Troubleshooting Kubernetes Deployments Applications

- Introducción a Kubernetes
- Conceptos básicos
- Despliegue de Aplicaciones
- Demo
- Estrategia de despliegues
- Helm despliegue de aplicaciones
- Troubleshooting Deployments

Contenedores en cluster

- **Orquestadores de contenedores**

https://docs.docker.com/engine/swarm/

http://kubernetes.io/

http://mesos.apache.org/

Contenedores en cluster

- **Orquestadores de contenedores**
  - Software encargado de **gestionar la ejecución** de contenedores en un **cluster** de máquinas
  - Pueden tratar **varios contenedores** como una única unidad lógica (**aplicación**)
  - **Servicios ofrecidos**
    - **Redes aisladas** para contenedores de la misma app
    - Políticas de **reinicio** en caso de **caída** del servicio
    - Políticas de **replicación** de contenedores por carga

Contenedores en cluster

- **Servicios y contenedores**
  - **Un contenedor con todos los servicios:** Una aplicación se puede empaquetar como una imagen docker con todos los servicios incluidos (web, BBDD, caché, etc…).
  - **Un contenedor por servicio:** También puede empaquetarse como muchas imágenes diferentes. Al arrancar habrá varios contenedores comunicados entre sí por red

Contenedores en cluster

- **Un contenedor con todos los servicios**
  - **Ventajas**
    - Más fácil de crear y probar (sólo un Dockerfile)
    - La comunicación de los servicios es siempre por localhost (más sencilla)
    - Se descarga de forma automática con un comando (ideal para distribuir)

Contenedores en cluster

- **Un contenedor con todos los servicios**
  - **Desventajas**
    - Un contenedor sólo puede escalar verticalmente (con una máquina más potente), pero no horizontalmente (usando varias máquinas)
    - No es tolerante a fallos (un error en un servicio afecta a todos los demás)

Contenedores en cluster

- **Un contenedor por servicio**
  - **Ventajas**
    - Las aplicaciones pueden escalar horizontalmente en un cluster
    - Cada contenedor puede estar en una máquina diferente
    - Se pueden clonar servicios que necesitan más potencia de cómputo en varias máquinas (web, servicios *stateless*)
    - Es tolerante a fallos (un servicio se cae y se puede reiniciar)

Contenedores en cluster

- **Un contenedor por servicio**
  - **Desventajas**
    - Puede ser más difícil de crear si tu código está en varios contenedores
    - Para webs sencillas se puede tener un contenedor para la BBDD y otros para la web (o varios)
    - Existen muchas formas diferentes de gestionar una aplicación formada por varios servicios
      - Una sola máquina: **docker-compose**
      - Cluster: **orquestadores de contendores**

Que es Kubernetes

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

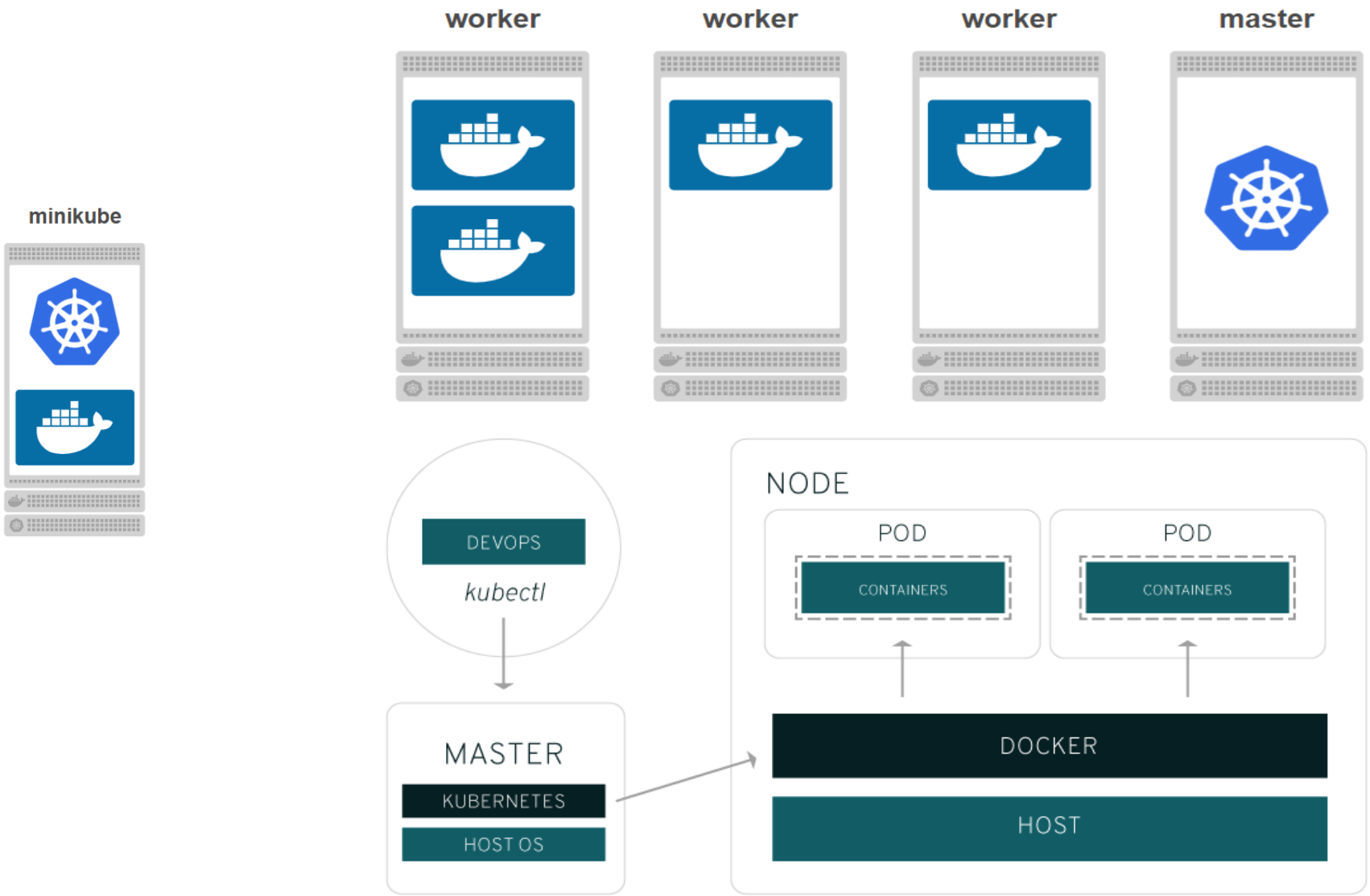
Que es Kubernetes

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, sobre todo para las empresas que hace software enterprise que ejecuta en la infraestructura del cliente

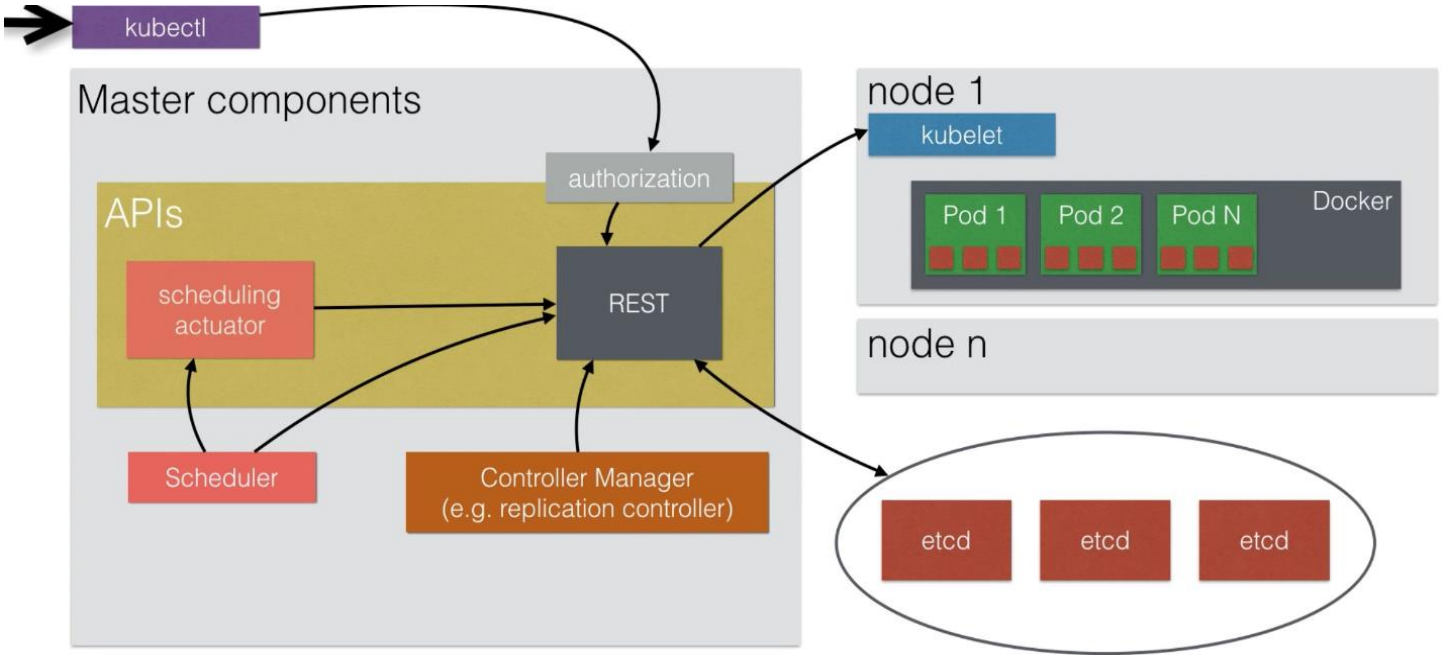
Arquitectura Básica Kubernetes



Arquitectura Básica 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.



