

BUILD, SHIP, RUN

Docker is the world's leading software containerization platform

[Get Started with Docker](#)

[Watch Video](#) ▶



docker 1.12 GA
Built-in Orchestration



pronoide

Version 1.6.1 2020-09-10

Contenidos

1. El concepto del contenedor	1
2. Instalación de Docker	30
2.1. Prerrequisitos	30
2.2. Lab: Instalación en RHEL	31
2.2.1. Instalación de docker con DNF	31
2.2.2. Prueba de ejecución	32
2.2.3. Instalación en grupo docker	33
2.2.4. Reinicio del perfil del usuario	34
2.2.5. Información básica del motor Docker Engine	34
3. Imágenes Docker	35
3.1. Criterio de la comunidad al nombrar imágenes docker	35
3.2. Sobre los Tags en docker	36
3.3. Recomendaciones en la gestión de imágenes docker	37
3.4. Lab: Imágenes Docker	38
3.4.1. Descarga de imágenes (docker pull)	38
3.4.2. Guardado de imágenes (docker save)	38
3.4.3. Listando las imágenes (docker images)	39
3.4.4. Eliminando imágenes (docker rmi)	39
3.4.5. Carga de imágenes (docker load)	40
3.4.6. Etiquetado de imágenes (docker tag)	41
4. Docker Hub	42
4.1. Características del portal	42
4.2. Lab: Docker Hub	45
4.2.1. Registro en el portal web	45
4.2.2. Búsqueda de imágenes (docker search)	46
4.2.3. Búsqueda de imágenes en el portal web	48
4.2.4. Realizando login por consola	50
4.2.5. Subida de imagen propia a docker hub (docker push)	51
5. Contenedores Docker	53
5.1. Ciclo de vida	53
5.2. Lab: Contenedores Docker	55
5.2.1. Arrancar un contenedor (docker run)	55
5.2.2. Listando contenedores activos (docker container ls docker ps)	55
5.2.3. Parar un contenedor (docker stop)	56
5.2.4. Listando los contenedores detenidos (docker container ls -a docker ps -a)	56
5.2.5. Iniciando un contenedor detenido (docker start)	57
5.2.6. Pausando un contenedor (docker pause)	57
5.2.7. Deshaciendo el pausado de un contenedor (docker unpause)	57

5.2.8. Reiniciando un contenedor (docker restart)	58
5.2.9. Detención forzada de un contenedor (docker kill)	58
5.2.10. Eliminando un contenedor (docker rm)	59
5.2.11. Creando un contenedor sin ejecutarlo (docker create)	59
5.2.12. Auto borrado de contenedor al detenerse (directiva --rm)	60
5.2.13. Borrando un contenedor que se encuentra en operación (borrado forzado -f)	60
6. Dockerfile	61
6.1. ¿Cómo se estructura?	61
6.2. ¿Qué es el multi-stage build?	61
6.3. Lab: Dockerfile	62
6.3.1. Construcción de nuestro primer Dockerfile	62
6.3.2. Pasos de construcción	62
6.3.3. Solución de problemas	63
6.3.4. Histórico	63
6.3.5. Instrucciones Dockerfile	64
6.3.6. CMD	64
6.3.7. ENTRYPOINT	64
6.3.8. WORKDIR	65
6.3.9. ENV	65
6.3.10. USER	66
6.3.11. VOLUME	66
6.3.12. ADD	68
6.3.13. COPY	68
6.3.14. ONBUILD	68
6.3.15. Creando imagen multi-stage	69
7. Contenedores Avanzados	70
7.1. Lab: Contenedores Avanzados	71
7.1.1. Exportación del sistema de archivos de un contenedor (docker export)	71
7.1.2. Importación del sistema de archivos de un contenedor (docker import)	71
7.1.3. Ejecución de comandos contra un contenedor (docker exec)	71
7.1.4. Cambios en el sistema de un contenedor que está en funcionamiento (docker commit)	72
7.1.5. Inspección de contenedores (docker inspect)	73
7.1.6. Aislamiento de procesos del contenedor	74
7.1.7. Adjuntándonos al proceso (docker attach)	75
7.1.8. Visualización de procesos (docker top)	75
7.1.9. Obtención de traza del contenedor (docker logs)	76
7.1.10. Destrucción del laboratorio	77
8. Volúmenes y puntos de montaje	78
8.1. ¿Cómo son definidos a nivel de sistema?	78
8.2. ¿Dónde guarda docker toda la gestión/administración de los volúmenes que el controla?	78
8.3. Lab: Volúmenes	79

8.3.1. Montando un volumen en un contenedor	79
8.3.2. Volumen bind.....	80
8.3.3. Volumen tmpfs.....	81
9. Networking	82
9.1. Lab: Networking	83
9.1.1. Creando un contenedor en la red Bridge	83
9.1.2. Bridge.....	84
9.1.3. Host	85
9.1.4. Container (Legacy)	86
9.1.5. None	86
9.1.6. Personalizada (Definida por el usuario)	87
9.1.7. Overlay	87
10. Docker Registry (Registros privados)	88
10.1. Lab: Docker Registry (Registros privados)	89
10.1.1. Registry Privado (registry)	89
10.1.2. Poniendo en marcha el registry	89
10.1.3. Persistencia en disco del registro	90
10.1.4. Registry Privado (Sonatype Nexus)	90
11. Integración continua	94
11.1. Lab: Integración continua	95
11.1.1. Creación del Dockerfile	95
11.1.2. Creación la imagen.....	96
11.1.3. Generación de la tarea.....	97
12. Monitorización	101
12.1. Docker stats	101
12.2. Docker events	101
12.3. CAdvisor	101
13. Seguridad	103
13.1. Namespaces del Kernel	103
13.2. CGroups	103
13.3. Ataques al daemon Docker	103
13.4. Otras características de seguridad	104
13.5. Lab: Seguridad	105
13.5.1. Proteger el socket del daemon Docker	105
14. Docker compose	110
14.1. Configuración	110
14.2. Directivas	110
14.2.1. build	111
14.2.2. deploy	112
14.2.3. depends_on	114
14.2.4. entrypoint	114

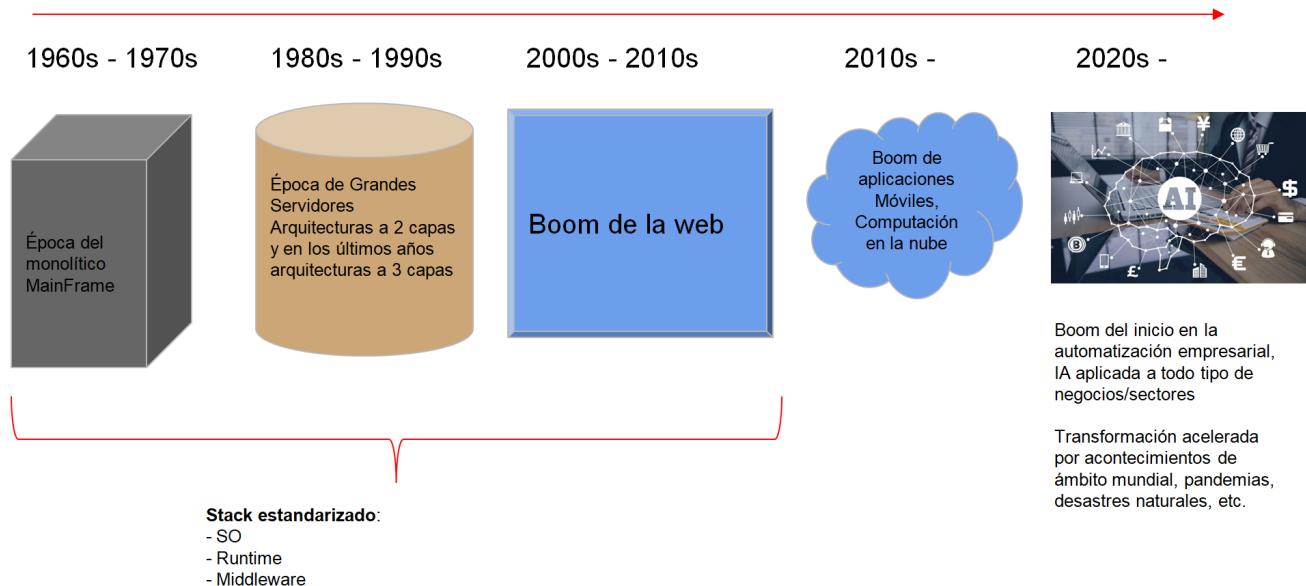
14.2.5. env_file	114
14.2.6. environment	114
14.2.7. expose	114
14.2.8. healthcheck	114
14.2.9. image	115
14.2.10. links	115
14.2.11. network_mode	115
14.2.12. networks	115
14.2.13. ports	115
14.2.14. ulimits	116
14.2.15. volumes	116
14.2.16. restart	116
14.2.17. privileged	116
14.2.18. driver	116
14.3. Inicio de docker-compose	117
14.4. Creación de imágenes	117
14.5. Ventajas e inconvenientes de docker-compose	123
14.6. Ejemplos en version 3	123
15. Docker swarm	126
15.1. Características	126
15.2. Conceptos	126
15.2.1. swarm	126
15.2.2. nodo	126
15.2.3. servicios y tareas	127
15.2.4. Balanceo de carga	127
15.3. Funcionamiento	127
15.4. Creación de swarm	128
15.5. Agregación de servicios	129
15.6. Monitorización de servicios	129
15.7. Escalar un servicio	130
15.8. Actualización de contenedores	130
15.9. Borrado de servicios	131
15.10. Drenar nodos	131
15.11. Deploy	131
15.11.1. Ejemplo	131
15.11.2. Ejemplo avanzado	133
16. Kubernetes (k8s)	136
16.1. Introducción	136
16.2. Características	137
16.3. Restricciones	137
16.4. Componentes	137

16.4.1. Master	137
16.4.2. Kube-apiserver	138
16.4.3. etcd	138
16.4.4. Kube-controller-manager	138
16.4.5. cloud-controller-manager (Nuevo en 1.6)	138
16.4.6. kube-scheduler	138
16.5. Componentes del nodo	138
16.5.1. kubelet	138
16.5.2. kube-proxy	139
16.5.3. docker	139
16.5.4. rkt	139
16.5.5. supervisord	139
16.5.6. fluentd	139
16.6. PODS	139
16.6.1. Gestión de múltiples contenedores	139
16.6.2. Red	140
16.6.3. Fases	140
16.6.4. Ejemplo de definición de un POD	140
16.6.5. Pods de infraestructura	141
16.7. Namespaces	141
16.8. Instalación de minikube	142
16.8.1. Instalación de kubectl	142
16.8.2. Instalación de minikube	142
16.8.3. Creación del cluster	142
16.9. Opciones especiales de Minikube	147
16.10. Ficheros de Configuración	148
16.11. Proyecto web con apache	152
17. Imágenes Docker Avanzadas	157
17.1. Lab: Imágenes Avanzadas	158
17.1.1. Soporte de clientes en sistema operativo	158
17.1.2. Pruebas de entorno gráfico	158
17.1.3. Ejecución Linux XEyes	158
17.1.4. Lanzamiento del IDE de desarrollo Eclipse	159
17.1.5. Lanzamiento del IDE de desarrollo VSCode	161
17.1.6. Conclusiones	162
18. Desarrollo con Docker	163
18.1. WebSite estático	163
18.2. Aplicación Web	164
19. Weblogic workshop	168
19.1. Preparación del entorno	168
19.2. Generación de un dominio básico:	169

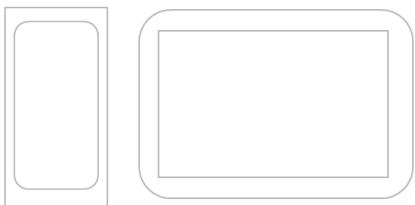
19.3. Clustering en un host	173
19.4. Clustering en múltiples hosts	174
19.4.1. AppDeploy	174
19.4.2. WebTier	174
19.5. Construcción del entorno multihost	175
19.6. Agregación de un ManagedServer.....	176
19.7. Inicio del WebTier	177
19.8. Otros contenidos del repositorio	177

Capítulo 1. El concepto del contenedor

Algo de historia evolutiva de las aplicaciones...



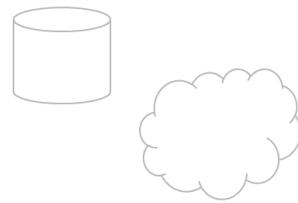
Paradigma Actual



Aplicaciones Ligeras en todo tipo de dispositivos ya sean móviles, tablets, smartwatch, etc.



Construcción de los sistemas basado en componentes, orientación a arquitectura de servicios



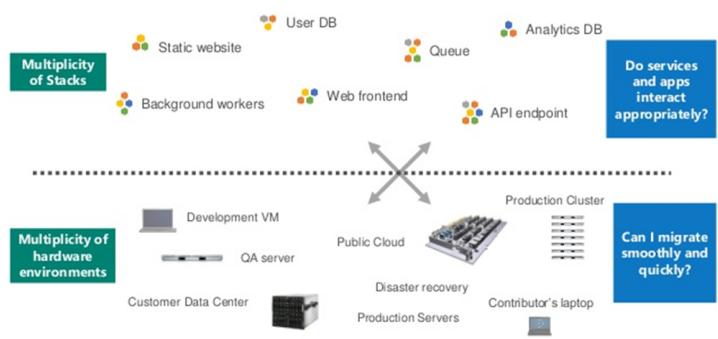
Funcionamiento en cualquier tipo de infraestructura, nube privada, infraestructura on premise”, virtualización

¿Exactamente en qué problemas puede salvarnos Docker?

- Stack de aplicaciones muy amplio tanto en tecnología como en infraestructura de funcionamiento
- Puesta en marcha continua de versiones cada vez con más frecuencia
- Necesidades de réplica de entornos de infraestructura y aplicaciones

La pesadilla de la compatibilidad NxN

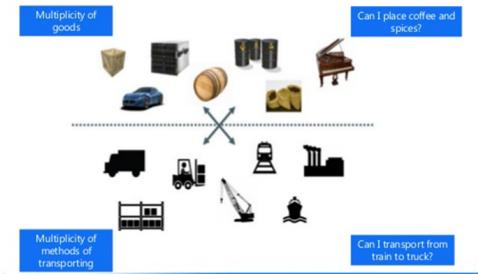
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?
	Development VM	QA Server	Single Prod Server	Grid Cluster	Public Cloud	Contributor's Laptop	Customer Servers



¿Alguien se ha enfrentado a este problema en otras ingenierías?

Problemática que se daba antes de 1960

- Cuando quieres transportar algo, puedes utilizar cualquier tipo de medio de transporte, con lo que eso conlleva...
- Sobre esta época, **se inventa** una forma de estandarizar el transporte de mercancías, **el contenedor**, con unas medidas standard



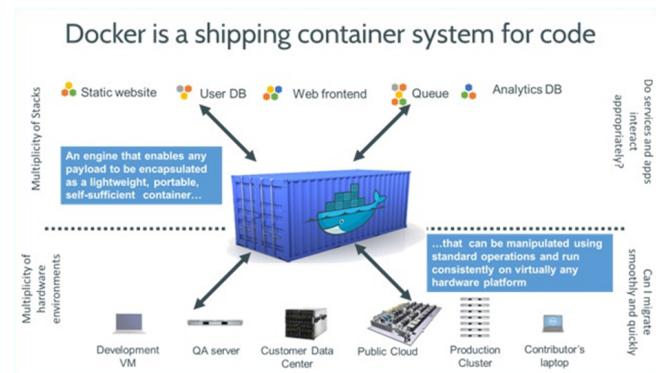
El Container al rescate...

- Revolución en la forma de transportar objetos
- El que genera el bien material, sólo tiene que tener en cuenta que bien de forma completa o por piezas tiene que caber en un container con unas medidas concretas
- Del mismo modo, el transportista sólo tiene que tener en cuenta que mueve objetos de un sólo tipo, contenedores con unas medidas muy concretas
- El coste disminuyó drásticamente
- La velocidad de entrega se incrementó de tal forma que supuso el mayor despegue en la historia del comercio a nivel mundial



Aplicando el concepto del container de transporte a las tecnologías de la información, nace Docker

- Empaqueamiento de una forma standard de stack tecnológico que necesitamos para que nuestro sistema funcione
- Posteriormente, independientemente del hardware podríamos poner en marcha el sistema en cuestión de una forma estandarizada



¿Qué beneficios obtengo en desarrollo?

- Entornos de funcionamiento limpios, aportando seguridad y portabilidad entre diferentes tipos de infraestructura
- Despliegues reproducibles (Entorno de pre, pro, etc. Sin pérdida o problemas con olvidos de librerías o sistemas en cuestión)
- Proporciona un aislamiento al despliegue de diferentes aplicaciones
- Los problemas de compatibilidad quedan prácticamente reducidos a la nada
- La velocidad de despliegue aumenta exponencialmente y ahorran costes en infraestructura (En una misma máquina puedo poner en marcha más sistemas que antes a igualdad de recursos)
- En esencia, es como si tuviéramos una Máquina Virtual sin los problemas de lentitud y penalizaciones en el consumo de recursos que tiene una Máquina Virtual

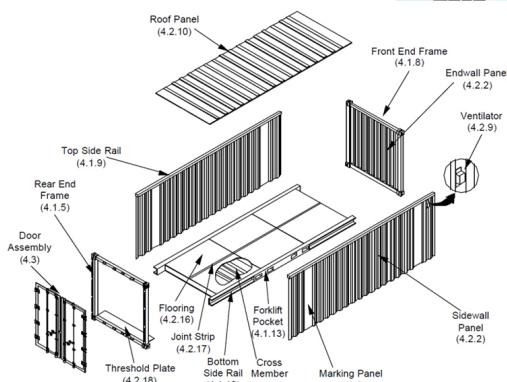
¿Qué beneficios obtengo respecto al DevOps?

- Si un container arranca bien el sistema, mientras no se altere la imagen, podré poner en marcha las réplicas del sistema que quiera de una forma sencilla y rápida
- Los despliegues quedan estandarizados y se pueden repetir cuanto se necesite
- Se acaban las inconsistencias entre entornos, (desarrollo, SQA, producción, etc.)
- Permite la segregación de responsabilidades (SoC)
- Aumenta la velocidad en entornos de integración continua (CI) y de entrega continua (CD)
- Aprovisionamiento menor de recursos hardware

Separación de Responsabilidades (SoC)

- **Equipo Desarrollo**

- Les interesa lo que hay dentro del container.
 - Código fuente
 - Librerías
 - Gestor de dependencias
 - Los datos
- Para el equipo todos los Servidores Linux son iguales



- **Equipo DevOps**

- Les interesa lo que hay fuera del contenedor
 - Login de acceso
 - Accesos remotos
 - Configuración de la red
 - Monitorización
- Todos los contendores se arrancan, paran, copian, pegan, migran de la misma forma



Situaciones típicas...

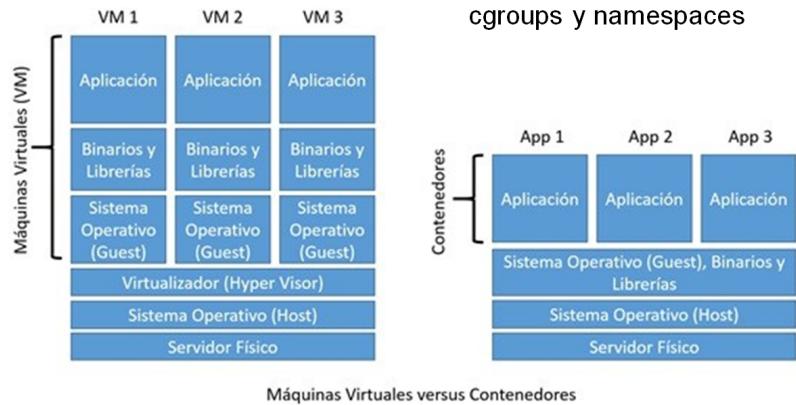


Sobre Docker

- Funciona en sistemas Linux
- Kernel 2.7+
- No tiene importancia la distribución
- Puede funcionar en hardware físico, ya sea nube o no
- Si funciona en el host... Funciona en el container
- Si la app funciona en un Kernel Linux, funcionará con Docker
- Orientación a sistemas del lado del servidor
- Virtualización de Alto Nivel, VM Ligera
- Espacio de proceso propio
- Configuración de red propia
- Puede funcionar como **root**
- Puede disponer de su propio **/sbin/init** (Su propio conjunto de herramientas de consola)
- A bajo nivel
 - Es un **chroot** con poderes :D
 - Cada container es un proceso aislado
 - Comparte Kernel con el host
 - No realiza emulación de dispositivo (Kernel)

Contenedores vs Maquinas Virtuales

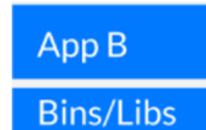
- Los contenedores están aislados, pero comparten OS y opcionalmente binarios y librerías
- El resultado...
 - Despliegues más rápidos
 - Menos coste de IT
 - Facilidad de migración
 - Facilidad de reinicio



¿Por qué los contenedores Docker son ligeros?



MVs
Cada cambio a una
Aplicación requiere una
nueva versión de la máquina
virtual



App Original
No hay OS que requiera
espacio adicional,
tampoco recursos que
deban de ser reiniciados

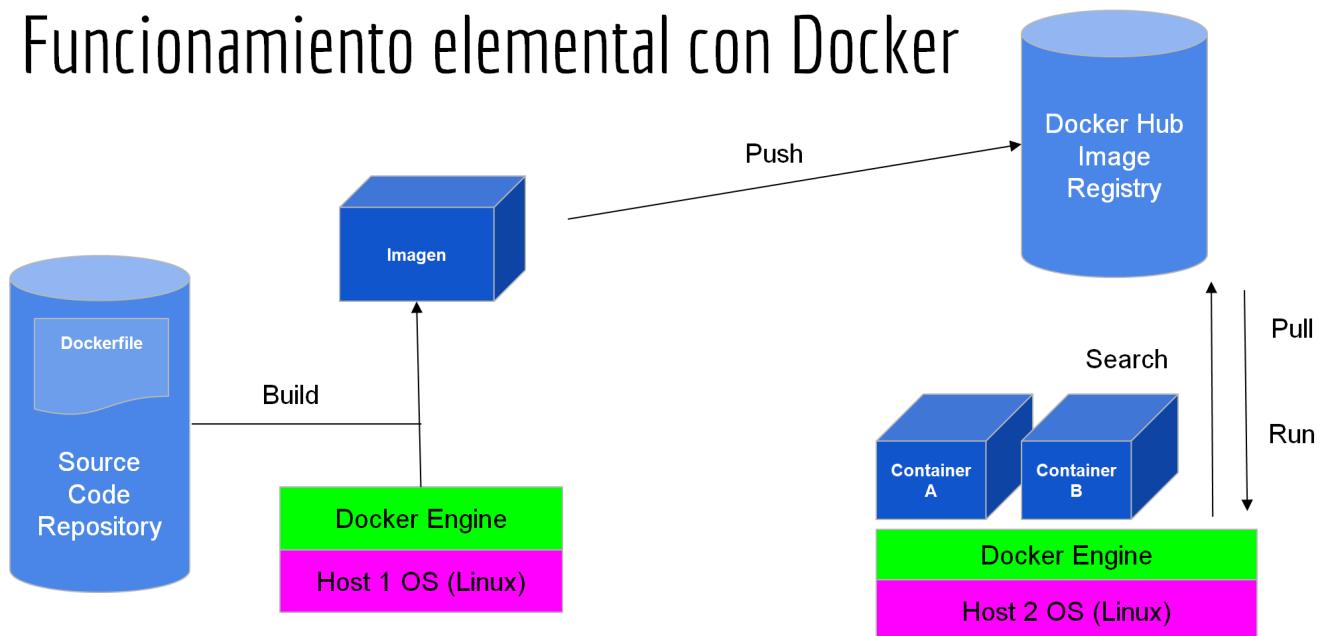


Copia de App
No hay OS
Puede compartir
binarios o librerías



Actualización
Sólo se copian en
el container el
delta o
incremental de la
aplicación

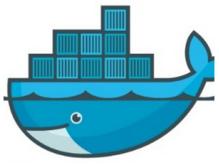
Funcionamiento elemental con Docker



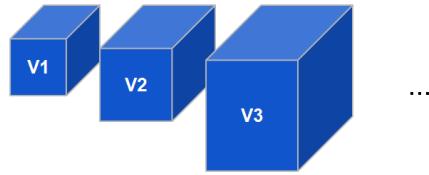
Tecnologías bajo Docker



- Linux Kernel
 - Servicios de aislamiento de recursos **cgroups**
 - Kernel namespaces
 - Union file-systems **aufs** (Sistema de archivo por capas)
 - **Libvirt, LXC**



- Git
 - Control de versiones: delta de las imágenes de contenedores
- Registro Docker Hub



Archivo Dockerfile

- Se puede decir que es la receta de como preparar un sistema con todo lo necesario para su puesta en funcionamiento
- Prepará una imagen específica de donde posteriormente sacar instancias (containers) para poner en marcha el sistema en sí
- Es un archivo de configuración que dice como se monta un container Docker

Dockerfile

```
# Apache example Server
```

```
FROM ubuntu:16.10
```

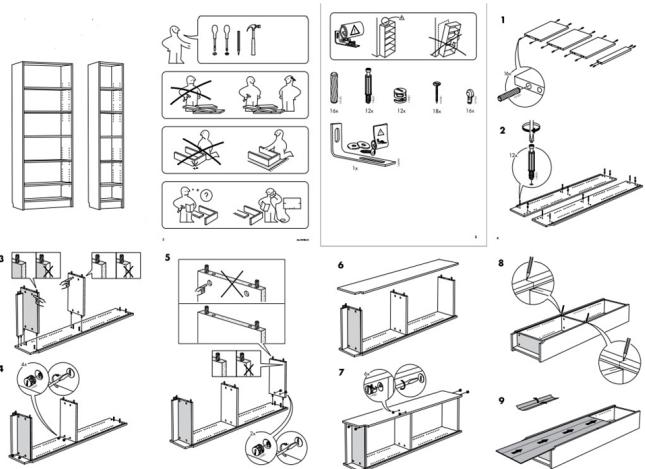
```
MAINTAINER Manuel Tomás Ortega Sánchez:  
1.0.0
```

```
RUN apt-get update && apt-get install -y apache2
```

```
ENV APACHE_LOG_DIR /var/log/apache2
```

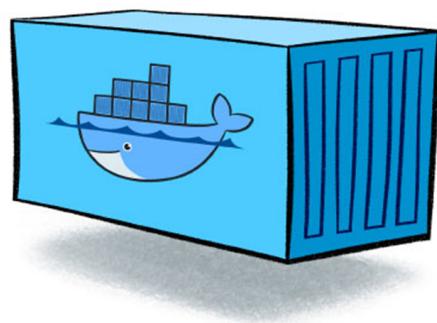
```
EXPOSE 80
```

```
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```



Container

- El container es el resultado de instanciar una imagen previamente definida por el Dockerfile
- Tenemos la aplicación paquetizada, aislada
- Una vez que le hemos asignado cuota de CPU, espacio de disco, red, etc.



Host (Anfitrión)

- Es el porta contenedores
- Puede ser local, nube privada, nube pública, formato de servidor virtual o físico.
- Expone los recursos disponibles
- Depende lo grande del buque... Cabrán más o menos containers
 - Dependiendo de CPU, Memoria, etc.

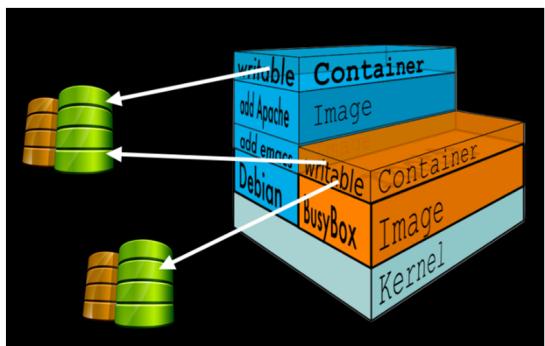


Imagen

- Fichero binario que contiene todo el sistema de ficheros de un contenedor
- Sistema de ficheros **Union**
- Estructurado en capas (layers) por delta

Volumen

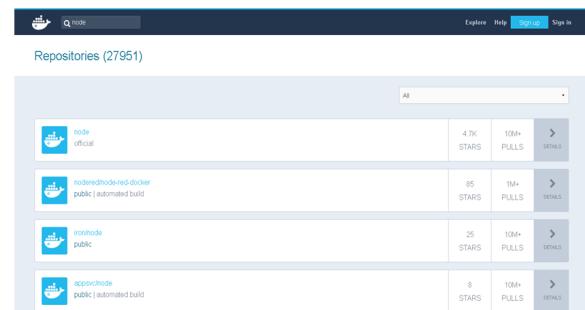
- Discos o directorios externos que podemos **montar** en el contenedor
- Recursos externos (alojados en el host) que sobreviven al contenedor
- Configuración / Datos /Recursos



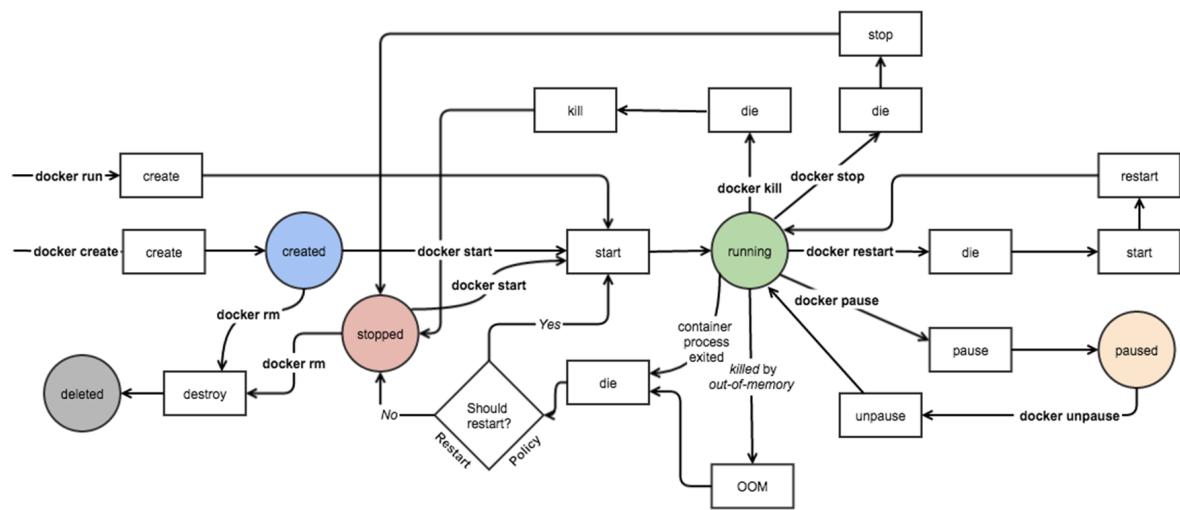
Registro

- Biblioteca de imágenes para poner en marcha containers
 - Totalmente operativas para poner en marcha
- Registro público
 - Compartidas por la comunidad
 - De libre acceso
- Registro privado
 - Containers corporativos o privados

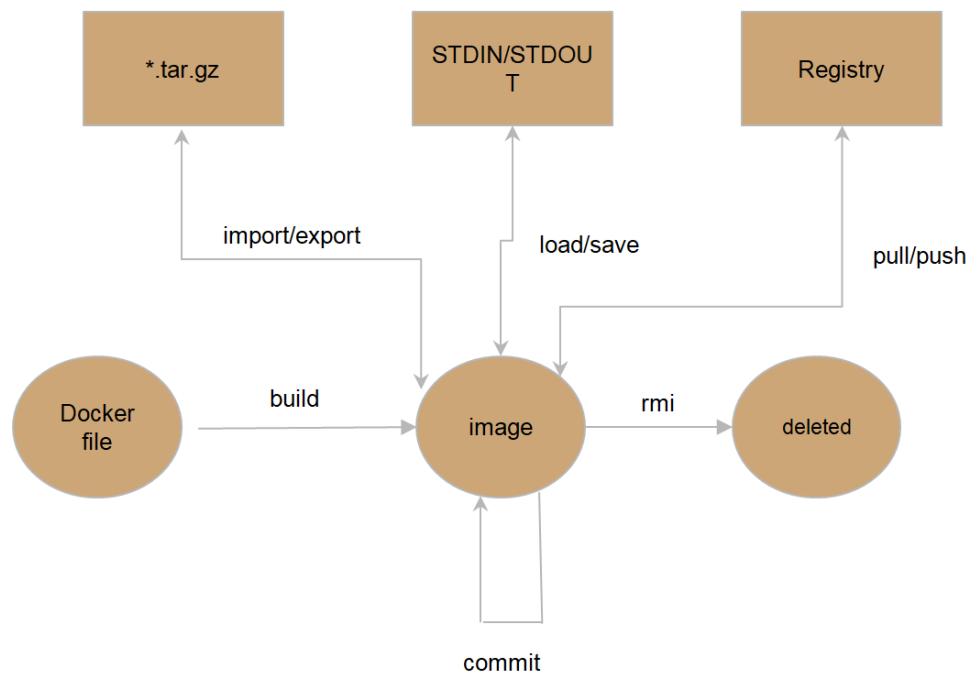
<https://hub.docker.com/>



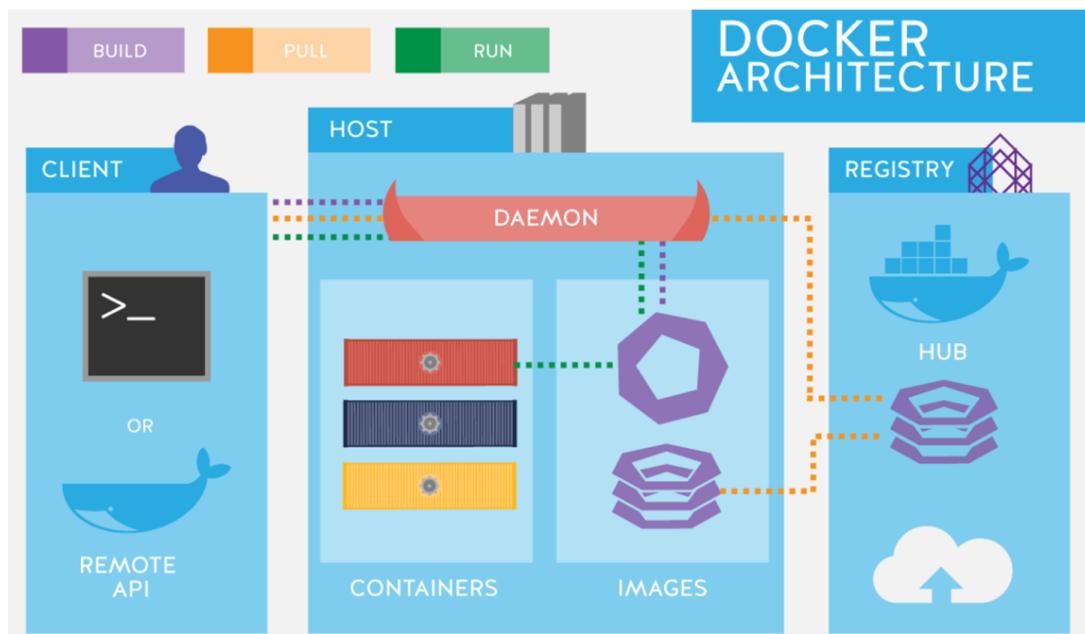
Ciclo de vida de un container



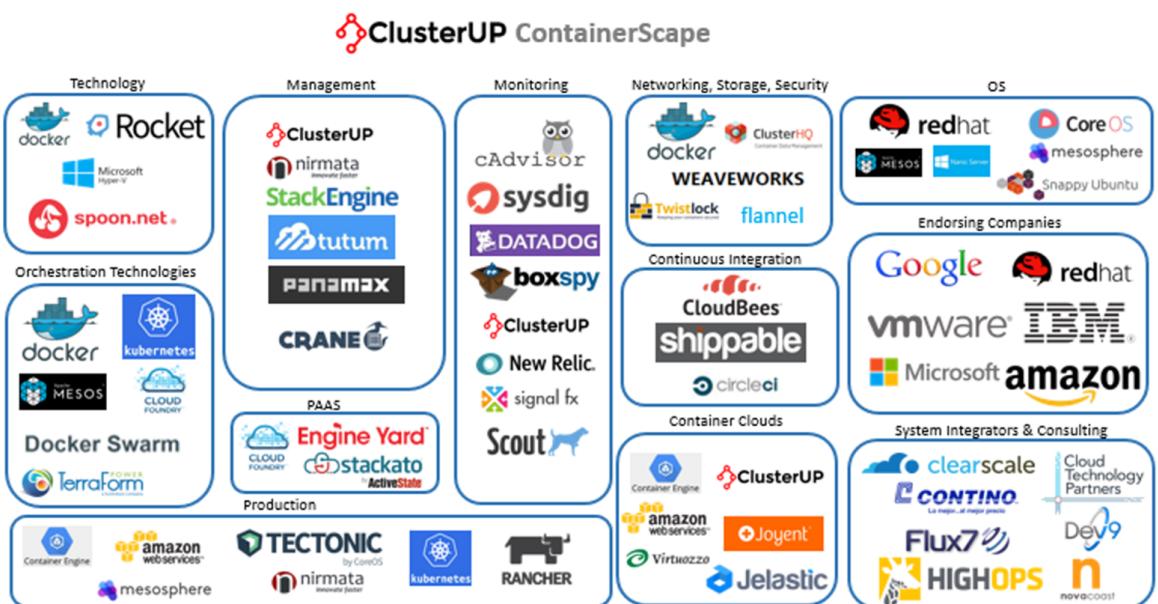
Ciclo de vida de una imagen



Infraestructura Docker



Ecosistema Docker



Características Generales

- **Portable:** Nos abstraemos del sistema operativo
- **Inmutable:** Empaque las dependencias necesarias
- **Ligero:** Usa cgroups y namespaces de linux

Orígenes de Docker

- La tecnología Docker, está basada en una tecnología que ya existía desde hace muchos años en tornos Linux, los LXC, o también llamados, linux containers
- Docker añade una capa interfaz línea de comandos de usuario, de forma que facilite toda la gestión

Lema de Docker

“Build, Ship & Run
Any Application, Anywhere”

Instalación

- Actualmente Docker dispone de dos versiones
 - Docker EE (Enterprise Edition)
 - Docker CE (Community Edition)
 - <https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/>
- Soporta la instalación en plataformas
 - Desktop
 - Cloud
 - Amazon AWS
 - Microsoft Azure
 - Servers
- Necesario x64 & Activación de instrucciones de virtualización

Capítulo 2. Instalación de Docker

- Docker está soportado en plataformas Linux con kernels superiores a 3.10.X
- Para obtener la última versión de Docker, debemos agregar los repositorios de Docker oficiales
- Una vez obtenido el repositorio oficial, descargamos el software

2.1. Prerrequisitos

- Se requiere un sistema de 64bit.
- Debe ser al menos un kernel de tipo 3.10+
- La nueva instalación de Docker permite separar dos versiones:
 - Docker-ce (community edition)
 - Docker-ee (enterprise edition)
- Desde Marzo de 2017 las versiones community pasaron de numerarse consecutivamente (1.10,1.11,...1.13,...) a tener versiones mensuales (AA.MM: 17.03 fue la primera) que se mantienen durante un mes (*Edge Releases*) o 3 meses (*Stable Releases*)
- Instalaremos la versión community edition para este curso.
- Sobre un entorno Linux podemos saber el kernel que poseemos

