PROYECTO

GESTIÓN DE CONTENEDORES DOCKER - KUBERNETES

Proyecto y formación en centros de trabajo
2º ASIR
María Cabrera Gómez de la Torre

Índice de contenido

INTRODUCCIÓN A DOCKER 3
UN POCO DE HISTORIA....3
REQUISITOS MÍNIMOS....3
CARACTERÍSTICAS....4
VENTAJAS Y DESVENTAJAS....5
USOS Y RECOMENDACIONES....6
ARQUITECTURA....6
COMPONENTES....7
DIFERENCIAS CON LAS MÁQUINAS VIRTUALES....9
INSTALACIÓN DE DOCKER....10
- Instalación y primeros pasos en Windows....10

INSTALACIÓN EN UBUNTU 14.04....17
- CONFIGURACIÓN OPCIONAL....19

COMENZANDO CON DOCKER....23
- PRINCIPALES COMANDOS DOCKER 23
- IMÁGENES....26
  - TRABAJANDO CON IMÁGENES....27
- EJECUCIÓN DE CONTENEDORES....31
  - COMANDO RUN....31
- MODO INTERACTIVO....33
- CREANDO IMÁGENES DOCKER....35
- DETACHED o BACKGROUND....38
- MAPEO DE PUERTOS....40
- DOCKERHUB....42
- VOLÚMENES....45
- VARIABLES DE ENTORNO....46
- CONFIGURACIÓN DE RED....46
- ELIMINAR CONTENEDORES....47

DOCKERFILE....48
- CONSTRUIR EL CONTENEDOR....48
- DOCKERFILE COMANDOS....49
- EJEMPLO DOCKERFILE....55
- LIMITACIÓN DE RECURSOS....58
- INICIO AUTOMÁTICO....59

DOCKER-MACHINE....60
BIBLIOGRAFÍA....65
KUBERNETES....66
- CARACTERÍSTICAS....66
- ARQUITECTURA....67
- KUBERNETES NODE....69
- OTROS CONCEPTOS....72
- REDES EN KUBERNETES....73
- VOLÚMENES EN KUBERNETES....74
- VOLÚMENES PERSISTENTES....78
- INSTALACIÓN....81
- INSTALACIÓN EN CENTOS....82

CREANDO PODS y RC..92
CREANDO SERVICIOS..102
BIBLIOGRAFÍA KUBERNETES...105

INTRODUCCIÓN A DOCKER

Docker es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones dentro de “contenedores”.

Éste contenedor empaqueta todo lo necesario para que uno o más procesos (servicios o aplicaciones) funcionen: código, herramientas del sistema, bibliotecas del sistema, dependencias, etc. Ésto garantiza que siempre se podrá ejecutar, independientemente del entorno en el que queramos desplegarlo. No hay que preocuparse de qué software ni versiones tiene nuestra máquina, ya que nuestra aplicación se ejecutará en el contenedor,.

UN POCO DE HISTORIA

Salomon Hykes comenzó Docker comenzó como un proyecto interno dentro de dotCloud, empresa enfocada a PaaS (plataforma como servicio). Fué liberado como código abierto en marzo de 2013.

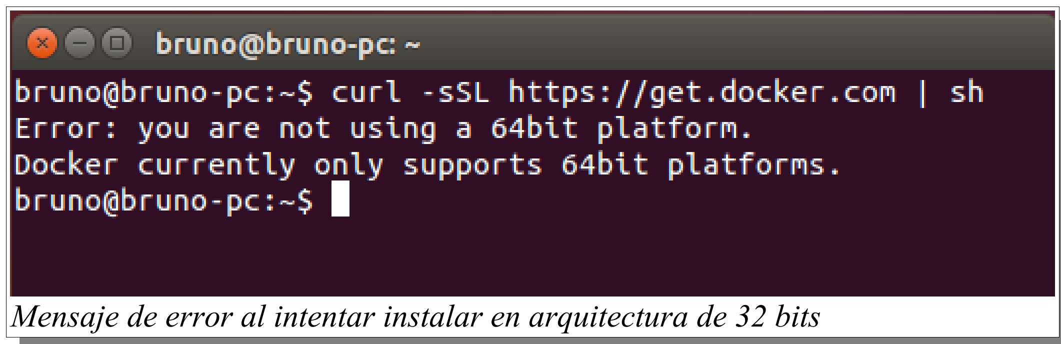
Con el lanzamiento de la versión 0.9 (en marzo de 2014) Docker dejó de utilizar LXC como entorno de ejecución por defecto y lo reemplazó con su propia librería, libcontainer (escrita en Go), que se encarga de hablar directamente con el kernel.

Actualmente es uno de los proyectos con más estrellas en GitHub, con miles de bifurcaciones (forks) y miles de colaboradores.

REQUISITOS MÍNIMOS

Docker funciona de forma nativa en entornos Linux a partir de la versión 3.8 del Kernel. Algunos Kernels a partir de la versión 2.6.x y posteriores podrían ejecutar Docker, pero los resultados pueden variar, por lo que oficialmente no está soportado.

Otro requisito es que sólo está preparado para arquitecturas de 64 bits (actualmente x86_64 y amd64).



Mensaje de error al intentar instalar en arquitectura de 32 bits

Para usar Docker en entornos Windows o MAC han creado una herramienta, **Boot2Docker**, que no es más que una máquina virtual ligera de Linux con Docker ya instalado. Dicha imagen la arrancamos con Virtualbox, Vmware o con la herramienta que tengamos instalada.

Otra manera es con un instalador "todo en uno" para MAC y Windows. Este instalador trae un cliente para Windows, la imagen de una máquina virtual Linux, Virtualbox y msys-git unix tools.

CARACTERÍSTICAS

Las principales características de Docker son:

- ***Portabilidad***: el contenedor Docker podemos desplegarlo en cualquier sistema, sin necesidad de volver a configurarlo o realizar las instalaciones necesarias para que la aplicación funcione, ya que todas las dependencias son empaquetadas con la aplicación en el contenedor.
- ***Ligereza***: los contenedores Docker sólo contienen lo que las diferencia del sistema operativo en el que se ejecutan, no se virtualiza un SO completo.
- ***Autosuficiencia***: un contenedor Docker no contiene todo un sistema operativo completo, sólo aquellas librerías, archivos y configuraciones necesarias para desplegar las funcionalidades que contenga.

VENTAJAS Y DESVENTAJAS

Usar contenedores Docker permite a des arrolladores y administradores de sistemas probar aplicaciones o servicios en un entorno seguro e igual al de producción, reduciendo los tiempos de pruebas y adaptaciones entre los entornos de prueba y producción.

Las principales ventajas de usar contenedores Docker son:

- Las instancias se inician en pocos segundos.
- Son fácilmente replicables.
- Es fácil de automatizar y de integrar en entornos de integración continua.
- Consumen menos recursos que las máquinas virtuales tradicionales.
- Mayor rendimiento que la virtualización tradicional ya que corre directamente sobre el Kernel de la máquina en la que se aloja, evitando al hypervisor.
- Ocupan mucho menos espacio.
- Permite aislar las dependencias de una aplicación de las instaladas en el host.
- Existe un gran repositorio de imágenes ya creadas sobre miles de aplicaciones, que además pueden modificarse libremente.

Por todo esto Docker ha entrado con mucha fuerza en el mundo del desarrollo, ya que permite desplegar las aplicaciones en el mismo entorno que tienen en producción o viceversa, permite desarrollarlas en el mismo entorno que tendrán en producción.

Aunque también tiene algunas desventajas:

- Sólo puede usarse de forma nativa en entornos Unix con Kernel igual o superior a 3.8.
- Sólo soporta arquitecturas de 64 bits.
- Como es relativamente nuevo, puede haber errores de código entre versiones.

USOS Y RECOMENDACIONES

El uso de Docker está recomendado en:

- Entornos de integración continua, es decir, cuando el paso de desarrollo a producción en un proyecto sea lo más a menudo posible, para así poder detectar fallos cuanto antes.
- Para garantizar la integridad de las aplicaciones en diferentes entornos.
- Cuando necesitemos tener entornos fácilmente desplegables, portables y desechables.
- Cuando necesitemos un entorno fácilmente escalable.

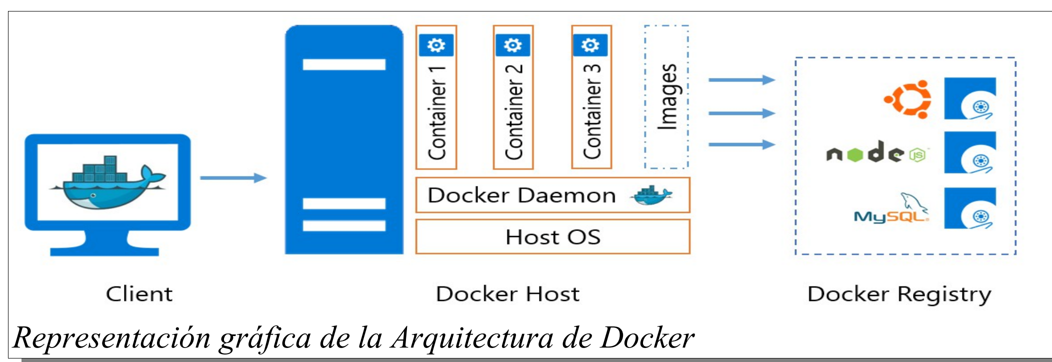
ARQUITECTURA

Docker usa una arquitectura **cliente-servidor**. El cliente de Docker habla con el demonio de Docker que hace el trabajo de crear, correr y distribuir los contenedores. Ambos pueden ejecutarse en el mismo sistema, o se puede conectar un cliente a un demonio Docker remoto. El cliente Docker y el demonio se comunican vía sockets o a través de una RESTfull API. (imagen pagina oficial).

Explicamos un poco por orden la arquitectura o funcionamiento de Docker:

El cliente de Docker (Docker Client) es la principal interfaz de usuario para Docker. Él acepta comandos del usuario y se comunica con el demonio Docker.

El demonio Docker (Docker Engine) corre en una máquina anfitriona (host). El usuario no interactúa directamente con el demonio, en su lugar lo hace a través del cliente Docker.



Representación gráfica de la Arquitectura de Docker

El demonio Docker levanta los contenedores haciendo uso de las imágenes, que pueden estar en local o en el Docker Registry.

Cada contenedor se crea a partir de una imagen y es un entorno aislado y seguro dónde se ejecuta nuestra aplicación.

COMPONENTES

Según la documentación oficial, Docker tiene dos principales componentes:

- ***Docker***

  Plataforma open source de virtualización con contenedores.

- ***Docker Hub***

  Plataforma de Software como servicio (SaaS, Software-as-a-Service) para compartir y administrar contenedores Docker.

Pero también necesitamos conocer otros componentes y conceptos:

- ***Docker Engine***

Es el demonio que se ejecuta dentro del sistema operativo (Linux) y que expone una API para la gestión de imágenes, contenedores, volúmenes o redes. Sus funciones principales son:

- La creación de imágenes Docker.
- Publicación de imágenes en Docker Registry.
- Descarga de imágenes desde Docker Registry.
- Ejecución de contenedores usando las imágenes.
- Gestión de contenedores en ejecución (pararlo, arrancarlo, ver logs, ver estadísticas).

- ***Docker Client***

Cualquier software o herramienta que hace uso de la API del demonio Docker, pero suele ser el comando ***docker***, que es la herramienta de línea de comandos para gestionar Docker Engine.

Éste cliente puede configurarse para hablar con un Docker local o remoto, lo que permite administrar nuestro entorno de desarrollo local como nuestros servidores de producción.

- ***Docker Images***

Son plantillas de sólo lectura que contienen el sistema operativo base (más adelante entraremos en profundidad) dónde correrá nuestra aplicación, además de las dependencias y software adicional instalado, necesario para que la aplicación funcione correctamente. Las plantillas son usadas por Docker Engine para crear los contenedores Docker.

- ***Docker Registries***

Los registros de Docker guardan las imágenes. Pueden ser repositorios públicos o privados. El registro público lo provee el Hub de Docker, que sirve tanto imágenes oficiales cómo las subidas por usuarios con sus propias aplicaciones y configuraciones.

Así tenemos disponibles para todos los usuarios imágenes oficiales de las principales aplicaciones (MySQL, MongoDB, Apache, Tomcat, etc.), así cómo no oficiales de infinidad de aplicaciones y configuraciones.

DockerHub ha supuesto una gran manera de distribuir las aplicaciones. Es un proyecto open source que puede ser instalado en cualquier servidor. Además nos ofrecen un sistema SaaS de pago.

- ***Docker Containers***

El contenedor de Docker aloja todo lo necesario para ejecutar un servicio o aplicación. Cada contenedor es creado de una imagen base y es una plataforma aislada.

Un contenedor es simplemente un proceso para el sistema operativo, que se aprovecha de él para ejecutar una aplicación. Dicha aplicación sólo tiene visibilidad sobre el sistema de ficheros virtual del contenedor.

- ***Docker Compose***

Es otro proyecto open source que permite definir aplicaciones multi-contenedor de una manera sencilla. Es una alternativa más cómoda al uso del comando *docker run*, para trabajar con aplicaciones con varios componentes.

Es una buena herramienta para gestionar entornos de desarrollo y de pruebas o para processos de integración continua.

- ***Docker Machine***

Es un proyecto open source para automatizar la creación de máquinas virtuales con Docker instalado, en entornos Mac, Windows o Linux, pudiendo administrar así un gran número de máquinas Docker.

Incluye drivers para Virtualbox, que es la opción aconsejada para instalaciones de Docker en local, en vez de instalar Docker directamente en el host. Esto simplifica y facilita la creación o la eliminación de una instalación de Docker, facilita la actualización de la versión de Docker o trabajar con distintas instalaciones a la vez.

Usando el comando **docker-machine** podemos iniciar, inspeccionar, parar y reiniciar un host administrado, actualizar el Docker client y el Docker daemon, y configurar un cliente para que hable con el host anfitrión. A través de la consola de administración podemos administrar y correr comandos Docker directamente desde el host. Éste comando **docker-machine** automáticamente crea hosts, instala Docker Engine en ellos y configura los clientes Docker.

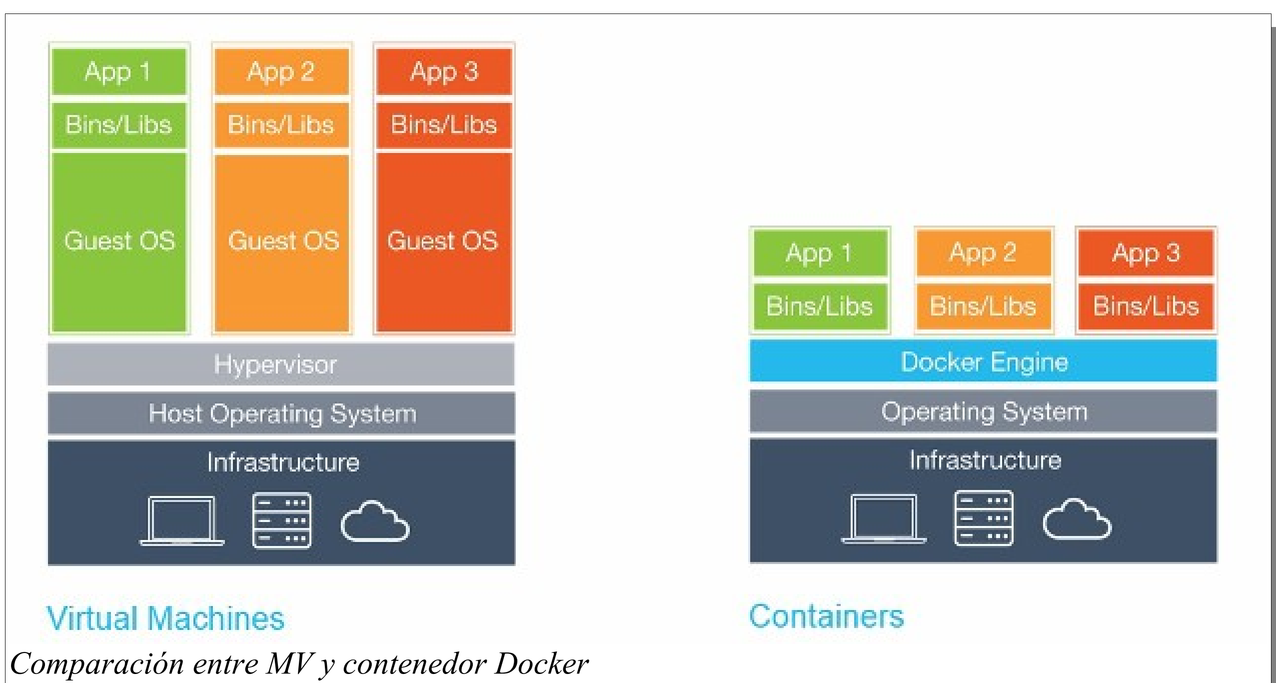
DIFERENCIAS CON LAS MÁQUINAS VIRTUALES

La principal diferencia es que una máquina virtual necesita tener virtualizado todo el sistema operativo, mientras que el contenedor Docker aprovecha el sistema operativo sobre el que se ejecuta, compartiendo el Kernel e incluso parte de sus bibliotecas. Para el SO anfitrión, cada contenedor no es más que un proceso que corre sobre el Kernel.

El concepto de contenedor o "virtualización ligera" no es nuevo. En Linux, LXC (Linux Containers) es una tecnología de virtualización a nivel de sistema operativo que utiliza dos características del Kernel, cgroups (que permite aislar y rastrear el uso de recursos) y namespaces (que permite a los grupos separarse, así no pueden verse unos a otros), para poder ejecutar procesos ligeros independientes en una sola máquina, aislados unos de otros, cada uno con su propia configuración de red. Ejemplos de esto son las jaulas de FreeBSD, OpenSolaris o Linux Vservers.

Otra diferencia es el tamaño, una máquina virtual convencional puede ocupar bastante, sin embargo los contenedores Docker sólo contienen lo que las diferencia del sistema operativo en el que se ejecutan, ocupando una media de 150-250 Mb.

En cuanto a recursos, el consumo de procesador y memoria RAM es mucho menor al no estar todo el sistema operativo virtualizado.



Comparación entre MV y contenedor Docker

INSTALACIÓN DE DOCKER

Podemos distinguir entre la instalación para el desarrollo local y la instalación en servidores en producción. Para los servidores en producción la mayoría de proveedores de servicio (AWS, GCE, Azure, Digital Ocean…) disponen de máquinas virtuales con versiones de Docker pre-instaladas.

En nuestro caso vamos a instalar Docker en Windows y en Ubuntu, pero en la página oficial vienen guías de instalación para múltiples distribuciones Linux (Ubuntu, Red Hat, CentOS, Fedora, Arch Linux, Gentoo, Oracle Linux, etc.).

Instalación y primeros pasos en Windows

Para Windows (versíón 7 o superiores) y Mac han creado una herramienta, "Boot2Docker", que es una máquina virtual de Linux con Docker ya instalado.

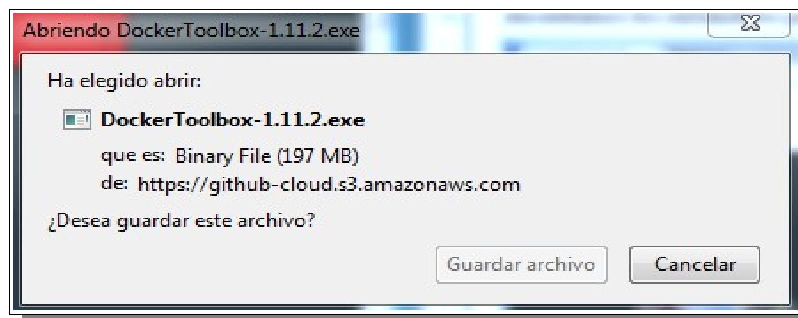
Vamos a la página oficial https://www.docker.com/products/docker-toolbox. En ella encontramos los instaladores para Mac y Windows.

Este toolbox contiene:

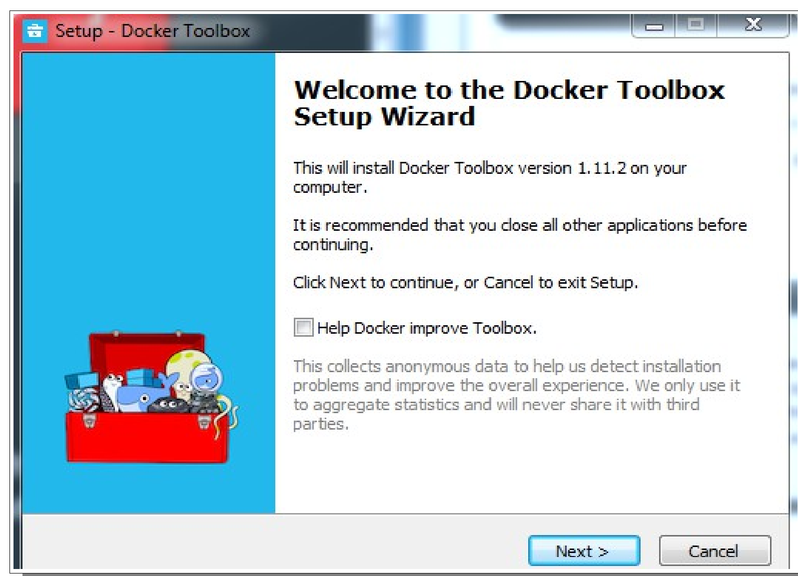
- Docker CLI client: para crear imágenes y contenedores con Docker Engine.
- Docker Machine: para poder correr comandos de Docker Engine en una terminal Windows.
- Docker Compose: para correr el comando docker-compose.
- Kitematic: entorno web para la creación y administración de los contenedores.

- Docker QuickStart: shell preconfigurada con un entorno de línea de comandos para acceder a la máquina virtual con Docker.
- Oracle VM VirtualBox: para crear una máquina virtual Linux.
- Git MSYS-git UNIX tools: versión de Git para Windows fácil de instalar.

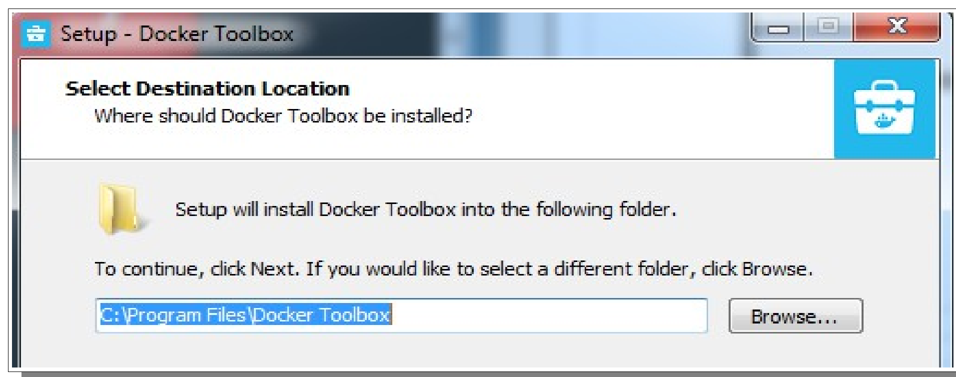
También podríamos optar por instalar sólo la máquina ligera de Linux preparada con Docker, boot2docker, pero en este caso descargamos el Toolbox.



