



**Juan Alberto García Fernández**  
Systems Engineer / Trainer



**Alberto García**  
**alberto@rootdesdezero.com**

---

# Kubernetes Despliegue de Aplicaciones

- Introducción a Kubernetes
- Conceptos básicos
- Despliegue de Aplicaciones
- Demo
  - Estrategia de despliegues
  - Helm

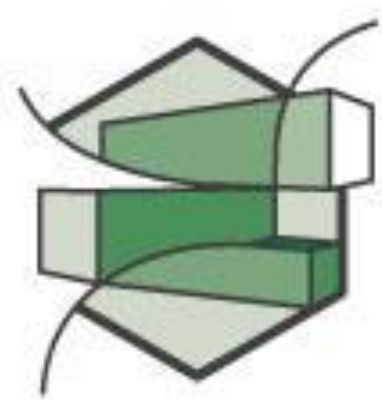


# Introducción

- Docker es un proyecto open source que ha revolucionado la manera de desarrollar software gracias a la sencillez con la que permite gestionar contenedores. Los contenedores LXC (Linux Containers) son un concepto relativamente antiguo y utilizado desde hace tiempo por grandes empresas como Amazon o Google, pero cuyo gestión era complicada. Sin embargo, Docker define APIs y herramientas de línea de comandos que hacen casi trivial la creación, distribución y ejecución de contenedores. De ahí que el lema de Docker sea: “Build, Ship and Run. Any application, Anywhere” y se haya convertido en una herramienta fundamental tanto para desarrolladores como para administradores de sistemas.

# Introducción

- Podríamos definir un contenedor Docker como una máquina virtual ligera, que corre sobre un sistema operativo Linux pero con su propio sistema de ficheros, su propio espacio de usuarios y procesos, sus propias interfaces de red... por lo que se dice que son sistemas aislados. Las características principales de Docker son su portabilidad, su inmutabilidad y su ligereza:



# Introducción

- **Portabilidad**

Un contenedor Docker es ejecutado por lo que se denomina el Docker Engine, un demonio que es fácilmente instalable en prácticamente todas las distribuciones Linux. Un contenedor ejecuta una imagen de docker, que es una representación del sistema de ficheros y otros metadatos que el contenedor va a utilizar para su ejecución. Una vez que hemos generado una imagen de Docker, ya sea en nuestro ordenador o vía una herramienta externa, esta imagen podrá ser ejecutada por cualquier Docker Engine, independientemente del sistema operativo y la infraestructura que haya por debajo.

# Introducción

- **Inmutabilidad**

Una aplicación la componen tanto el código fuente como las librerías del sistema operativo y del lenguaje de programación necesarias para la ejecución de dicho código. Estas dependencias dependen a su vez del sistema operativo donde nuestro código va a ser ejecutado, y por esto mismo ocurre muchas veces aquello de que “no sé, en mi máquina funciona”. Sin embargo, el proceso de instalación de dependencias en Docker no depende del sistema operativo, si no que este proceso se realiza cuando se genera una imagen de docker. Es decir, una imagen de docker (también llamada repositorio por su parecido con los repositorios de git) contiene tanto el código de la aplicación como las dependencias que necesita para su ejecución. Una imagen se genera una vez y puede ser ejecutada las veces que sean necesarias, y siempre ejecutará con las misma versión del código fuente y sus dependencias, por lo que se dice que es inmutable. Si unimos inmutabilidad con el hecho de que Docker es portable, decimos que Docker es una herramienta fiable, ya que una vez generada una imagen, ésta se comporta de la misma manera independientemente del sistema operativo y de la infraestructura donde se esté ejecutando.

# Introducción

- **Ligereza**

Los contenedores Docker corriendo en la misma máquina comparten entre ellos el sistema operativo, pero cada contenedor es un proceso independiente con su propio sistema de ficheros y su propio espacio de procesos y usuarios (para este fin Docker utiliza cgroups y namespaces, recursos de aislamiento basados en el kernel de Linux). Esto hace que la ejecución de contenedores sea mucho más ligera que otros mecanismos de virtualización. Comparemos por ejemplo con otra tecnología muy utilizada como es Virtualbox. Virtualbox permite del orden de 4 ó 5 máquinas virtuales en un ordenador convencional, mientras que en el mismo ordenador podremos correr cientos de containers sin mayor problema, además de que su gestión es mucho más sencilla.

# 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 contenedores**

# 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.

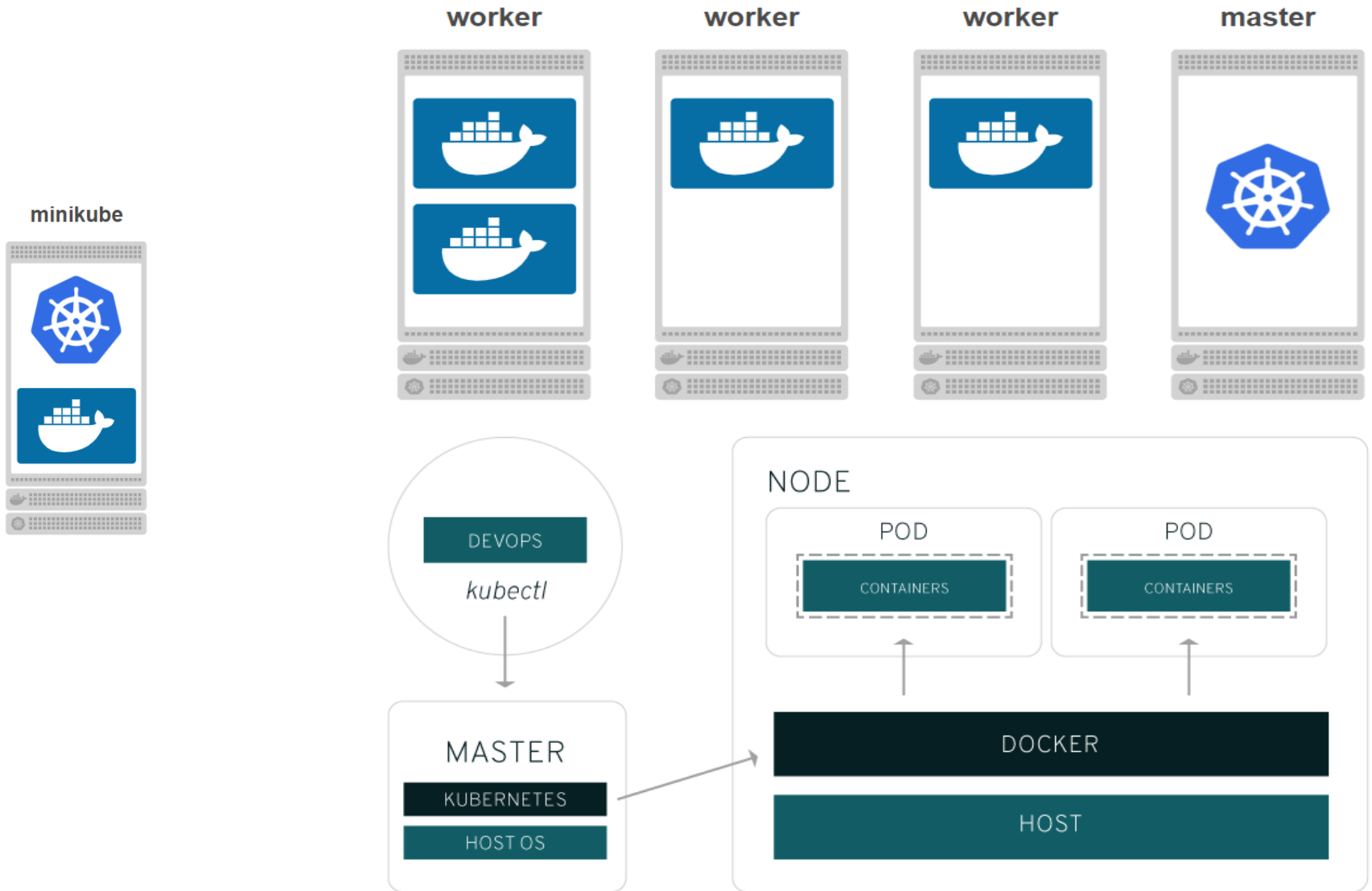
De hecho, Kubernetes ha conseguido hacer realidad el concepto de multi-cloud. Si empaquetamos nuestra aplicación para correr en Kubernetes, nos va a ser relativamente sencillo lanzar nuestra aplicación en un Kubernetes corriendo en AWS, o en Azure, o en Google Cloud, o en un clúster *on-premises*.

Esta característica es de vital importancia, sobre todo para las empresas que hace software enterprise que ejecuta en la infraestructura del cliente





