





Hola!

Soy Pedro Plasencia

Full Stack Web Developer desde 2014

@pedrovelasquez9 – pedro@impactotecnologico.net



Docker





docker



Máquinas virtuales vs Contenedores

Antes de comparar es importante poder entender el concepto de virtualización.

La virtualización consiste en añadir una capa de abstracción a los recursos físicos con el objetivo de mejorar el uso de los recursos del sistema. Existen tres tipos de virtualización:

- Completa
- Asistida por hardware
- A nivel de OS.



Tipos de virtualización

- **Completa:** La maquina virtual no tiene acceso directo a los recursos físicos y requiere una capa superior para acceder a ellos. Algunos ejemplos son VirtualBox, QEMU, Hyper-V, VMware ESXi.
- **Asistida por hardware:** Es el hardware el que facilita la creación de la máquina virtual y controla su estado. Algunos ejemplos son KVM, Xen, VMWare fusion.

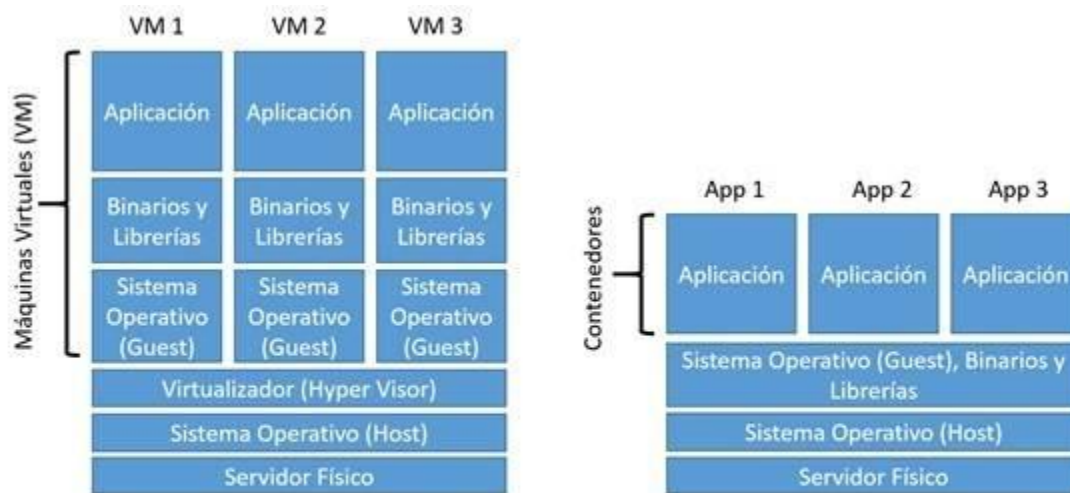


Tipos de virtualización

- **A nivel de sistema operativo:** El encargado de aislar los recursos y proporcionar las herramientas necesarias para crear, manipular o controlar el estado de los contenedores es el sistema operativo y no el hardware. En este grupo entran los contenedores Linux y Docker.



Máquinas virtuales vs Contenedores



Máquinas Virtuales versus Contenedores



Criterios de comparación

- **Rapidez:** Docker y sus contenedores son capaces de compartir un solo núcleo y compartir bibliotecas de aplicaciones, esto ayuda a que los contenedores presenten una carga más baja de sistema que las máquinas virtuales.

Una máquina virtual puede tardar hasta varios minutos para crearse y poner en marcha mientras que un contenedor puede ser creado y lanzado sólo en unos pocos segundos.



Criterios de comparación

- **Portabilidad:** Los contenedores Docker aportarán numerosos beneficios en comparación con las máquinas virtuales. En términos de tecnología, es bastante interesante en escenarios donde ayuda en la promoción de la portabilidad de la nube mediante la ejecución de las mismas aplicaciones en diferentes entornos virtuales esto es muy útil en el ciclo de vida para el desarrollo de software



Criterios de comparación

- **Seguridad:** Una de las ventajas de la utilización de máquinas virtuales es la abstracción a nivel de hardware físico que se traduce en kernels individuales, que limitan la superficie de ataque al hipervisor (el monitor o núcleo de la virtualización). En teoría, las vulnerabilidades particulares en las versiones de sistemas operativos no se pueden aprovechar para poner en peligro otras máquinas virtuales que se ejecutan en la misma máquina física.



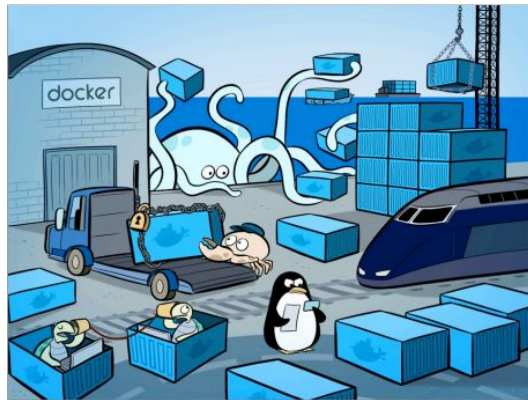
Criterios de comparación

- **Administración:** Soluciones tales como Docker hacen más fácil la gestión de contenedores, pero muchos clientes todavía encuentran la gestión de contenedores más un arte que una ciencia. Para varios usuarios que han estado trabajando con Docker recientemente y comparten su experiencia y su frustración de la gestión del Docker en un entorno de producción.



¿Qué es Docker?

Es una plataforma que se encarga de crear y administrar contenedores ligeros y portables para aplicaciones de software, que pueden ejecutarse en cualquier máquina que disponga de Docker.



Estos contenedores disponen de todo lo mínimo necesario para la ejecución adecuada de la aplicación alojada en ellos.



Introducción a Docker

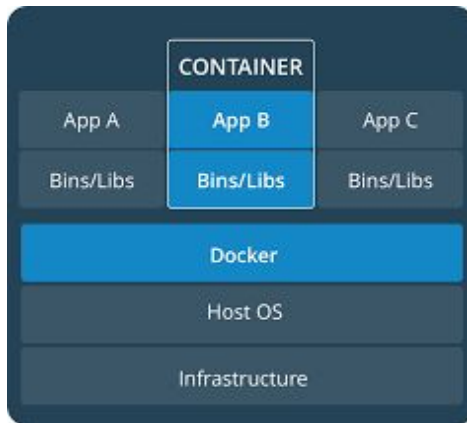
Las palabras claves que definen a Docker son las develop (desarrollar), ship (transportar), y run (ejecutarse).



Docker facilita el desarrollo de una aplicación, que puede ser transportada como un contenedor y ejecutada donde sea.



Docker permite la virtualización de aplicaciones a través de contenedores que pueden ejecutarse de forma independiente desde una misma instancia Linux.

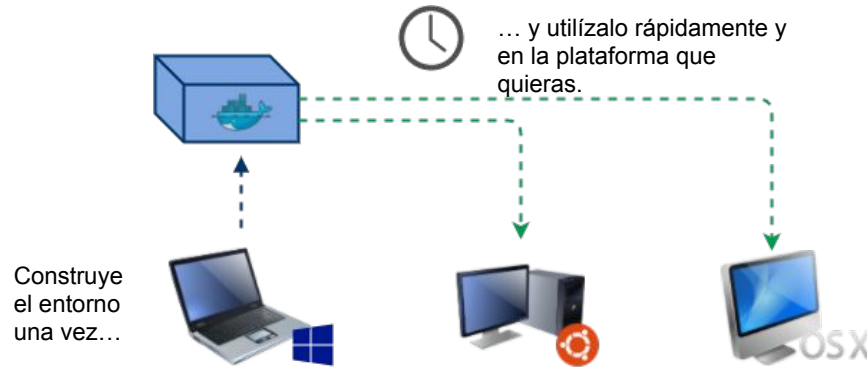


Los namespace aíslan el entorno operativo (filesystem, árbol de procesos, etc) de cada contenedor, y los cgroups los recursos (CPU, memoria, bloqueos I/O de red, etc).



Características de Docker

- Los contenedores pueden desplegarse con facilidad en cualquier otro sistema (que soporte Docker), ahorrándonos el tiempo de instalación.



Características de Docker

- Los equipos de diferentes unidades como: desarrollo, QA y operaciones, pueden trabajar sin problemas en todas las aplicaciones.
- Los contenedores se pueden implementar en cualquier entorno físico, virtual o en la nube.
- Los contenedores Docker son ligeros y fácilmente escalables.

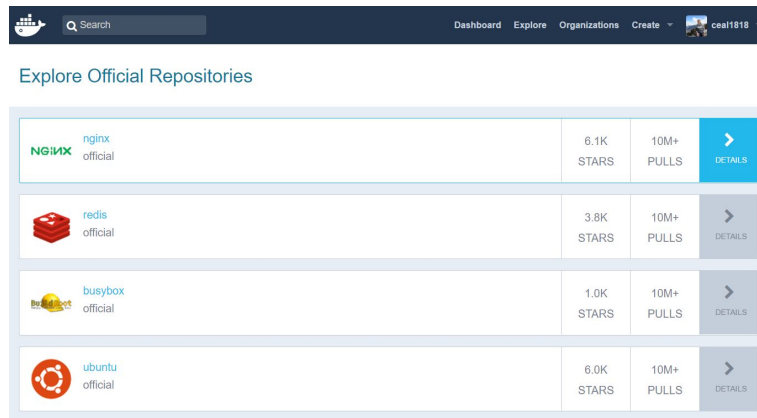


Introducción a Docker

Componentes de Docker

Docker está compuesta por los siguientes elementos:

- ◆ **Docker Registry:** es un servicio de registro de imágenes Docker (público o privado), donde podemos obtener o cargar diferentes imágenes. El registro público de Docker se le conoce como: **Docker Hub** (<https://hub.docker.com/>).



nginx official		6.1K STARS	10M+ PULLS	> DETAILS
redis official		3.8K STARS	10M+ PULLS	> DETAILS
busybox official		1.0K STARS	10M+ PULLS	> DETAILS
ubuntu official		6.0K STARS	10M+ PULLS	> DETAILS

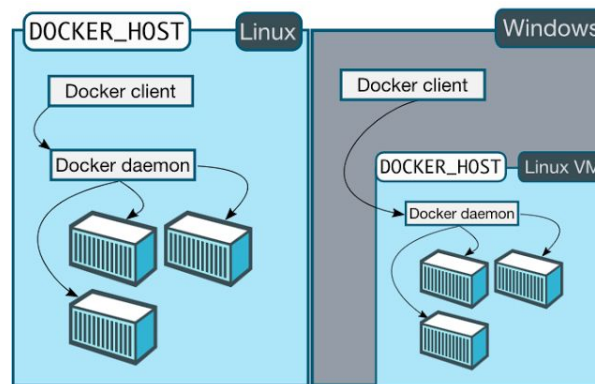


Componentes de Docker

- ◆ **Docker Engine:** es la pieza fundamental de Docker que se encarga de la orquestación de las imágenes y los contenedores, el balanceo de carga entre los contenedores, entre muchas otras cosas.

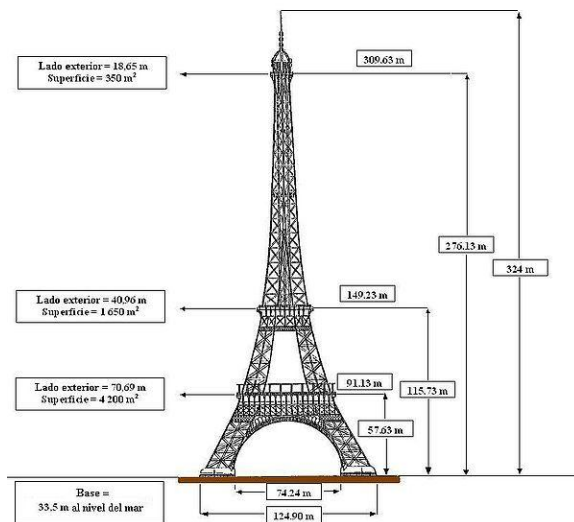
El Docker Engine está compuesto por:

- ◆ Docker daemon.
- ◆ Docker Host (API).
- ◆ Docker CLI.



Componentes de Docker

- ◆ **Docker Image:** es un componente estático de Docker que solo contiene un sistema operativo base y un conjunto de componentes que aportarán la funcionalidad específica de la misma.



Componentes de Docker

- ◆ **Docker Container:** es la instancia de una imagen Docker que presta el servicio para el cual fue diseñada.



Ventajas de Docker

- ◆ **Saca el mayor partido al sistema:** Docker se apoya en herramientas nativas del Kernel de Linux para aislar y securizar un contenedor. Así puede ejecutar N cantidad de contenedores, aprovechando los recursos del sistema anfitrión sin penalizar su rendimiento.
- ◆ **Construye uno, utilízalo donde quieras:** Los contenedores Docker son independientes al sistema donde se construyen. Por ello podemos compartirlos fácilmente solo con pasar el código que los genera.



Ventajas de Docker

- ◆ **Seguridad:** Los contenedores Docker son componentes aislados que no pueden acceder a otros contenedores y tampoco al sistema anfitrión. Esto garantiza que Docker no permitirá que un contenedor afecte a otros contenedores o al sistema anfitrión.
- ◆ **Despliegues en segundos:** Los contenedores Docker no son máquinas virtuales, por tanto no requieren de la ejecución de piezas de hardware o hipervisores para su ejecución. Por este motivo los contenedores tardan muchos menos en arrancar.



Ventajas de Docker

- ◆ **Muchas imágenes disponibles:** Docker dispone de varias imágenes que pueden descargarse y modificarse libremente, para construir un contenedor adecuado a las necesidades del proyecto.



Instalación de Docker Engine

- ◆ **Instalación de Docker Engine para equipos de desarrollo:** Recomendamos que sigan los pasos indicados para cada sistema operativo en esta página <https://www.docker.com/community-edition#/overview>

Nota: Para Windows 10, estas instrucciones solo aplican si la versión de sistema operativo es superior a Home y el equipo es 64bits.

- ◆ **Instalación alternativa de Docker Engine en PC con Windows 10 Home:** Es necesario que el equipo tenga el soporte de virtualización habilitado, Virtual Box instalado, y Git instalado con Git Bash.

Luego es necesario descargar Docker Toolbox desde esta página <https://www.docker.com/products/docker-toolbox>



Antes de crear un container es necesario aprender a interactuar y administrar con las imágenes. Conozcamos los comandos básicos que nos ayudarán a lograrlo.

Para empezar a trabajar con Docker CLI debemos conocer el modo de trabajo del comando.

```
docker [OPTIONS] COMMAND [ARG...]
```

- Listar imágenes disponible en nuestro registro privado

```
$ docker images
```

- Listar imagen de centos disponible en nuestro registro privado

```
$ docker images centos
```



Creación de Containers

- Descargar imagen de centos

```
$ docker pull centos
```

- Eliminar imagen de centos

```
$ docker rmi centos
```

- Listar los contenedores en ejecución

```
$ docker ps
```

- Arrancar contenedor ubuntu solicitando interacción con bash

```
$ docker run -it ubuntu /bin/bash
```



- Arrancar contenedor Apache publicando en el host su servicio por el puerto 8000.

```
$ docker run -itd -p 8000:80 --name svr1 httpd
```

- Arrancar contenedor Apache exponiendo su servicio por el puerto 8000.

```
$ docker run -itd --expose 8000 --name svr2 httpd
```



En cuanto hayamos creado nuestro primer contenedor, debemos ser capaces de controlarlo. Para ello es necesario conocer los comandos que nos permiten gestionar los container en Docker.

