kubernetes

Introducción

Kubernetes nació en Google y fue donado en 2014 como proyecto Open Source, está escrito Go/Golang y su referencias base:

- Github: https://github.com/kubernetes/kubernetes
- Slack: slack.k8s.io

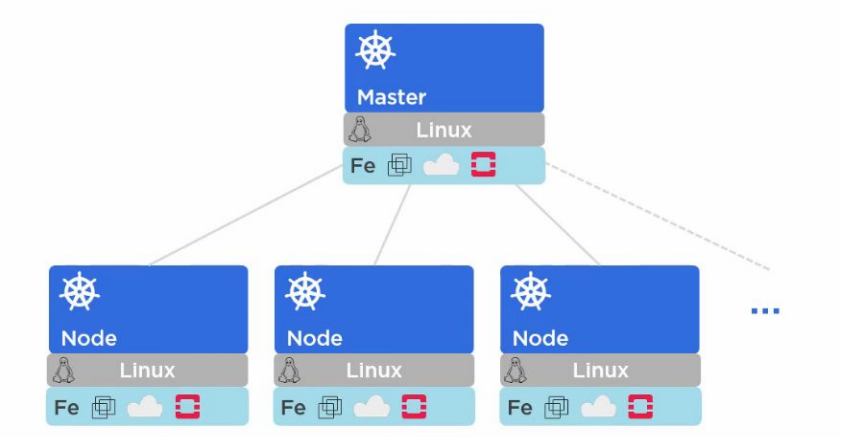
Es bastante común ver su abreviatura Kubernetes = K8s. Podríamos decir que K8s es un orquestador de microservicios para apps.

El modelo a gran rasgos está basado en:

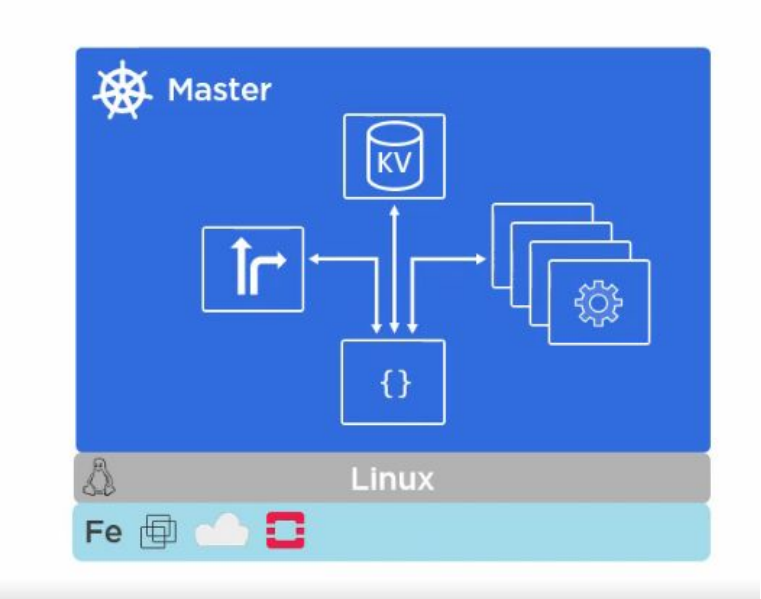
- Maestros
- Nodos
- Pods
- Servicios
- Despliegues
- Recaps

Maestros

K8s es efectivamente un conjunto de maestros y nodos, siendo un maestro un nodo más. Un nodo maestro en principio regula el funcionamiento de un conjunto de nodos.



Profundizando un poco más en el nodo maestro, éste es un conglomerado de partes.

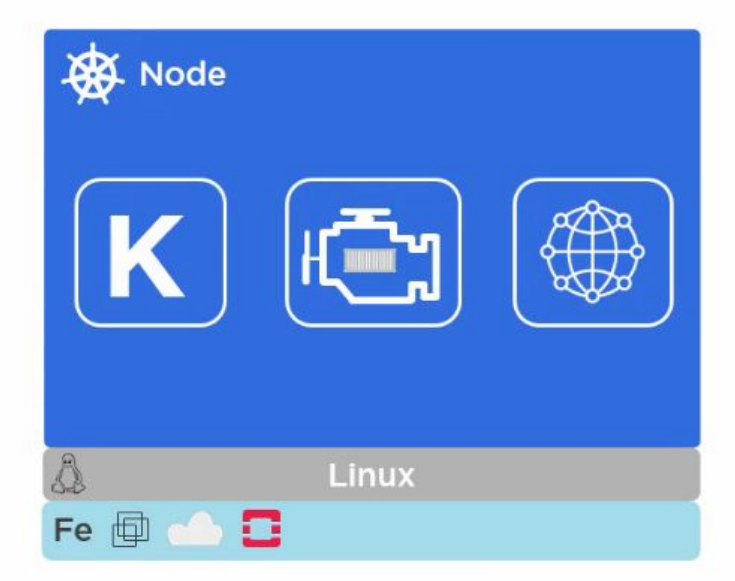


Empezando por arriba y en sentido de las agujas del reloj:

- **Cluster store**: mantiene el almacenamiento persistente. Se encarga de ver el estado del clúster y su configuración. Usa etcd. Suele emplear BBDD NoSQL.
- **Controller manager**: este es un controlador de controladores, nodo controlador, controlador endpoint, y controlador del namespace. Prácticamente tenemos un controlador para todo, ayuda manteniendo un estado deseado estable.
- **kube-apiserver**: es el fron-end de acceso al panel de control, expone la API de acceso y utiliza para ello JSON files o YAML. Es el único componente con el que podemos interactuar directamente, para ello emplearemos la utilidad de línea de comandos $**kubectl**.
- **kube-scheduler**: mira si hay nuevos pods y los asigna a nodos workers (recursos, restricciones …)

Nodos

Solían tener el nombre de MInions, son los k8s workers. Hacen lo que el nodo maestro les manda.



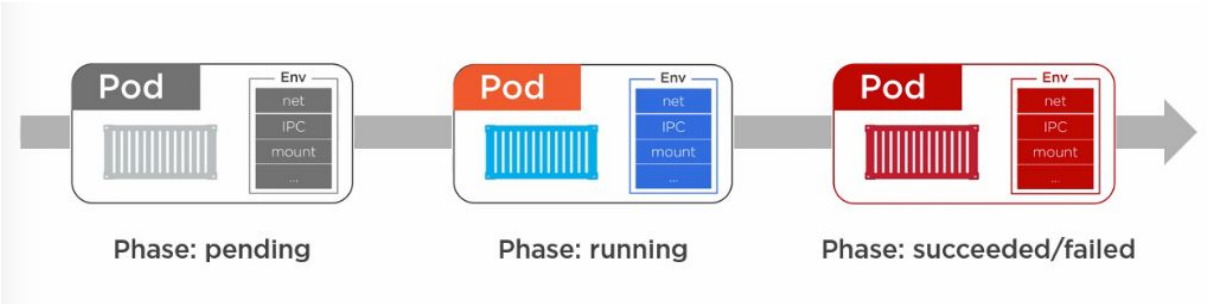
Son más simples que el nodo maestro, hay sólo tres cosas a tener en cuenta:

- **Kubelet**: es el principal agente en el nodo, registra el nodo en el k8s cluster y mira en la api server del master para ver sus asignaciones de trabajo. Instancia pods y reporta al master.Si un pod cae el kubelet no lo vuelve a levantar sino que lo reporta al master
- **Container engine**: realiza la gestión del contenedor, sube imágenes, para/arranca contenedores (usualmente emplea Docker aunque CoreOS Rocket (rkt) también tiene auge)
- **Kube-proxy**: es como el cerebro del nodo. Se encarga de la gestión de la red, todos los contenedores en un pod tienen una sola IP, si queremos acceder a los pods accederemos a ellos a través de un número de puerto. Balancea los pods a través de un servicio.

Pods

Las unidades mínimas a lo largo de estos años han sido:

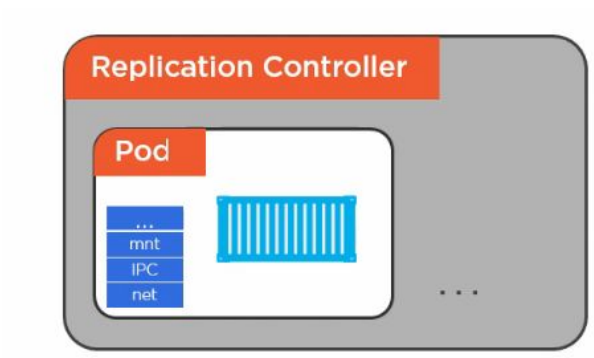
Para aclarar conceptos, k8s ejecuta contenedores sí, pero dentro de pods, no puede desplegar un contenedor fuera de un pod, de hecho un pod puede ejecutar más de un contenedor. El ciclo de vida de un pod es el siguiente:

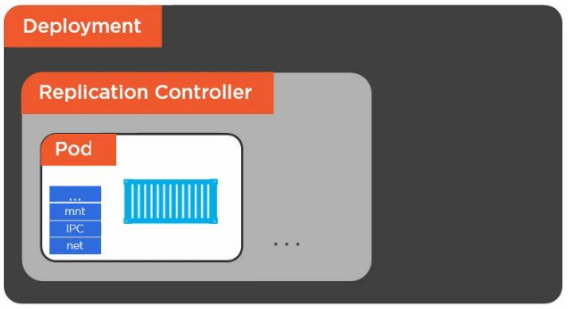


Si un contenedor dentro de un pod falla, no se intenta revivir o ver los fallos se levanta uno nuevo. Con este concepto aparece uno nuevo y es el de Replica Controller:

Replica Controller

Cuando queremos desplegar un pod podemos utilizar elementos de alto nivel como Replica Controllers que se basaran en la definición que tendremos en nuestro fichero YAML o JSON. Resumiendo establece réplicas del pod si éstos fallan.

