INTRODUCCIÓN A DOCKER

Docker es un proyecto open source que ha revolucionado la manera de desarrollar software gracias a la sencillez con la que permite gestionar contenedores. Los contenedores LXC (LinuX Containers) son un concepto relativamente antiguo y utilizado desde hace tiempo por grandes empresas como Amazon o Google, pero cuya gestión era complicada. Sin embargo, Docker define APIs y herramientas de línea de comandos que hacen casi trivial la creación, distribución y ejecución de contenedores. De ahí que el lema de Docker sea: "Build, Ship and Run. Any application, Anywhere" y se haya convertido en una herramienta fundamental tanto para desarrolladores como para administradores de sistemas.

Podríamos definir un contenedor Docker como una máquina virtual ligera, que corre sobre un sistema operativo Linux pero con su propio sistema de ficheros, su propio espacio de usuarios y procesos, sus propias interfaces de red… por lo que se dice que son sistemas aislados. Las características principales de Docker son su portabilidad, su inmutabilidad y su ligereza:

Portabilidad

Un contenedor Docker es ejecutado por lo que se denomina el Docker Engine, un demonio que es fácilmente instalable en prácticamente todas las distribuciones Linux. Un contenedor ejecuta una imagen de docker, que es una representación del sistema de ficheros y otros metadatos que el contenedor va a utilizar para su ejecución. Una vez que hemos generado una imagen de Docker, ya sea en nuestro ordenador o vía una herramienta externa, esta imagen podrá ser ejecutada por cualquier Docker Engine, independientemente del sistema operativo y la infraestructura que haya por debajo.

Inmutabilidad

Una aplicación la componen tanto el código fuente como las librerías del sistema operativo y del lenguaje de programación necesarias para la ejecución de dicho código. Estas dependencias dependen a su vez del sistema operativo donde nuestro código va a ser ejecutado, y por esto mismo ocurre muchas veces aquello de que "no sé, en mi máquina funciona". Sin embargo, el proceso de instalación de dependencias en Docker no depende del sistema operativo, si no que este proceso se realiza cuando se genera una imagen de docker. Es decir, una imagen de docker (también llamada repositorio por su parecido con los repositorios de git) contiene tanto el código de la aplicación como las dependencias que necesita para su ejecución. Una imagen se genera una vez y puede ser ejecutada las veces que sean necesarias, y siempre ejecutará con las misma versión del código fuente y sus dependencias, por lo que se dice que es inmutable. Si unimos inmutabilidad con el hecho de que Docker es portable, decimos que Docker es una herramienta fiable, ya que una vez generada una imagen, ésta se comporta de la misma manera independientemente del sistema operativo y de la infraestructura donde se esté ejecutando.

Ligereza

Los contenedores Docker corriendo en la misma máquina comparten entre ellos el sistema operativo, pero cada contenedor es un proceso independiente con su propio sistema de ficheros y su propio espacio de procesos y usuarios (para este fin Docker utiliza cgroups y namespaces, recursos de aislamiento basados en el kernel de Linux). Esto hace que la ejecución de contenedores sea mucho más ligera que otros mecanismos de virtualización.

Comparemos por ejemplo con otra tecnología muy utilizada como es Virtualbox. Virtualbox permite del orden de 4 ó 5 máquinas virtuales en un ordenador convencional, mientras que en el mismo ordenador podremos correr cientos de containers sin mayor problema, además de que su gestión es mucho más sencilla.

CONTENEDORES VS VIRTUALIZACION

Una de las primeras dudas que surguen con Docker es saber diferenciar claramente las ventajas y desventajas que existen entre el uso de contenedores y la virtualización.

Antes de introducirnos en el contexto de las ventajas y desventajas, debemos revisar que es la virtualización y los diferentes tipos que existen para evitar la confusión con los contenedores.

La virtualización consiste en añadir una capa de abstracion a los recursos físicos con el objetivo de mejorar el uso de los recursos del sistema. En el pasado, cada elemnto físico ejecutaba un recurso, Con la introducción de la virutalizacion des posibles crear varios entornos simulados (maquinas virtuales), para diversos recursos. Es decir, en el caso de un servidor, gracias a la virutlizacion, podemos ejecutar varios sistemas dentro del mismo con diferentes entornos. Por ejemplo, para diversos propósitos.

Los diferentes tipos de virtualización:

- **Virtualizacion completa:** la maquina virtual no tiene acceso directo a los recursos físicos y requiere de una capa superior para acceder a ellos.
    - Virtualbox
    - Qemu
    - Hyper-V
    - Vmware ESXI
- **Virtualizacion asistida por hardware:** el el hardware el que facilita la creación de la maquina virtual y controla su estado:
    - Kvm
    - Xen
- **Virtualizacion a nivel de sistema operativo:** aquí incluimos los contenedores.

Es el sistema operativo, y no el hardware el encargado de aislar los recursos y proporcionar las herramientas para crear, manipular o controlar el estado de los contenedores (termino utilizado en lugar de maquinas virtuales).

Contenedores vs VMs

La principal diferencia la encontraremos en sus componentes y la forma en la que se integran en la máquina Host donde corren.

Partes común de ambas tecnologías

Ambas tecnologías se ejecutan en una máquina física que la llamaremos Host. Esta máquina tendrá un Hardware específico y un Sistema Operativo.

Tanto el Hardware como el Sistema operativo es único en la máquina, y sobre él se ejecutarán las aplicaciones y procesos de la máquina.

Por lo tanto

- el Hardware de la máquina y
- el Sistema Operativo de la máquina

es compartido por

- el motor del contenedor o container engine
- el motor de las VMs o hypervisor

Container engine vs Hypervisor

Aquí es donde se presentan las principales diferencias.

- **El Hypervisor**

  Es el responsable de gestionar y ejecutar todas las VMs. Cada una de las VMs debe tener instalado un Sistema Operativo completo. Por lo tanto, dentro de nuestro Host vamos a tener ejecutándose diferentes VMs cada una de ellas con un sistema operativo completo.

  Tendremos que instalar y mantener en cada VM todas las librerías y aplicaciones que necesite el SO y las aplicaciones y servicios que queramos correr en la VM.

  Cada proceso ejecutado en un VM es un proceso que corre dentro del SO de la VM, orquestado todo ello a través del Hypervisor para competir con los recursos del Host.

- **El Container engine**

  Es el responsable de gestionar y ejecutar todos los contenedores. El Sistema Operativo del Host es compartido y utilizado por todos los contendores. Por lo tanto no tenemos que duplicar librerías ni aplicaciones en cada contenedor.

  Cada proceso ejecutado en cada contenedor es como un proceso ejecutado en el Host y por tanto más ligero que el mismo proceso ejecutado en una VM. Esto significa que cuando un contenedor termina de ejecutar su tarea, el contenedor se para y deja de consumir recursos del Host. En una VM esto significaría que el proceso dentro de la VM termina, pero sin embargo la VM sigue ejecutándose en el Host.

La compartición de recursos entre contenedores se realiza de una forma rápida y sencilla, al igual que el aislamiento entre contenedores.

Cuando se trata de comparar los dos tipos de tecnologías se podría decir que **Docker** y sus contenedores tienen mucho más potencial que las máquinas virtuales, vamos a detallar bien cada punto destacando los aspectos fuertes y débiles al usar Docker vs VM, distinguiendo categorías como rendimiento, rapidez, portabilidad, seguridad y administración.

Rapidez

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

En comparación con las máquinas virtuales, los contenedores pueden ser más rápidos y consumirán menos recursos siempre que el usuario está dispuesto a pegarse a una única plataforma para proporcionar el sistema operativo compartido.

Una máquina virtual puede tardar hasta varios minutos para crearse y poner en marcha mientras que un contenedor puede ser creado y lanzado sólo en unos pocos segundos. Aplicaciones contenidas en contenedores ofrecen un rendimiento superior, en comparación a la ejecución de la aplicación dentro de una máquina virtual.

Para dar un ejemplo claro, los tiempos de inicio y detención de Docker son menores a 50ms, mientras que las máquinas virtuales inician en 30-45 segundos, y se detienen en 10 segundos o menos.

Portabilidad

Todas las aplicaciones tienen sus propias dependencias, que incluyen tanto los recursos de software y hardware. Los Contenedores Docker aportarán numerosos beneficios en comparación con las tecnologías existentes. En términos de tecnología, es bastante interesante en escenarios donde ayuda en la promoción de la portabilidad de la nube mediante la ejecución de las mismas aplicaciones en diferentes entornos virtuales esto es muy útil en el ciclo de vida para el desarrollo de software.

Docker es una plataforma abierta para desarrolladores, es un mecanismo que ayuda a aislar las dependencias por cada aplicación mediante la creación de contenedores. Los contenedores son escalables y seguros si los comparamos con el enfoque anterior del uso de máquinas virtuales.

Seguridad

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

100% compatible con sistemas de seguridad de kernel como SELinux, GRSEC, etc. Docker también ofrece grupos de control de acceso al demonio que controla la virtualización, restringiéndolo así sobre posibles cambios que puedan hacerse al sistema.

Administración

Soluciones tales como Docker hacen más fácil la gestión de contenedores.

Solución Híbrida: Máquinas Virtuales y Docker juntos

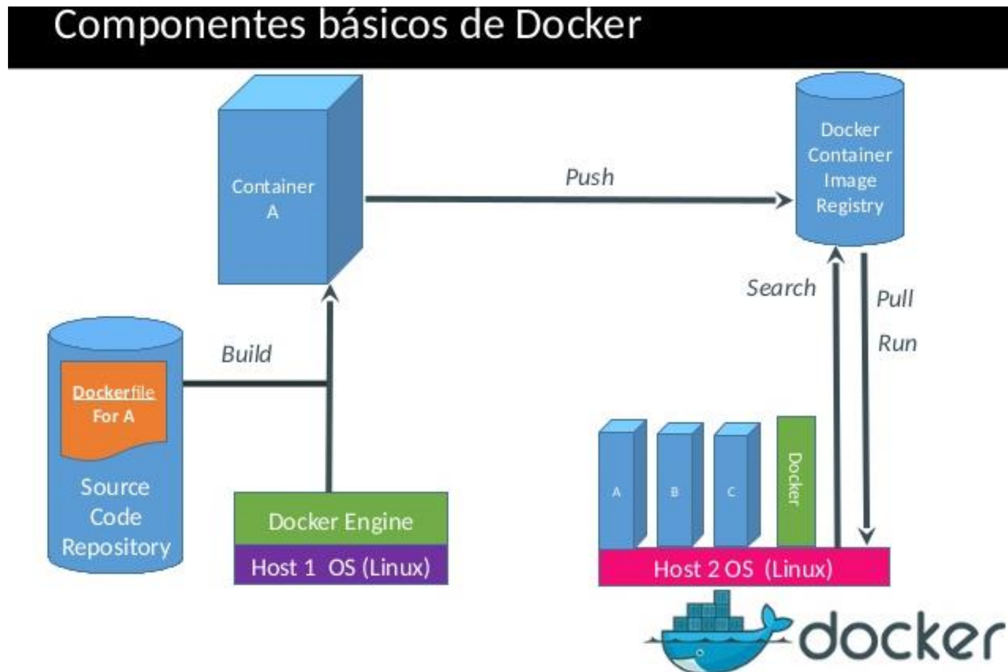
A veces se puede utilizar un enfoque híbrido que utiliza tanto la máquina virtual como Docker, la cual puede traer consigo muchos beneficios en determinados escenarios. Por ejemplo, en el desarrollo de una app podemos encontrar una forma de usar Docker y máquinas virtuales para probar el software que hemos desarrollado:

1. Correr varias pruebas para ver si una app está funcionando correctamente. Esto puede hacerse usando los contenedores de Docker.
2. Emular el entorno del cliente lo más idéntico posible a la realidad (topología de red, sistema operativo, firewall, servicios, etc), en este caso podemos usar también las máquinas virtuales junto con Docker para hacer las pruebas.

Conclusión

En el desafío Docker vs Máquinas Virtuales hemos visto de manear clara cómo Docker y sus contenedores están transformando las operaciones de los DevOps como una herramienta importante en el mundo del desarrollo web. Los casos de uso para los contenedores de Docker dentro del ámbito de DevOps son muchos, ejecutar aplicaciones en un contenedor y luego desplegar en cualquier lugar (la nube o en las instalaciones o cualquier distribución de Linux) es una realidad ahora.

COMPONENTES DE DOCKER



Docker está formado fundamentalmente por tres componentes:

- **Docker Engine**
- **Docker Client**
- **Docker Registry**

Docker Engine o Demonio Docker:

Es un demonio que corre sobre cualquier distribución de Linux y que expone una API externa para la gestión de imágenes y contenedores (y otras entidades que se van añadiendo en sucesivas distribuciones de docker como volúmenes o redes virtuales). Podemos destacar entre sus funciones principales:

- Creación de imágenes docker.
- Publicación de imágenes en un Docker Registry o Registro de Docker (otro componente Docker que se explicará a continuación).
- Descarga de imágenes desde un Registro de Docker
- Ejecución de contenedores usando imágenes locales.

Otra función fundamental del Docker Engine es la gestión de los contenedores en ejecución, permitiendo parar su ejecución, rearrancarla, ver sus logs o sus estadísticas de uso de recursos.

Docker Client o Cliente Docker

Es cualquier herramienta que hace uso de la api remota del Docker Engine, pero suele hacer referencia al comando docker que hace las veces de herramienta de línea de comandos (cli) para gestionar un Docker Engine. La cli de docker se puede configurar para hablar con un Docker Engine local o remoto, permitiendo gestionar

tanto nuestro entorno de desarrollo local, como nuestros servidores de producción. Los comandos de docker más comunes son:

•**docker info:** da información acerca de la cantidad de contenedores e imágenes que está gestionando la **máquina actual, así como los plugins actualmente instalados.**

•**docker images**: lista información de las imágenes que se encuentran disponibles en la máquina (nombre, id, espacio que ocupa, el tiempo que transcurrió desde que fue creada).

•**docker build:** crea una imagen desde el fichero Dockerfile del directorio actual.

•**docker pull <imagen>:<version>:** descarga en la máquina actual la versión de la imagen indicada. En caso de no indicar la versión descarga todas las que estén disponibles.

•**docker push <imagen>:<version>:** sube la versión de la imagen indicada a un Registro de Docker, permitiendo su distribución a otras máquinas.

•**docker rmi <imagen>:<version>:** elimina una imagen de la máquina actual.

•**docker run <imagen>:<version>:** crea un contenedor a partir de una imagen. Este comando permite multitud de parámetros, que son actualizados para cada versión del Docker Engine, por lo que para su **documentación lo mejor es hacer referencia a la página oficial.**

•**docker ps:** muestra los contenedores que están corriendo en la máquina. Con el flag **-a** muestra también los **contenedores que están parados.**

•**docker inspect contenedor:** muestra información detallada de un contenedor en formato json. Se puede acceder a un campo particular con el comando docker **inspect -f '{{.Name}}'** contenedor.

•**docker stop contenedor:** para la ejecución de un contenedor.

•**docker start contenedor:** reanuda la ejecución de un contenedor.

•**docker rm contenedor:** elimina un contenedor. Para borrar todos los contenedores de una máquina se puede ejecutar el comandodocker rm -fv $(docker ps -aq).

•**docker logs contenedor:** muestra los logs de un contenedor.

•**docker stats contenedor:** muestra las estadísticas de ejecución de un contenedor, como son la memoria utilizada, la CPU, el disco…

•**docker exec contenedor comando:** ejecuta un comando en un contenedor. Útil para depurar contenedores en ejecución con las opciones **docker exec -it contenedor bash**.

•**docker volume ls:** lista los volúmenes existentes en la máquina. Para un listado completo de los comandos relacionados con volúmenes ejecuta **docker volume --help.**

•docker network ls: lista las redes existentes en la máquina. Para un listado completo de los comandos relacionados con redes ejecuta **docker network --help.**

Docker Registry o Registro Docker

El Registro es otro componente de Docker que suele correr en un servidor independiente y donde se publican las imágenes que generan los Docker Engine de tal manera que estén disponibles para su utilización por cualquier otra máquina. Es un componente fundamental dentro de la arquitectura de Docker ya que permite distribuir nuestras aplicaciones. El Registro de Docker es un proyecto open sourceque puede ser instalado gratuitamente en cualquier servidor, pero Docker ofrece Dockerhub, un sistema SaaS de pago donde puedes subir tus propias

imágenes, acceder a imágenes públicas de otros usuarios, e incluso a imágenes oficiales de las principales aplicaciones como son: MySQL, MongoDB, RabbitMQ, Redis, etc.

El registro de Docker funciona de una manera muy parecida a git (de la misma manera que Dockerhub y us métodos de pago funcionan de una manera muy parecida a Github). Cada imagen, también conocida como repositorio, es una sucesión de capas. Es decir, cada vez que hacemos un build en local de nuestra imagen, el Registro de Docker sólo almacena el **diff** respecto de la versión anterior, haciendo mucho más eficiente el proceso de creación y distribución de imágenes.

INSTALACIÓN DE DOCKER

En la instalación de Docker queremos distinguir la instalación para el desarrollo en local, y la instalación en servidores para correr código en producción. En cuanto a la instalación en servidores de producción, la mayoría de proveedores de servicio: AWS, GCE, Azure, Digital Ocean… disponen de máquinas virtuales con versiones de Docker pre-instaladas. En cualquier caso, la instalación es bastante sencilla. Como el proceso varía un poco entre versiones de Docker, preferimos poner un enlace a la documentación oficial de lainstalación en Ubuntu.

Para la instalación en local para desarrolladores, Docker ofrece la instalación de Docker Toolbox. Docker Toolbox instala en Mac y en Windows, y consiste de los componentes:

- **Kitematic:** UI de gestión de contenedores.
- **Docker Machine**: herramienta para la creación de máquinas virtuales con Docker instalado. Incluye Virtualbox.
- **Docker Compose:** herramienta para gestionar contenedores basada en definición de contenedores y sus relaciones en un formato yml.

En este tutorial no explicaremos Kitematic por ser una herramienta en beta y en proceso de cambio, pero explicaremos un poco sobre cómo funcionan Docker Machine y Docker Compose.

DOCKER MACHINE

Docker Machine es un proyecto open source que como hemos dicho automatiza la creación de maquinas virtuales con Docker instalado. Incluye drivers para distintos proveedores de servicios en la nube, como AWS, GCE, Azure, Digital Ocean… y también para Virtualbox (ver lista completa de drivers aquí). **Virtualbox es la opción aconsejada** para instalaciones de Docker en local, en lugar de instalar Docker directamente en tu ordenador personal. Esto es debido a que nos permitirá crear o destruir la instalación de Docker con mayor facilidad, actualizar la versión de Docker, o trabajar con distintas instalaciones de Docker a la vez para aislar entornos de aplicaciones.

La documentación de los comandos disponibles en Docker Machine puede ser consultada aquí, pero queremos destacar algunas buenas prácticas:

- Crea una máquina de Virtualbox para cada aplicación con la que trabajes. Aunque Docker aísla la ejecución de contenedores, siempre hay inter-relaciones como los puertos donde escuchan los distintos componentes, por lo que correr todas las aplicaciones en el mismo Virtualbox puede dar lugar a conflictos. Además, el driver de Virtualbox tiene algunos bugs que te obligarán a recrear la máquina cada cierto tiempo, tener entornos separados reduce los inconvenientes de este proceso.
- El driver de Virtualbox permite definir la memoria máxima (**--virtualbox-memory**) y el disco máximo (**--virtualbox-disk-size**) de la máquina virtual. Modifica estos valores si tu aplicación necesita muchos recursos.
- Si tu presupuesto lo permite, puede ser conveniente crear tu máquina virtual de desarrollo en un proveedor de servicios como Amazon para no consumir recursos en tu ordenador personal, sobretodo si tu aplicación es muy pesada.
- Haz uso del comando **eval "$(docker-machine env name)"** para auto-configurar tu Docker Client respecto a la maquina **name,** de tal manera que tus commandos de Docker se ejecuten en la máquina **name.**

DOCKER COMPOSE

Docker compose es otro proyecto open source que permite definir aplicaciones muilti-contenedor de una manera sencilla y declarativa. Es una herramienta ideal para gestionar entornos de desarrollo y de pruebas, o para procesos de integración continua como veremos en la próxima sesión.

docker-compose es una alternativa más cómoda al uso del comando docker run, que resulta ingobernable cuando trabajamos con aplicaciones con varios componentes. Con Docker Compose se define un fichero docker-compose.yml que tiene esta forma:

```
web:
  build: .
  ports:
   - "5000:5000"
  links:
```

```
  - redis
redis:
  image: redis
```

Donde estamos definiendo una aplicación que se compone de un contenedor definido desde un Dockerfile local, que escucha en el puerto 5000, y que hace uso de redis como un servicio externo. Dada esta definición, la manera de levantar la aplicación es simplemente:

```
docker-compose up -d
```

docker-compose acepta distintos comando, una lista completa puede encontrarse aquí. Destacar los siguientes puntos sobre docker-compose:

- **docker-compose up -d** levanta la aplicación en modo demonio, **docker-compose up** la levanta en primer plano, mostrando los logs de los distintos contenedores. La ejecución sucesiva del comando **docker-compose up -d** sólo recrea los contenedores que hayan cambiado su imagen o su definición.

- **docker-compose up -d** no hace el build de las imágenes locales. Si deseas actualizar tu aplicación en base a los últimos cambios de tu código, necesitarás hacer **docker-compose build** antes de ejecutar **docker-compose up -d** nuevamente. Un truco para mejorar este proceso es montar tu código como un volumen en el fichero **docker-compose.yml**, de tal manera que tu container siempre ve los últimos cambios en tu código fuente.

docker-compose permite definir prácticamente todos los flags que soporta el comando docker run, pero docker-compose es mucho más fácil de utilizar. Las opciones más comunes son:

- build: para indicar que el container se construye desde un Dockerfile local.
- image: para indicar que el container corre un imagen remota.
- command: para redefinir el comando que ejecuta el container en lugar del comando definido en la imagen.
- environment: para definir variables de entorno en el contenedor. Se pueden pasar haciendo referencia a un fichero usando la propiedad env_file. Si la variable no tiene un valor dado, su valor se cogerá del entorno de shell que ejecuta el **docker-compose up**, lo que puede ser útil para pasar claves, por ejemplo.
- links: para definir relaciones entre contenedores.
- ports: para mapear los puertos donde el contenedor acepta conexiones.
- volumes: para definirir volúmes en el contenedor.
- volumes_from: para reusar los volúmenes de otro contenedor.

Aquí tenéis una lista completa y actualizada de las opciones que permite docker-compose.

CONCLUSIÓN

Docker está teniendo un notable éxito y éste no ha venido de forma gratuita, sino porque es una tecnología disruptiva que implica una mejora tremenda en los procesos de desarrollo de software. Nosotros aconsejamos su conocimiento y utilización tanto por parte de desarrolladores como por parte de administradores de sistema, ya que todas las grandes firmas de la nube, como son Amazon, Google, Microsoft, IBM, Oracle… están metidos en la carrera de dar el mejor soporte posible para Docker. Esto se debe a que todas las empresas, pequeñas, medianas y grandes multinacionales, están haciendo uso de Docker para desarrollar y gestionar sus aplicaciones (valga como ejemplo el gobierno de los EE.UU), por lo que no adoptar esta tecnología (u otras similares que puedan aparecer) será una desventaja competitiva en los mercados tecnológicos.

INTEGRACIÓN CONTINUA CON DOCKER

La Integración Continua se refiere a la práctica de automatizar tareas de modo que se ejecuten automáticamente cuando se produce un evento, por ejemplo, una nueva versión de código en nuestro repositorio. Ejemplos de las tareas que pueden ser automatizadas son: compilación de componentes, ejecución de pruebas unitarias, ejecución de pruebas de integración, ejecución de pruebas de aceptación, obtención de métricas de calidad de código…

Algunos de los objetivos que persigue la integración continua son:

- Facilitar la integración entre distintos los distintos componentes de nuestra aplicación.
- Automatizar la construcción de nuestra aplicación.
- Automatizar la ejecución de tests en la construcción de nuestra aplicación.
- Desplegar las cambios de nuestro código tan frecuentemente como sea posible.
- Probar en una réplica del entorno de producción.
- Permitir que todo el mundo pueda probar la última versión de nuestra aplicación de una manera sencilla.

La integración continua aporta aporta innumerables ventajas en el objetivo de conseguir un software de alta calidad, algunas de las cuales son:

- Detectar rápidamente posibles conflictos entre desarrolladores.
- Garantizar el correcto funcionamiento de nuestra aplicación antes de desplegar una nueva versión.
- Agiliza la corrección de los errores detectados.
- Permite resolver problemas de integración a lo largo de todo el proceso de desarrollo del software y no sólo al final del mismo, evitando situaciones caóticas previas a la fecha de entrega.

Aunque el concepto de integración continua es muy amplio, en este curso lo vamos a delimitar a:

- **Docker Build**: proceso de construcción de una imagen de docker.
- **Testing con Docker**: como utilizar Docker para facilitar la automatización del testeo de nuestras aplicaciones.

DOCKER BUILD

Como vimos en la introducción a Docker, una imagen se corresponde con la información necesaria para arrancar un contenedor, y básicamente se compone de un sistema de archivos y de otros metadatos como son el comando a ejecutar, las variables de entorno, los volúmenes del contenedor, los puertos que utiliza nuestro contenedor…
El build de una imagen termina una vez que la imagen de docker se sube a un Registro de Docker, momento en el cual comienza el periodo de despliegue de la aplicación.

La manera recomendada de construir una imagen es utilizar un fichero **Dockerfile,** un fichero con un conjunto de instrucciones que indican cómo construir una imagen de Docker. Las instrucciones principales que pueden utilizarse en un Dockerfile son:

- **FROM image**: para definir la imagen base de nuestro contenedor.
- **RUN comando**: para ejecutar un comando en el contexto de la imagen.
- **CMD comando**: para definir el comando que ejecuta el container al arrancar.
- **EXPOSE puerto**: para definir puertos donde el contenedor acepta conexiones.
- **ENV var=value**: para definir variables de entorno.
- **COPY origen destino**: para copiar ficheros dentro de la imagen.
- **VOLUME path**: para definir volúmenes en el contenedor.

Para una lista completa de las instrucciones disponibles ir a la documentación oficial.

La cache de Docker

La construcción de una imagen de Docker dado un Dockerfile puede ser un proceso costoso ya que puede implicar la instalación de un número elevado de librarías, y al mismo tiempo es un proceso bastante repetitivo porque sucesivos builds del mismo Dockerfile suele ser similares entre sí. Es por eso que Docker introduce el concepto de la **cache** para optimizar el proceso de construcción de imágenes.

La primera optimización que hace la cache de Docker es la descarga de la imagen base de nuestro Dockerfile. Docker descargará la imagen base siempre que la misma no se encuentre ya descargada en la máquina que hace el build. Esta optimización parece obvia ya que estas imágenes pueden tener un tamaño de cientos de MB, pero hay que tener cuidado ya que si la versión remota de la imagen cambia, **Docker seguirá utilizando la versión local**. Por tanto, si queremos ejecutar nuestro Dockerfile con la nueva versión de la imagen base deberemos de hacer un **docker pull** manual de la imagen base.

Como hemos comentado anteriormente, una imagen de Docker tiene una estructura interna bastante parecida a un repositorio de git. Lo que conocemos como commits en git lo denominamos capas de una imagen en Docker. Por lo tanto, una imagen (o repositorio) es una sucesión de capas en un Registro de Docker, donde cada capa almacena un diff respecto de la capa anterior. Esto es importante de cara a optimizar nuestros Dockerfiles, como veremos en la siguiente sección.

Por ahora bastará saber que cada instrucción de nuestro Dockerfile creara una y sólo una capa de nuestra imagen. Por lo tanto, la cache de Docker funciona a nivel de instrucción. En otras palabras, si una línea del Dockerfile no cambia, en lugar de recomputarla, Docker asume que la capa que genera esa instrucción es la misma que la ejecución anterior del Dockerfile. Por lo tanto, si tenemos una instrucción tal como:

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

que no ha cambiado entre 2 build sucesivos, los comandos apt-get no se ejecutarán, sino que se reusará la capa que generó el primer build. Por tanto, aunque antes de ejecutar el segundo build haya una nueva versión del paquete git, la imagen construida a partir de este Dockerfile tendrá la versión de git anterior, la que se instaló en el primer build de este Dockerfile.

Es importante destacar los siguientes aspectos sobre la cache de Docker:

- La cache de Docker es local, es decir, si es la primera vez que haces el build de un Dockerfile en una máquina dada, todas las instrucciones del Dockerfile serán ejecutadas, aunque la imagen ya haya sido construida en un Registro de Docker.
- Si una instrucción ha cambiado y no puede utilizar la cache, la cache queda invalidada y las siguientes instrucciones del Dockerfile serán ejecutadas sin hacer uso de la cache
.
- El comportamiento de las instrucciones **ADD** y **COPY** es distinto en cuanto al comportamiento de la cache. Aunque estas instrucciones no cambien, invalidan la caché si el contenido de los ficheros que se están copiando ha sido modificado.

Por último, si por algún motivo deseas hacer un build sin usar la cache, puedes hacer uso del flag **--no-cache=true** para dicho fin.

Consejos para escribir un Dockerfile

1. Usa .dockerignore

El build de una image se ejecuta a partir de un Dockerfile y de un directorio, que se conoce con el nombre de contexto. Este directorio suele ser el mismo que el directorio donde se encuentra el Dockerfile, por lo que si ejecutamos la instrucción:

```
ADD app.py /app/app.py
```

Estamos añadiendo a la imagen el fichero **app.py** del contexto, es decir, el fichero **app.py** que se encuentra en el directorio donde está el Dockerfile. Dicho directorio se comprime y se manda al Docker Engine para construir la imagen, pero puede que tenga ficheros que no son necesarios. Es por eso que este directorio puede tener un fichero **dockerignore,** que de una manera similar a **fichero .gitignore**, indica los ficheros que no deben ser considerados como parte del contexto del build.

2. Reduce el tamaño de tus imágenes al mínimo

Tu imagen Docker sólo debe contener lo estrictamente necesario para ejecutar tu aplicación. Con el objetivo de reducir complejidad, dependencias, tamaño de las imágenes, tiempos de build de una imagen, debes evitar la instalación de paquetes sólo por el hecho de que puedan ser útiles para depurar un contenedor. Como ejemplo, no incluyas editores de texto en tus imágenes.

3. Ejecuta sólo un proceso por contenedor

Salvo raras excepciones, es recomendable correr sólo un proceso por contenedor. Esto permite reutilizar contenedores más fácilmente, que sean más fáciles de escalar, y da lugar a sistemas más desacoplados. Por ejemplo saca tu lógica de logging a un contenedor independiente.

4. Minimiza el número de capas de tu imagen.

Como hemos dicho anteriormente, cada capa de una imagen se corresponde con una instrucción del Dockerfile. Compare el Dockerfile:

```
RUN apt-get update

RUN apt-get install -y bzr

RUN apt-get install -y cvs

RUN apt-get install -y git

RUN apt-get install -y mercurial
```

con este otro:

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

    bzr \

    cvs \

    git \

    mercurial \

    apt-get clean
```

Ambos son igualmente legibles, pero el primero genera 5 capas, y el segunda sólo una, que además ejecuta un **apt-get clean** que reduce el tamaño de dicha capa.

5. Optimiza el uso de la cache.

Optimiza el uso de la cache añadiendo al principio de tu Dockerfile las instrucciones que menos cambian (como la instalación de librerías), y dejando para el final las que más cambian (como el copiado del código fuente). Como ejemplo compare el Dockerfile:

```
FROM python:2.7



WORKDIR /app



ADD requirements.txt /app/requirements.txt

RUN pip install -r requirements.txt



ADD * /app



CMD ["python", "app.py"]
```

con este otro:

```
FROM python:2.7



WORKDIR /app



ADD * /app

RUN pip install -r requirements.txt



CMD ["python", "app.py"]
```

El primero cachea la instalaciones de las dependencias pip siempre que no añadamos nuevas dependencias al **fichero requirements.txt,** antes de añadir el código fuente. Sin embargo, el segundo, aunque genere menos capas, no reusa la instalación de las dependencias porque **ADD * /app** invalida la cache en cuanto hay un cambio en nuestro código fuente.

6. Parametriza tus Dockerfiles usando argumentos

Aumenta la reusabilidad de tus Dockerfiles entre distintos entornos y aplicaciones parametrizando tus Dockerfiles con argumentos. Los argumentos son valores que se pasan como parámetros a cada build (aunque pueden tener valores por defecto), y que puedes utilizar en las instrucciones de tu Dockerfile. Por ejemplo, el Dockerfile:

```
FROM ubuntu

ARG user=root

ARG password

RUN echo $user $password
```

puede ser parametrizado de la siguiente manera:

```
docker build -t imagen --build-arg password=secret .
```

TESTING DOCKER

El testeo de aplicaciones ha sido tradicionalmente tedioso ya que la ejecución de nuestra aplicación requiere de la instalación de sus dependencias, y además puede necesitar componentes externos como bases de datos. Sabemos que Docker soluciona estos problemas, y que gracias a la herramienta docker-compose, es prácticamente trivial el proceso de ejecución de una aplicación en local.

Pues bien, estas ideas pueden ser aplicadas al proceso de testeo de una aplicación. Lo primero que tenemos que hacer es dockerizar nuestros scripts de tests, de tal manera que podamos ejecutarlos haciendo uso de docker-compose al igual que hacemos con cualquier otra imagen de Docker. El proceso de dockerizar nuestros tests equivale a crear un fichero **Dockerfile.test** que instalará las librerías necesarias para la ejecución de nuestros tests.

Una vez dockerizados los tests, podremos crear un fichero **docker-test.yml** donde haremos referencia a la imagen que ejecuta nuestros tests, y donde añadiremos el resto de servicios que necesitemos para levantar nuestro entorno de pruebas. En un caso sencillo, supongamos que la ejecución de nuestros tests sólo necesita tener levantada una base de datos MySQL. Nuestro fichero **docker-test.yml** podría ser así:

```
test:
  build: .
  dockerfile: Dockerfile.test
  link:
    - db
db:
  image: mysql
```

Por tanto, la ejecución de nuestros tests puede ser automatizada de la siguiente manera:

```
docker-compose -f ~/testing/docker-test.yml -p ci build
docker-compose -f ~/testing/docker-test.yml -p ci up -d
docker logs -f ci_test_1
docker wait ci_test_1
```

La primera línea construye la imagen que corre los tests, la segunda levanta el entorno de pruebas (**-p** cisirve para prefijar los containers creados por docker-compose y **-f** para hacer referencia a un fichero yml que no sea **docker-compose.yml,** que es el valor por defecto), la tercera muestra la salida en streaming de nuestros tests, y finalmente la cuarta muestra el código de salida de nuestros tests, de tal manera que **0** indica que los tests han pasado, y cualquier otro valor, que los tests fallaron.

Este enfoque para la ejecución de tests basado en Docker y Docker Compose presenta las siguientes ventajas:

- Automatizable: esta manera de testear es independiente de la aplicación que queremos testear. Esto facilita enormemente la tarea del administrador que se dedique a automatizar el testeo de aplicaciones en el proceso de integración continua.

•Ligero: Docker es ligero, por lo que el fichero docker-test.yml puede contener tantos servicios como sea necesario y podrá ser ejecutado en una sola máquina. Esto permite la ejecución de entornos complejos para pruebas de integración y aceptación.

•Portable: gracias a que el proceso está basado en Docker y Docker es portable, la ejecución de pruebas puede realizarse en cualquier máquina, independientemente de su sistema operativo o el hardware interno.

•Inmutable: gracias a que Docker es inmutable, los procesos de pruebas también lo son, por lo que las pruebas que pasen en tu máquina local están garantizadas a pasar en la máquina utilizada para la integración continua.

Conclusión

Docker es una herramienta que también revoluciona los procesos de integración continua, al ofrecer cuatro ficheros que actúan como contrato entre el desarrollador y el personal de administración de sistema:

•**Dockerfile**: cómo construir la imagen de nuestra aplicación.

•**docker-compose.yml**: cómo ejecutar nuestra aplicación.

•**Dockerfile.test**: cómo construir la imagen que prueba nuestra aplicación.

•**docker-test.yml**: cómo testear nuestra aplicación.

Por tanto, el personal de sistemas puede añadir el proceso de build y de pruebas a su herramienta de integración continua independientemente de la aplicación en cuestión, o del sistema operativo o hardware que estén usando los desarrolladores de dicha aplicación.

Existen números herramientas para automatizar el build y las pruebas de aplicaciones, algunas puedes instalarlas en tu infraestructura, como:

•Jenkins

•Bamboo

•Drone.io

Y otras funcionan bajo el modelo SaaS, como:

•Travis

•Circleci

Docker ofrece sus propias herramienta de integración continua:

•Dockerhub: auto builds cuando hay un commit en Github.

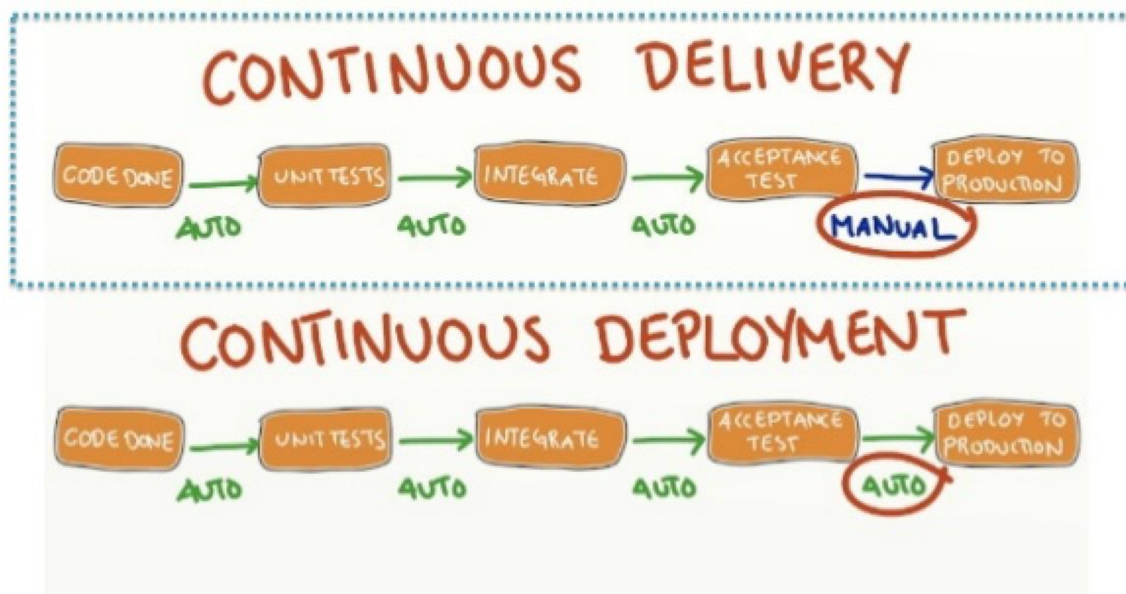
•Tutum: testing cuando hay una PR en Github.

Despliegue y gestión

ENTREGA CONTINUA CON DOCKER

La Entrega Continua es una metodología para el despliegue de aplicaciones en producción que surge de las mejores prácticas de gestión de configuración, pruebas automatizadas, integración continua, gestión de datos y gestión de versiones. El objetivo es automatizar al máximo el despliegue de un sistema en un ambiente productivo, minimizando los riesgos asociados a dicho proceso, y facilitando la detección de regresiones.

Tenemos que distinguir entre Entrega Continua (Continuous Delivery) y Despliegue Continuo (Continuous *Deployment*). La entrega continua se refiere a la práctica de tener automatizado todos los procesos de validación del software y su despliegue en producción, pero el proceso de actualizar el entorno de producción lo inicia el personal de operaciones. Por contra, nos referimos a despliegue continuo cuando el proceso de actualizar el entorno de producción es también automático y cada commit en nuestro repositorio de código va directamente a producción después de pasar todo el proceso de integración continua. La diferencia se muestra en la siguiente figura:



Desde un punto de vista tecnológico, ambas prácticas son equivalentes. La decisión de tomar un enfoque de entrega continua o despliegue continuo suele responder a decisiones tácticas de la empresa. Lo normal es que empresas de pequeño y mediano tamaño opten por una metodología de entrega continua para controlar cuando lanzan nuevas funcionalidades, mientras que grandes empresas como Amazon o Google suelen optar por el despliegue continuo. En esta lección vamos a centrarnos en cómo hacer entrega continua con Docker.

La entrega continua es la evolución natural de tener unos procesos de integración continua robustos y completos. La práctica de la entrega continua está condenada al fracaso si no disponemos de una buena cobertura en nuestras pruebas unitarias, de integración y de aceptación. También hay que destacar que todo proceso de entrega continua tiene que tener estas propiedades:

- Versionado de todos los componentes de nuestra aplicación. Todo nuestro código y nuestras configuraciones deben existir en un repositorio (por ejemplo tipo git) y cada release tiene que ser tageada para facilitar los procesos de identificación de errores y rollback.

•Nuestros build y tests deben mantenerse en tiempos de ejecución lo más bajos posibles. En caso contrario, un hotfix urgente a producción tardará demasiado en pasar por todo el proceso automatizado de integración y entrega continua y podemos tener la tentación de saltarnos los procesos automatizados de integración y entrega continua para adelantar la publicación del hotfix, con el riesgo de posibles regresiones. Además, el tener builds y tests rápidos acelerará todo el ciclo de desarrollo.

•Notificaciones: los desarrolladores y el personal de operaciones deben de recibir notificaciones cuando un build o una prueba haya fallado o haya tenido éxito. De igual manera, deben notificarse los despliegues a producción. Cuando una prueba o un build falla, la prioridad del equipo de desarrollo debe ser arreglarlo.

MOTIVACIÓN

Tradicionalmente el proceso de actualizar una aplicación en un entorno de producción ha sido considerada una práctica de alto riesgo por el riesgo de sufrir regresiones en el sistema. Esto ha tenido el efecto negativo de alargar las entregas por meses, incrementando el time to market de las empresas, y reduciendo sus posibilidades frente a la competencia. Hay dos motivaciones principales para la entrega continua:

•Reducir el time to market: de esta manera obtenemos feedback de nuestros usuarios tan pronto como es posible, disponiendo de más datos para mejorar el producto. Además, los usuarios tienden a usar el producto que primero les da la funcionalidad que están buscando.

•Reducir el coste de los errores del software (Fast Fail): cuanto antes detectemos un error en el ciclo de desarrollo del software, más fácil será identificarlo y corregirlo.

El concepto de Fast Fail siempre ha traído controversia. Algunos autores discuten que es mejor mejorar el desarrollo del software para que no se produzcan fallos en producción, pero la realidad es que esto no es posible. Veámoslo con un ejemplo:

Imaginemos una empresa que lleva 3 meses preparando una release. Todo está preparado para actualizar el entorno de producción. A las 9AM se actualiza el entorno de producción, y después de comprobar que todo parece funcionar correctamente, se inicia la campaña de marketing destinada a captar más usuarios. Minutos más tarde de haber actualizado el entorno de producción, se produce un fallo en cascada que tira abajo el entorno de producción al completo. Después de diez horas de duro trabajo por parte del equipo de desarrollo y de operaciones, descubren que una función no estaba cerrando conexiones a la base de datos, y que esto había derivado en la imposibilidad de aceptar nuevas conexiones para las peticiones de usuario. Una vez solucionado el problema, la empresa hace un análisis post mortem para ver cómo evitar este tipo de problemas en el futuro. Las propuestas que surgen de este análisis son:

•Más pruebas manuales.

•Más pruebas automáticas.

•Revisiones de código y programación en parejas.

•Más planificación al inicio.

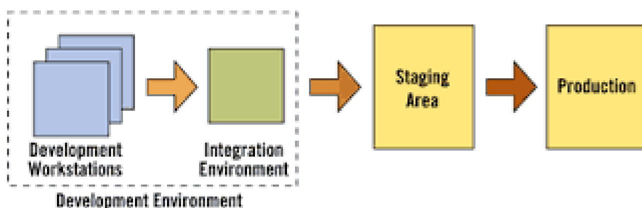
Ninguna de estas propuestas son una solución definitiva. Las pruebas manuales son de la largo la peor solución. Aquí aplica aquello de que “si estás haciendo más de dos veces lo mismo, algo estás haciendo mal, mejor haz un programa que resuelva el problema”. Pero tampoco las pruebas automáticas, aunque muy recomendables (por no decir indispensables), van evitar la presencia de errores en producción. Ninguna prueba automática se tan brutal, aleatoria, maliciosa, ignorante o agresiva como la suma de todos los usuarios del sistema. Las revisiones de código y la programación en parejas son también prácticas muy recomendables y reducen la cantidad de errores porque dos ojos ven más que uno, pero siempre habrá errores que sigan sin detectarse. En cuanto a una mayor planificación, puede ayudar a evitar urgencias de última hora, donde la presencia de errores aumenta

considerablemente, pero tampoco garantizan la no existencia de errores. La realidad es que este tipo de errores van a incrementarse a medida que crezca el producto y se vuelva más grande y complejo. Veamos porqué la entrega continua es la única solución escalable.

En un entorno de entrega continua, nuestro desarrollador realiza el commit de la función que no cierra las conexiones con la base de datos. El commit llega a producción y minutos más tarde hay una alerta indicando que producción se ha caído. Se comprueban los últimos commits y rápidamente se detecta donde está el problema, haciendo un rollback al commit anterior al que introdujo la regresión. Nuestro desarrollador prácticamente no pasa tiempo depurando ya que el error está localizado e implementa rápidamente el fix de su código. Este error causó una falla en cascada, pero el tiempo de caída fue mínimo.

ENTORNOS PARA ENTREGA CONTINUA

Para la implantación correcta de un proceso de entrega continua es necesario disponer de tres entornos de ejecución que resumimos en la siguiente figura:



Cada uno de estos entornos se corresponde con tres ramas distintas en nuestro repositorio de código, comúnmente: dev, staging y master.

- **Entorno de desarrollo**: incluye las máquinas de los desarrolladores y un entorno de ejecución (llamado de integración) que se actualiza para cada commit en la rama de dev. Las pruebas unitarias y de integración deben estar automatizadas a este nivel.
- **Entorno de staging**: es un reflejo del entorno de producción. Sirve para ejecutar pruebas de aceptación y de rendimiento, y para probar los procesos de despliegue automático en un entorno equivalente al de producción.
- **Entorno de producción**: es el entorno que utilizan los usuarios de la aplicación.

En empresas pequeñas o de mediano tamaño el entorno de desarrollo y el de staging se fusionan en uno sólo, principalmente por el costo asociado a lanzar un entorno equivalente al de producción, pero también para reducir el tiempo gastado en la gestión de los distintos entornos.

USO DE DOCKER PARA ENTREGA CONTINUA

En esta sección vamos a ver cómo utilizar Docker para mejorar nuestra metodología de entrega continua. Vamos a asumir que el entorno de desarrollo y el de staging han sido fusionados (al entorno resultante le seguiremos llamando staging).

Docker no sólo simplifica enormemente la metodología de entrega continua, si no que la hace mucho más fiable porque el salto del entorno de staging al de producción es menor, y porque las imágenes de Docker, portables e inmutables, has sido validadas por las pruebas unitarias, de integración y de aceptación.

Lo primero que tenemos que hacer es crear un servidor para realizar las labores de integración continua, y que vamos a llamar builder. También crearemos un servidor, llamado staging, para el entorno de staging. Por último, crearemos varios servidores para nuestro entorno de producción.

A continuación configuramos nuestro repositorio de código (por ejemplo, un repositorio git en Github). El repositorio constará de dos ramas, master y staging, donde no estará permitido comitear directamente. Cada

desarrollador hará un fork de este repositorio para hacer desarrollo. Para comitear código al repositorio principal, los desarrolladores crearán pull requests que pueden ser revisadas por el resto de desarrolladores.

Nuestros procesos de integración continua validarán el correcto funcionamiento de cada pull request. Cuando una pull request es revisada y mergeada, otro proceso de integración continua volverá a correr las pruebas en la rama de staging, y si estas pasan, creará una imagen de Docker con el tag staging y el entorno de staging se actualiza automáticamente.

Cuando el equipo de operaciones decida hacer una nueva release se mergea la rama de staging en la rama master, y otro proceso de integración construirá la imagen de Docker con el tag latest desde la rama master. Además, a este commit de master le añadiremos un tag en Github, y crearemos este mismo tag en la imagen de Docker desde la rama master. También está automatizado el proceso de apagar los contenedores que están corriendo en producción para recrearlos con la nueva imagen. Ante cualquier regresión en producción, podemos fácilmente actualizar los contenedores corriendo en producción para que usen un tag de la imagen de Docker anterior. Por último, todos estos procesos automatizados se notifican al equipo de desarrollo a través de email, o por canales como Slack o Hipchat.

VENTAJAS DE DOCKER EN PRODUCCIÓN

La primera ventaja ya la hemos comentado y es que Docker mejora enormemente la fiabilidad de la entrega continua, ya que las imágenes de Docker son portables inmutables y se generan una vez validadas contra la ejecución automática de pruebas.

La segunda ventaja es la facilidad que ofrece Docker para la automatización de tareas. El proceso de generación de imágenes de Docker y de su distribución es casi trivial gracias a la API que implementa el demonio de Docker. Pero también lo es automatizar la respuesta a eventos como aumentos de tráfico o caída de datacenters.

Por último, Docker es una herramienta ideal para implementar arquitecturas basadas en microservicios, y gracias al aislamiento entre contenedores, permite la combinación de microservicios en una misma máquina para la consecución de diversos objetivos. Por ejemplo, imagina una aplicación con una API pública que, entre otros, ofrece un servicio de autenticación. Imaginemos otro componente que hace uso de la API pública para autenticar peticiones. Además de servidores dedicados a ofrecer la API pública, Docker facilita enormemente correr siempre el segundo componente con un contenedor de API pública en la misma máquina (y no visible públicamente). De esta manera, aunque nuestra API pública sufra, por ejemplo, un ataque de denegación de servicio, el segundo componente podrá seguir funcionando. A esta combinación de contenedores, que tienen que correr conjuntamente, se les denomina pods.

Networking

NETWORKING EN DOCKER

Docker usa un bridge Ethernet para permitir al daemon comunicarse con el dispositivo de red de la máquina (a partir de ahora usaré máquina o host para referirme a lo mismo) en cuestión.

Esta bridge network se llama docker0 (ó bridge) y se crea cuando instalamos la Docker Engine. Este dispositivo de red siempre se activa cuando arrancas la máquina, docker instala el dispositivo dentro del kernel de Linux para habilitar y configurar esta network. A continuación docker crea una subnet virtual en la máquina para permitir la transmisión de paquetes entre contenedores.

Para ver esta docker brigde network puedes hacerlo utilizando el comando**: ifconfig** o **ipconfig**.

Antes de nada, debemos de saber que Docker no ofrece mecanismos de balanceo del tráfico, cortafuegos y otras características propias de sistemas SDNs.

Para permitir a nuestros contenedores tener visibilidad desde fuera y entre ellos debemos de ajustar tres aspectos en la configuración. Estos parametros de configuración se especifican cuando arrancamos el Docker daemon. Son los siguientes:

1.Que no haya reglas iptable ya definidas en la máquina que puedan bloquear el tráfico hacía y desde nuestros contenedores (FORWARD, INPUT). **$ sudo iptables -L**

2.Cuando los contenedores quieren comunicarse entre sí Docker necesita definir reglas iptables o delegarlo en otro sistema (weave, calico). Para permitir a Docker crear las reglas iptable entre contenedores que se comunican entre sí es necesario habilitar el parámetro 'iptables=true'. De ese modeo, Docker engine añadira las reglas a la filter chain de DOCKER. Por cierto, ten en cuenta que Docker no modificará ninguna regla existente.

```
$ sudo iptables -L DOCKER

iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

3.Otro argumento que puede afectar a la comunicación entre contenedores es `ip_forward`. Este habilita el tráfico entre contenedores y el mundo exterior cuando esta habilitado, `--ip-forward=true`. Puedes usar el siguiente comando para comprobarlo

$ sysctl net.ipv4.conf.all.forwarding

Si no habilitas ip_forward=true tus contenedores no podrán comunicarse entre sí, obviamente esto por un lado protegería los contenedores y el host de cualquier vulnerabilidad en temas de red. Pero por el otro lado, sí ip_forward=true los contenedores podrían comunicarse entre ellos de manera arbitraria. A continuación aprenderemos más a cerca de que configuración utilizar.

En nuestro entorno utilizaremos los siguientes parámetros de configuración:

$ docker -d --ip_forward=false --iptables=true -H fd://

$ sudo systemctl cat docker

Docker también permite utilizar IPv6 si se necesita, solo necesitamos arrancar el Docker daemon y pasarle el parámetro '--ipv6´ cuando arrancamos el daemon. También podemos definir la subnet.

$ docker daemon --ipv6 --fixe-cidr-v6="2001:db9:2::/64"

COMUNICACIÓN ENTRE CONTENEDORES EN UN HOST

Cuando un contenedor se despliega, Docker crea un par de interfaces Ethernet a las que le asigna una IP (aleatoriamente) y una subnet de un rango privado de IPs no usado previamente por el host.

La comunicación entre contenedores está controlada por el sistema operativo a través de dos controladores: network topology (que verifica que el bridge existe) y iptables rules.

Para tener control del tráfico entre contenedores debemos de configurar nuestro docker daemon con el argumento ip_forward=false. Para habilitar la comunicación entre contenedores es necesario que usemos Docker links **--link=CONTAINER_NAME:ALIAS.**

Hasta ahora la comunicación entre contenedores estaba basada en Docker links. Estos están solo soportados por la network por defecto de Docker, la brigde network (docker0).

Los links substituyen el nombre o alias del contenedor por su IP y creando una entrada en '/etc/hosts' en el contenedor que hace uso del link. Asímismo sí Docker daemon corre con **--icc=false** y **--iptables=true** cuando docker detecta un Docker link en la definición de alguno de nuestros contenedores. Docker añadirá un par de reglas iptable ACCEPT para permitir a ambos contenedores transmitir paquetes entre ellos usando sus puertos externos (EXPOSE en Dockerfiles).

Sí por el contrario utilizas Docker v1.9, entonces debe evitar su uso ya que Docker networking los reemplazará por el uso de networks personalizadas entre contenedores.

Tipos de Docker networks

Hasta ahora hemos visto, la Docker network de tipo bridge. Sin embargo a veces, no queremos usar la red de Docker y usar directamente la red de nuestro host. Esto es posible usando el argumento **--net=host c**uando despleguemos nuestro contenedor. Recordar que el argumento **--net** puede ser usado para determinar que red usar cuando despleguemos nuestro contenedor (igual que en rkt, coíncidencia ??).

$ docker inspect --format='{{json .NetworkSettings}}'

Sí por el contrario quieres que tu contenedor no tenga acceso a la red, puedes utilizar el mismo argumento pero pasándole el valor '--net=none'. Docker añadirá el contenedor a un network group pero sin interfaz de red.

A parte de usar estos tres tipos de redes es posible crear tu propia configuración de red para utilizar en tus contenedores Docker.

TIPOS DE DOCKER NETWORKS

Hasta ahora hemos visto, la Docker network de tipo bridge. Sin embargo a veces, no queremos usar la red de Docker y usar directamente la red de nuestro host. Esto es posible usando el argumento **--net=host**cuando despleguemos nuestro contenedor. Recordar que el argumento **--net** puede ser usado para determinar que red usar cuando despleguemos nuestro contenedor (igual que en rkt, coíncidencia ??).

$ docker inspect --format='{{json .NetworkSettings}}'

Sí por el contrario quieres que tu contenedor no tenga acceso a la red, puedes utilizar el mismo argumento pero pasándole el valor '--net=none'. Docker añadirá el contenedor a un network group pero sin interfaz de red.

A parte de usar estos tres tipos de redes es posible crear tu propia configuración de red para utilizar en tus contenedores Docker.

Comandos de Docker network

A continuación se muestra una lista de los comandos que se pueden usar con Docker networking:

```
$ docker network create

$ docker network connect

$ docker network ls

$ docker network rm

$ docker network disconnect

$ docker network inspect
```

COMANDOS DE DOCKER NETWORK

A continuación se muestra una lista de los comandos que se pueden usar con Docker networking:

```
$ docker network create

$ docker network connect

$ docker network ls

$ docker network rm

$ docker network disconnect

$ docker network inspect
```

docker network inspect

Este comando te permite saber los recursos empleados por una network así como su configuración

```
$ docker network inspect bridge
[
  {
      "Name": "bridge",
      "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
      "Scope": "local",
      "Driver": "bridge",
      "IPAM": {
         "Driver": "default",
         "Config": [
            {
               "Subnet": "172.17.0.1/16",
               "Gateway": "172.17.0.1"
            }
         ]
      },
      "Containers": {},
      "Options": {
         "com.docker.network.bridge.default_bridge": "true",
         "com.docker.network.bridge.enable_icc": "true",
         "com.docker.network.bridge.enable_ip_masquerade": "true",
         "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
         "com.docker.network.bridge.name": "docker0",
         "com.docker.network.driver.mtu": "9001"
```

```
    }
  }
]
```

Ahora vemos el estado después de haber creado dos contenedores asociados a esa network:

```
$ docker network inspect bridge
{[
  {
    "Name": "bridge",
    "Id": "f7ab26d71dbd6f557852c7156ae0574bbf62c42f539b50c8ebde0f728a253b6f",
    "Scope": "local",
    "Driver": "bridge",
    "IPAM": {
      "Driver": "default",
      "Config": [
        {
          "Subnet": "172.17.0.1/16",
          "Gateway": "172.17.0.1"
        }
      ]
    },
    "Containers": {
      "3386a527aa08b37ea9232cbcace2d2458d49f44bb05a6b775fba7ddd40d8f92c": {
        "EndpointID":
"647c12443e91faf0fd508b6edfe59c30b642abb60dfab890b4bdccee38750bc1",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
```

```
        },
        "94447ca479852d29aeddca75c28f7104df3c3196d7b6d83061879e339946805c": {
            "EndpointID":
"b047d090f446ac49747d3c37d63e4307be745876db7f0ceef7b311cbba615f48",
            "MacAddress": "02:42:ac:11:00:03",
            "IPv4Address": "172.17.0.3/16",
            "IPv6Address": ""
        }
    },
    "Options": {
        "com.docker.network.bridge.default_bridge": "true",
        "com.docker.network.bridge.enable_icc": "true",
        "com.docker.network.bridge.enable_ip_masquerade": "true",
        "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
        "com.docker.network.bridge.name": "docker0",
        "com.docker.network.driver.mtu": "9001"
    }
  }
]
```


Contenedores en la red por defecto pueden comunicarse entre ellos usando diferentes IPS siempre y cuando la configuración de Docker lo soporte.

Nota: Docker no ofrece soporte para un service discovery automático cuando usamos la network bridge, sí quieres que tus contenedores puedan ser accesibles unos a otros debes de hacer uso de los Docker links.

docker network ls

Muestra un listado de las networks que Docker tiene creadas (por defecto, o no), inicialmente son tres.

```
$ docker network ls
NETWORK ID        NAME              DRIVER
9f104ee27bf5      none              null
cf23ee007fb4      host              host
7fsa4eb8c647      bridge            bridge
```

docker network create

Te permite crear tus propias networks: bridge o overlay. También es posible crear tus propias networks or plugins de red. Contenedores pueden comunicarse dentro de su network pero no a través de networks. Contenedores puede pertenecer a más de una red.

Vamos a crear una docker network de tipo bridge:

```
$ docker network create --driver bridge webvinar-docker
c5ee82f76de30319c75554a57164c682e7372d2c694fec41e42ac3b77e570f6b

$ docker network inspect webvinar-docker
[
    {
        "Name": "webvinar-docker",
        "Id": "c5ee82f76de30319c75554a57164c682e7372d2c694fec41e42ac3b77e570f6b",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
```

```
            {}
          ]
        },
        "Containers": {},
        "Options": {}
    }
]

$ docker network ls
NETWORK ID         NAME               DRIVER
9f104ee27bf5       none               null
cf23ee007fb4       host               host
7fsa4eb8c647       bridge             bridge
c5ee82f76de3       webvinar-docker    bridge
```

Una vez la red fue creada correctamente podemos empezar a utilizar usando el parámetro **--net**.

```
$ docker run --net=webvinar-docker -itd --name=test busybox
885b7b4f792bae534416c95caa35ba123f201fa181e18e59beba0c80d7d77c1d
```

Cuando creas una network esta recibe una subnet que no ha sido utilizada todavía. Sin embargo, también es posible definir esa subnet usando el **parámetro --subnetwork**. Cuidado con usar la misma subnet para diferentes networks.

En una bridge network puedes solo una subnet mientras que en overlay networks puedes crear varias. Otro parámetro que se puede utilizar es **--gateway-ip-range** y **--aux-address**

$ docker network inspect webvinar-docker

Los contenedores dentro de esta red se podran comunicar entre sí e asimismo ellos estarán aislados del resto de networks. Para permitir a los contenedores comunicarse entre sí y entre otros contenedores, es posible publicar los puertos de los contenedores a nivel de la network. Esto facilita que tus contenedores puedan ser utilizados por otras redes.

Para crear networks con una gran cantidad de contenedores es recomendable utilizar el tipo de network llamado overlay.

$ docker network create --driver overlay webvinar-overlay-net

docker network connect/disconnect

Podemos conectar los contenedores una vez creados o cuando los desplegamos con **--net=NETWORK_NAME.**

$ docker network connect/disconnect NETWORK_NAME CONTAINER_NAME

Vemos el contenedor asignado a la red:

$ docker network inspect NETWORK_NAME $ docker inspect --format='{{json .NetworkSettings}}' CONTAINER_NAME | jq '.'

Ahora CONTAINER_NAME tendrá dos interfaces de red, una para la red creada y otra para la por defecto.

docker network rm

Cuando todos los contenedores de un red están parados o desconectados de la misma. Entonces, podremos borrar nuestra network usando el siguiente comando:

$ docker network rm NETWORK_NAME

COMUNICACIÓN ENTRE HOSTS

Con el uso de Docker networking v1.9 podemos definir overlay networks que nos permiten comunicar contenedores a través de hosts. Para construir una network de tipo overlay, Docker hace uso de libnetwork para construir una VXLAN overlay network y la librería libkv. Para mantener control de toda la información de la red, Docker necesita tener configurado un sistema de almacenamiento clave/valor.

Hasta el momento Docker soporta etcd, Zookeeper y Consul para el almacenamiento de la información de red.

Pasos para tener funcionando una red de tipo overlay:

1.Tener un sistema de almacenamiento configurado y soportado por Docker.

2.Tener Docker instalado en cada máquina y configurado con los siguientes parametros que permitiran al sistema descubrir todos los nodos que lo forman e intercambiar la información necesaria.

```
+--- Parámetro        -------------| Description -------------------------------------------------

  --cluster-store=PROVIDER://URL  | Especifica la localización del sistema de almacenamiento clave/valor

-------------------------------+-----------------------------------------------------------------

  --cluster-advertise=HOST_IP           | Pública los contenedores registrados en este host

---------------------------------------------------------------------------------------------------
```

Un ejemplo de como añadir un contenedor a una network overlay sería:

$ docker run -itd --net=webvinar-overlay-net busybox

VENTAJAS DE DOCKER NETWORKING V1.9

A continuación se detallan algunas de las ventajas que ofrece Docker v1.9 en su network plugin:

- Fácil de utilizar, sin necesidad de ser un experto en redes.
- Basado en estándares: OVS (Open Virtual Switching) and VXLAN para la encapsulación.
- Ilimitado, usuarios pueden crear tantas redes como necesiten.
- Incluye service discovery usando Docker Swarm.
- Es posible usar otros drivers para la creación y gestión de una network, por ejemplo **--driver weaveo --driver calico**

CONCLUSION

Todavía está algo inmaduro, se prevéen muchos cambios en la librería de red (libnetwork). Falta por definir como podría funcionar con otros sistemas de orquestación de contenedores como Kubernetes y Mesos.

Además uno de los problemas es el uso de una base de datos para almacenar la información de red lo que puede reducir el grado de disponibilidad y tolerancia a fallos de nuestro sistema.

Otro es el hecho de usar /etc/hosts como service discovery en bridge networks, cada contenedor en la red crea una entrada lo que puede crear problemas de escalabilidad.

OTROS DRIVERS PARA LA RED

Weave

Weave es independiente de la topología de red que utilices. Esta compuesto de varios componentes que se instalan en cada host:

- weave: El daemon que se encarga de todo la gestión de la red a nivel de cada host.
- weave-dns: Herramienta para crear dominios que pueden ser accedidos desde cualquier contenedor.
- weave-proxy: Crea un proxy que engloba y substituye el proxy de Docker para la comunicación entre Docker hosts.
- weave-scope: Herramienta para visualizer la topología que forman todos los contendores.

Algunos de los aspectos a destacar de Weave:

- De los primeros en ofrecer soporte para la comunicación entre hosts.
- Simple, weave routers se retro alimentan de la información de otros weave router en otros hosts. Esto les permite tener conocimiento de los contenedores y hosts con los que pueden comunicarse.
- La información de red está distribuida a través de todos los nodos que componen la Weave network. Esto mejora la tolerancia a fallos.
- Sistema de encriptado incorporado, aunque caro a nivel de performance.
- Es capaz de hacer tunneling incluso a través de cortafuegos.
- Service discovery usando weave DNS.
- Usa NAT multicast y MTUs.
- Simple DNS load balancing (round-robin), usand weave-dns.
- Funciona en Giant Swarm, Kubernetes, Mesos.
- Permite tener una visión de la topología de red y donde estan tus contenedores usando Weave-scope.
- Uso de Gossip protocol para compartir la información de red y la reglas de enrutado.

Desventajas de Weave:

- No es el más rápido de todos los drivers de networking. Aunque ha mejorado su performance recientemente.

Flannel

Desarrollado por CoreOS y pensado para Kubernetes con el concepto de subnets para pods. Al igual que Docker networking utiliza una base de datos clave/valor, Flannel hace uso de etcd para almacenar toda la información de red.