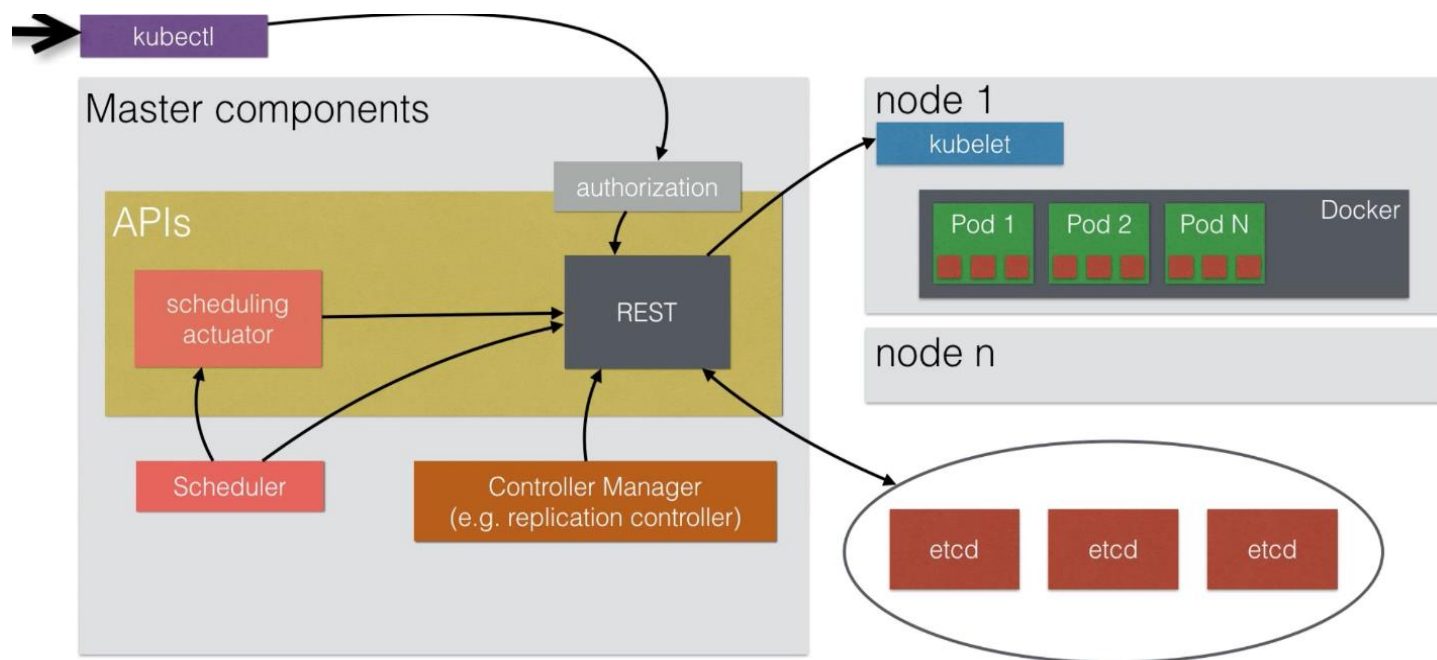
# Arquitectura Básica Kubernetes



# 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.

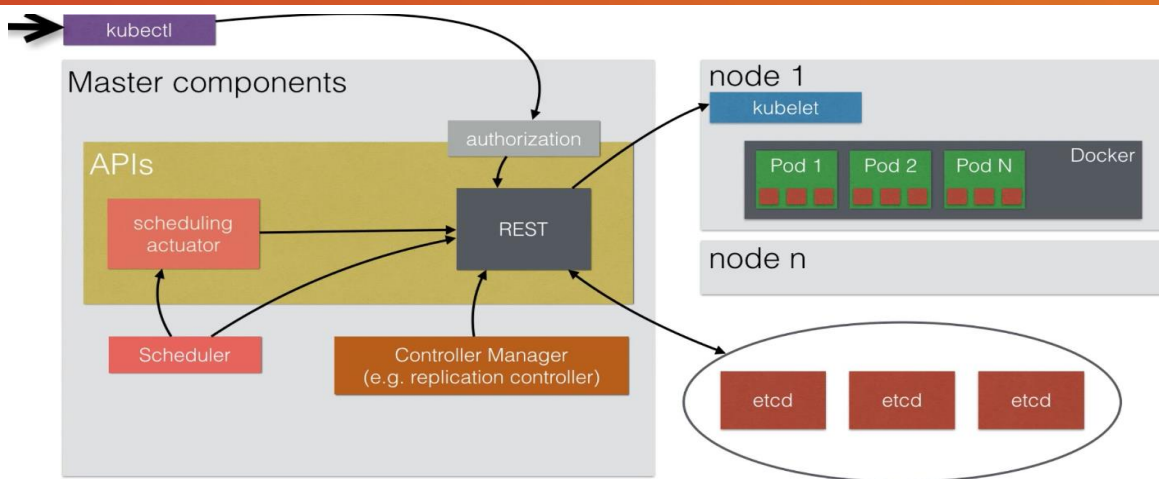
kubectl envía una solicitud de API con el contenido del archivo YAML a la API de Kubernetes.

La API almacena el estado del recurso en una base de datos.

**Etc****d**: Es el servicio de disponibilidad de cluster desarrollado por CoreOS y utilizado también 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 Kubernetes se 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 front del estado compartido del clúster a través del cual interactúan todos los demás componentes.

# Arquitectura Básica Kubernetes

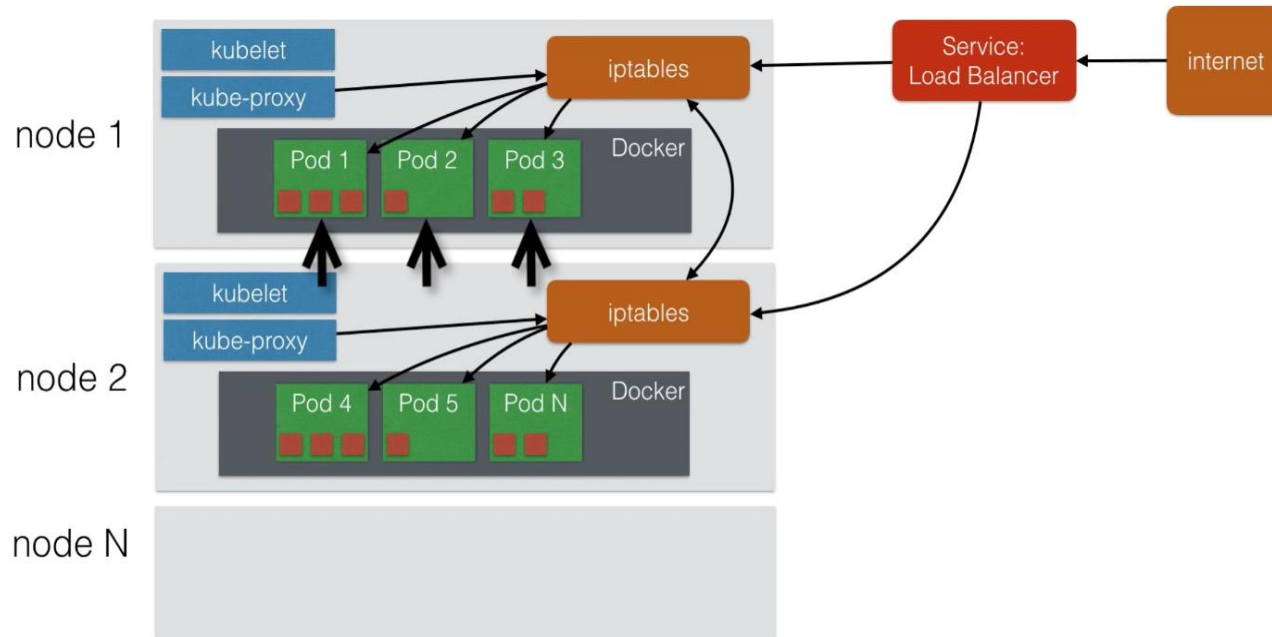


**En el master node** la lógica de la capa de control se fundamenta en **etcd**, una base de datos distribuida que nos asegura la consistencia de los datos usando el protocolo RAFT. Sobre **etcd** tenemos la capa REST, que es a través de la que pasan todos los accesos a la base de datos. Esa capa REST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

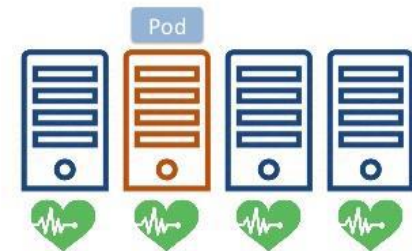
Es un servicio usado para manejar el proceso de replicación de las tareas definidas. Los detalles de estas operaciones son escritos en el etcd, donde el controllermanager observa los cambios y cuando un cambio es detectado, el controllermanager lee la nueva información y ejecuta el proceso de replicación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

## Scheduler

- Elige el lugar y levanta el Pod dentro de los nodos.
- El mejor lugar es elegido en base a los requerimientos del Pod.

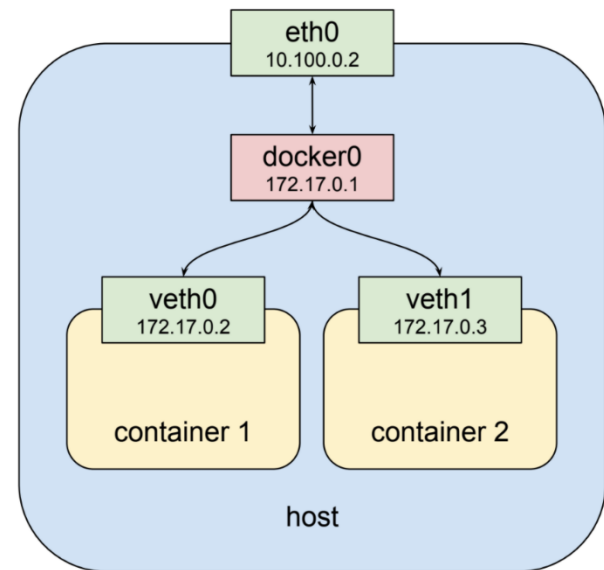


# Kubernetes Arquitectura

<https://kubernetes.io/docs/concepts/cluster-administration/networking/>

Con Kubernetes existen varios sistemas open-source que nos pueden hacer la vida más fácil cuando mantengamos el networking en un cluster kubernetes:

- Weave
- Flannel
- Calico



# Kubernetes Arquitectura Weave

## Weave

Weave es independiente de la topología de red que utilices. Esta compuesto de varios componentes que se instalan en cada host:

- **weave:** El daemon que se encarga de todo la gestión de la red a nivel de cada host.
- **weave-dns:** Herramienta para crear dominios que pueden ser accedidos desde cualquier contenedor.
- **weave-proxy:** Crea un proxy que engloba y substituye el proxy de Docker para la comunicación entre Docker hosts.
- **weave-scope:** Herramienta para visualizar la topología que forman todos los contendores.



# Kubernetes Arquitectura Weave

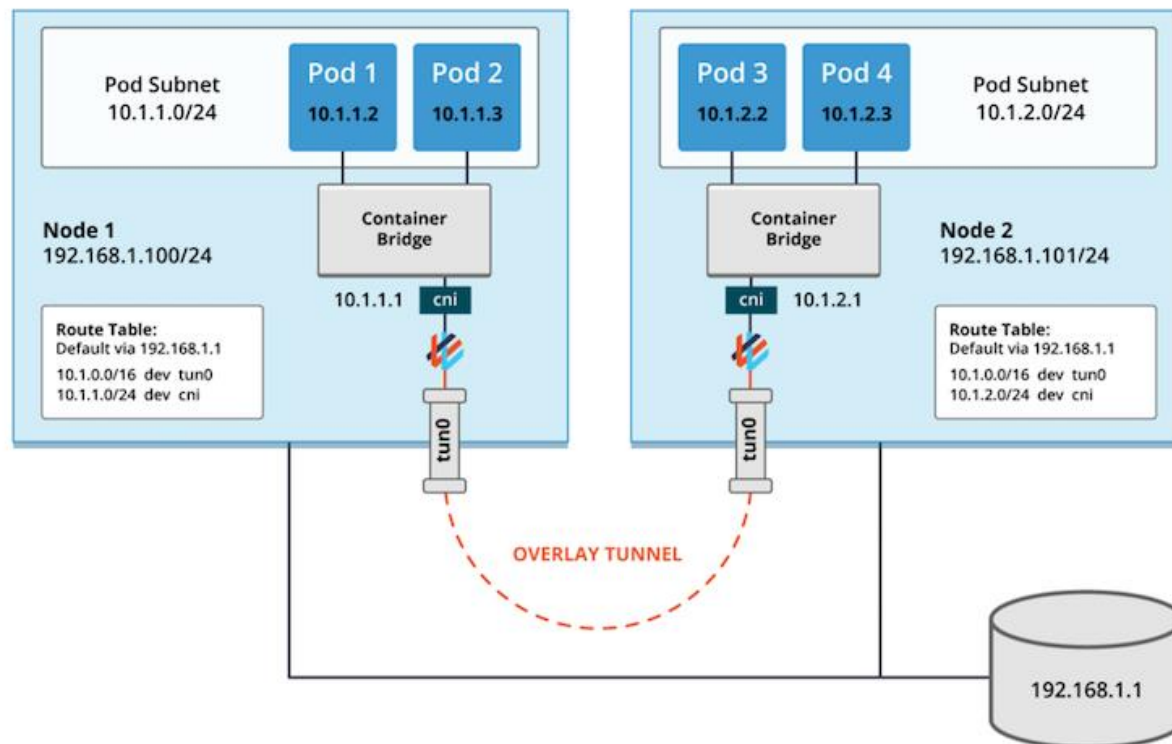
## **Algunos de los aspectos a destacar de Weave:**

- De los primeros en ofrecer soporte para la comunicación entre hosts.
- Se puede utilizar los binarios de Weave para configurar tu cluster de Kubernetes. No es necesario ejecutar los componentes de weave en contenedores.
- Simple, weave routers se retro alimentan de la información de otros weave router en otros hosts. Esto les permite tener conocimiento de los contenedores y hosts con los que pueden comunicarse.
- La información de red está distribuida a través de todos los nodos que componen la Weave network. Esto mejora la tolerancia a fallos.
- Sistema de encriptado incorporado, aunque caro a nivel de performance.
- Es capaz de hacer tunneling incluso a través de cortafuegos.
- Service discovery usando weave DNS.
- Usa NAT multicast y MTUs.
- Simple DNS load balancing (round-robin), usand weave-dns.
- Funciona en Giant Swarm, Kubernetes, Mesos.
- Permite tener una visión de la topología de red y donde estan tus contenedores usando Weave-scope.
- Uso de Gossip protocol para compartir la información de red y la reglas de enrutado.