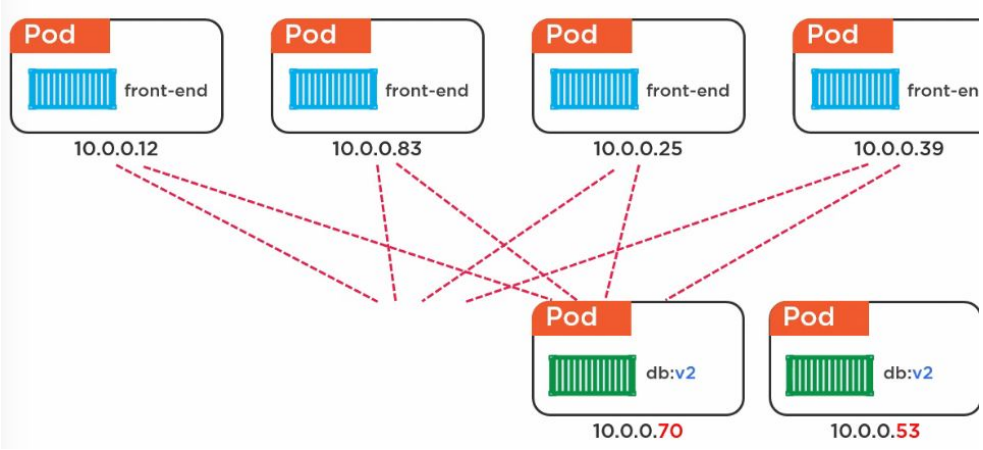




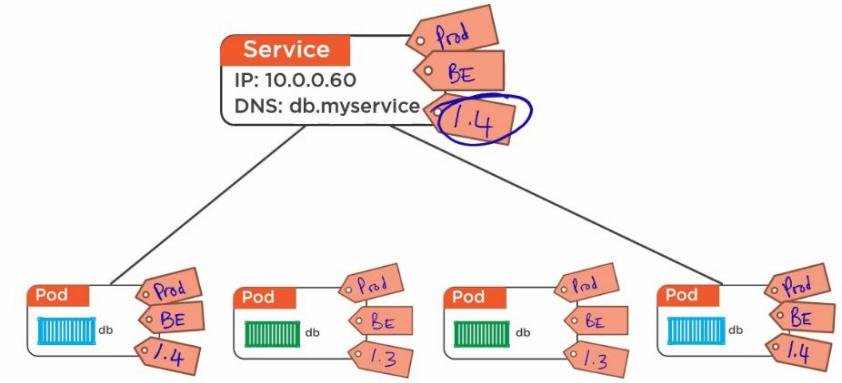
Services

Hemos visto que los pods dentro de los nodos tienen asignadas una IP. Si alguno muere se levanta otro y no necesariamente en el mismo Pod ni con la misma IP.

Entonces pudiera darse el caso siguiente, que un pod al que teníamos acceso falle se levante y ya no tengamos acceso a él y todos los pods que apuntaban a él pierdan su acceso:



Para evitar esta situación aparecen los servicios que harán de gestores de pods. Para facilitar esta faena también existen las etiquetas (labels), mediante las cuales podremos acceder a aquellos pods que nos interesen.



Deployments (despliegues)

Ahora que ya tenemos nuestra infraestructura, podemos desplegarla de una forma declarativa utilizando archivos YAML para especificar cuántos pods queremos, cuáles queremos etc .. Más adelante lo veremos en más profundidad.

YAML review

Los ficheros YAM están compuestos de mapas y listas.
La indentación importa y MUCHO.
Usa siempre espacios, NUNCA tabules.
Mapas:

- name: par de valores.
- pueden contener otros mapas para hacer estructuras de datos más complejos.

Podemos tener también una secuencia de ítems, son las listas.

- secuencia de ítems.
- múltiples mapas pueden ser definidos en una lista también.

```
key: value
complexMap:
  key1: value
  key2:
    subKey: value
items:
  - item1
  - item2
itemsMap:
  - map1: value
    map1Prop: value
  - map2: value
    map2Prop: value
```

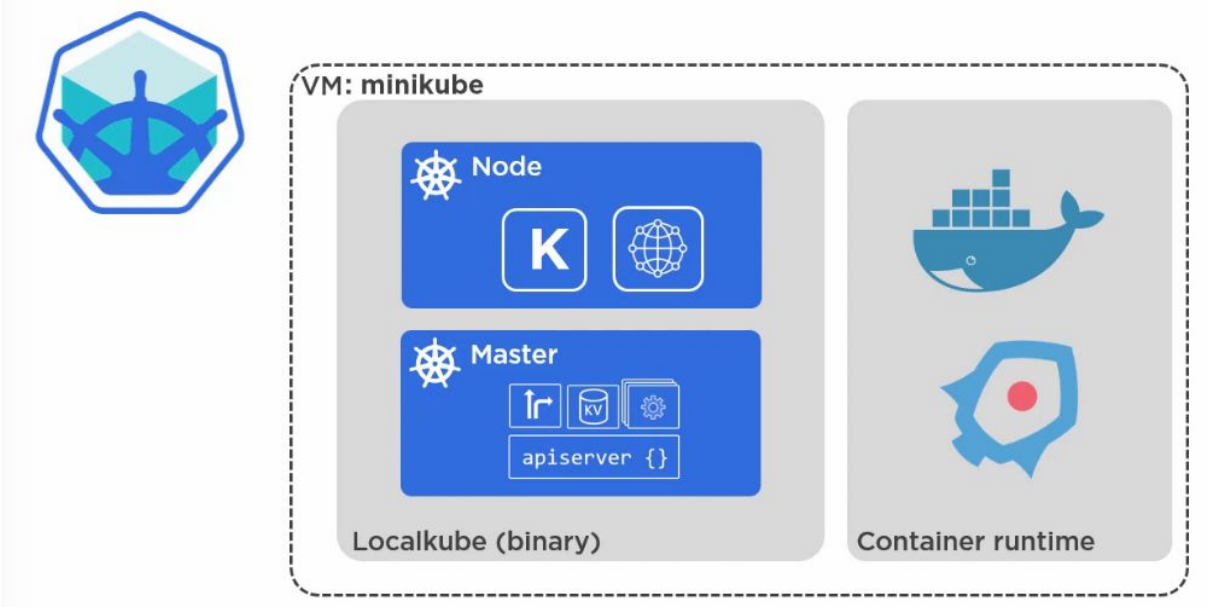
Instalando K8s

Tenemos diversas formas de instalar K8s, mencionaremos las principales aunque aquí nos vamos a centrar en aquella en la que podamos utilizar en nuestro portátil o PC. De mayor facilidad a menor tenemos las siguientes opciones:

- MiniKube
- Docker Desktop
- Google Container Desktop (GKE)
- AWS Provider
- Manual Install con Kubeadm

MiniKube

MiniKube tiene la siguiente estructura:



Y como referencia podemos utilizar los siguientes enlaces:

- https://github.com/kubernetes/minikube
- https://kubernetes.io/

Desarrollaré la explicación en entorno MAC pero no varía en demasía para Windows o Linux

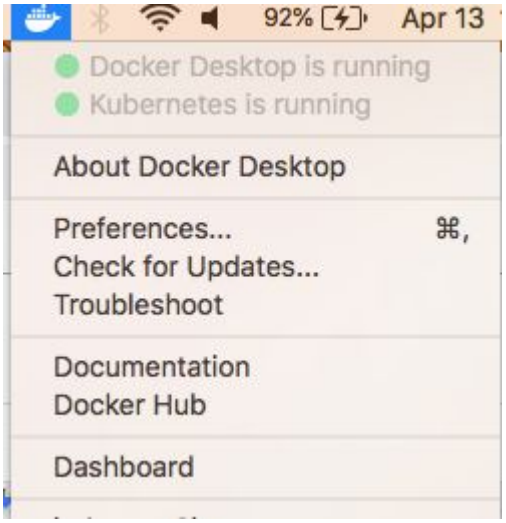
```
$brew install kubectl
$brew install minikube
```

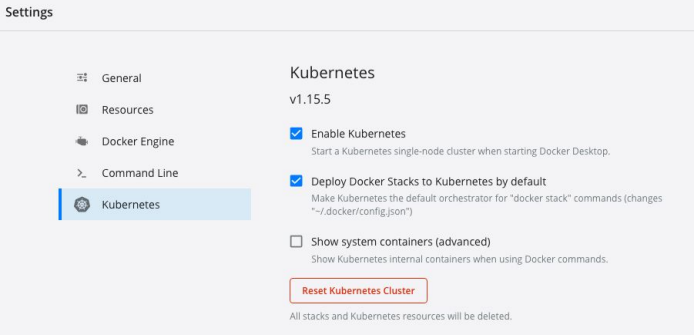
Con estos dos comando instalo la línea de comandos kubectl para poder interactuar por terminal y la máquina virtual de minikube que arrancaremos siempre por terminal.
Para poder acceder al comando kubectl desde cualquier sitio de la terminal tendremos que incluir en el path la ruta, adia de hoy habría que incluir en el archivo .bash_profile la siguiente línea:

```
export PATH=$PATH:/usr/local/Cellar/kubernetes-cli/1.18.1/bin
```

Docker Desktop

Realizando la instalación típica de Docker solo tendremos que acceder al panel de control para habilitar Kubernetes, aunque previamente habremos instalado kubectl y modificado el path para poder acceder. En el panel de control en la opción Preferences iremos a Kubernetes y lo habilitaremos, la primera vez tarda unos 5’ en realizar todas las acciones para habilitarlo.





