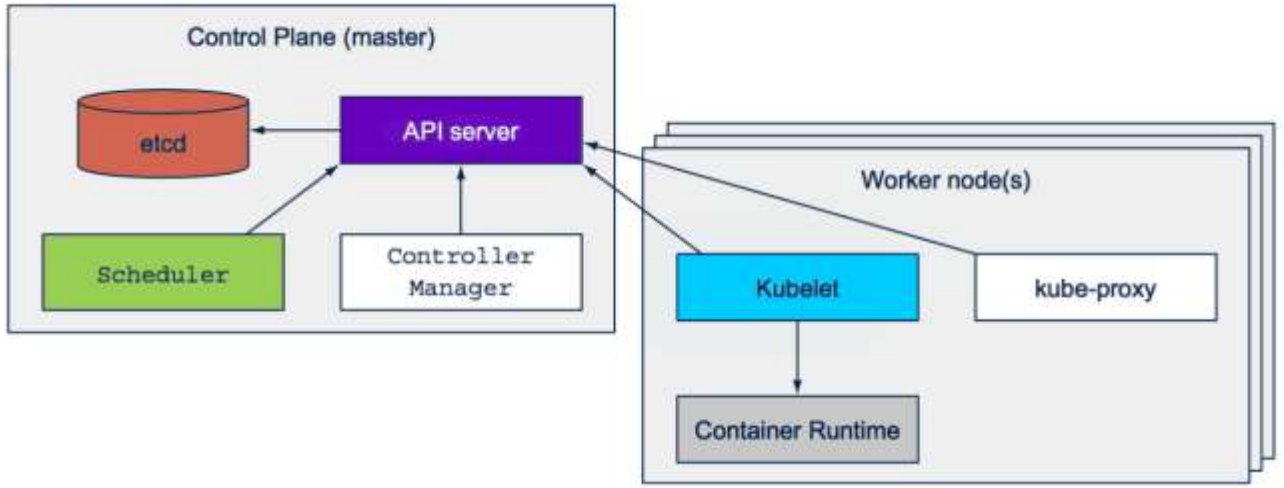
edix
Digital Workers

UF-3 KUBERNETES

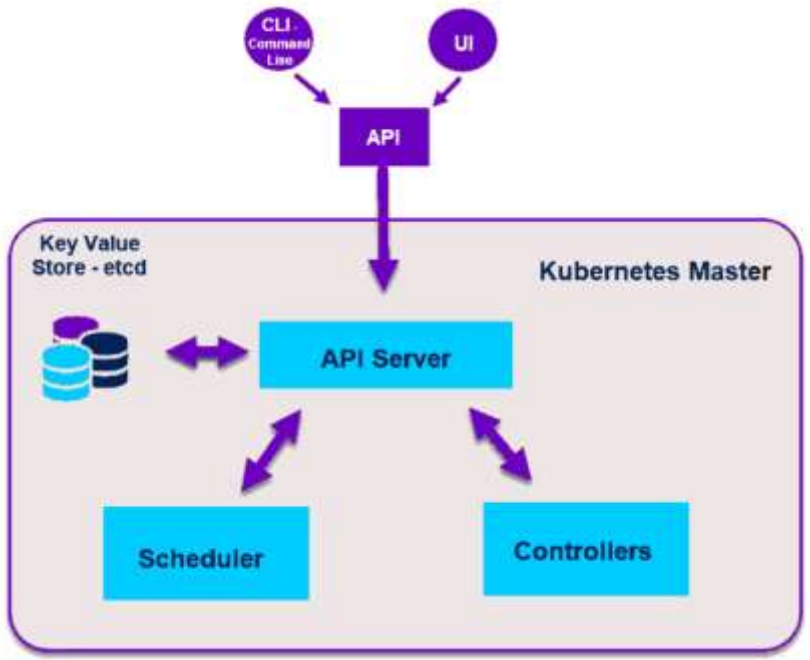
Profesor Raúl Salgado Vilas

- **Kubernetes:** Kubernetes es una herramienta de código abierto que se utiliza para el despliegue y la gestión de contenedores. No se trata de una plataforma de contenerización, sino que ofrece una gestión de aplicaciones multicontenedor. Esta herramienta fue desarrollada inicialmente por Google, basándose en soluciones propias con las que la compañía había estado trabajando internamente durante años. La primera versión se presentó en el año 2014 y, desde entonces, ha ido evolucionando continuamente con nuevas funcionalidades.

- **Arquitectura de Kubernetes:** La arquitectura de Kubernetes es modular, en ella se exponen diferentes servicios que se distribuyen por los múltiples nodos del clúster. Un clúster está formado por dos tipos de componentes. Por un lado, tendremos un nodo maestro o master, también conocido como Control Plane, y, por otro, los nodos de trabajo (worker nodes). Veamos gráficamente los componentes que contiene cada uno de ellos:

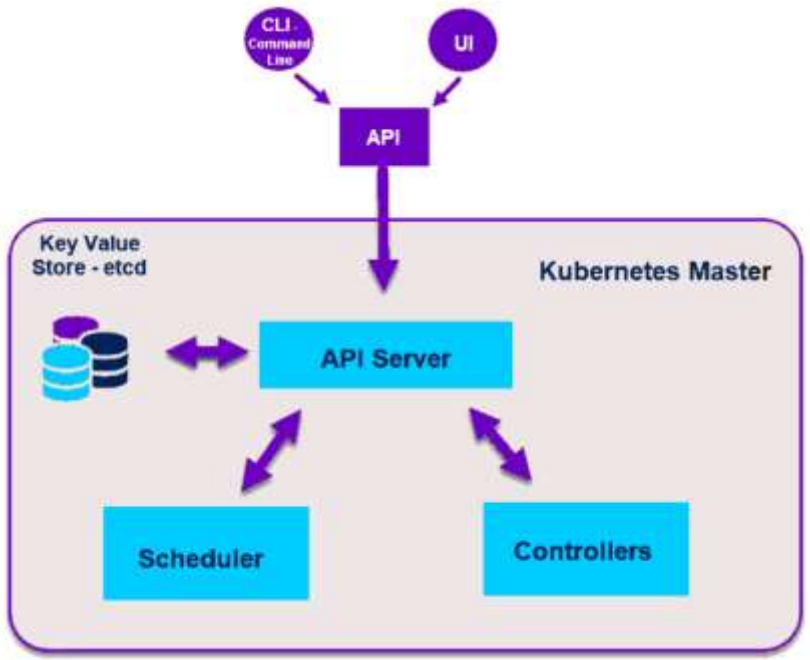


- **Nodos master (Control Plane):** En un clúster de Kubernetes, el nodo master será el responsable de su gestión. Este nodo es el encargado de realizar algunas decisiones que afectarán de forma global a todo el clúster. Por ejemplo, en él se repartirán los trabajos entre los nodos worker disponibles, se crearán nuevos Pods cuando se detecte que el número de réplicas es inferior al deseado, etc. Además, el nodo master será el encargado de recibir y procesar las peticiones a través de la API. Un clúster de Kubernetes puede funcionar con un único nodo master. Sin embargo, si queremos alta disponibilidad deberemos configurar varios nodos, para conseguir redundancia. El algoritmo utilizado para poder establecer quorum entre los nodos exige que el número de nodos master sea siempre impar.