# Kubernetes Arquitectura Weave

## Desventajas de Weave:

- No es el más rápido de todos los drivers de networking. Aunque ha mejorado su performance recientemente.



# Kubernetes Arquitectura Flannel

## Flannel

Desarrollado por CoreOS y pensado para **Kubernetes** con el concepto de subnets para pods. Al igual que Docker networking utiliza una base de datos clave/valor, Flannel hace uso de etcd para almacenar toda la información de red.

Algunos aspectos a destacar de Flannel son:

- Proporciona la posibilidad de crear una network overlay (L3)
- Una subnet CIDR (enrutamiento entre dominios sin clases) por host (como con Kubernetes) o

Host A: 11.0.47.1/24

o Host B: 11.0.87.1/24

- No hay necesidad de hacer mapeos estáticos de puertos y IPs
- Usa etcd para salvaguardar la información de la network
- Contenedores hablan via direcciones IP
- Uso de IPSEC como protocolo de encapsulación

# Kubernetes Arquitectura Flannel

Ofrece dos tipos de mecanismos para encapsular los paquetes:

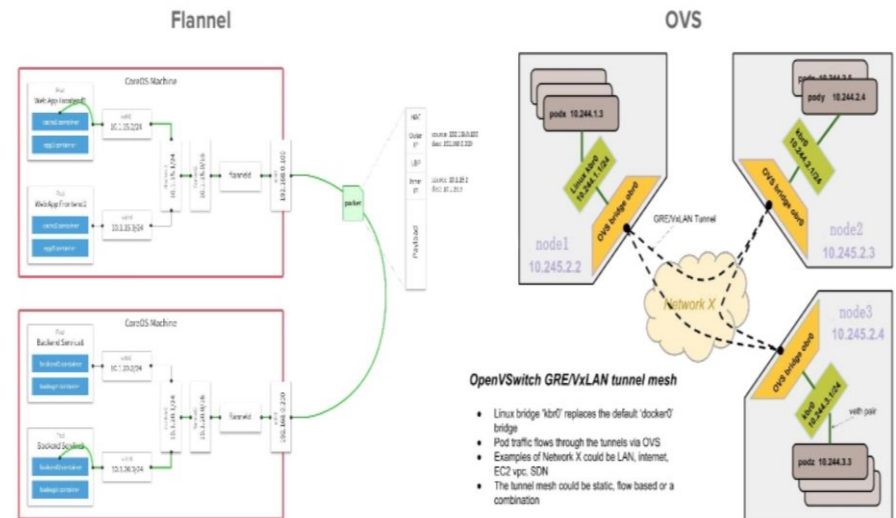
- o UDP
- o VXLAN

• Ofrece drivers para configurar distintos tipos de networks:

VxLAN, UDP, alloc, host-gw, aws-vpc.

**Flannel** es una solución de red de contenedores desarrollada por CoreOS. Flannel asigna una subred a cada host, el contenedor obtiene direcciones IP de esta subred, y estas IPs se pueden enrutar entre hosts y los contenedores se pueden comunicar a través de hosts sin NAT y port mapping.

Inter POD communication: other examples



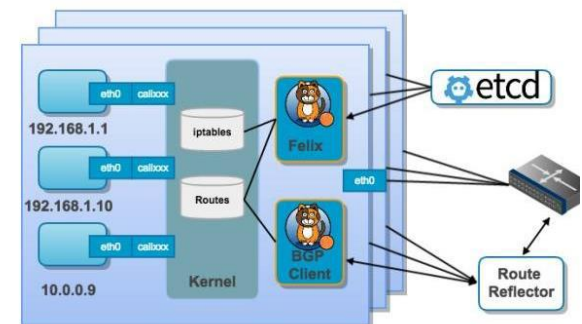
# Kubernetes Arquitectura Calico

**Project Calico** es una implementación de SDN de tres capas, sin usar redes sobrecargadas. Calico se basa en el protocolo BGP y en el propio mecanismo de enrutamiento de Linux, sin requerir de hardware especial.

Calico no usa tecnología NAT o de túnel. Se puede implementar fácilmente en el servidor físico o el entorno contenedor. Al mismo tiempo, viene con los componentes de administración de ACLs basados en Iptables que son muy flexibles, para satisfacer las necesidades de aislamiento de seguridad más complejas.

Todos los contenedores están configurados para usar calico-node para lograr la interoperabilidad de la red y el acceso a Ethernet.

<https://www.projectcalico.org/>



# Kubernetes Arquitectura Calico

## Calico

- **Ofrece** un modelo de networking parecido al de Internet (L2-L3)
- **Permite** definir perfiles ACLs para los contenedores. Incrementa la seguridad entre contenedores ya que no es necesario que los contenedores creen iptables si usan links entre ellos.
- **Usa BGP** para compartir la información de enrutado entre todos los hosts que componen el cluster de Calico.
- **Escala** bastante bien con una gran cantidad de contenedores.
- Uno de los mejores a nivel de Performance por detrás de Flannel.

# Kubernetes Arquitectura Network

Un breve resumen del modelo de red se muestra en la siguiente tabla:

	Calico	Flannel	Weave
Tipo de red	puro de Capa 3	VxLAN o UDP	VxLAN o UDP

## Aislamiento de aplicaciones

	Calico	Flannel	Weave
Aislamiento	Perfil de esquema	CIDR	CIDR

## Soporte de protocolo

	Calico	Flannel	Weave
Protocolo	TCP, UDP, ICMP & ICMPv6	Todos	Todos

## Canal de cifrado

	Calico	Flannel	Weave
Canal de cifrado	No	TLS	NaCl Library

# Kubernetes

## etcd - Almacén clave-valor

**etcd** es un almacén de clave-valor.

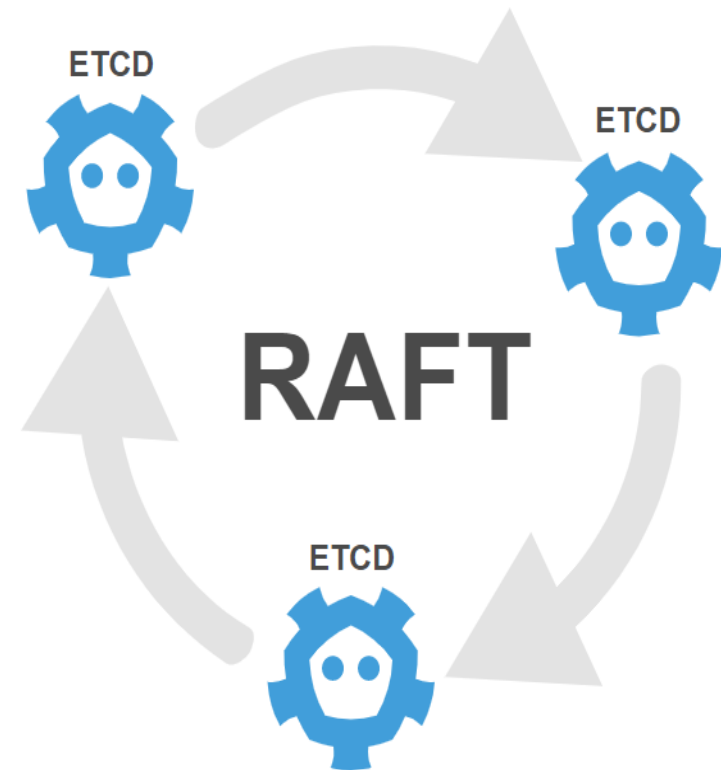
La razón por la que es tan popular y se usa en Kubernetes es que está diseñado para ejecutarse en sistemas distribuidos.

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 (imágenes, volúmenes, etc).

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

## Nodes

Colección de máquinas que son tratadas como una sola unidad lógica por Kubernetes.

- Docker
- Kubernetes Agents (kubelet, proxy)



# Kubernetes Conceptos Básicos

## Pod

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

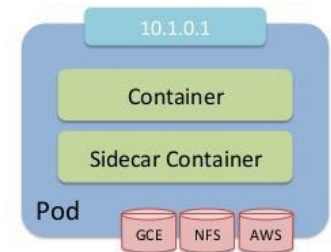
En general, un pod contendrá únicamente un contenedor, aunque no tiene que ser así: si tenemos dos contenedores que actúan de forma conjunta, podemos desplegarlos dentro de un solo pod. Dentro de un pod todos los contenedores se pueden comunicar entre ellos usando localhost, por lo que es una manera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

En este sentido, todos los contenedores dentro de un pod se podría decir que están instaladas en un 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.

## Pods

Mínima unidad lógica desplegable en Kubernetes.

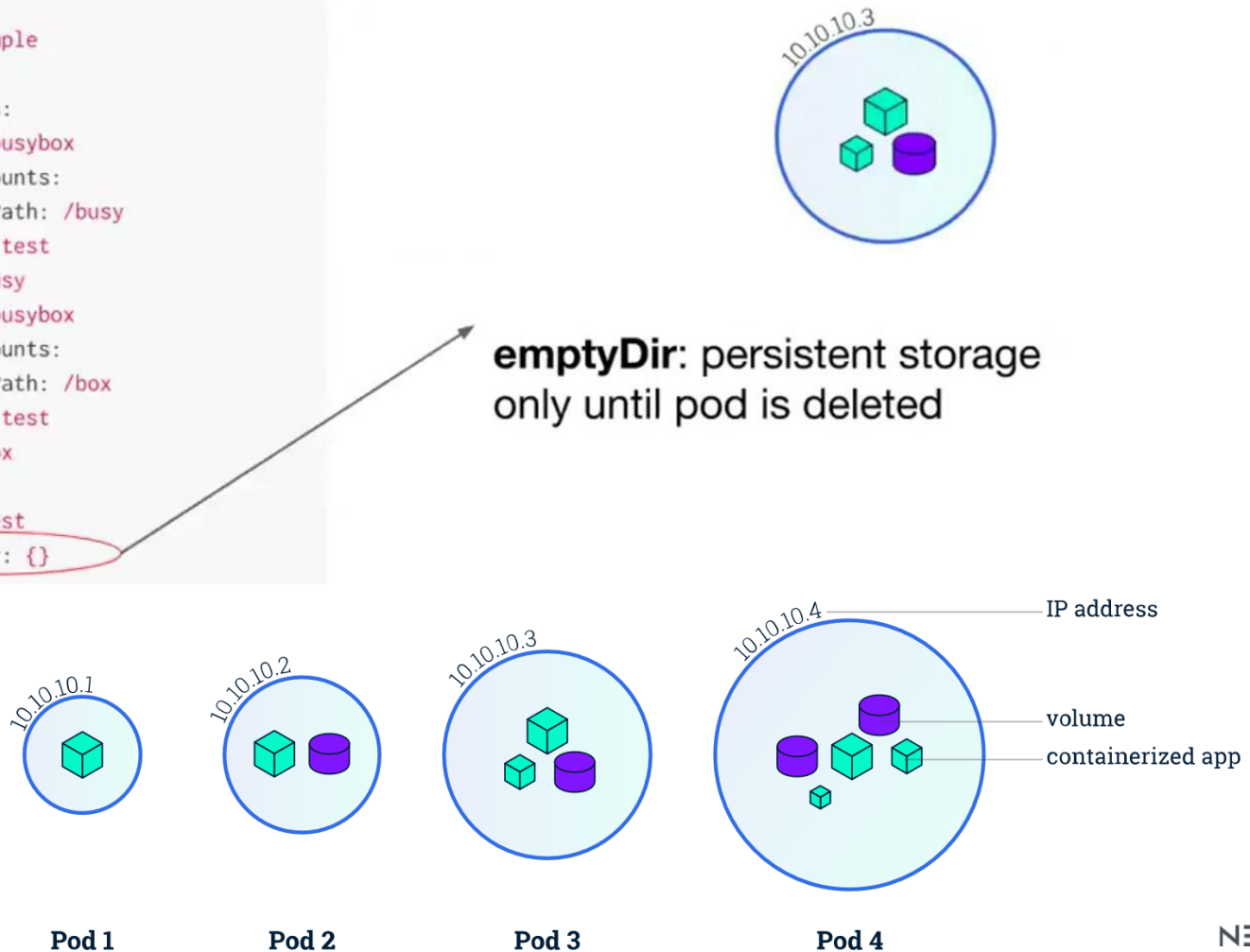
- Contienen un grupo de contenedores co-localizados (usualmente uno) y volúmenes.
- Share Namespace, Ip por Pod, localhost dentro del POD



