unir
LA UNIVERSIDAD DE INTERNET

# UNIDAD FORMATIVA 3

## Virtualización, Cloud y Contenedores

### Introducción a Kubernetes

# Índice

# Introducción a Kubernetes

Las aplicaciones continúan creciendo más allá de nuestra capacidad para gestionarlas con un nivel óptimo de seguridad en un único servidor. Por ello, ha surgido la necesidad de lo que se ha venido a denominar sistema de orquestación. Este sistema ayuda a los usuarios a ver un conjunto de servidores como un clúster unificado programable y fiable.

Kubernetes es un sistema de código abierto iniciado por Google para hacer frente a esta necesidad que hemos comentado y está basado en ideas que han sido validadas internamente por los sistemas de Google a lo largo de los últimos años (Borg y Omega). Estos últimos se utilizan para ejecutar y gestionar los innumerables servicios de Google, incluyendo la búsqueda de Google, el correo de Google, etc. La mayoría de los ingenieros que construyeron y operaron con clústeres de Borg a gran escala han estado involucrados en el diseño y construcción de Kubernetes.

# Objetivos

Al finalizar este tema, seremos capaces de dar respuesta a los siguientes interrogantes y conseguir las siguientes destrezas:

- Qué es Kubernetes y cuáles son sus principales características.
- Visión general de la arquitectura de Kubernetes
- Conocer los principales componentes de Kubernetes y para qué sirven.
- Poner en práctica lo aprendido con los Pods, el pilar fundamental sobre el que se construirá todo lo demás.

# ¿Qué es 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.

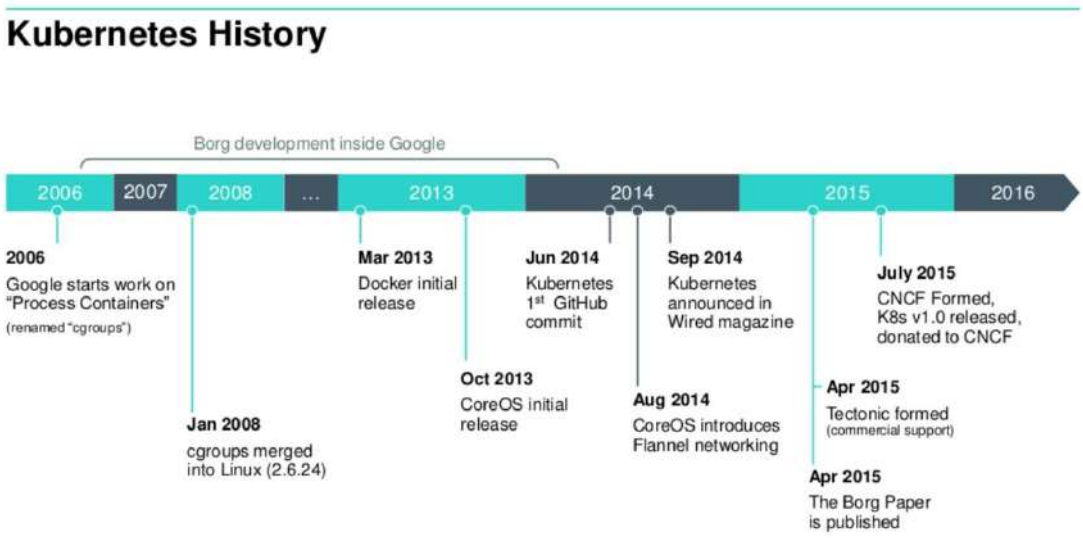



Imagen 1. Historia de Kubernetes [1]

Entre las principales características y capacidades de Kubernetes podemos encontrar:

- El empaquetado automático de contenedores.
- El descubrimiento de servicios y balanceo de carga.
- El almacenamiento desacoplado.
- La autorreparación.
- La gestión de la configuración de aplicaciones.
- La ejecución por lotes y tareas programadas.
- El escalado horizontal.
- El despliegue de actualizaciones automatizadas y sin cortes de servicio.
- Los rollbacks de despliegue a versiones previas.
- La gestión de los recursos del clúster.

## 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:

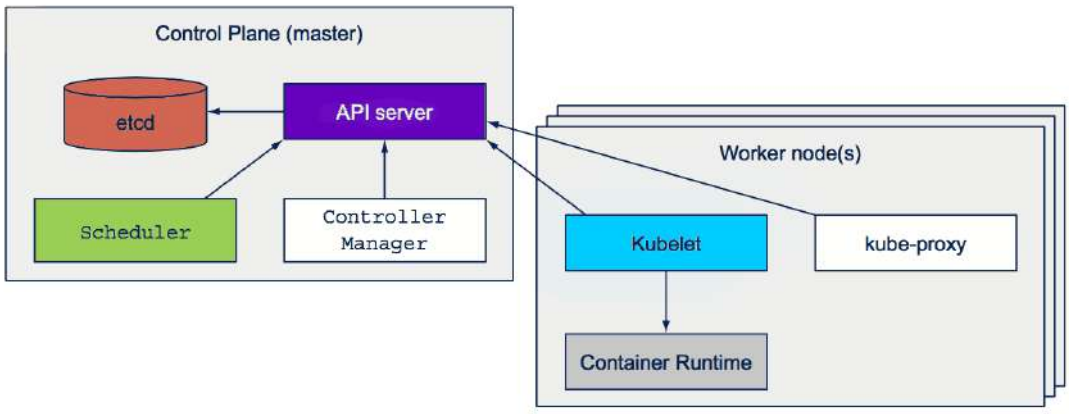



Imagen 2. Componentes que forman un clúster de Kubernetes [2]

A continuación, veremos con más detalle cada uno de los componentes de cada tipo de nodo.

## 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.

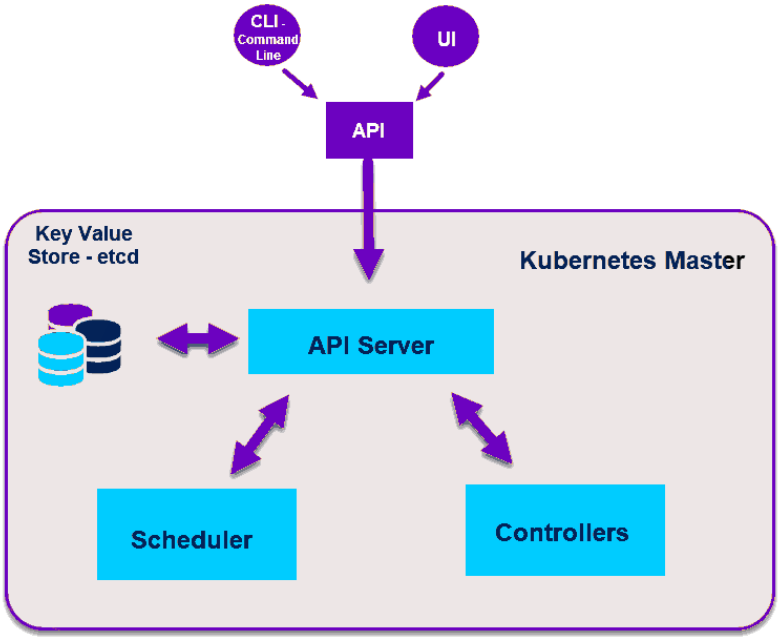



Imagen 3. Arquitectura del nodo *master* de Kubernetes [3]

Veamos en mayor profundidad cada uno de estos componentes.