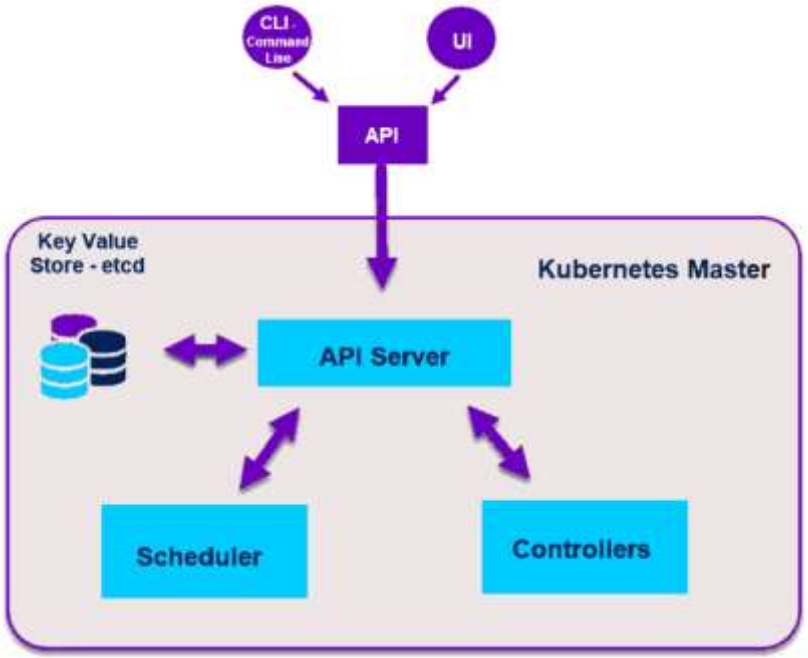


- API server: El componente API server es el encargado de exponer el API REST de Kubernetes para interactuar con el clúster. Actuará como punto de entrada de los comandos enviados para gestionar el clúster, ya sean de componentes internos del sistema o externos y será el encargado de procesar las peticiones y ejecutarlas en caso de ser válidas. Podremos utilizar el API directamente mediante llamadas REST, o bien utilizando una herramienta de línea de comandos, como son kuberctl o kubeadm.

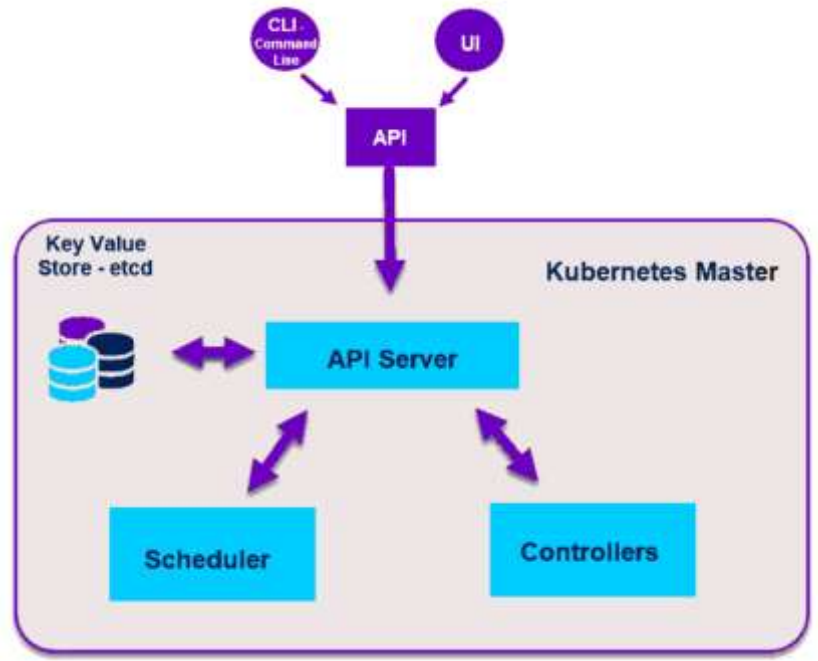
- **Nodos master (Control Plane):** En un clúster de Kubernetes, el nodo master será el responsable de su gestión. Este nodo es el encargado de realizar algunas decisiones que afectarán de forma global a todo el clúster. Por ejemplo, en él se repartirán los trabajos entre los nodos worker disponibles, se crearán nuevos Pods cuando se detecte que el número de réplicas es inferior al deseado, etc. Además, el nodo master será el encargado de recibir y procesar las peticiones a través de la API. Un clúster de Kubernetes puede funcionar con un único nodo master. Sin embargo, si queremos alta disponibilidad deberemos configurar varios nodos, para conseguir redundancia. El algoritmo utilizado para poder establecer quorum entre los nodos exige que el número de nodos master sea siempre impar.



- API server: El componente API server es el encargado de exponer el API REST de Kubernetes para interactuar con el clúster. Actuará como punto de entrada de los comandos enviados para gestionar el clúster, ya sean de componentes internos del sistema o externos y será el encargado de procesar las peticiones y ejecutarlas en caso de ser válidas. Podremos utilizar el API directamente mediante llamadas REST, o bien utilizando una herramienta de línea de comandos, como son kuberctl o kubeadm.



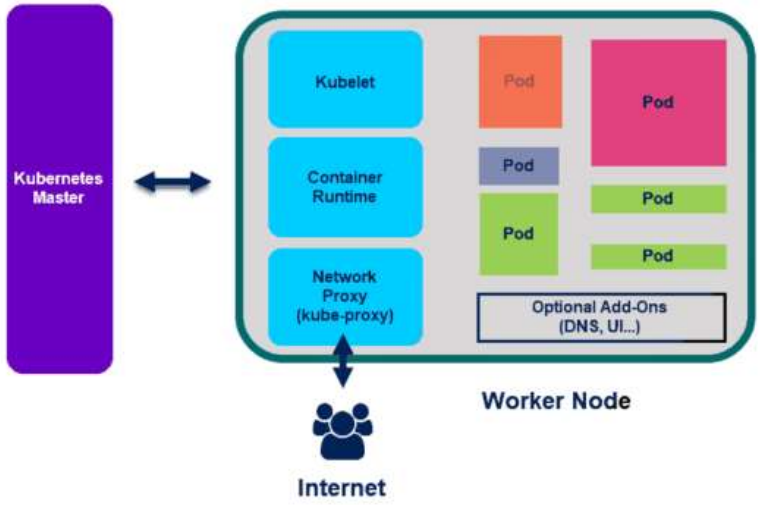
- Scheduler El planificador o Scheduler atiende las peticiones que recibe el API server y las va asignando a los diferentes nodos worker. Para determinar en qué nodo desplegar los Pods, el planificador tendrá en cuenta el estado de los nodos, si están corriendo adecuadamente, cuáles son los recursos disponibles de los que dispone y los requeridos para el nuevo despliegue. En caso de no disponer de ningún nodo adecuado para un Pod, se podrá permanecer en estado pendiente hasta que haya un nodo disponible que cumpla los requisitos