Kubernetes Pods

A continuación crearemos un pod con un contenedor nginx. Tenemos dos vías:

- kubectl run command
- kubectl create/apply command con un fichero YAML

```
#Lista sólo pods
kubectl get pods
```

```
#lista todos los recursos
kubectl get all
```

```
#activar contenedor del pod para ser llamado desde fuera
kubectl port-forward [podname] 8080:80
```

```
#borrado de un pod
kubectl delete pod [podname]
```

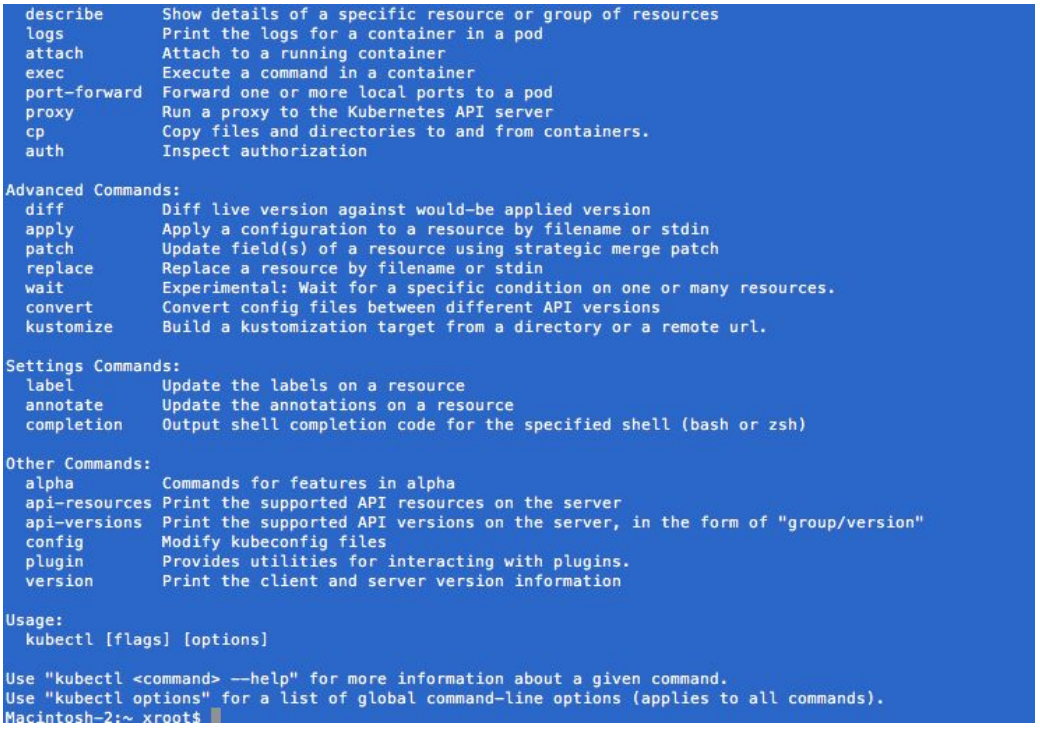
```
#borrado de un despliegue (deployment)
kubectl delete deployment [deploymentname]
```

```
#Ejecuta nginx:alpine
kubectl run [podname] --image=nginx:alpine
```

```
#Entrar dentro de un pod
kubectl exec [podname] -it sh
```

kubectl run command

Lo primero asegurarnos de que funciona kubectl, para ello tecleamos kubectl.



Ahora crearemos un pod con nombre mi_pod partiendo de una imagen nginx:alpine:

```
$kubectl run mi-pod --image=nginx:alpine
```

Ahora comprobamos:

$kubectl get pods

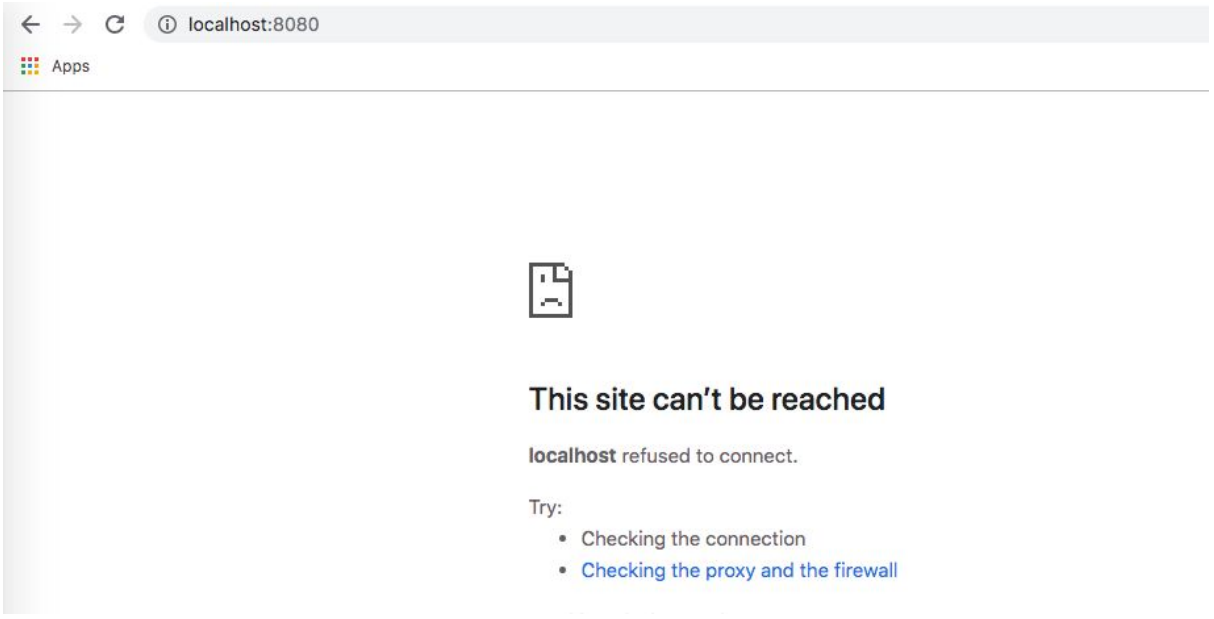

```
$kubectl get all
```

```
Macintosh-2:~ xroot$ kubectl get all
NAME         READY   STATUS    RESTARTS   AGE
pod/mi-pod   1/1     Running   0          70s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   21h
Macintosh-2:~ xroot$
```

Para borrar el pod:

```
$kubectl delete pod mi-pod
```

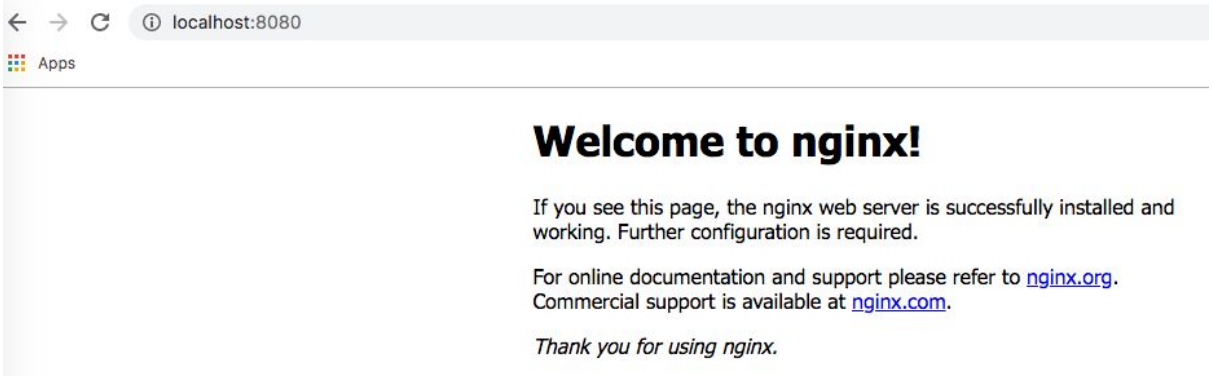
Bien, lo vuelvo a crear e intento acceder a él, y como es previsible no puedo acceder.



Para ello tendré que exponer el puerto:

```
$kubectl port-forward mi-pod 8080:80
```

```
Macintosh-2:~ xroot$ kubectl port-forward mi-pod 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```


localhost:8080
Apps
Welcome to nginx!
If you see this page, the nginx web server is successfully installed and working. Further configuration is required.
For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.
Thank you for using nginx.

kubectl create/apply command con un fichero YAML

Ahora vamos a definir un pod con un fichero yaml.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx
spec:
  containers:
  - name: my-nginx
    image: nginx:alpine
```

Y para ejecutar este fichero:

$kubectl create -f mificheropod.yml

ó

$kubectl apply -f micheropod.yml

```
Macintosh-2:~ xroot$ nano mi-pod.yaml
Macintosh-2:~ xroot$ cat mi-pod.yaml
apiVersion: v1
kind: Pod
metadata:
 name: mi-nginx
spec:
 containers:
 - name: mi-nginx
   image: nginx:alpine
Macintosh-2:~ xroot$ kubectl create -f mi-pod.yaml
pod/mi-nginx created
Macintosh-2:~ xroot$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
mi-nginx   1/1     Running   0          6s
mi-pod     1/1     Running   0          26m
Macintosh-2:~ xroot$
```

En este caso hemos creado el pod mi-nginx. Y ahora para poder borrarlo podemos utilizar la nomenclatura aprendida anteriormente o borrarlo utilizando el fichero que hemos empleado para crearlo.

```
Macintosh-2:~ xroot$ kubectl delete -f mi-pod.yaml
pod "mi-nginx" deleted
Macintosh-2:~ xroot$ kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
mi-pod   1/1     Running   0          29m
Macintosh-2:~ xroot$
```

Un comando muy interesante es:

```
$kubectl describe pod [nombrepod]
```

En nuestro caso en particular

```
Macintosh-2:~ xroot$ kubectl describe pod mi-pod
Name:         mi-pod
Namespace:    default
Priority:     0
Node:         docker-desktop/192.168.65.3
Start Time:   Tue, 14 Apr 2020 12:17:06 +0200
Labels:       run=mi-pod
Annotations:  <none>
Status:       Running
IP:           10.1.0.15
IPs:          <none>
Containers:
  mi-pod:
    Container ID:   docker://629d8ea6378738f37aed60d7d4a0d44369a8f7ea84503d3d36b5bee632134098
    Image:          nginx:alpine
    Image ID:       docker-pullable://nginx@sha256:abe5ce652eb78d9c793df34453fddde12bb4d93d9fbf2c363d0992726e4d2cad
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 14 Apr 2020 12:17:09 +0200
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-9h2cb (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-9h2cb:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-9h2cb
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:          <none>
```

Un apartado muy importante es el de status para saber en qué estado está el Pod y el contenedor que hay dentro.