Kubernetes Arquitectura

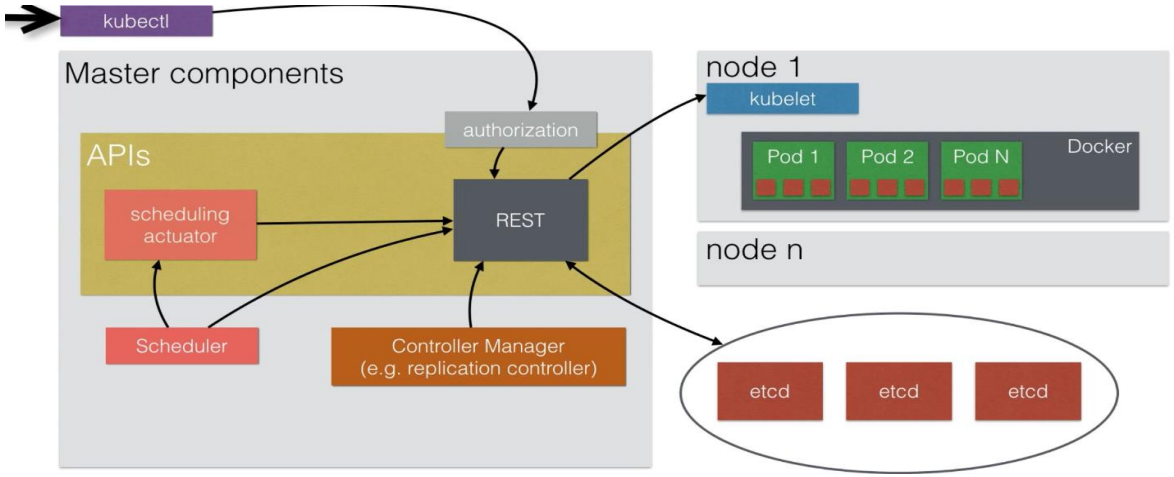
¿Qué es el nodo maestro, y qué hace?

Cuando le pide a Kubernetes que cree una implementación, utiliza kubectl una herramienta de línea de comandos para interactuar con el clúster.
kubectl envía una solicitud de API con el contenido del archivo YAML a la API de Kubernetes.
La API almacena el estado del recurso en una base de datos.

Etcd: Es el servicio de disponibilidad de cluster desarrollado por CoreOS y utilizado tambien por Centos/REDHAT. El demonio etcd es el encargado del almacenamiento distribuido.Utiliza una clave llave/valor que puede ser distribuida y leída por múltiples nodos. Kubernetes utiliza etcd para guardar la configuración y el estado del cluster leyendo la metadata de los nodos.

API server: API Server de Kubernetesse encarga de validar y configurar los datos de los objetos api que incluyen los pods, servicios, replicationcontrollers,etc. El servidor de API sirve para operaciones REST y proporciona al fronten del estado compartido del clúster a través del cual interactúan todos los demás componentes.

Arquitectura Básica Kubernetes

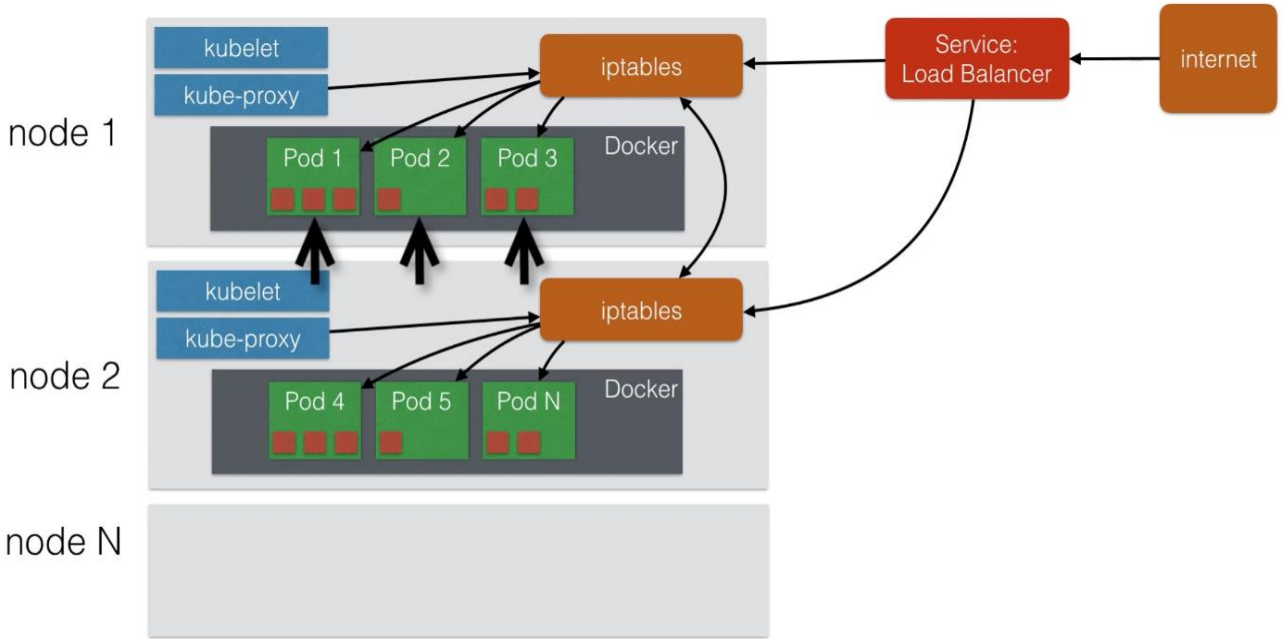


En el master node la lógica de la capa de control se fundamenta en **`etcd`**, una base de datos distribuida que nos asegura la consistencia de los datos usando el protocolo RAFT. Sobre **`etcd`** tenemos la capa REST, que es a través de la que pasan todos los accesos a la base de datos. Esa capa REST 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 master 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.

Arquitectura Básica Kubernetes



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*.

Kubernetes Arquitectura

ControllerManager Server:

Es un servicio usado para manejar el proceso de replicación de las tareas definidas. Los detalles de estas operaciones son escritos en el etcd, donde el controllermanager observa los cambios y cuanto un cambio es detectado, el controllermanager lee la nueva información y ejecuta el proceso de replicación hasta alcanzar el estado deseado

SchedulerServer:

Es el servicio que se encarga balancear la carga de los servidores y elegir el nodo mas saludable(procesamiento, memoria,etc) a la hora de deployar un pod. Tambien se utiliza para leer los requisitos de un pod, analizar el ambiente del clustery seleccionar el nodo más performance. Este servicio adicionalmente cumple la función de monitorear el estado de los recursos de los nodos en realtime.

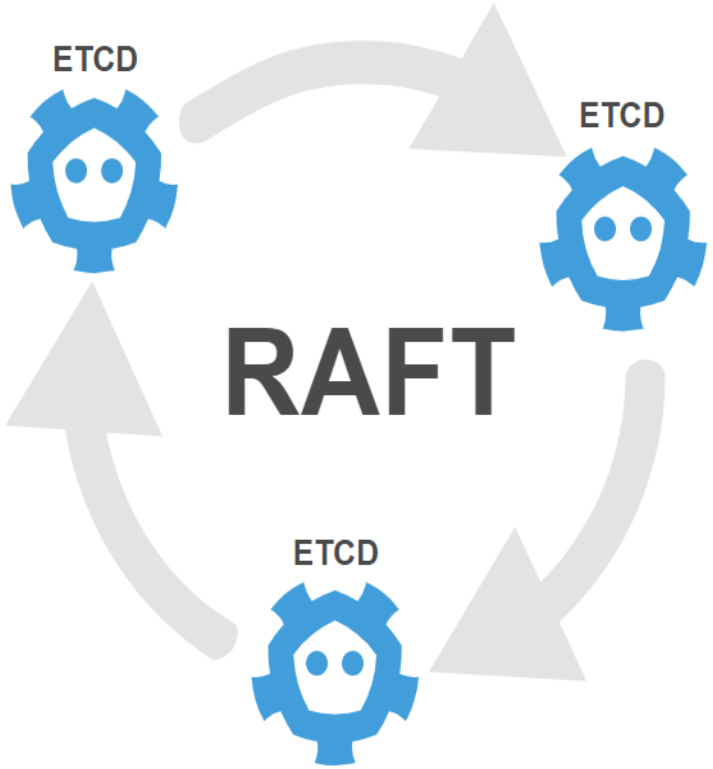
Kubernetes

etcd - Almacén clave-valor

etcd es un almacén de clave-valor.
La razón por la que es tan popular y se usa en Kubernetes es que está diseñado para ejecutarse en sistemas distribuidos.

Cuando inicias una colección de instancias de etcd, comienzan a hablar entre sí y forman una red.

Eventualmente eligen a un líder y el resto de los casos se convierten en seguidores.
Antes de escribir cualquier valor en el disco, las bases de datos acuerdan ese valor primero y luego lo persisten.
Las bases de datos de etcd coordinan la lectura y escritura utilizando el protocolo RAFT



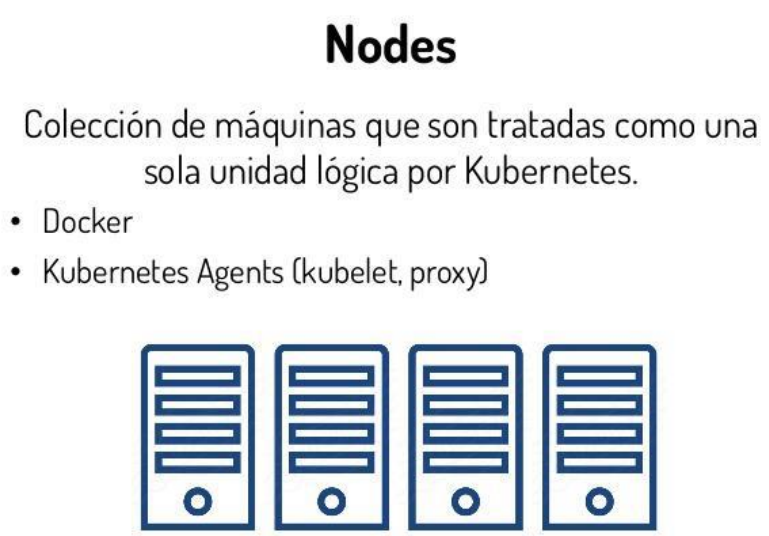
Kubernetes

Kubernetes Minion

Un minión es un nodo/servidor que forma parte de un Cluster, este nos aporta su Computo (procesamiento,memoria,etc) y los suma a los recursos del Cluster. Para poder lograr esto el minion utiliza los siguientes demonios que lo mantienen en contacto con el Master.

Kubelet: Este servicio se encarga de obtener las instrucciones del Master y realizar los cambios correspondientes dentro del nodo.Otra de sus funciones manejar la creación y configuración de los pods(imagenes,volumenes,etc).

Kube-Proxy: Este servicio monitorea los Servicios y Endpoints levantados en el master. Nos provee la apertura de un puerto especifico/aleatorio en el pod para que pueda ser accedido y configura las reglas de firewalls requeridas.



Kubernetes Conceptos Básicos

Pod

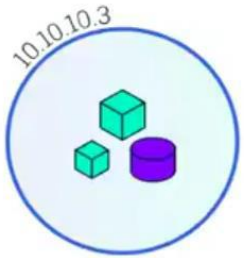
La unidad fundamental de despliegue en Kuberneteses el Pod. Un pod sería el equivalente a la mínima unidad funcional de la aplicación.