Algunos aspectos a destacar de Flannel son:

- Proporciona la posibilidad de crear una network overlay (L3)
- Una subnet CIDR (enrutamiento entre dominios sin clases) por host (como con Kubernetes)
  - Host A: 11.0.47.1/24
  - Host B: 11.0.87.1/24
- No hay necesidad de hacer mapeos estáticos de puertos y IPs
- Usa etcd para salvaguardar la información de la network
- Contenedores hablan via direcciones IP
- Uso de IPSEC como protocolo de encapsulación
- Ofrece dos tipos de mecanismos para encapsular los paquetes:
  - UDP
  - VxLAN

- Ofrece drivers para configurar distintos tipos de networks: VxLAN, UDP, alloc, host-gw, aws-vpc.

Calico

- Ofrece un modelo de networking parecido al de Internet (L2-L3)
- Permite definir perfiles ACLs para los contenedores. Incrementa la seguridad entre contenedores ya que no es necesario que los contenedores creen iptables si usan links entre ellos.
- Usa BGP para compartir la información de enrutado entre todos los hosts que componen el cluster de Calico.
- Escala bastante bien con una gran cantidad de contenedores.
- Uno de los mejores a nivel de Performance por detrás de Flannel.

NETWORKING SERVICE ORCHESTRATION

LibChan: es una librería para definir la forma de comunicar distintos contenedores entre sí. Podría ser considerado como una librería para implementar el nivel de comunicación de RabbitMQ.

Docker storage

DOCKER STORAGE DRIVERS

Docker ofrece la posibilidad de usar diferente sistemas de ficheros para el almacenamiento de los contenedores y toda la información necesaria para correr Docker en una máquina. Estos son:

- aufs
- btrfs
- devicemapper
- vfs
- overlayfs
- zfs (recientemente zfs)

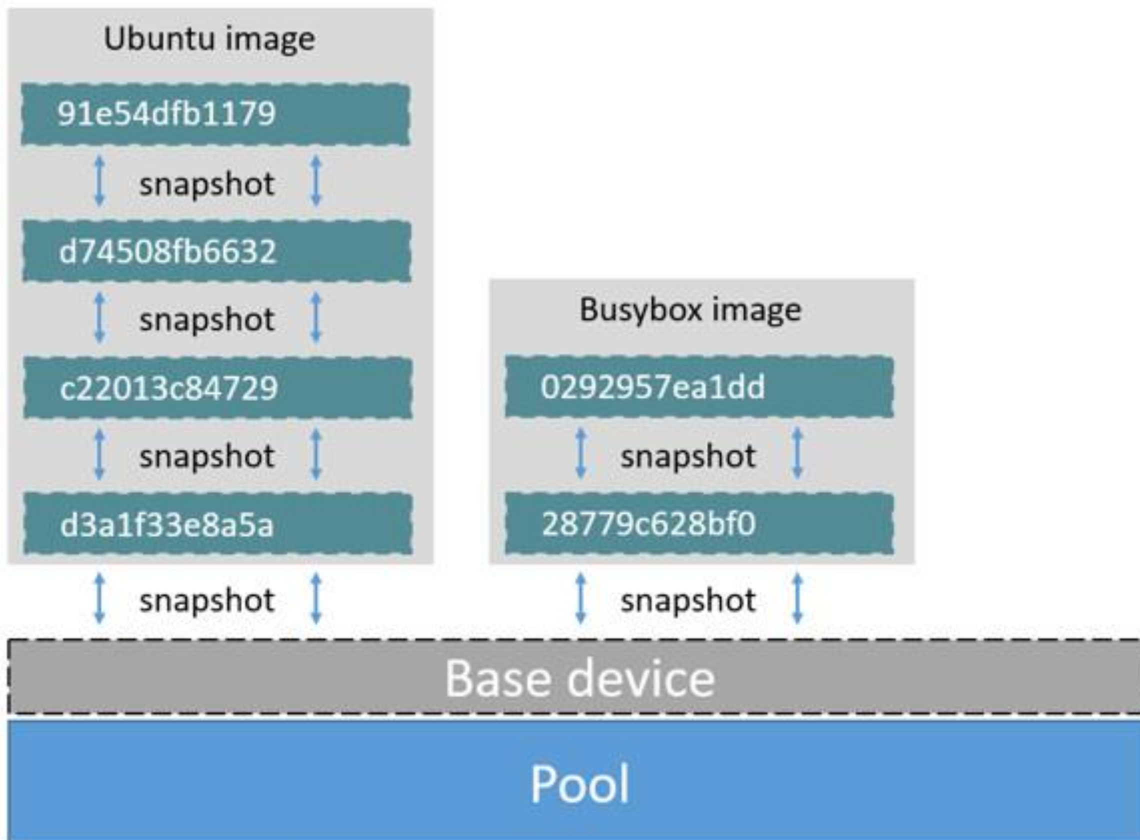
Para utilizar cualquiera de estos sistema de ficheros necesitamos pasar el siguiente argumento cuando arranquemos nuestro Docker daemon '–storage-driver='.

$ sudo docker daemon --storage-driver=STORAGE_DRIVER &

Aufs: Este sistema hace uso del sistema de ficheros Aufs union. Este sistema no está soportado en la mayoría de distros y por lo tanto no está recomendado su uso en producción.

Este almacena cada docker layer como un directorio normal y corriente, conteniendo ficheros y metadata de aufs. Este sistema de ficheros combina todos los layers en un unico mountpoint. Funciona como una pila donde cualquier cambio en este mountpoint va encima del primer layer.

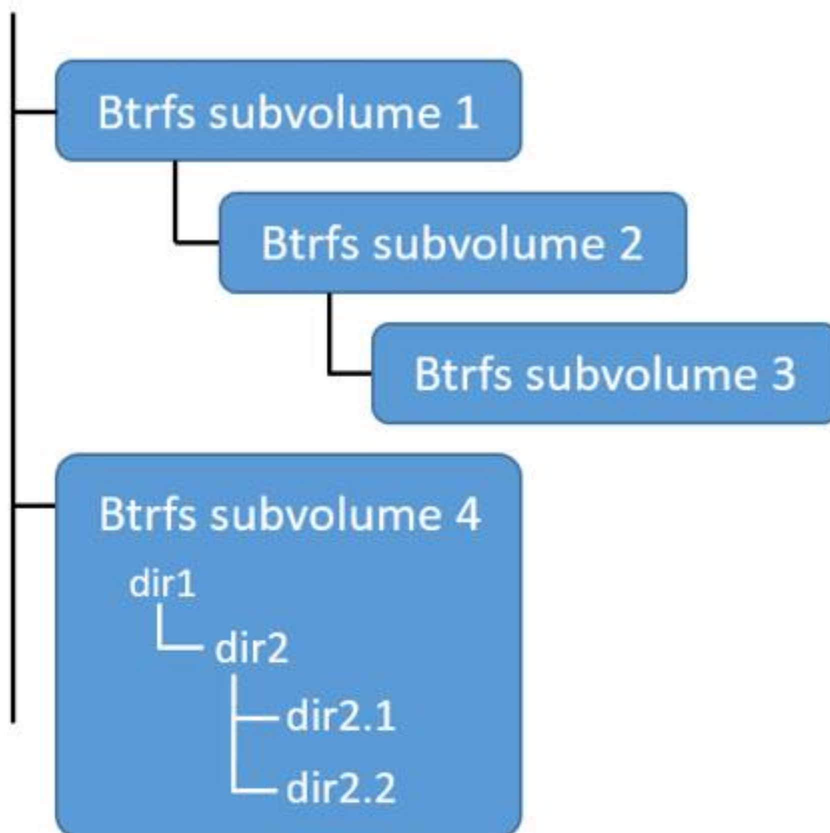
- **/var/lib/docker/aufs/mnt/diff** – donde la imagen layer y su contenido son almacenadores.
- **/var/lib/docker/aufs/layers/ –** metadata sobre como los layers están apilados (organizados).
- **/var/lib/docker/aufs/mnt/<container-id>** – donde los contenedores son almacenados.



Típica imagen describiendo el uso de Aufs para contenedores en Docker

Btrfs: Este sistema usa snapshotting en el sistema de ficheros para crear los layers de las imagenes Docker. Este requiere, al igual que overlay, tener el directorio '/var/lib/docker' sobre un sistema de ficheros de tipo btrfs. Cada layer es almacenado como un subvolumen dentro del directorio '/var/lib/docker/btrfs/subvolumes' y en la forma de snapshot de un subvolumen padre. La siguiente imagen ilustra un poco mejor cómo funciona este sistema de ficheros.

- **/var/lib/docker/btrfs/subvolumes** – donde los volumes son visibles como un sistema ficheros
- **$ sudo btrfs subvolume list /var/lib/docker**



En comparación con otros este sistema de ficheros es muy rápido pero tiene bastantes problemas de estabilidad debido a su escasa madurez.

Devicemapper: Este sistema de ficheros usa el device-mapper (dm-thinp) para crear layers. Device-mapper es la parte del kernel de linux llamada LVM2 volumenes lógicos del sistema. Es un sistema basado en bloques copy-on-write que basicamente usa dos bloques, uno para datos y el otro para metadatos de los dispositivos. Devicemapper crea un pool que puede ser usado para crear otros bloques a partir de este pool. Estos bloques que usa el pool están inicialmente vacíos y las partes no usadas no son reservadas para el uso de este bloque. Permite copy-on-write snapshotting de un dispositivo para crear uno nuevo.

Docker crea un dispositivo base en el pool conteniendo un sistema de ficheros ext4 vacío. El resto de layers seran snapshots del layer base. El sistema de ficheros tiene un tamaño fijo (por defecto 10Gb) con lo cual todo los containers y imágenes tienen un tamaño máximo.

El siguiente comando permite ver todos los dispositivos que devicemapper ha creado.

- **$ sudo lsblk**

Lista de directorios destacables:

- **/var/lib/docker/devicemapper/devicemapper** – contiene data y metadata bloques devices creados en el pool usando la técnica de loopback.
- **/var/lib/docker/devicemapper/devicemapper/json** – contiene la información que permite mapear los layers con los ids en el pool.

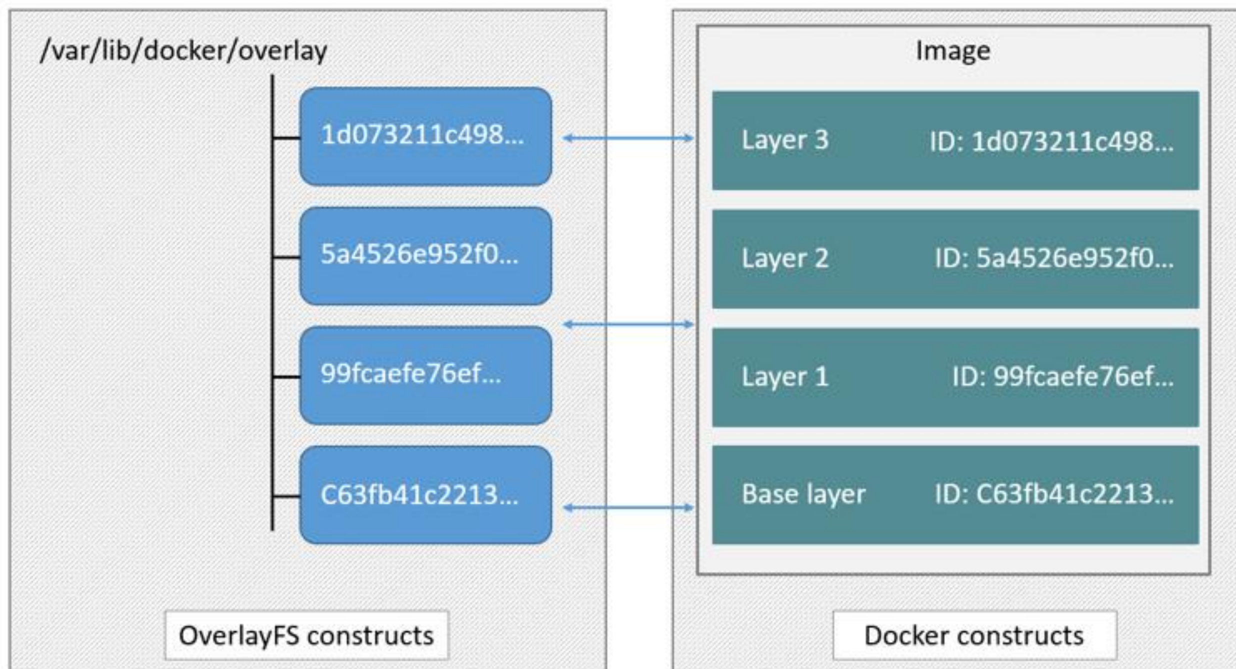
Vfs: Este sistema de ficheros usa un sencillo mecanismo de fallback que no soporta copy on write. Cada layer is un directorio a parte, cuando creamos a nuevo layer otro layer es creado como un clon del primero en un nuevo

directorio. La creación de layer es muy costosa a nivel de performance aunque es uno de los más robustos que funciona en cualquier sistema operativo. Este no comparte el uso de disco compartido entre layers lo cual es un problema a la hora de gestionar los layers de una imagen Docker.

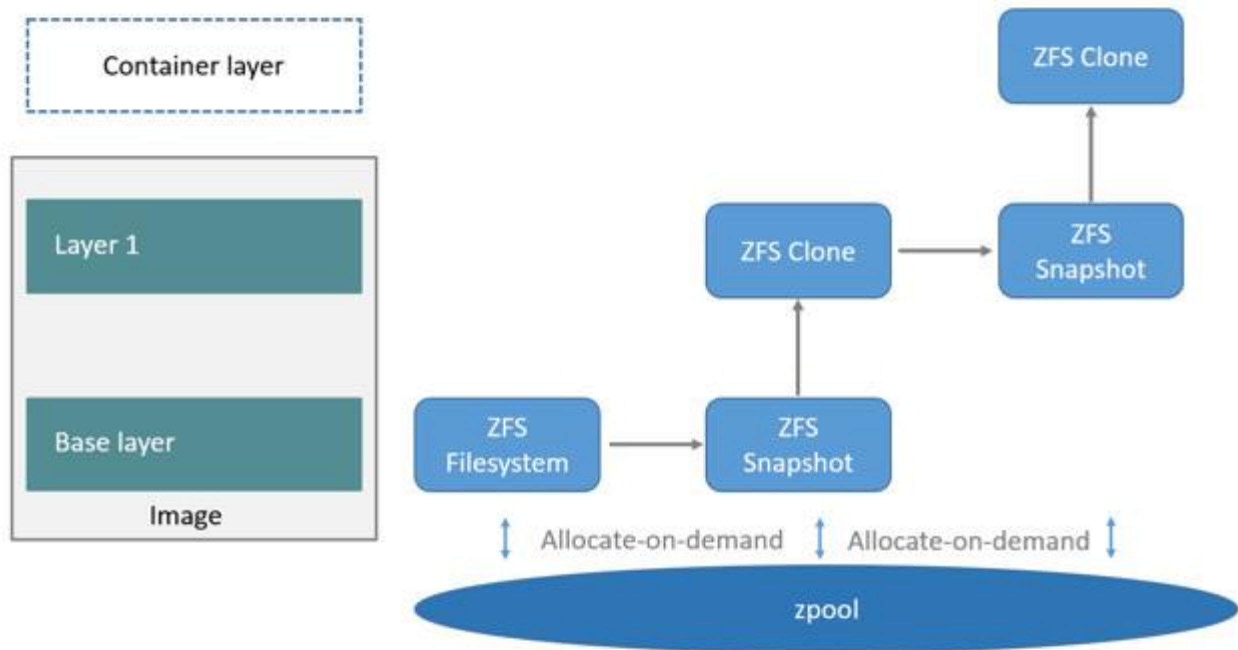
OverlayFS: Es visto como el sucesor de aufs aunque todavía es algo inmaduro para ser adoptado en producción. OverlayFS maneja un concepto basado en dos directorios en el cual uno se apoya sobre el otro mostrando una visión unificada de los layers de una imagen Docker. Para conseguir este efecto OverlayFS usa una tecnología llamada union mount. Los nombres de los directorios son **upperdir, lowerdir** mientras que la versión unificada es expuesta como un directorio llamada **merged.**

$ ls -l /var/lib/docker/overlay/

En OverlayFS cada image layer representa un directorio en la **/var/lib/docker/overlay.** Para crear un contenedor, overlay combina el directorio que representa el último layer con un nuevo directorio para el contenedor. Así el layer de la imagen es el 'lowerdir' en modo lectura, y el nuevo directorio para el contenedor es el 'upperdir' en modo lectura/escritura.



Zfs: Este eficiente sistema de ficheros combina snapshotting y clonado de layers. Las snapshots tiene acceso de lectura y los clones tienen acceso de lectura/escritura. Clones solo pueden ser creados a partir del snapshot de un layer. En Docker una imagen base forma un sistema de ficheros ZFS, donde sus hijos son clones de un snapshot del layer situado más abajo. Consequentemente, un contenedor es un clon basado en una snapshot del layer situado en la cola de la imagen a partir de la cual fue creado.



¿CÓMO SE CREA EL NAMESPACE DE UN CONTENEDOR?

Cuando ejecutamos Docker run este llama al Docker daemon que a su vez invoca **execdriver p**ara inicializar el contenedor. Esta inicialización es ejecutada por **libcontainer.** En Linux, libcontainer llama a **LinuxStandardInit** que lanza **setupRootfs** para crear el namespace inicial para un contenedor.

Esta imagen ofrece una visión de las operaciones que tienen lugar cuando creamos un contenedor independientemente del sistema de ficheros utilizado.

DOCKER VOLUMES

Docker volumes abstraen la vida de los datos que son almacenados en ellos de la vida de contenedor que los crea. Docker volumes permiten almacenar información en un contenedor fuera del boot volume (pero comprendido en el mismo sistema de ficheros).

Un contenedor puede ser creado con uno o más volumes usando el parámetro **-v,** el cual creará un scope dentro de la jerarquía de ficheros **docker /var/lib/docker** donde la información será almacenada.

Los datos de configuración de volumes son almacenados en **/var/lib/docker/volumes** en el cual cada subdirectorio representa un nombre de volumen identificado por un UUID. La información es almacenada en el directorio **/var/lib/docker/btrfs|aufs|xvfs|overlay/dir** con un identificador único. La información en los volumes puede ser navegada a través del sistema operativo. También podemos aplicar distintos permisos Unix a los volumenes (readonly ó readwrite).

El uso de volumes tienes sus ventajas y desventajas.

Ventajas:

- Al ser un fichero más en el sistema de operativo este puede ser copiado, movido como otro fichero normal.
- Un volumen puede estar asociado a un directorio en el contenedor, permitiendo así acceder a los datos persistentes del host desde un contenedor.

Desventajas:

- Es dificil de asociar el UUID de un volumen con su contenedor.

Podemos utilizar los volumes para acceder a un sistema de ficheros NFS o LUN accediendo así a almacenamiento distribuido externo lo que aumentaría la tolerancia a fallos.

CONTENEDOR DOCKER PARA DATOS

Otra opción para almacenar la información de los contenedores de una manera persistente es el uso de contenedores para datos. Estos contenedores tienen uno o más volumes asociados al contenedor, estos contenedores de datos pueden ser utilizados (exportados) por otros contenedores usando el parámetro **--volume-from=CONTAINER_DATA_NAME.** Para hacer uso de esta directiva los contenedores tiene que residir en la misma máquina.

Cuando usamos contenedores para datos y estos son compartidos por varios contendores, este escenario se similar al de un Docker NFS server proporcionando acceso a la información desde un contenedor central. Por lo tanto, los contenedores pueden ser creados y destruidos mientras que la información permanece en el contenedor de datos, lo que añade un nivel más de abstracción.

Hay varias aspectos a tener en cuenta cuando usemos este tipo de contenedores en nuestro sistema:

- **Seguridad:** No hay más seguridad que los permisos asociados al volumen como cualquier otro fichero en un sistema Unix. De ese modo, es conveniente controlar el acceso de los usuarios al sistema de ficheros y consequentemente al del volumen.
- **Almacenamiento huérfano:** Borrar todos los contenedores excepto el contenedor de datos.
- **Integridad de los datos:** Compartir contenedores de datos no excluye la necesidad de sistemas de bloqueo para sincronizar todas los operaciones sobre los datos cuando estos están compartidos. Docker no proporciona ningun mecanismo para hacer snapshots o replicar los datos.
- **Movilidad de los datos:** no hay ninguna solución nativa de Docker que permita migrar los datos de un host a otro.

En la actualidad hay varias soluciones que ofrecen la posibilidad de compartir contenedores para datos entre hosts. Algunas de ellas se explican a continuación.

SOLUCIONES DE ALMACENAMIENTO CON CONTENEDORES, SDS PARA DOCKER

En el mercado hay diversas soluciones para ofrecer funcionalidades de portabilidad de volumes, como Quobyte o Flocker entre otros. Asimismo Docker está intentando ofrecer un plugin que pueda cubrir las funcionalidades de estas dos soluciones.

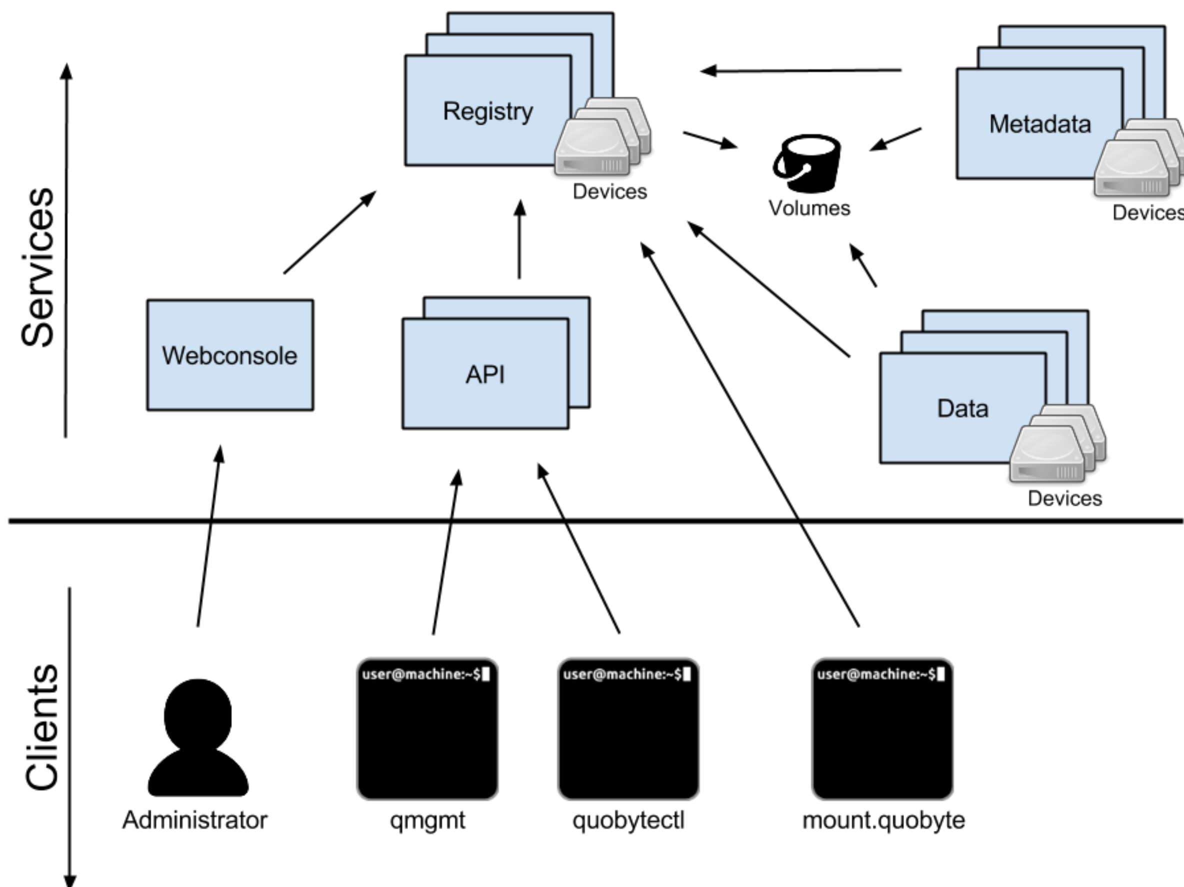
Quobyte

Este sistem distribuido para el almacenamiento de ficheros consiste de diferente componentes que puede residir en la misma maquina o estar distribuidos en un cluster. Los principales componentes son los servicios y los dispositivos. Los servicios son componentes que corren en los diferentes hosts y comunican entre ellos para mantener información sobre el sistema. Por otro lado, los dispositivos son unidades de almacenamiento conectadas al hardware de cada maquina para albergar la información de los servicios y otros componentes de Quobyte.

Para almacenar información, Quobyte ofrece volumes como las unidades de almacenamiento que pueden ser montadas en mountpoints en los hosts.

Cada uno de estos componentes necesita usar el componente registry para saber que nodos están registrados al igual que para mander señales de humo de sí mismo.

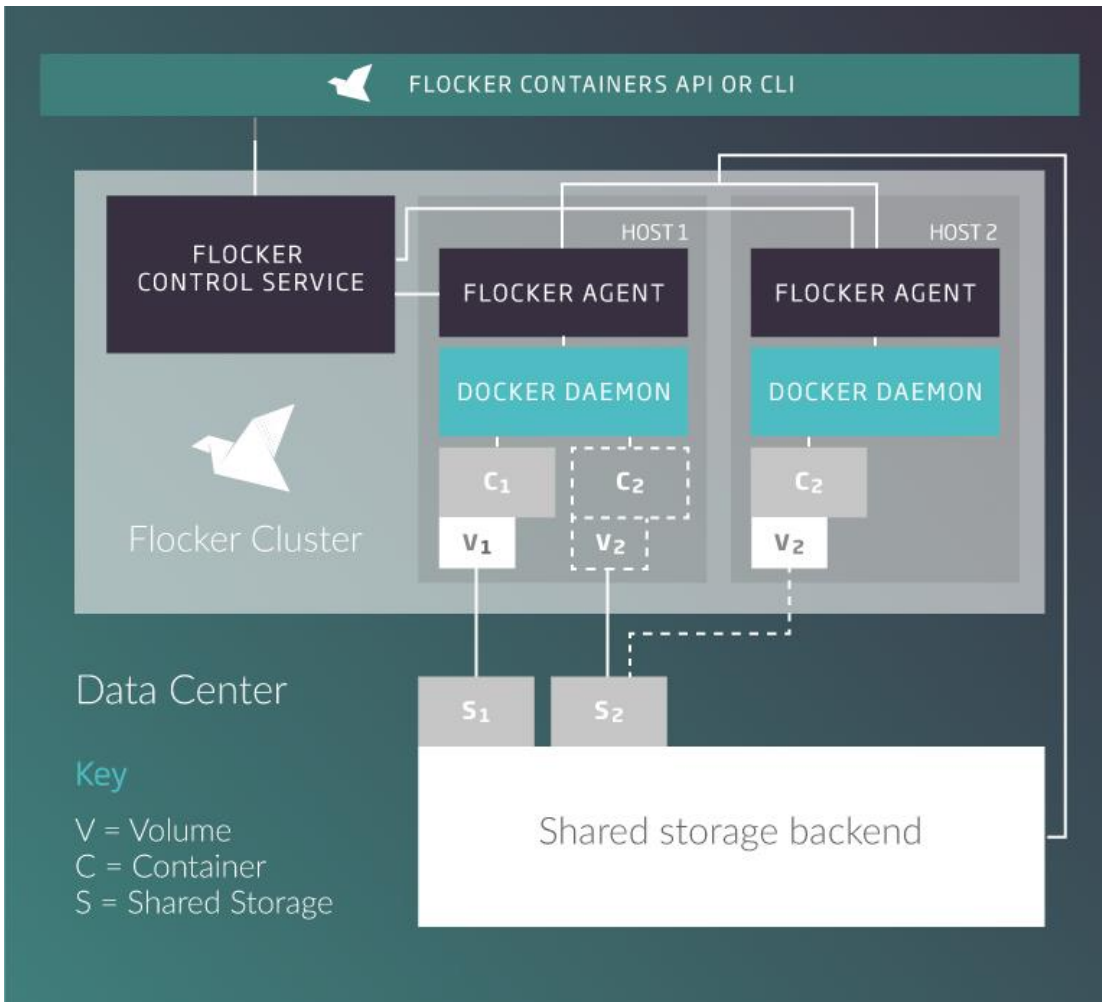
- **Registry:** es el núcleo del cluster deo Quobyte. Es el encargado de mantener el control de todos los dispositivos y services corriendo en el cluster, almacenamiento de la información del sistema, volumes asi como coordinación la localización de los ficheros de datos y metadata de los servicios.
- **Metadata:** Este component almacena metadatos sobre los ficheros, como sus permisos, tamaño.
- **Data:** Este componente almacena los datos de ficheros y hace uso de las operaciones POSIX para escribir y read. Usa dispositivos para replicar la información y migrarla a lo largo del cluster.
- **API:** Expone los services de la API de Quobyte que se usan para obtener información sobre el cluster or para create y administrar los recursos.
- **Webconsole:** es una interfaz web para administrar un cluster.



Flocker

Flocker es una solución para el almacenamiento de datos en Docker containers que proporciona las siguientes características:

- **Gestión de multiple hosts:** Contenedores pueden estar distribuidos entre diversos nodos o contenedores en el mismo host.
- **Sistema de fichero ZFS:** usa este tipo de sistema de ficheros para almacenar los dos y conseguir los mejores resultados con la máxima disponibilidad.



La magia del diseño de Flocker network proxy permite comunicar cualquiera de los contenedores de datos con cualquiera de los contenedores del cluster siempre y cuando los puertos asignados a esos contenedores sean únicos para evitar conflictos.

La configuración de Flocker se sustenta en dos ficheros YAML: application.yaml y deployment.yaml. Estos describen la topología de la aplicación. Uno application.yaml la base de datos, el servidor web y todos los links entre ellos. El otro fichero deployment.yaml contiene una definición de la localización del contenedor de datos dentro del Flocker cluster para saber en que nodo se encuentra.

Estos ficheros son declarativos, y así Flocker ejecutará las operaciones necesarias para aplicar de la configuración deseada. Asimismo Flocker es capaz de mover los contenedores de datos de un nodo a otro simplemente modificando la localización del contenedor en el fichero deployment.yaml usando después las herramientas de Flocker. Este mecanismo de migración de datos en la realidad sigue un proceso bastante sencillo ya que los contenedores son destruidos en el nodo actual y re-creados en el nuevo nodo.

PLUGIN PARA VOLUMENES DOCKER

Existen plugins en Docker para permitir a los contenedores integrarse con sistemas de almacenamiento externo como AWS EBS que persisten los datos del contenedor más alla de la vida del contenedor en sí mismo. También es posible crear tu propio plugin para la gestión de contenedores de datos en Docker.

Un plugin para volumes hace uso de los argumentos **-v** y **--volume-driver=DRIVER_NAME** cuando hacemos uso del comando 'docker run'. Estos plugin requieren un nombre para el volumen que puede ser pasado con el argumento **-v** y con el siguiente **formato -v VOLUME_NAME:/misdatos**. Este nombre permite asociar el volumen con el volumen persistente gestionado por el plugin y así ser capaz de re-utilizarlo cuando sea conveniente. El nombre del volumen no puede empezar por '/'. Uno de los drivers existentes es Flocker:

$ docker run -ti -v VOLUME_NAME:/misdatos --volume-driver=flocker busybox sh

Directivas para crear tu propio Driver

Para ello debemos de pasar un campo **VolumeDriver** cuando llamemos a la ruta de la Docker API (/containers/create) durante la creación de un contenedor. Este campo permite especificar el nombre de driver, es de tipo string, y su valor por defecto es **local.**

Cualquier driver para volumenes en Docker tiene que implementar los siguientes puntos de entrada:

- **/VolumeDriver.Create:** Como comentamos previamente la creación de un volumen necesita el nombre del volumén.
    - Argumento de entrada: 'volume_name'; Argument de salida: 'error'
- **/VolumeDriver.Remove:** Docker necesita el nombre del volumen para borrarlo. - Argumento de entrada: 'volume_name'; Argument de salida: 'error'
- **/VolumeDriver.Path:** Docker necesita saber el directorio donde está el volumen en el host.
    - Argumento de entrada: 'volume_name'; Argument de salida: 'Mountpoint', 'error'
- **/VolumeDriver.Unmount:** Esta operacion se usa para indicar que Docker no va a usar el volumen. Esta operación es llamada cuando usamos **docker stop.** - Argumento de entrada: 'volume_name'; Argument de salida: 'error'

Ejercicios para casa:

1. Crear un contenedor mysql y pasar como volumen un directorio de vuestro host con la directiva **-v.**
2. Crear un contenedor que hace uso de otro contenedor con la directiva **volume-from.**
3. Crear un Flocker cluster de 2 nodes en vuestra cuenta de AWS. Después intenta crear contenedores de datos y moverlos de un host a otro usando la cli de Flocker.

ALGUNOS EJERCICIOS

Volumes

Uso de volumes:

```
docker run -v /tmp -tid --name=vol1 busybox
```

Este comando crea un volumen en la ruta dependiente del tipo de sistema de ficheros.

```
docker inspect vol1

docker run -v /host/logs:/container/logs -tid --name=vol2 busybox
```

En cambio este comando, lo que hace es montar un directorio en un directorio del contenedor. Este uso es muy común y bastante utilizado para realizar tareas de administración.

Directiva volumes-from

Crea un volumen para datos y úsalo en otro contenedor.

```
docker run -v /tmp -tid --name=vol1 busybox

docker exec -ti vol1 sh

// Crea un directorio y un fichero.

docker run --volumes-from=vol1 -tid --name=test busybox

docker exec -ti test sh

// Lista el contenido de /tmp
```

Uso de volumes para otros propositos

El uso de volumenes también nos permite pasar sockets, binarios. Un uso no muy seguro pero normal es pasar a un contenedor el socket the Docker como un volumen **/var/run/docker.sock**. Esto nos permite acceder al socket de Docker y hacer llamadas a su API dentro del contenedor.

```
docker run -tid -v /var/run/docker.sock:/var/run/docker.sock -p 25565:25565 --name dockercraft
gaetan/dockercraft
```

Creacion de backups

```
$ docker run --rm --volumes-from mydb -v $(pwd):/backup debian tar cvf /backup/mybackup.tar
/var/lib/mysql/data
```

CREAR TU NFS SHARE

Como root del sistema ejecuta los siguientes commandos:

```
apt-get install nfs-kernel-server

mkdir /export

chmod 777 /export


mount --bind /home/vagrant/info_docker /export
```

En **/etc/fstab** añade la siguiente linea:

```
/home/vagrant/info_docker /export none bind 0 0
```

En **/etc/exports** añade la siguiente linea:

```
/export       127.0.0.1(rw,fsid=0,no_root_squash,insecure,no_subtree_check,async)


service nfs-kernel-server start


mkdir -p /test

mount -t nfs 127.0.0.1:/export /test


exportfs -a


service nfs-kernel-server restart



docker run -tid --name nfs_cli --privileged -v /test:/test busybox
```

```
docker run -tid --volumes-from nfs_cli --name=vol_test busybox
```

Prueba como se asignan permisos en volumenes:

```
FROM debian:jessie


RUN useradd hector

VOLUME /misDatos

RUN touch /misDatos/holaMundo && chown -R hector:hector /misDatos
```

Comprueba si holaMundo ha sido creado en /misDatos. ¿No, verdad?

En Docker cualquier cosa después de VOLUME no aplicará los cambios sobre ese volumen, piensa en LAYERS. Lo que pretendemos es que holaMundo se escriba en la imagen del sistema de ficheros pero lo que est'a haciendo es correr en el volumen del contenedor temporal.

Docker es inteligente y copia todos los ficheros que existen en una imagen sobre un volumen y establecen los privilegios necesarios.

Ahora prueba con esto:

```
FROM debian:jessie


RUN useradd hector

RUN mkdir -p /misDatos && touch /misDatos/holaMundo && chown -R hector:hector /misDatos

VOLUME /misDatos
```

OTRAS REFERENCIAS O ENLACES INTERESANTES

Issue: Capability to specify per volume mount propagation mode
(https://github.com/docker/docker/pull/17034#issuecomment-164443643)

- https://github.com/gondor/docker-volume-netshare
- https://github.com/docker/docker/issues/4213
- EMC ScaleIO: https://github.com/djannot/scaleio-docker
- https://docs.docker.com/engine/userguide/storagedriver/btrfs-driver/
- https://github.com/gondor/docker-volume-netshare
- https://dzone.com/articles/mount-aws-efs-nfs-or-cifssamba-volumes-in-docker

Monitoring en Docker

MONITORING DE CONTENEDORES

El ecosistema de los contenedores en general es:

- Desconocido.
- Inestable.
- Repleto de tecnologías bastante inmaduras.

El Reto: entender y aprender patrones en los que los contenedores fracasan.

De ahí que el Monitoring se vea como un mecanismo para aprender y anticipar a fallos.

Infraestructura para monitoring de contenedores

Una infraestructura para monitoring contenedores requiere:

- Un sistema de monitoring basado en contenedores.

Estos sistemas presentan una arquitectura jerárquica:

- Un sistema central monitoreo los clusters.
- Cada host del cluster monitoreo sus contenedores.

Las características de una infraestructura para monitorear contenedores:

- Ligera.
- Alta disponibilidad.
- Escalable.
- Contenerizada.
- Distribuida.
- Capaz de reducir el ruido para evitar distracción.
- Que tenga más código para analizar métricas que para coleccionarlas.

Las partes de una infraestructura de este tipo son:

- Logs
- Colección y almacenamiento de datos.
- Exploración del sistema.
  - Alertas o Notificaciones.
  - Análisis de los datos.

Logs

Es necesario visualizar los logs para saber que esta pasando en mí aplicación.

- Coleccionarlos.
- Procesarlos.
- Almacenarlos.

Docker ofrece un comando muy sencillo para mostrar los logs en stdout:

$ docker logs CONTAINER_NAME

Docker ofrece la posibilidad de utilizar diferentes plugins para redireccionar los logs:

- Json-file
- Fluentd.
- Journald.
- Syslog.
- Awslog.
- Gelf.

Este plugin hay que configurarlo cuando arranquemos el Docker daemon:

$ docker daemon --logs-driver=DRIVER_NAME

Logs

Una de las arquitecturas tradicionales y efectivas es el uso y almacenamiento de logs es:

- Elasticsearch.
- Logstash.
- Logstash-forwarder.
- Kibana.

Logs – Consejos

Una de las arquitecturas tradicionales y efectivas es el uso y almacenamiento de logs es:

- Descubrir lo deconocido a través de los logs.
- Minimizar el uso de **docker logs** ya que es CPU intensive.
- Usar containers para los componentes de ELK.
- Use logging levels (DEBUG, INFO, ERROR, etc…).

Colecciona y almacena métricas

Tres niveles de métricas son almacenados y analizados.

Uso de contenedores para instalar y correr estas aplicaciones:

- Influx DB
- Grafana
- cAdvisor

Colecciona y almacena métricas – Consejos –

- El uso de InfluxDB puede ser problemático, uso de CPU, no estable.
- cAdvisor puede ser CPU intensive.
- Distinguir entre métricas a nivel de cluster, host y contenedor.
- Limitar los recursos de los contenedores a nivel de CPU y Memoria.

Exploración del Sistema

- Prevenir y identificar patrones de errores antes de tiempo.
- Definir alertas de una manera jerárquica.
    - Cluster => Hosts => Contenedor

Principales herramientas:

- Riemann (stream processing system)
- Sensu (monitor router) - Host and cluster alert clients
- Comunicación:
    - Slack
    - Mailgun
    - OpsGenie

Otras herramientas usadas para explorar tu sistema (a nivel de host):

- Sysdig
- htop
- strace
- tcpdump
- sysstat
- tcpflow
- perf_events
- iotop
- …

Exploración del Sistema – Consejos –

Otras herramientas usadas para explorar tu sistema (a nivel de host):

- Uso de Riemann (cadvisor plugin for riemann)
- Evalua la latencia/ancho de banda

- Docker tareas de limpieza:
    - Images se apilan en el sistema de ficheros.
    - Contenedores se apilan en el sistema de ficheros.
- Uso del canal de comunicaión más adecuado para cada alerta (los emails no se leen).
- Healthchecks para contenedores de usuario o los nuestros propios.
- El ancho de banda difiere entre hosts y contenedores.
- Errores en el almacenamiento (AWS vols, quotas, movimiento de datos).
- Corrupción del sistema de ficheros.
- Mapeo de puertos con Docker (port binding, OOM, btfrs/overlay (readonly)).
- ICMP, SNMP pings.

Conclusiones

- Usa soluciones que esten contenerizadas. :D
- Usa las herramientas que mejor se adaptan a tu problema.
- Colecciona métricas para prevenir y entender los desconocido.
- Analiza los datos coleccionados para detectar patrones en los fallos.

SOLUCIONES DEL MERCADO

Prometheus

Creado por ex-Googlers en 2012, pertenece a los creados de SoundCloud.

Es un sistema para el monitoreo y alertas para sistemas distruidos.

Las características de Prometheus son:

- Opensource
- Almacenamiento usan LevelDB.
- Visualización de los datos coleccionados.
- Capacidad para crear alertas.

\includegraphics[width=0.8\paperwidth,height=0.7\paperheight]{prometheus-architecture.png}

Los componentes de Prometheus son:

- **Server:** nucleo de esta arquitectura, coordina y se encarga del almacenamiento. Proporciona herramientas para analizar los datos.
- **Node Exporter:** coleciona métrica de diferentes componentes o tipos (Node, cAdvisor, Redis).
- **Push Gateway:** transmite información para jobs de corta vida.
- **Alert Manager:** permite definir alertas con distintos niveles y integraciones.
- **Prom Dash:** dashboard para visualizar los datos (similar a Grafana).
- **Service Discovery:** require un sistema para descubrir los componentes (SkyDNS, etcd, Consul).

Las DESVENTAJAS de Prometheus son:

- Uso del módelo Pull-Push.
- No tiene un mecanismo para replicar los datos entre host.
- Uso de un lenguaje propio para la definición de alertas y queries.
- Depende de un service discovery para funcionar.

Sysdig

Es una herramienta opensource para analizar el comportamiento del sistema.

Captura, analize y permite filtrar la información almacenada.

Permite tener una visión gráfica del cluster incluyendo los contenedores.

Las características de Sysdig son:

- SaaS
- Análisis de los datos.
- Predicciones.
- Agente instalado en tu máquina que require permisos privilegiados.
- Capacidad para crear alertas.
- Orientado a contenedores.
- Muy rápido, instala un módulo en el kernel.

Las desventajas de Sysdig son:

- De pago.
- Datos son dinero.
- Consumo de red y recursos.

Datadog

Es una herramienta para monitorear aplicaciones que corren en un máquina o cluster.

Permite visualizar el performance de tus contenedores.

Las características de DataDog son:

- SaaS
- Análisis de los datos.
- Predicciones.
- Agente instalado en tu m'aquina que require permisos privilegiados.
- Ellos consumen y almacenan tu información.
- Correlaciona el performance de los contenedores con las aplicaciones corriendo en ellos.
- Capacidad para crear alertas.

Las desventajas de DataDog son:

- De pago.
- Datos son dinero.
- Consumo de red y recursos.

EJERCICIOS

cAdvisor la herramienta de monitoring

Esta herramienta te permite monitorear las métricas de los contenedores de una manera sencilla y práctica:

```
$ 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
```

Sysdig

Instala este agente en tu docker machine y accede a la informacion desde sysdig cloud.

```
$ docker run --name sysdig-agent --privileged --net host --pid host -e ACCESS_KEY=YOUR_KEY -e TAGS=openwebinars:hector,location:spain -v /var/run/docker.sock:/host/var/run/docker.sock -v /dev:/host/dev -v /proc:/host/proc:ro -v /boot:/host/boot:ro -v /lib/modules:/host/lib/modules:ro -v /usr:/host/usr:ro sysdig/agent
```

Herramienta opensource de sysdig para un solo host:

```
$ docker run --rm -it --name sysdig-$(uuidgen) --privileged -v /var/run/docker.sock:/host/var/run/docker.sock -v /dev:/host/dev -v /proc:/host/proc:ro -v /boot:/host/boot:ro -v /lib/modules:/host/lib/modules:ro -v /usr:/host/usr:ro sysdig/sysdig csysdig
```

Datadog

Instala este agente en tu docker machine y accede a la informacion desde datadog.com

```
$ docker run -d --name dd-agent -hhostname-v /var/run/docker.sock:/var/run/docker.sock -v /proc/:/host/proc/:ro -v /sys/fs/cgroup/:/host/sys/fs/cgroup:ro -e API_KEY=YOUR_KEY datadog/docker-dd-agent:latest
```

Prometheus

Simple manera de correr prometheus:

```
$ docker run -p 9090:9090 prom/prometheus
```

PromDASh

Herramienta parecida a Grafana pero para Prometheus:

```
$ docker run -p 3000:3000 -v /tmp/prom:/tmp/prom -e DATABASE_URL=sqlite3:/tmp/prom/file.sqlite3 prom/promdash
```

```
$ docker run --rm -v /tmp/prom:/tmp/prom -e DATABASE_URL=sqlite3:/tmp/prom/file.sqlite3 prom/promdash ./bin/rake db:migrate
```

Nota

En su medida intenta usar contenedores para todas tus aplicaciones de monitoring: strace, htop, etc…

```
$ docker run --rm -i realguess/jq:latest jq
```

Resumen Arquitectura en Docker

2005: OPEN VZ (OPEN VIRTUZZO)

Se trata de una tecnología de virtualización a nivel de sistema operativo que utiliza un kernel Linux parcheado para virtualización, aislamiento y checkpointing.

El código no fue lanzado como parte del kernel oficial de Linux.

2006: PROCESS CONTAINERS

Llanzado por Google en 2006.

Fue diseñado para limitar, contabilizar y aislar el uso de recursos (CPU, memoria, I/O de disco, red) de una colección de procesos.

Fue renombrado como «Grupos de control (cgroups)» un año después y finalmente se fusionó con el kernel 2.6.24 de Linux.

2008: LXC

LXC (LinuX Containers) fue la primera y más completa implementación del gestor de contenedores Linux.

Se dió en el año 2008 utilizando cgroups y namespaces de Linux, y funcionaba en un solo kernel de Linux sin necesidad de parches.

2011: WARDEN

CloudFoundry comenzó Warden en 2011, utilizando LXC en las etapas iniciales y posteriormente lo reemplazó con su propia implementación.

2013: LMCTFY

Let Me Contain That For You (LMCTFY) se lanzó en 2013 como una versión de código abierto de los contenedores de Google, proporcionando contenedores de aplicaciones Linux.

HISTORIA DE DOCKER

Docker Inc. se creó en un grupo de incubadoras entre 2010 y 2011.

Su producto Docker fue lanzado en marzo de 2013, bajo modalidad opensource.

- En 2013: Red Hat y Docker anunciaron una colaboración entre Fedora, Red Hat Enterprise Linux (RHEL) y OpenShift.
- En 2014: Microsoft anunció la integración del motor Docker en Windows Server.
- En 2014: Se anunciaron los servicios de contenedores Docker para Amazon Elastic Compute Cloud (EC2) y para IBM Cloud.
- En 2016: Microsoft da soporte nativo a Docker en Windows 10.

ECOSISTEMA DOCKER

Docker es un conjunto de piezas de software para trabajar con contenedores:

- Docker Engine: Es un motor de contenedores. Permite la descarga y creación de imágenes, y la ejecución de contenedores. Existen dos distribuciones:
  - Docker CE: Opensource y gratuita. Su sitio oficial es https://www.docker.com
  - Docker EE: Código privativo y de pago. Su sitio oficial es https://www.mirantis.com
- Docker desktop: Para Windows y Mac. Permite realizar las operaciones de docker engine a través de una interfaz gráfica..

DOCKER ENGINE

Docker Engine es una tecnología de contenedorización open source para construir y ejecutar aplicaciones.

Docker Engine actúa como una aplicación cliente-servidor con:

- Un servidor con un proceso de demonio de larga ejecución llamado dockerd. El daemon crea y administra objetos Docker, como imágenes, contenedores, redes y volúmenes.
- Un API (REST) que especifica interfaces que los programas pueden usar para comunicarse con el demonio Docker.
- Un CLI (interfaz de línea de comando) llamado docker, que utiliza las API de Docker para controlar o interactuar con el demonio Docker a través de scripts o comandos de CLI directos.

DOCKER ENGINE

Utiliza la virtualización a nivel del sistema operativo para ejecutar los contenedores.

Los contenedores están aislados unos de otros, pero pueden comunicarse entre sí.

Todos los contenedores son ejecutados por un solo núcleo del sistema operativo y, por lo tanto, usan menos recursos que las máquinas virtuales.

QUE PERMITE DOCKER ENGINE

- Docker tiene la capacidad de reducir el tamaño del desarrollo al proporcionar una huella más pequeña del sistema operativo a través de contenedores.
- Con los contenedores, es más fácil para los equipos de diferentes unidades, como desarrollo, control de calidad y operaciones, trabajar sin problemas en todas las aplicaciones.

CARACTERÍSTICAS DE LOS CONTENEDORES

- Estándar: Docker creó el estándar de la industria para contenedores, lo que permite su portabilidad
- Ligeros: los contenedores comparten el núcleo del sistema operativo de la máquina y, por lo tanto, no requieren un sistema operativo por aplicación, lo que aumenta la eficiencia del servidor y reduce los costos de servidor y licencia
- Más seguros: las aplicaciones son más seguras en contenedores por las capacidades de aislamiento de los contenedores

IMÁGENES Y CONTENEDORES DOCKER

Una imagen es un paquete de software ligero, independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación:

- Aplicación
- Herramientas del sistema
- Bibliotecas
- Configuraciones del sistema

Las imágenes se convierten en contenedores en tiempo de ejecución.

Los contenedores aíslan el software de su entorno y aseguran que el software que contienen siempre se ejecutará igual, independientemente de la infraestructura.

ARQUITECTURA DE DOCKER ENGINE

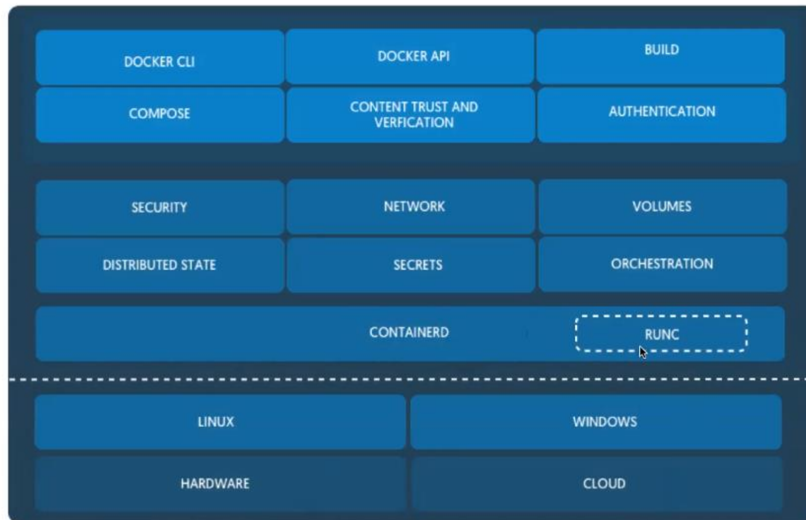
En su origen, Docker se basaba en la tecnología de virtualización a nivel de sistema operativo (SO) para Linux LXC.

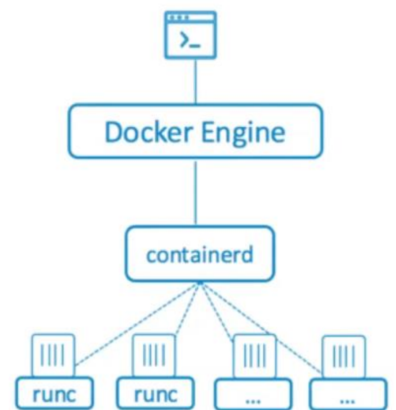
Al año fue sustituida por una tecnología propia de virtualización escrita en el lenguaje de programación Go, llamada libcontainer.

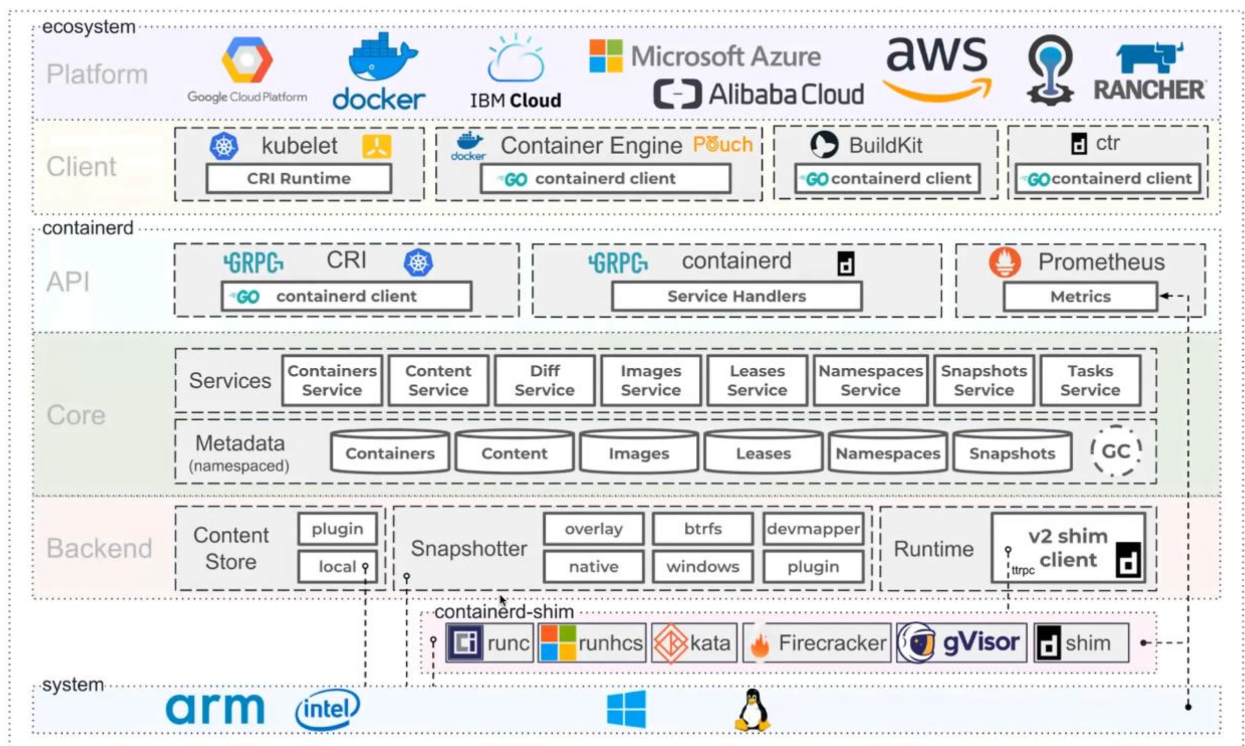
Libcontainer fue donada a la Open Containers Initiative, y hoy en día se denomina runc.

Posteriormente la capa de gestión de imágenes y contenedores fue también desligada y donada a la Cloud Native Computing Foundation, que la distribuye bajo el nombre containerd.

CONTAINERD







Laboratorio 1 Instalación de Docker

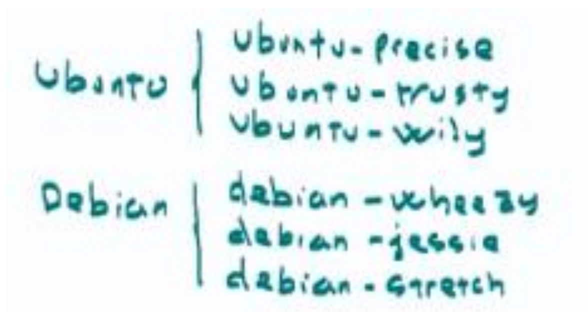
En este laboratorio veremos las diferentes formas de instalar nuestro servidor docker.

https://docs.docker.com/install/#upgrade-path

En Ubuntu/Debian:

- Crear el fichero /etc/apt/sources/list.d/docker.list
- Contenido:

  deb https://apt.dockerproject.org/repo distro main



Ejecutar:

apt-get update

apt-ger install docker-enfine

Comprobar:

systemctl status docker o /etc/init.d/docker status

docker info

docker versión

```
root@debian:/etc/apt/sources.list.d# lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 8.3 (jessie)
Release:        jessie/sid
Codename:       jessie
root@debian:/etc/apt/sources.list.d# cat docker.list
deb https://apt.dockerproject.org/repo debian-jessie main
root@debian:/etc/apt/sources.list.d# apt-get update
0% [Working]
```

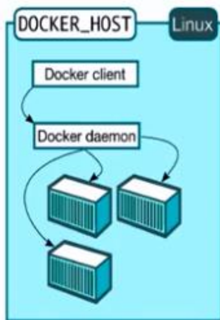
```
root@debian:/etc/apt/sources.list.d# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled)
   Active: active (running) since Sat 2016-02-20 19:55:07 CET; 14s ago
     Docs: https://docs.docker.com
 Main PID: 27148 (docker)
   CGroup: /system.slice/docker.service
           └─27148 /usr/bin/docker daemon -H fd://

Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.527340735+01:00" level=warni...und"
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.528168171+01:00" level=warni...iod"
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.528380217+01:00" level=warni...tas"
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.533598886+01:00" level=info ...rt."
Feb 20 19:55:07 debian docker[27148]: ..........
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.567016564+01:00" level=info ...ne."
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.567116348+01:00" level=info ...ion"
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.567491097+01:00" level=info ...10.1
Feb 20 19:55:07 debian systemd[1]: Started Docker Application Container Engine.
Feb 20 19:55:07 debian docker[27148]: time="2016-02-20T19:55:07.679717000+01:00" level=info ...ock"
Hint: Some lines were ellipsized, use -l to show in full.
root@debian:/etc/apt/sources.list.d#
```

```
root@debian:/etc/apt/sources.list.d# docker info | less
root@debian:/etc/apt/sources.list.d# docker version
Client:
 Version:      1.10.1
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 19:09:42 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.10.1
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   9e83765
 Built:        Thu Feb 11 19:09:42 2016
 OS/Arch:      linux/amd64
root@debian:/etc/apt/sources.list.d#
```

Instalación centos/RHEL7/fedora:

Instalación

- En Centos / RHEL7 / Fedora:



→ Crear el fichero /etc/yum.repos.d/docker.repo

→ Con el contenido

```
[dockerrepo]
name = Docker Repository
baseurl = https://yum.dockerproject.org/repo/main/os
enabled = 1
gpgcheck = 1
gpgkey = https://yum.dockerproject.org/yum
```

Centos: centos/$releasever
RHEL7: centos/7
Fedora: fedora/$releasever

→ Ejecutar:

- Centos / RHEL7: yum install docker-engine
- Fedora: dnf install docker-engine

→ Comprobar:

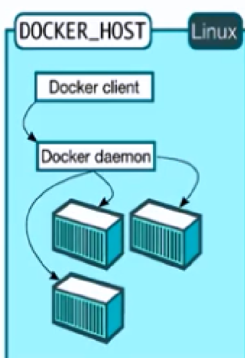
systemctl status docker-engine // /etc/init.d/docker status
docker info
docker version

```
root@centos7:/etc/yum.repos.d$ cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
root@centos7:/etc/yum.repos.d$ cat docker.repo
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
root@centos7:/etc/yum.repos.d$
```

```
root@centos7:/etc/yum.repos.d$ systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: inactive (dead)
     Docs: https://docs.docker.com
root@centos7:/etc/yum.repos.d$ systemctl start docker
root@centos7:/etc/yum.repos.d$ systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: active (running) since Sat 2016-02-20 13:58:52 EST; 1s ago
     Docs: https://docs.docker.com
 Main PID: 13210 (docker)
   CGroup: /system.slice/docker.service
           └─13210 /usr/bin/docker daemon -H fd://

Feb 20 13:58:51 centos7 docker[13210]: time="2016-02-20T13:58:51.703513506-05:00" level=info ...ds"
Feb 20 13:58:51 centos7 docker[13210]: time="2016-02-20T13:58:51.731957263-05:00" level=warning ...
Feb 20 13:58:51 centos7 docker[13210]: time="2016-02-20T13:58:51.762960841-05:00" level=info ...se"
Feb 20 13:58:51 centos7 docker[13210]: time="2016-02-20T13:58:51.871168227-05:00" level=info ...ss"
Feb 20 13:58:52 centos7 docker[13210]: time="2016-02-20T13:58:52.084770835-05:00" level=info ...t."
Feb 20 13:58:52 centos7 docker[13210]: time="2016-02-20T13:58:52.084856968-05:00" level=info ...e."
Feb 20 13:58:52 centos7 docker[13210]: time="2016-02-20T13:58:52.084874989-05:00" level=info ...on"
Feb 20 13:58:52 centos7 docker[13210]: time="2016-02-20T13:58:52.084895071-05:00" level=info ...0.1
Feb 20 13:58:52 centos7 systemd[1]: Started Docker Application Container Engine.
Feb 20 13:58:52 centos7 docker[13210]: time="2016-02-20T13:58:52.096353653-05:00" level=info ...ck"
Hint: Some lines were ellipsized, use -l to show in full.
```

Otras distribuciones:

Instalación



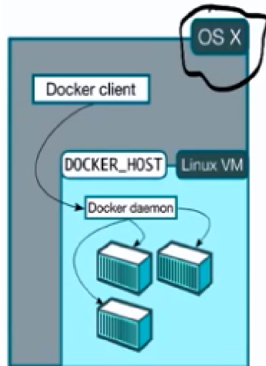
- Otras distribuciones Linux:
  - Arch Linux
  - CRUX Linux
  - FrugalWare
  - Gentoo
  - Oracle Linux
  - OpenSUSE / SUSE Linux Enterprise
- Instrucciones:

https://docs.docker.com/engine/installation/linux/

Instalación en Mac/OS X/ Windows

Instalación

· Instalación en Mac OS X / Windows



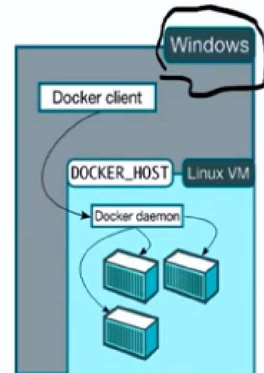
→ En Windows comprobar si la virtualización está activada.

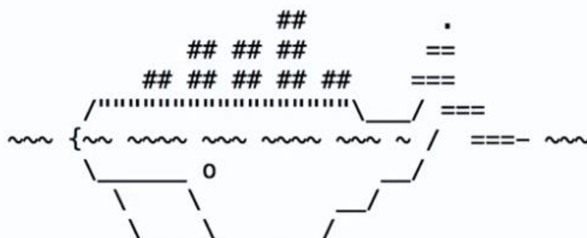
→ Instalar Docker toolbox

→ Ejecutar Docker Quickstart Terminal

→ Comprobar:

docker info
docker version





```
docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com
```

Laboratorio 2 Trabajando con imágenes y contenedores

En este laboratorio comenzaremos a trabajar con imágenes y contenedores de forma básica.

→ Ejecutar:

- docker run -ti debian echo hola
  - run → Acción
  - -ti → opciones | -t → pseudo TTY | -i → interactivo
  - debian → imagen
  - echo hola → comando

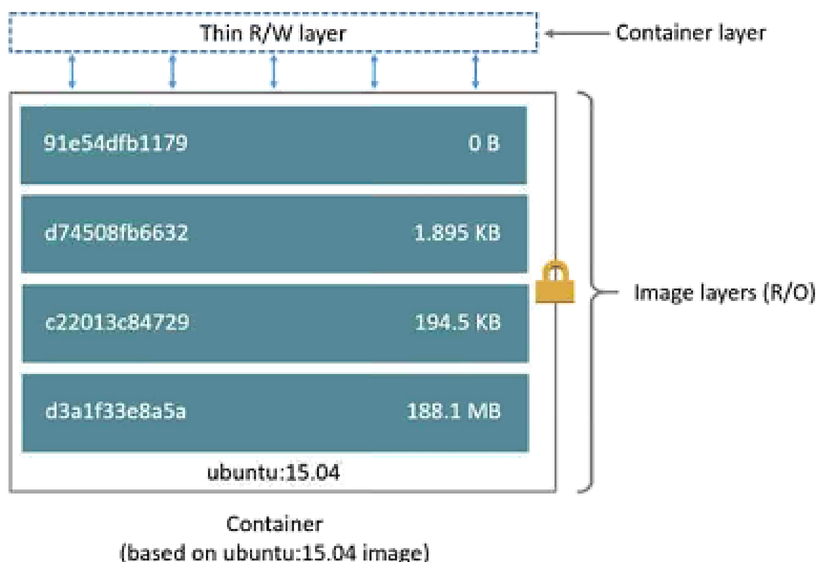
* comando puede ser /bin/sh para entrar dentro del contenedor

* con opción -d se ejecuta como un "daemon".

→ Comprobar:

- docker ps -a
  - -a → Listar contenedores detenidos
- docker logs idcontenedor

Imágenes de Docker (Docker Images)
Las imágenes de Docker son plantillas de solo lectura(podriamos decir que es como un archivo comprimido tarball que contiene todos los ficheros necesarios para la aplicación para la aplicación que se desea lanzar en el contenedor) de solo lectura, es decir, una imagen puede contener el Sistema operativo de CentOS o Ubuntu con apache instalados, pero esto solo nos permitirá crear los contenedores en esta configuración. Si hacemos cambios en el contenedor ya lanzado, al detenerlo esto no se verá reflejado en la imagen



Para listar las imágenes que disponemos actualmente en nuestro servidor Docker, utilizaremos el siguiente comando:

docker images opciones nombre de la imagen

Las opciones para la acción images son las siguientes:

- **-a/--all muestra todas las imágenes**
- **-f, --filter filtro: filtra la salida con el filtro especifico**
- **--format formato: formatea la salida**
- **--no-trunc: no trunca el tamaño de las columnas-**
- **-q/--quiet: muestra solo dos identificadores**

```
#docker images
```

El resultado será algo parecido a:

```
REPOSITORY          TAG                 IMAGE ID
CREATED         VIRTUAL SIZE
hello-world         latest              05a3bd381fc2
10 days ago      1.84kB
```

La información que se visualizara en columnas es la siguiente:

- REPOSITORY (hello-world): el nombre de la imagen en el almacén.
- TAG (latest): su versión, puede ser un número o una denominación textual. Veremos un poco más adelante que estos tags se pueden multiplicar respecto a una misma imagen.
- IMAGE ID (05a3bd381fc2): un identificador único para la imagen.
- CREATED (10 days ago): la fecha de creación de la imagen en la máquina host de Docker. Atención, no se trata de la fecha de creación de la imagen por el almacén, sino de la creación de la caché local.
- VIRTUAL SIZE (1.84kB): el tamaño de la imagen. Se trata de un tamaño virtual que representa a todas las capas que componen sucesivamente la imagen. El tamaño realmente ocupado en disco duro por una imagen, normalmente es muy inferior. En nuestro caso, la capa hello-world no pasa de 1,84 kilo-bytes y se trata además de su tamaño efectivo, porque no se basa en una capa inferior.

Utilizar imágenes Docker preexistentes

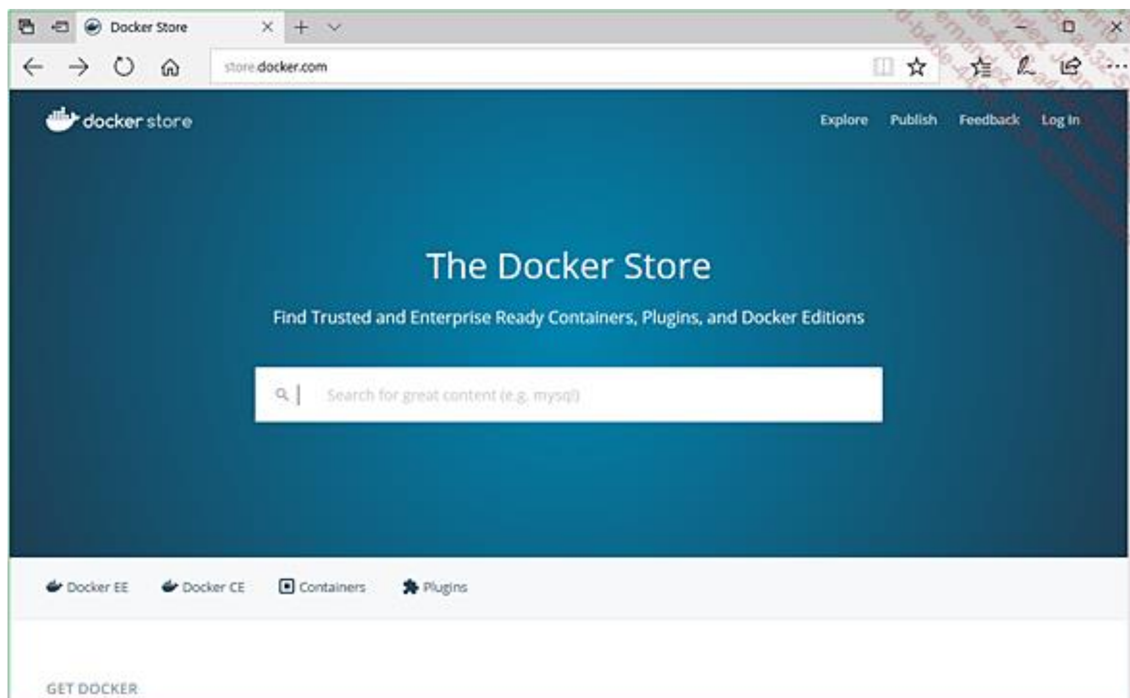
1. El Docker Store

a. El principio

El primer ejemplo de ejecución de Docker se ha mostrado anteriormente, en el que se utiliza una imagen existente, porque se trata del medio más sencillo de establecer los contenedores. En la práctica, utilizar contenedores existentes en lugar de crear los suyos propios representa la mayoría de los casos de uso.

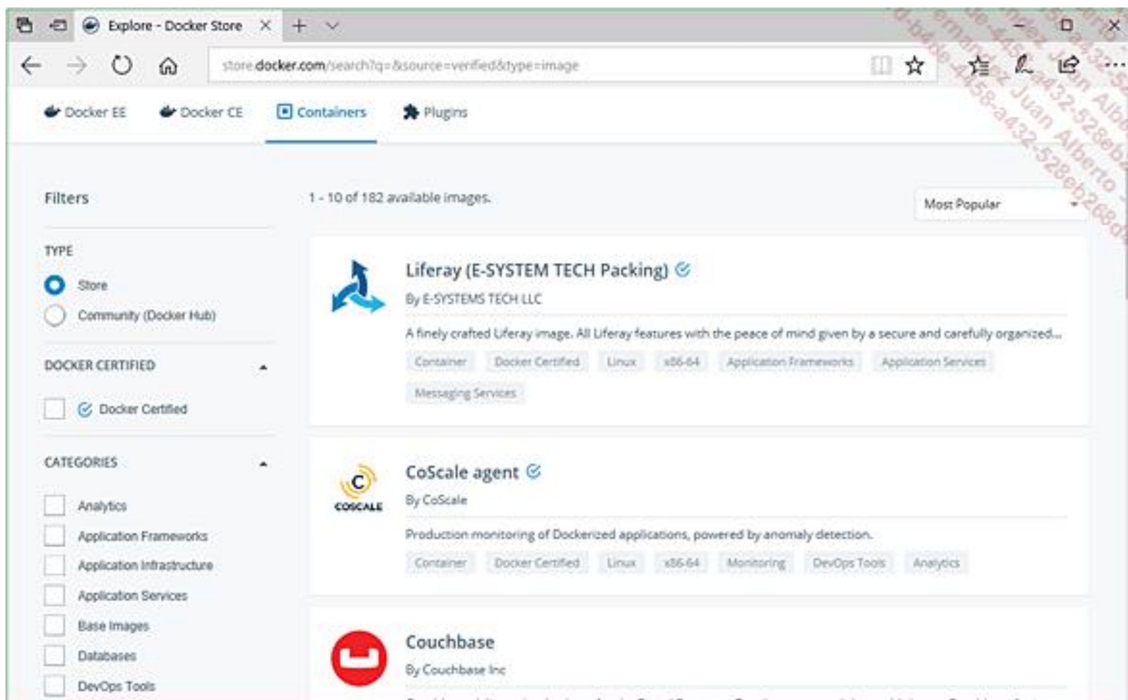
El éxito de Docker reside, no solamente en la herramienta en sí misma, sino también para una gran parte, en el hecho de que Docker - ayudado por una abundante comunidad - pone a disposición del público un gran número de imágenes pre-construidas. De esta manera es posible desarrollar contenedores con Apache, Nginx, PostgreSQL, MongoDB, WordPress y centenares de otros servidores y aplicaciones utilizando un sencillo comando, eventualmente acompañado de la configuración adecuada.

El sitio web Docker Store, accesible en https://store.docker.com, da acceso a estas imágenes:



b. Búsqueda y cualificación de imágenes

Un clic en el enlace **Explore**, en la parte superior derecha de la interfaz, no permite acceder a una interfaz de búsqueda de imágenes (observe que no es imprescindible crear una cuenta para acceder a esta parte del sitio web):



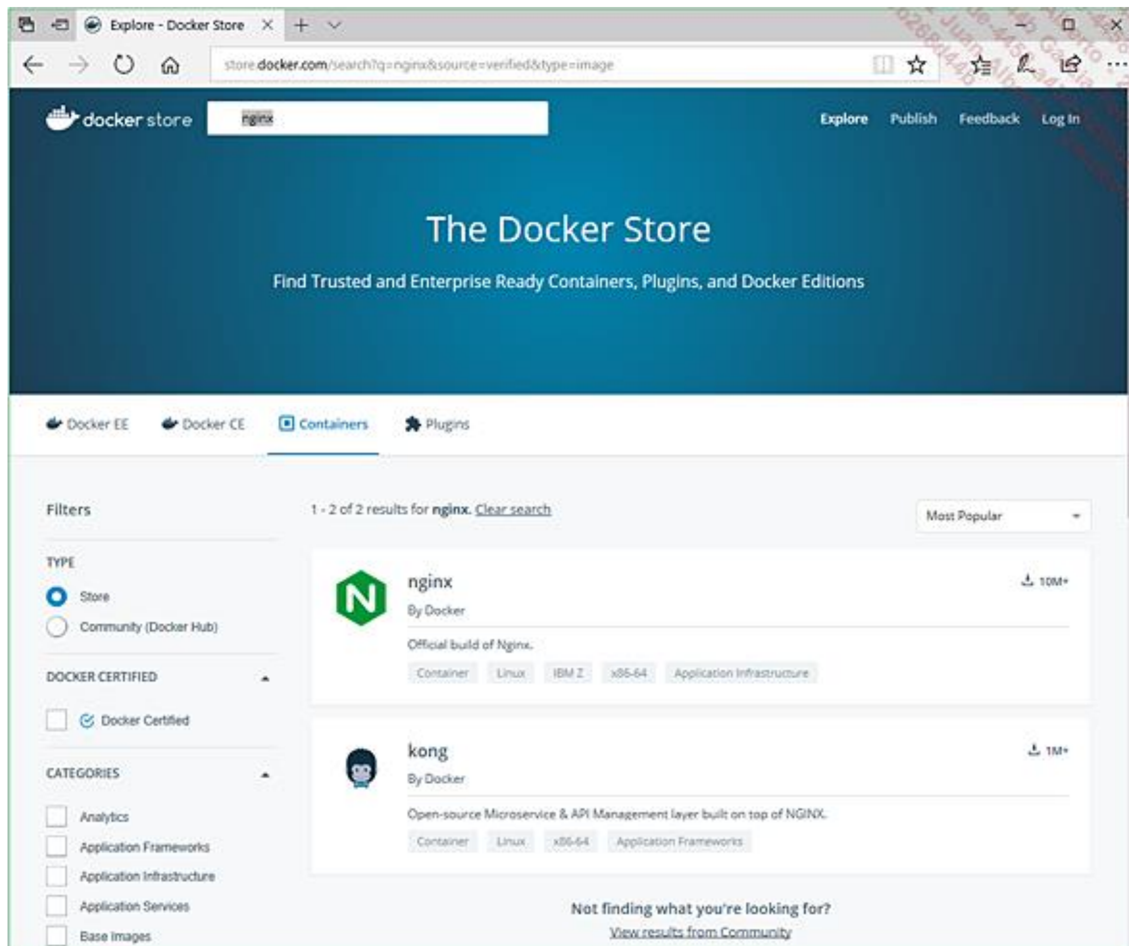
La columna de la izquierda permite filtrar la búsqueda por categorías y por medio de otras opciones que vamos a detallar. También es posible filtrar por una expresión textual, al entrar en la zona de texto junto a la lupa, en la parte superior de la interfaz (aunque no sea visible en la captura de pantalla anterior).

La primera opción de filtro en la parte superior izquierda, merece una explicación particular. Por defecto, se posiciona en **Store**, lo que quiere decir que las imágenes que aparecerán han sido verificadas por Docker, lo que hace que tengan un mayor nivel de seguridad que las imágenes comunitarias potencialmente archivadas por cualquiera en el registro. La única condición para esta operación era crear una cuenta, no hay absolutamente ninguna garantía asociada y las imágenes pueden contener errores de seguridad intencionados, virus o cualquier tipo de ataque. Por lo tanto, hay un interés real cuando están disponibles, en utilizar imágenes Docker de tipo Store. Por el contrario, teniendo en cuenta su trabajo de validación, lógicamente no son más que algunos centenares, en comparación con los centenares de miles de imágenes disponibles en la sección Community. Además, como el nombre Docker Store indica, algunas son de pago o gratuitas pero en este último caso, con una licencia de utilización que pueda constituir otro tipo de freno al uso.

La segunda opción de filtro de las imágenes se corresponde con el carácter oficialmente certificado de las imágenes. En este caso, la verificación por Docker va incluso más lejos, con un compromiso de la empresa sobre el contenido de las imágenes y su rendimiento en una infraestructura certificada. En caso de la Store, el nivel de confianza viene de la relación entre Docker y los editores establecidos, así como de un eventual soporte de pago, lo que ya es un primer nivel de calidad interesante para las empresas, tradicionalmente cautelosas respecto a Docker por motivos del origen incierto de las imágenes del registro.

c. Ejemplo de búsqueda

Supongamos que necesitamos establecer un servidor Nginx. Escribiendo esta palabra clave en la parte superior de la interfaz y validando la búsqueda, obtenemos lo siguiente:



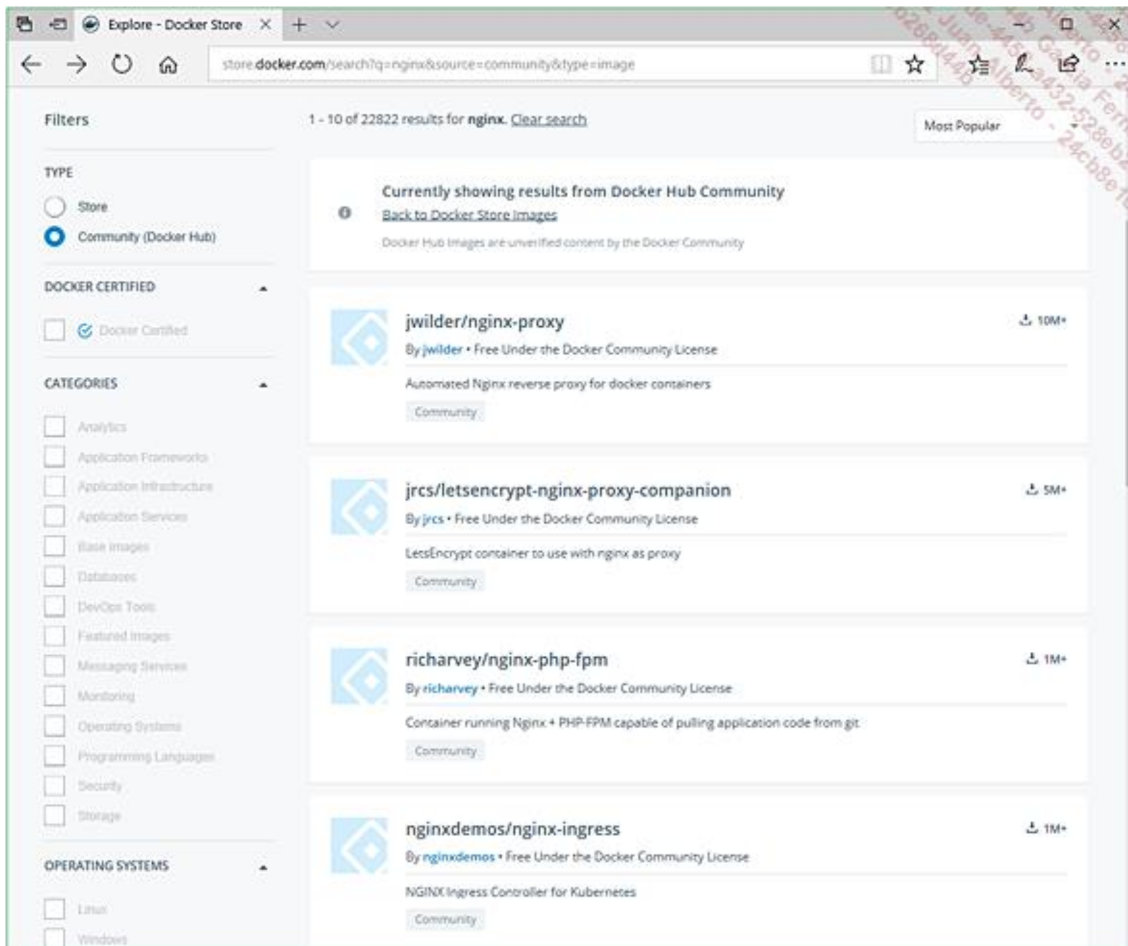
Como se puede ver, el filtro todavía está en **Store**, lo que quiere decir que queremos ver imágenes validadas por Docker. Por el contrario, la ausencia del icono azul **Docker Certified**, muestra que estas imágenes no están certificadas. En la práctica, se muestra la segunda imagen porque se trata de un producto basado en Nginx, pero esto no es lo que buscamos. Por lo tanto, la primera es la imagen oficial que buscamos (como se indica en la etiqueta, aunque no fuera lo previsto).

El número de descargas es también un buen indicador de calidad de una imagen, si es significativamente elevado. Después de algunos millones de descargas (en nuestro ejemplo, más de diez millones), se puede considerar que una mala calidad (rendimiento, adaptación, funcionalidad, etc.) o una corrupción (virus, backdoor, bomba lógica, etc) se hubiera detectado con toda seguridad.

Las etiquetas situadas debajo del nombre de la imagen dan información adicional de las plataformas destino y las categorías eventuales de pertenencia del resultado de la búsqueda. A continuación, un clic en el resultado de la búsqueda en sí misma conduce a la página correspondiente con, esta vez, todos los detalles asociados:

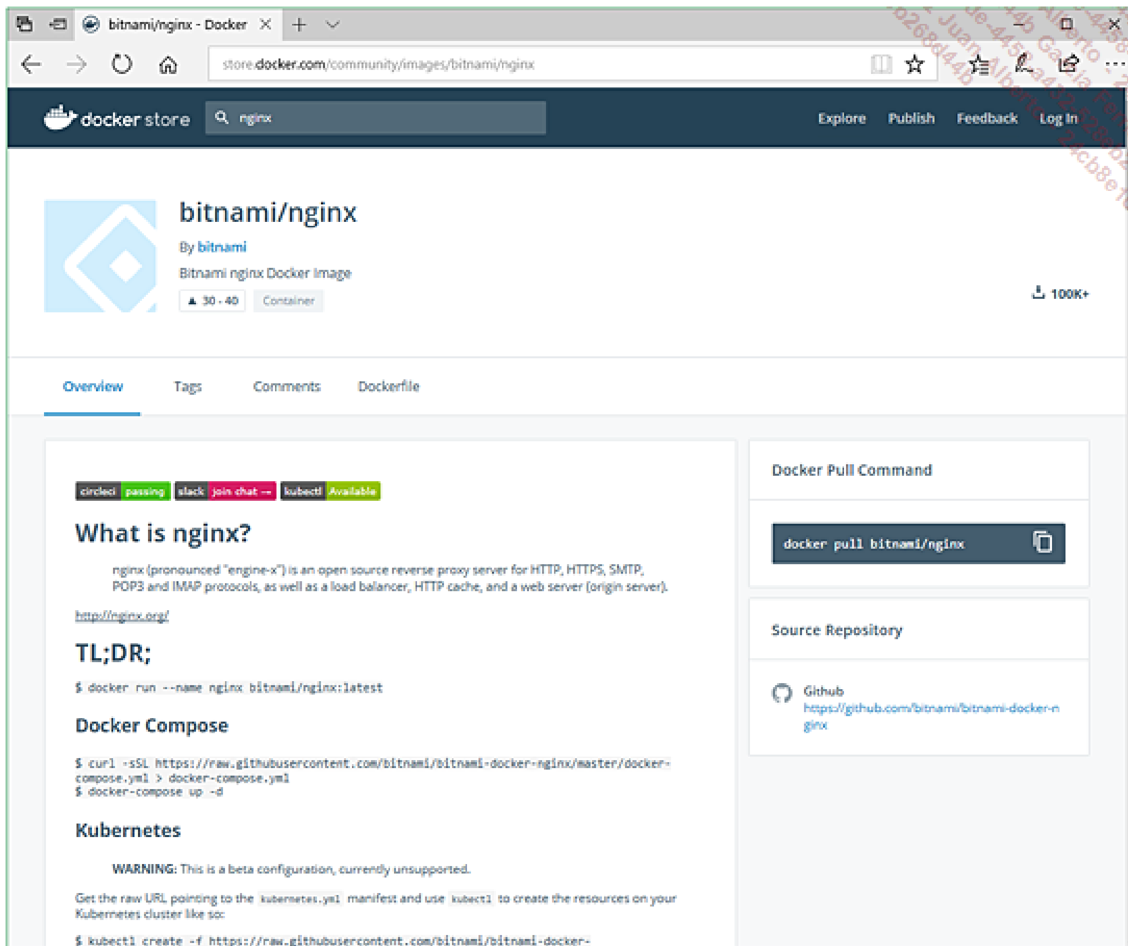
d. Caso de las imágenes de la comunidad

Volviendo una página atrás y cambiando el filtro para ver solo las imágenes generadas por la comunidad, en esta misma búsqueda con la palabra clave **nginx**, la gran cantidad de resultados muestra claramente el problema para un principiante:



Entre las 22.822 imágenes encontradas, algunas se corresponden con usos particulares de Nginx (como un proxy, controlador de tipo ingreso para Kubernetes, etc.), otras con instalaciones de Nginx que incluyen frameworks de desarrollos (Nginx con PHP y FPM, Nginx con Drupal, etc.); algunas son el resultado de empresas reconocidas por la calidad de sus imágenes, otras provienen de una persona que ha utilizado la parte pública de Docker Store para transmitir imágenes de una plataforma a otra sin voluntad real de compartir (documentación inexistente es un mal síntoma); algunas se han descargado millones de veces, otras casi nunca y son visiblemente obsoletas. En resumen, el efecto para una persona que busca simplemente una imagen con Nginx, puede ser el de una feria de imágenes sin ningún sentido, pero es necesario pasar por las imágenes de la comunidad si no hay disponible ninguna imagen oficial y crear una imagen por sí mismo no sea una opción.

En este caso, la calidad de la documentación es un primer indicio. Tomemos como ejemplo la imagen Nginx ofrecida por Bitnami.



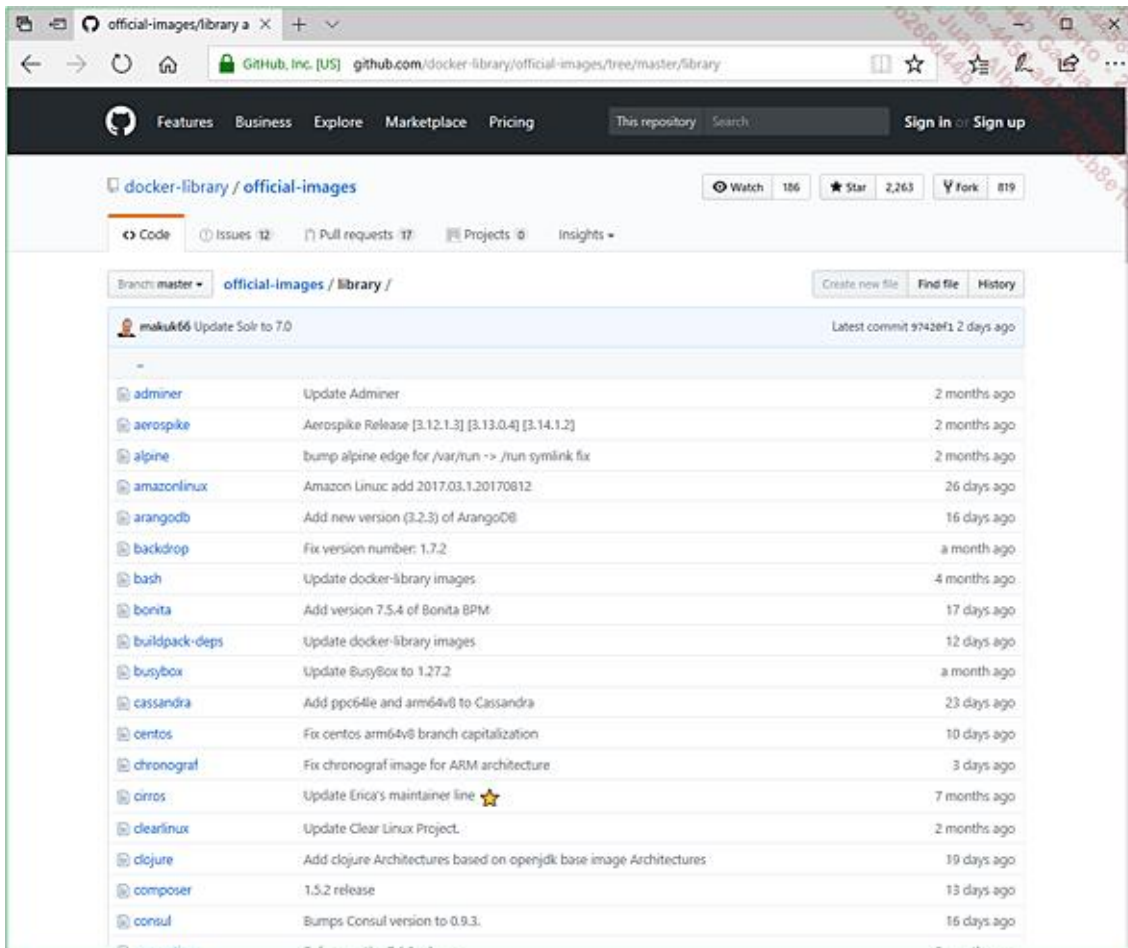
La presencia de etiquetas hace referencia a la integración continua CircleCI y la presencia de un chat dedicado, son síntomas de garantía. El contenido detallado de la sección **Overview** también es un buen signo para el potencial usuario. En la parte superior de la interfaz se encuentra el número de estrellas dadas por los usuarios a esta imagen (en nuestro ejemplo, la imagen bitnami/nginx tiene 36 estrellas, de ahí la visualización entre 30 y 40). Incluso aquí, se trata de un índice adicional de calidad. Independientemente, cada uno de estos índices solo tiene una fiabilidad limitada, pero tener índices positivos conduce al final a un nivel de garantía potencialmente suficiente, en todo caso para unas pruebas y una cualificación antes de entrar en producción.

Es necesario estar identificado para votar y asignar una estrella a una imagen dada.

e. Complementos a las imágenes oficiales

Contrariamente a las imágenes de la comunidad cuyo nombre utiliza el prefijo del identificador del propietario de la imagen (separado del nombre de la imagen por un símbolo /), las imágenes oficiales no usan prefijo. En el pasado, esto era equivalente a un prefijo de identificador "library", pero este ya no es el uso y la distinción entre imágenes oficiales y no oficiales a partir de ahora es más marcada.

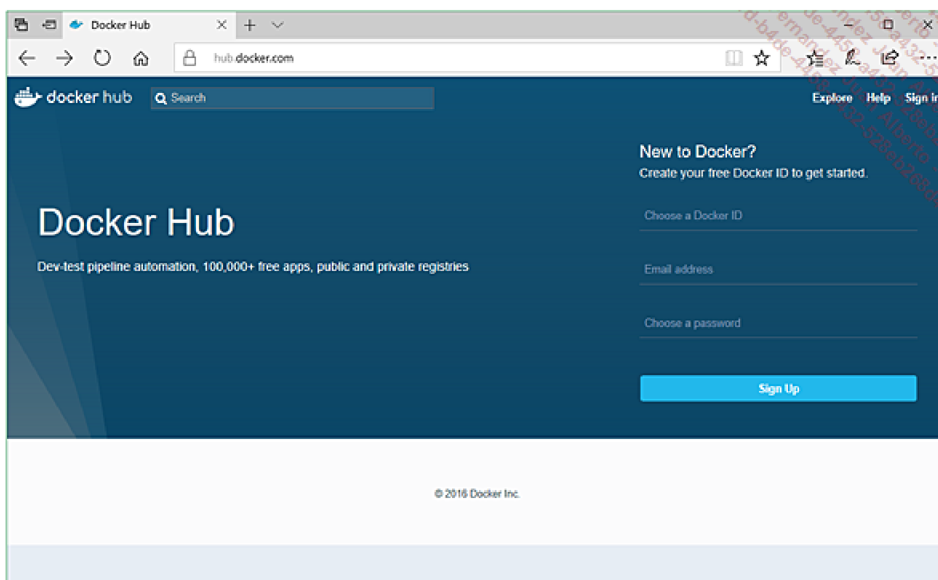
Observe también que la lista de las imágenes oficiales ahora está en el almacén GitHub llamado docker-library/official-images (https://github.com/docker-library/official-images, y después desciende en el directorio llamado library):



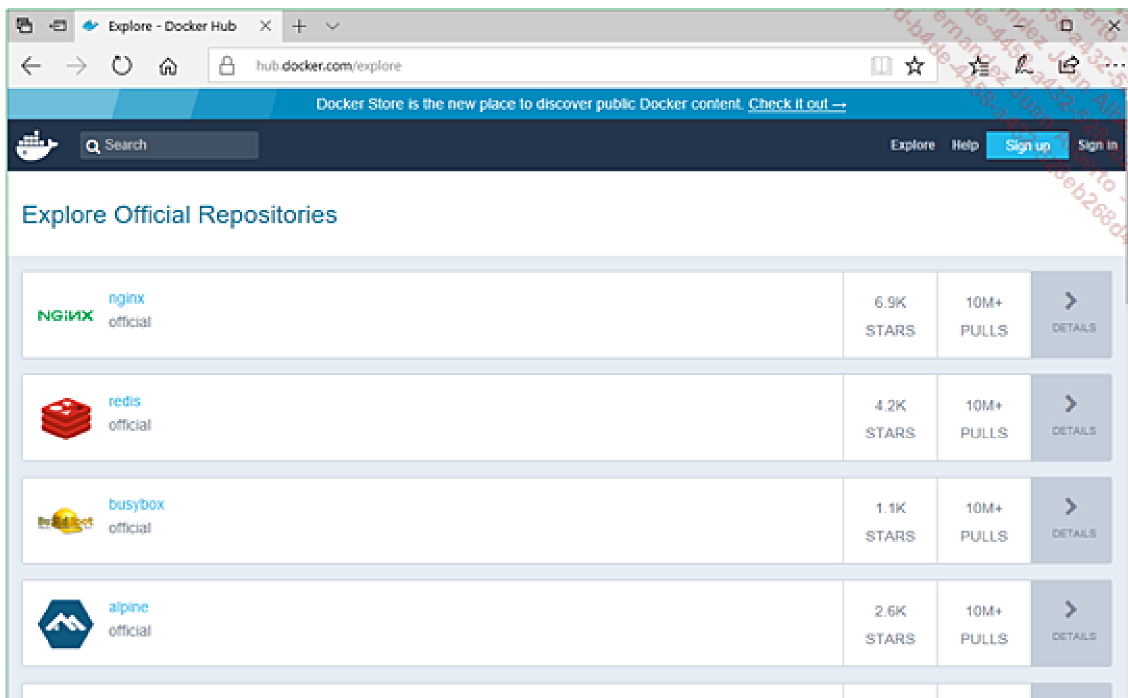
f. Enlace con el registro Docker Hub

El Docker Store es una creación relativamente reciente, pero desde siempre existe la capacidad de basarse en las imágenes Docker ofrecidas por otros (editores o comunidad) y es una de las razones principales del éxito de Docker. En el pasado, la publicación de imágenes era gestionada por el Docker Hub, algunas veces llamado registro Docker o registro público.

El Docker Hub existe siempre y es accesible en la dirección: http://hub.docker.com



Por supuesto, un clic en el enlace **Explore** en la parte superior derecha, redirige hacia una página donde claramente se indica en la parte superior que el nuevo lugar para encontrar el contenido Docker es el Docker Store, con un enlace que dirige hacia este último:



Claramente, el registro público Docker Hub sigue presente por razones de compatibilidad, pero parece una buena opción tener la costumbre de utilizar el Docker Store, que será más duradero.

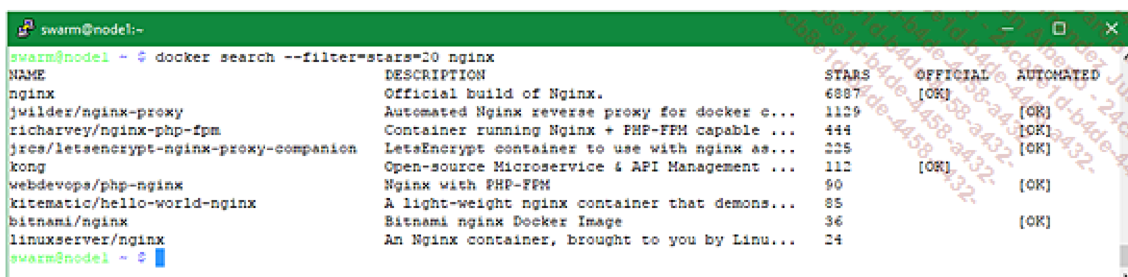
g. Búsqueda por la línea de comandos

El cliente Docker también permite buscar las imágenes en función de palabras clave:

Buscar una imagen Docker

```
docker search --filter=stars=20 nginx
```

La ejecución de este comando implica la búsqueda de todas las imágenes que contiene la palabra clave nginx y al menos dispongan de veinte estrellas. Esto conduce a la siguiente visualización o algo parecido:



Las columnas de información devuelven:

- El nombre de la imagen. Observe en particular que la primera imagen (con mejor nota, la ordenación se hace por el número de estrellas de manera descendente), está compuesta únicamente de la palabra nginx, sin prefijo correspondiente a un propietario. Como se ha explicado, se trata de la convención

para llamar a las imágenes oficiales. Salvo necesidad particular, si queremos un contenedor con el servidor web Nginx, esta es la imagen que utilizaremos.

- La descripción de la imagen, que se alienta especialmente durante el registro de una imagen, de manera que los potenciales usuarios puedan conocer las particularidades de una imagen. Por defecto, la visualización está cortada, pero la opción --no-trunc=true permite ver las descripciones completas si es necesario.
- El número de estrellas.
- Una indicación del carácter oficial de la imagen.
- Una indicación del hecho de que la compilación de la imagen se ha realizado de manera automática.

La antigua opción --stars está obsoleta y se sustituye por la opción --filter, que es más genérica. Observe también que esta última se puede sustituir por su versión abreviada -f, pero de manera general, las opciones se utilizan en el libro en su forma completa, lo que permite una mejor comprensión de su efecto.

Para encontrar las opciones, están disponibles los siguientes comandos:

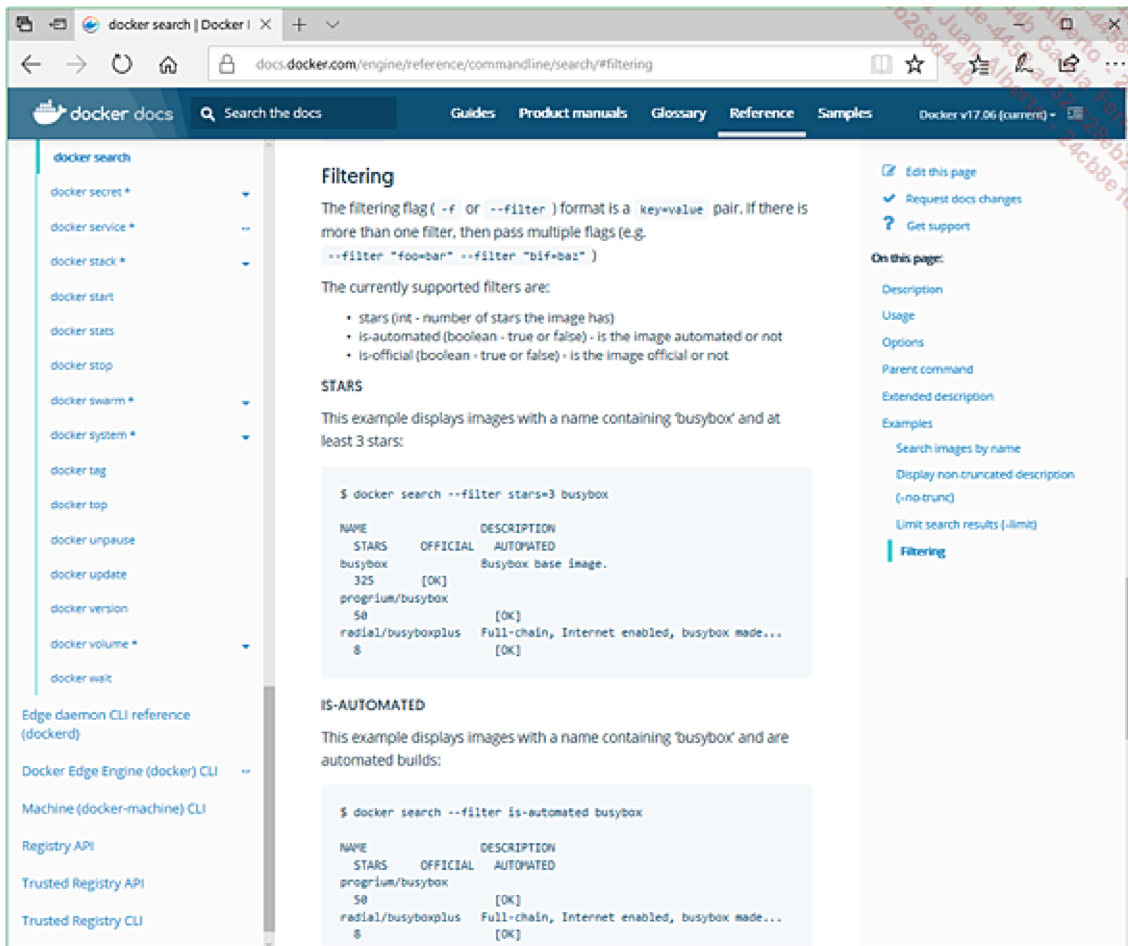
Obtener la lista de los comandos del cliente Docker

```
docker help
```

Obtener ayuda de las opciones de un comando

```
docker help nombre_del_comando
```

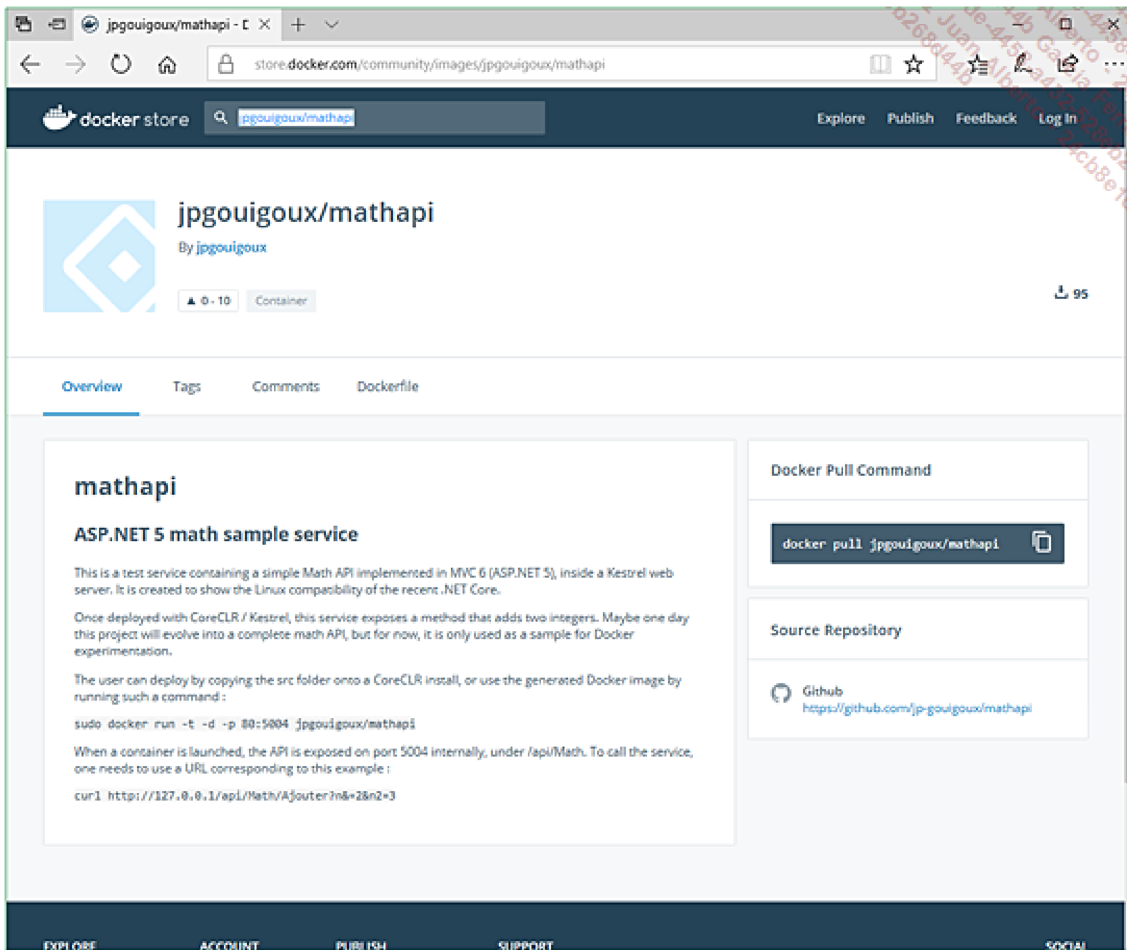
Por supuesto, el contenido de la ayuda se limita a la explicación de las opciones del comando, pero para obtener todo el detalle de su contenido algunas veces es necesario hacer referencia a la documentación en línea. En el caso de la opción --filter del comando docker search, la URL https://docs.docker.com/engine/reference/commandline/search/#filtering contiene las explicaciones completas:



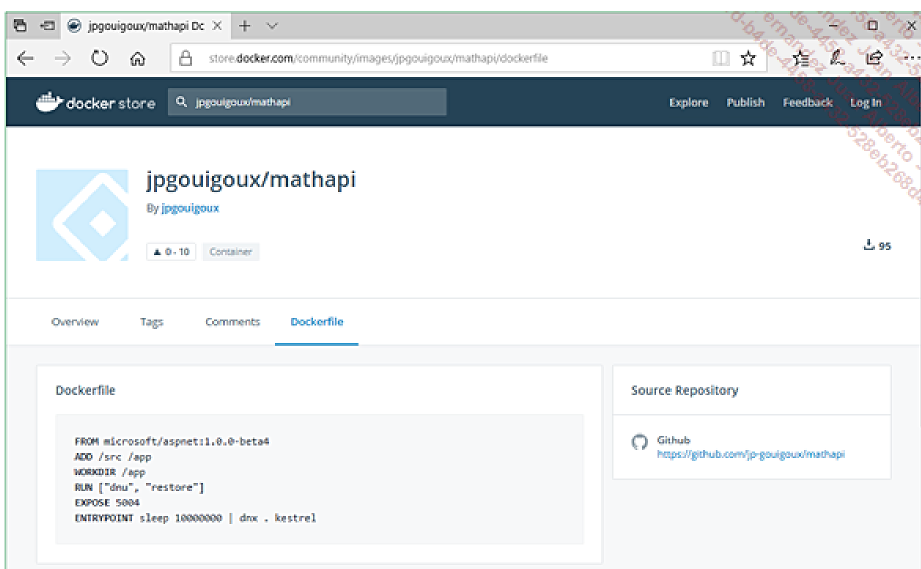
h. Precauciones con una imagen no oficial

Una imagen no oficial puede venir de cualquiera y contener cualquier funcionalidad. Es conveniente tener un mínimo de seguridad durante su implantación. Por supuesto, la estanqueidad inherente al modo de funcionamiento de Docker, da una cierta garantía de ausencia de impactos en la máquina host, pero esto no impide comportamientos potencialmente peligrosos en el contenedor en sí mismo.

Tomemos como ejemplo la imagen jpgouigoux/mathapi. Una búsqueda en el registro conduce a la página de bienvenida que describe rápidamente lo que se supone que se puede hacer con la imagen, cómo utilizarla y algunas propiedades adicionales.



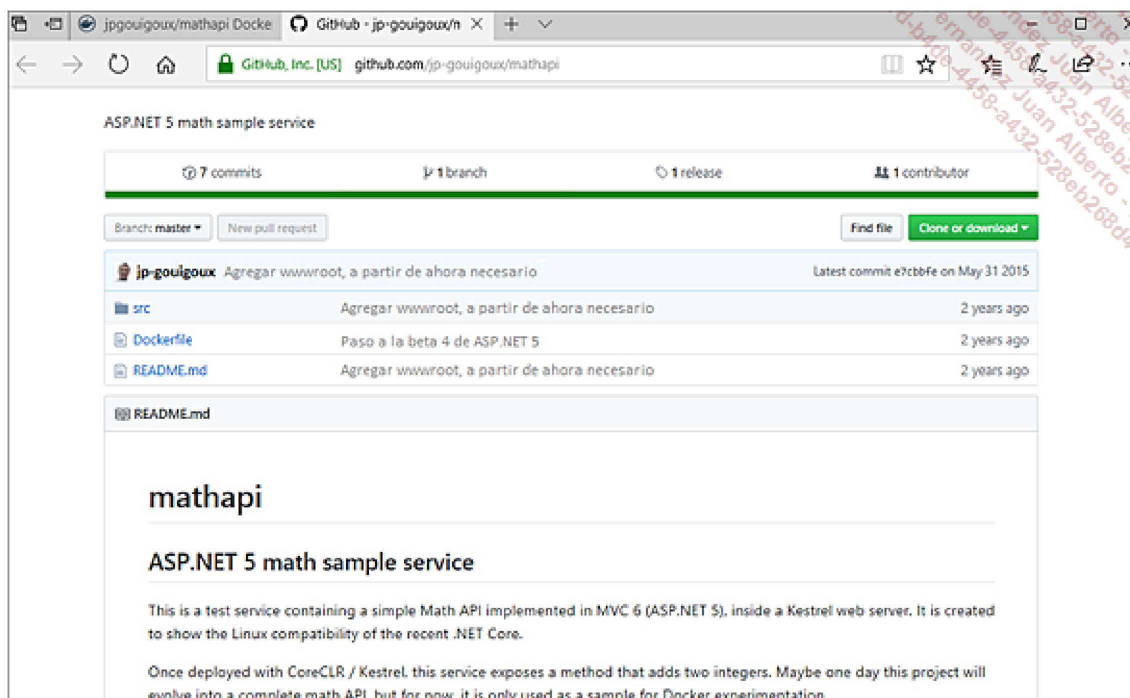
El hecho de que la documentación indique un almacén GitHub como fuente de la imagen, es un primer paso hacia una validación del contenido de la imagen. En este caso, Docker ha creado la imagen a partir del código fuente disponible y en particular, del archivo Dockerfile. Esto permite, no únicamente validar que no se ha traficado con la imagen disponible por parte de la persona propietaria del almacén en Docker Store (es Docker el que se encarga de su compilación a partir de los archivos fuente), sino que además permite echar un vistazo al Dockerfile, que aparece justamente en la cuarta pestaña de la interfaz:



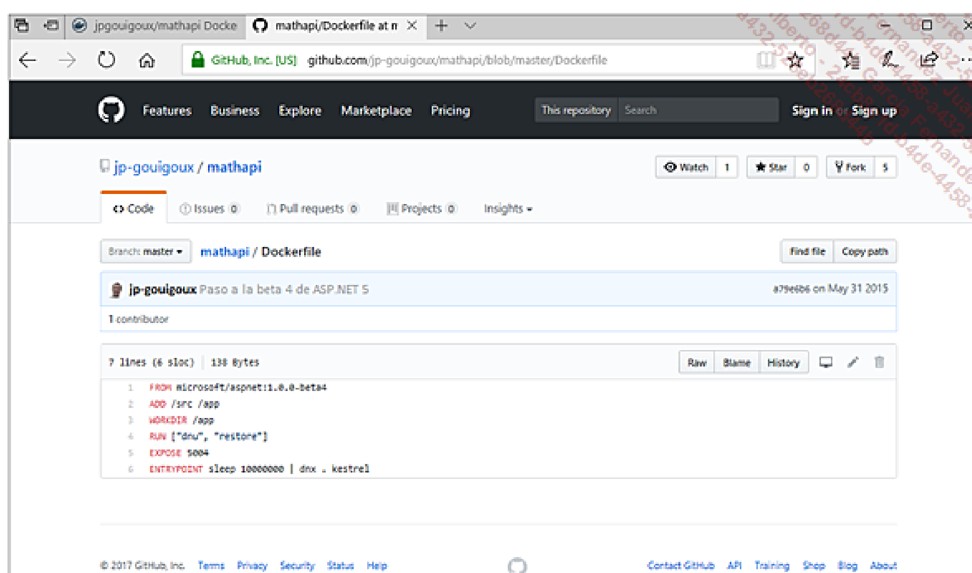
A continuación vamos a ver más en detalle el contenido de este archivo, pero de momento su presencia permite eliminar la opacidad sobre el contenido de la imagen.

Esta opacidad no es en sí misma un problema desde el momento en que la confianza se establece con el proveedor. En las arquitecturas orientadas a servicios, es incluso lógico que la implementación sea lo más oculta posible y que solo sea visible el contrato de servicio. Por ejemplo, antes de analizar la imagen mencionada anteriormente, no conocíamos la tecnología subyacente y es perfectamente posible utilizarla, sin conocer el servidor de aplicaciones ni el lenguaje utilizados.

Si es necesario, todos los archivos se podrán controlar yendo al almacén origen en GitHub:



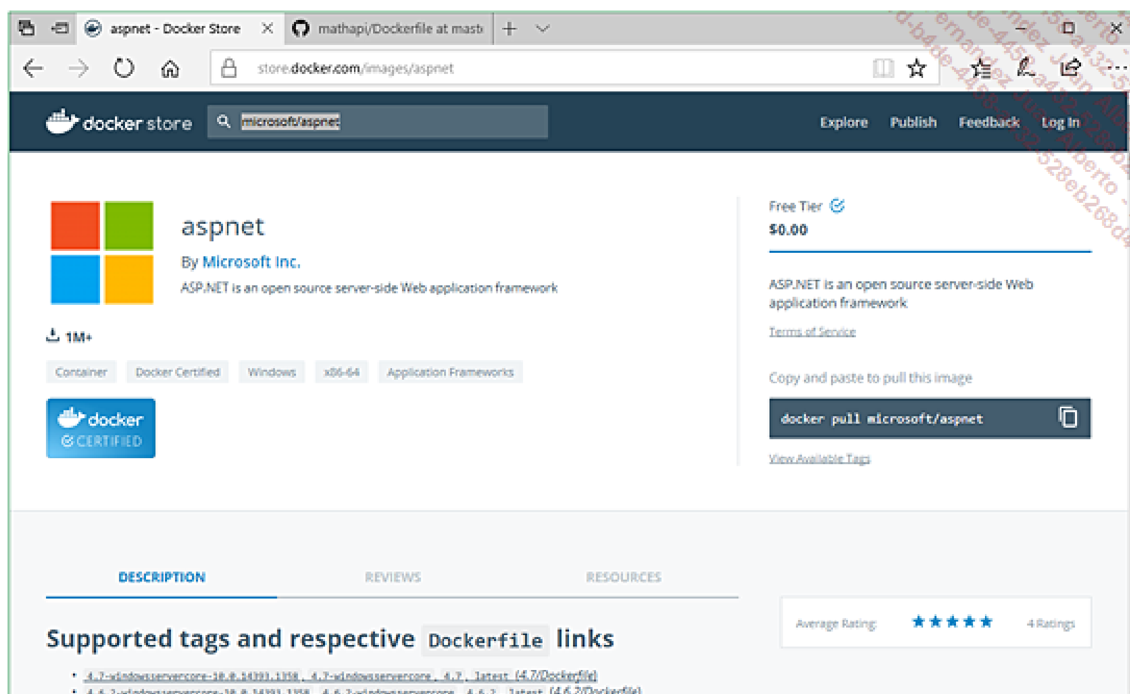
Por supuesto, el archivo Dockerfile es el mismo que el indicado en Docker Store:



Para completar el análisis, conviene verificar su contenido, con un sentido crítico, incluso la asistencia de un profesional si el uso previsto es para producción.

En primer lugar, la palabra clave FROM hace referencia a una imagen que, aunque no sea oficial, proviene de un origen más reconocido, a saber, la cuenta llamada microsoft. Atención, el hecho de tener una cuenta con nombre, no quiere decir que la persona represente a la empresa asociada. Docker no tiene ningún interés en dejar que las personas usurpen las cuentas de empresas conocidas. En el caso particular de esta imagen microsoft/aspnet, un vistazo a la página del registro permite encontrar un enlace al sitio web MSDN oficial de Microsoft y que reenvía a la página del registro Docker. Por tanto, se establece la relación de confianza.

La versión 1.0.0-beta4 es una versión obsoleta de microsoft/aspnet, pero cuando el presente libro salga a la venta, la imagen habrá evolucionado en una versión más reciente de .NET Core. De hecho, la imagen aspnet ha evolucionado de tal manera que a día de hoy, forma parte de las imágenes oficiales (y por lo tanto, ya no usa el prefijo de microsoft/) y de las imágenes certificadas (afortunadamente gratuita), como se muestra en la siguiente captura de pantalla. Por el contrario, se basa en .NET 4.7 y las versiones .NET Core están en la imagen oficial llamada dotnet.



Si este no es el caso, la práctica consiste en seguir el origen de las imágenes las unas respecto a las otras. En casi todos los casos, en algún momento nos encontraremos necesariamente con una imagen que provenga de un proveedor para el que, la confianza es lo suficientemente fuerte como para que no tengamos necesidad de su contenido. Falta comprobar todos los comandos adicionales que están en los archivos Dockerfile sucesivos, de manera que se valide que no añaden funcionalidades maliciosas.

En algunos casos es relativamente sencillo validar el contenido. Por ejemplo, si volvemos a la imagen jpgouigoux/mathapi, hay tres archivos guardados y su contenido es fácilmente verificable en el mismo almacén GitHub que el archivo Dockerfile. Un vistazo al archivo Service.cs, por ejemplo, muestra claramente que el contenido es el esperado.

La respuesta a la observación de que hace cierto tiempo es, para los más curiosos, que este servicio se implementa en ASP.NET 5 / MVC 6, en el que se utiliza el framework .NET Core, a saber, la versión ligera y moderna de .NET, soportada por Microsoft en Windows, pero también en Linux. Una parte de esta información estaba escrita en la página de bienvenida, pero un vistazo más en profundidad permite comprobar su veracidad.

En otros casos, el análisis es más complejo. Por ejemplo, si no se hubiera establecido que la imagen aspnet era de confianza, hubiera sido necesario analizar el contenido del Dockerfile, representado a continuación:

```
FROM microsoft/dotnet-framework:4.7

SHELL ["powershell", "-Command", "$ErrorActionPreference =
'Stop'; $ProgressPreference = 'SilentlyContinue';"]

RUN Add-WindowsFeature Web-Server; \
    Add-WindowsFeature NET-Framework-45-ASPNET; \
    Add-WindowsFeature Web-Asp-Net45; \
    Remove-Item -Recurse C:\inetpub\wwwroot\*

ADD ServiceMonitor.exe /

#download Roslyn nupkg and ngen the compiler binaries
RUN Invoke-WebRequest
https://api.nuget.org/packages/microsoft.net.compilers.2.3.1.nupkg
-OutFile c:\microsoft.net.compilers.2.3.1.zip; \
    Expand-Archive -Path c:\microsoft.net.compilers.2.3.1.zip
-DestinationPath c:\RoslynCompilers; \
    Remove-Item c:\microsoft.net.compilers.2.3.1.zip -Force; \
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\csc.exe
/ExeConfig:c:\RoslynCompilers\tools\csc.exe | \
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\vbc.exe
/ExeConfig:c:\RoslynCompilers\tools\vbc.exe  | \
    &C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\VBCSCompiler.exe
/ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe | \
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\csc.exe
/ExeConfig:c:\RoslynCompilers\tools\csc.exe | \
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\vbc.exe
/ExeConfig:c:\RoslynCompilers\tools\vbc.exe | \
    &C:\Windows\Microsoft.NET\Framework\v4.0.30319\ngen.exe
install c:\RoslynCompilers\tools\VBCSCompiler.exe
/ExeConfig:c:\RoslynCompilers\tools\VBCSCompiler.exe;

ENV ROSLYN_COMPILER_LOCATION c:\\RoslynCompilers\\tools

EXPOSE 80

ENTRYPOINT ["C:\\ServiceMonitor.exe", "w3svc"]
```

La imagen básica es oficial, el archivo es mucho más completo que el de la imagen estudiada antes. Sobre todo, descargar y descomprimir un archivo cuyo contenido es opaco, se debería estudiar con cuidado para eliminar cualquier duda. Este análisis hace necesario disponer de una experiencia avanzada para validar que el contenido no tiene efectos indeseados. En ausencia de una competencia como esta, la pregunta vuelve de nuevo una vez más a la confianza en el productor de la imagen. Esta es la razón por la que Docker orienta cada vez más al usuario a las imágenes oficiales.

El cliente docker también nos permite hacer busquedas en el repositorio oficial sin tener que abrir un navegador. Para ello utilizaremos el comando:

docker search búsqueda opciones

Las opciones son las siguientes:

-f, --filter: filtra la salida con el filtro especificado:

--limit numero: limita el numero de resultados por defecto 25

--no-trunc: no trunca el tamaño de las columnas.

Buscamos una imagen httpd:

 # docker search httpd

Buscando una imagen httpd y limitando a la salida de 5 resultados:

 # docker search –limit 5 httpd

Para descargar imágenes o actualizar una imagen desde el repositorio oficial, utilizaremos el siguiente comando:

docker pull imagen:tag

Esta acción realizara los siguientes pasos :

- Comprobara si en el repositorio existe dicha imagen.
- Comprobar si la imagen y la versión (la etiqueta especificada en el tag o latest si no se especifica) existen actualmente en nuestro servidor.
- En el caso de que exista una versión mas reciente, actualizara la imagen u la versión indicada.
- Descomprimira la imagen recién descargada
- Incluira la imagen a la lista de imágenes accesibles y le asociara un identificador corto y otro largo de 64 caracteres basándose en un algoritmo share256. Este identificador largo es importante para verificar la identidad de la imagen.

Descargamos la imagen del servidor web nginx:

docker pull nginx

Using default tag: latest

latest: Pulling from library/nginx

a5a6f2f73cd8: Already exists

1ba02017c4b2: Pull complete

33b176c904de: Pull complete

Digest: sha256:5d32f60db294b5deb55d078cd4feb410ad88e6fe77500c87d3970eca97f54dba

Status: Downloaded newer image for nginx:latest

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	568c4670fa80	9 days ago	109MB

Una imagen contiene distintas capas de las modificaciones que se han ido realizando en una imagen base. Las imágenes creadas entre la imagen base y la imagen definitiva se llaman imágenes intermedias. Un ejemplo es una imagen con una distribución que contiene los elementos básicos. Posteriormente, en esa imagen se han instalado algunas utilidades, una aplicación y sus librerias. Para finalizar, se establece un software de monitorización para dicha aplicación. En este caso dispondremos:

- Una imagen base: contiene lo básico del sistema operativo
- Dos imágenes intermedias:
  - La primera después de la instalación de las utilidades
  - La segunda después de la instalación de la aplicación y sus librerias
- Una imagen final: es la unión del resto de las imágenes

Para ver el historial de una imagen utilizaremos el comando:

docker history nginx

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
568c4670fa80	9 days ago	/bin/sh -c #(nop) CMD ["nginx" "-g" "daemon…	0B	
<missing>	9 days ago	/bin/sh -c #(nop) STOPSIGNAL [SIGTERM]	0B	
<missing>	9 days ago	/bin/sh -c #(nop) EXPOSE 80/tcp	0B	
<missing>	9 days ago	/bin/sh -c ln -sf /dev/stdout /var/log/nginx…	22B	
<missing>	9 days ago	/bin/sh -c set -x && apt-get update && apt…	53.8MB	
<missing>	9 days ago	/bin/sh -c #(nop) ENV NJS_VERSION=1.15.7.0.…	0B	
<missing>	9 days ago	/bin/sh -c #(nop) ENV NGINX_VERSION=1.15.7-…	0B	
<missing>	3 weeks ago	/bin/sh -c #(nop) LABEL maintainer=NGINX Do…	0B	
<missing>	3 weeks ago	/bin/sh -c #(nop) CMD ["bash"]	0B	
<missing>	3 weeks ago	/bin/sh -c #(nop) ADD file:dab9baf938799c515…	55.3MB	

Si no desea conservar la imagen, hay un comando que permite eliminarla.

Eliminar una imagen

```
docker rmi [nombre de la imagen]
docker rmi  --help
```

docker rmi nginx

Arrancamos un contenedor basado en una imagen a través de su IMAGE ID, nosotros selecionaremos el **IMAGE ID** de nuestra imagen que tengamos en nuestro servidor de docker:

[root@docker ~]# docker run -ti 568c4670fa80 echo hola

Hola

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1b5ae700e0a6 b45f1e1c24ef "echo hola" 26 seconds ago Exited (0) 19 seconds ago tiny_hugle

Arrancamos un contenedor y nos conectamos por Shell a nuestro conector:

[root@docker ~]# docker run -ti docker.io/nickistre/centos-lamp /bin/bash

[root@9494b5dec581 /]#

Para salirnos de este contenedor y que siga corriendo utilizaremos la cnbinacion de teclas **Control+PQ**, y el contenedor seguirá corriendo.

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9494b5dec581 docker.io/nickistre/centos-lamp "/bin/bash" About a minute ago Up About a minute 22/tcp, 80/tcp, 443/tcp mad_rosalind

Para ver los log de un contenedor:

[root@docker ~]# docker logs 1b5ae700e0a6

Hola

Para parar un contenedor que tenemos activo:

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9494b5dec581 docker.io/nickistre/centos-lamp "/bin/bash" 6 minutes ago Up 6 minutes 22/tcp, 80/tcp, 443/tcp mad_rosalind

1b5ae700e0a6 b45f1e1c24ef "echo hola" 8 minutes ago Exited (0) 8 minutes ago
tiny_hugle

[root@docker ~]# docker stop 9494b5dec581

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9494b5dec581 docker.io/nickistre/centos-lamp "/bin/bash" 7 minutes ago Exited (137) 39 seconds ago
mad_rosalind

1b5ae700e0a6 b45f1e1c24ef "echo hola" 9 minutes ago Exited (0) 9 minutes ago
tiny_hugle

Para arrancar el contenedor que hemos parado anteriormente:

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9494b5dec581 docker.io/nickistre/centos-lamp "/bin/bash" 8 minutes ago Up 1 seconds 22/tcp, 80/tcp,
443/tcp mad_rosalind

Paramos todos los contenedores y los eliminamos:

[root@docker ~]# docker stop 9494b5dec581

9494b5dec581

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9494b5dec581 docker.io/nickistre/centos-lamp "/bin/bash" 10 minutes ago Exited (137) 43 seconds ago
mad_rosalind

1b5ae700e0a6 b45f1e1c24ef "echo hola" 12 minutes ago Exited (0) 12 minutes ago
tiny_hugle

Eliminamos todos los contenedores, no las imágenes, para dejar nuestro sistema limpio y volver a comenzar:

```
[root@docker ~]# docker rm $(docker ps -a -q)
9494b5dec581
1b5ae700e0a6
[root@docker ~]# docker ps -a
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
```

Laboratorio 3 Trabajando con contenedores

· Obtener ayuda: docker --help

· Acciones:

→ Arrancar contenedor: docker start idcontenedor

→ Reiniciar contenedor: docker restart idcontenedor

→ Eliminar Contenedor: docker rm idcontenedor

→ Conectarse a un contenedor: docker attach idcontenedor (Ctrl-P Ctrl-Q)

→ Ejecutar un proceso dentro: docker exec idcontenedor comando

→ Información:

- Listar: docker ps [-a] [-l]
- Estadísticas: docker stats
- Listar procesos contenedor: docker top idcontenedor
- Registros: docker logs [-f] idcontenedor

En este lab vamos a lanzar contenedores en forma de daemon, y nos vamos a conectar a ellos y comprobaremos que no finalizan.

Parar todos los contenedores:

docker stop $(docker ps –a –q)

Eliminar todos los contenedores:

docker rm $(docker ps –a –q)

Eliminar todas las imágenes (Ojo no realizarlo en los laboratorios):

docker rmi $(docker images –q)

Para ver al ayuda de los comandos:

 # docker --help

Si queremos eliminar un contenedor:

#docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

013e49664e68 c319cf0f078c "/bin/bash -c 'while " 22 hours ago Exited (137) 21 hours ago
big_goldstine

[root@docker ~]# docker rm big_goldstine

big_goldstine

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Si queremos conectarnos a la consola de un conector que se este ejecutanto:

 # docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/jdeathe/centos-ssh-apache-php latest c319cf0f078c 6 weeks ago 278.5 MB

docker.io/nickistre/centos-lamp latest b45f1e1c24ef 10 months ago 538.4 MB

[root@docker ~]# docker run -it b45f1e1c24ef /bin/bash

Para conectacnos podemos usar el nombre genereado con el identificados:

[root@docker ~]# docker attach adoring_lalande

Para salirnos sin detener el contenedor utilizamos la combinación de teclas **Control+PQ**

Lanzar contenedores en forma de daemon, y nos vamos a conectar a ellos y comprobaremos que no finalizan, nos aseguamos de que no tenemos ningún contenedor ejecutándose, si lo tenemos lo eliminaremos:

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Lanzamos un conetenedor que refresca la fecha cada 2 segundos y podemos interacturar con el contenedor porque hemos utlizado –d

docker run -dti c319cf0f078c /bin/bash -c "while true; do date; sleep 2; done"

013e49664e68e237fb3bab37cec9e8728c42e82a34f4a67f6662f3b871b47b4f

Ejecutamos docker ps –a y vemos como nos da el identificador que son los primeros 12 caracteres en formato reducida, el comando que hemos lanzado. Cuando ha sido creado y el tiempo que lleva funcionando:

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

013e49664e68 c319cf0f078c "/bin/bash -c 'while " About a minute ago Up About a minute 22/tcp, 80/tcp, 443/tcp, 8443/tcp big_goldstine

Para ver la salida del comando que hemos lanzado en el contenedor, podemos utilizar el comando docker logs y el nombre del contenedor:

#docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

013e49664e68 c319cf0f078c "/bin/bash -c 'while " 4 minutes ago Up 4 minutes 22/tcp, 80/tcp, 443/tcp, 8443/tcp big_goldstine

[root@docker ~]# docker logs –f big_goldstine

Si queremos parar el contenedor:

[root@docker ~]# docker stop big_goldstine

big_goldstine

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

013e49664e68 c319cf0f078c "/bin/bash -c 'while " 10 minutes ago Exited (137) 5 seconds ago

Si lo queremos volver a arrancar:

[root@docker ~]# docker start big_goldstine

Si queremos ejecutar un comando en un contenedor que esta corriendo:

 # docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

cd88d1945efc b45f1e1c24ef "/bin/bash" 5 minutes ago Up 5 minutes 22/tcp, 80/tcp, 443/tcp adoring_lalande

[root@docker ~]# docker exec -ti adoring_lalande uptime

17:04:36 up 2:16, 0 users, load average: 0.07, 0.11, 0.13

Una forma que podemos utilizar para conectarnos a un contenedor y que se siga ejecutando sin ningún problema, es decir cuando estamos dentro del contenedor ejecutamos exit y al salirnos el contenedor sigue corriendo:

 # docker exec -ti adoring_lalande /bin/bash

[root@cd88d1945efc /]# **exit**

exit

[root@docker ~]# docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

cd88d1945efc b45f1e1c24ef "/bin/bash" 7 minutes ago Up 7 minutes 22/tcp, 80/tcp, 443/tcp adoring_lalande

Para obtener información sobre los contenedores podemos ejecutar el comando ps con la opción –a nos dara todos los contenedores los que se estan ejecutando y los que han finalizado:

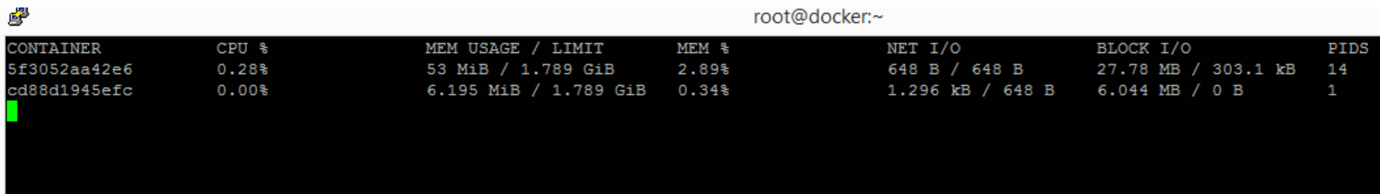
[root@docker ~]# docker ps –a

Para ver el ultimo contenedor que ha sido creado:

[root@docker ~]# docker ps –l

Para ver estadísticas de nuestros contenedores:

[root@docker ~]# docker stats



```
root@docker:~
CONTAINER       CPU %      MEM USAGE / LIMIT       MEM %     NET I/O              BLOCK I/O             PIDS
5f3052aa42e6    0.28%      53 MiB / 1.789 GiB      2.89%     648 B / 648 B        27.78 MB / 303.1 kB   14
cd88d1945efc    0.00%      6.195 MiB / 1.789 GiB   0.34%     1.296 kB / 648 B     6.044 MB / 0 B        1
```

Podemos observar el consumo de CPU, la memoria usuada que tenemos y el limite que tenemos de memoria, y lo que estamos utilizando de red tanto en entrada como en salida, y de disco tanto en entrada como en salida.

Tambien podemos utilizar para ver mas información sobre las aplicaciones que estan corriendo en un contenedor el comando:

 # docker top centos6

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	6097	6084	0	18:11	pts/2	00:00:00	/usr/bin/python /usr/bin/supervisord --configuration=/etc/supervisord.conf
root	6140	6097	0	18:11	pts/2	00:00:00	/usr/bin/python /usr/bin/supervisor_stdout
root	6142	6097	0	18:11	pts/2	00:00:00	php-fpm: master process (/etc/php-fpm.conf)
root	6143	6097	0	18:11	pts/2	00:00:00	/usr/sbin/httpd -c ErrorLog /dev/stdout -DFOREGROUND -D production
497	6247	6143	0	18:11	pts/2	00:00:00	/usr/sbin/httpd -c ErrorLog /dev/stdout -DFOREGROUND -D production
497	6248	6143	0	18:11	pts/2	00:00:00	/usr/sbin/httpd -c ErrorLog /dev/stdout -DFOREGROUND -D production
497	6249	6143	0	18:11	pts/2	00:00:00	/usr/sbin/httpd -c ErrorLog /dev/stdout -DFOREGROUND -D production

497 6250 6143 0 18:11 pts/2 00:00:00 /usr/sbin/httpd -c ErrorLog /dev/stdout -DFOREGROUND -D production

497 6251 6143 0 18:11 pts/2 00:00:00 /usr/sbin/httpd -c ErrorLog /dev/stdout -DFOREGROUND -D production

497 6252 6142 0 18:11 pts/2 00:00:00 php-fpm: pool app-www

497 6253 6142 0 18:11 pts/2 00:00:00 php-fpm: pool app-www

497 6254 6142 0 18:11 pts/2 00:00:00 php-fpm: pool app-www

497 6255 6142 0 18:11 pts/2 00:00:00 php-fpm: pool app-www

497 6256 6142 0 18:11 pts/2 00:00:00 php-fpm: pool app-www

Otro comando para ver información sobre un contenedor es docker logs contenedor:

[root@docker ~]# docker logs -f centos6

Podemos congelar un contenedor, con lo que pararemos los procesos de este contenedor con el comando;

[root@docker ~]# docker pause centos6

 # docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

5f3052aa42e6 c319cf0f078c "/usr/sbin/httpd-star" 36 minutes ago Up 36 minutes (Paused) 22/tcp, 80/tcp, 443/tcp, 8443/tcp centos6

Podemos decongelar un contenedor, con lo que continuara los procesos de este contenedor con el comando, es decir los procesos continuaran, en el STATUS veremos UP…

[root@docker ~]# **docker unpause centos6**

Si queremos lanzar un contedor con la opción **–d**, el contenedor se lanzará en modo Detached. En este modo cuando el proceso raíz haya finalizado el contenedor se detiene. Debemos tener pendiente que se le pasamos un comando al contenedor en ejecución, una vez el comando se ejecute, se detendrá. Es por esto que siempre debemos configura muy bien los que seseamos den los Dockerfiles, especialmente hablamos los EntryPoints y CMD.

 # docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.io/jdeathe/centos-ssh-apache-php latest c319cf0f078c 6 weeks ago 278.5 MB

docker.io/nickistre/centos-lamp latest b45f1e1c24ef 10 months ago 538.4 MB

[root@docker ~]# **docker run -dti --name centos6-lamp c319cf0f078c**

1b51865c48cdcc0f60f80538719fdb35c22361ea4171de30fe1b9dca6ce2fb56

[root@docker ~]# **docker ps -a**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1b51865c48cd c319cf0f078c "/usr/sbin/httpd-star" 8 seconds ago Up 6 seconds 22/tcp, 80/tcp, 443/tcp, 8443/tcp centos6-lamp

5f3052aa42e6 c319cf0f078c "/usr/sbin/httpd-star" 56 minutes ago Up 56 minutes 22/tcp, 80/tcp, 443/tcp, 8443/tcp centos6

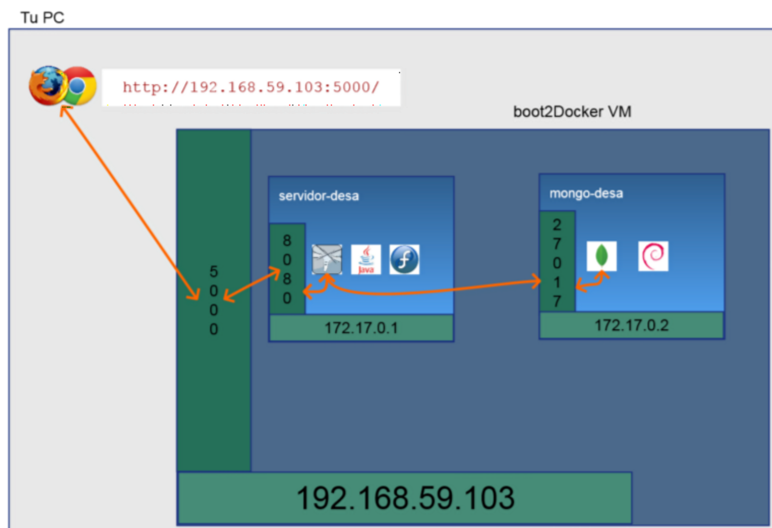
cd88d1945efc b45f1e1c24ef "/bin/bash" About an hour ago Up About an hour 22/tcp, 80/tcp, 443/tcp adoring_lalande

Laboratorio 4 Exponer puerto contenedor

Exponer puertos contenedor

- Para acceder a las aplicaciones dentro de los contenedores debemos "exponer" los puertos al servidor
- Usaremos las opciones
  - -P → Exponer puertos por defecto a puertos aleatorios.
  - -p puerto servidor : puerto contenedor → Manual
- utilizaremos docker inspect imagen para obtener los puertos.

Para ello se crea una regla de iptables.



Cuando nosotros ejecutraremos un contenedor que contenga una aplicación que deba ser acesible externamente, como podría ser un servidor web, nosotros por defecto solo podremos acceder a la misma desde el servidor Docker a la dirección privada del contenedo.

Ademas, un contenedor que este en la misma red de otro contenedor podrá acceder también a la dirección privada desde la aplicación que se esta ejectundo. Un ejemplo común es un contenedor que se esta ejecutando un servidor web y la aplicación que sirve se conecta a la base de datos que esta alojando este contenedor.

En este ejemplo utilizamos la imagen oficial del servidor web Apache:

 # docker run -dti httpd

Unable to find image 'httpd:latest' locally

Trying to pull repository docker.io/library/httpd ...

latest: Pulling from docker.io/library/httpd

10a267c67f42: Pull complete

0782edf7745a: Pull complete

f3a72c4d9d02: Pull complete

dd6ec65d8a55: Pull complete

1b7920e1c0df: Pull complete

5b99e4053015: Pull complete

e720548ad189: Pull complete

Digest: sha256:72b55a7c15a4ee3d56ff19f83b57b82287714f91070b1f556a54e90da5eee3fa

9aea6f58832aa66f86b080bc3d555d4c5c30b644d0fcfe45fa87cffe01a5b942

[root@docker ~]# **docker ps -a**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

9aea6f58832a httpd "httpd-foreground" 8 seconds ago Up 7 seconds 80/tcp fervent_bhabha

A través de la acción inspect podremos obtener que dirección IP esta utilizando:

[root@docker ~]# **docker inspect 9aea6f58832a | grep -i "iPAddress"**

"SecondaryIPAddresses": null,

"IPAddress": "172.17.0.3",

"IPAddress": "172.17.0.3",

Si utilizamos esta ip podremos acceder al contenedor de Docker o desde otro contenedor al contenedor recién creado:

[root@docker ~]# curl http://172.17.0.3

<html><body><h1>It works!</h1></body></html>

Con esto podemos acceder al apache del contenedor desde nuestro servidor de Docker, pero no desde la red de nuestra empresa.

Para ello tendremos que utilizar la exposición de puerto que consiste en reservar un puerto en el servidor Docker con el fin de redirigir las peticiones a un puerto específico del contenedor. Si queremos que el puerto del contenedor sea accesible a través de la dirección del servidor Docker podremos rederiguir las comunicaciones al puerto 80 dek servidor al puerto 80 del contenedor.

Para realizar esta tarea utilizaremos las opciones **-p** o -**P** el puerto o puerto a rediriguir:

-P (--publish-all): seleciona un puerto libre aleatorio del servidor donde se va a escuchar las peticiones a rediriguir.

-p (--publish-value): debemos de especificar un puerto manaualmente donde deseamos realizar la escucha. Si este puerto esta en uso, el contenedor fallara.

En el caso de la opción **-P**, Docker revisara los puerto que se han configurado en la imangen para ser expuestos y por cada uno de ellos generara uno aleatorio.

Podremos utilizar la acción ps para obtener dichos puertos.

[root@docker ~]# **docker stop 9aea6f58832a 0a7fc69afbdc**

9aea6f58832a

0a7fc69afbdc

[root@docker ~]# **docker rm 9aea6f58832a 0a7fc69afbdc**

9aea6f58832a

0a7fc69afbdc

[root@docker ~]# **docker ps -all**

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker.io/httpd	latest	e0645af13ada	2 weeks ago	177.5 MB
docker.io/jdeathe/centos-ssh-apache-php	latest	c319cf0f078c	3 months ago	278.5 MB
docker.io/nickistre/centos-lamp	latest	b45f1e1c24ef	12 months ago	538.4 MB

 # docker run -dtiP httpd

5eef23535eb6ef65684b37204e4e4349cfadcb6484ac4365b9f3528abf26d114

[root@docker ~]# docker ps -l

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
5eef23535eb6	httpd	"httpd-foreground"	12 seconds ago	Up 11 seconds	0.0.0.0:32768->80/tcp	mad_albattani

Ahora este puerto expuesto por el contenedor es accesibles desde el propio servidor Docker o desde la red de nuestra empresa a través de la ip del servidor de Docker:

http://192.168.1.150:32768/

It works!

Es este punto paramos el contenedor, lo destruimos y creamos uno nuevo para utilizar la opción **-p**:

En caso de utilizar la opción **-p** debemos elegir manualmente cual será el puerto del servidor que utiizaremos para la redireccion:

docker run -dti -p redireccion contenedor comando

 # docker run -dti -p 80:80 httpd

En este caso si nuestro servidor Docker no tenemos ninguna aplicación a la escucha en el puerto 80, Docker rediriguira las peticiones HTTP al contenedor.

En cualquier de los dos casos usando -p o -P podremos listar los puerto rediriguidos utilizando la acción port especificándole el contenedor del que queremos obtener la información:

#docker ps -l

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

5eef23535eb6 httpd "httpd-foreground" 11 minutes ago Up 11 minutes 0.0.0.0:32768->80/tcp mad_albattani

[root@docker ~]# docker port 5eef23535eb6

80/tcp -> 0.0.0.0:32768

Iptables

La redireccion de la comunicación entre el servidor y el contenedor se realiza utilizando iptables (permite acutar como firewall y como redireccionar de trafico a través de reglas). Podremos listar las reglas con el siguiente comando:

root@docker ~# **iptables -t nat -L DOCKER -v -n**

Chain DOCKER (2 references)

pkts bytes target prot opt in out source destination

0 0 RETURN all -- docker0 * 0.0.0.0/0 0.0.0.0/0

1 52 DNAT tcp -- !docker0 * 0.0.0.0/0 0.0.0.0/0 tcp dpt:32768 to:172.17.0.2:80

Laboratorio 5 Copia de Seguridad contenedores/imagenes

Copias de seguridad y restaurar

- Exportar contenedor:
  - Sintaxis: docker export [-o fichero] contenedor
  - Ejemplo: docker export web > backup-web.tar
- Importar contenedor: (como una imagen)
  - Sintaxis: docker import fichero/url nombre
  - Ejemplo: docker import backup-web.tar
- Exportar imagen:
  - Sintaxis: docker save [-o fichero] contenedor
  - Ejemplo: docker save imagen-web > imagen-web.tar
- Importar imagen:
  - Sintaxis: docker load [-i fichero]
  - Ejemplo: docker load -i imagen-web.tar

Es posible hacer una copia de seguridad de un contenedor, ya este en ejecución o detenido. A través de la acción export, empaquetara el contenido generara un fichero tar. La sintaxis es la siguiente:

 # docker export -o fichero.salida.tar contenedor

 # docker export contenedor > contenedor.tar

En este ejemplo vemos que tenemos el contenedor mad_albattani y vamos a salvarlo:

 # docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

5eef23535eb6 httpd "httpd-foreground" 28 minutes ago Up 28 minutes 0.0.0.0:32768->80/tcp mad_albattani

 # docker export mad_albattani > httpd.tar

 # ls -l

total 178196

-rw-r--r-- 1 root root 182453248 may 27 12:22 httpd.tar

Restaurar

No es posible restaurar el contenedor de forma automática, la restauración consiste en la creación de imagen basada en el contenido de la copia de seguridad del contenedor. Es posible hacer la restauración en un servidor diferente al que ha sido realizada la copia. La sintaxis es la siguiente:

 # docker import -M [Mensaje de importación] [fichero nombre:etiqueta]

 # docker import httpd.tar copaiapache

 # docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
copaiapache	latest	ccad1e13ae00	4 seconds ago	173.7 MB

Para realizar **copias de seguridad a imágenes**, a través de la acción save empaquetara el contenido y generara un fichero tar. La sintaxix posibles son las siguientes:

 # docker save [-o fichero.salida.tar] imagen

 # docker save imagen > imagen.tar

[root@docker ~]# docker save -o httpd.tar httpd

[root@docker ~]# ls -l

-rw------- 1 root root 186371072 may 27 12:35 httpd.tar

Restaurar:

Es posible restaurar una copia de seguridad previamente creada, ya sea en el mismo servidor o en otro, la sintaxis es la siguiente:

 # docker impor -i fichero.tar

 # docker impor < fichero.tar

Importamos la copia de seguridad de la imagen httpd.tar:

 # docker load -i httpd.tar

8d4d1ab5ff74: Loading layer [==>] 129.4 MB/129.4 MB

3291ee6ef6e0: Loading layer [==>] 3.584 kB/3.584 kB

457598cf1733: Loading layer [==>] 2.56 kB/2.56 kB

6f31e408653c: Loading layer [==>] 5.12 kB/5.12 kB

65956f7f583a: Loading layer [==>] 46.63 MB/46.63 MB

f1c7cb70e415: Loading layer [==>] 10.31 MB/10.31 MB

d4d87debc120: Loading layer [==>] 3.584 kB/3.584 kB

Loaded image: docker.io/httpd:latest

Para eliminar una imagen, tendremos que cumplir un requisito que es que no exista ningún contenedor que haga referencia la imagen que deseamos eliminar, el comando será el siguiente:

 # docker rmi imagen [opciones]

-f/--force: fuerza la eliminación de la imagen.

--no-prune: no elimina imágenes padre sin etiquetas.

En caso de querer eliminar una versión especificada y no la ultima (latest) deberemos especificar a la imagen la versión que deseamos eliminar:

 # docker rmi imagen:version [opciones]

En este ejemplo borramos la imagen httpd:

 # docker rmi httpd

Laboratorio 6 Convertir un contenedor en una Imagen

En una gran variedad de casos existirán cambios que hemos realizado en un contenedor que tendremos que mantener. En el caso de la instalación de una aplicación especifica, la configuración de la aplicación o distintas coniguraciones dentro del contenedor. A demás es habitual querer transferir esos cambios a otros sistemas de Docker para que puedan ejectuar en el contenedor con el mismo contenido.

La solución mas fácil, sin tener que hacer una copia de seguridad el contenedor y volver a restaurar el contenedor, es convertir dicho contenedor en una imagen. Esa imagen contendrá los cambios realizado en la imagen base, a demás podrmos definir el nombre de la imagen.

Docker nos ofrece la posibilidad de convertir un contenedor a una imagen a través del comando:

 # docker commit contenedor usuario/imagen:[versión]

Ademas, con las opciones -a para indicar el autor y -m para indicar la modificación realizada podemos mantener un buen control de versiones de nuestra propia imágenes.

En este laboratorio lanzaremos la imagen docker.io/nickistre/centos-lamp, la cual tendremos que configurar para realizar la instalación de Joomla_1.5.15-Spanish-pack_completo.tar, y la configuración de una base de datos llamada intranet, (el instructor guiara toda la practia de la instalación del productor joomal).

En el contenedor **docker.io/nickistre/centos-lamp**, tendremos que crar a un usuario llamado operador y cambiar el password al usuario root, para que podamos conectarnos a través de ssh

Comenzamos el laboratorio:

 # docker run -dtiP --name centos6 docker.io/nickistre/centos-lamp

Entramos en el contenedor y realizamos la creación del usuario operador y el cambio del password al usuario root:

 # docker exec -ti centos6 /bin/bash

[root@a56f79547575 /]#

Para salirnos del contenedor podemos ejecutar exit y comprobaremos que el contenedor seguirá corriendo con el comando:

 # docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a56f79547575 docker.io/nickistre/centos-lamp "supervisord -n" 3 minutes ago Up 3 minutes 0.0.0.0:32780->22/tcp, 0.0.0.0:32779->80/tcp, 0.0.0.0:32778->443/tcp centos6

Ahora podremos conectarnos a través de Winscp y **subir el archivo Joomla_1.5.15-Spanish-pack_completo.tar** al contenedor, en este caso al puerto 32780:

192.168.1.150:32780 como podemos observar es el puerto rediriguido del servidor Docker al contenedor centos6.

Tras subir el achivo de joomala nos conectamos a ssh al cotenedor y comanzamos la instalación de nuestra intranet Joomla en apache y mysql.

Tras la finalización y el correcto funcionamiento de la intranet, procedemos a converir el contenedor centos6 en una nueva imagen llamada intranet:

 # docker commit -m "Intranet con joomla" centos6 intranet

sha256:c1435f560b71dad3d785ad3793caa1872a356e3304c39570b1150f1683ea16fd

[root@docker ~]# docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
intranet	**latest**	**c1435f560b71**	**9 seconds ago**	**641.3 MB**
copaiapache	latest	ccad1e13ae00	About an hour ago	173.7 MB
docker.io/httpd	latest	e0645af13ada	2 weeks ago	177.5 MB
docker.io/jdeathe/centos-ssh-apache-php	latest	c319cf0f078c	3 months ago	278.5 MB
docker.io/nickistre/centos-lamp	latest	b45f1e1c24ef	12 months ago	538.4 MB

Si utilizamos el comando docker history podremos listar las modificaciones que hemos realizado a la imagen base hasta llegar a la imagen que hemos creado:

 # docker history intranet

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
c1435f560b71	4 minutes ago	supervisord -n	102.9 MB	Intranet con joomla
b45f1e1c24ef	12 months ago	/bin/sh -c #(nop) CMD ["supervisord" "-n"]	0 B	
<missing>	12 months ago	/bin/sh -c #(nop) EXPOSE 22/tcp 443/tcp 80/tc	0 B	
<missing>	12 months ago	/bin/sh -c #(nop) ADD file:c987af8588740fdd52	1.414 kB	
<missing>	12 months ago	/bin/sh -c #(nop) ADD file:77dbd518f0925a0244	22 B	
<missing>	12 months ago	/bin/sh -c mkdir -p /root/.ssh && touch /root	0 B	
<missing>	12 months ago	/bin/sh -c sed -ri 's/UsePAM yes/UsePAM no/g'	5.044 kB	
<missing>	12 months ago	/bin/sh -c ssh-keygen -q -N "" -t dsa -f /etc	3.353 kB	
<missing>	12 months ago	/bin/sh -c yum install -y openssh-server open	17.02 MB	
<missing>	12 months ago	/bin/sh -c pip install supervisor	2.707 MB	
<missing>	12 months ago	/bin/sh -c yum install -y python-pip && pip i	18.18 MB	
<missing>	12 months ago	/bin/sh -c yum install -y php php-mysql php-d	50.27 MB	
<missing>	12 months ago	/bin/sh -c service mysqld start	21.84 MB	
<missing>	12 months ago	/bin/sh -c echo "NETWORKING=yes" > /etc/sysco	15 B	
<missing>	12 months ago	/bin/sh -c yum install -y mysql mysql-server	47.11 MB	
<missing>	12 months ago	/bin/sh -c yum -y install httpd vim-enhanced	168.3 MB	
<missing>	12 months ago	/bin/sh -c rpm -Uvh https://dl.fedoraproject.	10.34 MB	
<missing>	12 months ago	/bin/sh -c #(nop) MAINTAINER Nicholas Istre <	0 B	

<missing> 19 months ago /bin/sh -c #(nop) ADD file:9ff85de7a936502e83 202.6 MB

<missing> 2 years ago /bin/sh -c #(nop) MAINTAINER The CentOS Proje 0 B

Etiquetado de imágenes

Observamos que con Docker podemos mantener un historial de versiones para las imágenes. Esto se realiza con etiquetas (tags). Vamos a utilizar la acción tag que nos permite modificar las etiquetas de una imagen.

 # docker tag imagen/imagen:tag

 # docker tag intranet intranet:0.1

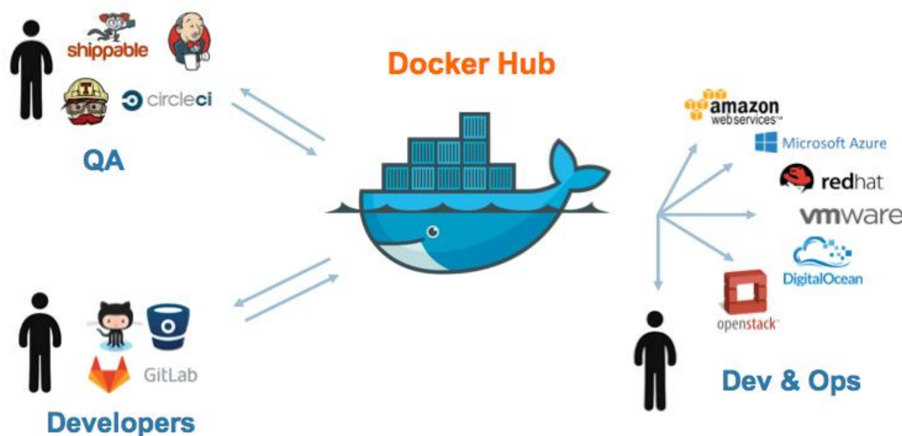
[root@docker ~]# docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
intranet	**0.1**	**c1435f560b71**	**8 minutes ago**	**641.3 MB**
intranet	latest	c1435f560b71	8 minutes ago	641.3 MB
copaiapache	latest	ccad1e13ae00	About an hour ago	173.7 MB
docker.io/httpd	latest	e0645af13ada	2 weeks ago	177.5 MB
docker.io/jdeathe/centos-ssh-apache-php	latest	c319cf0f078c	3 months ago	278.5 MB
docker.io/nickistre/centos-lamp	latest	b45f1e1c24ef	12 months ago	538.4 MB

Publicar una imagen en Docker Hub

Docker Hub que permite a los usuarios compartir las imágenes construidas, se podría decir que es el GitHub de los contenedores docker y quizá por ello el paralelismo en el nombre entre ambos. Docker Hub permite subir imágenes o usar las imágenes oficiales de postgresql, redis, mysql, ubuntu, rabbitmq, … y otra multitud de proyectos.

En la web oficial del repositorio de Docker **https://hub.docker.com/** es posible registrarse de forma gratuita para alojar las imágenes que hemos creado. Es importante tenen en consideración que las imágenes creadas pueden ser acesible públicamente, por lo cual no debemo de alojar datos comprometidos.



Una vez registrado con un usuario y una contraseña (llamada Docker ID), para almacenar imangenes dentro del repositorio nosotros debemos nombar a la imagen con el formato:

usuario/imangen:versión

Esta tarea puede realizarse con la acción tag.

Antes de realizar la acción de publicar la imagen debemos de autenticarnos con la cuenta previamente creada. Para ello utilizaremos la acción login.

 # docker login

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username:

Password:

Login Succeeded

Para publicar una imagen utilizaremos la siguiente sintaxis:

docker push usuario/imagen[:versión]

 # docker push agarciaf/intranet

Ahora podremos buscar las imágenes creadas por en nombre del usuario, o bien por el nombre de la imagen:

 # docker search agarcif

INDEX NAME DESCRIPTION STARS OFFICIAL AUTOMATED

[root@docker ~]# **docker search agarciaf**

INDEX	NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
docker.io	docker.io/agarciaf/centos6-joomla		0		
docker.io	docker.io/agarciaf/debian-vim		0		
docker.io	docker.io/agarciaf/intranet				

[root@docker ~]# docker search intranet

Los datos de loguin en Linux se almacenan en el siguiente fichero $HOME/.docker/config.json.

En el caso de Windows utilizara la siguiente ruta: $HOME\.docker\config.json.

Una vez almacenados los datos, es posible utilizar docker push, sin necesidad de introducir nuestras credenciales de nuevo.

Ademas es posible hacer Docker logout para cerrar la sesión en Docker Hub

 # docker logout

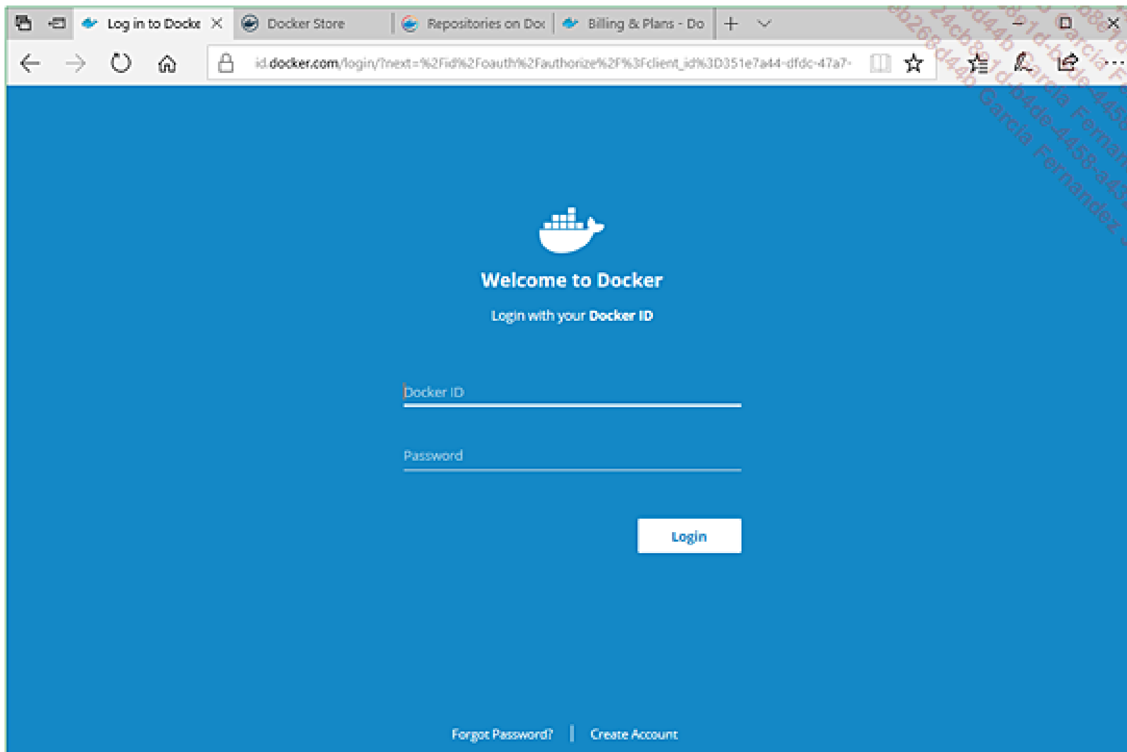
Gestión de la cuenta Docker Hub y almacenes privados

Hasta ahora, solo hemos consumido imágenes Docker desde el registro, y estas imágenes eran públicas. Por lo tanto, no hay necesidad de conectarse al Docker Store. Sin embargo, con el objetivo de acceder a las imágenes en el registro público, es necesario tener una cuenta activa y conectarse. Esta identificación también sirve para acceder a los almacenes privados. Como su nombre indica, los almacenes privados son almacenes de imágenes Docker, a los que solo pueden acceder las personas con autorización para explotarlas.

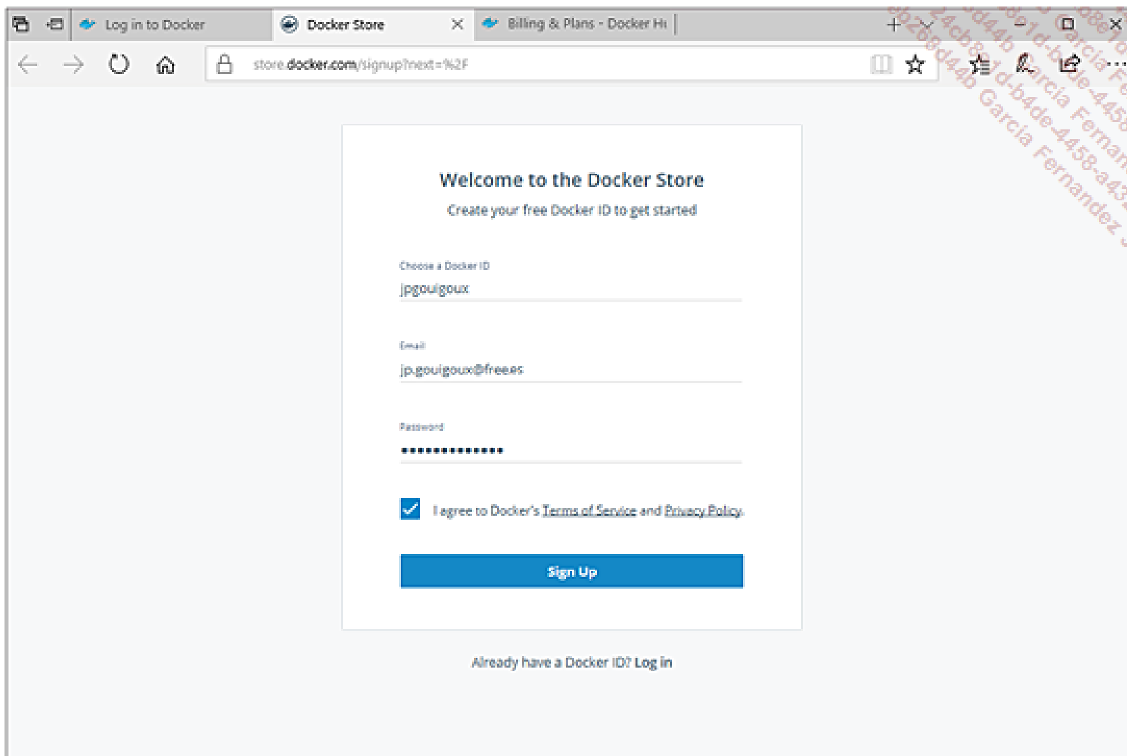
Docker ofrece cuentas gratuitas o de pago. La limitación de las primeras es que solo permiten un único almacén privado. El número de almacenes públicos es ilimitado (en la actualidad).

a. Creación de una cuenta

El procedimiento de creación de una cuenta no necesita explicaciones. Es suficiente con conectarse a https://hub.docker.com y hacer clic en el comando **Login**, para acceder a la ventana de conexión, en la que el enlace **Create Account** permite solicitar una cuenta:



La siguiente página no pide nada especial:



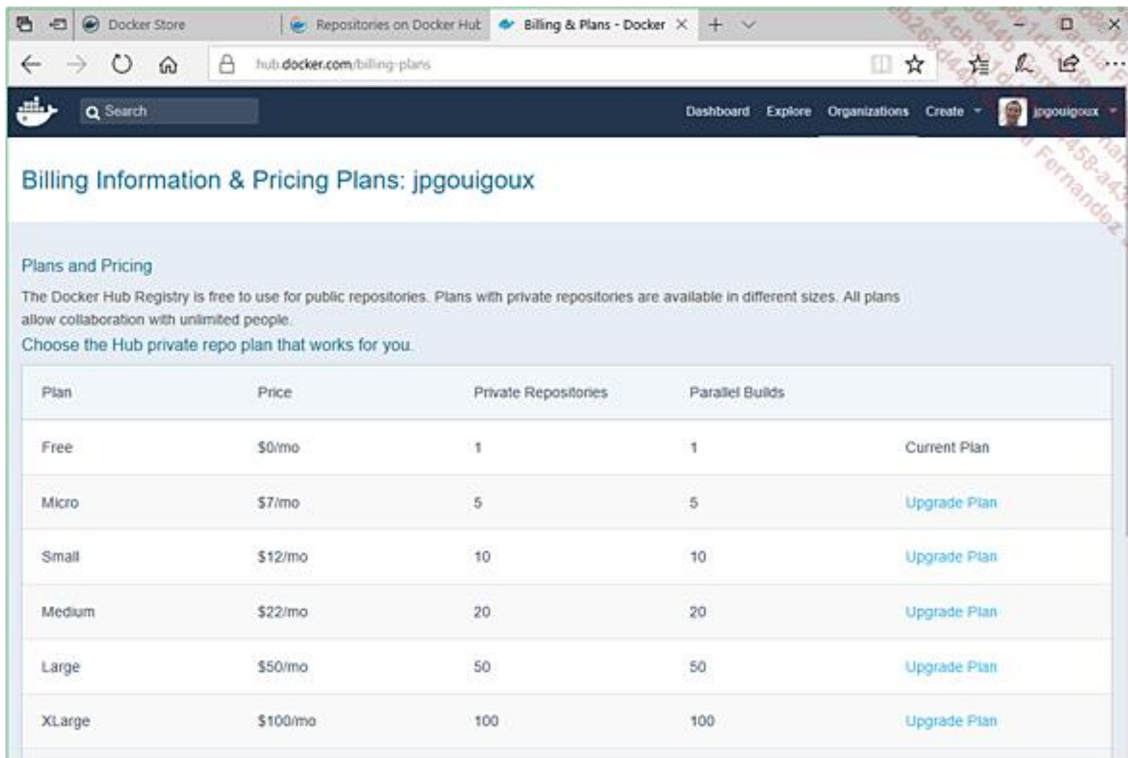
Una vez visitado el enlace de confirmación que se envía al correo electrónico, la cuenta está lista para la conexión y la imagen de su perfil aparece en la parte superior derecha de la interfaz y confirma que está conectado.

b. Características de la cuenta

La cuenta que se acaba de crear es una cuenta gratuita. No está limitada respecto al número de almacenes de imágenes públicos, pero solo permite un único almacén privado. Preste atención al sentido de la palabra almacén; en el lenguaje Docker, se trata de un conjunto de imágenes con el mismo nombre, pero potencialmente las etiquetas son diferentes.

Otro límite es que solo se puede lanzar a la vez una operación de compilación de una imagen, desde su código fuente en GitHub (ver un poco más adelante los detalles). Por necesidades de las pruebas o de los desarrollos, esto en general es suficiente.

Si estas condiciones son demasiado restrictivas para su uso, es posible recurrir a ofertas de pago, que le darán acceso a más almacenes privados y capacidad de compilaciones en paralelo (el vocablo inglés es build). La dirección https://hub.docker.com/billing-plans lista las diferentes ofertas existentes a día de hoy:



Un clic en el botón **Upgrade Plan**, correspondiente a la oferta elegida, conduce a una interfaz de elección del medio de pago. A continuación, la oferta puede evolucionar en función de las necesidades.

En función de sus necesidades, puede ser pertinente establecer su propio registro privado o bien echar un vistazo a las ofertas comerciales como Azure Container Registry. Un capítulo del presente libro se dedica a la gestión de los registros.

c. Automated build y cuenta GitHub

Como se ha explicado anteriormente es posible hacer que Docker Hub sea responsable de la compilación de una imagen Docker, a partir del código fuente depositado en GitHub. Este modo de funcionamiento marcado como Automated Build en los almacenes, permite dar una garantía al usuario final de que la imagen en Docker Hub se corresponde con el archivo Dockerfile mostrado, así como permitir al desarrollador deshacerse completamente de esta operación, que se realizará automáticamente en cada modificación del almacén del código fuente.

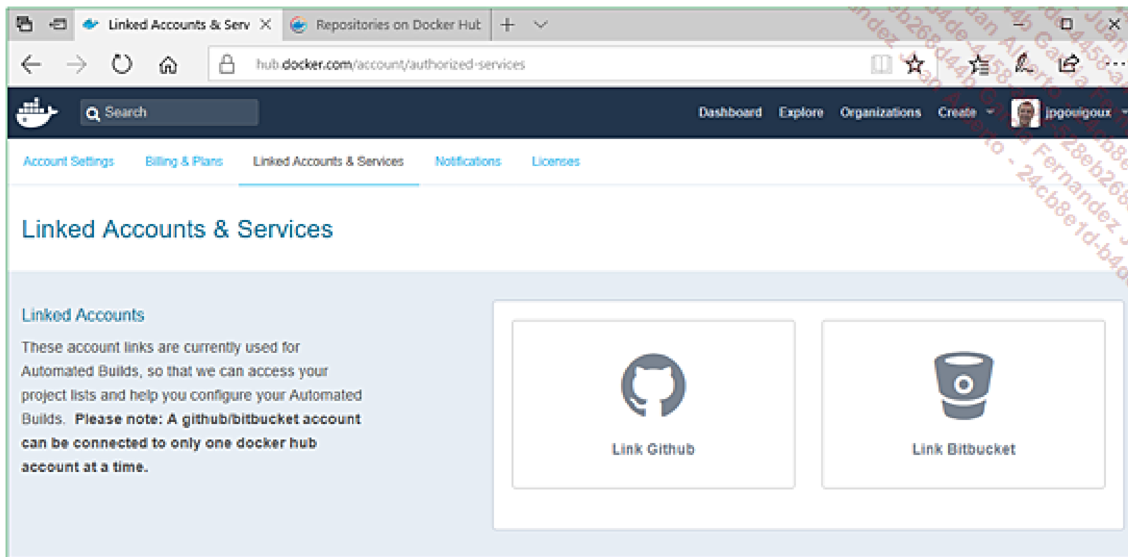
Para esto, es necesario que las cuentas Docker Hub y GitHub estén relacionadas. Este último punto merece una explicación un poco más detallada que para la creación de una cuenta.

Conéctese con la cuenta creada en Docker Hub: http://hub.docker.com

Pulse en el nombre de su cuenta, en la parte superior derecha.

Seleccione el comando **Settings**.

Vaya a la pestaña **Linked Accounts & Services**.



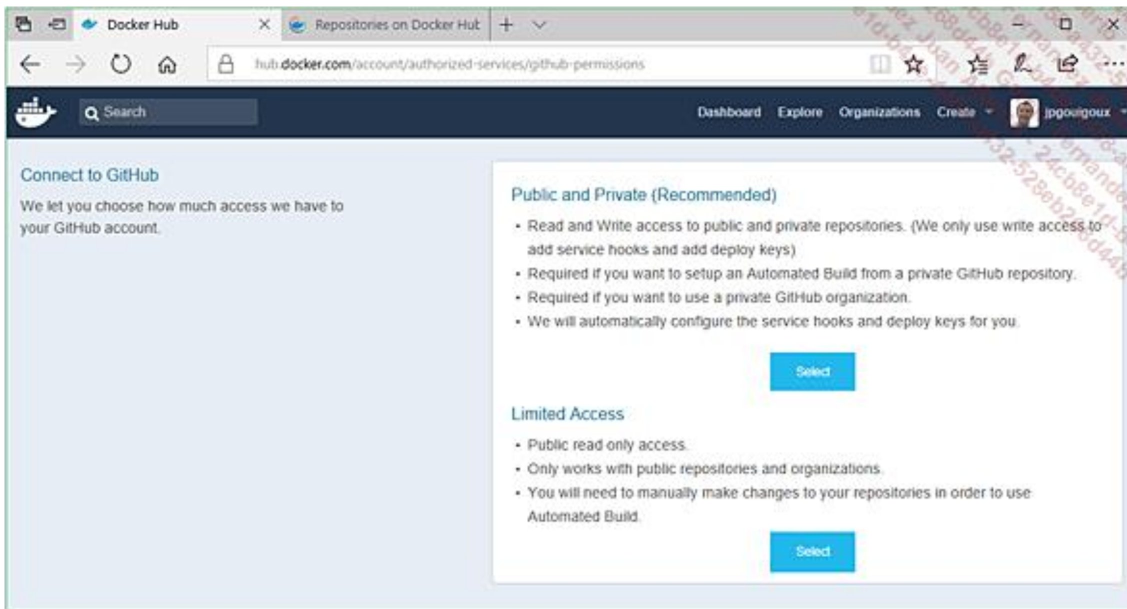
Como se ha explicado en la página anterior, el enlace con una cuenta de tipo GitHub o Bitbucket le permite establecer un sistema de build automático de sus imágenes. Si ha puesto en marcha un almacén GitHub que contiene un archivo Dockerfile (de manera tradicional en la raíz, incluso si es posible funcionar de otra manera), es posible utilizar un webhook para que cada modificación añadida al almacén GitHub lance la compilación de la imagen Docker y su puesta a disposición en la nueva versión en Docker Hub.

Un webhook es un tipo de evento transmitido entre dos aplicaciones web. Se caracteriza por una llamada de URL realizada por el emisor, acompañada de argumentos que dependen del contexto. El receptor puede desencadenar una operación particular en función de estos valores. La tecnología de las webhooks no se específica en Docker Hub o GitHub. Se trata de una práctica habitual en los servicios web.

Vamos a establecer esta solución en la solución mathapi, utilizada como el ejemplo mostrado anteriormente.

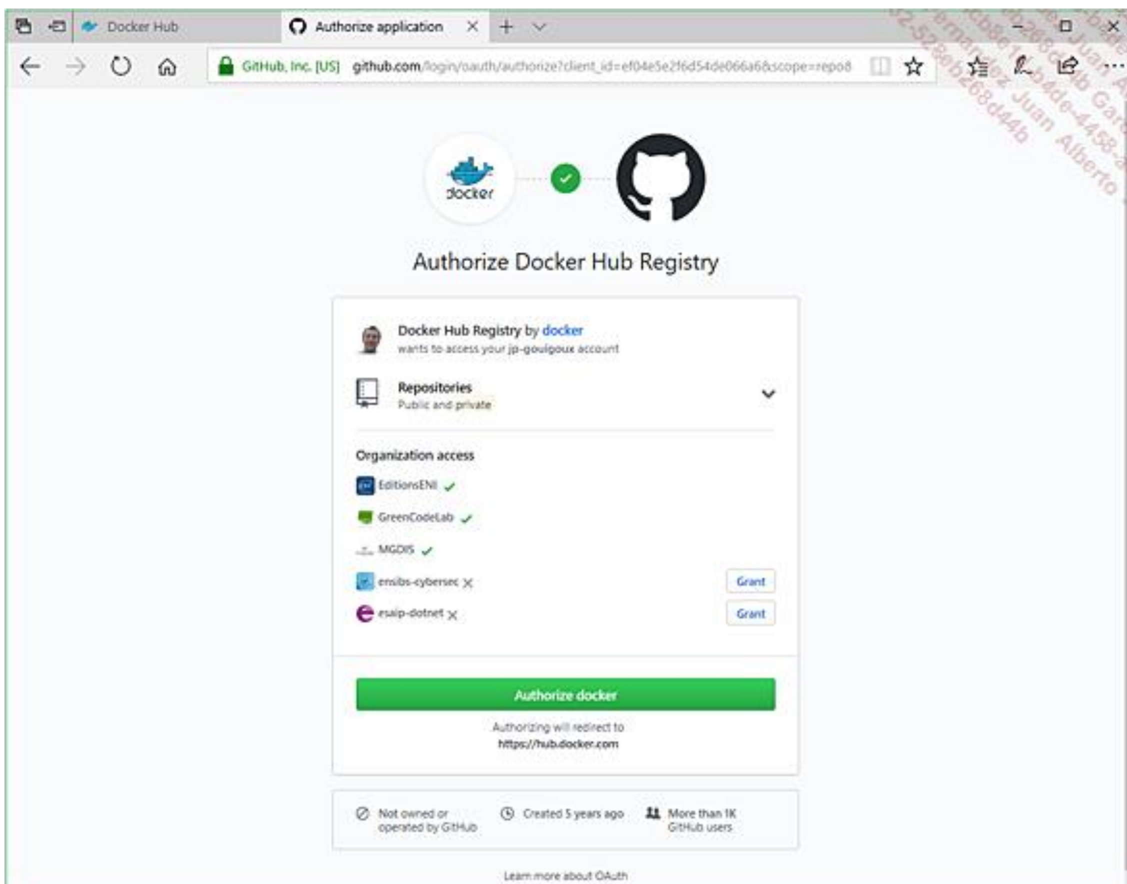
Pulse en **Link Github** en el icono GitHub.

En la pantalla siguiente, seleccione la opción de acoplamiento recomendada. Esta opción da más permisos a Docker Hub para manipular su cuenta GitHub y precisamente, establecer automáticamente las webhooks para usted.



Si no está conectado a GitHub, es necesario realizar esta autenticación.

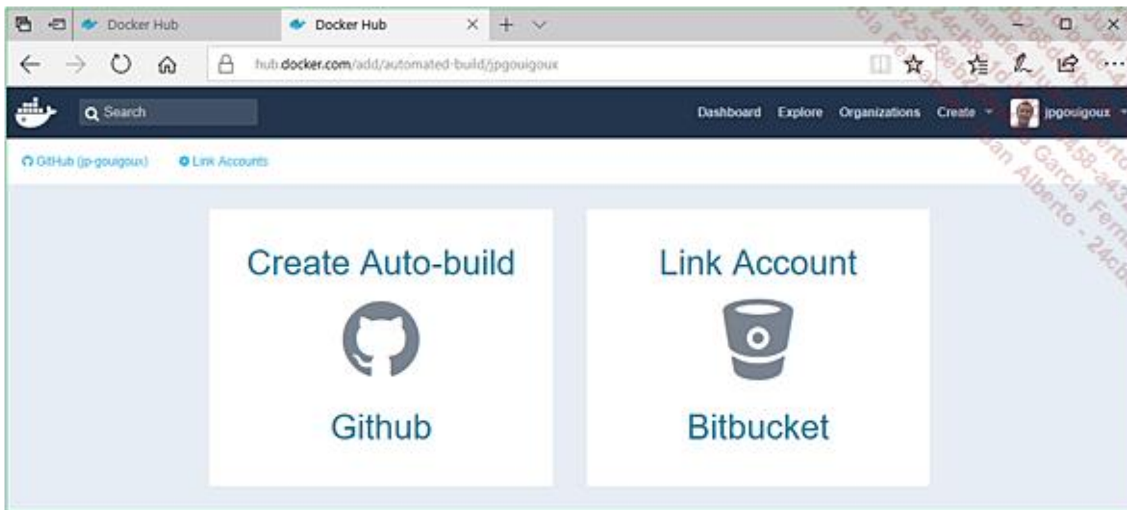
En la pantalla que sigue, valide la autorización dada sobre los almacenes GitHub que desee:



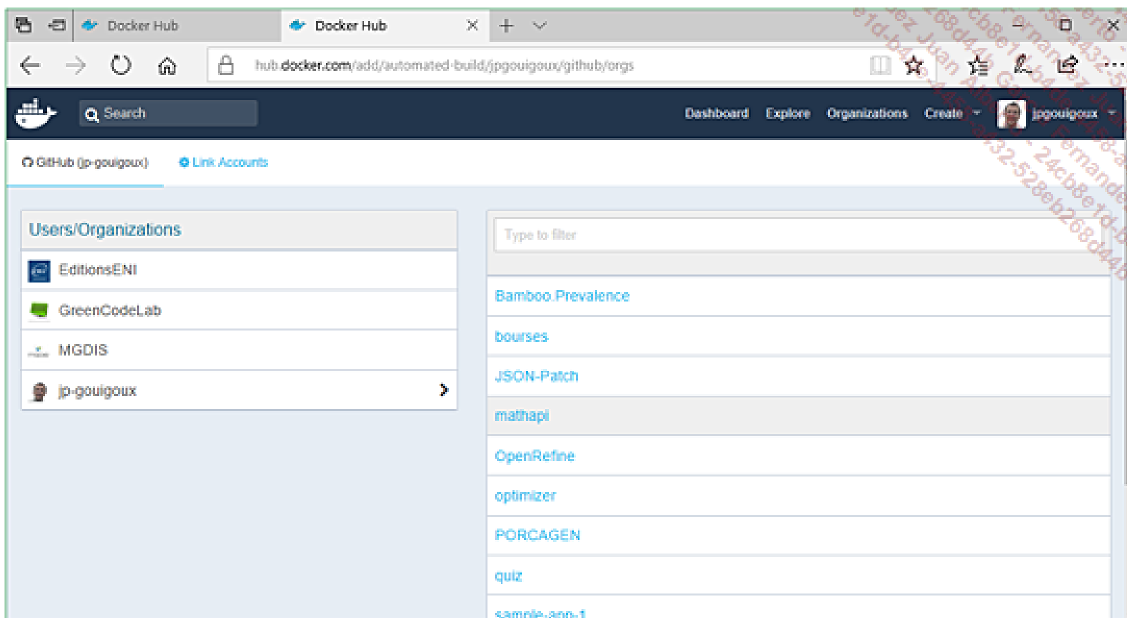
Es posible que tenga que introducir su contraseña GitHub de nuevo, para confirmar esta autorización.

De vuelta a su cuenta Docker Hub, pulse en el botón **Create** y después en **Create Automated Build**.

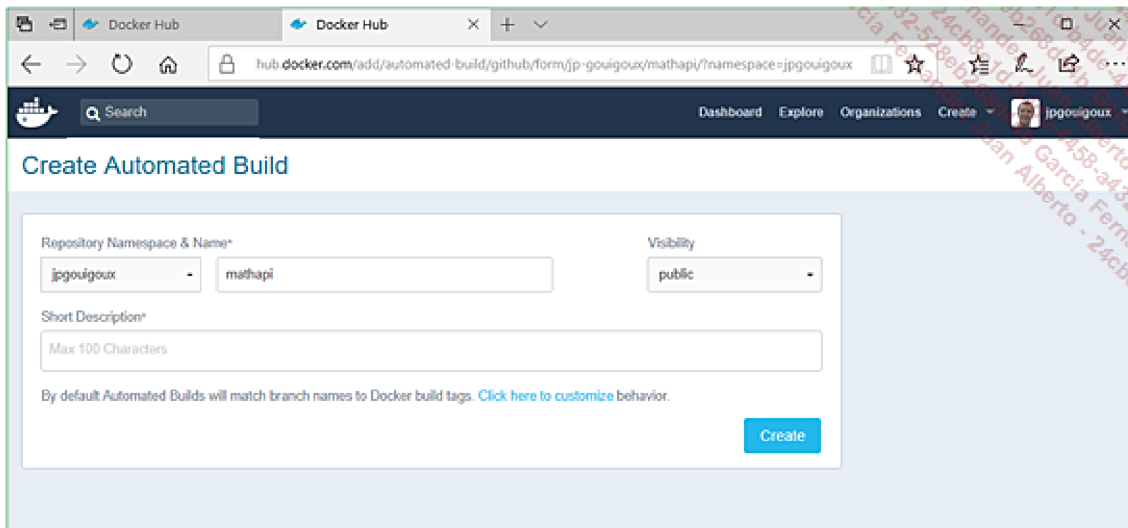
Seleccione el tipo de fuente que desea, en nuestro ejemplo **GitHub**.



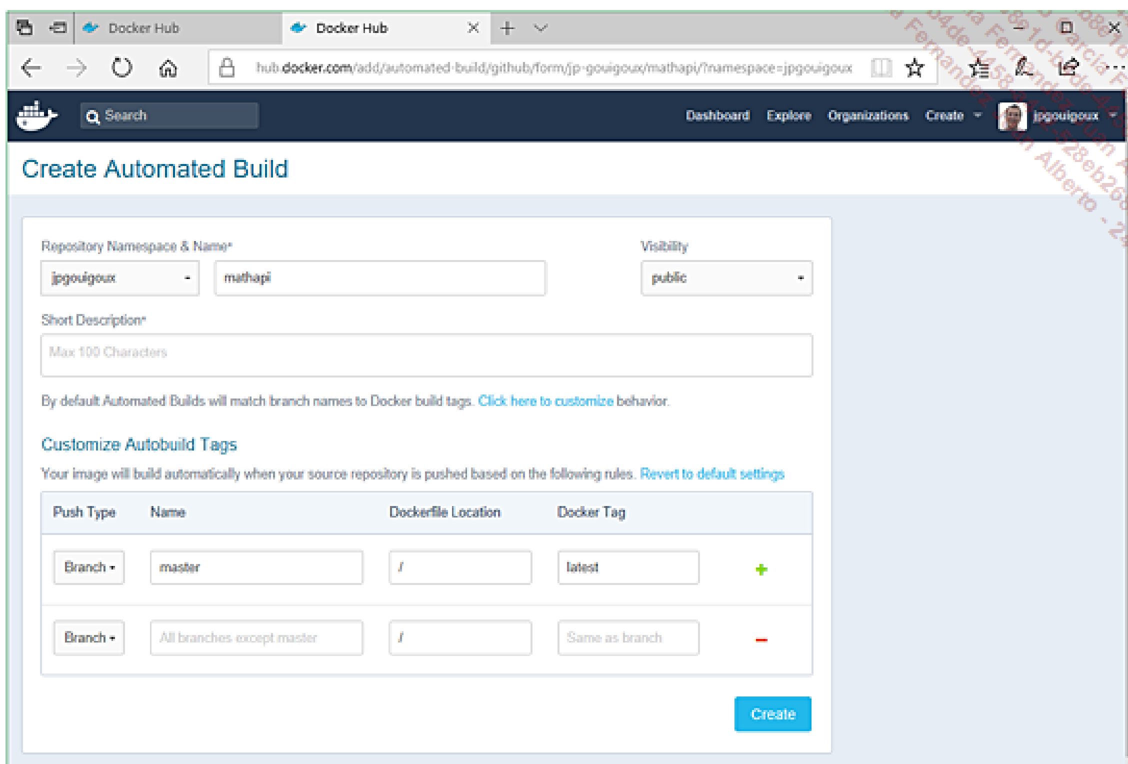
La lista de sus almacenes GitHub se muestra, agrupados por organización. Seleccione el almacén de origen a compilar.



Se muestra una ventana con varias opciones.

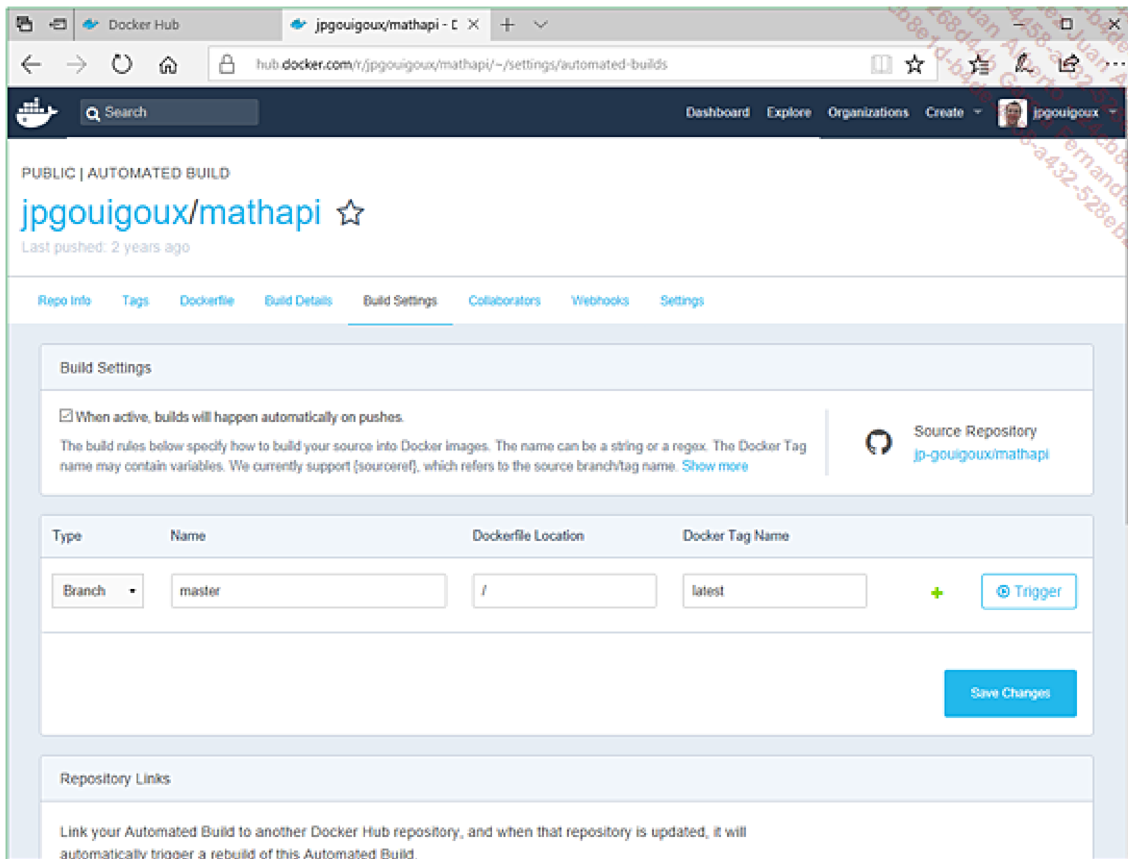


Pulse en **Click here to customize behavior** para que aparezcan las opciones de compilación:



Es posible lanzar la build automatizada cuando se modifica una rama o por supuesto, una etiqueta (Tag). La segunda columna permite especificar el nombre de la rama o de la etiqueta en cuestión. La tercera se corresponde con la localización del archivo Dockerfile que hay que utilizar. Habitualmente, está en la raíz del almacén GitHub pero si no es el caso, esta opción permite adaptar el comportamiento de la build automática. Para terminar, la cuarta columna en las líneas de opción permite especificar la etiqueta que se adjuntará a la imagen Docker, una vez compilada y ubicada en el almacén Docker Hub.

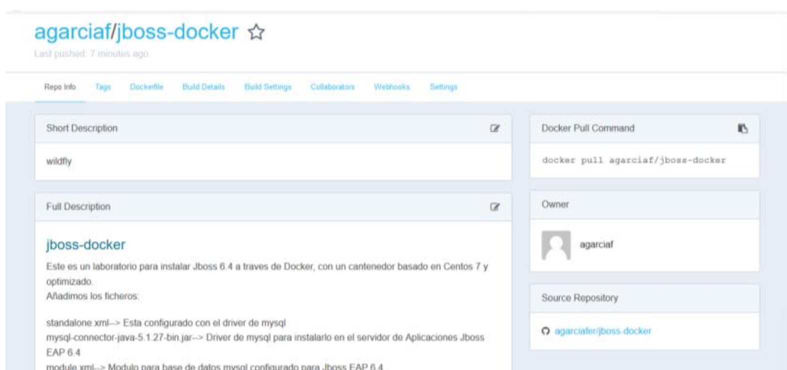
Como no utilizaremos estas opciones, puede pulsar en **Revert to default settings**.



Después de haber introducido una descripción (es un campo obligatorio), pulse en **Create**.

La interfaz apunta entonces al almacén creado junto con el código fuente y es posible encontrar los argumentos de lo que se acaba de configurar en la pestaña **Build Settings**.

El botón **Trigger** permite lanzar manualmente una build de la imagen a partir del código fuente. También hay un enlace a los resultados de la build, en la pestaña **Build Details**. Esto es práctico para solucionar posibles problemas. También podemos comprobar que el almacén Docker Hub ha recuperado y utilizado el Readme.md de GitHub y que una pestaña muestra el Dockerfile utilizado, lo que es mucho más limpio que dirigir simplemente por medio de un enlace al almacén GitHub, como se había implementado en el primer enfoque:



Esta manera de proceder con un almacén de código fuente, relacionado por medio de un evento al almacén de imagen por un webhook, es el estándar de facto de producción de imágenes Docker, y se corresponde con la filosofía de separación correcta de las responsabilidades que impera en los servicios web y más allá, en cualquier sistema informático correctamente urbanizado. Cada uno debe hacer su trabajo y solo su trabajo, de manera que se haga lo mejor posible.

Para más detalles de la configuración avanzada de las builds automatizadas, hay una documentación muy completa disponible en http://docs.docker.com/docker-hub/builds.

Para este laboratorio utilizaremos los archivos, de la carpeta **Dockerfile-Wildfly**, relacionando nuestro proyecto de github con nuestro repositorio de imágenes de docker en docker-hub.

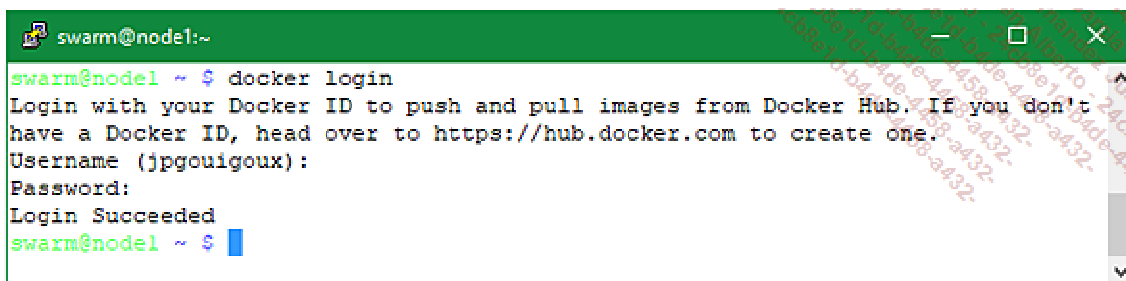
d.Conexión a la cuenta en línea de comandos

El modo de operación de las imágenes en el registro, sobre un navegador, se presta a la búsqueda de imágenes por palabras clave o a su administración, pero cuando se trata de poner en producción, es necesario tomar el control de estas imágenes e instalarlas de manera efectiva. Aunque existen interfaces gráficas para Docker, en la práctica se utiliza la línea de comandos. Con el objetivo de acceder a los almacenes privados, es necesario poder conectarse a su cuenta Docker Hub desde la línea de comandos.

Conexión a una cuenta Docker Hub

```
docker login
```

La conexión solicita indicar el nombre de la cuenta creada en Docker Hub, la contraseña, así como una dirección de correo electrónico:



En el siguiente capítulo volveremos sobre la herramienta de la cuenta Docker, para poder añadir imágenes compiladas localmente en el registro, de manera pública o privada. Por el momento, terminamos esta sección con el comando inverso del anterior:

Desconexión de la cuenta Docker Hub

```
docker logout
```

Es importante observar que es posible utilizar Docker sin tener cuenta en DockerHub. En efecto, hay numerosas ventajas que se pueden extraer de las imágenes existentes. Incluso cuando un usuario crea sus propias imágenes, perfectamente puede pasarlas de una máquina a otra sin recurrir al alojamiento ofrecido por Docker. La primera solución es exportar las imágenes en los archivos. La segunda es establecer su propio almacén de imágenes. Estudiaremos estas dos posibilidades en el siguiente capítulo.

e. Webhook en evento de push en Docker Hub

Más atrás hemos visto el uso de las webhooks de GitHub para Docker Hub, para lanzar la compilación automática de una imagen durante una operación de modificación de una rama o etiqueta, en un almacén GitHub dado. También existe una funcionalidad de webhook en la que Docker Hub es el emisor.

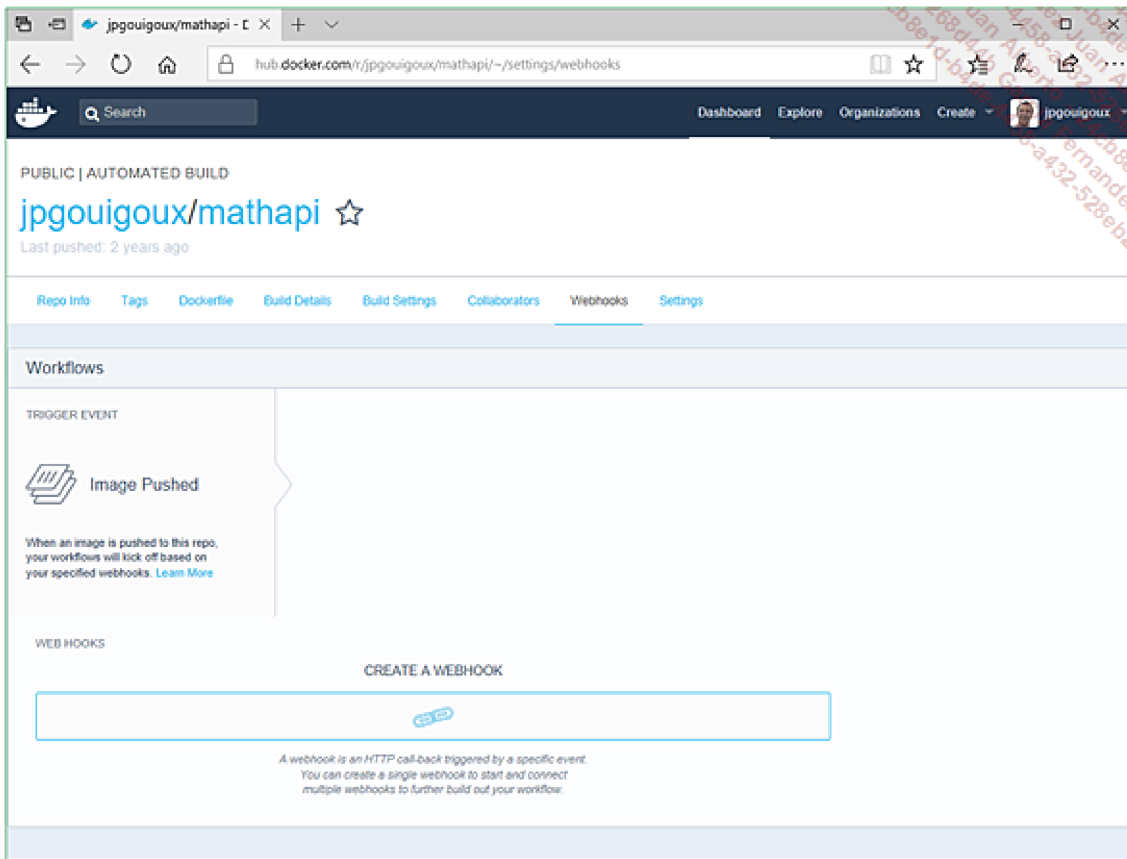
Las páginas de la documentación Docker correspondientes a esta funcionalidad (https://docs.docker.com/docker-hub/repos/#webhooks y https://docs.docker.com/docker-hub/webhooks/), ofrecen un ejemplo del contenido JSON (JavaScript Object Notation), que se enviará a un callback durante un evento de push:

```
{
  "callback_url":
```

```
"https://registry.hub.docker.com/u/svendowideit/busybox/hook/2141bc0cdec4hebec411i4c1g
40242eg110020/",
  "push_data": {
    "images": [
      "27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3",
      "51a9c7c1f8bb2fa19bcd09789a34e63f35abb80044bc10196e304f6634cc582c",
        "..."
    ],
    "pushed_at": 1.417566822e+09,
    "pusher": "svendowideit"
  },
  "repository": {
    "comment_count": 0,
    "date_created": 1.417566665e+09,
    "description": "",
    "full_description": "webhook triggered from a ’docker push’",
    "is_official": false,
    "is_private": false,
    "is_trusted": false,
    "name": "busybox",
    "namespace ": "svendowideit",
    "owner": "svendowideit",
    "repo_name": "svendowideit/busybox",
    "repo_url": "https://registry.hub.docker.com/u/svendowideit/busybox/",
    "star_count": 0,
    "status": "Active"
  }
}
```


El callback solo se llama si la operación de push se desarrolla correctamente.

En la actualidad, el único evento desencadenado por Docker Hub es el que se corresponde con esta operación de push. Este evento se repite en cada operación de almacenamiento, incluso si la imagen ubicada no ha cambiado. Además, se activa en un almacén manual, pero también durante un almacenamiento como consecuencia de una compilación automática. La configuración de un webhook se realiza en la pestaña del mismo nombre en un almacén Docker Hub.

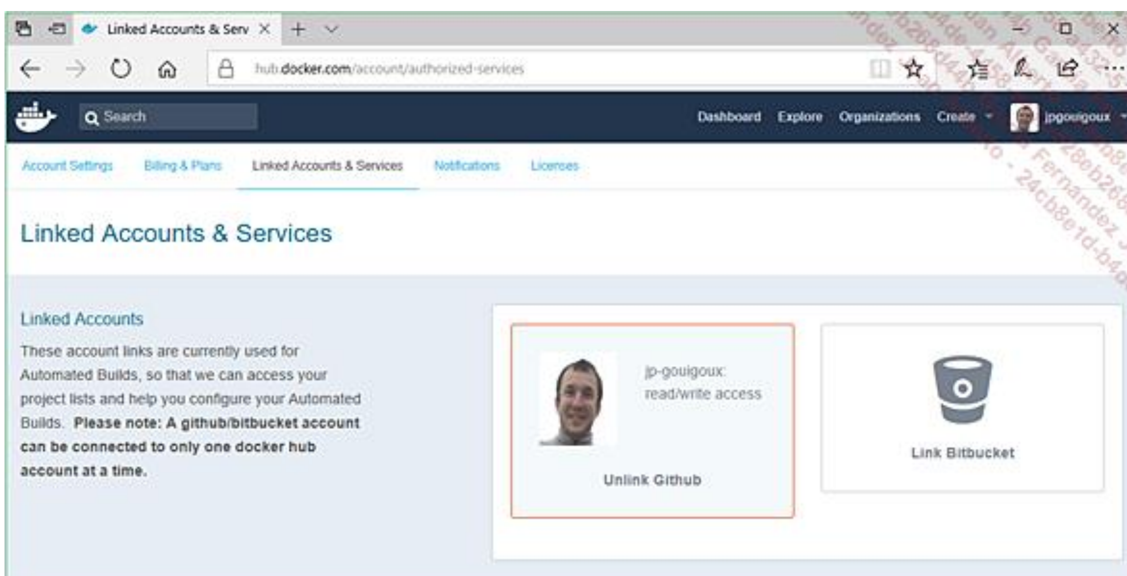


En un almacén estándar, también es posible encadenar las webhooks, con una ejecución en sucesión (la llamada a la siguiente callback en una lista encadenada, solo se realizará cuando la anterior se haya validado, devolviendo correctamente la URL llamada), pero esto va más allá del marco del presente libro.

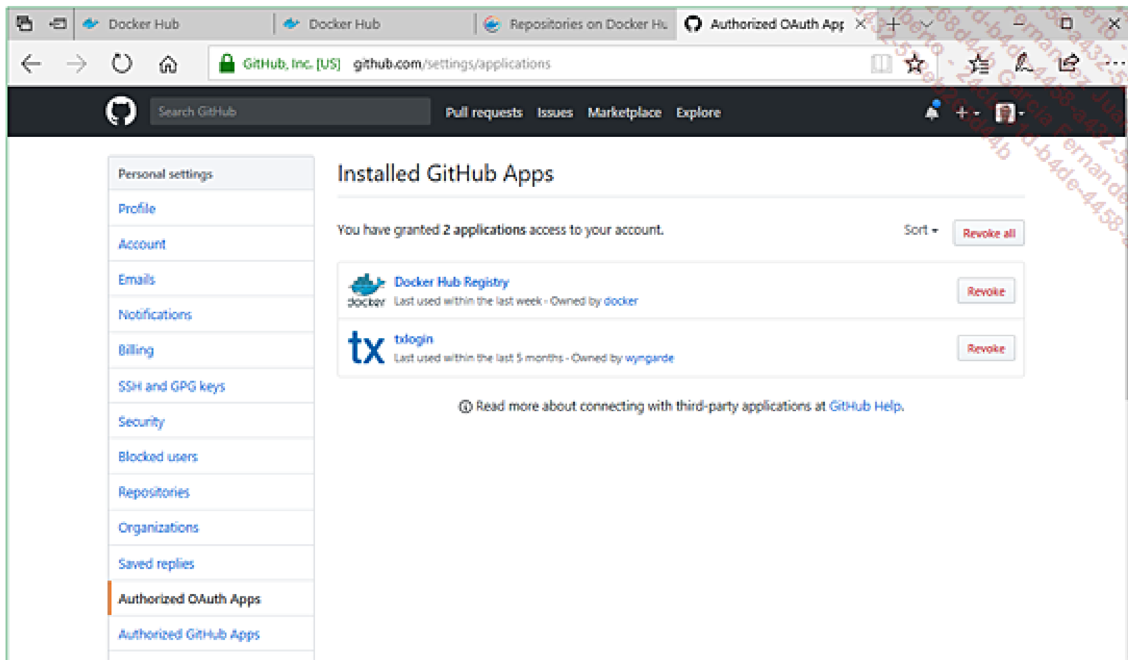
f. Desconexión de las cuentas Docker Hub y GitHub

Si por casualidad desea desconectar las cuentas Docker Hub y GitHub, el primer reflejo sería realizar la operación inversa a la mostrada más arriba, accediendo a su perfil Docker Hub y sus argumentos.

En la pestaña **Linked Accounts & Services**, sería posible eliminar el enlace a GitHub:



Sin embargo, esto no es suficiente para poner la situación en su estado inicial, porque también haría falta eliminar la autorización de Docker Hub en los argumentos de GitHub, haciendo clic en el enlace **Revoke** asociado en la lista **Authorized OAuth Apps**:



Preste atención con eliminar el enlace en primer lugar. Si elimina la autorización en primer lugar, el icono de eliminación de enlace ya no se podría pulsar y sería necesario desconectarse y volver a conectarse a su cuenta Docker Hub para poder desconectar la cuenta.

Repositorio Local

Primeros pasos para un registro privado

El manejo de archivos de almacenamiento sigue un enfoque un poco simplista respecto a la sofisticación de un registro, incluso si el hecho de utilizar Docker Hub pudiera ser un impedimento por razones de confidencialidad. Afortunadamente, existe una solución ideal que acumula las ventajas de las dos soluciones: el registro privado.

Hemos visto anteriormente que era posible beneficiarse, en la oferta gratuita, de un almacén privado para una imagen en el registro Docker Hub, pero aquí hablamos de crear nuestro propio registro y por tanto, eliminar esta limitación. Las herramientas necesarias son gratuitas, el coste reside en la explotación y el alojamiento eventual del servidor.

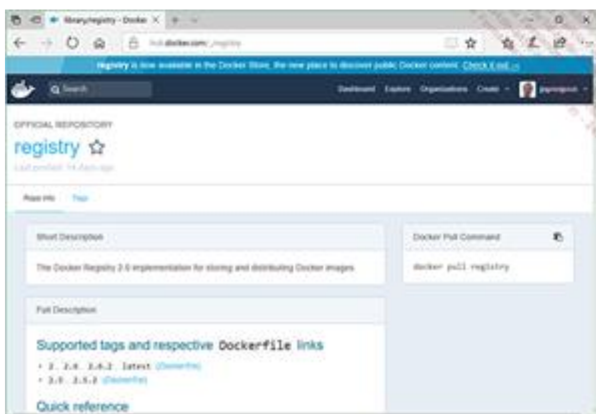
1. Advertencia sobre el antiguo registro

En primer lugar, tenga cuidado con no utilizar el registro antiguo. Hasta hace poco, el proyecto docker-registry (https://github.com/docker/docker-registry) se utilizaba para establecer un registro privado. Este proyecto, escrito en Python, se detiene en la versión 0.9.1. Ha salido una versión más reciente, completamente rescrita en Go (el lenguaje principal de desarrollo de Docker), y hace que el anterior registro haya quedado obsoleto. Esta versión se encuentra en https://github.com/docker/distribution.

La advertencia es necesaria porque este proyecto aparece en numerosos artículos y enlaces, así como en la primera página de resultados de una búsqueda sobre docker private registry. Al haber pasado dos años desde la primera edición del presente libro, puede haber dejado de pensar que esto siga sucediendo.

2. Imagen Docker en local

¿Qué hay más sencillo para desarrollar un servidor, que utilizar una imagen pública y arrancar un contenedor con ella? Esto es exactamente lo que vamos a hacer, utilizando la imagen oficial llamada registry, para crear nuestro propio servidor de registro. Una ventaja colateral es que la advertencia anterior sobre el carácter obsoleto del antiguo proyecto, es menos importante: en efecto, library/registry en Docker Hub, es el único almacén que contiene las versiones de Dockerfile, radicalmente diferentes entre la 0.9.1 y la 2.0. Además, para Docker Hub, las cosas están claras y solo se muestran desde ahora las versiones de la rama 2.0. La antigua versión ya no está disponible:



Algunas imágenes que creamos no deben ser accesibles públicamente, en estos casos, no tendremos acceso a intenet desde diferentes redes. La solución es desplegar un repositorio local donde alojaremos nuestras imágenes y serán acesibles a diferentes servidores Docker.

Para desplegar un repositorio local, ejecutaremos un contenedor usando una **imagen oficial** llamada **reistry:2**. El puerto por defecto para la comunicación a un registro Docker es **5000/Tcp**, el comando para ejecutar esta imagen y escuchas a dicho puerto es el siguiente:

 # docker run -d -p 5000:5000 --restart=always --name registry registry:2

 # docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

657c1b7cdbf1 registry:2 "/entrypoint.sh /etc/" 20 seconds ago Up 19 seconds 0.0.0.0:5000->5000/tcp registry

Una vez ejecutado nuestro repositorio local, podemos etiquetar las imágenes con el formato:

servidor:5000/nombre (por ejemplo, localhost:5000/imagen). Se muestras a continuación las acciones commit y con push:

 # docker commit intranet localhost:5000/intranet

sha256:27e5ade02c15449a6954caf577f9701510bfd091921cb5f66e3537a57659c4eb

 # docker push localhost:5000/intranet

 # docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost:5000/intranet	latest	27e5ade02c15	2 minutes ago	646.7 MB

Las imágenes publicadas con push se alamacenaran dentro del contenedor que ejecuta el repositorio local. Se recomienda utilizar un volumen para almacernarlo externamente al contenedor, porque permitirar ser reutilizado en otro servidor en caso de problema con el actual y nos dara mas flexibilidad a la hora de actualizar el repositorio. Para ello utilizaremos la opción -v para especificar un directorio del servidor Docker para enlazar con el directorio /var/lib/registry dentro del contenedor que es donde se alojan las imágenes.

#docker run -d -p 5000:5000 --restart=always --name registry -v /registro:/var/lib/registry registry

4c239c4147db2d5d51ec4523f7fa5a3cf3fa9f2098c25c52e711198bb00d6a44

Una vez vuelto a ejecutar nuestro registro local, deberemos de volver a publicar la imagen previamente etiquetada, posteriormente podemos ver en el directorio del servidor Docker el contenido (/registro).

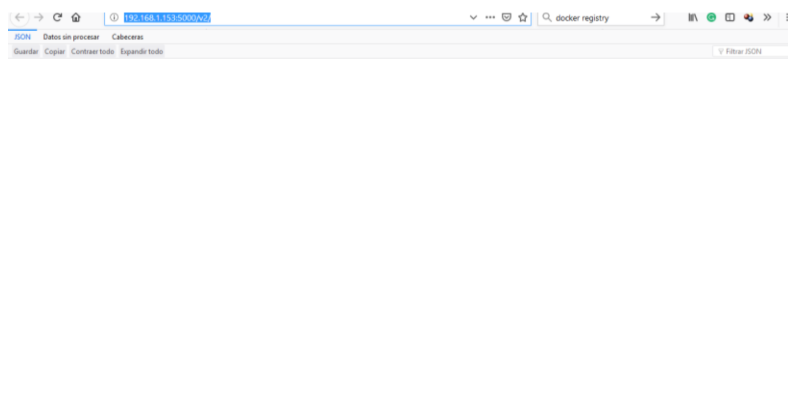
 # docker tag alpine:3.7 localhost:5000/alpine

 # docker push localhost:5000/alpine

 # ls -l /registro/docker/registry/v2/repositories/alpine

Un primer vistazo en un navegador que apunta al puerto 5000 (o a cualquier otro puerto que haya seleccionado para exponer su registro), permite ver que el servidor está activo:

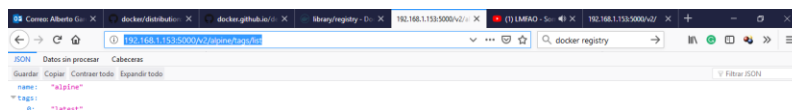
http://192.168.1.150:5000/v2/



La URL /v2/ nos da acceso al API en versión 2 del registro y en ausencia de cualquier argumento o porción de URL adicional, el API solicitado devuelve una lista vacía, pero el hecho de que veamos los separadores de lista de JSON (los paréntesis) en la pestaña **Datos sin procesar** que proporciona Firefox, prueba que el servidor responde correctamente.

Volviendo a nuestro navegador y escribiendo una URL con el nombre de nuestra imagen, seguida de /tags/list vemos que la imagen se ha situado con sus tags. De esta manera confirmamos, su correcto funcionamiento:

http://192.168.1.150:5000/v2/alpine/tags/list



La imagen ahora se almacena en un registro, desde donde usted puede recuperarla en teoría desde cualquier otra máquina de la red, salvo que nuestro ejemplo utiliza por el momento localhost, lo que impide que el cliente apunte a la máquina correcta. La siguiente sección realiza una implantación un poco más realista,

pero el objetivo era comenzar con un ejemplo sencillo para mostrar que el registro se puede utilizar de manera muy sencilla.

Para visualizar los logs de nuestro registro:

```
#docker logs registre
```

En este caso vemos que accedemos al registro local desde el propio servidor Docker donde esta alojado. **En caso de querer acceder desde un servidor externo, debemos de configurar el servicio Docker para permitir las conexiones a dicho repositorio**.

En la máquina desde la que desea acceder al registro, edite el archivo **/etc/docker/daemon.json**. Sin duda será necesario utilizar el prefijo del comando sudo. Es necesario elevar los permisos para modificar este archivo

Para ello configuraremos, o crearemos el fichero /**etc/docker/daemon.json** con el siguiente contenido:

#vi /etc/docker/daemon.json

{"insecure-registries" : ["docker.curso.esp:5000"]}

Despues de reiniciar nuestro servicio Doker:

 # systemctl restart docker

 # systemctl status docker

Despues de reiniciar nuestro servidor prodremos acceder a las imágenes desde un servidor remoto, en el servidor remoto tendremos que ralizar las siguientes modificaciones:

Servidor Remoto

 # vi /etc/docker/daemon.json

{"insecure-registries" : ["docker.curso.esp:5000"]}

#systemctl restart docker

 # docker pull docker.curso.esp:5000/intranet

Using default tag: latest

Trying to pull repository docker.curso.esp:5000/intranet ...

latest: Pulling from docker.curso.esp:5000/intranet

a3ed95caeb02: Pull complete

246fa74a6ffc: Pull complete

30cb2a8c3fb3: Pull complete

9a3d144bcbd0: Pull complete

766d08bc36d2: Pull complete

9271d91746a8: Pull complete

8ae4ab599326: Pull complete

8af48f505a44: Pull complete

5f9835a5a894: Pull complete

ed7f78e8c16a: Pull complete

cb7763e7f973: Pull complete

d732833f54c7: Pull complete

13a06c66a75f: Pull complete

ca9d3256a30a: Pull complete

21bde52c2a6e: Pull complete

f847fe003644: Pull complete

f8c1adcda761: Pull complete

c25c3914e0ea: Pull complete

Digest: sha256:16be00f0489f313f4e3315f78582dfa6a8159844a3749cc8cdb7103fd20ff755

Al habilitar la dirección IP o el nombre DNS a la lista de registros inseguros (insecure-registries), podemos descargar imágenes desde ese servidor que previamente fueron publicadas. Ademas desde el servidor que hemos configurado también podemos publicar imágenes utilizando push.

 # docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

docker.curso.esp:5000/intranet latest 27e5ade02c15 About an hour ago 646.7 MB

Ahora podemos iniciar el contenedor intranet en el nuevo servidor y tendremos que conectarnos a la intranet, creada en laboratorios anteriores:

 # docker run -dtiP --name intranet 27e5ade02c15

aa5582980c88732d8a4eff063b7c035fb9e07a7563c6aa20afb3a17674bfc61a

 # docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES

aa5582980c88 27e5ade02c15 "supervisord -n" 27 seconds ago Up 24 seconds 0.0.0.0:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp intranet

http://192.168.1.155:32769/

Eliminación de la imagen en la máquina origen

Lógicamente, una vez que la máquina Node2 ha enviado una imagen al registro central, podemos eliminar esta imagen de la caché local. Ya hemos visto el comando rmi, pero nuestro ejemplo nos permite profundizar en su comportamiento.

La siguiente captura de pantalla muestra el encadenamiento de comandos necesarios para limpiar definitivamente las imágenes:

 # docker images

alpine 3.7 791c3e2ebfcb 5 months ago 4.2MB

localhost:5000/alpine latest 791c3e2ebfcb 5 months ago 4.2MB

 # docker rmi alpine:3.7

 # docker rmi -f 791c3e2ebfcb

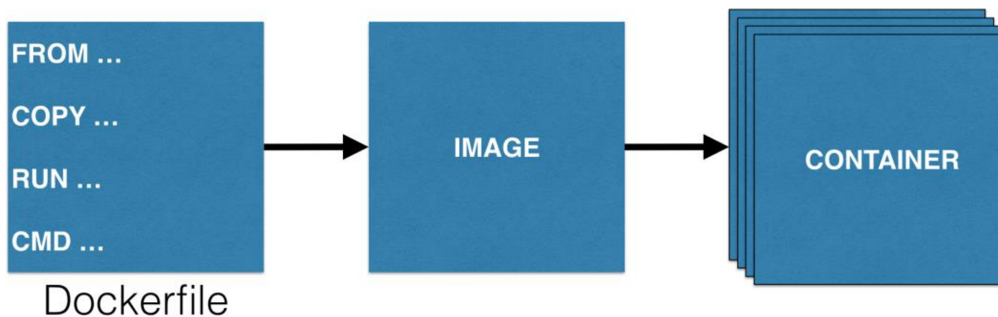
El detalle de los comandos y de los resultados es instructivo:

- La lista de las imágenes hace que aparezcan lógicamente los dos tags para el mismo identificador de imagen: se trata de la misma imagen Docker a la que simplemente hemos asociado un nombre, para situarla en el registro privado.

- Utilizando el comando rmi con el identificador en lugar del nombre, expresamos la voluntad de eliminar todos los tags, pero un mensaje explica que, como el tag está asociado a múltiples almacenes (el local y el del registro), es necesario utilizar la opción -f.
- La siguiente llamada con esta opción -f, elimina de manera forzada la imagen y todas sus etiquetas.
- Otra llamada al comando docker images permite validar que las dos etiquetas se han eliminado, así como las imágenes correspondientes en sentido de los identificadores hexadecimales.

https://docs.docker.com/registry/configuration/

Dockerfile



Docker nos ofrece la posibilidad de indicar las intruciones requeridas para crear una nueva imagen. Estas instrucciones se incluyen dentro de un fichero llamado Dockerfile.

Los pasos para crear una imagen a partir de un fichero Dockerfile son las siguientes:

- Crear un nuevo directorio que contenga un fichero Dockerfile y otros ficheros que fuesen necesarios dentro del contenedor.
- Crear el contenido de para el fichero Dockerfile.
- Ejecutar docker con la acción buil.

La sintaxis para la acción build es la siguiente:

docker buill opciones directorio

Donde las opciones mas comunes son:

- **-t nombre[:etiqueta]:** crear una imagen con el nombre y la etiqueta especificada a partir de las instrucciones de Dockerfile. Es muy recomendable utilizar esta opción.
- **--no-cache:** por defecto, Docker guarda en memoria cache las acciones realizadas recientemente. Si nosotros ejecutamos docker build varias veces, Docker comprobara si el fichero Dockerfile contiene las mismas instrucciones y en caso afirmativo, no genera una nueva imagen. Para generar siempre una nueva imagen sin hacer caso a la memoria utilizaremos esta opción.
- **--pull** por defecto, Docker solo descargara la imagen especificada en la expresión FROM si no existe. Para forzar que descargue la nueva versión de la imagen utilizaremos esta opción.
- **--quiet:** por defecto se muestra todo el proceso de creación ,los comando ejecutados y su salida. Utilizando esta opción solo mostrara el identificador de la imagen creada.

Ejemplo:

mkdir /debianvim2

cd /debianvim2

[root@docker debianvim2]# vi Dockerfile

FROM debian:latest

RUN apt-get update && apt-get install -y vim

CMD /bin/bash

Donde las expresiones tienen el siguiente significado:

- **FROM:** indica cual es la imagen base para crear la nueva.
- **RUN:** los comandos a ejecutrar dentro del contenedor con la imagen base para generar la imagen fianl.
- **CMD:** el comando por defecto a ejecutar cuando utilicemos esta imagen. Es el punto de entrada a la imagen si no especificamos otro comando con run o créate

En la acción build especificamos el (.) como el directorioactual como argumento, tabien se podría utilizar el directorio absoluto.

[root@docker debianvim2]# docker build -t debianvim2 .

Una vez finalizada todos las acciones y elimidados los contenedores intermedios, se creara la imagen defintiva con el nombre y la etiqueta espeficicados:

[root@docker debianvim2]# docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debianvim2	latest	aee276dba7e7	22 seconds ago	146 MB

Como podemos observar, con Dockerfile podemos de una forma fácil y simple generar nuestras imágenes. Tendremos las siguientes ventajas:

- Es posible reutilizar las plantillas para nuevas imágenes.
- Su sintaxis es sencilla tanto para administradores como para desarrolladores
- Utilizando un control de verisones (git,svn) es posible mantener versiones de estas plantillas.

A continuación**, vemos todas las expresiones** que podemos utilizar dentro de un fichero **Dockerfile:**

FROM

FROM indica la imagen base que va a utilizar para seguir futuras instrucciones. Buscará si la imagen se encuentra localmente, en caso de que no, la descargará de internet.

Sintaxis

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

MAINTAINER

Esta instrucción nos permite configurar datos del autor que genera la imagen.
Sintaxis

```
MAINTAINER <nombre> <Correo>
```

RUN

Esta instrucción ejecuta cualquier comando en una capa nueva encima de una imagen y hace un commit de los resultados. Esa nueva imagen intermedia es usada para el siguiente paso en el **Dockerfile**. RUN tiene 2 formatos:

- El modo shell: `/bin/sh -c`

```
RUN comando
```

- Modo ejecución:

```
RUN ["ejecutable", "parámetro1", "parámetro2"]
```

El modo ejecución nos permite correr comandos en imágenes bases que no cuenten con /bin/sh , nos permite además hacer uso de otra shell si así lo deseamos, ejemplo:

```
RUN ["/bin/bash", "-c", "echo prueba"]
```

ENV

Esta instrucción configura las variables de ambiente, estos valores estarán en los ambientes de todos los comandos que sigan en el **Dockerfile**.

Sintaxis

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

Estos valores persistirán al momento de lanzar un contenedor de la imagen creada y pueden ser usados dentro de cualquier fichero del entorno, por ejemplo un script ejecutable. Pueden ser sustituida pasando la opción **-env** en *docker run*. Ejemplo:

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

ADD

Esta instrucción copia los archivos o directorios de una ubicación especificada y los agrega al sistema de archivos del contenedor en la ruta especificada. Tiene dos formas:

Sintaxis

```
ADD <src>... <dest>
ADD ["<src>",... "<dest>"]
```

COPY

Al igual que ADD copia ficheros dentro de la imagen a crear. Es la expresión recomendada para copiar ficheros al contenedor, excepto que tengamos la necesidad de extraer ficheros comprimidos automáticamente.

COPY origen1 origen2   destino

Ejemplos:

COPY 004-miweb.conf /etc/httpd/conf.d

COPY miweb/ /var/www/html

COPY *.conf /etc/httpd/conf.d

Otra diferencia con ADD, es que la expresión COPY no permite especificar como origen un fichero alojado en un servidor web.

EXPOSE

Esta instrucción le especifica a docker que el contenedor escucha en los puertos especificados en su ejecución. EXPOSE no hace que los puertos puedan ser accedidos desde el host, para esto debemos mapear los puertos usando la opción **-p** en *docker run*.

Ejemplo:

```
EXPOSE 80 443
```

CMD y ENTRYPOINT

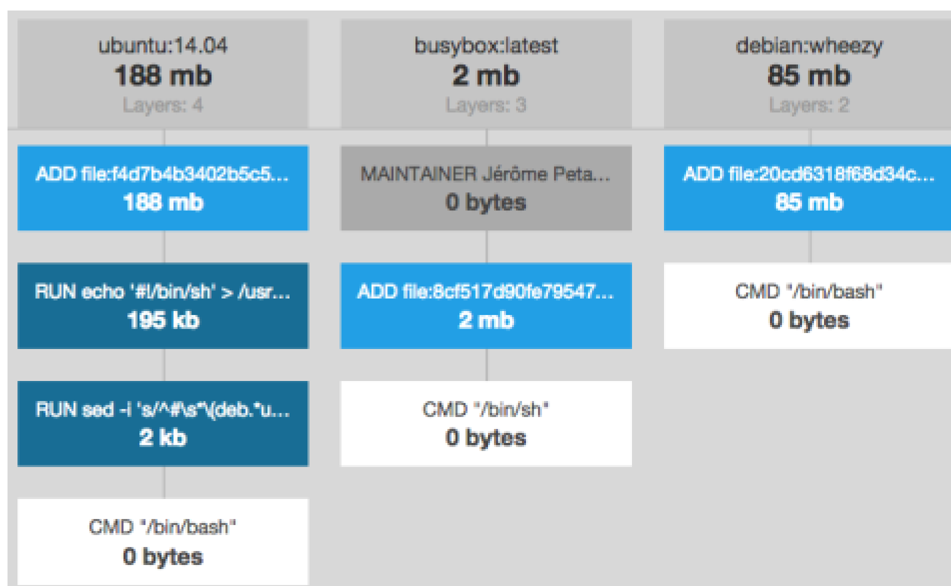
Estas dos instrucciones nos permiten especificar el comando que se va a ejecutar por defecto, sino indicamos ninguno cuando ejecutamos el `docker run`. Normalmente las imágenes bases (debian, ubuntu,…) están configuradas con estas instrucciones para ejecutar el comando */bin/sh* o */bin/bash*. Podemos comprobar el comando por defecto que se ha definido en una imagen con el siguiente comando:

```
$ docker inspect debian
...
 "Cmd": [
                "/bin/bash"
            ],
...
```

Por lo tanto no es necesario indicar el comando como argumento, cuando se inicia un contenedor:

```
$ docker run -i -t  debian
```

En el siguiente gráfico puedes ver los detalles de algunas imágenes oficiales: su tamaño, las capas que la conforman y el comando que se define por defecto:



CMD

CMD tiene tres formatos:

- Formato de ejecución:

```
CMD ["ejecutable", "parámetro1", "parámetro2"]
```

- Modo shell:

```
CMD comando parámetro1 parámetro2
```

- Formato para usar junto a la instrucción ENTRYPOINT

```
CMD ["parámetro1","parámetro2"]
```

Solo puede existir una instrucción *CMD* en un **Dockerfile**, si colocamos más de una, solo la última tendrá efecto.Se debe usar para indicar el comando por defecto que se va a ejecutar al crear el contenedor, pero permitimos que el usuario ejecute otro comando al iniciar el contenedor.

ENTRYPOINT

ENTRYPOINT tiene dos formatos:

- Formato de ejecución:

```
ENTRYPOINT ["ejecutable", "parámetro1", "parámetro2"]
```

- Modo shell:

```
ENTRYPOINT comando parámetro1 parámetro2
```

Esta instrucción también nos permite indicar el comando que se va ejecutar al iniciar el contenedor, pero en este caso el usuario no puede indicar otro comando al iniciar el contenedor. Si usamos esta instrucción no permitimos o no esperamos que el usuario ejecute otro comando que el especificado. Se puede usar junto a una instrucción *CMD*, donde se indicará los parámetro por defecto que tendrá el comando indicado en el *ENTRYPOINT*. Cualquier argumento que pasemos en la línea de comandos mediante `docker run` serán anexados después de todos los elementos especificados mediante la instrucción *ENTRYPOINT*, y anulará cualquier elemento especificado con CMD.

Ejemplo:

Si tenemos un fichero **Dockerfile**, que tiene las siguientes instrucciones:

```
ENTRYPOINT [“http”, “-v ]”
CMD [“-p”, “80”]
```

Podemos crear un contenedor a partir de la imagen generada:

- `docker run centos:centos7`: Se creará el contenedor con el servidor web escuchando en el puerto 80.
- `docker run centos:centros7 -p 8080`: Se creará el contenedor con el servidor web escuchando en el puerto 8080

VOLUME: Nos permite montar una carpeta de nuestra maquina host dentro de nuestra imagen. Se debe tener particular atención a que la carpeta sea algo que pueda existir en todas las maquinas donde se compile el Dockerfile ya que sino nos arrojara un error

Ejempo, podemos definir un volumen y realizar acciones dentro de el:

RUN mkdir /datos && > /datos/fecha.txt

VOLUME /datos

Al crear un contenedor basado en esta imagen veremos que se ha creado un volumen y todas las modificaciones hechas dentro del directorio especificado sen guardan dentro de el y es accesible a través de /var/lib/docker/volumes/.

WORKDIR: Este comando nos permitirá indicarle a Docker cual es el directorio por default donde ejecutara las acciones.

WORKDIR /var/www

USER

Por defecto, todos los comandos son ejecutados como el usuario root. A través de esta expresión podemos cambiar el comportamiento:

- USER usuario
- USER uid

SHELL

El punto de entrada para los contenedores de Linux es el comando /bin/sh -c para ejecutar los comandos especificados en CMD o los comandos especificados en línea de comandos para la acción run.

La sintaxis es la siguiente:

SHELL [“comando”, “argumento 1”, “ argumento2”]

LABEL

La expresión LABEL sirve para añadir metadatos a una imagen. Estos metadatos son muy utilies para añadir información importante como puede ser la versión de una aplicación o una descripción detallada, la sintaxis es la siguiente:

LABEL clave=valor [clave2=valor2]….

ARG

Podemos pasar argumentos a nuestro Dockerfile para distintos propósitos:

ARG nombre

ARG nombre=valorpordefecto.

ENV

La expresión ENV nos permite establecer variables de entorno dentro del contenedor que use la imagen que estamos creando. Las sintaxis posibles son las siguientes:

ENV clave valor

HEALTHCHECK

Indica el comando a comprobar si el estado del contenedor es correcto. La comprobación puede ser por ejemplo comprobar si un servidor web acepta conexiones al puerto 80. Se puede expecificar las siguientes opciones antes del comando:

- --interval=duración: indica el intervalo de comprobación, por defecto 30s.
- --timeout=duración: indica el tiempo de espera para la comprobación, por defecto también 30s.
- --retries=intentos: el numero de intentos antes de declarar un contenedor como fallido.

HEALTHCHECK –interval=2m –timeout=3s **CMD** curl -f http://127.0.0.1/ || exit 1

Ejemplo

```
FROM    centos:7

RUN yum -y  update &&  yum -y install httpd*

EXPOSE  80

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

En el siguiente laboratorio de **Dockerfile** creamos una imagen para ejecutar Apache2

```
# Colocamos la imagen base
FROM debian:latest

# Declaramos quien es el autor y el que se encagara de mantenerla
MAINTAINER usuario@correo.es

# Instalar Apache2 y configurar locales
RUN apt-get update && apt-get install -y locales locales-all apache2
RUN locale-gen es_ES.UTF-8

# Exponemos el puerto por default de Apache
EXPOSE 80
VOLUME /var/www

#Variables para apache2
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_LOG_DIR /var/log/apache

RUN mkdir -p $APACHE_RUN_DIR $APACHE_LOCK_DIR $APACHE_LOG_DIR

#Contenido Web
COPY index.html /var/www/html

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

Los comentarios dentro de un fichero Dockerfile se especifican empezando la línea con el símbolo de almohadilla # los pasos para crear la imagen y ejecutar un contenedor que tendrá un servidor web que muestre el contenido deseado se detallan a continuación:

Creamos el directorio llamado /apache2 y el archivo index.html

#mkdir /apache2

#**cd /apache2**

echo "HOLA DESDE EL CONTENEDOR" > index.html

Creamos nuestro archivo Dockerfile, con el contenido del laboratorio y lo compilamos:

 # docker build --no-cache --pull -t debianapache2 .

 # docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debianapache2	latest	dbdc8c47048d	8 minutes ago	363MB

 # docker run -dtiP --name serweb debianapache2

 # docker ps -l

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c5551b203fb6	debianapache2	"/usr/sbin/apache2 -…"	19 seconds ago	Up 17 seconds	0.0.0.0:32768->80/tcp	serweb

 # curl localhost:32768

HOLA DESDE EL CONTENEDOR

Laboratorio Generar Imágenes

En este laboratorio vamos a ver de forma práctica, como generear distintas imágenes del servidor de aplicaciones Jboss EAP, y sus distintas configuraciones.

El formador explicara las distintas configuraciones del fichero Dockerfile.

En nuestro servidor docker, tendremos el siguiente directorio con los siguientes archivos de configuración:

```
#/docker-jboss-eap-master

# ls

Dockerfile        hello.war  jboss-eap-6.4.0.zip     module.xml                           README.md

Dockerfile-Bueneo  java.sh    jdk-7u80-linux-x64.rpm  mysql-connector-java-5.1.27-bin.jar  standalone.xml
```

Dockerfile

```
# dockerfile to build image for JBoss EAP 6.4

# start from rhel 7.2

FROM centos

# file author / maintainer

MAINTAINER "FirstName LastName" "emailaddress@gmail.com"

# update OS

RUN yum -y update && \

yum -y install sudo openssh-clients telnet unzip java-1.8.0-openjdk-devel && \

yum clean all

WORKDIR /opt

RUN mkdir software

COPY jboss-eap-6.4.0.zip /opt/software

WORKDIR /opt/software

RUN unzip jboss-eap-6.4.0.zip -d /opt

### Set Environment

ENV JBOSS_HOME /opt/jboss-eap-6.4

COPY hello.war /opt/jboss-eap-6.4/standalone/deployments


### Create EAP User

RUN $JBOSS_HOME/bin/add-user.sh admin Pelicano,013 --silent
```

```
# Install mysql module
ADD module.xml /opt/jboss-eap-6.4/modules/com/mysql/main/module.xml
ADD mysql-connector-java-5.1.27-bin.jar /opt/jboss-eap-6.4/modules/com/mysql/main/mysql-connector-java-5.1.27-bin.jar
ADD standalone.xml /opt/jboss-eap-6.4/standalone/configuration/standalone.xml

# Install Java JDK
COPY jdk-7u80-linux-x64.rpm /opt/software
RUN  rpm -ivh /opt/software/jdk-7u80-linux-x64.rpm
ENV JAVA_HOME /usr/java/jdk1.7.0_80/
ENV PATH /usr/java/jdk1.7.0_80/bin:$PATH



### Configure EAP
RUN echo "JAVA_OPTS=\"\$JAVA_OPTS -Djboss.bind.address=0.0.0.0 -Djboss.bind.address.management=0.0.0.0\"" >>
$JBOSS_HOME/bin/standalone.conf
### Open Ports
EXPOSE 8080 9990 9999

WORKDIR /opt/jboss-eap-6.4/bin
RUN chmod 755 standalone.sh
ENTRYPOINT $JBOSS_HOME/bin/standalone.sh
```

Construimos la imgen, terminamos la orden con un .:

[root@docker docker-jboss-eap-master]**# docker build --rm -f Dockerfile -t jboss-eap-mysql-centos-jdk .**

Si todo ha ido correcto tendremos una nueva imagen creada:

```
# docker images
REPOSITORY                      TAG        IMAGE ID        CREATED          SIZE
jboss-eap-mysql-centos-jdk1.7   latest     8975d81e8667    58 minutes ago   1.213 GB
```

Lanzamos un nuevo contenedor con la imagen creada de Jboss:

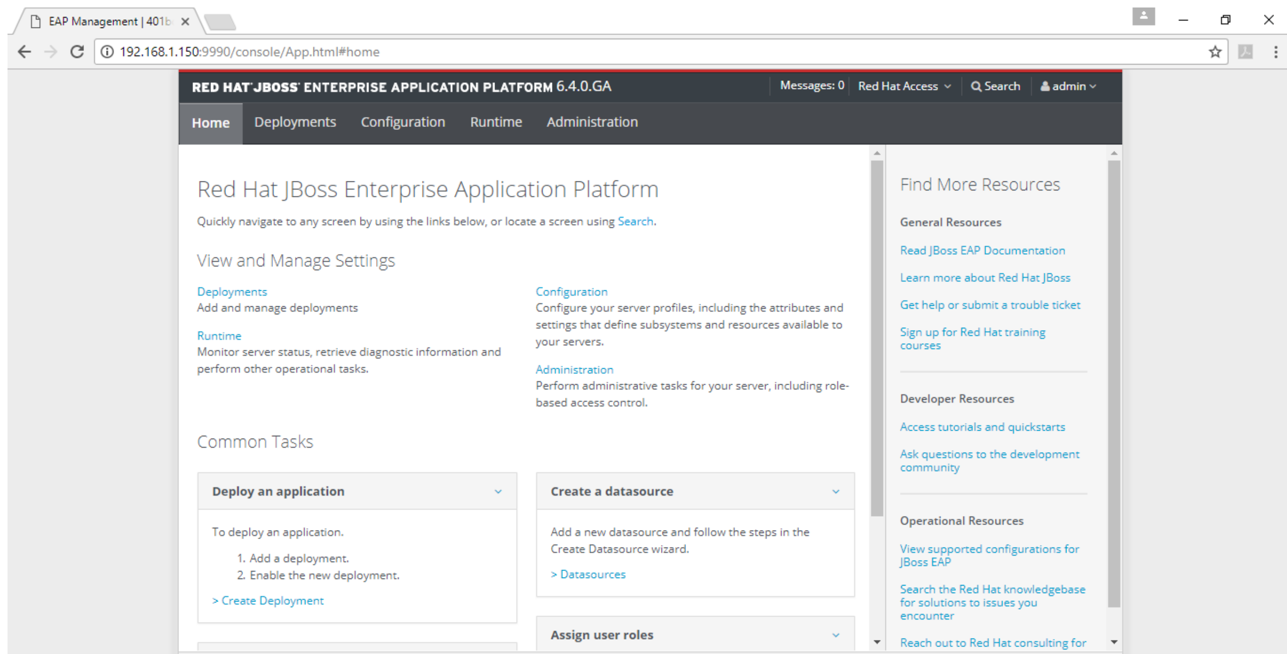
docker run -it -p 0.0.0.0:9990:9990 -p 0.0.0.0:8080:8080 --name jboss-serv1 -d jboss-eap-mysql-centos-jdk

Si queremos lanzar otro contenedor cambiamos los puertos del Docker server:

docker run -it -p 0.0.0.0:9991:9990 -p 0.0.0.0:8180:8080 -p 0.0.0.0:8109:8009 --name jboss-serv2 -d jboss-eap-mysql-centos-jdk

Ahora comprobamos que accemos a la consola de administración del servidor Jboss y al conector http:

http://192.168.1.150:9990



http://192.168.1.150:8080/hello/

Tambien podemos acceder al contenedor y comprobar la instalación y configuración de nuestro servidor Jboss EAP y las variables de entorno de java:

 # docker exec -ti jboss-serv1 /bin/bash

[root@401bc2a8708c bin]# **java -version**

java version "1.7.0_80"

Java(TM) SE Runtime Environment (build 1.7.0_80-b15)

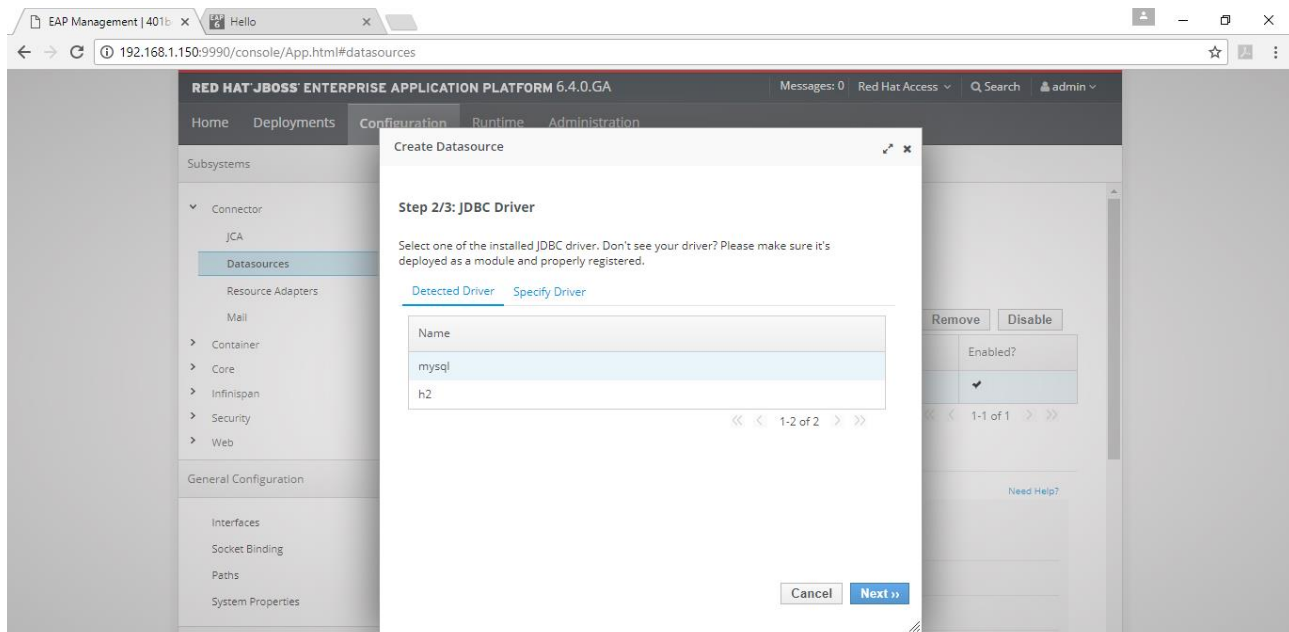
Java HotSpot(TM) 64-Bit Server VM (build 24.80-b11, mixed mode)

[root@401bc2a8708c bin]# **ls -l /opt/jboss-eap-6.4/**

total 396

-rw-rw-r-- 1 root root   419 Mar 27 2015 JBossEULA.txt

```
-rw-rw-r-- 1 root root  26530 Mar 27  2015 LICENSE.txt
drwxrwxr-x 3 root root     27 Mar 27  2015 appclient
drwxrwxr-x 4 root root   4096 Mar 27  2015 bin
drwxrwxr-x 3 root root     20 Mar 27  2015 bundles
drwxrwxr-x 5 root root     52 Mar 27  2015 docs
drwxrwxr-x 5 root root     50 Mar 27  2015 domain
-rw-rw-r-- 1 root root 363815 Mar 27  2015 jboss-modules.jar
drwxrwxr-x 4 root root     31 Jul  2 22:47 modules
drwxrwxr-x 8 root root     91 Jul  3 08:40 standalone
-rw-rw-r-- 1 root root     66 Mar 27  2015 version.txt
drwxrwxr-x 4 root root    158 Mar 27  2015 welcome-content
```



Consecuencias de la elección de las imágenes básicas

a. La imagen básica correcta

Las consecuencias del mecanismo de caché y de manera general, del modo de funcionamiento de las imágenes que se apilan las unas sobre las otras, son de varios tipos.

En primer lugar, es fundamental no multiplicar imágenes básicas. Salvo si hay una necesidad particular, se recomienda elegir una distribución y atenerse a ella lo máximo posible. Si todas las imágenes se construyen a partir de una imagen debian:wheezy, los 100 MB correspondientes se descargan una vez para todas y solo la parte superior de las imágenes fluctuará. En una ubuntu:xenial, hablamos alrededor de 125 MB, pero incluso si la imagen es más grande, lo importante sigue siendo sobre todo no multiplicar las distribuciones y las versiones. Si alguna de sus imágenes están basadas en una ubuntu:xenial, otras en una ubuntu:trusty, otras en debian:wheezy y las últimas en una centos:latest, está multiplicando la carga en el disco (lo que puede resultar molesto), así como sobre el ancho de banda (lo que generalmente es muy molesto, porque está relacionado directamente con el rendimiento).

La elección de la versión de la imagen, también es importante. Utilizar el defecto latest (la última), conduce al cabo de algún tiempo a la presencia de varias versiones efectivas en su caché de imágenes locales, si el tag latest se mueve mucho. Aquí de nuevo es más limpio validar una versión dada y ajustarse a ella. Salvo necesidad particular, cuando utilice una imagen ubuntu, es lógico utilizar la última LTS y el equipo que mantiene la imagen oficial, hace justamente el esfuerzo de no mover la etiqueta latest dentro de las versiones minores de la LTS, de manera que continúen los patches, pero sin imponer un cambio más importante de imagen a los usuarios.

Para terminar, no es necesario utilizar imágenes tan grandes en todos los casos. Las imágenes ubuntu y debian son muy conocidas, porque ofrecen distribuciones casi completas, aunque se haya hecho un esfuerzo enorme de reducción de tamaño, respecto a las versiones propuestas durante la instalación como OS principal de una máquina física y que ocupan un DVD de 4,7 GB. Pero en el 20% de los casos más sencillos, que representan el 80% de los usos, una imagen reducida como busybox puede servir perfectamente y solo pesa algunos MB.

Los editores de las distribuciones, prestan atención al tamaño de sus imágenes y el tamaño de la imagen básica Ubuntu ha bajado en los últimos años. Incluso si 125 MB parece todavía desmesurado para los usos de micro-servicios, conviene recordar que durante la primera versión de este libro (2015), la misma imagen pesaba 190 MB. La reducción también es visible en Debian. La imagen ya era más ligera con 125 MB y ahora no ocupa más de 100.

Debian va incluso más allá con sus imágenes de tipo slim, es decir, ligeras o delgadas según la traducción inglesa adoptada. Una Debian Jessie en versión slim solo ocupa 79 MB y la más reciente, Debian Wheezy en versión slim, baja hasta 47 MB. Todavía no estamos en tamaños ultra-reducidos de únicamente 4 MB, como una imagen Alpine, pero el esfuerzo ya es enorme.

Se debe reproducir el mismo esfuerzo de concisión en las imágenes básicas para los contenedores Windows, una imagen microsoft/windowsservercore que pesa varios GB, mientras que una imagen microsoft/nanoserver, que es hoy en día la alternativa más restringida posible para los contenedores Windows, todavía ocupa varios centenares de MB.

b. Su propia imagen básica

Otro medio eficaz de ganar espacio, es establecer una capa de imagen para todas las funcionalidades que utilice normalmente. Por ejemplo, un equipo de desarrollo va a estar acostumbrado a tener determinadas herramientas en las máquinas de ejecución de sus aplicaciones. Un equipo orientado a la web, necesitará Nginx NodeJS, AngularJS, etc. Si las imágenes se construyen normalmente para probar las aplicaciones desarrolladas por este equipo, una buena práctica es construir una imagen común de "desarrollo web", que servirá de base para todas estas máquinas de prueba que normalmente tienen una duración de vida muy limitada (el tiempo de las pruebas, porque otra versión llega pocas horas más tarde si el equipo trabaja de manera just-in-time).

El ahorro en términos de rendimiento es importante para trabajar en la integración continua. Pocos segundos para descargar, instalar y configurar las dependencias, representan un coste al final del año cuando decenas de imágenes de prueba se han generado cada día. La comodidad de los equipos también mejora.

Por ejemplo, imaginemos una imagen para establecer un servidor web escrito en Python, que utiliza un equipo de desarrollo para lanzar varias veces al día las pruebas en cada uno de los puestos de desarrollo. Estas pruebas normalmente se ejecutarían en un contenedor arrancado únicamente para validar que las modificaciones del código de cada desarrollador no presenten ningún problema.

Un **Dockerfile** como este, sería como sigue:

```
FROM ubuntu:trusty

RUN apt-get update \
  && apt-get install -y \
    python \
    python-pip \
    wget \
  && pip install Flask

COPY webtest.py webtest.py

EXPOSE 5000
ENTRYPOINT ["python"]
CMD ["webtest.py"]
```

Aprovechamos la presentación de este Dockerfile para recordar las **buenas prácticas**:

- Utilización de una imagen básica con una versión explícita.
- Agrupación de los comandos de instalación en una única operación RUN.
- Disposición en las líneas para facilitar su lectura.
- Añadir un espacio antes del símbolo \ de retorno de línea, también para esclarecer el texto durante la lectura.
- Respeto el orden alfabético para hacer visibles las eventuales repeticiones durante futuras adiciones.
- Utilización preferente de la operación COPY en lugar de ADD.
- Posicionamiento del comando de copia del archivo origen después del comando que gestiona las instalaciones de las herramientas, mucho más pesadas, de manera que no se invalide este nivel de caché cuando se realice una modificación del código fuente (es un aspecto esencial).
- Separación ENTRYPOINT y CMD.

Un vistazo a la estructura de la imagen utilizando el comando history, muestra que las herramientas instaladas ocupan más de 160 MB:

```
jpg@Ubuntu-VirtualBox: ~/docker/flask
jpg@Ubuntu-VirtualBox:~/docker/flask$ docker history flask
IMAGE               CREATED             CREATED BY                                      SIZE
4f5feb29b11b        11 minutes ago      /bin/sh -c #(nop) CMD ["webtest.py"]            0 B
86c16db940b9        11 minutes ago      /bin/sh -c #(nop) ENTRYPOINT ["python"]         0 B
d28bfc16e028        11 minutes ago      /bin/sh -c #(nop) EXPOSE 5000/tcp               0 B
8555f79cf592        11 minutes ago      /bin/sh -c #(nop) COPY file:afc5ff3fe91d6f018   164 B
feb0aada0cd9        17 minutes ago      /bin/sh -c apt-get update     && apt-get inst   161.3 MB
16de3ae0de29        19 minutes ago      /bin/sh -c #(nop) MAINTAINER jp.gouigoux@free   0 B
07f8e8c5e660        2 weeks ago         /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
37bea4ee0c81        2 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
a82efea989f9        2 weeks ago         /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
e9e06b06e14c        2 weeks ago         /bin/sh -c #(nop) ADD file:f4d7b4b3402b5c53f2   188.1 MB
jpg@Ubuntu-VirtualBox:~/docker/flask$
```

Claramente, la utilización de la caché es obligatoria. La idea de compartir una imagen básica que contiene estas herramientas entre todas las personas de un equipo de desarrollo, pueda evitar pérdidas de tiempo en las reconstrucciones. De esta manera, todos los desarrolladores pueden utilizar una imagen básica actualizada, sin tener que pasar por las etapas de compilación. El ahorro no es excepcionalmente elevado en este caso, porque cada máquina de desarrollo tiene su propia caché de imágenes de todas maneras, pero veremos en un siguiente capítulo un segundo ejemplo con un caso particular de compilación de aplicación, que hace el montaje con una imagen básica "desarrollo", particularmente útil.

Sin anticipar demasiado sobre este ejemplo, se trata de un entorno .NET donde NuGet recupera las dependencias y se descargan de manera dinámica en función del código fuente. Como la carga del código fuente en la imagen se hace antes de la recuperación de las dependencias por NuGet, esta última etapa de la construcción siempre empieza desde cero después de cada modificación de código, mientras que en una gran mayoría de los casos, se trata de las mismas librerías. El hecho de preparar una imagen básica con el código "estándar" (resultante de la descarga de todas las librerías más comunes), permite ganar mucho tiempo en el resto de compilaciones de imágenes posteriores.

Best practices for writing Dockerfiles

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Laboratorio 7 Redes

En este laboratorio trabajaremos con las redes internas dentro de Docker Engine. Veremos las redes creadas automáticamente por el instalador, como crear nuestras propias redes y como conectar contenedores existentes a diferentes redes.

Redes Predefinidas

Durante la instalación de Docker Engine se crean tres redes, estas son;

- **bridge:** red por defecto. En Linux durante la instalación se crea una nueva interfaz de red virtual llamada docker0. Cuando ejecutamos un contenedor, por defecto, utiliza esta red a no ser que especifiquemos lo contrario.

  # ifconfig docker0

```
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:ddff:fe65:cc1d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:dd:65:cc:1d  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 536 (536.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18  bytes 2254 (2.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- **none:** Utilizando esta red, el contenedor no tendrá asociada ninguna interfaz de red, solo la de loopback (lo).
- **host:** utilizando esta red, el contenedor tendrá la misma configuración que el servidor Docker Engine donde se este ejecutando.

A excepción de la red bridge, no es necesario interaccionar con las otras dos. No es posible eliminar las redes predefinidas creadas por el instalador ya que son necesarias para el servicio. Es por ello que veremos como crear nuestras propias redes, donde tendremos mas flexibilidad y será posible eliminarlas si fuera posible.

Listar redes

A través de la acción network del cliente Docker, podremos listar, crear, modificar o borrar redes. Ademas, podremos conectar un contenedor a una red existente, para listar las redes externas utilizaremos:

#docker network ls [opciones]

 # docker network list [opciones]

docker network ls

NETWORK ID	NAME	DRIVER	SCOPE
3fb2d3b156fb	bridge	bridge	local
1aa8d6b12369	host	host	local
d6e5abc27f84	none	null	local

-f/--filter filtro: filtra la salida a través del filtro especifico

-q: muestra solo los identificadores.

Crear redes

Cuando trabajamos con distintos entornos, como puede ser desarrollo, test, producción, vamos a necesitar separar los contenedores en diferentes redes. Docker engine nos permite crear redes que permiten aislar las comunicaciones entre ellos.

La sintaxis es la siguiente:

#docker network create [opciones] nombre

`-attachable`	`false`	Enable manual container attachment
`--aux-address`	`map[]`	Auxiliary IPv4 or IPv6 addresses used by Network driver
`--driver, -d`	`bridge`	Driver to manage the Network
`--gateway`		IPv4 or IPv6 Gateway for the master subnet
`--internal`	`false`	Restrict external access to the network
`--ip-range`		Allocate container ip from a sub-range
`--ipam-driver`	`default`	IP Address Management Driver
`--ipam-opt`	`map[]`	Set IPAM driver specific options
`--ipv6`	`false`	Enable IPv6 networking
`--label`		Set metadata on a network
`--opt, -o`	`map[]`	Set driver specific options
`--subnet`		Subnet in CIDR format that represents a network segment

Crear red con Rango Autogenerado

Si la opción **–subnet** no se especifica, Docker Engine buscara un rango libre y lo asignara a esta red. **La red por defecto bridge es 172.17.0.1/16.**

La primera red personalizada creada (sin especificar –subnet) será con la subnet 172.18.0.1/16 y la siguiente 172.19.0.1/16 y asi sucesivamente.

En el servidor se creará una intefaz virtual bridge con el formato br-identificador

 # docker network create desarrollo

e342b97fe80943490a99828c61e8b9b80bb6a391e882f0b34d095c9c28a4fc97

 # ip a show br-e342b97fe809

8: br-e342b97fe809: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN

link/ether 02:42:4c:03:6a:d7 brd ff:ff:ff:ff:ff:ff

inet 172.18.0.1/16 scope global br-e342b97fe809

valid_lft forever preferred_lft forever

Crear red con Rango Especificado

A través de la acción network créate podemos especificar diferente valores para personalizar nuestra red. En este laboratorio crearemos una red con un rango especifico, limitando las direcciones que se pueden utilizar a coun una ip especifica como puerta de enlace.

 # docker network create --subnet 192.168.100.1/24 --ip-range 192.168.100.100/30 --gateway 192.168.100.100 produccion

2a7b1ed7c6e2b409a7d96d09db30667efc42171016096286bfbfb48abe9703b9

 # ip a show br-2a7b1ed7c6e2

9: br-2a7b1ed7c6e2: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN

link/ether 02:42:5e:5c:33:08 brd ff:ff:ff:ff:ff:ff

inet 192.168.100.100/24 scope global br-2a7b1ed7c6e2

valid_lft forever preferred_lft forever

Crear red sin Acceso al Exterior

En algunas situaciones, por motivos de seguridad, deseamos que nuestros contenedores no tengan acceso al exterior, específicamente internet. Para ello especificamos la opción –internal.

 # docker network create --internal interna

Especificar red al crear un contenedor

Al crear un contenedor, ya sea a través de la acción run o con créate, podemos especificar la red donde va a formar parte el contenedor. Para ello utilizaremos la opción –network.

 # docker run --network desarrollo -ti httpd ip a show eth0

```
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link tentative
       valid_lft forever preferred_lft forever
```

 # docker run --network produccion -ti copaiapache ip a show eth0

```
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:64:65 brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.101/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:c0ff:fea8:6465/64 scope link tentative
       valid_lft forever preferred_lft forever
```

Podemos comprobar que en la red creada **como internal** no podemos acceder al exterior:

docker run --network **desarrollo** -ti httpd ping -c1 8.8.8.8

PING 8.8.8.8 (8.8.8.8): 56 data bytes

64 bytes from 8.8.8.8: icmp_seq=0 ttl=55 time=21.854 ms

--- 8.8.8.8 ping statistics ---

1 packets transmitted, 1 packets received, 0% packet loss

round-trip min/avg/max/stddev = 21.854/21.854/21.854/0.000 ms

 # docker run --network interna -ti httpd ping -c1 8.8.8.8

PING 8.8.8.8 (8.8.8.8): 56 data bytes

--- 8.8.8.8 ping statistics ---

1 packets transmitted, 0 packets received, 100% packet los

Inspecionar la Red

Al iguan que con los contenedores con la acción inspect es posibles inspeccionar una red utilizando en este caso la acción network inspect

La sisntaxis es la siguiente:

#docker network inspect [-f formato] nombre o identificador

 # docker network ls

NETWORK ID   NAME   DRIVER   SCOPE

3fb2d3b156fb   bridge   bridge   local

e342b97fe809   desarrollo   bridge   local

1aa8d6b12369   host   host   local

b1f8f3717a46   interna   bridge   local

d6e5abc27f84   none   null   local

2a7b1ed7c6e2   produccion   bridge   local

 # docker network inspect interna


```
[
    {
        "Name": "interna",
        "Id": "b1f8f3717a46755d4784b019df5e14ab5a3d0994c654f3adca5d33614c80fbd9",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1/16"
                }
            ]
        },
        "Internal": true,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```


 # docker network inspect produccion


```
[
    {
        "Name": "produccion",
        "Id": "2a7b1ed7c6e2b409a7d96d09db30667efc42171016096286bfbfb48abe9703b9",
        "Scope": "local",
```

```
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "192.168.100.1/24",
                    "IPRange": "192.168.100.100/30",
                    "Gateway": "192.168.100.100"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```


Podremos observar, si un contenedor o varios contenedores están en ejecuccion usando la red que inspecionamos, mostrara, mostrara por cada uno de ellos:

- Su nombre
- Su identificadores
- Su dirección MAC
- Su dirección IP

Si queremos ver las redes a las que el contenedor esta conectado, utilizaremos el comando (**registry es el contenedor):**

 # docker inspect --format="{{.NetworkSettings.Networks}}" "registry"

map[bridge:0xc4200bc6c0]

 # docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' registry

172.17.0.2

 # docker inspect --format='{{json .NetworkSettings.Networks}}' registry

{"bridge":{"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":"3fb2d3b156fba57e39359b6ed9ba5018dd62a24f3b30f70a60b658e7b43d2875","EndpointID":"f1b193c3f715747c96eced8528493aab2c85884300e9dc1f5d94c3e7508c9817","Gateway":"172.17.0.1","IPAddress":"172.17.0.2","IPPrefixLen":16,"IPv6Gateway":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,"MacAddress":"02:42:ac:11:00:02"}}

Conectar y Desconectar contenedor A/De una red

Hasta haroa hemos visto que en la fase de creación de un contenedor es posible elegir la red en la que queremos incluirlo. Una vez creado, Docker Engine nos permite conectarlo a otra red, creando una interfaz de red virtual dentro del contenedor utilizando la red especificada. Para ello utilizaremos la siguiente sintaxis:

#docker network connect [opciones] red contenedor

Donde red es el nombre o identificador de una red ya exixtentes y contenedor es el nombre o el identificador de un contenedor en ejecuccion. No es posible conectar a un contenedor detenido a una red.

En el caso de no especificar ninguna opción, obtendrá una dirección IP libre dentro de las red especificada. Las opciones posibles para network connect son las siguientes:

--ip dirección: especifica una dirección ip estatica.

--ip6 direccion: especifica una dirección ipv6 estatica

En nuestro laboratorio lanzaremos un contenedor dentro de una red previamente creada y una vez en ejecución lo conectamos a otra red.

Ejecutamos el contenedor solo a una red:

docker run --name centos6 --network desarrollo -dti httpd

a4b127221b099065ae4946454727b003f65f2383e647d92fb202b89a23c21443

Listar las redes del contenedor previamente creado:

 # docker inspect --format="{{.NetworkSettings.Networks}}" "centos6"

map[desarrollo:0xc4200bc6c0]

Conectar el contenedor a una nueva red:

 # docker network connect interna centos6

Listar los interfaces e IPs dentro del contenedor para comprobar ambas redes:

 # docker exec -ti centos6 ip a

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
21: eth0@if22: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
       valid_lft forever preferred_lft forever
23: eth1@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:13:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.19.0.2/16 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe13:2/64 scope link
       valid_lft forever preferred_lft forever
```

En caso de querer desconectar un contenedor de una red, utilizaremos el siguiente comando:

#docker network disconnect [--force] red contenedor

Es posible desconectar el contenedor tanto de la red conectada utilizando network connect como de las red especificada durante la creación con run o créate.

Listar las redes atuales del contenedor (centos6)

 # docker inspect --format="{{.NetworkSettings.Networks}}" "centos6"

map[desarrollo:0xc4200bc6c0 interna:0xc4200bc780]

Desconectar el contenedor de una red (centos6)

 # docker network disconnect desarrollo centos6

Comprobamos la desconexion (centos6)

docker inspect --format="{{.NetworkSettings.Networks}}" "centos6"

map[interna:0xc4200bc6c0]

Eliminrar una Red

Una vez creada una red es posible eliminarla, el único requisito es que esa red no este utilizada por ninguno de los contenedores que actualmente están en ejecución, la sintaxis es la siguiente:

#docker network rm red

#docker network remove red

En el caso de que la red sea eliminada, el comando nos devolverá en la salida el nombre de la misma. En caso de que la red este sinedo utilizada, nos dara el siguiente error:

 # docker network remove interna

Error response from daemon: network interna has active endpoints

Use Macvlan networks

https://docs.docker.com/network/macvlan/

Algunas aplicaciones, especialmente las aplicaciones heredadas o que monitorean el tráfico de la red, esperan **estar directamente conectadas a la red física**. En este tipo de situación, puede usar el controlador de red macvlan para asignar una dirección MAC a cada interfaz de red virtual del contenedor, lo que hace que parezca una interfaz de red física directamente conectada a la red física. En este caso, debe designar una interfaz física en su host de Docker para usar para Macvlan, así como la subred y la puerta de enlace de Macvlan. Incluso puede aislar sus redes Macvlan usando diferentes interfaces de red físicas. Tenga en cuenta lo siguiente:

- Es muy fácil dañar involuntariamente su red debido al agotamiento de la dirección IP o a la "difusión de VLAN", que es una situación en la que tiene una cantidad inapropiada de direcciones MAC únicas en su red.
- Su equipo de red debe poder manejar el "modo promiscuo", donde a una interfaz física se le pueden asignar varias direcciones MAC.
- Si su aplicación puede funcionar con un puente (en un solo host Docker) o superposición (para comunicarse a través de múltiples hosts Docker), estas soluciones pueden ser mejores a largo plazo.

En este ejemplo estoy creando una red llamada lan basada en la red 192.168.1.0/24 y esta excluido en el hdcp del router de esta red, para que se pueda dar ips a partir del rango 192.168.1.192 a los contenedores, de esta forma los contenedores están pinchados a la red local de la empresa:

```
[root@docker ~]#docker network create -d macvlan -o parent=ens33 \
  --subnet 192.168.1.0/24 \
  --gateway 192.168.1.1 \
  --ip-range 192.168.1.192/27 \
  lan

[root@docker ~]# docker network ls

[root@docker ~]# docker network inspect lan

 [root@docker ~]# docker run -dtiP  --net=lan --name conte1 debian

[root@docker ~]#  docker run -dtiP  --net=lan --name conte2 nginx

[root@docker ~]# docker inspect conte1
```

Si ahora entramos en nuestro contenedor, veremos que no tenemos comunicación con nuestro Docker engine:

[root@docker ~]# docker exec -ti conte1 /bin/bash

root@535178e9920f:/# ping 192.168.1.150

PING 192.168.1.150 (192.168.1.150) 56(84) bytes of data.

From 192.168.1.192 icmp_seq=1 Destination Host Unreachable

From 192.168.1.192 icmp_seq=2 Destination Host Unreachable

From 192.168.1.192 icmp_seq=3 Destination Host Unreachable

Cuando estamos utilizando el driver macvlan no concta los contenedores con el host realizaremos en nuestro, este tráfico se filtra explícitamente por los módulos del kernel para ofrecer aislamiento y seguridad adicionales al proveedor.

En nuestro docker engine 192.168.1.150, craremos un bridge y lo enlazamos para que nuestros contenedores puedan ver los servicios que esta ofreciendo nuestro Docker Engine, como puede ser traefick, el enlace físico en nuestro Docker engine es ens33, que es el intefaz a nuestra red local:

[root@docker ~]#ip link add mac0 link ens33 type macvlan mode bridge

[root@docker ~]#ip addr add 192.168.1.148/24 dev mac0

[root@docker ~]#ifconfig mac0 up

Si ahora volvemos a ejecutar ping dentro de un contenedor que este en la red lan, veremos como ya tenemos comunicación con nuestro Docker engine:

[root@docker ~]# docker exec -ti conte1 /bin/bash

root@535178e9920f:/# ping 192.168.1.150

PING 192.168.1.150 (192.168.1.150) 56(84) bytes of data.

64 bytes from 192.168.1.150: icmp_seq=1 ttl=64 time=2.94 ms

64 bytes from 192.168.1.150: icmp_seq=2 ttl=64 time=0.485 ms

64 bytes from 192.168.1.150: icmp_seq=3 ttl=64 time=0.063 ms

Ahora como ejemplo, podemos iniciar un contenedor de jboss sobre la imagen jboss-eap-mysql-centos-jdk, y realizar un despliegue sobre una base de datos que esta en la red en un servidor (192.168.1.201), esta demostración la realizara el formador:

[root@docker ~]# docker run -dti --rm --net=lan --name jboss1 --ip=192.168.1.194 jboss-eap-mysql-centos-jdk

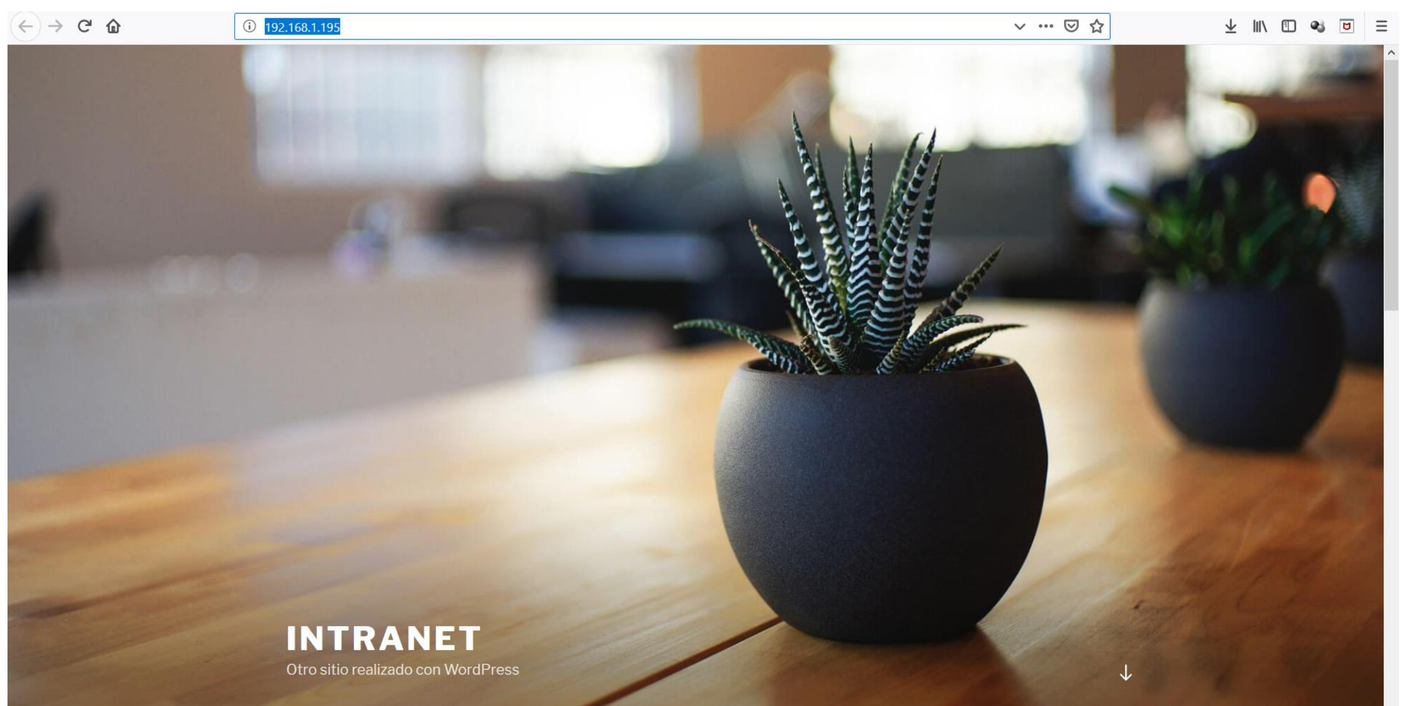
```
tarta@LAPTOP-BOSROO2V C:\Users\tarta
> ping 192.168.1.194

Haciendo ping a 192.168.1.194 con 32 bytes de datos:
Respuesta desde 192.168.1.2: Host de destino inaccesible.
Respuesta desde 192.168.1.194: bytes=32 tiempo=108ms TTL=64
Respuesta desde 192.168.1.194: bytes=32 tiempo=11ms TTL=64
Respuesta desde 192.168.1.194: bytes=32 tiempo=5ms TTL=64

Estadísticas de ping para 192.168.1.194:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    Mínimo = 5ms, Máximo = 108ms, Media = 41ms
```

http://192.168.1.194:9990/console/App.html#home

También podríamos realizarlo con un contenedor de Wordpress, contra un servidor de msyql de nuestra red (192.168.1.5):

[root@docker ~]# docker run -dti --rm --net=lan --name servidor_wp --ip=192.168.1.195 -e WORDPRESS_DB_HOST=192.168.1.5 -e WORDPRESS_DB_USER=root -e WORDPRESS_DB_PASSWORD=000000 -e WORDPRESS_DB_NAME=wordpress wordpress

http://192.168.1.195/



Laboratorio Redes docker

Enlazando contenedores docker

Para trabajar con el paradigma de "microservicio" donde cada contenedor ofrezca un servicio que funcione de forma autónoma y aislada del resto, pero que tenga cierta relación con otro contenedor (que ofrezca también un sólo servicio) para que entre todos ofrezcan una infraestructura más o menos compleja. En este laboratorio veremos como podemos aislar servicios en distintos contenedores y enlazarlos para que trabajen de forma conjunta.

Instalación de wordpress en docker

Vamos a crear un contenedor con un servidor web con wordpress instalado que lo vamos a enlazar con otro contenedor con un servidor de base de datos mysql. Para realizar el laboratorio vamos a utilizar las imágenes oficiales de wordpress y mysql que encontramos en docker hub.

```
$ docker images
REPOSITORY        TAG            IMAGE ID          CREATED          SIZE
wordpress         latest         55f2580b9cc9      5 days ago       516.5 MB
mysql             latest         e13b20a4f248      5 days ago       361.2 MB
```

Docker nos permite un mecanismo de enlace entre contenedores, posibilitando enviar información de forma segura entre ellos y pudiendo compartir información entre ellos, por ejemplo las variables de entorno. Para establecer la asociación entre contenedores es necesario usar el nombre con el que creamos el contenedor, el nombre sirve como punto de referencia para enlazarlo con otros contenedores.

Por lo tanto, lo primero que vamos a hacer es **crear un contenedor desde la imagen mysql** con el nombre ***servidor_mysql***, siguiendo las instrucción del repositorio de docker hub:

```
#docker run -dti  --name servidor_mysql -e MYSQL_ROOT_PASSWORD=Pelican0  mysql
```

En este caso sólo hemos indicado la variable de entrono *MYSQL_ROOT_PASSWORD*, que es obligatoria, indicando la contraseña del usuario root. Si seguimos las instrucciones del repositorio de docker hub podemos observar que podríamos haber creado más variables, por ejemplo: *MYSQL_DATABASE, MYSQL_USER, MYSQL_PASSWORD, MYSQL_ALLOW_EMPTY_PASSWORD*.

A continuación, vamos a crear un nuevo contenedor, con el nombre ***servidor_wp***, con el servidor web a partir de la imagen wordpress, enlazado con el contenedor anterior.

#docker run -dti --name servidor_wp -p 80:80 --link servidor_mysql:mysql wordpress

Para realizar la asociación entre contenedores hemos utilizado el parámetro **`--link`**, donde se indica el nombre del contenedor enlazado y un alias por el que nos podemos referir a él.

Podemos comprobar los contenedores con los que está asociado un determinado contenedor con la siguiente instrucción:

$ **docker inspect -f "{{ .HostConfig.Links }}" servidor_wp**

En esta situación el contenedor ***servidor_web*** puede acceder a información del contenedor ***servidor_mysql***, para hacer esto docker construye un túnel seguro entre los contenedores y no es necesario exponer ningún puerto entre ellos (cuando hemos creado el contenedor ***servidor_mysql*** no hemos utilizado el parámetro *-p*), por lo tanto al ***servidor_mysql*** no se expone al exterior. Para facilitar la comunicación entre contenedores, docker utiliza las variables de entrono y modifica el fichero */etc/hosts*.

Variables de entorno en contenedores asociados

Por cada asociación de contenedores, docker crea una serie de variables de entorno, en este caso, en el contenedor ***servidor_wp***, se crearán las siguientes variables, donde se utiliza el nombre del alias indicada en el parámetro `--link`:

- **MYSQL_NAME:** Con el nombre del contenedor *servidor_mysql.*
- **MYSQL_PORT_3306_TCP_ADDR:** Por cada puerto que expone la imagen desde la que hemos creado el contenedor se crea una variable de entorno de este tipo. El contenido de esta variable es la dirección IP del contenedor.
- **MYSQL_PORT_3306_TCP_PORT:** De la misma manera se crea una por cada puerto expuesto por la imagen, en este caso guardamos el puerto expuesto.
- **MYSQL_PORT_3306_TCP_PROTOCOL:** Una vez más se crean tantas variables como puertos hayamos expuesto. En esta variable se guarda el protocolo del puerto.
- **MYSQL_PORT:** En esta variable se guarda la url del contenedor, con la ip del mismo y el puerto más bajo expuesto. Por ejemplo *MYSQL_PORT=tcp://172.17.0.82:3306.*

Finalmente por cada variable de entorno definido en el contenedor enlazado, en este caso *servidor_mysql*, se crea una en el contenedor principal, en este caso *servidor_web.* Si en el contenedor_mysql hay una variable MYSQL_ROOT_PASSWORD, en el servidor web se creará la variable MYSQL_ENV_MYSQL_ROOT_PASSWORD.

Para ver las variables de entorno de los contenedores asociados:

 # docker exec -ti servidor_wp env

 # docker exec -ti servidor_wp 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

172.17.26.2 mysql bd52c7372e5a servidor_mysql

172.17.26.3 bf3af17a9402

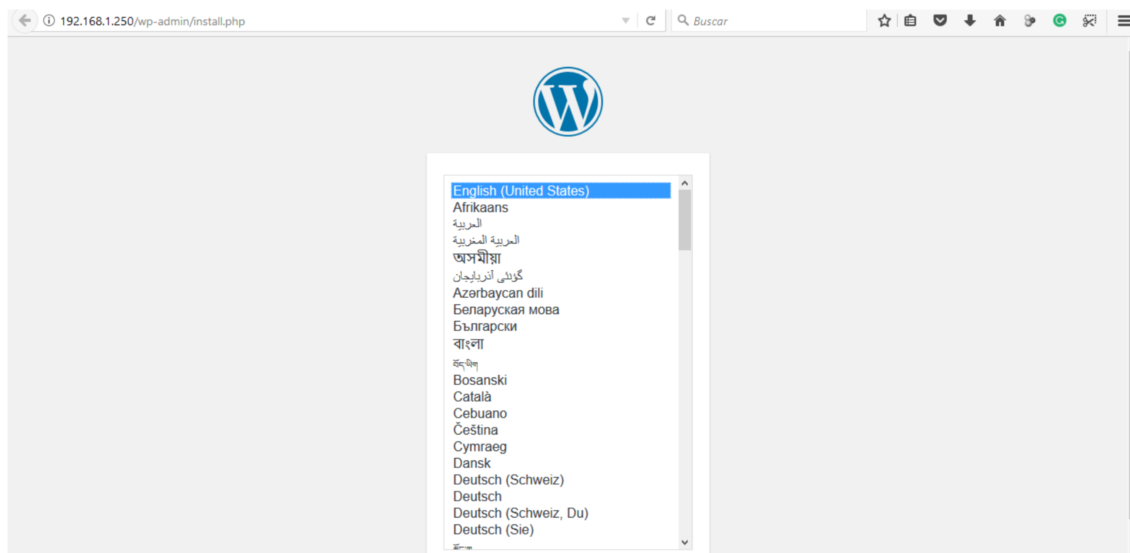
Por tanto llegamos a la conclusión que toda la información que necesitamos para instalar wordpress (dirección y puerto del servidor de base de datos, contraseña del usuario de la base de datos,…) lo tenemos a nuestra disposición en variables de entorno. El script bash que ejecutamos por defecto al crear el contenedor desde la imagen wordpress utilizará toda esta información, que tiene en variables de entorno, para crear el fichero de configuración de wordpress: *wp-config.php*. Además podremos crear nuevas variables a la hora de crear el contenedor como nos informa en la documentación del repositorio de docker hub:

- **-e WORDPRESS_DB_HOST=...** (defaults to the IP and port of the linked mysql container)
- **-e WORDPRESS_DB_USER=...** (defaults to “root”)
- **-e WORDPRESS_DB_PASSWORD=...** (defaults to the value of the **MYSQL_ROOT_PASSWORD** environment variable from the linked mysql container)
- **-e WORDPRESS_DB_NAME**=... (defaults to “wordpress”)
- **-e WORDPRESS_TABLE_PREFIX**=... (defaults to “”, only set this when you need to override the default table prefix in wp-config.php)
- **-e WORDPRESS_AUTH_KEY=...,** -e **WORDPRESS_SECURE_AUTH_KEY**=..., -e **WORDPRESS_LOGGED_IN_KEY**=..., -e WORDPRESS_NONCE_KEY=..., **-e WORDPRESS_AUTH_SALT**=..., -e **WORDPRESS_SECURE_AUTH_SALT**=..., -e **WORDPRESS_LOGGED_IN_SALT**=..., -e **WORDPRESS_NONCE_SALT**=... (default to unique random SHA1s).

Comprobación de la instalación de wordpress

Como hemos visto, al crear el contenedor *servidor_wp* asociado al contenedor *servidor_mysql,* el script bash que se está ejecutando en la creación es capaz de configurar la conexión a la base de datos con los datos de las variables de entorno que se han creado, además, al modificar su fichero */etc/hosts*, es capaz de conectar al contenedor utilizando el nombre del mismo. Al exponer el puerto 80 podemos acceder con un navegador web y comprobar que el wordpress está instalado, solo es necesario la configuración del mismo, aunque no será necesario realizar la configuración de la conexión a la base de datos:

http://192.168.1.250



Hola

Por favor, facilita un nombre de usuario válido.

Título del sitio: Intranet

Nombre de usuario: admin
Los nombres de usuario pueden tener únicamente caracteres alfanuméricos, espacios, guiones bajos, guiones medios, puntos y el símbolo @.

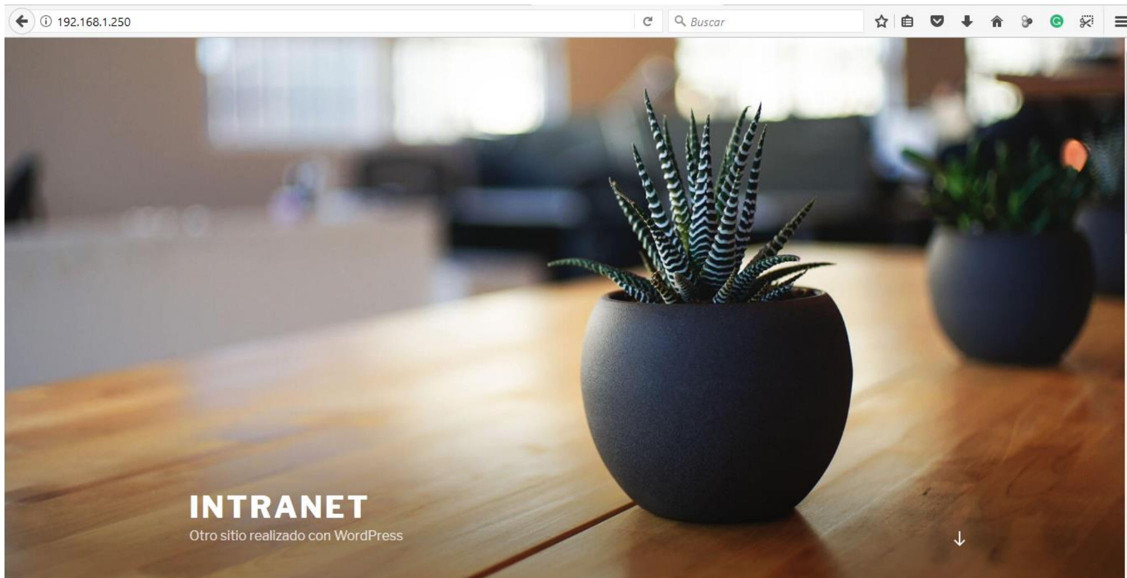
Contraseña: Pelican0 — Ocultar
Muy débil
Importante: Necesitas esta contraseña para acceder. Por favor, guárdala en un lugar seguro.

Confirma la contraseña: ☑ Confirma el uso de una contraseña débil.

Tu correo electrónico: a@curso.local
Comprueba bien tu dirección de correo electrónico antes de continuar.

Visibilidad en los motores de búsqueda: ☐ Disuade a los motores de búsqueda de indexar este sitio
Depende de los motores de búsqueda atender esta petición o no.

Instalar WordPress



Laboratorio enlazar contenedor contra un servicio externo

En este laboratorio veremos como conectar un contedor basado en la imagen de wordpress, contra un servidor de mariadb, que esta corriendo en una maquina virutal 192.168.1.5.

Para eso utilizaremos el parámetro –add-host del comando docker run, que lo que hace es añadir al /etc/hosts del contenedor el mapeo de la maquina con su ip en este caso mysqldb:

```
# docker run -dti --rm  --name  servidor_wp -p 85:80  --add-host=mysqldb:192.168.1.5 -
e WORDPRESS_DB_HOST=mysqldb -e WORDPRESS_DB_USER=root -e WORDPRESS_DB_PASSWORD=000000
-e WORDPRESS_DB_NAME=wordpress  wordpress
```

Podemos comprobar el contenido del archivo /etc/hosts del contenedor:

```
# docker exec -ti servidor_wp cat /etc/hosts
```

Para un servicio en Docker Swarm, utilizaremos el parámetro --host

```
#docker service create --name wp -p 85:80  --host=mysqldb:192.168.1.5 -e
WORDPRESS_DB_HOST=mysqldb -e WORDPRESS_DB_USER=root -e WORDPRESS_DB_PASSWORD=000000 -e
WORDPRESS_DB_NAME=wordpress wordpress
```

Laboratorio 8 Almacenamiento

Los conetenedores y las imágenes se almacenan en disco directamente. Docker pose una arquitectura de almacenamiento flexible que permite elegir entre diversos drivers dependiendo del sistema operativo y las necesidades del entorno. Cada driver de almacenamiento Docker se basa en el sistema de ficheros Linux o un administrador de volúmenes como (LVM).

En este laboratorio veremos las ventajas y desventajas de cada una de las opciones que nos proporciona Docker y veremos como configurar el servcion para utilizarlo. Es importante saber que cada driver administra los datos de forma particular y los contenedores e imágenes no son compatibles entre ellos. Es decir, si creamos imágenes y contenedores con un driver y decidimos utilizar otro, no serán accesibles en este ultimo. Si deseamos utilizar los contenedores e imágenes de un drivers a otro deberemos de hacer una copia de seguridad en el antiguo y restaurarlo en el nuevo.

Linux distribution	Supported storage drivers
Docker CE on Ubuntu	`aufs`, `devicemapper`, `overlay2` (Ubuntu 14.04.4 or later, 16.04 or later), `overlay`, `zfs`
Docker CE on Debian	`aufs`, `devicemapper`, `overlay2` (Debian Stretch), `overlay`
Docker CE on CentOS	`devicemapper`
Docker CE on Fedora	`devicemapper`, `overlay2` (Fedora 26 or later, experimental), `overlay` (experimental)

Storage driver	Supported backing filesystems
`overlay`, `overlay2`	`ext4`, `xfs`
`aufs`	`ext4`, `xfs`
`devicemapper`	`direct-lvm`
`btrfs`	`btrfs`
`Zfs`	`zfs`

Dependiendo del sistema operativo que utilizcemos y su versión de Docekr elegirá por defecto el driver mas optimo y estable, podemos comprobarlo con el comando:

[root@docker ~]# docker info

Server Version: 1.12.6

Storage Driver: devicemapper

Pool Name: docker-253:0-3485556-pool

Pool Blocksize: 65.54 kB

Base Device Size: 10.74 GB

Backing Filesystem: xfs

Data file: /dev/loop0

Metadata file: /dev/loop1

Data Space Used: 2.341 GB

Data Space Total: 107.4 GB

Data Space Available: 41.2 GB

Metadata Space Used: 5.292 MB

Metadata Space Total: 2.147 GB

Metadata Space Available: 2.142 GB

Thin Pool Minimum Free Space: 10.74 GB

Udev Sync Supported: true

Deferred Removal Enabled: false

Deferred Deletion Enabled: false

Las ventajas y desventajas de cada uno de los dirvers de almacenamiento:

AUFS: stable | production-ready | good memory use | smooth Docker experience | high write activity | PaaS-type work

Devicemapper (loop): stable | in mainline kernel | smooth Docker experience | production | performance | lab testing

Devicemapper (direct-lvm): stable | production-ready | in mainline kernel | smooth Docker experience | PaaS-type work

Btrfs: in mainline kernel | high write activity | container churn | build pools

Overlay: stable | good memory use | in mainline kernel | container churn | lab testing

ZFS native (ZoL): PaaS-type work

ZFS FUSE: stable | lab testing | production

Key

Has attribute: attribute

If good for use case: use case

If bad for use case: use case

Si deseamos utilizar un driver distinto en nuestro servidor Docker, debemos configurar el servicio a través de la opción –storage-driver=driver.

DEVIC MAPPER

Este driver fue desarrollado por Red Hat, debido a que su sistema operativo, incluia una versión del nuecleo de Linux sin soporte AUFS. Aunque la compañía contemplo incluir el soporte de AUFS, decidieron utilizar Device Mapper por su estabilidad y madurez.

Con devicemapper cada imagen y contenedor se alojan dentro de su propio dispositivo virtual, la gran diferencia de este drives es que funciona a nivel de bloque y no a nivel de fichero. Para ahorrar espacion y operaciones utiliza la técnica de snapshot para manipular imágenes y contenedores.

Al elegir devicemapper como driver tendremos dos opciones:

1. Crear un dispositivo virtual llamado loop, que será un fichero alojado dentro de :

   [root@docker ~]# ls -l **/var/lib/docker/devicemapper/devicemapper/**

   total 2274168

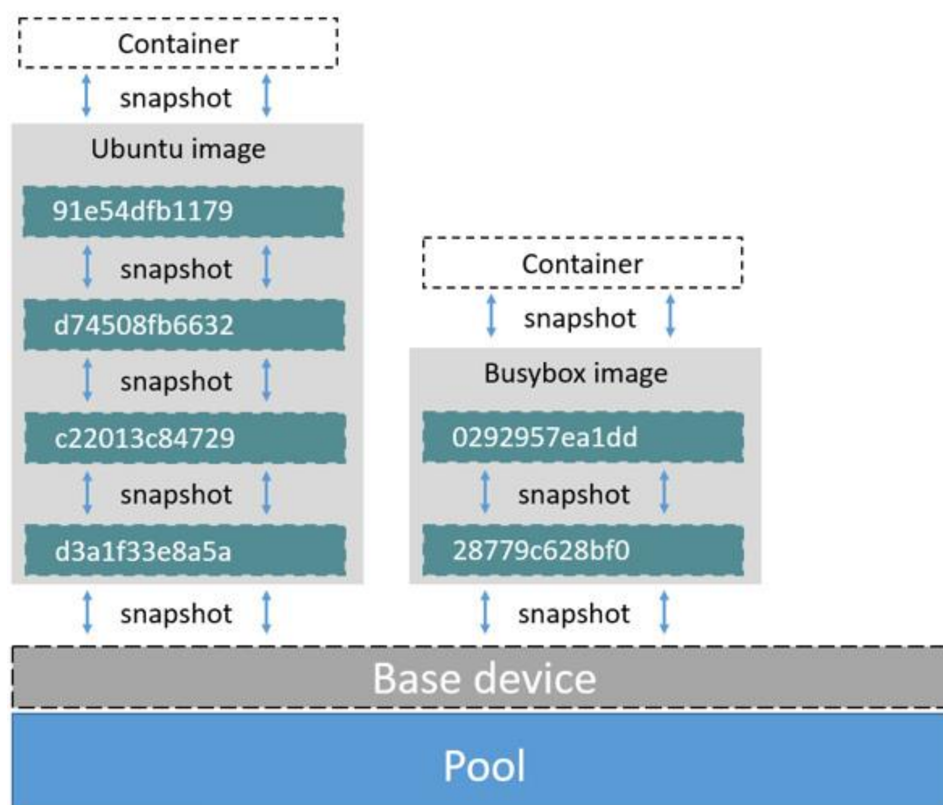
   -rw------- 1 root root 107374182400 jun 3 14:09 data

   -rw------- 1 root root 2147483648 jun 26 10:32 metadata

2. Utilizar un dispositivo físico (un disco, una particion), para ser utilizado como pool.

En este caso, el pool es un contenedor de las imágenes y contenedores, además de sus instataneas derivadas.

En el caso de los contenedores simplemente creamos una instantánea de la imagen como se muestra a continuación:



Las distribuciones que soportan devicemapper en la actualidad son las siguientes;

- RHEL/Centos/Fedora
- Ubuntu 12.04
- Ubuntu 14.04
- Debian
- Arch Linux

En el caso de utilizar este driver, la información mostrada por el comando docker info será la siguiente:

 # docker info

Storage Driver: devicemapper

Pool Name: docker-253:0-3485556-pool

Pool Blocksize: 65.54 kB

Base Device Size: 10.74 GB

Backing Filesystem: xfs

Data file: /dev/loop0

Metadata file: /dev/loop1

Como podemos observar, por defecto se utiliza lo denominado loop que es un fichero alojado dentro del disco y tratado como un dispositivo para ser usado como LVM. Los ficheros son los siguientes:

- Contine el contenido de imágenes y contenedores:

/var/lib/docker/devicemapper/devicemapper/data

- Contien la información de metadatos necesarios para el funcionamiento:

/var/lib/docker/devicemapper/devicemapper/metadata

Al utilizar por defecto el modo loop, Docker nos indicara con un aviso que no es recomendado para el uso en produccio. La recomendación es utilizar un dispositivo serparado para el uso de devicemapper. Antes de cambiar del modo loop al modo direct-lvm (utilizando dispositivos), es necesario realizar una copia de seguirdad de las imágenes y contenedores ya que el cambio de diver conlleva la perdida de todos los datos.

Conmenzamos con el labortatorio:

- Realizamos una copia de seguridad de nuestras imágenes:

#docker save -o centos-lamp.tar docker.io/nickistre/centos-lamp

- Detenemos docker

#systemctl stop docker

- Detectamos el nuevo disco:

#echo - - - > /sys/class/scsi_host/host0/scan

#cat /proc/partitions

8   16   20971520 sdb

- Eliminamos los datos antiguos, recordar que tendríamos que hacer una copia de seguridad antes:

  # rm -rf /var/lib/docker

- Configurar el fichero, para la opción DEVS para apuntar a nuestro dispositivo físico (disco, o particion) y el nombre del grupo de volumenes:

vi /etc/sysconfig/docker-storage-setup

DEVS="/dev/sdb"

VG="curso-docker-vg"

- Ejecutamos el comando:

 # docker-storage-setup

Comprobando que nadie esté utilizando este disco en este momento...

Correcto

Disco /dev/sdb: 2610 cilindros, 255 cabezas, 63 sectores/pista

sfdisk: /dev/sdb: tipo de tabla de particiones no reconocido

Situación anterior:

```
sfdisk: No se ha encontrado ninguna partición


Situación nueva:

Units: sectors of 512 bytes, counting from 0


   Disp.  Inicio  Principio   Fin   Nº sect.  Id  Sistema

/dev/sdb1          2048  41943039   41940992  8e  Linux LVM

/dev/sdb2             0         -          0   0  Vacía

/dev/sdb3             0         -          0   0  Vacía

/dev/sdb4             0         -          0   0  Vacía

Atención: la partición 1 no termina en un límite de cilindro

Atención: no hay ninguna partición primaria marcada como iniciable (activa).

Esto no es problema para LILO, pero el MBR de DOS no iniciará con este disco.

La nueva tabla de particiones se ha escrito correctamente


Volviendo a leer la tabla de particiones...


Si ha creado o modificado una partición DOS, como /dev/foo7, utilice dd(1)

para poner a cero los 512 primeros bytes: dd if=/dev/zero of=/dev/foo7 bs=512 count=1

(Véase fdisk(8).)

INFO: Device node /dev/sdb1 exists.

  Physical volume "/dev/sdb1" successfully created.

  Volume group "curso-docker-vg" successfully created

  Using default stripesize 64,00 KiB.

  Rounding up size to full physical extent 24,00 MiB

  Logical volume "docker-pool" created.

  Logical volume curso-docker-vg/docker-pool changed.
```

El comando docker-storage-setup realizara las siguientes configuraciones:

- Añadir una firma al dispositivo por motivos de seguridad.
- Particionar el disco
- Crear un grupo de volúmenes (Volume Group) y crear un volumen lógico (Logical Volume), por defecto con el 40 % de espacio libre.

- Configura el fichero /etc/sysconfig/docker-storage, para utilizar el dispositivo.

Iniciamos el servicio de docker:

#systemctl start docker

Ahora podemos detectar que tenemos un nuevo grupo de volúmenes que ha sido creado con el nombre de "docker-pool"

 # docker info

Storage Driver: devicemapper

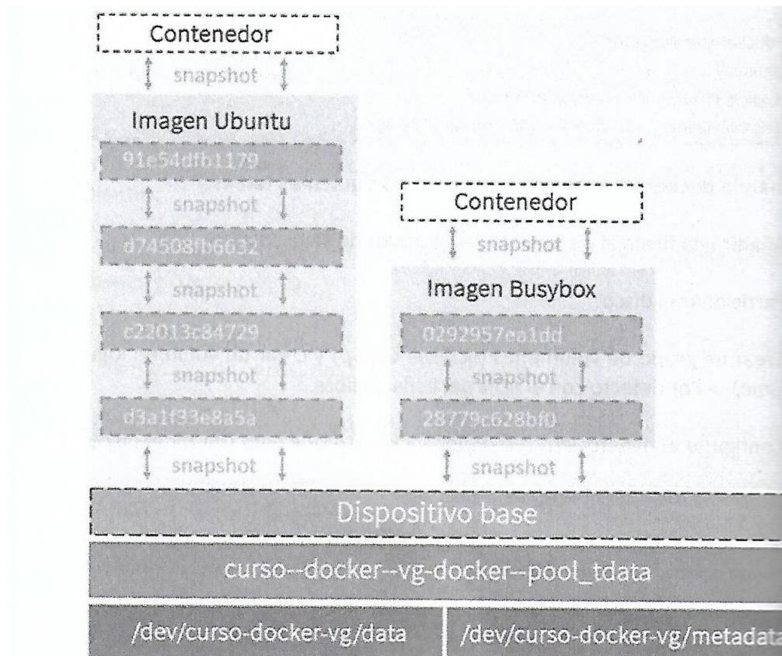
Pool Name: curso--docker--vg-docker--pool

Pool Blocksize: 524.3 kB

Base Device Size: 10.74 GB

Backing Filesystem: xfs

Al configurar nuestro dispositivo físico en devicemapper nos llevara a la siguiente imagen al trabajar con imágenes y contenedores en Docker:



Importamos las copias de seguridad de nuestras imágenes:

docker load -i centos-lamp.tar

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker.io/jdeathe/centos-ssh-apache-php	latest	c319cf0f078c	4 months ago	278.5 MB
docker.io/nickistre/centos-lamp	latest	b45f1e1c24ef	13 months ago	538.4 MB

Lanzamos un contenedor y comprobamos su correcto funcionamiento:

docker run -dtiP --name centos6 docker.io/nickistre/centos-lamp

docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

b064ceeba664 docker.io/nickistre/centos-lamp "supervisord -n" About a minute ago Up About a minute 0.0.0.0:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp centos6

Volumenes

Un volumen en Docker es un directorio especial asignado a un contenedor y que aloja los datos dentro del directorio /var/lib/docker/volumes.

Las ventajas de utilizar volúmenes son las siguientes:

- Se inicializan durante la creación del contenedor.
- Se pueden compartir entre diferentes contenedores.
- Las modificaciones realizadas inmediatamente a disco sin pasar primero por cache (con la posibilidad de que esto pueda significar perdida de datos).
- Los cambios en el volumen no se verán afectdos en el caso de que se actualice la imagen base del contenedor.

Para crear un contenedor durante las acciones run o créate, utilizaremos la opción -v (--volume) y el nombre del directorio como se muestra en el ejemplo:

docker run -dtiP --name centos6-lanp **-v /var/www/html** docker.io/nickistre/centos-lamp

Al igual que con otras configuraciones relacinadas con los contenedores, es posible utilizar la acción inspect:

docker inspect -f "{{ .Mounts }}" centos6-lanp

{0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b /var/lib/docker/volumes/0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b/_data /var/www/html local true }]

Podemos manipular los ficheros de configuración del volumen:

ls -l /var/lib/docker/volumes/0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b/_data

total 8

-rw-r--r-- 1 root root 159 jun 26 13:10 index.php

-rw-r--r-- 1 root root 22 may 18 2016 phpinfo.php

Podemos comprobar desde el contenedor que el fichero ha sido modificado:

docker exec -ti centos6-lanp /bin/bash

En el apartado **.Mounts** es posible obtener el directorio local del servidor Docker asociado para dicho volumen. En el podemos acceder a los ficheros relativos al contenedor, añadir o manipulas los existentes.

En el ejemplo anterior observamos que un volumen actua como un directorio normal dentro del servidor Docker, haciendo ideal este uso para realizar copias de seguridad, compartir datos entre contenedores o desde el propio servidor.

La opción -v (--volume) adminte otra sintaxis para compartir un directorio del servidor Docker al contenedor, la sintaxis es la siguiente:

-v/--volumen directorio del servidor:directorio contenedor.

En el siguiente ejemplo quearemos un directorio llamado /web, en el servidor de Docker, y este directorio lo hacemos accesible para los apaches que se ejecuten en los contenedores, con los queremos compartir:

[root@docker ~]# mkdir /web

Copiamos un archivo index.html a el directorio /web

Ejecutamos un contenedor utilizando un volumen con el directorio del servidor:

docker run -dtiP --name centos6-prueba-web -v /web:/var/www/html docker.io/nickistre/centos-lamp

docker run -dtiP --name centos6-prueba-web2 -v /web:/var/www/html docker.io/nickistre/centos-lamp

Comprobamos el contenido servidor por el servidor web del contenedor:

docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

3719436f90c6 docker.io/nickistre/centos-lamp "supervisord -n" 6 hours ago Up 6 hours 0.0.0.0:32779->22/tcp, 0.0.0.0:32778->80/tcp, 0.0.0.0:32777->443/tcp centos6-prueba-web2

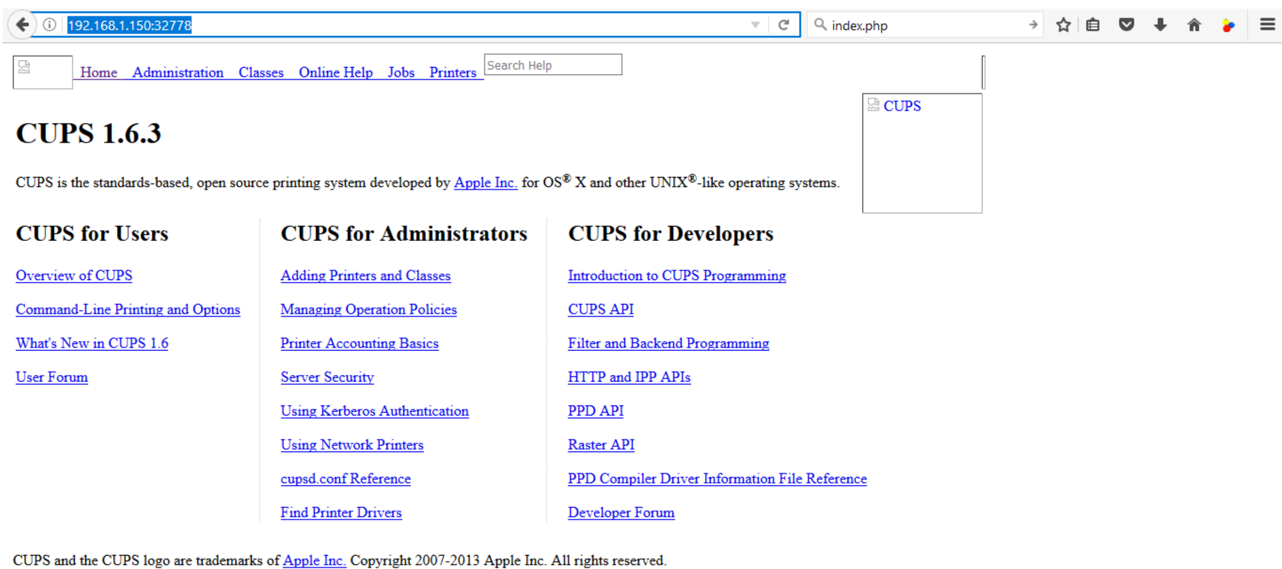
eb8d48807332 docker.io/nickistre/centos-lamp "supervisord -n" 6 hours ago Up 6 hours 0.0.0.0:32776->22/tcp, 0.0.0.0:32775->80/tcp, 0.0.0.0:32774->443/tcp centos6-prueba-web

9dc05bb1ca7a docker.io/nickistre/centos-lamp "supervisord -n" 6 hours ago Up 6 hours 0.0.0.0:32773->22/tcp, 0.0.0.0:32772->80/tcp, 0.0.0.0:32771->443/tcp centos6-lanp

b064ceeba664 docker.io/nickistre/centos-lamp "supervisord -n" 7 hours ago Up 7 hours 0.0.0.0:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp centos6

http://192.168.1.150:32775/

http://192.168.1.150:32778/



El cliente docker a través de la acción volumen nos permite realizar las siguientes tareas:

Listar los volúmenes, las opciones son las siguientes:

- -f/--filter filtro: filtra la salida a través del filtro especificado
- -q: muestra solo los identificadores

#docker volume ls

Mostrar información detallada de un volumen. La opción aceptada es la siguiente:

- -f/--format

docker volume ls

DRIVER VOLUME NAME

local 0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b

[root@docker ~]# docker volume inspect 0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b

```
[
    {
        "Name": "0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b",
        "Driver": "local",
        "Mountpoint":
"/var/lib/docker/volumes/0353ca0338ba0c02b8d6a42d940dac27dae4b909fb6d724ee8adbeb8ba46eb0b/_data",
        "Labels": null,
        "Scope": "local"
    }
]
```

Crear un volumen nuevo a utilizar:

Créate [opciones] nombre

-d/--driver diver: Especifica el driver a utilizar por defecto local

-o/--opt: opciones para driver especificado

Ejemplo crear un volumen con un nombre especifico:

docker volume create --name webapp

Creamos un fichero desde el servidor Docker dentro del volumen:

[root@docker ~]# echo "prueba de volumen" > /var/lib/docker/volumes/webapp/_data/ejemplo.txt

Arrancar un contenedor utilizando el volumen webapp y comprobar el contenido:

docker run -dtiP --name centos6-pruebacreacion2 -v webapp:/var/www/html docker.io/nickistre/centos-lamp

docker ps -a

CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES

4cf00648aa3e   docker.io/nickistre/centos-lamp   "supervisord -n"   7 seconds ago   Up 5 seconds   0.0.0.0:32785->22/tcp, 0.0.0.0:32784->80/tcp, 0.0.0.0:32783->443/tcp   centos6-pruebacreacion2

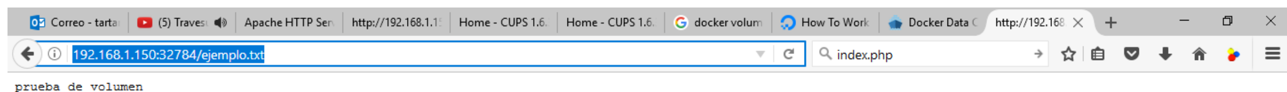
 # docker exec -ti centos6-pruebacreacion2 /bin/bash

[root@4cf00648aa3e /]# ls -l /var/www/html/

total 4

-rw-r--r-- 1 root root 18 Jun 26 18:03 ejemplo.txt

http://192.168.1.150:32784/ejemplo.txt



Laboratorios Volumenes

En estos laboratorios sobre volúmenes realizaremos todas las explicaciones anteriores utilizando la imagen docker.io/nickistre/centos-lamp.

Mounted volumes

Otras veces nos interesa montar ficheros o carpetas desde la máquina ***host***. En este caso, podemos montar la carpeta o el fichero especificando la ruta completa desde la máquina ***host*****,** y la ruta completa en el contenedor. Es posible también especificar si el volumen es de lectura y escritura (por defecto) o de solo lectura.

root@docker:~/docker$ **docker run -ti --rm -v /etc/hostname:/root/parent_name:ro -v /opt/:/data alpine:3.4 sh**

/ # cat /root/parent_name

docker

/ # ls /data/

Usage: docker volume [OPTIONS] [COMMAND]

Manage Docker volumes

Commands:

create Create a volume

inspect Return low-level information on a volume

ls List volumes

rm Remove a volumen

Docker Plugin

A partir de la versión 1.8.0 de Docker, es posible integrar pluguins externos para soportar diversos almacenamientos externos, a utilizar por los volúmenes. A través de estos almacenamientos externos podemos terner nuestros datos de forma persistente independientemente del estado del servidor de Docker. La lista de pluguins mas importantes y soportatdos se lista a continuación:

https://docs.docker.com/engine/extend/legacy_plugins/

Azure File Storage	Permite montar Microsoft Azure File Storage como volúmenes
Contiv Volume	Permite montar nodos CEPH y NFS
Convoy	Permite montar dispositivos diversos como: Device Mapper, Virtual File System/Network File Systems Amazon Elastic Bloc (EBS)
DRBD	Permite montar almacenamiento proporcionado por DRBD
Flocker	Permite montar almacenamiento proporcionado por Flocker
HPE 3Par	Permite montar almacenamiento HPE 3Par y StoreVirtual SCSI
Netapp (nDVP)	Permite montar almacenamiento proporcionado por NetApp
NetShare	Permite montar almacenamiento porocionado por NFS ¾, AWS EFS (Elastic Block Filesystem) y CIFS (Common Internet File System).
Vmware Vsphere	Permite montar almacenamiento proporcioando por vSphere.

En este laboratorio instalaremos el pluguin llamado **NetShare** para montar un directorio desde un servidor NFS:

Instalar el pluguin:

http://netshare.containx.io/docs/getting-started

https://github.com/ContainX/docker-volume-netshare

Si estamos utilizando Centos 7.x, tendremos que instalar en el servidor de NFS:

En el servidor docker (192.168.1.150)

```
#yum install nfs-utils
#mkdir /vol
#chmod 777 /vol
```

En la misma linea del fichero de configuración

Para que los contenedores puedan cambiar los permisos, deberemos de configurar:

(rw,async,no_subtree_check,no_wdelay,crossmnt,insecure,all_squash,insecure_locks,sec=sys,anonuid=0,anongid=0)

```
# vi /etc/exports

/vol
192.168.1.152(rw,async,no_subtree_check,no_wdelay,crossmnt,insecure,all_
squash,insecure_locks,sec=sys,anonuid=0,anongid=0)

/vol
192.168.1.150(rw,async,no_subtree_check,no_wdelay,crossmnt,insecure,all_
squash,insecure_locks,sec=sys,anonuid=0,anongid=0)
```

Arrancamos los servicios:

```
#systemctl enable rpcbind
#systemctl enable nfs-server
#systemctl start rpcbind
#systemctl start nfs-server
```

En la maquina docker2 (192.168.1.152)

```
#yum install nfs-utils
```

Ahora nos descargamos el binario de 64 bits, para amd

docker-volume-netshare_0.35_linux_amd64-bin

https://github.com/ContainX/docker-volume-netshare/releases

Y lo copiamos en el directorio /etc/init.d, de nuestros servidores de docker.

En los dos servidores compromamos que lo podemos ejecutar:

```
# /etc/init.d/docker-volume-netshare.bin -h
```

Ahora ejecutaremos un contenedor desde el servidor docker2 utilizando el pluguin para montar un directorio via NFS

Iniciamos el pluguin en los dos servidores:

```
[root@docker /]# /etc/init.d/docker-volume-netshare.bin nfs
[root@docker2 ~]# /etc/init.d/docker-volume-netshare.bin nfs
```

```
[root@docker /]# /etc/init.d/docker-volume-netshare.bin nfs
INFO[0000] == docker-volume-netshare :: Version: 0.35 - Built: 2018-01-27T22:43:03-08:00 ==
INFO[0000] Starting NFS Version 4 :: options: ''
```

Desde otra terminal,en el servidor docker2:

```
#docker run -dti --name contendor1 --volume-driver=nfs -v
192.168.1.150/vol:/mnt Ubuntu

# docker ps -l
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
e24fbcf8504b        ubuntu              "/bin/bash"         3 minutes ago       Up 2
minutes
```

Si entramos en el contenedor veremos el punto de montaje NFS, dentro del contenedor:

```
[root@docker2 ~]# docker exec -ti contendor1 /bin/bash
root@e24fbcf8504b:/# df -h
Filesystem             Size  Used Avail Use% Mounted on
overlay                 17G   15G  2.9G  84% /
tmpfs                   64M     0   64M   0% /dev
tmpfs                  5.1G     0  5.1G   0% /sys/fs/cgroup
192.168.1.150:/vol      26G   23G  3.1G  89% /mnt
/dev/mapper/cl-root     17G   15G  2.9G  84% /etc/hosts
shm                     64M     0   64M   0% /dev/shm
tmpfs                  5.1G     0  5.1G   0% /proc/acpi
tmpfs                  5.1G     0  5.1G   0% /proc/scsi
tmpfs                  5.1G     0  5.1G   0% /sys/firmware

root@e24fbcf8504b:/# cd /mnt/
root@e24fbcf8504b:/mnt# touch contenedor1
```

Ahora tendremos que ver el archivo creado en el servidor NFS:

```
[root@docker ~]# ls -l /vol/
-rw-r--r-- 1 root              root        0 ene 20 11:09 contenedor1
```

También podemos crear un volumen, con el driver NFS y asociarlo a un contenedor:

```
#docker volume create -d nfs --name vol1 -o share=192.168.1.150:/vol -o create=true

#docker run --name contenedor2-nfs  -dti  --rm  -v vol1:/datos ubuntu:trusty
```

docker exec -ti contenedor2-nfs /bin/bash

#root@80976bd79945:/# df -h
Filesystem   Size Used Avail Use% Mounted on
192.168.1.150:/vol/vol1   26G   23G 3.1G 89% /datos

```
#cd /datos/
#root@80976bd79945:/datos# ls
contenedor1
#root@80976bd79945:/datos# touch contenedor2
#root@80976bd79945:/datos# ls
contenedor1  contenedor2
```

Ahora podremos comprobar los volumentes con driver NFS:

```
[root@docker2 ~]# docker volume ls -f driver=nfs
DRIVER              VOLUME NAME
nfs                 192.168.1.150/vol
nfs                 vol1
```

A partir de la versión 1.12 de Docker, los pluguins cambian su arquitectura y se incluye la acción pluguin al cliente de docker.

La pagina oficial que describe el cambio de arquitectura es la siguiente:

https://docs.docker.com/engine/extend/

En este laboratorio instalaremos el pluguin sshfs sobre el servidor docker2, que nos permite montar un directorio utilizando SSH:

https://github.com/vieux/docker-volume-sshfs

Instalamos el pluguin utilizando la acción plugin install:

```
[root@docker2 ~]# docker plugin  install vieux/sshfs
Plugin "vieux/sshfs" is requesting the following privileges:
 - network: [host]
 - mount: [/var/lib/docker/plugins/]
 - mount: []
 - device: [/dev/fuse]
 - capabilities: [CAP_SYS_ADMIN]
Do you grant the above permissions? [y/N] y
latest: Pulling from vieux/sshfs
52d435ada6a4: Download complete
Digest: sha256:1d3c3e42c12138da5ef7873b97f7f32cf99fb6edde75fa4f0bcf9ed277855811
Status: Downloaded newer image for vieux/sshfs:latest
Installed plugin vieux/sshfs
```

Listamos los pluguins instalados actualmente:

```
[root@docker2 ~]# docker plugin ls
ID                  NAME                 DESCRIPTION              ENABLED
309fdb476300        vieux/sshfs:latest   sshFS plugin for Docker  true
```

Creamos los volúmenes utilizando el dirver instalado anteriormente en el servidor docker2:

```
[root@docker2 ~]# docker volume create -d vieux/sshfs --name volumenssh -o
sshcmd=root@192.168.1.152:/contenedores2 -o password=000000


[root@docker2 ~]# docker volume create -d vieux/sshfs --name volumenssh-docker -o
sshcmd=root@192.168.1.150:/contenedores -o password=000000
```

Nos aseguramos de que estén creado los directorios /contenedores2 en el servidor docker2 y contenedores en el servidor docker

Ejecutaremos nuestros contenedores desde el servidor docker2, utilizando los volumentes previamente creados:

```
#docker run -dti --name conte1  -v volumenssh:/data debian

#docker run -dti --name conte2  -v volumenssh-docker:/micontenedor
debian
```

Nos conectamos a los contenedore y veremos que estamos compartiendo los directorios de nuestros docker engine a través de sshfs:

```
# docker exec -ti conte1 /bin/bash
# docker exec -ti conte2 /bin/bash

# [root@docker2 ~]# docker volume ls
```

Limitar Recursos

Por defecto un contenedor puede utilizar todos los recursos disponibles del servidor donde este alojado. Esto puede causar que una aplicación ejecutándose dentro de uno de ellos pueda afectar al resto de ellos e incluso al propio servidor.

Docker proporciona la opción de limitar los recursos que un contenedor puede utilizr, memoria, procesador o entrada/salida de disco.

Para ello, utilizaremos diferentes opciones que nos proporciona Docker tanto en la acción **créate** o **run**.

Memoria

Aunque los contenedores no usan demasiada memoria las aplicaciones que alojan pueden utilizar mas memoria de la esperada. Ya sea por su uso o por errores en el código que deiven en un uso abusivo. A través de las siguientes opciones podemos limitar su uso:

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

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

Con limite para memoria hasta 300MB y sin swap:

```
$ docker run -ti -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

Con limite de memoria y con todo el swap que este disponible:

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

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

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

Cuando limitamos la memoria que puede utilizar un contenedor, es importante saber que desde dentro del contenedor no es posible ver esta limitación. El comando free u otros comandos como top, mostraran la memoria del servidor Docker y no la propia del mismo.

Para comprobar la limitación de memoria de un contenedor desde el servidor Docker podemos utilizar la acción inspect, las claves de relacionadas al limite de uso de la memoria son las siguientes:

- Memory: la cantidad de memoria máxima que puede utilizar con la especificación -m.
- Kernel/Memory: memoria de kernel limitada
- MemoryReservation: la memoria reservada para el contenedor especifico.
- MemorySwap: la cantidad de memoria swap máxima que puede utilizar.

- MemorySwappiness: el valor de swappiness para el contenedor.
- OOMKilled: indica si esta deshabilitad0 OOM killer.

Para obtener la limitación de memoria para el contenedor:

```
#docker inspect --format="{{ .HostConfig.Memory }}" <container_id>
```

INICIO AUTOMÁTICO

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

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

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

-always: reinicia el contenedor siempre.

Ejemplo:

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

Procesador

Al igual que con la memoria, el uso de procesador por un contenedor es ilimitado.

A través de las siguientes opciones es posible limitar tanto los procesadores a utilizar, como los ciclos

Para la limitación de CPU podemos limitar de la siguiente forma:

- **--cpu-shares** en un limitación relativa a la cantidad de CPUs y sería un tanto %. Osea si tenemos 8 CPUs podemos limitar a 50 y usariamos 4.
- **--cpu-period** especifica un tiempo limite de uso de CPU y normalmente va asociado al siguiente parámetro.
- **--cpu-quota** indica la cuota de uso de un CPU.
- **--cpus** especifica cuanto, del total disponible del procesador, puede utilizar (disponible a partir de Docker 1.13), ejemplo, --cpus “1.5” → Indica que tiene acceso garantizado a los recusos de un procesador y a la mitad de otro.
- **--cpuset-cpus** limita el uso de CPU/cores que puede utilizar, los formatos posibles son:
  - **n** un único procesador
  - **n-m** Rango de procesadores
  - **n,m** Lista de procesadores

Ejemplo:

--cpuset-cpus 0

--cpuset-cpus 0,2→Utilizara las cpus 0,1,2

--cpuset-cpus 0,2 → Utilizara los CPUs 0 y 2

Dispositivo entrada-salida y sistemas de ficheros.

En cuanto al acceso a los recursos conviene limitar en todo momento. Por ejemplo tenemos que limitar que el acceso al fichero null `/dev/zero` sea de solo lectura para evitar que un atacante modifique su contenido.

```
>_ docker run --device=/dev/zero:/dev/zero:r ...
```

Podemos limitar las operaciones de entrada-salida a disco por segundo. Esto nos puede sacar de un apuro si algo empieza a escribir en disco de manera masiva, sea un atacante o un error de programación. Para ello tenemos las siguientes opciones:

- ***--device-read-bps*** limita en bytes el tamaño de lecturas a un dispositivo, ejemplo –device-read-bps /dev/sda:10mb
- ***--device-write-bps*** limita en bytes el tamaño de escritura a un dispositivo, ejemplo, --device-write-bps /dev/sda:10mb
- ***--device-read-iops*** limita la veces que se lee de un dispositivo por segundo, ejemplo, --device-read-iops /dev/sda:100
- ***--device-write-iops*** limita la veces que se escribe de un dispositivo por segundo
- ***--blkio-weight*** especifica la prioridad para las acciones de IO, entre 10 a 1000, o para deshabilítalo. Mayor numero mallor prioridad, ejemplo –blkio-weigth 600
- ***--blkio-weight-device*** dispositivo a limitar por la prioridad de peso, ejemplo, --blkio-weight-device /dev/sda1

Vamos a limitar el acceso a disco con 1000 operaciones por segundo, se puede ir jugando con este número según necesidades pero lo importante es que el sistema lo soporte.