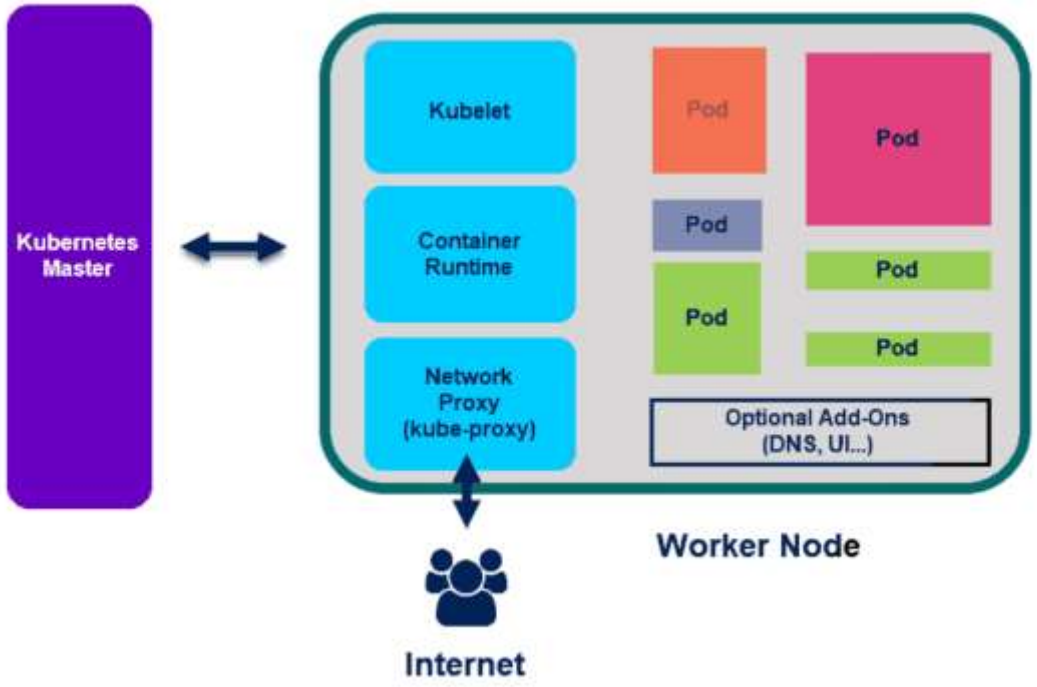
- Controller Manager El Controller Manager es el componente que ejecuta varios procesos controladores encargados de mantener el estado deseado en el clúster de Kubernetes. Mediante llamadas al API server se obtendrá el estado deseado en cada momento y, tras comprobar el estado actual de los nodos, se realizarán las acciones necesarias en caso de haber diferencias. El componente Controller Manager incluye los siguientes controladores:
  - Replication controller. El controlador de replicación será el encargado de la gestión de los Pods en el clúster. Crea nuevos Pods para mantener el número deseado y elimina los que han fallado.
  - Endpoint controller. Será el encargado de proporcionar los endpoints del clúster, asegurando la conexión entre servicios y Pods.
  - Node controller. El controlador de nodo es responsable de la monitorización de los nodos. Detectará cuando alguno deja de estar disponible, desplegando sus Pods en otros nodos.
  - Service controller. Se encarga de la creación de las cuentas predeterminadas, así como los tokens de acceso al API cuando se crea un namespaces.

- Almacenamiento clave-valor (etcd) etcd es un almacén de datos clave-valor distribuido y muy consistente que proporciona una forma confiable de almacenar datos a los que un sistema distribuido o clúster debe acceder. Está desarrollado en Go, ofreciendo un soporte multiplataforma y utiliza el algoritmo de consenso Raft para la comunicación entre las máquinas del sistema distribuido. Kubernetes utiliza etcd tanto para el almacenamiento de la configuración del clúster como para el descubrimiento de servicios. Además, permite la notificación de cambios de configuración en un nodo concreto al resto del clúster de una forma fiable.

- Nodos worker: Los nodos worker serán los encargados de la ejecución de los Pods de nuestras aplicaciones. Los Pods son agrupaciones lógicas de uno o más contenedores junto a un conjunto de recursos compartidos. En cada uno de estos nodos podremos ejecutar múltiples Pods simultáneamente. Aunque es posible tener un solo nodo worker, para conseguir una alta disponibilidad deberemos configurar varios nodos en nuestro clúster. En versiones iniciales de Kubernetes, a estos nodos también se les conocía con el nombre de minions.



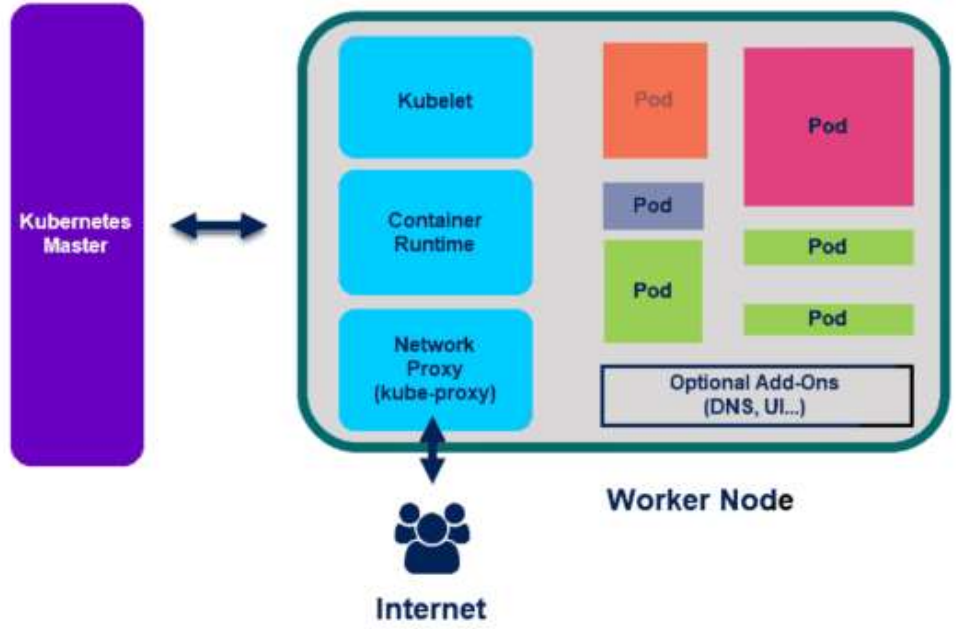
❑ Nodos worker:



- ❑ Kubelet En cada uno de los nodos worker tendremos un servicio llamado kubelet que será el encargado de comunicarse con el nodo master, obteniendo la configuración de los Pods y garantizando que los contenedores definidos en ellos se estén ejecutando correctamente. Además, se comunicará con el almacenamiento etcd del máster para recuperar información de los servicios, y así poder registrar los nuevos servicios que se hayan creado.

- ❑ Runtime de contenedores Un runtime de contenedores es aquel software responsable de la ejecución de los contenedores definidos en los Pods del clúster. Kubernetes soporta cualquier runtime de contenedores que implemente la especificación de runtime OCI (Open Container Initiative). Algunas de las implementaciones más populares son Docker, rkt y runc.