En general, un pod contendrá únicamente un contenedor, aunque no tiene que ser así: si tenemos dos contenedores que actúan de forma conjunta, podemos desplegarlos dentro de un solo pod. Dentro de un pod todos los contenedores se pueden comunicar entre ellos usando localhost, por lo que es una manera sencilla de desplegar en Kubernetes .

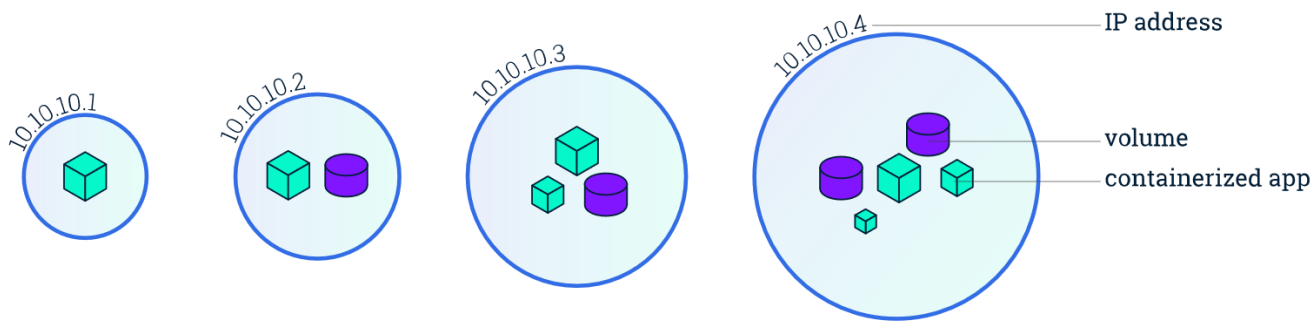
En este sentido, todos los contenedores dentro de un pod se podría decir que están instaladas en una mismo equipo (como un stack LAMP).
Sin embargo, un pod es un elemento no-durable, es decir, que puede fallar o ser eliminado en cualquier momento. Por tanto, no es una buena idea desplegar pods individuales en Kubernetes.

Kubernetes Pod

```yaml
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



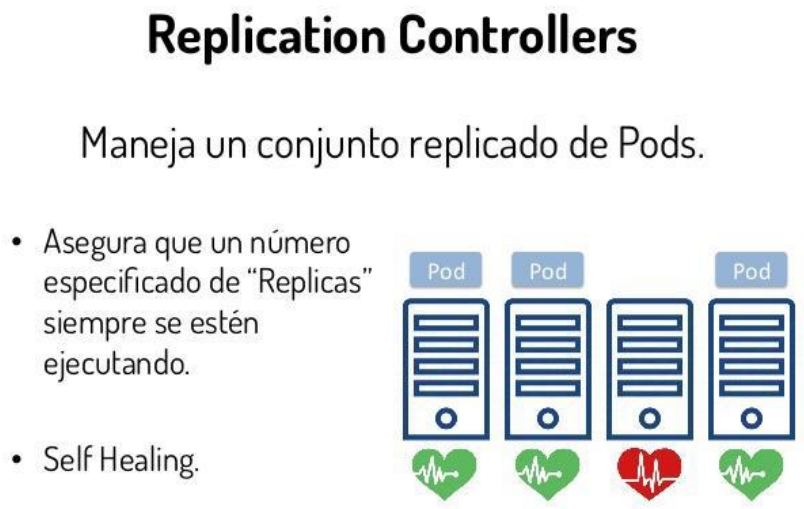
Kubernetes Conceptos Básicos

Replication Controller

Replication Controllerse encarga de mantener un determinado número de réplicas del pod en el clúster.

El Replication Controller asegura que un determinado número de copias -réplicas-del pod se encuentran en ejecución en el clúster en todo momento. Por tanto, si alguno de los pods es eliminado, el ReplicationControllerse encarga de crear un nuevo pod. Para ello, el ReplicationController incluye una plantilla con la que crear nuevos pods.

Así, el Replication Controller define el estado deseado de la aplicación: cuántas copias de mi aplicación quiero tener en todo momento en ejecución en el clúster. Modificando el número de réplicas para el Replication Controller podemos escalar(incrementar o reducir) el número de copias en ejecución en función de las necesidades.



Kubernetes Conceptos Básicos

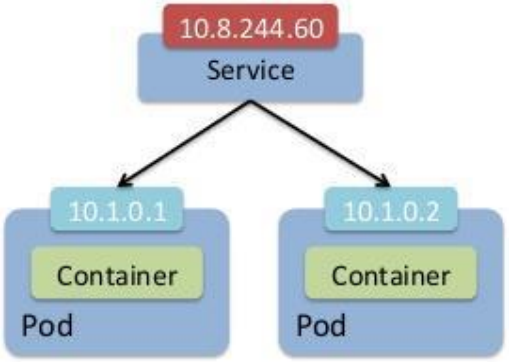
Services: Es el microservicio de una aplicación la cual fue deployada en un pod o ReplicationController y se puede conectar a otro servicio para Lograr la aplicación final.

Un ejemplo de esto puede ser un replicationcontroller llamado WordPress(nos realiza un deployde 3 containers/pod), el cual es asociado a un servicio con el mismo nombre y se conecta al replicationcontrol a través de un Label/Etiqueta.
Este servicio llamado WordPress ,lo podemos conectar a otro servicio llamado MYSQL(Mismo proceso que WordPress con el ReplicationController).

Services

Service Discovery para los Pods.

- Endpoints persistentes para los Pods.
- Backend dinámico basado en Labels.



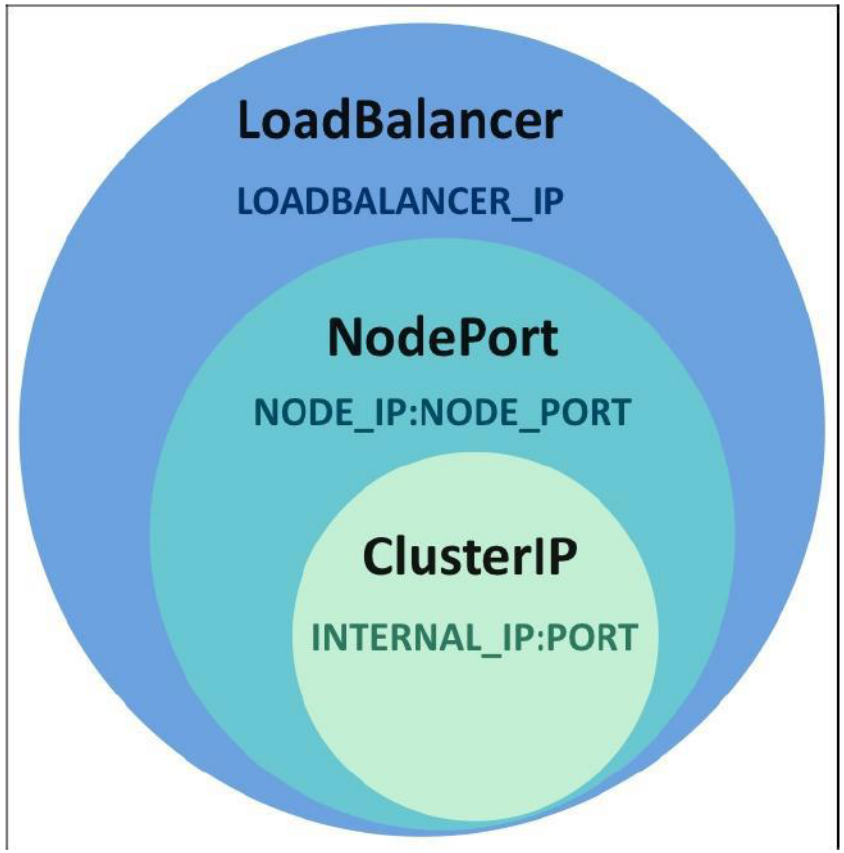
Kubernetes

Type Services

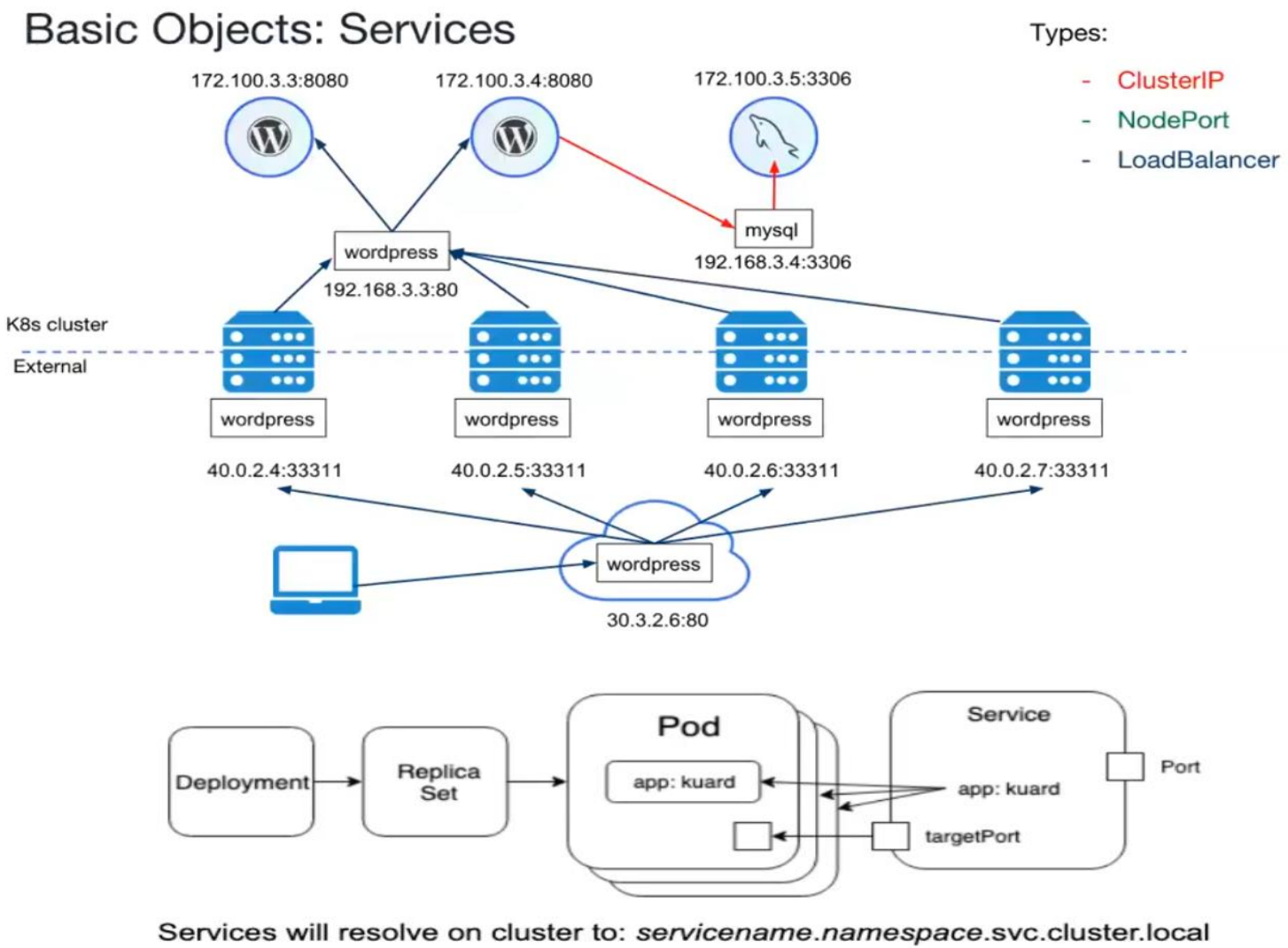
ClusterIP: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default ServiceType

NodePort: Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

LoadBalancer: Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.



Kubernetes Services



Kubernetes

Labels

Son utilizados para organizar y seleccionar un grupo de objetos basado en pares del tipo key:value. Labels son fundamentales para conectar los servicios a los replicationcontrollers, pods...

Deployment

El Deployment añade la capacidad de poder actualizar la aplicación definida en un ReplicationController sin pérdida de servicio, mediante actualización continua(rollingupdate).
Si el estado deseado de la aplicación son tres réplicas de un pod basado en cliente/app-1.0 y queremos actualizar a cliente/app-2.0, el Deployment se encarga de realizar la transición de la versión 1.0 a la 2.0 de forma que no haya interrupción del servicio. La estrategia de actualización puede definirse manualmente, pero sin entrar en detalles, Kubernetes se encarga de ir eliminado progresivamente las réplicas de la aplicación v1.0 y sustituirlas por las de la v2.0.
El proceso se hace de forma controlada, por lo que si surgen problemas con la nueva versión de la aplicación, la actualización se detiene y es posible realizar marcha atrás hacia la versión estable.

Kubernetes 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 nuestras 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.

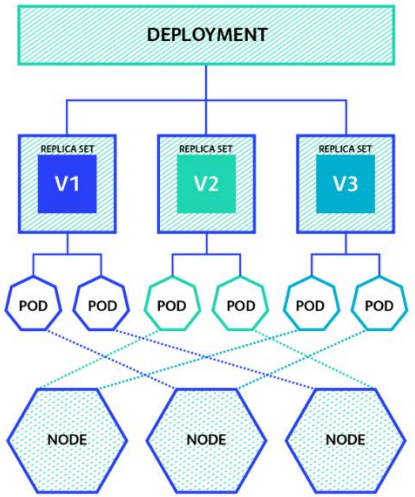
Filtrando pods, por label:

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

Kubernetes Deployments

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.



Kubernetes

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.

Kubernetes

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)

Kubernetes

Deployments

- This is an example of a deployment:

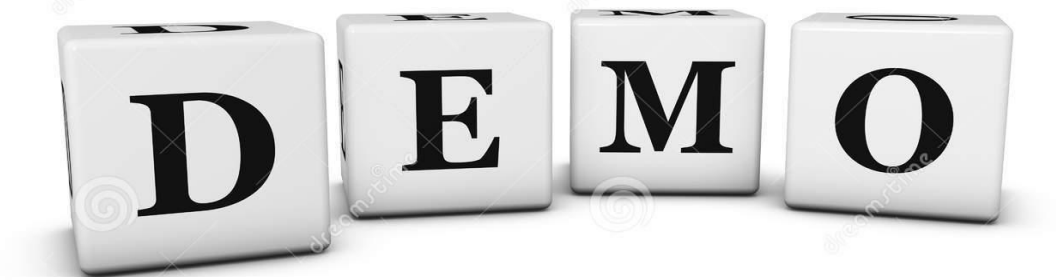
```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: wardviaene/k8s-demo
        ports:
        - containerPort: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