```
>_ docker run --device-write-iops /dev/sda:1000 ...
```

Docker Update

A través de la acción update es posible establecer nuevas restricciones a un contenedor que esta en ejecución, la sintaxis:

docker update opciones contenedor

Options

Name, shorthand	Default	Description
`--blkio-weight`	0	Block IO (relative weight), between 10 and 1000, or 0 to disable (default 0)
`--cpu-period`	0	Limit CPU CFS (Completely Fair Scheduler) period

Name, shorthand	Default	Description
`--cpu-quota`	0	Limit CPU CFS (Completely Fair Scheduler) quota
`--cpu-rt-period`	0	Limit the CPU real-time period in microseconds
`--cpu-rt-runtime`	0	Limit the CPU real-time runtime in microseconds
`--cpu-shares, -c`	0	CPU shares (relative weight)
`--cpuset-cpus`		CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems`		MEMs in which to allow execution (0-3, 0,1)
`--kernel-memory`		Kernel memory limit
`--memory, -m`		Memory limit
`--memory-reservation`		Memory soft limit
`--memory-swap`		Swap limit equal to memory plus swap: ‘-1’ to enable unlimited swap
`--restart`		Restart policy to apply when a container exits

Examples

The following sections illustrate ways to use this command.

Update a container's cpu-shares

To limit a container's cpu-shares to 512, first identify the container name or ID. You can use `docker ps` to find these values. You can also use the ID returned from the `docker run` command. Then, do the following:

```
$ docker update --cpu-shares 512 abebf7571666
```

Update a container with cpu-shares and memory

To update multiple resource configurations for multiple containers:

```
$ docker update --cpu-shares 512 -m 300M abebf7571666 hopeful_morse
```

Update a container's kernel memory constraints

You can update a container's kernel memory limit using the `--kernel-memory` option. On kernel version older than 4.6, this option can be updated on a running container only if the container was started with `--kernel-memory`. If the container was started *without* `--kernel-memory` you need to stop the container before updating kernel memory.

For example, if you started a container with this command:

```
$ docker run -dit --name test --kernel-memory 50M ubuntu bash
```

You can update kernel memory while the container is running:

```
$ docker update --kernel-memory 80M test
```

If you started a container *without* kernel memory initialized:

```
$ docker run -dit --name test2 --memory 300M ubuntu bash
```

Update kernel memory of running container `test2` will fail. You need to stop the container before updating the `--kernel-memory` setting. The next time you start it, the container uses the new value.

Kernel version newer than (include) 4.6 does not have this limitation, you can use `--kernel-memory` the same way as other options.

Update a container's restart policy

You can change a container's restart policy on a running container. The new restart policy takes effect instantly after you run `docker update` on a container.

To update restart policy for one or more containers:

```
$ docker update --restart=on-failure:3 abebf7571666 hopeful_morse
```

Laboratorio Limitar recursos contenedores

Si ejecutamos dos contenedores con diferentes prioridades realizando la misma tarea, veremos que los de mayor prioridad acederan al disco de una forma mas rápida:

docker run -dti --blkio-weight 300 --name prueba1 debian dd if=/dev/zero of=/root/prueba bs=10M count=100

docker run -dti --blkio-weight 600 --name prueba2 debian dd if=/dev/zero of=/root/prueba bs=10M count=100

En este laboratorio estamos ejecutando dos contenedores prueba1 y prueba2, con diferentes valores en la prioridad (peso), con --blkio-weight.

Si compromabos la salida del contenedor para observar la velociada de cada uno de ellos:

[root@docker ~]# docker logs prueba1

100+0 records in

100+0 records out

1048576000 bytes (1.0 GB, 1000 MiB) copied, 3.10053 s, 338 MB/s

[root@docker ~]# docker logs prueba2

100+0 records in

100+0 records out

1048576000 bytes (1.0 GB, 1000 MiB) copied, 4.75926 s, 220 MB/s

Como podemos observar en la salida del comando dd, el contenedor con mayor prioridad (prueba2), ejecuta la acción mas rápido debido a que la velociad de acceso al dispositivo es mas veloz.

En este laboratorio limitaremos la velocidad de escritura a un dispositivo podremos ulizar la opción ***--device-write-bps:***

En este caso estamos limitando la velocidad de escritura al dispositivo /dev/sdb (tendremos que aseguranos que estamos trabajando con este dispositivo, sino seria /dev/sda), a 10MB/s, utilizaremos el comando dd con las opciones oflag=direct para acceso directo al disco para poder observar la limitación.

docker run -dti --device-write-bps /dev/sdb:10mb --name prueba4 debian dd if=/dev/zero of=/root/prueba bs=10M count=100 oflag=direct

En este laboratorio restringiremos el numero de operaciones por segundo, utilizaremos la opción ***--device-write-iops***

docker run -dti --device-write-iops /dev/sdb:10 --name prueba4b debian dd if=/dev/zero of=/root/prueba bs=10M count=100 oflag=direct

En este laboratorio escribimos 100 veces un fragmento de 10MB. En este caso vemos que la velocidad se redujo debido a la limitación indicada. Si en ver de escribir 10Mb escribiéramos 1MB pero 1000 veces modemos comprobar lo que sucede:

docker run -dti --device-write-iops /dev/sdb:100 --name prueba4c debian dd if=/dev/zero of=/root/prueba bs=1M count=1000 oflag=direct

docker logs prueba4c

1000+0 records in

1000+0 records out

1048576000 bytes (1.0 GB, 1000 MiB) copied, 0.618129 s, 1.7 GB/s

Al necesitar mas operacions (1000), la velocidad se reduce a mas de la mitad, ya que hemos restringido el numero de operaciones por segundo.

Como observamos con las opciones que nos proporciona Docker, podemos restringir tanto la velocidad como el numero de operaciones por segundo, tanto para la escritura como para la lectura.

Logging Drivers

Por defecto, la salida que produce un contenedor es almacenada internamente por el servicio de Docker, a traces de la acción logs, ejemplo:

docker run -dti –name mysql mysql

docker logs mysql

Esta solución esta limitada por los siguientes motivos:

- Requiere que Docekr este en funcionamiento.
- Los registros no son permanentes, se pierden en cada reinicio.
- No se almacenan de forma centralizada, en caso de tener diversos servidores Docker, es necesario acceder al apropiado.

Docker a través de logging-drivers nos permite cambiar el comportamiento por defecto y almacenar los registros generados por los contenedores en diversas opciones.

Supported logging drivers

Driver	Description
`none`	No logs will be available for the container and `docker logs` will not return any output.
`json-file`	The logs are formatted as JSON. The default logging driver for Docker.
`syslog`	Writes logging messages to the `syslog` facility. The `syslog` daemon must be running on the host machine.
`journald`	Writes log messages to `journald`. The `journald` daemon must be running on the host machine.
`gelf`	Writes log messages to a Graylog Extended Log Format (GELF) endpoint such as Graylog or Logstash.
`fluentd`	Writes log messages to `fluentd` (forward input). The `fluentd` daemon must be running on the host machine.
`awslogs`	Writes log messages to Amazon CloudWatch Logs.
`splunk`	Writes log messages to `splunk` using the HTTP Event Collector.
`etwlogs`	Writes log messages as Event Tracing for Windows (ETW) events. Only available on Windows platforms.
`gcplogs`	Writes log messages to Google Cloud Platform (GCP) Logging.

https://docs.docker.com/engine/admin/logging/overview/#supported-logging-drivers

La configuración del driver de registros puede ser configurado a nivel del servicio y a nivel individual para cada uno de los contenedores. Tanto a nivel de servicio (opciones para dockerd) como a nivel de contenedor (con las acciones de run o créate) las opciones para especificar un nuevo driver y sus opciones son las siguientes:

- --log-driver driver: indica el dirver a utilizar a nivel global.
- --log-opt opciones: establece diferentes opciones de driver, como pueden ser el servidor a conectar, el puerto, o diferentes parámetros.

A través de la acción info, podemos ver las opciones del servidor Docker y entre ellas podemos ver el driver actual:

docker info |grep -i logging

Logging Driver: journald

En caso de un contenedor utilizaremos la acción inspect para optener el driver con el que se esta ejecutando:

docker inspect --format "{{ .HostConfig.LogConfig.Type }}" mysql

Journald

Journald

Este driver tiene dos características importantes:

- Aloja el registro dentro del journal del sistema, donde los demás servicios del sistema alojan los suyos. Es una forma centralizada dentro de un sistema individual de alojar toda la información generada.
- Es posible continuar utilizando docker logs, este es útil para evitar busquedas en ficheros con muchos registros.

La única opción a utilizar es una ya conocida: tag, para especificar una etiqueta que se incluirá en cada registro generado por el contenedor. Utilizando este drivers podemos filtrar los registros a través de dos claves: CONTAINER_NAME y CONTAINER-ID.

Ejemplo:

docker run --log-driver=journald -dti --name pruebajournald docker.io/centos

b17f79e573024542cf730f722996e35f28e9081ee4686405716d2d7f232ba81d

[root@docker /]# journalctl CONTAINER_NAME=pruebajournald -a

[root@docker /]# journalctl CONTAINER_NAME=pruebajournald -f

Syslog

El driver de syslog es uno de los mas utilizados a la hora de utilizar Docker, ya que la mayoría de empresas suelen poseer un servidor centralizado. Las opciones que podemos indicarle a este driver son las siguientes:

Option	Description	Example value
`syslog-address`	The address of an external `syslog` server. The URI specifier may be `[tcp | udp|tcp+tls]://host:port`, `unix://path`, or `unixgram://path`. If the transport is `tcp`, `udp`, or `tcp+tls`, the default port is `514`.	`--log-opt syslog-address=tcp+tls://192.168.1.3:514, --log-opt syslog-address=unix:///tmp/syslog.sock`
`syslog-facility`	The `syslog` facility to use. Can be the number or name for any valid `syslog` facility. See the syslog documentation.	`--log-opt syslog-facility=daemon`
`syslog-tls-ca-cert`	The absolute path to the trust certificates signed by the CA. **Ignored if the address protocol is not `tcp+tls`.**	`--log-opt syslog-tls-ca-cert=/etc/ca-certificates/custom/ca.pem`
`syslog-tls-cert`	The absolute path to the TLS certificate file. **Ignored if the address protocol is not `tcp+tls`.**	`--log-opt syslog-tls-cert=/etc/ca-certificates/custom/cert.pem`
`syslog-tls-key`	The absolute path to the TLS key file. **Ignored if the address protocol is not `tcp+tls`.**	`--log-opt syslog-tls-key=/etc/ca-certificates/custom/key.pem`
`syslog-tls-skip-verify`	If set to `true`, TLS verification is skipped when connecting to the `syslog` daemon. Defaults to `false`. **Ignored if the address protocol is not `tcp+tls`.**	`--log-opt syslog-tls-skip-verify=true`
`tag`	A string that is appended to the `APP-NAME` in the `syslog` message. By default, Docker uses the first 12 characters of the container ID to tag log messages. Refer to the log tag option documentation	`--log-opt tag=mailer`

Option	Description	Example value
	for customizing the log tag format.	
`syslog-format`	The `syslog` message format to use. If not specified the local UNIX syslog format is used, without a specified hostname. Specify `rfc3164` for the RFC-3164 compatible format, `rfc5424` for RFC-5424 compatible format, or `rfc5424micro` for RFC-5424 compatible format with microsecond timestamp resolution.	`--log-opt syslog-format=rfc5424micro`
`labels`	Applies when starting the Docker daemon. A comma-separated list of logging-related labels this daemon will accept. Used for advanced log tag options.	`--log-opt labels=production_status,geo`
`env`	Applies when starting the Docker daemon. A comma-separated list of logging-related environment variables this daemon will accept. Used for advanced log tag options.	`--log-opt env=os,customer`
`env-regex`	Applies when starting the Docker daemon. Similar to and compatible with `env`. A regular expression to match logging-related environment variables. Used for advanced log tag options.	`–log-opt env-regex=^(os

https://docs.docker.com/engine/admin/logging/syslog/#options

En este ejemplo, ejecutaremos un contedor y sus registros serán enviados al servidor syslog que esta en ejecuion en el mismo servidor Docker:

Tendremos que habilitar el syslog remoto en el purto/UDP

docker run --log-driver=syslog --log-opt syslog-address=udp://127.0.0.1:514 --log-opt syslog-facility=daemon --log-opt tag=pruebasyslog3 --name pruebasyslog3 -dti jboss-eap-mysql-centos-jdk1.7

En otra consola vermos como se registran los logs:

[root@docker /]# tail -f /var/log/messages

En este ejemplo, arrancamos un contenedor en el servidor docker2, con una images de tomcat, podremos comprobar como se registran los logs en el servidor remoto docker:

[root@docker2 ~]# docker run --log-driver=syslog --log-opt syslog-address=udp://192.168.1.150:514 --log-opt syslog-facility=daemon --log-opt tag=jboss-syslog --name pruebasyslog3 -dti tomcat

[root@docker /]# tail -f /var/log/messages

Jul 1 12:54:12 192.168.1.152 jboss-syslog[1002]: Using CATALINA_BASE: /usr/local/tomcat#015

Jul 1 12:54:12 192.168.1.152 jboss-syslog[1002]: Using CATALINA_HOME: /usr/local/tomcat#015

Jul 1 12:54:12 192.168.1.152 jboss-syslog[1002]: Using CATALINA_TMPDIR: /usr/local/tomcat/temp#015

Jul 1 12:54:12 192.168.1.152 jboss-syslog[1002]: Using JRE_HOME: /docker-java-home/jre#015

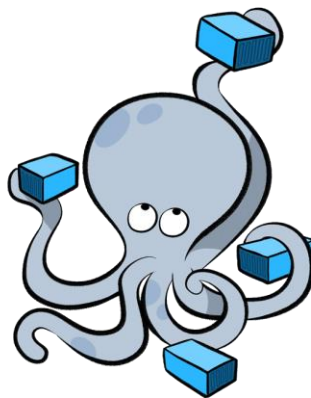
…

Laboratorios Registros logs

En este laboratorio, probaremos con varios contenedores tanto locales como remotos que se registren los logs en el syslog remoto del servidor docker (192.168.1.150).

Docker Compose

Con Dockerfile podemos usar Docker Engine para crear imágenes y posteriormente desplegar contenedores con ellas. Docker nos proporciona otro componente llamado Compose para definir y ejecutar contenedores basados en plantillas.



Esta plantilla contendrá:

- Lista de imágenes a utilizar para ejecutar contenedores.
- La ruta de los ficheros Dockerfile que crearan las imágenes previamente especificando si furea necesario.
- Los puertos a exponer para acceder a dicho contenedor
- Los volúmenes a utilizar
- Las variables necesarias para ejecutar las aplicaciones

Las principales características de este componente son:

- Compose fue diseñado para ejecutar contenedores en un solo servidor y no en alta disponibilidad.
- Mas centrado en entornos de desarrollo que de producción.

Los casos de uso mas común de Docker Compose se describen a continuación:

- Entornos de desarrollo: simplifica todo el proceso de creación de imágenes, de ejecución de contenedores y comunicación entre ellos.
- Automatizacion de tests a aplicaciones: es posible utilizar lo denomindado continous integration (CI), integración continua, para desplegar aplicaciones que están bajo desarrollo dentro de un contenedor y comprobar su estado y sus dependencias.

- En el caso de solo poseer un único servidor Docker, es posible utilizarlo como herramienta para desplegar aplicaciones.

Para entornos de producción con diferentes servidores, se recomienda utilizar Docker Swarm para crear un cluster que contendrá los contenedores en alta disponibilidad.

Sintaxis Plantilla

https://docs.docker.com/compose/compose-file/

La sintaxis en la plantilla del fichero de configuración docker-compose.yml, el formato utlizado es YAML, cuya sintaxis y legibilidad son muy fáciles de entender y aprender.

La plantilla definirá servicios (contenedores), redes, volúmenes. La extensión del fichero puede ser yml o yaml. Las instrucciones mas comunes son:

version: Indica la versión del formato a utilizar. Las versiones posibles son:

- 1: Las versión histórica, con ella se puede definir volúmenes, redes o incluir argumentos para crear imágenes.
- 2: En esta versión todos los servicios deben ser declarados bajo la instrucion services. Las redes pueden especificarse bajo la instrucción networks.
- 3: La versión recomendada en nuevas versiones al ser retrocompatible, soporta Docker Swarm.

context: directorio o url

Similar al build, pero solo compatible con la versión 2 y posterior, indica el directorio o una dirección de un repositiorio git que contenga el fichero Dockerfile.

dockerfile: fichero

Especifica un nombre fichero altenativo por defecto Dockerfile, ejemplo:

- dockerfile: Dockerfile-dev

args: argumentos

Especifila la lista de argumentos a pasar para crear la imagen basada en el fichero Dockerfile. El fichero Dockerfile deberá incluir las instrucciones ARG, ejemplo:

args:

usuario: root

configuración: dev.conf

web : Nombre que define el servicio

build : crea el contenedor, en este caso pasamos . , para que haga uso del Dockerfile que creamos recientemente.

command : El comando que le pasamos al contenedor al momento de su ejecución.

ports: Mapeo de Puertos

links : Definimos el enlace de los contenedores

volumes : Creamos el volumen que contiene nuestro código

db: Nombre que define nuestro contenedor con la base de datos

image: la imagen donde basará su contenedor

environment: aquí pasamos las variables, en este caso es un contenedor con MySQL y le pasamos la base de datos a crear y las credenciales.

entrypoint: comando

Indica el punto de entrada para la imagen distinto al definido, en el fichero Dockerfile. Esta espresion se usa para teneruna imagen para todos los entornos, pero el fichero plantilla definirá un punto de entrada para cada uno de ellos.

container_name: nombre

Especifica el nombre para el contenedor al crear el servicio

depends_on: servicios

Cuando desplegamos varios servicios (contenedores), es normal que uno de ellos dependa de los otros. Por ejemplo, un servidor web depende de un servidor de base de datos para su funcionamiento y queremos que primero se ejecute este ultimo.

env_file: fichero

Especifica un fichero que contiene las variables de entorno a pasar a la creación de la imgen.

expose: puertos

Lista de puertos a exponer, solo para la creación de imagen.

network_mode: red

Especifica el modo de red a utilizar por los servicioes, por ejemplo:

network_mode: “bridge”

https://docs.docker.com/compose/

Algunos comandos docker compose

- **docker-compose build**: Lanza el proceso de generación de imágenes docker de los servicios definidos en docker-compose.yml
- **docker-compose create**: Lanza el proceso de creación de los servicios indicado en el archivo docker-compose.yml
- **docker-compose start**: Inicia el servicio indicado
- **docker-compose stop**: Detiene el servicio indicado
- **docker-compose rm**: Elimina el servicio, no la imagen, indicado

- **docker-compose exec**: Ejecuta un comando en el container del servicio indicado
- **docker-compose up**: Crea imágenes, crea los servicios y los inicia según el docker-compose.yml
- **docker-compose down**: Detiene y elimina servicios

Instalar docker-compose

```
# curl -L https://github.com/docker/compose/releases/download/1.14.0/docker-compose-
`uname -s`-`uname -m` > /usr/local/bin/docker-compose

#chmod +x /usr/local/bin/docker-compose

# docker-compose --version
docker-compose version 1.14.0, build 1719ceb
```

https://docs.docker.com/compose/install/

Comenzando con Docker Compose

Definición de los servicios de Docker Compose

Cuando comenzamos a ejecutar contenedores utilizando cada vez más la funcionalidad de `docker run`, los comandos pueden comenzar a ser bastante largos y difíciles de recordar.

Afortunadamente, Docker Compose nos permite escribirlos en **YAML.**

Tomemos un comando `docker run` suficientemente complicado para iniciar un contenedor MySQL configurado usando variables de entorno y convertirlo en YAML en a `docker-compose.yml`. Aquí está el comando con el que comenzamos:

```
$ docker run -d

 --name mysql:5.7

 -p 3306:3306

 -e MYSQL_ROOT_PASSWORD=secret_password

 -e MYSQL_DATABASE=rootdesdezero

 mysql
```

Podemos crear este contenedor exacto como un "servicio" dentro de un archivo `docker-compose.yml` después de configurar un poco de la plantilla. Aquí está nuestro comienzo:

docker-compose.yml

```
version: "3"

services:

  mysql:

    image: "mysql"

    environment:

      MYSQL_ROOT_PASSWORD: "secret_password"

      MYSQL_DATABASE: "rootdesdezero"

    ports:

      - "3306:3306"
```

Hay algunas versiones diferentes de la sintaxis utilizada para el archivo `docker-compose.yml`, pero casi siempre deberíamos estar usando la más nueva, que es 3.x dependiendo de nuestra versión de docker. A partir de ahí, declaramos que queremos definir "servicios" y cualquiera de los elementos aquí debajo será un contenedor que Docker Compose creará y ejecutará para nosotros.

Dentro de la definicion `mysql` de nuestro servicio, tenemos claves que podemos establecer en ese mapa muy de cerca a las opciones que usamos cuando lo usamos manualmente `docker run`. Vale la pena señalar que si una opción toma una lista y los valores no incluyen un signo igual (`=`), usaremos una lista YAML (comenzando con `-`), de lo contrario usaremos pares clave/valor YAML como lo hicimos en la sección `environment`.

Desde el directorio donde creamos el archivo `docker-compose.yml`, podemos crear y ejecutar nuestros contenedores en segundo plano usando `docker-compose up -d,` para demonizar los contenedores:

```
$ docker-compose up -d
```

Eliminar variables de entorno codificadas

Una mejora que podemos hacer a nuestro servicio `mysql` sería no codificar las variables de entorno. Lo ideal es que deseemos agregar nuestros archivos `docker-compose.yml` a un repositorio de código y no queremos comprometer nuestras credenciales en nuestro repositorio de código git.

Afortunadamente, Docker Compose proporciona una excelente manera para que pasemos nuestros valores variables de entorno existentes. Lo primero que debemos hacer es eliminar los valores de nuestro archivo `docker-compose.yml` manteniendo las claves:

docker-compose.yml

```
version: "3"

services:

  mysql:

  image: "mysql:5.7"

    environment:

      MYSQL_ROOT_PASSWORD:

      MYSQL_DATABASE:

    ports:

      - "3306:3306"
```

Para probar esto, queremos eliminar el contenedor que Docker Compose creó usando `docker-compose down`:

```
$ docker-compose down
```

A continuación, definiremos nuestras variables de entorno en línea cuando estemos ejecutando `docker-compose up`:

```
$ MYSQL_ROOT_PASSWORD=secret_password MYSQL_DATABASE=rootdesdezero
docker-compose up -d
```

Ya no estamos configurando estos valores en nuestro archivo, pero si inspeccionamos el contenedor `MYSQL_` las variables de entorno, podemos ver que los valores están configurados correctamente.

Nota: Puede encontrar el nombre de su contenedor usando `docker-compose ps`

```
$ docker inspect docker-compose-example_mysql_1 | grep MYSQL_
```

Orquestando Contenedores

Saber cómo crear un contenedor con Docker Compose nos acerca bastante a poder orquestar múltiples contenedores. Seguiremos los mismos pasos creando otro objeto al mismo nivel que nuestra clave `mysql`.

En este ejemplo, creemos un contenedor de **blog Ghost** que requerirá y se conectará a nuestro servicio `mysql`:

docker-compose.yml

```
version: "3"
services:
  mysql:
    image: "mysql:5.7"
    environment:
      MYSQL_ROOT_PASSWORD:
      MYSQL_DATABASE:
    ports:
      - "3306:3306"
  blog:
    image: "ghost:2-alpine"
    ports:
      - "8080:2368"
    environment:
      DATABASE__CLIENT: mysql
      DATABASE__CONNECTION__HOST: mysql
      DATABASE__CONNECTION__USER: root
      DATABASE__CONNECTION__DATABASE:
      DATABASE__CONNECTION__PASSWORD:
    depends_on:
      - mysql
```

Tendremos que tener en cuenta aquí es nuestra clave `depends_on` en el servicio `blog`. Al especificar esto, le estamos diciendo a Docker Compose que el servicio `mysql` debe iniciarse antes que el servicio `blog` porque el servicio `blog` necesita que se ejecute.

Además, cuando estamos configurando la conexión de la base de datos para nuestro blog, especificamos el "host" de nuestra base de datos como tal `mysql`. Esto parece muy extraño, pero en la red interna que Docker Compose creó para nuestros contenedores, `mysql` otros contenedores (concretamente nuestro contenedor `blog`) pueden acceder al contenedor utilizando el nombre de dominio completo (FQDN) de `mysql`.

Ya tenemos el contenedor `mysql` en ejecución, pero aún podemos ejecutarlo `docker-compose up` y ejecutará nuestro contenedor adicional:

```
$ MYSQL_ROOT_PASSWORD=secret_password MYSQL_DATABASE=rootdesdezero
DATABASE__CONNECTION__PASSWORD=secret_password
DATABASE__CONNECTION__DATABASE=rootdesdezero docker-compose up -d
```

Ahora podemos acceder a nuestro blog conectándonos a la dirección IP de nuestro host Docker en el puerto `8080`.

Crear volúmenes

Vamos a echar un vistazo de cómo mejorar la forma en que almacenamos datos para nuestro blog montando un volumen para que nuestra base de datos persista. Este cambio nos permitirá conservar nuestros datos incluso si eliminamos el contenedor `mysql`.

Docker Compose facilita esto porque considera que los volúmenes y las redes son objetos de nivel superior al igual que los "servicios". Creemos un volumen llamado `db-data` y lo montamos en nuestro contenedor `mysql`:

docker-compose.yml

```
version: "3"
volumes:
  db-data:
    external: false
services:
  mysql:
    image: "mysql:5.7"
    environment:
      MYSQL_ROOT_PASSWORD:
      MYSQL_DATABASE:
    ports:
      - "3306:3306"
    volumes:
      - "db-data:/var/lib/mysql"
  blog:
    image: "ghost:2-alpine"
    ports:
      - "8080:2368"
    environment:
      DATABASE__CLIENT: mysql
      DATABASE__CONNECTION__HOST: mysql
      DATABASE__CONNECTION__USER: root
      DATABASE__CONNECTION__DATABASE:
      DATABASE__CONNECTION__PASSWORD:
    depends_on:
      - mysql
```

Esto creará el volumen para nosotros. La configuracion `external:false` le dice a Docker Compose que no creamos el volumen "externamente" para Docker Compose.

Para que nuestro servicio `mysql` use esto, necesitamos crear un nuevo contenedor.

Ejecutaremos `docker-compose down`:

Ahora podemos recrear nuestros contenedores para que los datos se escriban en un volumen.

```
$ MYSQL_ROOT_PASSWORD=secret_password MYSQL_DATABASE=rootdesdezero
DATABASE__CONNECTION__PASSWORD=secret_password
DATABASE__CONNECTION__DATABASE=rootdesdezero docker-compose up -d
```

Creando aplicaciones en contenedores con docker-compose

Vamos a crear una aplicación en python y la vamos a guardarla en un contenedor. Comenzamos creando un nuevo *build context*:

```
#mkdir -p  /miaplicacion
#cd /miaplicacion
```

El código de la aplicación es el siguiente, lo guardaremos en un archivo llamado app.py:

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(),  visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Y por último definimos nuestro *Dockerfile*:

 # Partimos de una base oficial de python

FROM python:2.7-slim

 # El directorio de trabajo es desde donde se ejecuta el contenedor al iniciarse

WORKDIR /app

 # Copiamos todos los archivos del build context al directorio /app del contenedor

COPY app.py /app

 # Ejecutamos pip para instalar las dependencias en el contenedor

RUN pip install --upgrade pip

#RUN pip install --trusted-host pypi.python.org -r requirements.txt

RUN pip install Flask

RUN pip install Redis

 # Indicamos que este contenedor se comunica por el puerto 80/tcp

EXPOSE 80

 # Declaramos una variable de entorno

ENV NAME World

 # Ejecuta nuestra aplicación cuando se inicia el contenedor

CMD ["python", "app.py"]

Para conocer todas las directivas visita la documentación oficial de Dockerfile.

En total debemos tener 2 archivos:

```
$ ls
app.py  Dockerfile
```

Ahora construimos la imagen de nuestra aplicación, no olvidarse del . final:

```
docker build -t miaplicacion .
```

Y comprobamos que está creada:

```
$ docker image ls
REPOSITORY        TAG              IMAGE ID          CREATED            SIZE
miaplicacion      latest           88a822b3107c      56 seconds ago     132MB
```

Probar nuestro contenedor

Vamos a arrancar nuestro contenedor y probar la aplicación:

docker run --rm -p 4000:80 miaplicacion

Lo que arranca la aplicación Flask:

```
$ docker run --rm -p 4000:80 miaplicacion
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Comprobamos en el puerto 4000 si efectivamente está iniciada o no: http://localhost:4000.

Obtendremos un mensaje como este:

Hello World!

Hostname: 0367b056e66e
Visits: cannot connect to Redis, counter disabled

Ya tenemos una imagen lista para ser usada. Pulsamos Control+C para interrumpir y borrar nuestro contenedor.

Creando la aplicación

En este caso nuestro contenedor no funciona por sí mismo. Es muy habitual que dependamos de servicios para poder iniciar la aplicación, habitualmente bases de datos. En este caso necesitamos una base de datos *Redis* que no tenemos.

Vamos a aprovechar las características de ***Compose*** para levantar nuestra aplicación.

Vamos a crear el siguiente archivo ***docker-compose.yaml***:

```
version: "3"
services:
  web:
    build: .
    ports:
      - "4000:80"
  redis:
    image: redis
    ports:
      - "6379:6379"
    volumes:
      - "./data:/data"
    command: redis-server --appendonly yes
```

La principal diferencia con respecto al paso anterior, es que en un servicio podemos indicar una imagen (parámetro imagen) o un *build context* (parámetro build).

Esta es una manera de integrar las dos herramientas que nos proporciona *Docker*: la creación de imágenes y la composición de aplicaciones con servicios.

Ahora podemos comprobar el correcto funcionamiento de nuestra aplicación:

```
# docker-compose up
```

http://192.168.33.10:4000/

Hello World!

Hostname: 2ee47b37f067
Visits: 1

Hello World!

Hostname: 2ee47b37f067
Visits: 7

Balanceo de carga

Nos aseguramos de eliminar todos los contenedores del laboratorio anterior, incluida la imagen creada:

Vamos a modificar nuestro *docker-compose.yaml*:

```
version: "3"
services:
  web:
    build: .
  redis:
    image: redis
    volumes:
     - my-db-redis:/data
    command: redis-server --appendonly yes
  lb:
    image: dockercloud/haproxy
    ports:
      - 4000:80
    links:
      - web
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
volumes:
  my-db-redis:
```

En este caso, el servicio web no va a tener acceso al exterior (hemos eliminado el parámetro ports). En su lugar hemos añadido un balanceador de carga (el servicio lb).

Vamos a arrancar esta nueva aplicación, pero esta vez añadiendo varios servicios web:

```
docker-compose up -d --scale web=5
```

Esperamos a que terminen de iniciar los servicios:

```
$ docker-compose up -d --scale web=5

Creating network "miaplicacion_default" with the default driver
Creating miaplicacion_redis_1 ... done
Creating miaplicacion_web_1   ... done
Creating miaplicacion_web_2   ... done
Creating miaplicacion_web_3   ... done
Creating miaplicacion_web_4   ... done
Creating miaplicacion_web_5   ... done
Creating miaplicacion_lb_1    ... done
```

Podemos comprobar como del servicio web nos ha iniciado 5 instancias, cada uno con su sufijo numérico correspondiente. Si usamos docker ps para ver los contenedores disponibles tendremos:

```
$ docker ps
CONTAINER ID  IMAGE              [...]   PORTS                                NAMES
77acae1d0567  dockercloud/haproxy  [...]   443/tcp, 1936/tcp, 0.0.0.0:4000->80/tcp  miaplicacion_lb_1
5f12fb8b80c8  miaplicacion_web    [...]   80/tcp                              miaplicacion_web_5
fb0024591665  miaplicacion_web    [...]   80/tcp                              miaplicacion_web_2
a20d20bdd129  miaplicacion_web    [...]   80/tcp                              miaplicacion_web_4
53d7db212df8  miaplicacion_web    [...]   80/tcp                              miaplicacion_web_3
41218dbbb882  miaplicacion_web    [...]   80/tcp                              miaplicacion_web_1
06f5bf6ed070  redis               [...]   6379/tcp                            miaplicacion_redis_1
```

Vamos a fijarnos en el CONTAINER ID y vamos a volver a abrir nuestra aplicación:

http://192.168.33.10:4000/

Si en esta ocasión vamos recargando la página, veremos como cambian los *hostnames*, que a su vez coinciden con los identificadores de los contenedores anteriores.

Info

Esta no es la manera adecuada de hacer balanceo de carga, puesto que todos los contenedores están en la misma máquina, lo cual no tiene sentido. Solo es una demostración. Para hacer balanceo de carga real necesitaríamos tener o emular un clustes de máquinas y crear un enjambre (*swarm/kubernetes*).

Laboratorios docker-compose

1. En este primer laboratorio utilizaremos un sistema de foros (MyBB), y lo conectaremos contra un servidor de base de datos mysql.

Docker Compose se basa en un fichero con extensión yml donde vamos a indicar que imagen queremos desplegar, cómo se va a configurar y de qué depende. Todas aquellas dependencias se van a ejecutar con anterioridad, para así poder realizar una instalación correcta.

Esta seria la configuración de despliegue:

#vi docker-compose.yml

```
version: '2'

 services:

  db:

    image: mysql:5.7

    volumes:

      - "./.data/db:/var/lib/mysql"

    ports:

      - "3306:3306"

    restart: always

    environment:

      MYSQL_ROOT_PASSWORD: 1234

      MYSQL_DATABASE: foro

      MYSQL_USER: admin

      MYSQL_PASSWORD: 1234

  mybb:

    depends_on:

      - db

    image: fbmac/mybb:latest

    links:

      - db

    ports:
```

- "8000:80"

restart: always

Como podemos ver hay dos servicios:

- La base de datos, en este caso uso MySQL v5.7, le digo que me redirija el tráfico para poder acceder a la misma y además configuro algunos datos como las contraseñas o una primera base de datos.
- El foro, en este caso se usa MyBB (al no ser una imagen oficial, hay que poner el usuario que la hizo disponible “fbmac”), he indicado la dependencia y también he redirigido el tráfico.

Para realizar el despliegue, lo único que hay que hacer es ejecutar el comando **docker-compose up -d,** desde la carpeta donde tenemos guardado el fichero yml.

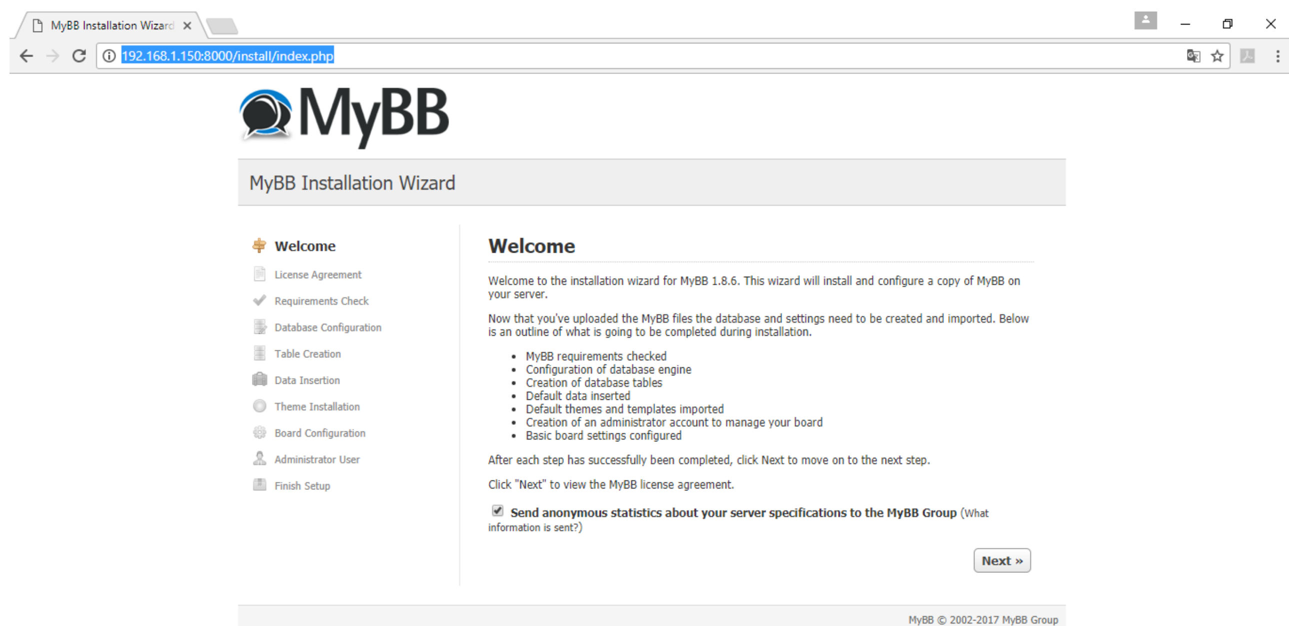
docker-compose up

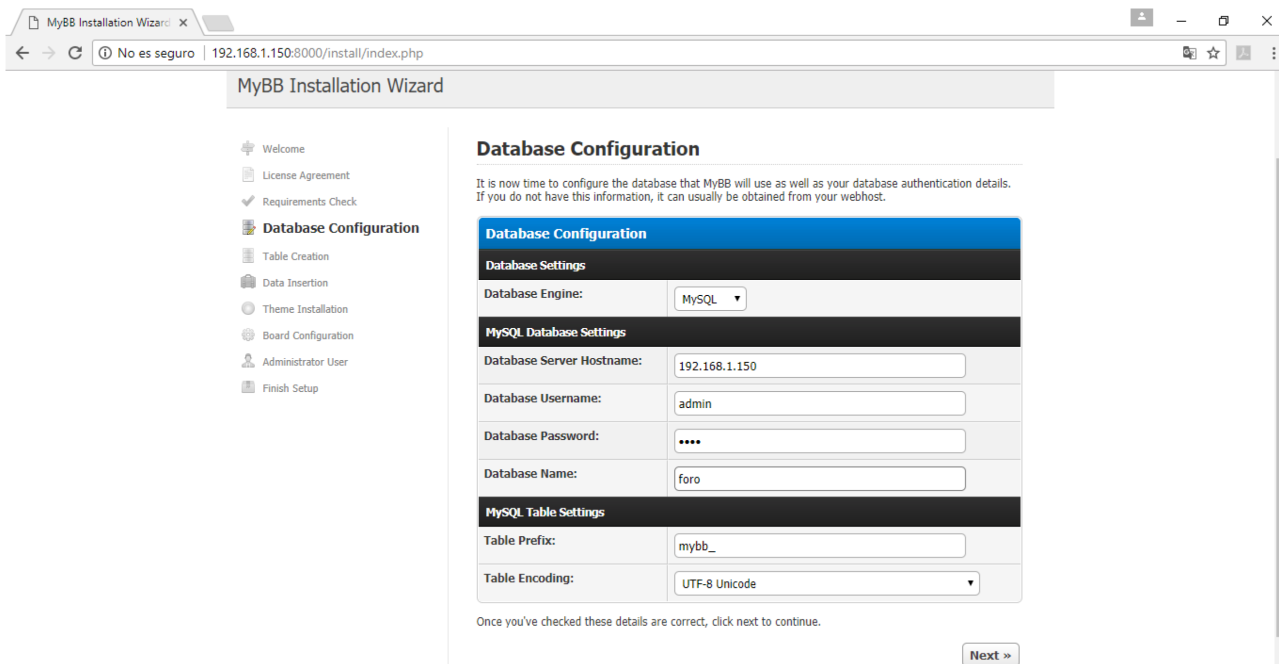
Una vez que lo hacemos nos dirá que se ha creado db_1 y mybb_1, por ese orden.

Una vez que ya tenemos todo listo, procedemos a realizar la instalación de MyBB mediante la interfaz web.

Lo primero que hacemos es indicar los datos de nuestra base de datos:

http://192.168.1.150:8000/install/index.php





Tras finalizar el asistente, tendremos que eliminar el directorio install, del contenedor mybb:

[root@docker ~]# docker exec -ti bf435f10c499 /bin/bash

root@bf435f10c499:/var/www# ls

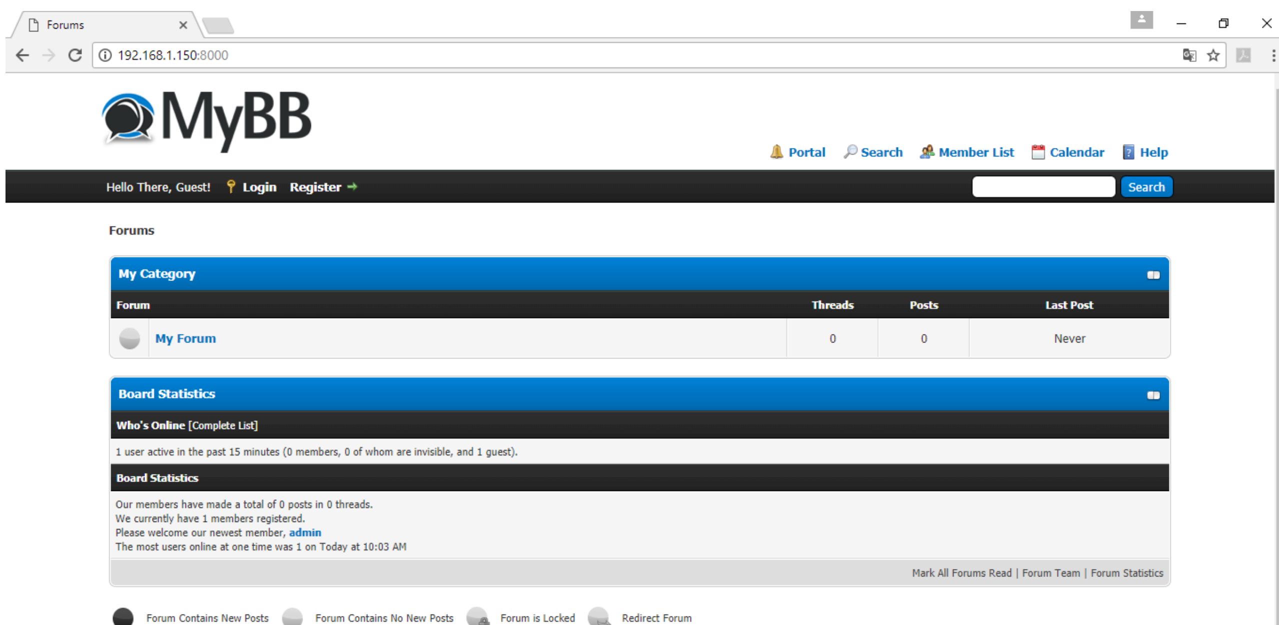
html mybb

root@bf435f10c499:/var/www# cd mybb/

root@bf435f10c499:/var/www/mybb# rm -rf in

inc/   index.php install/

root@bf435f10c499:/var/www/mybb# rm -rf install/



2. Laboratorio Wordpress con Docker Compose

En este laboratorio de Docker Compose instalaremos WordPRess en un entorno aíslado contruido mediante contenedores Docker.

mkdir -p /docker-compose/Wordpress/

Creamos el fichero **docker-compose.yml** que contendrá la definición de nuestro proyecto. Nos basaremos en dos instancias separadas: Una instancia para Wordpress y otra instancia separada para MySQL con un volumen montado para la persistencia de los datos de mysql .

Este será el fichero **docker-compose.yml** :

```
version: '2'
services:
  db:
    image: mysql:5.7
    volumes:
      - "./.data/db:/var/lib/mysql"
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    links:
      - db
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress
```

* El directorio **./.data/db** será automáticamente creado en el directorio del proyecto, a la misma altura que el fichero docker-compose.yml , y contendrá los datos persistentes de la base de datos de nuestro WordPress.

Contruyendo el proyecto

Ahora vamos a ejecutar el comando **docker-compose up -d** estando situados dentro del directorio del proyecto.

Este comando descargará (pull) las imágenes necesarias y iniciará el contenedor de Wordpress y de la base de datos.

```
$ docker-compose up -d
Creating network "wordpresswptesteando_default" with the default driver
Pulling db (mysql:5.7)...
5.7: Pulling from library/mysql
Digest: sha256:e6dc999f4e2d5982e74a008ff08c0641f0832e19339aaf4f4c2bc2ca426e96c6
Status: Downloaded newer image for mysql:5.7
Creating wordpresswptesteando_db_1
Pulling wordpress (wordpress:latest)...
latest: Pulling from library/wordpress
357ea8c3d80b: Already exists
Digest: sha256:56cd7233bf69a580d823d29ad16c085392abf3fc00b1e4ed7b955b83db2544f7
Status: Downloaded newer image for wordpress:latest
Creating wordpresswptesteando_wordpress_1
```

Con esto ya tendremos creados y corriendo dos contenedores:

- `wordpresswptesteando_db_1` : Es el contenedor MySQL para nuestra base de datos de Wordpress.
- `wordpresswptesteando_wordpress_1` : Es el contenedor que contiene nuestra instancia de WordPress además del servidor Web apache.

Accediendo a WordPress desde el navegador

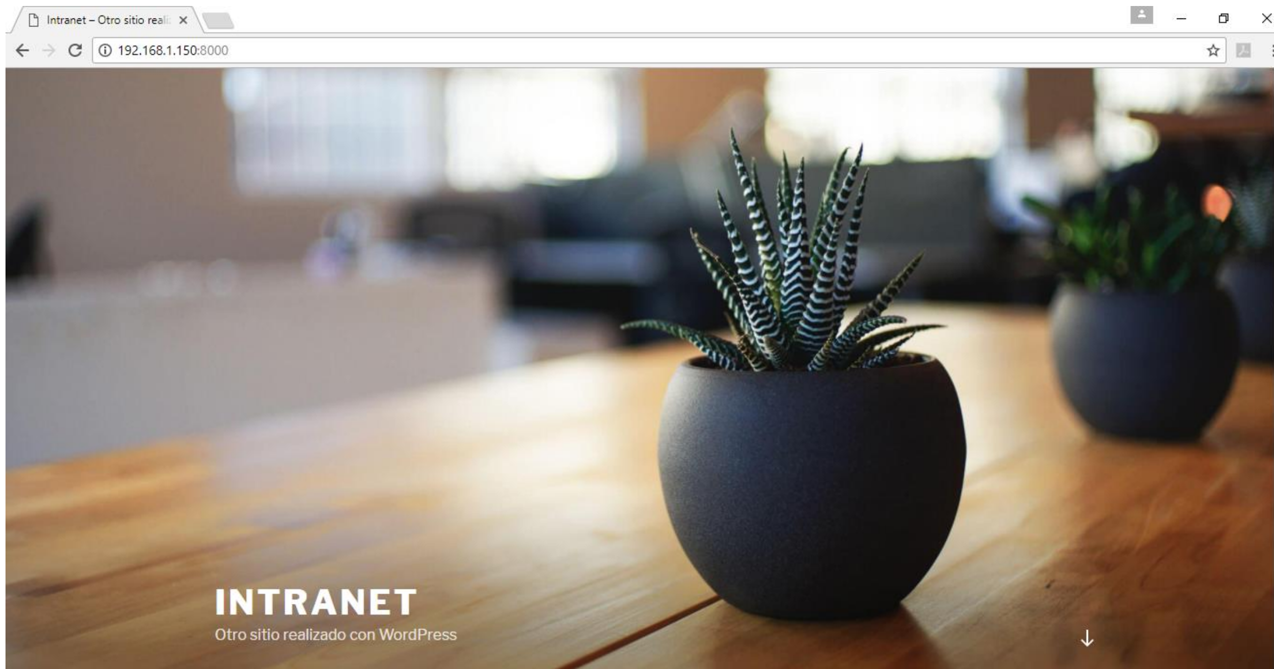
Al definir nuestro proyecto WordPress anteriormente indicamos la siguiente opción:

```
ports:
  - "8000:80"
```

Con esto hicimos un link entre los puertos `80` del servidor web que está corriendo en nuestro contenedor WordPress y el puerto `8000` de nuestra máquina local.

Para acceder a nuestro sitio web WordPress recien creado, deberemos abrir el navegador web en la dirección ip local de nuestra máquina o `localhost` pero en el puerto que tenemos linkeado al contenedor Wordpress, es decir, el puerto `8000` .

Para ello vamos a abrir `http://192.168.1.150:8000` en nuestro navegador y ya podremos comenzar con la instalación de WordPress.



3. Laboratorio Jboss EAP con Docker Compose

En este laboratorio, configuraremos con docker-copose, un proyecto para desplegar un contenedor de Jboss EAP, y linkarlo con una base de datos mysql, a continuacion desplegaremos una aplicaciones en Jboss llamada SGAJBoss.war, la cual tendra una conexión JDBC al contenedor de mysql, y lo hacemos pasar a traves de nuestro balanceador de Apache.

Nos posicionamos en el directorio del proyecto:

cd /docker-jboss-eap-master

Creamos el fichero **docker-compose.yml** que contendrá la definición de nuestro proyecto. Nos basaremos en dos instancias separadas: Una instancia para Jboss EAP a través de la imagen jboss-eap-mysql-centos-jdk1.7 y otra instancia separada para MySQL con un volumen montado para la persistencia de los datos de mysql, y donde copiaremos un script para la carga de la base de datos que necesita la aplicacione SGAJBoss .

En el archivo de configuración (yml), estamos exponiendo puertos del contenedor de Jboss, esto no será útil para escalar luego contenedores de Jboss, y hacerlos pasar a través de nuestros frontales Web, para que todo fuese elástico, en nuestros servidores web, tendríamos que utilizar el modulo **mod_cluster**, que entre sus ventajas tiene:

- El factor de carga se calcula en base a lo dice cada nodo.
- **Tiene auto Discovery de nodos**, algo importantísimo en un ambiente elastic cloud, donde los nodos pueden levantarse dinámicamente en base a la demanda.
- El módulo de nodo de mod_cluster ya viene integrado en JBoss AS6.
- Autodetección de nodos caídos.
- HeartBeat, Balanceo basado en la carga de los nodos,
- Transparent Failover, Sticky Session,

Este será el fichero **docker-compose.yml** :

```
version: '2'
 services:
  jboss:
   image: jboss-eap-mysql-centos-jdk1.7
  #container_name: jboss1
   volumes:
       - "./.data/jboss/logs:/opt/jboss-eap-6.4/standalone/log"
       - "./.data/jboss/deployments:/opt/jboss-eap-6.4/standalone/deployments"
   depends_on:
      - db
  #ports:
   # - 8009:8009
    #- 8080:8080
    #- 9990:9990
   expose:
       - 8080
       - 8009
       - 9990

  db:
     image: docker.io/mysql/mysql-server
     container_name: mysql
     volumes:
       - "./.data/db:/var/lib/mysql"
     ports:
       - "3306:3306"
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: 123456
       MYSQL_DATABASE: test2
       MYSQL_USER: admin
       MYSQL_PASSWORD: 123456
```

Contruyendo el proyecto

Ahora vamos a ejecutar el comando **docker-compose up** estando situados dentro del directorio del proyecto.

Este comando descargará (pull) las imágenes necesarias y iniciará el contenedor de Jboss y de la base de datos.

[root@docker docker-jboss-eap-master]# docker-compose up

Como podemos observar ha creado dos contenedores dockerjbosseapmaster_jboss_1 y mysql

[root@docker docker-jboss-eap-master]# docker-compose ps

```
Name                          Command              State      Ports
----------------------------------------------------------------------------------------------------------
dockerjbosseapmaster_jboss_1   /bin/sh -c $JBOSS_HOME/bin ...   Up   8009/tcp, 8080/tcp, 9990/tcp, 9999/tcp
mysql                          /entrypoint.sh mysqld            Up   0.0.0.0:3306->3306/tcp, 33060/tcp
```

A continuacion vamos a copar el script de base de datos de mysql ScriptBaseDatosSGA.sql, en:

```
volumes:
       - "./.data/db:/var/lib/mysql"
```

[root@docker db]# cp /root/ScriptBaseDatosSGA.sql /docker-jboss-eap-master/.data/db

Ahora entramos en el contenedor de mysql y creamos la base de datos necesaria llamata test, para poder desplegar la aplicación SGAJBoss.

[root@docker docker-jboss-eap-master]# **docker exec -ti mysql /bin/bash**

bash-4.2# cd /var/lib/mysql

bash-4.2# mysql -u root -p < ScriptBaseDatosSGA.sql

Enter password:

bash-4.2# ls -l

drwxr-x--- 2 mysql mysql   96 Jul 7 11:04 test

A continuacion salimos del contenedor, recordar que estos datos siempre persistiran, ya que tenemos un volumen montado contra el contenedor de la base de datos.

Para realizar el despliegu de la aplicación tendremos que averiguar que direcionamiento ip tienen nuestros contenedores, en este lab:

docker network ls

NETWORK ID	NAME	DRIVER	SCOPE
57bb5f8c4cf7	**dockerjbosseapmaster_default**	**bridge**	**local**

[root@docker docker-jboss-eap-master]# **docker network inspect dockerjbosseapmaster_default**

Donde podemos observar los contenedores asociados:

"Containers": {

"3ff4f84d67cf9c4ba4dff8b64521cb6edbd189e4afaa552acaed5bffb577b03d": {

"Name": "dockerjbosseapmaster_jboss_1",

"EndpointID": "185d2466e049ad28df6fa5921c07474b36946e93d76ff31e8b44d150ac9cc92f",

"MacAddress": "02:42:ac:12:00:03",

"IPv4Address": "172.18.0.3/16",

"IPv6Address": ""

},

"7f533dbcafc5c9235960601a8ce31c4a5db5a7d27533592232e3b18307458b57": {

"Name": "mysql",

"EndpointID": "692633f43fc0ae05c4a5f09b446f40265f952665ed4feba4d36bc88a42e5d555",

"MacAddress": "02:42:ac:12:00:02",

"IPv4Address": "172.18.0.2/16",

"IPv6Address": ""

}

Para poder llegar a esta red **172.18.0.0/16,** desde nuestro red de la empresa necesitamos enrutarla, para ello realizaremos los siguientes pasos, en nuestro servidor docker Engine habilitamos Forwarding:

vi /usr/lib/sysctl.d/50-default.conf

net.ipv4.ip_forward = 1

Guardamos los cambios y ejecutamos:

/sbin/sysctl -p

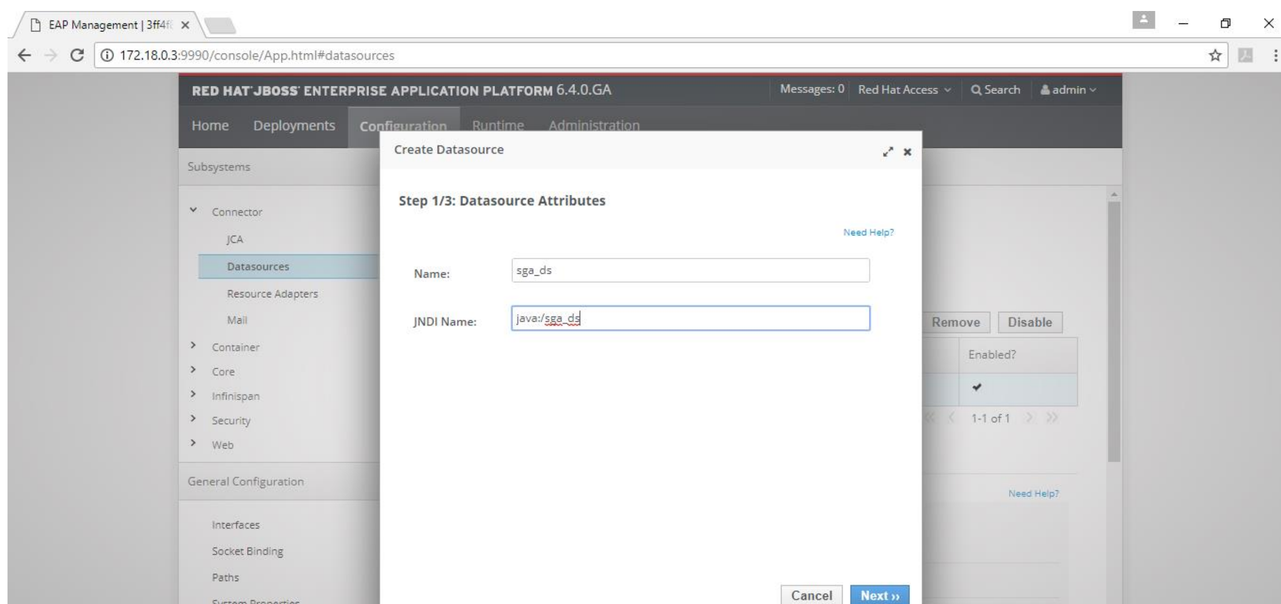
A continuacion desde nuestro equipo Windows que esta en la red local, abrimos una comsola CMD, **con permisos de Administrador** y ejecutamos:

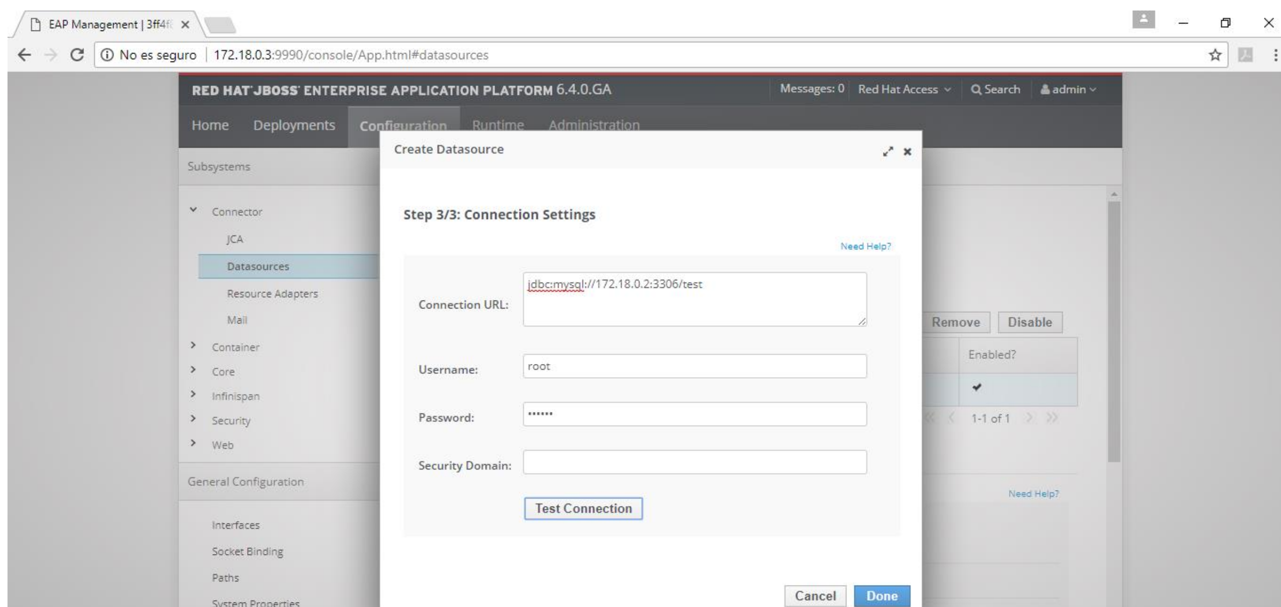
route add 172.18.0.0 mask 255.255.0.0 192.168.1.150 metric 1

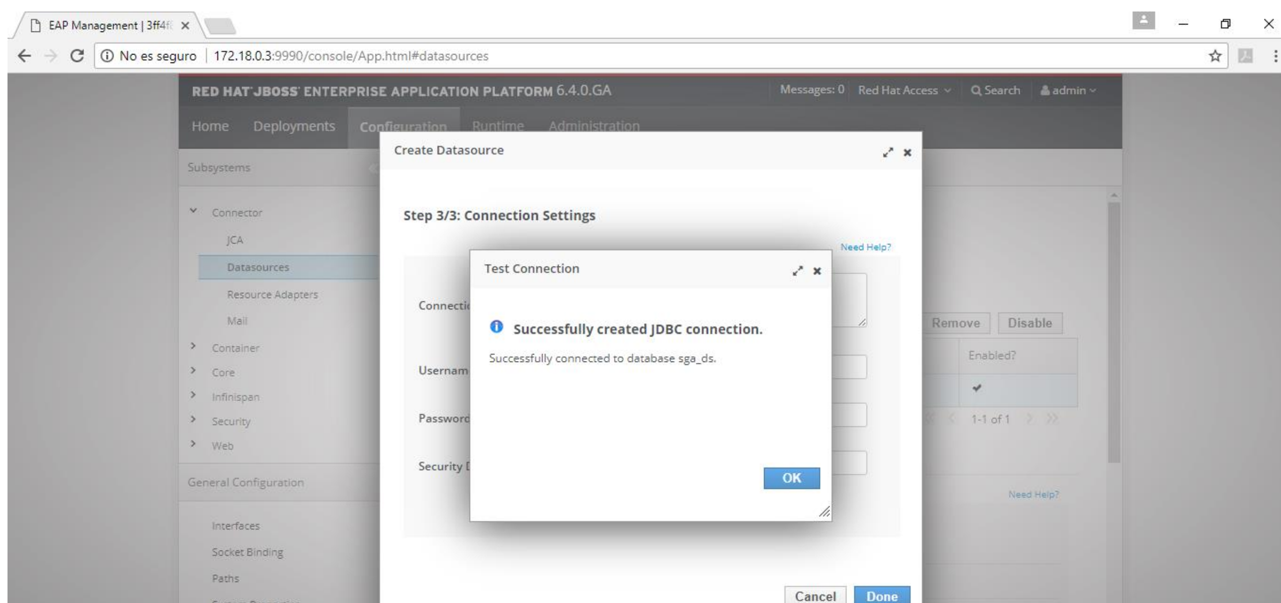
```
Administrador: C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.14393]
(c) 2016 Microsoft Corporation. Todos los derechos reservados.

C:\Windows\system32>route add 172.18.0.0 mask 255.255.0.0 192.168.1.150 metric 1
 Correcto

C:\Windows\system32>ping 172.18.0.3

Haciendo ping a 172.18.0.3 con 32 bytes de datos:
Respuesta desde 192.168.1.2: Host de destino inaccesible.
Respuesta desde 172.18.0.3: bytes=32 tiempo=2ms TTL=63
Respuesta desde 172.18.0.3: bytes=32 tiempo=2ms TTL=63
Respuesta desde 172.18.0.3: bytes=32 tiempo=8ms TTL=63

Estadísticas de ping para 172.18.0.3:
    Paquetes: enviados = 4, recibidos = 4, perdidos = 0
    (0% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    Mínimo = 2ms, Máximo = 8ms, Media = 4ms

C:\Windows\system32>
```

Ahora ya nos podemos conectar a nuestro servidor de aplicaciones a la consola de administracion y comenzar a crear el datasource y el despliegue de la aplicación SGAJBoss.

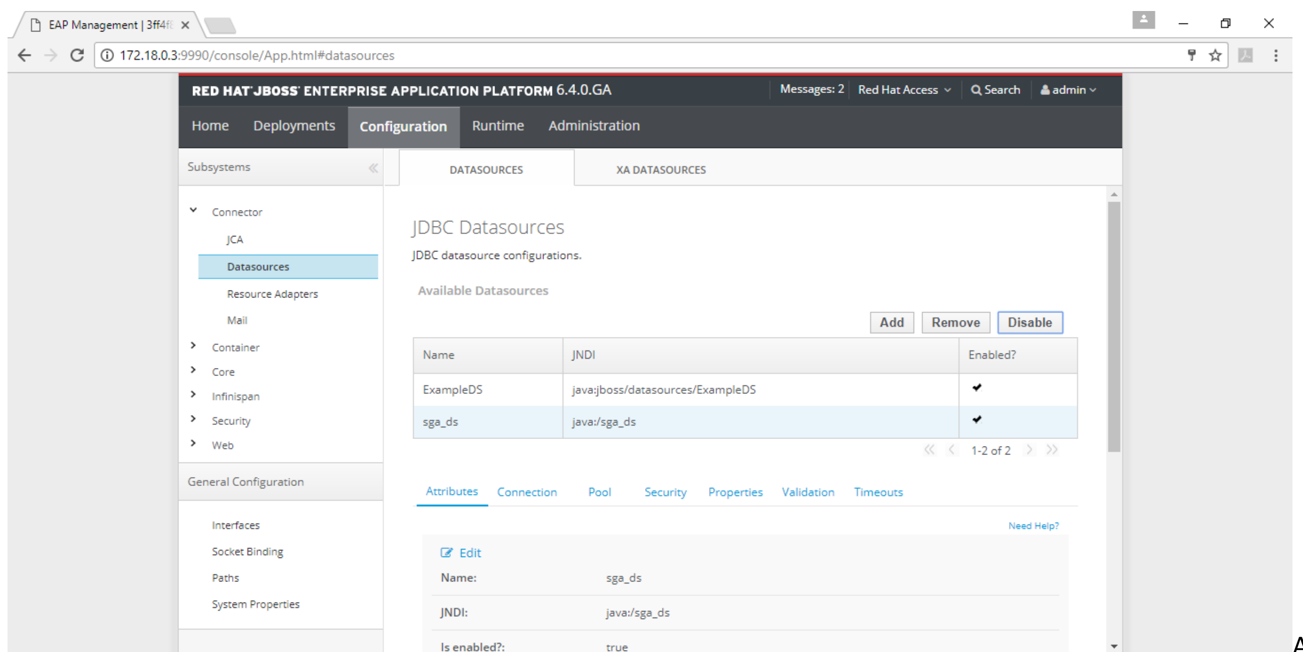
http://172.18.0.3:9990/console/App.html


EAP Management | 3ff4f
172.18.0.3:9990/console/App.html#datasources
RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM 6.4.0.GA
Messages: 0
Red Hat Access
Search
admin
Home
Deployments
Configuration
Runtime
Administration
Subsystems
Connector
JCA
Datasources
Resource Adapters
Mail
Container
Core
Infinispan
Security
Web
General Configuration
Interfaces
Socket Binding
Paths
System Properties
Create Datasource
Step 1/3: Datasource Attributes
Need Help?
Name:
sga_ds
JNDI Name:
java:/sga_ds
Remove
Disable
Enabled?
1-1 of 1
Cancel
Next ››

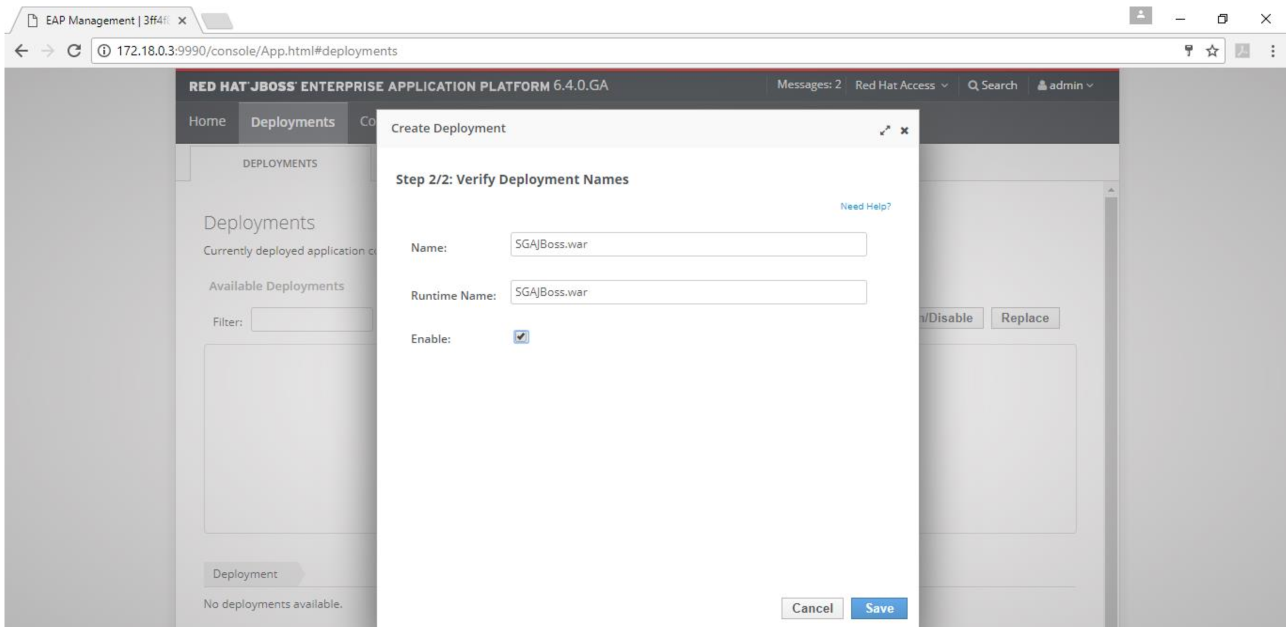

EAP Management | 3ff4f
No es seguro
172.18.0.3:9990/console/App.html#datasources
RED HAT JBOSS ENTERPRISE APPLICATION PLATFORM 6.4.0.GA
Messages: 0
Red Hat Access
Search
admin
Home
Deployments
Configuration
Runtime
Administration
Subsystems
Connector
JCA
Datasources
Resource Adapters
Mail
Container
Core
Infinispan
Security
Web
General Configuration
Interfaces
Socket Binding
Paths
System Properties
Create Datasource
Step 3/3: Connection Settings
Need Help?
Connection URL:
jdbc:mysql://172.18.0.2:3306/test
Username:
root
Password:
Security Domain:
Test Connection
Remove
Disable
Enabled?
1-1 of 1
Cancel
Done

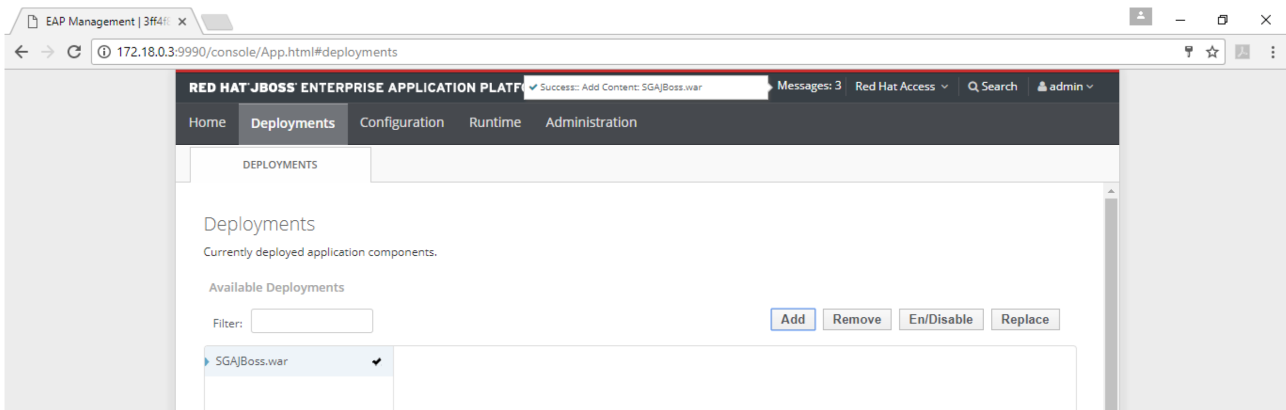


Activamos la nueva conexión JDBC



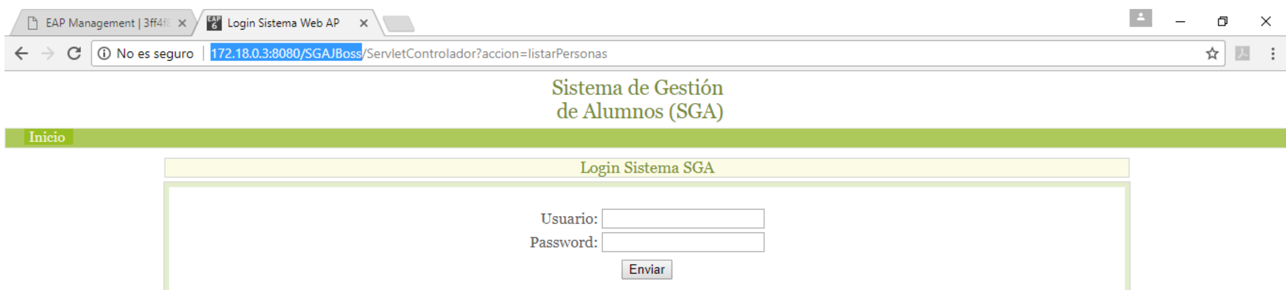
A continuacion realizamos el despliegue de la aplicación:





Ahora acedemos a la aplicación y nos validamos con el usuario admin y password admin:

http://172.18.0.3:8080/SGAJBoss



Ahora podremos agrear usuarios al la base de datos a traves de la aplicación:

Ahora hacemos pasar la aplicacione SGAJBoss, a traves del modulo mod_jk de nuestro apache, (el instructor guiara toda la configuracion).

http://aplicaciones.miempresa.com/SGAJBoss/

Ahora podemos escalar el servicio de Jboss a traves de **docker-compose** y podriamos añadirlo a nuestros frontales Web, altamete recomendado utilizar el modulo de Jboss mod_cluster:

[root@docker docker-jboss-eap-master]# **docker-compose scale jboss=4**

```
Starting dockerjbosseapmaster_jboss_1 ... done
Creating dockerjbosseapmaster_jboss_2 ...
Creating dockerjbosseapmaster_jboss_3 ...
Creating dockerjbosseapmaster_jboss_4 ...
Creating dockerjbosseapmaster_jboss_2 ... done
Creating dockerjbosseapmaster_jboss_3 ... done
Creating dockerjbosseapmaster_jboss_4 ... done
```

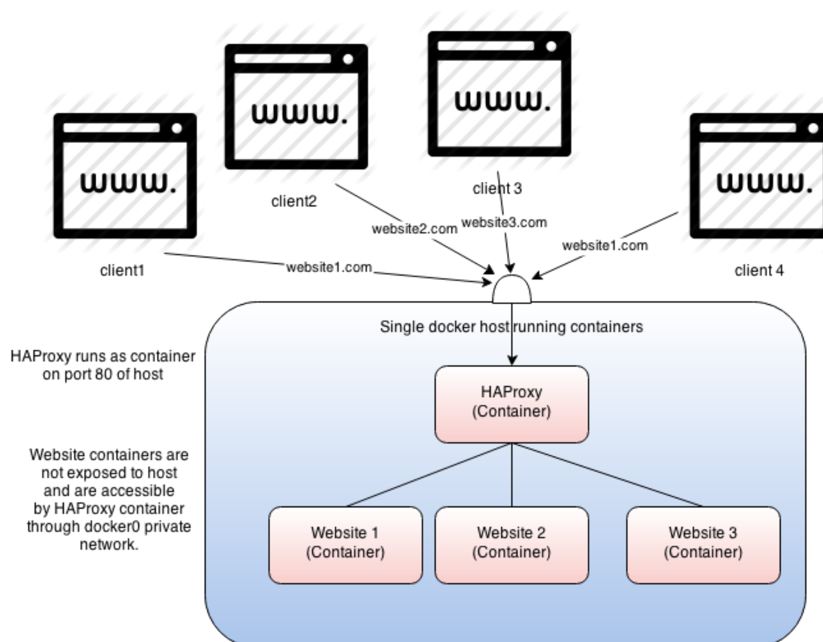
[root@docker docker-jboss-eap-master]# **docker-compose ps**

```
      Name                  Command              State            Ports
--------------------------------------------------------------------------------------------------------------
dockerjbosseapmaster_jboss_1   /bin/sh -c $JBOSS_HOME/bin ...   Up      8009/tcp, 8080/tcp, 9990/tcp, 9999/tcp
dockerjbosseapmaster_jboss_2   /bin/sh -c $JBOSS_HOME/bin ...   Up      8009/tcp, 8080/tcp, 9990/tcp, 9999/tcp
dockerjbosseapmaster_jboss_3   /bin/sh -c $JBOSS_HOME/bin ...   Up      8009/tcp, 8080/tcp, 9990/tcp, 9999/tcp
dockerjbosseapmaster_jboss_4   /bin/sh -c $JBOSS_HOME/bin ...   Up      8009/tcp, 8080/tcp, 9990/tcp, 9999/tcp
mysql                   /entrypoint.sh mysqld          Up      0.0.0.0:3306->3306/tcp, 33060/tcp
```

Ahora podriamos averiguar el direcionamiento ip de los contenedores y pasarlos a traves de nuestros balanceadores:

[root@docker docker-jboss-eap-master]# **docker network inspect dockerjbosseapmaster_default**

Laboratorio Confiuracion contenedores docker a traves de balanceador haproxy



HAProxy es un balanceador TCP que se suele usar para peticiones HTTP pero se puede usar para cualquier protocolo TCP. Vamos a ver como configurarlo:

Con HAPROXY podemos montar un servidor frontal que proporcione balanceo de carga en el acceso a nuestros servidores web.