Podemos incluir también el puerto que expone el contenedor, siendo en nuestro caso el puerto 80:

```
$ kubectl port-forward hello-pod 8080:80
```

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

Ahora vamos incluir un paso más, un estado deseado en el pod, y para ello nos vamos a apoyar en el Replication Controller. Por ejemplo podemos definir que queremos 5 réplicas de ese pod y si alguna falla el RC se encargará de levantar otra instancia, no intentará arreglarla ni ver dónde ha fallado.

Vamos a partir del siguiente fichero:

```
Macintosh-2:~ xroot$ cat  mi-pod.yaml
apiVersion: v1
kind: ReplicationController
metadata:
 name: hello-rc
spec:
 replicas: 4
 selector:
  app: hello-world
 template:
  metadata:
   labels:
     app: hello-world
  spec:
   containers:
   - name: mi-nginx
     image: nginx:alpine
     ports:
     - containerPort: 8080
Macintosh-2:~ xroot$ kubectl create -f mi-pod.yaml
replicationcontroller/hello-rc created
Macintosh-2:~ xroot$
```

kind: ReplicationController, cuando antes teníamos el tipo Pod.

metadata: describimos el nombre y las etiquetas asociadas a este ReplicationController.

spec: la especificación, queremos cuatro réplicas y para ello utilizará una plantilla que contendrá un el contenedor que queremos y el puerto que se va a exponer. Los cuatro contenedores no se exponen a la vez sino que se hace balanceo entre ellos. Debemos tener muy en cuenta los espacios en blanco, puede ser que quiera un map y me genere un error porque detecta un String y esto puede ser porque no/si detecta el espacio en blanco.

Como he especificado cuatro réplicas, en cuanto yo borre un pod, automáticamente se levantará otro, y de esta faena se encarga el RC.

```
Macintosh-2:~ xroot$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
hello-rc-7kzvc   1/1     Running   0          2m13s
hello-rc-f5f7v   1/1     Running   0          2m13s
hello-rc-j2jz7   1/1     Running   0          2m13s
hello-rc-nqztg   1/1     Running   0          2m13s
Macintosh-2:~ xroot$ kubectl delete pod hello-rc-7kzvc
pod "hello-rc-7kzvc" deleted
Macintosh-2:~ xroot$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
hello-rc-f5f7v   1/1     Running   0          2m32s
hello-rc-gdxkf   1/1     Running   0          4s
hello-rc-j2jz7   1/1     Running   0          2m32s
hello-rc-nqztg   1/1     Running   0          2m32s
Macintosh-2:~ xroot$
```

Ahora vamos a modificar el fichero y donde ponía 4 réplicas ahora vamos a poner 5. Cuando modificamos estos ficheros sobre templates y plantillas o RC, para volver a ejecutarlos escribiremos el siguiente comando.

```
$kubectl apply -f mi-pod.yml
```

```
Macintosh-2:~ xroot$ kubectl apply -f mi-pod.yaml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
replicationcontroller/hello-rc configured
Macintosh-2:~ xroot$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
hello-rc-f5f7v   1/1     Running   0          13m
hello-rc-gdxkf   1/1     Running   0          11m
hello-rc-j2jz7   1/1     Running   0          13m
hello-rc-nqztg   1/1     Running   0          13m
hello-rc-z7w5s   1/1     Running   0          28s
Macintosh-2:~ xroot$
```

No hay que preocuparse del mensaje, si volvemos a ver los pods veremos que hay uno más, y tenemos los cinco que queríamos.

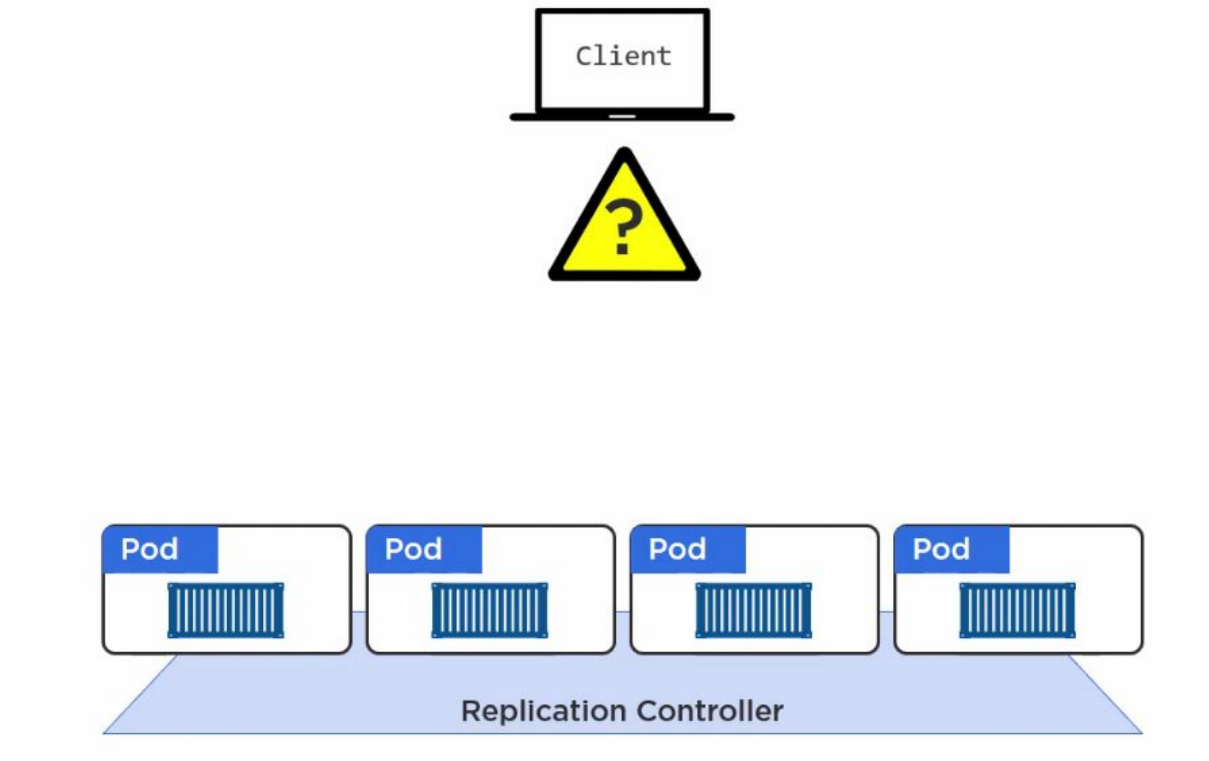
Si queremos ver algún pod podemos exponer su puerto en particular y ejecutarlo en el navegador.

Pod Health

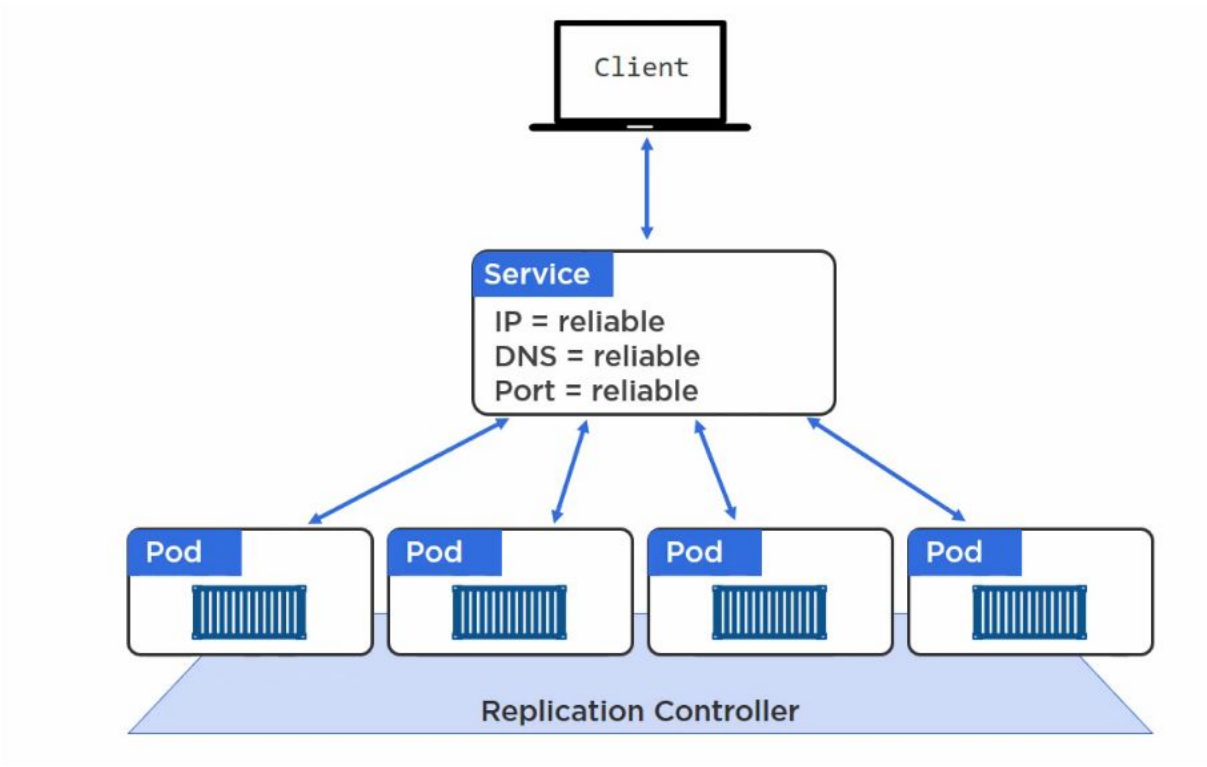
Kubernetes Services

Bien, tenemos una app ejecutándose, aunque es sólo la evidencia de que hay algo ejecutándose, tenemos alguno pods corriendo en la máquina. Tenemos un RC y si perdemos algún pod éste levanta uno nuevo.