```
$ uname -r  
5.14.17-201.fc34.x86_64
```

- Se recomienda actualizar el sistema operativo para tener los últimos parches

```
$ sudo dnf -y update
```

2.2. Lab: Instalación en RHEL

- En nuestro caso, vamos a realizar una instalación de Docker con Fedora, ya que Fedora posee el mismo sistema de binarios que RedHat.
 - En caso de que no haya conectividad de internet en la máquina virtual, seguramente será por el uso de un proxy.
 - Para instalar software en Fedora a través de un proxy, debemos escribir las siguientes directivas dentro del fichero /etc/dnf/dnf.conf dentro del apartado [main]
 - Debemos conocer las directivas de Proxy para poder tener conexión a internet.



```
[main]
proxy=https://ip-proxy:puerto-proxy
proxy_username=usuario-autenticacion-proxy
proxy_password=contraseña-autenticacion-proxy
```

2.2.1. Instalación de docker con DNF

- Para instalarlo con DNF, instalamos primero las utilidades de DNF

Operación opcional

```
$ sudo dnf install -y dnf-utils
```

- Instalamos el repositorio de la edición community

```
$ sudo dnf config-manager --add-repo https://download.docker.com/linux/fedora/docker-ce.repo
```

- Actualizamos la caché de dnf

Operación opcional

```
$ sudo dnf makecache
```

- Instalamos el paquete Docker

```
$ sudo dnf -y install docker-ce
```

- Iniciamos el daemon de docker

```
$ sudo systemctl start docker
```

- Definimos el servicio como automático para los siguientes reinicios del sistema

```
$ sudo systemctl enable docker
```

- En caso de que poseamos un proxy, también hay que definir las variables que pasaremos al cliente para que en la ejecución del contenedor aplique las directivas de proxy.
- Con esto conseguimos que las descargas desde el contenedor como yum update funcionen a través del proxy.



```
{  
  "proxies":  
  {  
    "default":  
    {  
      "httpProxy": "http://127.0.0.1:3001",  
      "httpsProxy": "http://127.0.0.1:3001",  
      "noProxy": "*.test.example.com,.example2.com"  
    }  
  }  
}
```

2.2.2. Prueba de ejecución

- Comprobamos la ejecución de un contenedor básico de docker

```
$ sudo docker run hello-world
```

- La imagen se descargará y ejecutará

- En caso de que no haya conectividad con el registro externo, podemos configurar un proxy
- Para ello creamos el fichero http-proxy.conf en /etc/systemd/system/docker.service.d/ con el siguiente contenido

/etc/systemd/system/docker.service.d/http-proxy.conf

```
[Service]
Environment="HTTP_PROXY=http://ip-proxy:puerto-proxy/"
```

- En caso de necesitar un proxy https

/etc/systemd/system/docker.service.d/https-proxy.conf

```
[Service]
Environment="HTTPS_PROXY=https://ip-proxy:puerto-proxy/"
```

- Se puede definir también aquellas entradas sin proxy

/etc/systemd/system/docker.service.d/no-proxy.conf

```
[Service]
Environment="NO_PROXY=localhost,127.0.0.1,192.168.250.101,..."
```

- Podemos definirlo todo en el primer fichero sin problemas, pero así está clasificado.

2.2.3. Instalación en grupo docker

- Por defecto, docker se ejecuta como usuario Root.
- Podemos configurar un usuario para que ejecute los procesos de Docker
- Según el tipo de instalación, el grupo docker debería estar creado, sino se debe crear y reiniciar el servicio

Ejecución de Docker no-root

- Agregamos si no existe el grupo docker

Operación opcional, solo si el grupo no existe

```
$ sudo groupadd docker
```

- Agregamos el usuario al grupo docker

```
$ sudo usermod -aG docker $USER
```



Al ejecutar la instrucción, se sustituirá de forma automática, el token \$USER por el login del usuario con el que estemos conectados

- Salimos y entramos a la sesión, y podremos ejecutar procesos de Docker.
- Debemos ser conscientes de que el usuario docker es similar a root debido a sus permisos especiales. No se debe otorgar a la ligera ni publicar apis externas que controlen usuarios del grupo docker

2.2.4. Reinicio del perfil del usuario

Para que todas las capas de la shell que corresponden al perfil de usuario contemplen el cambio de inclusión del usuario en un nuevo grupo, será necesario reiniciar dichas capas.

Ejecutamos el siguiente comando en consola:

```
$ sudo systemctl restart gdm
```

2.2.5. Información básica del motor Docker Engine

Una vez llevada a cabo la instalación de forma correcta, vamos a ejecutar el comando que nos da información sobre las versiones del cliente de interacción con Docker así como la versión del servidor:

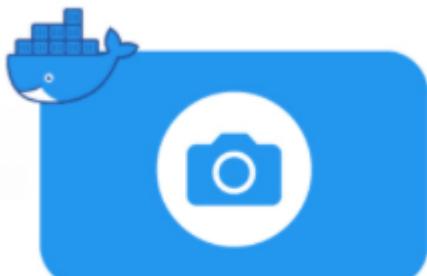
```
$ docker version
```

También procedemos a ejecutar y comentamos el detalle de otro comando interesante, que nos proporciona información detallada sobre el propio motor Docker Engine, informándonos entre otras cosas:

- Las imágenes que tenemos
- La cantidad de contendores
- Versión del kernel que tiene el nodo
- Etc.

```
$ docker info
```

Capítulo 3. Imágenes Docker

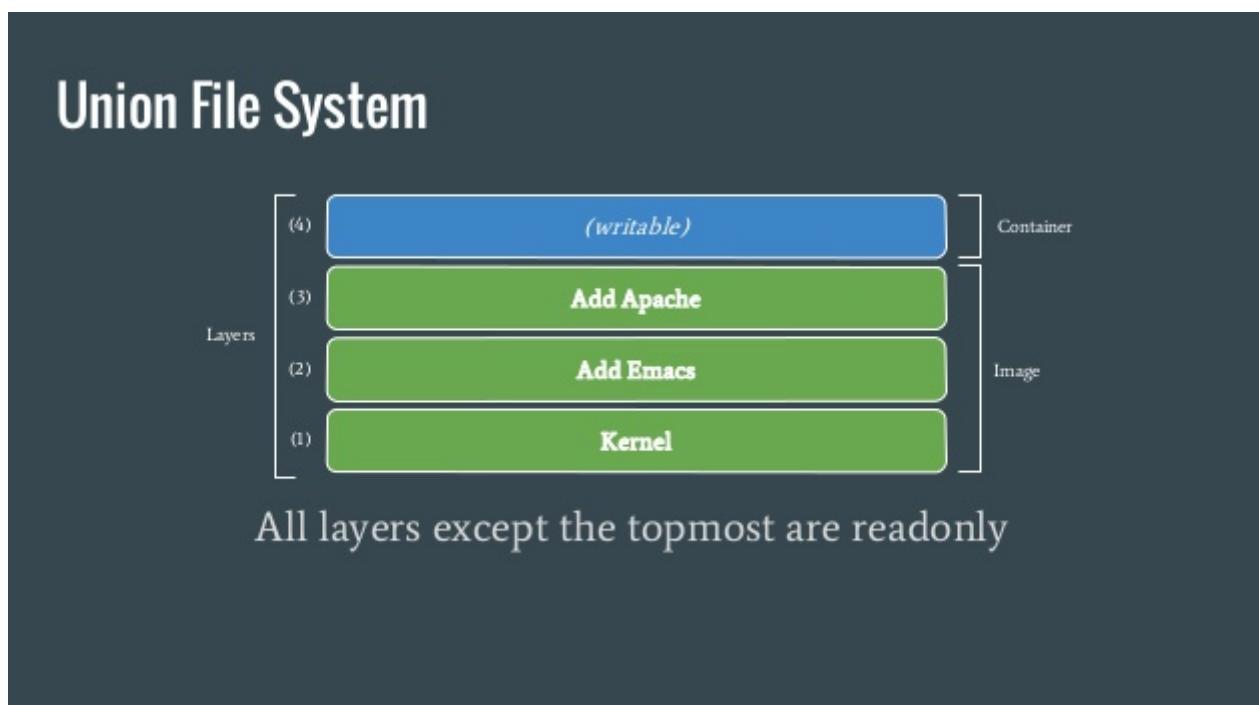


Docker Image

La imagen en el mundo docker, es el fichero binario que contiene todo el sistema de ficheros, que posteriormente podrá utilizar un contenedor.

Sistema de ficheros **Union**

Las imágenes se encuentran estructuradas por capas (layers) delta, de manera que se pueda tener capacidad de re-construir el binario, compilando únicamente aquellas partes que hayan cambiado.



Cuando hacemos referencia a una imagen Docker, el formato estándar sería el siguiente:

- <nombre-imagen-docker>:<version>

3.1. Criterio de la comunidad al nombrar imágenes docker

Cuando estemos creando nuestras propias imágenes docker, es de vital importancia establecer una buena nomenclatura/versionado para poder tener siempre una gestión profesional de los sistemas, su actualización y posterior despliegue.

En principio, en cuanto al nombre de las imágenes docker, la comunidad establece el siguiente criterio:

- <nombre-empresa/sistema>

Por ejemplo, si tenemos en cuenta estos datos:

- **Nombre de la empresa**
 - mycompany1
- **Nombre del producto**
 - bill-system
- **Versión del sistema compilado**
 - 2.0.5

Como resultado de nombre de imagen podríamos tener:

- mycompany1/bill-system:2.0.5

3.2. Sobre los Tags en docker

Docker tiene un tag especial, denominado **latest**.

Cuando descargamos una imagen del hub, o bien nosotros creamos nuestra propia imagen, docker siempre requiere de forma explícita que indiques una versión en concreta.

Si no indicamos la versión de forma explícita al arrancar un contenedor, por ejemplo:

```
$ docker run ubuntu
```

Lo que estamos haciendo de forma implícita, es como indicáramos esto:

```
$ docker run ubuntu:latest
```

También debemos de tener en cuenta, que cuando queremos ejecutar un contenedor de una imagen y una versión específica, docker primero le echa un vistazo al propio docker engine local, si la imagen ya se encuentra en el propio docker engine, docker no descarga nada del hub...

Esto que acabamos de comentar puede ser un problema ya que si nos estamos gestionando, de forma que siempre utilizamos el mismo TAG cuando regeneramos la imagen docker con los últimos cambios de desarrollo... Al ejecutar el comando anterior... Como docker detecta que la versión en concreto de esa imagen ya se encuentra en el propio docker engine local... ¡No descarga los cambios del hub!

Este problema podríamos solventarlo, ejecutando en tandem 2 comandos, un comando de descarga explícita, seguido de un comando de ejecución explícita, para una versión latest que se ha regenerado con cambios en el hub, y queremos actualizarnos los cambios locales y proceder al

arranque de un contenedor fresco, ejecutaríamos lo siguiente:

```
$ docker pull ubuntu:latest && docker run ubuntu:latest
```

 Llevar a cabo una gestión de compilación - despliegue de esta forma es un grave error, por que por un lado, desarrollo iría con un versionado y operaciones con despliegue por otro... Sin tener en principio sentido lógico las imágenes docker respecto a los sistemas internos que están desplegando.

3.3. Recomendaciones en la gestión de imágenes docker

- Indica como versión de la imagen docker, la misma con la que desarrollo haya etiquetado la release de la aplicación
 - Esto facilita cuando detectes una incidencia en producción, que indiques la misma versión de la imagen docker que desarrollo tiene con el TAG del código fuente
- Nunca lleves a cabo despliegues en producción de versiones latest
 - Uno de los problemas de realizar esto, es que si sale alguna incidencia... Lo único que vas a poder decirle a desarrollo, es que la incidencia en producción se está experimentando en la versión latest... Y desarrollo seguramente dirá, que ellos han liberado por ejemplo, la versión 2.0.4 y la 2.0.5, que para corregir la incidencia es necesario indicar la versión del sistema en producción concreta...

3.4. Lab: Imágenes Docker

Mediante este laboratorio vamos a practicar operaciones básicas de gestión con imágenes docker.

3.4.1. Descarga de imágenes (docker pull)

Vamos a practicar la descarga de una imagen docker.

- Indicamos que queremos descargar la imagen ubuntu
- No indicamos versión, por lo que el sistema descargará la última que haya disponible

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu
54ee1f796a1e: Pull complete
f7bfea53ad12: Pull complete
46d371e02073: Pull complete
b66c17bbf772: Pull complete
Digest: sha256:31dfb10d52ce76c5ca0aa19d10b3e6424b830729e32a89a7c6eee2cda2be67a5
Status: Downloaded newer image for ubuntu:latest
```

Observamos la descarga de las diferentes capas que componen la imagen Docker

Volvemos a ejecutar el comando, de la misma imagen y la misma versión, en este caso, lo que vamos a observar, es que docker primero verifica que en nuestro docker engine local ya existe la imagen y no procede a la descarga:

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu
Digest: sha256:31dfb10d52ce76c5ca0aa19d10b3e6424b830729e32a89a7c6eee2cda2be67a5
Status: Image is up to date for ubuntu:latest
```

También observamos que el ID de la imagen, es exactamente igual en todas las descargas, ya que, mientras que la imagen no se regenere, el ID sha256 será el mismo.

3.4.2. Guardado de imágenes (docker save)

Otra opción interesante, consiste en guardar las imágenes docker que se encuentren en el propio docker engine, en un archivo comprimido, para posteriormente poder llevarlas a otra máquina, o bien, realizar una copia de seguridad en caso necesario.

Vamos a guardar la imagen que hemos descargado previamente, la imagen **ubuntu:latest**:

```
$ docker save ubuntu:latest | gzip > ubuntu_latest.tar.gz
```

Si listamos el sistema de archivos, observaremos que ha aparecido un nuevo archivo comprimido con nombre **ubuntu_latest.tar.gz**

3.4.3. Listando las imágenes (docker images)

Podemos visualizar las imágenes que tenemos en nuestro docker engine, ejecutando el comando:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	4e2eef94cd6b	3 weeks ago	73.9MB

Observamos diferentes columnas en la información que observamos:

- **REPOSITORY**

- El nombre de la imagen docker también se le conoce como el repositorio

- **TAG**

- La versión de la imagen en cuestión, en este caso se trata de una imagen etiquetada como latest

- **IMAGE ID**

- Identificador único de imagen, hash sha256

- Este Id lo genera automáticamente el docker engine cada vez que una imagen es reconstruida

- **CREATED**

- Esta es la fecha de la última vez que realizó una operación de construcción (build) contra esa imagen

- Esta fecha puede dar lugar a confusión, ya que no es la fecha en la que la imagen se creó en el docker engine, sino, como se ha comentado, la fecha en la que se realizó la última construcción

- **SIZE**

- Nos indica el tamaño

3.4.4. Eliminando imágenes (docker rmi)

Cuando tengamos imágenes docker en nuestro sistema que ya no vayamos a utilizar, podemos eliminarlas.

Podemos indicar de forma explícita incluso el TAG en cuestión que queramos eliminar de dicha imagen.



Deberemos de recordar que cuando eliminamos una imagen, si no somos explícitos y no indicamos el TAG, docker entenderá que queremos eliminar la imagen con el TAG latest

Ahora, vamos a eliminar una imagen que está presente en nuestro docker engine, ejecutando el siguiente comando:

```
$ docker rmi ubuntu:latest  
Untagged: ubuntu:latest  
Untagged: ubuntu@sha256:31dfb10d52ce76c5ca0aa19d10b3e6424b830729e32a89a7c6eee2cda2be67a5  
Deleted: sha256:4e2eef94cd6b93dd4d794c18b45c763f72edc22858e0da5b6e63a4566a54c03c  
Deleted: sha256:160004bdd9a2800d0085be0315b769a9ce04c07ca175ecae89593eeee9aeb944  
Deleted: sha256:9ed638911072c3379e75d2eaf7c2502220d6757446325c8d96236410b0729268  
Deleted: sha256:ce7da152e578608030e9a05f9f5259b329fe5dcc5bf48b9f544e48bd69a5f630  
Deleted: sha256:2ce3c188c38d7ad46d2df5e6af7e7aed846bc3321bdd89706d5262fef6a3390
```

El sistema informa sobre las capas que han sido eliminadas.

Listamos de nuevo las imágenes del docker engine, para confirmar que ya no está presente dicha imagen:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

3.4.5. Carga de imágenes (docker load)

Si tenemos una imagen previamente guardada con formato comprimido, podemos llevar a cabo la inyección en nuestro docker engine, en lugar de descargarla de un hub.

Vamos a cargar en nuestro docker engine la imagen comprimida que tenemos en el archivo **ubuntu_latest.tar.gz**:

```
$ docker load < ubuntu_latest.tar.gz  
2ce3c188c38d: Loading layer [=====] 75.23MB/75.23MB  
ad44aa179b33: Loading layer [=====] 1.011MB/1.011MB  
35a91a75d24b: Loading layer [=====] 15.36kB/15.36kB  
a4399aeb9a0e: Loading layer [=====] 3.072kB/3.072kB  
Loaded image: ubuntu:latest
```

Observamos el proceso de carga de la imagen en nuestro docker engine, inyectando las capas que son necesarias.

Listamos de nuevo las imágenes del docker engine, en este caso confirmamos que la imagen vuelve a estar presente en nuestro docker engine:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	4e2eef94cd6b	3 weeks ago	73.9MB

3.4.6. Etiquetado de imágenes (docker tag)

Otro comando que puede resultar útil, es llevar a cabo un re-etiquetado de una imagen docker que ya existe en nuestro docker engine.

Vamos a re-etiquetar la imagen ubuntu:latest al tag... pako:1.0.4 :D

Ejecutamos el siguiente comando:

```
$ docker tag ubuntu:latest pako:1.0.4
```

Listamos de nuevo las imágenes del docker engine, en este caso confirmamos la existencia de dos imágenes en nuestro docker engine:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
pako	1.0.4	4e2eef94cd6b	3 weeks ago	73.9MB
ubuntu	latest	4e2eef94cd6b	3 weeks ago	73.9MB

Lo que estamos observando realmente, no son dos imágenes docker diferentes, si atendemos al hash de la columna IMAGE ID, vamos a observar que nos aparecen los primeros 12 caracteres del sha256.

Realmente se trata de la misma imagen, ya que el conjunto de capas que conforman la imagen, realmente no ha cambiado... ¡Tenemos 2 punteros que apunta al mismo bloque de capas!



Otro aspecto también a tener en cuenta, es que la columna SIZE indica lo que ocupa ese conjunto de capas, en este caso **no significa que lo que tenemos ahora mismo en disco ocupe 147,8MB**, sino, únicamente 73.9MB, ya que, como hemos comentado, se trata de 2 punteros que apuntan a un mismo conjunto de capas.

Podemos crear tantos TAG's de una imagen como queramos, en esencia serán punteros a modo de referencia, que en ningún caso (mientras la imagen sea exactamente la misma), producirá duplicidad de capas o aumento de tamaño en disco.

Capítulo 4. Docker Hub

Docker Hub es un servicio proporcionado por Docker para buscar y compartir imágenes de diferentes sistemas para facilitar la gestión y despliegue de las mismas.

Se estaba dando la dualidad de tener 2 portales hub diferentes en el mundo Docker.

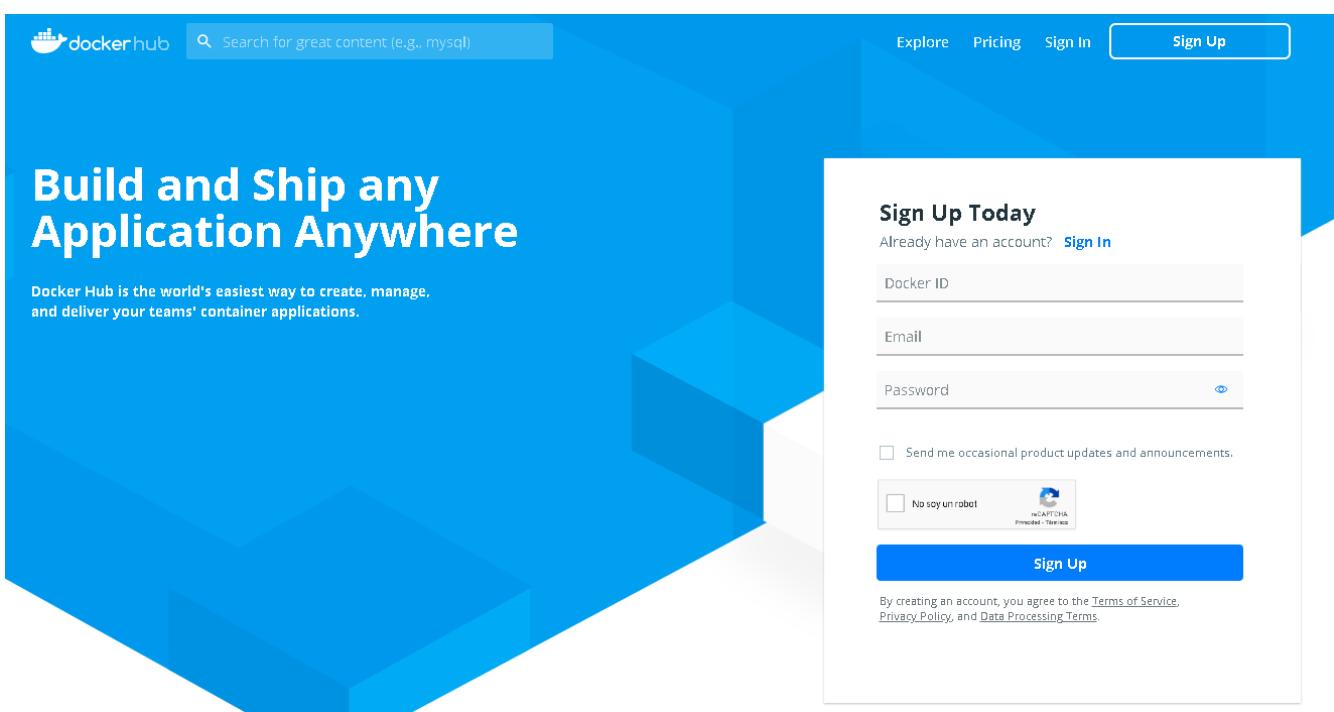
Desde últimos de 2018 - principios de 2019, Docker realizó la fusión de los portales **Docker Store** y **Docker Cloud**, pasando a ser unificados en un único portal, denominado **Docker Hub**.

Esta unificación proporciona una única experiencia en cuanto a las imágenes, nos permite:

- Encontrar
- Almacenar
- Compartir

De una forma sencilla y rápida.

Si accedemos a la URL: <https://hub.docker.com/>, observaremos la pantalla de inicio de dicho portal.



Millones de usuarios y más de cien mil organizaciones utilizan Docker Hub, Store y Cloud para sus necesidades de contenido de imágenes.

4.1. Características del portal

Repositorios

Los repositorios en el mundo Docker, son lo que serían un proyectos a nivel software en desarrollo.

- 1 repositorio en el mundo docker, tiene únicamente almacenada una imagen de un sistema específico.

- 1 repositorio puede tener 1 o más TAG's.
- Permite llevar a cabo filtrado de búsquedas para encontrar aquella imagen que más nos encaje.

Organizaciones y Equipos

Podemos utilizar docker hub de dos formas:

- **Modo Personal (Modo Gratuito)**
 - Uso gratuito de descarga de imágenes, sin necesidad de estar registrados
 - Sólo admite 1 imagen privada
 - El resto de imágenes tienen que ser forzosamente públicas y cualquiera podría descargarlas
- **Modo Empresarial (Modo Pago)**
 - Uso gratuito de descarga de imágenes, sin necesidad de estar registrados
 - Debemos de pasar a un plan de pago si queremos custodiar imágenes
 - Las imágenes serán privadas y dependiendo del plan que elijamos, tendremos una cantidad de imágenes (repositorios) que podremos custodiar
 - Permite llevar a cabo la implementación de builds automáticas, si vinculamos nuestra cuenta de docker hub con una cuenta de repositorio de código fuente donde se encuentre custodiado el archivo de construcción Dockerfile

Búsqueda mejorada de imágenes

Permite llevar a cabo filtros de imágenes por criterios como:

- Imágenes Oficiales acreditadas
- Imágenes que han sido Verificadas y Certificadas
- Categoría del sistema que estamos buscando (Servidores web, bases de datos, etc.)

Esto nos garantizando un nivel de calidad en las imágenes Docker listadas por la consulta de búsqueda que hayamos realizado.

Imágenes oficiales y Verified publisher

Al igual que las Imágenes Oficiales, las imágenes Verified Publisher, han sido examinadas por Docker.



Mientras que Docker mantiene la biblioteca de imágenes oficiales, las imágenes Verified Publisher y Certified Images son proporcionadas por proveedores de software externos.

Los proveedores interesados pueden inscribirse en la siguiente URL: <https://goto.docker.com/Partner-Program-Technology.html>.

Imágenes certificadas

Las imágenes certificadas también están disponibles en Docker Hub.

Las imágenes certificadas son una categoría especial de imágenes de Verified Publisher que superan los requisitos adicionales de calidad, mejores prácticas y soporte de Docker:

- Han sido probadas y soportadas en la plataforma Docker Enterprise por editores verificados.
- Se adhieren a las mejores prácticas de contenedores de Docker
 - Podemos encontrar más información sobre estas buenas prácticas en la siguiente URL:
<https://docs.docker.com/docker-hub/publish/#create-great-content>
- Pasar una serie de pruebas funcionales de API.
- Completar una evaluación de exploración de vulnerabilidades.
- Proporcionado por socios con una relación de apoyo de colaboración.
- Llevar una marca de calidad única **Docker Certified**.

El portal Docker Hub se encuentra en constante evolución y no es de extrañar que nos vayamos encontrando en espacios de tiempo relativamente cortos, mejoras o cambios.

4.2. Lab: Docker Hub

Mediante este laboratorio, vamos a llevar a cabo tareas relacionadas con el portal docker hub.

4.2.1. Registro en el portal web

Lo primero que vamos a realizar, es llevar a cabo el registro en el portal Docker Hub, esto nos permitirá descargar imágenes propias, así como posteriormente subir

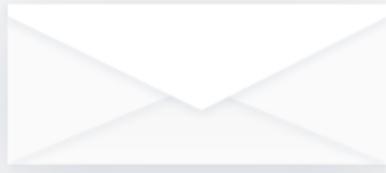
Para registrarnos, accedemos a la URL: <https://hub.docker.com/>

Rellenamos el formulario con nuestros datos y pulsamos sobre el botón **Sign Up**

Elegimos el plan **Free**

The screenshot shows the 'Choose a Plan' section of the Docker Hub website. It features three plans: 'Free', 'Pro', and 'Team'.
Free: \$0/month. Includes: Unlimited public repositories, 1 private repository, Community support. A red circle highlights the 'Continue with Free' button.
Pro: \$5/month. With annual plan. Includes: Unlimited public repositories, Unlimited private repositories, 2 parallel builds, Email support.
Team: Starts at \$25 for 5 users. Includes: Unlimited public repositories, Unlimited private repositories, User management with role-based access controls, Unlimited teams, 3 parallel builds, Email support.
Each plan has a 'Buy Now' button below it.

La plataforma nos enviará un correo en unos minutos para verificar la cuenta, confirmamos y listo, ya tendremos nuestra cuenta operativa.



Please verify your email address

Great! You're almost there. Before you can create a repository or configure Docker Hub, you'll need to verify your email address.

We've sent a verification email to

4.2.2. Búsqueda de imágenes (docker search)

Además del portal web, para casos en los que tengamos que comprobar de forma rápida la existencia de una imagen docker específica, disponemos del comando de consola **docker search**

El uso genérico del comando sería:

- docker search <texto_busqueda>

Vamos a buscar coincidencias del servidor web nginx, a ver que encontramos.

Ejecutamos el siguiente comando en consola:

```
$ docker search nginx
```

NAME	DESCRIPTION	STARS	OFFICIAL
AUTOMATED			
nginx	Official build of Nginx.	13731	[OK]
jwilder/nginx-proxy	Automated Nginx reverse proxy for docker containerization	1872	
[OK]			
richarvey/nginx-php-fpm	Container running Nginx + PHP-FPM capable of serving static files and PHP scripts	786	
[OK]			
linuxserver/nginx	An Nginx container, brought to you by LinuxServer	128	
tiangolo/nginx-rtmp	Docker image with Nginx using the nginx-rtmp module	91	
[OK]			
bitnami/nginx	Bitnami Nginx Docker Image	90	
[OK]			
jc21/nginx-proxy-manager	Docker container for managing Nginx proxy hosts	87	
alfg/nginx-rtmp	NGINX, nginx-rtmp-module and FFmpeg from source	76	
[OK]			
nginxdemos/hello	NGINX webserver that serves a simple page containing your message	60	
[OK]			
jlesage/nginx-proxy-manager	Docker container for Nginx Proxy Manager	53	
[OK]			
nginx/nginx-ingress	NGINX Ingress Controller for Kubernetes	41	
privatebin/nginx-fpm-alpine	PrivateBin running on an Nginx, php-fpm & Alpine Linux stack	35	
[OK]			
schmunk42/nginx-redirect	A very simple container to redirect HTTP traffic to HTTPS	19	
[OK]			
nginxinc/nginx-unprivileged	Unprivileged NGINX Dockerfiles	17	
nginx/nginx-prometheus-exporter	NGINX Prometheus Exporter	15	
centos/nginx-112-centos7	Platform for running nginx 1.12 or building nginx modules	14	
centos/nginx-18-centos7	Platform for running nginx 1.18 or building nginx modules	13	
raulr/nginx-wordpress	Nginx front-end for the official wordpress:fast image	13	
[OK]			
sophos/nginx-vts-exporter	Simple server that scrapes Nginx vts stats and exposes them via Prometheus	7	
[OK]			
bitwarden/nginx	The Bitwarden nginx web server acting as a reverse proxy	7	
mailu/nginx	Mailu nginx frontend	7	
[OK]			
bitnami/nginx-ingress-controller	Bitnami Docker Image for NGINX Ingress Controller	6	
[OK]			
flashspys/nginx-static	Super Lightweight Nginx Image	6	
[OK]			
wodby/nginx	Generic nginx	1	
[OK]			
ansibleplaybookbundle/nginx-apb	An APB to deploy NGINX	1	
[OK]			

Observamos que nos aparecen un montón de coincidencias, una lista donde el nombre de la imagen contenga la palabra **nginx**.

En la lista podemos observar las columnas que nos dan información de:

- **NAME**

- El nombre del repositorio en sí, el nombre de la imagen

- **DESCRIPTION**

- La descripción de la imagen que ha realizado el autor de la misma

- **STARS**

- El otorgar estrellas a un repositorio de Docker funciona de forma parecida a cuando otorgamos estrellas a un repositorio de código fuente en GitHub por ejemplo

- Si un repositorio tiene más estrellas, más popular es, y saldrá en primeras posiciones ante búsquedas de dicha plataforma

- **OFFICIAL**

- Indica si la imagen está acreditada como oficial

- **AUTOMATED**

- Indica si el repositorio de imagen lleva ajustado algún tipo de pipeline, de forma que cuando se detecten cambios en el archivo fuente de construcción Dockerfile, se desencadene de forma automática la reconstrucción de la imagen

4.2.3. Búsqueda de imágenes en el portal web

Cuando buscamos una imagen docker a través del portal web, obtenemos ciertos detalles visuales que nos dan un plus respecto a la búsqueda por consola.

Buscamos mediante el portal web, la imagen **wordpress**

The screenshot shows the Docker Hub search interface. At the top, there's a search bar with 'wordpress' typed in. Below it, a navigation bar includes 'Explore', 'Repositories', 'Organizations', and 'Get Help'. Underneath, tabs for 'Docker', 'Containers', and 'Plugins' are visible, with 'Containers' being the active tab. On the left, there are 'Filters' for 'Docker Certified' (unchecked), 'Verified Publisher' (unchecked), and 'Official Images' (unchecked). The main search results area displays 1 - 25 of 7987 results for 'wordpress'. A specific result for 'wordpress' is highlighted, showing its official Docker image. The image has a circular icon with a white 'W' inside, the name 'wordpress' in bold, and a note that it was updated 10 hours ago. Below the image, a brief description states: 'The WordPress rich content management system can utilize plugins, widgets, and themes.' A horizontal bar below the image lists supported architectures: Container, Linux, PowerPC 64 LE, x86-64, 386, ARM 64, IBM Z, mips64le, ARM, Application Services.

Hacemos clic sobre la misma para que nos lleve al detalle.

The screenshot shows the detailed view of the 'wordpress' Docker image on Docker Hub. At the top, the navigation bar shows 'Explore > wordpress'. To the right, it says 'Using 0 of 1 private repositories. [Get more](#)'. Below the navigation, there's a large image of the WordPress logo (a stylized 'W' inside a circle). The image is labeled 'wordpress' with a star icon and 'Docker Official Images'. A brief description follows: 'The WordPress rich content management system can utilize plugins, widgets, and themes.' A download count of '500M+' is shown with a downward arrow icon. Below the image, a horizontal bar lists supported architectures: Container, Linux, ARM, PowerPC 64 LE, x86-64, 386, ARM 64, IBM Z, mips64le, Application Services. An 'Official Image' button is also present. To the right, there's a dropdown menu set to 'Linux - ARM (latest)', a link to copy/paste the pull command ('Copy and paste to pull this image'), a command input field containing 'docker pull wordpress' with a copy icon, and a link to 'View Available Tags'. At the bottom, there are tabs for 'Description' (which is currently selected), 'Reviews', and 'Tags'. A 'Quick reference' section at the very bottom contains links to 'Maintained by: the Docker Community' and 'Where to get help: the Docker Community Forums, the Docker Community Slack, or Stack Overflow'.

En el detalle de la misma, podemos consultar elementos como:

- **Description**

- Una descripción bastante elaborada, dependiendo de la imagen encontraremos más o menos grado de detalle

- **Reviews**

- Anotaciones o valoraciones que pueden realizar los usuarios de la comunidad, con algún dato o prueba relevante

- **Tags**

- Información sobre los tags que han sido liberados del repositorio
- Información sobre el sistema operativo/arquitectura sobre la que funciona
- Tamaño que ocupa cada tag
- Los identificadores sha256 de las capas que conforman la imagen

Relacionado con los tags, otro punto muy interesante, es que si hacemos clic sobre el identificador de cualquier tag en concreto, por ejemplo, de la versión **latest**

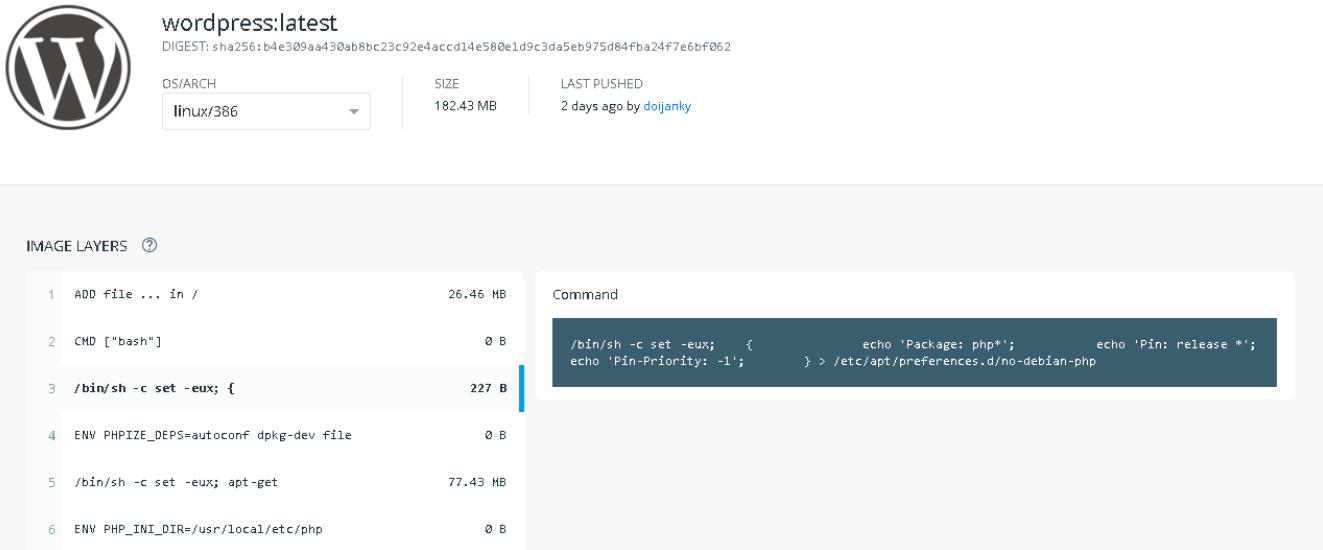
The screenshot shows the Docker Hub interface for the 'wordpress' image. The 'Tags' tab is selected. A search bar at the top says 'Filter Tags'. On the right, there's a 'Sort by' dropdown set to 'Latest'. The 'latest' tag is highlighted with a red circle. Below it, the page shows the following information:

IMAGE	DIGEST	OS/ARCH	COMPRESSED SIZE
latest	b4e309aa430a	linux/386	182.43 MB
	7c7f8e8a2d33	linux/amd64	177.12 MB
	cac6b0c7b928	linux/arm/v5	154.98 MB
	+5 more...		

At the bottom right, there's a button labeled 'docker pull wordpress:latest' with a blue icon.

Al acceder al detalle, vamos a observar el conjunto de instrucciones que son aplicadas para conformar el tag de la imagen.

También podemos observar detalles interesantes, como el tamaño que ocupa cada capa, el comando que se ha ejecutado exactamente, así como indicación del tamaño total que el tag pesa.



Si por el contrario, realizamos una búsqueda de otro tipo de imágenes propietarias, como por ejemplo, la base de datos propietaria de Microsoft, Sql Server, podremos observar información sustancialmente diferente.

En el buscador indicamos **Microsoft SQL Server** y navegamos hacia la misma

4.2.4. Realizando login por consola

Para poder descargar imágenes propietarias, así como para poder subir las nuestras propias, debemos de realizar login por consola e introducir nuestro ID de la cuenta docker con la que nos hemos registrado en el portal web.

Ejecutamos el siguiente comando:

```
$ docker login <credentials>

Username: <usuario>
Password: <password>

WARNING! Your password will be stored unencrypted in /etc/docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

En cuanto a **credentials**, tenemos dos opciones.



Si indicamos únicamente nuestro ID de usuario del portal Docker Hub con el que previamente nos hemos registrado, estaremos haciendo login directamente sobre dicho portal.

En el caso de que queramos loguearnos en otro registry docker diferente al Docker Hub, indicaríamos lo siguiente: <https://HOST:PUERTO/REPO>

4.2.5. Subida de imagen propia a docker hub (docker push)

Vamos a subir una imagen propia al registry de Docker Hub.

La premisa para poder subir una imagen a nuestro repositorio, es que, la imagen esté etiquetada con el ID de la cuenta de acceso a docker hub.

Esto es necesario, (además de previamente haber hecho login por consola), para poder autenticarse y subir el contenido al repositorio específico.

Lo primero que vamos a realizar, es proceder con un re-etiquetado de una imagen existente, para nuestro caso, vamos a re-etiquetar la imagen **ubuntu:latest** a **<user-docker-hub>/app1:1.0.8**

Sustituímos **<user-docker-hub>** por el usuario que cada uno tenga:

```
$ docker tag ubuntu:latest <user-docker-hub>/app1:1.0.8
```

Seguidamente, realizamos la operación de subida:

```
$ docker push <user-docker-hub>/app1:1.0.8
```

```
The push refers to repository [docker.io/.../app1]
a4399aeb9a0e: Mounted from library/ubuntu
35a91a75d24b: Mounted from library/ubuntu
ad44aa179b33: Mounted from library/ubuntu
2ce3c188c38d: Mounted from library/ubuntu
1.0.8: digest: sha256:6f2fb2f9fb5582f8b587837af6ea8f37d8d1d9e41168c90f410a6ef15fa8ce5 size: 1152
```

A continuación, comprobamos en el portal web docker hub, en nuestro perfil, que efectivamente nos aparece la nueva imagen con el tag específico que hemos indicado.

Borramos la imagen local:

```
$ docker rmi <user-docker-hub>/app1:1.0.8
```

```
Untagged: .../app1:1.0.8
Untagged: .../app1@sha256:6f2fb2f9fb5582f8b587837af6ea8f37d8d1d9e41168c90f410a6ef15fa8ce5
```

Procedemos ahora a intentar descargar nuevamente la imagen del docker hub, ejecutando el siguiente comando:

```
$ docker pull <user-docker-hub>/app1:1.0.8
```

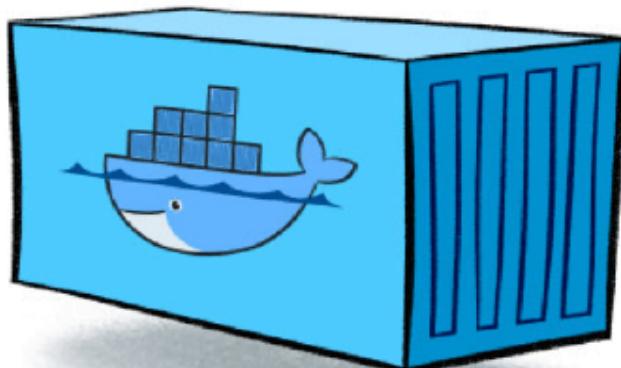
```
1.0.8: Pulling from .../app1
Digest: sha256:6f2fb2f9fb5582f8b587837af6ea8f37d8d1d9e41168c90f410a6ef15fa8ce5
Status: Downloaded newer image for .../app1:1.0.8
```

Observamos que la imagen vuelve a estar presente en nuestro docker engine local:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
.../app1	1.0.8	4e2eef94cd6b	3 weeks ago	73.9MB

Capítulo 5. Contenedores Docker



El contenedor Docker, es el resultado de haber **instanciado** una imagen Docker.

Podremos tener tantos contenedores de una misma imagen como necesitemos (y el hardware nos acompañe :D).

Cuando tenemos un contenedor, independientemente de la fase del ciclo de vida en el que se encuentre, se dice que el contenedor cumple el requisito de la inmutabilidad.

Esto quiere decir, que por ejemplo, si un contenedor ha partido de una imagen del servidor web nginx, con un intérprete propio específico así como una estructura de sistema de archivo específica, eso no se puede cambiar.

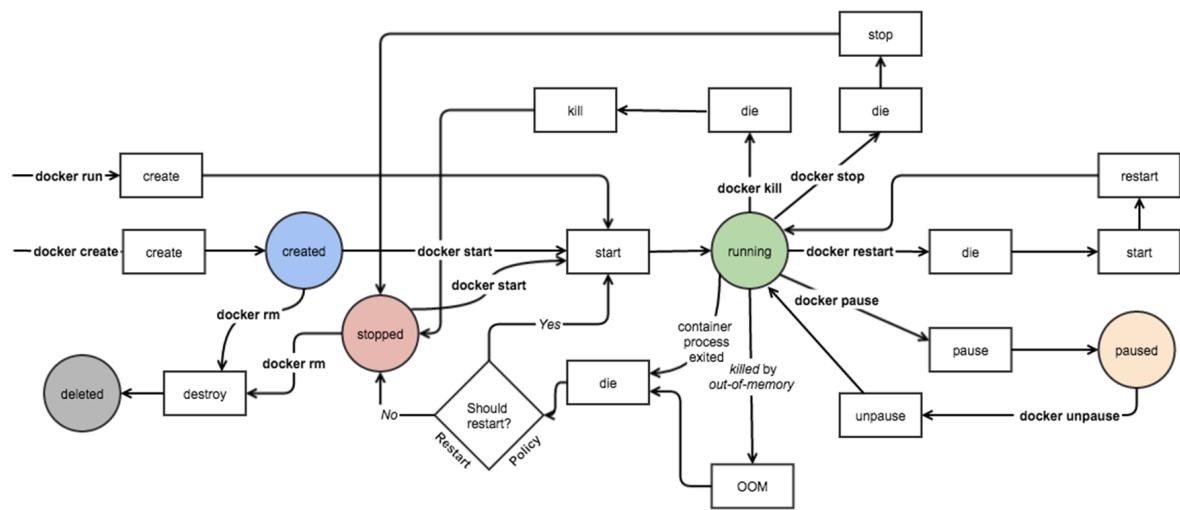
Cuando necesitemos actualizar un sistema que está en funcionamiento, para subirlo de versión por ejemplo, deberíamos de seguir estos pasos:

- Recrear la imagen docker con el nuevo contenido que queremos poner en producción
- Instalar en el servidor/servidores necesarios la imagen docker en cuestión
- Eliminar los contenedores antiguos
- Crear los nuevos contenedores de la nueva imagen
- Eliminar en caso necesario, las imágenes antiguas que ya no vamos a usar más

5.1. Ciclo de vida

Los contenedores docker poseen un ciclo de vida propio, separado del de las imágenes, de manera que podrán cambiar de una fase a otra, dependiendo de las circunstancias de operación.

Ciclo de vida de un container



5.2. Lab: Contenedores Docker

Mediante este laboratorio, vamos a practicas las operaciones del ciclo de vida de los contenedores docker.

5.2.1. Arrancar un contenedor (docker run)

Vamos a arrancar un nuevo contenedor, nos basamos en la imagen tomcat.

Como nombre de contenedor, indicaremos tomcat.

```
$ docker run --name=tomcat -d tomcat

Unable to find image 'tomcat:latest' locally
latest: Pulling from library/tomcat
57df1a1f1ad8: Pull complete
71e126169501: Pull complete
1af28a55c3f3: Pull complete
03f1c9932170: Pull complete
881ad7aafb13: Pull complete
9c0ffd4062f3: Pull complete
bd62e479351a: Pull complete
48ee8bc64dbc: Pull complete
6daad3485ea7: Pull complete
bc07a0199230: Pull complete
Digest: sha256:c2b033c9cee06d6a3eb5a4d082935bbb8afee7478e97dc6bc452bb6ab28da4b
Status: Downloaded newer image for tomcat:latest
7761b1db7077f2d7f1259fd77a49cea45a86cdb815605c70e7862f0495e450a5
```

Observamos como docker nos indica que no ha encontrado la imagen en el docker engine local, procede a la descarga del conjunto de capas que conforman la imagen y posteriormente arranca un contenedor



Con el modificador **-d**, estamos indicando que el contenedor pase a un segundo plano y libere la consola una vez ejecutamos el comando.



Si volvemos a ejecutar el comando de docker run, para obtener otro contenedor que tenga como base la misma imagen tomcat, observaremos que ahora docker tirará de caché y no procederá a llevar a cabo ningún tipo de descarga del hub.

5.2.2. Listando contenedores activos (docker container ls | docker ps)

Ahora, vamos a listar los contenedores activos que tengamos en funcionamiento.

Tenemos dos comandos principales para llevar a cabo esta acción, siendo alias el uno del otro:

Ejecutamos el comando:

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES 7761b1db7077 tomcat	tomcat	"catalina.sh run"	12 minutes ago	Up 12 minutes	8080/tcp

Ejecutamos también el siguiente comando y observamos los mismos resultados:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES 7761b1db7077 tomcat	tomcat	"catalina.sh run"	12 minutes ago	Up 12 minutes	8080/tcp

- IMAGE: Indica el nombre de la imagen original de la que el contenedor parte
- COMMAND: Indica el comando del proceso que ha ejecutado el contenedor (ID 1)
- CREATED: Indica la fecha de creación del contenedor
- STATUS: Nos indica el estado del contenedor. En caso de salida, indicará el id de salida del proceso (0 OK)
- PORTS: Que puertos han sido publicados al exterior
- NAMES: Por defecto, los contenedores poseen un nombre único que no se puede repetir para un contenedor.

5.2.3. Parar un contenedor (docker stop)

Procedemos a parar el contenedor con nombre tomcat, de forma que dejará de dar servicio.

Ejecutamos el siguiente comando:

```
$ docker stop tomcat
```

5.2.4. Listando los contenedores detenidos (docker container ls -a | docker ps -a)

Si los contenedores están detenidos, tenemos que añadir un nuevo modificador **-a** a los comandos de listar contenedores.

```
$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES 7761b1db7077 tomcat	tomcat	"catalina.sh run"	19 minutes ago	Exited (143) 2 minutes ago	

La columna STATUS indica el estado en el que se encuentra el contenedor, el estado Exited es detenido, y entre paréntesis obtendremos el código de error a bajo nivel del sistema operativo para el proceso.



En el mundo de los sistemas operativos, todo código de error distinto de 0, indicará una parada del proceso no natural.

En este caso, nos aparece el código numérico 143, esta situación es normal, ya que el proceso de tomcat es un proceso que siempre está en funcionamiento (no es un proceso por lotes), y nosotros hemos parado de forma explícita el contenedor.

5.2.5. Iniciando un contenedor detenido (docker start)

Cuando tenemos un contenedor detenido, podemos volver a ponerlo en marcha.

Ejecutamos el siguiente comando:

```
$ docker start tomcat  
tomcat
```

5.2.6. Pausando un contenedor (docker pause)

Otra operación interesante, es llevar a cabo una pausa del contenedor.

Mediante esta operación, lo que conseguimos es **congelar** el programa en sí, de manera que se interrumpirá de manera temporal el procesamiento que esté llevando a cabo.

Ejecutamos el siguiente comando:

```
$ docker pause tomcat  
tomcat
```

Si listamos ahora el estado de los contenedores, observaremos que docker nos indica que el contenedor está en una fase de operación **Up**, pero que el estado es pausado.

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
7761b1db7077	tomcat	"catalina.sh run"	27 minutes ago	Up 3 minutes (Paused)	8080/tcp
tomcat					

5.2.7. Deshaciendo el pausado de un contenedor (docker unpause)

Cuando un contenedor está pausado, podemos deshacer la operación, volviendo el contenedor a su flujo de procesamiento normal previo a la pausa.

Ejecutamos el siguiente comando:

```
$ docker unpause tomcat  
tomcat
```

Verificamos, volviendo a listar de nuevo los contenedores, que efectivamente, el contenedor vuelve a estar en operación normal.

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
7761b1db7077	tomcat	"catalina.sh run"	32 minutes ago	Up 7 minutes	8080/tcp
tomcat					

5.2.8. Reiniciando un contenedor (docker restart)

Un contenedor que se encuentre operando normalmente **Up**, podemos reiniciarlo.

Ejecutamos el siguiente comando:

```
$ docker restart tomcat  
tomcat
```

Listamos de nuevo los contenedores para verificar:

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
7761b1db7077	tomcat	"catalina.sh run"	34 minutes ago	Up 2 seconds	8080/tcp
tomcat					

Este proceso de reinicio, envía una señal al proceso del sistema operativo para que se detenga, pero no para que sea eliminado.



Si atendemos al dato del Id del contenedor, observaremos que es el mismo, tanto antes como después de reiniciar, lo que si ha cambiado es la fecha desde la cual se puso el marcha **STATUS**.

5.2.9. Detención forzada de un contenedor (docker kill)

En el caso de que por alguna razón, tengamos que detener un contenedor que no responde a un comando de parada natural (docker stop), tenemos esta opción.

Docker recomienda de forma encarecida, que paremos nuestros contenedores con el comando de parada programada **docker stop**, ya que si hacemos un kill, debemos de ser conscientes de lo que estamos haciendo, sobre todo sin son sistemas críticos que están realizando peticiones de lectura/escritura como las bases de datos, etc.

```
$ docker kill tomcat  
tomcat
```

Para verificar, volvemos a visualizar los contenedores que están detenidos:

```
$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
7761b1db7077	tomcat	"catalina.sh run"	40 minutes ago	Exited (137) 6 seconds ago	
	tomcat				

En este caso, observamos que el estado del contenedor es detenido, pero que el código numérico del sistema operativo es diferente que en el caso de una parada con docker stop.

5.2.10. Eliminando un contenedor (docker rm)

Una vez el contenedor se encuentre detenido, ejecutamos el comando de eliminación del mismo.

Para referirnos al contenedor, podemos hacerlo de dos formas:

- Por su Id
- Por su nombre

Lo mas cómodo en la mayoría de los casos será hacer referencia por nombres.

Ejecutamos el siguiente comando:

```
$ docker rm tomcat  
tomcat
```

5.2.11. Creando un contenedor sin ejecutarlo (docker create)

En lugar de llevar a cabo una ejecución directa con el comando **docker run**, tenemos la opción de crear contenedores pero que no sean ejecutados.

De esta forma, podemos instanciar una imagen docker creando su correspondiente contenedor y llevar a cabo alguna tarea que necesitemos, como comprobar su estructura, inspección, metadatos asociados, etc.

Ejecutamos el siguiente comando:

```
$ docker create tomcat  
308eeb30473a0e80b1bbe8292791c1c115c9e6473ace550a4d6e8df397ba2302
```

Si listamos ahora los contenedores, observaremos como aparece el contenedor, que no está en funcionamiento, y que como status indica **Created**.

Ejecutamos el siguiente comando:

```
$ docker container ls -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
308eeb30473a	tomecat	"catalina.sh run"	53 seconds ago	Created	
distracted_cray					

5.2.12. Auto borrado de contenedor al detenerse (directiva --rm)

Si queremos que un contenedor se auto-elimine cuando el contenedor se pare, simplemente tenemos añadir la directiva **--rm** en su sentencia de arranque.

Ejecutamos el siguiente comando:

```
$ docker run --name=tomcat -d --rm tomcat  
002f27d98ba4223191fc29685eadf2b08bc26da2ad8649d87f1049d7147a8590
```

5.2.13. Borrando un contenedor que se encuentra en operación (borrado forzado -f)

Un contenedor que se encuentra en producción en estado de operación normal (funcionamiento), no debería de ser eliminado directamente.

De hecho, si intentamos borrar un contenedor directamente que se encuentre en estado de funcionamiento, docker denegará la ejecución del comando, por que primero quiere que el propio contenedor pase a la fase de parada.

Pero podemos encontrarnos en ciertas situaciones, que queramos del tirón llevar a cabo esta eliminación sin pasar antes por la fase de parada.

Ejecutamos el siguiente comando:

```
$ docker rm -f tomcat  
tomcat
```

Capítulo 6. Dockerfile

El archivo Dockerfile es la forma recomendada, en lugar de la obtención de imágenes a partir de commits :D

El archivo Dockerfile usa un lenguaje de dominio de especificación (DSL) con instrucciones de construcción para generar las imágenes Docker.

Para construir imágenes propias a partir de un archivo Dockerfile, utilizamos la instrucción **docker build**

6.1. ¿Cómo se estructura?

El archivo Dockerfile es un archivo de texto plano sin extensión, que se estructura con una serie de instrucciones que indican las operaciones que hay que realizar para poder obtener la imagen en cuestión.

El archivo Dockerfile se procesa secuencialmente de arriba hacia abajo, y el orden en el que se encuentre una instrucción, determinará lo que se hace primero y lo que se hace después.

Es altamente recomendable que este archivo se agregue al repositorio de código fuente que esté custodiado con el control de versiones, de manera que formará parte del mismo, ya que especificará la forma en la que se construye una imagen docker para poner en producción el sistema en sí.

6.2. ¿Qué es el multi-stage build?

A partir de Docker 17.05 aparece una nueva funcionalidad llamada **multi-stage**.

Utilizando la principal característica de Docker (las capas), esta nueva funcionalidad nos va a ayudar a simplificar el proceso de construcción, ya que nos va a permitir tener en un fichero las **stages** o etapas a contemplar, para por un lado, indicar la imagen base sobre la que vamos a crear nuestro sistema, y por otro lado, poder indicar la imagen en cuestión que utilizaremos para ir a producción.

6.3. Lab: Dockerfile

Mediante este laboratorio, vamos a soltarnos con el lenguaje de definición de esquema DSL que se utiliza en la definición de un archivo Dockerfile

6.3.1. Construcción de nuestro primer Dockerfile

Vamos a crear nuestro primer archivo Dockerfile, le daremos contenido de forma que crearemos nuestra propia imagen:

```
# Version: 0.0.1
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y nginx
RUN echo '<marquee>Mira, un nuevo contenedor!</marquee>' \
    >/usr/share/nginx/html/index.html
EXPOSE 80
```

- El fichero Dockerfile contiene una serie de instrucciones con argumentos
- Las instrucciones deben escribirse en mayúsculas, seguido de sus argumentos.
- Las instrucciones se ejecutan de forma secuencial
- Cada instrucción agrega una nueva capa y luego genera la imagen.
- FROM especifica una imagen existente, donde el resto de comandos se van a ejecutar
- RUN especifica los comandos a ejecutar en un contenedor
- El comando EXPOSE permite ejecutar el contenedor con un puerto específico
- Esto no implica que el puerto esté accesible desde el contenedor, solo indica los Docker no abre los puertos automáticamente, espera a que lo hagamos en el comando run
- Sirve para ayudar a enlazar otros contenedores

6.3.2. Pasos de construcción

- Docker ejecuta el contenedor de la imagen seleccionada
- Se ejecuta una instrucción que realiza cambios en el contenedor
- Docker ejecuta el equivalente al docker commit por cada instrucción
- Docker ejecuta el nuevo contenedor de esa nueva imagen
- Se ejecuta otra nueva instrucción y vuelve a repetir el proceso hasta acabar
- Si por alguna razón, docker falla en alguno de los comandos, se puede acceder al último contenedor y explorar porqué no se realizó el proceso
- Para construir el contenedor, usamos el comando build

```
$ docker build -t <nombre_imagen>:<tag> <context_path>
```

- Se puede indicar como path una ruta de un repositorio git como github

```
$ docker build -t <image>:<tag> github.com/<path_repo>
```

- Podemos agregar un fichero .dockerignore para que no se suban al docker daemon si no son necesarios

6.3.3. Solución de problemas

- Cada vez que la imagen está construida, trata las imágenes de los pasos anteriores como caché
- Las imágenes construidas en los pasos anteriores implica que ya no se procesan
- Podemos forzar a que se realice de nuevo la construcción por medio de la opción --no-cache en caso de que no detecte
- Un truco para refrescar la caché cuando queramos es agregando variables de entorno
- Al agregar una variable de entorno, si esta es modificada, la caché es ignorada automáticamente desde la posición del cambio

```
# Version: 0.0.2
FROM ubuntu:14.04
ENV key value <-- cambios en esta linea aplica a todas las lineas inferiores
RUN apt-get update
RUN apt-get install -y nginx
RUN echo '<marquee>Mira, un nuevo contenedor!</marquee>' \
    >/usr/share/nginx/html/index.html
EXPOSE 80
```

6.3.4. Histórico

- Podemos observar el histórico de la creación de la imagen

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
30d96d3de199	4 hours ago	/bin/sh -c #(nop) EXPOSE 80/tcp	0 B	
c06aa252ac7	4 hours ago	/bin/sh -c echo '<marquee>Mira, un nuevo cont	46 B	
2c7607683901	4 hours ago	/bin/sh -c apt-get install -y nginx	18.15 MB	
1442b9191433	4 hours ago	/bin/sh -c apt-get update	22.16 MB	
8503fa4e4d43	4 hours ago	/bin/sh -c #(nop) ENV UPDATE_AT=2016-09-01	0 B	
0485842b3a0e	4 hours ago	/bin/sh -c #(nop) MAINTAINER Ruben Gomez "rg	0 B	
4a725d3b3b1c	9 days ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0 B	
<missing>	9 days ago	/bin/sh -c mkdir -p /run/systemd && echo 'doc	7 B	
<missing>	9 days ago	/bin/sh -c sed -i 's/^#\s*\\$(deb.*universe\)\$/	1.895 kB	
<missing>	9 days ago	/bin/sh -c rm -rf /var/lib/apt/lists/*	0 B	
<missing>	9 days ago	/bin/sh -c set -xe && echo '#!/bin/sh' > /u	194.6 kB	
<missing>	9 days ago	/bin/sh -c #(nop) ADD file:ada91758a31d8de3c7	187.8 MB	

6.3.5. Instrucciones Dockerfile

- Existen una serie de instrucciones preparadas para construir un Dockerfile y dar mayor flexibilidad
- Algunos de ellos son: CMD, ENTRYPOINT, ADD, COPY, VOLUME, WORKDIR, USER, ONBUILD, ENV

6.3.6. CMD

- Permite ejecutar comandos tras la ejecución del contenedor.
- Similar a la instrucción RUN, pero este no se ejecuta en construcción
- Una equivalencia a docker run -it <imagen> /bin/sh

```
CMD ["/bin/sh"]
```

- Podemos concatenar argumentos

```
CMD ["/bin/bash", "-l"]
```

- De esta manera, al terminar de ejecutar el contenedor ya no es necesario indicarlo

```
docker run -it kane_project/testing
```

- Si indicamos el comando a ejecutar, el comando CMD se sobreescribe y no se ejecuta

```
$ docker run -it kane_project/testing /bin/ps
  PID TTY      TIME CMD
    1 ?        00:00:00 ps
$
```

6.3.7. ENTRYPOINT

- Provee de un comando que no se puede sobreescribir como CMD
- De hecho, cualquier argumento especificado en Docker Run se hará sobre el ENTRYPOINT

```
ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
CMD ["-h"]
```

- El comando ejecutará sobre el entrypoint
- Podemos modificar el entrypoint, pero explicitandolo con --entrypoint

```
$ docker run -it --entrypoint /bin/sh kane_project/testing
```

6.3.8. WORKDIR

- Provee una forma de establecer un directorio de trabajo
- Permite establecer comandos a ejecutar a partir del workdir

```
WORKDIR /usr
```

- Permite realizar distintas operaciones en distintos directorios durante la construcción
- Permite usar un directorio final de uso en el contenedor

```
docker run kane_project/testing ls -la
total 28
drwxr-xr-x 10 root root 97 Dec 15 17:44 .
drwxr-xr-x 21 root root 4096 Jan 13 12:41 ..
drwxr-xr-x 2 root root 8192 Jan 13 12:36 bin
drwxr-xr-x 2 root root 6 Apr 10 2014 games
drwxr-xr-x 2 root root 26 Dec 14 01:10 include
drwxr-xr-x 26 root root 4096 Dec 14 01:10 lib
drwxr-xr-x 10 root root 105 Dec 14 01:10 local
drwxr-xr-x 2 root root 4096 Jan 13 12:37 sbin
drwxr-xr-x 61 root root 4096 Jan 13 12:37 share
drwxr-xr-x 2 root root 6 Apr 10 2014 src
```

- Se puede cambiar el workdir por medio de la opción -w

```
docker run -w /root kane_project/testing ls -la
total 12
drwx----- 2 root root 35 Dec 14 01:11 .
drwxr-xr-x 21 root root 4096 Jan 13 12:41 ..
-rw-r--r-- 1 root root 3106 Feb 20 2014 .bashrc
-rw-r--r-- 1 root root 140 Feb 20 2014 .profile
```

6.3.9. ENV

- Se utilizan para establecer variables de entorno en el contenedor

```
ENV USR_HOME /root
```

- La variable de entorno se usará en el siguiente comando
- Las variables de entorno serán persistentes en el contenedor

```
[vagrant@localhost mi_primer_dockerfile]$ docker run -it kane_project/testing bash  
root@3c9629f61a2b:/# echo $USR_HOME  
/root
```

- Con la opción -e se pueden definir variables de entorno

```
$ docker run -it -e USR_HOME=/Otro kane_project/testing bash  
root@ec2fb2d5b5bb:/# echo $USR_HOME  
/Otro
```

6.3.10. USER

- Especifica el usuario que debe ejecutar los comandos

```
RUN useradd nginx  
USER nginx
```

- Todas las siguientes líneas se ejecutan con el usuario definido.