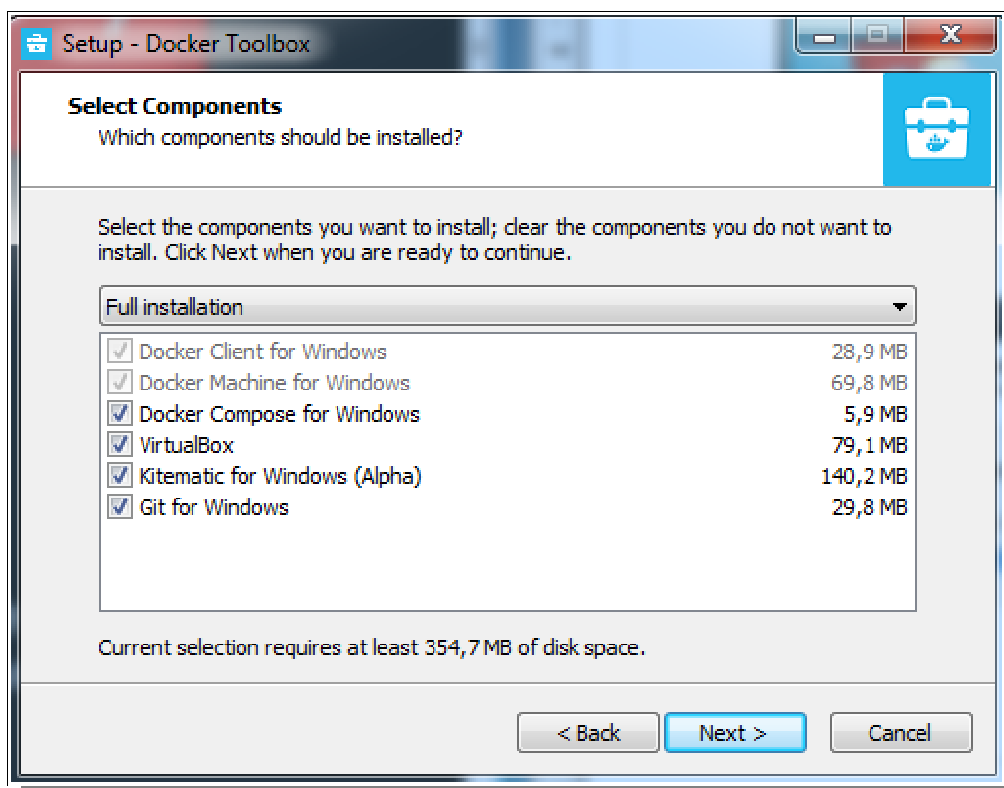
Ejecutamos el instalador, que recordemos sólo funcionará en arquitecturas de 64 bits. Para iniciar el instalador, que nos indica la versión que nos va a instalar.



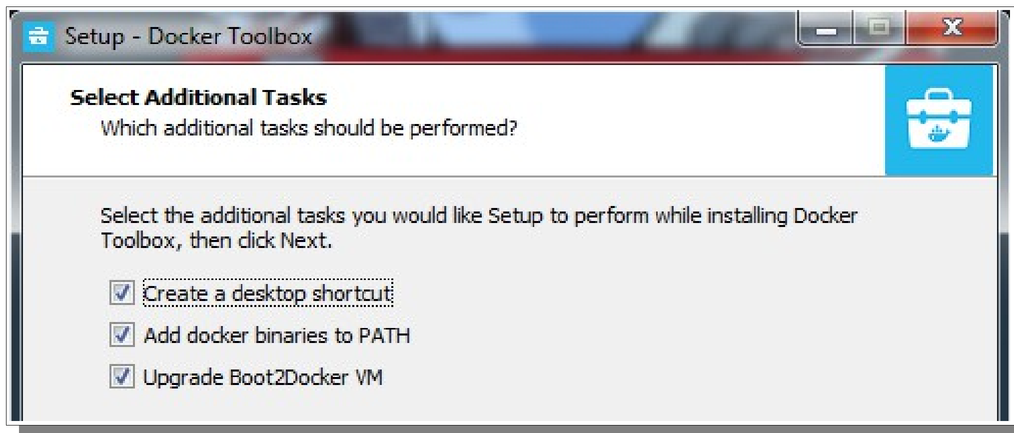
Continuamos con la instalación y dejamos que lo instale en la ruta por defecto.



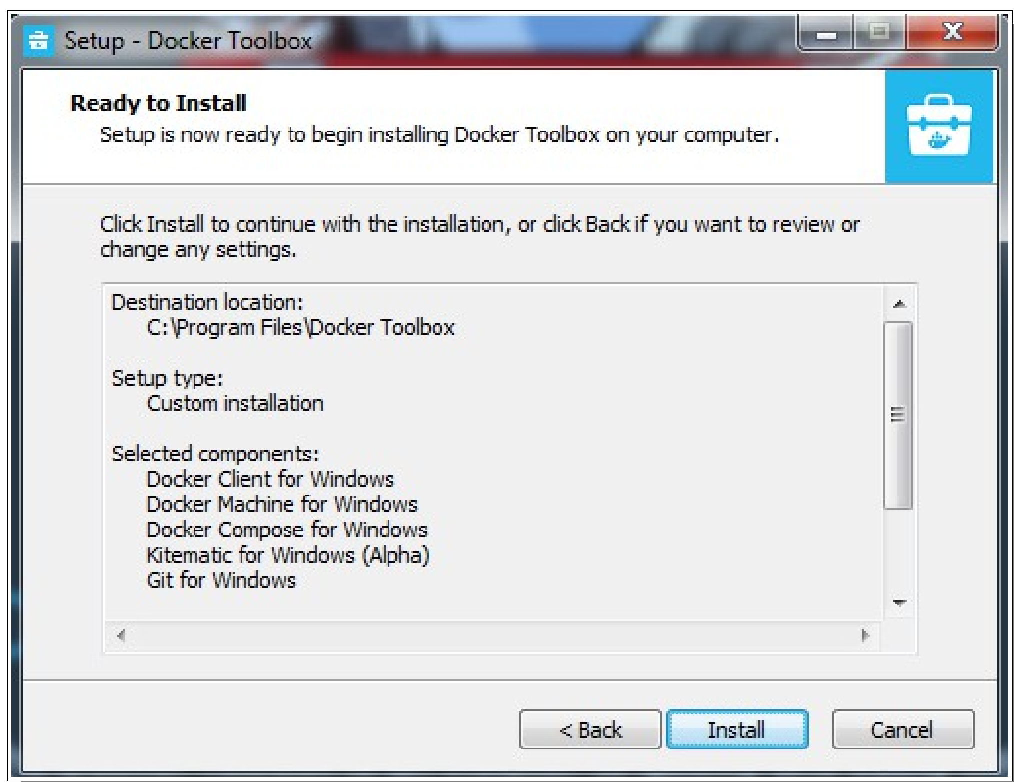
En el siguiente paso elegimos los componentes que queremos instalar, por ejemplo yo tengo ya instalado VirtualBox, por lo que lo desmarcaré.



Dejamos marcadas todas las opciones. Importante es marcar que nos actualice las variables de entorno añadiendo los binarios de Docker. Docker-machine requiere acceso a las herramientas que nos proporciona Git, por lo que si no lo tenemos en el path de Windows, no funcionará.



Nos resume la instalación.



Hasta aquí la instalación. Ahora vamos a ver cómo ejecutar nuestra máquina.

Ejecutamos Docker Quickstart Terminal.

Nos fijamos en lo que va haciendo.

```
MINGW64:/c/Users/Maria
Creating CA: C:\Users\Maria\.docker\machine\certs\ca.pem
Creating client certificate: C:\Users\Maria\.docker\machine\certs\cert.pem
Running pre-create checks...
(default) You are using version 4.3.6r91406 of VirtualBox. If you encounter issu
es, you might want to upgrade to version 5 at https://www.virtualbox.org
Creating machine...
(default) Copying C:\Users\Maria\.docker\machine\cache\boot2docker.iso to C:\Use
rs\Maria\.docker\machine\machines\default\boot2docker.iso...
(default) Creating VirtualBox VM...
(default) Creating SSH key...
(default) Starting the VM...
(default) Check network to re-create if needed...
(default) Windows might ask for the permission to create a network adapter. Some
times, such confirmation window is minimized in the taskbar.
(default) Found a new host-only adapter: "VirtualBox Host-Only Ethernet Adapter
#2"
(default) Windows might ask for the permission to configure a network adapter. S
ometimes, such confirmation window is minimized in the taskbar.
(default) Windows might ask for the permission to configure a dhcp server. Somet
imes, such confirmation window is minimized in the taskbar.
(default) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this vi
rtual machine, run: C:\Program Files\Docker Toolbox\docker-machine.exe env defau
lt
```

1.- Primero crea los certificados (esto sólo lo hace la primera vez que lo ejecutamos).

2.- Copia la imagen boot2docker.iso de la caché al directorio dónde guarda las máquinas.

3.- Crea la máquina virtual en VirtualBox

4.- Crea las claves ssh.

5.- Inicia la máquina virtual.

6.- Hace un chequeo de la red para recrearla si es necesario.

7.- Configura la red (adaptador, dhcp y espera a recibir una IP.

8.- Crea la instancia y la aprovisiona con boot2docker.

9.- Ajusta la configuración de Docker en el demonio remoto.

10.- Chequea la conexión con Docker y nos dice que la máquina está lista y corriendo.

También nos dice la ruta al comando docker-machine para que nos conectemos al cliente Docker y ya tenemos una shell de la máquina.

Si entramos en VirtualBox vemos que nos ha creado una máquina Linux con el nombre default, que está corriendo.

Ahora es en esta máquina dónde podemos ejecutar nuestros contenedores Dockers y podemos acceder a ella como a cualquier máquina virtual de VirtualBox.

```
default [Corriendo] - Oracle VM VirtualBox
Máquina  Ver  Dispositivos  Ayuda
    inet6 fe80::a00:27ff:fe6e:7048/64 scope link
       valid_lft forever preferred_lft forever
-------------------
Linking /etc/docker to /var/lib/boot2docker for persistence
Need TLS certs for default,127.0.0.1,10.0.2.15,192.168.99.100
-------------------
Finished boot2docker init script...
login[1349]: root login on 'tty1'
Boot2Docker version 1.11.2, build HEAD : a6645c3 - Wed Jun  1 22:59:51 UTC 2016
Docker version 1.11.2, build b9f10c9
root@default:~# _
CTRL DERECHA
```

Probamos a ejecutar un par de comandos Docker.

```
root@default:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
root@default:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
root@default:~# _
```

Cómo a partir de aquí la ejecución de los comandos es igual que en Linux, será en los próximos capítulos dónde profundizaremos en ellos. Ahora vamos a ver la instalación en Ubuntu.

INSTALACIÓN EN UBUNTU 14.04

En la documentación oficial podemos ver las distribuciones de Ubuntu que están soportadas. Aquí vamos a instalar Docker en Ubuntu Trusty 14.04 (LTS).

Primero vamos a comprobar los requisitos, que son arquitectura de 64 bits y un Kernel igual o superior a 3.10.

```
root@docker:~# uname -a
Linux docker 3.13.0-65-generic #105-Ubuntu SMP Mon Sep 21 18:50:58 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
```

Los pasos a seguir para la instalación son:

1.- Actualizamos los paquetes del sistema.

```
root@docker:~# apt-get update
```

2.- Instalamos los paquetes para https y certificados CA.

```
root@docker:~# apt-get install apt-transport-https ca-certificates
```

3.- Añadimos una nueva clave GPG

```
root@docker:~# sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

4.- Abrimos /etc/apt/sources.list.d/docker.list. Si apt-get no existe lo creamos y si existe borramos su contenido. Y añadimos el repositorio:

```
deb https://apt.dockerproject.org/repo ubuntu-trusty main
```

5.- Actualizamos la lista de paquetes.

```
root@docker:~# apt-get update
```

6.- Si tuviéramos una versión antigua debemos eliminarla.

```
root@docker:~# apt-get purge lxc-docker
```

7.- Comprobamos el candidato para la instalación.

```
root@docker:~# apt-cache policy docker-engine
docker-engine:
  Installed: (none)
  Candidate: 1.11.2-0~trusty
  Version table:
     1.11.2-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trus
     1.11.1-0~trusty 0
```

8.- La documentación oficial recomienda para nuestra versión de Ubuntu, instalar el paquete **linux-image-extra**, que permite utilizar los drivers de almacenamiento aufs (AnotherUnionFS), que es una versión alternativa de UnionFS, un servicio de archivos que implementa una unión para montar sistemas de archivos Linux.

Y también recomienda instalar el paquete **apparmor**, que es una extensión de seguridad que provee de una variedad de políticas de seguridad para el Kernel de Linux. Es una alternativa a SELinux.

```
root@docker:~# apt-get install linux-image-extra-3.13.0.65-generic apparmor
```

9.- Instalamos Docker-Engine.

```
root@docker:~# apt-get install docker-engine
```

10.- Iniciamos el demonio docker

```
root@docker:~# service docker start
```

11.- Comprobamos que docker está instalado correctamente.

```
root@docker:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
a9d36faac0fe: Pull complete
Digest:
sha256:e52be8ffeeb1f374f440893189cd32f44cb166650e7ab185fa7735b7dc48d619
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
```

```
This message shows that your installation appears to be working
correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub
account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Hasta aquí la instalación.

CONFIGURACIÓN OPCIONAL

La documentación oficial nos ofrece una serie de configuraciones y procedimientos para que Ubuntu trabaje mejor con Docker.

1.- Crear el grupo Docker si no existe.

El demonio Docker se comunica a través del socket de Unix en vez de por un puerto TCP. Por defecto el socket Unix es propiedad del usuario root, por esta razón el demonio Docker siempre corre como el usuario root.

Para evitar tener que usar sudo cuando se utilice el comando docker, se crea un grupo llamado Docker y se añaden usuarios a el. Cuando el demonio docker se inicie con un usuario perteneciente al grupo Docker, lo hará con los permisos de lectura y escritura equivalentes a root.

```
root@docker:~# groupadd docker
root@docker:~# usermod -aG docker ubuntu
```

Reiniciamos la sesión del usuario Ubuntu y comprobamos.

```
ubuntu@docker:~$ docker run hello-world

Hello from Docker.
This message shows that your installation appears to be working
correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub
account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

ubuntu@docker:~$
```

2.- Ajustar la memoria y la swap.

Dice la documentación que a veces aparece un mensaje de warning de la memoria swap para cgroup.

```
WARNING: Your kernel does not support cgroup swap limit. WARNING: Your
kernel does not support swap limit capabilities. Limitation discarded.
```

Para evitar estos mensajes activamos la memoria de intercambio en nuestro sistema. Editamos el fichero **/etc/default/grub** y actualizamos el valor **GRUB_CMDLINE_LINUX:**

```
GRUB_DEFAULT=0

GRUB_HIDDEN_TIMEOUT=0

GRUB_HIDDEN_TIMEOUT_QUIET=true
```

```
GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0"
GRUB_CMDLINE_LINUX=""
```

Y lo dejamos:

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

Luego actualizamos GRUB y reiniciamos el sistema.

```
root@docker:/home/ubuntu# update-grub
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.13.0-65-generic
Found initrd image: /boot/initrd.img-3.13.0-65-generic
done
```

3.- Configurar las políticas del Firewall.

Docker usa un bridge para gestionar las redes de contenedores, por lo que si tenemos un Firewall (UFW) que por defecto elimine el tráfico de forwarding, habrá que configurarlo adecuadamente.

Comprobamos el estado de nuestro firewall.

root@docker:/home/ubuntu# ufw status

Status: inactive

En este caso está inactivo. Si estuviera activo los pasos a seguir serían los siguientes.

a.- Editar el fichero **`/etc/default/ufw`** y actualizar la siguiente política a **"ACCEPT"**.

```
DEFAULT_FORWARD_POLICY=”ACCEPT”
```

b.- Reiniciar el Firewall y permitir las conexiones entrantes en el puerto de Docker.

```
$ sudo ufw reload
$ sudo ufw allow 2375/tcp
```

4.- Configurar un servidor DNS para Docker.

Sistemas como Ubuntu generalmente usan la 127.0.0.1 como servidor DNS por defecto en el fichero **/etc/resolv.conf**. El NetworkManager también configura el dnsmasq para que use el servidor local.

Al iniciar contenedores en máquinas con esta configuración, Docker nos mostrará la siguiente advertencia:

```
WARNING: Local (127.0.0.1) DNS resolver found in resolv.conf and containers
can't use it. Using default external servers : [8.8.8.8 8.8.4.4]
```

Esto ocurre porque los contenedores Docker no usan el DNS local, usan uno externo.

Para evitar esta advertencia podemos especificar un servidor DNS para que lo usen los contenedores. Esto se hace editando el fichero /etc/default/docker y especificando uno o más servidores DNS en la siguiente línea:

```
DOCKER_OPTS=”--dns 8.8.8.8”
```

Y reiniciamos el demonio Docker.

Si usáramos NetworkManager con dnsmasq, tendríamos que deshabilitarlo comentando en el fichero **/etc/NetworkManager/NetworkManager.conf** la línea:

```
# dns=dnsmasq
```

Y reiniciando NetworkManager y Docker.

5.- Configurar Docker para iniciar en el arranque.

Ubuntu usa systemd como su gestor de arranque. Para configurar el demonio de Docker para arrancar en el inicio:

```
$ systemctl enable docker
```

COMENZANDO CON DOCKER

Ya tenemos instalado Docker en nuestra máquina anfitriona cuyo SO es Ubuntu 14.04 LTS. Si escribimos "`docker version`" en la línea de comandos, nos da la versión del cliente y del servidor, además de la versión de la API y de Go (lenguaje de programación en el que está escrito Docker).

```
ubuntu@docker:~$ docker version
Client:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:47:50 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:47:50 2016
 OS/Arch:      linux/amd64
```

Con "docker -v" vemos sólo la versión de Docker instalada.

```
ubuntu@docker:~$ docker -v
Docker version 1.11.2, build b9f10c9
```

PRINCIPALES COMANDOS DOCKER

Antes de empezar a usar Docker en la máquina que hemos preparado, vamos a familiarizarnos con los comandos que nos ofrece Docker. Escribiendo ***docker*** en la terminal nos aparece una lista de las opciones disponibles:

- **attach**: para acceder a la consola de un contenedor que está corriendo.

- **build**: construye un contenedor a partir de un Dockerfile.
- **commit**: crea una nueva imagen de los cambios de un contenedor.
- **cp**: copia archivos o carpetas desde el sistema de ficheros de un contenedor a el host.
- **create**: crea un nuevo contenedor.
- **daemon**: crea un proceso demonio.
- **diff**: comprueba cambios en el sistema de ficheros de un contenedor.
- **events**: muestra eventos en tiempo real del estado de un contenedor.
- **exec**: ejecuta un comando en un contenedor activo.
- **export**: exporta el contenido del sistema de ficheros de un contenedor a un archivo .tar.
- **history**: muestra el historial de una imagen.
- **images**: lista las imágenes que tenemos descargadas y disponibles.
- **import**: crea una nueva imagen del sistema de archivos vacío e importa el contenido de un fichero .tar.
- **info**: muestra información sobre los contenedores, imágenes, versión de docker.
- **inspect**: muestra informaciones de bajo nivel del contenedor o la imagen.
- **kill**: detiene a un contenedor activo.
- **load**: carga una imagen desde un archivo .tar.
- **login**: para registrarse en un servidor de registro de Docker, por defecto "https://index.docker.io/v1/".
- **logout**: se desconecta del servidor de registro de Docker.
- **logs**: obtiene los registros de un contenedor.
- **network connect**: conecta un contenedor a una red.
- **network create**: crea una nueva red con un nombre especificado por el usuario.
- **network disconnect**: desconecta un contenedor de una red.
- **network inspect**: muestra información detallada de una red.
- **network ls**: lista todas las redes creadas por el usuario.
- **network rm**: elemina una o más redes.
- **pause**: pausa todos los procesos dentro de un contenedor.
- **port**: busca el puerto público, el cual está nateado, y lo hace privado.
- **ps**: lista los contenedores.

- **pull**: descarga una imagen o un repositorio del servidor de registros Docker.
- **push**: envía una imagen o un repositorio al servidor de registros de Docker.
- **rename**: renombra un contenedor existente.
- **restart**: reinicia un contenedor activo.
- **rm**: elimina uno o más contenedores.
- **rmi**: elimina una o más imágenes.
- **run**: ejecuta un comando en un nuevo contenedor.
- **save**: guarda una imagen en un archivo .tar
- **search**: busca una imagen en el índice de Docker.
- **start**: inicia un contenedor detenido.
- **stats**: muestra el uso de los recursos de los contenedores.
- **stop**: detiene un contenedor.
- **tag**: etiqueta una imagen en un repositorio.
- **top**: busca los procesos en ejecución de un contenedor.
- **unpause**: reanuda un contenedor pausado.
- **update**: actualiza la configuración de uno o más contenedores.
- **version**: muestra la versión de Docker instalada.
- **volume create**: crea un volumen.
- **volume inspect**: devuelve información de bajo nivel de un volumen.
- **volume ls**: lista los volúmenes.
- **volume rm**: elimina un volumen.
- **wait**: bloquea hasta detener un contenedor, entonces muestra su código de salida.

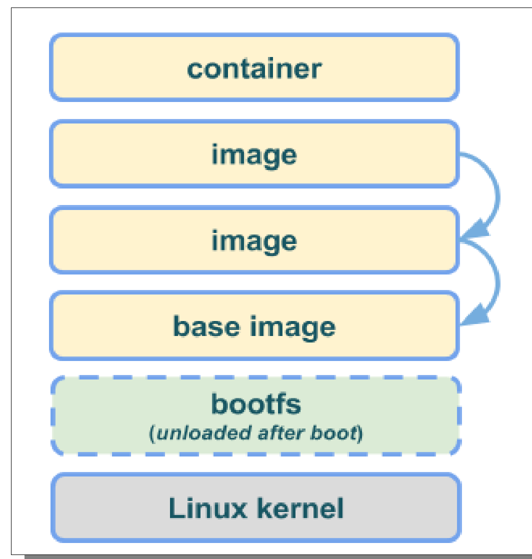
Y si queremos saber cómo funciona un comando concreto debemos escribir "**`docker COMMAND --help`**", por ejemplo vemos cómo usar el comando save:

```
ubuntu@docker:~$ docker save --help
Usage:     docker save [OPTIONS] IMAGE [IMAGE...]
Save one or more images to a tar archive (streamed to STDOUT by default)
  --help             Print usage
  -o, --output       Write to a file, instead of STDOUT
```

IMÁGENES

Las imágenes son plantillas de sólo lectura que usamos como base para lanzar un contenedor. Una imagen Docker se compone de un sistema de archivos en capas una sobre la otra. En la base tenemos un sistema de archivos de arranque, bootfs (parecido al sistema de archivos de Linux) sobre el que arranca la imagen base.