### *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.

Aunque desde un punto de vista lógico cada controlador se ejecutaría como un proceso independiente, con el fin de reducir la complejidad, todos los controladores están compilados dentro del mismo binario y son ejecutados como un único proceso.

### *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.

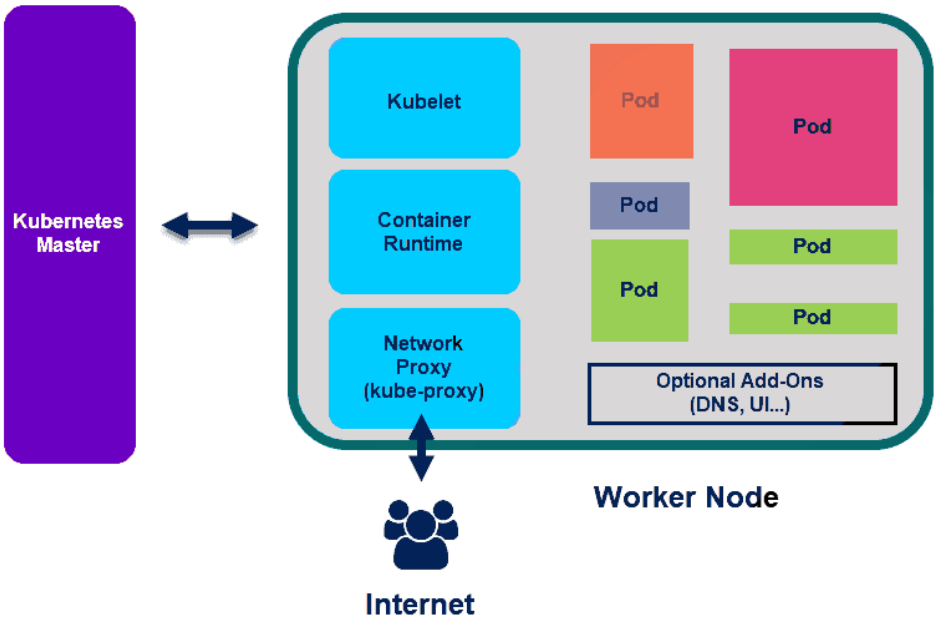



Imagen [4]. Arquitectura del nodo worker de Kubernetes [3]

A continuación, veremos con más detalle los componentes existentes en los 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.

#### *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.

## Objetos básicos de Kubernetes

Los objetos proporcionados por Kubernetes son entidades utilizadas para el despliegue, mantenimiento y escalado de nuestras aplicaciones en el clúster, ya sea en una infraestructura localizada en la nube o en una propia. Mediante estos objetos, describiremos las aplicaciones que queremos ejecutar en el clúster y cuáles son los recursos disponibles para ellas, además de una serie de políticas que definirán cómo se comportan nuestras aplicaciones.

La creación de los objetos se hará siempre mediante llamadas al API de Kubernetes, el cual recibirá la especificación de los objetos en formato JSON. Sin embargo, si en vez de realizar llamadas directamente al API utilizamos el cliente de línea de comandos *kubectl*, podremos utilizar el formato YAML para la descripción de nuestros objetos. Será el propio cliente kubectl el encargado de convertir la información a JSON antes de realizar las llamadas al API.

Veamos a continuación algunos de los objetos básicos de Kubernetes que utilizaremos al definir aplicaciones para ser ejecutadas en el clúster.

### Pods

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**.

Cada uno de los contenedores de un Pod se ejecutarán dentro de su propio *cgroup* (verificar que se ha introducido el concepto en Docker), sin embargo, todos compartirán algunos namespaces de Linux. Es decir, existe un aislamiento parcial entre los contenedores que pertenecen a un mismo Pod. Además, al ejecutarse los contenedores de un Pod bajo el mismo namespace de IPC, todos los procesos podrán comunicarse entre sí mediante IPC (IPC (*Inter process communication*) es un mecanismo que permite la comunicación entre procesos).

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.

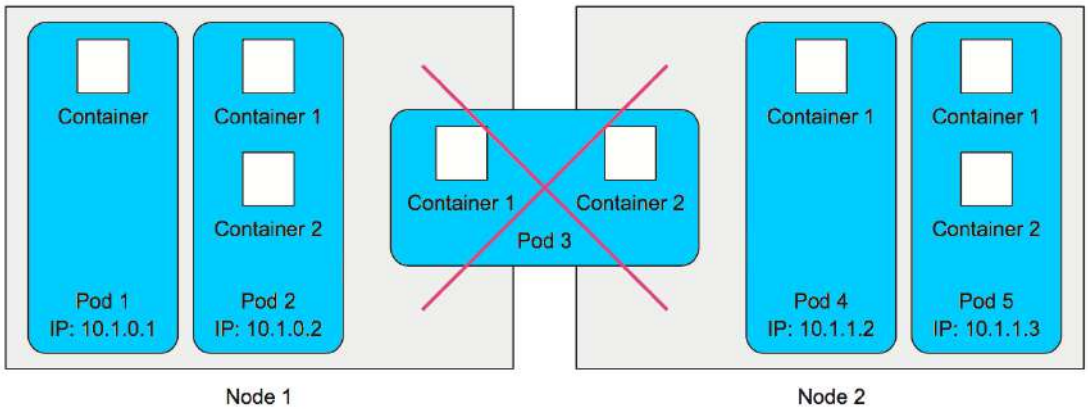



Imagen 1. Los contenedores de un Pod se despliegan en el mismo nodo y comparten IP [2]

En Kubernetes, todos los Pods desplegados en cualquiera de los nodos comparten el mismo espacio de direcciones de red. Esto significa que, por defecto, cualquier Pod del clúster podría comunicarse con otro a partir de su dirección IP. Esta red plana entre los Pods del clúster es independiente de la topología real de la red entre las máquinas de los nodos.