Los comandos que debemos conocer son los siguientes:

◆ Arrancar un contenedor detenido

```
$ docker start 145637e76dcb
```

```
$ docker container start 145637e76dcb
```

◆ Parar contenedor en ejecución

```
$ docker stop 145637e76dcb
```

```
$ docker container stop 145637e76dcb
```



Control de Containers en Docker

- ◆ Detener abruptamente un contenedor en ejecución

```
$ docker kill 145637e76dcb
```

```
$ docker container kill 145637e76dcb
```

- ◆ Reiniciar contenedor en ejecución

```
$ docker restart 145637e76dcb
```

```
$ docker container restart 145637e76dcb
```

- ◆ Obtener información de bajo nivel del contenedor

```
$ docker inspect 145637e76dcb
```

- ◆ Listar los logs del contenedor

```
$ docker logs 145637e76dcb
```



Control de Containers en Docker

- ◆ Muestra estadísticas de uso de los recursos del contenedor

```
$ docker stats 145637e76dcb
```

```
$ docker container stats 145637e76dcb
```

- ◆ Muestra los procesos en ejecución del contenedor

```
$ docker top 145637e76dcb
```

```
$ docker container top 145637e76dcb
```



◆ `docker run -d -p 33061:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=secret mysql`

- ◆ **-d**: Deattached Mode es la forma en que indicamos que corra en background.
- ◆ **-p** : puerto, el contenedor corre en el puerto 3306 pero hacemos un bind para que lo escuchemos en Host el puerto 33061.
- ◆ **--name** : para no tener que hacer referencia al hash le asignamos un nombre.
- ◆ **-e** : environment le asignamos la contraseña.



◆ Entrar al contenedor

◆ `docker exec -it mysql-db mysql -p`

◆ **exec**: indicamos que vamos a pasar un comando.

◆ **-it** Modo interactivo.

◆ **mysql -p**: es el comando para entrar a la consola de mysql con el usuario root



Construyendo y administración de DockerFiles

Es un script que simplifica y automatiza el proceso de aprovisionamiento de una imagen Docker, partiendo de un conjunto de instrucciones sencillas de comprender.

El aprovisionamiento a través de un fichero DockerFile parte de una imagen base, y forma una nueva con las nuevas características indicadas en el DockerFile.

Para trabajar construir una imagen partiendo de un DockerFile, es necesario ejecutar el siguiente comando en el directorio donde se encuentra el fichero DockerFile.

```
$ docker build .
```



Construyendo y administración de DockerFiles

Sintaxis del fichero DockerFile

La sintaxis básica del fichero es el siguiente:

```
# Line blocks used for commenting  
command argument argument
```

Este sería un ejemplo sencillo con DockerFile

```
# DockerFile simple example  
RUN echo "Hello world!!!"
```



Sintaxis del fichero DockerFile

- **FROM:** a través de esta instrucción definimos la imagen base.

```
# Base image is ubuntu:latest  
FROM ubuntu
```

- **LABEL:** con esta instrucción se puede declarar etiquetas informativas en la imagen.

```
# Author of this image  
LABEL maintainer=josej@gmail.com
```

- **COPY:** permite copiar los ficheros o directorios desde una fuente (contexto de construcción) a un destino.

```
# Copy files in img/ to ing/  
COPY img/ img/
```



Sintaxis del fichero DockerFile

- **ADD:** permite copiar ficheros o directorios (contexto de construcción), o recursos remotos desde una fuente a un destino.

```
# Add files in css/ to css/  
ADD css/ css/
```

- **ENV:** permite declarar variables de entornos en el sistema de la imagen.

```
# Usage: ENV key value  
ENV SERVER_WORKS 4
```

- **USER:** permite definir el usuario que estará activo en la construcción de la imagen y la instanciación de la misma.

```
# Set root like active user  
USER root
```



Construyendo y administración de DockerFiles

Sintaxis del fichero DockerFile

- **WORKDIR:** establece el directorio de trabajo dentro de la imagen en tiempo de construcción e instanciación de la imagen.

```
# Set work directory /var/usr/httpd  
WORKDIR /var/usr/httpd
```

- **RUN:** permite ejecutar comandos durante la construcción de la imagen.

```
# Run update apt-get list and install mongodb-org  
RUN apt-get update && apt-get install -y mongodb-org
```



Construyendo y administración de DockerFiles

Sintaxis del fichero DockerFile

- **CMD:** permite ejecutar comandos en el contenedor en cuanto se ha instanciado.

```
# Execution echo 'hello world!' when the container is instantiate.  
CMD ["echo", "'hello world!'"]
```

- **ENTRYPOINT:** permite configurar un contenedor para que sea ejecutable. CMD puede suministrarle parámetros.

```
# As soon as the container is instantiated, it will execute this command  
ENTRYPOINT ["top", "-b"]
```



```
FROM alpine:3.4
```

```
RUN apk update
```

```
RUN apk add vim
```

```
RUN apk add curl
```

```
RUN apk add git
```

Construimos la imagen:

```
$ docker build -t my-basic-app .
```

Arrancamos la imagen:

```
$ docker run -it --rm --name my-running-app my-basic-app
```



FROM ubuntu

MAINTAINER Pedro Plasencia

RUN apt-get update

RUN apt-get install -y apache2

RUN echo "<h1>Apache desde Docker</h1>" > /var/www/html/index.html

EXPOSE 80

ENTRYPOINT apache2ctl -D FOREGROUND

◆ Crear imagen y contenedor



Docker Volume

Los contenedores en Docker son capaces de trabajar con un almacenamiento alternativo, que facilite la gestión de los datos que requiera o genere el servicio.

Dentro de Docker se puede manejar datos de la siguiente forma:

- En volúmenes
- En contenedores como volúmenes.



Volumen de datos

- Se pueden utilizar para guardar e intercambiar información independiente de la vida del contenedor.
- Facilitan el intercambio de datos entre contenedores.
- Los volúmenes son directorios en el host montados en el contenedor como directorios.
- En el caso de montar en un directorio ya existente de un contenedor un volumen de datos , su contenido no será eliminado.



Administración de Docker Volumes y Networking

- ◆ Los volúmenes de docker son una forma especial de crear espacios para persistencia de archivos
- ◆ La principal diferencia es que estos volúmenes pueden usar una serie de drivers que permiten la integración de un volumen con algún tipo específico de filesystem como por ejemplo Azure Filesystem, S3 de AWS, IPFS entre algunos.
- ◆ El driver por defecto es el local, que permite el uso del propio filesystem (ninguna novedad) como volumen.



Administración de Docker Volumes y Networking

- ◆ Cada volumen es identificado con una etiqueta y no es necesario especificar un path en el SO host, ya que este volumen tiene una zona donde se crean todos los volúmenes dentro del filesystem del host.
- ◆ Una vez creado un volumen, puede ser montado dentro de un contenedor asociándolo a un directorio dentro del contenedor:
- ◆ `docker run -d -v data:/var/lib/mysql -p 3306:3306 mysql`



Ejercicio guiado

- ◆ Eliminamos el proceso que corre el contenedor **creado** para MySQL
 - ◆ docker **rm** -f mysql-**db**
- ◆ Eliminamos todos los volúmenes ya que Docker crea volúmenes temporales sin pedir permiso
 - ◆ docker **volume** prune
- ◆ Creamos un volumen
 - ◆ docker volume **create** mysql-db-data
- ◆ Verificamos que se haya creado
 - ◆ docker volume ls



- ◆ Levantamos nuevamente el Docker y agregamos el volumen con la opción `--mount`
 - ◆ `docker run -d -p 33060:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=secret --mount src=mysql-db-data,dst=/var/lib/mysql mysql`
- ◆ Entramos al contenedor
 - ◆ `docker exec -it mysql-db mysql -p`
- ◆ Creamos una BD
 - ◆ `create database demo;`
 - ◆ `show databases;`
- ◆ Terminamos el proceso
 - ◆ `docker rm -f mysql-db`



- ◆ Arrancamos nuevamente el contenedor con el mismo volumen
 - ◆ `docker run -d -p 33060:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=secret --mount src=mysql-db-data,dst=/var/lib/mysql mysql`
- ◆ Entramos nuevamente al contenedor y verificamos la base de datos
 - ◆ `docker exec -it mysql-db mysql -p`
- ◆ Verificamos que existe la BD
 - ◆ `show databases;`
- ◆ Inspeccionemos el volumen
 - ◆ `docker volume inspect mysql-db-data`



Trabajando con volúmenes

- ◆ **rm:** Eliminar un volumen de datos específico.

```
$ docker volume rm [VOLUME]
```

Al arrancar un contenedor podemos asociarle un volumen a través del parámetro `-v`. El valor que recibe el parámetro es el nombre del volumen y el nombre del directorio en el contenedor separado por dos puntos.

```
$ docker run -it --name contenedor2 -v vol1:/data  
ubuntu bash
```



Trabajando con volúmenes

Desde DockerFile

Entre las instrucciones reservadas se encuentra VOLUME. Esta instrucción crea un punto de montaje asociado a un directorio dentro del contenedor.

```
VOLUME [Container directory]
```



```
FROM alpine  
VOLUME ["/data"]  
ENTRYPOINT ["/bin/sh"]
```

```
docker image build -t img1 .  
docker container run --name c2 -ti img1
```

Entramos al contenedor y:

```
cd /data  
touch hello.txt  
ls
```

Desde otra consola inspeccionamos el contenedor y volumen



Networking

Docker permite crear redes que faciliten la comunicación de los contenedores con el host, la comunicación entre contenedores, y la comunicación entre los nodos de un clúster Docker.

En Docker existen los siguientes tipos de redes:

- **host**: es la red del propio equipo y suele hacer referencia a eth0
- **bridge**: es una red puente que facilita la comunicación entre contenedores y el host. Por defecto, Docker crea la red bridge docker0 y a ella se conectan todos los contenedores.
- **none**: representa la no conexión a ninguna interfaz de red.
- **overlay**: es una red que sirve para comunicar diferentes nodos de un cluster Docker, y para ello requiere de un servicio de almacenamiento Key-Value (Zoopeker, etc)



Networking

Comandos básicos

Para trabajar con redes en Docker es necesario conocer los comandos que tenemos a nuestra disposición a través de **docker network**.

- ◆ **ls**: permite listar las redes disponibles en el demonio docker.

```
$ docker network ls
```

- ◆ **create**: permite crear una nueva red indicando el driver.

```
$ docker network create --driver bridge my-network
```



Networking

Comandos básicos

- ◆ **connect/disconnect:** A través de estos comandos podemos conectar o desconectar una red y un contenedor en ejecución.

```
$ docker network connect [NETWORK] [CONTAINER]
```

```
$ docker network disconnect [NETWORK] [CONTAINER]
```

- ◆ **inspect:** Muestra los detalles de una o más redes.

```
$ docker network inspect [NETWORK]
```

- ◆ **rm:** Elimina la red indicada.

```
$ docker network rm [NETWORK]
```



Networking

Conectar una red a un contenedor desde el arranque

Para que conectemos un contenedor a una red desde el mismo arranque del contenedor, es necesario que ejecutemos el siguiente comando:

```
$ docker run --net=my-network -itd --name=container3 ubuntu
```



◆ Una aplicación por contenedor

- ◆ Es bastante común pensar en contenedores como si de una máquina virtual se tratase. La idea es que los contenedores, por un lado, tengan el mismo ciclo de vida de una aplicación y, por otro lado, que estos sean efímeros (podamos ejecutar o destruir uno en cualquier momento, así como, levantar varias instancias de un mismo contenedor).



Entender el contexto de construcción

Cuando se ejecuta el comando *docker build* para construir una imagen a partir de un *Dockerfile* el directorio donde se encuentra dicho fichero se conoce como contexto de construcción. Es este directorio el que se envía como contexto de construcción al demonio de Docker, es decir, los ficheros y directorios dentro de él.

Esto provoca que tener ficheros en dicho directorio que no son necesarios para la construcción de la imagen provocará contexto de construcción más grandes y, en consecuencia, imágenes de mayor tamaño. También, resulta en un tiempo de construcción mayor y en un contenedor (una vez ejecutado) con mayor uso de memoria.



Entender el contexto de construcción

- ◆ En este caso, podemos aislar dentro de nuestro proyecto el fichero *Dockerfile* dentro de un subdirectorio que contendrá sólo aquello necesario. O bien, utilizar *.dockerignore* para indicar al demonio de Docker que no tenga en cuenta aquello que no necesitamos



Optimización de la caché de construcción de Docker

- ◆ Docker utiliza una memoria caché con la idea de agilizar la construcción de imágenes. Como ya se ha comentado previamente, las imágenes son construidas por capas, cada instrucción dentro de un fichero *Dockerfile* resulta en una capa de la imagen.
- ◆ Durante la construcción, siempre que se pueda, Docker tiende a reutilizar las capas de una imagen de una construcción anterior, obviando un paso que podría resultar costoso.



Optimización de la caché de construcción de Docker

- ◆ Agrupar instrucciones en una misma capa (instrucción del fichero *Dockerfile*). Esto es por diversos motivos, desde reutilizar la capa, hasta mantener el mismo contexto, ya que puede ser necesario mantener un contexto entre dos capas. Para evitar esto, el ejemplo claro de esto es el comando *apt* o *yum* (instalador de paquetes), que normalmente requiere de una actualización de repositorios y de paquetes previos. A continuación, se muestra con un ejemplo:

```
FROM debian:9
```

```
RUN apt-get update
```

```
RUN apt-get install -y nginx
```



```
FROM debian:9
```

```
RUN apt-get update && \
```

```
apt-get install -y nginx
```



Eliminar herramientas innecesarias

- ◆ Dejar el entorno sucio o con elementos innecesarios no es recomendable y, además, puede suponer un problema de seguridad. Por ejemplo, si durante la construcción se necesitan herramientas como *unzip*, *netcat* o *wget*, es recomendable dejar el paso inverso. No debemos olvidar seguir recomendaciones como la anterior, hacerlo en la propia instrucción *RUN*. Por ejemplo:

```
FROM debian:9
```

```
RUN apt-get update && \  
    apt-get install -y netcat && \  
DO SOMETHING && \  
    apt-get remove -y netcat && \  

```



Iniciar contenedores en modo sólo lectura

- ◆ Es una buena práctica iniciar el contenedor en modo *sólo lectura*. Para ello, utilizar junto al comando *docker run* con el flag *read-only*:
 - ◆ `docker run -d --read-only nginx`
- ◆ Si el contenedor necesita escribir en el sistema de ficheros, se puede proveer un volumen para evitar errores y, además, hacer persistente los cambios una vez muera el contenedor.



Utilizar imágenes bases reducidas

- ◆ En los ficheros *Dockerfile* se parte de una base a través de la instrucción *FROM*, la cual se encuentra al principio del fichero. El resto de instrucciones dependen de dicha instrucción (se puede partir de una imagen vacía).
- ◆ Por ejemplo, si partimos de una imagen tipo CentOS el gestor de paquetes a utilizar en la instrucción *RUN* será *yum* en lugar de *apt-get*. Esto es porque en esa imagen base de la que partimos no tenemos disponible *apt-get* como gestor de paquetes.



Utilizar imágenes bases reducidas

- ◆ Por lo tanto, una imagen puede ser más ligera que otra.
- ◆ Por ejemplo, si necesitamos *centos* como imagen base, debemos saber que existe una alternativa más ligera como *alpine*, en este caso, pesa 71 MB menos.



