con un archivo de manifiesto.

En el siguiente ejemplo, se muestra un manifiesto de :

```
apiVersion: batch/v1
kind: Job
metadata:
  # Unique key of the Job instance
  name: example-job
spec:
  template:
    metadata:
      name: example-job
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl"]
          args: ["-Mbignum=bpi", "-wle", "print bpi(2000)"]
        # Do not restart containers after they exit
      restartPolicy: Never
```

Copia el manifiesto a un archivo llamado **config.yaml** y crea el jobs:

```
kubectl apply -f config.yaml
```

Este jobs procesa pi a 2,000 lugares y, luego, lo imprime.

El único requisito para un objeto de jobs es que el campo de la `template` del Pod es obligatorio.

Recuento de finalizaciones de jobss

Un jobs se completa cuando una cantidad específica de Pods finaliza de manera correcta. De forma predeterminada, un jobs no paralelo con un solo Pod se completa apenas el Pod finaliza de manera correcta.

Si tienes un jobs paralelo, puedes establecer un *recuento de finalización* mediante el campo de `completions` opcional. Este campo especifica cuántos Pods deben finalizar de manera correcta antes de que se complete el jobs. El campo de `completions` acepta un valor positivo distinto de cero.

Omitir `completions` o especificar un valor cero hace que el éxito de cualquier Pod indique el éxito de todos los Pods.

Copia config.yaml del ejemplo anterior a un archivo llamado config-2.yaml. En config-2.yaml, cambia el name a example-job-2 y agrega completions: 8 al campo spec del jobs. Esto especifica que debe haber ocho finalizaciones exitosas:

Crea el jobs:

```
kubectl apply -f config-2.yaml
```

Nota: Para un jobs no paralelo, no establezcas el campo de completions,

El valor predeterminado de completions es 1. Cuando se configuran las completions, el campo de parallelism establece de manera predeterminada 1 a menos que se establezca lo contrario. Si no están configurados ambos campos, sus valores predeterminados son 1.

Administra el paralelismo

De forma predeterminada, los Pods de jobs no se ejecutan en paralelo. El campo de parallelism opcional especifica la cantidad máxima deseada de Pods que un jobs debe ejecutar al mismo tiempo en un momento determinado.

La cantidad real de Pods que se ejecutan en un estado estable puede ser menor que el valor de parallelism si el jobs restante es menor que el valor de parallelism. Si también configuraste las completions, la cantidad real de Pods que se ejecutan en paralelo no excede la cantidad de finalizaciones restantes. Un jobs puede limitar la creación de Pods en respuesta a un fallo excesivo en la creación de Pods.

Copia config.yaml del ejemplo anterior a un archivo llamado config-3.yaml. En config-3.yaml, cambia el name a example-job-3 y agrega parallelism: 5 al campo spec del jobs. Esto especifica que debe haber cinco Pods simultáneos en ejecución:

Crea el jobs:

```
kubectl apply -f config-3.yaml
```

Nota: Para un jobs no paralelo, no establezcas el campo de parallelism.

El valor predeterminado de parallelism es 1 si el campo se omite o si se establece lo contrario. Si el valor se establece en 0, el jobs se pone en pausa hasta que se aumenta el valor.

Especifica una fecha límite

De forma predeterminada, un jobs crea Pods nuevos para siempre, si sus Pods fallan con continuidad. Si prefieres no tener un reintento de jobs para siempre, puedes especificar un valor de *fecha límite* mediante el campo opcional activeDeadlineSeconds.

Una fecha límite le otorga a un jobs una cantidad específica de tiempo, en segundos, para completar sus tareas con éxito antes de finalizar.

Para especificar una fecha límite, agrega el valor `activeDeadlineSeconds` al campo `spec` del `jobs` en el archivo de manifiesto. Por ejemplo, la siguiente configuración especifica que el `jobs` tiene 100 segundos para completarse con éxito:

Nota: Asegúrate de agregar el valor `activeDeadlineSecond` al campo `spec` del `jobs`. El campo `spec` en el campo de la `template` de Pod también acepta un valor `activeDeadlineSeconds`.

Si un `jobs` no se completa con éxito antes de la fecha límite, el `jobs` finaliza con el estado `DeadlineExceeded`. Esto hace que la creación de Pods se detenga y que los Pods existentes se borren.

Especifica un selector de Pods

Especificando un selector de forma manual es útil si quieras actualizar una plantilla de Pod del `jobs`, pero quieres que los Pods actuales del `jobs` se ejecuten en el `jobs` actualizado.

Se crea una instancia de `jobs` con un campo `selector`. El `selector` genera un identificador único para los Pods del `jobs`. El ID generado no se superpone con ningún otro `jobs`. Por lo general, no establecerías este campo tú mismo: establecer un valor `selector` que se superponga con otro `jobs` puede causar problemas con los Pods del otro `jobs`. Para configurar el campo, debes especificar `manualSelector: True` en el campo `spec` del `jobs`.

Por ejemplo, puedes ejecutar `kubectl get job my-job --output=yaml` si quieres ver la especificación del `jobs`, que contiene el selector generado para sus Pods:

Cuando creas un `jobs` nuevo, puedes configurar el valor `manualSelector` a `True`, luego configura el valor `job-uid` del campo `selector` de la siguiente manera:

Los Pods que crea `my-new-job` usan el UID del Pod anterior.

Nota: Los `jobs` tienen sus propios UID. El UID del `jobs` nuevo es diferente del UID del `jobs` anterior.

Inspecciona un jobs

kubectl

Console

Para verificar el estado de un jobs, ejecuta el siguiente comando:

```
kubectl describe job my-job
```

Para ver todos los recursos de Pod en tu clúster, incluidos los Pods creados por el jobs que se completaron, ejecuta lo siguiente:

```
kubectl get pods -a
```

La marca `-a` especifica que se deben mostrar todos los recursos del tipo especificado (en este caso, Pods).

Escala un jobs

kubectl

Console

Para escalar un jobs, ejecuta el siguiente comando:

```
kubectl scale job my-job --replicas=[VALUE]
```

`kubectl scale` hace que cambie la cantidad de Pods que se ejecutan al mismo tiempo. En particular, cambia el valor del `parallelism` al `[VALOR]` que especifiques.

Borra un jobs

Cuando se completa, el jobs deja de crear Pods. El objeto de la API de jobs no se quita cuando se completa, lo que permite que veas su estado. Los Pods que crea el jobs no se borran, pero finalizan. La retención de los Pods te permite ver sus registros y, también, interactuar con ellos.

kubectl

Console

Para borrar un jobs, ejecuta el siguiente comando:

```
kubectl delete job my-job
```

Cuando borras un jobs, también se borran todos sus Pods.

Para borrar un jobs, pero conservar sus Pods, especifica la marca --cascade false:

```
kubectl delete jobs my-job --cascade false
```

Cronjobs

Descripción general

Puedes crear **CronJobs** para realizar tareas limitadas y relacionadas con la hora que se ejecutan una o varias veces en una hora especificada. CronJobs se puede usar para tareas puntuales automáticas, como las copias de seguridad, los informes y el envío de correos electrónicos.

CronJobs usa objetos [trabajo](#) para completar sus tareas. CronJobs crea un objeto trabajo alrededor de una vez por hora de ejecución de su programación.

Los CronJobs se crean, administran, escalan y borran de la misma manera que los jobs.

Nota: Para usar CronJobs, tu clúster debe ejecutar Kubernetes versión 1.8.x o posterior.

Crea un CronJob

El el ejemplo siguiente de un CronJob, se imprime la hora actual y una string cada un minuto:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo "Hello, World!"
  restartPolicy: OnFailure
```

CronJobs usa el campo requerido `jobTemplate` que contiene la especificación de trabajo `jobTemplate: spec` para los trabajos que crea. La especificación de trabajo contiene una especificación del pod `spec: template: spec` de los pods que crea para llevar a cabo las tareas que específicas.

CronJobs usa el campo `schedule` requerido que acepta una hora en el formato [crontab](#) estándar de Unix. Todas las horas de CronJob están en UTC:

- El primer valor indica el minuto (entre 0 y 59).
- El segundo valor indica la hora (entre 0 y 23).
- El tercer valor indica el día del mes (entre 1 y 31).
- El cuarto valor indica el mes (entre 1 y 12).
- El quinto valor indica el día de la semana (entre 0 y 6).

`schedule` también acepta * y ? como comodines. La combinación de / con los rangos especifica que la tarea se debe repetir a intervalos regulares. En el ejemplo anterior, */1 * * * * indica que la tarea se debe repetir cada un minuto, todos los días, todos los meses.

Nota: Para obtener más información sobre cómo especificar las programaciones de CronJob, consulta [CRON expression](#).

Para crear este CronJob, guarda el manifiesto anterior como `config.yaml` y, a continuación, ejecuta el comando siguiente:

```
kubectl apply -f cronjob.yaml
```

Como alternativa, para crear un CronJob sin crear un archivo de manifiesto, usa `kubectl run`:

```
kubectl run hello --schedule="*/1 * * * *" --restart OnFailure \
--image busybox -- /bin/sh -c "date; echo Hello, World\!"
```

Especifica una fecha límite

El campo opcional `startingDeadlineSeconds` indica el plazo límite (en segundos) para iniciar el CronJob en caso de que no cumpla con su hora programada por alguna razón. Los CronJobs que no se cumplen se consideran errores.

Para especificar un plazo límite, agrega el valor `startingDeadlineSeconds` al campo de CronJob `spec` en el archivo del manifiesto. Por ejemplo, en el manifiesto siguiente, se especifica que el CronJob tiene 100 segundos para comenzar:

Si no especificas un valor `startingDeadlineSeconds`, no se usa un plazo límite.

Especifica una política de simultaneidad

En el campo opcional `concurrencyPolicy`, se especifica cómo tratar las ejecuciones simultáneas de un trabajo creado mediante el controlador de CronJob. Debes especificar `concurrencyPolicy` en el campo `spec` de CronJob.

`concurrencyPolicy` admite los valores siguientes:

- `Allow`: permite trabajos simultáneos por configuración predeterminada.
- `Forbid`: prohíbe trabajos simultáneos y omite la próxima ejecución si la anterior aún no finalizó.
- `Replace`: cancela el trabajo en ejecución actual y lo reemplaza con uno nuevo.

Suspende las ejecuciones posteriores

En el campo opcional `suspend`, cuando se establece en `true`, se suspenden todas las ejecuciones posteriores. Sin embargo, no suspende las ejecuciones en curso. Debes especificar `suspend` en el campo `spec` de CronJob.

El valor predeterminado de `suspend` es `false`.

Especifica los límites del historial

El `successfulJobsHistoryLimit` y `failedJobsHistoryLimit` opcional especifican la cantidad de trabajos completados y con errores que se deben conservar. Debes especificarlos en el campo `spec` de CronJob.

Según la configuración predeterminada, `successfulJobsHistoryLimit` se establece en 3 y `failedJobsHistoryLimit` se establece en 1. Establecer el valor de cualquiera de estos campos en 0 provoca que ninguno de los trabajos se conserve después de que finalicen.

Inspecciona un CronJob

Para verificar el estado de un CronJob, ejecuta el comando siguiente:

```
kubectl describe cronjob [CRON_JOB]
```

Para ver todos los recursos del pod en tu clúster, incluso los pods completados que creó CronJob, ejecuta el comando siguiente:

```
kubectl get pods -a
```

La marca `-a` especifica que se deben mostrar todos los recursos del tipo especificado (en este caso, Pods).

Borra un CronJob

Para borrar un CronJob, ejecuta el comando siguiente:

```
kubectl delete cronjob [CRON_JOB]
```

Cuando borras un CronJob, el recolector de elementos no utilizados de Kubernetes borra de forma automática los trabajos asociados y no se inician trabajos nuevos.

<https://cloud.google.com/kubernetes-engine/docs/concepts/pod?hl=es-419>

Monitorizacion con Heapster with an InfluxDB backend and a Grafana UI

Enlace: <https://github.com/kubernetes/heapster/blob/master/docs/influxdb.md>

Instalar Heapster

<https://github.com/kubernetes/heapster.git>

```
git clone https://github.com/kubernetes/heapster.git
```

<https://github.com/kubernetes/heapster/blob/master/docs/influxdb.md>

```
kubectl create -f deploy/kube-config/influxdb/  
kubectl create -f deploy/kube-config/rbac/heapster-rbac.yaml  
  
# Forwarding desde contenedor  
kubectl port-forward grafana-core 3000  
  
# Entraremos desde localhost al puerto 9090. Usuario y contraseña por  
defecto admin/admin  
http://localhost:3000
```

Lab Realizar una actualización continua de una aplicación en Kubernetes

Descripción:

En este laboratorio práctico, se le presentará un clúster de tres nodos. Deberá implementar su aplicación para poder comenzar a servir a sus usuarios finales.

Implementará la imagen desde **linuxacademycontent/kubeserve:v1** y luego verificará que la implementación fue exitosa.

Una vez que su aplicación se esté ejecutando y sirviendo a los clientes, realizará una actualización continua, asegurándose de que la implementación sea exitosa y no haya tiempo de inactividad para sus usuarios finales.

Hará uso de la herramienta de línea de comandos kubectl para realizar todas las operaciones, en combinación con el comando **set image** para realizar la actualización continua a la nueva versión (**linuxacademycontent/kubeserve:v2**).

Cuando haya verificado que los usuarios finales ahora usan la versión 2 de la aplicación frente a la versión 1, a continuación tendremos que volver a la versión 1 de nuestra aplicación, puede considerar que este laboratorio práctico está completo, tras realizar todos los pasos anteriores.

Objetivos de aprendizaje

Cree y despliegue la versión 1 de la aplicación, y verifique que el deployment es correcto.

1. Use el siguiente YAML nombrado `kubeserve-deployment.yaml` para crear su implementación:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubeserve
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubeserve
  template:
    metadata:
      name: kubeserve
      labels:
        app: kubeserve
    spec:
      containers:
        - image: linuxacademycontent/kubeserve:v1
          name: app
```

2. Use el siguiente comando para crear el deployment:

```
kubectl apply -f kubeserve-deployment.yaml --record
```

3. Use el siguiente comando para verificar que la implementación se realizó correctamente:

```
kubectl rollout status deployments kubeserve
```

4. Use el siguiente comando para verificar que la aplicación tenga la versión correcta:

```
kubectl describe deployment kubeserve
```

Amplíe la aplicación para crear alta disponibilidad.

1. Use el siguiente comando para escalar su aplicación a cinco réplicas:

```
kubectl scale deployment kubeserve --replicas=5
```

2. Use el siguiente comando para verificar que se hayan creado réplicas adicionales:

```
kubectl get pods
```

Cree un servicio para que los usuarios puedan acceder a la aplicación de tipo nodeport.

1. Use el siguiente comando para crear un servicio para su implementación:

```
kubectl expose deployment kubeserve --port 80 --target-port 80 --type NodePort
```

2. Use el siguiente comando para verificar que el servicio esté presente y recopile la IP del clúster:

```
kubectl get services
```

3. Use el siguiente comando para verificar que el servicio responde:

```
curl http://<ip-address-of-the-service>
```

Realice una actualización continua a la versión 2 de la aplicación y verifique su éxito.

1. Use este comando curl loop para ver el cambio de versión a medida que realiza la actualización continua:

```
while true; do curl http://<ip-address-of-the-service>; done
```

2. Use este comando para realizar la actualización (mientras se ejecuta el bucle curl):

```
kubectl set image deployments/kubeserve app=linuxacademycontent /kubeserve:v2 --v 6
```

3. Use este comando para ver el ReplicaSet adicional creado durante la actualización:

```
kubectl get replicasets
```

4. Use este comando para verificar que todos los pods estén en funcionamiento:

```
kubectl get pods
```

5. Use este comando para ver el historial de implementación:

```
kubectl rollout history deployment kubeserve
```

```
deployment.extensions/kubeserve
REVISION  CHANGE-CAUSE
1          kubectl apply --filename=kubeserve-deployment.yaml --
record=true
2          kubectl apply --filename=kubeserve-deployment.yaml --
record=true
```

Ahora volvemos a la versión 1 primero la visualizamos y luego realizamos un undo con el parámetro `--to-revision=1`:

```
kubectl rollout history deployment/kubeserve --revision 1

deployment.extensions/kubeserve with revision #1
Pod Template:
  Labels:      app=kubeserve
               pod-template-hash=968646c97
  Annotations: kubernetes.io/change-cause: kubectl apply --
filename=kubeserve-deployment.yaml --record=true
  Containers:
    app:
      Image:      linuxacademycontent/kubeserve:v1
      Port:       <none>
      Host Port: <none>
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
```

```
$ kubectl rollout undo deployment/kubeserve --to-
revision=1
```

Kubernetes Cluster Dashboard y METALLB EN MODO LAYER 2

KUBERNETES ON PREMISE CON METALLB EN MODO LAYER 2.

Una de las desventajas de tener un *cluster* de *k8s* On Premise/BareMetal es la falta de *LoadBalancer*.

Gracias a MetalLB esto está resuelto de una manera fácil y elegante.

Cuando queremos publicar servicios en k8s, lo hacemos usando un *Ingress Controller* (**nginx** o **Traefik** por nombrar algunos). Este servicio se apoya de las direcciones IP de los hosts si no tenemos LoadBalancer.

El tema se complica cuando queremos publicar servicios que no son HTTP o HTTPS.

La ventaja de un IP via LoadBalancer es que podemos usarlos en varios servicios (pods) y publicar cualquier puerto en TCP o UDP.

METALLB

<https://metallb.universe.tf/>

Los requerimientos son los siguientes:

- Un cluster de k8s en la versión 1.9.0 o más reciente y que no tenga un tipo de LoadBalancer en funcionamiento, esto quiere decir que tendríamos problemas usando esta solución en GKE por ejemplo, podríamos utilizarlo a nivel baremetal.
- Una configuración de cluster que pueda coexistir con MetalLB
<https://metallb.universe.tf/installation/network-addons/>
- Direcciones IPv4 para asignar usando este servicio.
- Dependiendo del modo operativo, podríamos necesitar un router que soporte BGP.

<https://metallb.universe.tf/#requirements>

Aunque pongamos los servicios en type LoadBalancer no se asigna ninguna ip externa y aunque se la pongamos manualmente no podemos llegar a ella.

La documentación que hay sobre Ingress controller sirve para configurarla en los proveedores aws google azure... que expongamos el servicio y asignemos una ip no es suficiente dado que ni flannel ni kubernetes se encargan de la configuración y conexión con redes externas, esto sucede en las instalaciones que tenemos de kubernetes a nivel de baremetal.

MetalLB que se encargara de esta tarea por nosotros

Para instalar MetalLB, aplicamos este manifiesto:

<https://metallb.universe.tf/installation/>

```
$ kubectl apply -f
https://raw.githubusercontent.com/google/metallb/v0.7.3/manifests/metallb.yaml
```

This will deploy MetalLB to your cluster, under the `metallb-system` namespace. The components in the manifest are:

- The `metallb-system/controller` deployment. This is the cluster-wide controller that handles IP address assignments.
- The `metallb-system/speaker` daemonset. This is the component that speaks the protocol(s) of your choice to make the services reachable.
- Service accounts for the controller and speaker, along with the RBAC permissions that the components need to function.

Ahora crearemos el manifiesto de configuracion, en este ejemplo otorgaremos 20 ip a Metallb, estoy utilizando este rango de red, porque estamos con mv creadas a través de vagrant con virtualbox y es el rango en el que está el direccionamiento de las maquinas virutales:

<https://metallb.universe.tf/configuration/>

Estamos utilizando el modo capa 2

\$ vi metallb.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 10.0.0.25-10.0.0.60
```

Desplegamos el ConfigMap:

\$ kubectl create -f metallb.yaml

Ahora comenzamos a desplegar el dashboard de kubernetes:

<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>

Creamos un certificado ssl para el dashboard

Antes de desplegar el dashboard vamos a crear un certificado y añadirlo como secreto para que sustituya al que se despliega por defecto pues solo responde al domain localhost y no vamos a usar este dominio. En vez de localhost usaremos dashboard

```
$mkdir $HOME/certs
$cd $HOME/certs
$openssl genrsa -out dashboard.key 2048
$openssl rsa -in dashboard.key -out dashboard.key
$openssl req -sha256 -new -key dashboard.key -out dashboard.csr -subj
'/CN=dashboard'
$openssl x509 -req -sha256 -days 365 -in dashboard.csr -signkey
dashboard.key -out dashboard.crt
```

Creamos el secreto

```
$kubectl create namespace kubernetes-dashboard
$kubectl -n kubernetes-dashboard create secret generic kubernetes-
dashboard-certs --from-file=$HOME/certs
```

Desplegar el dashboard

```
$kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-
beta1/aio/deploy/recommended.yaml
```

A continuacion tenemos que editar el servicio del dashboard y cambiar el type de ClusterIP a LoadBalancer, para integrarlo con metallb:

```
$ kubectl -n kubernetes-dashboard edit service kubernetes-dashboard

apiVersion: v1
kind: Service
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":{"k8s-app":"kubernetes-dashboard"},"name":"kubernetes-dashboard","namespace":"kube-system"},"spec":{"ports":[{"port":443,"targetPort":8443}],"selector":{"k8s-app":"kubernetes-dashboard"}}}
    creationTimestamp: "2019-04-24T22:21:15Z"
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
  resourceVersion: "1753"
  selfLink: /api/v1/namespaces/kube-system/services/kubernetes-dashboard
  uid: 4612785f-66df-11e9-8180-000c29e7b067
spec:
  clusterIP: 10.110.50.44
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31394
    port: 443
    protocol: TCP
    targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
    sessionAffinity: None
    type: LoadBalancer
status:
{}
```

Salimos y guardamos los cambios.

Esto genera una ip publica, del rango generado anteriormente (10.0.0.25–10.0.0.60), para el servicio de dashboard, configuramos nuestro hosts para que resuelva dashboard a la ip publica generada.