Podríamos pensar que los Pods actúan como máquinas virtuales donde alojar toda nuestra aplicación. Sin embargo, deberemos aprender a descomponer nuestra aplicación y organizarla en múltiples Pods, de manera que sea posible escalar cada capa de la aplicación de manera individual. Normalmente, utilizaremos Pods con un único contenedor, sin embargo, habrá ocasiones en las que tendremos un contenedor principal y varios de apoyo para tareas específicas, pero que están relacionadas con el principal.

## 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.

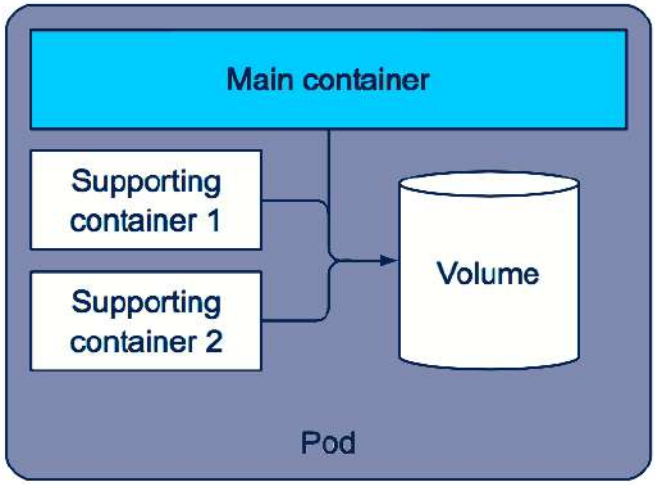



Imagen 2. Múltiples contenedores en un Pod compartirán acceso a los volúmenes. [2]

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.

## Controladores de Kubernetes

En Kubernetes existen, además, otras abstracciones de mayor nivel denominadas controladores. Estos se apoyan en los objetos básicos que hemos visto, proporcionando funcionalidad adicional en la gestión y ejecución de los Pods.

Estudiaremos brevemente los principales controladores disponibles:

### Deployment

En Kubernetes utilizaremos los Deployments para gestionar los Pods de nuestra aplicación; estos son ideales para aplicaciones o microservicios sin estado. Además, nos permitirán actualizar los Pods de las aplicaciones sin cortes de actividad. Cuando creamos un Deployment, automáticamente se creará un ReplicaSet para la creación y gestión de los Pods según el número de réplicas deseadas.

## ReplicaSets

Los ReplicaSets son, generalmente, creados por los Deployments, aunque pueden crearse independientemente. Serán los encargados de crear en nuestro clúster múltiples copias de un mismo Pod. Asimismo, garantizan que nuestra aplicación tiene el número deseado de Pods, creándolos y escalándolos según los disparadores configurados en el Deployment. En otras palabras, se encargará de crear un nuevo Pod en caso de que uno de los Pods en ejecución muera o se detenga por algún motivo.

## Replication Controller

Los ReplicationControllers funcionan de manera similar a los ReplicaSets, pero con alguna limitación, por ejemplo, no soportan el uso de selectores para definir un conjunto de recursos. Se recomienda la utilización de los ReplicaSets en su lugar.

## StatefulSets

Al igual que ocurría con los ReplicaSets, los StatefulSets nos permitirán gestionar el despliegue y escalado de un conjunto de Pods según se haya definido. Sin embargo, se diferencia de los Deployments en que los Pods de los StafulSets no se sustituyen completamente, es decir, cada Pod tendrá asociado un identificador único que el controlador se encargará de persistir y mantener ante cualquier replanificación del Pod.

Los StatefulSets se utilizan para aplicaciones **con estado,** como puede ser una base de datos. Toda la información asociada al estado de los Pods gestionados por los StatefulSets será almacenada en un volumen asociado.

## DaemonSets

Utilizaremos los DaemonSets cuando queramos que un determinado Pod se ejecute en cada uno de los nodos del clúster. Estos serán los encargados de crear un Pod cada vez que un nodo se una al clúster. Asimismo, nos serán útiles para la ejecución de tareas en segundo plano, como, por ejemplo, para tareas motorización o recopilación de logs.

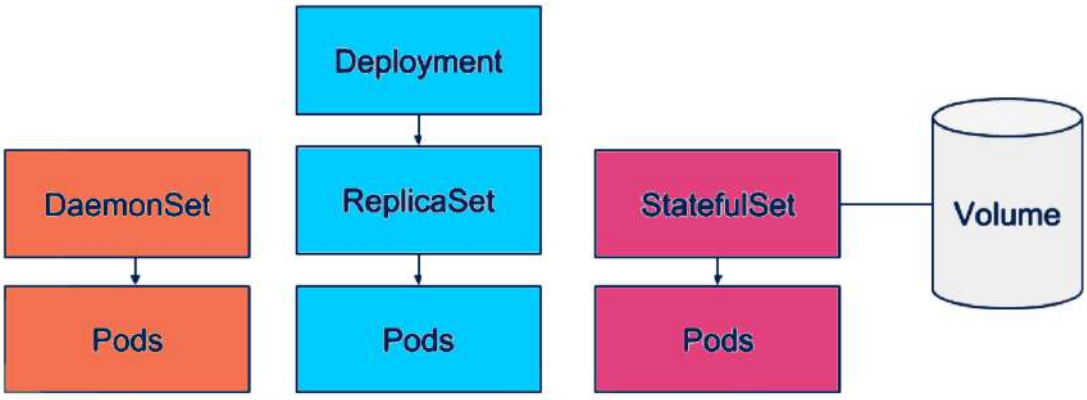



Imagen 5. Controladores de alto nivel de Kubernetes [4]

Los StatefulSets y DaemonSets estarían al mismo nivel de abstracción que los ReplicaSets, sin embargo, no tienen un nivel de abstracción superior que los controle, como sí ocurría con los ReplicaSets y el Deployment.

### Jobs

Los Jobs permitirán crear Pods para la ejecución de un proceso por lotes, supervisando que la tarea ha finalizado con éxito. Se diferencia de los ReplicaSets en que una vez que el proceso del contenedor finaliza con éxito, el Pod no se vuelve a reiniciar.