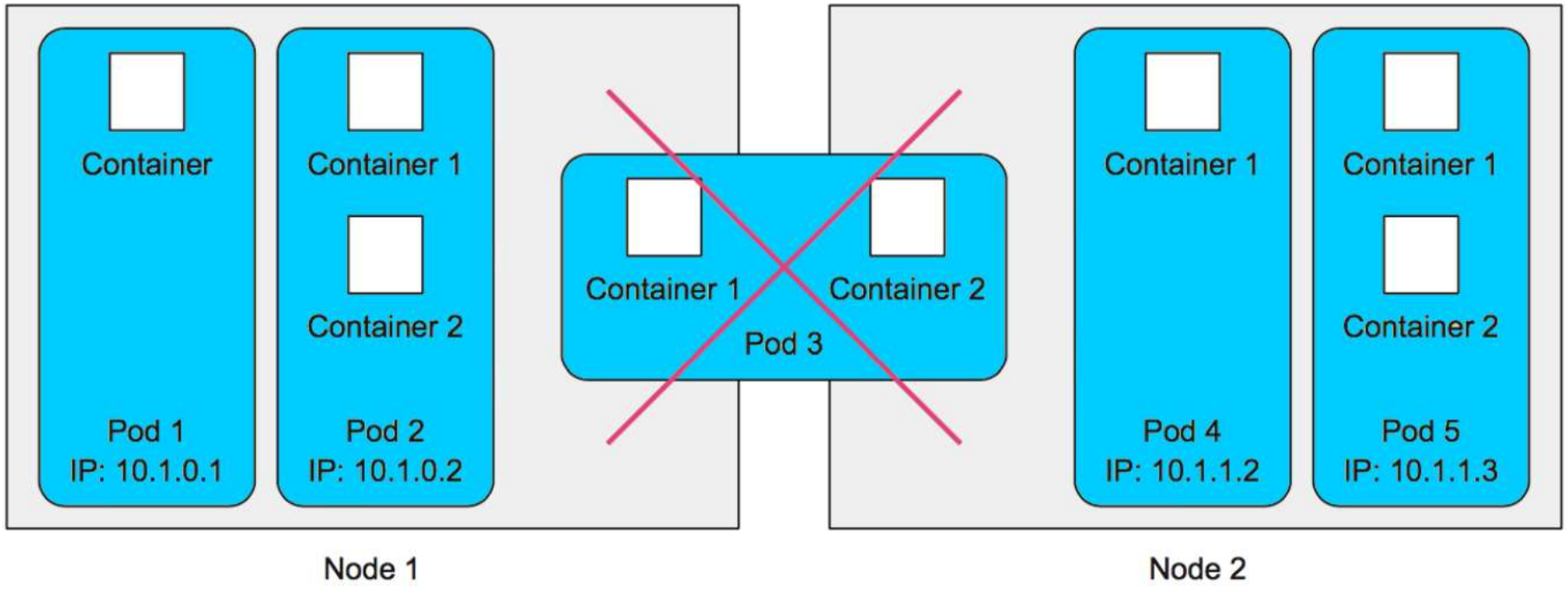
❑ Nodos worker:



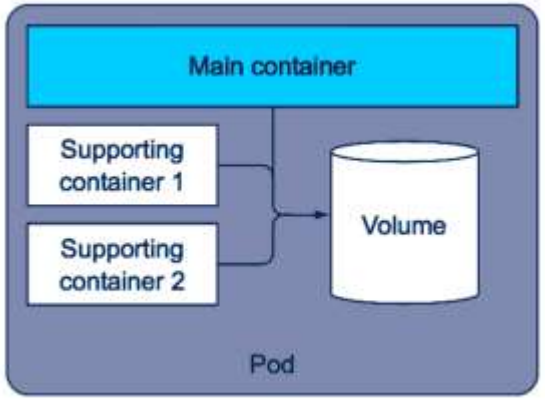
- **Kube-proxy:** El componente kube-proxy es un agente de red que se ejecuta en cada uno de los nodos y será el responsable de las actualizaciones dinámicas y el mantenimiento de las reglas de red. Además, hará funciones de proxy de red y balanceo de carga, redirigiendo el tráfico a los contenedores según la dirección IP y puerto de la petición.

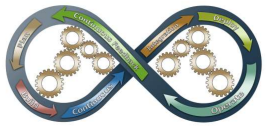
- **Pods:** Como ya comentamos, las aplicaciones se ejecutan mediante Pods. Estos serán desplegados siempre en nodos worker. El Pod es la unidad mínima desplegable dentro de un clúster de Kubernetes. Cuando se despliegan en un nodo, permanecerán allí durante toda su ejecución, hasta que finalicen o sean eliminados. Importante: nunca se moverán de nodo. En caso de fallo, se planificará la creación de un nuevo Pod en otro nodo disponible del clúster. Los contenedores están diseñados para ejecutar un único proceso. Sin embargo, en nuestras aplicaciones, a veces queremos que varios procesos se ejecuten lo más cerca posible y se comuniquen entre sí. Este es el principal motivo por el que en Kubernetes la unidad mínima de despliegue, el Pod, puede estar formada por más de un contenedor.

- **Pods:** Un Pod se desplegará por completo en un mismo nodo, y todos sus contenedores compartirán la misma dirección IP y puertos, estando asociados al mismo hostname. Deberemos tener cuidado de no generar conflictos exponiendo el mismo puerto por varios contenedores de un mismo Pod:



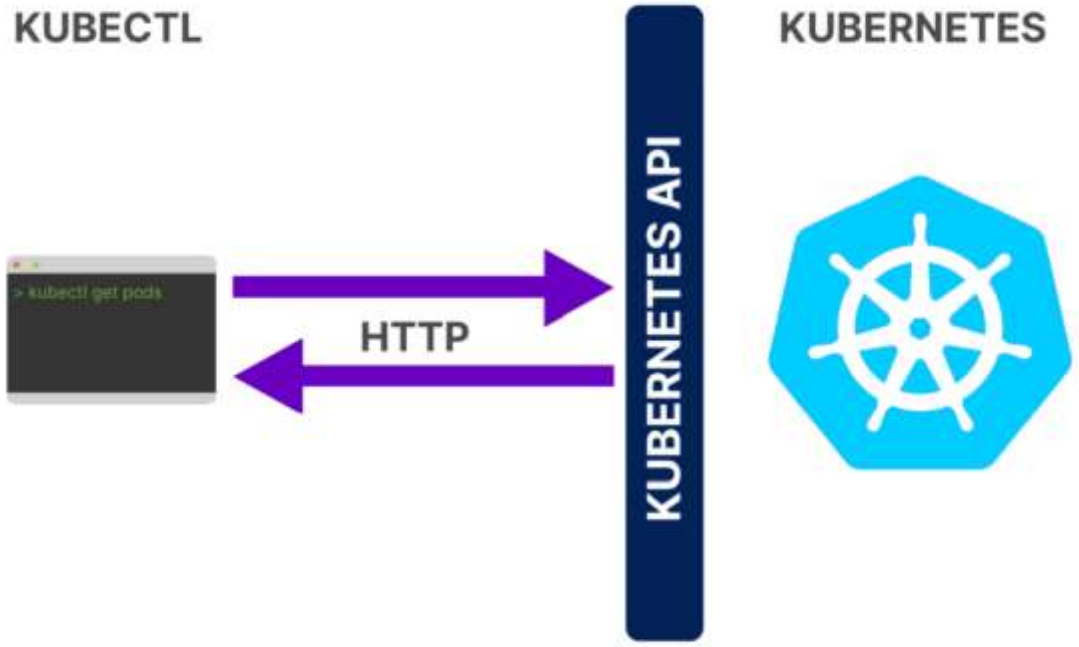
- **Servicios:** En Kubernetes, un servicio define un único punto de acceso para un conjunto de Pods. De esta manera, los servicios proporcionan una dirección IP y uno o varios puertos para acceder a los Pods subyacentes, permitiendo tanto a usuarios y aplicaciones externas al clúster, como a Pods internos comunicarse con los Pods a los que referencia un servicio.
- **Namespace:** Utilizaremos los Namespaces para organizar nuestros objetos en el clúster. Estos nos permiten agrupar recursos para realizar acciones sobre todos ellos. Un caso de uso típico de los Namespaces es el de crear diferentes entornos de ejecución (desarrollo, test, producción, etc.) para nuestras aplicaciones.
- **Volúmenes:** Aunque el sistema de ficheros de un contenedor está completamente aislado, Kubernetes introduce el concepto de volúmenes dentro de los Pods, el cual permitirá compartir almacenamiento temporal o persistente entre los contenedores del Pod. Estarán mapeados a directorios accesibles por los contenedores en rutas locales definidas. Las modificaciones realizadas en el sistema de ficheros local de un contenedor se perderán cuando se reinicien, sin embargo, la información en los volúmenes sí se mantendrá tras un reinicio.