```
$ kubectl -n kubernetes-dashboard get service kubernetes-dashboard
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes-dashboard	LoadBalancer	10.108.178.92	10.0.0.25	443:31787/TCP	27d

Resolvemos **dashboard.miempresa.com, a través del fichero hosts de nuestro Windows:**

C:\Windows\System32\drivers\etc\hosts

10.0.0.25 dashboard.curso.local

Con esto el dashboard ya sería accesible pero no tendríamos privilegios.

Creando el rol dashboard-admin:

<https://github.com/kubernetes/dashboard/wiki/Creating-sample-user>

```
$ vi admin-user.yaml
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

```
$ kubectl create -f admin-user.yaml
```

```
$ vi cluster-role.yaml
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

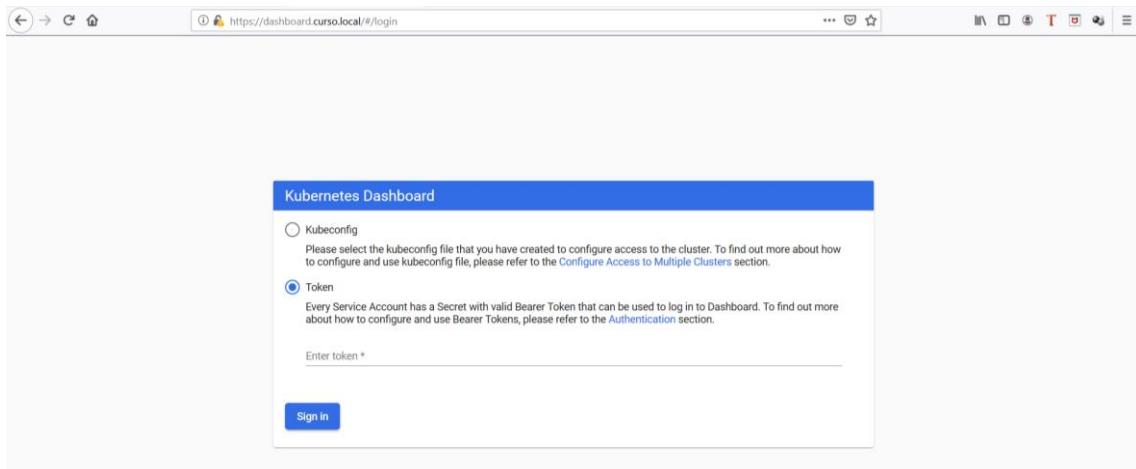
```
$ kubectl create -f cluster-role.yaml
```

Copiar el token de acceso

```
$ kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret | grep admin-user | awk '{print $1}')
```

Una vez hemos configurado nuestro archivo host en local para que resuelva la ip publica **10.0.0.25** a **dashboard.curso.local**

En el navegador <https://dashboard.curso.local/> y copiar el token la pagina de login:



The screenshot shows the Kubernetes Dashboard Overview page. The left sidebar shows the 'default' namespace with 'Workloads' expanded, listing Deployments, Pods, Replica Sets, and Stateful Sets. The main area displays 'Workload Status' with four green circles at 100% for Deployments, Pods, Replica Sets, and Stateful Sets. Below this, the 'Deployments' table lists three entries:

Name	Namespace	Labels	Pods	Age	Images
holo	default	app: hola	3 / 3	5 days	nbrown/nginxhello:1.12.1
adios	default	app: adios	3 / 3	5 days	nbrown/nginxhello:1.13.5
my-apache	default	app.kubernetes.io/instance: my-apache app.kubernetes.io/managed-by: Tiller	1 / 1	7 days	docker.io/bitnami/apache:2.4.39-debian-9-r53

K8Dash - Kubernetes Dashboard

<https://github.com/herbrandson/k8dash>

K8Dash is the easiest way to manage your Kubernetes cluster. Why?

- Full cluster management: Namespaces, Nodes, Pods, Replica Sets, Deployments, Storage, RBAC and more
- Blazing fast and Always Live: no need to refresh pages to see the latest
- Quickly visualize cluster health at a glance: Real time charts help quickly track down poorly performing resources
- Easy CRUD and scaling: plus inline API docs to easily understand what each field does
- Simple OpenID integration: no special proxies required
- Simple installation: use the provided yaml resources to have K8Dash up and running in under 1 minute (no, seriously)

Getting Started

Deploy k8dash with something like the following...

NOTE: never trust a file downloaded from the internet. Make sure to review the contents of [kubernetes-k8dash.yaml](#) before running the script below.

```
kubectl apply -f
https://raw.githubusercontent.com/herbrandson/k8dash/master/kubernetes
-k8dash.yaml
```

Desplegamos un ingress en nuestro cluster [ingress-k8dash.yaml](#):

```
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: k8dash
  namespace: kube-system
spec:
  rules:
  -
    host: k8dash.curso.local
    http:
      paths:
      -
        path: /
        backend:
          serviceName: k8dash
          servicePort: 80
```

```
$ kubectl apply -f ingress-k8sdash.yaml
```

Service Account Token

The first (and easiest) option is to create a dedicated service account. This can be accomplished using the following script.

```
# Create the service account in the current namespace (we assume default)
kubectl create serviceaccount k8dash-sa

# Give that service account root on the cluster
kubectl create clusterrolebinding k8dash-sa --clusterrole=cluster-admin --serviceaccount=default:k8dash-sa

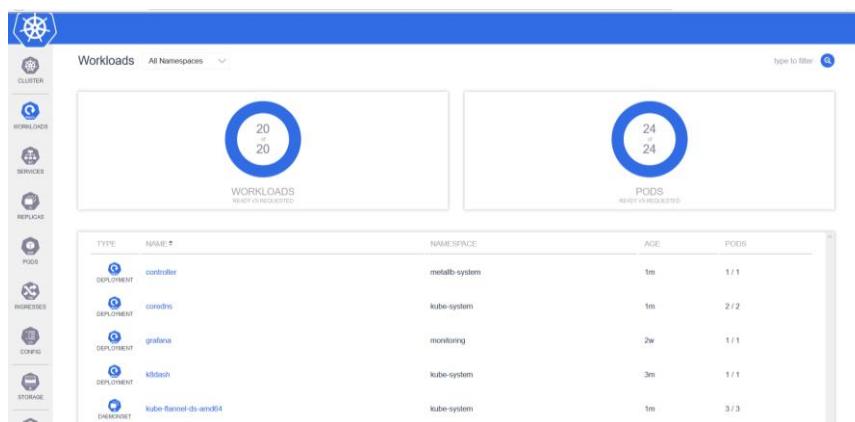
# Find the secret that was created to hold the token for the SA
kubectl get secrets

# Show the contents of the secret to extract the token
kubectl describe secret k8dash-sa-token-xxxxx
```

Resolvemos `k8dash.curso.local`, a través del fichero hosts de nuestro Windows:

C:\Windows\System32\drivers\etc\hosts

10.0.0.12 k8dash.curso.local



Deployment strategies

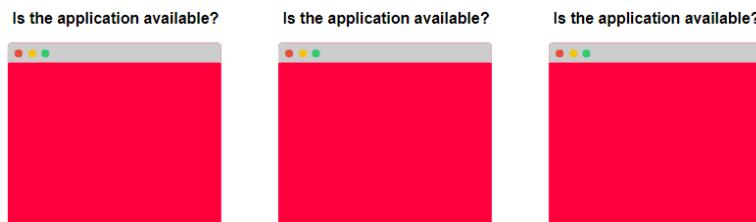


Cuando implementa una aplicación, ¿cómo sabe si la aplicación se implementó correctamente y puede responder al tráfico en vivo?

Puede tener la tentación de visitar la aplicación después de cada implementación para asegurarse de que funciona de la manera esperada.

Si tiene una sola aplicación, verificar el estado después de cada implementación es molesto pero factible.

Cuando su aplicación se vuelve popular, puede escalar la aplicación a tres instancias y aún verificar que la implementación fue exitosa visitando cada aplicación.



Es más difícil pero aún manejable.

Si tiene cientos de aplicaciones y docenas de instancias cada una, se convierte en un trabajo de tiempo completo para verificar el estado de cada una de ellas individualmente.

No se escala.



Sería mucho mejor si en lugar de que usted vigilara la implementación, podría hacer que la aplicación le informe el estado.

En lugar de buscar una respuesta exitosa, puede tener un proceso automatizado que verifique la salud de sus Pods.

Si la aplicación no es saludable, tal vez podría intentar reiniciarla (tal vez para solucionar un error transitorio) o al menos podría eliminarla del Servicio si está devolviendo errores como respuesta.

No desea que las réplicas que fallen respondan con errores al tráfico de producción.

Application health checks

Kubernetes ofrece dos mecanismos para rastrear el ciclo de vida de sus contenedores y Pods: **sondas de vida y de preparación (liveness and readiness probes)**.

Readiness probe

Una sonda de preparación está diseñada para sondear los contenedores dentro de un Pod y decidir si pueden recibir tráfico.

La sonda conecta y separa el contenedor al Servicio cuando detecta una respuesta saludable o no saludable del contenedor.

Imagina tener una aplicación web escrita en Spring Boot.

Dependiendo de los recursos asignados a él, la aplicación podría tardar unos 30 segundos en estar lista para atender el tráfico.

No querrás servir el tráfico a esa aplicación hasta que esté completamente cargada.

La sonda de preparación está diseñada para resolver tal problema.

Puede agregar una sonda de preparación (**readiness probes**) a sus contenedores de esta manera:

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
  - name: sise
    image: learnk8s/demo:1.0.0
    ports:
    - containerPort: 3000
  readinessProbe:
    initialDelaySeconds: 2
    periodSeconds: 5
    httpGet:
      path: /health
      port: 3000
```

Liveness probe

Kubernetes tiene una segunda sonda llamada la sonda de vida.

La sonda de vida está diseñada para controlar su contenedor y reiniciarlo cuando no es saludable.

Cuando tiene una pérdida de memoria en uno de sus contenedores, la aplicación podría rechazar las solicitudes entrantes o devolver malas respuestas.

Podría tener una sonda de preparación configurada para separar el contenedor que falló del Servicio.

Sin embargo, el contenedor (y el Pod) todavía se contabilizarán en las réplicas totales para su deployment.

Si tiene un deployment con tres réplicas y cada una de ellas tiene un solo contenedor, una no podrá recibir tráfico y Kubernetes no realizará ninguna otra acción.

A menos que la sonda de vida esté fallando.

En tal escenario, Kubernetes intenta reiniciar el contenedor dentro del Pod con la esperanza de que sea suficiente para solucionar el problema.

Puede agregar una sonda de vida a sus contenedores de esta manera:

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
    - name: sise
      image: learnk8s/demo:1.0.0
      ports:
        - containerPort: 3000
      livenessProbe:
        initialDelaySeconds: 2
        periodSeconds: 5
        httpGet:
          path: /health
          port: 3000
```

Implementación de sondas de preparación y vida.