Kubernetes

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

Kubernetes



kubernetes

Kubernetes Ingress

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.

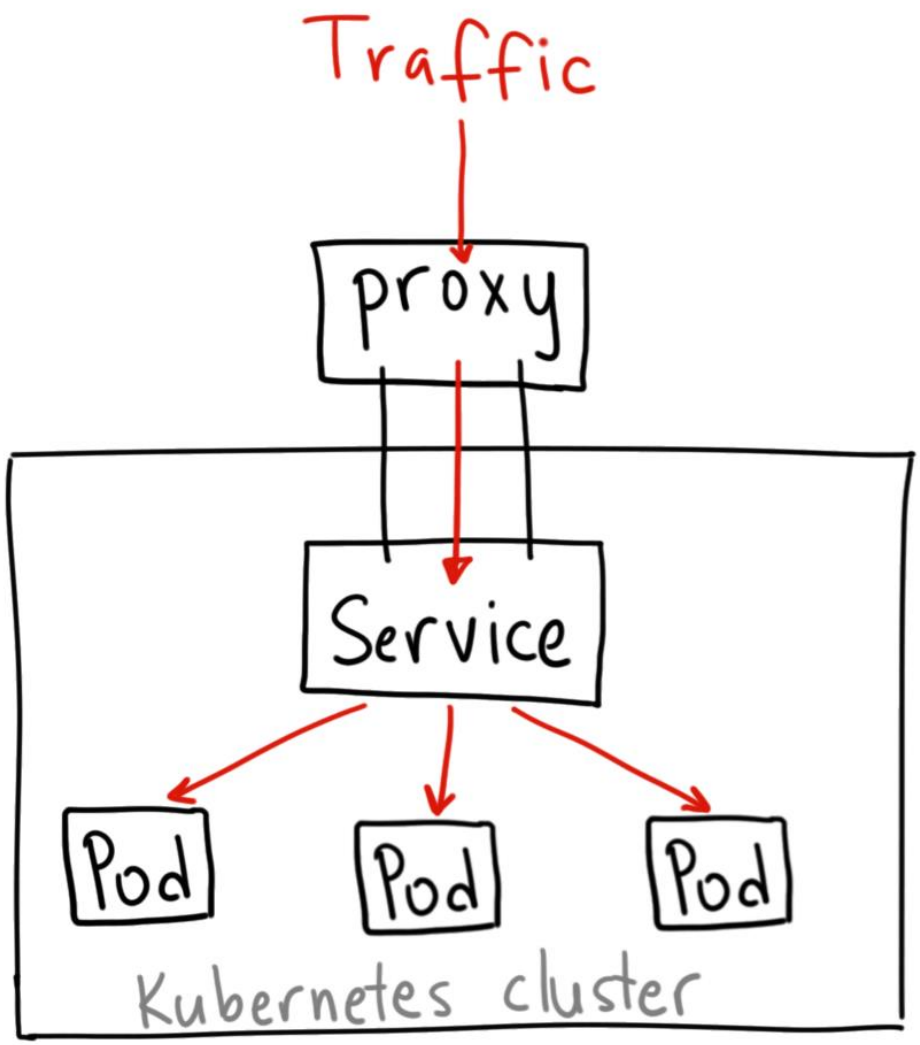
Un Ingress nos permite definir rustas de entrada a nuestro servicio de una manera programática. Existen numerosos Ingress Controllers, como el de Nginx, HAProxy o traefik, cada uno de ellos permitiendo distintas configuraciones.

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.

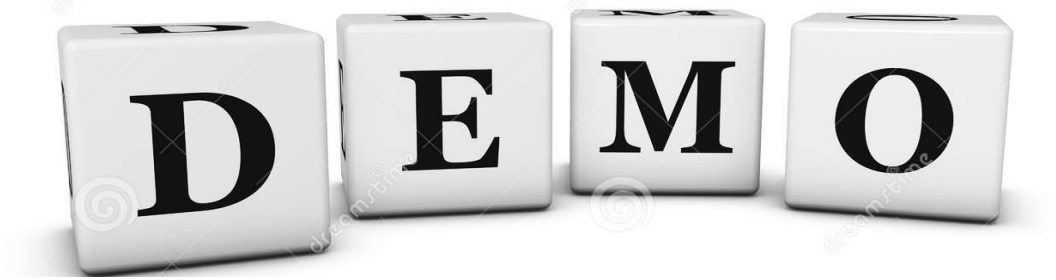
En el laboratorio veremos como redireccionar el tráfico de entrada a mi clúster en función de la cabecera `Host` de la petición entrante.

Hay aplicaciones que dependen mucho de la ruta donde se expongan, entonces como programadores o administradores tenemos que saber en que rutas o dominios se expone su aplicación.

Kubernetes Ingress



Kubernetes



kubernetes

Kubernetes Healthchecks

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

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 healchecks

Por comandos:

Podemos configurar la ejecución de comando periódico que se ejecuta en contexto de uno de los contenedores de mi pod.

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

Kubernetes Healthchecks

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 esta lista aunque yo como contenedor estoy listo si no tengo la cache de redis no podre 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.

Kubernetes Healthchecks

Resumen:

- **Readiness:** ¿El contenedor esta listo para recibir trafico?
- **Liveness:** ¿El contenedor sigue vivo?

El flujo es el siguiente:

- Si **Readiness** falla, `Kubernetes detiene el trafico 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

Kubernetes Healthchecks

kubernetes

Kubernetes Canary Deployments

Canary deployments

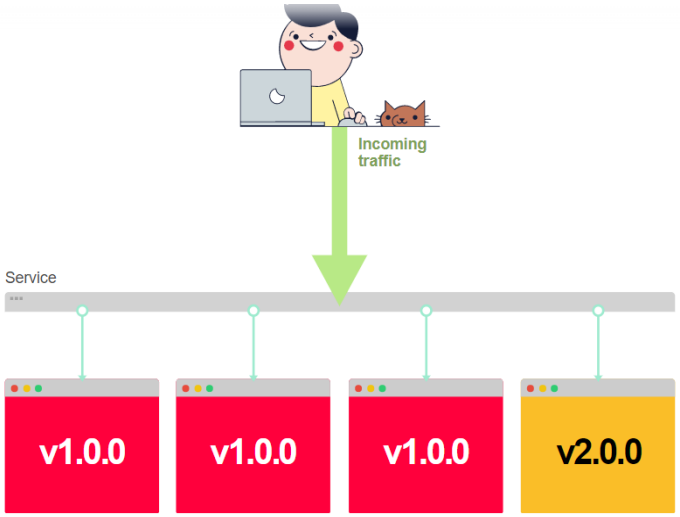
Otra opción para desplegar en producción sin interrumpir el tráfico en vivo es usar una implementación de Canary.