### CronJobs

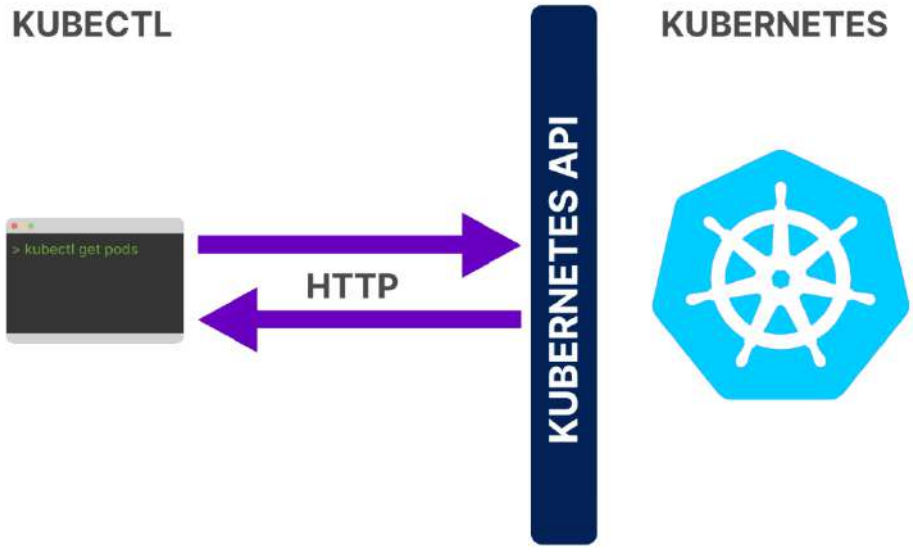
Los CronJobs son similares a los Jobs, pero permiten su programación para repetirse periódicamente, por ejemplo, cada hora o diariamente, o a ciertas fechas y horas establecidas.

## Instalación y configuración

Ahora que tenemos una visión general de cómo funciona Kubernetes, el siguiente paso será preparar un entorno con el que podamos trabajar. La instalación y configuración de un clúster de Kubernetes es algo más compleja que el orquestador Docker Swarm.

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:

https://kubernetes.io/es/docs/tasks/tools/install-kubectl/

## Tipos de despliegues de Kubernetes

Existen multitud de herramientas para facilitarnos el despliegue de un clúster de Kubernetes, así como de servicios de proveedores en la nube para crear clústeres directamente en su infraestructura.

Dependiendo de nuestros requerimientos y casos de uso, optamos por un modelo de despliegue u otro. Además, debemos tener en cuenta algunos criterios como el soporte para alta disponibilidad, la frecuencia de actualizaciones, soporte para entornos híbridos, etc.

### En un solo nodo o máquina

Para entornos de desarrollo y pruebas es muy habitual utilizar un clúster de Kubernetes local formado por un solo nodo, normalmente, en nuestra propia máquina. Existen varias alternativas para dicho entorno local, siendo las más populares Docker Desktop y Minikube.

#### *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. Cuando habilitamos el soporte para Kubernetes en Docker Desktop, podremos seguir desplegando contenedores Docker en Swarm o independientes, sin que se vea afectado de ninguna manera. Además, nos ofrece integración con Docker, pudiendo desplegar nuestras stacks o pilas de Docker directamente en Kubernetes.


Imagen 6. Versiones de los productos instalados con Docker Desktop.

Una vez que tengamos Docker Desktop instalado en nuestra máquina, la habilitación de Kubernetes es muy sencilla. Iremos a la opción 'Preferences' del menú de Docker y seleccionaremos 'Kubernetes'. Veremos que por defecto la opción Kubernetes no está habilitada, así que, marcaremos la opción 'Enable Kubernetes' y aplicaremos los cambios.

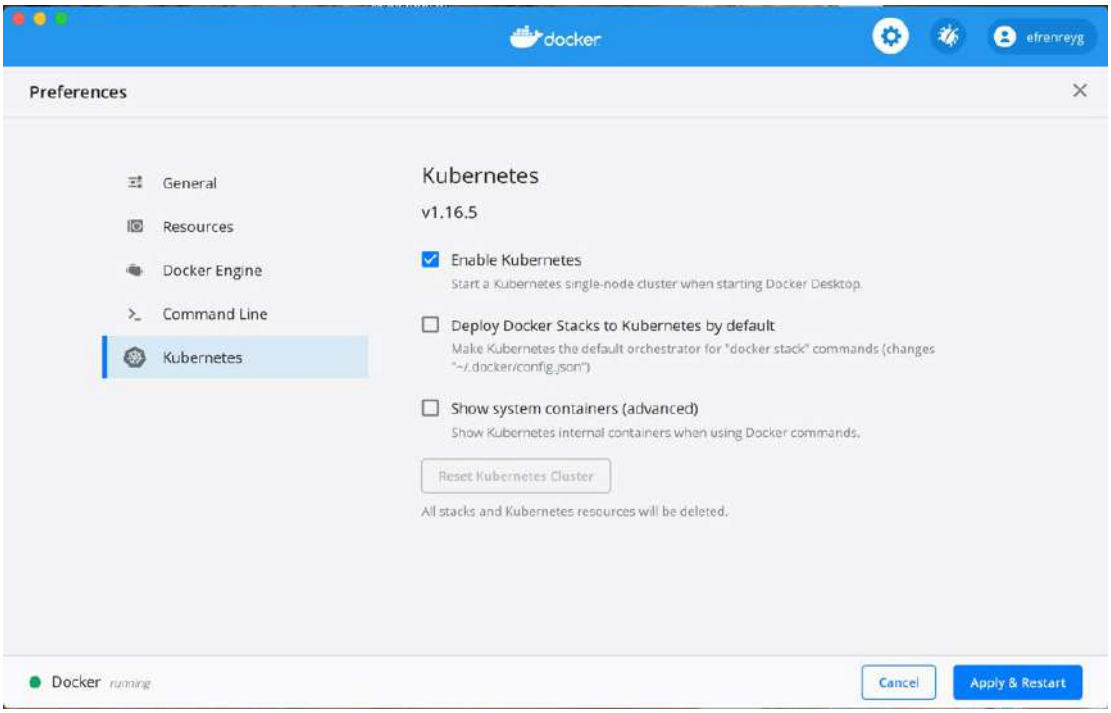



Imagen 7. Configuración de Kubernetes en Docker Desktop.

La instalación comenzará y tardará unos minutos. Una vez finalizada, veremos en la parte inferior de la misma ventana que ambos están corriendo en la máquina:


Imagen 8. El estado indica que el cliente y servidor de Kubernetes están ejecutándose.

***Minikube***

Minikube 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.