# Kubernetes Pod

```
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 Controller** se encarga de mantener un determinado número de réplicas del pod en el clúster.

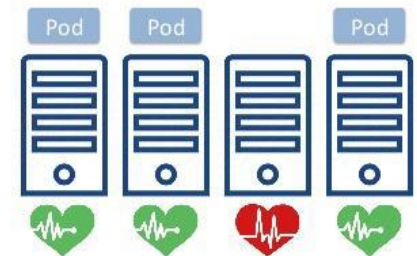
El Replication Controller asegura que un determinado número de copias -réplicas- del pod se encuentran en ejecución en el clúster en todo momento. Por tanto, si alguno de los pods es eliminado, el ReplicationController se encarga de crear un nuevo pod. Para ello, el ReplicationController incluye una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

## Replication Controllers

Maneja un conjunto replicado de Pods.

- Asegura que un número especificado de "Replicas" siempre se estén ejecutando.
- Self Healing.



# 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.

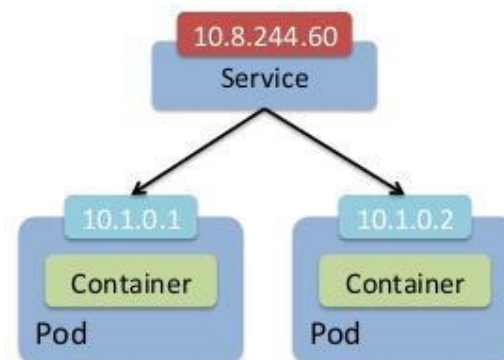
Un ejemplo de esto puede ser un replicationcontroller llamado WordPress(nos realiza un deploy de 3 containers/pod), el cual es asociado a un servicio con el mismo nombre y se conecta al replicationcontrol a través de un Label/Etiqueta.

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.



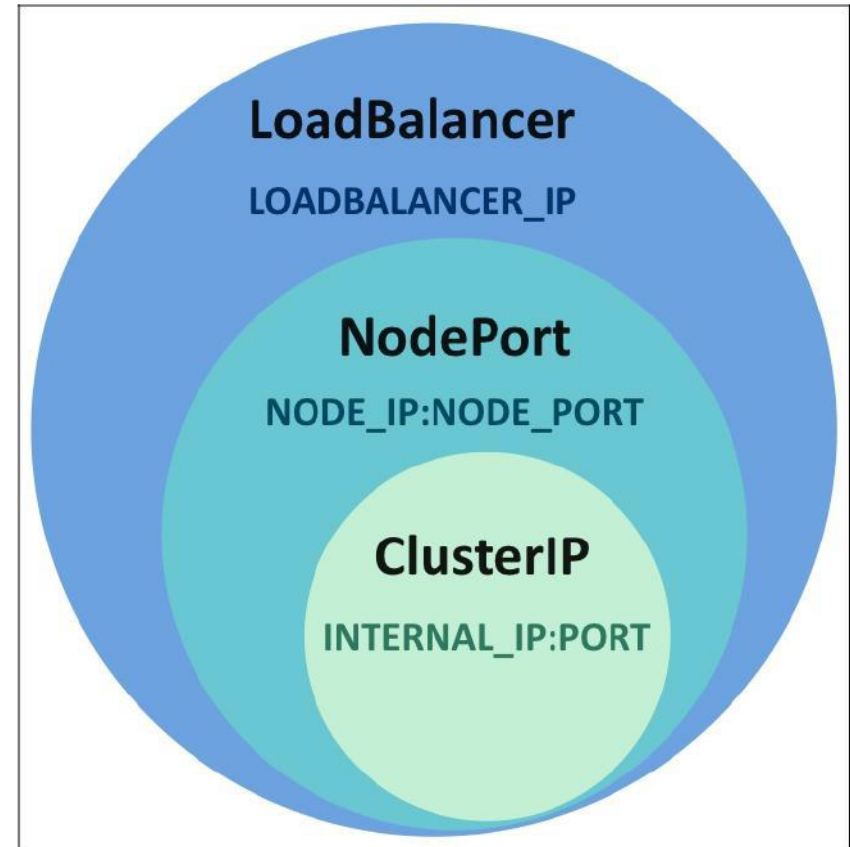
# Kubernetes

## 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.

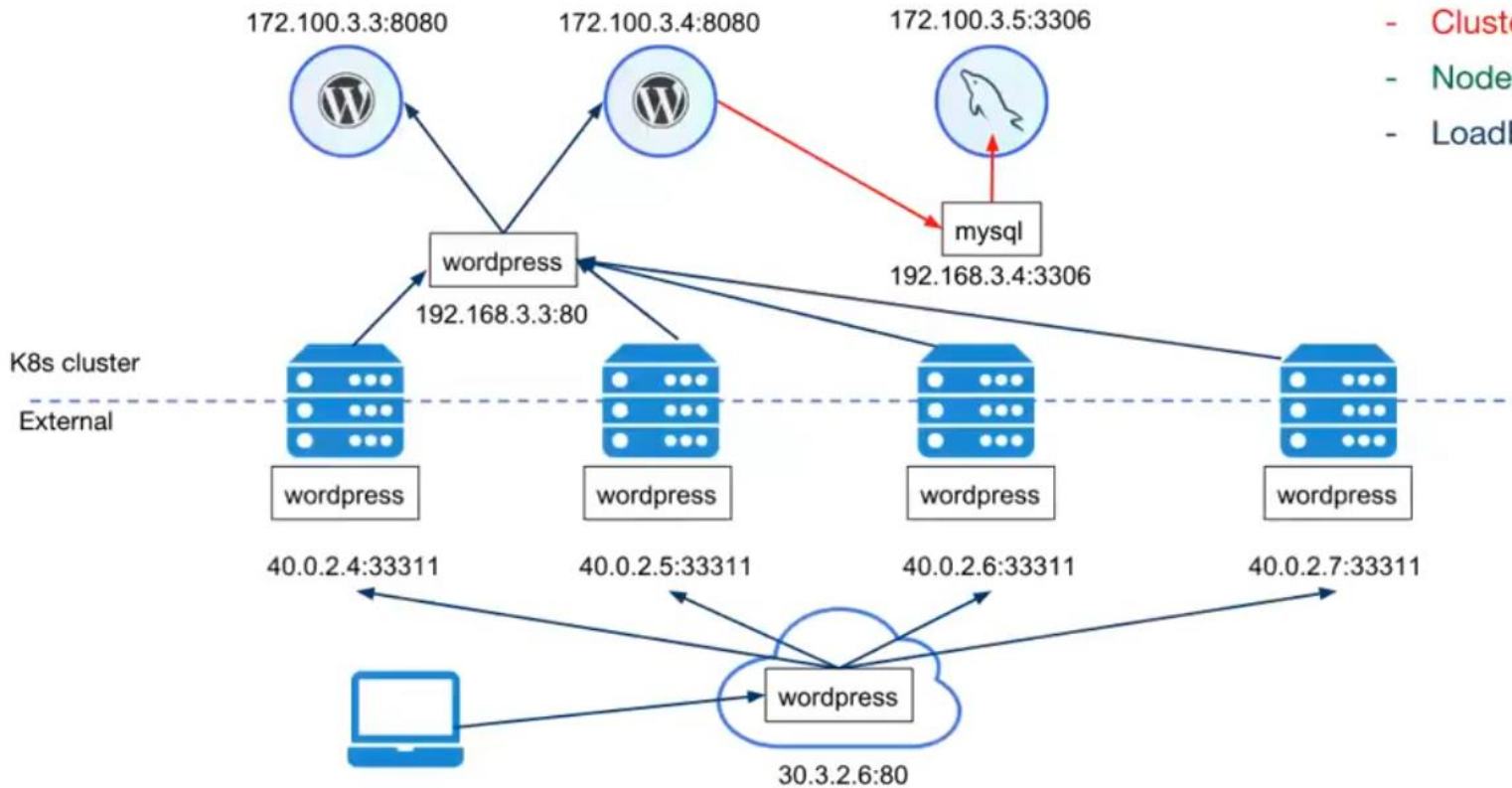


# Kubernetes Services

## Basic Objects: Services

Types:

- ClusterIP
- NodePort
- LoadBalancer



# 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 eliminando 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.

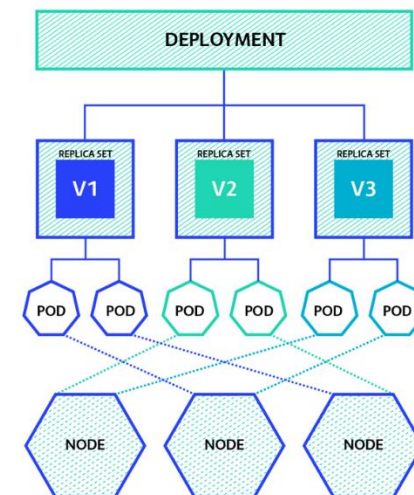
**Filtrando pods, por label:**

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

# Kubernetes kubectl

## ¿Qué es kubectl?

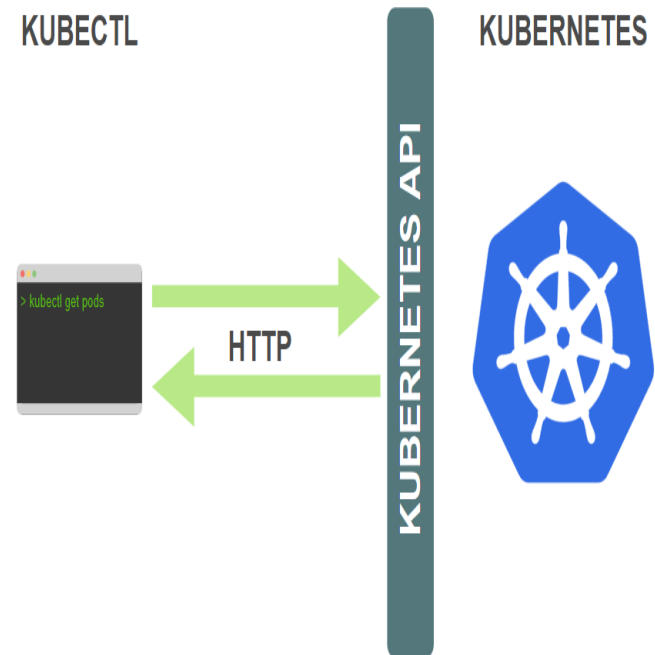
Desde el punto de vista de un usuario, kubectl es un comando para controlar Kubernetes. Le permite realizar todas las operaciones posibles de Kubernetes.

Desde un punto de vista técnico, kubectl es un cliente para la **API de Kubernetes**.

La API de Kubernetes es una **API REST de HTTP**. Esta API es la **interfaz de usuario** real de Kubernetes. Kubernetes está totalmente controlado a través de esta API.

Esto significa que cada operación de Kubernetes está expuesta como un punto final de API y puede ejecutarse mediante una solicitud HTTP a este punto final.

En consecuencia, el trabajo principal de kubectl es realizar solicitudes HTTP a la API de Kubernetes:



# Kubernetes kubectl

## Kubernetes Cheat Sheet

### What is Kubernetes?

Kubernetes is a platform for managing containerized workloads. Kubernetes orchestrates computing, networking and storage to provide a seamless portability across infrastructure providers.

### Viewing Resource Information

#### Nodes