No utilices la etiqueta de imagen base latest

- ◆ La etiqueta latest es la que se utiliza por defecto, cuando no se especifica ninguna otra etiqueta.
- ◆ Así que nuestra instrucción FROM ubuntu en realidad hace exactamente lo mismo que FROM ubuntu:latest.
- ◆ Pero la etiqueta latest apuntará a una imagen diferente cuando se publique una nueva versión, y el *build* puede romperse.
- ◆ Por lo tanto, a menos que estemos creando un Dockerfile genérico que tenga que estar actualizado con la imagen base, debemos proporcionar una etiqueta específica:
 - ◆ FROM ubuntu:16.04



◆ Crear un contenedor Docker que:

- ◆ Parta de Tomcat (libre elección de versión)
- ◆ Que copie **de local** el war demo: <https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war>
- ◆ Y que arranque Tomcat cuando se inicie el contenedor y que se quede en ejecución
- ◆ El contenedor tiene que quedarse ejecutando en background
- ◆ Probar ingreso desde el navegador

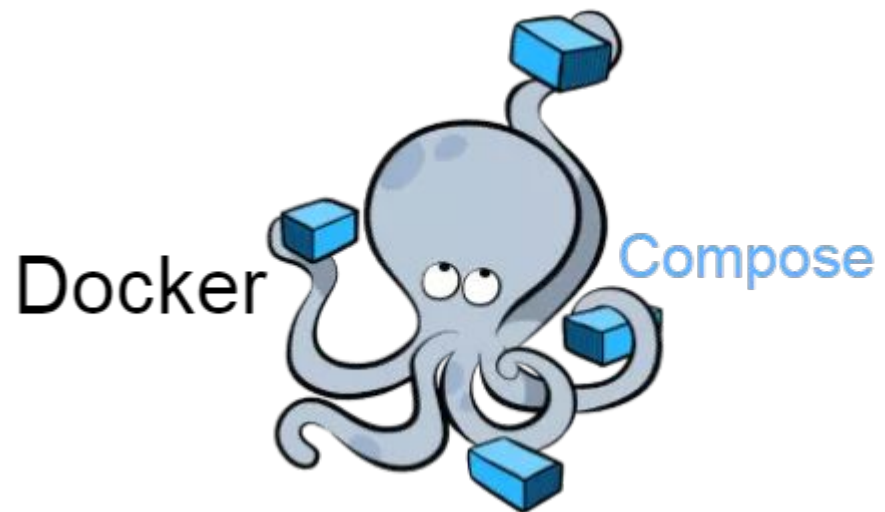


Instalando Wordpress con contenedores



Trabajando con volúmenes





Introducción a Docker Compose

- ◆ Las aplicaciones basadas en microservicios se prestan a usar múltiples contenedores cada uno con un servicio, uno puede contener la base de datos postgresql, otro una base de datos clave/valor redis o de documentos como elasticsearch para hacer búsquedas, otro un sistema de mensajería como rabbitmq, otro tomcat o wildfly que use los anteriores y un servidor web como Nginx.
- ◆ Teniendo múltiples contenedores usar el comando docker run para cada uno de ellos nos resultará incómodo.



Introducción a Docker Compose

- ◆ En este punto entra **Docker Compose** permitiéndonos definir nuestra aplicación multicontenedor en un archivo con las mismas propiedades que indicaríamos con el comando `docker run` individualmente.
- ◆ Con un único comando podremos iniciar todos los contenedores y en el orden que los especifiquemos.



Instalación

◆ Docker compose se instala fácilmente en distribuciones linux populares:

◆ `sudo apt install docker-compose`

◆ `docker-compose --version`



El descriptor de contenedores

- ◆ Es un archivo de texto con formato yaml en la que especificamos los diferentes contenedores y sus propiedades, básicamente podemos indicar las mismas propiedades que indicamos arrancando los contenedores individualmente con el comando docker run.



Docker-compose.yml

```
redisdb:
  image: busybox
  volumes:
    - /var/lib/redis

postgresldb:
  image: busybox
  volumes:
    - /var/lib/postgresql/data

redis:
  image: redis:3
  volumes_from:
    - redisdb
  ports:
    - "6379:6379"
```



Docker-compose.yml

```
postgresql:
  image: postgres:9
  volumes_from:
    - postgresqldb
  environment:
    POSTGRES_USER: posgresql
    POSTGRES_PASSWORD: posgresql
  ports:
    - "5432:5432"

apps:
  image: library/tomcat:8-jre8
  links:
    - redis
    - postgresql
  ports:
    - "8080:8080"
```



Comandos comunes

- ◆ `docker-compose up -d`
- ◆ `docker-compose ps`
- ◆ `docker-compose stop`
- ◆ `docker-compose restart`
- ◆ `docker-compose logs`



- ◆ `docker-compose` inicia los contenedores en el orden que hemos indicado en el archivo de definición, las trazas emitidas de los servicios de los contenedores aparecerán en la terminal si iniciamos los contenedores en primer plano y con `Ctrl+C` se pararán los contenedores.
- ◆ Indicando la opción `-d` los contenedores se iniciarán en segundo plano, con `docker-compose stop` podremos pararlos, con `docker-compose restart` reiniciarlos, `docker-compose rm` para eliminar completamente los contenedores y con `docker-compose logs` veremos las trazas emitidas por los servicios que nos serán de utilidad si iniciamos los contenedores en segundo plano.



Integrar Docker + Jenkins + GitLab

Crear un docker-compose.yml con lo siguiente:

```
version: '3'
```

```
services:
```

```
  web:
```

```
    image: 'gitlab/gitlab-ce:latest'
```

```
    restart: always
```

```
    hostname: 'gitlab'
```

```
    environment:
```

```
      GITLAB_OMNIBUS_CONFIG: |
```

```
        external_url 'http://localhost:9091'
```



ports:

- '81:80'
- '4443:443'
- '24:24'
- '9091:9091'

networks:

- red-local

jenkins:

image: 'jenkins/jenkins:its'

restart: always

hostname: 'jenkins'



ports:

- '8081:8080'

- '2121:21'

networks:

- red-local

networks:

red-local:

driver: bridge





Subir una imagen a Docker Hub



Para subir una imagen propia y que esté disponible en el repositorio en línea de Docker (Docker Hub) podemos hacer lo siguiente:

- ◆ Primero que nada, debemos tener una cuenta en <https://hub.docker.com/> y verificar nuestra dirección de correo
- ◆ Una vez dentro de nuestra cuenta, podemos crear un repositorio nuevo
- ◆ Tras crearlo, podemos iniciar sesión desde nuestra consola:
- ◆ **\$ docker login --username=yourhubusername --email=youremail@company.com**
- ◆ Si todo ha ido bien, podemos proceder a agregar un tag a la imagen a subir: **docker tag bb38976d03cf yourhubusername/verse_gapminder:firsttry**



- ◆ Por último, ejecutamos el comando para subir la imagen en cuestión:
- ◆ **`docker push yourhubusername/verse_gapminder`**

Ya con esto nuestra imagen estará disponible en Docker Hub para uso global





Docker Machine



Docker Machine es una herramienta que nos ayuda a crear, configurar y manejar máquinas (virtuales o físicas) con Docker Engine. Con Docker Machine podemos iniciar, parar o reiniciar los nodos docker, actualizar el cliente o el demonio docker y configurar el cliente docker para acceder a los distintos Docker Engine. El propósito principal del uso de esta herramienta es la de crear máquinas con Docker Engine en sistemas remotos y centralizar su gestión.



Docker Machine utiliza distintos drivers que nos permiten crear y configurar Docker Engine en distintos entornos y proveedores, por ejemplo virtualbox, AWS, VMWare, OpenStack, etc.

Las tareas fundamentales que realiza Docker Machine, son las siguientes:

- ◆ Crea una máquina en el entorno que hayamos indicado (virtualbox, openstack...) donde va a instalar y configurar Docker Engine.
- ◆ Genera los certificados TLS para la comunicación segura.

También podemos utilizar un driver genérico (generic) que nos permite manejar máquinas que ya están creadas (físicas o virtuales) y configurarlas por SSH.



¿Por qué usar Docker Machine?

- ◆ Nos permite provisionar distintos hosts remotos con Docker
- ◆ Nos permite ejecutar y administrar comandos de Docker en máquinas que no lo soportan o no tienen los requisitos mínimos necesarios para ejecutarlo
- ◆ Entre los usos más destacados, figuran los siguientes casos de uso:
 - ◆ **Tengo una máquina antigua con mac o windows:** en este caso, podemos usar Docker de forma local a través de una Docker Machine en nuestra máquina aunque no cumpla con los requisitos de instalación de Docker



Docker Machine on Mac



Docker Machine on Windows



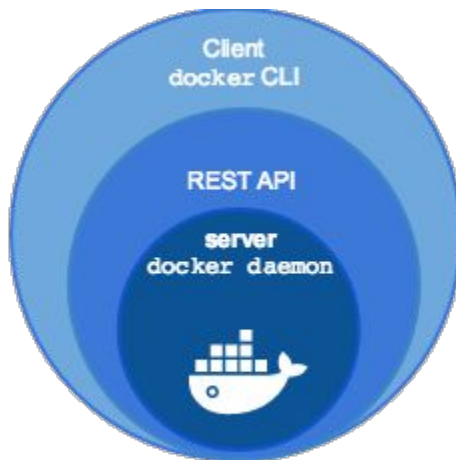
¿Por qué usar Docker Machine?

- ◆ **Quiero usar Docker en hosts remotos desde mi máquina local:** sin importar tu sistema operativo local, a través de Docker Machine, podremos crear y administrar instancias remotas de Docker en máquinas conectadas a una red, centralizando el control en nuestra máquina local



Docker Engine vs Docker Machine

Cuando hablamos de Docker Engine, nos referimos a la instalación “usual” de Docker: el DAEMON, el CLI y la API de Docker que nos permiten usar la herramienta de forma local para crear, configurar y administrar contenedores, imágenes y servicios dockerizados.



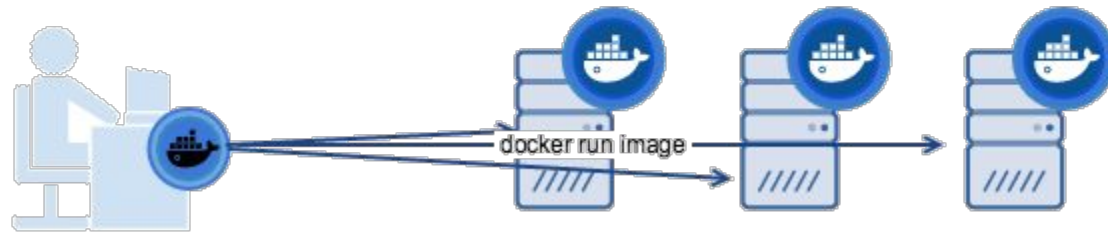
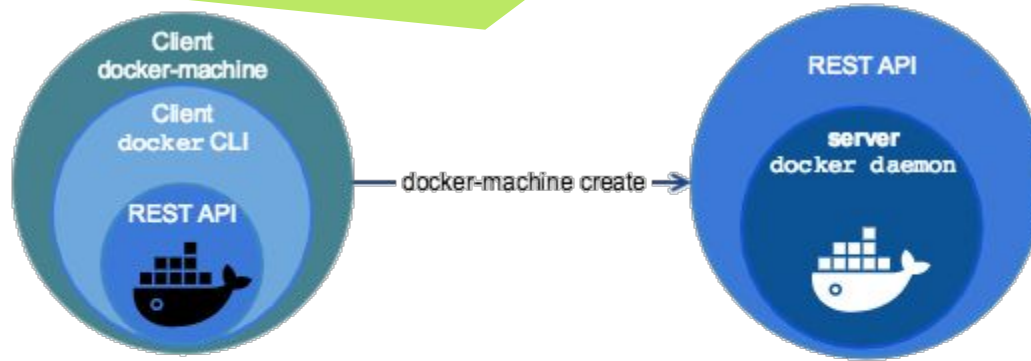
Docker Engine vs Docker Machine

Por otro lado, cuando hablamos de Docker Machine, nos referimos a una herramienta para administrar hosts remotos dockerizados (máquinas remotas con Docker Engine instalado).

Usualmente, instalamos Docker Machine en nuestra máquina local y podemos usar su CLI para instalar Docker Engine en máquinas remotas creadas con Docker Machine.

Estas máquinas virtuales pueden ser locales o remotas.





Instalación

<https://docs.docker.com/machine/install-machine/>



Práctica

Instalamos Docker-machine:

```
base=https://github.com/docker/machine/releases/download/v0.16.0 && curl -L  
$base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine && sudo install  
/tmp/docker-machine /usr/local/bin/docker-machine
```



Práctica

Comprobamos la instalación:

\$ docker-machine -version



Práctica

Vamos a utilizar el driver de VirtualBox que nos permitirá crear una máquina virtual con Docker Engine en un ordenador donde tengamos instalado VirtualBox. Para ello ejecutamos la siguiente instrucción:

```
$ docker-machine create -d virtualbox nodo1
```



Práctica

Esta instrucción va a crear una nueva máquina (nodo1) donde se va a instalar una distribución Linux llamada boot2docker con el Docker Engine instalado. Utilizando el driver de VirtualBox podemos indicar las características de la máquina que vamos a crear por medio de parámetros, por ejemplo podemos indicar las características hardware (--virtualbox-memory, --virtualbox-disk-size, ...)

Más parámetros: <https://docs.docker.com/machine/drivers/virtualbox/>



Práctica

Una vez creada la máquina podemos comprobar que lo tenemos gestionados por Docker Machine, para ello ejecutamos:

```
$ docker-machine ls
```



Práctica

A continuación para conectarnos desde nuestro cliente docker al Docker Engine de la nueva máquina necesitamos declarar las variables de entornos adecuadas, para obtener las variables de entorno de esta máquina podemos ejecutar:

```
$ docker-machine env nodo1
```



Práctica

Y para ejecutar estos comandos y que se creen las variables de entorno, ejecutamos:

```
$ eval $(docker-machine env nodo1)
```



Práctica

A partir de ahora, y utilizando el cliente docker, estaremos trabajando con el Docker Engine de nodo1:

```
$ docker run -d -p 80:5000 training/webapp python app.py
```



Práctica

Y para acceder al contenedor deberíamos utilizar la ip del servidor docker, que la podemos obtener:

```
$ docker-machine ip nodo1
```



Práctica

Otras opciones de docker-machine que podemos utilizar son:

- ◆ inspect: Nos devuelve información de una máquina.
- ◆ ssh, scp: Nos permite acceder por ssh y copiar ficheros a una determinada máquina.
- ◆ start, stop, restart, status: Podemos controlar una máquina.
- ◆ rm: Es la opción que borra una máquina de la base de datos de Docker Machine. Con determinados drivers también elimina la máquina.
- ◆ upgrade: Actualiza a la última versión de docker la máquina indicada.





Docker Swarm



A partir de la versión 1.12 de Docker es muy sencillo hacer un clúster formado por varios Docker Hosts mediante lo que se conoce como Swarm Mode.

Swarm es ahora parte integral del engine y no tendremos que instalar elementos adicionales.



Además, se han incluido nuevas funcionalidades como escalado, balanceo de carga y rolling updates, todo ello completamente integrado en la instalación de Docker. Veamos algunos conceptos básicos antes de ver Docker Swarm en acción.