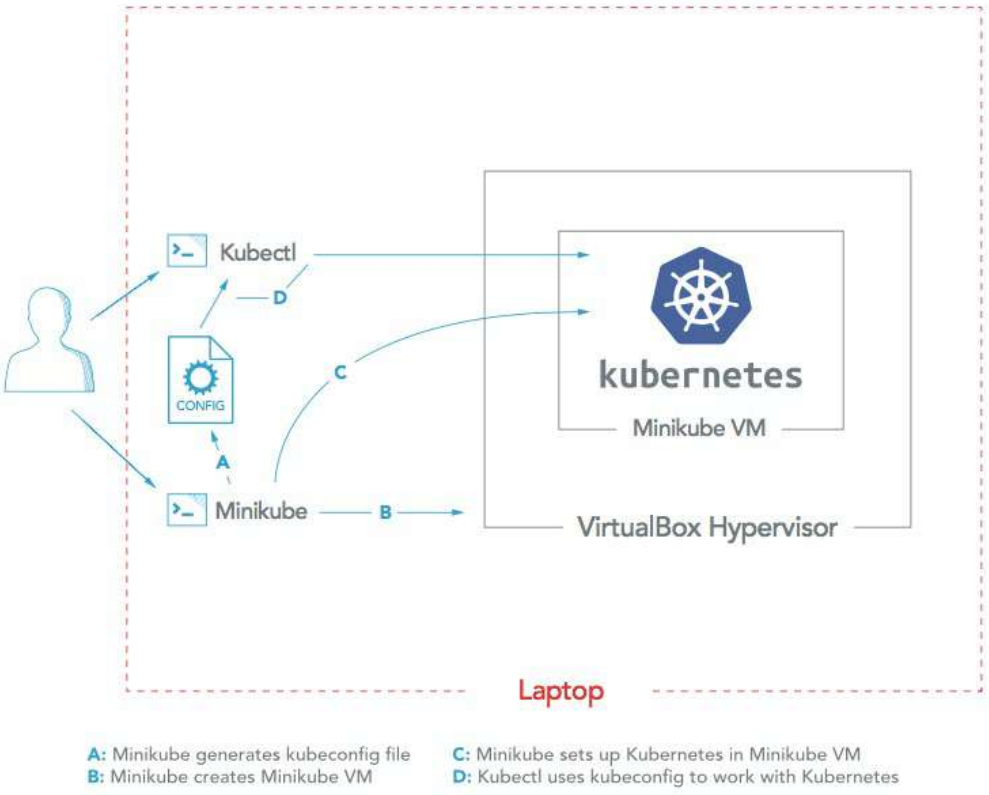



Imagen 11. Despliegue de Kubernetes con Minikube. [5]

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, aunque deberemos instalar previamente algún hipervisor, como, por ejemplo, Virtual Box. Por ejemplo, en Linux bastaría con descargar el ejecutable y otorgarle los permisos adecuados:

```
$ 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...
...
```

Para más detalles acerca de la instalación en las diferentes plataformas, consulta la documentación de Kubernetes.

### En servidores, mediante un instalador

Este método nos permite desplegar los nodos de Kubernetes en servidores, ya sean máquinas físicas de un datacenter o máquinas virtuales, ya estén dentro del mismo datacenter o en la nube.

La utilización de este tipo de instaladores requiere un amplio conocimiento técnico del diseño interno de Kubernetes y nos permitirá una mayor personalización del clúster.

Como contrapartida, el despliegue y mantenimiento requieren de más tiempo y recursos. Algunas de las herramientas de instalación más populares son las siguientes:

- Kubeadm. Permite el despliegue de clústeres tanto en máquinas físicas como en entornos virtualizados. El despliegue es más sencillo que con kops y kargo. Por defecto no soporta alta disponibilidad, aunque puede ser configurada.

- Kops: Permite desplegar el clúster utilizando un documento de especificación generado por los administradores. Solamente soporta despliegues en AWS, sin embargo, será la propia herramienta kops la encargada de crear los recursos necesarios en AWS. Soporta alta disponibilidad y extensiones para gestión de red y monitorización del clúster (Flannel, Calico, Dashboard, etc.).

- Kubespray. Soporta el despliegue del clúster en múltiples plataformas, como son AWS, GCE, Azure, OpenStack y servidores bare metal. Está basado en **Ansible**, por lo que requiere un buen conocimiento de esta herramienta y, además, es necesaria la creación de un fichero de inventario para Ansible.

### Como servicio en la nube

La manera más fácil de desplegar un clúster de Kubernetes es a través de los servicios que nos ofrecen los proveedores de nube pública. Estos nos permiten reducir el esfuerzo dedicado a la gestión y, además, son rápidos y escalan bien. Serán una opción válida siempre y cuando sea adecuado alojar los datos y procesos de nuestra aplicación en una nube pública.

Algunos de los principales servicios de Kubernetes para la nube son:

- Amazon Elastic Kubernetes Service (EKS).

- Azure Kubernetes Service (AKS).

- Google Kubernetes Engine (GKE).

- Alibaba Cloud Container Service for Kubernetes (ACK).

Veamos con más detalle la propuesta de Amazon:

*Amazon EKS*

El servicio Amazon EKS (Elastic Kubernetes Service) nos permite ejecutar un clúster de Kubernetes con la ventaja de no tener que preocuparnos por la gestión del Control Plane. Será el propio servicio EKS el encargado de aprovisionar, escalar y gestionar los nodos del Control Plane según las necesidades del clúster. Nuestra gestión se limitará a la creación de los nodos worker mediante plantillas de CloudFormation ya predefinidas.

Además, el servicio Amazon EKS está perfectamente integrado con otros servicios de AWS que nos permitirán garantizar la escalabilidad y seguridad de las aplicaciones. Entre ellos están Amazon ECR (registro de imágenes), Amazon ELB (balanceador de carga), IAM (seguridad) y Amazon VPC (aislamiento de red).

## Verificación de Kubernetes

Ahora que ya hemos habilitado Kubernetes en la máquina, debemos comprobar que tenemos todo funcionando correctamente.

**Nota**

Asumimos, a continuación, que estamos usando la versión de Docker Desktop y que por tanto está instalado kubectl.

El siguiente comando nos muestra las versiones utilizadas en el cliente y el servidor del contexto actual:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"16+", GitVersion:"v1.16.6-beta.0",
GitCommit:"e7f962ba86f4ce7033828210ca3556393c377bcc", GitTreeState:"clean", BuildDate:"2020-
01-15T08:26:26Z", GoVersion:"go1.13.5", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"16+", GitVersion:"v1.16.6-beta.0",
GitCommit:"e7f962ba86f4ce7033828210ca3556393c377bcc", GitTreeState:"clean", BuildDate:"2020-
01-15T08:18:29Z", GoVersion:"go1.13.5", Compiler:"gc", Platform:"linux/amd64"}
```

En Kubernetes, un contexto se refiere a un grupo de parámetros de acceso, formado por la combinación de un determinado clúster, un usuario y un namespace. Si consultamos el contexto actual, veremos que tenemos uno creado, llamado docker-desktop:

```
$ kubectl config current-context
docker-desktop
```

Para comprobar que el cliente kubectl está correctamente configurado y tiene acceso a nuestro clúster, mostraremos la información de este de la siguiente forma. En caso de que el clúster no estuviese accesible nos mostraría un error.