Cada imagen, también conocida como repositorio, es una sucesión de capas. Es decir, al arrancar un contenedor lo hacemos sobre una imagen, a la que llamamos imagen base. Con el contenedor corriendo, cada vez que realizamos un cambio en el contenedor Docker añade una capa encima de la anterior con los cambios , pero dichas modificaciones no serán persistentes, los cambios no los hacemos en la imagen (recordemos que es de sólo lectura), por lo que deberemos guardarlos creando una nueva imagen con los cambios.



Como vemos en la imagen cuando un contenedor se pone en marcha a partir de una imagen, Docker monta un sistema de ficheros de lectura escritura en la parte superior de las capas. Aquí es donde los procesos de nuestro contenedor Docker serán ejecutados (container).

Cuando Docker crea un contenedor por primera vez, la capa inicial de lectura y escritura esta vacía. Cuando se producen cambios, éstos se aplican a esta capa, por ejemplo, si desea cambiar un archivo, entonces ese archivo se copia desde la capa de sólo lectura inferior a la capa de lectura y escritura. Seguirá existiendo la versión de sólo lectura del archivo, pero ahora está oculto debajo de la copia.

TRABAJANDO CON IMÁGENES

Como hemos explicado anteriormente, los contenedores se construyen a partir de imágenes, que se pueden encontrar en local, en el Docker Hub o en repositorios privados. Más adelante veremos Docker Hub un poco más en profundidad.

Para comprobar las imágenes que tenemos en local tenemos que ejecutar "`docker images`".

```
ubuntu@docker:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
hello-world         latest              693bce725149        4 days ago
967 B
ubuntu@docker:~$
```

Ahora mismo sólo tenemos la imagen que se descargó cuándo comprobamos que se había instalado correctamente Docker.

Podemos buscar la imagen que queremos con el comando search. Por ejemplo si queremos saber que imágenes hay disponibles de Ubuntu.

```
ubuntu@docker:~$ docker search ubuntu
NAME                              DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
ubuntu                            Ubuntu is a Debian-based Linux operating s...   4082      [OK]
ubuntu-upstart                    Upstart is an event-based replacement for ...   64        [OK]
rastasheep/ubuntu-sshd            Dockerized SSH service, built on top of of...   28                   [OK]
torusware/speedus-ubuntu          Always updated official Ubuntu docker imag...   26                   [OK]
ubuntu-debootstrap                debootstrap --variant=minbase --components...   25        [OK]
ioft/armhf-ubuntu                 [ABR] Ubuntu Docker images for the ARMv7(a...   14                   [OK]
nickistre/ubuntu-lamp             LAMP server on Ubuntu                           7                    [OK]
nuagebec/ubuntu                   Simple always updated Ubuntu docker images...   5                    [OK]
nickistre/ubuntu-lamp-wordpress   LAMP on Ubuntu with wp-cli installed            5                    [OK]
nimmis/ubuntu                     This is a docker images different LTS vers...   4                    [OK]
maxexcloo/ubuntu                  Docker base image built on Ubuntu with Sup...   2                    [OK]
admiringworm/ubuntu               Base ubuntu images based on the official u...   1                    [OK]
darksheer/ubuntu                  Base Ubuntu Image -- Updated hourly             1                    [OK]
jordi/ubuntu                      Ubuntu Base Image                               1                    [OK]
esycat/ubuntu                     Ubuntu LTS                                      0                    [OK]
konstruktoid/ubuntu               Ubuntu base image                               0                    [OK]
webhippie/ubuntu                  Docker images for ubuntu                        0                    [OK]
lynxtp/ubuntu                     https://github.com/lynxtp/docker-ubuntu         0                    [OK]
life360/ubuntu                    Ubuntu is a Debian-based Linux operating s...   0                    [OK]
teamrock/ubuntu                   TeamRock's Ubuntu image configured with AW...   0                    [OK]
widerplan/ubuntu                  Our basic Ubuntu images.                        0                    [OK]
datenbetrieb/ubuntu               custom flavor of the official ubuntu base ...   0                    [OK]
ustclug/ubuntu                    ubuntu image for docker with USTC mirror        0                    [OK]
rallias/ubuntu                    Ubuntu with the needful                         0                    [OK]
uvatbc/ubuntu                     Ubuntu images with unprivileged user            0                    [OK]
ubuntu@docker:~$
```

Vemos una lista con las imágenes disponibles.

Para descargar una imagen debemos hacerlo con el comando **`pull`** (también podemos hacerlo con el el comando **run** como veremos más adelante, pero ésta es la forma correcta).

Sintaxis:

```
docker pull [options] NAME[:TAG] |
[REGISTRY_HOST[:REGISTRY_PORT]/]NAME[:TAG]
```

Donde las opciones disponibles son:

`-a, --all-tags`: descarga todas imágenes etiquetadas en el repositorio.

`--disable-content-trust=true`: se salta la verificación de la imagen.

`--help`: muestra ayuda sobre el comando.

Si no especificamos el tag se descargará la última versión. Vamos a probarlo.

```
ubuntu@docker:~$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
ac6ad7efd0f9: Pull complete
e7491a747824: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:46fb5d001b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
ubuntu@docker:~$
```

Si nos fijamos en la descarga vemos 5 líneas que ponen “pull complete”. Esto es que la imagen está formada por 5 capas o layers. Estas capas pueden ser reutilizadas por otras imágenes, que evitan así el tener que volver a descargarlas, por ejemplo si descargáramos otra imagen de Ubuntu.

En la descarga también vemos una línea que pone Digest. Éste código sirve si queremos asegurarnos de usaruna versión de la imagen en concreto, y no la última por ejemplo, cómo pasa si usamos el nombre y el tag.

Comprobamos las imágenes disponibles ahora.

```
ubuntu@docker:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              693bce725149        4 days ago          967 B
ubuntu              latest              2fa927b5cdd3        2 weeks ago         122 MB
ubuntu@docker:~$
```

Para ver los detalles de una imagen ejecutamos el siguiente comando:

```
ubuntu@docker:~$ docker inspect ubuntu
[
    {
                                                                        "Id":
"sha256:2fa927b5cdd31cdec0027ff4f45ef4343795c7a2d19a9af4f32425132a222330"
,
        "RepoTags": [
            "ubuntu:latest"
        ],
        "RepoDigests": [],
        "Parent": "",
        "Comment": "",
        "Created": "2016-05-27T14:15:02.359284074Z",
                                                                 "Container":
"b8bd6a8e8874a87f626871ce370f4775bdf598865637082da2949ee0f4786432",
        "ContainerConfig": {
            "Hostname": "914cf42a3e15",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) CMD [\"/bin/bash\"]"
            ],
                                                                     "Image":
"b873f334fa5259acb24cf0e2cd2639d3a9fb3eb9bafbca06ed4f702c289b31c0",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "1.9.1",
        "Author": "",
        "Config": {
            "Hostname": "914cf42a3e15",
            "Domainname": "",
```

```
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [],
            "Cmd": [
                "/bin/bash"
            ],
                                                                  "Image":
"b873f334fa5259acb24cf0e2cd2639d3a9fb3eb9bafbca06ed4f702c289b31c0",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 121989688,
        "VirtualSize": 121989688,
        "GraphDriver": {
            "Name": "aufs",
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:9436069b92a3ec23351313d6decc5fba2ce1cd52aac77dd8d
a3b03b3dfcb5382",
                "sha256:19429b698a2283bfaece7f590e438662e2078351a9ad077c6
d1f7fb12f8cd08d",
                "sha256:82b57dbc5385a2a0edd94dfc86f9e4d75487cce30250fec94
147abab0397f2b8",
                "sha256:737f40e80b7ff641f24b759d64f7ec489be0ef4e0c16a9780
b795dbe972b38d2",
                "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6c
ddfaf10ace3c6ef"
            ]
        }
    }
]
```


Una vez que tenemos disponible una imagen podemos ejecutar cualquier contenedor.

EJECUCIÓN DE CONTENEDORES

COMANDO RUN

Podemos crear un contenedor con el comando run.

Sintaxis:

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

Entre las opciones se encuentran (lista completa aquí):

`-a, --attach`: para conectarnos a un contenedor que está corriendo.

`-d, --detach`: corre un contenedor en segundo plano.

`-i, --interactive`: habilita el modo interactivo.

`--name`: le pone nombre a un contenedor.

Ahora vamos a ejecutar un contenedor sobre la imagen de ubuntu que tenemos descargada.

```
ubuntu@docker:~$ docker run ubuntu echo hello world
hello world
ubuntu@docker:~$
```

El comando **run** primero crea una capa del contenedor sobre la que se puede escribir y a continuación ejecuta el comando especificado. Con este comando hemos ejecutado un contenedor, sobre la imagen Ubuntu, que ha ejecutado el comando echo. Cuando ha terminado de ejecutar el comando que le hemos pedido se ha detenido. Los contenedores están diseñados para correr un único servicio, aunque podemos correr más si hiciera falta. Cuando ejecutamos un contenedor con run debemos especificarle un comando a ejecutar en él, y dicho contenedor sólo se ejecuta durante el tiempo que dura el comando que especifiquemos, funciona como un proceso.

Algo a tener en cuenta es que si la imagen que estamos poniendo en el comando run no la tuviéramos en local, Docker primero la descargaría y la guardaría en local y luego seguiría con la construcción capa por capa del contenedor.

Vamos a ver los contenedores que tenemos en nuestro host con el comando ps.

```
ubuntu@docker:~$ docker ps
CONTAINER ID          IMAGE                 COMMAND                 CREATED
STATUS                PORTS                 NAMES
```

No nos aparece ningún contenedor, y esto es porque el contenedor ya no está activo. Para ver todos los contenedores (activos e inactivos) usamos el flag -a.

```
ubuntu@docker:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND              CREATED             STATUS                      PORTS               NAMES
170c062a279c        ubuntu              "echo hello world"   3 minutes ago       Exited (0) 3 minutes ago                        serene_heyrovsky
18dfb4249654        hello-world         "/hello"             38 hours ago        Exited (0) 38 hours ago                         agitated_pasteur
1769cd61916d        hello-world         "/hello"             38 hours ago        Exited (0) 38 hours ago                         determined_blackwell
ubuntu@docker:~$
```

Vemos el id del contenedor, la imagen del contenedor, el comando que se ha ejecutado, cuando se creó, el estado, el mapeo de puertos y el nombre.

Éste ID es abreviado, para verlo completo tenemos que ejecutar:

```
ubuntu@docker:~$ docker ps -a --no-trunc
CONTAINER ID                                                       IMAGE               COMMAND
    NAMES
dfb581bd638ca0493c8122a1dc290727ab81bd0aa2f68a5ab55f3a07603f5639   ubuntu              "/bin/bash"
    tiny_turing
18dfb4249654af7b449648eca5d04935215f2f474e404467df53094d893bc37e   hello-world         "/hello"
    agitated_pasteur
1769cd61916d943e566e62dec1eda715e39b4c3464dd1244cd7da38a347d5e5d   hello-world         "/hello"
    determined_blackwell
ubuntu@docker:~$
```

También vemos que Docker genera automáticamente un nombre al azar por cada contenedor que creamos. Si queremos especificar un nombre en particular podemos hacerlo con el parámetro --name.

```
ubuntu@docker:~$ docker run --name prueba01 ubuntu echo prueba nombre
prueba nombre
```

Vemos el contenedor con el nombre que hemos puesto.

```
ubuntu@docker:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
8c5ab634d291        ubuntu              "echo prueba nombre"     4 seconds ago       Exited (0) 4 seconds ago                        prueba01
18dfb4249654        hello-world         "/hello"                 40 hours ago        Exited (0) 40 hours ago                         agitated_pasteur
1769cd61916d        hello-world         "/hello"                 40 hours ago        Exited (0) 40 hours ago                         determined_blackwell
ubuntu@docker:~$
```

MODO INTERACTIVO

Cómo hemos visto, cuando creamos un contenedor con **`run`**, debemos especificar un comando que se va a ejecutar y cuando se acabe su ejecución el contenedor se detendrá. Básicamente el contenedor se crea para ejecutar dicho comando.

Tenemos la opción de ejecutar un contenedor en modo interactivo con los flags:

-t: ejecuta una terminal.

-i: nos comunicamos con el contenedor en modo interactivo.

```
ubuntu@docker:~$ docker run -it ubuntu /bin/bash
root@dfb581bd638c:/# pwd
/
root@dfb581bd638c:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@dfb581bd638c:/#
```

Cómo vemos en la imagen estamos conectados al contenedor. Si abrimos otra terminal y ejecutamos un docker ps, vemos que tenemos el contenedor corriendo.

```
ubuntu@docker: ~
root@dfb581bd638c: / 117x11
122 MB
ubuntu@docker:~$ docker -it ubuntu /bin/bash
flag provided but not defined: -it
See 'docker --help'.
ubuntu@docker:~$ docker run -it ubuntu /bin/bash
root@dfb581bd638c:/# pwd
/
root@dfb581bd638c:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@dfb581bd638c:/#
ubuntu@docker: ~ 117x10
ubuntu@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
   NAMES
dfb581bd638c        ubuntu              "/bin/bash"         2 minutes ago       Up 2 minutes
   tiny_turing
ubuntu@docker:~$
```

Pero si nos salimos de la shell del contenedor, éste se detendrá.

```
ubuntu@docker: ~
ubuntu@docker: ~ 117x11
flag provided but not defined: -it
See 'docker --help'.
ubuntu@docker:~$ docker run -it ubuntu /bin/bash
root@dfb581bd638c:/# pwd
/
root@dfb581bd638c:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@dfb581bd638c:/# exit
exit
ubuntu@docker:~$
ubuntu@docker: ~ 117x10
ubuntu@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
   NAMES
ubuntu@docker:~$
```

Lo vemos con docker ps -a.

```
ubuntu@docker:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                    PORTS
               NAMES
dfb581bd638c        ubuntu              "/bin/bash"         4 minutes ago       Exited (0) About a minute ago
               tiny_turing
18dfb4249654        hello-world         "/hello"            2 days ago          Exited (0) 2 days ago
               agitated_pasteur
1769cd61916d        hello-world         "/hello"            2 days ago          Exited (0) 2 days ago
               determined_blackwell
ubuntu@docker:~$
```

El contenedor sigue ahí así que podemos reiniciarlo.

```
ubuntu@docker: ~ 168x45
ubuntu@docker:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                    PORTS               NAMES
dfb581bd638c        ubuntu              "/bin/bash"         56 minutes ago      Exited (0) 6 seconds ago                      tiny_turing
18dfb4249654        hello-world         "/hello"            2 days ago          Exited (0) 2 days ago                         agitated_pasteur
1769cd61916d        hello-world         "/hello"            2 days ago          Exited (0) 2 days ago                         determined_blackwell
ubuntu@docker:~$ docker start dfb
dfb
ubuntu@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
dfb581bd638c        ubuntu              "/bin/bash"         56 minutes ago      Up 4 seconds                            tiny_turing
ubuntu@docker:~$
```

Y conectarnos a él.

```
ubuntu@docker:~$ docker attach tiny_turing
root@dfb581bd638c:/#
```

O ejecutar con **exec** comandos dentro de un contenedor activo, por ejemplo una shell.

```
ubuntu@docker:~$ docker exec -it dfb /bin/sh
# exit
ubuntu@docker:~$ docker ps
CONTAINER ID          IMAGE                    COMMAND                  CREATED
STATUS               PORTS                    NAMES
dfb581bd638c          ubuntu                    "/bin/bash"            3 hours ago
Up 19 seconds                                 tiny_turing
```

Vemos que cuando salimos el contenedor no se detiene.

Podemos abrir otra terminal y parar el contenedor:

```
ubuntu@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
dfb581bd638c        ubuntu              "/bin/bash"         About an hour ago   Up About a minute
ubuntu@docker:~$ docker stop dfb581bd638c
dfb581bd638c
ubuntu@docker:~$
```

Pero no resulta muy funcional un contenedor que se pare cuando terminemos de actuar sobre él. Nos interesa que continúe ejecutándose con el servicio que queramos. Para ello está el modo detached, que veremos un poco más adelante.

CREANDO IMÁGENES DOCKER

Las imágenes, como hemos visto, son plantillas de sólo lectura, que usamos de base para lanzar contenedores. Por tanto lo que hagamos en el contenedor sólo persiste en ese contenedor, las modificaciones no las hacemos en la imagen.

Si queremos que dichos cambios sean permanentes, debemos crear una nueva imagen con el contenedor personalizado.

Vemos las imágenes que tenemos disponibles.

```
root@docker:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              693bce725149        9 days ago          967 B
ubuntu              latest              2fa927b5cdd3        2 weeks ago         122 MB
```

Vamos a realizar un ejemplo. Tenemos la imagen base de ubuntu. Lanzamos un contenedor con esa imagen base en el modo interactivo que vimos anterioramentey vamos a instalar una serie de paquetes.

```
root@docker:~# docker run -it --name maria01 ubuntu /bin/bash
root@c906d56c7aa5:/# apt-get update
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [94.5 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [94.5 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
…
…
```

Instalamos por ejemplo git.

```
root@c906d56c7aa5:/# apt-get install git
```

Ahora vamos a guardar los cambios realizados en la imagen. Tenemos que salir del contenedor y ejecutar el comando “**`commit`**”.

Cuando salimos de un contenedor interactivo éste se detiene. Lo vemos con **`docker ps -a`**

```
root@docker:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                          PORTS               NAMES
c906d56c7aa5        ubuntu              "/bin/bash"         6 minutes ago       Exited (0) About a minute ago                       maria01
fc6e05dedc96        ubuntu              "/bin/bash"         26 minutes ago      Exited (127) 19 minutes ago                         2plano
e4a81bcfcd80        hello-world         "/hello"            50 minutes ago      Exited (0) 50 minutes ago                           tiny_engelbart
```

Ahora mismo este contenedor está formado por la capa con la imagen base y la capa en la que hemos instalado git.

Para poder utilizar esta imagen con los cambios, tenemos que crear una nueva imagen, con el comando:

```
docker commit -m "Instalado Git" -a "Maria Cabrera" c906d56c7aa5 git/ubuntu:v1
```

Con **commit** creamos una nueva imagen en nuestro repositorio local.

-m: añadimos un comentario.

-a: autor de la imagen.

c906d56c7aa5: identificador del contenedores.

git/ubuntu:v1: el nombre que le damos a la imagen.

Al crearla nos devuelve un identificador único de imagen. La podemos ver en la lista de imágenes ahora.

```
root@docker:~# docker commit -m "Instalado Git" -a "Maria Cabrera" c906d56c7aa5 git/ubuntu:v1
sha256:8a1408e736fd05e4b13ff3b1f85535d7659b8254cd3beb583f7f2d8546aa6b02
root@docker:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
git/ubuntu          v1                  8a1408e736fd        8 seconds ago       250.2 MB
hello-world         latest              693bce725149        9 days ago          967 B
ubuntu              latest              2fa927b5cdd3        3 weeks ago         122 MB
root@docker:~#
```

A partir de aquí podemos crear un contenedor con esta nueva imagen como base y ya tendrá instalado git. Lo comprobamos:

```
root@docker:~# docker run -it --name git git/ubuntu:v1 /bin/bash
root@bfa268acf89b:/# dpkg -l | grep git
ii  findutils                4.6.0+git+20160126-2            amd64
es for finding files--find, xargs
ii  git                      1:2.7.4-0ubuntu1                amd64
calable, distributed revision control system
ii  git-man                  1:2.7.4-0ubuntu1                all
calable, distributed revision control system (manual pages)
ii  libasn1-8-heimdal:amd64  1.7~git20150920+dfsg-4ubuntu1   amd64
 Kerberos - ASN.1 library
ii  libgssapi3-heimdal:amd64 1.7~git20150920+dfsg-4ubuntu1   amd64
 Kerberos - GSSAPI support library
```

DETACHED o BACKGROUND

Nos permite ejecutar un contenedor en segundo plano y poder correr comandos sobre el mismo en cualquier momento mientras esté en ejecución. Lo hacemos con el flag -d. Se dice que es un contenedor demonizado y se ejecutará indefinidamente.

Vamos a ver un ejemplo. Primero creamos un contenedor basado en la imagen ubuntu que hemos descargado en modo interactivo.

```
root@docker:~# docker run -it --name 2plano ubuntu /bin/bash
root@fc6e05dedc96:/#
```

Y le vamos a instalar un servidor web.

```
root@docker:~# docker run -it --name apache ubuntu /bin/bash
root@2ca26b7f9d5c:/# apt-get install apache2
```

Ahora creamos una nueva imagen del contenedor con apache instalado.

```
root@docker:~# docker commit -m "Instalado Apache" -a "Maria Cabrera" 2ca26b7f9d5c  apache/ubuntu:v1
sha256:defab75e4eb991475e943513f52002b76f5eaa65b5d69c998228c469e99093ab
```

Ahora podemos arrancar un contenedor en segundo plano.

```
root@docker:~# docker run -d --name server apache/ubuntu:v1 /usr/sbin/apache2ctl -DFOREGROUND
9bb15ff58f5da24e2513e69f0c4301577eb10dd07c173729d472a94b8b20234d
```

Vamos a hacer que el servicio apache no se detenga con -D y que el contenedor se ejecute en segundo plano con -d.

Comprobamos nuestros contenedores.

```
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
9bb15ff58f5d         apache/ubuntu:v1     "/usr/sbin/apache2ctl"    24
seconds ago      Up 24 seconds                              server
```

Podemos ver los procesos que se están ejecutando en un contenedor con “top”.

```
root@docker:~# docker top web_server
UID                     PID                     PPID                    C
STIME                   TTY                     TIME                    CMD
root                    10543                   10529                   0
15:19                  ?                    00:00:00             /bin/sh
/usr/sbin/apache2ctl -DFOREGROUND
root                    10564                   10543                   0
15:19                          ?                                  00:00:00
/usr/sbin/apache2 -DFOREGROUND
www-data                10565                   10564                   0
15:19                          ?                                  00:00:00
/usr/sbin/apache2 -DFOREGROUND
www-data                10566                   10564                   0
15:19                          ?                                  00:00:00
/usr/sbin/apache2 -DFOREGROUND
```

Una vez que hemos creado el contenedor y lo tenemos corriendo en segundo plano, podemos conectarnos a él mediante el comando exec, que ejecuta un proceso en el contenedor.

```
root@docker:~# docker exec -it server /bin/bash
root@5659f2d7e356:/#
```

Vemos que al salirnos del contenedor, éste no se detiene:

```
root@5659f2d7e356:/# exit
exit
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
5659f2d7e356          apache/ubuntu:v1      "/usr/sbin/apache2ctl"    4
minutes ago       Up 4 minutes                               server
```

MAPEO DE PUERTOS

Para que la aplicación que nos está sirviendo nuestro contenedor, por ejemplo el servidor web instalado en el contenedor anteior, tenemos que mapear los puertos. Es decir, al crear el contenedor, éste no está disponible al exterior.