```
[vagrant@localhost mi_primer_dockerfile]$ docker run -it -e USR_HOME=/Otro kane_project/testing bash  
nginx@08bd9cd0a85b:/# whoami  
nginx
```

6.3.11. VOLUME

- Agrega volúmenes a un contenedor creado de una imagen.
- Permite definir un directorio dentro de uno o más contenedores para datos compartidos persistentes
 - Pueden ser compartidos y reutilizados entre contenedores
 - Un contenedor no tiene porqué estar ejecutándose para compartir sus volúmenes
 - Los cambios en el volumen son inmediatos
 - Los volúmenes persisten hasta que ninguna imagen los utilice

```
VOLUME ["/data"]
```

- Es la misma instrucción que

```
docker run -v <path_host>:<path_contenedor>
```

- Para observar donde se encuentra el punto de montaje

SHELL1

```
docker run -it --name volume-data -v /mnt kane_project/testing
nginx@9a6321b474f8:$
```

SHELL2

```
$ docker inspect -f "{{json .Mounts}}" volume-data
[
{
  "Name": "91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266",
  "Source": "/var/lib/docker/volumes/91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266/_data",
  "Destination": "/mnt",
  "Driver": "local",
  "Mode": "",
  "RW": true,
  "Propagation": ""
}
]
$ docker volume inspect 91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266
[
{
  "Name": "91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266",
  "Driver": "local",
  "Mountpoint": "/var/lib/docker/volumes/91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266/_data",
  "Labels": null,
  "Scope": "local"
}
]
```

- Podemos acceder a la unidad montada y escribir datos en ella, que se reflejarán en el volumen del contenedor

SHELL2

```
sudo touch /var/lib/docker/volumes/91def767f74c736a0fe2449ca4f4702413352f20e8fdc46eff4f9ec14234b266/_data/prueba.txt
```

- Automáticamente aparece en el nuevo contenedor:

SHELL1

```
nginx@9a6321b474f8:$ ls /mnt
prueba.txt
```

- Otra opción más común es la de definir el punto de montaje host y guest

```
$ docker run -it -v $PWD:/mnt kane_project/testing
nginx@e9751f87d080:$ ls /mnt
Dockerfile
```

6.3.12. ADD

- Agrega ficheros y directorios de nuestro entorno de construcción a la imagen
- Sirve para instalar aplicaciones
- Especifica un origen y un destino

```
ADD software.lic /opt/application/software.lic
ADD http://wordpress.org/latest.zip /root/wordpress.zip
ADD latest.tar.gz /var/www/wordpress/
```



Copia fichero Descarga Descomprime. El truco está en la barra del final.

6.3.13. COPY

- Igual que el add pero solo copia, no descomprime
- El directorio debe ser un path completo
- El UID y GID son 0
- Si el destino no está creado, es similar a mkdir -p

6.3.14. ONBUILD

- Agrega disparadores de las imágenes
- Se ejecuta cuando la imagen es base de otra imagen
- Permite ejecutar un script dependiente del entorno donde se ejecute

```
ONBUILD ADD . /miapp/src
ONBUILD RUN cd /miapp/src && make
```

- Por ejemplo, creamos una imagen base

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y apache2
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ONBUILD ADD . /var/www/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2"]
CMD ["-DFOREGROUND"]
```

Ejecutamos la siguiente instrucción para cambiar el entrypoint y poder trabajar con la imagen:

```
$ docker run -it --entrypoint=/bin/bash kaneproject/webapp -s
```

6.3.15. Creando imagen multi-stage

En este ejercicio, nuestra idea es generar al vuelo un archivo **index.html** sobre un intérprete que tiene presente el servidor web nginx.

Posteriormente, queremos que el archivo que se ha generado al vuelo (**index.html**) del primer stage FROM, se copie a la construcción final de la imagen docker del segundo stage con una orden COPY.

Creamos un archivo Dockerfile con el siguiente contenido:

```
FROM nginx:latest as builder
RUN echo 'Hello from first multi-stage' > /usr/share/nginx/html/index.html

FROM httpd:latest
COPY --from=builder /usr/share/nginx/html/index.html /usr/local/apache2/htdocs/
```

- Observamos que con la directiva **--from=builder**, estamos haciendo referencia a la ruta de un archivo de un stage anterior con el alias **builder**



También podríamos indicar en lugar de una capa anterior, una imagen directamente que se encuentre en el propio Dockerhub, indicando **--from=nginx:latest**

Seguidamente, llevamos a cabo la construcción de la imagen:

```
$ docker build -t httpd-multistage .
```

Por último, arrancamos un contenedor para comprobar que observamos el contenido del archivo **index.html** que creó el primer stage:

```
$ docker run -d -p 8081:80 httpd-multistage
```

Realizamos una petición de curl:

```
$ curl localhost:8081
Hello from first multi-stage
```

Capítulo 7. Contenedores Avanzados

Además del propio ciclo de vida de los contenedores, que es necesario conocer bien para poder detectar fallas en producción y llevar a cabo una administración de una forma profesional, también resulta interesante conocer otro conjunto de operaciones más especiales que nos permitan llevar a cabo ciertas acciones que pudiéramos necesitar en nuestro día a día con docker.

Acciones del tipo:

- Obtención de traza de un contenedor en streaming
- Conectarse dentro del propio contenedor para ver qué está pasando en su interior (Modo Origen :D)
- Adjuntar otra consola al proceso del contenedor principal para propósitos de monitoreo
- Visualizar los procesos internos del contenedor
- Inspeccionar la estructura del container de forma externa
- Etc.

7.1. Lab: Contenedores Avanzados

Mediante este laboratorio, practicaremos operaciones más avanzadas en la gestión de los contenedores.

7.1.1. Exportación del sistema de archivos de un contenedor (docker export)

Mediante esta operación, vamos a exportar el sistema de archivo de un contenedor.

Suponemos que tenemos un contenedor con nombre **tomcat** funcionando.

Ejecutamos el siguiente comando:

```
$ docker export tomcat > tomcat_export.tar
```

7.1.2. Importación del sistema de archivos de un contenedor (docker import)

Este comando, importa el contenido de un sistema de archivos para crear una imagen.

Podemos indicar como parámetro incluso una URL en lugar de un archivo local, en nuestro caso, vamos a hacer referencia al archivo local previamente exportado:

Ejecutamos el siguiente comando:

```
$ docker import tomcat_export.tar
```

Si seguidamente, ejecutamos un comando para listar las imágenes, observaremos que tendremos una referencia a una imagen con las columnas de repository y tag a <none>.

Esto nos valdría en el caso de que si ahora, quisiéramos arrancar un contenedor pero que se basara en esa imagen específica, indicaríamos a un comando docker run, dicho Id

7.1.3. Ejecución de comandos contra un contenedor (docker exec)

En ocasiones nos va a interesar ejecutar comandos contra un contenedor para que este nos devuelva un resultado.

En otras ocasiones, incluso, se nos va a dar la necesidad, de conectarnos internamente al contenedor para visualizar el sistema de archivos interno para detectar anomalías.

Vamos a ejecutar un contenedor basado en **tomcat** y nos conectamos dentro, nuestro objetivo será, poner al día los repositorios e instalarle una herramienta que no está en el sistema, la herramienta **htop**.

En primer lugar, arrancamos un contenedor basado en la imagen tomcat:

Ejecutamos el siguiente comando:

```
$ docker run --name=tomcat -d tomcat  
bc7f16c5b9d50cb8638af958a78a0eb4a3b6e6c0826a728e42d8f42f9f0f8017
```

Una vez estando el contenedor en ejecución, vamos a proceder a conectarnos dentro:

```
$ docker exec -it tomcat bash
```

El argumento `-it` permite habilitar una terminal interactiva de entrada - salida

El último argumento, será el nombre del binario que se encuentre dentro del contenedor que queremos ejecutar, en nuestro caso, `bash`, abrirá un proceso de consola para conectarnos dentro.

Una vez conectados dentro, vamos a observar como el prompt del sistema cambia, y en la shell se comuta a un usuario root:

```
root@bc7f16c5b9d5:/usr/local/tomcat#
```

Desde este momento, todo comando que ejecutemos, lo estaremos realizando a nivel del sandbox del contenedor docker, no en el anfitrión directamente.

Procedemos a actualizar los repositorios:

```
root@bc7f16c5b9d5:/# apt-get update
```

Seguidamente, procedemos con la instalación de la herramienta `htop`:

```
root@bc7f16c5b9d5:/# apt-get install htop -y
```

Una vez realizada la operación de instalación, salimos del contenedor:

```
root@bc7f16c5b9d5:/# exit
```

7.1.4. Cambios en el sistema de un contenedor que está en funcionamiento (docker commit)

Tenemos un comando que es un tanto peculiar, y siempre debemos de ser conscientes de lo que estamos haciendo :D

Dado un contenedor que tengamos, podemos crear una imagen en caliente a partir de mismo, sin necesidad de haber llevado a cabo una operación de construcción de imagen (docker build) ni de disponer de un archivo de manifiesto asociado (Dockerfile)

El peligro de realizar esta operación, es que no vamos a dejar constancia en ningún sitio de las operaciones que hemos realizado... Por lo cual es un método bastante poco aconsejado -

Uno de los pocos motivos para proceder con esta operación, podría ser que en producción tienes un sistema funcionando, detectas una falla que ves que es posible corregir rápidamente por que estamos en modo (se me quema la casa), la corriges rápidamente modificando el propio contenedor y le comentas a desarrollo lo que has hecho, para que en una versión posterior, lo incorporen al manifiesto Dockerfile correspondiente.

Vamos a generar una imagen nueva a partir de los cambios que hemos realizado en el contenedor con nombre **tomcat**.

De forma genérica, podemos indicar que el comando sería el siguiente:

```
$ docker commit <container_id> <new_image_name_to_generate>
```

Ejecutamos el siguiente comando para llevar a cabo la operación:

```
$ docker commit -m="My Modified Tomcat Image" --author="Pako" tomcat my_new_tomcat_image:1.0.6  
d3c888cfea99ffdade0008176c0b757427751fab7fedee59f693a8a7b8df4b41
```

- Docker commit solo guarda las diferencias entre la última imagen de contenedor y el estado actual.
- Las actualizaciones son muy ligeras, ya que solo almacenan los cambios entre la imagen de ejecución del contenedor y la nueva imagen generada.
- También está permitido agregar más información a la imagen
- Como el autor, una descripción y un tag a la imagen

7.1.5. Inspección de contenedores (docker inspect)

Si por alguna causa, queremos comprobar la estructura de armado del contenedor, tenemos a nuestra disposición un comando bastante útil para llevar a cabo inspecciones.

El comando no sólo vale para los contenedores, puede aplicar a cualquier entidad docker, como redes, volúmenes, imágenes, etc.

Vamos a inspeccionar y a comentar lo que observamos cuando inspeccionamos la imagen que acabamos de crear de forma personalizada.

Ejecutamos el siguiente comando:

```
$ docker inspect my_new_tomcat_image:1.0.6

[
  {
    "Id": "sha256:d3c888cfea99ffdade0008176c0b757427751fab7fedee59f693a8a7b8df4b41",
    "RepoTags": [
      "my_new_tomcat_image:1.0.6"
    ],
    "RepoDigests": [],
    "Parent": "sha256:5f47aad0b70e1d6a8324543d3b7536f23baaa09b09661dbf4e52d7b822c0157",
    "Comment": "My Modified Tomcat Image",
    ...
    "Author": "Pako",
  }
]
```

Entre otros elementos, observamos el comentario que hemos indicado así como el autor, estos datos al hacer un commit forman parte de la metadata de la propia imagen.

7.1.6. Aislamiento de procesos del contenedor

En referencia al aislamiento de procesos, vamos a comprobar la cantidad de procesos que observamos dentro de un contenedor.

Un contenedor, realmente es un proceso aislado del propio kernel, de forma que internamente si ejecutamos un comando tipo **ps** únicamente deberíamos de observar el propio lanzamiento del comando, así como el proceso que mantenga al contenedor activo como hilo principal, ya sea un servidor web, una consola shell, etc.

En primer lugar, iniciamos una ejecución bash para conectarnos al contenedor **tomcat**:

```
$ docker exec -it tomcat bash
root@bc7f16c5b9d5:/usr/local/tomcat#
```

Una vez conectados dentro, vamos a listar los procesos del sistema:

```
root@bc7f16c5b9d5:/# ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        1  0.4  1.1 5175388 91092 ?      Ssl 14:47   0:15 /usr/local/openjdk-11/bin/ja
root      329  0.1  0.0  5752  3580 pts/0     Ss 15:44   0:00 bash
root      334  0.0  0.0  9392  3040 pts/0     R+ 15:45   0:00 ps -aux
```

 Un contenedor posee como proceso principal (PID 1) el proceso de ejecución del contenedor, si este cae, el contenedor muere.

Se trata del mismo comportamiento que cualquier sistema linux, si el proceso 1 (initd/systemd) muere, el sistema operativo muere.

- Si el proceso de tomcat tuviera algún problema y se cayera a nivel interno, el propio contenedor

se pararía de forma directa

Por último, salimos del contenedor para volver a la shell del sistema operativo:

```
root@bc7f16c5b9d5:/# exit  
exit
```

7.1.7. Adjuntándonos al proceso (docker attach)

El comando docker exec permite llevar a cabo la ejecución de un comando concreto sobre un contenedor, de manera adicional en otro proceso.

Pero en ciertas ocasiones, quizás nos interese conectarnos a ese mismo contenedor con el mismo proceso.

Mediante el comando attach, lo que vamos a hacer con el contenedor **tomcat** que tenemos en ejecución, es adquirir a primer plano el proceso.

Vamos a observar que la consola se queda como pillada, esto es normal, por que realmente, ahora es como si el proceso tomcat lo hubiéramos levantando con la propia consola y no en segundo plano.

Ejecutamos en una terminal el siguiente comando:

```
$ docker attach tomcat
```

Ejecutamos en una segunda terminal de nuevo el comando:

```
$ docker attach tomcat
```

El efecto que ahora tenemos, es que si desde cualquiera de ella, finalizamos el proceso tomcat, pulsando **Ctrl + Z**, el proceso finalizará en ambas terminales, ya que se trata del mismo proceso.

7.1.8. Visualización de procesos (docker top)

Otro comando bastante interesante que aporta docker y que nos permite directamente obtener en consola el conjunto de procesos interno que está ejecutando un contenedor, sin meternos dentro (hacer un top desde fuera), es el comando **docker top**.

Eliminamos primero el contenedor con nombre tomcat del sistema (recordamos que no puede haber 2 conteenedores cuyos nombres sean iguales en un mismo motor docker engine):

```
$ docker rm tomcat  
tomcat
```

Arrancamos de nuevo un contenedor basado en tomcat:

```
$ docker run -d --name=tomcat tomcat
```

Seguidamente, ejecutamos el comando de visualización de procesos contra el contenedor que está en ejecución:

```
$ docker top tomcat
```

UID	PID	PPID	C	STIME	TTY
TIME	CMD				
root	3019	3002	17	16:05	?
00:00:08	/usr/local/openjdk-11/bin/java -Djava.util.logging.config.file=/usr/local/tomcat/conf/logging.properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -Djdk.tls.ephemeralDHKeySize=2048 -Djava.protocol.handler.pkgs=org.apache.catalina.webresources -Dorg.apache.catalina.security.SecurityListener.UMASK=0027 -Dignore.endorsed.dirs= -classpath /usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-juli.jar -Dcatalina.base=/usr/local/tomcat -Dcatalina.home=/usr/local/tomcat -Djava.io.tmpdir=/usr/local/tomcat/temp org.apache.catalina.startup.Bootstrap start				

Observamos en este caso, a nivel de proceso del sistema operativo, el proceso en cuestión que está ejecutando el contenedor, con todos los argumentos que han sido proporcionados para su arranque.

7.1.9. Obtención de traza del contenedor (docker logs)

Vamos a crear un contenedor como servicio:

```
$ docker run --name test_daemon -d ubuntu /bin/sh -c "while true; do echo hola mundo;  
sleep 1; done"  
fc90b1bd1312ac3c81cef3e0b603fe40c469dec86a8878579793480a6a731b33  
[docker@docker-master.local ~]$
```

- Comprobamos que el contenedor no nos genera prompt y no vemos la salida del comando en ejecución.
- Vamos a mostrar la información de log del contenedor.
- Para ello lanzamos el comando:

```
$ docker logs -f test_daemon  
hola mundo  
hola mundo  
hola mundo
```

- Pulsamos CTRL+C y volvemos a ejecutar el comando para ver la información de la fecha incluida.

```
$ docker logs -ft test_daemon
2016-09-01T09:23:51.700058936Z hola mundo
2016-09-01T09:23:52.707106869Z hola mundo
2016-09-01T09:23:53.708836849Z hola mundo
2016-09-01T09:23:54.715426920Z hola mundo
```

- Comprobamos los procesos reales del contenedor.

```
$ docker top test_daemon
PID          USER          TIME          COMMAND
3098        root          0:00          /bin/sh -c while true; do echo hola mundo; sleep 1; done
4007        root          0:00          sleep 1
```

- Filtramos la información de inspección para localizar la dirección ip del contenedor.

```
$ docker inspect --format='{{ .NetworkSettings.IPAddress}}' test_daemon
172.17.0.3
```

- Indicamos el estado de contenedor y la ip en una plantilla GO.

```
docker inspect --format='Running: {{ .State.Running }} Network:{{ .NetworkSettings.IPAddress}}' test_daemon
Running: true Network:172.17.0.3
```

- Paramos el contenedor con el comando docker kill

```
$ docker kill test_daemon
```

7.1.10. Destrucción del laboratorio

- Para destruir todos los contenedores, lanzamos el siguiente comando:

```
$ docker rm $(docker ps -a -q) -f
```

Capítulo 8. Volúmenes y puntos de montaje

Los volúmenes de Docker permiten obtener una persistencia después de haber destruido un contenedor.

Existen tres tipos de volúmenes y/o puntos de montaje:

- bind o puntos de montaje
- volume o volumen controlado por Docker
- tmpfs p volumen temporal

8.1. ¿Cómo son definidos a nivel de sistema?

En el ecosistema docker, los volúmenes se comportan como carpetas a nivel del propio sistema operativo.

A nivel interno de los contenedores, consiste en que aparece una carpeta en el sistema de archivo.

Los volúmenes permiten que la aplicación interna que está funcionando dentro del contenedor, pueda guardar los archivos necesarios en una carpeta concreta, sin que la aplicación sea consciente de que los datos realmente se están guardando en una carpeta del anfitrión.

La gestión la lleva a cabo completamente el motor de docker.

8.2. ¿Dónde guarda docker toda la gestión/administración de los volúmenes que el controla?

En la ruta **/var/lib/docker**, docker guarda todos los datos referentes a la gestión de todas las entidades que el gestiona, entre otros, los volúmenes.

Para acceder a dicha ruta, necesitamos acceder como root por consola, ya que es un directorio protegido sólo para usuarios de altos privilegios como es el caso del root.

8.3. Lab: Volúmenes

Mediante este laboratorio, vamos a llevar a cabo algunas de las operaciones más comunes en la gestión de los volúmenes.

8.3.1. Montando un volumen en un contenedor

Vamos a montar un volumen directamente en el contenedor que se va a poner en marcha:

```
docker run -v /mnt tomcat:latest
```

- Al destruir el contenedor, ese volumen no se borrará y aparecerá como un hash en el listado de volúmenes locales disponibles

```
$ docker volume ls
DRIVER      VOLUME NAME
local      84fd14d44259f7bb4befb5268c76f6c68af8b8fbe10a5af0b5c83f6eaef9a7ce
local      dca0c7c730822de136c8f4cdeeaeb8aca3e319e9135e3fa4a8d4381249990f6
local      git-global
local      maven-data
```

- Para borrar un volumen podemos usar el comando docker volume rm

```
$ docker volume rm 84fd14d44259f7bb4befb5268c76f6c68af8b8fbe10a5af0b5c83f6eaef9a7ce
84fd14d44259f7bb4befb5268c76f6c68af8b8fbe10a5af0b5c83f6eaef9a7ce
```

- También podemos definir un alias para el volumen
 - Volúmenes con alias asociado para poder usar el punto de montaje de forma sencilla

```
$ docker volume ls
DRIVER      VOLUME NAME
local      dca0c7c730822de136c8f4cdeeaeb8aca3e319e9135e3fa4a8d4381249990f6
local      git-global
local      maven-data
local      mi-alias
```

- El volumen está disponible y podemos ver información relacionada con el mismo

```
$ docker volume inspect mi-alias
[
  {
    "CreatedAt": "2019-03-06T15:23:13Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/mi-alias/_data",
    "Name": "mi-alias",
    "Options": null,
    "Scope": "local"
  }
]
```

- Si miramos en la información del contenedor, veremos que aparece el alias como información adicional del mismo

```
$ docker inspect d6 | less
"HostConfig": {
  "Binds": [
    "mi-alias:/mnt"
  ],
  ...
}
"Mounts": [
  {
    "Type": "volume",
    "Name": "mi-alias",
    "Source": "/var/lib/docker/volumes/mi-alias/_data",
    "Destination": "/mnt",
    "Driver": "local",
    "Mode": "z",
    "RW": true,
    "Propagation": ""
  }
],
```

8.3.2. Volumen bind

- Son aquellos volúmenes donde definimos ruta del host con ruta del invitado.
- Para ello indicamos la ruta host en vez del alias
- La ruta en el host debe existir

```
$ docker run -v /home/alumno/Desktop:/mnt tomcat:latest
```

- De esta forma, el contenido de Desktop de host aparecerá en el contenedor en la ruta /tmp

8.3.3. Volumen tmpfs

- Permite crear un volumen con datos en entorno temporal
- Cuando el contenedor se pare, este volumen se destruye.
- Se mantiene en la memoria del host fuera de la capa de escritura del contenedor.

```
$ docker run --tmpfs /mnt tomcat:latest
```

Capítulo 9. Networking

Al instalar Docker, se crean tres redes automáticamente

```
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
7798eb42ea63    bridge    bridge      local
b205310f4be6    host      host       local
b15351d82e1b    none     null       local
```

- Son las implementaciones por defecto.
- Al definir `--network=<red>`, estamos indicando que red queremos implementar en el contenedor.
- El Daemon docker crea una interfaz virtual llamada docker0
- Permite reenviar paquetes entre otras interfaces de red
- Por defecto docker conecta las interfaces de red a esta red virtual

Existe un cuarto tipo de red (overlay) que en principio no aparece con Docker, ya que esta red únicamente se activa cuando el nodo de docker está participando en un clúster, en modo docker swarm.

La red overlay es una red distribuída entre varios nodos.

9.1. Lab: Networking

Mediante este laboratorio, llevaremos a cabo una gestión de las operaciones más comunes de las redes docker.

9.1.1. Creando un contenedor en la red Bridge

Vamos a ejecutar un contenedor de la imagen nginx y de forma explícita, vamos a circunscribirlo a la red **bridge**

```
$ docker run -d -P --name=nginx --network=bridge nginx:1.9.1  
17271e1c7e78b923dfb6a448fccb91328bcecccd25c0a8990202b6d72bf21c045
```

Listamos los contenedores:

```
$ docker ps  
CONTAINER ID        IMAGE               COMMAND      CREATED          STATUS              PORTS  
STATUS            PORTS  
17271e1c7e78        nginx:1.9.1        nginx -g      19 seconds ago  
Up 18 seconds      0.0.0.0:49153->443/tcp,  
                      0.0.0.0:49154->80/tcp    nginx
```

- Con la opción **-P**, docker analiza todas las sentencias EXPOSE que contenga la imagen, de forma que realiza el binding automático de todos los puertos que requiera el contenedor.
- Si indicamos **-p 80:80**, lo que estamos haciendo es llevando a cabo un binding explícito, de forma que el puerto interno 80 del contenedor, lo estamos bindeando con el socket del puerto 80 del anfitrión (si el puerto estuviera ocupado, el proceso de arranque daría un error)
- Si indicamos **-p 127.0.0.1:8080:8080**, estamos bindeando un puerto del anfitrión específico (8080) con una interfaz de red específica del anfitrión (127.0.0.1), de forma que los paquetes de red únicamente podrían salir por esa interfaz y no por ninguna otra, como pudiera ser el caso de una interfaz con una IP de la red local del servidor.

Vamos a conectarnos dentro del contenedor para visualizar algunos datos relacionados con la red:

```
$ docker exec -it nginx bash
```

Al iniciar el contenedor veremos un prompt. Usamos hostname para comprobar que el id del contenedor aparece en el hostname

```
root@17271e1c7e78:/# hostname  
17271e1c7e78
```

Comprobamos el /etc/hosts que posee la línea de ip e identificador del contenedor:

```
root@17271e1c7e78:/# 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
172.17.0.3 17271e1c7e78
```

- Vamos a comprobar ahora la dirección ip del contenedor.
- Para saber la IP, podemos usar las net-tools de ubuntu, para lo cual, primero vamos a tener que actualizar el sistema de archivo con un apt-get update, ya que estamos en el mundo debian:

```
root@17271e1c7e78:/# apt-get update
```

Instalamos las net-tools:

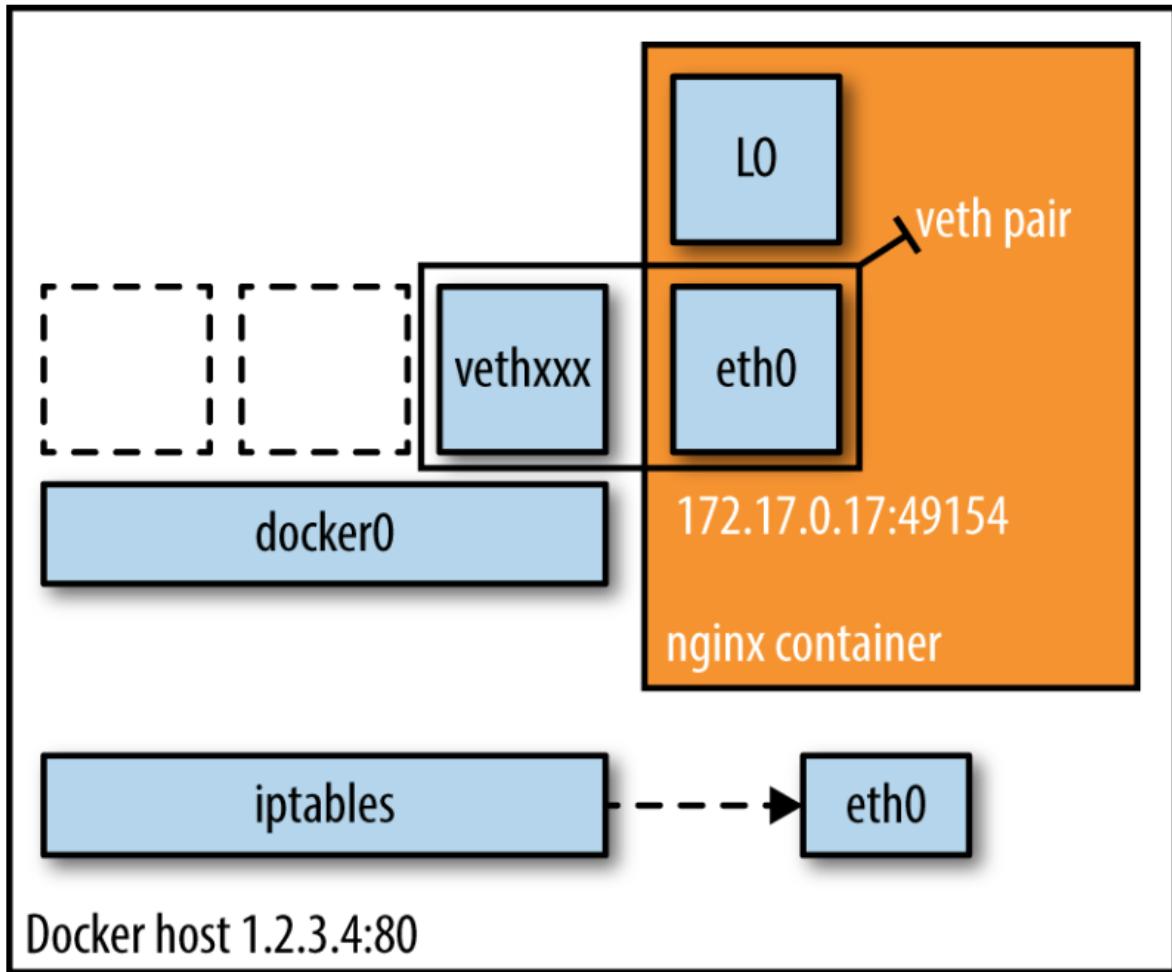
```
root@17271e1c7e78:/# apt-get install net-tools -y
```

Ahora comprobamos las direcciones ip del contenedor:

```
root@17271e1c7e78:/# ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:03
          inet addr:172.17.0.3 Bcast:172.17.255.255 Mask:255.255.0.0
...
lo      Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
```

9.1.2. Bridge

- El modo bridge es la interfaz por defecto
- La red *Bridge* representa la interfaz docker0
- Por defecto, todos los contenedores se conectan a esta red en particular.
- Sin definir la opción -P o -p los paquetes no pueden salir del host



- Podemos usar el comando `docker network inspect` para gestionar las redes de Docker

```
$ docker network inspect bridge
...
    "Name": "bridge",
    "Id": "32c64aca12eba4569b4dc9a23eff2e7d65999d21b87ec1309ea73fff16716b56",
    "Scope": "local",
    "Driver": "bridge",
    "Subnet": "172.17.0.0/16",
    "Gateway": "172.17.0.1"
```

- posee información de los contenedores que están adscritos a él

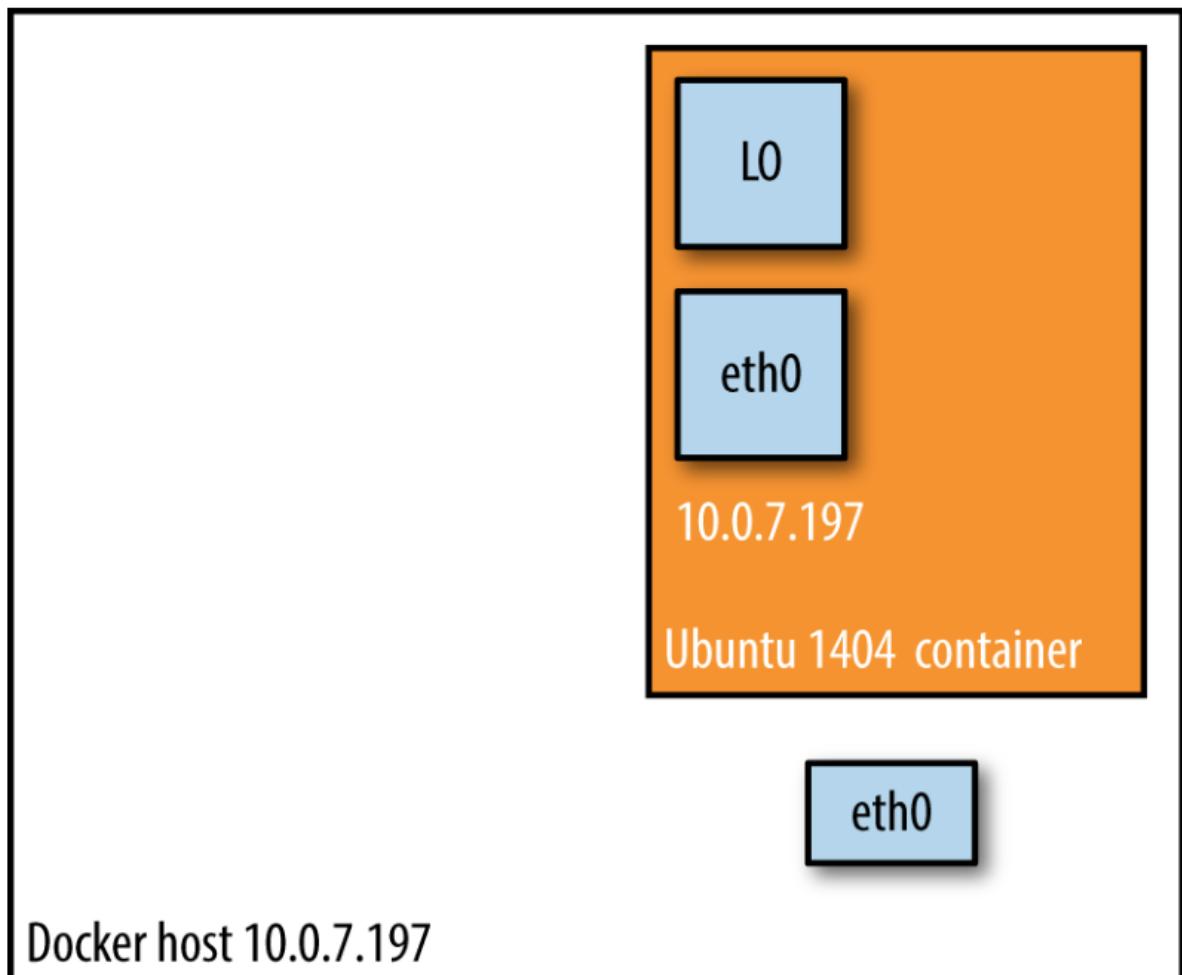
9.1.3. Host

- Es un modo que deshabilita el aislamiento de red en un contenedor
- Comparte el espacio del host y se expone directamente en la red pública

```
$ docker run -d --net=host ubuntu:14.04 tail -f /dev/null
```

- Las dos máquinas comparten la dirección ip

- Este modo es más rápido que el modo bridge



9.1.4. Container (Legacy)

- Permite usar el mismo espacio de trabajo de un contenedor en otro
- Permite crear varios contenedores con la misma ip

```
$ docker run -d --name nginx -P --net=bridge nginx:1.9.1
$ docker exec -it nginx ip addr
$ docker run -it --net=container:nginx ubuntu:14.04 ip addr
```

9.1.5. None

- Permite crear un contenedor en su propia red privada
- No posee una interfaz de red real.
- Permite usarlo de plantilla para otras redes personalizadas
- Util para redes

```
$ docker run -d -P --net=none nginx:1.9.1
```

9.1.6. Personalizada (Definida por el usuario)

- Podemos crear varios tipos de redes y configurarlas
- Usamos las creadas para clonar la red

```
$ docker network create --driver bridge mi_red  
$ docker network inspect mi_red
```

- Comprobamos que la creación de la red es igual a la red anterior
- El antiguo enlazado o linking en redes internas no está soportado

9.1.7. Overlay

- En caso de poseer un cluster de docker (docker swarm) podemos crear una red de tipo overlay

```
# docker network create --driver overlay --subnet 10.0.9.0/24 red_multihost  
# docker service create --replicas 2 --network red-multihost --name web nginx
```

- No se permite la creación y uso de redes con docker run, debe lanzarse con docker service
- Se pueden crear nuevas redes de tipo bridge, overlay o MACVLAN.
- Incluso podemos implementar nuestros propios drivers.