```
$ kubectl cluster-info
Kubernetes master is running at https://kubernetes.docker.internal:6443
KubeDNS is running at
https://kubernetes.docker.internal:6443/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Para finalizar, si listamos los nodos del clúster, veremos que está formado por un único nodo master:

```
$ kubectl get nodes
NAME          STATUS   ROLES    AGE   VERSION
docker-desktop   Ready    master   15m   v1.16.6-beta.0
```

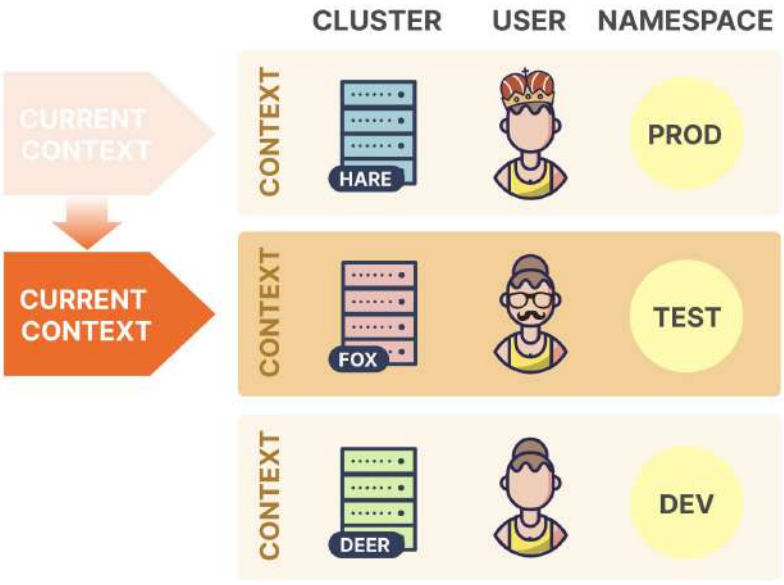
## Acceso al clúster mediante contextos

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.




Los contextos nos permiten cambiar de clúster y namespace fácilmente. [1]

De entre todos los contextos definidos en la configuración, siempre tendremos uno que será el contexto actual, el cual será el que usará kubectl si no le indicamos otra cosa en los argumentos. Podemos consultar la lista de contextos disponibles, así como cuál es el contexto actual de la siguiente manera:

```
$ kubectl config get-contexts
CURRENT   NAME             CLUSTER        AUTHINFO       NAMESPACE
*         docker-desktop   docker-desktop   docker-desktop

$ kubectl config current-context
docker-desktop
```

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 **k*ubectl config use-context***:

```
$ 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. Veamos algunos ejemplos:

- **pods**, *po.*
- **namespaces**, *ns.*
- **nodes**, *no.*
- **deployments**, *deploy.*
- **replicasets**, *rs.*
- **daemonsets**, *ds.*
- **statefulsets**, *sts.*
- **jobs.**
- **cronjobs**, *cj.*
- **services**, *svc.*
- **persistentvolumes**, *pv.*
- **persistentvolumeclaims**, *pvc.*

Para obtener un listado completo de los recursos soportados en nuestro clúster, podemos utilizar el comando kubectl api-resources, consultar la ayuda en la propia línea de comandos con -h, o bien revisar la documentación de kubectl en este enlace.

## 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).

**Estructura del manifiesto de un Pod**

Veamos, como ejemplo, un manifiesto sencillo. En este caso, un sencillo 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.

## Documentación de los recursos de Kubernetes

Cuando estemos definiendo nuestros los recursos de la aplicación en ficheros YAML, necesitaremos consultar algunos de los atributos soportados y sus posibles valores. Para ello, podemos consultar la documentación de referencia de Kubernetes en la web.

Otra opción, quizás más cómoda, sería utilizar el comando ***kubectl explain***, el cual nos permite listar los atributos soportados por un recurso. El comando recupera la información utilizando el API de nuestro clúster, que debe estar ejecutándose, recuperando así la versión específica de Kubernetes que está instalada. Además, nos permite consultar atributos al nivel que queramos. El comando recibirá la ruta de la propiedad a consultar. Veamos un par de ejemplos:

```
$ kubectl explain pods
KIND:     Pod
VERSION:  v1
DESCRIPTION:
 Pod is a collection of containers that can run on a host. This resource is
 created by clients and scheduled onto hosts.
FIELDS:
   apiVersion <string>
     ...
   kind <string>
     ...
   metadata <Object>
     ...
   spec <Object>
     ...
   status <Object>
     ...

$ kubectl explain pods.spec.containers.imagePullPolicy
KIND:     Pod
VERSION:  v1
FIELD:    imagePullPolicy <string>
DESCRIPTION:
   Image pull policy. One of Always, Never, IfNotPresent. Defaults to Always
   if :latest tag is specified, or IfNotPresent otherwise. Cannot be updated.
   More info:
   https://kubernetes.io/docs/concepts/containers/images#updating-images
```

## 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 get po ejemplo-nginx -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2021-11-17T17:12:52Z"
  name: ejemplo-nginx
  namespace: default
  resourceVersion: "2411"
  uid: a527a4a5-3945-4ba8-88e6-ac0dfdce50ca
spec:
  containers:
  - image: nginx:latest
    imagePullPolicy: Always
    name: servidor-nginx
    ports:
    - containerPort: 8080
      protocol: TCP
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-8rvbl
      readOnly: true
..
$ kubectl delete -f ejemplo-nginx.yaml
pod "ejemplo-nginx" deleted

$ kubectl delete po ejemplo-nginx
Error from server (NotFound): pods "ejemplo-nginx" not found
```

Como vemos en el caso del borrado, las operaciones se pueden hacer sobre un manifiesto o sobre un identificador del recurso que ese manifiesto contiene. La diferencia fundamental es que la operación '**delete'** se aplica, en este caso, a **todos los recursos definidos** por ese manifiesto.