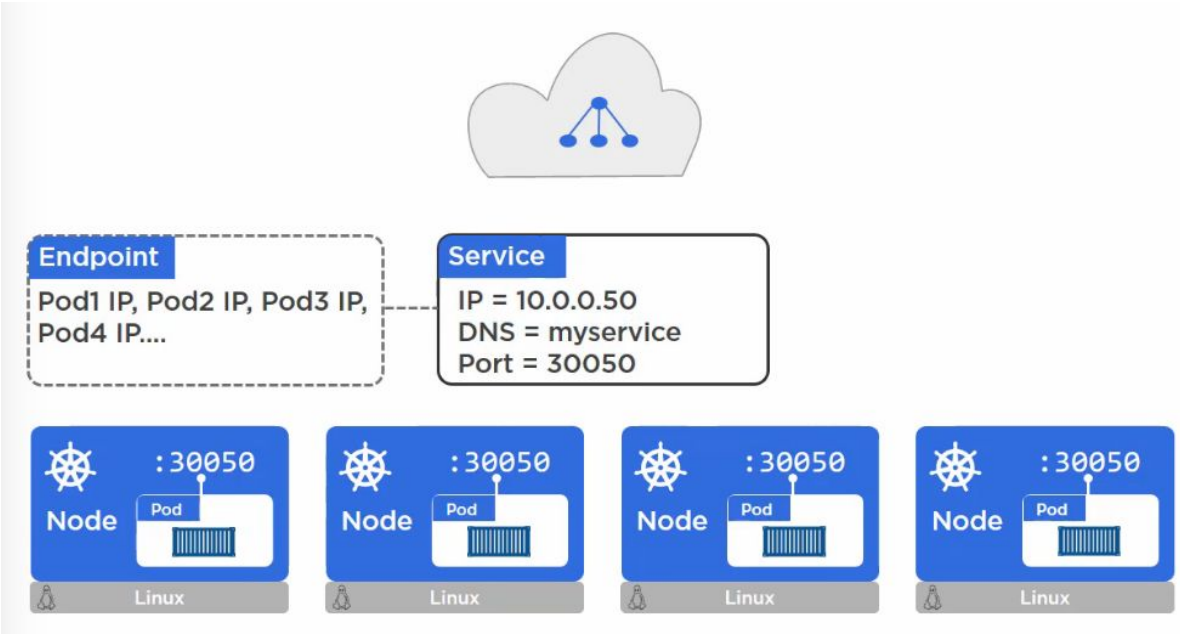
Para acceder a nuestras apps aparecen los servicios (services), que son objetos RESTs en la API de K8s, para empezar veamos los problemas principales:



A partir de aquí, aparece el servicio que nos va a hacer transparente el acceso a los pods.



El servicio expondrá un puerto al exterior, puerto que se hará extensible a los pods:



Además también generará una lista de endpoints, que es una lista de los pods con sus ips disponibles en este momento.

Modo Iterativo

Partiendo del siguiente estado:

```
hello-rc-27w5s   1/1     Running   0          28s
Macintosh-2:~ xroot$ cat mi-pod.yaml
apiVersion: v1
kind: ReplicationController
metadata:
 name: hello-rc
spec:
 replicas: 5
 selector:
  app: hello-world
 template:
  metadata:
   labels:
     app: hello-world
  spec:
   containers:
   - name: mi-nginx
     image: nginx:alpine
     ports:
     - containerPort: 8080
Macintosh-2:~ xroot$ kubectl get nodes
NAME             STATUS   ROLES    AGE   VERSION
docker-desktop   Ready    master   47h   v1.15.5
```

Si accedemos a localhost:8080, veremos que no hay acceso, para mejorar el acceso vamos a crear un servicio y exponer uno de sus puertos que se extenderá al resto de sus puertos.

$kubectl expose rc hello-rc --name=hello-svc --target-port=8080 --type=LoadBalancer

```
Macintosh-2:~ xroot$ kubectl expose rc hello-rc --name=hello-svc --target-port=8080 --type=NodePort
service/hello-svc exposed
Macintosh-2:~ xroot$
Macintosh-2:~ xroot$ kubectl describe svc hello-svc
Name:                     hello-svc
Namespace:                default
Labels:                   app=hello-world
Annotations:              <none>
Selector:                 app=hello-world
Type:                     NodePort
IP:                       10.109.227.112
LoadBalancer Ingress:     localhost
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30961/TCP
Endpoints:                10.1.0.17:8080,10.1.0.19:8080,10.1.0.20:8080 + 2 more...
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
Macintosh-2:~ xroot$
```

Ahora haciendo http://localhost:8080/ deberíamos tener acceso al servidor web.

Podemos ver los servicios ejecutándose:

```
Macintosh-2:~ xroot$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
hello-svc    NodePort    10.109.227.112   <none>        8080:30961/TCP   12m
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP          47h
Macintosh-2:~ xroot$
```

Modo Declarativo

Primero borramos el servicio creado anteriormente:

```
Macintosh-2:~ xroot$ kubectl delete svc hello-svc
service "hello-svc" deleted
Macintosh-2:~ xroot$
```

Entonces ahora vamos a crear nuestro manifiesto:

```
 GNU nano 2.0.6                                File: mi

apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: hello-world
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 300001
    protocol: TCP
  selector:
    app: hello-world
```

Tenemos tres tipos de type:

- **ClusterIP**: IP estable interna de cluster.
- **NodePort**: expone la fuera del clúster incluyendo un puerto.
- **LoadBalancer**: integra el NodePort con los balanceadores de nube.

Ahora si compruebo los pods que oy a manejar con el yml creado, aunque para hacer pruebas en local **cambiaré el NodePort a LoadBalancer.**

```
Macintosh-2:~ xroot$ kubectl describe pods | grep app
Labels:         app=hello-world
Labels:         app=hello-world
Labels:         app=hello-world
Labels:         app=hello-world
Labels:         app=hello-world
Macintosh-2:~ xroot$ cat mi-svc.yml
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: hello-world
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 300001
    protocol: TCP
  selector:
    app: hello-world
```

$kubectl create -f mi-svc.yml

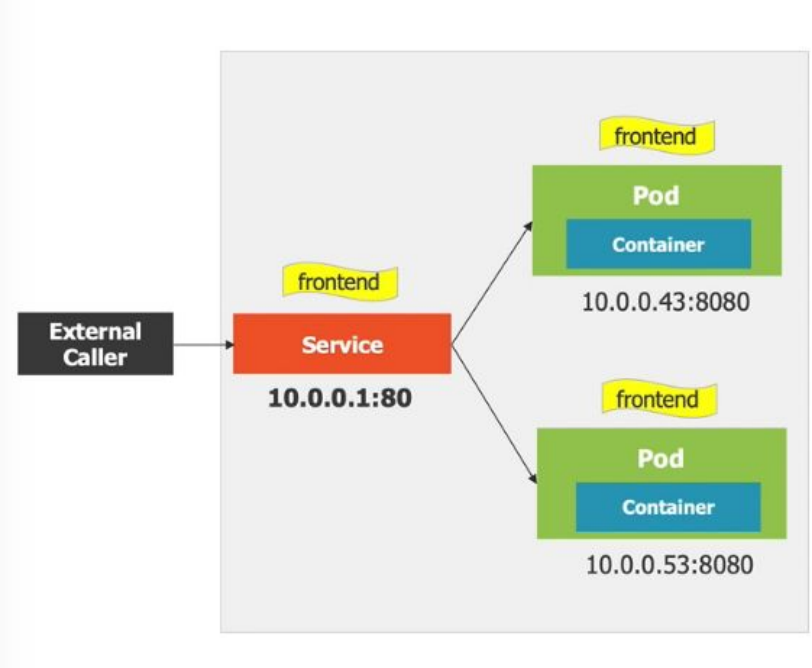
```
Macintosh-2:~ xroot$ kubectl create -f mi-svc.yml
service/hello-svc created
Macintosh-2:~ xroot$ kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hello-svc    NodePort    10.111.94.195   <none>        8080:30001/TCP   18s
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP          47h
Macintosh-2:~ xroot$ kubectl describe svc hello-svc
Name:                     hello-svc
Namespace:                default
Labels:                   app=hello-world
Annotations:              <none>
Selector:                 app=hello-world
Type:                     NodePort
IP:                       10.111.94.195
LoadBalancer Ingress:     localhost
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30001/TCP
Endpoints:                10.1.0.17:8080,10.1.0.20:8080,10.1.0.21:8080 + 2 more...
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
Macintosh-2:~ xroot$
```

Y los endpoints asociados, que son los pods:

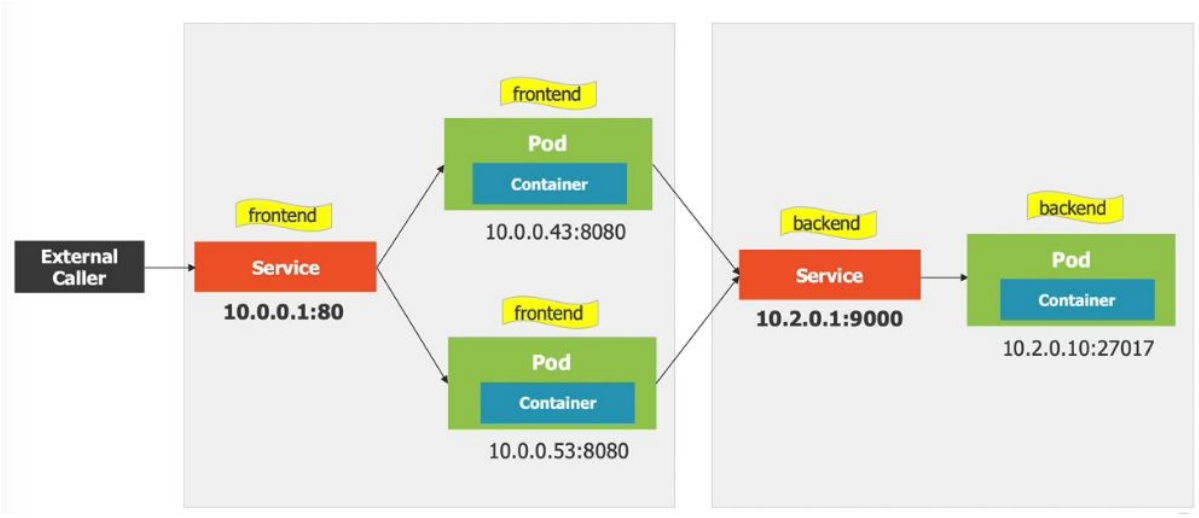
```
Macintosh-2:~ xroot$ kubectl describe ep hello-svc
Name:         hello-svc
Namespace:    default
Labels:       app=hello-world
Annotations:  endpoints.kubernetes.io/last-change-trigger-time: 2020-04-15T12:17:14Z
Subsets:
  Addresses:          10.1.0.17,10.1.0.20,10.1.0.21,10.1.0.22,10.1.0.23
  NotReadyAddresses:  <none>
  Ports:
    Name     Port  Protocol
    ----     ----  --------
    <unset>  8080  TCP

Events:  <none>
Macintosh-2:~ xroot$
```

Entendiendo los tipos de Servicios:

La pregunta principal viendo los problemas que hemos tenido hasta ahora intentando ver los pods como se ejecutan, ¿podemos confiar en la ip? La cuestión es depende del tipo de ip que tengamos definida podremos acceder de una manera u otra. Pero no podremos creer en ellas porque las ips de los pods cambian.

