

Docker

⚠ DOCUMENTO EN DESARROLLO ⚠

Overview

Docker permite empaquetar las aplicaciones en **contenedores** que incluyen todo lo necesario para que se puedan ejecutar en un **entorno aislado**. Cada contenedor almacena el código fuente de la aplicación, los archivos de configuración y todas las dependencias de software que necesita. Esta estrategia permite que las aplicaciones se puedan ejecutar de la misma manera sobre cualquier infraestructura que tenga soporte para Docker, tanto de forma local como en la nube.

Con la tecnología de contenedores para aplicaciones, ya no es necesario preocuparse por el software que está instalado en la máquina donde se ejecuta el contenedor, porque **todo lo que la aplicación necesita** está incluido dentro del propio contenedor. Esta forma de trabajar resuelve el problema de *'funciona en mi máquina'*, donde una aplicación puede funcionar correctamente en el entorno de desarrollo, pero tiene errores en el entorno de producción porque los dos entornos no son idénticos y contienen versiones de software diferentes.

Cada vez hay más equipos de desarrollo y operaciones que están utilizando la tecnología de contenedores Docker en sus flujos de trabajo. Esto ha permitido acelerar el proceso de desarrollo de las aplicaciones, ha facilitado la forma de distribuirlas y ha acelerado la automatización del despliegue en producción.

Máquina virtual vs contenedor

Una **máquina virtual (VM)** es un entorno que emula la misma funcionalidad de una máquina física. Una máquina virtual utiliza los **recursos** que se le han asignado, como por ejemplo su propia CPU, memoria, interfaz de red, almacenamiento y su propio sistema operativo.

Las máquinas virtuales (VM) se crean y se ejecutan sobre un software llamado **hipervisor** o **virtual machine monitor (VMM)**. El hipervisor se ejecuta en la máquina física y actúa como una **capa intermedia** entre el hardware de la máquina anfitriona o *'host'* y la máquina virtual. El hipervisor se encarga de **gestionar y distribuir** los recursos de la máquina física entre las máquinas virtuales.

Es posible crear varias máquinas virtuales sobre una misma máquina física. Cada una de las máquinas virtuales estará aislada del resto, tendrá sus propios recursos y contará con su **propio sistema operativo**, que no necesariamente será el mismo que el de la máquina anfitriona.

Un **contenedor** se puede definir como una **unidad estándar de software** que permite empaquetar el código fuente de una aplicación y todas sus dependencias, para que se pueda **distribuir y ejecutar** de forma rápida y fiable en diferentes entornos.

También se puede definir como un proceso que ha sido aislado de todos los demás procesos de la máquina anfitriona en la que está ejecutando. Aunque es posible tener más de un proceso en un contenedor, las buenas prácticas recomiendan ejecutar solo un proceso por contenedor.

Los contenedores deben cumplir con los **estándares abiertos** de la industria de los contenedores software desarrollados por la **OCI (Open Container Initiative)**.

La principal diferencia entre una máquina virtual y un contenedor es que la **máquina virtual necesita un sistema operativo** completo para poder funcionar mientras que un **contenedor no lo necesita** ya que comparte el kernel del sistema operativo de la máquina en la que se está ejecutando.

Por lo tanto, **los contenedores requieren menos recursos** que las máquinas virtuales. Con el mismo hardware es posible tener un mayor número de contenedores que de máquinas virtuales.

Además, los contenedores son más livianos y arrancan más rápido que las máquinas virtuales.

Por último, **un contenedor se puede ejecutar dentro de una máquina virtual** pero no al revés.

La arquitectura Docker

La arquitectura de Docker fue diseñada inicialmente de forma monolítica, pero más tarde fue rediseñada hacia una **arquitectura modular**, compuesta por diferentes componentes que pueden ser reemplazados e incluso utilizados en otros proyectos.

Cada uno de los componentes de Docker se desarrolla por separado y muchos de ellos forman parte del "**Proyecto Moby**", un proyecto *open source* creado por la compañía Docker Inc. en 2017, en el cual se desarrollan componentes y herramientas que pueden ser utilizadas para crear productos basados en la tecnología de contenedores.