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 desee.**

Es una práctica común crear solo unos pocos Pods con la versión actual y monitorear el tráfico en vivo que llega a ellos.

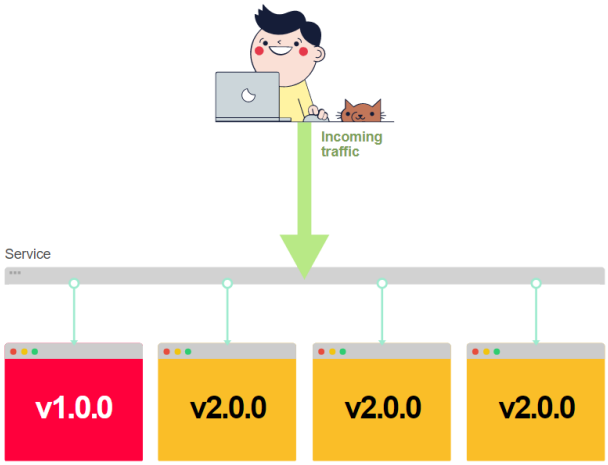
Kubernetes Canary Deployments



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.

Kubernetes Canary Deployments



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.

Kubernetes Canary Deployments

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 back-end: component: backend.

Puedes tener cualquier número de etiquetas.

De hecho, podría etiquetar sus Pods con una etiqueta para la versión como esta:

pod.yaml

```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.

Kubernetes Canary Deployments

Se utiliza un selector para apuntar Pods con una etiqueta específica como `component: frontend`.

service.yaml

```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: frontend` 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.

Kubernetes Canary Deployments

Se utiliza un selector para apuntar Pods con una etiqueta específica como `component: frontend`.

service.yaml

```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: frontend` de etiqueta.

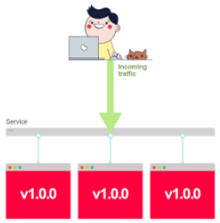
Cuando un conjunto de Pods comparte la misma etiqueta, el Servicio enruta el tráfico a todos ellos, incluso si los Pods pertenecen a deployments diferentes.

Kubernetes Canary Deployments

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 deployemnts: 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.

Kubernetes Canary Deployments

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.

Kubernetes Canary Deployments

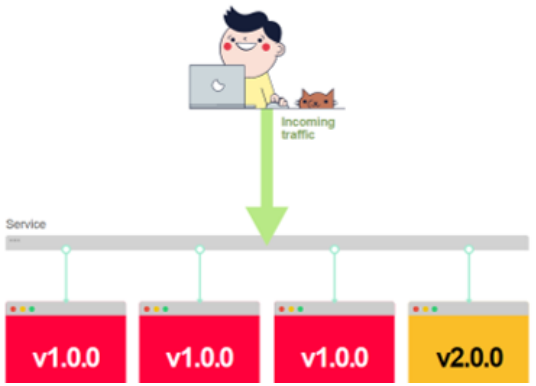
Pero necesita una etiqueta compartida para que el selector de su Servicio dirija el tráfico a ambos.

Puede agregar una etiqueta **component: frontend** a ambos y tener el Nombre de servicio de destino en lugar de la versión.

service.yaml

```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.



Kubernetes Canary Deployments

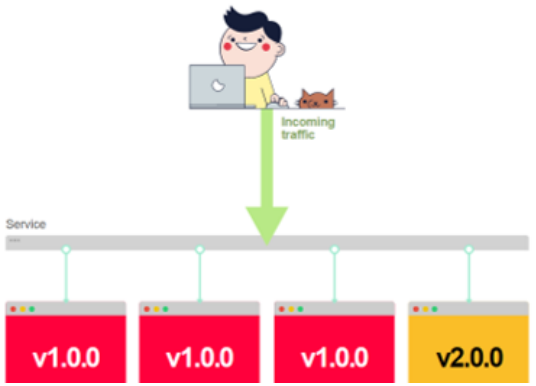
Pero necesita una etiqueta compartida para que el selector de su Servicio dirija el tráfico a ambos.

Puede agregar una etiqueta **component: frontend** a ambos y tener el Nombre de servicio de destino en lugar de la versión.

service.yaml

```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.

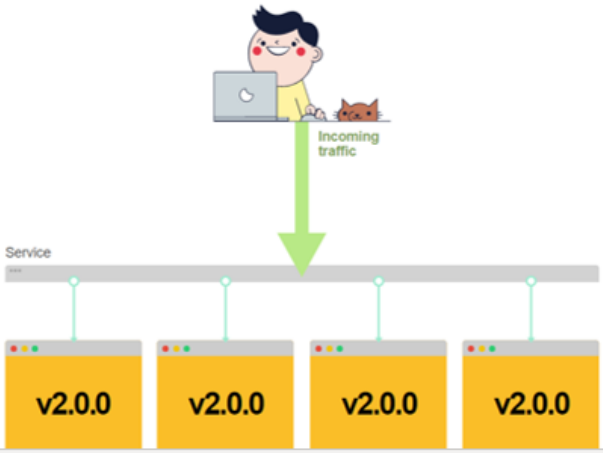


Kubernetes Canary Deployments

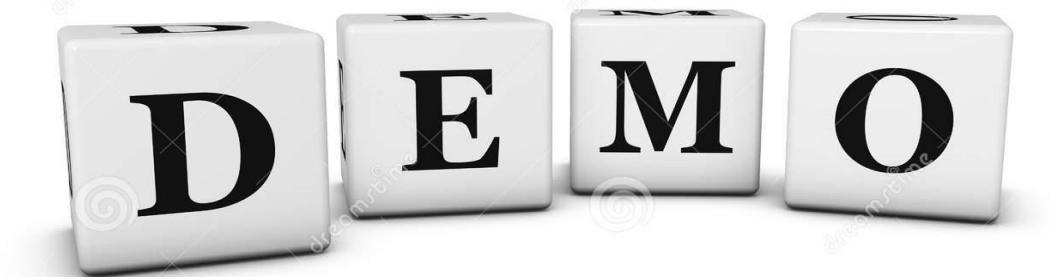
Cuando esté dispuesto a cambiar todo el tráfico, puede editar el Servicio y señalarlo en version: 2.0.0 en lugar de la etiqueta compartida.

service.yaml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: canary-service
spec:
  selector:
    version: "2.0.0"
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```



Kubernetes



kubernetes

Kubernetes 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.

deploy-blue.yaml

```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
```

Kubernetes Blue-Green Deployments

Y crea un archivo **service-blue-green.yaml** con el siguiente contenido:

service-blue-green.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: entry-point
spec:
  ports:
  - port: 80
    targetPort: 8080
  type: NodePort
  selector:
    version: "1.0.0"
```

Kubernetes Blue-Green Deployments

Creamos otro deployment para una versión más nueva de la aplicación: **versión 2.0.0**

deploy-green.yaml

```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
```

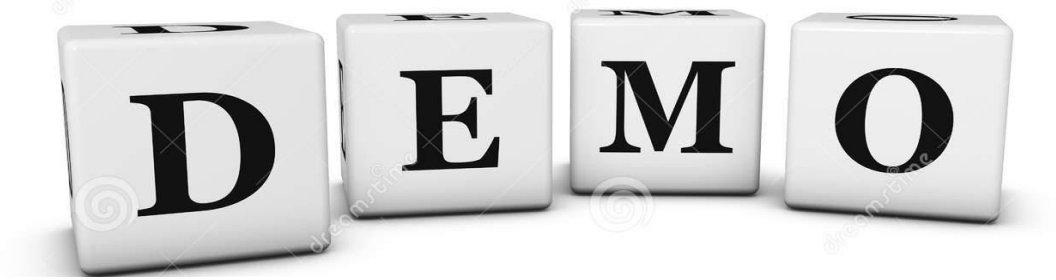
Kubernetes Blue-Green Deployments

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:**

```
kubectl edit service entry-point
```

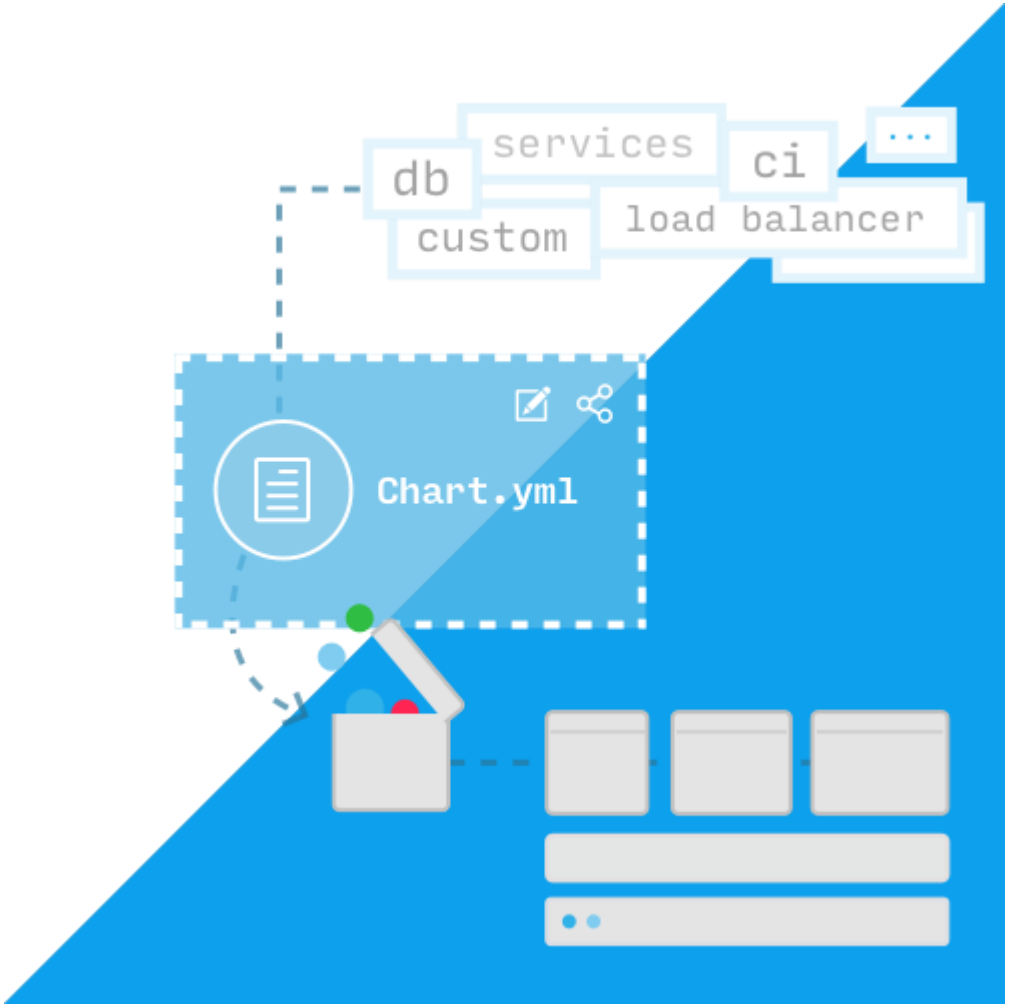
Kubernetes



kubernetes

Kubernetes HELM

https://helm.sh/



Kubernetes HELM

Helm es un gestor de paquetes para kubernetes. Esto es, una herramienta que permite crear, distribuir e instalar aplicaciones en una plataforma, de una manera sencilla y estándar.
Toma por ejemplo NPM para NodeJS o pip para Python.