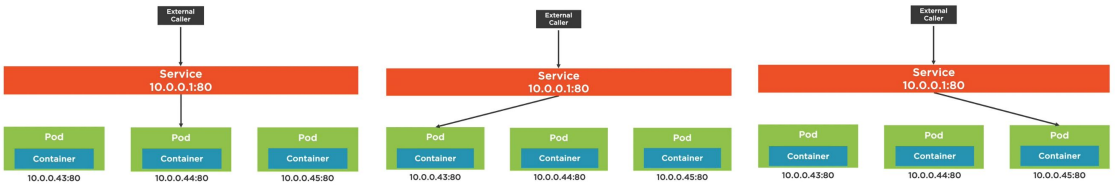


Ante un agente externo que llama hay que ver cómo llamaremos a las ips de los servicios o de los pods (como accederemos a ellos). La etiqueta “frontend” es como enlaza todos los objetos dentro del nodo. E incluso el gráfico de arriba se puede complicar:



Y ¿cómo accederemos al backend?

Bien, ante diferentes llamadas el servicio irá balanceando las llamadas para que sean atendidas por los diferentes pods.



Bien, pero los navegadores funcionan de una manera un poco diferente, éstos usan la misma conexion una y otra vez sobre el servidor y será respetada por K8s y esa conexión seguirá llamando al mismo pod una y otra vez, si éste desaparece puede ser reprogramado y el servicio puede lidiar con esto.

¿Cómo podemos acceder a un Pd desde fuera de K8s? la respuesta es sencilla no se puede ano ser que empleemos el port-forwarding.

#escucha en el puerto 8080 local y redirige al puerto 80 del pod

```
$kubectl port-forward pod/pod-name 8080:80
```

#escucha en el puerto 8080 local y redirige al puerto 80 del deploy

```
$kubectl port-forward deployment/deployment-name 8080:80
```

#escucha en el puerto 8080 local y redirige al puerto 80 del servicio

```
$kubectl port-forward service/service 8080:80
```

Ahora partiendo del siguiente despliegue:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: hello-deploy
 labels:
  app: my-nginx
spec:
 replicas: 2
 selector:
  matchLabels:
   app: my-nginx
 minReadySeconds: 5
 strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
 template:
  metadata:
   labels:
    app: my-nginx
  spec:
   containers:
   - name: my-nginx
     image: nginx:alpine
     ports:
     - containerPort: 80
     resources:
      limits:
       memory: "128Mi" #128MB
       cpu: "200m"     #200 millicpu .2 cpu o 20% cpu
```

Tiene el contenedor nginx:alpine y despliega el puerto 80 del contenedor, así como limita a un 20% de cpu y a 128 MB de espacio, y dos réplicas. Pues vamos a desplegarlo:

```
$ kubectl apply -f mi-deploy.yml
```

Si vemos lo que se ha creado, los dos pods, un deploy y un replicaset, y tenemos un ClusterIP en el servicio, podríamos pensar en llamarlo desde el navegador pero no funcionará, por ahora sólo tenemos un ClusterIP configurado para nuestros servicios, con lo que para conectar con los pods tendremos que usar el comando port-forward:

```
NAME                                 READY   STATUS    RESTARTS   AGE
pod/hello-deploy-77f7664f5f-dh4bf    1/1     Running   0          2m9s
pod/hello-deploy-77f7664f5f-wfjg5    1/1     Running   0          2m9s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3d22h

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hello-deploy   2/2     2            2           2m9s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/hello-deploy-77f7664f5f    2         2         2       2m9s
Macintosh-2:~ xroot$
```

Para conectar con el pod:

```
NAME                                READY   STATUS    RESTARTS   AGE
pod/hello-deploy-77f7664f5f-dh4bf   1/1     Running   0          2m9s
pod/hello-deploy-77f7664f5f-wfjg5   1/1     Running   0          2m9s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3d22h

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hello-deploy   2/2     2            2           2m9s

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/hello-deploy-77f7664f5f   2         2         2       2m9s
Macintosh-2:~ xroot$
```

Y ejecutando en el navegador localhost:8080, podremos ver la página de inicio de nginx. Ahora puedo intentar hacer este port-forward al deploy, y obtengo el mismo resultado:

```
Macintosh-2:~ xroot$ kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
hello-deploy   2/2     2            2           18m
Macintosh-2:~ xroot$ kubectl port-forward deployments/hello-deploy 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Definiendo un servicio

```
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: nginx
spec:
  selector:
   app: nginx
  ports:
  - name: http
    port: 8081
    targetPort: 80
```

Se define el target port del container y el puerto del servicio. El selector app, se aplicará a los recursos con una etiqueta de “app: nginx”.

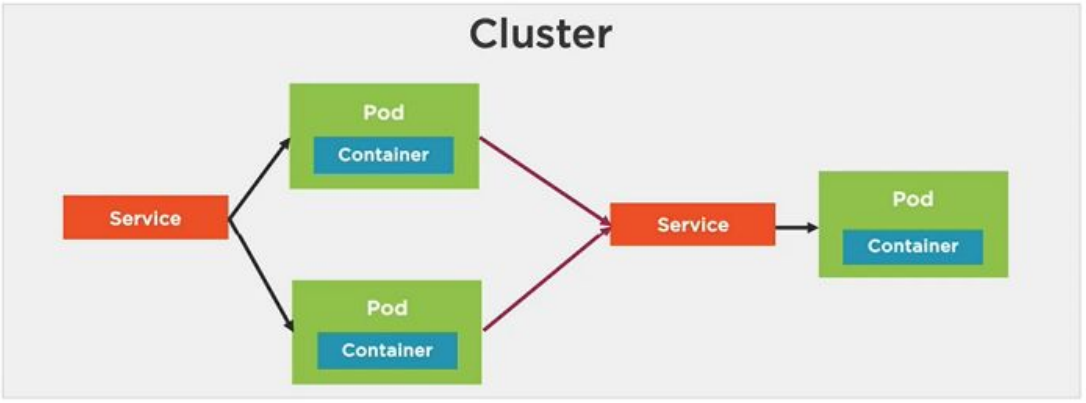
Tenemos los siguientes tipos de servicios:

- ClusterIP
- NodePort
- LoadBalancer

- ExternalName

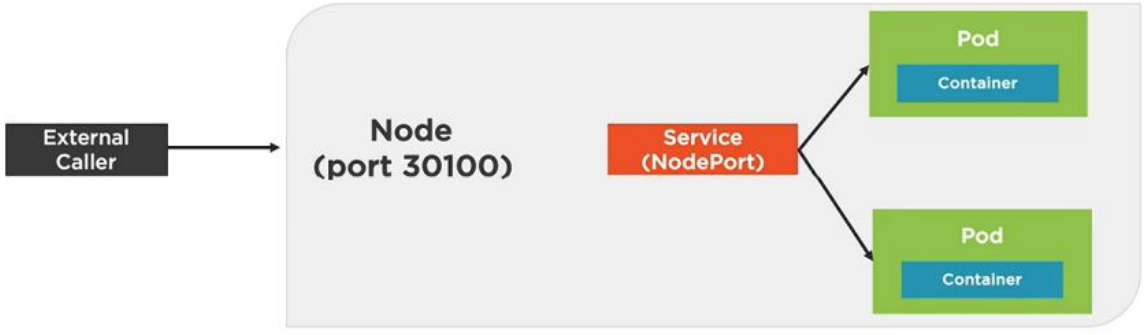
ClusterIP

Expone el servicio en una ip del cluster interno. El servicio puede hablar con los pods internamente, sólo los pods del clúster podrán hablar con el servicio, pero permite a pods hablar con otros pods.



NodePort

Expone el servicio en cada ip del nodo en un puerto estático. Expone el servicio en cada ip de un nodo en un puerto estático. Aloja un puerto por defecto aunque puede ser configurado, podremos ir a través de ese nodo

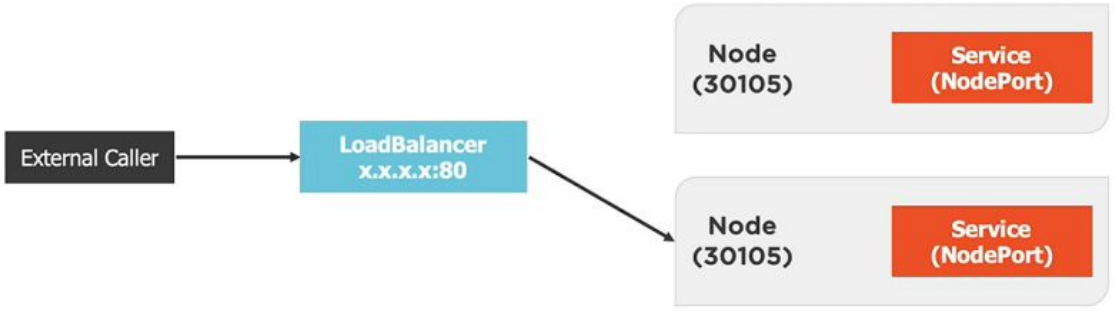


Exponemos el puerto 30100 del nodo, así que si alguien llama desde fuera llamará a la ip del nodo en ese puerto y entonces podrá llamar a los servicios que hay por detrás y los pods que hayan. Muy útil por las siguientes razones.
Un ejemplo de YAML:

```yaml
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: nginx
spec:
  type: NodePort
  selector:
   app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 31000
```

LoadBalancer

Provee una ip externa para actuar como balanceador del servicio. Expone un servicio al exterior y las llamadas son balanceadas a los diferentes pods que hay en el interior.



targetPort es el puerto interno al cual va a llamar

```yaml
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: nginx
spec:
  type: LoadBalancer
  selector:
   app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
```

ExternalName

Mapea un servicio como un nombre DNS. Servicio que actúa como un alias para un servicio externo, permite a un servicio actuar como un proxy para un servicio externo. Los detalles del servicio están ocultos del clúster.

```
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: nginx
spec:
  type: ExternalName
  externalName: api.acmecorp.com
  selector:
   app: nginx
  ports:
  - port: 80
```

Testing un servicio y Pod con curl

Podemos ejecutar un shell en un Pod y testear una URL en caso que hayan múltiples contenedores ejecutándose en un Pod.