Capítulo 10. Docker Registry (Registros privados)

Cuando queremos almacenar imágenes Docker, ya sean imágenes propias que hemos construído, o bien, imágenes de terceros sin alterar, necesitamos un almacén binario.

Dicho almacén binario en el mundo docker se le conoce como un registry.

Tenemos la opción de utilizar el registry público por excelencia (Docker Hub), pero si queremos que las imágenes sean privadas, no nos queda más remedio que pasar por caja.

Como alternativa, podríamos tener en nuestros propios servidores nuestro propio registry.

Existen un par de alternativas al propio Docker Hub que son las que más se suelen utilizar:

Por un lado, tenemos un registry que no requiere autentificación y que se monta bastante rápido, para propósitos del equipo de desarrollo y pruebas en lo referente a la custodia de imágenes compiladas, viene genial.

Por otro lado, tenemos un registry bastante popular en el mundo del desarrollo, ya que es bastante versátil y se utiliza como repositorio binario para un amplio abanico de tecnologías, se trata del Sonatype Nexus.

10.1. Lab: Docker Registry (Registros privados)

Mediante este laboratorio, vamos a interactuar con los registry privados, llevando a cabo la instalación, puesta en marcha y gestión de los mismos.

10.1.1. Registry Privado (registry)

Se trata de un servidor que almacena y distribuye imágenes de Docker Es un producto OpenSource bajo licenciamiento apache:

http://en.wikipedia.org/wiki/Apache_License

Permite control en el almacenamiento de las imágenes

Existen distintas distribuciones de registros de Docker.

Para la versión de contenedor de Docker, poseemos la versión gratuita y la versión de registro oficial con soporte.

Para la versión de contenedor, debemos usar una versión de servidor Docker 1.6.x Las versiones anteriores trabajan con registros antiguos de python.

10.1.2. Poniendo en marcha el registry

Creamos el registry utilizando la imagen oficial docker:

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

Este repositorio posee un sistema de autenticación nulo, con cualquier usuario podemos conectarnos y subir imágenes.

Para subir una imagen de contenedor, hay que marcarla con el nombre del servidor y el nombre del grupo/imagen:tag que queramos subir.

Vamos a re-etiquetar una imagen docker llamada **alpine:latest** a **localhost:5000/contenedores/alpine:v1**, de forma que quedará fijado en la metadata del nombre de la imagen, la dirección IP y el PUERTO al cual se debe de subir (push) y descargar (pull) la imagen docker.

```
$ docker tag alpine:latest localhost:5000/contenedores/alpine:v1
```

Para subir la imagen construida, con subir la imagen tageada con la ruta del servidor es suficiente:

```
$ docker push localhost:5000/contenedores/alpine:v1
The push refers to a repository [localhost:5000/contenedores/alpine]
e154057080f4: Mounted from alpine
v1: digest: sha256:c0537ff6a5218ef531ece93d4984efc99bbf3f7497c0a7726c88e2bb7584dc96 size: 528
```

10.1.3. Persistencia en disco del registro

Un punto importante con la utilización de un registry es la persistencia del propio registry.

Nuestro registry se está montando sobre un contenedor, y por funcionamiento, un contenedor es efímero y no persiste los datos que dentro del mismo se estén generando.

Para conseguir el efecto de que las propias imágenes que subimos a nuestro registry quedan almacenadas en disco aún cuando el contenedor es destruido, necesitamos llevar a cabo el montaje de un volumen:

```
$ docker run -d -p 5000:5000 --restart=always --name registry \
-v `pwd`/data:/var/lib/registry \
registry:2
```

Subimos la imagen al nuevo registro y luego lo destruimos:

```
$ docker tag alpine:latest localhost:5000/contenedores/alpine:v1
$ docker push localhost:5000/contenedores/alpine:v1
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
cd6c3a9987e7        registry:2        "/entrypoint.sh /e..."   About an hour ago   Up About an hour   0.0.0.0:5000->5000/tcp   registry
[vagrant@localhost ~]$ docker rm -f cd6c3a9987e7
cd6c3a9987e7
```

A continuación, borramos la imagen local, creamos el registro y nos la descargamos del mismo:

```
$ docker rmi localhost:5000/contenedores/alpine:v1
Untagged: localhost:5000/contenedores/alpine:v1
Untagged: localhost:5000/contenedores/alpine@sha256:c0537ff6a5218ef531ece93d4984efc99bbf3f7497c0a7726c88e2bb7584dc96

$ docker run -d -p 5000:5000 --restart=always --name registry \
> -v `pwd`/data:/var/lib/registry \
> registry:2
36259e2a67906e29aedff591dac5567235a27e52aba09e08a913be84a2dcff22

$ docker pull localhost:5000/contenedores/alpine:v1
v1: Pulling from contenedores/alpine
Digest: sha256:c0537ff6a5218ef531ece93d4984efc99bbf3f7497c0a7726c88e2bb7584dc96
Status: Downloaded newer image for localhost:5000/contenedores/alpine:v1
```

De esta forma comprobamos que el contenido de las imágenes es almacenado de forma externa

10.1.4. Registry Privado (Sonatype Nexus)

Otra opción es un gestor mas general de artefactos como **Sonatype Nexus**

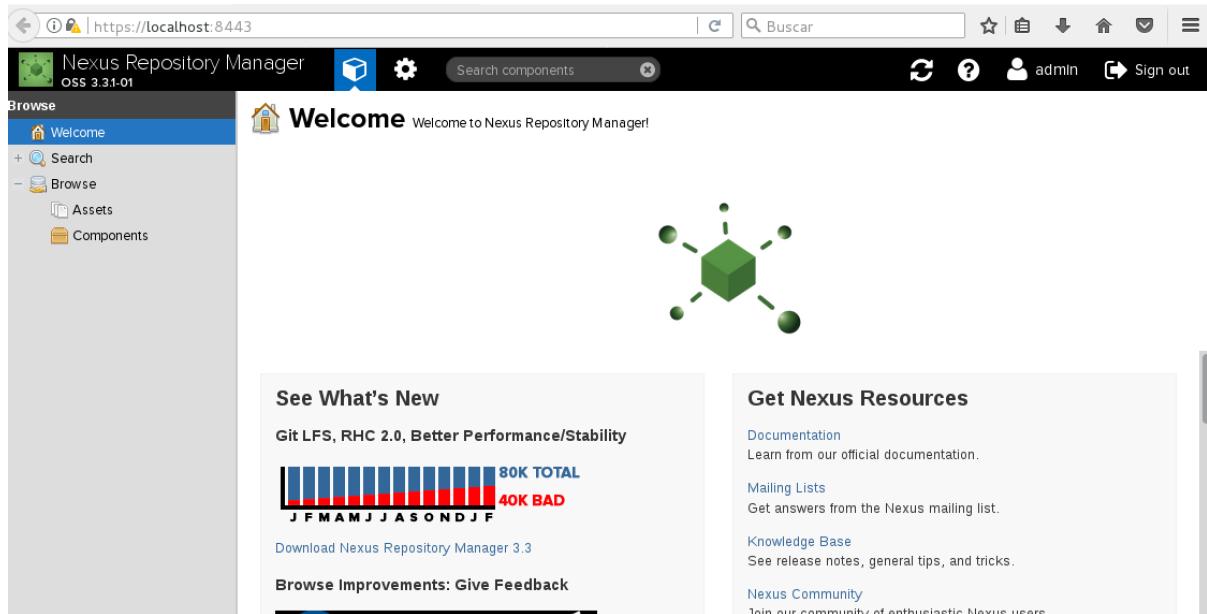
Si usamos usuario y contraseña, es necesario poseer ssl para que el cliente de Docker permita el uso de contraseñas.

El uso es similar al del repositorio.

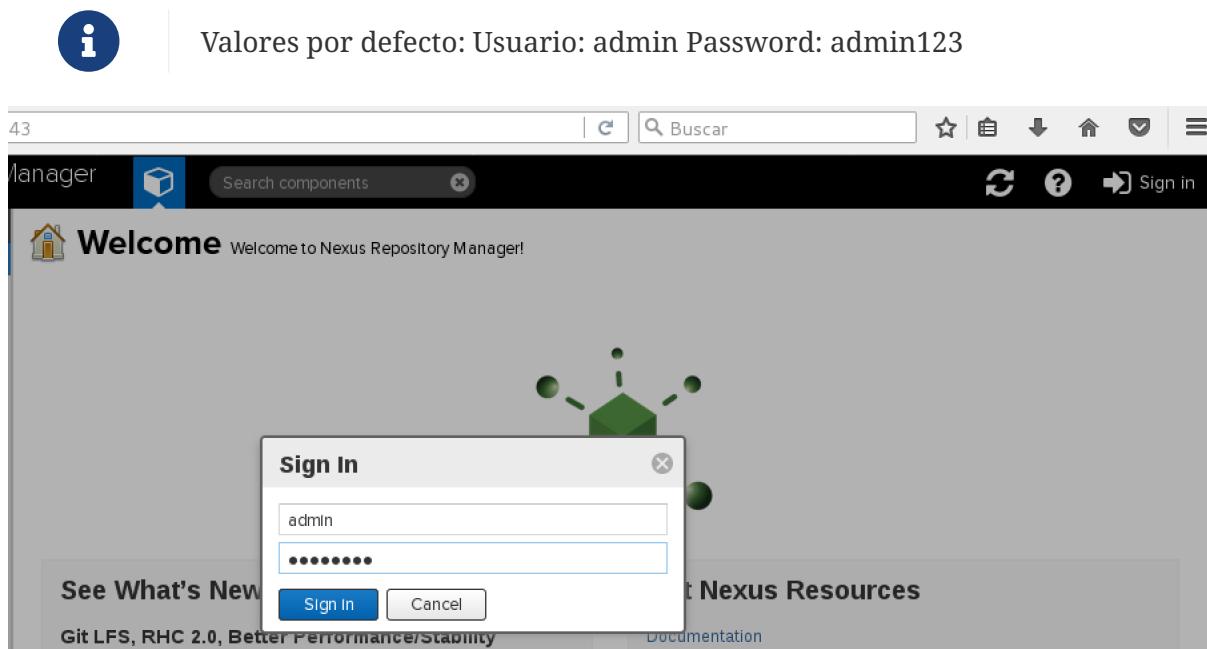
En este caso usaremos una imagen generada con ssl preconfigurado para activar el registro de Docker con ssl.

```
$ docker run -d -p 8081:8081 -p 8443:8443 -p 8444:8444 -v ~/nexus-data:/nexus-data -v ~/nexus-ssl:/opt/sonatype/nexus/etc/ssl --name nexus bradbeck/nexus-https:3.6.0 e3adfc643bae0d542904e7d0faa0c32c1d7910d812132f617c024749f4ff0650
```

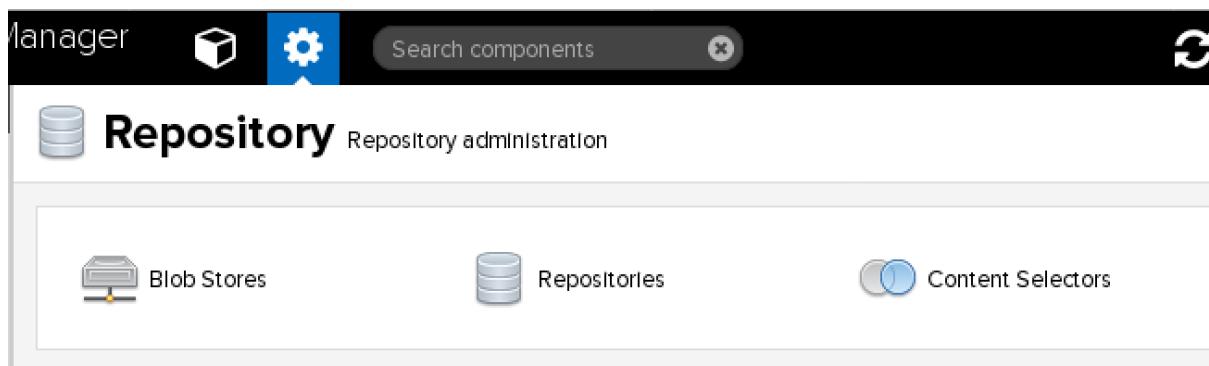
Una vez Nexus quede inicializado abrimos un navegador y accedemos a <https://localhost:8081>



Nos registramos para crear un nuevo registro de Docker:



Accedemos al menú de creación de repositorios y creamos un nuevo recurso de tipo **Docker Host**:



Seleccionamos la creación de repositorios:

This screenshot shows the "Repositories" management page in Nexus. On the left, a sidebar menu under "Administration" has "Repositories" selected, indicated by a blue background. The main area displays a table of existing repositories. One row is highlighted with a blue background, showing "nuget-group" as the name, "group" as the type, "nuget" as the format, and "Online" as the status. A prominent blue "Create repository" button is located above the table.

Seleccionamos Docker[host] y rellenamos la opción de nombre y conector https:



No es posible crear un conector http inseguro para usarlo con docker sin configurar el cliente para que acepte conexiones inseguras

This screenshot shows the "Create repository" form. The "Name" field is filled with "releases". The "Online" checkbox is checked. The "Repository Connectors" section contains two parts: "HTTP" (with a note about using it behind a proxy and a dropdown menu) and "HTTPS" (with a note about using it for https and a dropdown menu set to "8444"). The "Docker Registry API Support" section includes a checkbox for enabling the V1 API.

Bajamos en la página y creamos el repositorio:

Storage

Blob store:

Blob store used to store asset contents

default

Strict Content Type Validation:

Validate that all content uploaded to this repository is of a MIME type appropriate for the repository format

Hosted

Deployment policy:

Controls if deployments of and updates to artifacts are allowed

Allow redeploy

[Create repository](#)

[Cancel](#)

Ahora en el listado de repositorios ya debería aparecer el repositorio creado:



Para validarnos podemos usar el usuario admin y pass admin123 o crear nuestro usuario.

Usando por defecto, realizamos el login en el repositorio

```
$ docker login https://localhost:8444/releases
Username: admin
Password:
Login Succeeded
```

Creamos una imagen como el tag usado anteriormente:

```
$ docker tag alpine:latest localhost:8444/server-alpine:v1
$ docker push localhost:8444/server-alpine:v1

The push refers to a repository [localhost:8444/server-alpine]
e154057080f4: Pushed
v1: digest: sha256:c0537ff6a5218ef531ece93d4984efc99bbf3f7497c0a7726c88e2bb7584dc96 size: 528
```

Podemos ver el recurso ahora en el repositorio:

Group ↑	Name	Version
	server-alpine	v1

Capítulo 11. Integración continua

Podemos utilizar docker en el mundo de desarrollo enfocado a la integración continua.

Para ello, podemos ceder el trabajo de construcción y pruebas a Jenkins.

Lo mejor para pruebas es compilar nuestra propia imagen, realizar las pruebas y destruir el contenedor una vez ya no lo necesitemos.

11.1. Lab: Integración continua

Mediante este laboratorio, crearemos nuestra propia imagen docker de Jenkins para su utilización con tareas de integración continua.

También realizaremos algunas pruebas de funcionamiento con la plataforma.

11.1.1. Creación del Dockerfile

Creamos el archivo de definición para construir la imagen:

```
FROM ubuntu:16.04
MAINTAINER james@example.com
ENV REFRESHED_AT 2014-06-01
RUN apt-get update -qq && apt-get install -qq curl
RUN apt-get install -qqy apt-transport-https ca-certificates
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
58118E89F3A912897C070ADBF76221572C52609D
RUN echo deb https://apt.dockerproject.org/repo ubuntu-xenial main >
/etc/apt/sources.list.d/docker.list
RUN apt-get update -qq
RUN apt-get install -y iptables lxc openjdk-8-jdk git-core docker.io
ENV JENKINS_HOME /opt/jenkins/data
ENV JENKINS_MIRROR http://mirrors.jenkins-ci.org
RUN mkdir -p $JENKINS_HOME/plugins
RUN curl -sf -o /opt/jenkins/jenkins.war -L $JENKINS_MIRROR/war-
stable/latest/jenkins.war
RUN for plugin in chucknorris greenballs scm-api git-client git ws-cleanup ;\
do curl -sf -o $JENKINS_HOME/plugins/${plugin}.hpi \
-L $JENKINS_MIRROR/plugins/${plugin}/latest/${plugin}.hpi ; done
ADD ./dockerjenkins.sh /usr/local/bin/dockerjenkins.sh
RUN chmod +x /usr/local/bin/dockerjenkins.sh
VOLUME /var/lib/docker
EXPOSE 8080
ENTRYPOINT [ "/usr/local/bin/dockerjenkins.sh" ]
```

- Usamos como base Ubuntu 18.04
- Actualizamos e instalamos las configuraciones básicas
- Instalamos el certificado de docker
- Agregamos a la lista de repositorios de docker
- Instalamos los paquetes de Docker io y java
- Descargamos jenkins
- Publicamos el punto de montaje y el puerto
- Generamos un fichero que permita lanzar Jenkins y el proceso Docker añadido.
- Es el fichero del entrypoint **dockerjenkins.sh**

```

#!/bin/bash

# First, make sure that cgroups are mounted correctly.
CGROUP=/sys/fs/cgroup

[ -d $CGROUP ] ||
mkdir $CGROUP

mountpoint -q $CGROUP ||
mount -n -t tmpfs -o uid=0,gid=0,mode=0755 cgroup $CGROUP || {
    echo "Could not make a tmpfs mount. Did you use -privileged?"
    exit 1
}

# Mount the cgroup hierarchies exactly as they are in the parent system.
for SUBSYS in $(cut -d: -f2 /proc/1/cgroup)
do
    [ -d $CGROUP/$SUBSYS ] || mkdir $CGROUP/$SUBSYS
    mountpoint -q $CGROUP/$SUBSYS ||
        mount -n -t cgroup -o $SUBSYS cgroup $CGROUP/$SUBSYS
done

# Now, close extraneous file descriptors.
pushd /proc/self/fd
for FD in *
do
    case "$FD" in
        # Keep stdin/stdout/stderr
        [012])
            ;;
        # Nuke everything else
        *)
            eval exec "$FD>&="
            ;;
    esac
done
popd

dockerd &
exec java -jar /opt/jenkins/jenkins.war

```

11.1.2. Creación la imagen

El directorio debe tener el siguiente aspecto:

```

/
└── Dockerfile
└── dockerjenkins.sh

```

- Dentro, creamos la imagen y levantamos el contenedor

```
$ docker build -t cursodocker/jenkins .
$ docker run -p 8080:8080 --name jenkins --privileged -d cursodocker/jenkins
```

- La opción --privileged es un peligro, ya que permite ejecutar el contenedor como si fuera el propio host
- Permite ejecutar el comando docker dentro de un contenedor docker
- Se puede considerar como un acceso root.
- Accedemos al log de Docker para poder configurar jenkins

```
*****
*****
*****
Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*****
*****
*****
```

11.1.3. Generación de la tarea

- Creamos una tarea de tipo libre

Enter an item name

Tarea nueva Docker

» Required field

Crear un proyecto de estilo libre



Esta es la característica principal de Jenkins, la de ejecutar el proyecto combinando cualquier tipo de repositorio de software (SCM) con cualquier modo de construcción o ejecución (make, ant, mvn, rake, script ...). Por tanto se podrá tanto compilar y empaquetar software, como ejecutar cualquier proceso que requiera monitorización.

- Definimos un espacio de trabajo distinto

Configurar el almacenamiento cuando haya un proyecto para ejecutarlo

Bloquear la ejecución cuando un proyecto relacionado está en ejecución

Utilizar un directorio de trabajo personalizado

Directorio /tmp/jenkins-buildenv/\${JOB_NAME}/workspace

Nombre a mostrar

¿Conservar los 'logs' de dependencias de las ejecuciones?

- Usaremos un repositorio de un proyecto con pruebas incluidas

<https://github.com/jamtur01/docker-jenkins-sample.git>

Configurar el origen del código fuente

Ninguno
 Git

Repositories

Repository URL: <https://github.com/jamtur01/docker-jenkins-sample.git>

Credentials: - none -

Avanzado...
Add Repository

Branches to build

Navegador del repositorio: (Auto)

- Creamos una nueva tarea de shell

Ejecutar

Añadir un nuevo paso ▾

Ejecutar Ant

Ejecutar linea de comandos (shell) **•**

Ejecutar tareas 'maven' de nivel superior

- El contenido es el siguiente

```
IMAGE=$(docker build . | tail -1 | awk '{ print $NF }')
MNT="$WORKSPACE/.."
CONTAINER=$(docker run -d -v "$MNT:/opt/project" $IMAGE /bin/bash -c 'cd
/opt/project/workspace && rake spec')
docker attach $CONTAINER
RC=$(docker wait $CONTAINER)
docker rm $CONTAINER
exit $RC
```

- En este paso se construye la imagen de docker
- Se ejecuta el contenedor
- se accede al contenedor y se observa el resultado

- Se van generando los resultados de las pruebas
- Se borra el contenedor y se sale
- Luego se define una post-tarea

Activate Chuck Norris

Agregar los resultados de los tests de los proyectos padre

Almacenar firma de ficheros para poder hacer seguimiento

Ejecutar otros proyectos

Guardar los archivos generados

Publicar los resultadood de tests JUnit

Git Publisher

Editable Email Notification

Notificación por correo

Set build status on GitHub commit [deprecated]

Set status for GitHub commit [universal]

Delete workspace when build is done

Añadir una acción ▾

- Agregamos la dirección de los informes que se generan
 - spec/reports/*.xml

Acciones para ejecutar después.

Publicar los resultadood de tests JUnit

Ficheros XML con los informes de tests `spec/reports/*.xml`

El atributo '@includes' de la etiqueta 'fileset' especifica dónde estarán los ficheros XML generados, por ejemplo: 'myproject/target/test-reports'

- Lanzamos el proyecto

 Volver al Panel de Control

 Estado Actual

 Cambios

 Zona de Trabajo

 Construir ahora

 Borrar Proyecto

 Configurar

 Move

Proyecto Tarea nueva Docker



- Ahora vemos que el resultado es correcto

 Historia de tareas Tendencia 

X

 #2	06-sep-2016 23:26
 #1	06-sep-2016 23:24

 [RSS Para Todos](#)  [RSS para los errores](#)

Capítulo 12. Monitorización

- Los comandos más accesibles de monitorización son
 - docker stats
 - docker events

12.1. Docker stats

- Estadísticas del contenedor
- Agregado en la versión 1.5.0
- Indica la información del consumo del contenedor

CONTAINER PIDS	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O
d6ba16b145c8 0	0.27%	48.44 MiB / 489 MiB	9.90%	1.206 kB / 648 B	25.13 MB / 0 B
0b296962ba3e 0	0.10%	12.57 MiB / 489 MiB	2.57%	1.296 kB / 648 B	12.59 MB / 0 B
a19c6981e0ce 0	0.08%	3.828 MiB / 489 MiB	0.78%	7.983 kB / 648 B	12.53 MB / 0 B

- Indica la memoria utilizada y la máxima
- Porcentaje de memoria utilizada
- Tráfico de red
- Consumo de CPU

12.2. Docker events

- El demonio de Docker genera un flujo de eventos en el ciclo de vida del contenedor

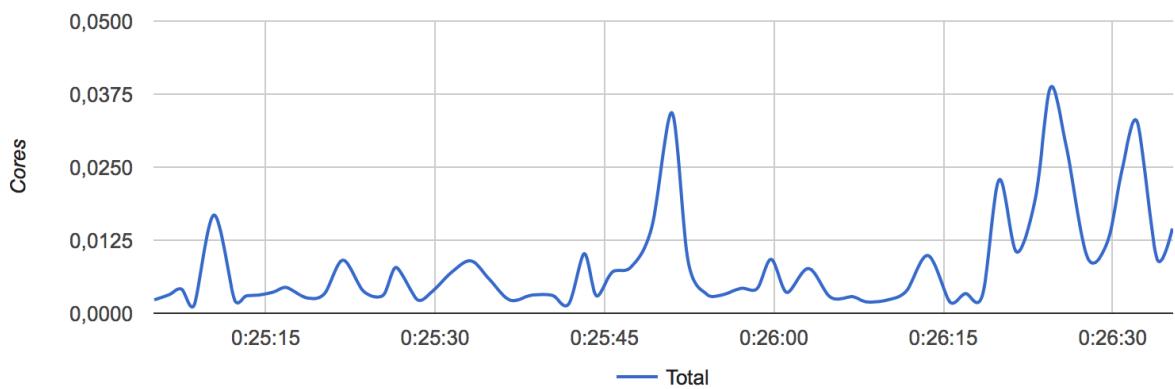
```
MacBook-Pro-de-Ruben:test_network kane_project$ docker events
2016-09-01T00:21:56.590145490+02:00 container create 48a8990c4d7c18130b12d54a840375779da72c577684d628b6cd347ba79c3815
(image=ubuntu:14.04, name=tiny_mestorf)
2016-09-01T00:21:56.594358020+02:00 container attach 48a8990c4d7c18130b12d54a840375779da72c577684d628b6cd347ba79c3815
(image=ubuntu:14.04, name=tiny_mestorf)
```

12.3. CAdvisor

- Permite mostrar la información de docker a nivel web
- Presenta los datos por medio de gráficas

```
docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker/:/var/lib/docker:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

- Si visitamos la web:



- Publica una interfaz rest con la información de Docker

```
curl http://172.17.42.10:8080/api/v1.3/containers/
```

Capítulo 13. Seguridad

Existen cuatro áreas definidas para seguridad en Docker

- Seguridad del kernel y soporte para espacios de trabajo y cgroups
- El daemon de Docker
- Perfiles de configuración de contenedores
- Las características de seguridad del kernel y su forma de interactuar con los contenedores

13.1. Namespaces del Kernel

- Al iniciar un contenedor por medio de `docker run`, Docker crea un conjunto de namespaces y grupos de control (cgroups) para el contenedor.
- Los **namespaces** permiten un enorme aislamiento. Los procesos que se ejecutan dentro de un contenedor no se ven.
- Cada contenedor obtiene su **pila de red independiente**. El contenedor no puede obtener acceso privilegiado a los sockets o interfaces de otro contenedor.
- Esta tecnología lleva disponible desde 2008, con las versiones de kernel 2.6.15 en adelante, por lo que se consideran suficientemente probados.
- Los namespaces son reimplementaciones de las características de OpenVZ, sobre 2005.

13.2. CGroups

- Permite definir recursos y limitarlos por cuentas.
- Provee de métricas y garantiza que cada contenedor tiene un uso razonable de memoria, CPU, I/O de disco, e impide que el uso exhaustivo de alguno de estos recursos pueda tirar el sistema.
- No trata de evitar el acceso de un contenedor a otro, sino que evita los ataques DDOS de los contenedores.
- Muy útil como PaaS, garantizando un tiempo de inicio y un rendimiento consistente.
- El código se inició en 2006, y fue incluido en el kernel 2.6.24

13.3. Ataques al daemon Docker

- Ejecutar contenedores y aplicaciones con Docker. Este daemon requiere de privilegios de `root`, luego se deben tener en cuenta lo siguiente:
 - Solo usuarios autorizados deben acceder al daemon de Docker.
 - Docker permite compartir directorios entre el host y el contenedor, y el contenedor accede con todos los permisos a la unidad compartida.
 - Puede iniciar un contenedor que vea el directorio raíz / y alterarlo a voluntad.
 - Si se instrumentaliza Docker por medio de un servidor web, podemos abrir la puerta a que ese daemon cree contenedores maliciosos en nuestro sistema.

- El API REST usa en las nuevas versiones (a partir de la 0.5.2) un socket UNIX, en vez de TCP al 127.0.0.1
- Podemos definir permisos de acceso al socket para evitar que se pueda acceder y controlarlo desde ahí.
- Podemos exponer el API REST de forma explícita, pero conociendo las consecuencias.
- Es recomendable exponerlo solo a redes autorizadas o por VPN, o protegida por https, e incluso por stunnel con certificados ssl
- El daemon es vulnerable a la carga de imágenes *docker load* o de la red *docker pull*. Pero desde la versión 1.10.0, las imágenes son accedidas y almacenadas por medio de un checksum criptográfico, lo que impide que un atacante cause una colisión con una imagen existente.
- Si se ejecuta docker en un servidor, es recomendable que solo se ejecute docker en el mismo. Con él se pueden instalar otros procesos de monitorización y de auditoría.

13.4. Otras características de seguridad

- Podemos usar otras características de seguridad avanzadas, como TOMOYO, AppArmor, SELinux, GRSEC, etc.
- Docker no interfiere con otros sistemas, solo activa capacidades a los contenedores.
 - Puedes iniciar un kernel con GRSecurity (<https://grsecurity.net>) y PaX (<https://es.wikipedia.org/wiki/PaX>). Permite realizar comprobaciones de seguridad a nivel de kernel en compilación y en ejecución. Evita exploits, y no afecta a Docker, ya que es independiente de los contenedores
 - Si el servidor viene con plantillas de seguridad para los contenedores de Docker, como en AppArmor o en RHEL con SELinux, proveen de seguridad extra.

13.5. Lab: Seguridad

Mediante este laboratorio, vamos a llevar a cabo la gestión de la protección del socket del docker engine, para que nadie que no esté autorizado pueda conectarse a el.

13.5.1. Proteger el socket del daemon Docker

- Por defecto, docker ejecuta un unix socket, pero se puede modificar para que sea http
- Para que sea accesible desde la web, debemos usar certificados firmados por una CA
- para crear un CA y clientes con OpenSSL, generamos las claves públicas y privadas:

```
$ openssl genrsa -aes256 -out ca-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
```

```
$ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
\-----
Country Name (2 letter code) [XX]:ES
State or Province Name (full name) []:Valladolid
Locality Name (eg, city) [Default City]:Valladolid
Organization Name (eg, company) [Default Company Ltd]:Curso Docker
Organizational Unit Name (eg, section) []:Formacion
Common Name (eg, your name or your server's hostname) []:localhost.localdomain
Email Address []:rubengomez78@gmail.com
```

- Ya poseemos el CA, asi que podemos crear la clave de servidor y el CSR (Certificate Signing Request)

```
$ openssl genrsa -out server-key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
```

```
$ openssl req -subj "/CN=localhost.localdomain" -sha256 -new -key server-key.pem -out server.csr
```

- Ahora firmamos las claves públicas con el CA
- Podemos crear certificados que solo permitan el acceso a determinadas máquinas por medio de direcciones ip

```
$ echo subjectAltName = DNS:localhost.localdomain,IP:10.0.2.15,IP:127.0.0.1,IP:192.168.122.1 > extfile.cnf
```

```
$ openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem
-extfile extfile.cnf
Signature ok
subject=/CN=localhost.localdomain
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

- Para la autenticación de clientes, creamos una clave de cliente y CSR

```
$ openssl genrsa -out key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
```

```
$ openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

- Ahora firmamos la clave pública

```
$ openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem -extfile
extfile.cnf
Signature ok
subject=/CN=client
Getting CA Private Key
Enter pass phrase for ca-key.pem:
```

- Protegemos las claves para que no tenga permisos de escritura

```
$ chmod -v 0400 ca-key.pem key.pem server-key.pem
el modo de «ca-key.pem» cambia de 0664 (rw-rw-r--) a 0400 (r-----)
el modo de «key.pem» cambia de 0664 (rw-rw-r--) a 0400 (r-----)
el modo de «server-key.pem» cambia de 0664 (rw-rw-r--) a 0400 (r-----)
```

- Ya podemos eliminar los CSR por seguridad

```
$ rm -v client.csr server.csr
«client.csr» borrado
«server.csr» borrado
```

- Ahora podemos hacer que el daemon de Docker solo acepte conexiones de clientes que posean un certificado permitido por el CA
- Iniciamos el daemon de docker con el certificado

```
$ sudo dockerd --tlsverify --tlscacert=ca.pem --tlscert=server-cert.pem --tlskey=server-key.pem -H=0.0.0.0:2376
INFO[0000] libcontainerd: new containerd process, pid: 20010
WARN[0000] containerd: low RLIMIT_NOFILE changing to max current=1024 max=4096
WARN[0001] devmapper: Usage of loopback devices is strongly discouraged for production use. Please use '--storage-opt dm.thinpooldev' or use 'man docker' to refer to dm.thinpooldev section.
WARN[0001] devmapper: Base device already exists and has filesystem xfs on it. User specified filesystem will be ignored.
INFO[0001] [graphdriver] using prior storage driver "devicemapper"
INFO[0001] Graph migration to content-addressability took 0.00 seconds
WARN[0001] mountpoint for pids not found
INFO[0001] Loading containers: start.
.....INFO[0001] Firewalld running: false
INFO[0002] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon option --bip can be used to set a preferred IP address

INFO[0002] Loading containers: done.
INFO[0002] Daemon has completed initialization
INFO[0002] Docker daemon commit=78d1802 graphdriver=devicemapper version=1.12.6
INFO[0002] API listen on [::]:2376
```

- Intentamos conectar via http sin permisos

```
$ docker version
Client:
Version: 1.12.6
API version: 1.24
Go version: go1.6.4
Git commit: 78d1802
Built: Tue Jan 10 20:20:01 2017
OS/Arch: linux/amd64
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

- Podemos observar que el servidor no ofrece comunicación con el cliente. Hagámoslo ahora con el certificado

```
$ docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --tlskey=key.pem
-H=localhost.localdomain:2376 version
Client:
Version: 1.12.6
API version: 1.24
Go version: go1.6.4
Git commit: 78d1802
Built: Tue Jan 10 20:20:01 2017
OS/Arch: linux/amd64

Server:
Version: 1.12.6
API version: 1.24
Go version: go1.6.4
Git commit: 78d1802
Built: Tue Jan 10 20:20:01 2017
OS/Arch: linux/amd64
```

- Si queremos securizar el cliente por defecto:
- Creamos la carpeta .docker en el directorio \${HOME} del usuario

```
$ mkdir -pv .docker
mkdir: se ha creado el directorio «.docker»
```

- Copiamos los certificados ca.pem cert.pem y key.pem al directorio .docker/

```
$ cp -v {ca,cert,key}.pem .docker/
«ca.pem» -> «.docker/ca.pem»
«cert.pem» -> «.docker/cert.pem»
«key.pem» -> «.docker/key.pem»
```

- Almacenamos la variable de exportación en la sesión de usuario (.bash_profile)

```
$ export DOCKER_HOST=tcp://10.0.2.15:2376 DOCKER_TLS_VERIFY=1
```

- Ya podemos comprobar como no necesitamos definir las entradas de los certificados.

```
$ docker version
Client:
Version:      1.12.6
API version:  1.24
Go version:   go1.6.4
Git commit:   78d1802
Built:        Tue Jan 10 20:20:01 2017
OS/Arch:      linux/amd64
```

```
Server:
Version:      1.12.6
API version:  1.24
Go version:   go1.6.4
Git commit:   78d1802
Built:        Tue Jan 10 20:20:01 2017
OS/Arch:      linux/amd64
```

- Otra opción de comunicación por medio de curl

```
$ curl https://10.0.2.15:2376/images/json \
> --cert .docker/cert.pem \
> --key .docker/key.pem \
> --cacert .docker/ca.pem
```

Capítulo 14. Docker compose

- Se trata de una herramienta disponible en las docker tools. (Incluido en instaladores Windows y Mac OSX)
- En los repositorios oficiales no está disponible, hay que descargarlo por separado.
- Permite definir aplicaciones docker multi-contenedor
- Por medio de un comando, permite crear e iniciar servicios de la configuración
- Perfecto para desarrollo y pruebas, u orquestación de servicios
- Docker compose puede ejecutarse con docker swarm y en las últimas versiones incluse se puede ejecutar bajo Kubernetes (beta)
- Existen traductores de ficheros docker-compose a kubernetes

14.1. Configuración

- El fichero docker-compose se basa en la siguiente tabla de compatibilidad.
- Cuanta mayor sea la versión, mayores las capacidades de configuración.