```
$ kubectl get no
$ kubectl get no -o wide
$ kubectl describe no
$ kubectl get no -o yaml
$ kubectl get node --selector=[label_name]
$ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="ExternalIP")].address}'
$ kubectl top node [node_name]
```

#### Pods

```
$ kubectl get po
$ kubectl get po -o wide
$ kubectl describe po
$ kubectl get po --show-labels
$ kubectl get po -l app=nginx
$ kubectl get po -o yaml
$ kubectl get pod [pod_name] -o yaml --export
$ kubectl get pod [pod_name] -o yaml --export > nameoffile.yaml
$ kubectl get pods --field-selector status.phase=Running
```

#### Namespaces

```
$ kubectl get ns
$ kubectl get ns -o yaml
$ kubectl describe ns
```

#### Deployments

```
$ kubectl get deploy
$ kubectl describe deploy
$ kubectl get deploy -o wide
$ kubectl get deploy -o yaml
```

#### Services

```
$ kubectl get svc
$ kubectl describe svc
$ kubectl get svc -o wide
$ kubectl get svc -o yaml
$ kubectl get svc --show-labels
```

#### DaemonSets

```
$ kubectl get ds
$ kubectl get ds --all-namespaces
$ kubectl describe ds [daemonset_name] -n [namespace_name]
$ kubectl get ds [ds_name] -n [ns_name] -o yaml
```

#### Events

```
$ kubectl get events
$ kubectl get events -n kube-system
$ kubectl get events -w
```

#### Logs

```
$ kubectl logs [pod_name]
$ kubectl logs --since=1h [pod_name]
$ kubectl logs --tail=20 [pod_name]
$ kubectl logs -f -c [container_name] [pod_name]
$ kubectl logs [pod_name] > pod.log
```

#### Service Accounts

```
$ kubectl get sa
$ kubectl get sa -o yaml
$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ kubectl replace serviceaccount default -f ./sa.yaml
```

#### ReplicaSets

```
$ kubectl get rs
$ kubectl describe rs
$ kubectl get rs -o wide
$ kubectl get rs -o yaml
```

#### Roles

```
$ kubectl get roles --all-namespaces
$ kubectl get roles --all-namespaces -o yaml
```

#### Secrets

```
$ kubectl get secrets
$ kubectl get secrets --all-namespaces
$ kubectl get secrets -o yaml
```

#### ConfigMaps

```
$ kubectl get cm
$ kubectl get cm --all-namespaces
$ kubectl get cm --all-namespaces -o yaml
```

#### Ingress

```
$ kubectl get ing
$ kubectl get ing --all-namespaces
```

#### PersistentVolume

```
$ kubectl get pv
$ kubectl describe pv
```

#### PersistentVolumeClaim

```
$ kubectl get pvc
$ kubectl describe pvc
```



# Kubernetes kubectl

## Kubernetes Cheat Sheet

page 2

### Viewing Resource Information (cont.)

#### StorageClass

```
$ kubectl get sc
$ kubectl get sc -o yaml
```

#### Multiple Resources

```
$ kubectl get svc, po
$ kubectl get deploy, no
$ kubectl get all
$ kubectl get all --all-namespaces
```

### Changing Resource Attributes

#### Taint

```
$ kubectl taint [node_name] [taint_name]
```

#### Labels

```
$ kubectl label [node_name] disktype=ssd
$ kubectl label [pod_name] env=prod
```

#### Cordon/Uncordon

```
$ kubectl cordon [node_name]
$ kubectl uncordon [node_name]
```

#### Drain

```
$ kubectl drain [node_name]
```

#### Nodes/Pods

```
$ kubectl delete node [node_name]
$ kubectl delete pod [pod_name]
$ kubectl edit node [node_name]
$ kubectl edit pod [pod_name]
```

### Deployments/Namespace

```
$ kubectl edit deploy [deploy_name]
$ kubectl delete deploy [deploy_name]
$ kubectl expose deploy [deploy_name]
  --port=80 --type=NodePort
$ kubectl scale deploy [deploy_name]
  --replicas=5
$ kubectl delete ns
$ kubectl edit ns [ns_name]
```

#### Services

```
$ kubectl edit svc [svc_name]
$ kubectl delete svc [svc_name]
```

#### DaemonSets

```
$ kubectl edit ds [ds_name] -n kube-system
$ kubectl delete ds [ds_name]
```

#### Service Accounts

```
$ kubectl edit sa [sa_name]
$ kubectl delete sa [sa_name]
```

#### Annotate

```
$ kubectl annotate po [pod_name]
  [annotation]
$ kubectl annotate no [node_name]
```

### Adding Resources

#### Creating a Pod

```
$ kubectl create -f [name_of_file]
$ kubectl apply -f [name_of_file]
$ kubectl run [pod_name] --image=nginx
  --restart=Never
$ kubectl run [pod_name]
  --generator=run-pod/v1 --image=nginx
$ kubectl run [pod_name] --image=nginx
  --restart=Never
```

#### Creating a Service

```
$ kubectl create svc nodeport [svc_name]
  --tcp=8080:80
```

### Creating a Deployment

```
$ kubectl create -f [name_of_file]
$ kubectl apply -f [name_of_file]
$ kubectl create deploy [deploy_name]
  --image=nginx
```

#### Interactive Pod

```
$ kubectl run [pod_name] --image=busybox
  --rm -it --restart=Never -- sh
```

#### Output YAML to a File

```
$ kubectl create deploy [deploy_name]
  --image=nginx --dry-run -o yaml >
  deploy.yaml
$ kubectl get po [pod_name] -o yaml --export
  > pod.yaml
```

#### Getting Help

```
$ kubectl -h
$ kubectl create -h
$ kubectl run -h
$ kubectl explain deploy.spec
```

### Requests

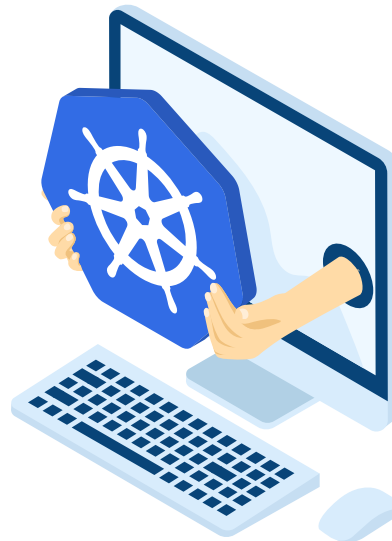
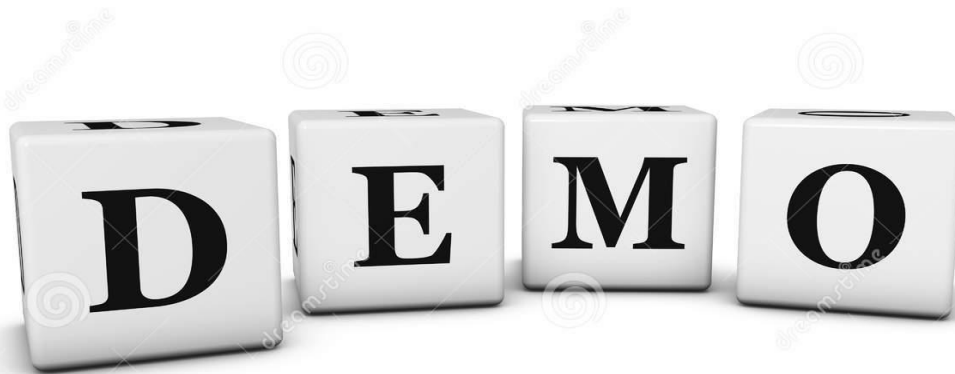
#### API Call

