Docker

Overview

Docker es una plataforma abierta para desarrollar, enviar y ejecutar aplicaciones. Docker le permite separar sus aplicaciones de su infraestructura para que pueda entregar software rápidamente. Con Docker, puede administrar su infraestructura de la misma manera que administra sus aplicaciones. Al aprovechar las metodologías de Docker para enviar, probar e implementar el código rápidamente, puede reducir significativamente la demora entre la escritura del código y su ejecución en producción.

La plataforma Docker

Docker ofrece la capacidad de empaquetar y ejecutar una aplicación en un entorno aislado y holgado llamado contenedor. El aislamiento y la seguridad le permiten ejecutar muchos contenedores simultáneamente en un host determinado. Los contenedores son livianos porque no necesitan la carga adicional de un hipervisor, sino que se ejecutan directamente dentro del núcleo del equipo host. Esto significa que puede ejecutar más contenedores en una combinación de hardware dada que si estuviera usando máquinas virtuales. Es incluso posible ejecutar contenedores Docker dentro de máquinas host que en realidad son una máquina virtual.

Los contenedores permiten a un desarrollador empaquetar una aplicación con todas las partes que necesita, como bibliotecas y otras dependencias, y enviar todo en un solo paquete. Al hacerlo, gracias al contenedor, el desarrollador puede estar seguro de que la aplicación se ejecutará en cualquier otra máquina Linux, independientemente de las diferencias de configuración que haya entre la máquina utilizada para escribir el código y la máquina donde se ejecute el contenedor.

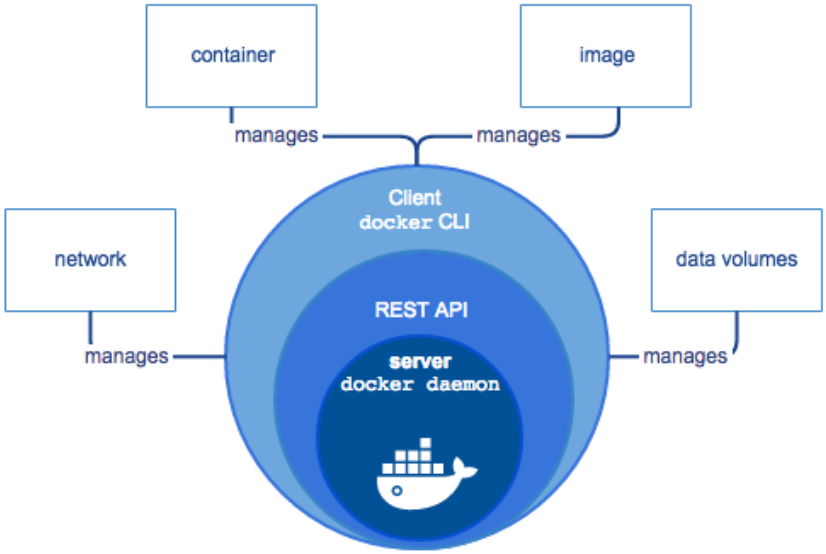
En cierto modo, **Docker es un poco como una máquina virtual**. Pero a diferencia de una máquina virtual, en lugar de crear un sistema operativo virtual completo, Docker permite que las aplicaciones utilicen el mismo núcleo de Linux que el sistema en el que se ejecutan y sólo requiere que las aplicaciones se envíen con las aplicaciones, bibliotecas o dependencias que no se estén ejecutando en el equipo anfitrión. Esto proporciona un aumento significativo del rendimiento y reduce el tamaño de la aplicación.

El paquete que contiene la aplicación, que es la **imagen** es típicamente de decenas o cientos de megabytes, por lo que es fácil de almacenar y rápido de mover. Cuando ejecute un contenedor desde la imagen, se iniciará en segundos y el proceso de la aplicación en realidad se ejecuta en el host, lo que significa que puede ejecutar cientos de contenedores en una sola máquina.

Docker Engine

Docker Engine es una aplicación cliente-servidor con estos componentes principales:

- Un **servidor** que es un tipo de programa de larga duración llamado proceso daemon (el comando `dockerd`).
- Una **API REST** que especifica las interfaces que los programas pueden usar para hablar con el daemon e indicarle qué hacer.
- Un **cliente** de la interfaz de línea de comandos (CLI) (el comando `docker`).