- **PersistentVolumes:** Kubernetes introduce una nueva abstracción denominada PersistentVolumes, la cual nos permitirá desacoplar los Pods de la infraestructura asociada al almacenamiento. A diferencia de los objetos 'volumes', los recursos de tipo PersistentVolume tienen un ciclo de vida independiente a los Pods.
- **ConfigMaps y Secrets:** Los ConfigMaps nos permitirán separar los datos de configuración de los contenedores; están pensados para el almacenamiento de información no confidencial. Por el contrario, para la gestión de información sensible, como contraseñas y certificados, utilizaremos los Secrets.

- **Instalación y configuración:** En Kubernetes necesitaremos instalar un cliente de línea de comandos específico llamado kubectl y, a continuación, instalar y configurar los nodos de nuestro clúster. El cliente kubectl es la herramienta de línea de comandos de Kubernetes que nos permitirá desplegar y gestionar aplicaciones en el clúster. Los comandos que ejecutemos con el cliente serán enviados a Kubernetes mediante llamadas HTTP a su API REST.

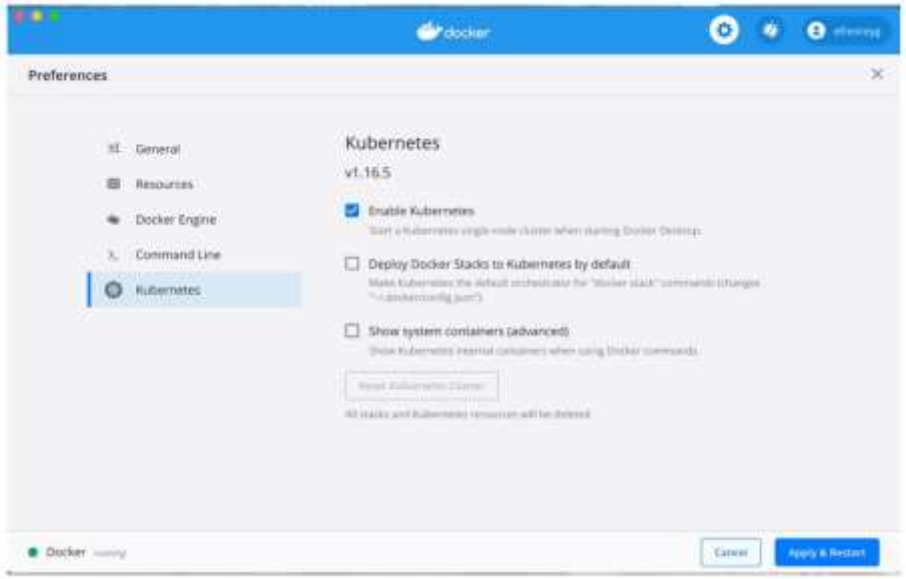


El cliente *kubectl* envía comandos mediante llamadas al API de Kubernetes [1]

- Dependiendo del sistema operativo del que se trate tendremos diferentes opciones de instalación: si hemos instalado un clúster en local, como Minikube o Docker Desktop, entonces ya lo tendremos instalado. Para más detalles sobre la instalación consultar: Kubernetes

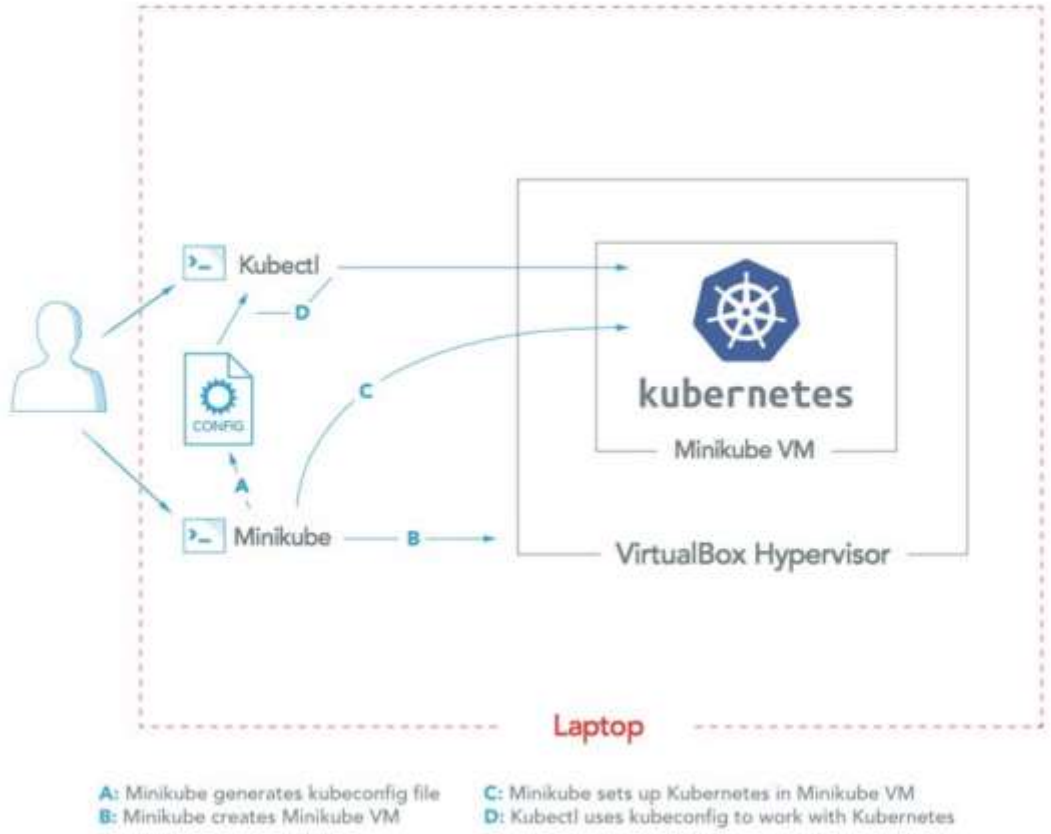
raul@raul-virtual-machine:~/Desktop$ **sudo snap install kubectl --classic**
kubectl 1.26.1 from Canonical✓ installed
raul@raul-virtual-machine:~/Desktop$

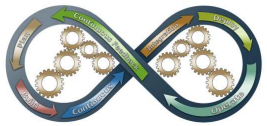
https://kubernetes.io/docs/reference/kubectl/



- Docker Desktop Kubernetes está disponible en la última versión de Docker Desktop, tanto para MacOS como para Windows, e incluye al cliente kubectl y a un servidor Kubernetes local.