En este caso, Helm es un gestor para Kubernetes. Verás que existen cientos de aplicaciones listas para instalar en tu clúster. Que puedes crear y distribuir las tuyas propias de manera sencilla, a partir de los YAML con los que despliegas en Kubernetes.

Es un proyecto mantenido por la Cloud Native Computing Foundation (CNCF)

Más sobre Helm en su página principal: helm.sh.

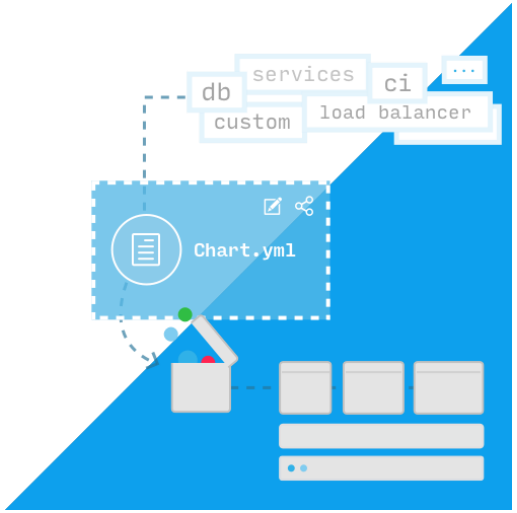
Helm significa timón en inglés. Esto tiene que ver con el significado de Kubernetes, que en griego antiguo era el capitán o timonel. Simboliza que Helm es una herramienta para controlar mejor la plataforma de Kubernetes.

Kubernetes HELM

HELM

- Aplica estos principios a Kubernetes
- Define una estructura y un lenguaje(YAML + Go) para las aplicaciones
- Proporciona un repositorio para compartirlas
- Mantenido por CNCF Permite reutilizar recursos entre distintos clusters de Kubernetes

Permite reutilizar recursos entre distintos clusters de kubernetes.



Kubernetes HELM

Ventajas de utilizar Helm

El principal uso de esta herramienta es **la instalación de aplicaciones de terceros**, para lo que ofrece esta serie de ventajas:

- Tiene cientos de paquetes disponibles en su repositorio oficial, listos para ser instalados en tu clúster con un solo comando.

- Estos paquetes son **completamente personalizables**, de forma que lo puedes aceptar a tu entorno, a las particularidades de tu clúster o a tu caso de uso.

- Cada vez más proveedores oficiales, es decir, los creadores de contenido original, cómo pueden ser las empresas detrás de los desarrollos de bases de datos, de servidores web o de herramientas DevOps, se suman a la ola de publicar sus desarrollos directamente como paquetes de Helm, para que puedan ser instalados automáticamente en cualquier clúster. De hecho **se ha convertido en el estándar de facto para la distribución de aplicaciones en Kubernetes**.

- Además, todas estas aplicaciones al proceder de los proveedores originales y a tener empresas detrás que se preocupan por estos desarrollos, **van a estar siempre actualizadas a última versión**.

Kubernetes HELM

Componentes y Vocabulario

CLI (Command Line Interface)

Es la aplicación para el usuario de Helm. Desde ahí puedes realizar todas las funciones.
Internamente y de forma transparente se comunica con el Tiller.
Disponible para todos los SOs, es la aplicación de consola, se comunica solamente con el Tiller.

Tiller

Es un agente que está desplegado en el clúster de Kubernetes.
En ningún momento se interactúa directamente con él, el CLI es el único punto de interacción con el usuario.
El Tiller simplemente recibe acciones del CLI y aplica los cambios conectándose con Kubernetes y crea las releases y ejecuta el código dinamico

API de Kubernetes

Al final, Helm es solo una capa por encima de Kubernetes. Todo lo que haces con Helm se traduce en comandos que Helm envía automáticamente a la API de Kubernetes, como cuando utilizas `kubectl`.
De nuevo, la comunicación con Kubernetes se realiza de forma transparente para el usuario, que solo utiliza el CLI.

Kubernetes HELM

Vocabulario

Chart

Chart es como se llama a una aplicación en Helm. Es sinónimo de paquete. Un chart define cómo es esa aplicación y es la pieza que se distribuye en los repositorios.

Template

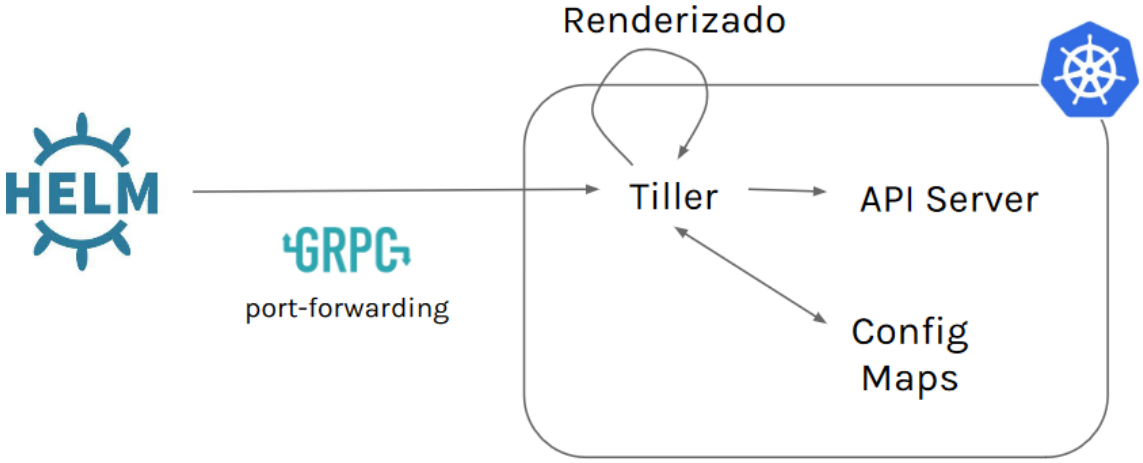
Los chart se componen básicamente de templates. Un template es simplemente un archivo YAML de Kubernetes, que se ha modificado (o no) para que se pueda personalizar. Estos templates al instalar la aplicación se convierten en los YAML que se despliegan en Kubernetes, al final es un archivo YAML de kubernetes con partes deinamicas en Go.

Release

Release es como se llama a las instalaciones concretas de una aplicación (chart). En tú clúster puedes tener varias releases del mismo chart, por ejemplo con distintas versiones o con distintos parámetros.

Es la aplicación desplegada en kubernetes a partir de un chart

Kubernetes HELM



Helm se compone de dos partes el cliente y el tiller que esta desplegado en kubernetes y se comunicar a través de port-forwarding, lo que hace es un túnel entre nuestro local y el clúster de kubenetes que seria el pod del tiller, esto se realiza automáticamente a través de la API de kubernetes, cuando se levanta este túnel se comunica a través de GRPC (*gRPC es un sistema de llamada a procedimiento remoto de código abierto desarrollado inicialmente en Google*.), esta especialmente pensado para que se comuniquen dos piezas de software entre si.

La interface de comandos no realiza ninguna carga de trabajo, simplemente le pasa las acciones al tiller, el tiller es el que realiza el trabajo, la mayor tarea que tiene el tiler es el renderizado, el renderizado es coger el código dinámico que tiene los templates del chart y renderizarlos con los valores personalizador que nosotros queramos, nombre, permisos, etc…

Kubernetes HELM

Instalación de Aplicaciones en Helm

Repositorios de aplicaciones para Kubernetes

Son APIs (normalmente con una web para consultarlo desde el navegador) a las que accede el CLI de Helm para poder instalar los charts.

Kubeapps Hub es el estándar, viene configurado por defecto en la instalación de Helm (con el nombre *stable*). Es mantenido por la CNCF y solo contiene aplicaciones preparadas para producción.

Aparte del repositorio *stable* en KubeApps puedes encontrar muchos otros repositorios de distintos proveedores, ya mantenidos por cada creador. También encontrarás uno llamado *incubator* que en este caso también es oficial, y contiene aplicaciones que están siendo mejoradas para promocionar a *stable*

Puedes acceder a KubeApps Hub, buscar entre los charts que ofrece y los distintos proveedores y repositorios en:

https://hub.kubeapps.com/

HELM Comandos Básicos Helm (I)

helm init # Inicializa el entorno de Helm. Crea y gestiona el agente Tiller.
helm repo update # Actualiza la información de los repositorios
helm install # Instala un chart en tu clúster. Esto es, crea una nueva release a partir de un chart.
helm upgrade # Actualiza una release
helm ls # Lista todas las releases (instalaciones) que hay desplegadas en tu clúster
helm get # Información sobre una release existente
helm history # Muestra las versiones de una release
helm rollback # Permite volver a una versión anterior de la release
helm delete # Elimina una release del cluster

HELM Comandos Básicos Helm (II)

helm init --upgrade # Upgradea el tiller a la ultima versiones, es decir a la del cliente local Helm, si actualizamos la versión de Helm, tendremos que actualizar el tiller a esa versión.

helm init –help # Nos dará la ayuda del comando.

helm install stable/joomla --name joomla1 --namespace nombrenamespace

helm upgrade joomla1 stable/joomla # Upgradeamos el chart de nuestro despliegue

helm ls -a # Nos permite ver todas las releases que tengamos en el cluster, es decir incluso las que hemos borrado, es interesante el campo STATUS

helm get joomla1 # Nos permite inspeccionar una reléase.