Si queremos redimensionar nuestra web, podríamos tener instalados varios servidores web encargados de gestionar nuestro dominio y como frontend de los mismos un equipo con haproxy instalado encargado de repartir las peticiones entre los distintos servidores web.

En este laboratorio configuraremos un balanceador por software (haproxy), el cual nos permitirá balancear peticiones hacia nuestros contenedores httpd, dependiendo del dominio del que venga en el cliente.

Tendremos el dominio **www.miempresa.com**, el cual nos tiene que balancear a tres contenedores que servirán la web del dominio.

Tendremos el domino **aplicaciones.miempresa.com**, el cual servirá la web sobre dos contenedores en apache.

Para realizar este lab en los contenedores tendremos que trabajar con volúmenes persistentes y parseando el puerto apache de nuestros contenedores, con un puerto en nuestro Doceker Engine.

Lanzamos los contenedores que vamos a pasar a traves de **www.miempresa.com**, mostraran el nombre del contenedor a traves de haproxy

En **/web** esta index.php que mapeamos con en contenedor y mostrara el nombre del contenedor.

index.php

```
<?php

echo gethostname();

?>
```

Lanzamos los contenedores, mapeando los puertos locales en nuestro Docker Engine (81,82,83)

```
# docker run -dtiP --name centos6-web1 -p 81:80  -v /web:/var/www/html  docker.io/nickistre/centos-lamp

# docker run -dtiP --name centos6-web2 -p 82:80  -v /web:/var/www/html  docker.io/nickistre/centos-lamp

# docker run -dtiP --name centos6-web3 -p 83:80  -v /web:/var/www/html  docker.io/nickistre/centos-lamp
```

Lanzamos los contenedores para la web **aplicaciones.miempresa.com**, mapeando los puertos locales en nuestro Docker Engine (84,85), creamos el directorio **/web2**

index.php

```
<?php phpinfo(); ?>
```

```
#docker run -dtiP --name centos6-web4 -p 84:80 -v /web2:/var/www/html  docker.io/nickistre/centos-lamp

#docker run -dtiP --name centos6-web5 -p 85:80 -v /web2:/var/www/html  docker.io/nickistre/centos-lamp
```

Ahora configuraremos nuestro servidor Haproxy, añadimos y comprobamos lo que esta en negrita:

 # vi /etc/haproxy/haproxy.cfg

```
global
    log         127.0.0.1 local2

    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    user        haproxy
    group       haproxy
    daemon

    # turn on stats unix socket
    stats socket /var/lib/haproxy/stats

#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#---------------------------------------------------------------------
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
```

```
    option              redispatch
    retries             3
    timeout http-request    10s
    timeout queue           1m
    timeout connect         10s
    timeout client          1m
    timeout server          1m
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 3000

frontend http-in
    bind *:80
    acl is_site1 hdr_end(host) -i www.miempresa.com
    acl is_site2 hdr_end(host) -i aplicaciones.miempresa.com

    use_backend site1 if is_site1
    use_backend site2 if is_site2

#default_backend Granja_puerto_80

backend site1
        balance roundrobin
option httpclose
    option forwardfor
        server web1  127.0.0.1:81 maxconn 400 check
        server web2  127.0.0.1:82 maxconn 400 check
```

server web3 127.0.0.1:83 maxconn 400 check

backend site2

balance roundrobin

option httpclose

option forwardfor

server web4 127.0.0.1:85 maxconn 400 check

server web5 127.0.0.1:84 maxconn 400 check

 # Acceso a estadisticas de haproxy

listen stats :4443

mode http

stats enable

stats hide-version

stats realm Balanceador Haproxy\ Estadisticas

stats uri /

stats refresh 5s

stats auth admin:000000

Iniciamos el servicio haproxy:

systemctl start haproxy

¿Qué es la integración continua?

Continuous Integration

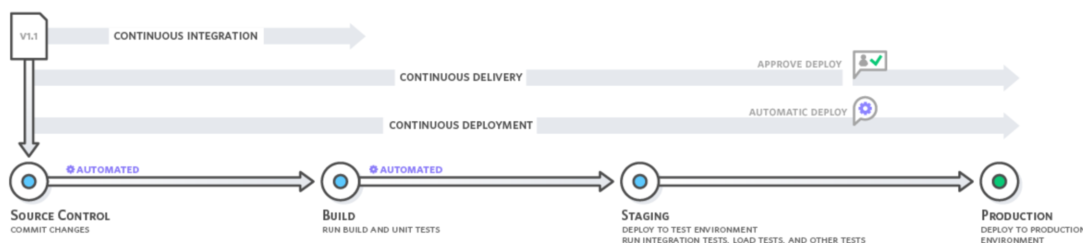
La **integración continua** es una práctica de desarrollo de software mediante la cual los desarrolladores combinan los cambios en el código en un repositorio central de forma periódica, tras lo cual se ejecutan versiones y pruebas automáticas. La integración continua se refiere en su mayoría a la fase de creación o integración del proceso de publicación de software y conlleva un componente de automatización (p. ej., CI o servicio de versiones) y un componente cultural (p. ej., aprender a integrar con frecuencia). Los objetivos clave de la integración continua consisten en encontrar y arreglar errores con mayor rapidez, mejorar la calidad del software y reducir el tiempo que se tarda en validar y publicar nuevas actualizaciones de software.

¿Por qué es necesaria la integración continua?

Anteriormente, era común que los desarrolladores de un equipo trabajasen aislados durante un largo periodo de tiempo y solo intentasen combinar los cambios en la versión maestra una vez que habían completado el trabajo. Como consecuencia, la combinación de los cambios en el código resultaba difícil y ardua, además de dar lugar a la acumulación de errores durante mucho tiempo que no se corregían. Estos factores hacían que resultase más difícil proporcionar las actualizaciones a los clientes con rapidez.

¿En qué consiste la integración continua?

Con la integración continua, los desarrolladores envían los cambios de forma periódica a un repositorio compartido con un sistema de control de versiones como Git. Antes de cada envío, los desarrolladores pueden elegir ejecutar pruebas de unidad local en el código como medida de verificación adicional antes de la integración. Un servicio de integración continua crea y ejecuta automáticamente pruebas de unidad en los nuevos cambios realizados en el código para identificar inmediatamente cualquier error.



La integración continua se refiere a la fase de creación y pruebas de unidad del proceso de publicación de software. Cada revisión enviada activa automáticamente la creación y las pruebas.

Con la entrega continua, se crean, prueban y preparan automáticamente los cambios en el código y se entregan para la fase de producción. La entrega continua amplía la integración continua al implementar todos los cambios en el código en un entorno de pruebas y/o de producción después de la fase de creación.

¿Qué es la entrega continua?

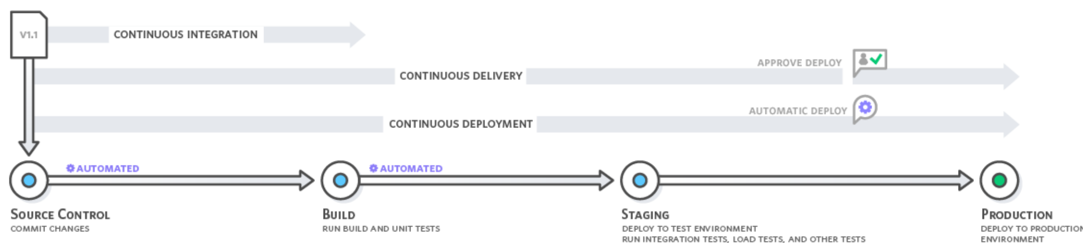
continuous delivery

La **entrega continua** es una práctica de desarrollo de software mediante la cual se preparan automáticamente los cambios en el código y se entregan a la fase de producción. Fundamental para el desarrollo de aplicaciones modernas, la entrega continua amplia la integración continua al implementar todos los cambios en el código en un entorno de pruebas o de producción después de la fase de compilación. Cuando la entrega continua se implementa de manera adecuada, los desarrolladores dispondrán siempre de un artefacto listo para su implementación que se ha sometido a un proceso de pruebas estandarizado.

La entrega continua permite a los desarrolladores automatizar las pruebas más allá de las pruebas de unidades, por lo que pueden verificar actualizaciones en las aplicaciones en varias dimensiones antes de enviarlas a los clientes. Las pruebas pueden incluir pruebas de la UI, de carga, de integración, de fiabilidad de la API, etc. De este modo, los desarrolladores pueden validar las actualizaciones de forma más exhaustiva y descubrir problemas por anticipado. Con la nube, resulta sencillo y rentable automatizar la creación y replicación de varios entornos de pruebas, algo que anteriormente era complicado en las instalaciones.

Entrega continua e implementación continua

Con la entrega continua, todos los cambios en el código se crean, se prueban y se envían a un entorno de almacenamiento o pruebas de no producción. Pueden efectuarse varias pruebas al mismo tiempo antes de la implementación en producción. La diferencia entre la entrega continua y la implementación continua es la diferencia de aprobación manual para actualizar la producción. Con la implementación continua, la producción tiene lugar de manera automática, sin aprobación explícita.



La entrega continua automatiza todo el proceso de publicación de software. Cada revisión efectuada activa un proceso automatizado que crea, prueba y almacena la actualización. La decisión definitiva de implementarla en un entorno de producción en vivo la toma el desarrollador.

Continuous Deployment

Continuous Deployment o despliegue continuo significa que cada cambio pasa por un “pipeline” automatizado y se pone en producción, lo que suele generar sitios donde hay muchos despliegues en producción cada día.

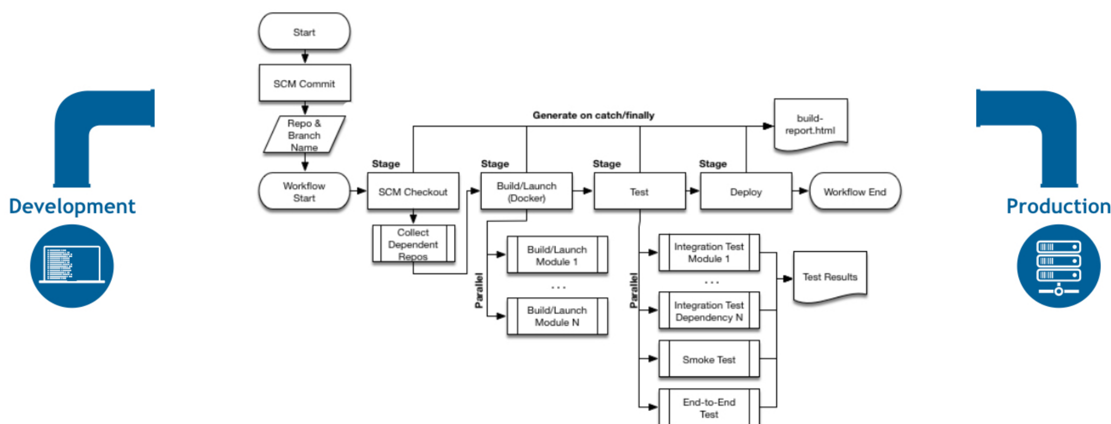
Para tener Continuous Deployment antes debe haber Continuous Delivery y en cualquiera de los casos debe haber una buena integración continua

Pipelines: definición

Un pipeline es una forma de trabajar en el mundo devops en la integración continua.

Utilizando pipeline y Jenkins, podemos definir el ciclo de vida completo de una aplicación (descargar código, compilar, test, desplegar, etc.) mediante código.

De esta forma, resulta mucho más sencillo replicar los diferentes pasos con distintas aplicaciones y gestionar mejor los cambios en cada paso.



Jenkins

https://es.wikipedia.org/wiki/Jenkins

Jenkins es un servidor de automatización open source escrito en Java. Está basado en el proyecto Hudson y es, dependiendo de la visión, un fork del proyecto o simplemente un cambio de nombre.

Jenkins ayuda en la automatización de parte del proceso de desarrollo de software mediante integración continua y facilita ciertos aspectos de la entrega continua. Admite herramientas de control de versiones como CVS, Subversion, Git, Mercurial, Perforce y Clearcase y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como secuencias de comandos de consola y programas por lotes de Windows.

Laboratorio Jenkins Docker

En este laboratorio instalaremos un contenedor a través de docker-compose, con jenkins, y lo integraremos con un contenedor basado en centos que construiremos durante la ejecución de docker-compose, a continuación, veremos como podemos integrar dicho contenedor a través de jenkins via ssh, para simular CI/CD

Dockerfile

```
FROM centos

RUN yum -y install openssh-server

RUN useradd remote_user && \
    echo "1234" | passwd remote_user  --stdin && \
    mkdir /home/remote_user/.ssh && \
    chmod 700 /home/remote_user/.ssh

COPY remote-key.pub /home/remote_user/.ssh/authorized_keys

RUN chown remote_user:remote_user   -R /home/remote_user && \
    chmod 600 /home/remote_user/.ssh/authorized_keys

RUN /usr/sbin/sshd-keygen > /dev/null 2>&1

CMD /usr/sbin/sshd -D
```

docker-compose.yml

```
version: '3'
services:
  jenkins:
    container_name: jenkins
    image: jenkins/jenkins
    ports:
      - "8080:8080"
    volumes:
      - $PWD/jenkins_home:/var/jenkins_home
    networks:
      - net
  remote_host:
    container_name: remote-host
    image: remote-host
    build:
      context: centos7
    networks:
      - net
networks:
  net:
```

En el material del curso, tendremos la carpeta **Laboratorio jenkins-docker**, con todos los archivos de configuración para realizar el lab.

Para la creación de las llaves ssh que están en el directorio centos7, se ha utilizado el comando:

#ssh-keygen -f remote-key

Comenzamos el laboratorio:

#cd /Laboratorio jenkins-docker

docker-compose buil

Verificar permisos en la capeta del **volumen jenkins_home**, porque sino no arrancara nuestro Jenkins, es debido a que la imagen de Jenkins arranca con el usuario jenkins con el UID 1000:

#chown -R 1000 jenkins_home

#chown -R 1755 jenkins_home

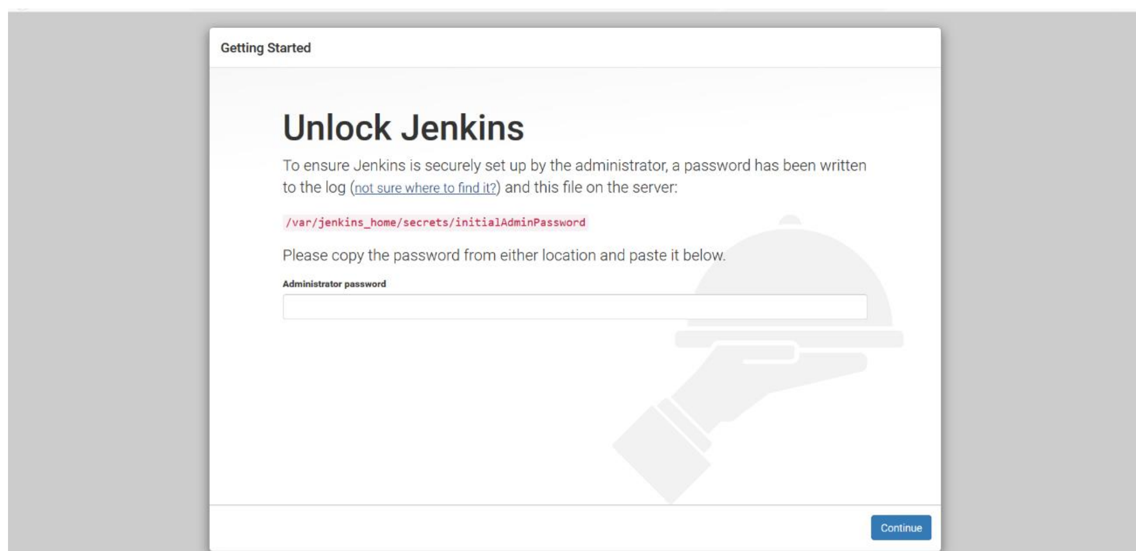
docker-compose up

http://192.168.1.150:8080/

Para obtener el password inicial:

cat jenkins_home/secrets/initialAdminPassword

b3871de8d6274815886f15f82d2956fa



Install suggested plugins

Customize Jenkins

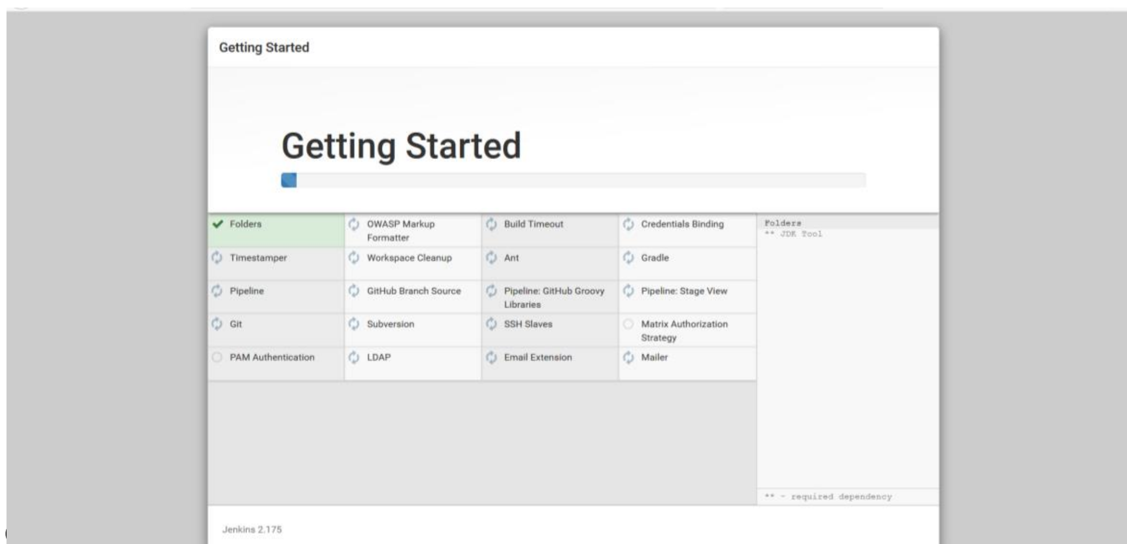
Plugins extend Jenkins with additional features to support many different needs.

Install suggested plugins

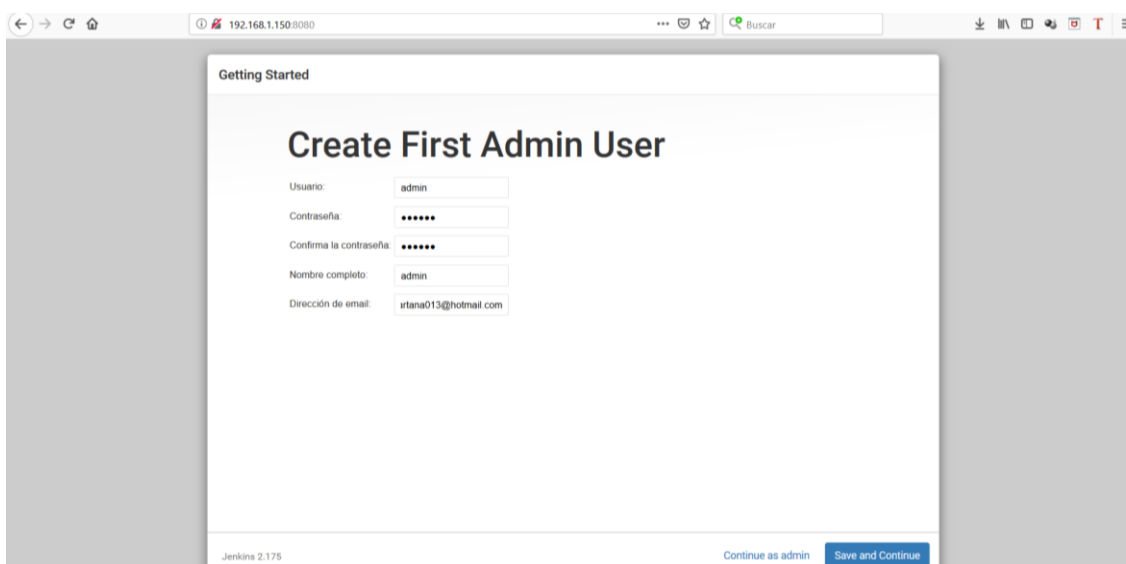
Install plugins the Jenkins community finds most useful.

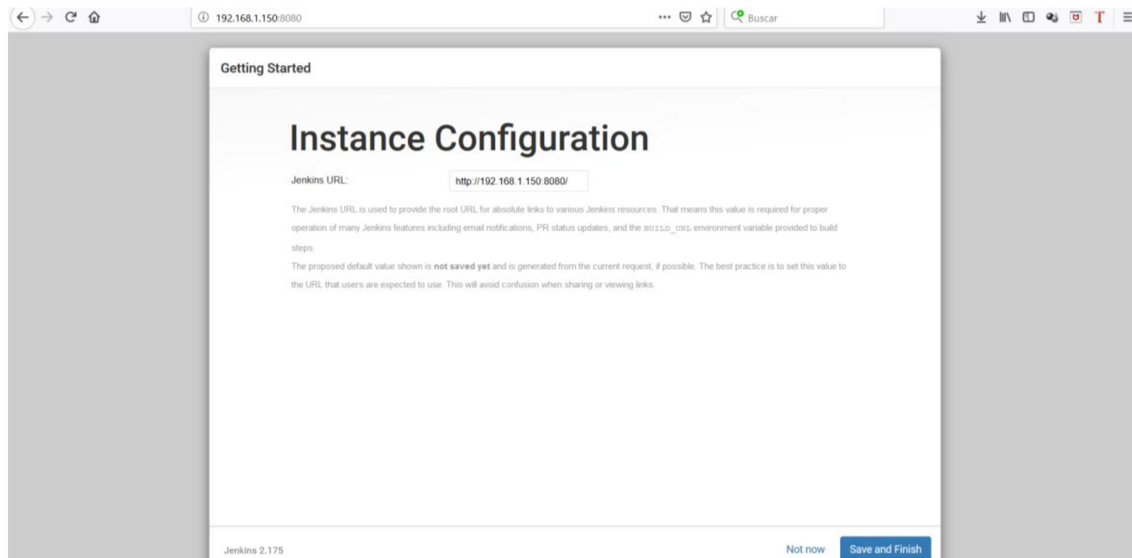
Select plugins to install

Select and install plugins most suitable for your needs.



Creamos la cuenta inicial del Admin:





Ahora configuramos nuestro Jenkins para conectarse al hosts que acabamos de crear en Docker **remote-host**:

#cd /Laboratorio jenkins-docker

docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

0f2d878c50bf jenkins/jenkins "/sbin/tini -- /usr/…" 10 minutes ago Up 10 minutes 0.0.0.0:8080->8080/tcp, 50000/tcp jenkins

c055efce3624 remote-host "/bin/sh -c '/usr/sb…" 10 minutes ago Up 10 minutes remote-host

En administrar Jenkins

Ahora instalamos el **pluguin** de **ssh**, en **Administrar Pluguins**:

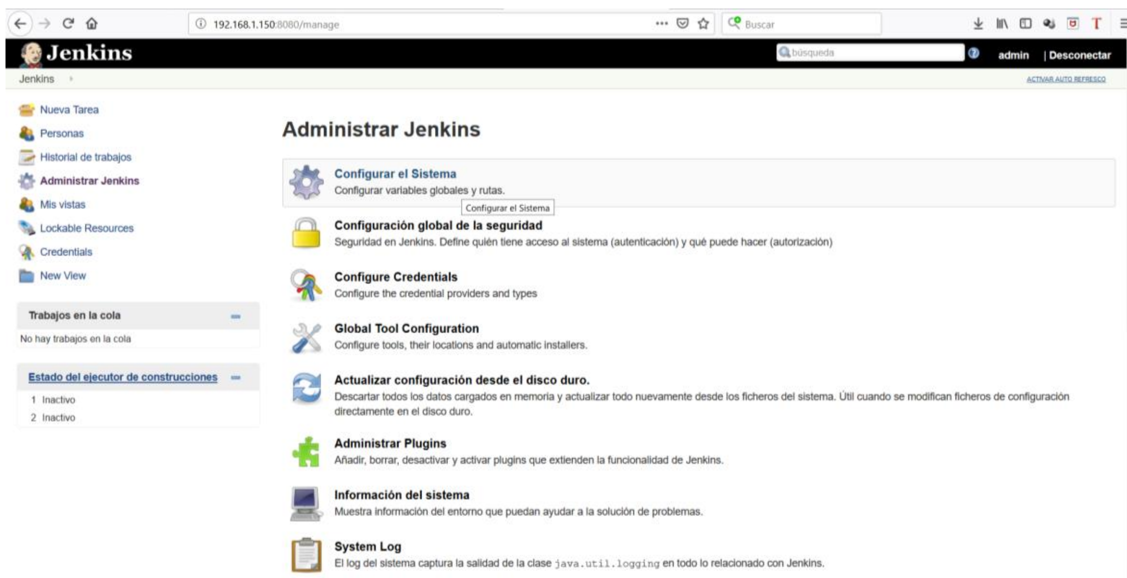
En Todos los **pluguins** filtraremos por **ssh** y lo instalamos

Seleccionamos instalar sin reiniciar

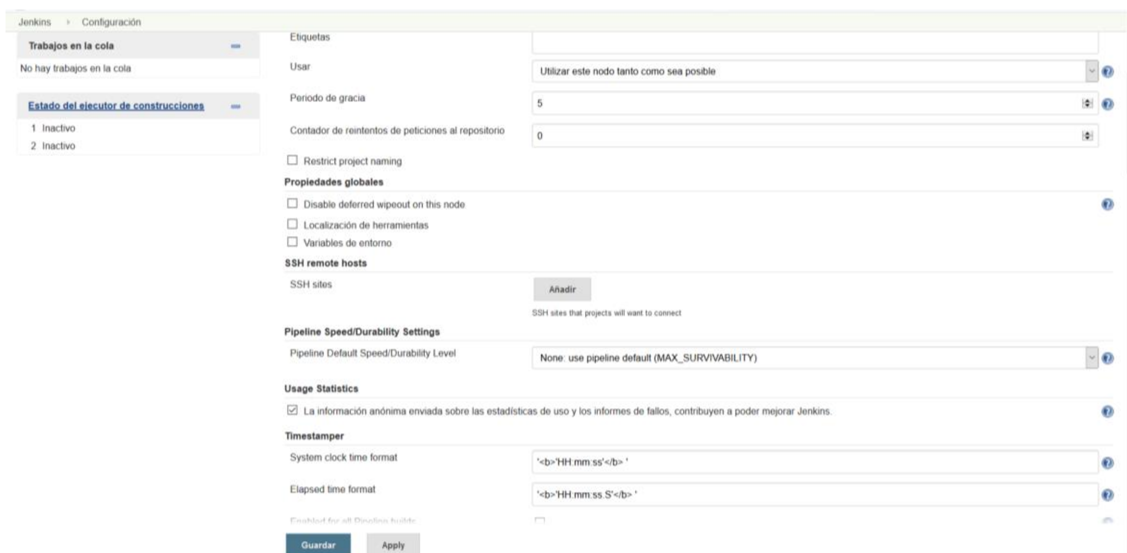
Seleccionamos Reiniciar Jenkins cuando termine la instalación y no queden trabajos en ejecución:

Ahora en el **menú Jenkins**, administrar **Jenkins-Administrar Plugins-Plugins** instalados, si filtramos por ssh veremos el plugin instalado.

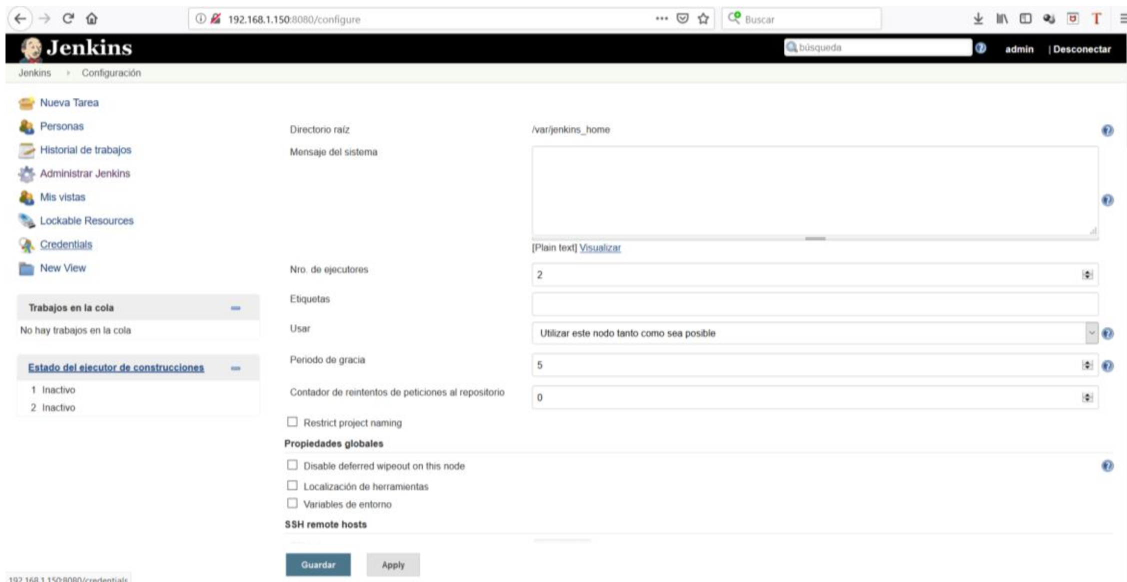
Ahora en configuración del sistema:



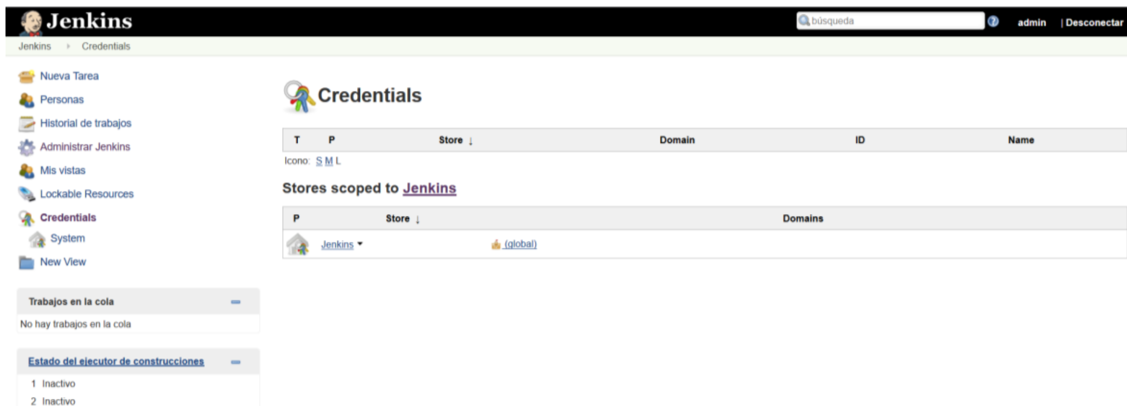
En ssh remote hosts:



Primero configuraremos las credenciales, antes de configurar la conexión ssh:



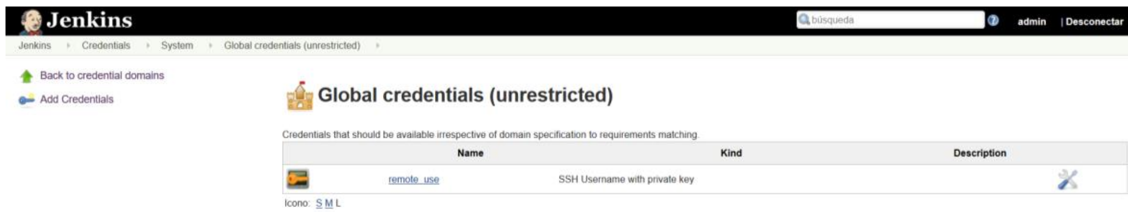
Seleccionamos Jenkins:



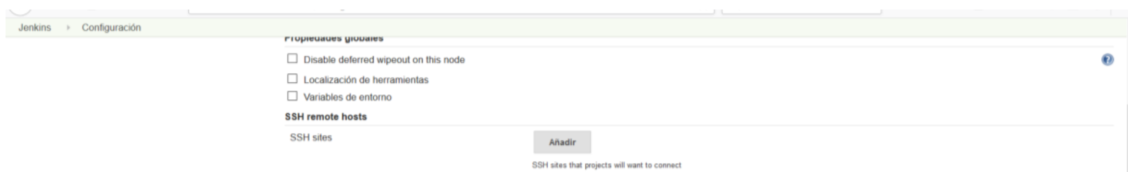
Global credentials:

Y seleccionamos Add credentials:

Creamos la configuración, el usuario es de docker-compose de la imagen de centos, pegamos el contenido de la llave privada **remote-key**, para poder conectarnos al contenedor remote-host:



Ahora nos vamos a **Jenkis-Administrar Jenkis-Configurar** el sistema y en SSH remote hosts, establecemos la conexión:



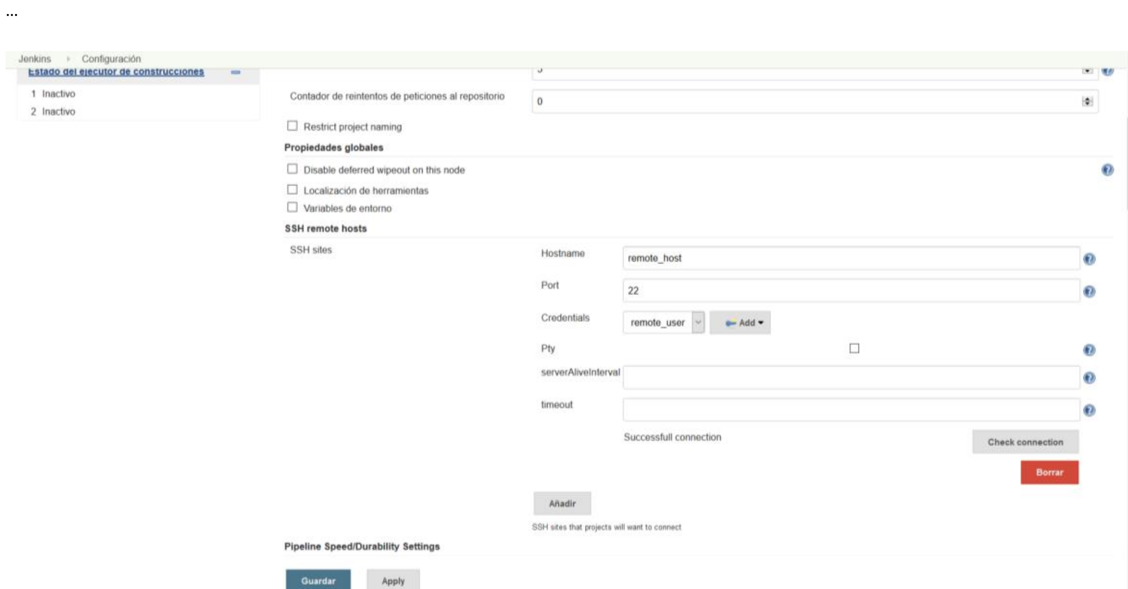
Seleccionamos el hostname que es **remote_host**, es el servicio declarado en docker-compose, puerto 22 y le damos las credenciales, creadas anteriormente, le daremos a **Check connection**, y nos tiene que dar Successful….. finalizaremos dándole a guardar:

remote_host:

container_name: remote-host

image: remote-host

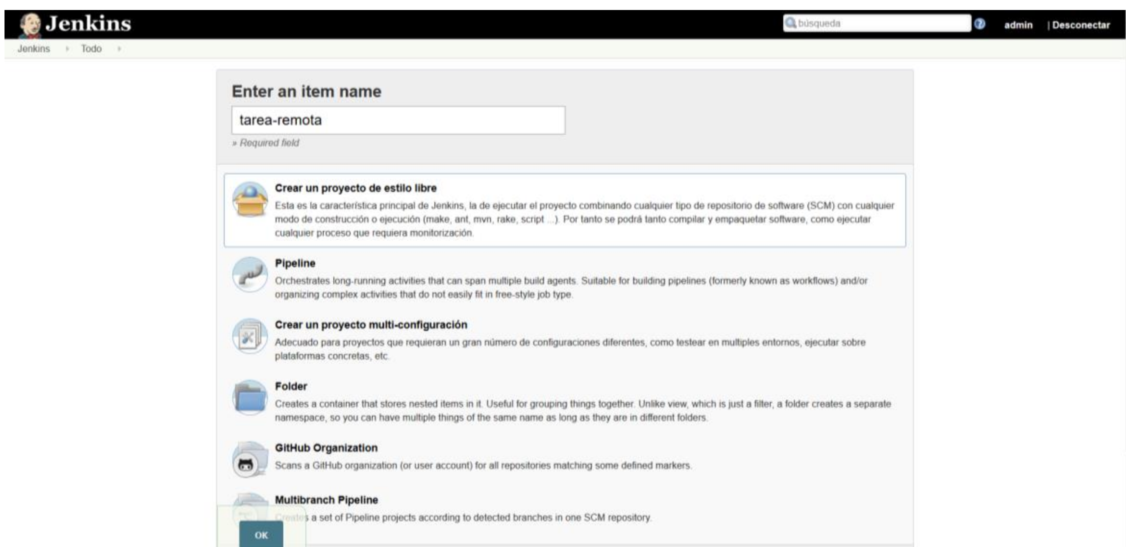
…



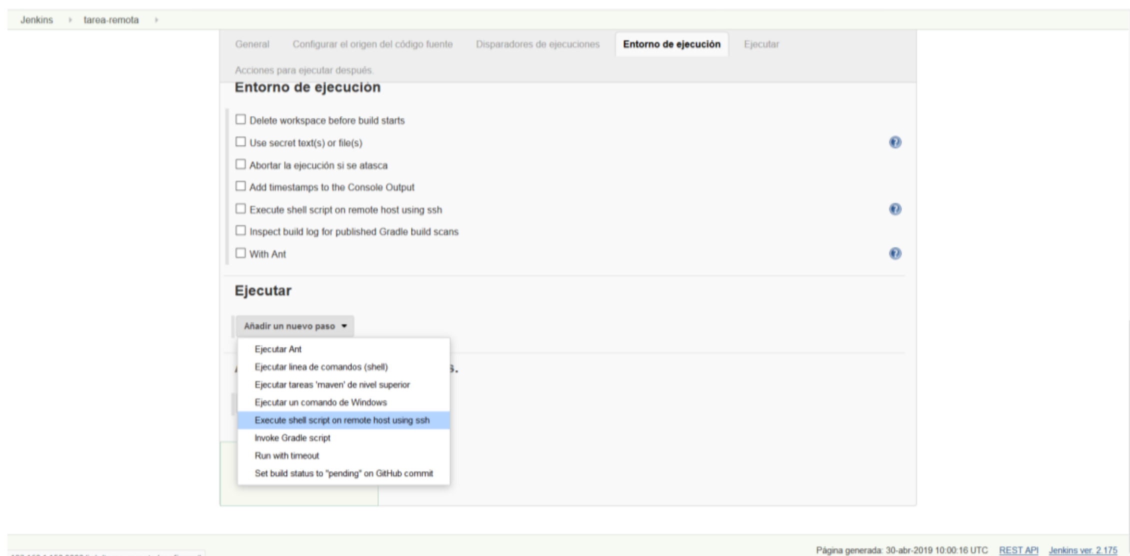
Ejecutar un Job en un host remoto vía SSH

Nos creamos un nuevo (job), Nueva Tarea:

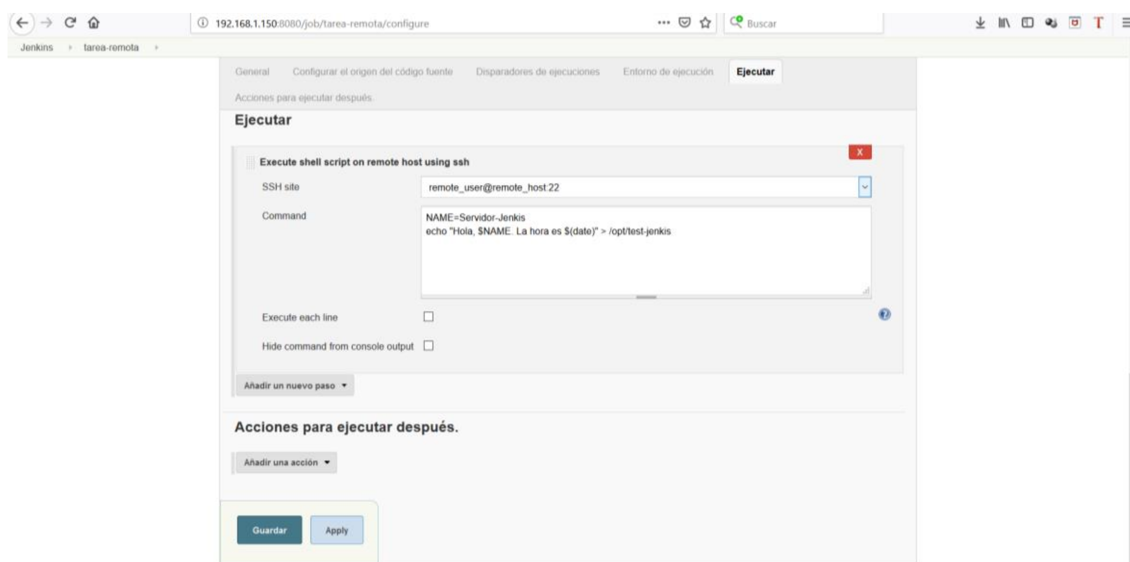
Creamos un proyecto de estilo libre y le damos a ok:



Seleccionamos en **Ejecutar-Ejecutar Shell script on remote host using ssh:**



Añadiríamos este comando y le damos a guardar



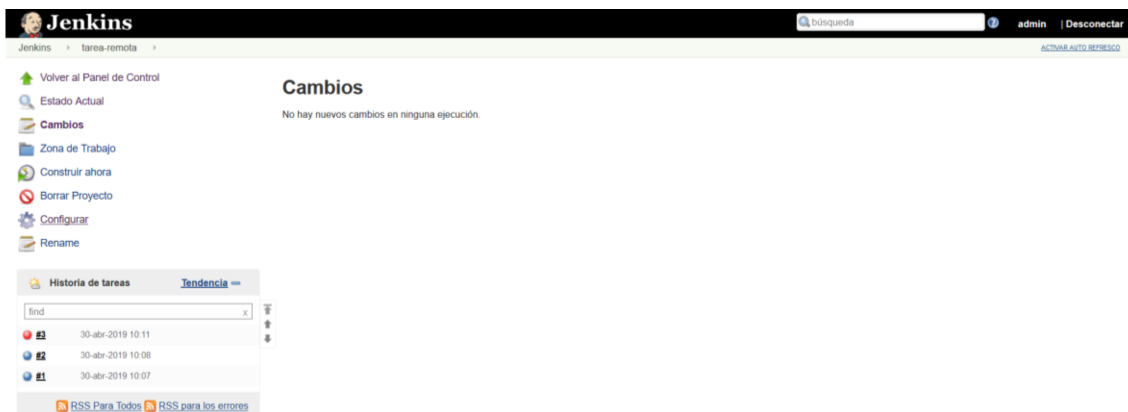
Ahora construiremos nuestro Job, sobre la maquina remota:

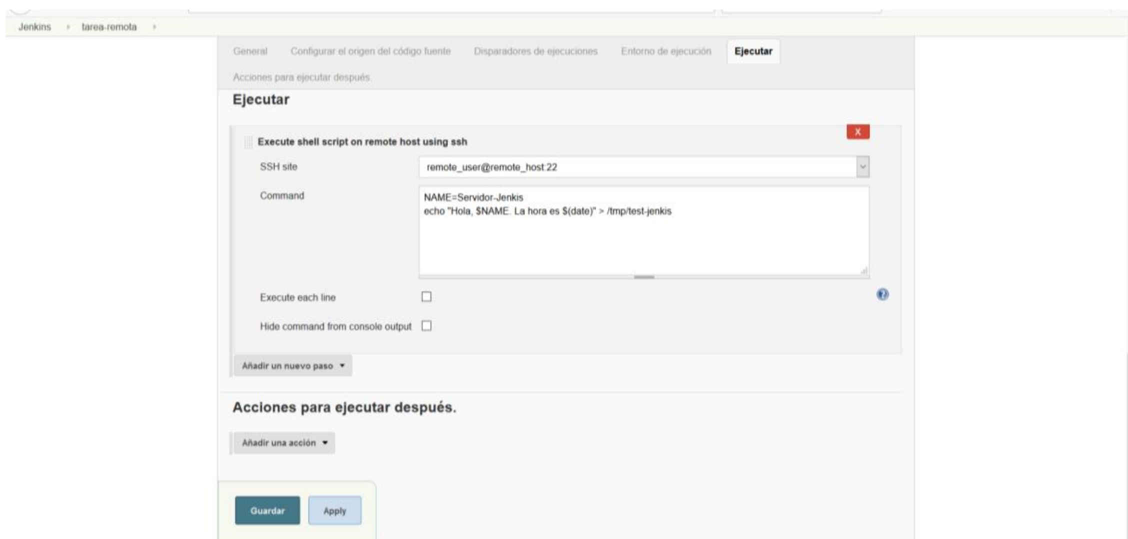
En Construir ahora:

Veremos que falla, ¿Por qué?....., veremos en historial de tareas la bola en color rojo…

Si vemos el error, nos dice que no tenemos permisos para escribir en opt, recordar que la tarea la estamos haciendo con el usuario **remote_user**

Si ahora seleccionamos configurar, lo dejamos en el directorio /tmp:





Si ahora seleccionamos **Construir** ahora, veré que la tarea se ejecutado sobre el contenedor a trabes de un job de jenkins:

Podemos acceder al contenedor **remote-host** y ver que se ha creado en el directorio /tmp/test-jenkis

#docker exec -ti remote-host /bin/bash

ls -l /tmp/

total 8

-rw-rw-r-- 1 remote_user remote_user 63 Apr 8 10:17 test-jenkis

Laboratorio Informacion del Sistema

```
[root@docker ~]#docker system

Usage: docker system COMMAND

Manage Docker

Options:
      --help   Muestra su uso

Commands:
  df          Muestra el disco usado por Docker
  events      Muestra los eventos del servidor Docker en tiempo real
  info        Muestra información del todo sistema
  prune       Elimina datos no usados

Run 'docker system COMMAND --help' for more information on a command.
```

Mostrar espacio de disco usado por Docker

Muestra el espacio usado por Docker, detallando cuánto es ocupado por las imágenes de Docker, cuánto por los contenedores y poor los volúmenes. Incluye el número total de elementos y los activos actualmente.

```
[root@docker ~]#  docker system df
TYPE                TOTAL               ACTIVE              SIZE                RECLAIMABLE
Images              4                   3                   248.4 MB            15.49 MB (6%)
Containers          10                  0                   11 B                11 B (100%)
Local Volumes       20                  0                   625.6 MB            625.6 MB
(100%)
```

Mostrar los eventos de Docker en tiempo real

Muestra en tiempo real los eventos que se ocurren en la instalación de Docker. En este caso un extracto tras la ejecución de **docker system prune**

```
[root@docker ~]#docker system events
2017-05-16T11:06:55.312285709+02:00 container destroy
18a5b622035f92456711e50ee5bf417050b4478c81437444264b87643e55a97a (image=hello-world,
name=prueba)
2017-05-16T11:06:55.273533792+02:00 container destroy
dc41ea7b360599745a2c812f05f2b846b65c697f30db5a43b1e3d213c861e377 (image=hello-world,
name=dreamy_cray)
2017-05-16T11:06:55.351868008+02:00 container destroy
020cece2c065de3bacf1d25da5170d5bb34da4243f6bc1172575c9988f019f36 (image=nginx,
name=modest_shockley)
(...)
```

Mostrar información de la instalación de Docker

Con la ejecución de este comando podemos ver toda la información del sistema Docker.

Muestra información variada de Docker (número de contenedores y imágenes, versión de Docker, número de nodos de Swarm...) así como información propia del sistema host (sistema operativo, versión del kérnel, CPU's, memoria RAM...)

```
[root@docker ~]#docker info
Containers: 10
 Running: 0
 Paused: 0
 Stopped: 10
Images: 4
Server Version: 17.03.1-ce
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: xfs
 Dirs: 35
 Dirperm1 Supported: true
Logging Driver:
json-file Cgroup
Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: active
 NodeID: 3xwj9vrt255deqistbwcebdk2
 Is Manager: true
 ClusterID: zur9v9u86u1jwzfr1bg5483gj
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
   Election
    Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
Default Runtime: runc
(…)
```

Eliminar elementos que no estén en uso

Unifica todas las opciones prune de contenedores, volúmenes, redes e imágenes, borrando todas aquellas que no estén en uso.

```
[root@docker ~]#docker system prune
WARNING! This will remove:
      - all stopped containers
      - all volumes not used by at least one container
      - all networks not used by at least one container
      - all dangling images
Are you sure you want to continue? [y/N] y Deleted Containers:
dc41ea7b360599745a2c812f05f2b846b65c697f30db5a43b1e3d213c861e377
18a5b622035f92456711e50ee5bf417050b4478c81437444264b87643e55a97a
020cece2c065de3bacf1d25da5170d5bb34da4243f6bc1172575c9988
f019f36 (...)

Deleted Volumes:
04a60c7c7125f72e35b138da4af3a1634e9262caf92ccb6a06cdab7e9ac7aa7c
66c142b61eb1376e7a55f9c35c60ee6e3356880869d2cc42894a8c99634fb75c
8967c318cf59bb1a9435330bb371af0c9f16abf7c6af1c3abe61ac403
7d54d61 (...)

Deleted Networks:
prueba

Total reclaimed space: 625.6 MB
```

```
[root@docker ~]#  docker system df
TYPE                TOTAL               ACTIVE              SIZE                RECLAIMABLE
Images              4                   0                   248.4 MB            248.4 MB
(100%)
Containers          0                   0                   0 B                 0 B
Local Volumes       0                   0                   0 B                 0 B
```

Instalamos portainer

https://portainer.readthedocs.io/en/stable/deployment.html

Portainer es una interfaz de usuario de administración ligera que le permite administrar fácilmente su host Docker o en un clúster de Swarm.

Portainer está destinado a ser tan simple de implementar como es para usar. Consiste en un único contenedor que puede ejecutarse en cualquier motor Docker (Docker para Linux y Docker para Windows son compatibles).

¡Portainer le permite administrar sus contenedores, imágenes, volúmenes, redes..Es compatible con el motor Docker Engine y con Docker Swarm.

Lanzamos portainer en un docker engine:

#docker volume create --name portainer_data

#docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer

Sin embargo, si queremos hacerlo en un Docker Swarm, lo hacemos creando un servicio con el comando:

```
 # docker service create \
--name portainer \
--publish 9000:9000 \
--constraint 'node.role == manager' \
--mount type=bind,src=//var/run/docker.sock,dst=/var/run/docker.sock portainer/portainer -H unix:///var/run/docker.sock
```

Si en la versión 1.20 de portainer obtenemos este error:

Unable to query endpoint (err=Endpoint is down) (code=503)

Tendremos que utilizar esta imagen de portainer:

portainerci/portainer:fix2556-frequent-offline-mode

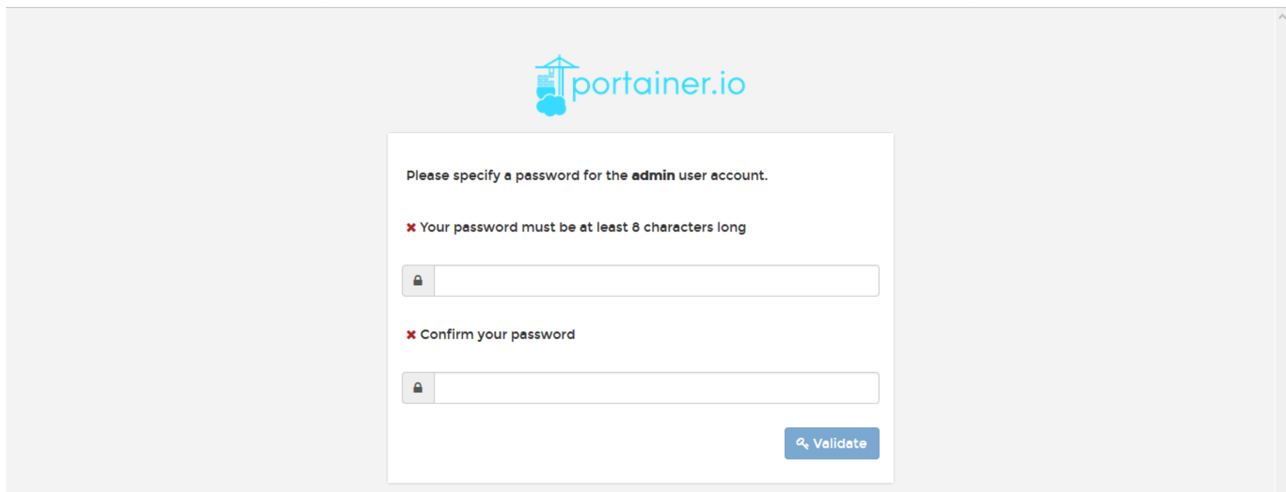
#docker service create --name portainer --publish 9000:9000 --constraint 'node.role == manager' --mount type=bind,src=//var/run/docker.sock,dst=/var/run/docker.sock **portainerci/portainer:fix2556-frequent-offline-mode** -H unix:///var/run/docker.sock

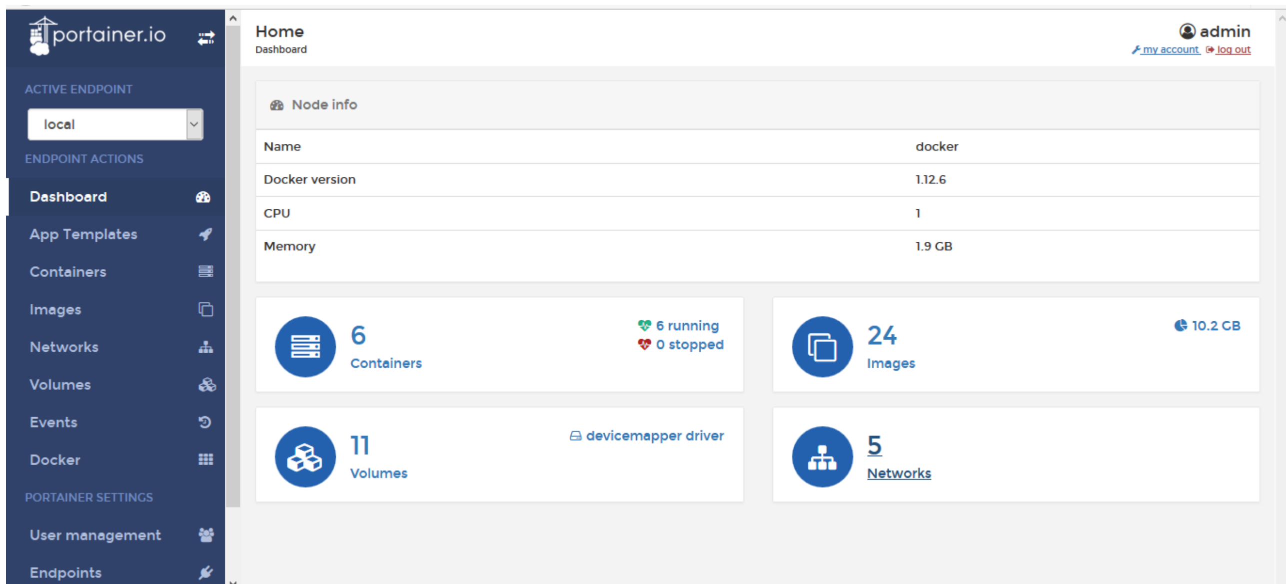
Creará un servicio llamado portainer, se deba ejecutar en un nodo que cumpla las funciones de manager, puentea el puerto 9000 del servicio al 9000 del nodo, monte el socket de Docker para que puedan ejecutarse comando dentro de él y lo hace con la imagen portainer/portainer.

Existen otras herramientas más complejas y con más funcionalidades que Portainer, como es el caso de Rancher, una potente aplicación que permite incluso gestionar nodos en otras plataformas como AWS o DigitalOcean.

Accedemos a nuestro portainer:

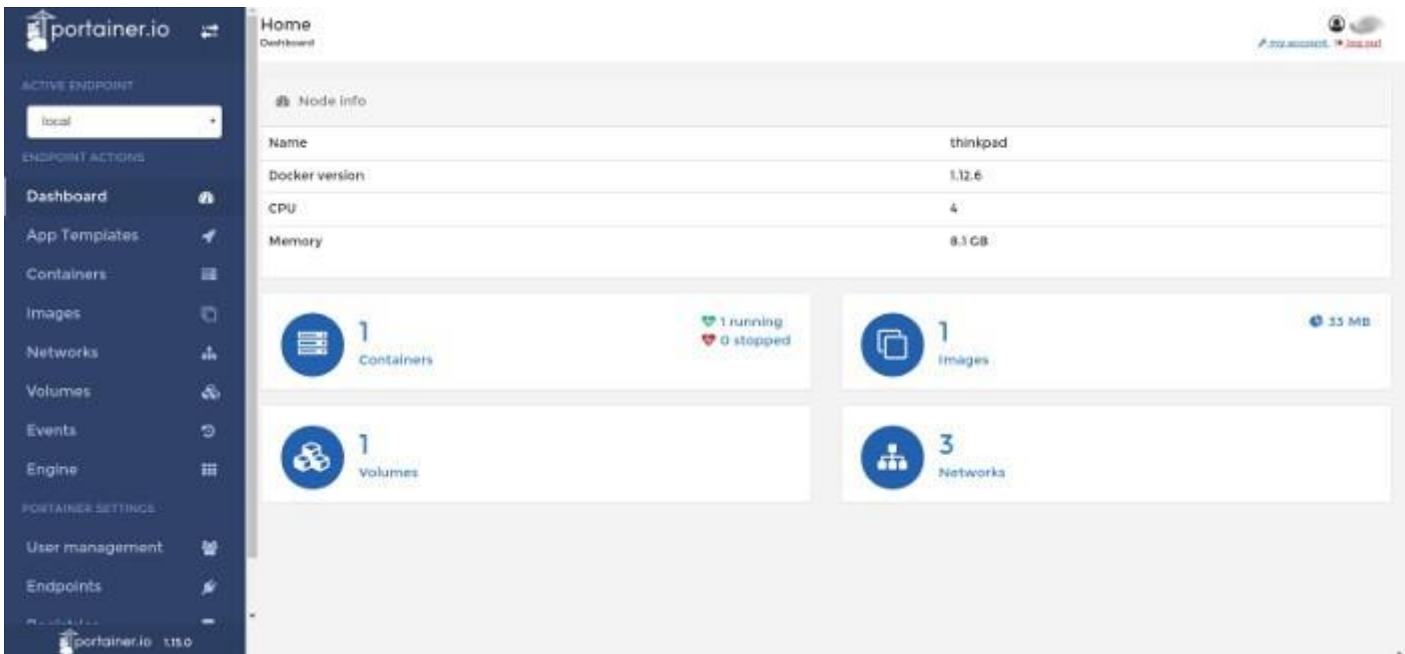
http://192.168.1.150:9000





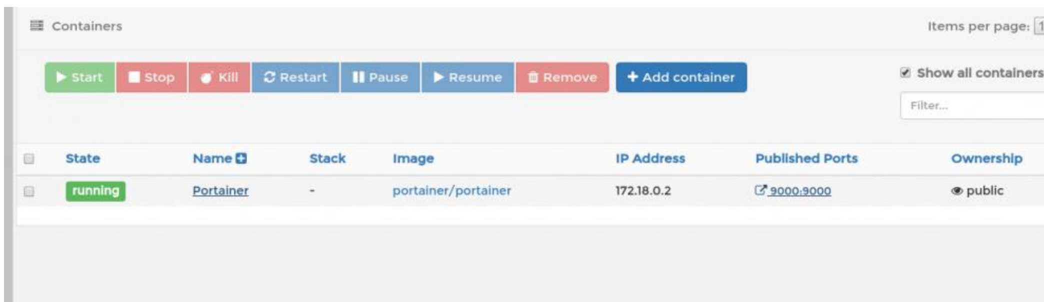
Una vez ejecutado esto ya podemos entrar con el navegador en http://localhost:9000

Os preguntará que pongáis un usuario y una contraseña y pasaréis a la siguiente ventana donde nos preguntará si queremos levantarlo en local o en remoto, en nuestro caso seleccionamos en local y así poderlo ejecutar directamente en nuestro servidor



El Dashboard es la página principal, la de entrada, donde os muestra un resumen muy visual de lo que hay, el número de contenedores, de imágenes, de redes, esta página es simplemente informativa y nos sirve para ver un poco por encima qué es lo que tenemos.

El siguiente menú es el de **App Templates**. Aquí podemos ver las plantillas que hay disponibles para descargar y posteriormente instalar, están ordenadas por categorías, y podemo ver plantillas como por ejemplo nginx, gitlab, ghost, drupal, El siguiente menú es el de Containers, aquí veremos todos nuestros contenedores, y podremos hacer varias cosas, como encenderlos, apagarlos, o ver detalles del propio contenedor, pinchando en la columna Name, de la imagen pinchando en la columna image o entrar en el pinchando enb published ports.



Ahora pasamos a la pantalla de imagenes en la columna de la izquierda donde podemos hacer lo mismo de antes, pero con imágenes, además de bajar alguna imagen directamente si conocemos el nombre en DockerHub.

En networks podemos ver, añadir o eliminar redes de docker de una forma muy sencilla

En volumenes veremos los directorios del sistema que usa docker, podremos añadir volúmenes, o borrarlos, aunque lo normal es que simpelmente se gestionen al instalar nuevas imágenes.

Y luego ya pasaríamos a los logs y configuraciones del propio portainer.