- **Mikube:** despliega un clúster de Kubernetes de un solo nodo de forma local o en una máquina virtual. Soportado en diferentes sistemas operativos (Linux, Windows, OSX) e hipervisores (VirtualBox, VMware Fusion, Hyper-V, etc). Además, es una de las herramientas oficiales de Kubernetes: Welcome! | minikube (k8s.io)



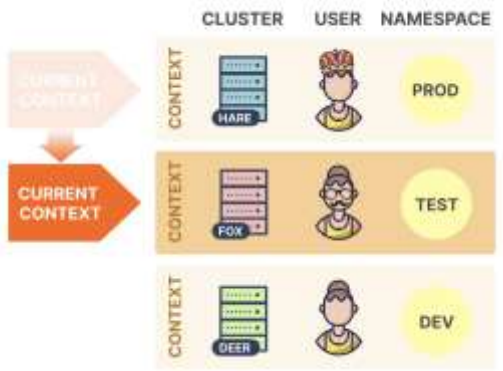


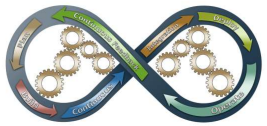
- **Mikube:** Algunas de las características de Kubernetes soportadas por Minikube son DNS, NodePorts, ConfigMaps y Secrets, Dashboard, Container Runtime (Docker, CRI-O, containerd) o Ingress. La instalación es bastante sencilla:

```
curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube
$ sudo cp minikube /usr/local/bin && rm minikube
$ minikube start Starting local Kubernetes cluster.

$ kubectl version
$ kubectl config current-context
$ kubectl cluster-info
$ kubectl get nodes
```

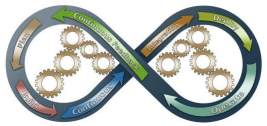
- **Acceso al clúster mediante contextos:** Cada vez que hacemos una petición al API de Kubernetes con el cliente kubectl, se consultan previamente los parámetros de conexión al clúster. Dicha información se almacena habitualmente en el fichero ~/.kube/config. Si queremos trabajar con diferentes clústeres desde la misma máquina (por ejemplo, el clúster por defecto de Minikube y uno productivo en EKS), deberemos tener la configuración de conexión a cada uno definida en dicho fichero. Además, y como veremos, dentro de un mismo clúster podemos tener múltiples namespaces, creando así clústeres virtuales con los que agrupar los recursos. Si queremos que kubectl utilice un namespace específico para un clúster, también estará definido en dicho fichero de configuración.

- Toda esta información estará almacenada en el fichero de configuración mediante la definición de contextos. En Kubernetes los contextos están formados por tres elementos:
  - Clúster. Vendrá especificado por la URL al API de Kubernetes.
  - Usuario. Incluirá las credenciales para un usuario concreto en dicho clúster.
  - Namespace. Será el namespace que se utilizará cuando seleccionemos este contexto.



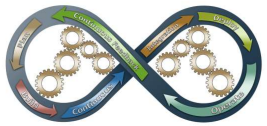


root@raul-virtual-machine:/home/raul/Desktop# **kubectl config set-context dev-context --namespace=dev-ns --cluster=docker-desktop --user=docker-user**
Context "dev-context" created.
root@raul-virtual-machine:/home/raul/Desktop**# kubectl config get-contexts**
CURRENT NAME CLUSTER AUTHINFO NAMESPACE
dev-context docker-desktop docker-user dev-ns
root@raul-virtual-machine:/home/raul/Desktop#

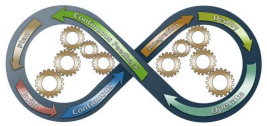


- **Acceso al clúster mediante contextos:** Podemos definir nuevos contextos con el comando kubectl config set-context, al cual le podremos indicar el clúster, el usuario y el namespace. Si no especificamos alguno de ellos, utilizará el actual o por defecto. Si queremos establecer el contexto que acabamos de crear como el contexto actual, deberemos hacer un cambio de contexto con kubectl config usecontex:

```
$ kubectl config set-context dev-context \
 --namespace=dev-ns --cluster=docker-desktop --user=docker-desktop
Context "dev-context" created.
$ kubectl config get-contexts
CURRENT NAME CLUSTER AUTHINFO NAMESPACE
dev-context docker-desktop docker-desktop dev-ns
* docker-desktop docker-desktop docker-desktop
 $ kubectl config use-context dev-context
Switched to context "dev-context".
```



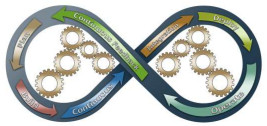
- **Operaciones con Kubernetes:** Para ver realmente de lo que es capaz Kubernetes, lo mejor es empezar a practicar. Veremos de lo que es capaz la herramienta kubectl, usando múltiples sub comandos o acciones. Cada acción tendrá sus propias opciones disponibles. El cliente kubectl soporta multitud de tipos de recursos, los cuales pueden especificarse en formatos largo o corto: pods, po. ,namespaces, ns., nodes, no., deployments, deploy. , replicasets, rs. , daemonsets, ds., statefulsets, sts., jobs., cronjobs, cj. ,services, svc., persistentvolumes, pv. persistentvolumeclaims, pv
- https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands
- **Modo imperativo y declarativo:** En Kubernetes, tendremos dos maneras de trabajar para desplegar nuestras aplicaciones y crear objetos. Por un lado, podremos utilizar el modo imperativo, para el cual utilizaremos comandos de cliente kubectl para la gestión de nuestros objetos, como son run, create, delete o edit (este modo actúa directamente sobre los objetos, permitiendo alterar el yaml de los manifiestos que vamos a ver a continuación). También podremos trabajar de forma declarativa, creando manifiestos en formato YAML, que aplicaremos en el clúster mediante comandos kubectl apply. Los manifiestos indicarán lo que queremos, es decir, el estado deseado, y serán los controladores de Kubernetes los encargados de crear los objetos necesarios y realizar modificaciones cuando haga falta. Esta es la forma recomendada de Kubernetes cuando trabajamos con entornos de producción, ya que por ser ficheros en formato yaml los tendremos versionados en nuestro repositorio, y esto posibilitará que otros servicios y herramientas empleen esos manifiestos (Jenkins o Ansible, por poner ejemplos



❑ Manifiesto de un POD:

```
apiVersion: v1
kind: Pod
metadata:
  name: ejemplo-nginx
spec:
  containers:
  - image: nginx:latest
    name: servidor-nginx
    ports:
    - containerPort: 8080
      protocol: TCP
```

- El fichero de definición de un Pod consta de varias partes. En primer lugar, deberemos indicar la versión del API que estamos utilizando, así como el tipo de recurso que estamos describiendo: en nuestro caso, un Pod. Para ello, utilizaremos las etiquetas apiVersion y kind. A continuación, se definirán las tres secciones más importantes que se usarán en la mayoría de los objetos de Kubernetes:
  - La sección ‘metadata’, que incluirá información como son el nombre, el namespace, etiquetas, etc.
  - En la sección ‘spec’ describiremos el comportamiento deseado del Pod, además de su contenido, es decir, los contenedores, volúmenes, etc… Además, cuando usemos el comando kubectl edit para editar el manifiesto de un componente ya desplegado en el cluster, se añadirá a esto la sección ‘status’. La sección ‘status’ es de solo lectura e incluirá información relativa al estado de la ejecución del Pod.