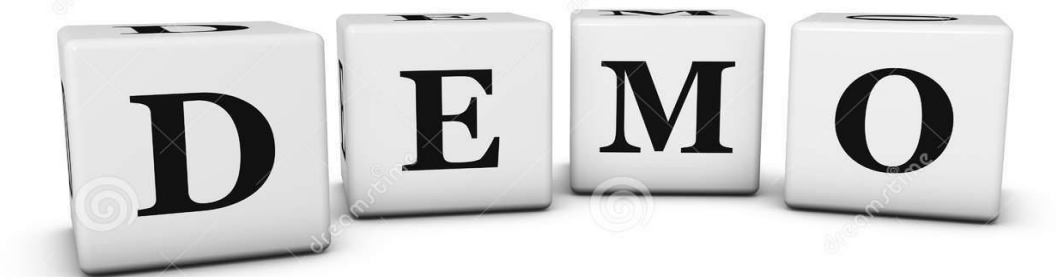
helm get notes joomla1 # Nos vuelve a mostrar las notas que ha lanzado cuando lo hemos desplegado.

helm history joomla1 # Permite inspeccionar una versión, mas que versiones se les llama revisiones.

helm rollback joomla1 1 # Le damos para volver a atrás el nombre de la reléase y el numero de version

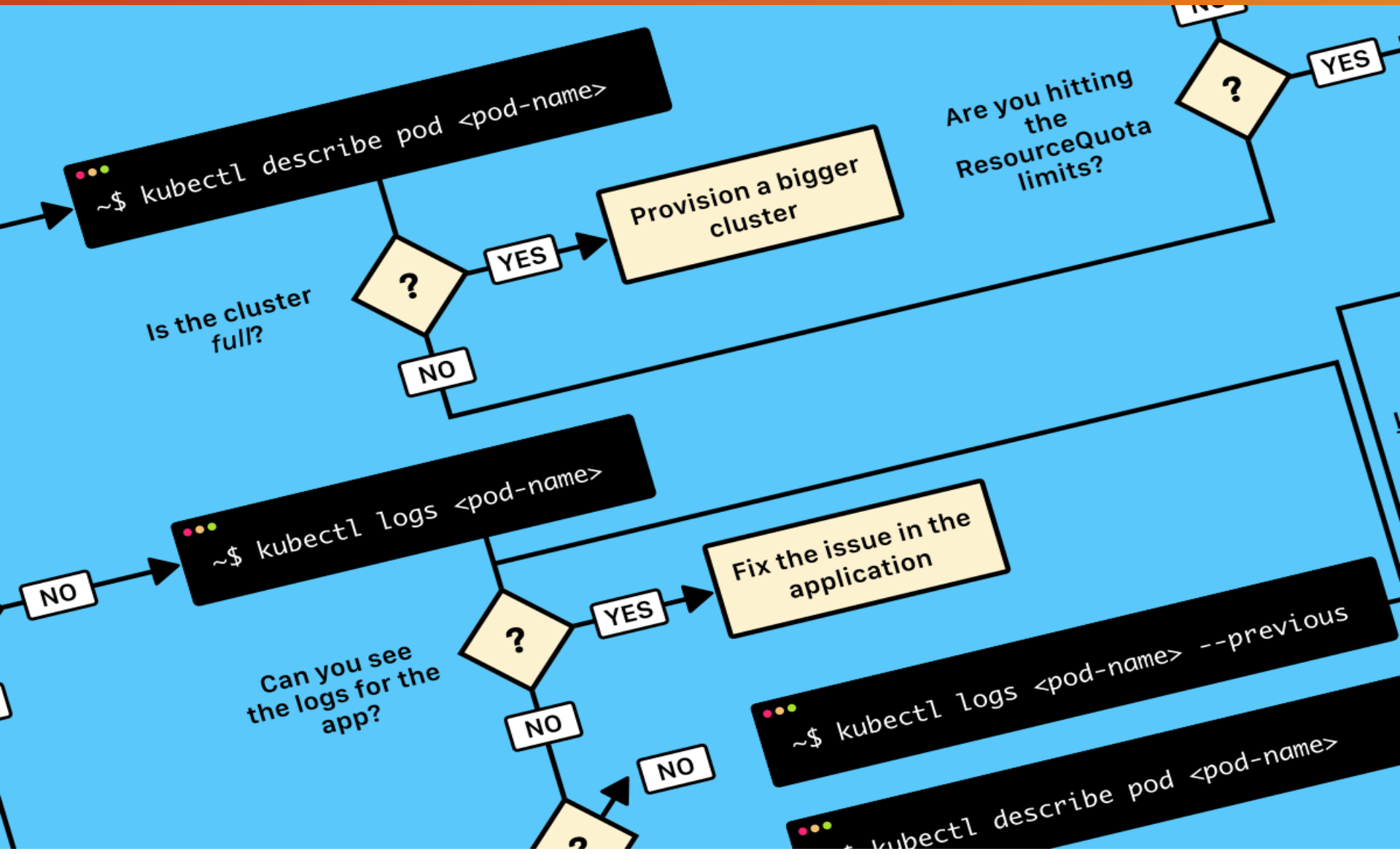
helm delete --purge joomla1 # si queremos eliminar la reléase y todo su contenido y que no quede rastro en nuestro kubernetes.

Kubernetes



kubernetes

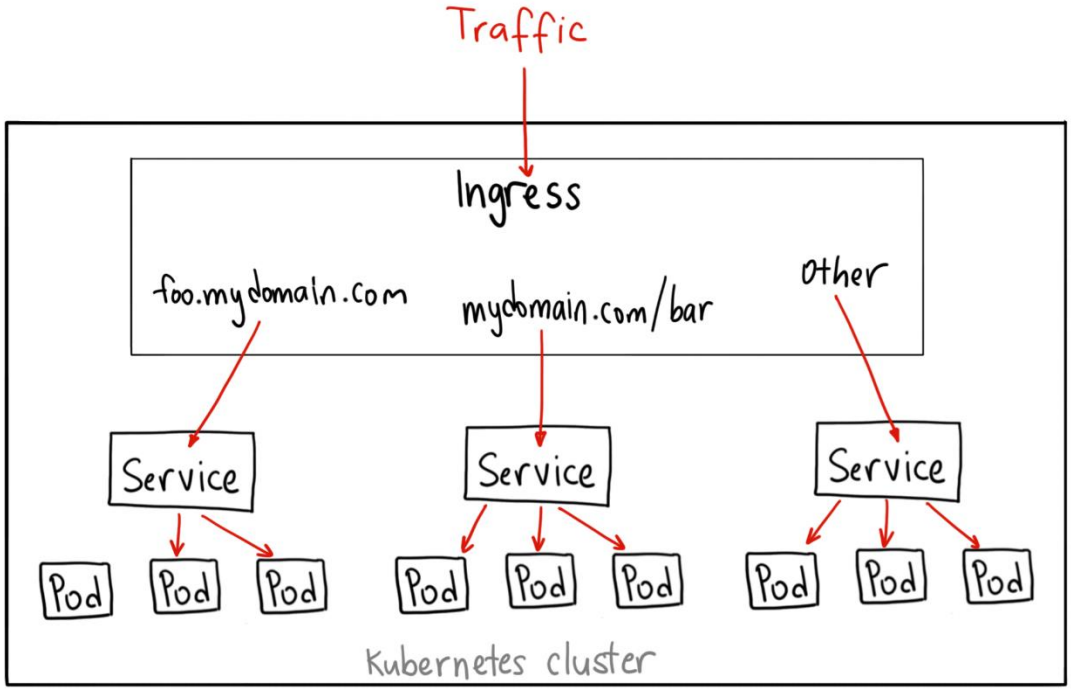
Troubleshooting Kubernetes Deployments



Troubleshooting Kubernetes Deployments

Cuando desea desplegar una aplicación en Kubernetes, generalmente define tres componentes:

- Un **deployment,** que es una receta para crear copias de su aplicación llamada Pods
- Un **servicio** : un balanceador de carga interno que enruta el tráfico a Pods
- Un **ingress** - una descripción de cómo el tráfico llega desde fuera del clúster para su servicio.



Troubleshooting Kubernetes Deployments

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    track: canary
spec:
  selector:
    matchLabels:
      any-name: my-app
  template:
    metadata:
      labels:
        any-name: my-app
    spec:
      containers:
      - name: cont1
        image: learnk8s/app:1.0.0
        ports:
        - containerPort: 8080
---
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    name: app
---
```

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
    paths:
    - backend:
        serviceName: app
        servicePort: 80
      path: /
```

Connecting Deployment and Service

El Servicio y el deployment no están conectados en absoluto.
En cambio, el Servicio apunta a los Pods directamente y omite el deployment por completo.
Entonces, a lo que debe **prestar atención es a cómo se relacionan los Pods y el Servicio**.

Debes recordar tres cosas:

- El selector del servicio debe coincidir con al menos una etiqueta del Pod
- El Servicio **targetPort** debe coincidir con el **containerPort** del contenedor dentro del Pod
- El servicio **port** puede ser cualquier número.

Varios servicios pueden usar el mismo puerto porque tienen diferentes direcciones IP asignadas

Troubleshooting Kubernetes Deployments

En la práctica, debería mirar estas líneas:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    track: canary
spec:
  selector:
    matchLabels:
      any-name: my-app
  template:
    metadata:
      labels:
        any-name: my-app
    spec:
      containers:
      - name: cont1
        image: learnk8s/app:1.0.0
        ports:
        - containerPort: 8080
---
```

Connecting Deployment and Service

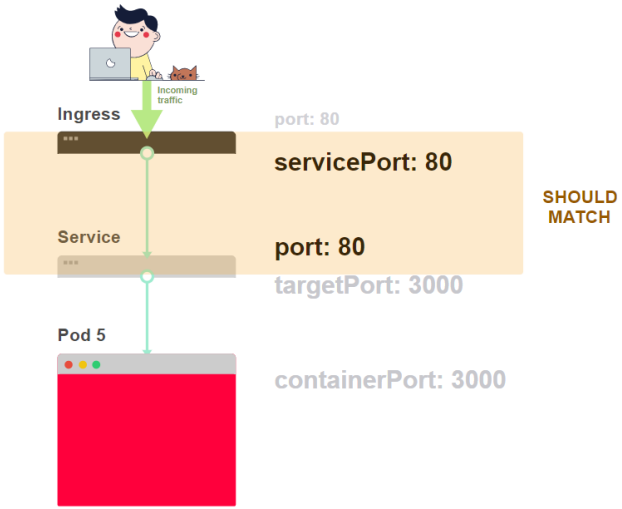
```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    any-name: my-app
```

Connecting Service and Ingress

El siguiente paso para exponer su aplicación es configurar el Ingress.
El Ingress tiene que saber cómo recuperar el Servicio para luego recuperar los Pods y enrutar el tráfico hacia ellos.
El Ingress recupera el Servicio correcto por nombre y puerto expuesto.

Deben coincidir dos cosas en Ingress y Service:

- El **servicePort** del Ingress debe coincidir con el **port** del Servicio
- El **serviceName** del Ingress debe coincidir con el **name** del Servicio



Connecting Service and Ingress

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
    paths:
    - backend:
        serviceName: my-service
        servicePort: 80
      path: /
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    any-name: my-app
---
```

Troubleshooting Kubernetes Deployments

¿Qué pasa cuando algo sale mal?