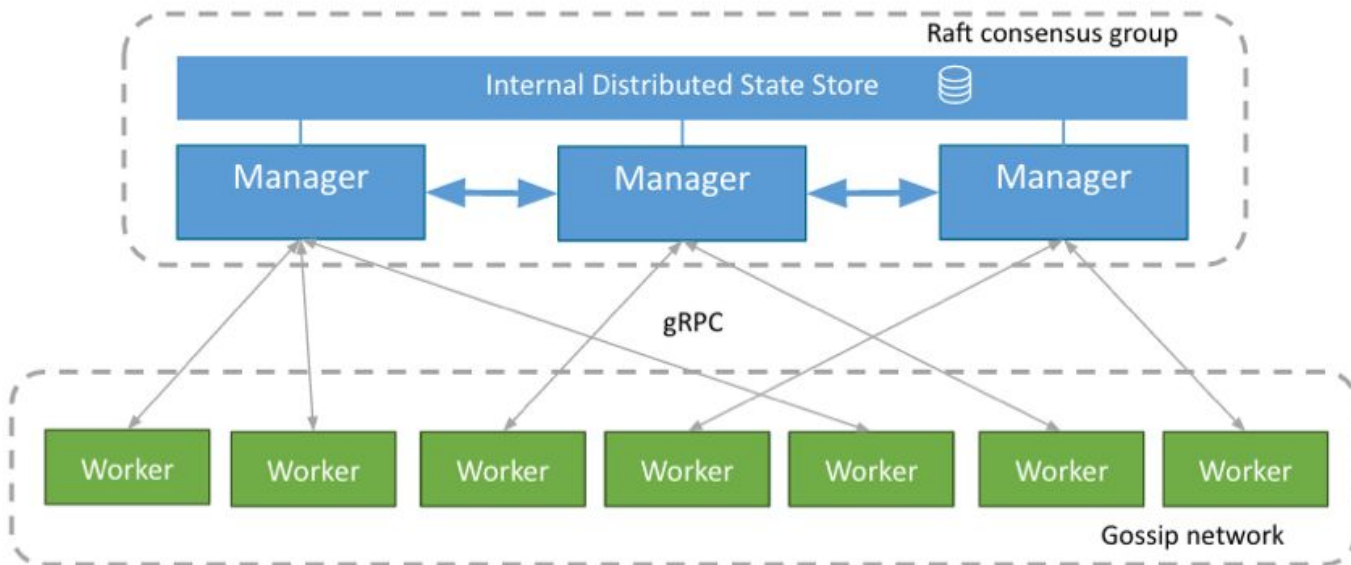
Nodos

En un clúster de Swarm existen dos tipos de nodo, Manager y Worker.

Los nodos Manager son los encargados de gestionar el clúster. Entre todos los Manager se elige automáticamente un líder y éste es el encargado de mantener el estado del clúster.



Los Manager son también los encargados de distribuir las tareas o tasks (unidades básicas de trabajo) entre todos los nodos Worker, los cuales reciben estas tareas y las ejecutan. Los nodos Manager por defecto también actúan como nodos Worker aunque se puede cambiar su configuración para que



Servicios y tareas

Un servicio define las tareas que serán ejecutadas dentro del clúster. Cuando creamos un servicio le indicamos a Swarm qué imagen y qué parametrización se utilizará para crear los contenedores que se ejecutarán después como tareas dentro del clúster.

Existen dos tipos de servicios, replicados y globales:

1. En un servicio replicado, Swarm creará una tarea por cada réplica que le indiquemos para después distribuirlas en el clúster. Por ejemplo, si creamos un servicio con 4 réplicas, Swarm creará 4 tareas.
2. En un servicio global, Swarm ejecutará una tarea en cada uno de los nodos del clúster.



Servicios y tareas

Como hemos dicho antes, las tareas son la unidad de trabajo dentro de Swarm. Realmente son la suma de un contenedor más el comando que ejecutaremos dentro de ese contenedor.

Los Manager asignan tareas a los nodos Worker de acuerdo al número de réplicas definidas por el servicio. Una vez que la tarea es asignada a un nodo ya no se puede mover a otro, tan sólo puede ejecutarse o morir.

Ante la caída de una tarea, Swarm es capaz de crear otra similar en ese u otro nodo para cumplir con el número de réplicas definido.



Balanceo

Swarm tiene un sistema de balanceo interno para exponer servicios hacia el exterior del clúster. Un Manager es capaz de publicar automáticamente un puerto generado al azar en el rango 30000-32767 para cada servicio, o bien, nosotros podemos publicar uno específico.

Cualquier sistema externo al clúster podrá acceder al servicio en este puerto publicado a través de cualquier nodo del clúster, independientemente de que ese nodo esté ejecutando una tarea del servicio o no.

Todos los nodos del clúster enrutarán a una tarea que esté ejecutando el servicio solicitado. Además, Swarm cuenta con un DNS interno que asigna automáticamente una entrada a cada uno de los servicios desplegados en el clúster.



Crear un clúster

Para crear un clúster con Swarm Mode tenemos que partir de un nodo destinado a ser Manager. Este nodo debe tener Docker 1.12 o superior ya instalado.

Suponiendo que la IP del nodo es 192.168.3.80, ejecutamos el siguiente comando:

```
$ docker swarm init --advertise-addr 192.168.3.80
```



Crear un clúster

Al ejecutar el comando hemos inicializado este nodo como Manager.

Con el parámetro obligatorio `--advertise-addr` le indicamos la IP del Manager que se utilizará internamente para las conexiones de Swarm. Si omitimos el puerto tomará el 2377 por defecto.

La salida del comando nos muestra dos tokens. Cada uno de ellos sirve para unir nodos Manager y Worker adicionales.



Crear un clúster

Añadiremos ahora un nodo Worker al clúster. Para ello y desde la consola del Worker ejecutamos:

```
$ docker swarm join --token
```

```
SWMTKN-1-50edlk935u9qgvrs8alhpzf1awgdil2dmfs4zgpd8ue2ltkmlww-3y9tb0pjnieqao9ahkutp  
vpxe 192.168.3.80:2377
```



Crear un clúster

Como podemos ver hemos utilizado el token para nodos Worker. Le indicamos también la IP y puerto de uno de los nodos Manager del clúster existente.

Si queremos añadir otro Manager el comando sería el mismo, pero usaríamos el otro token.



Gestión de servicios

Una vez tengamos el clúster preparado, podemos empezar a correr servicios sobre él. Para dar de alta un nuevo servicio nos vamos a la consola de un Manager y ejecutamos lo siguiente:

```
$ docker service create --replicas 1 --name helloworld alpine ping google.com
```

Donde `--name` es el nombre del servicio y `--replicas` es el número de tareas de este servicio que queremos crear.



Gestión de servicios

Ahora podemos ver un listado de todos los servicios en el clúster:

```
$ docker service ls
```

Y podemos ver información sobre el servicio:

```
$ docker service inspect --pretty helloworld
```



Gestión de servicios

Podemos ver todas las tareas que se están ejecutando para este servicio:

```
$ docker service ps helloworld
```

Si queremos aumentar o disminuir el número de réplicas ejecutamos:

```
$ docker service scale helloworld=5
```



Gestión de servicios

Y después comprobamos en qué nodos están corriendo las nuevas réplicas:

```
$ docker service ps helloworld
```



Ejemplo con Vagrant

Pre-requisitos:

- ◆ Vagrant
- ◆ Virtualbox
- ◆ Copiar y guardar de forma local el vagrantfile de este enlace:

<https://github.com/pedrovelasquez9/vagrant-docker/tree/master>



Ejemplo con Vagrant

Desde una terminal ubicada donde hemos guardado el archivo anterior ejecutar:

\$ vagrant up

Tardará un rato en crear y provisionar las 4 máquinas. Una vez terminado podrás ver las máquinas corriendo en la consola de VirtualBox.

El proceso de provisión dentro del Vagrantfile inicializa el clúster, por lo que la máquina manager será un nodo Manager y las máquinas worker01, worker02 y worker03 serán nodos Worker.



Ejemplo con Vagrant

De vuelta en la línea de comandos y ejecutamos lo siguiente:

```
$ vagrant ssh manager
```

Con este comando establecemos una conexión SSH con el nodo Manager. Desde aquí podemos lanzar comandos contra el clúster.

Por ejemplo, podemos ver todos los nodos:

```
$ docker node ls
```



Ejemplo con Vagrant

Crear un servicio nuevo:

```
$ docker service create --replicas 10 --name helloworld -p 8080:8080  
drhelius/helloworld-node-microservice
```

Y ver sus tareas:

```
$ docker service ps helloworld
```



Ejemplo con Vagrant

Para consumir este servicio podemos realizar una petición sobre cualquier nodo del clúster. Swarm enrutará nuestra petición a un nodo que tenga una tarea de ese servicio corriendo.

Por ejemplo, podemos probarlo apuntando al Manager y utilizando el puerto que expusimos en su creación, 8080:

```
$ curl localhost:8080
```



Ejemplo con Vagrant

Aunque esta prueba funcionaría desde cualquier nodo o desde el exterior del clúster si apuntamos a la IP de un nodo. En este momento podemos simular el fallo del nodo worker03 con 3 tareas corriendo.

Swarm tratará de mantener el estado dentro del clúster. Como se pierden 3 tareas, tendrá que ejecutar 3 nuevas en otros nodos. Para probarlo, terminamos la sesión SSH con el Manager, entramos en el worker03 y lo apagamos:

```
$ exit
```

```
$ vagrant ssh worker03
```

```
$ sudo shutdown -h now
```



Ejemplo con Vagrant

Entramos de nuevo en el Manager y vemos qué ha pasado:

```
$ docker service ps helloworld
```

Si en este momento arrancamos de nuevo el worker03 veremos que las tareas se quedan como están, ya que ahora no es necesario ningún cambio para cumplir con las 10 réplicas.





DockStation



Si bien Docker es sencillo de usar, hay usuarios que prefieren una interfaz gráfica para administrar sus contenedores, imágenes e instancias de Docker, para ello, contamos con la disponibilidad de DockStation.

DockStation es una aplicación centrada en el desarrollo para administración de proyectos con Docker. Nos permite administrar, interactuar y configurar servicios, imágenes y contenedores desde una interfaz gráfica y sin tener que salir de ella.



Entre otras cosas, DockStation permite:

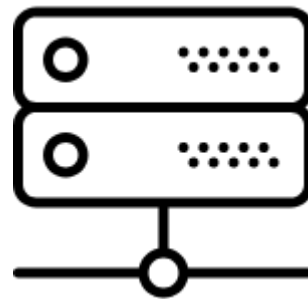
Trabajar con servicios y contenedores

Ayuda a administrar proyectos y configuraciones de contenedor, por ejemplo, vincular un host local a un proyecto, cambiar de versión, asignar puertos, asignar y reasignar variables de entorno, cambiar el punto de entrada e indicar comandos, configurar volúmenes, acceso rápido a la documentación de imágenes, limpieza de contenedores de servicios y muchas otras funcionalidades útiles.



Trabajo con contenedores remotos

La aplicación ayuda a gestionar y observar contenedores remotos. Proporciona muchas herramientas, como la supervisión de registros, la búsqueda de registros, la agrupación, la ejecución de herramientas y la obtención de información de contenedor. También proporciona herramientas de autorización para conexiones remotas.



Independencia

No requiere la instalación local de Docker para controlar los contenedores remotos.

También puede usarse como una herramienta de administración y monitoreo para contenedores Docker remotos.

Tiene soporte de Docker Machine (Oracle VirtualBox, VMware Fusion, Microsoft Hyper-V).



Compatibilidad hacia atrás

La aplicación funciona con Docker Compose. Podemos usar archivos `docker-compose.yml` de terceros. DockStation genera un archivo `docker-compose.yml` nativo y limpio que se puede usar incluso fuera de la aplicación, utilizando los comandos del CLI de Docker Compose nativos.



Monitor de estadísticas y recursos

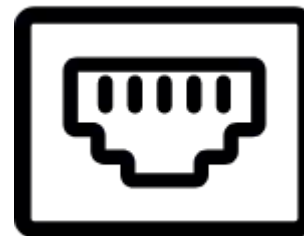
Excelentes herramientas para el monitoreo común, múltiple y único de los recursos de contenedores.

Fácil seguimiento del uso de la CPU, uso de la memoria, E / S de redes, E / S de bloques.



Monitor de puertos

Ayuda a ver y administrar puertos abiertos de contenedores.



Crear proyectos

Podemos crear proyectos nuevos usando Docker con solo unos pocos clicks a través de la interfaz de usuario de DockStation.



GUI

La aplicación combina muchos comandos CLI en una interfaz gráfica conveniente. De modo que en lugar de muchos comandos solo se necesita un clic.



Observador

Con la aplicación es muy fácil ver el estado de los contenedores y ver sus registros.



Descarga e instalación:

<https://dockstation.io/>

<https://github.com/DockStation/dockstation>



Práctica



Add new project



● Connected: localhost



Práctica

Create a new project

Project title

For import existing project you should set a path to your **docker-compose.yml** file.

Path SET PATH

You can paste a **docker run** command to parse to docker compose format. ** Not required*

docker run --name test -it debian

Compose file version

2.0

Override Docker Compose configs

+ ADD OVERRIDE CONFIG

CANCEL CREATE

Connected: localhost



hello-world
~/home/pedro/Documents

WEBRESTARTSTOPREMOVEBUILD

GENERALSCHEMESettings

web (No containers)
gitlab/gitlab-ce/latest

jenkins (1 container)
jenkins/jenkins:lts

>EXECSTOPRESTARTCLEARREMOVEINFOBUILDSLOGSSETTINGS

```
jenkins_1 | INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext@3435664e]: org.springframework.beans.factory.support.DefaultListableBeanFactory@b7366aa
jenkins_1 | Mar 20, 2019 8:50:20 AM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
jenkins_1 | INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@b7366aa: defining beans [filter,legacy]; root of factory hierarchy
jenkins_1 | Mar 20, 2019 8:50:20 AM jenkins.install.SetupWizard init
jenkins_1 | INFO:
jenkins_1 | .....
jenkins_1 | .....
jenkins_1 | .....
jenkins_1 | Jenkins initial setup is required. An admin user has been created and a password generated. Please use the following password to proceed to installation:
jenkins_1 | 
jenkins_1 | 13cbf34fb0b334a3fd8ba7e780922c9d6
jenkins_1 | This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
jenkins_1 | .....
jenkins_1 | .....
jenkins_1 | ---> setting agent port for jnlp
jenkins_1 | --> setting agent port for jnlp... done
jenkins_1 | Mar 20, 2019 8:50:39 AM hudson.model.UpdateSite updateData
jenkins_1 | INFO: Obtained the latest update center data file for UpdateSource default
jenkins_1 | Mar 20, 2019 8:50:39 AM hudson.model.UpdateSite updateData
jenkins_1 | INFO: Obtained the latest update center data file for UpdateSource default
jenkins_1 | Mar 20, 2019 8:50:41 AM jenkins.InitReactorRunner$1.onAttained
jenkins_1 | INFO: Completed initialization
jenkins_1 | Mar 20, 2019 8:50:41 AM hudson.WebAppMain$3.run
jenkins_1 | INFO: Jenkins is fully up and running
jenkins_1 | Mar 20, 2019 8:50:42 AM hudson.model.DownloadService$Downloadable load
jenkins_1 | INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
jenkins_1 | Mar 20, 2019 8:50:42 AM hudson.model.AsyncPeriodicWork$1.run
jenkins_1 | INFO: Finished Download metadata. 24,483 ms
```

Search projectNEW

Connected: localhost



Práctica

The screenshot displays the DockStation web interface. At the top, there is a navigation bar with tabs: NEWS, Projects, Containers, Stats Monitor, Ports, and Preferences. Below this, a sidebar on the left shows a project named 'hello-world' with the path '/home/pedro/Documents'. The main area is divided into two sections. The left section, titled 'Docker Hub', lists several official Docker images: 'jenkins' (76M, 4.1K), 'mongo' (1.5G, 5.6K), 'mysql' (734M, 7.7K), and 'ghost' (75M, 939). The right section, titled 'Local Images', shows a running container named 'jenkins' with a status of 'running'. A tooltip for the 'jenkins' service is visible, stating 'Service: web' and 'Service doesn't have containers. Please run the service.' The bottom of the interface includes a search bar for projects and images, and a status indicator showing 'Connected: localhost'.