$kubectl exec podname -- curl -s http://podIP

EN caso que haya que instalar CURL

kubectl exec podname -it sh

>apk add curl

>curl -s http:podIP

Tenemos los dos pods ejecutándose del deploy hecho hace poco:

```
Macintosh-2:~ xroot$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
hello-deploy-77f7664f5f-dh4bf   1/1     Running   0          96m
hello-deploy-77f7664f5f-wfjg5   1/1     Running   0          96m
Macintosh-2:~ xroot$
```

Ahora voy a crear un pod por separado e intentar comunicar con alguno de los otros dos.

```
kubectl run mipod --image=nginx:alpine
```

Y ahora:

```
Macintosh-2:~ xroot$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
hello-deploy-77f7664f5f-dh4bf   1/1     Running   0          100m
hello-deploy-77f7664f5f-wfjg5   1/1     Running   0          100m
mipod                         1/1     Running   0          44s
```

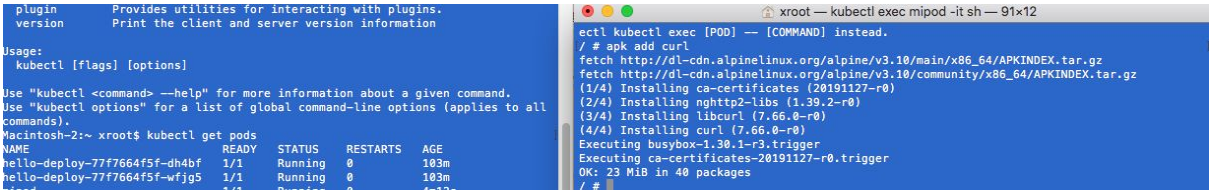
Y vamos a entrar dentro de él:

```
kubectl exec mipod -it sh
```

E instalamos curl

```
Macintosh-2:~ xroot$ kubectl exec mipod -it sh
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. U
ectl kubectl exec [POD] -- [COMMAND] instead.
/ # apk add curl
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/community/x86_64/APKINDEX.tar.gz
(1/4) Installing ca-certificates (20191127-r0)
(2/4) Installing nghttp2-libs (1.39.2-r0)
(3/4) Installing libcurl (7.66.0-r0)
(4/4) Installing curl (7.66.0-r0)
Executing busybox-1.30.1-r3.trigger
Executing ca-certificates-20191127-r0.trigger
OK: 23 MiB in 40 packages
/ #
```

Lanzamos otra terminal y vemos los pods



Vamos a coger el primer pod y ver su ip, pudiendo comprobar su podIP

```
kubectl get pod hello-deploy-77f7664f5f-dh4bf -o yaml
```

```
    lastState: {}
    name: my-nginx
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: "2020-04-17T10:48:07Z"
  hostIP: 192.168.65.3
  phase: Running
  podIP: 10.1.0.70
  qosClass: Guaranteed
  startTime: "2020-04-17T10:48:04Z"
Macintosh-2:~ xroot$
```

Cogeré esa IP y me iré a la primera terminal que es donde estaba dentro del contenedor:

```
/ # curl http://10.1.0.70
```

Y nos responde el servidor web

```
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Esto es ya un buen comienzo!

Si ese pod desapareciese supondría un problema, teniendo el siguiente servicio, a través del selector localizaré todas aquellos pods que tengan matchLabels en el deploy y como label =my-nginx:

```
apiVersion: v1
kind: Service
metadata:
 name: nginx-clusterip
spec:
  type: ClusterIP
  selector:
   app: my-nginx
  ports:
  - port: 8080
    targetPort: 80
```

Voy a ejecutarlo:

```
Macintosh-2:~ xroot$ kubectl apply -f mi-svc.yml
service/nginx-clusterip created
Macintosh-2:~ xroot$
```

Y me aparece mi servicio con su ip, con lo que vamos a utilizar esta IP, con lo que debería ser capaz de llamar al otro pod desde el servicio. Volvemos al contenedor en el que estábamos dentro e intentamos hacer un curl a la ip del servicio:

```
service/nginx-clusterip created
Macintosh-2:~ xroot$ kubectl get services
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes       ClusterIP   10.96.0.1        <none>        443/TCP    4d
nginx-clusterip  ClusterIP   10.101.237.244   <none>        8080/TCP   7s
Macintosh-2:~ xroot$
```

```
^Z[1]+  Stopped                 curl http://10.99.16
/ # curl http://10.96.0.1
^Z[2]+  Stopped                 curl http://10.96.0.
/ # curl http://10.96.0.1:443
Client sent an HTTP request to an HTTPS server.
/ # curl http://10.101.237.244:8080
<!DOCTYPE html>
```

Y el curl devuelve la página web del servicio y si lo siguiera ejecutando se balancea entre uno y otro pod. Si veo el nombre del servicio también podría realizar el curl con el nombre:

```
curl http://nginx-clusterip:8080
```

Ahora a continuación vamos a limpiar el servicio:

```
kubectl delete service nginx-clusterip
```

Y seguiremos teniendo los pods que habíamos creado previamente.

Ahora vamos a abrir un NodePort service tal y como muestra la siguiente figura y haremos un apply del service.

```
apiVersion: v1
kind: Service
metadata:
 name: nodeport-svc
spec:
  type: NodePort
  selector:
   app: my-nginx
  ports:
  - port: 80
    targetPort: 80
    nodePort: 31000
```

```
kubectl apply -f npmisvc-yml
```

```
Macintosh-2:~ xroot$ kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP        4d1h
nodeport-svc   NodePort    10.96.47.127   <none>        80:31000/TCP   51s
Macintosh-2:~ xroot$
```

Tenemos mapeado 80:31000, y en localhost:31000 funciona perfectamente. Ahora para finalizar probaremos el LoadBalancer.

```
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
spec:
  type: LoadBalancer
  selector:
   app: my-nginx
  ports:
  - name: "80"
    port: 80
    targetPort: 80
```

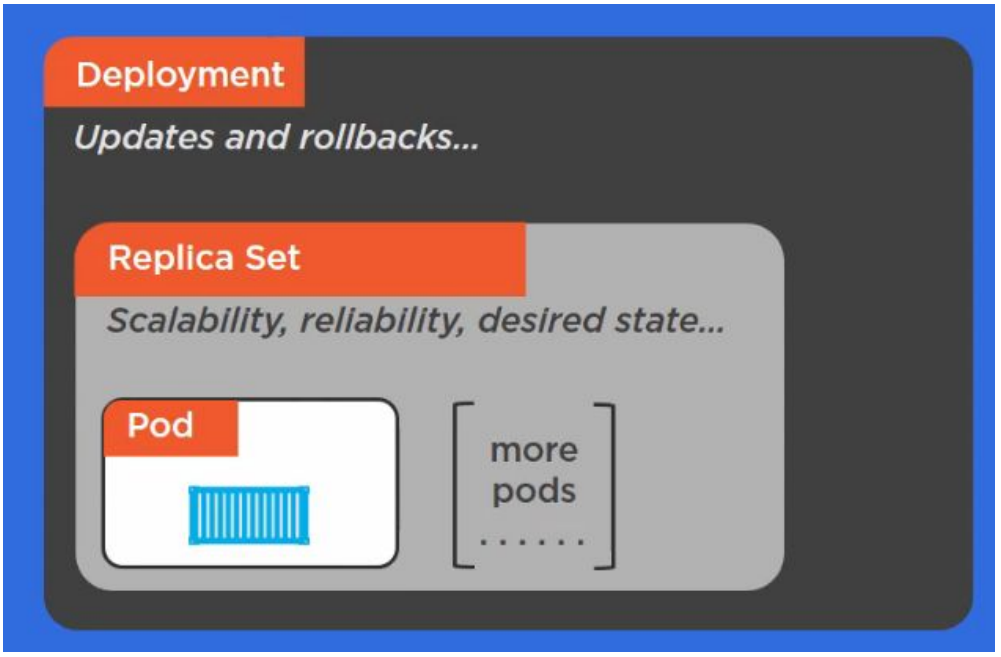
Previamente