Para que lo esté tenemos que redireccionar el puerto 22 del contenedor a uno de nuestra máquina. Usamos para ello el flag -p.

Vamos a ejecutar un contenedor basado en la imagen apache, en segundo plano y con la redirección de puertos activa.

Podemos hacer la redirección de puertos con las opciones:

-p: especificamos a qué puerto queremos la redirección.

-P: le dice a Docker que si expone algún tipo de puerto haga el fordwarding a un puerto aleatorio. Este se una cuando usamos un Dockerfile.

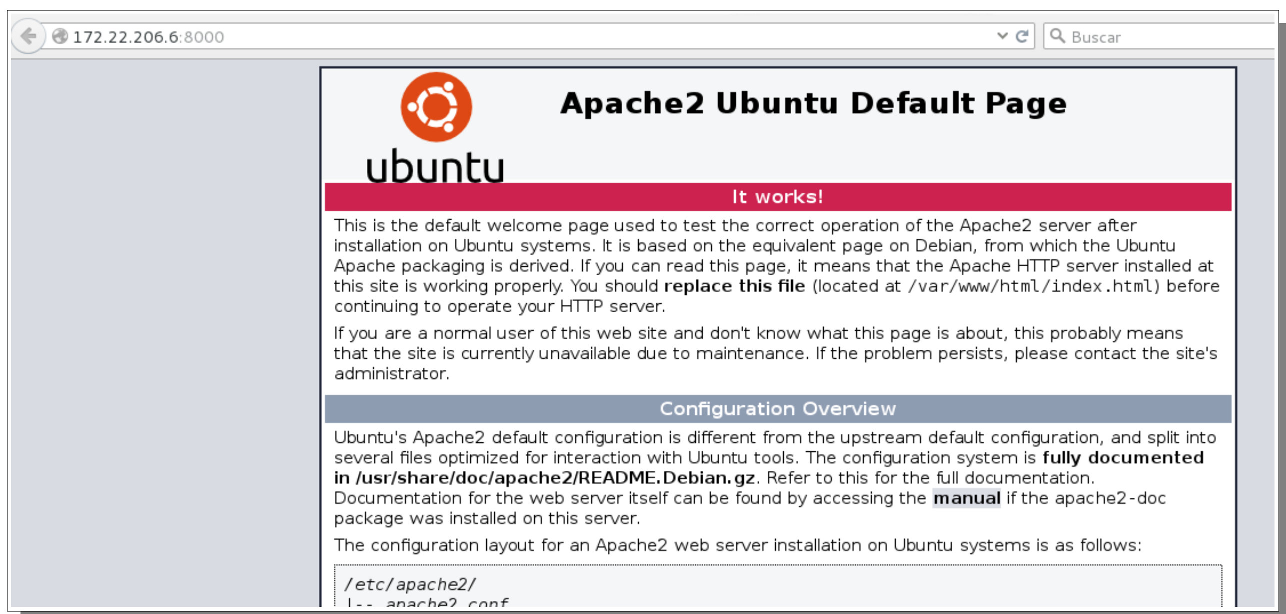
Lanzamos el contenedor:

```
root@docker:~# docker run -d -p 8000:80 --name apache_2
apache/ubuntu:v1 /usr/sbin/apache2ctl -D FOREGROUND
```

Comprobamos

```
root@docker:~# docker ps
CONTAINER ID        IMAGE                   COMMAND                      CREATED
STATUS              PORTS                   NAMES
c1fbe4ea1b9e        apache/ubuntu:v1        "/usr/sbin/apache2ctl"       9
seconds ago        Up 8 seconds                                         apache_3
1d47feeb39ec        apache/ubuntu:v1        "/usr/sbin/apache2ctl"       48
seconds ago        Up 47 seconds     0.0.0.0:8000->80/tcp    apache_2
```

Vemos que nos indica la redirección. Si accedemos desde nuestra máquina local veremos el servidor web en el puerto 8000 del anfitrión.



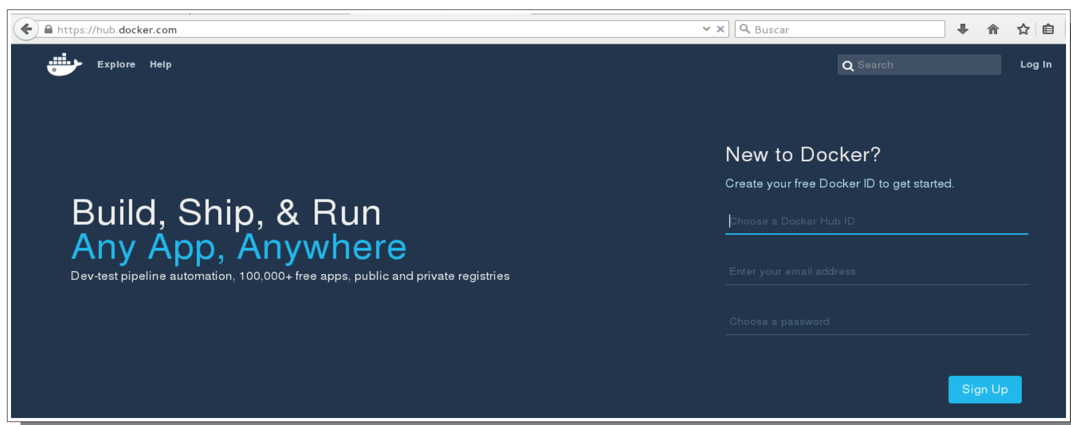
DOCKERHUB

Aquí vamos a hablar un poco de la plataforma Docker Hub, que sirve como repositorio de imágenes oficiales y de terceros.

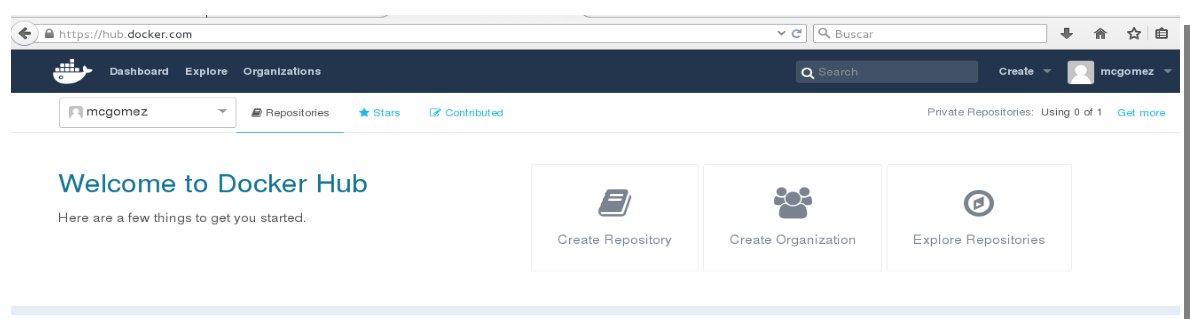
Las características de Docker Hub son:

- Repositorios de imágenes: encuentra, administra, sube y descarga imágenes oficiales y de la comunidad.
- Imágenes automáticas: crea nuevas imágenes cuando haces un cambio en la fuente de Github o BitBucket.
- Webhooks: crea automáticamente imágenes al hacer un push a un repositorio.

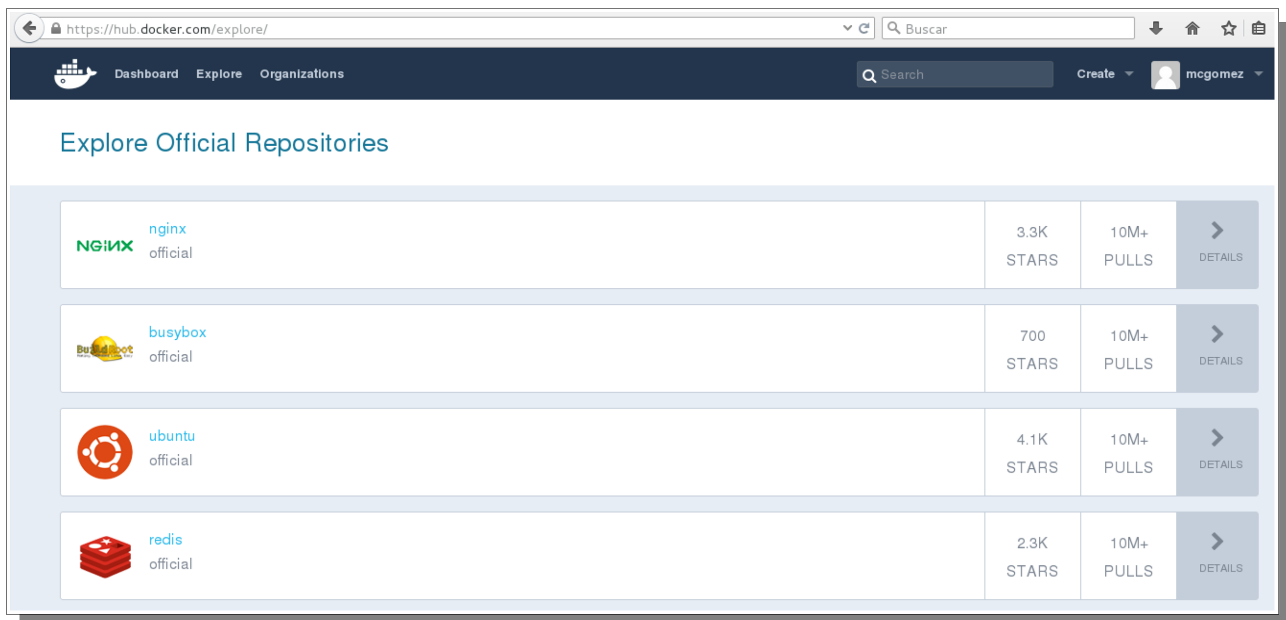
En Docker Hub podemos tener nuestro propio repositorio de imágenes, público o privado. Ya hemos visto cómo usar los repositorios desde la línea de comandos, ahora lo vemos desde el sitio web.



Creamos una cuenta y al acceder vemos que podemos crear nuestro propio repositorio, organización o buscar en los repositorios.



Al explorar vemos primero los repositorios oficiales.



Si entramos en el repositorio oficial de nginx por ejemplo, vemos las versiones disponibles para descargar y el comando.

LINKS

Para que dos contenedores colaboren entre ellos podemos abrir puertos y que se comuniquen por ahí, pero Docker nos ofrece la posibilidad de crear links entre contenedores.

Primero creamos un contenedor.

```
root@docker:~# docker run -d --name links01 apache/ubuntu:v1 /usr/sbin/apache2ctl -D FOREGROUND
bcaf034f661cfa05f4f520f42914620b06b8137a9c62a9117997cc69bd952a34
```

No hemos mapeado ningún puerto, el servidor no es accesible desde fuera.

Ahora creamos otro con la opción --link para que Docker cree un túnel entre los dos contenedores.

```
root@docker:~# docker run -d --name links02 --link links01 apache/ubuntu:v1 /usr/sbin/apache2ctl -D FOREGROUND
848a28b767733f7d791a4e3de0ae88423d1f71213c2dde4eeb6f9feec0fc8d4c
```

Comprobamos los contenedores activos con docker ps.

```
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS              PORTS               NAMES
848a28b76773        apache/ubuntu:v1    "/usr/sbin/apache2ctl"   54 seconds ago       Up 52 seconds                           links02
bcaf034f661c        apache/ubuntu:v1    "/usr/sbin/apache2ctl"   About a minute ago   Up About a minute                       links01
a6066a7a2383        apache/ubuntu:v1    "/usr/sbin/apache2ctl"   2 minutes ago        Up 2 minutes                            link_prueba
```

Este túnel es unidireccional y sólo podremos acceder desde el que se ha ejecutado con --link al otro.

```
root@docker:~# docker exec links02 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=848a28b76773
LINKS01_NAME=/links02/links01
HOME=/root
```

Vemos que aparecen variables del contenedor links01.

Podemos modificar desde links02 las variables de entorno del otro contenedor.

Docker internamente gestiona las Ips de los contenedores, añadiéndolas al fichero /etc/hosts de los contenedores.

```
root@docker:~# docker exec links02 cat /etc/hosts
127.0.0.1	localhost
::1	localhost ip6-localhost ip6-loopback
fe00::0	ip6-localnet
ff00::0	ip6-mcastprefix
ff02::1	ip6-allnodes
ff02::2	ip6-allrouters
172.17.0.6 links01 bcaf034f661c
172.17.0.7 848a28b76773
```

VOLÚMENES

Podemos montar volúmenes de datos en nuestros contenedores. Con el flag -v podemos montar un directorio dentro de nuestro contenedor. Así podemos incorporar datos que estarían disponibles para nuestro contenedor. Además seguirá apareciendo después de un reinicio del contenedor, serán persistentes.

También podemos compartir volúmenes de datos entre contenedores. Si dichos volúmenes están anclados al sistema operativo anfitrión, no serán eliminados por Docker cuando desaparezca el contenedor.

Vamos a crear un contenedor al que le pasemos un volumen.

```
root@docker:~# docker run -ti --name ejemplo_volumen -v /volumen01/ apache/ubuntu:v
1 /bin/bash
root@689345492ecf:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr  volumen01
root@689345492ecf:/#
```

Cuando listamos vemos la carpeta que encontramos montado en nuestro contenedor el directorio volumen01.

Esto es muy útil para pasarle ficheros a los contenedores y que los datos que nos interesen en ellos sean persistentes.

También podemos crear un contenedor que monte un volumen de otro con la opción –volume-from.

```
root@docker:/volumen01# docker run -d -p 8085:80 --name volumen01 --volumes-from ejemplo_volumen apache/ubuntu:v1 /usr/sbin/apache2ctl -D FOREGROUND
cf51f79ccb03f9042362d9ff962aba3ebd4c5a35ace3514600d42a37a5976036
```

VARIABLES DE ENTORNO

Podemos modificar las variables de entorno de un contenedor con el flag -e (--env)

También podemos pasarlas desde un fichero externo con ---env-file

Y podemos ver las variables con:

```
root@docker:/volumen01# docker exec a60 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=a6066a7a2383
HOME=/root
```

CONFIGURACIÓN DE RED

Por defecto los contenedores tienen las conexiones de redes habilitadas. Podemos deshabilitarlas pasando la opción --net none.

Con el comando docker run podemos especificar 3 configuraciones referentes a la red:

-dns: para configurar un servidor dns en el contenedor.

```
# docker run -i -t -dns=”8.8.8.8” ubuntu /bin/bash
```

-net: crear una red y conectar el servidor a ella.

```
# docker network create -d overlay mi-red
# docker run -net=mi-red -i -t -d ubuntu
```

-add-host: agregando una entrada en el fichero /etc/hosts del contenedor.

```
# docker run -i -t –add-host apache:10.0.0.10 ubuntu /bin/bash
```

ELIMINAR CONTENEDORES

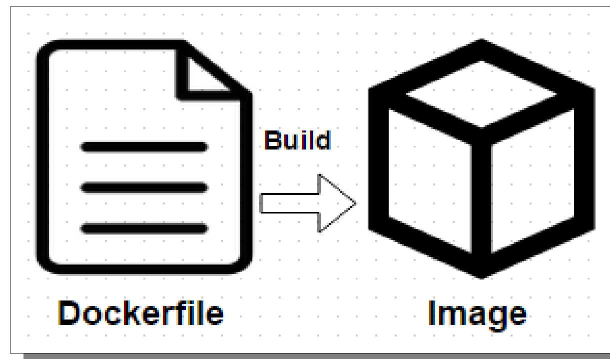
Para eliminar un contenedor podemos hacerlo por el nombre o por el ID. Realmente sólo nos hace falta los 3 primeros dígitos del ID.

Para poder eliminarlo debe estar parado. Si no lo estuviera tendríamos que pararlo con “stop”.

Utilizamos el comando **rm.** Si queremos ahorrarnos el paso de pararlo usamos -f

```
ubuntu@docker:~$ docker rm 8c5
8c5
ubuntu@docker:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                     PORTS               NAMES
18dfb4249654        hello-world         "/hello"            40 hours ago        Exited (0) 40 hours ago                        agitated_pasteur
1769cd61916d        hello-world         "/hello"            40 hours ago        Exited (0) 40 hours ago                        determined_blackwell
ubuntu@docker:~$
```

DOCKERFILE



Un Dockerfile es un archivo legible por el demonio Docker, que contiene una serie de instrucciones para automatizar el proceso de creación de un contenedor.

CONSTRUIR EL CONTENEDOR

El comando docker build irá siguiendo las instrucciones del Dockerfile y armando la imagen. El Dockerfile puede encontrarse en el directorio en el que estemos o en un repositorio.

El demonio de Docker es el que se encarga de construir la imagen siguiendo las instrucciones línea por línea y va lanzando los resultados por pantalla. Cada vez que ejecuta una nueva instrucción se hace en una nueva imagen, son imágenes intermedias, hasta que muestra el ID de la imagen resultante de ejecutar todas las instrucciones. El daemon irá haciendo una limpieza automática de las imágenes intermedias.

Estas imágenes intermedias con la caché de Docker. Y ¿para qué sirve que Docker cree una caché de imágenes intermedias? Pues si por alguna razón la creación de la imagen falla (por un comando erróneo por ejemplo), cuando lo corregimos y volvemos a construir la imagen a partir del dockerfile, el demonio no iniciará todo el proceso, sino que usará las imágenes intermedias y continuará en el punto dónde falló.

DOCKERFILE COMANDOS

FROM

Indicamos una imagen base para construir el contenedor y opcionalmente un tag (si no la indicamos docker asumirá “latest” por defecto, es decir buscará la última versión).

Lo que hace docker al leer esto es buscar en la máquina local una imagen que se llama, si no la encuentra la descargará de los repositorios.

Sintaxis:

```
FROM <imagen>
FROM <imagen>:<tag>
```

MAINTAINER

Es información del creador y mantenedor de la imagen, usuario, correo, etc.

Sintaxis:

```
MAINTAINER <nombre> <correo> <cualquier_info>
```

RUN

Ejecuta directamente comandos dentro del contenedor y luego aplica/persiste los cambios creando una nueva capa encima de la anterior con los cambios producidos de la ejecución del comando y se hace un commit de los resultados. Posteriormente sigue con la siguiente instrucción.

Sintaxis:

`RUN <comando>` → modo shell, /bin/sh -c

`RUN ["ejecutable","parámetro1","parámetro2"]` → modo ejecución, que permite correr comandos en imágenes base que no tengan /bin/sh o hacer uso de otra shell.

ENV

Establece variables de entorno del contenedor. Dichas variables se pasarán a todas las instrucciones RUN que se ejecuten posteriores a la declaración de las variables. Podemos especificar varias variables de entorno en una sóla instrucción ENV.

Sintaxis:

```
ENV <key> <value> <key> <value>
ENV <key>=<value> <key>=<value>
```

Si queremos sustituir una variable aunque esté definida en el Dockerfile, al ejecutar un contenedor podemos especificarla y tomará dicho valor, tenga el que tenga en el Dockerfile.

```
docker run -env <key>=<valor>
```

También podemos pasar variables de entorno con el comando “docker run” utilizando el parámetro -e, para especificar que dichas variables sólo se utilizarán en tiempo de ejecución.

Podemos usar estas variables en otras instrucciones llamándolas con *$nombre_var* o *${nombre_var}*. Por ejemplo:

```
ENV DESTINO_DIR /opt/app
WORKDIR $DESTINO_DIR
```

ADD

Esta instrucción copia los archivos o directorios de una ubicación especificada en <fuente> y los agrega al sistema de archivos del contenedor en la ruta especificada en <destino>.

En fuente podemos poner una URL, docker se encargará de descargar el archivo y copiarlo al destino.

Si el archivo origen está comprimido, lo descomprime en el destino cómo si usáramos “tar -x”.

Sintaxis:

```
ADD <fuente>..<destino>
ADD [“fuente”,...”destino”]
```

El parámetro <fuente> acepta caracteres comodín tipo ?,*, etc.

Una de las cosas a tener en cuenta (entre otras → https://docs.docker.com/engine/reference/builder/#add) es que el <origen> debe estar donde esté el Dockerfile, no se pueden añadir archivos desde fuera del directorio de construcción.

Si el destino no existe, Docker creará la ruta completa incluyendo cualquier subdirectorio. Los nuevos archivos y directorios se crearán con los permisos 0755 y un UID y GID de 0.

También tenemos que tener en cuenta que si los archivos o directorios agregados por una instrucción ADD cambian, entonces se invalida la caché para las siguientes instrucciones del Dockerfile.

COPY

Es igual que ADD, sólo que NO admite URLs remotas y archivos comprimidos como lo hace ADD.

WORKDIR

Permite especificar en qué directorio se va a ejecutar una instrucción RUN, CMD o ENTRYPOINT.

Puede ser usada varias veces dentro de un Dockerfile. Si se da una ruta relativa, esta será la ruta relativa de la instrucción WORKDIR anterior.

Podemos usar variables de entorno previamente configuradas, por ejemplo:

```
ENV rutadir /ruta
WORKDIR $rutadir
```

USER

Sirve para configurar el nombre de usuario a usar cuando se lanza un contenedor y para la ejecución de cualquier instrucción RUN, CMD o ENTRYPOINT posteriores.

VOLUME

Crea un punto de montaje con un nombre especificado que permite compartir dicho punto de montaje con otros contenedores o con la máquina anfitriona. Es un directorio dentro de uno o más contenedores que no utiliza el sistema de archivos del contenedor, aunque se integra en el mismo para proporcionar varias funcionalidades útiles para que los datos sean persistentes y se puedan compartir con facilidad.

Esto se hace para que cuando usemos el contenedor podamos tener acceso externo a un determinado directorio del contenedor.

Las características de estos volúmenes son:

- Los volúmenes pueden ser compartidos y reutilizados entre los contenedores.
- Un contenedor no tiene que estar en ejecución para compartir sus volúmenes.
- Los cambios en un volumen se hacen directamente.
- Los cambios en un volumen no se incluirán al actualizar una imagen.
- Los volúmenes persisten incluso cuando dejan de usarlos los contenedores.

Esto permite añadir datos, BBDD o cualquier otro contenido sin comprometer la imagen.