Compose file format	Docker Engine release
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+
1.0	1.9.1.+

- El fichero docker-compose.{yml,yaml} es un fichero YAML que define servicios, redes y volúmenes. Por defecto se busca en el directorio actual
- Una definición de servicio contiene la configuración que debe ser aplicada a cada contenedor. Muy similar al comando docker run y docker volume create
- Las instrucciones por defecto dentro del Dockerfile son respetados por docker compose
- La versión actual soportada es la versión 3.

14.2. Directivas

- Esta es la lista de directivas del fichero docker-compose.yml

14.2.1. build

Opciones de configuración en tiempo de construcción

Ejemplo de búsqueda de Dockerfile en subdirectorio

```
build: ./frontend
```

Ejemplo de búsqueda en subdirectorio y cambio de fichero

```
build:  
  context: ./backend  
  dockerfile: Dockerfile-development  
  args:  
    dbname: dev-bbdd
```

- Podemos definir la construcción de una imagen con tag

```
build: ./frontend  
image: app-angular:0.0.1-SNAPSHOT
```



En docker swarm no se permite la creación de imágenes, solo permite descarga de imágenes preconstruida (versión 3)

Context

- Directorio donde se encuentra el Dockerfile

Dockerfile

- Nombre alternativo de Dockerfile

ARGS

- Argumentos aplicables a los dockerfiles para construirlo en base a ellos

```
ARG db-name
```

```
RUN createdb.sh "$db-name"
```

command

- Permite sobreescribir el comando por defecto

```
command: java -jar service-spring-boot.war
```

14.2.2. deploy

- Permite definir directivas exclusivas de swarm, e ignoradas con docker-compose up o docker-compose run

```
version: '3'  
services:  
  redis:  
    image: redis:alpine  
    deploy:  
      replicas: 6  
      update_config:  
        parallelism: 2  
        delay: 10s  
      restart_policy:  
        condition: on-failure
```

mode

- global: un contenedor por swarm

```
version '3'  
services:  
  backend:  
    image: kane_project/backend:0.0.1-SNAPSHOT  
    deploy:  
      mode: global
```

- replicated: un número específico de imágenes

replicas

- Indica el número de réplicas

```
version '3'  
services:  
  frontend:  
    image: kane_project/frontend:0.0.1-SNAPSHOT  
    deploy:  
      mode: replicated  
      replicas: 3
```

placements

- Permite indicar restricciones de despliegues

```
deploy:  
  placement:  
    constraints:  
      - node.role == manager
```

update_config

- INDICA como debe actualizarse el servicio
- **parallelism** : Contenedores a actualizar en paralelo
- **delay** : tiempo de actualizaciones entre contenedores en paralelo
- **failure_action** : que hacer si falla la actualización **continue** o **pause**
- **monitor** : Tiempo de comprobacion para monitorizar el fallo, (0s por defecto)
- **max_failure_ratio** : tolerancia a fallos en actualizaciones

```
deploy:  
  replicas: 2  
  update_config:  
    parallelism: 2  
    delay: 10s
```

resources

- Indica que recursos consume cada contenedor

```
deploy:  
  resources:  
    limits:  
      cpus: '0.02'  
      memory: 200M  
    reservations:  
      cpus: '0.01'  
      memory: 100M
```

restart_policy

- Política de reinicio al salirse el contenedor
- **condition** : **none**, **on-failure**, **any**
- **delay** : espera entre reinicios
- **max_attempts** : máximo número de intentos
- **window** : tiempo maximo para considerar que se ha realizado el reinicio

14.2.3. depends_on

- Dependencias entre servicios. Indica el orden de inicio de los servicios

```
version '3'  
services:  
  frontend:  
    image: kane_project/frontend:0.0.1-SNAPSHOT  
    depends_on:  
      - backend  
  backend  
    image: kane_project/backend:0.0.1-SNAPSHOT
```

14.2.4. entrypoint

- Permite modificar el entrypoint definido

14.2.5. env_file

- Fichero de variables de entorno

```
env_file:  
  - ./base.env  
  - ./domain.env
```

14.2.6. environment

- Lista de variables de entorno

```
environment:  
  - APP_VERSION=0.0.1-SNAPSHOT  
  - SETTINGS_KEY=TESTING
```

14.2.7. expose

- Exposición de puertos

```
expose:  
  - "8080"  
  - "9990"
```

14.2.8. healthcheck

- Configuración de chequeo de salud para comprobar si un servicio funciona correctamente

```
healthcheck:  
  test: curl -f http://localhost  
  interval: 2m  
  timeout: 15s  
  retries: 2
```

14.2.9. image

- Especifica el nombre de la imagen

```
image: redis:latest
```

14.2.10. links

- Enlace a contenedores en servicios distintos

```
frontend:  
  links:  
    - backend-db  
    - backend-service
```

14.2.11. network_mode

- Opción similar a --net , con opción de definir servicio:nombre-servicio

```
network_mode: "host"
```

14.2.12. networks

- Permite indicar a que redes se unirán

```
services:  
  frontend:  
    networks:  
      - network-app1  
      - network-app2
```

14.2.13. ports

- Exposición de puertos

```
ports:  
- "8080"  
- "9990:9990"  
- "1024-1031"
```

14.2.14. ulimits

- Permite definir los límites de apertura de ficheros por contenedor

```
ulimits:  
nproc: 65535  
nofile:  
soft: 20000  
hard: 40000
```

14.2.15. volumes

- Permite indicar los volúmenes a resolver en los contenedores

```
volumes:  
- /var/log/httpd  
- /var/lib/mysql:/opt/mysql
```

14.2.16. restart

- condiciones de reinicio
- **no** bajo ningún concepto
- **always** siempre
- **on-failure** si se produce un error
- **unless-stopped** en caso de parada

14.2.17. privileged

- Si queremos iniciararlo como privilegiado

```
privileged: true
```

14.2.18. driver

- Indica el tipo de driver de red a utilizar. Overlay en swarm, bridge en docker

```
driver: overlay
```

Ejemplo de fichero docker-compose

```
version: '2' # indica la versión
services: # lista de servicios
  web: # nombre de servicio
    build: . # ruta de Dockerfile
    ports: # lista de puertos expuestos
      - "5000:5000"
    volumes: # lista de puntos de montaje
      - ./code
    links: # enlace al contenedor redis
      - redis
  redis: # nombre del servicio
    image: redis # nombre de imagen a utilizar
```

14.3. Inicio de docker-compose

- La instalación más sencilla es por medio de la descarga del comando y asignación de permisos

```
$ sudo curl -L "https://github.com/docker/compose/releases/download/1.16.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
$ docker-compose -version
```

14.4. Creación de imágenes

- Creamos una imagen para realizar una prueba de servidor con nodejs y express. .Dockerfile

```
FROM node

# Agregamos el directorio src
ADD src/ /src
# Accedemos al directorio src
WORKDIR /src
# Definimos las dependencias
RUN npm install
# Exponemos el puerto 80
EXPOSE 80
# Iniciamos el proceso node
CMD ["node", "index.js"]
```

- Generamos el fichero de ejecución de nodejs para ejecutar en el contenedor de docker

```
var express = require('express');
var os = require("os");

var app = express();
var hostname = os.hostname();

app.get('/', function (req, res) {
  res.send('<html><body>ejemplo desde express con Node.js desde el contenedor ' + 
hostname + '</body></html>');
});

app.listen(80);
console.log('Iniciado en http://localhost');
```

- Incluimos el fichero de empaquetado

```
{
  "name": "ejemplo-node-express",
  "private": true,
  "version": "0.0.1",
  "description": "Ejemplo de node y express con docker",
  "author": "rubengomez78@gmail.com",
  "dependencies": {
    "express": "4.12.0"
  }
}
```

- Creamos el contenedor y probamos que el servicio con express funciona correctamente

```
$ docker build -t cursodocker/express-example .
$ docker run -p 81:80 --name express -d cursodocker/express-example
```

- Ahora comprobemos el rendimiento de nuestra aplicación usando el potencial de docker y el Apache Bench

```
$ docker run --rm --network=host jordi/ab ab -n 10000 -c 10 http://localhost:81/
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

```
Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
```

```
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests
```

Server Software:

```
Server Hostname:      localhost
Server Port:          81
```

```
Document Path:        /
Document Length:      67 bytes
```

```
Concurrency Level:    10
```

```
Time taken for tests: 9.479 seconds
```

```
Complete requests:   10000
```

```
Failed requests:     0
```

```
Total transferred:  2470000 bytes
```

```
HTML transferred:   670000 bytes
```

```
Requests per second: 1054.95 [#/sec] (mean)
```

```
Time per request:    9.479 [ms] (mean)
```

```
Time per request:    0.948 [ms] (mean, across all concurrent requests)
```

```
Transfer rate:       254.47 [Kbytes/sec] received
```

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	0	0.3	0
Processing:	2	9	2.9	9
Waiting:	1	9	2.8	8
Total:	2	9	2.9	43

Percentage of the requests served within a certain time (ms)

50%	9
66%	9
75%	10
80%	10
90%	12
95%	13
98%	16
99%	22
100%	43 (longest request)

- Estos son los datos que nos muestra la aplicación: 1054,95 peticiones por segundo.
- Aprovechemos docker compose para desplegar un cluster de servidores web y repitamos las pruebas

- Creamos un HAProxy para que haga de balanceador entre los contenedores
- Usamos la misma imagen oficialde HAProxy, por medio de el repositorio haproxy y tag alpine
- Definimos el fichero haproxy.cfg que redirige las peticiones entre los clientes

haproxy/haproxy.cfg

```
global
  log 127.0.0.1 local0
  log 127.0.0.1 local1 notice

defaults
  log global
  mode http
  option httplog
  option dontlognull
  timeout connect 5000
  timeout client 10000
  timeout server 10000

frontend balancer
  bind 0.0.0.0:80
  mode http
  default_backend aj_backends

backend aj_backends
  mode http
  option forwardfor
  # http-request set-header X-Forwarded-Port %[dst_port]
  balance roundrobin
  server express1 express1:80 check
  server express2 express2:80 check
  server express3 express3:80 check
  # option httpchk OPTIONS * HTTP/1.1\r\nHost:\localhost
  option httpchk GET /
  http-check expect status 200
```

- Generamos ahora el fichero .yml que pertenece a docker compose:

docker-compose.yml

```
version: '3'
services:
  express1:
    build: .
    expose:
      - 80

  express2:
    build: .
    expose:
      - 80

  express3:
    build: .
    expose:
      - 80

  haproxy:
    image: haproxy:alpine
    volumes:
      - ./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro
    ports:
      - "80:80"
    expose:
      - "80"
```

- Ahora levantamos el servicio con docker compose

```
$ docker-compose up
Starting dockercompose_express2_1
Starting dockercompose_express1_1
Starting dockercompose_express3_1
Creating dockercompose_haproxy_1
Attaching to dockercompose_express2_1, dockercompose_express3_1,
dockercompose_express1_1, dockercompose_haproxy_1
express2_1 | Iniciado en http://localhost
express3_1 | Iniciado en http://localhost
express1_1 | Iniciado en http://localhost
haproxy_1  | <7>haproxy-systemd-wrapper: executing /usr/local/sbin/haproxy -p
/run/haproxy.pid -f /usr/local/etc/haproxy/haproxy.cfg -Ds
```

- Ahora podemos realizar peticiones contra el puerto 80 y comprobar como redirige a los distintos contenedores

```
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 22468f2328a6</body></html>
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 2f9d169e7f6b</body></html>
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 07431198daef</body></html>
$ curl localhost
<html><body>ejemplo desde express con Node.js desde el contenedor 22468f2328a6</body></html>
```

- Podemos realizar de nuevo la prueba del cluster de servidores, aunque en el caso de virtualización no ofrecerá los resultados deseados. No siempre por tener más es mejor!

```
$ docker run --rm --network=host jordi/ab ab -n 10000 -c 10 http://localhost/
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

Benchmarking localhost (be patient)

```
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests
```

Server Software:

```
Server Hostname:      localhost
Server Port:         80
```

Document Path: /

```
Document Length:    92 bytes
```

Concurrency Level: 10

Time taken for tests: 12.528 seconds

Complete requests: 10000

Failed requests: 0

Total transferred: 2730000 bytes

HTML transferred: 920000 bytes

Requests per second: 798.21 [#/sec] (mean)

Time per request: 12.528 [ms] (mean)

Time per request: 1.253 [ms] (mean, across all concurrent requests)

Transfer rate: 212.80 [Kbytes/sec] received

Connection Times (ms)

	min	mean	[+/-sd]	median	max
Connect:	0	0	0.4	0	19
Processing:	2	12	3.9	12	84
Waiting:	2	12	3.8	11	84
Total:	2	12	4.0	12	85

Percentage of the requests served within a certain time (ms)

50%	12
66%	13
75%	14
80%	14
90%	15
95%	18
98%	22
99%	25
100%	85 (longest request)

14.5. Ventajas e inconvenientes de docker-compose

- Por defecto, genera una interfaz de red única por aplicación
- No usa el link entre contenedores, sino la interfaz
- Usa network de tipo overlay en deployments de Swarm y despliega en bridge bajo el comando de docker-compose
- El comando de docker-compose no es capaz de desplegar por si mismo en un swarm.

14.6. Ejemplos en version 3

```
version: "3"

services:
  vote:
    build: ./vote
    command: python app.py
    volumes:
      - ./vote:/app
    ports:
      - "5000:80"
    networks:
      - front-tier
      - back-tier

  result:
    build: ./result
    command: nodemon --debug server.js
    volumes:
      - ./result:/app
```

```

ports:
  - "5001:80"
  - "5858:5858"
networks:
  - front-tier
  - back-tier

worker:
  build:
    context: ./worker
  networks:
    - back-tier

redis:
  image: redis:alpine
  container_name: redis
  ports: ["6379"]
  networks:
    - back-tier

db:
  image: postgres:9.4
  container_name: db
  volumes:
    - "db-data:/var/lib/postgresql/data"
  networks:
    - back-tier

volumes:
  db-data:

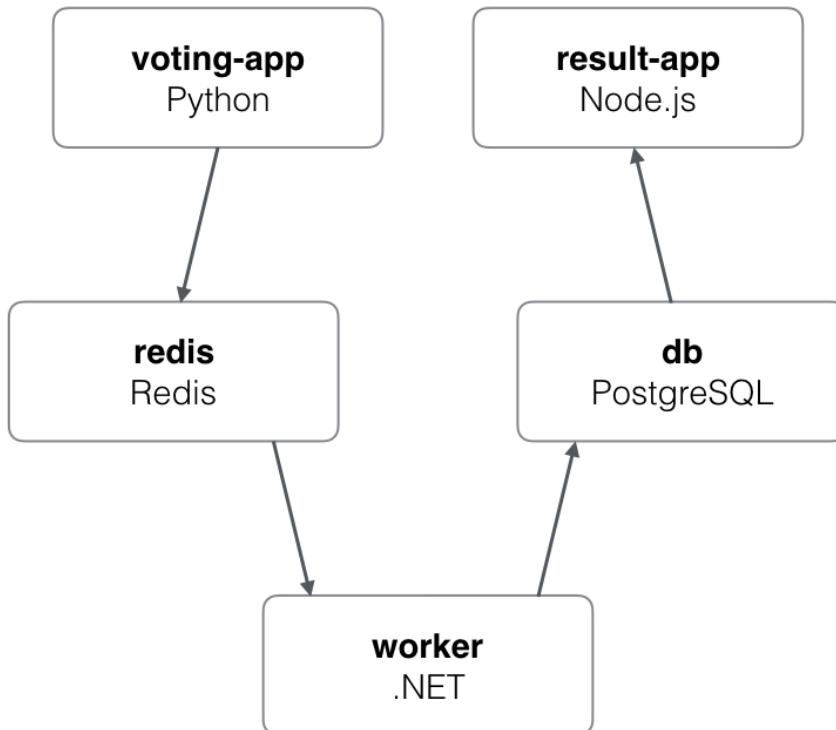
networks:
  front-tier:
  back-tier:

```

- Para iniciar el proyecto, podemos bajarlo de
 - <https://github.com/dockersamples/example-voting-app>

```
$ git clone https://github.com/dockersamples/example-voting-app
```

- La arquitectura de la aplicación es:



- Podemos ver los resultados de la app en :

- <http://localhost:5000> (app)
- <http://localhost:5001> (resultados)



Si no inicia por fallo del comando mv de la imagen de nodejs, editar el Dockerfile y usar: RUN cp -R node_modules /

Capítulo 15. Docker swarm

- En las últimas versiones, a partir de la versión v1.12 incluye su modo swarm
- Podemos usar Docker CLI para crear un swarm, desplegar servicios de aplicación y gestionar un entorno swarm

15.1. Características

- Gestión de cluster integrado en el motor de Docker
- Diseño descentralizado
- Modelo de servicio declarativo
- escalado - Generación de tareas y adaptación de estados
- Monitorización de nodos
- Gestión de redes multihost. Uso de redes overlay
- Descubrimiento de servicios por nombre dns
- Balanceo de carga
- Seguro al usar TLS para autenticación y encriptación de comunicaciones
- Actualizaciones en cadena

15.2. Conceptos

15.2.1. swarm

- Swarmkit está embebido en el docker engine.
- Se trata de un kit de gestión de cluster y orquestación
- Se inicializar swarm o unirse a un swarm existente
- Diferencia entre ejecutar contenedores y orquestarlos

15.2.2. nodo

- Es una instancia participante en swarm
- Para desplegar una aplicación, se accede al nodo manager que genera tareas en nodos
- Los nodos gestionan la orquestación y mantienen el estado
- El manager permiten elegir un líder para gestionar las tareas
- los nodos workers reciben y ejecutan tareas de los nodos manager
- Estos nodos por defecto son también workers, pero puede obligarse a que no lo sean.

15.2.3. servicios y tareas

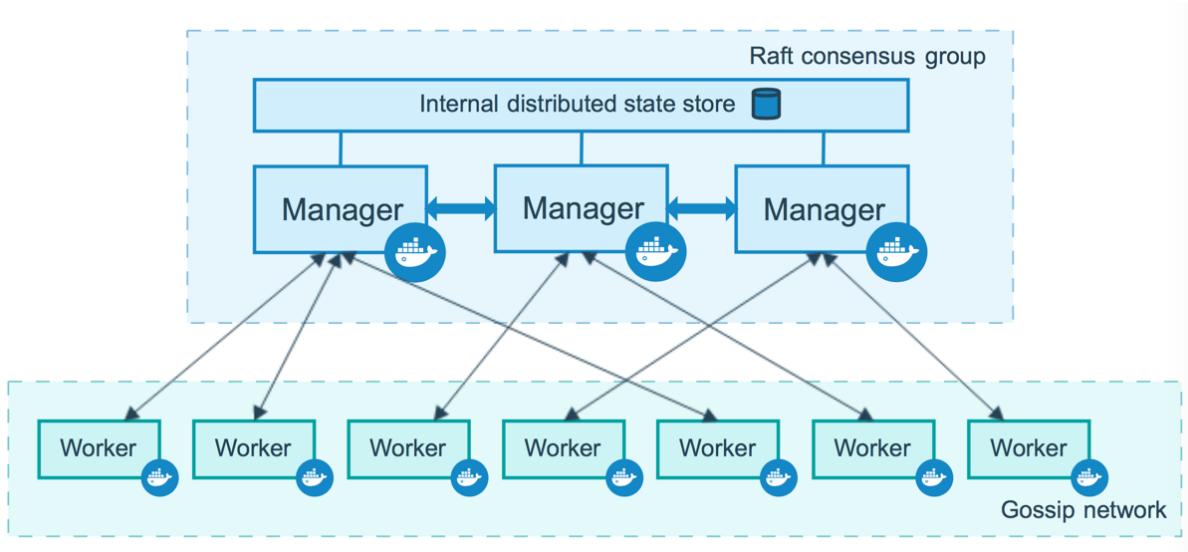
- Un servicio es una tarea a ejecutar por un worker
- Al crear un servicio, se especifica la imagen y los comandos a ejecutar
- Para servicios replicados, el manager distribuye las tareas réplica para ejecutar en nodos para establecer el estado final
- Para servicios globales, swarm ejecuta una sola tarea para el servicio por cada nodo disponible en el cluster
- Una tarea lanza los comandos dentro de un contenedor
- El manager asigna las tareas a los nodos según el número de réplicas

15.2.4. Balanceo de carga

- Uso de **ingress** para balanceo de carga
- Permite exponer asignar automáticamente un puerto publicado o asignar uno
- Swarm posee un dns interno que asigna cada servicio una entrada DNS
- Uso de un balanceo de carga interno

15.3. Funcionamiento

- Swarm define uno o más nodos físicos
- El modelo es el siguiente:



- Los nodos manejados gestionan las tareas del cluster
 - Permite mantener el estado del cluster
 - Programación de servicios
 - Define un Endpoint API
- Por medio de un algoritmo de consenso Raft, los managers mantienen un estado interno consistente

- En desarrollo, lo normal es usar un solo manager, en producción se recomienda usar un número de managers superior
 - tres managers toleran la caída de un solo manager
 - Cinco managers toleran dos managers caídos
 - La fórmula es $(n-1)/2$ nodos caídos
 - Máximo de 7
- Los workers son instancias de docker engine
- Se dedican a ejecutar contenedores
- Por defecto, un manager es un worker también y puede desplegar contenedores
- Se puede evitar drenando el nodo.
- Los workers se pueden promover a managers

```
$ docker node promote
```

- Y permite dejar que un manager pase a ser solo worker

```
$ docker node demote
```

15.4. Creación de swarm

- Para iniciar swarm, debemos declarar la dirección inicial del manager

```
$ docker swarm init --advertise-addr 192.168.250.100
```

- Configura el nodo manager para publicar su dirección y poder agregar workers o managers
- Segundo el --token que se manda

```
# docker swarm join-token manager
docker swarm join \
--token SWMTKN-1-0us2esc9srlieht8zq11t1p0svw5iiayit0ref44cb5grvfoes-
ajw3o57zjiw7m140kj0kw4o8g \
192.168.250.100:2377
docker swarm join-token worker
docker swarm join \
--token SWMTKN-1-0us2esc9srlieht8zq11t1p0svw5iiayit0ref44cb5grvfoes-
0f10rikyfye401hzxnm4p9xko \
192.168.250.100:2377
```

- Para comprobar el estado de los nodos. El asterisco indica a cual estamos conectados

```
$ docker node ls
[root@manager ~]# docker node ls
ID                      HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
0blcxu4g9o2sq856xvxdegy55  worker   Ready   Active
9grec76ftgzaybpgc0h6fggag  worker1  Ready   Active
dlvv9imw0r1m86li67ls4ntra *  manager   Ready   Active      Leader
```

15.5. Agregación de servicios

- para crear contenedores, creamos un servicio con el comando de creación

```
docker service create --replicas 1 --name test ubuntu sleep 10000
```

- las replicas indica cuantos servicios se van a replicar
- service create es la creación del servicio
- Para monitorizar los servicios ejecutados

```
# docker service ls
ID          NAME    REPLICAS  IMAGE        COMMAND
1c3igjg01ejx  test     1/1      ubuntu      sleep 10000
```

15.6. Monitorización de servicios

- Con el comando docker service inspect podemos mostrar la información del contenedor

```
# docker service inspect test --pretty
ID:      1c3igjg01ejxa12rjxe1jwe43
Name:    test
Mode:    Replicated
Replicas: 1
Placement:
UpdateConfig:
  Parallelism: 1
  On failure:  pause
ContainerSpec:
  Image:    ubuntu
  Args:    sleep 10000
Resources:
```

- podemos mostrar los servicios ejecutados

```
# docker service ps test
ID                  NAME     IMAGE      NODE      DESIRED STATE  CURRENT STATE
ERROR
9zpl5eit7subbav0up9jrv0k9  test.1  ubuntu    worker1  Running      Running 5 minutes
ago
```

- Si ejecutamos docker ps en el servidor, muestra los contenedores ejecutandose en el nodo actual

15.7. Escalar un servicio

- Para gestionar el número de contenedores en swarm

```
# docker service scale test=3
test scaled to 3
```

```
# docker service ps test
ID                  NAME     IMAGE      NODE      DESIRED STATE  CURRENT STATE
ERROR
9zpl5eit7subbav0up9jrv0k9  test.1  ubuntu    worker1  Running      Running 32 minutes
ago
54th2cay4z05aub7d5it1un4l  test.2  ubuntu    manager   Running      Preparing 21
seconds ago
c71xwtynp59isiml06o2fw7j5  test.3  ubuntu    worker1  Running      Running 15 seconds
ago
```

15.8. Actualización de contenedores

- Podemos actualizar los contenedores secuencialmente delegando la responsabilidad a swarm

```
$ docker service update --image ubuntu:16.04 test
$ docker service ps test
ID                  NAME     IMAGE      NODE      DESIRED STATE  CURRENT
STATE      ERROR
67cfssuuomd97dbycmdgyc9z  test.1  ubuntu:16.04  manager   Ready      Ready 2
seconds ago
9zpl5eit7subbav0up9jrv0k9  \_ test.1  ubuntu      worker1  Shutdown   Running
37 minutes ago
2gj34x8px4gei17jllob8rcnp  test.2  ubuntu:16.04  worker1  Running   Running 8
seconds ago
54th2cay4z05aub7d5it1un4l  \_ test.2  ubuntu      manager   Shutdown   Shutdown
8 seconds ago
c71xwtynp59isiml06o2fw7j5  test.3  ubuntu      worker1  Running   Running 5
minutes ago
```

- Por defecto, las actualizaciones dejan un histórico de versiones anteriores disponibles

- Para manipular el número máximo de históricos, podemos indicarlo con:

```
$ docker swarm update --task-history-limit 2
```

15.9. Borrado de servicios

- docker service rm test permite borrar el servicio test y sus tres contenedores

15.10. Drenar nodos

- Podemos definir el drenado de nodos para que un nodo deje de recibir nuevas tareas

```
docker node update --availability drain worker1
```

- Para ver su estado

```
docker node inspect --pretty worker1
```

15.11. Deploy

- Una de las últimas opciones de despliegue incluye la posibilidad de usar los ficheros docker-compose.yml de tercera generación como orquestadores de servicios.
- Sin embargo, esta opción se considera experimental. Para poder usarla, debemos crear un fichero en /etc/docker/daemon.json con el siguiente contenido

```
{"experimental":true}
```

- Si reiniciamos los servicios de docker ya podremos usar **docker deploy**

15.11.1. Ejemplo

- Vamos a desplegar con docker una solución similar al primer ejemplo donde usamos un haproxy para balancear las peticiones entre servicios
- En este caso no poseemos un registro público donde poder subir las imágenes, así que generaremos la imagen en los tres nodos para que estén en sus repositorios locales.

/repetimos en los tres nodos

```
docker build -t cursodocker/express:latest /vagrant/01_docker-compose-base-swarm
```

- Una vez generado, podemos publicar el nuevo servicio

```

version: '3'
services:
  express:
    image: cursodocker/express:latest
    ports:
      - 80:80
    deploy:
      mode: replicated
      replicas: 3
      placement:
        constraints: [node.role == worker]

```

- Podemos comprobar como no usamos haproxy para balancear los servicios, y usamos el puerto 80 para comunicarnos.
- Desplegamos el servicio

```
docker deploy --compose-file docker-compose-swarm.yml BASE
```

- La palabra BASE se usará como alias para crear la red overlay y el servicio
- Si consultamos con distintos navegadores por el puerto 80 de cualquiera de los servidores, veremos como nos responde cualquiera de los servicios.
- Para consultar el estado del cluster, y saber donde están los nodos, podemos usar la imagen de visualizer, lo que nos permite ver información a nivel de swarm

```
docker run -it -d --name swarm_visualizer -p 8000:8080 -e HOST=localhost -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer
```

- Consultando el servicio:



15.11.2. Ejemplo avanzado

- También podemos aumentar la complejidad del ejemplo, usando múltiples contenedores y reglas

Ejemplo avanzado

```
version: "3"

services:

  redis:
    image: redis:3.2-alpine
    ports:
```

```

    - "6379"
networks:
  - voteapp
deploy:
  placement:
    constraints: [node.role == manager]

db:
  image: postgres:9.4
  volumes:
    - db-data:/var/lib/postgresql/data
networks:
  - voteapp
deploy:
  placement:
    constraints: [node.role == manager]

voting-app:
  image: gaiadocker/example-voting-app-vote:good
  ports:
    - 5000:80
networks:
  - voteapp
depends_on:
  - redis
deploy:
  mode: replicated
  replicas: 2
  labels: [APP=VOTING]
  placement:
    constraints: [node.role == worker]

result-app:
  image: gaiadocker/example-voting-app-result:latest
  ports:
    - 5001:80
networks:
  - voteapp
depends_on:
  - db

worker:
  image: gaiadocker/example-voting-app-worker:latest
  networks:
    voteapp:
      aliases:
        - workers
  depends_on:
    - db
    - redis
  deploy:

```

```

mode: replicated
replicas: 2
labels: [APP=VOTING]
resources:
  # Consumo máximo
  limits:
    cpus: '0.25'
    memory: 512M
  # Intenta volver a estos valores
  reservations:
    cpus: '0.25'
    memory: 256M
# Políticas de reinicio
restart_policy:
  condition: on-failure
  delay: 5s
  max_attempts: 3
  window: 120s
# Políticas de actualización
update_config:
  parallelism: 1
  delay: 10s
  failure_action: continue
  monitor: 60s
  max_failure_ratio: 0.3
# solo en workers
placement:
  constraints: [node.role == worker]

networks:
  voteapp:

volumes:
  db-data:

```

- Si iniciamos este sistema:

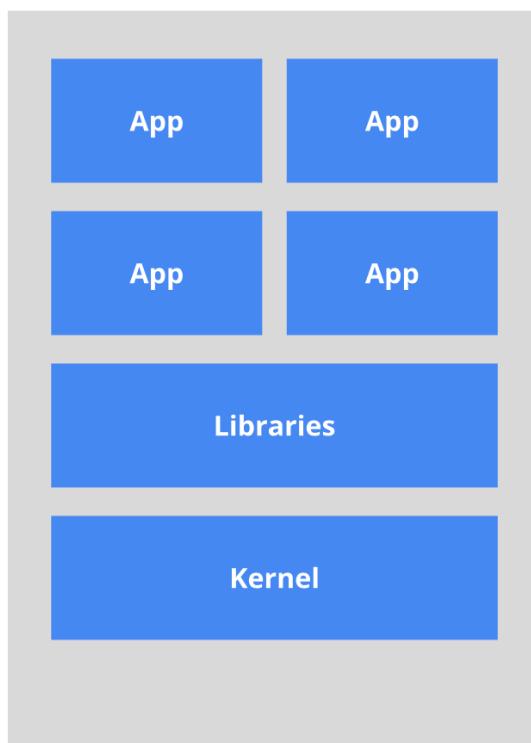
```
docker deploy --compose-file docker-compose-adv-swarm.yml VOTOS
```

- La gran ventaja es que podremos consultar cualquiera de los servicios públicos desde cualquier nodo, y todos los recursos están repartidos por todo el cluster.

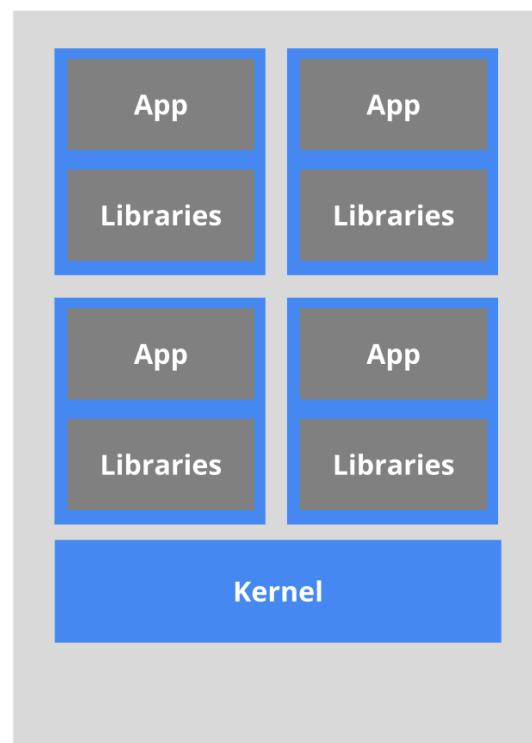
Capítulo 16. Kubernetes (k8s)

16.1. Introducción

- Helmsman o Pilot
- Sistema open-source de automatización de despliegues, escalado y gestión de aplicaciones en contenedores.
- Entre las características principales se encuentran:
 - Despliegue de contenedores por requerimientos y otras variables.
 - Reinicio de contenedores en caso de fallo.
 - Recolocación de contenedores en caso de caída de nodos.
 - Escalado horizontal
 - Descubrimiento de servicios y balanceo de carga
 - Actualizaciones y rollbacks automáticos
 - Orquestación de almacenamiento.
 - Gestión de ejecuciones Batch.
- El despliegue de aplicaciones permite usar contenedores que contienen las librerías necesarias para su funcionamiento aislando la aplicación de las demás



*Heavyweight, non-portable
Relies on OS package manager*



*Small and fast, portable
Uses OS-level virtualization*

- Los contenedores son pequeños y rápidos.
- Una aplicación se empaqueta en una imagen de contenedor.