```
Macintosh-2:~ xroot$ kubectl get services
NAME           TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP   10.96.0.1      <none>        443/TCP        4d1h
nodeport-svc   NodePort    10.96.47.127   <none>        80:31000/TCP   6m18s
Macintosh-2:~ xroot$ kubectl delete service nodeport-svc
service "nodeport-svc" deleted
Macintosh-2:~ xroot$ kubectl apply -f lbmi-svc.yml
service/hello-svc created
Macintosh-2:~ xroot$ kubectl get services
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
hello-svc    LoadBalancer   10.98.44.188   localhost     80:32432/TCP   5s
kubernetes   ClusterIP      10.96.0.1      <none>        443/TCP        4d1h
Macintosh-2:~ xroot$
```

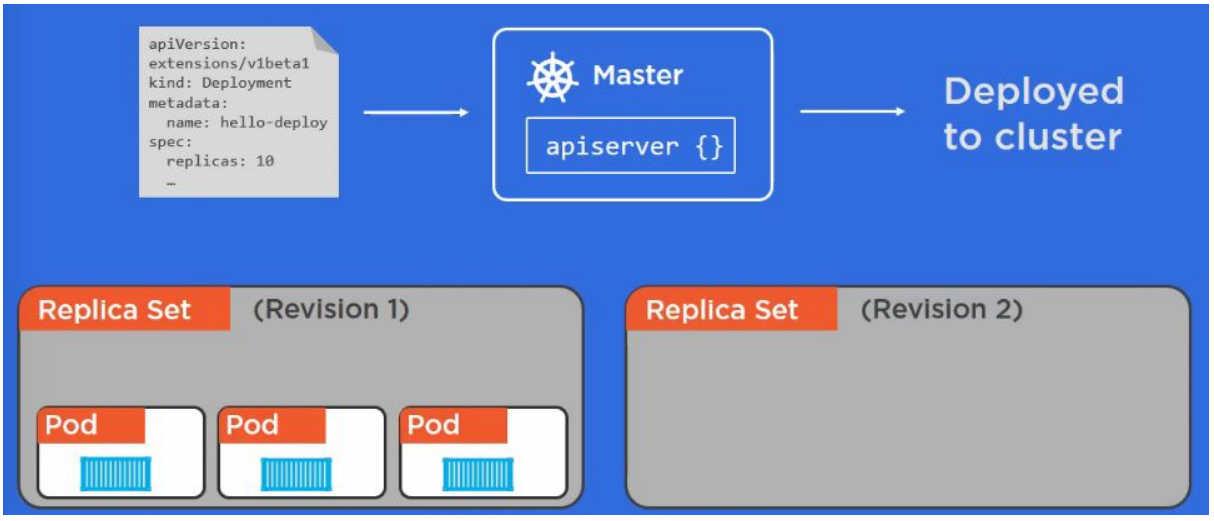
Y ahora solo en localhost:80 funciona perfectamente.

Kubernetes Deployments

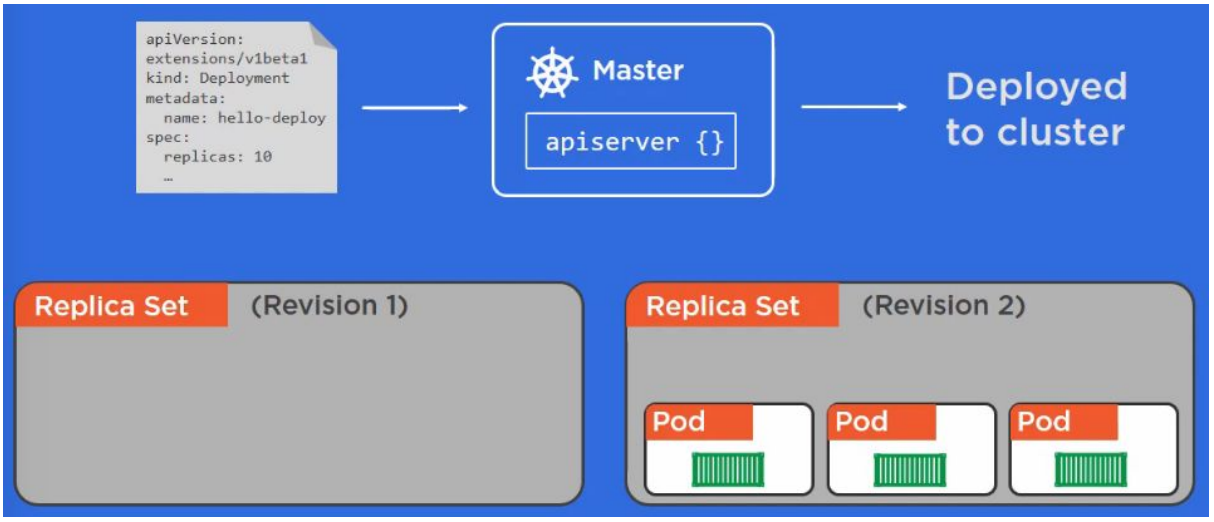
Despliegues (deployments) en K8s tratan sobre rolling updates y rollbacks, como hacer versiones de nuestros propios despliegues y poder volver atrás. Podemos crearlos como hemos hecho previamente: de una forma iterativa o de una forma declarativa, aquí nos vamos a centrar en este apartado en la forma declarativa, utilizando ficheros YAML o JSON.



Nos encontramos en una capa superior a los pods y a los RC, que aquí en este apartado se llaman ReplicaSet y se diferencian en aspectos muy sutiles de los RC, pero para lo que nos interesa a nosotros su funcionamiento es prácticamente el mismo.



A la hora de realizar un deployment, establecemos versiones y entonces los pods que teníamos en la revisión 1 pasan a la rev 2, pero el RS no se borra, permanece caso que quisiéramos realizar un rollback.



YAML

Partiendo de la siguiente situación

Lo primero es deshacernos del RC.

```
$kubectl delete rc hello-rc
```

Al realizar esta acción también desaparecerán los pods asociados a este RC.

```
Macintosh-2:~ xroot$ kubectl delete rc hello-rc
replicationcontroller "hello-rc" deleted
Macintosh-2:~ xroot$ kubectl geto pods
Error: unknown command "geto" for "kubectl"

Did you mean this?
        set
        get

Run 'kubectl --help' for usage.
Macintosh-2:~ xroot$ kubectl get pods
No resources found in default namespace.
Macintosh-2:~ xroot$
```

Veré que el servicio aún sigue en pie.

```
Macintosh-2:~ xroot$ kubectl describe svc
Name:                     hello-svc
Namespace:                default
Labels:                   app=hello-world
Annotations:              <none>
Selector:                 app=hello-world
Type:                     LoadBalancer
IP:                       10.107.134.221
LoadBalancer Ingress:     localhost
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30387/TCP
Endpoints:                <none>
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Partiremos del siguiente fichero mi-deploy.yml

Es importante el nombre apiVersion, tiene que empezar por extensions/ y luego ya pondremos la versión con la que trabajamos. El resto del fichero tiene una estructura muy similar al RC visto anteriormente.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: hello-deploy
spec:
 replicas: 3
 template:
  metadata:
   labels:
    app: hello-world
  spec:
   containers:
   - name: hello-pod
     image: nginx:alpine
     ports:
     - containerPort: 8080
```

Y para ejecutarlo:

```
$kubectl run -f mi-deploy.yml
ó
$kubectl apply -f mi-deploy.yml
```

```
Macintosh-2:~ xroot$ kubectl describe deploy hello-deploy
Name:                   hello-deploy
Namespace:              default
CreationTimestamp:      Thu, 16 Apr 2020 07:52:22 +0200
Labels:                 app=hello-world
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=hello-world
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:  app=hello-world
  Containers:
   hello-pod:
    Image:        nginx:alpine
    Port:         8080/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   hello-deploy-6c6449744d (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  60s   deployment-controller  Scaled up replica set hello-deploy-6c6449744d to 3
```

Así como también podemos obtener el ReplicaSet creado;

```
Macintosh-2:~ xroot$ kubectl get rs
NAME                       DESIRED   CURRENT   READY   AGE
hello-deploy-6c6449744d   3         3         3       3m59s
Macintosh-2:~ xroot$
```

Comprobamos que el servicio se está ejecutando para ver que está en funcionamiento el LoadBalancer para poder verlo en localhost.

```
Macintosh-2:~ xroot$ cat mi-svc.
apiVersion: v1
kind: Service
metadata:
 name: hello-svc
 labels:
  app: hello-world
spec:
  selector:
   app: hello-world
  type: LoadBalancer
  ports:
  - port: 8080
    nodePort: 30001
    protocol: TCP
```

Y comprobaremos el acceso a la app.

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Hasta ahora prácticamente no hemos hecho nada nuevo, aunque sí, hemos creado un deployment y dentro de él un ReplicaSet con tres réplicas.

Ahora que tenemos nuestra appl desplegada (deployed) y queremos actualizarla a que tenga dos réplicas y una imagen de apache.

```
$kubectl get pods
```

Comprobamos que sólo hay 3 pods en funcionamiento.

```
Macintosh-2:~ xroot$ kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
hello-deploy-6c6449744d-2h4sb   1/1     Running   0          25m
hello-deploy-6c6449744d-nxv7w   1/1     Running   0          25m
hello-deploy-6c6449744d-tmh4c   1/1     Running   0          25m
Macintosh-2:~ xroot$
```

Entonce modificaremos el yml file:

Incluimos minReadySeconds: mínimo número de segundos que espera antes de levantar otro pod.

Además incluiremos el siguiente mapa:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

Que lo que me indica es que queremos una estrategia rollingUpdate en la cual bajaremos un pod y nunca tendremos uno extra de más. Quedando de la siguiente manera:

y para ejecutarlo lo haremos con el siguiente comando:

```
$kubectl apply -f mi-deploy.yml --record
```

```
Macintosh-2:~ xroot$ kubectl apply -f mi-deploy.yml --record
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
deployment.extensions/hello-deploy configured
Macintosh-2:~ xroot$
```

Un comando interesante para ver el status del deployment

```
kubectl rollout status deployment hello-deploy
```

Comprobaremos que el servicio funciona correctamente y volveremos a lanzar servidor web. y comprobaremos los pods en funcionamiento. Y hemos pasado de tres a dos pods y la imagen de apache

Revisando el historial de revisiones:

```
Macintosh-2:~ xroot$ kubectl rollout history deployment hello-deploy
deployment.extensions/hello-deploy
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=mi-deploy.yml --record=true

Macintosh-2:~ xroot$
```

Puede ser que tengamos problemas a la hora de refrescar y ver correctamente el servidor web nuevo, para ello podemos acceder directamente a un pod de los nuevos creados y ver si realmente lo ha creado como toca. Incluso podemos ir cambiando el puerto donde el servicio se muestra por problemas en la caché.

Links de interés: https://kubernetes.io/docs/reference/kubectl/cheatsheet/