NEWS Projects Containers Stats Monitor Ports Preferences

hello-world
/home/pedro/Documents

WEB RESTART STOP REMOVE BUILD

GENERAL SCHEME SETTINGS

Docker Hub Local Images

official jenkins 76M 4.1K
Official Jenkins Docker image

official mongo 1.5G 5.6K
MongoDB document databases provide high availability and easy scalability.

official mysql 734M 7.7K
MySQL is a widely used, open-source relational database management system...

official ghost 75M 939
Ghost is a free and open source blogging platform written in JavaScript

official 1.4G

Search project + NEW Search images

Connected: localhost

Service: web
Service doesn't have containers. Please run the service.

jenkins jenkins
Container #1 running



Práctica

NEWS

Projects

Containers

Stats Monitor

Ports

Preferences

hello-world
/home/pedro/Documents

WEBRESTARTSTOPREMOVBUILD

GENERALSCHEMESSETTINGS

General settings of project

hello-world

/home/pedro/Documents

SET PATH

Compose file version

3.0

Override Docker Compose configs

+ ADD OVERRIDE CONFIG

Host settings

Assign local host

select service

APPLY

Search project

+ NEW

Connected: localhost



Práctica

NEWS Projects Containers Stats Monitor Ports Preferences

localhost DISCONNECT

helloworld_jenkins_1 jenkins/jenkins:its

relaxed_brahmagupta jenkins

helloworld_jenkins_1 Status: Up 3 minutes Created: 20 March 2019 Project: helloworld

Mar 20, 2019 8:50:20 AM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext@34356646]: org.springframework.beans.factory.support.DefaultListableBeanFactory@b7366aa
Mar 20, 2019 8:50:20 AM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@b7366aa: defining beans [filter, legacy]; root of factory hierarchy
Mar 20, 2019 8:50:20 AM jenkins.install.SetupWizard init
INFO:
.....
.....
.....
Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

13cbf34feb334a3f8dba7e780922c9d6

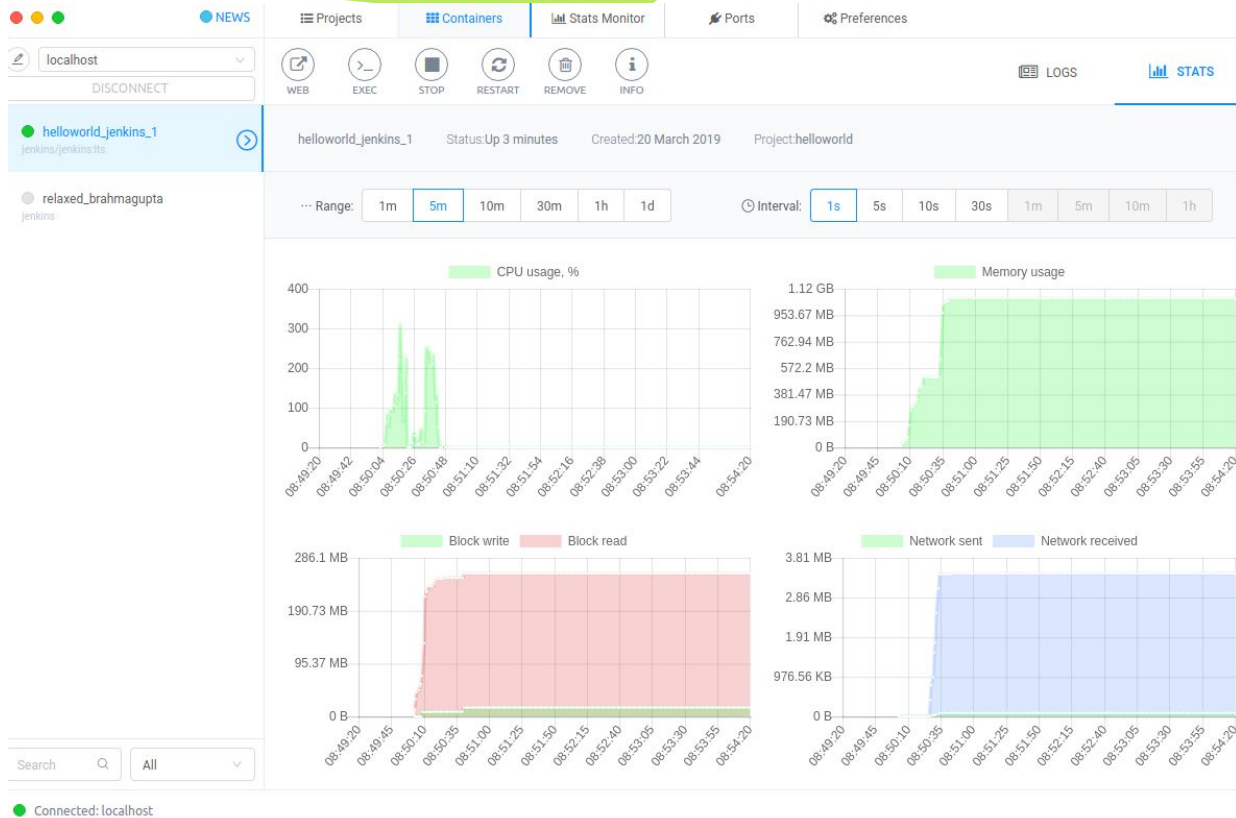
This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
.....
.....
.....
--> setting agent port for jnlp
--> setting agent port for jnlp... done
Mar 20, 2019 8:50:39 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Mar 20, 2019 8:50:39 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Mar 20, 2019 8:50:41 AM jenkins.InitReactorRunner\$1 onAttained
INFO: Completed initialization
Mar 20, 2019 8:50:41 AM hudson.WebAppMain\$3 run
INFO: Jenkins is fully up and running
Mar 20, 2019 8:50:42 AM hudson.model.DownloadService\$Downloadable load
INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Mar 20, 2019 8:50:42 AM hudson.model.AsyncPeriodicWork\$1 run
INFO: Finished Download metadata. 24,483 ms

Search All

Connected: localhost



Práctica



Práctica

NEWS

ProjectsContainersStats MonitorPortsPreferences

localhost

DISCONNECT

☐ helloworld_jenkins_1
jenkins/jenkins:its

☐ relaxed_brahmagupta
jenkins

Search

All

Connected: localhost

Containers: 2Running: 1CPU: 0.137 %MEMORY USAGE: 1.05 GB

Name	CONTAINER	CPU	MEM USAGE / LIMIT	NET I/O	BLOCK I/O	PIDS
helloworld_jenkins_1	2f5907b6aba4	0.179 %	1.05 GB / 7.69 GB	3.35 MB / 92.47 KB	241.2 MB / 16.3 MB	42
relaxed_brahmagupta	9f07335f05cf	-	-	-	-	0



Práctica

NEWS

Projects

Containers

Stats Monitor

Ports

Preferences

ID	Name	Ports	Actions
2f5907b6aba4	helloworld_jenkins_1	2020, 8081	<div><div></div><div></div><div></div></div>