```
$ kubectl get --raw /apis/metrics.k8s.io/
```

#### Cluster Info

```
$ kubectl config
$ kubectl cluster-info
$ kubectl get componentstatuses
```

# 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.

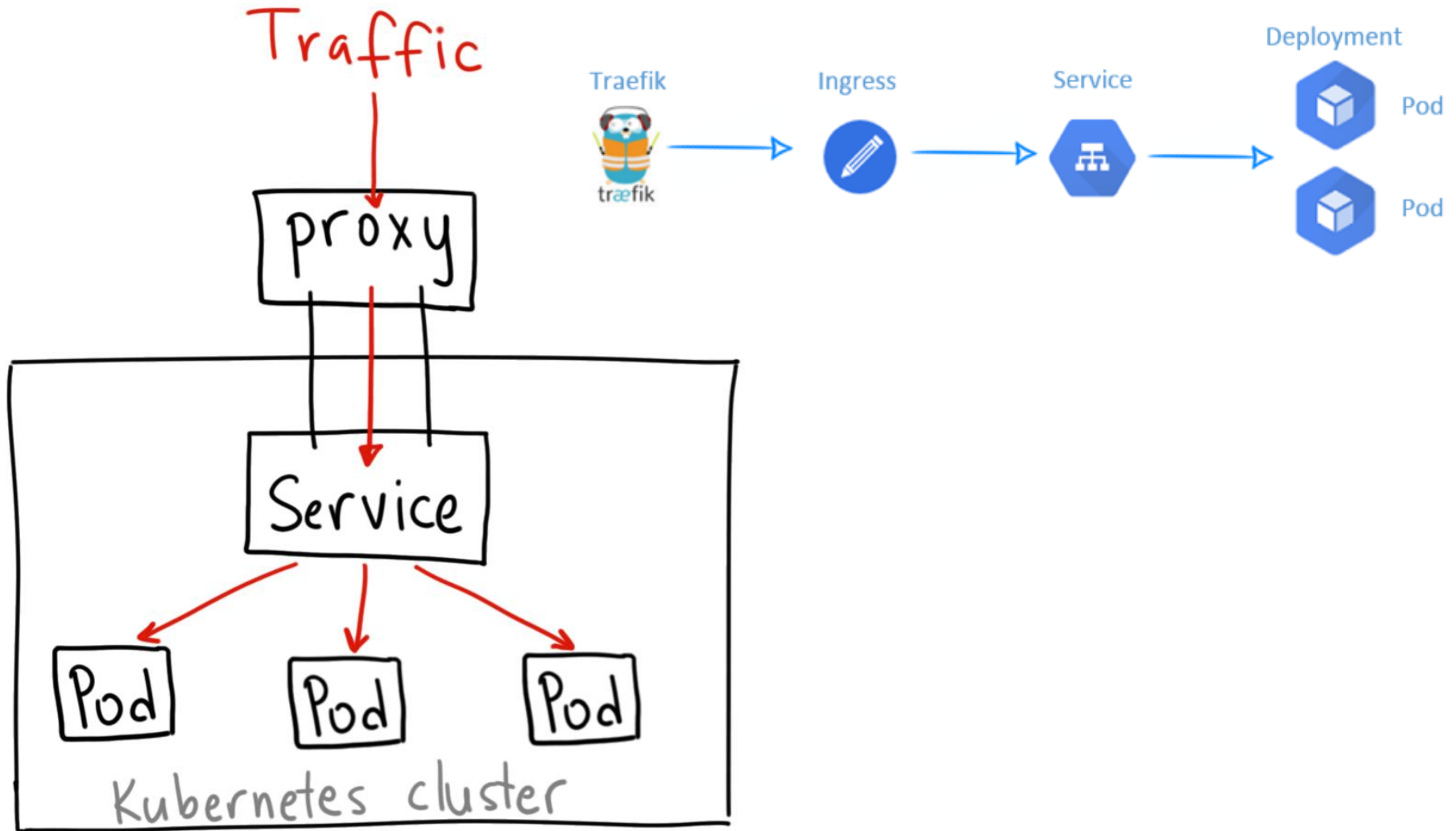
**Un Ingress nos permite definir rutas de entrada a nuestro servicio de una manera programática.** Existen numerosos Ingress Controllers, como el de Nginx, HAProxy o traefik, cada uno de ellos permitiendo distintas configuraciones.

Por ejemplo, con el Nginx Ingress Controller, podemos definir a dónde redirigir una petición en base al dominio solicitado, o al **path** solicitado dentro de esa petición.

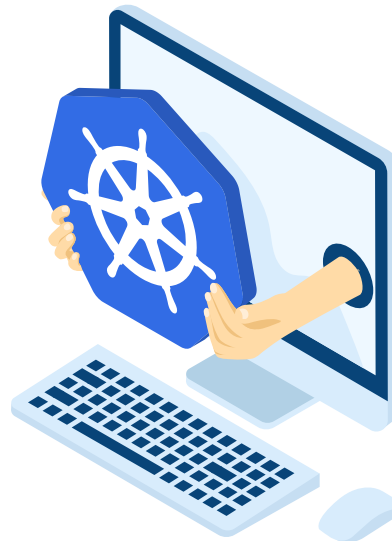
En el laboratorio veremos como redireccionar el tráfico de entrada a mi clúster en función de la cabecera **Host** de la petición entrante.

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

# Kubernetes Ingress



# 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?

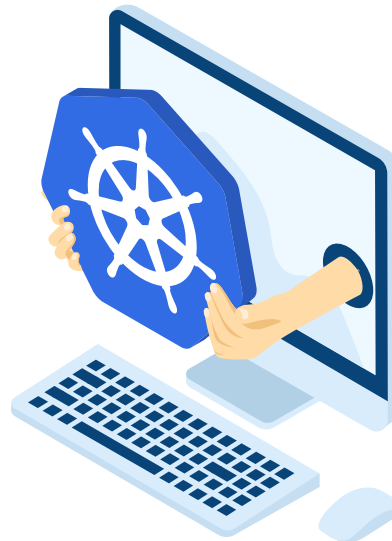
## El flujo es el siguiente:

- 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

DEMO



# Kubernetes Canary Deployments

## Canary deployments

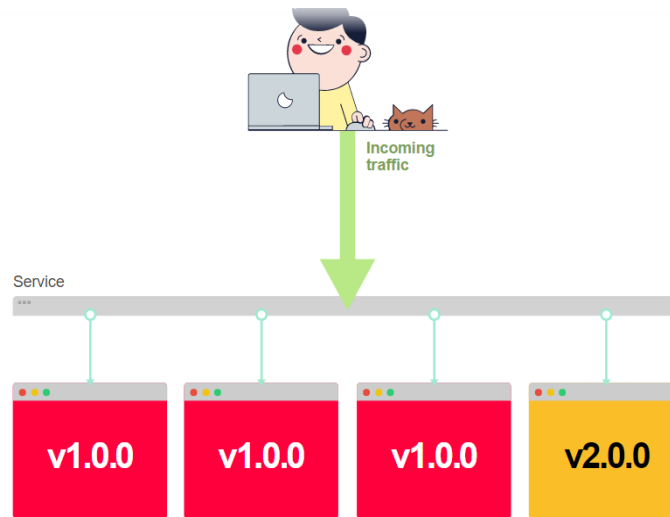
Otra opción para desplegar en producción sin interrumpir el tráfico en vivo es usar una implementación de Canary.

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.

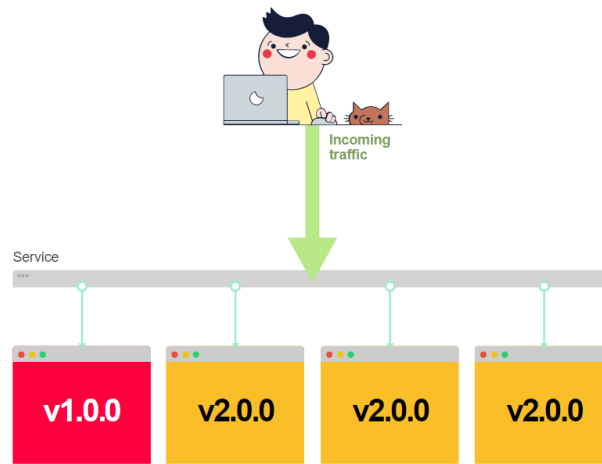
# Kubernetes Canary Deployments



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](#)

```
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](#)

```
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](#)

```
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 deployments: una para la aplicación existente con Pods con una etiqueta `version: 1.0.0` y otra para la nueva aplicación con una etiqueta `version: 2.0.0`

Cuando se inicia, el 100% del tráfico se enruta a la aplicación existente.



El selector de servicio apunta a **version: 1.0.0**.

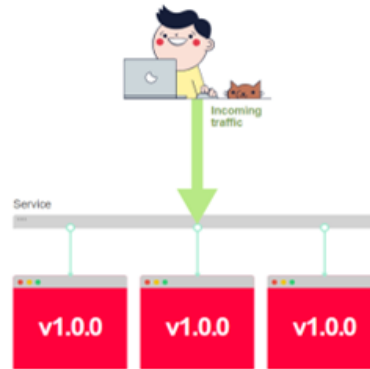
```
service.yaml
kind: Service
apiVersion: v1
metadata:
  name: canary-service
spec:
  selector:
    version: "1.0.0"
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Ahora decide balancear una fracción del tráfico en vivo a la nueva versión.

Con una sola instancia de la aplicación actual y tres instancias de la anterior, el 75% del tráfico se enrutará a la aplicación existente y solo el 25% a la última versión.



# 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

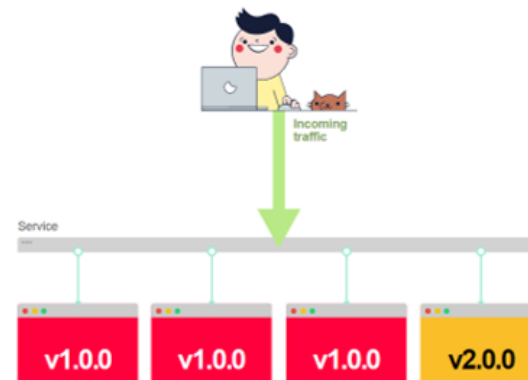
Pero necesita una etiqueta compartida para que el selector de su Servicio dirija el tráfico a ambos.

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

[service.yaml](#)

```
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

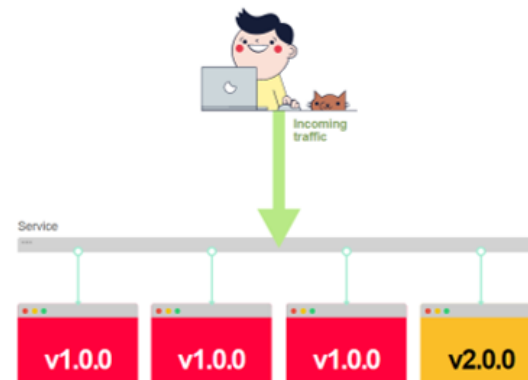
Pero necesita una etiqueta compartida para que el selector de su Servicio dirija el tráfico a ambos.

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

[service.yaml](#)

```
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.

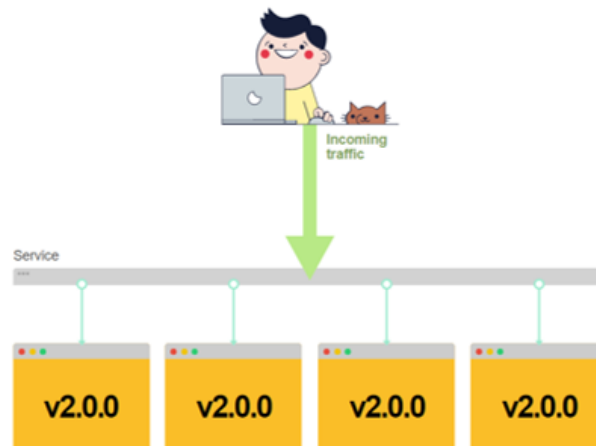


# Kubernetes Canary Deployments

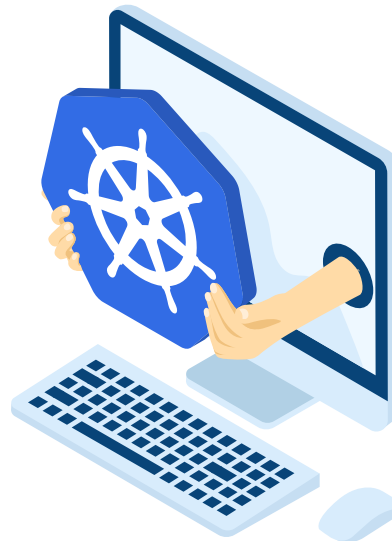
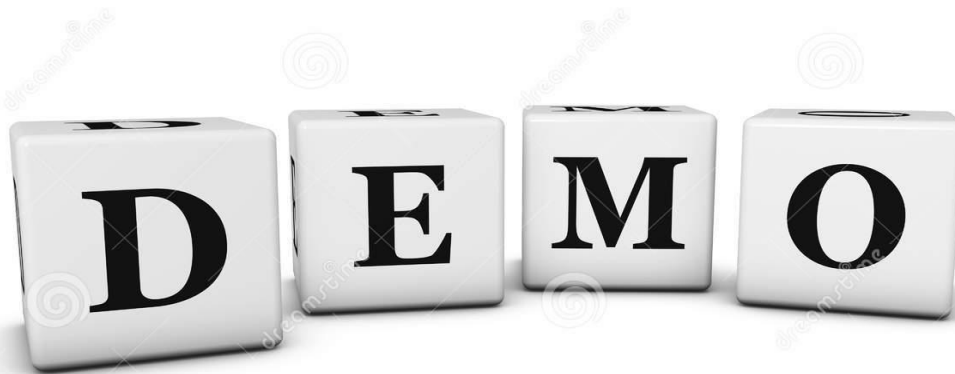
Cuando esté dispuesto a cambiar todo el tráfico, puede editar el Servicio y señalarlo en `version: 2.0.0` en lugar de la etiqueta compartida.

`service.yaml`

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



# 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

```
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

```
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