La consola de comandos (CLI) utiliza la API REST de Docker para controlar o interactuar con el demonio de Docker mediante secuencias de comandos o comandos CLI directos. Muchas otras aplicaciones de Docker utilizan la API y la CLI subyacentes.

El demonio crea y administra objetos Docker, como imágenes, contenedores, redes y volúmenes.

Docker se construye utilizando el lenguaje **Go**, pero se envía como binarios compilados, lo que significa que no es necesario instalar el lenguaje Go previamente.

Usos de Docker

Entrega rápida y consistente de sus aplicaciones

Docker optimiza el ciclo de vida del desarrollo al permitir que los desarrolladores trabajen en entornos estandarizados utilizando contenedores locales que proporcionan sus aplicaciones y servicios. Los contenedores son excelentes para flujos de trabajo de integración continua y entrega continua (CI/CD).

Considere el siguiente escenario de ejemplo:

- Sus desarrolladores escriben el código localmente y comparten su trabajo con sus colegas utilizando contenedores Docker.
- Utilizan Docker para impulsar sus aplicaciones en un entorno de prueba y ejecutar pruebas automáticas y manuales.
- Cuando los desarrolladores encuentran errores, pueden solucionarlos en el entorno de desarrollo y volver a implementarlos en el entorno de prueba para realizar pruebas y validaciones.
- Cuando finaliza la prueba, obtener la solución para el cliente es tan simple como enviar la imagen actualizada al entorno de producción.

Despliegue sensible y escalado

La plataforma basada en contenedor de Docker permite cargas de trabajo altamente portátiles. Los contenedores Docker pueden ejecutarse en una computadora portátil local de un desarrollador, en máquinas físicas o virtuales en un centro de datos, en proveedores en la nube o en una combinación de entornos.

La portabilidad y la naturaleza liviana de Docker también facilitan la gestión dinámica de las cargas de trabajo, ampliando o reduciendo las aplicaciones y servicios según lo requieran las necesidades comerciales, casi en tiempo real.

Ejecutando más cargas de trabajo en el mismo hardware

Docker es ligero y rápido. Proporciona una alternativa viable y rentable a las máquinas virtuales basadas en hipervisores, para que pueda utilizar más de su capacidad de cómputo para alcanzar sus objetivos comerciales. Docker es perfecto para entornos de alta densidad y para implementaciones pequeñas y medianas donde necesita hacer más con menos recursos.

Arquitectura Docker

Docker utiliza una arquitectura cliente-servidor. El cliente de Docker habla con el demonio de Docker, que se encarga de construir, ejecutar y distribuir sus contenedores de Docker. El cliente y el daemon de Docker pueden ejecutarse en el mismo sistema, o puede conectar un cliente de Docker a un demonio de Docker remoto. El demonio y el cliente Docker se comunican mediante una API REST, a través de sockets UNIX o una interfaz de red.

Demonio Docker

El demonio de Docker (`dockerd`) escucha las solicitudes de la API de Docker y administra los objetos de Docker, como imágenes, contenedores, redes y volúmenes. Un demonio también puede comunicarse con otros demonios para administrar los servicios de Docker.

El cliente Docker

El cliente Docker (`docker`) es la forma principal en que muchos usuarios de Docker interactúan con Docker. Cuando usa comandos como `docker run` , el cliente envía estos comandos a `dockerd` , lo que los ejecuta. El comando `docker` utiliza la API de Docker. El cliente Docker puede comunicarse con más de un demonio.

Registros Docker

Un registro de Docker almacena imágenes de Docker. **Docker Hub** es un registro público que cualquiera puede usar, y Docker está configurado para buscar imágenes en Docker Hub de forma predeterminada. Incluso puede ejecutar su propio registro privado. Si usa Docker Datacenter (DDC), incluye Docker Trusted Registry (DTR).

Cuando usa los comandos `docker pull` o `docker run` , las imágenes requeridas se extraen de su registro configurado. Cuando usa el comando `docker push` , su imagen se inserta en su registro configurado.

Objetos Docker

Cuando utiliza Docker, está creando y utilizando imágenes, contenedores, redes, volúmenes, complementos y otros objetos:

- **Images**: Una imagen es una plantilla de solo lectura con instrucciones para crear un contenedor Docker. A menudo, una imagen se basa en otra imagen, con alguna personalización adicional. Por ejemplo, puede crear una imagen basada en la imagen de Ubuntu, pero instala el servidor web Apache y su aplicación, así como los detalles de configuración necesarios para hacer que su aplicación se ejecute.