El Pod no se inicia o se está bloqueando.

3 pasos para solucionar problemas de despliegues de Kubernetes

Es esencial tener un modelo mental bien definido de cómo funciona Kubernetes antes de sumergirse en la depuración de un despliegue fallido.

Dado que hay tres componentes en cada deployment, debe depurarlos todos en orden, comenzando desde abajo:

1. Debes asegurarte de que tus Pods estén funcionando
2. Concéntrese en hacer que el Servicio enrute el tráfico a los Pods
3. Verifique que el Ingress esté configurado correctamente y se conecta al service

Troubleshooting Kubernetes Deployments

Hay cuatro comandos útiles para solucionar problemas de Pods:

- **kubectl logs <pod name>** es útil para recuperar los registros de los contenedores del Pod
- **kubectl describe pod <pod name>** es útil para recuperar una lista de eventos asociados con el Pod
- **kubectl get pod <pod name>** es útil para extraer la definición YAML del Pod tal como está almacenado en Kuberr
- **kubectl exec -ti <pod name> bash** es útil para ejecutar un comando interactivo dentro de uno de los contenedores del

Errores comunes de pods

Los pods pueden tener errores de inicio y tiempo de ejecución.

Los errores de inicio incluyen:

- ImagePullBackoff
- ImageInspectError
- ErrImagePull
- ErrImageNeverPull
- Registro no disponible
- InvalidImageName

Los errores de tiempo de ejecución incluyen:

- CrashLoopBackOff
- RunContainerError
- KillContainerError
- VerifyNonRootError
- RunInitContainerError
- CreatePodSandboxError
- ConfigPodSandboxError
- KillPodSandboxError
- SetupNetworkError
- TeardownNetworkError

Depuración del Ingress Nginx

El proyecto Ingress-nginx tiene un complemento oficial para Kubectl .

Puedes usar **kubectl ingress-nginx** para:

- inspeccionar registros, backends, certs, etc.
- conectarse a la entrada
- examinar la configuración actual

Los tres comandos que debes probar son:

- **kubectl ingress-nginx lint**, que comprueba el nginx.conf
- **kubectl ingress-nginx backend**, para inspeccionar el backend (similar a kubectl describe ingress <ingress-nam
- **kubectl ingress-nginx logs**, para verificar los registros

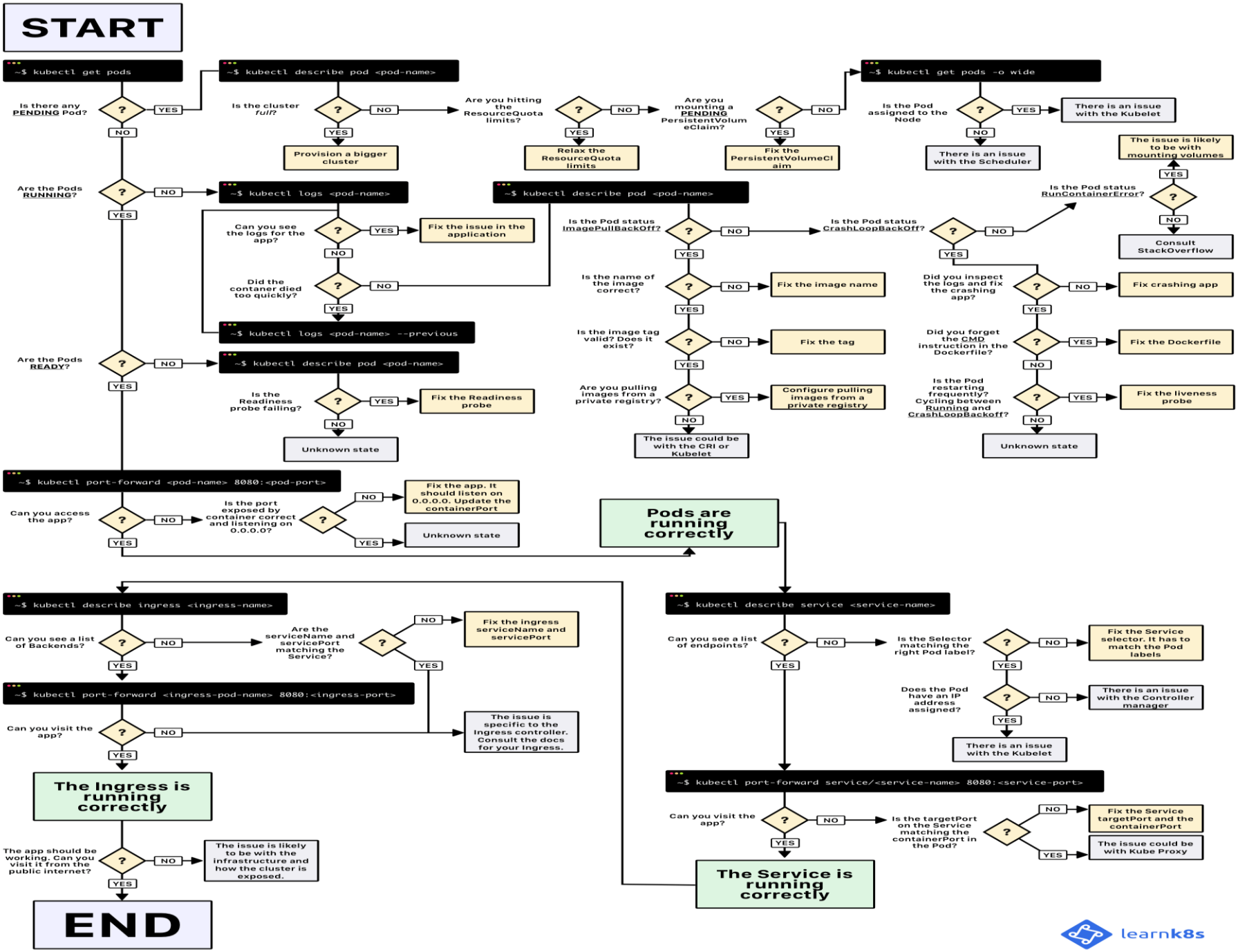
Tenga en cuenta que es posible que deba especificar el espacio de nombres correcto para su controlador Ingress --namespace <name>.

Troubleshooting Kubernetes Deployments

Resumen

La resolución de problemas en los despliegues Kubernetes puede ser una tarea desalentadora si no sabe por dónde empezar.

Siempre debe recordar abordar el problema de abajo hacia arriba: comience con los Pods y suba la pila comprobando la conexión con el Servicio e Ingress.


START
~$ kubectl get pods
Is there any PENDING Pod?
YES
NO
~$ kubectl describe pod <pod-name>
Is the cluster full?
NO
YES
Provision a bigger cluster
Are you hitting the ResourceQuota limits?
NO
YES
Relax the ResourceQuota limits
Are you mounting a PENDING PersistentVolumeClaim?
NO
YES
Fix the PersistentVolumeClaim
~$ kubectl get pods -o wide
Is the Pod assigned to the Node
YES
NO
There is an issue with the Kubelet
There is an issue with the Scheduler
Are the Pods RUNNING?
NO
YES
~$ kubectl logs <pod-name>
Can you see the logs for the app?
YES
NO
Fix the issue in the application
Did the contaner died too quickly?
NO
YES
~$ kubectl logs <pod-name> --previous
~$ kubectl describe pod <pod-name>
Is the Pod status ImagePullBackOff?
NO
YES
Is the Pod status CrashLoopBackOff?
NO
YES
Is the Pod status RunContainerError?
YES
NO
The issue is likely to be with mounting volumes
Consult StackOverflow
Is the name of the image correct?
NO
YES
Fix the image name
Is the image tag valid? Does it exist?
NO
YES
Fix the tag
Are you pulling images from a private registry?
YES
NO
Configure pulling images from a private registry
The issue could be with the CRI or Kubelet
Did you inspect the logs and fix the crashing app?
NO
YES
Fix crashing app
Did you forget the CMD instruction in the Dockerfile?
YES
NO
Fix the Dockerfile
Is the Pod restarting frequently? Cycling between Running and CrashLoopBackoff?
YES
NO
Fix the liveness probe
Unknown state
Are the Pods READY?
NO
YES
~$ kubectl describe pod <pod-name>
Is the Readiness probe failing?
YES
NO
Fix the Readiness probe
Unknown state
~$ kubectl port-forward <pod-name> 8080:<pod-port>
Can you access the app?
NO
YES
Is the port exposed by container correct and listening on 0.0.0.0?
NO
YES
Fix the app. It should listen on 0.0.0.0. Update the containerPort
Unknown state
Pods are running correctly
~$ kubectl describe ingress <ingress-name>
Can you see a list of Backends?
NO
YES
Are the serviceName and servicePort matching the Service?
NO
YES
Fix the ingress serviceName and servicePort
~$ kubectl port-forward <ingress-pod-name> 8080:<ingress-port>
Can you visit the app?
NO
YES
The issue is specific to the Ingress controller. Consult the docs for your Ingress.
The Ingress is running correctly
The app should be working. Can you visit it from the public internet?
NO
YES
The issue is likely to be with the infrastructure and how the cluster is exposed.
END
~$ kubectl describe service <service-name>
Can you see a list of endpoints?
NO
YES
Is the Selector matching the right Pod label?
NO
YES
Fix the Service selector. It has to match the Pod labels
Does the Pod have an IP address assigned?
NO
YES
There is an issue with the Controller manager
There is an issue with the Kubelet
~$ kubectl port-forward service/<service-name> 8080:<service-port>
Can you visit the app?
NO
YES
Is the targetPort on the Service matching the containerPort in the Pod?
NO
YES
Fix the Service targetPort and the containerPort
The issue could be with Kube Proxy
The Service is running correctly
learnk8s

GRACIAS


THANK
YOU
GRACIAS
ARIGATO
SHUKURIA
BOLZÏN
MERCI
BÏYAN
SHUKRIA
DANKSCHEEN
TASHAKKUR ATU
YAQHANYELAY
SUKSAMA
EKHMET
MEHRBANI
GRAZIE
MAAKE
KOMAPSUMNIDA
JUSPAXAR
GOZAIMASHITA
EFCHARISTO
PALDIES
TINGKI
SPASSIBO
SNACHALHUYA
NUHUN
CHALTU
WABEEJA
MAITEKA
YUSPAGARÀTAM
HUI
GUI
HATUR
EKOJU
SIKOMO
MAKETAI
MINMONCHAR
SPASIBO
DENKAUJA
NENACHALHYA
UNALCHÉESH
MERSI
ATTO
ANHA
DHANYABAAD
LAH
SANCO
MERASTAWHY
GAEJTHO
TAVTAPUCH
MEDAWAGSE
AGUYJE
FAKAAUE
BAIKA