```
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
```

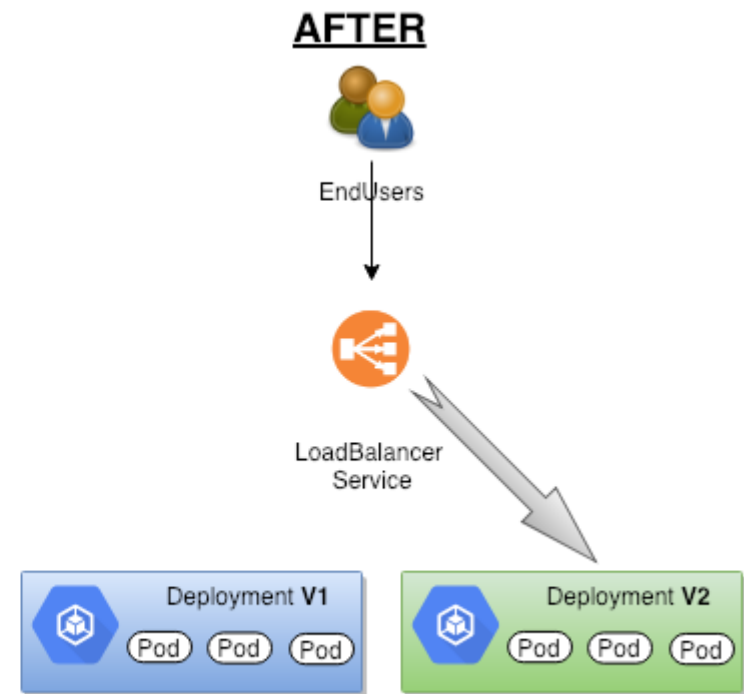
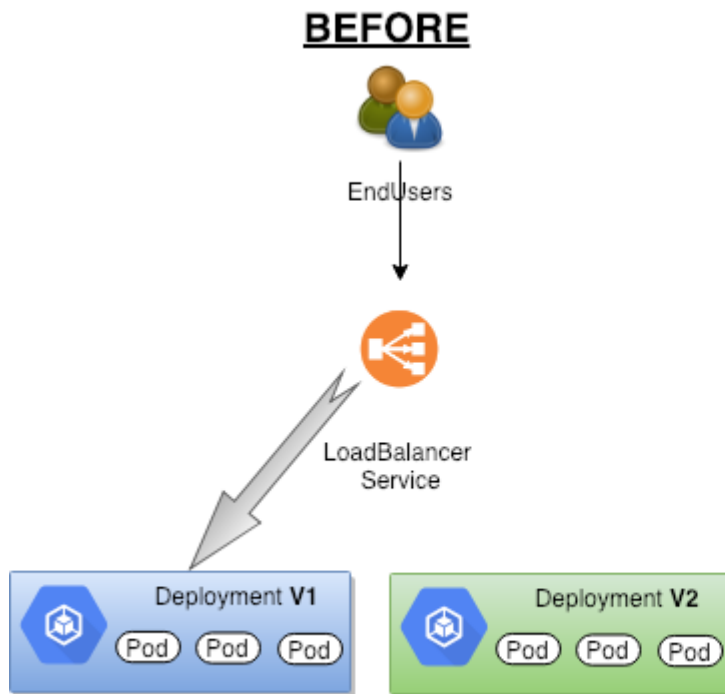


# Kubernetes Blue-Green Deployments

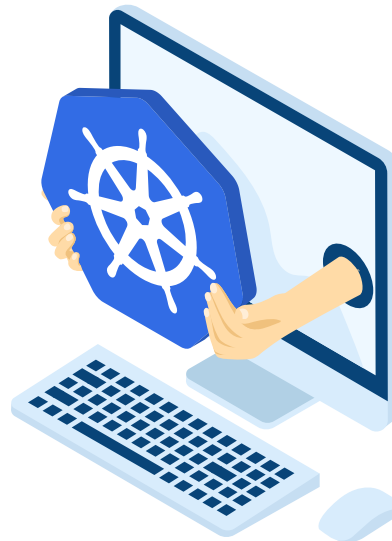
Para enrutar todo el tráfico de la versión **1.0.0** a **2.0.0** y lograr un despliegue azul-verde, abra otra terminal.

Actualice su servicio para apuntar a la versión **2.0.0**:

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



# Kubernetes



# Kubernetes Almacenamiento

<https://kubernetes.io/docs/concepts/storage/>

**En Kubernetes**, un volumen tiene un ciclo de vida explícito, que va en relación con la vida pod. El volumen puede sobrevivir al pod, y la información es preservada incluso con el restart de los contenedores. Por supuesto cuando un pod deja de existir el volumen también. De ahí, que Kubernetes ofrezca múltiple tipos de volúmenes, cada pod puede usar cualquiera de esos tipos y en la cantidad deseada.

Un volumen es un directorio con información la cual es accesible a través de los contenedores del pod. Como ese volumen es creado, que contenido y como se comporta dependerá del tipo de volumen. Para especificar el tipo de volumen utiliza para un pod hay que utilizar el campo `spec.volumes` y para especificar el punto de montaje, `spec.containers.volumeMounts`.

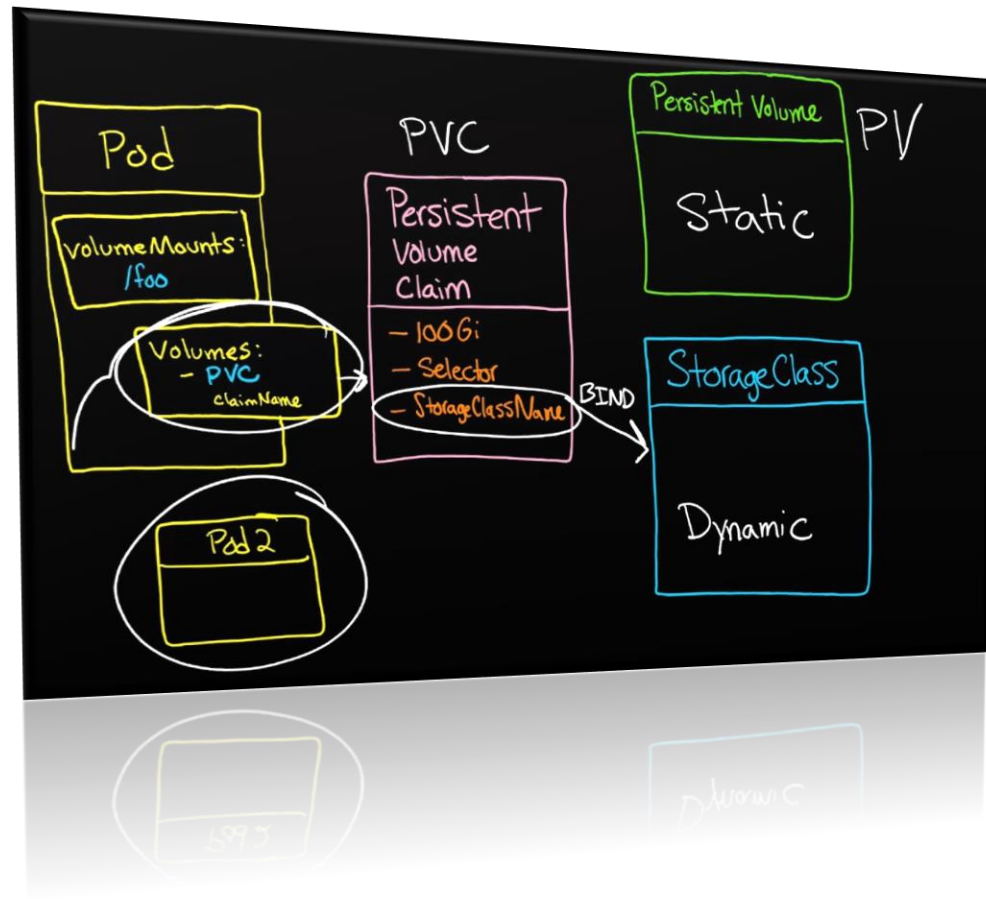
Recuerda que en Docker, un proceso en un contenedor ve el sistema de ficheros como un conjunto de docker layers y volúmenes. La imagen de docker represente sistema de ficheros raíz y los volúmenes son montados a posterior en una ruta determinada. Volúmenes no pueden ser montados en otros volúmenes o tener enlaces a otros volúmenes. Cada contenedor en un pod puede especificar una ruta diferente donde montar el volumen.

# Kubernetes Almacenamiento

## Tipos de volúmenes en Kubernetes:

<https://kubernetes.io/docs/concepts/storage/>

- emptyDir
- hostPath
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- gitRepo
- secret
- downwardAPI
- azureFileVolume
- awsElasticBlockStore
- gcePersistentDisk



# Kubernetes Almacenamiento

## Almacenamiento disponible en Kubernetes: PersistentVolumen

Para añadir almacenamiento a un pod, tenemos que distinguir **dos conceptos**:

- Por ejemplo nuestros desarrolladores no debería conocer con profundidad las características de almacenamiento que le ofrece el cluster. Desde este punto de vista, al desarrollador le puede dar igual que tipo de volumen puede utilizar (aunque en algún caso puede ser interesante indicarlo), lo que le interesa es, por ejemplo, el tamaño y las operaciones (lectura, lectura y escritura) del almacenamiento que necesita, y obtener del cluster un almacenamiento que se ajuste a esas características.
- La solicitud de almacenamiento se realiza con un elemento del cluster llamado ***PersistentVolumenClaims***.

# Kubernetes Almacenamiento

## Almacenamiento disponible en Kubernetes: PersistentVolumen

### Definiendo un PersistentVolumen

Un *PersistentVolumen* es un objeto que representa los volúmenes disponibles en el cluster. En él se van a definir los detalles del [backend](#) de almacenamiento que vamos a utilizar, el tamaño disponible, los [modos de acceso](#), las [políticas de reciclaje](#), etc.