El valor puede ser pasado en formato JSON o como un argumento, y se pueden especificar varios volúmenes.

```
VOLUME [“/var/tmp”]
VOLUME /var/tmp
```

LABEL

LABEL añade metadatos a una imagen Docker. Se escribe en el formato etiqueta=”valor”. Se pueden añadir varios metadatos separados por un espacio en blanco.

```
LABEL version=”1.0”
LABEL localizacion=”Barbate” tipo=”BBDD”
```

Podemos inspeccionar las etiquetas en una imagen usando el comando docker inspect.
$ docker inspect <nombre_imagen>/<tag>

STOPSIGNAL

Le indica al sistema una señal que será enviada al contenedor para salir. Puede ser un número válido permitido por el Kernel (por ejemplo 9) o un nombre de señal en el formato SIGNAME (por ejemplo SIGKILL).

ARG

Define una variable que podemos pasar cuando estemos construyendo la imagen con el comando docker build, usando el flag --build-arg <varname>=<value>. Si especificamos un argumento en la construcción que no está definido en el Dockerfil, nos dará un error.

El autor del Dockerfile puede definir una o más variables. Y también puede definir un valor por defecto para una variable, que se usará si en la construcción no se especifica otro.

```
ARG user1
ARG user1=someuser ARG user2
```

Se deben usar estas variables de la siguiente forma:

```
docker build –build-arg <variable>=<valor> ….
```

Docker tiene un conjunto de variables predefinidas que pueden usarse en la construcción de la imagen sin que estén declaradas en el Dockerfile:

```
HTTP_PROXY http_proxy
HTTPS_PROXY https_proxy
FTP_PROXY ftp_proxy
NO_PROXY no_proxy
```

ONBUILD

Añade triggers a las imágenes. Un disparador se utiliza cuando se usa una imagen como base de otra imagen.

El disparador inserta una nueva instrucción en el proceso de construcción como si se especificara inmediatamente después de la instrucción FROM.

Por ejemplo, tenemos un Dockerfile con la instrucción ONBUILD, y creamos una imagen a partir de este Dockerfile, por ejemplo, IMAGEN_padre.

Si escribimos un nuevo Dockerfile, y la sentencia FROM apunta a IMAGEN_PADRE, cuando construyamos una imagen a partir de este Dockerfile, IMAGEN_HIJO, veremos en la creación que se ejecuta el ONBUILD que teníamos en el primer Dockerfile.

Los disparadores ONBUILD se ejecutan en el orden especificado en la imagen padre y sólo se heredan una vez, si construimos otra imagen a partir de la IMAGEN_HIJO, los disparadores no serán ejecutados en la construcción de la IMAGEN_NIETO.

Hay algunas instrucciones que no se pueden utilizar en ONBUILD, como son FROM, MAINTAINER y ONBUILD, para evitar recursividad.

Un ejemplo de uso:

```
FROM Ubuntu:14.04
MAINTAINER mcgomez
RUN apt-get update && apt-get install -y apache2
ONBUILD ADD ./var/www/
EXPOSE 80
CMD [“D”,”FOREGROUND”]
```

EXPOSE

Se utiliza para asociar puertos, permitiéndonos exponer un contenedor al exterior (internet, host,etc.). Esta instrucción le especifica a Docker que el contenedor escucha en los puertos especificados. Pero EXPOSE no hace que los puertos puedan ser accedidos desde el host, para esto debemos mapear los puertos usando la opción **-p** en docker run.

Por ejemplo:

```
EXPOSE 80 443
docker run centos:centos7 -p 8080:80
```

CMD

Esta instrucción es similar al comando RUN pero con la diferencia de que se ejecuta cuando instanciamos o arrancamos el contenedor, no en la construcción de la imagen.

Sólo puede existir una única instrucción CMD por cada Dockerfile y puede ser útil para ejecutar servicios que ya estén instalados o para correr archivos ejecutables especificando su ubicación.

ENTRYPOINT

Cualquier argumento que pasemos en la línea de comandos mediante docker run serán anexados después de todos los elementos especificados mediante la isntrucción ENTRYPOINT, y anulará cualquier elemento especificado con CMD. Esto permite pasar cualquier argumento al punto de entrada.

Sintaxis:

`ENTRYPOINT [“ejecutable”,”parámetro1”,”parámetro2”]` → forma de ejecución

`ENTRYPOINT comando parámetro1 parámetro 2` → forma shell

ENTRYPOINT nos permite indicar qué comando se va a ejecutar al iniciar el contenedor, pero en este caso el usuario no puede indicar otro comando al iniciar el contenedor. Si usamos esta opción es porque no esperamos que el usuario ejecute otro comando que el especificado.

ARCHIVO DOCKERIGNORE

Es recomendable colocar cada DockerFile en un directorio limpio, en el que sólo agregemos los ficheros que sean necesarios. Pero es posible que tengamos algún archivo en dicho directorio, que cumpla alguna función pero que no queremos que sea agregado a la imagen. Para esto usamos .dockerignore, para que docker build excluya esos archivos durante la creación de la imagen.

Un ejemplo de .dockerignore

```
*/prueba*
*/*/prueba
prueba?
```

EJEMPLO DOCKERFILE

Vamos a lanzar un contenedor haciendo uso del siguiente Dockerfile, que hemos colocado en una carpeta llamada /wordpress.

```
FROM debian
#FROM nos indica la imagen base a partir de la cual crearemos la imagen con "wordpress" que construirá el Dockerfile.

MAINTAINER maria <marcabgom@gmail.com>

ENV HOME /root
#ENV HOME: Establecerá nuestro directorio "HOME" donde relizaremos los comandos "RUN".

RUN apt-get update

RUN apt-get install -y nano wget curl unzip lynx apache2 php5 libapache2-mod-php5 php5-mysql

RUN echo "mysql-server mysql-server/root_password password root" | debconf-set-selections

RUN echo "mysql-server mysql-server/root_password_again password root" | debconf-set-selections

RUN apt-get install -y mysql-server

ADD https://es.wordpress.org/wordpress-4.2.2-es_ES.zip /var/www/wordpress.zip
#ADD nos permite añadir un archivo al contenedor, en este caso nos estamos bajando Wordpress

ENV HOME /var/www/html/

RUN rm /var/www/html/index.html

RUN unzip /var/www/wordpress.zip  -d /var/www/

RUN cp -r /var/www/wordpress/* /var/www/html/

RUN chown -R www-data:www-data /var/www/html/

RUN rm /var/www/wordpress.zip

ADD /script.sh /script.sh
```

```
RUN ./script.sh

#ADD nos añadirá en este caso la configuración de la base de datos que una vez realizada el despliegue tendremos que poner para su utilización.

EXPOSE 80
#EXPOSE indica los puertos TCP/IP por los que se pueden acceder a los servicios del contenedor 80 "HTTP".

CMD ["/bin/bash"]
#CMD nos establece el comando del proceso de inicio que se usará.

ENTRYPOINT ["./script.sh"]
```

También colocamos en la misma carpeta el script donde vamos a establecer una comunicación con la base de datos y nos va a crear una base de datos "wordpress" usuario "wordpress" y contraseña "wordpress":

```
#!/bin/bash

#Iniciamos el servicio mysql

/etc/init.d/mysql start

#Guardamos en variables los datos de la base de datos

DB_ROOT="root"
DB_ROOT_PASS="root"
DB_NAME="wordpress"
DB_USER="wordpress"
DB_PASS="wordpress"

#Nos conectamos a la BBDD como root y creamos el usuario sql

mysql -u ${DB_ROOT} -p${DB_ROOT_PASS} -e "CREATE USER '${DB_USER}';"

#Creamos la base de datos

mysql -u ${DB_ROOT} -p${DB_ROOT_PASS} -e "CREATE DATABASE ${DB_NAME};"

#Le damos permisos al usuario sobre la base de datos y le ponemos contraseña

mysql -u ${DB_ROOT} -p${DB_ROOT_PASS} -e "GRANT ALL ON ${DB_NAME}.* TO $
{DB_USER} $
```

```
#Reiniciamos los servicios

/etc/init.d/apache2 start

/bin/bash
```

Nos situamos en la carpeta donde tenemos el dockerfile y creamos la imagen a partir de nuestro dockerfile con el siguiente comando:

```
root@docker:/Wordpress# docker build -f dockerfile -t wordpress .
Sending build context to Docker daemon 4.608 kB
Step 1 : FROM debian
latest: Pulling from library/debian
5c90d4a2d1a8: Downloading 3.669 MB/51.35 MB
```

Vemos que empieza descargando la imagen base.

Ya tenemos creada nuestra imagen. Podemos listarla como una más.

```
root@docker:/Wordpress# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
ZE
wordpress3          latest              dde622bcda02        About a minute ago
5.1 MB
wordpress2          latest              2391f2d8ff8b        5 minutes ago
0.7 MB
wordpress           latest              d29ad8177663        16 minutes ago
0.7 MB
apache/ubuntu       v1                  defab75e4eb9        2 hours ago
8.7 MB
```

El siguiente paso es correr la imagen en nuestro puerto 80, para lo que usamos el siguiente comando:

```
root@docker:/Wordpress# docker run -d -p 80:80 -i wordpress3
e1f9169992d6e066d30bb88cf96d915b8325593585bebb654ca7b920d871257f
root@docker:/Wordpress#
```

Y comprobamos:

LIMITACIÓN DE RECURSOS

Podemos limitar los recursos usados por los contenedores.

-m, --memory=""	Límite Memoria (formato: [], donde unidad= b, k, m or g)
--memory-swap=""	Total límite de memoria (memory + swap, formato: [], donde unidad = b, k, m or g)
--memory-reservation=""	Límite flexible de memoria (formato: [], donde unidad= b, k, m or g)
--kernel-memory=""	Límite memoria Kernel (formato: [], donde unidad= b, k, m or g)
-c, --cpu-shares=0	CPU (peso relativo)
--cpu-period=0	Limitar Período CPU CFS (Completely Fair Scheduler)
--cpuset-cpus=""	CPUs en donde permitir ejecución (0-3, 0,1)
--cpuset-mems=""	Nodos de memoria en donde permitir ejecución (0-3, 0,1)
--cpu-quota=0	Limitar cuota CPU CFS (Completely Fair Scheduler)
--blkio-weight=0	Bloquear Peso IO (Peso relativo) aceptar valor de peso entre 10 y 1000.
--oom-kill-disable=false	Desahabilitar OOM Killer para el contenedor
--memory-swappiness=""	Configurar el comportamiento d Swappines del contenedor

memory=inf, memory-swap=inf (default)	**No hay límites de memoria para el contenedor**
memory=L<inf, memory-swap=inf	(Especificar memoria y configurar memory-swap como -1) El contenedor no puede usar más de la memoria especificada por L, pero puede usar toda la memoria swap que necesite
memory=L<inf, memory-swap=2*L	(Especificar memoria sin memory-swap) El contenedor no puede usar mas de la memoria especificada por L, y usar el doble como swap
memory=L<inf, memory-swap=S<inf, L<=S	(Especificar memoria y memory-swap) El contenedor no puede usar más de la memoria especificada por L, y la memoria Swap especificada por S

Algunos ejemplos de creación de contenedores con limitación de recursos serían:

```
root@docker:/# docker run -i -t -m 500m ubuntu /bin/bash
root@fbc27c5774aa:/#
```

Hemos creado un contenedor con un límite de memoria de 500 megas.

INICIO AUTOMÁTICO

Para iniciar un contenedor y hacerlo disponible desde el inicio del sistema anfitrión usamos --restart, que admite tres valores:

-no: valor predeterminado, no estará disponible al reiniciar el sistema anfitrión.

-on-failure[max-retries]: reinicia si ha ocurrido un fallo y podemos indicarle los intentos.

-always: reinicia el contenedor siempre.

Ejemplo:

```
root@docker:/# docker run -i -t --restart=always ubuntu /bin/bash
```

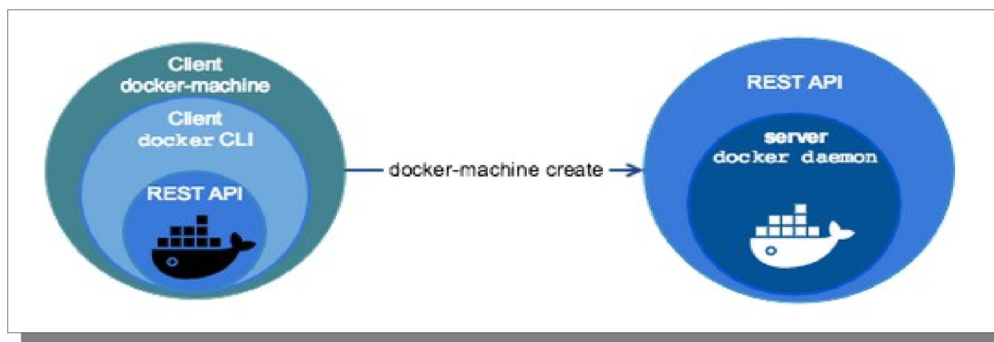
DOCKER-MACHINE

Con docker-machine podemos aprovisionar y administrar múltiples Dockers remotos, además de aprovisionar clústers Swarm tanto en entornos Windows, Mac como Linux.

Es una herramienta que nos permite instalar el demonio Docker en hosts virtuales, y administrar dichos hosts con el comando docker-machine. Además esto podemos hacerlo en distintos proveedores (VirtualBox, OpenStack, VmWare, etc.)

Usando el comando docker-machine podemos iniciar, inspeccionar, parar y reiniciar los hosts administrados, actualizar el cliente y demonio Docker y configurar un cliente Docker para que interactue con el host.

Con el cliente podemos correr dicho comando directamente en el host.



Probamos a crear una máquina virtual llamada ***prueba*** con VirtualBox. Lo hacemos desde una CMD o una terminal Linux con el comando:

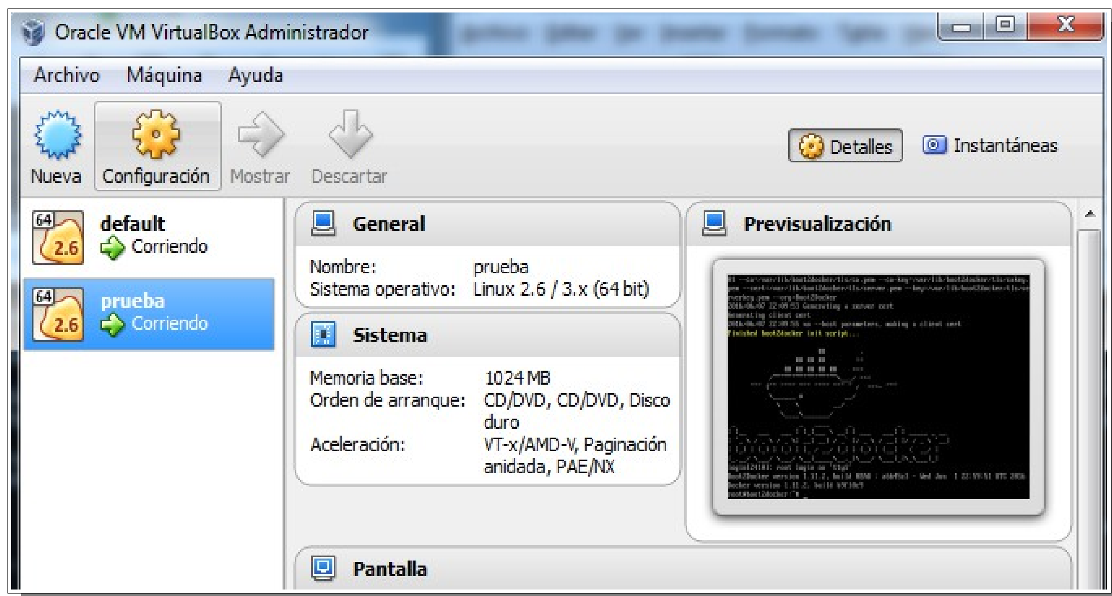
```
docker-machine create --driver virtualbox prueba
```

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\Maria>docker-machine create --driver virtualbox prueba
Running pre-create checks...
(prueba) You are using version 4.3.6r91406 of VirtualBox. If you encounter issue
s, you might want to upgrade to version 5 at https://www.virtualbox.org
Creating machine...
(prueba) Copying C:\Users\Maria\.docker\machine\cache\boot2docker.iso to C:\User
s\Maria\.docker\machine\machines\prueba\boot2docker.iso...
(prueba) Creating VirtualBox VM...
(prueba) Creating SSH key...
(prueba) Starting the VM...
(prueba) Check network to re-create if needed...
(prueba) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this vi
rtual machine, run: docker-machine env prueba

C:\Users\Maria>
```

La vemos en VirtualBox.



Vamos a ver los principales comandos de docker-machine que podemos ejecutar. Para ver una lista completa tenemos la documentación oficial.

- Crear una máquina.

```
docker-machine create --driver <nombre_driver> <nombre_máquina>
docker-machine create -d <nombre_driver> <nombre_máquina>
```

- Iniciar una máquina.

```
docker-machine start <nombre_máquina>
```

```
C:\Users\Maria>docker-machine start prueba
Starting "prueba"...
(prueba) Check network to re-create if needed...
(prueba) Waiting for an IP...
Machine "prueba" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run the `docker-machine env` command.

C:\Users\Maria>
```

- Listar las máquinas y ver su estado.

```
docker-machine ls
```

```
C:\Users\Maria>docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER    ERRORS
default   -        virtualbox   Stopped                                       Unknown
prueba    -        virtualbox   Running   tcp://192.168.99.100:2376           Unknown   Unable to qu
ery docker version: Get https://192.168.99.100:2376/v1.15/version: x509: certificate is valid for 19
2.168.99.101, not 192.168.99.100
```

- Detener una máquina.

```
docker-machine stop <nombre_máquina>
```

- Obetener la ip de una máquina.

```
docker-machine ip <nombre_máquina>
```

- Actualizar la máquina con la última versión de Docker.

```
docker-machine upgrade <nombre_máquina>
```

- Inspeccionar una máquina. Nos muestra información sobra la máquina en formato JSON, por defecto (podemos especificarlo con la opción --format (-f).

```
docker-machine inspect [options] <nombre_máquina>
```

```
C:\Users\Maria>docker-machine inspect prueba
{
    "ConfigVersion": 3,
    "Driver": {
        "IPAddress": "192.168.99.100",
        "MachineName": "prueba",
        "SSHUser": "docker",
        "SSHPort": 52187,
        "SSHKeyPath": "C:\\Users\\Maria\\.docker\\machine\\machines\\prueba\\id_r
        "StorePath": "C:\\Users\\Maria\\.docker\\machine",
        "SwarmMaster": false,
        "SwarmHost": "tcp://0.0.0.0:3376",
        "SwarmDiscovery": "",
        "VBoxManager": {},
        "HostInterfaces": {},
        "CPU": 1,
        "Memory": 1024,
        "DiskSize": 20000,
        "NatNicType": "82540EM",
        "Boot2DockerURL": "",
        "Boot2DockerImportVM": "",
        "HostDNSResolver": false,
        "HostOnlyCIDR": "192.168.99.1/24",
        "HostOnlyNicType": "82540EM",
        "HostOnlyPromiscMode": "deny",
        "NoShare": false,
        "DNSProxy": true,
        "NoVTXCheck": false
    },
    "DriverName": "virtualbox",
    "HostOptions": {
        "Driver": "",
        "Memory": 0,
        "Disk": 0,
        "EngineOptions": {
            "ArbitraryFlags": [],
            "Dns": null,
            "GraphDir": "",
            "Env": [],
            "Ipv6": false,
            "InsecureRegistry": [],
            "Labels": [],
            "LogLevel": "",
            "StorageDriver": "",
            "SelinuxEnabled": false,
            "TlsVerify": true,
            "RegistryMirror": [],
            "InstallURL": "https://get.docker.com"
        },
```

Todos estos comandos y el resto que se encuentran en la documentación oficial, están orientados a la administración de las máquinas sobre las que corre el demonio Docker.

A las máquinas podemos acceder a través de la consola de VirtualBox, o través de ssh , pero también tenemos la opción de ejecutar comandos en nuestra máquina creada directamente desde nuestra terminal o CMD. Esto es conectar nuestro cliente Docker con el demonio Docker de la máquina virtual.

Esto lo hacemos declarando las variables de entorno de nuestra máquina para indicar a docker que debe ejecutar los siguientes comandos en una máquina en concreto.

Primero exportamos las variables de la máquina:

```
docker-machine env <nombre_máquina>
```

```
C:\Users\Maria>docker-machine env prueba
SET DOCKER_TLS_VERIFY=1
SET DOCKER_HOST=tcp://192.168.99.100:2376
SET DOCKER_CERT_PATH=C:\Users\Maria\.docker\machine\machines\prueba
SET DOCKER_MACHINE_NAME=prueba
REM Run this command to configure your shell:
REM     @FOR /f "tokens=*" %i IN ('docker-machine env prueba') DO @%i
```

Si nos fijamos nos dice el comando que tenemos que ejecutar para configurar nuestro shell y que se ejecuten los comandos docker en la máquina virtual.

```
C:\Users\Maria> @FOR /f “tokens=*” %i IN ('docker-machine env prueba') DO @%i
```

Lo ejecutamos y probamos un comando docker.