- **Lanzamiento de un Pod:** Vamos a ver la secuencia de comandos necesarios para lanzar un pod a partir del manifiesto que acabamos de ver, consultar su estado y destruirlo:

```
$ kubectl create -f ejemplo-nginx.yaml
pod "ejemplo-nginx" created
$ kubectl delete -f ejemplo-nginx.yaml
$ kubectl get po ejemplo-nginx -o yaml
```

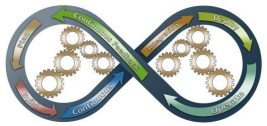
- **Versión imperativa:** Si quisiésemos hacer lo mismo, pero solo con comandos, y sin usar un manifiesto, crearíamos el Pod con kubectl run en lugar de create:

```
$ kubectl run ejemplo-nginx --port=8080 --image=nginx:latest
```

- **Comprobación de salud de los Pods:** Cuando desplegamos aplicaciones en Kubernetes, este se encargará de mantener siempre corriendo los contenedores de los Pods. Si falla o se cae un contenedor, Kubernetes creará uno nuevo. Si un nodo deja de estar operativo, los Pods que estaban en él se desplegarán en un nuevo nodo disponible. Sin embargo, el hecho de que los contenedores estén ejecutándose no quiere decir que nuestra aplicación funcione correctamente. Para comprobar que la aplicación de un Pod está funcionando adecuadamente podemos definir una prueba de vida (liveness probe) para los contenedores. Estas pruebas se podrán realizar de tres maneras:
  - Mediante la ejecución de un comando (exec).
  - Realizando una llamada HTTP (httpGet).
  - Conectándose a un socket TCP (tcpSocket)

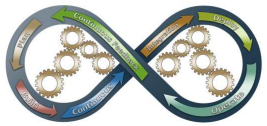
  $kubectl describe pod liveness-exec

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands



❑ **Kubernetes:**

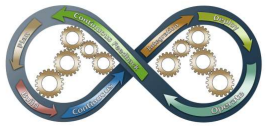
```
# List all pods in plain-text output format.
kubectl get pods

# List all pods in plain-text output format and include additional information (such as
node name).
kubectl get pods -o wide

# List the replication controller with the specified name in plain-text output format. Tip:
You can shorten and replace the 'replicationcontroller' resource type with the alias 'rc'.
kubectl get replicationcontroller <rc-name>

# List all replication controllers and services together in plain-text output format.
kubectl get rc,services

# List all daemon sets in plain-text output format.
kubectl get ds

# List all pods running on node server01
kubectl get pods --field-selector=spec.nodeName=server01
```



❑ **Kubernetes:**

raul@raul-virtual-machine:~/Desktop$ kubectl explain pods.spec.containers.volumeMounts

```
$ kubectl explain pods.spec.containers.volumeMounts
KIND:     Pod
VERSION:  v1

RESOURCE: volumeMounts <[]Object>

DESCRIPTION:
     Pod volumes to mount into the container's filesystem. Cannot be updated.

     VolumeMount describes a mounting of a Volume within a container.

FIELDS:
   mountPath     <string> -required-
     Path within the container at which the volume should be mounted. Must not
     contain ':'.

   mountPropagation      <string>
     mountPropagation determines how mounts are propagated from the host to
     container and the other way around. When not set, MountPropagationNone is
     used. This field is beta in 1.10.
```

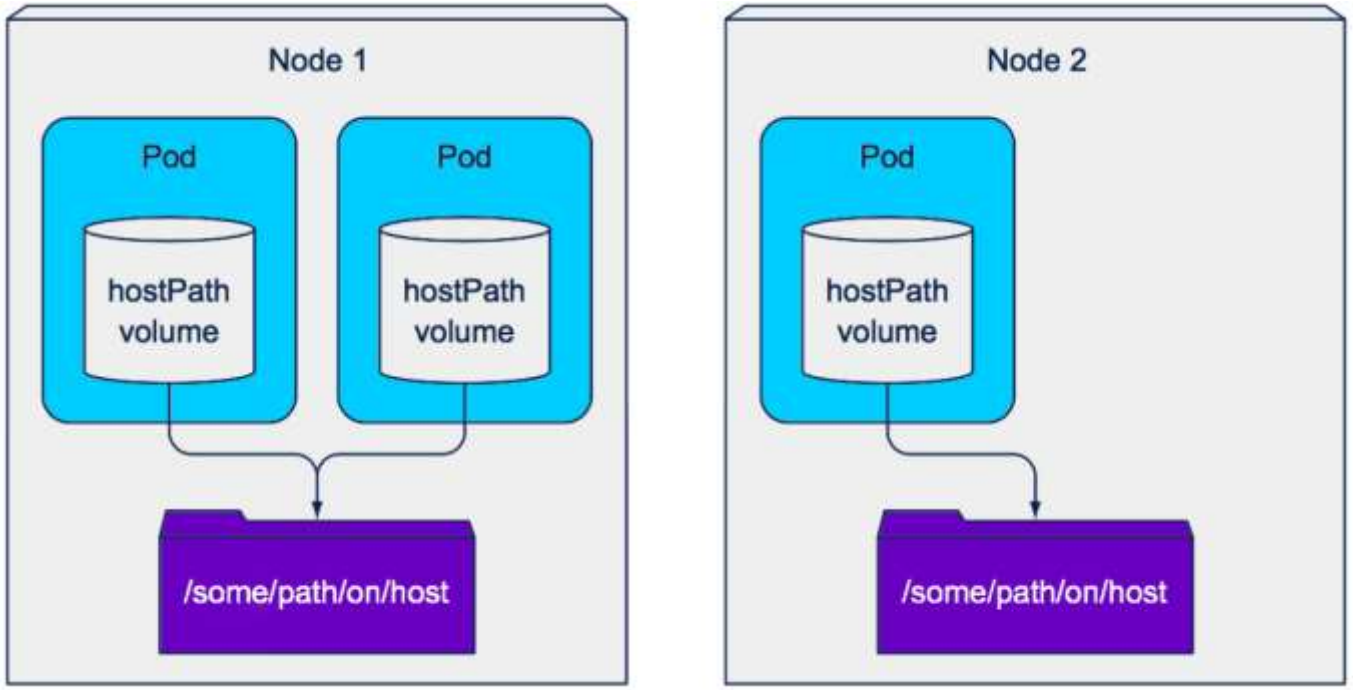


- **Kubernetes:** Almacenamiento de datos en Kubernetes En Kubernetes, los volúmenes son directorios accesibles por todos los contenedores que forman parte de un Pod. Las modificaciones realizadas en el sistema de ficheros locales de los contenedores se perderán cuando se reinicien; sin embargo, la información en los volúmenes sí se mantendrá tras un reinicio del contenedor. Además de los volúmenes existen otros tipos de recursos para el almacenamiento de la configuración.
- En general, podemos clasificar el almacenamiento de Kubernetes en base a la infraestructura que lo respalda y la persistencia que ofrece:
  - Persistencia local al Pod o al nodo (emptyDir, hostPath).
  - Compartición de ficheros (nfs).
  - Asociados a proveedor cloud (awsElasticBlockStore, azureDisk, gcePersistentDisk).
  - Sistemas de ficheros distribuidos (glusterfs, cephfs).
  - Almacenamiento orientado a un propósito específico (gitRepo, configMap, secret).