- Son imágenes inmutables

16.2. Características

- Kubernetes permite programar y ejecutar aplicaciones de contenedores en clusters físicos o virtuales
- Permite la abstracción de esta característica y pasar a una infraestructura de contenedores como la que kubernetes ofrece.
- Kubernetes permite
 - Monitorización del sistema de almacenamiento
 - Distribución de claves
 - Monitorización de salud de aplicaciones
 - Replicación de instancias de aplicación
 - Autoescalado de Pods horizontal
 - Nombrado y descubrimiento
 - Balanceo de carga
 - Actualizaciones en cascada
 - Monitorización de recursos
 - Logs
 - Debug de aplicaciones
 - Autenticación y autorización

16.3. Restricciones

- No limita las aplicaciones soportadas basadas en lenguajes
- No distingue entre aplicaciones y servicios
- No provee de ningún tipo de middleware, frameworks de procesamiento de datos, base de datos, ni sistema de almacenamiento de disco en cluster, pero permite la ejecución de dichos recursos en kubernetes
- No existe un marketplace para instalación sencilla
- No realiza CI ni compila.
- Permite que los usuarios definan el log de aplicación, monitorización y alerta.
- No provee de una configuración, gestión o sistema de auto-healing

16.4. Componentes

16.4.1. Master

- Provee del panel de control de Kubernetes. Permite la toma de decisiones global sobre el cluster

- Se pueden ejecutar en cualquier nodo del cluster, aunque suelen ejecutarse en la misma VM, y prohíbe la ejecución de contenedores de usuario dentro del nodo.

16.4.2. Kube-apiserver

- Expone el API de kubernetes. Es el panel de control, designado para escalado horizontal

16.4.3. etcd

- Usado como almacenamiento back. Todos los datos del cluster se almacenan aquí. Se deben realizar backups continuos de los datos de este contenedor.

16.4.4. Kube-controller-manager

- Ejecuta hilos que gestionan tareas en el cluster. Cada controlador es un proceso separado. Está compilado como un binario y ejecutado en un solo proceso
 - Control de nodos: Responsable de indicar cuando un nodo se cae
 - Control de replicación: Mantiene el número correcto de PODS para cada objeto de control de replicación en el sistema
 - Punto de entrada de Control: Publica Endpoints, uniones entre servicios y pods
 - Servicio de cuentas y controladores de token. Crea las cuentas por defecto y los tokens de acceso a las apis para nuevos namespaces

16.4.5. cloud-controller-manager (Nuevo en 1.6)

- Ejecuta controladores que interactúan con los proveedores de cloud.
- Permite una capa de abstracción con el código de los proveedores de servicios de cloud

16.4.6. kube-scheduler

- Vigila que los pods creados que no tienen nodo asignado sean reasignados

16.5. Componentes del nodo

- Estos componentes se ejecutan en cada nodo

16.5.1. kubelet

- Agente del nodo
- Busca los pods asignados al nodo
 - Monta los volúmenes requeridos del POD
 - Descarga los *secrets* del POD
 - Periódicamente ejecuta un test de vida al contenedor indicado
 - Informa del estado del POD, y crea un POD espejo si es necesario

- Informa del estado del nodo al sistema

16.5.2. kube-proxy

- Activa la abstracción manteniendo las reglas de red en el host y realizando forwarding de conexiones

16.5.3. docker

- Ejecución de contenedores

16.5.4. rkt

- Alternativa a docker

16.5.5. supervisord

- Proceso ligero para monitorización y control que vigila docker y kubelet

16.5.6. fluentd

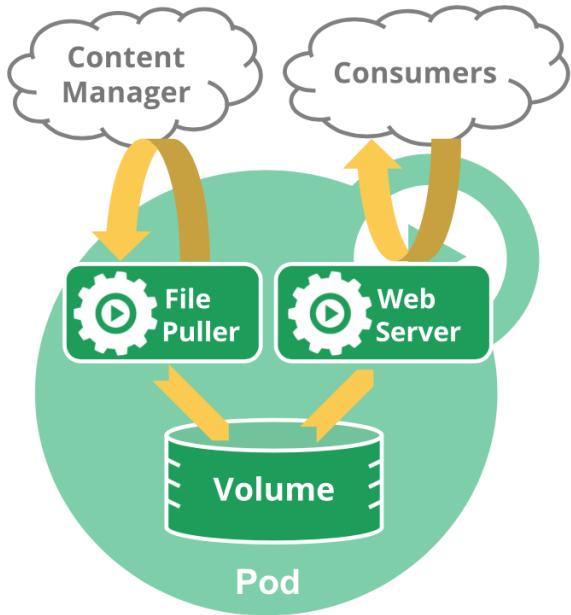
- Permite el logging a nivel de cluster.

16.6. PODS

- Se trata de un bloque básico de construcción de kubernetes
- Es la unidad mas simple
- Encapsula un contenedor de aplicación, o en algunos casos, múltiples contenedores
- Encapsula almacenamiento, una ip de red única, y opciones para orquestar los contenedores
- Representa una instancia con contenedores acoplados
- Objetivo
 - Un contenedor por POD. Es el caso de uso más común.
 - Muchos contenedores por POD. Contenedores que deben estar juntos para compartir recursos.

16.6.1. Gestión de múltiples contenedores

- Soportan múltiples procesos intercomunicados (contenedores) que conforman una unidad de servicio
- El POD siempre se crea en la misma máquina física, y solo se deben definir si hay recursos compartidos y acoplados



16.6.2. Red

- Cada POD posee una dirección de red única que comparte el namespace de red, dirección ip y puertos de red.
- Cada contenedor se comunica con cualquier otro por medio de localhost en el mismo POD

16.6.3. Fases

- Pending: Aceptado por kubernetes pero la imagen no ha sido creada
- Running: POD asignado a un nodo, y los contenedores han sido creados. Proceso de inicio o iniciando
- Succeeded: Contenedores en ejecucion sin problemas
- Failed: Todos los contenedores han terminado
- Unknown: No es posible saber el estado del POD

16.6.4. Ejemplo de definición de un POD

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
    - args:
        - /server
      image: gcr.io/google_containers/liveness
      livenessProbe:
        httpGet:
          # when "host" is not defined, "PodIP" will be used
          # host: my-host
          # when "scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" and "HTTPS" are allowed
          # scheme: HTTPS
          path: /healthz
          port: 8080
          httpHeaders:
            - name: X-Custom-Header
              value: Awesome
      initialDelaySeconds: 15
      timeoutSeconds: 1
      name: liveness

```

16.6.5. Pods de infraestructura

DNS

- Todos los clusters de kubernetes deben tener un DNS.
- Se trata de un DNS para servicios de kubernetes
- Todos los contenedores inicializados por kubernetes poseen este DNS por defecto

Interfaz de usuario (kube-ui)

- Se trata de una interfaz de lectura para ver el estado del cluster

Log de Cluster

- Responsable de guardar los logs del contenedor en un almacen central con una interfaz de búsqueda y exploración

16.7. Namespaces

- Kubernetes soporta multiples clusters virtuales gestionados desde el mismo cluster físico. Estos clusters virtuales se les llama namespaces
- Permite gestionar visibilidad de objetos a partir del namespace
- Para ver los namespaces

```
$ kubectl get namespaces
NAME        LABELS      STATUS
default     <none>     Active
kube-system <none>     Active
```

- kube-system es el namespace para los objetos creados por kubernetes
- Para ver los pods de un namespace o ejecutar un pod en un namespace

```
$ kubectl --namespace=default run nginx --image=nginx
$ kubectl --namespace=kube-system get pods
```

16.8. Instalación de minikube

- Minikube es una distribución basada en virtualización que permite crear un cluster de kubernetes de un solo nodo de forma sencilla para pruebas conceptuales.
- Para ello se necesita instalar VirtualBox, minikube y kubectl para gestionar el cluster

16.8.1. Instalación de kubectl

- Para instalar kubectl descargamos la última versión por medio del comando de descarga:
 - [Windows](#)
 - [Linux](#)
 - [OS X](#)
- Para linux y mac cambiamos los permisos de ejecución

```
$ chmod +x ./kubectl
```

16.8.2. Instalación de minikube

- La instalación de minikube es similar al anterior. Podemos descargarlo para cada distribución en:
 - <https://storage.googleapis.com/minikube/releases/>
- Para windows podemos usar el binario y renombrarlo a minikube.exe
 - <https://storage.googleapis.com/minikube/releases/v0.19.0/minikube-windows-amd64.exe>
- Es imprescindible que tanto minikube como kubectl tengan permiso de ejecución y estén en el path

16.8.3. Creación del cluster

```
$ minikube start
Starting local Kubernetes v1.6.0 cluster...
Starting VM...
Downloading Minikube ISO
 89.51 MB / 89.51 MB [=====] 100.00% 0s
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

- Una vez creado, podemos iniciar la aplicación hello-minikube

```
$ kubectl run hello-minikube --image=gcr.io/google_containers/echoserver:1.4 --port=8080
deployment "hello-minikube" created
```

- Para exponer el despliegue al exterior

```
$ kubectl expose deployment hello-minikube --type=NodePort
service "hello-minikube" exposed
```

- Podemos listar los pods desplegados

```
$ kubectl get pod
NAME                  READY   STATUS    RESTARTS   AGE
hello-minikube-938614450-v3tq7   1/1     Running   0          33s
```

- Podemos saber la dirección expuesta del servicio

```
$ minikube service hello-minikube --url
```

- El resultado de la petición via web con curl

```

$ curl $(minikube service hello-minikube --url)
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://192.168.99.100:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=*/
host=192.168.99.100:32099
user-agent=curl/7.51.0
BODY:
-no body in request-

```

- Con kubectl version y cluster info podemos ver mas información del cluster

```

$ kubectl version
Client Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.4",
GitCommit:"d6f433224538d4f9ca2f7ae19b252e6fcbb66a3ae", GitTreeState:"clean", BuildDate:"2017-05-19T18:44:27Z",
GoVersion:"go1.7.5", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"6", GitVersion:"v1.6.0",
GitCommit:"fff5156092b56e6bd60fff75aad4dc9de6b6ef37", GitTreeState:"clean", BuildDate:"2017-05-09T23:22:45Z",
GoVersion:"go1.7.3", Compiler:"gc", Platform:"linux/amd64"}
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

```

- Instalamos una aplicación de testing de kubernetes

```

$ kubectl run kubernetes-bootcamp --image=docker.io/jocatalin/kubernetes-bootcamp:v1 --port=8080
deployment "kubernetes-bootcamp" created
$ kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-minikube   1         1         1           1          11m
kubernetes-bootcamp   1         1         1           0          14s

```

- Podemos iniciar un proxy para publicar el API de comunicaciones

```

$ kubectl proxy
Starting to serve on 127.0.0.1:8001

```

- Desde otra consola podemos consultar información de un pod

```

$ curl http://localhost:8001/api/v1/proxy/namespaces/default/pods/hello-minikube-938614450-v3tq7/
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://172.17.0.2:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=*/
accept-encoding=gzip
host=172.17.0.2:8080
user-agent=curl/7.51.0
x-forwarded-for=127.0.0.1
x-forwarded-uri=/api/v1/proxy/namespaces/default/pods/hello-minikube-938614450-v3tq7/
BODY:
-no body in request-

```

- Podemos ver el estado real de los pods

kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
hello-minikube-938614450-v3tq7	1/1	Running	1	14m
kubernetes-bootcamp-3271566451-0ng25	0/1	ContainerCreating	0	3m

- Usando el comando kubectl describe pods, permite mostrar información de pods y sus eventos para evaluar posibles problemas en despliegues
- Se puede completar con el comando logs disponible (El comando log ha sido deprecado)

```

$ kubectl logs hello-minikube-938614450-v3tq7
172.17.0.1 - - [24/May/2017:16:57:55 +0000] "GET / HTTP/1.1" 200 538 "-" "curl/7.51.0"

```

- Como con Docker, podemos conectarnos al contenedor y lanzar comandos o ejecutar instrucciones

```
$ kubectl exec hello-minikube-938614450-v3tq7 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=hello-minikube-938614450-v3tq7
KUBERNETES_PORT=tcp://10.0.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
HELLO_MINIKUBE_PORT=tcp://10.0.0.177:8080
HELLO_MINIKUBE_PORT_8080_TCP=tcp://10.0.0.177:8080
KUBERNETES_SERVICE_PORT=443
HELLO_MINIKUBE_PORT_8080_TCP_PORT=8080
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.0.0.1
HELLO_MINIKUBE_SERVICE_HOST=10.0.0.177
HELLO_MINIKUBE_SERVICE_PORT=8080
HELLO_MINIKUBE_PORT_8080_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://10.0.0.1:443
HELLO_MINIKUBE_PORT_8080_TCP_ADDR=10.0.0.177
HOME=/root
```

- Podemos acceder a una terminal de un contenedor de un pod

```
$ kubectl exec -ti kubernetes-bootcamp-3271566451-0ng25 bash
root@kubernetes-bootcamp-3271566451-0ng25:/# cat server.js
var http = require('http');
var requests=0;
var podname= process.env.HOSTNAME;
var startTime;
var host;
var handleRequest = function(request, response) {
  response.setHeader('Content-Type', 'text/plain');
  response.writeHead(200);
  response.write("Hello Kubernetes bootcamp! | Running on: ");
  response.write(host);
  response.end(" | v=1\n");
  console.log("Running On:" ,host, "| Total Requests:", ++requests,"| App Uptime:", (new Date() - startTime)/1000 ,
"seconds", "| Log Time:",new Date());
}
var www = http.createServer(handleRequest);
www.listen(8080,function () {
  startTime = new Date();
  host = process.env.HOSTNAME;
  console.log ("Kubernetes Bootcamp App Started At:",startTime, "| Running On: " ,host, "\n");
});
```

- Si queremos obtener la lista de servicios y comprobar la ip del cluster de acceso

\$ kubectl get services				
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hello-minikube	10.0.0.177	<nodes>	8080:32099/TCP	33m
kubernetes	10.0.0.1	<none>	443/TCP	34m

- Exponemos la aplicación y comprobamos el nuevo puerto abierto

```
$ kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080
service "kubernetes-bootcamp" exposed
$ kubectl get services
NAME           CLUSTER-IP   EXTERNAL-IP     PORT(S)        AGE
hello-minikube 10.0.0.177  <nodes>        8080:32099/TCP 34m
kubernetes      10.0.0.1    <none>         443/TCP       35m
kubernetes-bootcamp 10.0.0.70  <nodes>        8080:30927/TCP 3s
```

- Mostramos los servicios publicados

```
$ kubectl describe services/kubernetes-bootcamp
Name:           kubernetes-bootcamp
Namespace:      default
Labels:         run=kubernetes-bootcamp
Annotations:   <none>
Selector:      run=kubernetes-bootcamp
Type:          NodePort
IP:            10.0.0.70
Port:          <unset> 8080/TCP
NodePort:      <unset> 30927/TCP
Endpoints:    172.17.0.5:8080
Session Affinity: None
Events:        <none>
```

- Si consultamos el servidor, podemos visitar el servicio publicado a partir de la dirección ip del nodo y el puerto exportado (NodePort)

```
$ curl 192.168.99.100:30927
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-3271566451-0ng25 | v=1
```

16.9. Opciones especiales de Minikube

- Podemos acceder directamente a la máquina virtual minikube por medio del comando

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
9a9be42b598d	gcr.io/google_containers/pause-amd64:3.0	"/pause"	16 minutes ago	Up 16 minutes
k8s_POD_kubernetes-bootcamp-3271566451-0ng25_default_c7deb7c5-40a1-11e7-9bd8-0800275879ae_0				
8e5388ea6ed3	fc5e302d8309	"/sidecar --v=2 --log"	20 minutes ago	Up 20 minutes
k8s_sidecar_kube-dns-268032401-dpbm3_kube-system_19dd9625-40a0-11e7-aba3-0800275879ae_1				
2378b46f3dd6	416701f962f2	"/dashboard --port=90"	20 minutes ago	Up 20 minutes
k8s_kubernetes-dashboard_kubernetes-dashboard-6zm3q_kube-system_19c28b8e-40a0-11e7-aba3-0800275879ae_1				
56acfaf1b043	1091847716ec	"/dnsmasq-nanny -v=2"	20 minutes ago	Up 20 minutes
k8s_dnsmasq_kube-dns-268032401-dpbm3_kube-system_19dd9625-40a0-11e7-aba3-0800275879ae_1				
a71171ccaa36	gcr.io/google_containers/pause-amd64:3.0	"/pause"	20 minutes ago	Up 20 minutes
k8s_POD_kubernetes-dashboard-6zm3q_kube-system_19c28b8e-40a0-11e7-aba3-0800275879ae_1				
09b54862fef8	f8363dbf447b	"/kube-dns --domain=c"	20 minutes ago	Up 20 minutes
k8s_kubedns_kube-dns-268032401-dpbm3_kube-system_19dd9625-40a0-11e7-aba3-0800275879ae_1				
5f22e83b2574	85809f318123	"/opt/kube-addons.sh"	20 minutes ago	Up 20 minutes
k8s_kube-addon-manager_kube-addon-manager-minikube_kube-system_8538d869917f857f9d157e66b059d05b_1				
d24e38745091	a90209bb39e3	"nginx -g 'daemon off'"	20 minutes ago	Up 20 minutes
k8s_hello-minikube_hello-minikube-938614450-v3tq7_default_26345e7c-40a0-11e7-aba3-0800275879ae_1				
9fabf9f1316d	gcr.io/google_containers/pause-amd64:3.0	"/pause"	20 minutes ago	Up 20 minutes
k8s_POD_kube-addon-manager-minikube_kube-system_8538d869917f857f9d157e66b059d05b_1				
4249466997bb	gcr.io/google_containers/pause-amd64:3.0	"/pause"	20 minutes ago	Up 20 minutes
k8s_POD_kube-dns-268032401-dpbm3_kube-system_19dd9625-40a0-11e7-aba3-0800275879ae_1				
c977d321abd3	gcr.io/google_containers/pause-amd64:3.0	"/pause"	20 minutes ago	Up 20 minutes
k8s_POD_hello-minikube-938614450-v3tq7_default_26345e7c-40a0-11e7-aba3-0800275879ae_1				

16.10. Ficheros de Configuración

- Kubernetes permite desplegar distintas soluciones basadas en ficheros de configuración con formato yaml o json.
- La unidad mas sencilla es un pod, que se puede generar directamente con run, aunque siempre se recomienda usar el api

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```

- Para definir un pool de recursos podemos generar un **Deployment**

deployment-nginx.yaml

```
# Indica cual es la versión con que aplicará y validará kubernetes el fichero
apiVersion: apps/v1beta1
# Tipo de recurso. v1 solo permite pods, ya que no existían los deployments. Algunos de ellos son service, job, pod, deployment, etc. La recomendación es que siempre iniciemos deployments, no pods, ya que genera pools de recursos que es capaz de monitorizar.
kind: Deployment
# Definición de metadatos para el recurso. En este caso, es el nombre del recurso para identificar el despliegue en kubernetes
metadata:
  name: test-nginx
# Las especificaciones indican los objetos que inician el recurso
spec:
  # Genera dos pods
  replicas: 2
  # Plantilla con lo que se genera el pod. Almacenamientos, nombres, imágenes, puertos, etc.
  template:
    metadata:
      labels:
        run: test-nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

- Para desplegarlo, aplicamos el fichero generado

```
$ kubectl create -f pod-nginx.yaml
deployment "test-nginx" created
```

- Al iniciar el proceso, observamos la creación de dos pods en despliegue

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
hello-minikube-938614450-x1bb2   1/1     Running   0          28m
kubernetes-bootcamp-3271566451-kszbv   1/1     Running   0          25m
test-nginx-4077391163-cspg1       1/1     Running   0          16s
test-nginx-4077391163-k907g       1/1     Running   0          16s
```

- Podemos comprobar los despliegues y su estado

```
kubectl get deployments
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-minikube   1         1         1           1          29m
kubernetes-bootcamp   1         1         1           1          25m
test-nginx       2         2         2           2          34s
```

- Una vez desplegado, podemos aprovechar y exponer el despliegue.
- La exposición del despliegue permite acceder a una url de cluster y el acceso a un endpoint con

todos los servicios publicados

```
$ kubectl expose deployment test-nginx --type=NodePort
service "test-nginx" exposed
```

- Para comprobar la url de cluster que accede a los dos pods

```
$ minikube service test-nginx --url
http://192.168.99.100:30951
```

- Los pods poseen su propia red, y podemos ver su configuración

```
$ kubectl get pods -l run=test-nginx -o yaml|grep podIP
podIP: 172.17.0.7
podIP: 172.17.0.6
```

- Accediendo a la máquina virtual, podemos acceder a las urls de los pods

```
$ minikube ssh
$ curl 172.17.0.6
<!DOCTYPE html>
...
$ curl 172.17.0.7
<!DOCTYPE html>
...
```

- Al publicar el despliegue, este genera un servicio que podemos observar

```
$ kubectl get svc
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
hello-minikube 10.0.0.171  <nodes>       8080:30051/TCP 35m
kubernetes      10.0.0.1    <none>        443/TCP       36m
kubernetes-bootcamp 10.0.0.202 <nodes>       8080:32044/TCP 28m
test-nginx      10.0.0.248  <nodes>       80:30951/TCP  5m
```

- En la ip del cluster, podemos acceder al servicio balanceado

```
$ minikube ssh
$ curl 10.0.0.248
<!DOCTYPE html>
...
```

- Tras exponer el servicio, podemos consultar información sobre sus Endpoints y direcciones de acceso

```
$ kubectl describe svc test-nginx
Name:           test-nginx
Namespace:      default
Labels:         run=test-nginx
Annotations:    <none>
Selector:       run=test-nginx
Type:          NodePort
IP:            10.0.0.248
Port:          <unset> 80/TCP
NodePort:       <unset> 30951/TCP
Endpoints:     172.17.0.6:80,172.17.0.7:80
Session Affinity: None
Events:        <none>
```

- Para obtener los endpoints

```
$ kubectl get ep test-nginx
NAME      ENDPOINTS      AGE
test-nginx  172.17.0.6:80,172.17.0.7:80  7m
```

- La resolución de nombres dentro de los pods está resuelta por defecto con el servicio kube-dns disponible en el namespace reservado kube-system

```
$ kubectl get services kube-dns --namespace=kube-system
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kube-dns  10.0.0.10      <none>          53/UDP,53/TCP  39m
```

- Para comprobar la resolución de nombres dentro del cluster, desplegaremos un pod con una imágenes base que posee el comando nslookup

```
$ kubectl run -it curl --image=radial/busyboxplus:curl
If you don't see a command prompt, try pressing enter.
[ root@curl-57077659-58gh4:/ ]$
```

- Si comprobamos el nombre del servicio, ahora es accesible desde cualquier sitio

```
[ root@curl-57077659-58gh4:/ ]$ nslookup test-nginx
Server:  10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      test-nginx
Address 1: 10.0.0.248 test-nginx.default.svc.cluster.local
```

16.11. Proyecto web con apache

- Para realizar el ejemplo de una aplicación web que se conecta a una base de datos, creamos un pod como base de datos para iniciar una sola instancia individual, aunque la recomendación siempre es la de creación de un recurso de tipo deployment

pod-mysql.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: db
  labels:
    name: db
spec:
  containers:
  - name: db
    image: mysql:5.7
    ports:
    - containerPort: 3306
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
```

- Para desplegar el pod

```
$ kubectl create -f pod-mysql.yml
```

- Para que esté accesible a otros pods

```
kubectl expose pod db --type=NodePort
```

- Al exponer el servicio, podemos observar la dirección de acceso, aunque desde cualquier otro pod se puede acceder con el nombre del pod

```
$ kubectl get svc
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
db           10.0.0.81    <nodes>       3306:32116/TCP  2h
hello-minikube 10.0.0.171  <nodes>       8080:30051/TCP  4h
kubernetes   10.0.0.1     <none>        443/TCP       5h
kubernetes-bootcamp 10.0.0.202  <nodes>       8080:32044/TCP  4h
test-nginx   10.0.0.110   <pending>    80:30379/TCP   3h
```

- En este caso, se necesita compartir una unidad de disco con el fichero index.php para su publicación.
- Para ello usaremos como ejemplo un volumen persistente y un objeto de tipo claim

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: www-data
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/home/docker/web/www"
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: www-data-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- De esta manera, quedan asociados y se publican para que las aplicaciones puedan usarlo como unidad compartida.

```
$ kubectl create -f persistent-volume-web.yml
```

```
$ kubectl create -f persistent-volume-claim-web.yml
```

- Podemos desplegar la aplicación web php que se conecta con la base de dato como un pod individual

```

apiVersion: v1
kind: Pod
metadata:
  name: web
spec:

  volumes:
    - name: www-data
      persistentVolumeClaim:
        claimName: www-data-claim

  containers:
    - name: web
      image: app-web-php:5.6
      ports:
        - containerPort: 80
      volumeMounts:
        - mountPath: "/var/www/html"
          name: www-data

```

- Desplegamos de la misma manera

```
$ kubectl create -f pod-web.yml
```

- Ahora, si probamos el pod accediendo y consultando a la capa web:

```

$ kubectl exec -it web bash
root@web:/var/www/html# curl localhost
Conexión realizada correctamente. Hurra!!

```

- Si ahora realizamos la operación por medio de un objeto de tipo deployment, para crear dos instancias del mismo cliente web

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: web-balancer
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: web-balancer
  spec:

    volumes:
      - name: www-data
        persistentVolumeClaim:
          claimName: www-data-claim
    containers:
      - name: web
        image: app-web-php:5.6
        ports:
          - containerPort: 80
        volumeMounts:
          - mountPath: "/var/www/html"
            name: www-data

```

- desplegamos el recurso que depende de ellos para poder montar las unidades en las imágenes.

```
$ kubectl create -f deployment-web.yml
```

- Exponemos el despliegue como NodePort

```
$ kubectl expose deployment web-balancer --type=NodePort
```

- Comprobamos la existencia del servicio

\$ kubectl get svc				
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
db	10.0.0.81	<nodes>	3306:32116/TCP	2h
hello-minikube	10.0.0.171	<nodes>	8080:30051/TCP	5h
kubernetes	10.0.0.1	<none>	443/TCP	5h
kubernetes-bootcamp	10.0.0.202	<nodes>	8080:32044/TCP	5h
test-nginx	10.0.0.110	<pending>	80:30379/TCP	3h
web-balancer	10.0.0.189	<nodes>	80:30758/TCP	2h

- Ahora podemos publicar el servicio y consultararlo desde el exterior

```
$ minikube service web-balancer --url  
http://192.168.99.100:30758
```

- Si consultamos el servicio, comprobamos que resuelve sin problemas e internamente consulta el pod del servicio mysql. *

Capítulo 17. Imágenes Docker Avanzadas

En el mundo de las imágenes docker, si le damos alguna que otra vuelta más a la tecnología, podemos llegar a conseguir efectos espectaculares.

Docker en principio, parece una tecnología puramente de servidor, sin interfáz gráfica alguna y que a priori, parece que únicamente vale para levantar API Rest :D

Sin embargo, podemos utilizar la magia de docker, para levantar incluso aplicaciones de escritorio con su interfáz gráfica completa en un contenedor, siguiendo el principio de instalación 0 y con todo lo necesario para que operen :)

17.1. Lab: Imágenes Avanzadas

Mediante este laboratorio, vamos a llevar a cabo demostraciones con imágenes que van más allá, levantando aplicaciones de escritorio de forma dockerizada.

El uso de entornos gráficos de Docker es posible.

Existen dos posibilidades:

- Creación de un servicio VNC
- Exportación de las X para su uso en equipos con X instaladas

17.1.1. Soporte de clientes en sistema operativo

- Para poder visualizar las X directamente, debemos instalar las X en cliente:
 - Clientes Windows: Cualquier cliente para windows, como por ejemplo XMing.

<https://sourceforge.net/projects/xming/>

- Clientes Linux: Todos los clientes con entorno gráfico usan las X, pero debemos agregar a los usuarios a la lista de acceso para poder ejecutarlos.

```
$ xhost local:docker@  
non-network local connections being added to access control list
```

- Cliente Mac: Es necesario instalar QuartzX para poseer soporte de X, ya que el cliente gráfico es privativo y no lo permite

17.1.2. Pruebas de entorno gráfico

- Se pueden lanzar múltiples recursos en entorno gráfico, pero no todo es compatible. Se han detectado algunos errores en ciertos productos que impiden su ejecución a nivel de X remota. Esto no tiene nada que ver con Docker.

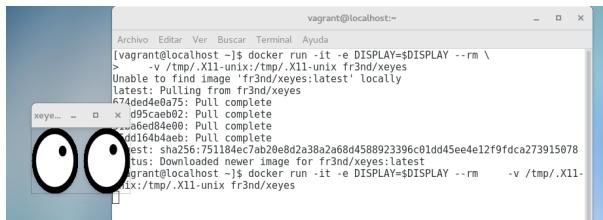
17.1.3. Ejecución Linux XEyes

Ahora, vamos a lanzar unos XEyes, que son una imagen de dos ojos muy común en Linux.

```
$ docker run -it -e DISPLAY=$DISPLAY --rm \
>     -v /tmp/.X11-unix:/tmp/.X11-unix fr3nd/xeyes

Unable to find image 'fr3nd/xeyes:latest' locally
latest: Pulling from fr3nd/xeyes
674ded4e0a75: Pull complete
a3ed95caeb02: Pull complete
51ba6ed84e00: Pull complete
a5dd164b4aeb: Pull complete
Digest: sha256:751184ec7ab20e8d2a38a2a68d4588923396c01dd45ee4e12f9fdca273915078
Status: Downloaded newer image for fr3nd/xeyes:latest
```

Tendríamos el siguiente resultado:



Para entorno Windows, se debe saber cual es la dirección IP que poseen las X instaladas y usarlo para lanzar el comando:

```
$ docker run -it -e DISPLAY=172.16.182.130:0.0 --rm fr3nd/xeyes
```

Como podemos comprobar, el display apunta a la dirección IP donde se despliegan las X en Windows.

17.1.4. Lanzamiento del IDE de desarrollo Eclipse

Otra herramienta gráfica que podemos lanzar, sería el IDE de desarrollo Eclipse.

En un entorno Linux ejecutamos los siguiente comandos:

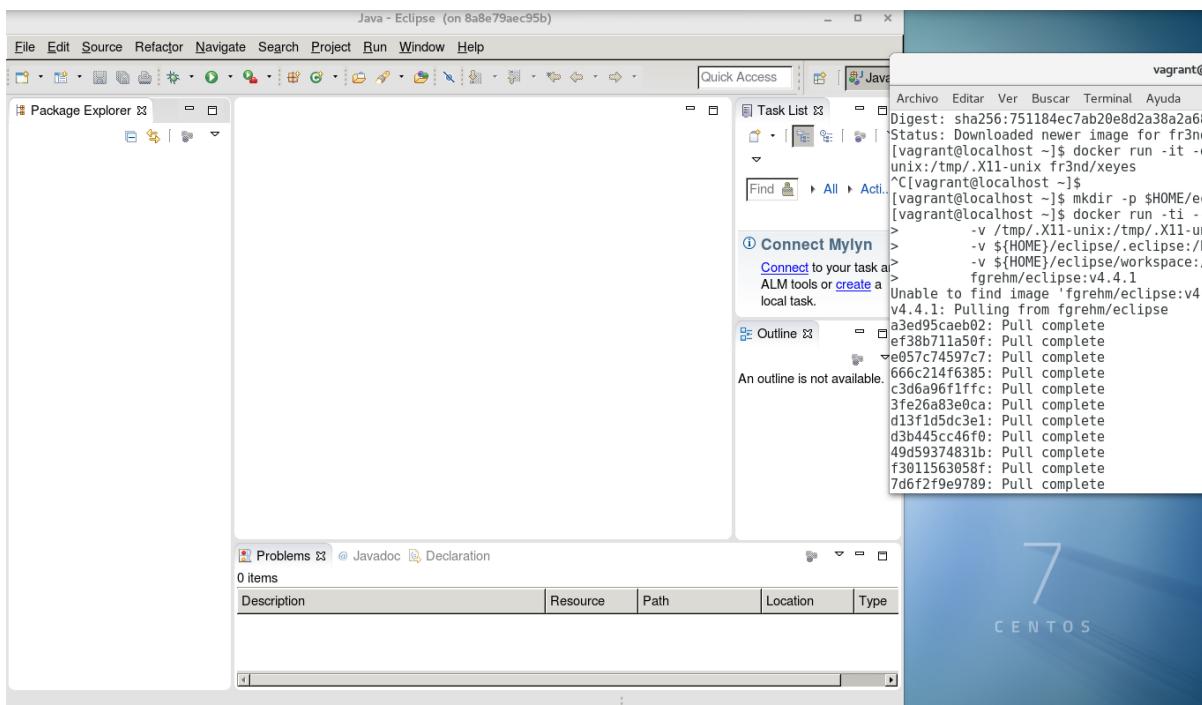
```

$ mkdir .eclipse
$ mkdir workspace
$ docker run -ti --rm -e DISPLAY=$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -v ${HOME}/eclipse/.eclipse:/home/developer \
    -v ${HOME}/eclipse/workspace:/workspace \
    --ulimit nofile=65536:65536 -u root
fgrehm/eclipse:v4.4.1

Unable to find image 'fgrehm/eclipse:v4.4.1' locally
v4.4.1: Pulling from fgrehm/eclipse
a3ed95caeb02: Pull complete
ef38b711a50f: Pull complete
e057c74597c7: Pull complete
666c214f6385: Pull complete
c3d6a96f1ffc: Pull complete
3fe26a83e0ca: Pull complete
d13f1d5dc3e1: Pull complete
d3b445cc46f0: Pull complete
49d59374831b: Pull complete
f3011563058f: Pull complete
7d6f2f9e9789: Pull complete
Digest: sha256:a025bef2fa0c7c1e71206ad6566ce9bae7927d7037c7be50209ac7e25e6d2e6e
Status: Downloaded newer image for fgrehm/eclipse:v4.4.1
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0

```

Como resultado, obtenemos la ejecución del IDE Eclipse en un entorno gráfico remoto.