```
C:\Users\Maria>@FOR /f "tokens=*" %i IN ('docker-machine env prueba') DO @%i

C:\Users\Maria>docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
PORTS               NAMES
```

Vemos que se ha ejecutado correctamente. A partir de este punto el uso de los comandos docker es el mismo que en cualquier máquina Linux.

BIBLIOGRAFÍA

https://sotoca.es/2014/06/20/jugando-con-docker/

http://www.tecnologia-informatica.es/manuales/writer4-ant.php

https://www.jsitech.com/linux/docker-conociendo-un-poco-mas-del-comando-docker-run/

https://openwebinars.net/academia/18/curso-de-docker/870/componentes-de-docker/preview/

https://docs.docker.com/windows/step_one/

http://blog.marcnuri.com/docker-instalando-docker-en-windows/

http://www.blog.teraswap.com/docker-comando-utiles/

https://docs.docker.com/engine/reference/commandline/cli/

https://www.jsitech.com/linux/docker-conociendo-los-comandos/

https://www.digitalocean.com/community/tutorials/docker-explicado-como-crear-contenedores-de-docker-corriendo-en-memcached-es

http://rm-rf.es/como-instalar-configurar-usar-docker-linux-containers/

http://portallinux.es/comandos-basicos-docker-parte-2/

https://platzi.com/blog/desplegar-contenedores-docker/

http://www.javiergarzas.com/2015/07/entendiendo-docker.html

http://www.cristalab.com/tutoriales/instalacion-y-primeros-pasos-en-docker-c114081l/

http://koldohernandez.com/instalacion-y-primeros-pasos-con-docker/

https://fportavella.wordpress.com/2015/01/08/instalacion-comandos-y-creacion-de-contenedor-docker-con-servidor-web-apache/

https://github.com/brunocascio/docker-espanol/blob/master/README.md → buen ejemplo links

http://www.josedomingo.org/pledin/2016/02/ejecutando-una-aplicacion-web-en-docker/

http://www.josedomingo.org/pledin/2016/02/primeros-pasos-con-docker/

http://www.josedomingo.org/pledin/2016/05/creando-servidores-docker-con-docker-machine/

https://docs.docker.com/engine/reference/builder/ → DOCUMENTACIÓN OFICIAL

http://codehero.co/como-construir-imagenes-usando-dockerfiles/

http://koldohernandez.com/instrucciones-de-un-dockerfile/

https://www.jsitech.com/generales/docker-creando-imagenes-basada-en-un-dockerfile/

http://www.josedomingo.org/pledin/2016/02/dockerfile-creacion-de-imagenes-docker/

http://www.nacionrosique.es/2016/06/instrucciones-en-el-fichero-dockerfile.html

https://github.com/mxabierto/hackea-el-sistema/wiki/%C2%BFC%C3%B3mo-escribir-un-buen-Dockerfile%3F

KUBERNETES

Es un proyecto open source de Google para la gestión de aplicaciones en contenedores, en especial los contenedores Docker, permitiendo programar el despliegue, escalado, monitorización de los contenedores, etc.

A pesar de estar diseñado para contenedores Docker, Kubernetes puede supervisar servidores de la competencia como Amazon o Rackspace.

Permite empaquetar las aplicaciones en contenedores y trasladarlos fácil y rápidamente a cualquier equipo para ejecutarlas.

Kubernetes es similar a Docker Swarm, que es la herramienta nativa de Docker para construir un clúster de máquinas. Fue diseñado para ser un entorno para la creación de aplicaciones distribuidas en contenedores. Es un sistema para la construcción, funcionamiento y gestión de sistemas distribuidos.

CARACTERÍSTICAS

Las principales características de Kubernetes son:

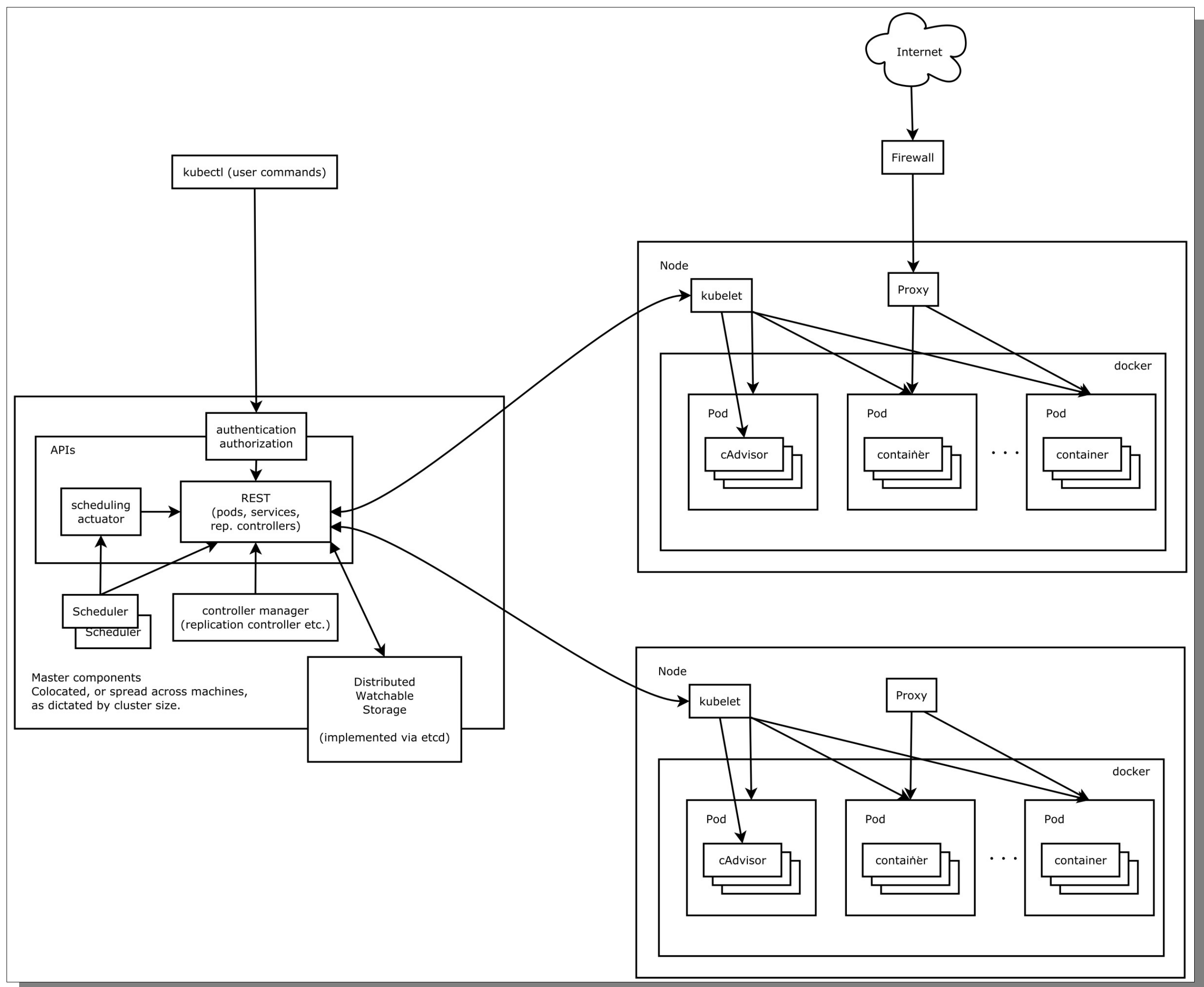
- Distribución inteligente de contenedores en los nodos.
- Otra característica es que tiene un fácil escalado, tanto horizontal como vertical, sólo hay que especificar con un comando la cantidad de nuevas aplicaciones.
- Administración de cargas de trabajo ya que nos provee de balanceadores de cargas.
- Facilita la gestión de gran cantidad de servicios y aplicaciones.
- Provee de alta disponibilidad.
- Muy modular, mucha flexibilidad.

Otra buena característica de Kubernetes es que está diseñado para mantener activos el mismo número de contenedores, es decir, si un contenedor se detiene por cualquier motivo, se creará uno nuevo con una copia casi exacta (99%).

ARQUITECTURA

Un clúster de Kubernetes está formado por nodos o minions (kubelet) y por los componentes del Master (APIs, scheduler, etc) encima de una solución de almacenamiento distribuido.

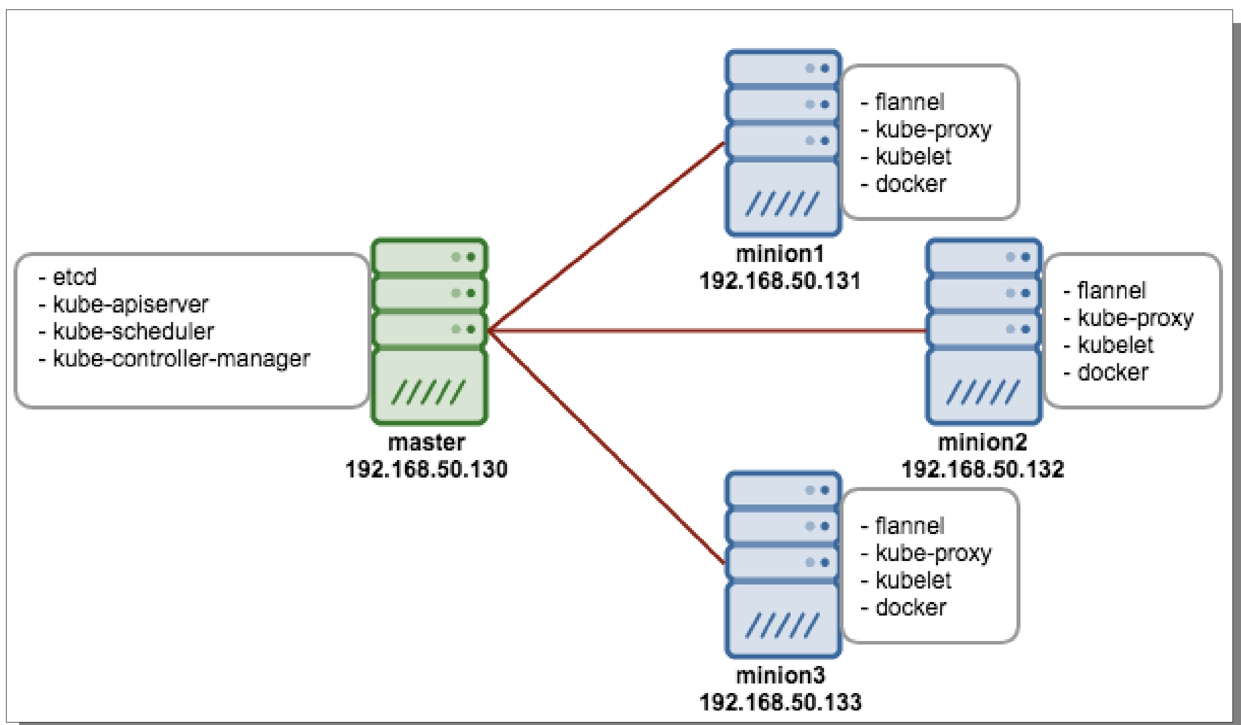
El siguiente diagrama muestra el estado actual, pero nos dicen en la documentación oficial, que se sigue trabajando en algunas cosas como la fabricación de kubelets por sí mismos corriendo en containers, que el scheduler sea 100% pluggable (enchufable).



En él podemos ver los componentes del Master y los componentes de los Nodos o Minions y cómo se comunican a través de la API y Kubelet.

MASTER	NODO
SCHEDULER	KUBE-PROXY
KUBE-CONTROLLER	PODSDOCKER
KUBE-APISERVER	cADVISOR
ETCD	KUBELET

Vemos una representación más simplificada.



KUBERNETES NODE

En el nodo se ejecutan todos los componentes y servicios necesarios para correr aplicaciones y balancear el tráfico entre servicios (endpoints). Es una máquina física o virtual que ejecuta Docker, dónde pods pueden ser programados. Docker se encarga de los detalles de la descarga de imágenes y funcionamiento de los contenedores. Este componente es el responsable de informar sobre la utilización de recursos y del estado del nodo al API server.

El nodo o minion provee los siguientes servicios:

Docker o rkt: son los motores de contenedores que funcionan en cada nodo descargando y corriendo las imágenes docker. Son controlados a través de sus APIs por Kubelet.

Kubelet: gestiona los pods y sus contenedores, sus imágenes, sus volúmenes, etc. Cada nodo corre un Kubelet, que es el responsable del registro de cada nodo y de la gestión de los pods corriendo en ese nodo. Kubelets pregunta al API server por pods para ser creados y desplegados por el Scheduler, y por pods para ser borrados basándose en eventos del cluster. También gestiona y comunica la utilización de recursos, el estado de los nodos y de los pods corriendo en él.

cAdvisor: es un recurso diseñado para los contenedores Docker e integrado en Kubelet. Es un agente de uso de los recursos y análisis que descubre todos los contenedores en la máquina y recoge información sobre CPU, memoria, sistema de ficheros y estadísticas de uso de la red. También nos proporciona el uso general de la máquina mediante el análisis del contenedor raíz en la máquina.

Flannel: provee redes y conectividad para los nodos y contenedores en el clúster. Utiliza etcd para almacenar sus datos.

Proxy (Kube-proxy): provee servicios de proxy de red. Cada nodo también ejecuta un proxy y un balanceador de carga. Los servicios endpoints se encuentran disponibles a través de DNS o de variables de entorno de Docker y Kubernetes que son compatibles entre ambos. Estas variables se resuelven en los puertos administrados por el proxy.

KUBERNETES MASTER

El servidor master va a controlar el clúster. Es el punto dónde se otorga a los servicios de clúster información de todos los nodos, y corre los siguientes servicios:

etcd: es una base de datos altamente disponible (distribuida en múltiples nodos) que almacena claves-valor en el que Kubernetes almacena información (configuración y metadatos) acerca de él mismo, pods, servicios, redes, etc, para que pueda ser utilizada por cualquier nodo del clúster. Esta funcionalidad coordina los componentes ante cambios de esos valores. Kubernetes usa etcd también para almacenar el estado del cluster.

Esta base de datos es simple (usa una API sencilla), segura, rápida y robusta.

Scheduler (Kube-scheduler): se encarga de distribuir los pods entre los nodos, asigna los pods a los nodos. Lee los requisitos del pod, analiza el clúster y selecciona los nodos aceptables. Se comunica con el apiserver en busca de pods no desplegados para desplegarlos en el nodo que mejor satisface los requerimientos.

También es el responsable de monitorizar la utilización de recursos de cada host para asegurar que los pods no sobrepasen los recursos disponibles.

API Server (kube-apiserver)– Provee la API que controla la orquestación de Kubernetes, y es el responsable de mantenerla accesible. El apiserver expone una interfaz REST que procesa operaciones cómo la creación/configuración de pods y servicios, actualización de los datos almacenados en etcd (es el único componente que se comunica con etcd).

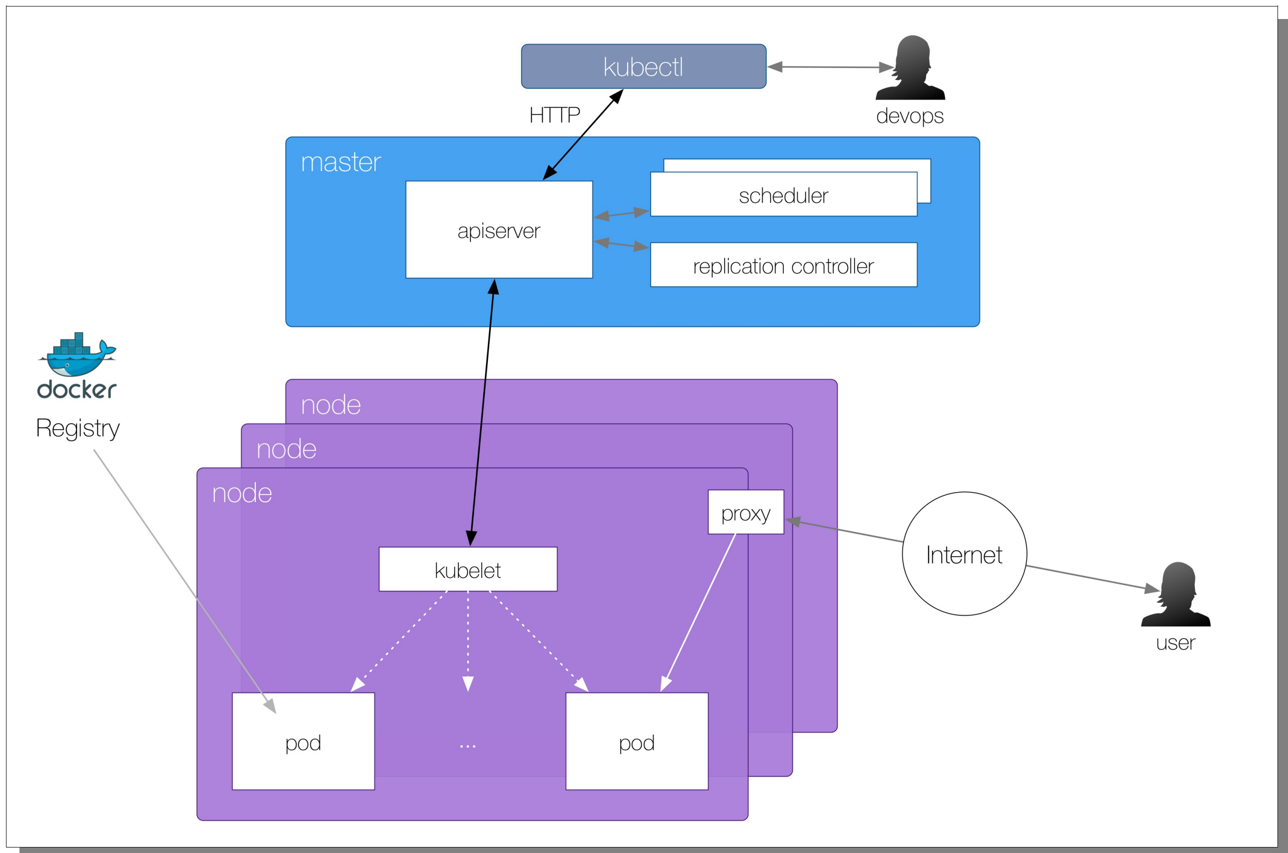
Es el responsable de que los datos de etcd y las características de los servicios de los contenedores desplegados sean coherentes.

Permite que distintas herramientas y librerías puedan comunicarse de manera más fácil.

El cliente kubecfg permite comunicarse con el master desde otro nodo.

Controller manager: es un servicio usado para manejar el proceso de replicación definido en las tareas de replicación. Los detalles de estas operaciones son descritas en el etcd, dónde el controller manager observa los cambios. Cuando un cambio es detectado, el controller manager lee la nueva información y ejecuta el proceso de replicación hasta alcanzar el estado deseado.

En Kubernetes hay muchos controladores encargados de gestionar los endpoints de los servicios, conocidos como service controllers. Los responsables de controlar el ciclo de vida de los nodos es el node controllers. En el caso de los pods tenemos los replication controllers, que escalan los pods a lo largo del cluster y funcionen correctamente.



En esta imagen vemos cómo se relacionan los componentes de Kubernetes (Master y Minions) con los desarrolladores (devops) y con los usuarios.

OTROS CONCEPTOS

Para entender Kubernetes debemos tener claros también algunos conceptos como:

PODs (dockers): son la unidad más pequeña desplegable que puede ser creada, programada y manejada por Kubernetes. Son un grupo de contenedores Dockers, de aplicaciones que comparten volúmenes y red. Son las unidades más pequeñas que pueden ser creadas, programadas y manejadas por Kubernetes. Deben ser desplegados y gestionados por controladores de réplicas.

Replication Controller: se asegura de que el número especificado de réplicas del pod estén ejecutándose y funcionando en todo momento. Manejan el ciclo de vida de los pods, creándolos y destruyendolos como sea requerido. Permite escalar de forma fácil los sistemas y maneja la re-creación de un Pod cuando ocurre un fallo. Gestiona los Pods a través de labels.

Clúster: Conjunto de máquinas físicas o virtuales y otros recursos (almacenamiento, red, etc.) utilizados por Kubernetes dónde pods son desplegados, gestionados y replicados. Los clústeres de nodos están conectados por red y cada nodo y pod pueden comunicarse entre sí. El tamaño de un clúster Kubernetes está entre 1 y 1000 nodos y es común tener más de un cluster en un datacenter.

Service: es una abstracción (un nombre único) que define un conjunto de pods y la lógica para acceder a los mismos. Un conjunto de Pods están destinados a un servicio normalmente. El conjunto de Pods que está etiquetado por un servicio está determinado por un "Selector de Labels" (también podríamos querer tener un servicio sin un Selector).

Con esto se consigue que si hay un cambio en un Pod, esto es totalmente transparente para el usuario y el servicio.

Los servicios ofrecen la capacidad para buscar y distribuir el tráfico proporcionando un nombre y dirección o puerto persistente para los pods con un conjunto común de labels.

Para aplicaciones nativas de Kubernetes, Kubernete ofrece una API de puntos finales que se actualiza cada vez que un conjunto de Pods en un servicio cambia. Para aplicaciones no nativas Kubernetes ofrece un puente basado en IP's virtuales para servicios que redirige a Pods back-end.

Labels: son pares clave/valor usados para agrupar, organizar y seleccionar un grupo de objetos tales como Pods. Son fundamentales para que los services y los replications controllers obtengan la lista de los servidores a donde el tráfico debe pasar.

REDES EN KUBERNETES

En Kubernetes es primordial que los nodos tengan acceso a internet para poder ofrecer sus servicios. También es importante que los nodos se comuniquen entre ellos y compartan recursos y datos. En el caso de un clúster Kubernetes existen dos elementos para resolver esto:

Docker0

Para el acceso de los contenedores a la red se crea una subred virtual con un segmento de red privado, administrado por una interfaz virtual que sirve de puente. Esta interfaz se crea en el momento de instalar Docker y se llama Docker0. El demonio Docker añade la interfaz a dicha subred.

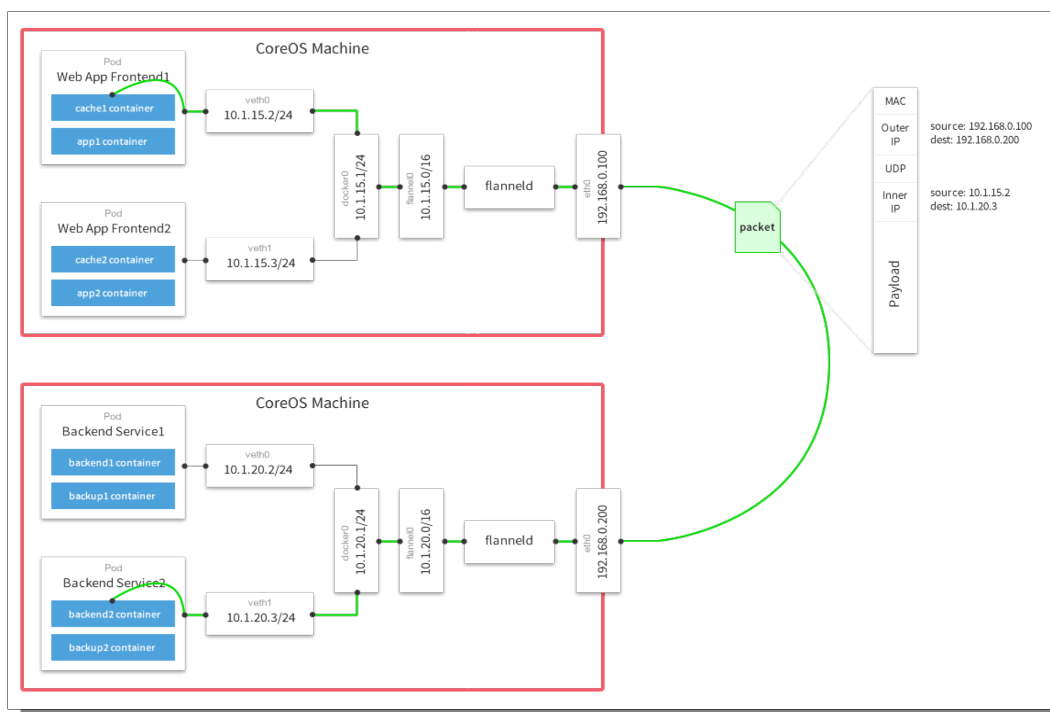
Cada contenedor tiene su propia IPv4 en la subred y hacen NAT.

Flannel

Desarrollado por CoreOS y pensado para Kubernetes, permite manejar subredes grandes. Permite que las distintas subredes entre los nodos se comuniquen entre sí. Funciona como una virtual extended LAN (VXLAN), que crea un número identificador de red.

Ésta información (identificador de red, status, IP, etc) es administrada por el servicio etcd que corre en el Master. Cada nodo se suscribe a etcd y sabe el estado de los demás nodos.

Una vez que las redes están identificadas, los participantes pueden navegar en esta red a través de su interfaz virtual. Cada nodo utiliza una interfaz virtual llamada flannel que administra el segmento de red de cada subred.



VOLÚMENES EN KUBERNETES

Los archivos en disco en un contenedor son efímeros, lo que representa algunos problemas para las aplicaciones cuando corren en contenedores. En primer lugar, cuando un contenedor se rompe, Kubelet lo reiniciará pero los archivos se perderán y el contenedor se iniciará limpio. En segundo lugar cuando dos contenedores corren en un Pod, a veces es necesario que compartan archivos entre ellos. Los volúmenes de Kubernetes resuelven ambos problemas.

En Docker existe también el concepto de volúmenes, pero no es más que un directorio en el disco o en otro contenedor. No se gestionan los tiempos de vida y hace poco que Docker ha empezado a proporcionar drivers para los volúmenes, pero tienen poca funcionalidad por ahora.

Un volumen Kubernetes, por otro lado, tiene un tiempo de vida explícito, el mismo que el Pod que lo contiene. En consecuencia, un volumen sobrevive a cualquier contenedor que se ejecuta dentro del Pod, y los datos se conservan tras un reinicio del contenedor. Por supuesto, si un Pod deja de existir, el volumen también dejará de existir.

Kubernetes nos ofrece múltiples tipos de volúmenes. En esencia un volumen sólo es un directorio con datos, que son accesibles por los contenedores de un Pod. Cómo se crea el directorio, sus características, configuraciones y usos viene determinado por el tipo de volumen elegido.

Para especificar el tipo de volumen tenemos el campo spec.volumes y para especificar el punto de montaje spec.containers.volumeMounts. Cada pod puede tener especificado un punto de montaje distinto.

En la documentación oficial vemos ejemplos de cómo definir distintos tipos de volúmenes en el fichero de creación de un Pod. Un ejemplo de archivo .yml con la definición de un volúmen sería:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

Los tipos de volúmenes en Kubernetes son:

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

emptyDir

Se crea cuando un pod es asignado a un nodo, y existirá tanto tiempo como el mismo pod. Inicialmente este volumen está vacío y permite a contenedores compartir ficheros. Todos estos puede leer y escribir del volumen. El punto de montaje puede diferir en los contenedores que usan este volumen.

Con este tipo de volumen es posible usar como dispositivo de almacenamiento la memoria. Sí usamos la propiedad emptyDir.medium con valor Memory este se montará usando tmpfs (RAM filesystem). Este tipo de sistema de almacenamiento es bastante rápido pero: 1) consume memoria de la maquina; y 2) no mantiene la información.

hostPath

Este volumen monta un fichero o directorio del host en el sistema de ficheros del pod. Esto no es algo que los Pods necesiten, pero si es una vía de escape para algunas aplicaciones.

Otros aspectos a tener en cuenta con el almacenamiento, en especial con volúmenes de tipo emptyDir y hostPath:

- No hay límite de espacio en disco utilizado.
- No hay aislamiento entre contenedores o pods.
- Tipo de dispositivo de almacenamiento, tales como discos o SSD, es determinado por el sistema de ficheros que alberga kubelet.

En el futuro, emptyDir y hostPath tendrán quotas con lo que el espacio de disco utilizado se limitará.

GcePersistentDisk

Monta un volumen de Google Compute Engine (CME) persistente en un Pod. El contenido se conserva y el volumen simplemente se desmonta. Esto significa que una PD puede ser reutilizado y sus datos traspasarse entre Pods.

Se debe crear un PD usando gcloud o la api GCE antes de poder usarlo.

Este tipo de volumen sólo puede ser utilizado por máquinas virtuales GCE.

Otra característica es que puede ser monta como sólo lectura por múltiples consumidores simultáneamente. Pero en modo lectura y escritura sólo puede ser montado por un consumidor.

awsElasticBlockStore

Este tipo de volúmenes usan el sistema EBS de AWS (Amazon Web Services). Los volúmenes tienen que estar en instancias de tipo AWS, en la misma región para poder ser utilizados entre contenedores. Solo un contenedor puede ser montado en una maquina.

nfs

Permite montar un volumen NFS de un recurso compartido existente en un pod. El contenido del volumen persiste cuando el Pod es eliminado y el volumen es simplemente desmontado. Este tipo de volúmenes permite tener múltiples puntos de escritura, la información puede ser compartida entre pods.

iscsi

Un volumen iSCSI permite montar un volumen existente de tipo iSCSI. Este tipo persiste una vez que el pod es borrado y la información puede ser reutilizada.

Una característica de iSCSI es que puede ser montado como solo lectura y accedido por múltiples consumidores simultáneamente. Sin embargo, este tipo solo permite tener un consumidor de tipo lectura-escritura.

glusterfs

Un volumen glusterfs permite usar el sistema de ficheros en red de glusterfs. El contenido de este volumen permanece después de borrar el pod. Al igual que nfs, puede ser montado con múltiples escrituras accediendo simultáneamente.

flocker

Es un sistema opensource para gestionar contenedores de datos, volúmenes. Proporciona una herramienta para gestionar y orquestar la gestión de nuestros volúmenes con soporte para diversos tipos de sistemas de almacenamiento.

Un volumen Flocker permite montar un dataset (conjunto de datos) en un pod. Si el dataset no existe en Flocker, necesitará ser creado primero. Si el dataset existe será unido al nodo que tiene el Pod por Flocker.

rbd

Es un volumen para Rados Block Device que puede ser montado en los Pods. Al igual que la mayoría, la información persiste incluso después de borrar el Pod. Como iSCSI, solo puede haber un consumidor con derecho a lectura mientras que no hay limite con consumidores de tipo lectura

gitRepo

Es un ejemplo de lo que se puede conseguir con un plugin para volúmenes. Este monta un directorio vacío y clona un repositorio git en el Pod para su uso. En el futuro, este tipo de volúmenes deberán ser más independientes y no depender de la API de kubernetes.

secret

Es un volumen que se utiliza para compartir información sensible entre Pods (por ejemplo contraseñas). A través de la API de kubernetes podemos almacenar credenciales que luego pueden montarse como volúmenes en los pods. Los volúmenes de tipo secret usan tmpfs (RAM filesystem) por lo que nunca son almacenados en discos persistentes.

downwardAPI

Este volumen se usa para hacer accesible información downward API a las aplicaciones. Monta un directorio y escribe las peticiones en forma de ficheros de texto.

FlexVolume

Permite a los usuarios de un proveedor montar volúmenes en un Pod. Los drivers del proveedor serán instalados en la ruta de plugin en cada nodo kubelet. Esto está en prueba y puede cambiar en el futuro.

AzureFileVolume

Se usa para montar un volumen de Microsoft Azure en un Pod.

vsphereVirtualDisk

Se usa para montar un volumen vSphere VMDK en un Pod. El contenido del volumen es persistente.

persistentVolumeClaim

Se utiliza para montar un volumen persistente en un Pod.

VOLÚMENES PERSISTENTES

Kubernetes proporciona una API para que usuarios y administradores puedan aprovisionar el almacenamiento dependiendo del consumo. La API proporciona dos recursos: PersistentVolume y PersistentVolumeClaim.

PersistentVolume (PV)

Es un recurso del clúster que ha sido aprivisionado por un administrador. Con el especificamos el volumen persistente. Su ciclo de vida no depende de los Pods que hacen uso de él.

PersistentVolumeClaim

Es una petición de almacenamiento por parte del usuario, reclamamos espacio en el volumen. Es similar a un Pod. Consume recursos del PV. Se pueden especificar tamaño y modo de acceso.

Ciclo de vida de un volumen y del reclamo (claim).

Los PVs son recursos del clúster. Los PVCs son las solicitudes de esos recursos, y también actúan como controles de solicitud de recursos. El ciclo de vida es:

- Aprovisionamiento

Un administrador crea una serie de PVs. Existen en la API de Kubernetes y están disponibles para el consumo. El administrador lleva los detalles del almacenamiento disponible para los usuarios del clúster.

- Unión (binding)

Un usuario crea un PVC con una cantidad específica de almacenamiento requerido y un modo de acceso. El nodo Master ve un nuevo PVC y encuentra un PV y los une. El PV tendrá el tamaño requerido o superior. La reclamación de un PV es indefinida si no se encuentra un PV.

- Uso

El clúster busca los reclamos para montar los volúmenes para un Pod. El usuario debe especificar el modo de acceso para usar el volumen.

Una vez que se le ha proporcionado un PV a un usuario, pertenece a él mientras lo necesite.

- Liberación

El usuario puede eliminar la reclamación, PVC de la API. El volumen se considera entonces liberado, pero no disponible para otro reclamo. Los datos del anterior usuario permanecen en el volumen y serán manejados de acuerdo a la política.

- Recuperación

La política de recuperación para un PV le dice al clúster qué hacer cuando el volumen haya sido liberado. Actualmente los volúmenes pueden ser retenidos, reciclados o eliminados:

- Retain: retenidos, se reclaman manualmente.
- Recycle: reciclados, se limpia con rm -rf /thevolume/*
- Delete: eliminados, se elimina el PVC y su contenido.

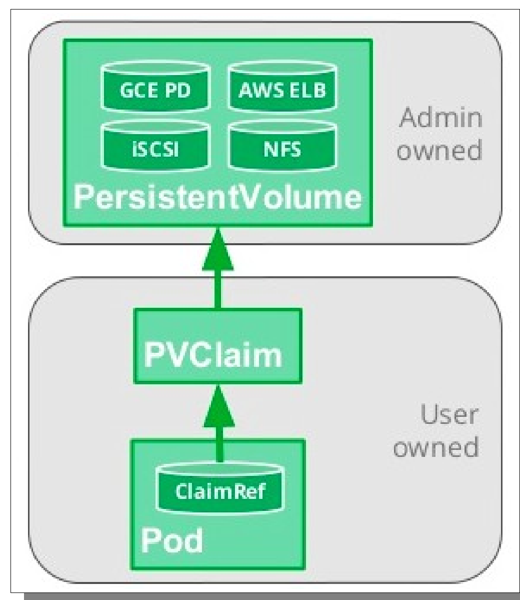
Tipos de volúmenes persistentes:

- GCEPersistentDisk
- AWSElasticBlockStore
- NFS
- iSCSI
- RBD (Ceph Block Device)

- Glusterfs
- HostPath

Modos de acceso

- ReadWriteOnce: lectura-escritura solo para un nodo (RWO)
- ReadOnlyMany: solo lectura para muchos nodos (ROX)
- ReadWriteMany: solo escritura para muchos nodos (RWX)



Características

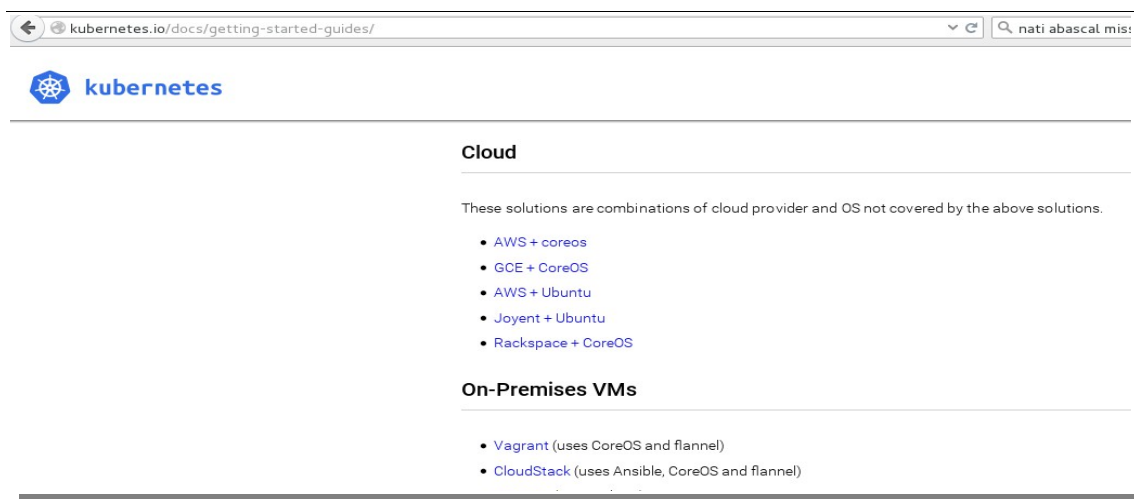
- Alto nivel de abstracción-aislamiento desde cualquier entorno en la nube.
- Lo aprovisiona el administrador, lo reclaman los usuarios.
- Tiempo de vida independiente.
- Puede ser compartido entre pods.
- Programado y gestionado dinámicamente como los nodos y los pods.

INSTALACIÓN

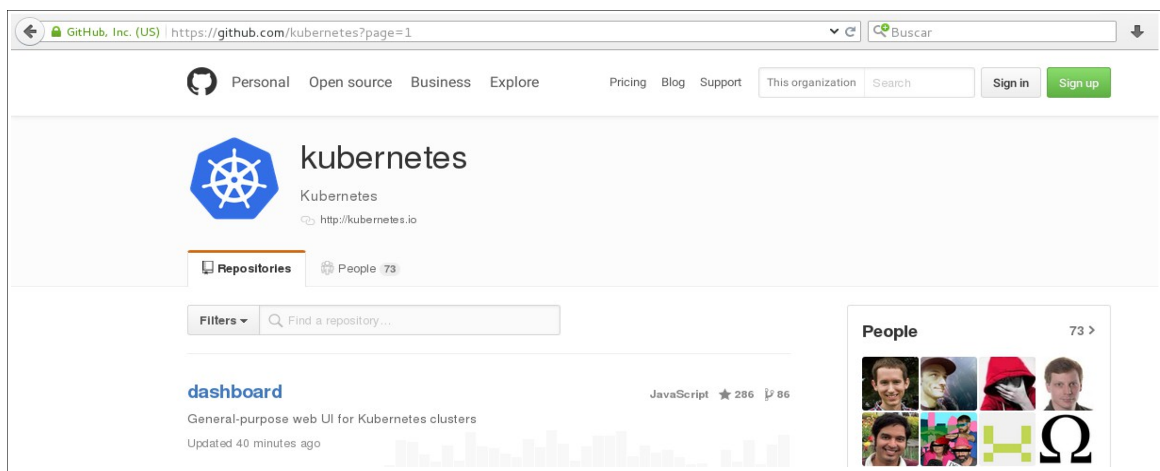
Podemos instalar Kubernetes de distintas formas dependiendo de las características del entorno en el que queramos instalar. Por ejemplo si vamos a usar Vagrant, Cloud, Ansible, OpenStack, un sistema operativo u otro, máquinas virtuales, servidores, etc.

Por tanto en la documentación oficial (enlace) nos ofrecen una serie de guías que pueden adaptarse a nuestras necesidades.

Hay algunas soluciones que requieren sólo unos cuantos comandos y otras que requieren un mayor esfuerzo para la configuración.



También disponemos del repositorio oficial de GitHub de Kubernetes, en el que tenemos infinidad de releases y contribuciones que pueden servirnos para nuestro escenario.



Si nos vamos al repositorio Kubernetes/kubernetes, tenemos una subcarpeta llamada cluster, donde encontramos scripts que nos generan clústers automáticamente, incluyendo la red, DNS, nodos y los distintos componentes del Master.

INSTALACIÓN EN CENTOS

En este escenario vamos a tener 3 máquinas.

10.0.0.20 .Master: CENTOS7.1

10.0.0.21 Minion1: CENTOS7.1

10.0.0.22 Minion2: CENTOS7.1

1.- Configuración de repositorios para la instalación de paquetes.

Esto lo hacemos en todos los nodos del clúster (Master y Minions).

Creamos el fichero **/etc/yum.repos.d/virt7-docker-common-release.repo** con el siguiente contenido:

```
[virt7-docker-common-release]
name=virt7-docker-common-release
baseurl=http://cbs.centos.org/repos/virt7-docker-common-
release/x86_64/os/
gpgcheck=0
```

2.- Configuración DNS

En todos los nodos del clúster configuramos el fichero /etc/hosts, para evitar el uso de un servidor DNS.

```
127.0.0.1   master
::1         master
10.0.0.21   minion1
10.0.0.22   minion2
```

3.- Instalación de Kubernetes y Docker

- MASTER

Instalamos el paquete kubernetes, que entre otras dependencias, instalará Docker, y los paquetes flannel y etcd.

```
[root@master ~]# yum install kubernetes flannel etcd
```

- MINIONS

Instalamos:

```
[root@minion2 ~]# yum install kubernetes flannel
```

4.- Firewall

Desactivamos el Firewall en todos los nodos del clúster, si lo tuviéramos instalado.

```
[root@master ~]# systemctl stop firewalld.service
[root@master ~]# systemctl disable firewalld.service
[root@master ~]# sed -i s/SELINUX=enforcing/SELINUX=disabled/g /etc/selinux/config
[root@master ~]# setenforce 0
```

5.- Configuración de Kubernetes en el MASTER

- *Kubernetes Config*

Configuramos el fichero **`/etc/kubernetes/config`** para que la línea KUBE_MASTER apunte a la IP del Master.

```
###
# kubernetes system config
#
# The following values are used to configure various aspects of all
# kubernetes services, including
#
#   kube-apiserver.service
```

```
#   kube-controller-manager.service
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow-privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER="--master=http://10.0.0.20:8080"
```

- *Kubernetes apiserver*

Modificamos el fichero **`/etc/kubernetes/apiserver`**., en las líneas:

ANTIGUO → `KUBE_API_ADDRESS="--insecure-bind-address=127.0.0.1"`
NUEVO → `KUBE_API_ADDRESS="--address=0.0.0.0"`

ANTIGUO → `KUBE_ETCD_SERVERS="--etcd-servers=http://127.0.0.1:2379"`
NUEVO → `KUBE_ETCD_SERVERS="--etcd-servers=http://10.0.0.20:2379"`

Comentar la siguiente línea:

```
#KUBE_ADMISSION_CONTROL="--admission-control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota"
```

- *Etcd*

Verificamos el puerto de etcd (2379) y lo configuramos para que se puedan unir desde 0.0.0.0:2379

```
# nano /etc/etcd/etcd.conf
ETCD_NAME=default
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

- *Reiniciar servicios y los habilitamos para que se inicien en el arranque.*

```
[root@master ~]# for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler; do
> systemctl restart $SERVICES
> systemctl enable $SERVICES
> systemctl status $SERVICES
> done
```

6.- Definición de una red FLANNEL

Vamos a crear una red para que se asigne a cada nuevo contenedor del clúster. Normalmente se trata de una red 172.17.0.0/16.

Creamos un archivo .json en cualquier carpeta del MASTER, con el siguiente contenido:

```
[root@master ~]# nano flannel-config.json
{
        "Network": "10.0.10.0/16",
        "SubnetLen": 24,
        "Backend": {
                "Type":    "vxlan",
                "VNI": 1
        }
}
```

En el fichero de configuración **/etc/sysconfig/flanneld** vemos la URL Y el lugar del archivo para etcd.

```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://127.0.0.1:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/atomic.io/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

Por tanto tenemos que añadir el fichero .json que creamos antes en la ruta /atomic.io/network.

```
[root@master ~]# etcdctl set /atomic.io/network/config < flannel-config.json
{
     "Network": "10.0.10.0/16",
     "SubnetLen": 24,
     "Backend": {
          "Type": "vxlan",
          "VNI": 1
     }
}
```

Lo verificamos:

```
[root@master ~]# etcdctl get /atomic.io/network/config
{
      "Network": "10.0.10.0/16",
      "SubnetLen": 24,
      "Backend": {
            "Type": "vxlan",
            "VNI": 1
      }
}
```

7.- Configuración de Kubernetes en los MINIONS

Realizamos la misma configuración en los dos minions.

- *Kubernetes config*

Ponemos la información del Master en la línea KUBE_MASTER del fichero **/etc/kubernetes/config.**

```
#   kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow-privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER="--master=http://10.0.0.20:8080"
```

- *Kubelet*

  Aquí proporcionamos la información sobre el servidor de la API y el hostname.

```
#nano /etc/kubernetes/kubelet
###
# kubernetes kubelet (minion) config

# The address for the info server to serve on (set to 0.0.0.0 or "" for
all interfaces)
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
# KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname-override=minion2"

# location of the api-server
KUBELET_API_SERVER="--api-servers=http://10.0.0.20:8080"

# pod infrastructure container
KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-
image=registry.access.redhat.com/rhel7/pod-infrastructure:latest"

# Add your own!
KUBELET_ARGS=""
```

- *Red FLANNEL*

  Modificamos sólo una línea dónde informamos del servidos Master.

```
#nano /etc/sysconfig/flanneld
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://10.0.0.20:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/atomic.io/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

- *Reiniciamos servicios en los Minions.*

```
# for SERVICES in kube-proxy kubelet docker flanneld; do
systemctl restart $SERVICES
systemctl enable $SERVICES
systemctl status $SERVICES
done
```

- *Comprobamos la red.*

```
[root@minion2 ~]# ip -4 a|grep inet
    inet 127.0.0.1/8 scope host lo
    inet 10.0.0.22/24 brd 10.0.0.255 scope global dynamic eth0
    inet 172.17.0.1/16 scope global docker0
    inet 10.0.81.0/16 scope global flannel.1
```

```
[root@minion2 ~]# ip -4 a|grep inet
    inet 127.0.0.1/8 scope host lo
    inet 10.0.0.22/24 brd 10.0.0.255 scope global dynamic eth0
    inet 172.17.0.1/16 scope global docker0
    inet 10.0.81.0/16 scope global flannel.1
[root@minion2 ~]#
```

También podemos hacer una consulta a etcd.

```
[root@minion1 ~]# curl -s http://10.0.0.20:2379/v2/keys/atomic.io/network/subnets | python -mjson.tool
{
    "action": "get",
    "node": {
        "createdIndex": 14,
        "dir": true,
        "key": "/atomic.io/network/subnets",
        "modifiedIndex": 14,
        "nodes": [
            {
                "createdIndex": 14,
                "expiration": "2016-06-18T00:53:49.78188336Z",
                "key": "/atomic.io/network/subnets/10.0.65.0-24",
                "modifiedIndex": 14,
                "ttl": 86172,
                "value": "{\"PublicIP\":\"10.0.0.21\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"52:db:30:ee:7d:8f\"}}"
            },
            {
                "createdIndex": 24,
                "expiration": "2016-06-18T00:54:18.551834481Z",
                "key": "/atomic.io/network/subnets/10.0.81.0-24",
                "modifiedIndex": 24,
                "ttl": 86201,
                "value": "{\"PublicIP\":\"10.0.0.22\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"ba:95:a5:05:19:59\"}}"
            }
        ]
    }
}
```

```
[root@minion1 ~]# curl -s http://10.0.0.20:2379/v2/keys/atomic.io/network/subnets |
 python -mjson.tool
{
    "action": "get",
    "node": {
        "createdIndex": 14,
        "dir": true,
        "key": "/atomic.io/network/subnets",
        "modifiedIndex": 14,
        "nodes": [
            {
                "createdIndex": 14,
                "expiration": "2016-06-18T00:53:49.78188336Z",
                "key": "/atomic.io/network/subnets/10.0.65.0-24",
                "modifiedIndex": 14,
                "ttl": 86172,
                "value": "{\"PublicIP\":\"10.0.0.21\",\"BackendType\":\"vxlan\",\"B
ackendData\":{\"VtepMAC\":\"52:db:30:ee:7d:8f\"}}"
            },
            {
                "createdIndex": 24,
                "expiration": "2016-06-18T00:54:18.551834481Z",
                "key": "/atomic.io/network/subnets/10.0.81.0-24",
                "modifiedIndex": 24,
                "ttl": 86201,
                "value": "{\"PublicIP\":\"10.0.0.22\",\"BackendType\":\"vxlan\",\"B
ackendData\":{\"VtepMAC\":\"ba:95:a5:05:19:59\"}}"
            }
        ]
    }
}
```

Podemos verificar los nodos del clúster si en el Master ejecutamos:

```
# kubectl get nodes
```

```
[root@master ~]# kubectl get nodes
NAME      STATUS    AGE
minion1   Ready     6m
minion2   Ready     5m
[root@master ~]#
```

Hasta aquí la instalación. Los siguientes pasos para la creación de Pods y servicios son comunes a las distintas configuraciones. Primero podemos crear los servicios y luego los Pods o al revés.