- **Kubernetes:** Volúmenes emptyDir Los volúmenes de tipo emptyDir nos permitirán compartir ficheros entre los contenedores de un Pod. Como su nombre indica, estos se crean vacíos. Todos los contenedores que forman parte del Pod podrán tener acceso de lectura y escritura a los ficheros del volumen, aunque cada contenedor podrá montar el volumen en una ruta diferente.
- Los volúmenes emptyDir están ligados a la vida útil del Pod, es decir, los datos almacenados serán borrados cuando el Pod sea eliminado del nodo.
- Para configurar un volumen de este tipo en la definición YAML de un Pod, lo añadiremos en la propiedad spec.volumes, donde le daremos un nombre e indicaremos el tipo. Además, en cada uno de los contenedores que vaya a hacer uso del volumen, deberemos indicar en spec.containers.volumeMounts su nombre, la ruta local donde se montará y, opcionalmente, si queremos solamente acceso de lectura desde el contenedor. En el siguiente ejemplo se define un volumen de tipo emptyDir en un Pod, que será utilizado por dos contenedores, cada uno en una ruta distinta y uno de ellos con solo acces

```
apiVersion: v1
kind: Pod
metadata:
  name: web-applicaton
spec:
  containers:
  - name: generador-html
    image: ubuntu:alpine
    volumeMounts:
      - name: html-content
        mountPath: /var/web-site
  - name: servidor-web
    image: nginx:alpine
    volumeMounts:
      - name: html-content
        mountPath: /usr/share/nginx/html
        readOnly: true
  volumes:
  - name: html-content
    emptyDir: {}
```

- **Kubernetes:** Volúmenes hostPath Por lo general, los Pods no deberían acceder al sistema de ficheros del nodo. Sin embargo, en ocasiones tendremos algunos Pods que realizan tareas a nivel del sistema que sí necesitarán acceder al sistema de ficheros de los nodos. Este tipo de Pods habitualmente actuarán como demonios (daemons) y, seguramente, serán desplegados mediante un DaemonSet:

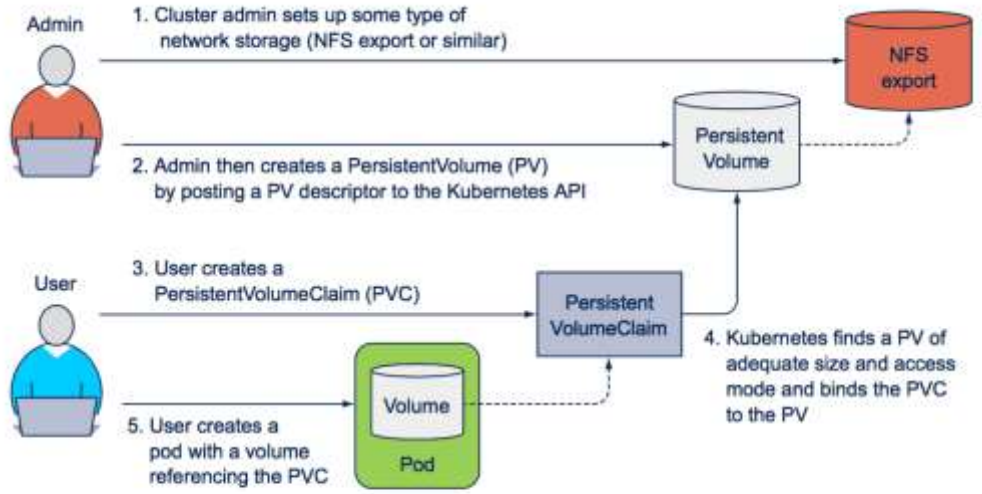


- **Kubernetes:** Los volúmenes de tipo hostPath referencian un directorio o archivo específico del sistema de ficheros del nodo, permitiendo a los Pods que lo monten, acceder a su contenido en la ruta local especificada. Es importante tener en cuenta que dicha ruta será la misma en todos los nodos y deberemos saber previamente si existe o no y si tenemos los permisos necesarios. Algunos ejemplos de casos de uso son:
  - El contenedor necesita acceder a la información interna de Docker (/var/lib/Docker).
  - Se quiere ejecutar cAdvisor en el contenedor para el análisis de recursos (/sys).

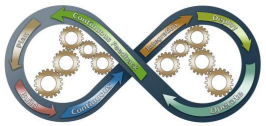
```
---
apiVersion: v1
kind: Pod
metadata:
  name: app-daemon
spec:
  containers:
  - image: ubuntu:alpine
    name: daemon-container
    volumeMounts:
    - mountPath: /data/node
      name: data-node
  volumes:
  - name: data-node
    hostPath:
      path: /data
      type: DirectoryOrCreate
```

- **Kubernetes:** Los Kubernetes PersistentVolume nos permite desacoplar el método de almacenamiento, ocultando la infraestructura subyacente a los desarrolladores. Estos últimos solo tendrán que solicitar la cantidad de almacenamiento requerido, con independencia de la nube donde estén alojados.

- Por lo tanto, serán los administradores del clúster los encargados de configurar el almacenamiento disponible, indicando el tamaño y los modos de acceso permitidos, y registrarlo en Kubernetes utilizando recursos de tipo PersistentVolume. Posteriormente, cuando un usuario necesite almacenamiento lo solicitará por medio de un manifiesto PersistenVolumeClaim, indicando cuánta capacidad necesita y el modo de acceso requerido. Finalmente, será Kubernetes el encargado de asociarle un PersistentVolume adecuado:



Pasos de la creación y uso de un PersistentVolume.



- **Kubernetes:** Creación de un PersistentVolume Antes de crear el recurso PersistentVolume, deberemos tener disponible el almacenamiento físico. Una vez dado este paso, crearemos el manifiesto en formato YAML, donde deberemos indicar la capacidad, los modos de acceso que queremos permitir y la referencia al disco. Los modos de acceso permitidos son:

  - ReadWriteOne (RWO). Únicamente se permite que un nodo monte el volumen para lectura y escritura.
  - ReadOnlyMany (ROX). Está permitido que múltiples nodos monten el volumen para solo lectura.
  - ReadWriteMany (RWX). Varios nodos pueden montar el volumen tanto para lectura como para escritura.