En un entorno Windows sería más sencillo de lanzar:

```
$ mkdir .eclipse  
$ mkdir workspace  
$ docker run -ti --rm -e DISPLAY=172.16.182.133:0.0 \  
-v $HOME/eclipse/.eclipse:/home/developer  
-v $HOME/eclipse/workspace:/workspace fgrehm/eclipse:v4.4.1
```



En ambos casos, se puede cambiar la ruta del workspace por la de nuestro proyecto, y además, el directorio **.eclipse** se almacena para poder guardar los plugins de Eclipse y que sobrevivan tras las instalaciones y posterior eliminación del contenedor.

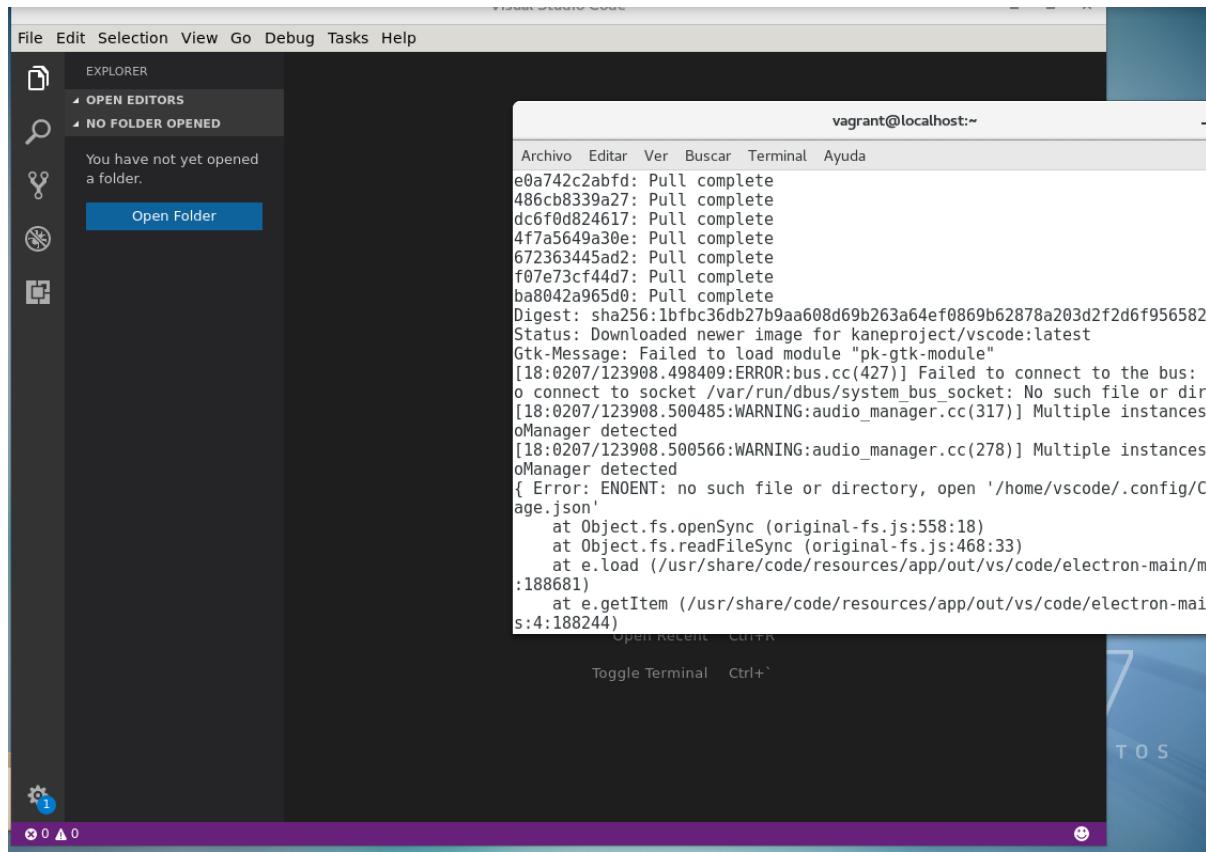
17.1.5. Lanzamiento del IDE de desarrollo VSCode

Visual Studio Code es una herramienta muy conocida para los desarrolladores de JavaScript.

Ejecutamos el siguiente comando para lanzar la aplicación en un entorno Linux:

```
$ docker run -it --user $(id -u) --network=host -e DISPLAY=$DISPLAY \  
> -v /tmp/.X11-unix:/tmp/.X11-unix -v $HOME/vscode:/home/vscode \  
> --privileged kaneproject/vscode:latest  
  
Unable to find image 'kaneproject/vscode:latest' locally  
latest: Pulling from kaneproject/vscode  
e0a742c2abfd: Pull complete  
486cb8339a27: Pull complete  
dc6f0d824617: Pull complete  
4f7a5649a30e: Pull complete  
672363445ad2: Pull complete  
f07e73cf44d7: Pull complete  
ba8042a965d0: Pull complete  
Digest: sha256:1bfbc36db27b9aa608d69b263a64ef0869b62878a203d2f2d6f9565826e4bc7b  
Status: Downloaded newer image for kaneproject/vscode:latest  
Gtk-Message: Failed to load module "pk-gtk-module"  
[18:0207/123908.498409:ERROR:bus.cc(427)] Failed to connect to the bus: Failed to connect to socket  
/var/run/dbus/system_bus_socket: No such file or directory  
[18:0207/123908.500485:WARNING:audio_manager.cc(317)] Multiple instances of AudioManager detected
```

Tendríamos el siguiente resultado:



17.1.6. Conclusiones

La verdadera dificultad se encuentra en coordinar los puertos de distintos contenedores.

Si no podemos coordinarlos, se pueden lanzar distintos comandos en el mismo contenedor, con lo cual podríamos desarrollar de forma sencilla, pero debemos recordar que los puntos de montaje en Windows no están perfeccionados todavía, y que si modificamos un contenedor para acercarlo a nuestro desarrollo, debemos crear un Dockerfile para poder repetir la instalación o alterarla.

Si queremos gestionar un entorno completo, podemos iniciar un servicio de tipo **docker-compose** para orquestar todos los servicios incluyendo el IDE.

Antes de dockerizar todo, debemos tener en cuenta que el entorno gráfico nos puede dar una mala experiencia de uso, no todo tiene porqué estar dockerizado.

Capítulo 18. Desarrollo con Docker

- Hemos aprendido a crear imágenes, lanzar y trabajar con contenedores
- Vamos a usar docker para
 - Probar un website estático
 - Construir y probar una aplicación web
 - Usar Docker para integración continua
- Los dos primeros casos se orientarán a desarrollo y pruebas individuales
- La tercera servirá como pruebas para un ciclo de vida de múltiples desarrolladores

18.1. WebSite estático

- Para un ejemplo de web estático, creamos un Dockerfile

Dockerfile

```
FROM ubuntu:14.04
MAINTAINER Ruben Gomez "rgomez@pronoide.es"
ENV REFRESHED_AT 2016-09-01
RUN apt-get update
RUN apt-get -y -q install nginx
RUN mkdir -p /var/www/html
ADD nginx/global.conf /etc/nginx/conf.d/
ADD nginx/nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

- Instala Nginx
- Crea un directorio para el website
- Agrega la configuración de Nginx
- Expone el puerto 80
- En la configuración obligamos a que el proceso se quede en foreground
- Construimos la imagen con los ficheros de apoyo

```
$ docker build -t cursodocker/nginx .
```

- Podemos observar todas las operaciones de la imagen realizadas

```
$ docker history cursodocker/nginx
```

- Desde el directorio donde está el Dockerfile, creamos directorios y fichero web

contenido del fichero website/index.html

```
<html><body><marquee><strong>Web Nginx</strong></marquee></body></html>
```

- Ahora conectamos el contenedor con una unidad externa

```
$ docker run -d -p 80 --name website -v $PWD/website:/var/www/html/website  
cursodocker/nginx nginx
```

- Los volúmenes pueden compartirse entre unidades y montarse en modo lectura/escritura o solo lectura
- El directorio destino es creado por docker si es necesario

```
$ docker run -d -p 80 --name website -v $PWD/website:/var/www/html/website:ro  
cursodocker/nginx nginx
```

- Ahora ya podemos ver el website desde nuestra máquina local buscando en localhost en el puerto externo compartido
- De hecho, ahora podemos modificar las páginas web y se verá el contenido nuevo sin necesidad de recargar

18.2. Aplicación Web

- En este caso , vamos a probar una aplicación web más compleja
- Aplicación basada en Sinatra
- Para ello, creamos otro contenedor con el siguiente Dockerfile

```
FROM ubuntu:16.04  
MAINTAINER Ruben Gomez "rgomez@pronoide.es"  
ENV REFRESHED_AT 2016-07-11  
  
RUN apt-get -yqq update && apt-get -yqq install ruby ruby-dev build-essential redis-tools  
RUN gem install --no-rdoc --no-ri sinatra json redis  
  
RUN mkdir -p /opt/webapp  
  
EXPOSE 4567  
  
CMD [ "/opt/webapp/bin/webapp" ]
```

NOTE

Para realizar este ejercicio, es necesario descargarse el workspace y alojarse en el directorio

13_testing_webapp_link

- Se ha creado una imagen con ruby, con herramientas de Redis, Sinatra y Json
- Se expone el puerto por defecto del servidor WEBrick
- Se ejecuta el comando webapp
- Creamos la imagen

```
$ docker build -t cursodocker/sinatra .
```

- Tras descargar el código de aplicación en el directorio webapp, creamos el contenedor

```
$ docker run -d -p 4567:4567 --name basewebapp -v $PWD/webapp:/opt/webapp  
cursodocker/sinatra
```

- Podemos ver que el proceso está ejecutándose correctamente

```
$ docker logs basewebapp  
[2017-01-13 15:28:36] INFO WEBrick 1.3.1  
[2017-01-13 15:28:36] INFO ruby 2.3.1 (2016-04-26) [x86_64-linux-gnu]  
-- Sinatra (v1.4.7) has taken the stage on 4567 for development with backup from WEBrick  
[2017-01-13 15:28:36] INFO WEBrick::HTTPServer#start: pid=1 port=4567  
$ docker top basewebapp  
UID          PID      PPID      C      STIME      TTY  
TIME          CMD  
root        20077    20062      0      16:28      ?  
0:00:00      /usr/bin/ruby /opt/webapp/bin/webapp
```

- Podemos mostrar los puertos compartidos por el contenedor

```
$ docker port basewebapp 4567
```

- Para probar que la web es correcta, lanzamos el siguiente comando

```
curl -i -H 'Accept: application/json' -d 'nombre=Ruben&apellido=Gomez' http://localhost:4567/json  
{ "nombre": "Ruben", "apellido": "Gomez"}
```

- Vamos a ampliar la imagen para que use otro contenedor
- Para ello, vamos a comunicar las aplicaciones por medio de un link o enlace.
- La imagen de Redis es la siguiente

```

FROM ubuntu:16.04
MAINTAINER Ruben Gomez "rgomez@pronoide.es"
ENV REFRESHED_AT 2016-09-01
RUN apt-get update
RUN apt-get -y install redis-server redis-tools
EXPOSE 6379
ENTRYPOINT ["/usr/bin/redis-server"]

```

- Construimos la imagen y la ejecutamos
- Observamos que no publicamos los puertos en ningún momento pero están expuestos

```

$ docker build -t cursodocker/redis -f Dockerfile-redis .
$ docker run -d --name redis cursodocker/redis

```

- Para conectar al contenedor de Redis, no podemos aprovechar las direcciones ip
- Estas direcciones cambian según el numero de contenedores existentes.
- La mejor forma es por medio de linkado
- Para establecer el linkado, es necesario saber el nombre del contenedor
- Iniciamos un nuevo contenedor usando la plantilla redis de la aplicación web

```

docker run -p 4567 --name webapp --link redis:db -t -i -v $PWD/webapp-
redis:/opt/webapp cursodocker/sinatra /bin/bash

```

- Accediendo al prompt de la shell podemos ver que el argumento link permite
 - Establecer una relación padre-hijos
 - La relación de redis es db, en las máquinas aparecerá un recurso db para acceder a la dirección de redis
 - No es necesario exponer el puerto fuera del contenedor, solo lo ve las máquinas enlazadas
- Se puede configurar el demonio docker, para que no se admitan comunicaciones entre aplicaciones no enlazadas
- Podemos enlazar tantos contenedores como queramos

```

docker run -p 4567 --name webapp2 --link redis:db -t -i -v $PWD/webapp-
redis:/opt/webapp cursodocker/sinatra
docker run -p 4567 --name webapp3 --link redis:db -t -i -v $PWD/webapp-
redis:/opt/webapp cursodocker/sinatra

```

- podemos observar los cambios en dos puntos
- /etc/hosts donde aparece la referencia al enlace
- Podemos hacer ping a db

- Podemos comprobar las variables de entorno con el comando env



Las imágenes por defecto tienen el software restringido. Por eso se debe instalar lo estrictamente necesario apt-get install iutils-ping

- Si observamos ahora y realizamos la petición, esta se realizará desde redis

```
curl -i -H 'Accept: application/json' -d 'nombre=Ruben&apellido=Gomez'  
http://localhost:32768/json  
{"nombre":"Ruben", "apellido": "Gomez"}
```

Capítulo 19. Weblogic workshop

- Aquí se referencia la documentación para la dockerización desde el repositorio oficial de Oracle en github

19.1. Preparación del entorno

- Para poder trabajar con la documentación oficial, descargaremos del repositorio el proyecto de weblogic

<https://github.com/oracle/docker-images>

```
git clone https://github.com/oracle/docker-images
```

- Para ello, debemos construir la imagen base JRE

<http://download.oracle.com/otn/java/jdk/8u101-b13/server-jre-8u101-linux-x64.tar.gz>

- Tras alojarlo en el directorio docker-images/OracleJava/java-8 (es un requerimiento)
- Creamos la imagen de java-8 desde el script

```
$ ./build.sh
```

- Esto creará la siguiente imagen

\$ docker images					
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE	
oracle/serverjre	8	7f5371331f31	4 minutes ago	382.2 MB	

- Si observamos la imagen, se obtiene a partir de la última versión de oraclelinux. Si queremos estabilizar la versión, se debería generar con una versión concreta.
- Para generar las imágenes de instalación de la imagen de weblogic, debemos descargar weblogic a la carpeta docker-images/OracleWebLogic/dockerfiles/12.2.1
- El aspecto debe ser el siguiente:

```
$ ls
Checksum.developer      fmw_12.2.1.0.0_infrastructure_Disk1_1of1.zip
Checksum.generic        fmw_12.2.1.0.0_wls_Disk1_1of1.zip
Checksum.infrastructure fmw_12.2.1.0.0_wls_quick_Disk1_1of1.zip
Dockerfile.developer    install.file
Dockerfile.generic      oraInst.loc
Dockerfile.infrastructure
```

19.2. Generación de un dominio básico:

- La generación del dominio se genera a partir de un script generado llamado *buildDockerImage.sh*

```
$ ./buildDockerImage.sh -h
```

```
Usage: buildDockerImage.sh -v [version] [-d | -g | -i] [-s] [-c]
Builds a Docker Image for Oracle WebLogic.
```

Parameters:

- v: version to build. Required.
Choose one of: 12.1.3 12.2.1 12.2.1.1 12.2.1.2
- d: creates image based on 'developer' distribution
- g: creates image based on 'generic' distribution
- i: creates image based on 'infrastructure' distribution
- c: enables Docker image layer cache during build
- s: skips the MD5 check of packages

* select one distribution only: -d, -g, or -i

LICENSE CDDL 1.0 + GPL 2.0

Copyright (c) 2014-2015 Oracle and/or its affiliates. All rights reserved.

- Las imágenes que se generan a partir de esta creación, no poseen un dominio preconfigurado.
- Para ello, se debe extender la imagen con un nuevo Dockerfile y crear el dominio por medio del lenguaje *WLST*
- Construimos la versión developer de weblogic:

```
$ ./buildDockerImage.sh -v 12.2.1 -d
```

- En la construcción se utiliza el siguiente Dockerfile de desarrollo

Dockerfile.developer

```
# LICENSE CDDL 1.0 + GPL 2.0
#
# Copyright (c) 2014-2015 Oracle and/or its affiliates. All rights reserved.
#
# ORACLE DOCKERFILES PROJECT
# -----
#
# This is the Dockerfile for WebLogic 12.2.1 Quick Install Distro
#
# REQUIRED FILES TO BUILD THIS IMAGE
# -----
#
# (1) fmw_12.2.1.0.0_wls_quick_Disk1_1of1.zip
#     Download the Developer Quick installer from http://www.oracle.com/technetwork/middleware/weblogic/downloads/wls-for-dev-1703574.html
#
# (2) server-jre-8uXX-linux-x64.tar.gz
```

```

# Download from http://www.oracle.com/technetwork/java/javase/downloads/server-jre8-downloads-2133154.html
#
# HOW TO BUILD THIS IMAGE
# -----
# Put all downloaded files in the same directory as this Dockerfile
# Run:
#     $ docker build -t oracle/weblogic:12.2.1-developer .
#
# IMPORTANT
# -----
# The resulting image of this Dockerfile DOES NOT contain a WLS Domain.
# For that, look into the folder 'samples' for an example on how
# to create a domain on a new inherited image.
#
# You can go into 'samples/1221-domain' after building the developer install image
# and build the domain image, for example:
#
#   $ cd samples/1221-domain
#   $ docker build -t mywls .
#
# Pull base image
# -----
FROM oracle/serverjre:8

# Maintainer
# -----
MAINTAINER Bruno Borges <bruno.borges@oracle.com>

# Environment variables required for this build (do NOT change)
# -----
ENV FMW_PKG=fmw_12.2.1.0.0_wls_quick_Disk1_1of1.zip \
    FMW_JAR=fmw_12.2.1.0.0_wls_quick.jar \
    ORACLE_HOME=/u01/oracle \
    USER_MEM_ARGS="-Djava.security.egd=file:/dev/.urandom" \
    DEBUG_FLAG=true \
    PRODUCTION_MODE=dev \
    PATH=$PATH:/usr/java/default/bin:/u01/oracle/oracle_common/common/bin

# Copy packages
# -----
COPY $FMW_PKG install.file oraInst.loc /u01/

# Setup filesystem and oracle user
# Install and configure Oracle JDK
# Adjust file permissions, go to /u01 as user 'oracle' to proceed with WLS installation
# -----
RUN chmod a+xr /u01 && \
    useradd -b /u01 -m -s /bin/bash oracle && \
    echo oracle:oracle | chpasswd && \
    cd /u01 && $JAVA_HOME/bin/jar xf /u01/$FMW_PKG && cd - && \
    su -c "$JAVA_HOME/bin/java -jar /u01/$FMW_JAR -invPtrLoc /u01/oraInst.loc -jreLoc $JAVA_HOME -ignoreSysPrereqs -force \
    -novalidation ORACLE_HOME=$ORACLE_HOME" - oracle && \
    chown oracle:oracle -R /u01 && \
    rm /u01/$FMW_JAR /u01/$FMW_PKG /u01/oraInst.loc /u01/install.file

USER oracle
WORKDIR $ORACLE_HOME

# Define default command to start bash.
CMD ["bash"]

```

- Para construir un dominio personalizado para desarrollo, poseemos el siguiente ejemplo en el directorio sample alojado en: docker-images/OracleWebLogic/samples/1221-domain

```
$ docker build -t 1221-domain --build-arg ADMIN_PASSWORD=welcome1 .
```

- Comprobemos el contenido del fichero docker:

dockerfile

```
# LICENSE CDDL 1.0 + GPL 2.0
#
# Copyright (c) 2014-2015 Oracle and/or its affiliates. All rights reserved.
#
# ORACLE DOCKERFILES PROJECT
# -----
#
# This Dockerfile extends the Oracle WebLogic image by creating a sample domain.
#
# Util scripts are copied into the image enabling users to plug NodeManager
# automatically into the AdminServer running on another container.
#
# HOW TO BUILD THIS IMAGE
# -----
#
# Put all downloaded files in the same directory as this Dockerfile
# Run:
#      $ sudo docker build -t 1221-domain --build-arg ADMIN_PASSWORD=welcome1 .
#
# Pull base image
# -----
FROM oracle/weblogic:12.2.1-developer

# Maintainer
# -----
MAINTAINER Bruno Borges <bruno.borges@oracle.com>

# WLS Configuration (editable during build time)
# -----
ARG ADMIN_PASSWORD
ARG DOMAIN_NAME
ARG ADMIN_PORT
ARG CLUSTER_NAME
ARG DEBUG_FLAG
ARG PRODUCTION_MODE

# WLS Configuration (editable during runtime)
# -----
ENV ADMIN_HOST="wlsadmin" \
    NM_PORT="5556" \
    MS_PORT="7001" \
    DEBUG_PORT="8453" \
    CONFIG_JVM_ARGS="-Dweblogic.security.SSL.ignoreHostnameVerification=true"

# WLS Configuration (persisted. do not change during runtime)
# -----
```

```

ENV DOMAIN_NAME="${DOMAIN_NAME:-base_domain}" \
    DOMAIN_HOME=/u01/oracle/user_projects/domains/${DOMAIN_NAME:-base_domain} \
    ADMIN_PORT="${ADMIN_PORT:-8001}" \
    CLUSTER_NAME="${CLUSTER_NAME:-DockerCluster}" \
    debugFlag="${DEBUG_FLAG:-false}" \
    PRODUCTION_MODE="${PRODUCTION_MODE:-prod}" \
    PATH=$PATH:/u01/oracle/oracle_common/common/bin:/u01/oracle/wlserver/common/bin:/u01/oracle/user_projects/domains/${DOMAIN_NAME:-base_domain}/bin:/u01/oracle

# Add files required to build this image
USER oracle
COPY container-scripts/* /u01/oracle/

# Configuration of WLS Domain
RUN /u01/oracle/wlst /u01/oracle/create-wls-domain.py && \
    mkdir -p \
    /u01/oracle/user_projects/domains/$DOMAIN_NAME/servers/AdminServer/security && \
    echo "username=weblogic" > \
    /u01/oracle/user_projects/domains/$DOMAIN_NAME/servers/AdminServer/security/boot.properties && \
    echo "password=$ADMIN_PASSWORD" >> \
    /u01/oracle/user_projects/domains/$DOMAIN_NAME/servers/AdminServer/security/boot.properties && \
    echo ". /u01/oracle/user_projects/domains/$DOMAIN_NAME/bin/setDomainEnv.sh" >> \
    /u01/oracle/.bashrc

# Expose Node Manager default port, and also default for admin and managed server
EXPOSE $NM_PORT $ADMIN_PORT $MS_PORT $DEBUG_PORT

WORKDIR $DOMAIN_HOME

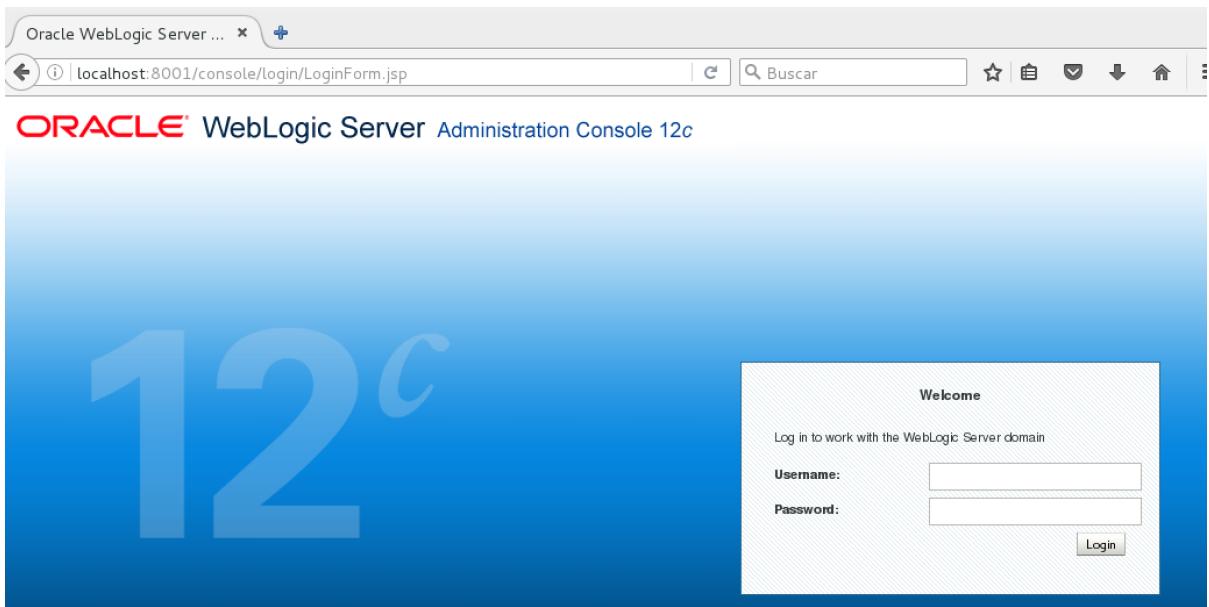
# Define default command to start bash.
CMD ["startWebLogic.sh"]

```

Para iniciar el AdminServer:

```
docker run -d --name=wlsadmin -p 8001:8001 1221-domain
```

- Podemos visitar el Adminserver en : <http://localhost:8001/console> con el usuario *weblogic* y *welcome1*



- Podemos observar que el AdminServer es el único servidor en el dominio

Servers (Filtered - More Columns Exist)							
	New	Clone	Delete	Showing 1 to 1 of 1 Previous Next			
	Name	Type	Cluster	Machine	State	Health	Listen Port
<input type="checkbox"/>	AdminServer(admin)	Configured			RUNNING	✓ OK	8001

19.3. Clustering en un host

- Podemos desplegar un cluster de Weblogic con Docker.
- Tras definir e iniciar el AdminServer, podemos definir nuevos contenedores de Docker como servidores manejados.

```
$ docker run -d --link wlsadmin:wlsadmin 1221-domain createServer.sh
```

- De esta forma, se crea un NodeManager dentro del contenedor y un ManagerServer, registrándolos en el AdminServer. Para ver el resultado, debemos esperar un tiempo a que aparezca el nuevo servidor
- Podemos crear tantos contenedores como deseemos:

Servers (Filtered - More Columns Exist)							
	New	Clone	Delete	Showing 1 to 3 of 3 Previous Next			
	Name	Type	Cluster	Machine	State	Health	Listen Port
<input type="checkbox"/>	AdminServer(admin)	Configured			RUNNING	✓ OK	8001
<input type="checkbox"/>	ManagedServer-3OqKZR@36e8e867b678	Configured	DockerCluster	Machine-36e8e867b678	RUNNING	✓ OK	7001
<input type="checkbox"/>	ManagedServer-e3Pfg5@2c6194b01591	Configured	DockerCluster	Machine-2c6194b01591	RUNNING	✓ OK	7001

19.4. Clustering en múltiples hosts

- En este caso, se va a generar un entorno de pruebas con tres máquinas virtuales.
 - La primera se usará de registro local para almacenar imágenes
 - La segunda almacenará el servidor principal con un AdminServer y el webtier
 - La tercera almacenará un contenedor de weblogic manejado
- Para realizar el laboratorio debemos generar las imágenes que vamos a utilizar

19.4.1. AppDeploy

- Construimos la imagen 1221-appdeploy

```
$ docker build -t 1221-appdeploy .
```

19.4.2. WebTier

- El plugin de Apache provee de balanceo de caraga a los servidores manejados en el cluster de Weblogic.
- Construimos la capa web alojada en 1221-webtier-apache

```
$ docker build -t 1221-webtier .
```

Dockerfile

```
# Example of Apache Web Server with WebLogic plugin for load balancing WebLogic on Docker Containers
#
# Copyright (c) 2015 Oracle and/or its affiliates. All rights reserved.
#
# Author: Bruno Borges <bruno.borges@oracle.com>
#
FROM httpd:2.4

ENV PLUGIN_PKG="WLSPlugin12.2.1-Apache2.2-Apache2.4-Linux_x86_64-12.2.1.0.0.zip" \
    PLUGIN_HOME="/root" \
    MOD_WLS_PLUGIN="mod_wl_24.so" \
    LD_LIBRARY_PATH="/root/lib" \
    WEBLOGIC_CLUSTER="server0:7001,server1:7001"

COPY $PLUGIN_PKG weblogic.conf /root/

RUN apt-get update && apt-get install -y unzip && \
    unzip /root/$PLUGIN_PKG -d $PLUGIN_HOME && \
    cat /root/weblogic.conf >> /usr/local/apache2/conf/httpd.conf && \
    rm /root/$PLUGIN_PKG /root/weblogic.conf
```

19.5. Construcción del entorno multihost

- Para construir el entorno multihost, debemos tener el entorno correcto:
 - **Docker Machine:** Herramienta que instala el motor de docker en hosts virtuales y los gestiona por medio de comandos docker-machine. Creará imágenes en VirtualBox usando la imagen boot2docker
 - **Docker swarm:** El sistema de clustering nativo de docker
 - **Docker Overlay Network:** Soporte de redes multihost
 - **Docker compose:** Definiciones de aplicaciones Docker multi-contenedor.
 - **Docker registry:** Servidor escalable que almacena y distribuye las imágenes Docker
 - **Consul:** Autorregistro y descubrimiento por DNS o HTTP
- EL directorio 1221-multihost posee el fichero bootstrap.sh
- El script inicia dos maquinas de VirtualBox, **weblogic-orchestator** y **weblogic-master**
 - **weblogic-orchestator** posee el Docker Registry en ejecución y **Consul** para iniciar servicios
 - **weblogic-master** Posee el Docker Swarm, la red Overlay y el AdminServer ejecutándose en el contenedor
- Prerequisitos de la máquina:
 - Debe ser un Linux en máquina host con VirtualBox instalado
 - Podemos usar Virtualización bajo virtualización, pero no es compatible con VirtualBox. Sí lo es con VMWare. Una solución es una VM de VMWare Linux y una instalación de VirtualBox dentro de la VM.
 - Se tiene que activar la característica de permitir VT / VT-x en la máquina virtual.
- Para iniciar la instalación del entorno, nos dirigimos al directorio docker-images/OracleWeblogic/samples/1221-multihost y ejecutamos el script ./bootstrap.sh

```
$ ./bootstrap.sh
```

- Al terminar la instalación aparecen las dos vm
 - Weblogic master.
 - Weblogic Orchestrator.
- Si accedemos al *weblogic-master* podemos observar las imágenes cargadas y la red overlay

```

$ docker-machine ssh weblogic-master
$ docker network ls
NETWORK ID      NAME        DRIVER      SCOPE
e4799c56b316   bridge      bridge      local
62f7cd479972   docker_gwbridge  bridge      local
aacb349898cb   host        host        local
bfa5a55dbd24   none        null       local
8e692cea67dd   weblogic-net  overlay    global
$ exit

```

- Al terminar podemos acceder a la consola. La ruta aparece al final de la instalación
- AppServer crea un AdminServer con una aplicación llamada sample

19.6. Agregación de un ManagedServer

- Ejecutamos el script create-machine.sh
- Permite construir una nueva máquina agregada a Swarm

```

./create-machine.sh
Creating new Docker Machine weblogic-marvel ...
Running pre-create checks...
Creating machine...
(weblogic-marvel) Copying /Users/kane_project/.docker/machine/cache/boot2docker.iso to
/Users/kane_project/.docker/machine/machines/weblogic-marvel/boot2docker.iso...
(weblogic-marvel) Creating VirtualBox VM...
(weblogic-marvel) Creating SSH key...
(weblogic-marvel) Starting the VM...
(weblogic-marvel) Check network to re-create if needed...
(weblogic-marvel) 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...
Configuring swarm...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-machine env
weblogic-marvel

Machine weblogic-marvel successfully created.
Deploy containers on this machine by calling:

```

 \$./create-container.sh weblogic-marvel

Or just call the script above without arguments, and a new container will be created in the Swarm.

- Ejecutamos create-container <nombre_machine> sustituyendo el nombre_machine por el nombre de máquina creada

```

./create-container.sh weblogic-marvel
Creating container instance weblogic-instance-9C22E246 on specific Docker Machine weblogic-marvel ...
Unable to find image '192.168.99.108:5000/1221-appdeploy:latest' locally
latest: Pulling from 1221-appdeploy
2c48edfee2a0: Pull complete
111e88d4e055: Pull complete
b58e7be02758: Pull complete
02f125d8a3a5: Pull complete
0ee8b50b2fb6: Pull complete
9fa38a2b3abd: Pull complete
0eea126244d5: Pull complete
ef35b8bf4f8d: Pull complete
9d026fbcd51c: Pull complete
Digest: sha256:fb5fa8a96e69ca95a81dc7db5c8ff4e4c8cd4f2440a1f426432036245b3445c7
Status: Downloaded newer image for 192.168.99.108:5000/1221-appdeploy:latest
7bf4a29dbeed520573711fce8e8870851128496d4c3bc16f654dd7a949e6e0b

```

19.7. Inicio del WebTier

- Para iniciar el WebTier, solo necesitamos lanzar el script start-webtier.sh
- Se instalará en el weblogic-master

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
74c948e40369	192.168.99.108:5000/1221-webtier	"httpd-foreground"	35 minutes ago	Up 35 minutes
0.0.0.0:80->80/tcp				

- Para acceder, podemos observar la url tras la finalización del script

19.8. Otros contenidos del repositorio

- Podemos ver otros ejemplos utilizados para desplegar soluciones en Docker:
- XXXX-docker-compose: Un ejemplo de docker compose que permite gestionar managed servers y nodemanagers
- XXXX-domain-with-resource: Es un ejemplo de dominio con recursos usando scripting WLST offline