**Tenemos tres modos de acceso**, que depende del backend que vamos a utilizar:

- ReadWriteOnce: read-write solo para un nodo (RWO)
- ReadOnlyMany: read-only para muchos nodos (ROX)
- ReadWriteMany: read-write para muchos nodos (RWX)

**Las políticas de reciclaje** de volúmenes también depende del backend y son:

- Retain: Reclamación manual
- Recycle: Reutilizar contenido
- Delete: Borrar contenido

# Kubernetes Almacenamiento

## Solicitud de almacenamiento en Kubernetes: **PersistentVolumenClaims**

Si nuestros pods necesita un volumen, necesitamos hacer una solicitud de almacenamiento usando un objeto del tipo **PersistentVolumenClaims**.

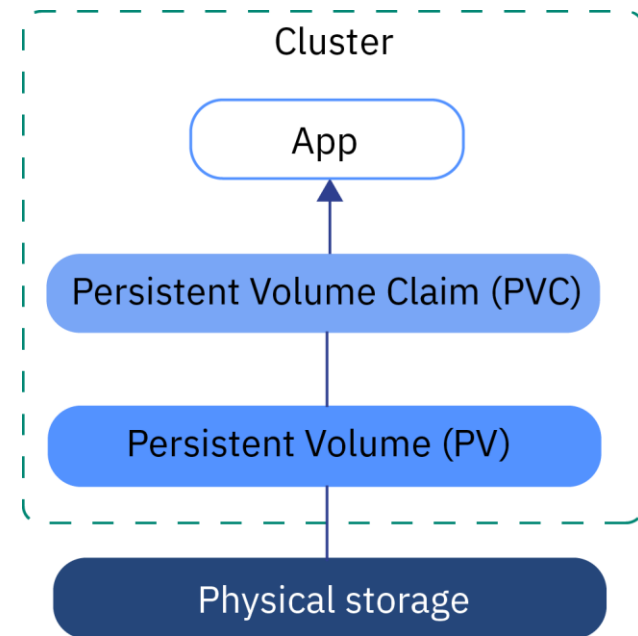
Cuando creamos un **PersistentVolumenClaims**, se asignará un *PersistentVolumen* que se ajuste a la petición. Está asignación se puede configurar de dos maneras distintas:

**Estática:** Primero se crea todos los *PersistentVolumenClaims* por parte del administrador, que se irán asignando conforme se vayan creando los *PersistentVolumen*.

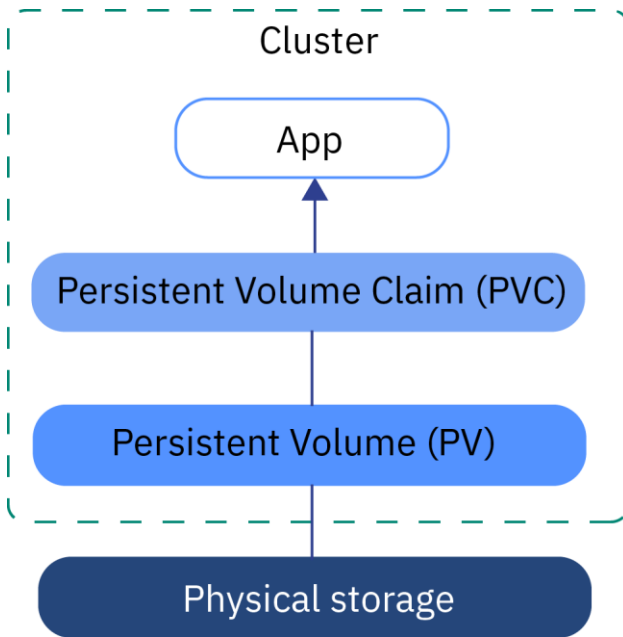
**Dinámica:** En este caso necesitamos un “provisionador” de almacenamiento (para cada uno de los backend), de tal manera que cada vez que se cree un *PersistentVolumenClaim*, se creará bajo demanda un *PersistentVolumen* que se ajuste a las características seleccionadas

**PersistentVolume** -- Donde especificamos el volumen persistente

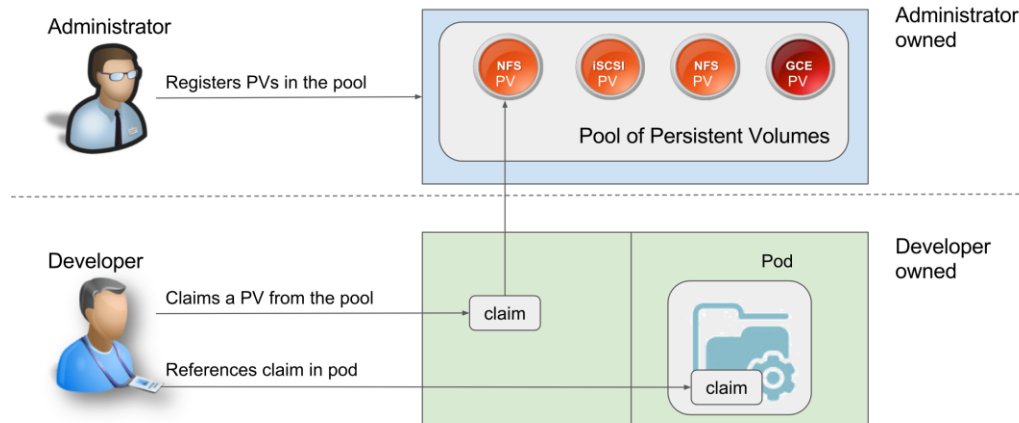
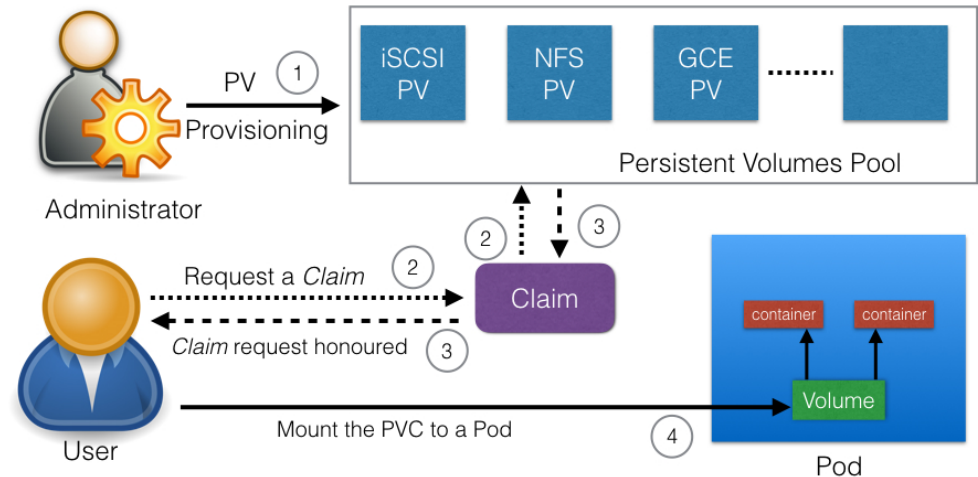
**PersistentVolumeClaim** -- Donde reclamamos espacio en el volumen



# Kubernetes Almacenamiento



## Persistent Volumes Claim (PVC)

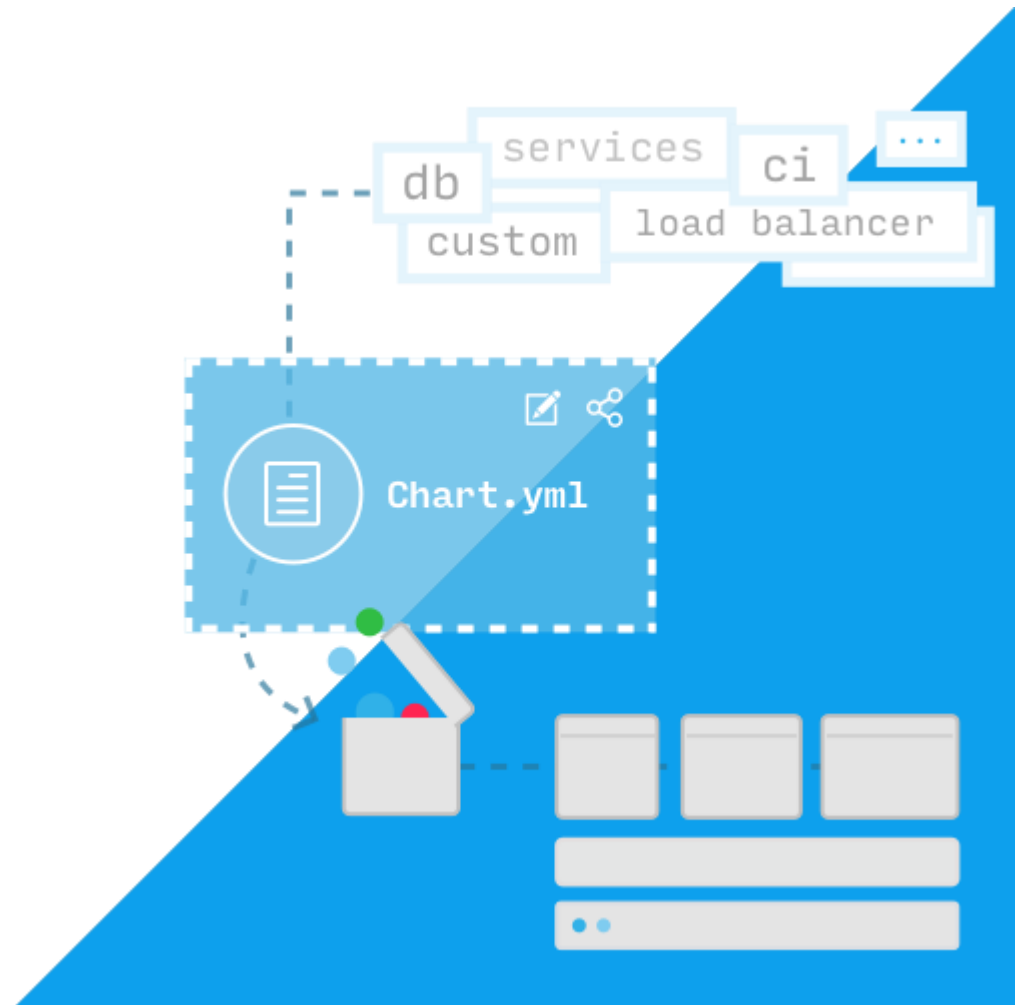




# Kubernetes HELM



<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 \(CNCF\)](https://cloudnativecomputing.org/)

Más sobre Helm en su página principal: [helm.sh](https://helm.sh).

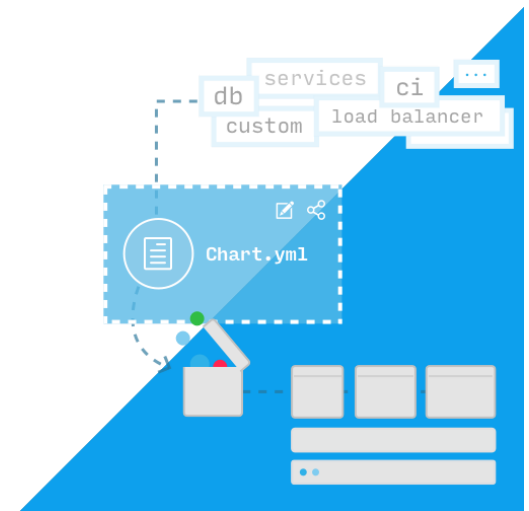
**Helm** significa timón en inglés. Esto tiene que ver con el significado de Kubernetes, que en griego antiguo era el capitán o timonel. Simboliza que Helm es una herramienta para controlar mejor la plataforma de Kubernetes.

# 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 dinámico

### 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

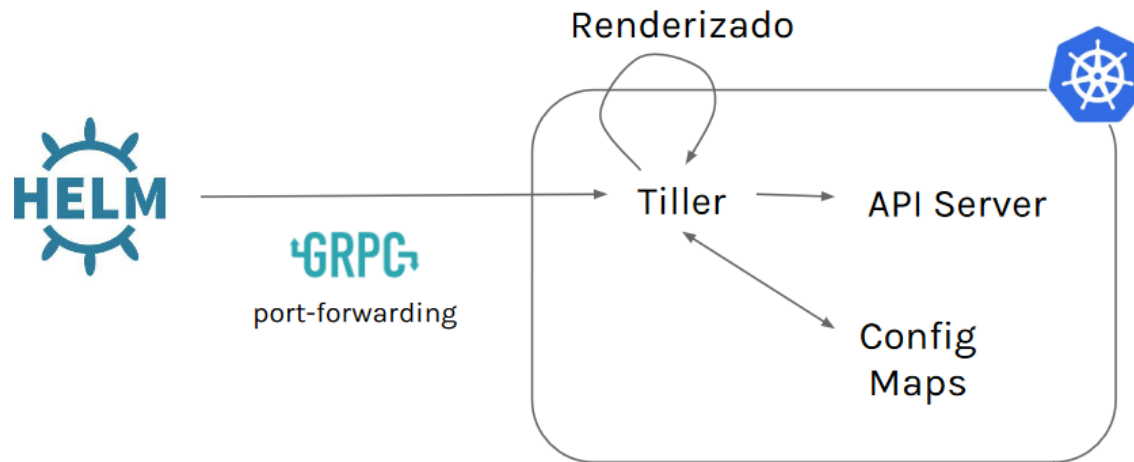
Los chart se componen básicamente de templates. Un template es simplemente un archivo YAML de Kubernetes, que se ha modificado (o no) para que se pueda personalizar. Estos templates al instalar la aplicación se convierten en los YAML que se despliegan en Kubernetes, al final es un archivo YAML de kubernetes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

### Release

Release es como se llama a las instalaciones concretas de una aplicación (chart). En tu 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 comunican a través de port-forwarding, lo que hace es un túnel entre nuestro local y el clúster de kubernetes 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 tiller 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.

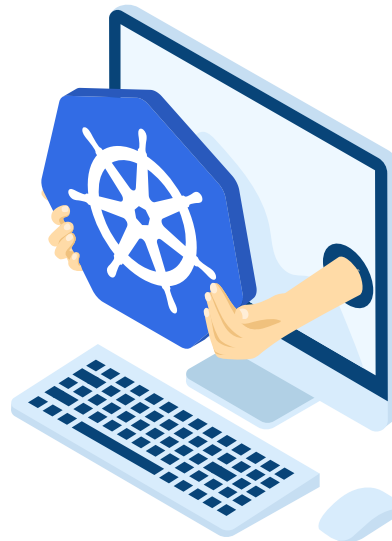
**helm get notes joomla1** # Nos vuelve a mostrar las notas que ha lanzado cuando lo hemos desplegado.

**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



[illegible]