Práctica

localhost:8081/login?from=%2F

Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

```
/var/jenkins_home/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

Administrator password



Otras GUI para Docker:

1. **Kitematic:** <https://kitematic.com/>
2. **Portainer:** <https://www.portainer.io/>
3. **Shipyards:** <https://shipyards-project.com/>
4. **Docker-compose UI:** <https://github.com/francescou/docker-compose-ui>



Kubernetes



Kubernetes

- ¿Qué es?
- ¿Para qué se usa?

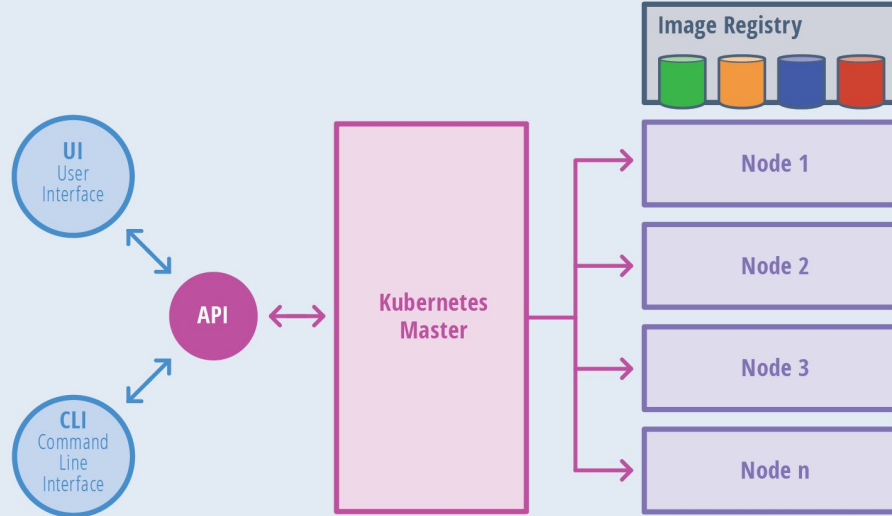


kubernetes

Kubernetes - Diseño

- Kubernetes es un sistema open source para la gestión de aplicaciones en contenedores, un sistema de **orquestación** para contenedores Docker, permitiendo acciones como programar el despliegue, escalado y la monitorización de nuestros contenedores, entre muchas otras más. Fue originalmente diseñado por Google y donado a la Cloud Native Computing Foundation (parte de la Linux Foundation).

Kubernetes Architecture



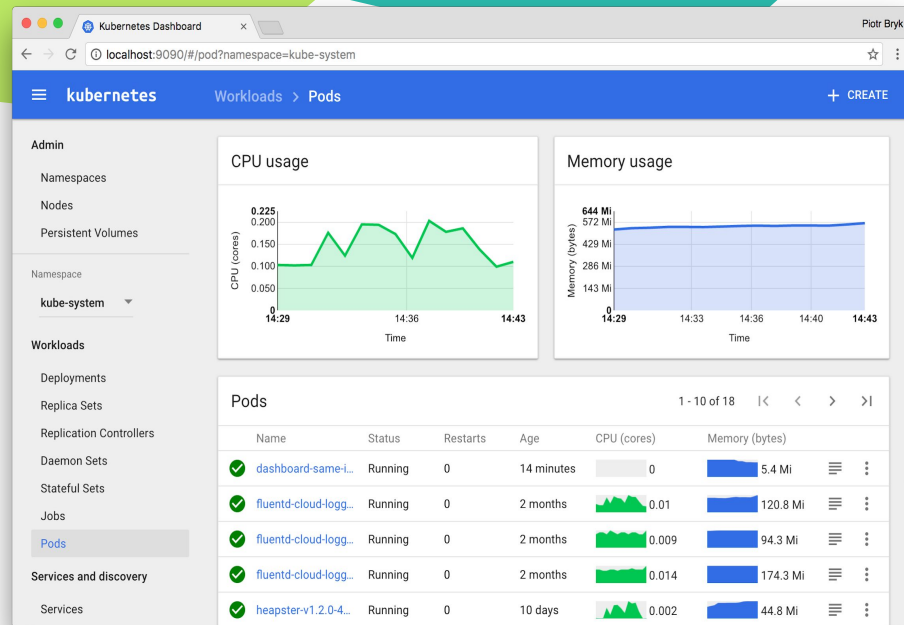
Kubernetes

Podemos encontrar como complemento el dashboard de Kubernetes, una interfaz gráfica que nos permite desplegar nuestras aplicaciones en contenedores a un clúster con Kubernetes, hacer debug y monitorear nuestras aplicaciones desplegadas y gestionar el clúster como tal.

Además, tendremos acceso a las aplicaciones o el listado de aplicaciones desplegadas en el clúster de Kubernetes, acceso a la gestión de los recursos del clúster y estadísticas de rendimiento y ejecución de una forma muy intuitiva.

Más información acerca del dashboard:

<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>



Kubernetes - Diseño

Kubernetes define un conjunto de bloques de construcción que conjuntamente proveen los mecanismos para el despliegue, mantenimiento y escalado de aplicaciones.

Los componentes que forman Kubernetes están diseñados para estar débilmente acoplados pero a la vez ser extensibles para que puedan soportar una gran variedad de flujos de trabajo.

Kubernetes - Diseño

Kubernetes coordina un grupo de máquinas de alta disponibilidad que están conectadas para trabajar como una sola unidad. Las abstracciones en Kubernetes le permiten desplegar aplicaciones en contenedores en un cluster sin atarlas específicamente a máquinas individuales.

Para hacer uso de este nuevo modelo de despliegue, las aplicaciones deben empaquetarse de forma que se desacoplen de los hosts individuales: deben ser empaquetadas en contenedores.

Kubernetes - Diseño

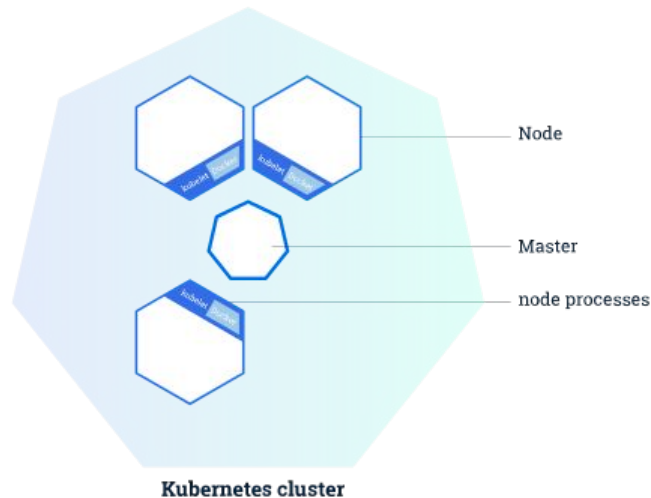
Las aplicaciones en contenedor son más flexibles y disponibles que en los modelos de implementación anteriores, en los que las aplicaciones se instalaban directamente en máquinas específicas como paquetes profundamente integrados en el host.

Kubernetes automatiza la distribución y programación de contenedores de aplicaciones a través de un cluster de una manera más eficiente.

Kubernetes - Master

El Master es responsable de la gestión del cluster.

El maestro coordina todas las actividades de su clúster, como la programación de aplicaciones, el mantenimiento del estado deseado de las aplicaciones, el escalado de aplicaciones y la implementación de nuevas actualizaciones.



Kubernetes - Diseño

Un nodo es una VM o un ordenador físico que sirve como una máquina de trabajo en un cluster de Kubernetes.

Cada nodo tiene un **Kubelet**, que es un agente para gestionar el nodo y comunicarse con el maestro Kubernetes.

El nodo también debe tener herramientas para manejar operaciones de contenedores, como Docker

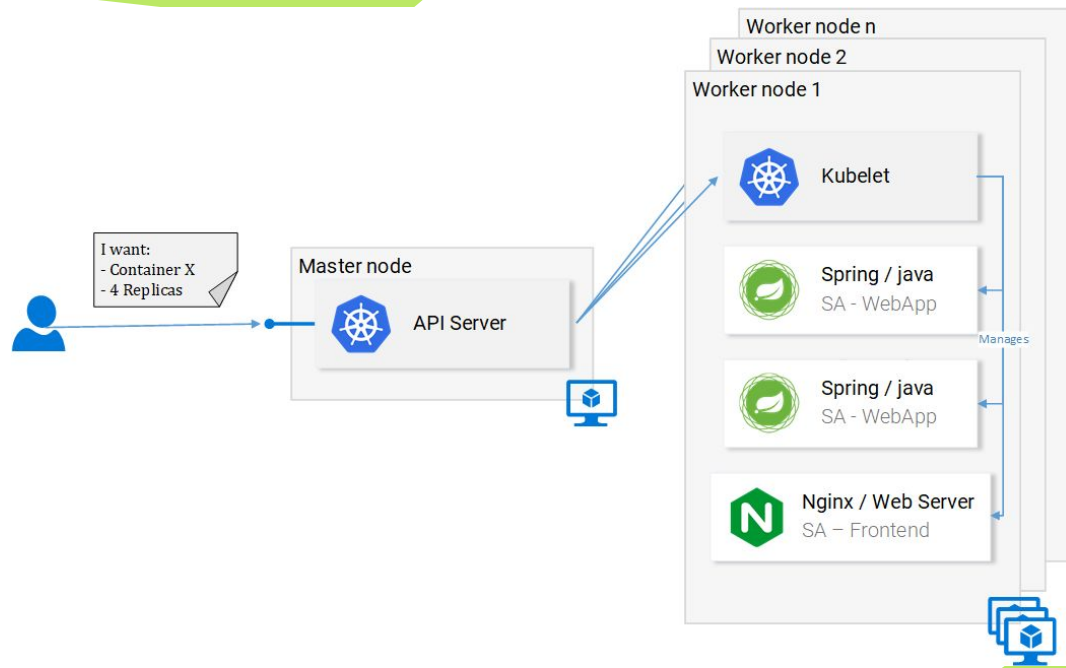
Kubernetes - Diseño

Cuando se despliegan aplicaciones en Kubernetes, se le dice al maestro que inicie los contenedores de aplicación.

El maestro programa los contenedores para que se ejecuten en los nodos del cluster. Los nodos se comunican con el maestro usando la API de Kubernetes, que el maestro expone.

Kubernetes - Arquitectura

- ◆ Los usuarios finales también pueden utilizar la API de Kubernetes directamente para interactuar con el clúster. Un clúster de Kubernetes puede desplegarse en máquinas físicas o virtuales.



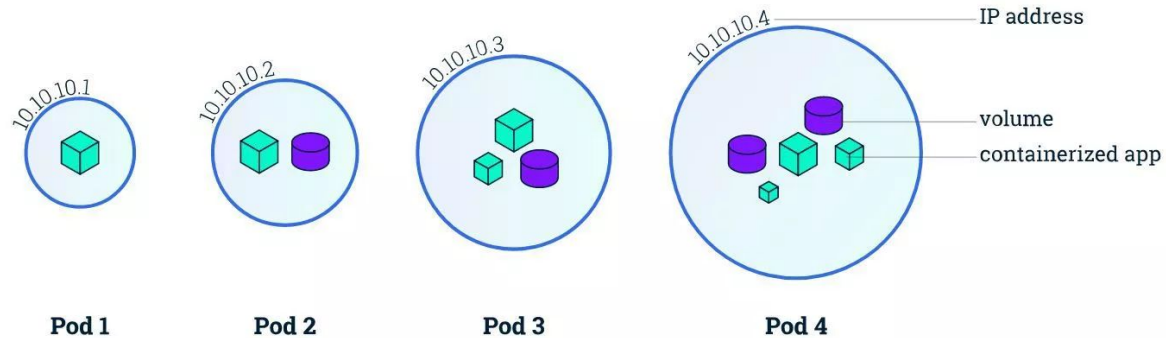
Kubernetes - Diseño

Podemos resumir el diseño interno de Kubernetes en los siguientes componentes:

- Pods: La unidad básica de planificación en Kubernetes es llamada capsula (“pod”). Este agrega un nivel de abstracción más elevado a los componentes en contenedores.
- Un pod consta de **uno o más contenedores** en los que se garantiza su ubicación en el mismo equipo anfitrión y pueden compartir recursos.

Kubernetes - Diseño

- **Pods:** conjunto de bloques de construcción



Los pods pueden utilizarse para realizar escalado horizontal, aunque fomentan el trabajo con microservicios puestos en contenedores diferentes para crear un sistema distribuido mucho más robusto.

Kubernetes - Diseño

- Cada pod en Kubernetes es asignado a una única dirección IP (dentro del clúster) que permite a las aplicaciones utilizar puertos sin riesgos de conflictos.
- Un pod puede definir un volumen, como puede ser un directorio de disco local o un disco de red, y exponerlo a los contenedores dentro del pod. Los pods pueden ser administrados manualmente a través de la API de Kubernetes, o su administración puede ser delegada a un controlador.

Kubernetes - Diseño

- Hay que tener en cuenta que los **pods** son entidades efímeras. En el ciclo de vida de un pod estos se crean y se les asigna un UID hasta que terminen o se borren. **Si un nodo que contiene un pod es eliminado, todos los pods que contenía ese nodo se pierden.**
- Este pod puede ser reemplazado en otro nodo, aunque el UID será diferente. Esto es importante porque un pod no debería de tener información almacenada que pueda ser utilizada después por otro pod en caso de que a este le pasara algo. Para compartir información entre pods están los volúmenes

Kubernetes - Diseño

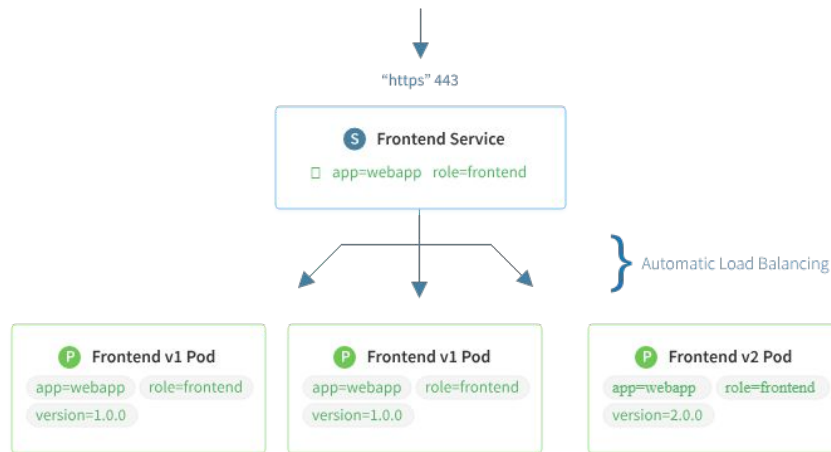
Servicios: un conjunto de pods que trabajan para un fin

El conjunto de pods que constituyen un servicio está definido por el selector de etiquetas.

Kubernetes provee de un servicio de descubrimiento y enrutamiento de pedidos mediante la asignación de una dirección IP estable y un nombre DNS al servicio, y balancea la carga de tráfico en un estilo round-robin hacia las conexiones de red de las direcciones IP entre los pods que verifican el selector (incluso cuando fallas causan que los pods se muevan de máquina en máquina).

Kubernetes - Diseño

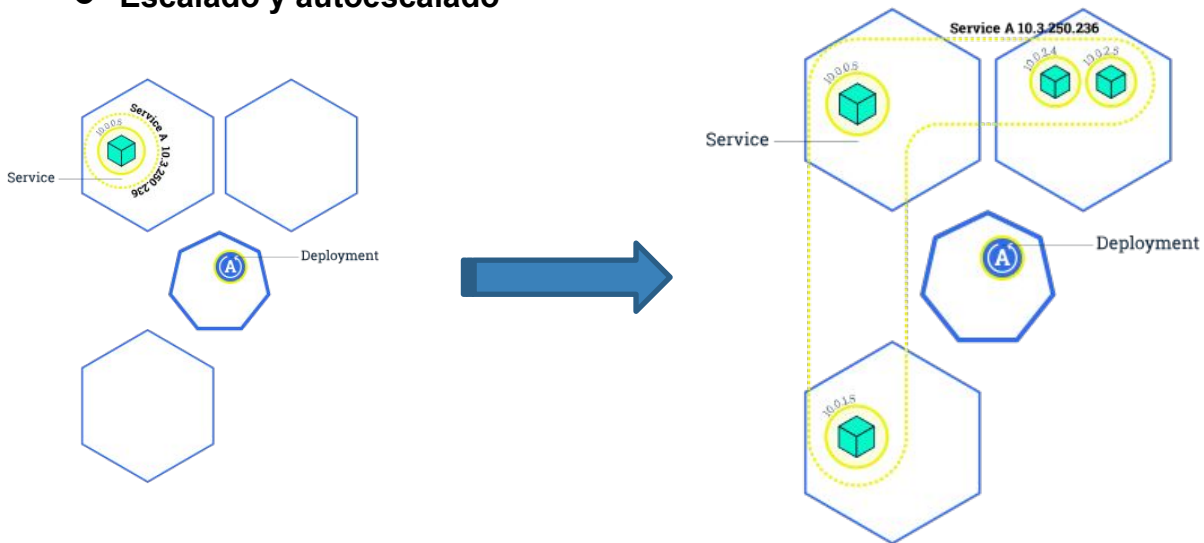
- Por defecto un servicio es expuesto dentro de un cluster (por ejemplo, pods de un backend pueden ser agrupados en un servicio, con las peticiones de los pods de frontend siendo balanceadas entre ellos), pero un servicio también puede ser expuesto hacia afuera del clúster.



Kubernetes - Escalado



- Escalado y autoescalado



Kubernetes - Escalado

- ◆ El escalado de un Despliegue asegurará que se creen nuevos Pods y se programen en Nodos con los recursos disponibles. El escalamiento aumentará el número de pods hasta alcanzar el nuevo estado deseado.
- ◆ Kubernetes también soporta la autoescala de Pods.
- ◆ El escalamiento se logra cambiando el número de réplicas en una implementación.

Kubernetes - Glosario de términos

Antes de continuar, tengamos en cuenta algunos términos comunes de Kubernetes:

Annotation: Una pareja clave-valor utilizada para añadir metadatos a los objetos. La información adicional proporcionada por las anotaciones puede ser grande o pequeña, estructurada como JSON o texto plano, y además permite caracteres que no están soportados por las Labels . Esta información puede ser útil para librerías o clientes que puedan necesitar información adicional.

Applications: representa la capa donde se ejecutan las aplicaciones contenidas

Clúster: un conjunto de máquinas, llamadas nodos, que ejecutan aplicaciones en contenedores administradas por Kubernetes. Un clúster tiene varios nodos de trabajo y como mínimo un nodo maestro.

Kubernetes - Glosario de términos

Container: imagen ejecutable y portable que contiene una aplicación y todas sus dependencias.

Container Environment Variables: variables de entorno de un contenedor en formato clave-valor

Container Runtime: representa el software encargado de la ejecución de contenedores

Container runtime interface (CRI): API de integración de los runtimes de contenedores con kubelet en un nodo

Control Plane: capa de orquestación de contenedores que expone la API para gestionar el ciclo de vida de los contenedores

Kubernetes - Glosario de términos

Controller: un loop de control que monitorea el estado del clúster a través del apiserver y realiza cambios en función de llevar ese estado al deseado

CustomResourceDefinition: código custom que define un recurso a añadir al apiserver de Kubernetes sin necesidad de definir un custom server nuevo

DaemonSet: capa que asegura que una copia de un Pod está ejecutándose en un set de nodos en un clúster

Data Plane: capa que provee de recursos de memoria, procesamiento, etc. para que los contenedores se ejecuten en una red

Kubernetes - Glosario de términos

Deployment: un objeto API que gestiona una aplicación replicada

Device Plugin: contenedores en ejecución dentro de kubernetes que proveen acceso a recursos específicos

Extensions: componentes de software que se integran para el soporte de nuevos tipos de Hardware con Kubernetes

Job: Una tarea finita o por lotes que se ejecuta hasta su finalización

Kubernetes - Glosario de términos

Kubectl: herramienta de línea de comandos para comunicarse con la API del servidor Kubernetes

Kubelet: agente ejecutándose en cada nodo de un clúster que asegura que los contenedores se ejecuten en los pods

Kubernetes API: aplicación que sirve la funcionalidad de Kubernetes a través de una interfaz RESTFull y almacena el estado en el clúster

Label: etiquetas para darle valor semántico a componentes, pods, servicios, etc. para dar al usuario información útil acerca de ellos

Kubernetes - Glosario de términos

LimitRange: provee de normas para limitar el consumo de recursos usados por contenedores o Pods en un namespace

Logging: listado de eventos exportados en un log por una aplicación o un clúster

Minikube: herramienta para ejecutar kubernetes de forma local

Namespace: abstracción usada por kubernetes para ejecutar varias instancias de clústers virtuales en el mismo clúster físico

Kubernetes - Glosario de términos

Node: representa una máquina del clúster de kubernetes

Name: Una cadena de caracteres proporcionada por el cliente que identifica un objeto en la URL de un recurso, como por ejemplo, /api/v1/pods/nombre-del-objeto

Pod: El objeto más pequeño y simple de Kubernetes. Un Pod es la unidad mínima de computación en Kubernetes y representa uno o más contenedores ejecutándose en el clúster

Pod Lifecycle: conjunto de fases por las que puede pasar un Pod durante su ciclo de vida útil

Kubernetes - Glosario de términos

Pod Security Policy: habilita un sistema de autenticación para gestionar Pods

QoS (Quality of Service) Class: provee a kubernetes la capacidad para clasificar Pods en el clúster y tomar decisiones de gestión sobre ellos

RBAC (Role-Based Access Control): provee un sistema de gestión de acceso según roles

ReplicaSet: provee un controlador de réplicas y replicación

Resource Quotas: provee normas para limitar el consumo de recursos de agregación por namespace

Kubernetes - Glosario de términos

Selector: permite a los usuarios filtrar recursos por labels

Service: es el objeto de la API de Kubernetes que describe cómo se accede a las aplicaciones, tal como un conjunto de Pods , y que puede describir puertos y balanceadores de carga

Volume: un directorio contenedor de data accesible por los contenedores en un pod

Instalación de Kubernetes

- ◆ Kubernetes puede funcionar en varias plataformas: desde un portátil, a VMs en un proveedor de nube, a un rack de servidores...
- ◆ El esfuerzo requerido para configurar un clúster varía desde ejecutar un solo comando hasta crear su propio clúster personalizado.
- ◆ Tenemos dos formas principales de tener kubernetes...

1. Local-machine Solutions

- ◆ **Minikube** es un método para crear un clúster local de Kubernetes de un solo nodo para desarrollo y pruebas. La configuración está completamente automatizada y no requiere una cuenta de proveedor de cloud computing.
- ◆ **microk8s** proporciona una única instalación de comando de la última versión de Kubernetes en una máquina local para desarrollo y pruebas. La configuración es rápida (~30 seg.)

Instalación de Kubernetes

- ◆ Un clúster de Kubernetes puede desplegarse en máquinas físicas o virtuales.
- ◆ Para empezar con el desarrollo de Kubernetes, podemos usar Minikube. Minikube es una implementación ligera de Kubernetes que crea una VM en su máquina local y despliega un clúster simple que contiene sólo un nodo.
- ◆ <https://github.com/kubernetes/minikube>

Instalación de Kubernetes

- ◆ Minikube está disponible para sistemas Linux, macOS y Windows. El CLI de Minikube proporciona operaciones básicas de bootstrapping para trabajar con su cluster, incluyendo inicio, parada, estado y borrado.
- ◆ Para nuestras prácticas vamos a usar un terminal en línea con Minikube preinstalado.

Demo de Kubernetes

- ◆ <https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-interactive/>
- ◆ Kubectl: es el programa que vamos a utilizar para interactuar con el api de kubernetes.

Demo de Kubernetes – Creación de Pods vía CLI

◆ <https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-interactive/>

Demo de Kubernetes – Interactuando con el POD

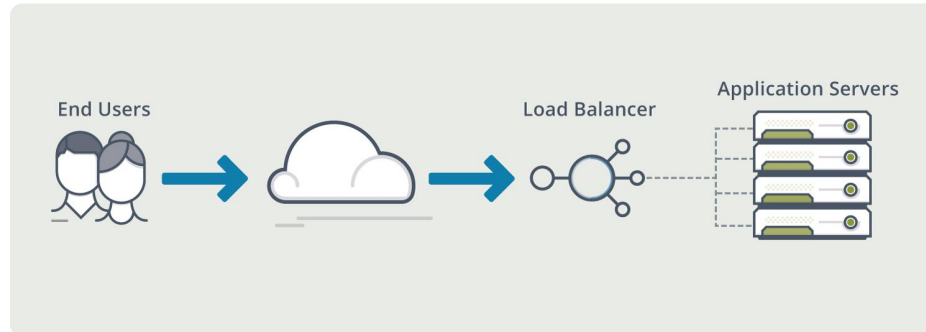
◆ <https://kubernetes.io/docs/tutorials/kubernetes-basics/explore-interactive/>

Demo de Kubernetes – Ingresando al Dashboard UI

◆ Minikube dashboard

Kubernetes - Ventajas

- Descubrimiento de servicios y balanceo de carga



Kubernetes - Ventajas

- **Autorreparación**



Kubernetes - Ventajas

- **Despliegues y rollbacks automáticos**
- cuando hay que actualizar una aplicación o cambiar su configuración, Kubernetes despliega los cambios de forma progresiva mientras monitoriza su salud para asegurar que no mata todas las instancias a la vez, y en caso de fallo, hace un rollback automático.



Kubernetes - Ventajas

- **Planificación**
- Se encarga de decidir en qué nodo se ejecutará cada contenedor de acuerdo a los recursos que requiera y a otras restricciones.
- Mezcla cargas de trabajo críticas y best-effort para potenciar la utilización y el ahorro de recursos.



Kubernetes - Ventajas

- **Gestión de la configuración y secrets:** la información sensible, como las passwords o las claves ssh, se almacena en Kubernetes oculta en secrets. Tanto la configuración de la aplicación como los secrets se despliegan y se actualizan sin tener que reconstruir la imagen ni exponer información confidencial.

A terminal window with a black background and green text. The text reads "INSERT PASSWORD:" on the first line and "*****" on the second line, with a small green square cursor at the end of the asterisks.

INSERT PASSWORD:

Kubernetes - Ventajas

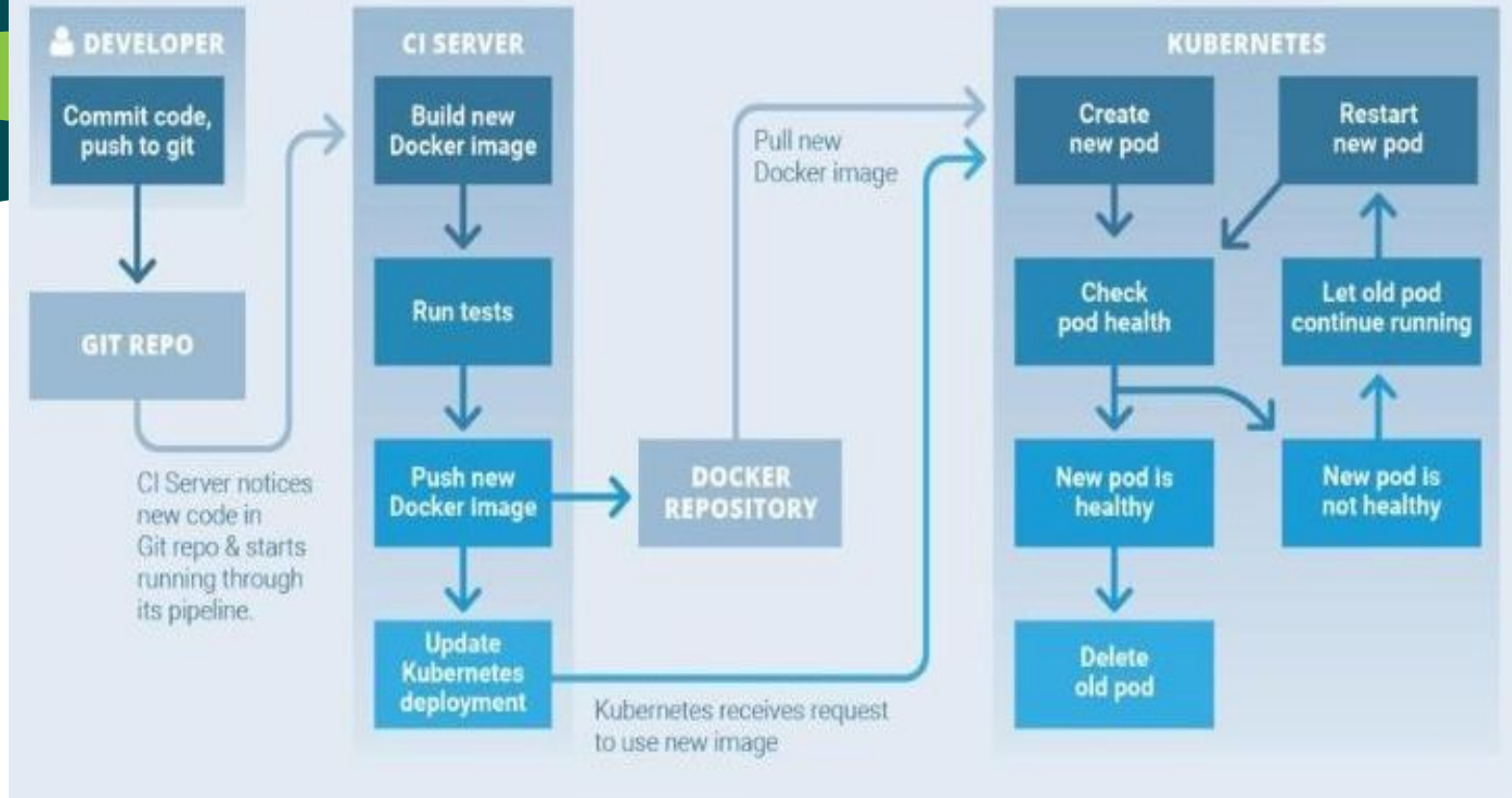
- Orquestación del almacenamiento



Kubernetes - Ventajas

- **Ejecución Batch**





<https://www.katacoda.com/courses/kubernetes>

Kubernetes - Práctica local

Instalación:

- `curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && sudo install minikube-linux-amd64 /usr/local/bin/minikube`
- `curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -`
- `echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/sources.list.d/kubernetes.list`
- `sudo apt-get update`
- `sudo apt-get install -y kubectl`

Kubernetes - Práctica local

Iniciar minikube:

- `minikube start`
- `minikube status`
- `minikube dashboard`

Crear un pod:

- `kubectl run mynginxapp --image=nginx:latest --port=80`

Crear un servicio

- `kubectl expose deployment mynginxapp --type=NodePort`

Kubernetes - Práctica local

Revisar el estatus del pod y el servicio:

- `kubectl get pods`
- `kubectl get services`

Ver url del servicio iniciado con minikube:

- `minikube service list mynginxapp`

Kubernetes - Práctica local

Borrar el servicio y detener minikube

- `kubectl delete service mynginxapp`
- `kubectl delete deployment mynginxapp`
- `minikube stop`

Kubernetes - Práctica local

Práctica guiada trabajando con formato YAML

- **Creando un local volume**
- **Crear secret**
- **Crear archivos de configuración de mysql y wordpress**
- **Visualizar en minikube dashboard**
- **Yendo un paso más allá: automatizando con shell scripting**

Kubernetes - Práctica local

Práctica:

Crear un archivo que levante un servidor NodeJS con el siguiente contenido:

Kubernetes - Práctica local