Las sondas de vitalidad y disponibilidad son totalmente independientes.

Podrías tener una sonda de preparación pero no una vida.

O viceversa. O ninguno de ellos. O ambos.

Cuando tienes ambos, todavía son independientes.

La sonda de vida no espera a que una sonda de preparación retire el contenedor del Servicio antes de reiniciarlo.

Si no tiene cuidado, puede reiniciar el contenedor mientras aún está conectado al Servicio y está recibiendo tráfico en vivo.

La mayoría de las veces, expondrá un punto final HTTP de su contenedor para cada una de sus sondas.

Es común que los puntos finales del API /`health` y /`ready` devuelvan 200 cuando el contenedor está en buen estado y listo respectivamente.

Sin embargo, no siempre tiene sentido tener puntos finales HTTP.

Si empaqueta una base de datos MySQL dentro de un contenedor, no desea agregar un servidor web con dos puntos finales solo para las sondas de preparación y disponibilidad.

Puede decirle a Kubernetes que ejecute un comando dentro del contenedor en lugar de solicitar un punto final HTTP.

En el caso de su contenedor MySQL, podría tener una sonda de preparación (**readiness probes**) como esta:

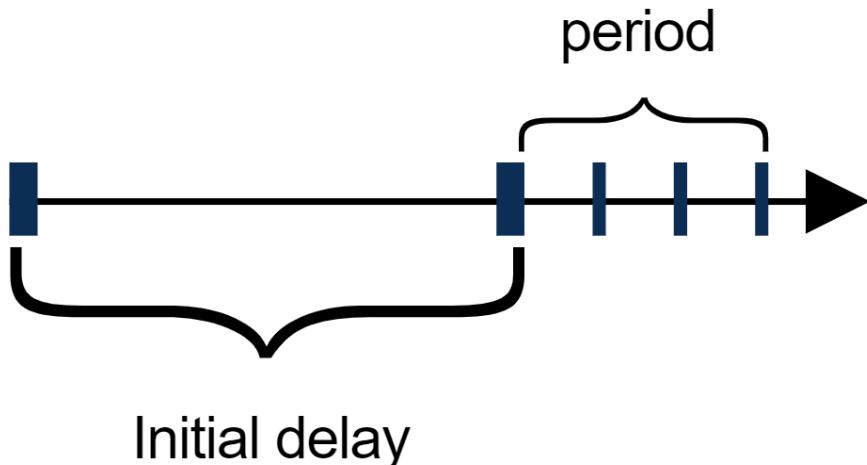
```
pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
  - name: mysql
    image: mysql:5.7
    readinessProbe:
      exec:
        command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
      initialDelaySeconds: 5
      periodSeconds: 2
      timeoutSeconds: 1
```

La sonda ejecuta un comando dentro del contenedor para verificar si la base de datos está lista.

Las sondas de actividad y disponibilidad verifican el estado de sus contenedores a intervalos regulares.

Puede personalizar la frecuencia con la que debe ejecutarse cada sonda y si debe esperar un retraso antes de comprobarlo por primera vez.

También puede personalizar la cantidad de respuestas exitosas que debe recibir antes de considerar que el contenedor es saludable o no.



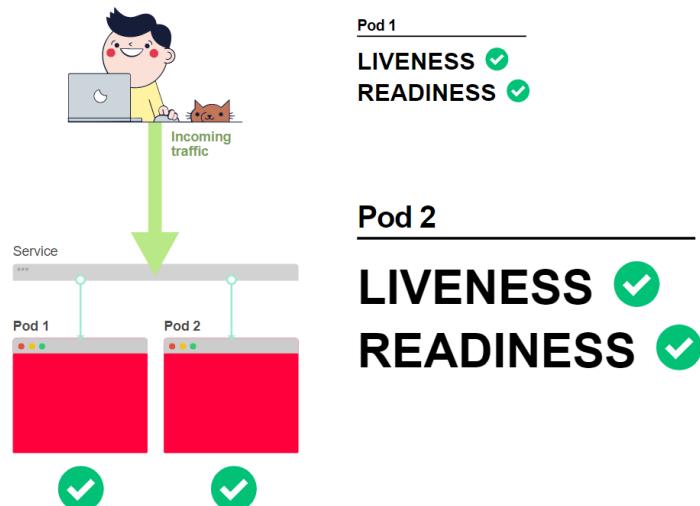
1. Las sondas comienzan a monitorear su Pod cuando se inicia su contenedor.
2. Si sabe que la aplicación no está lista y no tiene sentido realizar una prueba, puede establecer un retraso inicial.
3. Despues de la demora inicial, la sonda verifica el contenedor a intervalos regulares (el valor predeterminado es 10 segundos).
4. Despues de la demora inicial, la sonda verifica el contenedor a intervalos regulares (el valor predeterminado es 10 segundos).
5. Puede personalizar el período a cualquier valor (tenga en cuenta que el valor mínimo es de 1 segundo).

También puedes personalizar:

- El número de segundos después de los cuales la sonda se agota con **timeoutSeconds**
- Los éxitos consecutivos mínimos para que la sonda se considere exitosa después de haber fallado con **successThreshold**
- El número de reintentos consecutivos antes de considerar una sonda como fallida **failureThreshold**

Sondas de vitalidad y disponibilidad.

El siguiente es un ejemplo de una sonda de preparación y vida en la práctica.



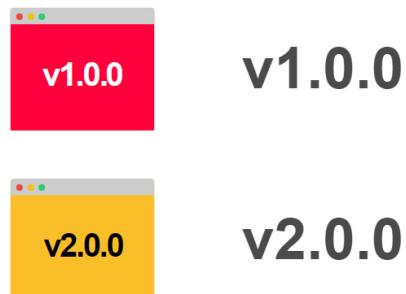
1. Usted creó un deployment con dos réplicas. El tráfico se enruta utilizando un servicio. Ambas aplicaciones tienen buena preparación y sondeo de vida.
2. La aplicación recibe demasiado tráfico y comienza a devolver errores. Dado que los desarrolladores no quieren dar una mala respuesta, diseñaron la sonda de preparación para devolver una respuesta fallida cuando eso sucede.
3. Kubernetes monitorea la respuesta de la sonda de preparación y se da cuenta de que no es saludable. Desconecta la aplicación del Servicio y ningún tráfico llega a ese contenedor.
4. La aplicación se encontró en un estado irrecuperable. No hay nada que puedas hacer aparte de reiniciarla. Nuevamente, los desarrolladores diseñaron la prueba de vida para fallar cuando se produce una excepción irrecuperable.
5. Kubernetes vigila las sondas de vida y observa que no está sano. Decide reiniciar el contenedor.
6. El contenedor vuelve a subir y se ve saludable. La sonda de vitalidad y preparación también son saludables.
7. Dado que la sonda de vida es verde, Kubernetes no reinicia más el contenedor.
8. Y como la sonda de preparación está en buen estado, Kubernetes vuelve a colocar el contenedor en el balanceador de carga.
9. Y como la sonda de preparación está en buen estado, Kubernetes vuelve a colocar el contenedor en el balanceador de carga.

Las sondas de preparación y vida son aún más críticas cuando decide enviar características a producción.

Rolling updates

Imaginemos que tienes tu aplicación desplegada en Kubernetes.

Desea actualizar la aplicación a la versión 2.0 y desea hacerlo con una actualización de cero tiempo de inactividad.

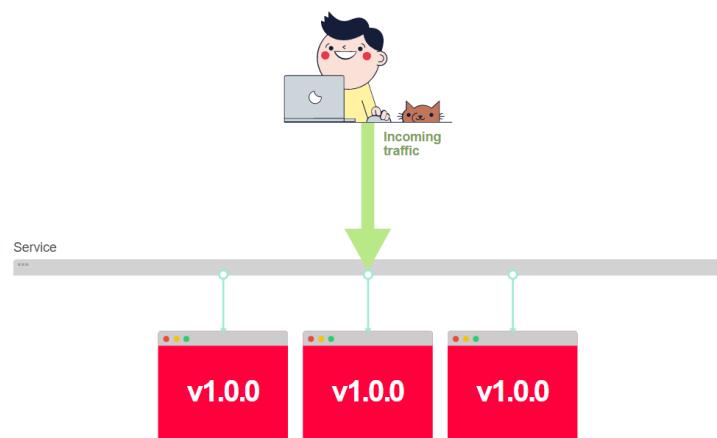


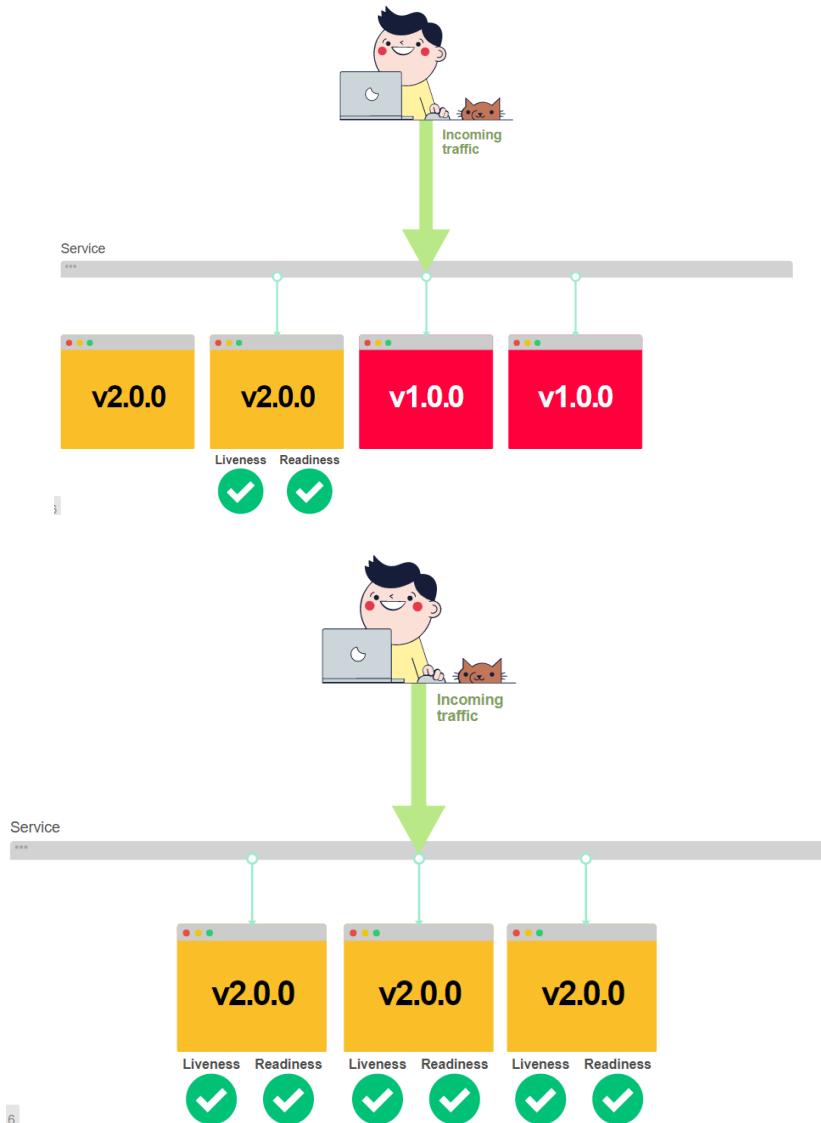
La opción más fácil a su disposición es agregar la versión más reciente de la aplicación junto con las existentes y eliminar una antigua.