Puede crear sus propias imágenes o solo usar las creadas por otros y publicadas en un registro. Para crear su propia imagen, cree un fichero *Dockerfile* con una sintaxis simple para definir los pasos necesarios para crear la imagen y ejecutarla. Cada instrucción en un fichero *Dockerfile* crea una capa en la imagen. Cuando cambia el fichero *Dockerfile* y reconstruye la imagen, solo se reconstruyen las capas que han cambiado. Esto es parte de lo que hace que las imágenes sean tan livianas, pequeñas y rápidas, en comparación con otras tecnologías de virtualización.

- **Containers**: Un contenedor es una instancia ejecutable de una imagen. Puede crear, iniciar, detener, mover o eliminar un contenedor utilizando la API o la CLI de Docker. Puede conectar un contenedor a una o más redes, adjuntar almacenamiento o incluso crear una nueva imagen en función de su estado actual.

De forma predeterminada, un contenedor está relativamente bien aislado de otros contenedores y su máquina host. Puede controlar el aislamiento de la red, el almacenamiento u otros subsistemas subyacentes de un contenedor desde otros contenedores o desde la máquina host.

Un contenedor se define por su imagen, así como por las opciones de configuración que le proporciona al crearlo o iniciarlo. Cuando se elimina un contenedor, desaparecen todos los cambios en su estado que no se almacenan en el almacenamiento persistente.

- **Servicios**: Los servicios le permiten escalar contenedores a través de múltiples demonios Docker, que trabajan todos juntos como un enjambre ('*swarm*') con múltiples '*managers*' y '*workers*' . Cada miembro de un '*swarm*' es un demonio Docker, y todos los demonios se comunican utilizando la API de Docker. Un servicio le permite definir el estado deseado, como la cantidad de réplicas del servicio que debe estar disponible en un momento dado. De forma predeterminada, el servicio tiene carga equilibrada en todos los nodos de trabajo. Para el consumidor, el servicio Docker parece ser una aplicación única. Docker Engine admite el modo '*swarm*' en Docker 1.12 y superior.

Ejemplo de Docker

El siguiente comando ejecuta un contenedor 'ubuntu', se adjunta interactivamente a su sesión local de línea de comandos y se ejecuta `/bin/bash` :

```
docker run -i -t ubuntu /bin/bash
```

1. Si no tiene la imagen 'ubuntu' localmente, Docker la extrae de su registro configurado, como si hubiera ejecutado `docker pull ubuntu` manualmente.
2. Docker crea un nuevo contenedor, como si hubiera ejecutado `docker container create` manualmente.
3. Docker asigna un sistema de archivos de lectura-escritura al contenedor, como su capa final. Esto permite que un contenedor en ejecución cree o modifique archivos y directorios en su sistema de archivos local.

4. Docker crea una interfaz de red para conectar el contenedor a la red predeterminada, ya que no se especificó ninguna opción de red. Esto incluye la asignación de una dirección IP al contenedor. De forma predeterminada, los contenedores pueden conectarse a redes externas mediante la conexión de red de la máquina host.

5. Docker inicia el contenedor y ejecuta `/bin/bash`. Debido a que el contenedor se ejecuta de forma interactiva y está conectado a su terminal (debido a los indicadores `-i` y `-t`), puede proporcionar entrada utilizando su teclado mientras la salida se registra en su terminal.

6. Cuando escribe `exit` para terminar, el contenedor se detiene pero no se elimina. Puedes comenzar de nuevo o eliminarlo.

Running containers

Las **imágenes Docker** son aplicaciones empaquetadas. Puede subirlas a un almacén central (llamado registro) y descargarlas en cualquier máquina que tenga acceso al registro.

La imagen es una **sola unidad lógica** que contiene la aplicación empaquetada. Para iniciar la aplicación, hay que ejecutar un contenedor desde la imagen.

Las imágenes se construyen normalmente para ejecutar un solo proceso. Si la aplicación necesita trabajar con otros servicios, se ejecutan esos servicios en sus propios contenedores y se orquestan para que todos los contenedores puedan trabajar juntos.

Cuando ejecuta un contenedor desde una imagen, puede ser una aplicación de corta duración que ejecuta alguna funcionalidad y luego termina; puede ser una aplicación de larga duración que se ejecuta como un servicio de fondo; o puede ser un contenedor interactivo con el que se puede conectar como si fuera una máquina remota.

Ejemplo de un contenedor que muestra "Hello World":

```
docker container run hello-world
```

- El cliente Docker local envía una solicitud al servidor Docker para ejecutar un contenedor desde la imagen llamada "hello-world".
- El servidor Docker comprueba si tiene una copia de la imagen en su caché. Si no, descargará la imagen desde el **Docker Hub**.
- Cuando la imagen se descarga localmente, el servidor Docker ejecuta un contenedor desde la imagen y envía la salida al cliente Docker.

Con esta imagen de ejemplo, el proceso dentro del contenedor finaliza cuando se ha escrito la salida de la consola. Los contenedores Docker salen cuando no hay procesos ejecutándose en su interior.

Este tipo de contenedor, que ejecuta un código y luego sale, es un patrón muy útil. Puede utilizar este enfoque para los contenedores que crean secuencias de comandos de tareas repetitivas, como realizar copias de seguridad de datos, crear infraestructura en la nube o procesar un mensaje de una cola de mensajes. Pero los contenedores son igualmente adecuados para **procesos en *background* de larga duración**.

Docker es una plataforma ideal para este tipo de contenedores ya que el contenedor **no necesita los recursos** hasta que no sea necesario, lo que permite ejecutar muchos contenedores en máquinas modestas. Siempre y cuando los patrones de uso sean variados y las cargas de los contenedores no sean todas al mismo tiempo, el host puede compartir recursos entre todos los contenedores.

Docker soporta procesos en segundo plano de larga duración, como servidores web, permitiendo que los contenedores se ejecuten en modo independiente, por lo que el contenedor sigue funcionando en segundo plano.

Por ejemplo, podemos ejecutar un contenedor desde una imagen del servidor web Nginx. Este comando publica el puerto 80 dentro del contenedor en el puerto 80 del host. Los puertos no se pueden compartir, así que esto fallará si tiene otro proceso escuchando en el puerto 80. Sin embargo, puede publicar el puerto de contenedor en cualquier puerto libre del host. Una vez que el contenedor arranca, podremos acceder al servidor escribiendo http://localhost en el navegador, lo que nos mostrará la página de bienvenida de Nginx:

```
docker container run --detach --publish 80:80 nginx:alpine
```

La salida del Docker muestra el **ID único** generado del nuevo contenedor y luego el control regresa a la terminal. Los ID en Docker son un código alfanumérico, como por ejemplo "7cfb3e6a2c0f21d867dc90708a6bbac7e079c". Docker también le asigna al contenedor un nombre generado aleatoriamente si no se le asigna un nombre manualmente.

El último tipo de contenedor es uno que se ejecuta de **forma interactiva**. Permanece vivo mientras esté conectado a él con el Docker CLI, y se comporta como una conexión remota a una máquina separada. Puede utilizar los contenedores de esta manera para evaluar imágenes, utilizar imágenes como herramientas de software o para seguir los pasos cuando está construyendo su propia imagen.

La mayoría de las imágenes del Docker Hub utilizan Linux como sistema operativo base, y Ubuntu es una de las imágenes base más populares. Canonical publica la imagen oficial de Ubuntu, y ha integrado Docker Hub con su ciclo de lanzamiento para que las últimas versiones de Ubuntu estén disponibles en el Docker Hub. Estas imágenes son **mínimas** de forma deliberada. Imágenes pequeñas equivalen a menos paquetes innecesarios e imágenes más seguras ya que con menos paquetes se reducen las posibles vulnerabilidades.

Para ejecutar un contenedor de forma interactiva se utiliza este comando. Docker ejecuta el contenedor de forma interactiva con emulación de terminal, lo cual permite ejecutar comandos como si estuvieramos conectados por SSH:

```
docker container run --interactive --tty ubuntu:16:04
docker container run -it ubuntu:16:04
```

Podemos ejecutar comandos de Linux en el contenedor como cualquier otra instalación de Ubuntu, en este caso. Esta versión de Ubuntu es pequeña y no tiene todos los comandos. Además, cualquier cambio en la instancia del contenedor **no se guarda**, lo cual quiere decir que si apagamos el contenedor y lo volvermos arrancar, estará tal y como se descargó por primera vez.

Meet the Dockerfile

Para empaquetar una imagen personalizada de Docker, se escribe un documento de texto llamado **Dockerfile** que enumera todos los pasos necesarios tanto para crear la imagen como para utilizar la línea de comandos de Docker para crearla. El Dockerfile utiliza un lenguaje específico de dominio muy sencillo que sólo requiere un puñado de instrucciones. El nombre por defecto del fichero es `Dockerfile` aunque puede usarse cualquier nombre y luego indicarle a Docker el nombre del fichero con `--file`

```
// Ejemplo de Dockerfile
FROM ubuntu

RUN apt-get update && apt-get install nano
```

En este ejemplo, al construir una imagen a partir de este Dockerfile, tendremos un contenedor con Ubuntu y con el paquete 'nano' instalado. Estos comandos **se ejecutan dentro del contenedor** y no en la máquina host.

Para construir la imagen, se utiliza el comando:

```
docker image build --tag myrepository/ubuntu-with-nano .
```

También puede dar a su imagen un nombre de repositorio. Cuando se construye localmente se puede llamar a la imagen como se quiera, pero la convención es usar un formato como **{usuario}/{aplicación}**, donde la parte de usuario es el ID de cuenta en el Docker Hub como por ejemplo "microsoft/azure-cli". Cuando son **imágenes oficiales**, en vez de usar el formato "canonical/ubuntu" se usa directamente "ubuntu".

Para listar las imágenes guardadas en la caché de Docker se usa el comando:

```
docker image ls
```

The main Dockerfile instrucctions

La única instrucción requerida es **FROM**. El resto son opcionales:

- **FROM (requerida)**: especifica la imagen base desde la cual se construirá la imagen.
- **RUN**: ejecuta un comando.
- **ENV**: define variables de entorno.
- **COPY**: copia ficheros desde el contexto a la imagen.
- **EXPOSE**: expone puertos desde el contenedor que pueden ser mapeados a puertos del host.
- **VOLUME**: crea un directorio dentro de la imagen que puede ser mapeado con sistema de almacenamiento externo.
- **CMD**: especifica el comando que se ejecuta cuando el contenedor arranca.

```
// Ejemplo de Dockerfile
FROM ubuntu
RUN apt-get update && \
 apt-get install -y netcat-openbsd
ENV LOG_FILE echo.out
COPY ./echoserver.sh /echoserver.sh
```

```
RUN chmod +x /echoserver.sh
EXPOSE 8082
VOLUME /server-logs
CMD /echoserver.sh
```

Docker utiliza un sistema de archivos en capas para las imágenes. A partir de la imagen base, el servidor Docker ejecuta un contenedor temporal desde la imagen para cada instrucción en el archivo Dockerfile, ejecuta la instrucción y guarda el contenedor temporal como una nueva imagen, añadiéndola a la caché de imagen local. Docker utiliza la caché durante el proceso de compilación, lo que significa que si encuentra una imagen que captura la pila de instrucciones actual, es decir, una que coincida con el estado que le está pidiendo a Docker que cree, reutilizará la imagen en caché.

Esto significa que si por ejemplo la instucción `apt-get update` esta en la caché, no se ejecutará la actualización de los paquetes en sucesivos creaciones de imagen a partir del Dockerfile ya que está en la caché. Se puede consultar la documentación oficial para crear ficheros Dockerfile.

About image registries

Empaquetar una aplicación en una imagen de contenedor es fácil con el comando `docker image build` pero la imagen sólo se crea en la caché de imagen local de su propio equipo. Para hacer que esa imagen sea útil, se necesita poder enviarla a diferentes hosts, y es ahí donde entran los registros de imágenes. Un registro es simplemente una ubicación compartida para las imágenes: se puede subir imágenes que creadas y descargas las imágenes creadas por otras personas. El registro es un concepto clave en Docker, y se utiliza la herramienta normal de la línea de comandos de Docker para trabajar con registros.

El **Docker Hub** es un repositorio gratuito y público mantenido por Docker. Además de imágenes subidas por usuarios, hay imágenes oficiales seleccionadas por Docker como Ubuntu, Nextcloud, Owncloud, MariaDB, etc...

No hay que confundir la terminología aquí. Un **"registro"** es un servicio para alojar muchos repositorios de imágenes, potencialmente de diferentes usuarios. Un **"repositorio"** es una biblioteca de imágenes que potencialmente contiene muchas imágenes etiquetadas diferentes.

Por ejemplo, el **Docker Hub** es un registro público, y uno de sus usuarios es **Microsoft**. Microsoft tiene un repositorio para .NET Core, "microsoft/dotnet", que ofrece muchas versiones de imágenes. Se puede descargar imágenes .NET Core específicas del repositorio "microsoft/dotnet" en el registro **Docker Hub**.

Antes de subir las imágenes creadas a **Docker Hub** hay que autenticarse en la plataforma. Sólo se subirán las capas o *layers* que no estén en el registro, es decir, si nuestra imagen tiene como base una imagen de Ubuntu que ya está en el registro, esta base no se sube para mantener las imágenes lo más livianas posibles:

```
docker login
docker image push {usuario}/{aplicación}
```

Tags and image versions

Hasta ahora nos hemos referido mayormente a las imágenes por el nombre de su repositorio sin especificar una etiqueta en particular. Los repositorios de imágenes Docker utilizan el formato básico **{user}/{app}**, pero a menudo la etiqueta se utiliza para especificar una versión de publicación, o una variante de la imagen, en el formato **{user}/{app}:{tag}**. Si no especifica una etiqueta cuando se refiere a una imagen, se utiliza la etiqueta predeterminada '***latest***'.

```
docker image ls --filter reference=ubuntu
```

Choose your base image carefully

Docker Hub es el registro por defecto, y hay miles de imágenes de alta calidad disponibles allí, pero sólo los repositorios oficiales están revisados.

Los repositorios oficiales son el mejor lugar para empezar. No sólo reciben asistencia técnica activa, sino que Docker Hub analiza las imágenes a partir de una base de datos de vulnerabilidades de seguridad y marca cualquier problema que encuentre en la imagen.

Data Storage in Docker

Docker utiliza un sistema de archivos en capas para contenedores llamado **Union File System**. Cuando se ejecuta un contenedor a partir de una imagen, todas las capas que entran en la imagen se montan en el contenedor como partes de sólo lectura de un sistema de archivos. A continuación, se crea una capa de lectura-escritura en la parte superior de esta instancia del contenedor. Se puede cambiar los datos dentro del contenedor, pero esto no cambiará la imagen en sí misma. Si se crea un fichero en un contenedor en ejecución y se ejecuta un segundo contenedor a partir del mismo el nuevo contenedor no tendrá el archivo que creó en el primer contenedor.

Esta restricción garantiza que cada contenedor que se ejecute desde una imagen comenzará exactamente con el mismo estado, ya que la única manera de cambiar una imagen es crear explícitamente una nueva. En Docker, si desea mantener los datos en su contenedor y tenerlos disponibles la próxima vez que ejecute un contenedor desde esa imagen, o si desea compartirlos con otros contenedores o con su equipo host, para ello se utilizan el concepto de **volumen**.

Los volúmenes son como **unidades de disco virtual** dentro del contenedor. Cuando se escribe en ellos, se está escribiendo a una ubicación permanente en el disco del equipo host (u otra plataforma de almacenamiento), pero dentro del contenedor se verá como una ruta de archivo local. Los volúmenes son propiedad de un contenedor, pero pueden ser compartidos con otros contenedores.

Al ejecutar un contenedor, se le pueden añadir volúmenes o se pueden configurar en el Dockerfile para que formen parte de la imagen. La instrucción **VOLUME** especifica un volumen de datos con una ruta con nombre que puede utilizar como ubicación de archivo en el contenedor.

```
// Dockerfile
FROM ubuntu:16.04

RUN mkdir /v1 && touch /v1/file1.es.txt

VOLUME /v1
```

También se puede montar un volumen usando un parámetro en el comando de arranque del contenedor:

```
docker container run -it --rm -v ~/v1:/v1 repo/myimage
```

Montar un directorio local en un contenedor permite muchos escenarios, pero el montaje es limitado si necesita mover su contenedor a un host diferente: los datos que necesita ya no estarán disponibles en el nuevo host. Para estos escenarios, Docker soporta sistemas de almacenamiento compartido como fuente de montajes de volumen a través de la arquitectura de plugins de volúmenes.

Sharing data volumes

Los volúmenes de datos existen fuera de un contenedor, lo que significa que cualquier dato escrito en ellos persiste incluso cuando el contenedor no está en funcionamiento o ha sido retirado. Esto también significa que los volúmenes de un contenedor pueden ser compartidos entre contenedores de la misma imagen o de diferentes imágenes, lo que es útil para escenarios como la realización de copias de seguridad de bases de datos. Puede tener una base de datos ejecutándose en un contenedor de larga duración y ejecutar un contenedor de tareas para realizar una copia de seguridad de los archivos de datos, compartiendo el volumen desde el contenedor de la base de datos.

Por ejemplo la imagen oficial de MongoDB almacena la base de datos en un volumen en "/data/db". Con el parámetro `--volumes-from` podríamos cargar dicho volumen en otro contenedor y acceder a la base de datos para, por ejemplo, hacer una copia de seguridad. En el nuevo contenedor, el volumen de datos tiene el mismo path, "/data/db":

```
// Arranque de un contenedor con MongoDB a partir de una imagen oficial de MongoDB
docker container run -d --name mongo mongo

// Cargar el volumen compartido en un contenedor interactivo a partir de una imagen ubuntu
docker container run -it --volumes-from mongo ubuntu
```

Con el siguiente comando podemos inspeccionar un contenedor y saber, por ejemplo, donde está físicamente almacenado en el host el volumen de datos compartido:

```
docker container inspect [IMAGE]
```

Removing containers, data volumes and images

Los contenedores terminan cuando el proceso dentro de ellos termina o cuando son externamente terminados con los comandos `docker container kill` o `docker container stop`. Pero el contenedor no se elimina, sino que se detiene. El comando `docker container ls` no mostrará el contenedor a menos que añada la flag `--all`, que listará todos los contenedores, incluyendo los detenidos.

Para contenedores de una sola tarea y contenedores interactivos, es una buena práctica usar la flag `--rm` al iniciar el contenedor para que Docker elimine automáticamente el contenedor cuando termine.

Para borrar explícitamente un contenedor se usa el comando `docker container rm {container}`. Se pueden **encadenar** comandos (sintaxis de Linux) de forma que, por ejemplo, podríamos eliminar todos los contenedores que estén detenidos:

```
docker container rm $(docker container ls -a -q)
```

Para borrar también el volumen asociado al contenedor, usamos la flag `-v` :

```
docker container rm -v {container}
```

Cuando eliminamos un contenedor pero no el volumen de datos que tuviera asociado, ese volumen queda "huérfano" o "*dangling*". Cuando no hay un contenedor creado a partir de una imagen en caché, se considera que esa imagen también es "huérfana" o "*dangling*". Al trabajar con Docker, ya que Docker no elimina nada de forma explícita, se suele dar el caso de acabar con imágenes, volúmenes y contenedores sin utilizar. Para ello podemos usar una serie de comandos. Docker también proporciona un comando que elimine todos los elementos no utilizados del sistema:

Orchestrating Systems with Docker

Cuando se descompone una aplicación grande en partes más pequeñas, cada una de las cuales se ejecuta de forma separada en un contenedor se necesita una forma de que los contenedores funcionen juntos. Eso se llama **orquestación**. La orquestación requiere un *framework* que permita la comunicación entre contenedores, configuración de cómo deben conectarse los contenedores, programación de la creación de contenedores, monitoreo de la salud y escalamiento. Hay plataformas externas para la orquestación de contenedores como **Mesos** y **Kubernetes**, que son muy populares, pero Docker tiene orquestación integrada en la plataforma llamado **Docker Swarm**.

Docker development best practices

Las imágenes pequeñas son más rápidas de extraer sobre la red y más rápidas de cargar en la memoria al iniciar contenedores o servicios. Existen algunas reglas básicas para mantener pequeño el tamaño de la imagen:

- Comience con una imagen base apropiada. Por ejemplo, si necesita un JDK, considere basar su imagen en la imagen oficial openjdk, en lugar de empezar con una imagen ubuntu genérica e instalar openjdk como parte del Dockerfile.
- Si tiene varias imágenes con muchas partes en común, considere la posibilidad de crear su propia imagen base con los componentes compartidos y basar sus imágenes únicas en ella. Docker sólo necesita cargar las capas comunes una vez, y se almacenan en caché. Esto significa que las imágenes derivadas utilizan la memoria del host Docker de forma más eficiente y se cargan con mayor rapidez.
- Para mantener su imagen de producción reducida pero permitir la depuración, considere usar la imagen de producción como la imagen base para la imagen de depuración. Se pueden agregar herramientas adicionales de prueba o depuración en la parte superior de la imagen de producción.
- Al crear imágenes, siempre etiquételas con etiquetas útiles que codifiquen la información de la versión, el destino deseado (por ejemplo, 'prod' o 'test'), la estabilidad u otra información que sea útil al implementar la aplicación en diferentes entornos. No confíe en la última etiqueta creada automáticamente.

- Evite almacenar datos de aplicaciones en la capa grabable de su contenedor utilizando controladores de almacenamiento. Esto aumenta el tamaño de su contenedor y es menos eficiente desde una perspectiva de E / S que el uso de volúmenes o montajes de enlace.
- En su lugar, almacenar datos utilizando volúmenes.
- Un caso en el que es apropiado usar *bind mounts* es durante el desarrollo, cuando usted quiera montar su directorio fuente o un binario que acaba de incorporar a su contenedor. Para la producción, utilice en su lugar un volumen, montándolo en la misma ubicación en la que montó un soporte de encuadernación durante el desarrollo.
- Para la producción, utilice *secrets* para almacenar datos de aplicaciones sensibles utilizados por los servicios, y utilice *configs* para datos no sensibles como archivos de configuración. Si actualmente utiliza contenedores independientes, considere la posibilidad de migrar para utilizar servicios de réplica única, de modo que pueda aprovechar estas funciones de sólo servicio.
- Cuando sea posible, diseñe su aplicación con la capacidad de escalar usando servicios '*swarm*'.
- Incluso si sólo necesita ejecutar una única instancia de su aplicación, los servicios de '*swarm*' proporcionan varias ventajas sobre los contenedores independientes. La configuración de un servicio es declarativa, y Docker siempre está trabajando para mantener el estado deseado y actual sincronizado.
- Las redes y los volúmenes pueden conectarse y desconectarse de los servicios '*swarm*', y Docker se encarga de redistribuir los contenedores de servicio individuales de forma no disruptiva. Los contenedores autónomos necesitan ser detenidos manualmente, removidos y recreados para adaptarse a los cambios de configuración.
- Varias características, como la capacidad de almacenar '*secrets*' y '*configs*', sólo están disponibles para los servicios en lugar de para los contenedores independientes. Estas funciones le permiten mantener sus imágenes lo más genéricas posible y evitar almacenar datos confidenciales dentro de las imágenes o contenedores del Docker.

Source

Appendix: Common Docker Commands

Image

Build an image:

```
docker image build --rm=true .
```

Install an image:

```
docker image pull ${IMAGE}
```

List of installed images:

```
docker image ls
```

List of installed images (detailed listing):

> docker image ls --no-trunc

Remove an image:

> docker image rm ${IMAGE_ID}

Remove unused images:

> docker image prune

Remove all images:

> docker image rm $(docker image ls -aq)

Para listar las instrucciones usadas en la creación de la imagen:

> docker image history ${IMAGE_ID}

Containers

Run a container:

> docker container run

List of running containers:

> docker container ls

List of all containers:

> docker container ls -a(--all)

Stop a container:

> docker container stop ${CONTAINER_ID}

Stop all running containers:

> docker container stop $(docker container ls -q)

List all exited containers with status 1:

> docker container ls -a --filter "exited=1"

Remove a container:

> docker container rm ${CONTAINER_ID}

Remove container by a regular expression:

> docker container ls -a | grep wildfly | awk '{print $1}' | xargs docker container rm -f

Remove all exited containers:

> docker container rm -f $(docker container ls -a | grep Exit | awk '{ print $1 }')

Remove all containers:

```
docker container rm $(docker container ls -aq)
```

Find IP address of the container:

```
docker container inspect --format '{{ .NetworkSettings.IPAddress }}' ${CONTAINER_ID}
```

Attach to a container:

```
docker container attach ${CONTAINER_ID}
```

Open a shell into a container:

```
docker container exec -it ${CONTAINER_ID} bash
```

Get container id for an image by a regular expression:

```
docker container ls | grep wildfly | awk '{print $1}'
```

Volume

Eliminar volúmenes huérfanos ('*dangling*')

```
docker volume rm $(docker volume ls -qf dangling=true)
```

Others

Limpieza general del sistema:

```
docker system prune
```

Show version

```
docker --version
```

Show client and server version:

```
docker version
```

Start the daemon manually

```
dockerd
```

Reference

Docker Docs
Docker Docs in Github
Docker for Java Developers
What is Docker?