### 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
```

Como vemos, la sintaxis es mucho más sencilla. Pero en cuanto la complejidad de nuestros componentes empieza a crecer, deja de ser tan útil como usar manifiestos.

## 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).

Si la prueba de vida definida falla, Kubernetes entiende que no está funcionando bien y reiniciará el Pod. Veamos algunos ejemplos de cómo configurarlos en la definición de los Pods:

```
spec:
  containers:
  - name: liveness
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy

      httpGet:
        path: /health
        port: 8080

      tcpSocket:
        port: 8080
```

Además de las pruebas de vida, también podemos definir dos tipos más de pruebas:

- Pruebas de arranque (**startup probe**) para detectar problemas en el arranque de un Pod. Si el Pod tarda más de lo esperado creará uno nuevo.
- Pruebas de disponibilidad (**readiness probe**) para detectar interrupciones temporales del servicio. En este caso, el Pod no se reiniciará, sino que dejará de recibir tráfico mientras esté fallando la prueba.

Para obtener más información acerca de la configuración de estas pruebas de salud, consultad la documentación oficial en el siguiente enlace.

## Gestión de los recursos de un Pod

A veces, queremos controlar la cantidad de recursos que utilizan los contenedores de nuestros Pods, ya sea estableciendo unos recursos mínimos requeridos o fijando unos límites de uso. En estos casos, utilizaremos la sección ***spec.containers.resources,*** donde podremos especificar los requisitos mínimos (**requests**) o los límites de uso (**limits**). Veamos un ejemplo:

```
...
spec:
  containers:
    - image: app-image:latest
      name: app
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
```

Como práctica, veamos qué nos dice Kubernetes sobre estos dos valores, para entender la diferencia:

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

RESOURCE: resources <Object>

DESCRIPTION:
     Compute Resources required by this container. Cannot be updated. More info:
     https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

     ResourceRequirements describes the compute resource requirements.

FIELDS:
   limits     <map[string]string>
     Limits describes the maximum amount of compute resources allowed. More
     info:
     https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

   requests              <map[string]string>
     Requests describes the minimum amount of compute resources required. If
     Requests is omitted for a container, it defaults to Limits if that is
     explicitly specified, otherwise to an implementation-defined value. More
     info:
     https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
```

- **limits**: Marca el máximo que puede llegar a consumir este Pod.
- **requests**: El mínimo imprescindible para arrancar el Pod, con un valor por defecto igual a limits.

Para consultar y monitorizar el consumo de recursos de los Pods o Nodos del clúster podemos utilizar el comando ***kubectl top.***

```
$ kubectl top nodes
NAME                  CPU(cores)   CPU%     MEMORY(bytes)   MEMORY%
ip-10-10-1-247.ec2.internal  38m        1%       410Mi         5%

$ kubectl top pods --all-namespaces
NAMESPACE    NAME                      CPU(cores)   MEMORY(bytes)
kube-system   aws-node-9nmnw               2m         20Mi
kube-system   coredns-7bcbfc4774-q4pjj       2m         7Mi
kube-system   coredns-7bcbfc4774-wwlcr        2m         7Mi
...
```

Importante

Este comando requiere que esté desplegado el servidor de métricas (Metrics server) en el clúster. En los despliegues de un solo nodo (Minikube, Docker Desptop, etc.) normalmente no viene instalado, por lo que sería necesario desplegar uno.

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/high-availability.yaml
```

Para más detalles al respecto, y diferentes opciones de instalación, consultar el README.md del repositorio de Metrics server.

## Etiquetas y anotaciones

Kubernetes nos permite adjuntar ciertos metadatos a nuestros objetos del clúster mediante el uso de etiquetas y anotaciones.

- Las **etiquetas** serán pares clave/valor y su principal uso será para seleccionar colecciones de objetos que cumplan determinadas condiciones.
- Utilizaremos las **anotaciones** para añadir información adicional a nuestros objetos en forma de metadatos, ya sean estructurados o no.

## Etiquetas (*labels*)

En Kubernetes, las etiquetas nos permitirán **organizar** de manera lógica los recursos y serán utilizadas por la mayoría de los recursos de mayor nivel mediante **selectores**. Además, nos permitirán **filtrar** de manera selectiva los objetos de los comandos kubectl para seleccionar solamente aquellos que necesitamos, independientemente de que estemos consultando o realizando modificaciones sobre recursos.

Si modificamos la definición de nuestro Pod añadiendo la sección labels, todas las etiquetas que definamos en ella serán aplicadas al recurso cuando ejecutemos el comando ***kubectl apply***. Es decir, si tenemos definida una etiqueta que no tenía el objeto, se le añadirá, si ya existía y el valor es diferente, se actualizará, y si el recurso tiene una etiqueta asociada que no está en nuestro YAML, será eliminada del recurso:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-nginx
  labels:
    environment: "development"
    app: "demo-app"
spec:
  containers:
  - image: nginx:latest
    name: servidor-nginx
    ports:
    - containerPort: 8080
      protocol: TCP
```

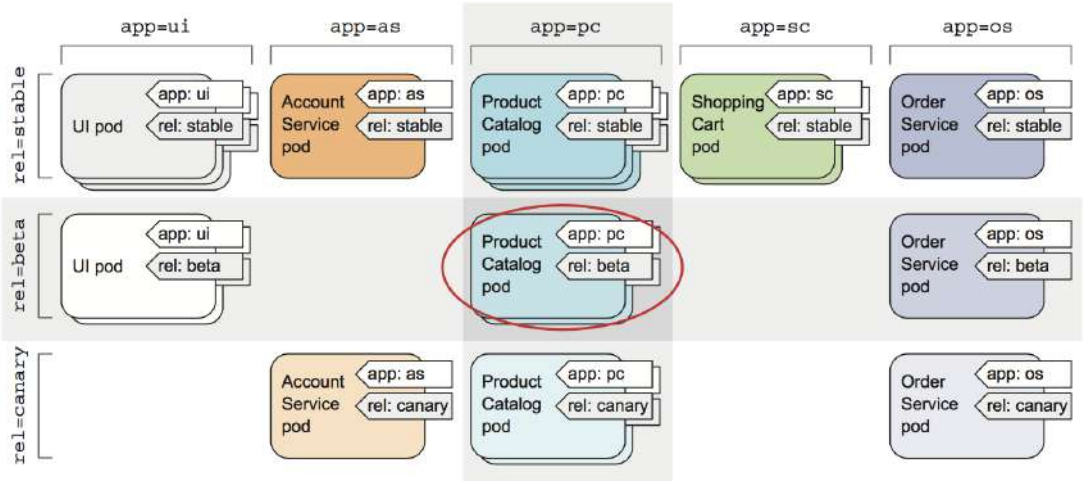
Una vez que apliquemos las modificaciones, podremos listar los Pods para comprobar los cambios sobre nuestro Pod:

```
$ kubectl apply -f app-nginx.yaml
pod/app-nginx configured

$ kubectl get pods --show-labels
NAME          READY   STATUS    RESTARTS   AGE   LABELS
ejemplo-nginx   1/1     Running   0          9s    app=demo-app,environment=development
```

### Selectores de etiquetas (selectors)

Los selectores nos van a permitir filtrar objetos en base a un conjunto de etiquetas. Podremos utilizar los selectores con el cliente kubectl, pero además algunos de los objetos de Kubernetes también los utilizaran para su funcionamiento como iremos viendo.