Si repite los mismos pasos para todas las instancias en su ReplicaSet, eventualmente migrará su carga de trabajo de producción de la versión uno a la versión dos.

Reemplazar Pods uno a la vez generalmente se conoce como actualización continua.

Echemos un vistazo a un ejemplo:





1. Tiene un servicio y un deployment con tres réplicas en la versión 1.0.0. Cambia la `image` en su deployment a la versión 2.0.0, esto es lo que sucede a continuación.
2. En una actualización progresiva, Kubernetes crea un Pod con una nueva versión de la imagen.
3. Kubernetes espera la preparación y la indiferencia de la sonda. Cuando ambos están en buen estado, el Pod se está ejecutando y puede recibir tráfico.
4. Se quita el Pod anterior y Kubernetes está listo para comenzar de nuevo.
5. Se crea otro Pod con la imagen actual.
6. Kubernetes espera la preparación y la indiferencia de la sonda. Cuando ambos están en buen estado, el Pod se está ejecutando y puede recibir tráfico.
7. Se quita el Pod anterior.
8. Y por última vez, se crea un Pod con la imagen actual.
9. Kubernetes espera la preparación y la indiferencia de la sonda. Cuando ambos están en buen estado, el Pod se está ejecutando y puede recibir tráfico.
10. La migración de la versión anterior a la actual está completa.

En la actualización continua que se muestra en la imagen anterior, tiene tres réplicas del mismo Pod.

Cuando decide cambiar la versión de su imagen en su deployment y usar la versión dos, esto es lo que sucede.

- Kubernetes creará un nuevo Pod con la última versión.
- Kubernetes esperará a que la sonda de vida esté saludable
- Tan pronto como la sonda de disponibilidad pasa, el Pod se adjunta al Servicio y está listo para recibir tráfico, el Pod está listo
- Kubernetes tomará uno de los Pods en el Despliegue

Los mismos pasos se repiten para cada otro Pod en el deployment hasta que se complete el despliegue.

Las actualizaciones continuas son excelentes cuando desea ofrecer funciones en producción de forma incremental.

Las nuevas instancias sirven gradualmente el tráfico en vivo.

Tenga en cuenta que en cualquier momento su usuario podría recibir tráfico de la versión actual o anterior.

Las actualizaciones continuas no son una estrategia de despliegue adecuada si tiene cambios de última hora.

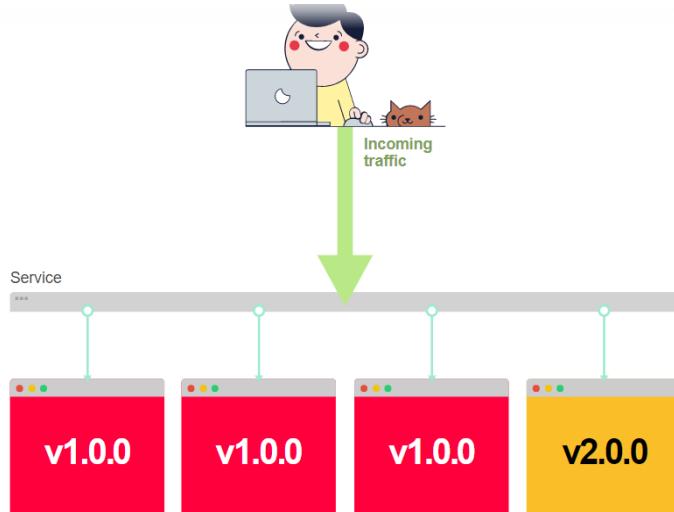
Canary deployments

Otra opción para desplegar en producción sin interrumpir el tráfico en vivo es usar una implementación de Canary.

Las implementaciones de Canary son similares a las actualizaciones sucesivas, en el sentido de que su tráfico en vivo llega a la versión actual y anterior de la aplicación.

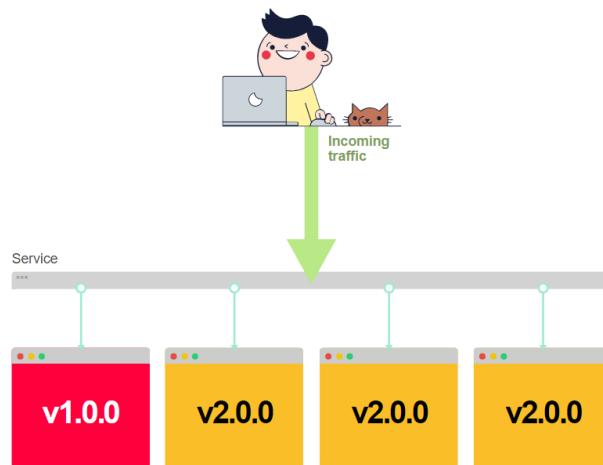
Pero mientras que con una actualización progresiva las actualizaciones se ejecutan una después de la otra, **con una implementación canary tiene dos versiones de su aplicación (actual y anterior) implementadas al mismo tiempo durante el tiempo que deseé.**

Es una práctica común crear solo unos pocos Pods con la versión actual y monitorear el tráfico en vivo que llega a ellos.



En este ejemplo, el 25% del tráfico llega a la versión actual y el 75% de la anterior.

Si todo sale según lo planeado, puede aumentar el número de réplicas en su versión actual y servir gradualmente más tráfico mientras disminuye las réplicas de la versión anterior de la aplicación.



En este ejemplo, el tráfico se balancea a favor de la versión actual: el 75% del tráfico llega a la versión actual y el 25% la anterior.

El proceso de enrutar el tráfico progresivamente a los Pods más nuevos es manual, y puede ajustar el tiempo que desea seguir haciéndolo.

En una implementación de Canary, desea que sus Servicios puedan enrutar el tráfico a dos conjuntos de Pods: actual y anterior.

Pero, ¿cómo hace un Servicio para dirigir el tráfico al Pod correcto?

Usando selectores y etiquetas

Los Pods en Kubernetes se pueden configurar con pares de valor-clave arbitrarios denominados etiquetas.

Las etiquetas son convenientes porque podría etiquetar sus Pods con un par clave-valor, como **component: frontend** para un componente de front-end.

Y se podría utilizar una estrategia similar para un componente de **backend: component: backend**.

Puedes tener cualquier número de etiquetas.

De hecho, podría etiquetar sus Pods con una etiqueta para la versión como esta:

```
pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-labels
  labels:
    component: frontend
    version: "1.0.0"
spec:
  containers:
  - name: myapp-container
    image: busybox
```

Los servicios pueden dirigir el tráfico a Pods cuando coinciden con un selector.

Se utiliza un selector para apuntar Pods con una etiqueta específica como **component: frontend**.

```
service.yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    component: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

El Servicio anterior enruta el tráfico a todos los Pods que tienen una **component: fronten** de etiqueta.

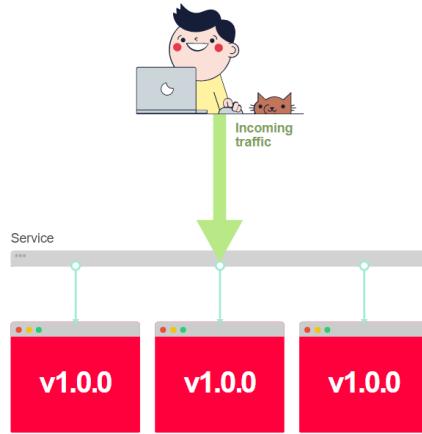
Cuando un conjunto de Pods comparte la misma etiqueta, el Servicio enruta el tráfico a todos ellos, incluso si los Pods pertenecen a deployments diferentes.

Uso de etiquetas y selectores con despliegues canarys.

Podría usar selectores y etiquetas para crear un deployment de tipo Canary y enrutar el tráfico a dos conjuntos diferentes de Pods.

Debe crear dos deployments: una para la aplicación existente con Pods con una etiqueta **version: 1.0.0** y otra para la nueva aplicación con una etiqueta **version: 2.0.0**

Cuando se inicia, el 100% del tráfico se enruta a la aplicación existente.



El selector de servicio apunta a **version: 1.0.0**.

service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: canary-service
spec:
  selector:
    version: "1.0.0"
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

Ahora decide balancear una fracción del tráfico en vivo a la nueva versión.

Con una sola instancia de la aplicación actual y tres instancias de la anterior, el 75% del tráfico se enrutará a la aplicación existente y solo el 25% a la última versión.

Pero necesita una etiqueta compartida para que el selector de su Servicio dirija el tráfico a ambos.

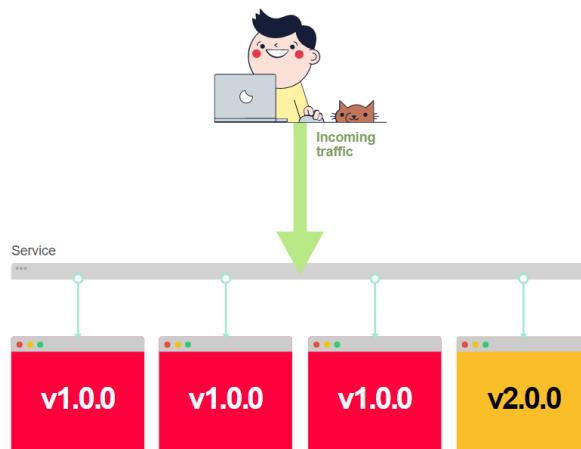
Puede agregar una etiqueta **component: frontend** a ambos y tener el Nombre de servicio de destino en lugar de la versión.

service.yaml

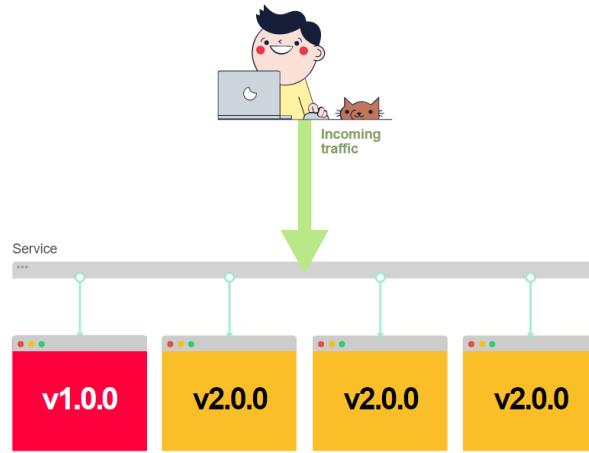
```
kind: Service
apiVersion: v1
metadata:
```

```
name: canary-service
spec:
  selector:
    component: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Tan pronto como realice el cambio, el Servicio enruta el tráfico a la aplicación actual (con suerte) una vez cada cuatro visitas.



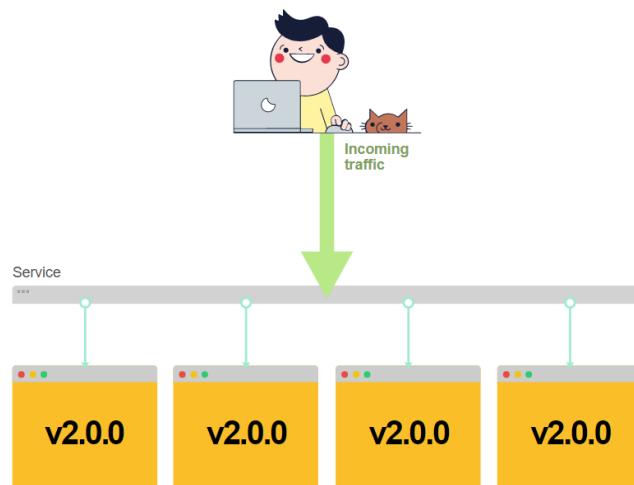
Si está satisfecho con el resultado, puede mover gradualmente su tráfico de producción a la nueva versión aumentando las réplicas en el deployment y disminuyendo el conteo anterior.



Cuando esté dispuesto a cambiar todo el tráfico, puede editar el Servicio y señalarlo en `version: 2.0.0` en lugar de la etiqueta compartida.

service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: canary-service
spec:
  selector:
    version: "2.0.0"
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```



Los servicios hacen de balanceador dinámico de carga.

Puede reconfigurar dinámicamente el Servicio para enrutar el tráfico a un conjunto de Pods ajustando el selector y las etiquetas en sus Pods.

Blue-green deployments

Las actualizaciones continuas y las implementaciones de Canary son excelentes estrategias para introducir actualizaciones incrementales.

Sin embargo, hay momentos en los que tiene que introducir cambios de última hora en su API o aplicación.

En ese caso, no puede tener dos versiones de su aplicación en vivo al mismo tiempo, ya que romperá el contrato existente con sus usuarios.

Una estrategia más adecuada para implementar cambios de última hora es utilizar blue-green Deployment

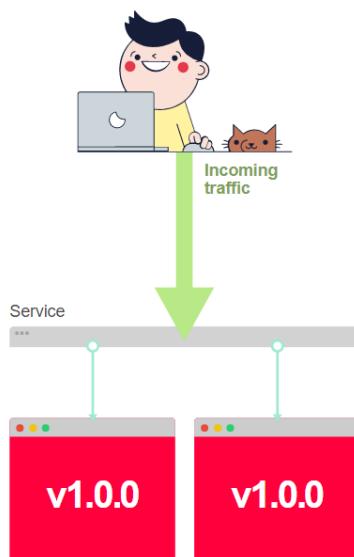
En una deployment **blue-green** crea una copia de su aplicación pero con la versión más reciente instalada y luego hace que el Servicio existente apunte a la última versión.

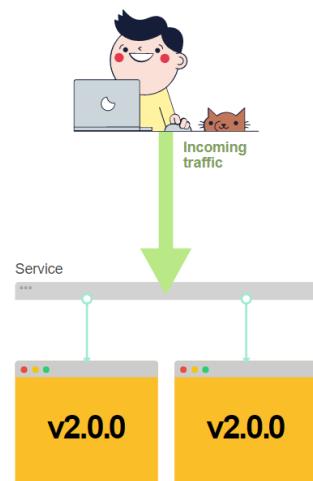
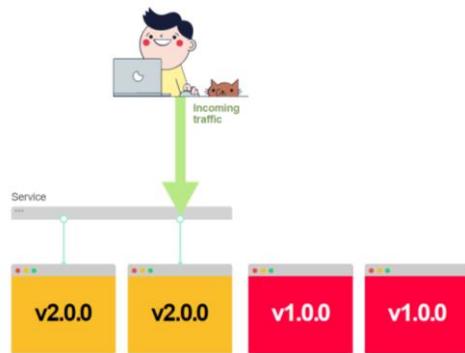
Comienzas con un servicio y un deployment.

El selector para los Pods está apuntando a `version: 1.0.0`.

A continuación, crea una segunda deployment con la aplicación más nueva y donde los Pods tienen una etiqueta de `version: 2.0.0`

Cuando esté listo, se cambia el selector en el Servicio de `version: 1.0.0` a `version: 2.0.0` y de repente todo su tráfico se sirve en el nuevo deployment





1. Comienzas con un deployment y servicio. Todo el tráfico se enruta a la versión anterior.
2. Se crea otro Despliegue. Una copia exacta de la anterior.
3. Cambia el selector en el Servicio para orientar la Implementación actual v2.0.0.
4. Eliminaremos el deployment anterior v1.0.0

Tenga en cuenta que si tiene un deployment con cientos de réplicas, cuando realice una blue-green Deployment tendrá el doble de Pods hasta que destruya el deployment anterior. Asegúrate de tener suficiente capacidad, para poder utilizar este método de deployment.

Deshacer un Deployment

Incluso si utiliza técnicas como las actualizaciones continuas, las implementaciones de Canary y Blue-green, todavía existe el riesgo de que su aplicación no funcione en producción de la forma que espera.

Cuando introduce un cambio que rompe producción, debe tener un plan para revertir ese cambio.

Kubernetes y `kubectl` ofrecen un mecanismo simple para revertir los cambios a recursos tales como deployments.

Cuando creas o modificas un recurso con `kubectl`, puedes agregar la bandera `--record`

A partir de ese momento, cada cambio en el recurso YAML se registra en una tabla inmutable.

Revision	What happened
1	<code>kubectl create -f deployment.yaml</code>
2	<code>kubectl scale --replicas=5 deploy/app</code>
3	<code>kubectl set image app=v2.0.0</code>

1. Cuando usas la bandera `--record`, cada acción es parte de la historia.
2. La primera fila de la tabla es cómo se creó el recurso
3. Si decide cambiar el recurso, por ejemplo, escalar una implementación, el cambio se almacena en el historial.
4. Si actualiza el deployment para usar una nueva imagen, eso también se registra.

Lo que es aún mejor es que puedes saltar a cualquier punto en el historial cuando el recurso fue creado o modificado.

Retroceder en el tiempo no reescribirá el historial (¡aunque sea!), Pero crea una nueva entrada en la tabla inmutable.

Revision	What happened
1	kubectl create -f deployment.yaml
2	kubectl scale --replicas=5 deploy/red
3	kubectl set image app=v2.0.0

1. La actualización a una nueva versión del deployment podría haber introducido una nueva entrada.
2. Puedes usar kubectl rollout undo deployment/app --to-revision=1 para volver a la primera versión. Se crea una nueva fila en la historia.

Revision	What happened
1	kubectl create -f deployment.yaml
2	kubectl scale --replicas=5 deploy/red
3	kubectl set image app=green
4	kubectl create -f deployment.yaml

Puede usar el siguiente comando para revisar una revisión previa de un deployment:

```
bash
kubectl rollout undo deployment/app --to-revision=1
```

Laboratorio

Antes de que empieces

Debes haber instalado:

- docker
- minikube
- kubectl

Además, antes de comenzar, familiarícese con las siguientes imágenes de contenedores:

- learnk8s/app:1.0.0
- learnk8s/app:2.0.0
- learnk8s/app:3.0.0

Puedes probar esas imágenes con:

bash

```
docker run -dti -p 8080:8080 learnk8s/app:1.0.0
```

La aplicación está disponible en <http://localhost: 8080> .

La aplicación expone un punto final de salud en <http://localhost: 8080/health> .

Usted debe probar learnk8s/app:2.0.0 y learnk8s/app:3.0.0

El primer contenedor tiene un fondo rojo liso.

El segundo contenedor tiene un fondo amarillo.

El tercer contenedor tiene un fondo verde.

Rolling updates

Las actualizaciones continuas permiten que la aplicación se actualice sin ningún tiempo de inactividad, un Pod en ese momento.

En esta sección, realizará una actualización sucesiva: actualizará la aplicación de la versión `1.0.0` a `2.0.0` sin interrumpir la conexión.

Creación de un Deployment

Debe crear un deployment para la aplicación.

Crea un archivo `deployment.yaml` con el siguiente contenido:

```
deploy.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: app
spec:
  minReadySeconds: 10
  replicas: 1
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
        - name: web
          image: learnk8s/app:1.0.0
          ports:
            - containerPort: 8080
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 1
        timeoutSeconds: 1
        periodSeconds: 5
      livenessProbe:
        httpGet:
```

```

path: /health
port: 8080
initialDelaySeconds: 1
timeoutSeconds: 1
periodSeconds: 5

```

Puedes crear el recurso con:

```

bash
kubectl create -f deployment.yaml

```

La mayor parte del contenido debe ser familiar:

- Es una definición para un despliegue.
- La imagen utilizada para los Pods es. learnk8s/app:1.0.0
- El contenedor expone el puerto 8080.
- solo hay una réplica

Puede verificar que la implementación fue exitosa con:

```

bash
kubectl get pods

```

Debes prestar atención a las dos nuevas secciones: readinessProbe y livenessProbe.

Las sondas son independientes, pero pueden consumir el mismo punto final.

Readiness probe (Sonda de preparación)

La sonda de preparación se utiliza para probar si la aplicación puede recibir tráfico.

Si su aplicación tarda algún tiempo en iniciarse antes de que pueda recibir tráfico, puede usar la sonda de preparación para indicar cuándo está lista.

Liveness probe (Sonda de vitalidad)

La sonda de vida se utiliza para indicar que la aplicación se inició correctamente.

Si su aplicación no puede iniciarse porque quizás no pudo obtener una configuración correctamente, puede usar la sonda de activación para indicar que el contenedor debe reiniciarse.

Creando un servicio

Puede exponer la Implementación con un Servicio.

Crea un archivo `service.yaml` con el siguiente contenido:

```
service.yaml
apiVersion: v1
kind: Service
metadata:
  name: entry-point
spec:
  ports:
    - port: 80
      targetPort: 8080
  type: NodePort
  selector:
    name: app
```

Puedes crear el recurso con:

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

Puedes verificar que tu servicio fue expuesto correctamente con:

```
bash
minikube service entry-point
```

Debería ver una página web con un fondo rojo.

Escalar la aplicación

Antes de comenzar, escala la implementación a cinco réplicas:

```
bash
kubectl edit deployment app
```

Y cambia las réplicas para que sean `replicas: 5`.

Seguimiento de las actualizaciones sucesivas.

Verá el clúster a medida que actualiza el contenedor en tiempo real.

Recupere la URL del servicio actual con:

```
bash
SERVICE_URL=$(minikube service entry-point --url=true)
```

Puede solicitar la aplicación implementada actual cada segundo con:

```
bash
while true; do printf "%s\n" $(curl -s $SERVICE_URL) | grep h1; sleep .5;
done
```

Debería poder ver la aplicación respondiendo con la versión 1.0.0.

En otra terminal, puede editar la Implementación y actualizar el contenedor a una versión más nueva de la aplicación:

```
bash
kubectl edit deployment app
```

Y cambiando la imagen sea: `image: learnk8s/app:2.0.0`

Kubernetes comienza a actualizar los Pods.

Puede inspeccionar el estado del despliegue con:

```
bash
kubectl rollout status deploy/app
```

Una vez que se completa la implementación, la aplicación debe ejecutarse en la versión 2.0.0.

Puedes verificarlo con:

```
bash
minikube service entry-point
```

Observe cómo las actualizaciones continuas y las implementaciones de tiempo de inactividad cero forman parte de la Implementación.

No tenía que codificar ninguna lógica para realizar una actualización sucesiva.

Más actualizaciones continuas

¿Puede realizar una actualización sucesiva y actualizar la aplicación a la versión 3.0.0?

Eliminar

Debe eliminar las implementaciones existentes con:

```
bash  
kubectl delete deployment/app
```

Puedes eliminar el Servicio con:

```
bash  
kubectl delete service entry-point
```

Services and selectors

En Kubernetes, puede asignar etiquetas a recursos como Pods, Deployments o Servicios.

Las etiquetas son dinámicas: puede agregarlas y eliminarlas incluso después de crear el recurso.

En esta sección, aprenderá a usar los selectores en Servicios para enrutar el tráfico a un subconjunto de Pods.

Desplegando las aplicaciones

Para empezar, vamos a crear dos deployments: uno para la versión 1.0.0 y otro para la versión 2.0.0.

Despliegue de la versión 1.0.0

Debes crear un archivo `deployment-v1.yaml` con el siguiente contenido:

```
despliegue-v1.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: v1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: app
        version: "1.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:1.0.0
      ports:
        - containerPort: 8080
  readinessProbe:
    httpGet:
      path: /health
      port: 8080
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
```

Puedes crear el recurso con:

```
bash
kubectl create -f deployment-v1.yaml
```

Despliegue de la versión 2.0.0

Debes crear un archivo `deployment-v2.yaml` con el siguiente contenido:

```
despliegue-v2.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: v2
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: app
        version: "2.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:2.0.0
          ports:
            - containerPort: 8080
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
      livenessProbe:
        httpGet:
          path: /health
          port: 8080
```

Puedes crear el recurso con:

```
bash
kubectl create -f deployment-v2.yaml
```

Debes verificar que las implementaciones fueron exitosas con:

```
bash
kubectl get pods
```

Distribuyendo tráfico a una aplicación a la vez.

Puedes exponer el Deployment `v1` con un Servicio.

Crea un archivo `service.yaml` con el siguiente contenido:

```
service.yaml
apiVersion: v1
kind: Service
metadata:
  name: entry-point
spec:
  ports:
    - port: 80
      targetPort: 8080
    type: NodePort
    selector:
      version: "1.0.0"
```

Debes visitar la página y confirmar que funciona:

```
bash
minikube service entry-point
```

Puede apuntar el mismo Servicio a la implementación de la versión `2.0.0` cambiando el sector para que sea `version: "2.0.0"`:

```
bash
kubectl edit service entry-point
```

Debes actualizar tu navegador o escribir de nuevo:

```
bash
minikube service entry-point
```

Kubernetes está sirviendo la versión `2.0.0` de su aplicación.

Observe cómo el Servicio es un conveniente balanceador de carga que puede apuntar a Pods.

Cuando el selector estaba apuntando a la versión `1.0.0`, el tráfico se servía desde Pods desde la versión `1.0.0`.

Cuando el selector cambió a punto a versión `2.0.0`, el tráfico se servía solo desde Pods con versión `2.0.0`.

Distribuyendo tráfico a ambas aplicaciones.

Pero ambas implementaciones comparten una etiqueta de tipo `name: app`.

Si cambia el selector para que apunte a `name: app` usted, puede servir el tráfico de ambas deployments.

Edita el servicio y cambia el selector para que esté `name: app` con:

bash

```
kubectl edit service entry-point
```

El contenido debería verse así:

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: entry-point
spec:
  ports:
  - port: 80
    targetPort: 3000
  type: NodePort
  selector:
    name: app
```

Actualice la página varias veces para ver las páginas de la versión `1.0.0` y `2.0.0` aparecen intermitentemente.

Si está usando Chrome y no puede ver la página que aparece de forma intermitente, intente nuevamente usando `curl`. Chrome almacena en caché las solicitudes en gran medida.

Puedes usar etiquetas para seleccionar pods `kubectl get pods` también:

bash

```
kubectl get pods -l version=1.0.0
kubectl get pods -l version=2.0.0,name=app
```

Laboratorio

Crear un deployment para la versión 3.0.0.

- El despliegue debe tener una réplica.
- la imagen debe ser learnk8s/app:3.0.0
- El Despliegue debe tener sondas de preparación y vida.
- Los Pods deben tener las siguientes etiquetas: version: 3.0.0 y region: uk-south

¿Se puede cambiar el selector de servicios para que el tráfico se enrute solo a la versión 2.0.0 y 3.0.0?

Eliminar

Debe eliminar las implementaciones existentes con:

```
bash  
kubectl delete deployment/v1  
kubectl delete deployment/v2
```

Puedes eliminar el Servicio con:

```
bash  
kubectl delete service entry-point
```

Canary deployments

Las implementaciones de Canary son útiles si desea probar una característica con un pequeño subconjunto de sus usuarios de producción.

La estrategia consiste en crear una segunda implementación y enrutar una parte más pequeña del tráfico de producción hacia ella.

Asegúrese de no tener ningún Despliegue o Servicio desplegado en su grupo antes de probar el despliegue del canary.

Puede listar las implementaciones en su clúster con:

```
bash  
kubectl get deployments
```

Y puedes eliminar un solo despliegue con:

```
bash  
kubectl delete deployment <deployment_name>
```

Puede listar los Servicios en el cluster con:

```
bash  
kubectl get services
```

Puede eliminar un solo servicio con:

```
bash  
kubectl delete service <service_name>
```

Despliegue de la aplicación de producción.

Debes desplegar la versión 1.0.0 de tu aplicación.

Crea un `deployment-prod.yaml` archivo con el siguiente contenido:

despliegue-prod.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: v1
spec:
  minReadySeconds: 10
  replicas: 3
  template:
    metadata:
      labels:
        name: app
        version: "1.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:1.0.0
      ports:
        - containerPort: 8080

  readinessProbe:
    httpGet:
      path: /health
      port: 8080
    initialDelaySeconds: 1
    timeoutSeconds: 1
    periodSeconds: 5

  livenessProbe:
    httpGet:
      path: /health
      port: 8080
    initialDelaySeconds: 1
    timeoutSeconds: 1
    periodSeconds: 5
```

Puedes crear el recurso en Kubernetes con:

```
bash  
kubectl create -f deployment-prod.yaml
```

Puedes verificar que tres Pods se estén ejecutando con:

```
bash  
kubectl get pods
```

Y crea un archivo `service.yaml` con el siguiente contenido:

```
service.yaml  
apiVersion: v1  
kind: Service  
metadata:  
  name: entry-point  
spec:  
  ports:  
    - port: 80  
      targetPort: 8080  
  type: NodePort  
  selector:  
    version: "1.0.0"
```

Puedes crear el recurso con:

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

Puedes verificar que tu servicio fue expuesto correctamente con:

```
bash  
minikube service entry-point
```

Deploying the canary

Debes crear un Despliegue para la aplicación canary.

Deberías usar la versión 2.0.0 para el deployment canary.

Crea un archivo `deployment-canary.yaml` con el siguiente contenido:

despliegue-canary.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: canary
spec:
  minReadySeconds: 10
  replicas: 1
  template:
    metadata:
      labels:
        name: app
        version: "2.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:2.0.0
          ports:
            - containerPort: 8080
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 1
        timeoutSeconds: 1
        periodSeconds: 5
      livenessProbe:
        httpGet:
          path: /health
          port: 8080
        initialDelaySeconds: 1
        timeoutSeconds: 1
        periodSeconds: 5
```

Puede desplegar el recurso a Kubernetes con:

```
bash
kubectl create -f deployment-canary.yaml
```

Debes verificar que se estén ejecutando cuatro Pods:

- Tres pods ejecutan versión 1.0.0
- Un Pod ejecuta la versión 2.0.0 de la aplicación

Los cuatro pods comparten la misma `name: app` etiqueta.

Puedes usar el siguiente comando:

```
bash
kubectl get pods --show-labels
```

Monitoreo del tráfico en vivo

Puedes acceder al servicio con:

```
bash
minikube service entry-point
```

Debe verificar que, independientemente de la cantidad de veces que actualice la página, siempre verá la versión 1.0.0 implementada.

Recupere la URL del servicio actual con:

```
bash
SERVICE_URL=$(minikube service entry-point --url=true)
```

Debes monitorear la respuesta de la aplicación en vivo:

```
bash
while true; do printf "%s\n" $(curl -s $SERVICE_URL) | grep h1; sleep .5;
done
```

Deberías poder ver la aplicación respondiendo 1.0.0 todas las veces.

Sólo la versión 1.0.0 de la aplicación está recibiendo tráfico activamente.

Exposing the canary deployment

En otro terminal, cambie el Servicio para que apunte a la versión `1.0.0` y `2.0.0` de la aplicación con:

```
bash
kubectl edit service entry-point
```

El selector debe ser `name: app`.

Dado que todos los Pods comparten la misma etiqueta, el tráfico se cargará en los cuatro Pods.

Y como hay tres réplicas de la aplicación en la versión `1.0.0` y una para el canary, cuando actualice la página, debería ver la versión `1.0.0` 3 de cada 4 veces.

Debería ver el canary por el resto (75% -25% dividido).

Balancing the split

Escale el despliegue de canary a 3 réplicas con:

```
bash
kubectl edit deployment canary
```

y actualicé las réplicas a `replicas: 3`.

Actualice la página varias veces: debería ver la versión `1.0.0` y `2.0.0` aparecer con la misma frecuencia (división 50% -50%).

Acabas de realizar una actualización de canary.

Inicialmente, introdujo un cambio (una versión actualizada de la aplicación) a solo un subconjunto de sus usuarios (75% -25% dividido).

Luego aumentó gradualmente la división a medida que ganó confianza en que la función no interrumpiría la producción (división del 50% -50%).

Eventualmente, puede enrutar el 100% del tráfico a la versión `2.0.0`.

Laboratorio

Crear una implementación para la versión 3.0.0.

- El despliegue debe tener una réplica.
- la imagen debe ser learnk8s/app:3.0.0
- El Despliegue debe tener sondas de preparación y vida.
- Los Pods deben tener las siguientes etiquetas: version: 3.0.0 y region: uk-south

¿Se puede cambiar el selector para el servicio, para que pueda enrutar el tráfico a la versión 1.0.0, 2.0.0 y 3.0.0 con la siguiente división?

- 1.0.0 50%
- 2.0.0 25%
- 3.0.0 25%

Eliminar

Debe eliminar las implementaciones existentes con:

```
bash
kubectl delete deployment/v1
kubectl delete deployment/canary
```

Puedes eliminar el Servicio con:

```
bash
kubectl delete service entry-point
```

Blue-green deployments

Los deployments blue-green es una técnica que reduce el tiempo de inactividad y el riesgo al ejecutar dos entornos de producción idénticos llamados Azul y Verde.

En cualquier momento, solo uno de los entornos recibe tráfico.

Antes de continuar, debe eliminar todos los deployments anteriores.

Puede listar las implementaciones en su clúster con:

```
bash  
kubectl get deployments
```

Y puedes eliminar un solo despliegue con:

```
bash  
kubectl delete deployment <deployment_name>
```

Puede listar los Servicios en el cluster con:

```
bash  
kubectl get services
```

Puede eliminar un solo servicio con:

```
bash  
kubectl delete service <service_name>
```

Despliegue de la aplicación de producción.

La aplicación web principal que recibe tráfico de producción se ejecuta en la versión 1.0.0.

Crea un archivo `deployment-blue.yaml` con el siguiente contenido:

```
deploy-blue.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: blue
spec:
  minReadySeconds: 10
  replicas: 3
  template:
    metadata:
      labels:
        name: app
        version: "1.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:1.0.0
      ports:
        - containerPort: 8080
  readinessProbe:
    httpGet:
      path: /health
      port: 8080
    initialDelaySeconds: 1
    timeoutSeconds: 1
    periodSeconds: 5
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
    initialDelaySeconds: 1
    timeoutSeconds: 1
    periodSeconds: 5
```

Puedes crear el recurso en Kubernetes con:

```
bash  
kubectl create -f deployment-blue.yaml
```

Puedes verificar que tres Pods se estén ejecutando con:

```
bash  
kubectl get pods
```

Y crea un archivo **service-blue-green.yaml** con el siguiente contenido:

```
service-blue-green.yaml  
apiVersion: v1  
kind: Service  
metadata:  
  name: entry-point  
spec:  
  ports:  
  - port: 80  
    targetPort: 8080  
  type: NodePort  
  selector:  
    version: "1.0.0"
```

Puedes crear el recurso con:

```
bash  
kubectl create -f service-blue-green.yaml
```

Puedes verificar que tu servicio fue expuesto correctamente con:

```
bash  
minikube service entry-point
```

Monitoring

Recupere la URL del servicio actual con:

```
bash  
SERVICE_URL=$(minikube service entry-point --url=true)
```

Escribe un bucle para solicitar la aplicación:

```
bash  
while true; do printf "%s\n" $(curl -s $SERVICE_URL) | grep h1; sleep .5;  
done
```

Debería poder ver la aplicación respondiendo con la versión 1.0.0.

Cree otra implementación para una versión más nueva de la aplicación: versión 2.0.0.

Debes crear un **deployment-green.yaml** archivo con el siguiente contenido:

```
deploy-green.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: green
spec:
  replicas: 3
  template:
    metadata:
      labels:
        name: app
        version: "2.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:2.0.0
          ports:
            - containerPort: 8080
      readinessProbe:
        httpGet:
          path: /health
          port: 8080
      livenessProbe:
        httpGet:
          path: /health
          port: 8080
```

Para enrutar todo el tráfico de la versión 1.0.0 a 2.0.0 y lograr un despliegue azul-verde, abra otra terminal.

Actualice su servicio para apuntar a la versión 2.0.0:

```
bash
kubectl edit service entry-point
```

y cambia el selector para que apunte a `version: "2.0.0"`.

Verifique que la aplicación que recibe el tráfico se ejecute en la versión 2.0.0:

```
bash
minikube service entry-point
```

En la ventana del terminal, debería notar un cambio brusco en las respuestas de la versión 1.0.0 a 2.0.0.

Acabas de realizar un despliegue blue-green.

Usted creó una copia exacta de la producción con algunos cambios nuevos (versión 2.0.0) y reconfiguró el balanceador de carga (Servicio) para que apunte a la nueva stack.

Laboratorio

Crear un deployment para la versión 3.0.0.

- El despliegue debe tener una réplica.
- la imagen debe ser learnk8s/app:3.0.0
- El Despliegue debe tener sondas de preparación y vida.
- Los Pods deben tener las siguientes etiquetas: version: 3.0.0

Debe actualizar su aplicación a la versión 3.0.0 utilizando una implementación azul-verde.

Eliminar

Debe eliminar las implementaciones existentes con:

```
bash
kubectl delete deployment/v1
kubectl delete deployment/canary
```

Puedes eliminar el Servicio con:

```
bash
kubectl delete service entry-point
```

Rolling back a deployment

Deshacer un despliegue

Asegúrese de no tener ningún Despliegue o Servicio desplegado en su grupo antes de probar el despliegue del canario.

Puede listar las implementaciones en su clúster con:

```
bash  
kubectl get deployments
```

Y puedes eliminar un solo despliegue con:

```
bash  
kubectl delete deployment <deployment_name>
```

Puede listar los Servicios en el cluster con:

```
bash  
kubectl get services
```

Puede eliminar un solo servicio con:

```
bash  
kubectl delete service <service_name>
```

Creando un despliegue

Debes desplegar la versión `1.0.0` de tu aplicación.

Crea un `deployment-prod.yaml` con el siguiente contenido:

```
despliegue-prod.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        name: app
        version: "1.0.0"
    spec:
      containers:
        - name: application
          image: learnk8s/app:1.0.0
      ports:
        - containerPort: 8080
  readinessProbe:
    httpGet:
      path: /health
      port: 8080
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
```

Crea el recurso con:

```
bash
kubectl create -f deployment.yaml --record
```

Tenga en cuenta la bandera `--record` estará al final

La bandera `--record` le permite grabar el comando actual como parte de los recursos que se crean o actualizan.

Puedes verificar que tres Pods se estén ejecutando con:

```
bash
kubectl get pods
```

Y crea un `service.yaml` con el siguiente contenido:

```
service.yaml
apiVersion: v1
kind: Service
metadata:
  name: entry-point
spec:
  ports:
  - port: 80
    targetPort: 8080
  type: NodePort
  selector:
    name: app
```

Puedes crear el recurso con:

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

Puedes verificar que tu servicio fue expuesto correctamente con:

```
bash
minikube service entry-point
```

Realice una actualización sucesiva y actualice la aplicación a la versión 2.0.0:

```
bash
kubectl edit deployment app
```

Cambia la imagen a `image: learnk8s/app:2.0.0`.

Debería verificar que la implementación fue exitosa y que la aplicación implementada es la versión 2.0.0:

```
bash
minikube service entry-point
```

Inspeccione el historial de despliegue con:

```
bash
kubectl rollout history deployment app
```

Vamos a revertir la implementación a la versión 1.0.0:

```
bash
kubectl rollout undo deployment app
```

Debe verificar que la aplicación que recibe el tráfico se ejecuta en la versión 1.0.0:

```
bash
minikube service entry-point
```

Implementó una actualización (versión 2.0.0), se dio cuenta de que no era lo que quería y regresó al estado original (versión 1.0.0).

Implementemos la versión 2.0.0 y 3.0.0 de la aplicación en ese orden:

```
bash
kubectl set image deployment app application=learnk8s/app:2.0.0
kubectl set image deployment app application=learnk8s/app:3.0.0
```

Inspeccione el historial de despliegue con:

```
bash
kubectl rollout history deployment app
```

Puede volver a un punto específico en el tiempo con **--to-revision**.

Volvamos a la primera implementación:

```
bash
kubectl rollout undo deployment app --to-revision=3
```

Se reversionó la aplicación a la primera implementación.

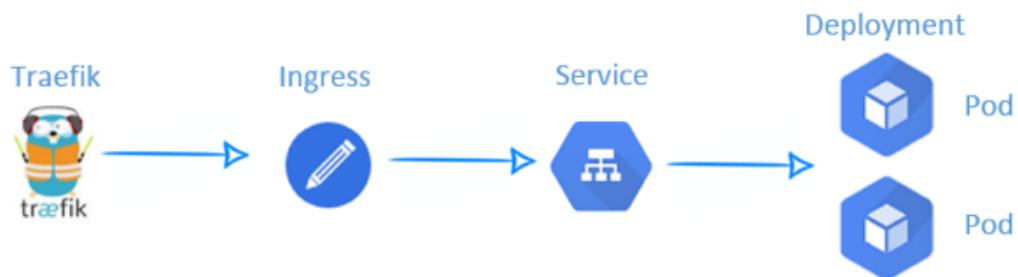
¿Se puede revertir la aplicación a la versión 3.0.0?

Lab instalacion Ingress-Controller

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

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

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



- Vamos a desplegar una pod que implementa un proxy inverso (en nuestro laboratorio, traefik,...) que esperará peticiones HTTP y HTTPS.
- Por lo tanto el `Ingress Controller` será el nodo del cluster donde se ha instalado el pod. En nuestro caso, para realizar el despliegue vamos utilizar un recurso de Kubernetes llamado `DaemonSet` que asegura que el despliegue se hace en todos los nodos del cluster y que los puertos que expone los pods (80 y 443) están mapeado y son accesible en los nodos. Por lo que cada uno de los nodos del cluster va a poseer una copia del `Ingress Controller`.
- No es necesario que al servicio al que accede el `Ingress Controller` este configurado del tipo *NodePort*.

Instalación de Ingress Controller en Kubeadm con Traefik

Siguiendo los pasos que obtenemos en la guía de usuario de [traefik](#) vamos a realizar la instalación del ingress controller.

Lo primero es crear las reglas de acceso (RBAC) necesarias (puedes ver el contenido del los ficheros yaml en la documentación):

```
kubectl apply -f
https://raw.githubusercontent.com/containous/traefik/v1.7/examples/k8s
/traefik-rbac.yaml
```

Como hemos comentado podemos desplegar el proxy utilizando un `Deployment` o un `DaemonSet`, hemos elegido esta última opción como indicábamos anteriormente:

```
kubectl apply -f
https://raw.githubusercontent.com/containous/traefik/v1.7/examples/k8s
/traefik-ds.yaml
```

Comprobamos el despliegue:

```
kubectl get ds -n kube-system
```

NAME	READY	CURRENT	UP-TO-DATE
AVAILABLE	AGE		
<code>...</code>			
traefik-ingress-controller	2	2	2
2	4h		

Comprobamos la creación de los pods:

```
kubectl get pods -n kube-system -o wide
```

traefik-ingress-controller-5m7t2 10.244.1.203	worker1	<none>	1/1	Running	28	32d
traefik-ingress-controller-ktfw2 10.244.2.151	worker2	<none>	1/1	Running	27	32d

Como vemos se ha creado un pod en cada nodo.

La IP de acceso a uno de los nodos es la 10.0.0.11, y 10.0.0.12, por lo que podemos hacer la prueba de que el proxy inverso está funcionando:

```
curl http://10.0.0.11/  
404 page not found
```

El dashboard en los dos nodos seria:

<http://10.0.0.11:8080/dashboard/>
<http://10.0.0.12:8080/dashboard/>