CREANDO PODS y RC

Ahora vamos a explicar cómo crear Pods y Replications Controller. Podemos hacerlo de dos maneras, con un fichero (yaml, json) o por línea de comandos. Vemos las dos.

1.- Por línea de comandos

Lo hacemos con el comando kubectl, que es el que se encarga de interactuar con la API de Kubernetes.

```
# kubectl run webserver-nginx –image=nginx --replicas=3 --port=80
```

`run`: sirve para arrancar un pod.

`my-nginx:` es el nombre que va ha recibir.

`--image`: la imagen base para construir el pod.

`--replicas`: número de pods que se han de crear.

`--port`: el puerto en el que escucha.

```
[root@master ~]# kubectl run webserver-nginx --image=nginx --replicas=3 --port=80
deployment "webserver-nginx" created
[root@master ~]#
```

Comprobamos los pods que tenemos con kubectl get pods.

```
[root@master ~]# kubectl get pods
NAME                            READY     STATUS              RESTARTS   AGE
webserver-nginx-124355620-24z2p   0/1       ContainerCreating   0          1m
webserver-nginx-124355620-6owxg   0/1       ContainerCreating   0          1m
webserver-nginx-124355620-v5lnq   0/1       ContainerCreating   0          1m
[root@master ~]#
```

Si nos fijamos en el estado pone que se está creando. Esperamos un poco para que temrinen de crearse y vemos el cambio de estado en pocos segundos.

```
[root@master ~]# kubectl get pods
NAME                            READY     STATUS    RESTARTS   AGE
webserver-nginx-124355620-24z2p   1/1       Running   0          4m
webserver-nginx-124355620-6owxg   1/1       Running   0          4m
webserver-nginx-124355620-v5lnq   1/1       Running   0          4m
[root@master ~]#
```

Vemos que los pods se crean todos con el mismo nombre y un identificador único al final.

Para ver información de un pod:

```
[root@master ~]# kubectl describe pod webserver-nginx-124355620-24z2p
Name:      webserver-nginx-124355620-24z2p
Namespace: default
Node:      minion1/10.0.0.21
Start Time:      Fri, 17 Jun 2016 06:15:14 +0000
Labels:          pod-template-hash=124355620,run=webserver-nginx
Status:          Running
IP:        172.17.0.2
Controllers:     ReplicaSet/webserver-nginx-124355620
Containers:
  webserver-nginx:
                                                 Container            ID:

docker://0b05eb407058a55a0a31cc30eca335418e90c206e6c9312f11c150b0e8884143
    Image:       nginx
    Image ID:

docker://89732b811e7f30254886e565657c6849a754e088ba37c5d2fc6d5dbcf7fc7df0
    Port:        80/TCP
    QoS Tier:
      memory:         BestEffort
      cpu:       BestEffort
    State:       Running
      Started:        Fri, 17 Jun 2016 06:17:18 +0000
    Ready:       True
    Restart Count:    0
    Environment Variables:
Conditions:
  Type           Status
  Ready    True
No volumes.
Events:
  FirstSeen      LastSeen   Count From            SubobjectPath
          Type       Reason               Message
  ---------      --------   ----- ----            -------------
          --------   ------               -------
  23m      23m        1    {default-scheduler }
     Normal          Scheduled       Successfully   assigned   webserver-
nginx-124355620-24z2p to minion1
  23m      23m        1    {kubelet                               minion1}
     spec.containers{webserver-nginx} Normal          Pulling
     pulling image "nginx"
```

```
  23m      21m        2     {kubelet minion1}
     Warning         MissingClusterDNS     kubelet    does    not    have
ClusterDNS  IP  configured  and  cannot  create  Pod  using  "ClusterFirst"
policy. Falling back to DNSDefault policy.
  21m      21m        1     {kubelet                               minion1}
     spec.containers{webserver-nginx} Normal         Pulled
     Successfully pulled image "nginx"
  21m      21m        1     {kubelet                               minion1}
     spec.containers{webserver-nginx} Normal         Created
     Created container with docker id 0b05eb407058
  21m      21m        1     {kubelet                               minion1}
     spec.containers{webserver-nginx} Normal         Started
     Started container with docker id 0b05eb407058
```

Nos ofrece muchísima información, entre ella:

- ID del contenedor.
- Imagen base
- ID imagen
- Estado
- Fecha de creación
- Puerto
- IP
- Nodo

Para borrar un pod usamos:

```
# kubectl delete pod <nombre_pod>
```

```
[root@master ~]# kubectl delete pod webserver-nginx-124355620-6owxg
pod "webserver-nginx-124355620-6owxg" deleted
[root@master ~]# kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
my-nginx-3800858182-ccxy1         1/1       Running   0          24m
webserver-nginx-124355620-24z2p   1/1       Running   0          33m
webserver-nginx-124355620-pkcnf   1/1       Running   0          7s
webserver-nginx-124355620-v5lnq   1/1       Running   0          33m
[root@master ~]#
```

Vemos que lo borra, pero que automáticamente se crea uno nuevo con un nuevo nombre y su edad es menor que la del resto.

Cuando creamos un pod con el comando run, se crea automaticamente un "deployment" que administra los pods. El deployment tendrá el mismo nombre que los pods.

```
[root@master ~]# kubectl get deployment
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
my-nginx          1         1         1            1           2h
webserver-nginx   3         3         3            3           2h
[root@master ~]#
```

Para eliminar los pods permanentemente debemos eliminar primero el deployment, para lo que nos hace falta su nombre (lo conseguimos con el comando de la imagen anterior).

Ejecutamos:

```
# kubectl delete deployment DEPLOYMENT_NAME
```

```
[root@master ~]# kubectl delete deployment webserver-nginx
deployment "webserver-nginx" deleted
[root@master ~]# kubectl delete deployment my-nginx
deployment "my-nginx" deleted
[root@master ~]#
```

Ahora comprobamos

```
[root@master ~]# kubectl get pods
NAME                               READY     STATUS        RESTARTS   AGE
my-nginx-3800858182-rbl3v          1/1       Terminating   0          18m
webserver-nginx-124355620-24z2p    1/1       Terminating   0          2h
webserver-nginx-124355620-pkcnf    1/1       Terminating   0          1h
webserver-nginx-124355620-v5lnq    1/1       Terminating   0          2h
[root@master ~]#
```

Vemos que en el estado los está eliminando. Al cabo de unos segundos no aparece nada al ejecutar el mismo comando.

http://kubernetes.io/docs/user-guide/pods/single-container/

Cuando creamos uno o varios pods es recomendable crear un replication controller que se encarga de que dicho grupo esté siempre disponible, actualizado, etc. (en versiones anteriores, la creación por línea de comandos de un pod, generaba automáticamente un RC, pero eso ya no es así,

ahora se crea el deployment). Si hay muchos pods, eliminará algunos, si hay pocos los creará. Es recomendable crear siempre un RC aunque sólo tengamos un Pod.

Una vez creado un RC, te permite:

- Escalarlo: puedes escoger el número de réplicas que tiene un pod de forma dinámica.
- Borrar el RC: podemos borrar sólo el RC o hacerlo junto a los pods de los que se encarga.
- Aislarlo: Los pods pueden no pertenecer a un RC cambiando los labels. Si un pod es removido de esta manera, será reemplazado por otro creado por el RC.

Los Replication Controller sólo se pueden crear con un fichero .yml. así que lo vemos en el punto siguiente junto a la creación de los Pods por fichero.

1.- Fichero yml

Creamos un fichero .yml con la definición de nuestro RC, que debe tener la siguiente estructura (recordemos que también podría ser a partir de un fichero en formato json).

```
{
  "apiVersion": "v1",
  "kind": "ReplicationController",
  "metadata": {
    "name": "",
    "labels": "",
    "namespace": ""
  },
  "spec": {
    "replicas": int,
    "selector": {
      "":""
    },
    "template": {
      "metadata": {
        "labels": {
          "":""
        }
      },
      "spec": {
        // See 'The spec schema' below
      }
    }
  }
}
```

Donde.

- **Kind:** siempre tiene que ser ReplicationController.
- **ApiVersion**: actualmente la v1.
- **Metadata:** contiene metadatos del RC como.

  Name: se requiere si no se especifica generateName. Debe ser un nombre único dentro del espacio de nombres y un valor compatible con RFC1035.

  Labels: opcional. Las etiquetas son claves arbitrarias que se usan para agrupar y "ser visto" por recursos y servicios.

  Éste campo es muy importante ya que será el campo en el que se fije un RC para saber qué pods tiene que gestionar.

  GenerateName: se requiere si "name" no está definido. Tiene las mismas reglas de validación que "name".

  Namespace: opcional. El espacio de nombres del replication controller.

  Annotations: opcional. Un mapa de clave/valor que puede ser utilizado por herramientas externas para almacenar y recuperar metadatos sobre objetos.

- **Spec**: especificaciones. Debe contener:

  Replicas: el número de Pods a crear.

  Selector: un mapa de clave/valor asignao al conjunto de Pods que este RC administra. Debe coincidir con la clave/valor del campo labels en la sección template.

  Template: contiene

  metadata: los metadatos con los labels para los pods.

  Labels: el esquema que define la configuración de los pods.

  spec: vemos cómo debe ser su estructura.

```
spec:
  containers:
    -
      args:
        - ""
      command:
        - ""
      env:
        -
          name: ""
          value: ""
      image: ""
```

```
      imagePullPolicy: ""
      name: ""
      ports:
        -
          containerPort: 0
          name: ""
          protocol: ""
      resources:
        cpu: ""
        memory: ""
  restartPolicy: ""
  volumes:
    -
      emptyDir:
        medium: ""
      name: ""
      secret:
        secretName: ""
```

La definición de cada campo lo vemos en la documentación oficial.

Ahora creamos nuestro fichero yaml para crear un replication controller.

```
$ nano /opt/kubernetes/examples/nginx/nginx-rc.yaml

apiVersion: v1
kind: ReplicationController
metadata:
    name: rc-nginx
spec:
    replicas: 2
    selector:
      app: nginx
    template:
      metadata:
            name: nginx
            labels:
                app: nginx
        spec:
            containers:
                - name: nginx
                  image: nginx
                  ports:
                    - containerPort: 80
```

Ahora creamos el RC con el siguiente comando:

```
$ kubectl create -f <path.yml>
```

```
[root@master ~]# kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
replicationcontroller "rc-nginx" created
```

Podemos comprobar el estado de nuestro RC con el comando:

```
# kubectl describe rc rc-nginx
```

```
[root@master ~]# kubectl describe rc rc-nginx
Name:           rc-nginx
Namespace:      default
Image(s):       nginx
Selector:       app=nginx
Labels:         app=nginx
Replicas:       3 current / 3 desired
Pods Status:    0 Running / 3 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen     LastSeen        Count   From                            SubobjectPath   Type            Reason                  Message
  ---------     --------        -----   ----                            -------------   --------        ------                  -------
  3m            3m              1       {replication-controller }                       Normal          SuccessfulCreate        Created pod: rc-nginx-nj79y
  3m            3m              1       {replication-controller }                       Normal          SuccessfulCreate        Created pod: rc-nginx-ueayo
  3m            3m              1       {replication-controller }                       Normal          SuccessfulCreate        Created pod: rc-nginx-jdihm

[root@master ~]#
```

Podemos escalar un RC, es decir, que añada pods, con el comando:

```
# kubectl scale rc <nombre_rc> --replicas=<numero>
```

Tenemos que tener en cuenta que el número será el total de pods, no el número a sumar. Es decir si tenemos 3 pods y queremos dos más, el número tendrá que ser 5.

Tenemos que esperar un poco para verlos ya que los va escalando uno a uno.

```
[root@master ~]# kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
rc-nginx-q3uyx   0/1       Pending   0          4s
rc-nginx-v4dfb   0/1       Pending   0          2m
[root@master ~]# kubectl scale rc rc-nginx --replicas=4
replicationcontroller "rc-nginx" scaled
[root@master ~]# kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
rc-nginx-2r9k1   0/1       Pending   0          2m
rc-nginx-f3irm   0/1       Pending   0          2m
rc-nginx-q3uyx   0/1       Pending   0          3m
rc-nginx-v4dfb   0/1       Pending   0          5m
[root@master ~]#
```

Para eliminar un RC.

```
# kubectl delete rc <nombre_rc>
```

```
[root@master ~]# kubectl delete rc rc-nginx
replicationcontroller "rc-nginx" deleted
[root@master ~]#
```

Ahora vamos a ver cómo tiene que ser un fichero yaml de creación de un pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: ""
  labels:
    name: ""
  namespace: ""
  annotations: []
  generateName: ""
spec:
  ? "// See 'The spec schema' for details."
  : ~
```

Donde la definición de cada campo es la misma que en el fichero de creación de un RC, por lo que no lo repetimos.

Nuestro fichero yaml quedará:

```
# nano /opt/kubernetes/examples/nginx/pod-nginx.yaml

apiVersion: v1
kind: Pod
metadata:
    name: my-nginx
# Especificamos que el pod tenga un label con clave "app" y valor "nginx"
que será lo que vea el RC para saber que tiene que gestionarlo.
    labels:
      app: nginx
spec:
    containers:
```

```
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
    restartPolicy: Always
```

Ejecutamos el comando:

```
# kubectl create -f /op     t/kubernetes/examples/nginx/pod-nginx.yaml
```

```
[root@master ~]# kubectl create -f /opt/kubernetes/examples/nginx/pod-nginx.yaml
pod "my-nginx" created
[root@master ~]#
```

Vemos el plod creado.

```
[root@master ~]# kubectl get pods
NAME        READY     STATUS     RESTARTS    AGE
my-nginx    0/1       Pending    0           1m
[root@master ~]#
```

Vemos si tenemos algún RC o deployment.

```
[root@master ~]# kubectl get rc
[root@master ~]# kubectl get deployment
[root@master ~]#
```

No se ha creado ninguno, por lo que esta metodología no es la más adecuada para crear nuestros pods.

CREANDO SERVICIOS

Como hemos explicado anteriormente, podemos crear un servicio (services), y dentro de él podemos tener infinidad de contenedores, dónde Kubernetes hará de balanceador de carga con lo que se llama Replication Controller.

Los pods son volátiles, son creados y destruidos trivialmente. Su ciclo de vida es manejado por los Replication Controllers.

Cada Pod tiene su propia dirección IP, que incluso puede cambiar a lo largo de su vida, lo que supone un problema a la hora de comunicarse con otros Pods. Entonces ¿cómo pueden comunicarse?

Lo hacen con lo que se llama "service". Un service define un grupo lógico de pods y una política de acceso a los mismos. Los pods apuntan a un servicio por la propiedad label. De esta manera un servicio en Pod que es destruido seguirá siendo accesible en otro Pod gracias a los Labels, incluso si está expuesto desde fuera del clúster.

Una buena práctica es la de crear el servicio primero y luego el RC, por lo que vamos a eliminar el que creamos anteriormente.

Creamos un servicio que será expuesto al exterior.

```
# nano /opt/kubernetes/examples/nginx/svc-nginx.yaml

apiVersion: v1
kind: Service
metadata:
    name: my-nginx-service
spec:
    type: NodePort
    selector:
      app: nginx
    ports:
      - port: 80
        protocol: TCP
        name: http
```

Levantamos el servicio con el comando:

```
# kubectl create -f /opt/kubernetes/examples/nginx/svc-nginx.yaml
```

```
[root@master ~]# kubectl create -f /opt/kubernetes/examples/nginx/svc-nginx.yaml
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet, you may
need to set up firewall rules for the service port(s) (tcp:31958) to serve traffic.

See http://releases.k8s.io/release-1.2/docs/user-guide/services-firewalls.md for mo
re details.
service "my-nginx-service" created
[root@master ~]#
```

Lo podemos ver con:

```
[root@master ~]# kubectl get svc
NAME               CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes         10.254.0.1       <none>        443/TCP   27m
my-nginx-service   10.254.159.198   nodes         80/TCP    54s
[root@master ~]#
```

Ahora creamos el RC que se va a encargar de los Pods que sirven el service.

```
# kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
```

```
[root@master ~]# kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
replicationcontroller "rc-nginx" created
[root@master ~]#
```

Vemos que se van creando.

```
[root@master ~]# kubectl get pods
NAME             READY     STATUS              RESTARTS   AGE
my-nginx-k3gnp   0/1       ContainerCreating   0          10s
my-nginx-rqleo   0/1       ContainerCreating   0          10s
[root@master ~]#
```

Una vez creados.

```
[root@master ~]# kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
my-nginx-k3gnp   1/1       Running   0          2m
my-nginx-rqleo   1/1       Running   0          2m
[root@master ~]#
```

Podemos ver datos del servicio, como la IP del clúster, con el comando:

```
# kubectl describe service <nombre_servicio>
```

```
[root@master ~]# kubectl describe service my-nginx
Name:                   my-nginx-service
Namespace:              default
Labels:                 <none>
Selector:               app=nginx
Type:                   NodePort
IP:                     10.254.159.198
Port:                   <unset> 80/TCP
NodePort:               <unset> 31958/TCP
Endpoints:              172.17.0.2:80,172.17.0.2:80
Session Affinity:       None
No events.
[root@master ~]#
```

Las Ips de Endpoints son las ips de los minions que sirven el servicio, en este caso nginx.

Nos conectamos a otra máquina que está en la misma red y con curl accedemos a nginx.

```
root@docker: ~ 83x34
Setting up libcurl3:amd64 (7.35.0-1ubuntu2.6) ...
Setting up curl (7.35.0-1ubuntu2.6) ...
Processing triggers for libc-bin (2.19-0ubuntu6.6) ...
root@docker:~# curl 172.17.0.2:80

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/T
R/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <!--
    Modified from the Debian original for Ubuntu
    Last updated: 2014-03-19
    See: https://launchpad.net/bugs/1288690
  -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Apache2 Ubuntu Default Page: It works</title>
    <style type="text/css" media="screen">
  * {
    margin: 0px 0px 0px 0px;
    padding: 0px 0px 0px 0px;
  }

  body, html {
    padding: 3px 3px 3px 3px;

    background-color: #D8DBE2;

    font-family: Verdana, sans-serif;
    font-size: 11pt;
    text-align: center;
  }

  div.main_page {
    position: relative;
```

BIBLIOGRAFÍA KUBERNETES

http://kubernetes.io/

http://kubernetes.io/docs/user-guide/

http://es.slideshare.net/paradigmatecnologico/introduccin-a-kubernetes

http://es.slideshare.net/paradigmatecnologico/introduccin-a-arquitecturas-basadas-en-microservicios?next_slideshow=1

http://javierjeronimo.es/2014/11/15/kubernetes-cluster-servicios-docker-facil/

https://www.adictosaltrabajo.com/tutoriales/primeros-pasos-con-kubernetes/

http://blog.juliovillarreal.com/instalando-y-configurando-kubernetes-en-centos-o-rhel-7/

http://www.cyliconvalley.es/2016/03/21/recursos-de-las-charlas-sobre-docker-y-kubernetes/

http://kubernetes.io/docs/admin/cluster-components/

https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes

https://www.digitalocean.com/community/tutorials/el-ecosistema-de-docker-una-introduccion-a-los-componentes-mas-comunes-es

http://profesores.elo.utfsm.cl/~agv/elo323/2s15/projects/reports/Diaz_Reyes.pdf

https://es.ikoula.wiki/es/Implementar_un_cl%C3%BAster_Kubernetes_con_CoreOS

https://coreos.com/kubernetes/docs/latest/deploy-master.html

http://kubernetes.io/docs/user-guide/connecting-applications/

https://www.adictosaltrabajo.com/tutoriales/instalacion-de-kubernetes-en-ubuntu-con-ansible/#03

https://github.com/kubernetes/kubernetes/tree/master/cluster/ubuntu

http://tedezed.github.io/Celtic-Kubernetes/HTML/8-Kubernetes_ansible.html

http://www.aventurabinaria.es/kubernetes-desplegado-centos/

http://tedezed.github.io/Celtic-Kubernetes/HTML/1-Portada.html

http://www.mundodocker.com.br/kubernetes-parte-ii/

https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-kubernetes-on-top-of-a-coreos-cluster

http://devopslab.com.br/kubernetes-como-instalar-e-configurar-o-kubernetes-k8s-gerencia-de-containers-docker/

http://severalnines.com/blog/installing-kubernetes-cluster-minions-centos7-manage-pods-services

http://blog.juliovillarreal.com/instalando-y-configurando-kubernetes-en-centos-o-rhel-7/

https://github.com/kubernetes/kubernetes/blob/master/examples/simple-nginx.md

http://containertutorials.com/get_started_kubernetes/k8s_example.html

http://kubernetes.io/docs/user-guide/simple-nginx/

http://kubernetes.io/docs/getting-started-guides/docker/

https://github.com/kubernetes/kubernetes/tree/master/examples/guestbook#step-one-start-up-the-redis-master

https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/7/getting-started-with-containers/chapter-4-creating-a-kubernetes-cluster-to-run-docker-formatted-container-images

https://github.com/tidchile/taller-hands-on-kubernetes

https://www.adictosaltrabajo.com/tutoriales/primeros-pasos-con-kubernetes/

http://kubernetes.io/docs/user-guide/kubectl-overview/

http://kubernetes.io/docs/user-guide/kubectl/kubectl/