Los principales componentes de Docker que debemos conocer son:

- Cliente Docker
 - Docker CLI
 - Docker Compose
- Docker Engine
 - Docker Engine API
 - Docker Daemon
- Container Runtime
 - Containerd
 - Runc
- Docker Registry

Cliente Docker

Docker utiliza una arquitectura **cliente-servidor**, donde una aplicación cliente interactúa con un servicio llamado **Docker Daemon**. Un mismo cliente puede comunicarse con más de un servicio **Docker Daemon**.

El **Docker Daemon** es el proceso que administra los objetos de Docker, como contenedores, imágenes, redes y volúmenes, y se encarga de ejecutar las instrucciones enviadas por el cliente.

La comunicación entre cliente y servidor se realiza a través de una API HTTP conocida como **Docker Engine API**, que permite la interacción mediante comandos HTTP, posibilitando la automatización y la integración con otras herramientas de terceros en procesos de CI/CD.

Las aplicaciones oficiales que se pueden utilizar como cliente son **Docker CLI ('Command Line Interface')** y **Docker Compose** aunque cualquier aplicación cliente que haga uso de la API de **Docker Engine** puede ser un cliente válido.

El cliente y el servidor se pueden ejecutar en la **misma máquina** o en **máquinas separadas**. Cuando están en la misma máquina, la comunicación entre ambos se realiza a través de un **socket IPC** o un **socket TCP**. En cambio, cuando se encuentran en máquinas separadas, la comunicación se realiza mediante un **socket TCP**.

Es importante considerar la seguridad al utilizar **sockets TCP** en máquinas separadas, recomendándose la configuración de conexiones seguras mediante SSL/TLS para proteger la comunicación.

Docker CLI

Docker CLI es el cliente oficial de Docker. Es una interfaz de línea de comandos que permite a los usuarios interactuar con el servicio **Docker Daemon** para ejecutar, gestionar y supervisar contenedores.

```
# Muestra la ayuda de Docker
```

```
$ docker help
```

El uso más habitual de **Docker CLI** es para interactuar con **contenedores individuales**, aunque también permite gestionar redes, volúmenes e imágenes.

Docker Compose

Docker Compose es una herramienta que permite definir, ejecutar y gestionar aplicaciones con **múltiples contenedores en un sólo host**. Utiliza un archivo de configuración con formato YAML para definir los servicios, las redes y los volúmenes que componen la aplicación.

```
# Muestra la ayuda de Docker Compose
$ docker compose help
```

Una de las ventajas que ofrece **Docker Compose** es que basta con ejecutar un solo comando para crear y ejecutar todos los servicios definidos en el archivo de configuración en formato YAML:

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "8080:80"
  db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: example
```

La especificación Compose es un estándar oficial, unificado y abierto donde se define cómo debe estructurarse un archivo `docker-compose.yml`. Esta especificación es un proyecto open source mantenido por la comunidad y por Docker en su conjunto, disponible [aquí](#).

Docker Compose es ideal para entornos de desarrollo local o para manejar aplicaciones que no requieren orquestación compleja. Es excelente para definir pilas de aplicaciones (e.g., una aplicación con un servidor web, base de datos y servicio de caché) en un solo *host*.

Docker Compose no solo gestiona los servicios, sino también las redes y volúmenes. Se pueden definir cómo los servicios se comunican entre sí y si ciertos volúmenes son compartidos o persistentes, lo que es ideal para casos donde las aplicaciones requieren bases de datos u otros servicios que dependen de almacenamiento persistente.

En la actualidad existen dos versiones de **Docker Compose**:

- **v1**: debe instalarse como una herramienta adicional y se ejecuta con `docker-compose`. Esta implementada en Python.
- **v2**: integra el comando dentro del cliente oficial de **Docker CLI**. Por lo tanto, la nueva versión se ejecuta con `docker compose`. Esta versión está implementada en Go.

Compose file reference

- [Compose file reference - docker.com](#)

```
name: mi-app

services:
  app:
```

```

container_name: mi-app
build:
  context: .
  dockerfile: Dockerfile
image: mi-app:latest
ports:
  - "3000:3000"
environment:
  - NODE_ENV=development
  - API_KEY=${API_KEY}
volumes:
  - ./usr/src/app
  - app-node-modules:/usr/src/app/node_modules
depends_on:
  - db
  - redis
networks:
  - backend
  - frontend

db:
  image: postgres:15
  container_name: mi-db
  restart: unless-stopped
  environment:
    POSTGRES_USER: usuario
    POSTGRES_PASSWORD: password
    POSTGRES_DB: midb
  volumes:
    - db-data:/var/lib/postgresql/data
  networks:
    - backend

redis:
  image: redis:7
  container_name: mi-redis
  restart: always
  ports:
    - "6379:6379"
  networks:
    - backend

volumes:
  db-data:
  app-node-modules:

networks:
  backend:
  frontend:

```

Buenas prácticas para evitar errores comunes al escribir archivos `docker-compose.yml` :

- Uso de comillas dobles (") en puertos, rutas con dos puntos o valores con variables:

```

ports:
  - "3000:3000"
  - "127.0.0.1:8080:80"

environment:
  - "API_KEY=${API_KEY}"

```

- YAML solo permite espacios, no tabulaciones. Usar tabulaciones puede hacer que `docker compose` falle silenciosamente.
- No utilizar números con ceros a la izquierda ya que YAML los interpreta como **número octal**. Si hay que usar ceros, utilizar las comillas:

```
- "PIN=0123"
```

- Las variables se pueden cargar desde uno o varios archivos `.env` :

```
env_file:  
- .env  
- .env.local
```

Dentro del `.env` no hay que usar espacios ni comillas:

```
API_KEY=abc123  
NOMBRE=M MyApp
```

- Para validar el fichero se puede usar el comando `docker compose config` .

Docker Engine

Docker Engine es el componente principal de Docker, responsable de crear, ejecutar y gestionar contenedores.

Tiene un **diseño modular** y está compuesto por varios componentes que cumplen con los estándares abiertos de la **OCI (Open Container Initiative)**:

- *Docker Engine API*
- *Docker Daemon* (componente principal)
- *Container runtime (containerd)*
- Gestión de redes (*libnetwork*)
- Creación de imágenes (*buildkit*)
- Interacción con los registros de contenedores (*distribution*)
- Soporte nativo para la orquestación de contenedores con Docker Swarm (*swarmkit*)
- Gestión de plugins

⚠ Estos son proyectos que cumplen los estándares abiertos y están alojados en **"Moby Project"**.

Docker Engine se ejecuta de forma nativa en los sistemas Linux y Windows Server. En otros sistemas operativos de Windows y en macOS, se ejecuta sobre una máquina virtual Linux.

Docker Engine API

La comunicación entre un cliente Docker y el servicio **Docker Daemon** se realiza a través de una API HTTP conocida como **Docker Engine API**.

Esta API implementa todas las operaciones que un usuario puede realizar desde un cliente Docker. Por ejemplo, cuando un usuario ejecuta el comando `docker ps` desde la línea de comandos, el cliente Docker está haciendo una petición GET al *endpoint* `/containers/json` .

La API puede cambiar cada vez que se libera una nueva versión de **Docker Engine**. Para mantener la compatibilidad, en los *endpoints* se incluye un prefijo en la URL para indicar la versión, como por ejemplo `/v1.41/containers/json` o `/v1.40/containers/json`. Si el cliente no especifica una versión, Docker usa la versión más reciente compatible.

En la documentación de Docker se puede consultar la [referencia completa de la API](#)

Como demostración, se puede hacer uso de la herramienta `curl` para hacer la petición y `jq` para formatear la salida JSON. Con el siguiente comando se puede hacer una petición directa al *endpoint* que nos devolverá la lista de imágenes sin hacer uso del cliente Docker:

```
curl --unix-socket /var/run/docker.sock http://localhost/v1.47/images/json | jq
```

En este ejemplo, se utiliza un socket UNIX porque el cliente y el servicio **Docker Daemon** están en la misma máquina.

Docker daemon

El servicio **Docker Daemon** es el encargado de crear y gestionar todos los objetos con los que trabaja Docker, como las imágenes, los contenedores, las redes y los volúmenes. Este servicio se ejecuta en un proceso llamado *'dockerd'*.

El servicio **Docker Daemon** expone una API HTTP, y los clientes Docker se comunican con el **Docker Daemon** mediante esta API, utilizando tres tipos diferentes de *sockets*: `unix`, `tcp` y `fd`. A su vez, este **Docker Daemon** se comunica con el *'container runtime'*.

En versiones anteriores, el **Docker Daemon** también incluía el *'container runtime'* pero actualmente, son componentes independientes.

El *'container runtime'* está formado por *'containerd'* y *'runc'*. *'containerd'* se encarga de las operaciones de alto nivel (como la gestión de imágenes y la supervisión de contenedores), mientras que *'runc'* ejecuta los contenedores propiamente dichos a nivel de sistema operativo.

En una instalación típica, el cliente Docker y el **Docker Daemon** se encuentran en la misma máquina. Para contenedores en Linux, se utiliza un *socket* de tipo UNIX. Para contenedores en Windows, se utiliza un *'named pipe'*.

En entornos donde el cliente Docker y el **Docker Daemon** están en máquinas distintas, se utiliza un *socket* de tipo TCP. Por defecto, esta comunicación se realiza sobre un canal no cifrado en el puerto 2375, adecuado para entornos de desarrollo. Para entornos de producción, es necesario usar una conexión cifrada con TLS, comúnmente utilizando el puerto 2376.

Container runtime

El *'container runtime'* es el software encargado de ejecutar los contenedores Docker:

- **'containerd'**: *'container runtime'* de alto nivel
- **'runc'**: *'container runtime'* de bajo nivel

containerd

containerd es el componente responsable de gestionar el ciclo de vida de un contenedor dentro de un *host*. Realiza tareas como descargar las imágenes desde los registros de los contenedores, almacenarlas en el *host*, supervisar la ejecución de los contenedores y gestionar el almacenamiento y las redes asociadas.

Este componente puede ser utilizado desde la línea de comandos a través del cliente *ctr*, que se incluye por defecto con la instalación de *containerd*:

```
# Muestra la ayuda del comando
$ ctr help
```

El componente *containerd* también implementa la interfaz CRI (*Container Runtime Interface*) de Kubernetes. Esto significa que este '*container runtime*' puede ser utilizado en un *cluster* de Kubernetes para crear y ejecutar contenedores a partir de imágenes Docker, que son compatibles con la especificación OCI (*Open Container Initiative*).

runc

runc es un componente de bajo nivel encargado de interactuar directamente con el *kernel* del sistema operativo del *host* donde se ejecutan los contenedores. Utiliza la librería **libcontainer** para gestionar esta interacción con el sistema operativo.

Es una implementación de código abierto de la **OCI Runtime Specification**, que define cómo deben configurarse, ejecutarse y gestionarse los contenedores a lo largo de su ciclo de vida.

Docker Registry

Un **Docker Registry** es un servicio encargado de almacenar y distribuir repositorios de imágenes Docker.

Un repositorio de imágenes es un conjunto de imágenes que se agrupan bajo el mismo nombre dentro del registro. Cada imagen en un repositorio está etiquetada con un **tag**, que generalmente se utiliza para indicar su versión.

Docker Hub es el **registro de contenedores oficial** de Docker. Este registro viene configurado por defecto al instalar **Docker Engine**, aunque puede ser reemplazado por otros registros de contenedores.

Algunos de los registros disponibles son:

- [Docker Hub \(oficial\)](#)
- [Github Container Registry](#)
- [DigitalOcean Container Registry](#)
- [Amazon Elastic Container Registry](#)
- [Azure Container Registry](#)
- [Google Cloud Container Registry](#)

Objetos de Docker

Los principales objetos de Docker son:

- Imágenes
- Contenedores
- Volúmenes
- Redes

Imágenes

Las imágenes son **plantillas inmutables** que contienen el sistema de archivos y los parámetros necesarios para configurar un contenedor. Representan el estado inicial del sistema de archivos raíz del contenedor y pueden incluir aplicaciones preinstaladas, configuraciones y dependencias.

Las imágenes se pueden crear manualmente utilizando un archivo llamado `Dockerfile`, que contiene una serie de instrucciones para ensamblar la imagen paso a paso. También es posible obtener imágenes predefinidas desde registros públicos como **Docker Hub**.

Docker utiliza un **sistema de capas** para construir imágenes, lo que permite reutilizar partes comunes entre diferentes imágenes y optimizar el almacenamiento y la transferencia.

A partir de una misma imagen, se pueden **crear múltiples contenedores**, lo que facilita la escalabilidad y consistencia en la implementación de aplicaciones.

Las imágenes suelen construirse 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 colaborar entre sí.

Al ejecutar un contenedor desde una imagen, este puede de corta duración, ejecutando alguna funcionalidad y luego terminando; 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.

Un **tag** es una etiqueta que se le asigna a una imagen dentro de un repositorio. Se utiliza para indicar la versión de una imagen y permite diferenciar de forma sencilla las imágenes que forman parte de un repositorio.

Contenedores

Un contenedor es una **instancia ejecutable de una imagen**. Representa una unidad de software ligera y autónoma que incluye todo lo necesario para ejecutar una aplicación.

Puede crear, iniciar, detener, mover o eliminar un contenedor utilizando la API o la CLI de Docker. Además, es posible conectar un contenedor a una o más redes, adjuntar almacenamiento persistente o incluso crear una nueva imagen basada en el estado actual del contenedor mediante la ejecución de comandos como `docker commit`.

De forma predeterminada, un contenedor está relativamente **aislado** de otros contenedores y de la máquina *host*. Este aislamiento abarca el sistema de archivos, la red y otros recursos del sistema. Sin embargo, Docker proporciona opciones para controlar el nivel de aislamiento según las necesidades específicas.

Es importante notar que, a menos que se utilicen volúmenes o montajes de enlaces, **los cambios realizados dentro de un contenedor no se preservan** una vez que el contenedor se elimina.

En Docker, existen contenedores que se ejecutan y se detienen automáticamente cuando no hay procesos activos en su interior. Este tipo de contenedor es especialmente útil para tareas repetitivas como copias de seguridad, creación de infraestructura en la nube o procesamiento de mensajes, ya que permite automatizar y simplificar estos procesos. Por otro lado, Docker también es ideal para manejar procesos en segundo plano de larga duración, como servidores web. Gracias a su capacidad para utilizar recursos solo cuando es necesario, Docker permite ejecutar múltiples contenedores en máquinas modestas, optimizando así el uso del hardware.

Un ejemplo práctico de esto es la ejecución de un contenedor de Nginx en segundo plano, lo que hace que el servidor web sea accesible en el puerto 80 del *host*. Docker genera un ID único para cada contenedor y, si no se especifica un nombre, asigna uno aleatorio. Este enfoque asegura que el contenedor siga funcionando en segundo plano una vez iniciado, permitiendo un acceso continuo al servicio.

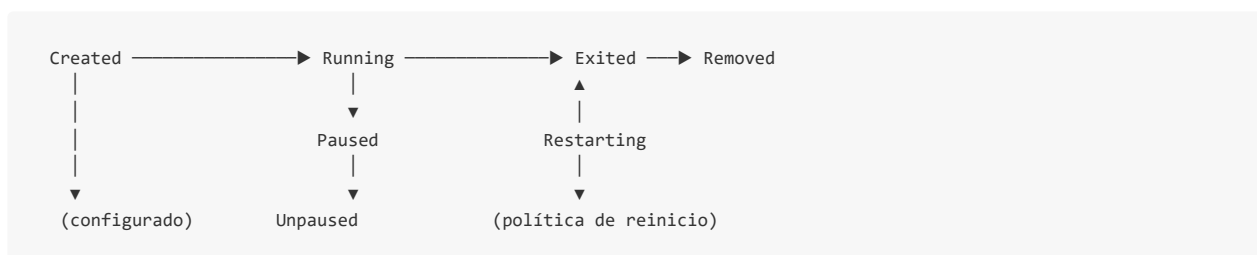
Además, Docker permite ejecutar contenedores de **forma interactiva**, permaneciendo en ejecución mientras la conexión esté activa y comportándose como una máquina remota. Esto es útil para evaluar imágenes, utilizar herramientas de software o seguir pasos en la creación de nuevas imágenes. Las imágenes de Ubuntu, por ejemplo, son populares en Docker debido a su tamaño reducido, lo que minimiza vulnerabilidades. Sin embargo, cualquier cambio realizado en estos contenedores no se guarda al reiniciarlos, manteniendo la imagen en su estado original.

Hay dos tipos de contenedores:

- **Contenedores Linux:** estos contenedores se pueden ejecutar en los sistemas operativos Linux, macOS y Windows. En macOS y Windows, Docker utiliza una máquina virtual ligera (basada en Linux) para ejecutar estos contenedores, ya que necesitan el núcleo de Linux para funcionar. Solo cuando se ejecutan sobre un sistema operativo Linux pueden utilizar directamente el kernel del sistema operativo anfitrión, lo que les permite ser más eficientes en cuanto a recursos.
- **Contenedores Windows:** estos contenedores sólo se pueden ejecutar en sistemas operativos Windows y Windows Server. A diferencia de los contenedores Linux, los contenedores Windows utilizan el kernel de Windows y requieren un sistema operativo Windows de la misma versión o similar para poder funcionar. Esto significa que no se pueden ejecutar contenedores Windows en sistemas basados en Linux sin usar alguna solución de virtualización adicional.

El ciclo de vida de un contenedor está compuesto por los siguientes estados:

- **Created.** En este estado, el contenedor ha sido creado, pero todavía no se está ejecutando. Un contenedor se encuentra en este estado después de ejecutar el comando `docker container create`.
- **Running.** Un contenedor está en ejecución cuando está ejecutando la aplicación que contiene (definido en `CMD` o `ENTRYPOINT`). Puede pasar del estado '**Created**' al estado '**Running**' con el comando `docker container start` o se puede ejecutar directamente con `docker container run`.
- **Paused.** Un contenedor que está en ejecución puede ser pausado para detener su ejecución de forma temporal. Para pausar la ejecución de un contenedor, se ejecuta el comando `docker container pause`, que se encarga de enviar la señal '**SIGSTOP**' al proceso principal del contenedor.
- **Exited.** Para detener de forma temporal un contenedor que está en ejecución, se utiliza el comando `docker container stop`, que se encarga de enviar las señales '**SIGTERM**' y '**SIGKILL**' al proceso principal del contenedor.
- **Removed / Deleted.** El contenedor ha sido completamente eliminado con el comando `docker container rm`.



Volúmenes

Los volúmenes son el mecanismo que ofrece Docker para gestionar la **persistencia de datos** en los contenedores. Permiten almacenar datos fuera del sistema de archivos del contenedor, asegurando que la información persista incluso si el contenedor se elimina.

Pueden ser creados y gestionados con comandos de Docker, como `docker volume create`. Una vez creados, los volúmenes pueden ser montados en uno o varios contenedores, permitiendo compartir datos entre ellos.

El ciclo de vida de los volúmenes es **independiente** del ciclo de vida de los contenedores. Esto significa que los volúmenes permanecen disponibles para otros contenedores incluso después de que los contenedores que los utilizaban hayan sido eliminados.

Además de los volúmenes, Docker también soporta montajes de enlaces (*'bind mounts'*), que permiten montar directorios o archivos específicos del sistema de archivos de la máquina host dentro de un contenedor, proporcionando mayor flexibilidad para interactuar directamente con los datos del *host*.

Redes

Docker proporciona capacidades avanzadas para la creación y gestión de redes, facilitando la **comunicación entre contenedores y con sistemas externos**. El componente principal de **Docker Engine** que se encarga de la gestión de las redes es *libnetwork*.

Docker soporta varios tipos de redes, incluyendo:

- **Bridge** → Es la red por defecto en Docker. Permite la comunicación entre contenedores en la misma máquina host.
- **Host** → Utiliza la pila de red de la máquina host, permitiendo que los contenedores compartan la misma dirección IP y puertos que el host.
- **Overlay** → Permite la comunicación entre contenedores que residen en diferentes máquinas host, comúnmente utilizada en configuraciones de clústeres y orquestación.
- **Macvlan** → Asigna una dirección MAC a cada contenedor, permitiendo que aparezcan como dispositivos físicos en la red.
- **None** → Desconecta el contenedor de cualquier red. Ideal cuando se busca aislamiento total y no se desea comunicación externa.

A través de comandos como `docker network create`, `docker network connect` y `docker network inspect`, los usuarios pueden crear redes personalizadas, conectar contenedores a ellas y obtener información detallada sobre su configuración.

Orquestación de contenedores

En muchas situaciones, podemos necesitar que las aplicaciones se ejecuten sobre un **cluster** de servidores para garantizar la alta disponibilidad y escalabilidad de los servicios.

Para desplegar una aplicación basada en contenedores en un **cluster**, se necesitan herramientas conocidas como **orquestadores de contenedores**. Las más conocidas dentro del ecosistema de Docker son **Docker Swarm** y **Kubernetes**.

Entre las tareas más destacadas de un orquestador de contenedores podemos encontrar:

- Automatizar el despliegue de una aplicación en un **cluster** de servidores.
- Crear y ejecutar los contenedores entre los diferentes nodos del **cluster**.
- Balancear la carga entre todos los contenedores.
- Escalar los servicios de forma automática cuando sea necesario.
- Permitir que una aplicación se recupere automáticamente de los errores.
- Posibilita actualizar una aplicación sin que exista tiempo de inactividad.

Docker Swarm

Docker Swarm es un orquestador que viene integrado de forma nativa en **Docker Engine**. Este orquestador, desarrollado por Docker, forma parte del **"Proyecto Moby"** con el nombre de **SwarmKit**.

Docker Swarm es una opción ideal para aquellos que buscan una solución de orquestación sencilla y totalmente integrada dentro del ecosistema Docker, lo que facilita su implementación sin necesidad de herramientas externas.

Kubernetes

Kubernetes, también conocido como **K8s**, es el orquestador más utilizado en la actualidad.

Este orquestador fue desarrollado originalmente por Google, pero fue donado a la **Cloud Native Computing Foundation (CNCF)**.

Kubernetes puede utilizar diferentes '*container runtimes*' para ejecutar contenedores. El único requisito es que sean compatibles con una API llamada **Container Runtime Interface (CRI)**.

Kubernetes es compatible con *containerd*, que es el '*container runtime*' que utiliza **Docker Engine**. Por lo tanto, en un *cluster* de Kubernetes se pueden crear y ejecutar contenedores a partir de imágenes Docker, que cumplen con la especificación OCI.

Dev containers

TODO

Appendix: Common Docker Commands (🚀)

Images (🚀)

```
# Subcomandos de 'image'
docker image --help

# Listar todas las imágenes instaladas localmente
# Aliases 'docker image list', 'docker images'
docker image ls

# Mostrar información detallada sobre una imagen específica
docker image inspect [image_id]

# Ver historial de instrucciones de una imagen
# Aliases 'docker history'
docker image history [image_id]

# Eliminar una imagen específica a partir de su ID
# Aliases 'docker image remove', 'docker rmi'
docker image rm [image_id]

# Eliminar una imagen específica a partir de su nombre (y tag)
docker image rm [nombre_imagen:etiqueta]

# Eliminar todas las imágenes
docker image rm $(docker image ls --all --quiet) # -aq

# Eliminar imágenes sin etiqueta (o 'dangling') no asociadas a contenedores activos
docker image prune

# Eliminar imágenes sin etiquetar y etiquetadas no asociadas a contenedores activos
docker image prune --all
```

Docker Hub

```
# Autenticarse en un registro
docker login

# Cerrar sesión del registro
docker logout

# Buscar una imagen por algún término en Docker Hub
docker search [término_de_búsqueda]

# Descarga una imagen de Docker Hub
# Aliases 'docker pull'
```

```
docker image pull [nombre[:tag]]

# Subir una imagen a Docker Hub
# Aliases 'docker push'
docker image push [nombre[:tag]]
```

Containers

```
# Subcomandos de 'container'
docker container --help

# Crear un contenedor (en estado 'created', sin arrancarlo) y le asigna un nombre
# Aliases 'docker create'
docker container create --name [name] [IMAGE]

# Arrancar un contenedor
# Aliases 'docker start'
docker container start [container_id]

# Detener un contenedor
# Aliases 'docker stop'
docker container stop [container_id]

# Detener todos los contenedores en ejecución
docker container stop $(docker container ls --quiet)

# Listar contenedores en ejecución
# Aliases 'docker container list', 'docker container ps', 'docker ps'
docker container ls

# Listar todos los contenedores
docker container ls --all

# Listar todos los contenedores que han salido con un estado específico
docker container ls -a --filter "status=exited"

# Obtener el ID del contenedor para una imagen específica usando una expresión regular
docker container ls | grep wildfly | awk '{print $1}'

# Ver detalles de un contenedor
docker container inspect [container_id]

# Encontrar la IP de un contenedor
docker container inspect --format '{{ .NetworkSettings.IPAddress }}' [container_id]

# Mostrar los logs del contenedor
# Aliases 'docker logs'
docker container logs [container_id]

# Mostrar últimos logs
# Aliases 'docker logs'
docker container logs --tail 100 [container_id]

# Ver estadísticas en tiempo real
# Aliases 'docker stats'
docker container stats [container_id]

# Ejecutar un comando en el contenedor
# Aliases 'docker exec'
docker container exec [container_id] [comando] [args...]

# Renombrar un contenedor
# Aliases 'docker rename'
docker container rename [old_name] [new_name]

# Conectar a un contenedor
# Aliases 'docker attach'
docker container attach [container_id]
```

```

# Eliminar un contenedor
# Aliases 'docker container remove', 'docker rm'
docker container rm [container_id]

# Eliminar contenedores que coincidan con una expresión regular
docker container ls -a | grep wildfly | awk '{print $1}' | xargs docker container rm -f

# Eliminar todos los contenedores que han salido
docker container rm -f $(docker container ls --all --filter "status=exited" --quiet)

# Eliminar todos los contenedores
docker container rm $(docker container ls --all --quiet)

# Crea y ejecuta un nuevo contenedor
# Aliases 'docker run'
docker container run --detach --name my-container --publish 80:80 my-image

# Abrir una shell interactiva en un contenedor
docker container exec --interactive --tty ${CONTAINER_ID} bash # -it

```

Volume

```

# Subcomandos de 'volume'
docker volume --help

# Listar todos los volúmenes
# Aliases 'docker volume list'
docker volume ls

# Eliminar todos los volúmenes no utilizados
docker volume prune

# Eliminar volúmenes huérfanos ('dangling')
docker volume rm $(docker volume ls --quiet --filter dangling=true) # -qf

```

Others

```

# Inicializar un proyecto con los archivos necesarios para ejecutarlo en un contenedor
docker init

# Mostrar el número de versión de Docker
docker --version

# Mostrar la versión del cliente y del servidor
docker version

# Muestra los registros del servicio de Docker en sistemas basados en 'systemd'
journalctl -u docker

## Comandos de 'system'
docker system --help

# Proporciona un resumen completo del estado del daemon de Docker
# Aliases 'docker info'
docker system info

# Muestra el uso de espacio en disco por imágenes, contenedores, y volúmenes
docker system df

# Limpieza general del sistema
docker system prune

# Iniciar el daemon manualmente

```

dockerd

Verifica el estado del servicio Docker en sistemas que usan 'systemd'

sudo systemctl status docker

Reinicia el servicio Docker en sistemas que usan 'systemd'

sudo systemctl restart docker

Permite especificar un archivo de configuración personalizado para el daemon de Docker




docker daemon --config-file /path/to/config.json

Actualizar Docker a la última versión disponible

sudo apt-get update && sudo apt-get upgrade docker-ce

Enlaces

Docker

-  [Docker - Build, Ship, and Run Any App, Anywhere](#)
- [Docker - Explore official repositories](#)
-  <https://github.com/veggie Monk/awesome-docker>
-  <https://github.com/docker/awesome-compose>
- [Moby is an open framework created by Docker to assemble specialized container systems](#)

Docker - Learning

- <https://docs.docker.com/>
- <https://roadmap.sh/docker>
- <https://cheatsheets.zip/docker>
- <https://github.com/wsargent/docker-cheat-sheet/tree/master/es-es>
- <https://github.com/collabnix/dockerlabs/blob/master/docker/cheatsheet/README.md>
- <https://labs.play-with-docker.com/>
- <https://www.youtube.com/playlist?list=PLO9JpmNASqM6PxlmKj6kfX-a8WwZJnwD9>

Docker - Security

- <https://github.com/docker/docker-bench-security>
- https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

Container

-  [Open Container Initiative \(OCI\)](#)
-  [Cloud Native Computing Foundation \(CNCF\)](#)
-  [Containerd - An industry-standard container runtime](#)
- [Welcome to the Oracle Container Registry](#)
- [LXC - Linux Containers](#)
- [The Fn project is an open-source container-native serverless platform that you can run anywhere](#)
- [Podman](#)

Container - Tools

- <https://github.com/containers>
- <https://github.com/wagoodman/dive>
- [Container tools by Google](#)

- [Crossplane - The open source multicloud control plane](#)
- [Kool makes using Docker for local development easier, simpler, faster, and better](#)

Dev Containers

- [Development Containers \(dev containers\)](#)
- [Dev Containers](#)

Licencia



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](#).