```
var http = require('http');

var handleRequest = function(request, response) {
  console.log('Received request for URL: ' + request.url);
  response.writeHead(200);
  response.end('Hola Docker!');
};

var www = http.createServer(handleRequest);
www.listen(8888);
```

Kubernetes - Práctica local

Probamos nuestro nuevo server con:

```
$ node server.js
```

E ingresando a:

```
http://127.0.0.1:8888
```


Kubernetes - Práctica local

Creamos un Dockerfile con el contenido:

```
FROM node:6.9.2
```

```
EXPOSE 8080
```

```
COPY server.js .
```

```
CMD node server.js
```

Kubernetes - Práctica local

Seteamos una variable de entorno para usar imágenes de docker locales:

```
eval $(minikube docker-env)
```

Y procedemos a construir la imagen:

```
$ docker build -t hello-node:v1 .
```

Podemos verificar que la imagen se ha creado con:

```
$ docker images
```

Kubernetes - Práctica local

Teniendo la imagen base, podemos poner en marcha el contenedor dentro de Kubernetes con el comando:

```
$ kubectl run hello-node --image=hello-node:v1 --port=8888 --image-pull-policy=Never
```

Y verlo entre los deployments de KubeCtl:

```
$ kubectl get deployments
```

Kubernetes - Práctica local

También podemos listar los Pods para ver en cuál está corriendo:

```
$ kubectl get pods
```

Si queremos ver los eventos disparados por nuestro nuevo deploy podemos ejecutar:

```
$ kubectl get events
```

Kubernetes - Práctica local

Si ingresamos a la web del dashboard de minikube, también podremos ver y gestionar nuestro nuevo despliegue.

Para exponerlo a la red, ejecutamos:

```
$ kubectl expose deployment hello-node --type=NodePort --port=8080 --target-port=8888
```

Y para entrar desde el navegador al servicio expuesto:

```
$ minikube service hello-node
```

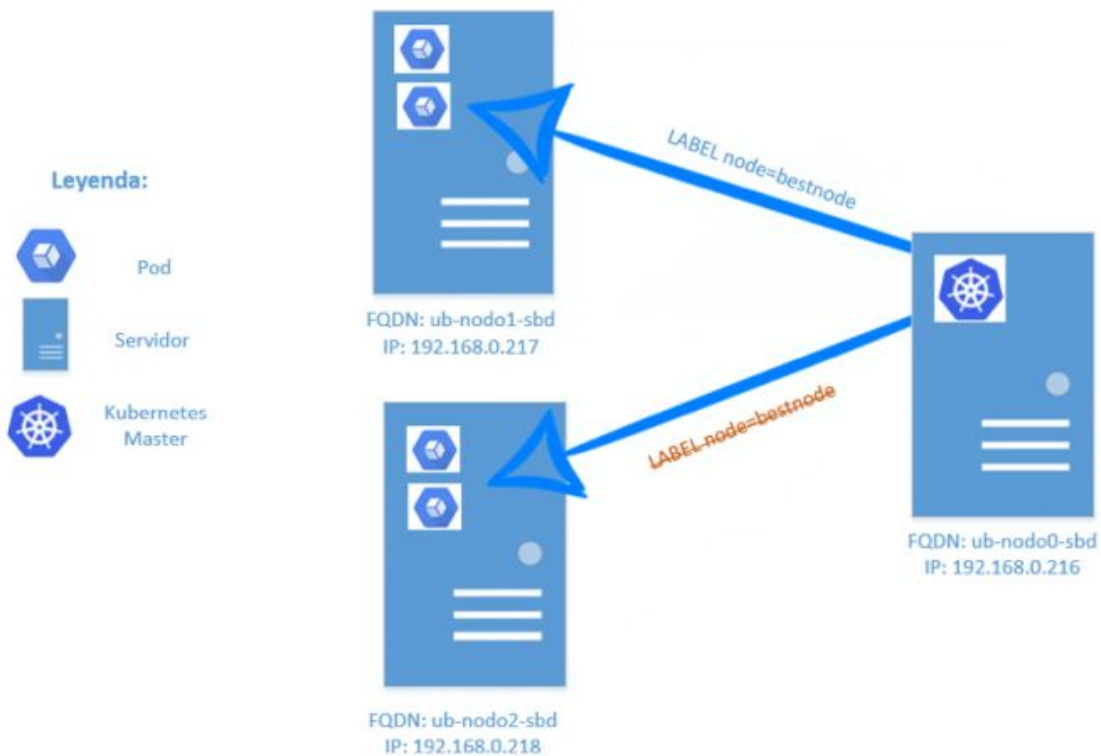
Kubernetes

Labels en Kubernetes

Los labels en Kubernetes funcionan como en otras herramientas como Docker, por ejemplo, básicamente definen un mecanismo de organizar los objetos en Kubernetes. Un label, realmente es un key-value que se le asigna a un objeto, como podría ser un service, un pod, un nodo, etc....

Un ejemplo de uso básico, es querer desplegar varios pods en un solo nodo (ub-nodo1-sbd) mediante un label, tal como se muestra a continuación:

Kubernetes



Kubernetes

Labels en Kubernetes

Para usar labels en Kubernetes, debemos asignar estos a, por ejemplo, un Pod que se esté ejecutando en nuestra instancia.

Lo primero que haremos es asignar el label “node=xxxxx” al servidor donde queremos que se desplieguen los pods y una vez hecho veremos que se ha creado el label.

Supongamos que queremos asignar el label “bestnode” a un nodo “ub-nodo1-sbd”

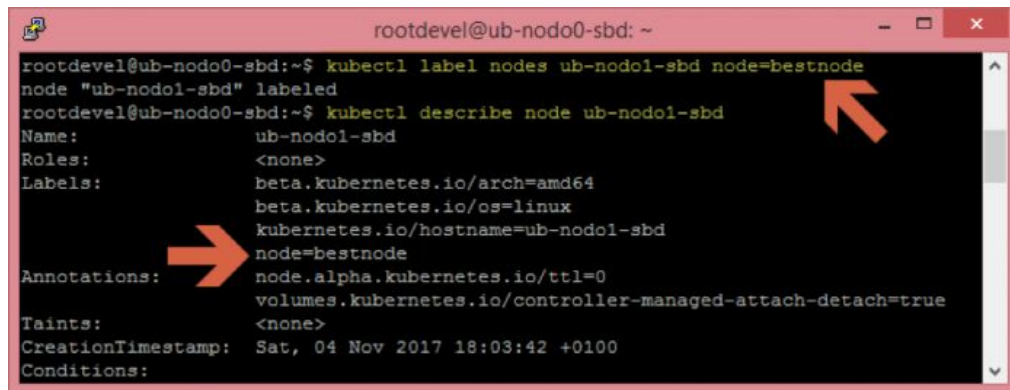
Kubernetes

Esto se logra ejecutando:

```
$ kubectl label nodes ub-nodo1-sbd node=bestnode
```

Y si hacemos un describe:

```
$ kubectl describe node ub-nodo1-sbd
```



```
rootdevel@ub-nodo0-sbd: ~  
rootdevel@ub-nodo0-sbd:~$ kubectl label nodes ub-nodo1-sbd node=bestnode  
node "ub-nodo1-sbd" labeled  
rootdevel@ub-nodo0-sbd:~$ kubectl describe node ub-nodo1-sbd  
Name:                ub-nodo1-sbd  
Roles:                <none>  
Labels:               beta.kubernetes.io/arch=amd64  
                     beta.kubernetes.io/os=linux  
                     kubernetes.io/hostname=ub-nodo1-sbd  
                     node=bestnode  
Annotations:          node.alpha.kubernetes.io/ttl=0  
                     volumes.kubernetes.io/controller-managed-attach-detach=true  
Taints:               <none>  
CreationTimestamp:    Sat, 04 Nov 2017 18:03:42 +0100  
Conditions:
```

The terminal window shows the execution of two kubectl commands. The first command labels the node 'ub-nodo1-sbd' with the label 'node=bestnode'. The second command describes the node, showing its name, roles, labels, annotations, taints, creation timestamp, and conditions. Two orange arrows are present: one pointing to the 'node=bestnode' label in the output of the first command, and another pointing to the 'node=bestnode' label in the output of the second command.

Kubernetes

Labels en Kubernetes

Para usar labels en Kubernetes, debemos asignar estos a, por ejemplo, un Pod que se esté ejecutando en nuestra instancia.

Lo primero que haremos es asignar el label “node=xxxxx” al servidor donde queremos que se desplieguen los pods y una vez hecho veremos que se ha creado el label.

Supongamos que queremos asignar el label “bestnode” a un nodo “ub-nodo1-sbd”

Kubernetes

Labels en Kubernetes

Ya que tenemos nuestro label, podemos usarlo, en el siguiente YML crearemos 3 pods y los asignaremos al equipo que hemos etiquetado:

Kubernetes

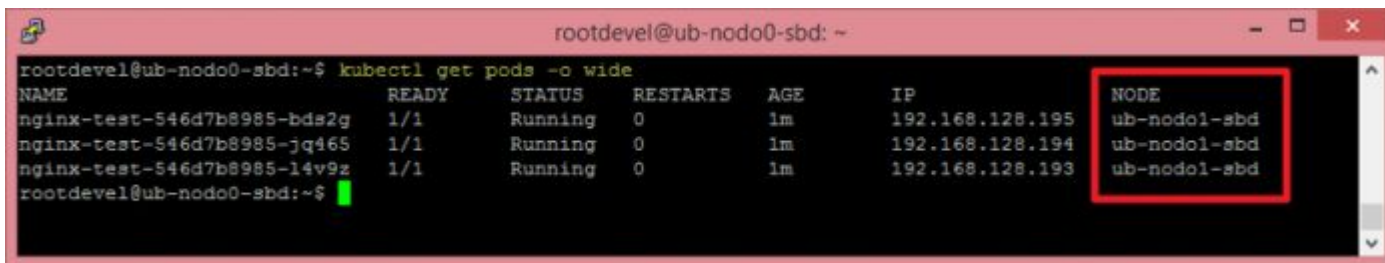
```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-test
spec:
  replicas: 3
  template:
    metadata:
      name: nginx
      namespace: default
      labels:
        env: beta
    spec:
      containers:
        - name: nginx
          image: nginx
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
      nodeSelector:
        mini: kube
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-test
spec:
  ports:
    - name: "www"
      port: 80
      targetPort: 80
  selector:
    run: my-nginx
  type: LoadBalancer
```

Kubernetes

Labels en Kubernetes

`$ kubectl create -f nginx-en-nodo1.yml`

Si ejecutamos `$ kubectl get pods -o wide` veremos algo como:



```
rootdev@ub-nodo0-sbd:~$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
nginx-test-546d7b8985-bds2g         1/1     Running   0           1m    192.168.128.195  ub-nodo1-sbd
nginx-test-546d7b8985-jq465         1/1     Running   0           1m    192.168.128.194  ub-nodo1-sbd
nginx-test-546d7b8985-l4v9z         1/1     Running   0           1m    192.168.128.193  ub-nodo1-sbd
rootdev@ub-nodo0-sbd:~$
```

The terminal output shows a table of pods. The 'NODE' column is highlighted with a red box, showing that all three pods are running on 'ub-nodo1-sbd'.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx-test-546d7b8985-bds2g	1/1	Running	0	1m	192.168.128.195	ub-nodo1-sbd
nginx-test-546d7b8985-jq465	1/1	Running	0	1m	192.168.128.194	ub-nodo1-sbd
nginx-test-546d7b8985-l4v9z	1/1	Running	0	1m	192.168.128.193	ub-nodo1-sbd

Kubernetes - Réplicas

Escalar una aplicación en kubernetes es sumamente fácil, una de las formas más directas de hacerlo es a través de la creación de un número de réplicas, veamos cómo hacerlo en la siguiente práctica donde:

- Crearemos un deployment de nginx.
- Usaremos kubectl para obtener información acerca del deployment.
- Actualizaremos el deployment añadiendo réplicas

Kubernetes - Réplicas

Comencemos creando un fichero YAML describe un deployment que corre la imagen Docker

nginx:1.7.9:

apiVersion: apps/v1 # Usa apps/v1beta2 para versiones anteriores a 1.9.0

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 2 # indica al controlador que ejecute 2 pods

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

Kubernetes - Réplicas

Acto seguido, podemos crear un deployment basado en el fichero anterior ejecutando:

```
$ kubectl apply -f deployment.yaml
```

Si queremos obtener información acerca del deployment podemos ejecutar:

```
$ kubectl describe deployment nginx-deployment
```


Kubernetes - Réplicas

Para ver los Pods creados por el deployment podemos ejecutar:

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

Deberíamos ver algo como:

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1771418926-7o5ns	1/1	Running	0	16h
nginx-deployment-1771418926-r18az	1/1	Running	0	16h

Kubernetes - Réplicas

Si queremos ver aún más detalles de alguno de los pods listados anteriormente, podemos hacerlo ejecutando:

```
$ kubectl describe pod <pod-name>
```

Kubernetes - Réplicas

Actualizando el deployment

Podemos actualizar el deployment aplicando un nuevo fichero YAML. El siguiente fichero YAML especifica que el deployment debería ser actualizado para usar nginx 1.8.

```
apiVersion: apps/v1 # Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.8 # Actualiza la versión de nginx de 1.7.9 a 1.8
          ports:
            containerPort: 80
```

Kubernetes - Réplicas

Aplicamos el nuevo fichero YAML:

```
$ kubectl apply -f deployment-update.yml
```

Comprobamos como el deployment crea nuevos pods con la nueva imagen mientras va eliminando los pods con la especificación antigua:

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

Kubernetes - Réplicas

Escalando la app

Podemos aumentar el número de pods en el deployment aplicando un nuevo fichero YAML. El siguiente fichero YAML especifica un total de 4 réplicas, lo que significa que el deployment debería tener cuatro pods:

Kubernetes - Réplicas

apiVersion: apps/v1 # Usa apps/v1beta2 para versiones anteriores a 1.9.0

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: nginx

replicas: 4 # Actualiza el número de réplicas de 2 a 4

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.8

ports:

- containerPort: 80

Kubernetes - Réplicas

Aplicamos el fichero

```
$ kubectl apply -f deployment-scale.yml
```

Y verificamos que el deployment tenga, en efecto, 4 pods:

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

Kubernetes - Réplicas

Deberíamos ver algo como:

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-148880595-4zdqg	1/1	Running	0	25s
nginx-deployment-148880595-6zgi1	1/1	Running	0	25s
nginx-deployment-148880595-fxcez	1/1	Running	0	2m
nginx-deployment-148880595-rwovn	1/1	Running	0	2m

Kubernetes - Réplicas

Si hemos terminado, podemos eliminar el deployment con:

```
$ kubectl delete deployment nginx-deployment
```

Kubernetes - Log y Debug

Una vez tengamos una (o varias) apps contenidas y desplegadas en Kubernetes, será necesario realizar labores de debug por si se presenta algún inconveniente.

Para encontrar problemas dentro de un pod en Kubernetes tenemos a disposición distintos comandos, veamos algunos de ellos:

Kubernetes - Log y Debug

Supongamos que tenemos dos pods en ejecución, podemos partir del siguiente YML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

Kubernetes - Log y Debug

Y creamos un deployment a partir de él:

```
$ kubectl apply -f nginx-with-request.yml
```

Para ver el estatus de los pods podemos ejecutar:

```
$ kubectl get pods
```

Kubernetes - Log y Debug

Esto nos da una vista bastante general y poco informativa, pero si queremos ver más detalles de cada Pod, podemos ejecutar:

\$ kubectl describe pod <nombre-del-pod>

Ejecutando el comando anterior podemos ver información acerca de la configuración del contenedor y del pod (labels, recursos, requerimientos, etc) así como información general acerca de ambos (estado, eventos, conteo de reinicios, etc)

Kubernetes - Log y Debug

Dependiendo del state del contenedor (waiting, running, terminated) el comando retornará información adicional.

Pero, ¿y los Pods que están en Pending?

Generalmente veremos pods que se quedan en un state pendiente, esto sucede por varias razones, la más común es que el Pod requiere más recursos de los disponibles, si queremos ver por qué un Pod no está en ejecución, debemos ver los eventos que ocurren dentro de él, para esto, podemos seguir la siguiente línea de acciones:

Kubernetes - Log y Debug

Supongamos que tenemos 5 pods, podemos listarlos con:

```
$ kubectl get pods
```

Y la salida será:

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1006230814-6winp	1/1	Running	0	7m
nginx-deployment-1006230814-fmgu3	1/1	Running	0	7m
nginx-deployment-1370807587-6ekbw	1/1	Running	0	1m
nginx-deployment-1370807587-fg172	0/1	Pending	0	1m
nginx-deployment-1370807587-fz9sd	0/1	Pending	0	1m

Kubernetes - Log y Debug

Para descubrir por qué un Pod no está ejecutándose, podemos recurrir a:

```
$ kubectl describe pod <nombre-del-pod>
```

La salida será algo como:

Kubernetes - Log y Debug

Name: nginx-deployment-1370807587-fz9sd
Namespace: default
Node: /
Labels: app=nginx,pod-template-hash=1370807587
Status: Pending
IP:
Controllers: ReplicaSet/nginx-deployment-1370807587
Containers:

nginx:
Image: nginx
Port: 80/TCP
QoS Tier:
memory: Guaranteed
cpu: Guaranteed
Limits:
cpu: 1
memory: 128Mi
Requests:
cpu: 1
memory: 128Mi
Environment Variables:

Volumes:

default-token-4bcbi:
Type: Secret (a volume populated by a Secret)
SecretName: default-token-4bcbi

Events:

FirstSeen	LastSeen	Count	From	SubobjectPath	Type	Reason	Message
-----	-----	----	-----	-----	-----	-----	-----
1m	48s	7	{default-scheduler }			Warning	FailedScheduling pod

(nginx-deployment-1370807587-fz9sd) failed to fit in any node
fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU, requested: 1000, used: 1420, capacity: 2000
fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU, requested: 1000, used: 1100, capacity: 2000

Kubernetes - Log y Debug

En el resultado anterior se ve que el Pod falla por un FailedScheduling y vemos que el mensaje del detalle nos indica la falta de recursos necesarios para iniciar la ejecución.

Para corregir este tipo de situaciones, podemos optar por un escalado como los que hemos hecho y tener menos réplicas del pod. (si dejáramos el pod en pending no pasaría nada, no afecta a los demás pods ni los recursos asignados a ellos)

Otra cosa que debemos notar es que, si bien pudimos ver los últimos eventos ocurridos en el Pod, quizás nos haga falta información más detallada o, incluso, nos sobren detalles que no tienen que ver con el arranque de nuestra app en Kubernetes.

Kubernetes - Log y Debug

Para este caso, tenemos a disposición el comando:

```
$ kubectl get events
```

Nota: debemos recordar que el comando `get events` recibe un namespace, por ejemplo:

```
$ kubectl get events --namespace=my-namespace
```

Esto retornará los eventos de los pods incluidos en el namespace señalado.

Kubernetes - Log y Debug

Sin embargo, también podemos usar la opción general **--all-namespaces** como argumento del comando.

Kubernetes - Log y Debug

Formateando la información

Algunas veces queremos ver información como detalles de un pod, logs y últimos eventos de una forma más cómoda, para ello, Kubernetes nos permite formatear el output del comando get pod a YAML.

Para poder hacer esto, basta con ejecutar:

```
kubectl get pod <nombre-del-pod> -o yaml
```

Kubernetes - Log y Debug

Accediendo a logs del contenedor

Para acceder a los logs de un contenedor, podemos ejecutar:

```
$ kubectl logs <pod-name> -c <init-container-2>
```

Kubernetes - Log y Debug

Más recursos para debug y troubleshooting

<https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/>

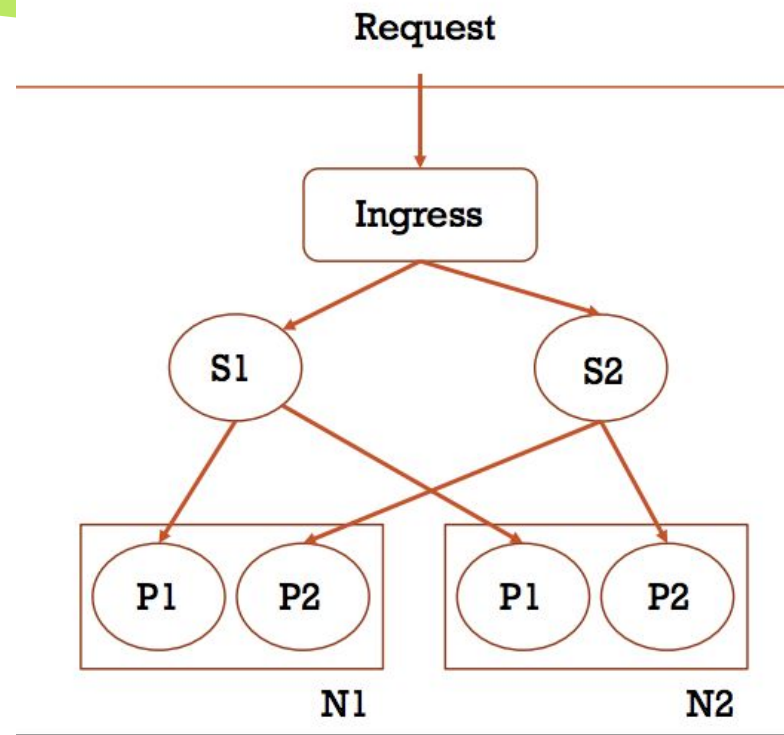
Kubernetes - Ingress

Puede darse el caso en el que los servicios desplegados en un clúster no sean accesibles de forma directa desde su ip a través de un navegador, para estos casos, suele requerirse de un proxy, una puerta o un proxy inverso para temas de redirección y apertura más allá del clúster.

Justo aquí nos ayuda Ingress, en estos escenarios, nuestra app dentro de un pod desplegado en Kubernetes es lo que está esperando sentado tras la puerta, el servicio es aquello que está justo tras la puerta e ingress sería la puerta en sí que nos permite alcanzar al servicio

Kubernetes - Ingress

S=Service, P=Pod, N=Node.



Kubernetes - Ingress

Por lo general, Ingress viene como un addon de plataformas como Minikube, por lo que podríamos habilitarlo y comenzar a usarlo con el comando:

```
> minikube addons enable ingress
```

Kubernetes - Helm

Como hemos visto, tenemos varias opciones a la hora de definir nuestras aplicaciones o la estructura de nuestros contenedores que serán desplegados en servidores o instancias de kubernetes, sin embargo, para esta última, también sería útil tener un gestor de paquetes como dockerhub para ahorrarnos la definición de tantos archivos a la hora de hacer un despliegue rápido, justo eso nos da Helm, es básicamente un gestor de paquetes para Kubernetes.

Kubernetes - Helm

Para instalarlo podemos ejecutar:

```
> curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > get_helm.sh  
> chmod 700 get_helm.sh  
> ./get_helm.sh
```

Una vez instalado, iniciamos con:

```
> helm init
```

Kubernetes - Helm

A partir de aquí podemos realizar búsquedas con el comando search, por ejemplo:

```
> helm search
```

E instalar paquetes con:

```
> helm install stable/jenkins
```

Ya con esto, Helm se encarga de desplegar nuestro servicio, solo nos quedaría ubicarlo y entrar en él: `kubectl get services`

Ventajas de usar Docker



Retorno de la inversión y ahorro de costes

Casos de uso e integración

Estandarización y productividad

Eficiencia de imágenes de contenedor

Casos de uso e integración

Compatibilidad y mantenimiento más fácil

Despliegue y escalabilidad rápidos

Plataformas multi-cloud

Aislamiento

Casos de uso e integración

Seguridad