Selección de Pods mediante selector de múltiples etiquetas (*app=pc,rel=beta*). [2]

Los selectores estarán formados por una serie de condiciones booleanas separadas por comas. Un recurso deberá cumplir con todas las condiciones para ser seleccionado por el filtro definido. La siguiente tabla muestra los diferentes operadores que podemos utilizar en los selectores cuando utilizamos el cliente.

Si quisiéramos listar solamente los Pods cuya etiqueta enviroment tiene el valor desarrollo, podríamos utilizar la opción **'--selector'** con la condición deseada:

```
$ kubectl get pods --selector="environment=desarrollo"
$ kubectl get pods --selector="environment!=production"
$ kubectl get pods --selector="environment!=production, app in (demo, test)"
```

## Anotaciones (annotations)

Mediante las **anotaciones** podremos agregar metadatos a los objetos del clúster que sirvan de ayuda a herramientas y librerías externas. Serán datos no identificativos de los objetos, pero que aportarán información adicional útil a aquellas herramientas que interactúen con Kubernetes a través de su API. Algunos de los casos de uso habituales de las anotaciones son:

- Registrar el motivo de la última actualización de un objeto.
- Adjuntar información sobre la compilación, la release o la imagen, como, por ejemplo, número de pull requests, rama de Git, hash de la imagen, registro utilizado, marcas de tiempo, etc.
- Proporcionar información adicional para interfaces externas, por ejemplo, referenciando un icono o enlace relacionado.
- Datos de contacto del equipo responsable del objeto.

Kubernetes, en ocasiones, también hará uso de las anotaciones. Por ejemplo, al realizar un despliegue en fases (*rolling deployments*) se utilizarán las anotaciones para registrar el estado del despliegue, así como la información necesaria para volver al estado anterior (*rollback*) en caso de fallo en el despliegue.

### Definiendo anotaciones

El uso de prefijos en la definición de las claves es recomendable para evitar que diferentes herramientas sobreescriban valores debido a un conflicto en el uso de claves. El formato del prefijo será un subdominio DNS seguido de una barra. Las anotaciones de un objeto las definiremos en la sección ***metadata.annotations*** del fichero YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-example
  annotations:
    imageregistry: "https://hub.docker.com/"
    example.com/icon-url: "https://example.com/icon.png"
...
```

También podemos añadir anotaciones de manera imperativa a objetos existentes mediante el comando kubectl annotate. Además, si consultamos los detalles del objeto mediante el comando kubectl describe veremos las anotaciones asociadas al mismo.

```
$ kubectl annotate pod mi-pod example.com/icon="icon.png"
pod "mi-pod" annotated

$ kubectl describe pod mi-pod
...
Annotations:    example.com/icon=icon.png
```

## Agrupación de recursos con namespaces

Los **namespaces** son una abstracción que permite dividir un clúster de Kubernetes en múltiples clústeres virtuales. Por ejemplo, podemos hacer que cada equipo que utilice el clúster tenga su propio namespace con una cuota de recursos asociada. Este es el principal mecanismo que nos ofrece Kubernetes para establecer el alcance y limitar el acceso.

La utilización de los namespaces nos va a permitir organizar nuestros objetos en el clúster, dividiendo sistemas complejos con múltiples componentes en grupos más pequeños. Podemos imaginarnos los namespaces como una carpeta que contiene un conjunto de objetos de nuestro clúster.

Cuando creamos un clúster de Kubernetes habitualmente este comienza con tres namespaces por defecto (aunque dependiendo de la instalación podría tener inicialmente alguno más):

- **Default**: Será el utilizado cuando no especifiquemos un namespace.
- **Kube-system**: Es el namespace para los objetos que han sido creados por el propio sistema de Kubernetes. Los usuarios no deberían desplegar aplicaciones en él.
- **Kube-public**: Visible y accesible por todos los usuarios del clúster. Aunque normalmente se utiliza de manera interna al clúster; puede ser utilizado para que algunos recursos sean visibles en todo el clúster y accesible por cualquier usuario.

Cuando hemos utilizado el comando kubectl en los apartados anteriores, este lo ha hecho operando sobre los objetos del namespace default. El siguiente comando obtiene la lista de namespaces definidos en nuestro clúster:

```
$ kubectl get ns
NAME              STATUS   AGE
default           Active   12d
docker            Active   12d
kube-node-lease   Active   12d
kube-public       Active   12d
kube-system       Active   12d
```

Si quisiéramos utilizar un namespace específico cuando ejecutamos un comando con kubectl, podríamos hacerlo utilizando la opción **'--namespace'**. El siguiente ejemplo lista los Pods creados en el namespace del sistema:

```
$ kubectl get pods --namespace kube-system
NAME                                     READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-4n69f                 1/1     Running   0          12d
coredns-5644d7b6d9-v2kkx                 1/1     Running   0          12d
etcd-docker-desktop                      1/1     Running   0          12d
kube-apiserver-docker-desktop            1/1     Running   0          12d
kube-controller-manager-docker-desktop   1/1     Running   0          12d
kube-proxy-j6cll                         1/1     Running   0          12d
kube-scheduler-docker-desktop            1/1     Running   0          12d
storage-provisioner                      1/1     Running   0          12d
vpnkit-controller                        1/1     Running   0          12d
```

En caso de querer interactuar con los objetos de todos los namespaces definidos podemos utilizar la opción '--all-namespaces'.

```
$ kubectl get pods --all-namespaces
```

El siguiente comando crearía un namespace de manera imperativa indicando el nombre que queremos utilizar:

```
$ kubectl create namespace custom-namespace
namespace "custom-namespace" created
```

En la forma declarativa, la definición del namespace es muy sencilla. Simplemente tendremos que especificar el nombre que queremos darle:

```
$ kubectl create -f - <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
EOF

namespace "custom-namespace" created
```

Aunque como acabamos de ver, los namespaces nos permiten operar sobre grupos de objetos, por defecto **no se proporciona ningún tipo de aislamiento** entre los objetos de distintos namespaces ejecutándose en el clúster. Por ejemplo, dos Pods de diferente namespace podrían comunicarse entre sí directamente mediante la IP privada.

Para poder controlar el tráfico de red y aislar los namespaces, debemos implementar alguna política de red de Kubernetes. Para más detalles, consulta el enlace siguiente: Prácticas recomendadas utilizando namespaces. Y échale un vistazo a este vídeo.

unir LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA