Docker Básico

Arturo Silvelo

Try New Roads

Introducción

¿Qué es Docker?

- **Docker** es una plataforma de código abierto que automatiza el desarrollo, despliegue y ejecución de aplicaciones. Permite separar las aplicaciones desarrolladas de la infraestructura donde se desarrollan.
- Docker se ejecuta en entornos totalmente aislados llamados **contenedores**. Estos se ejecutan directamente sobre el kernel de la máquina por lo que son mucho más ligeros que las máquinas virtuales.

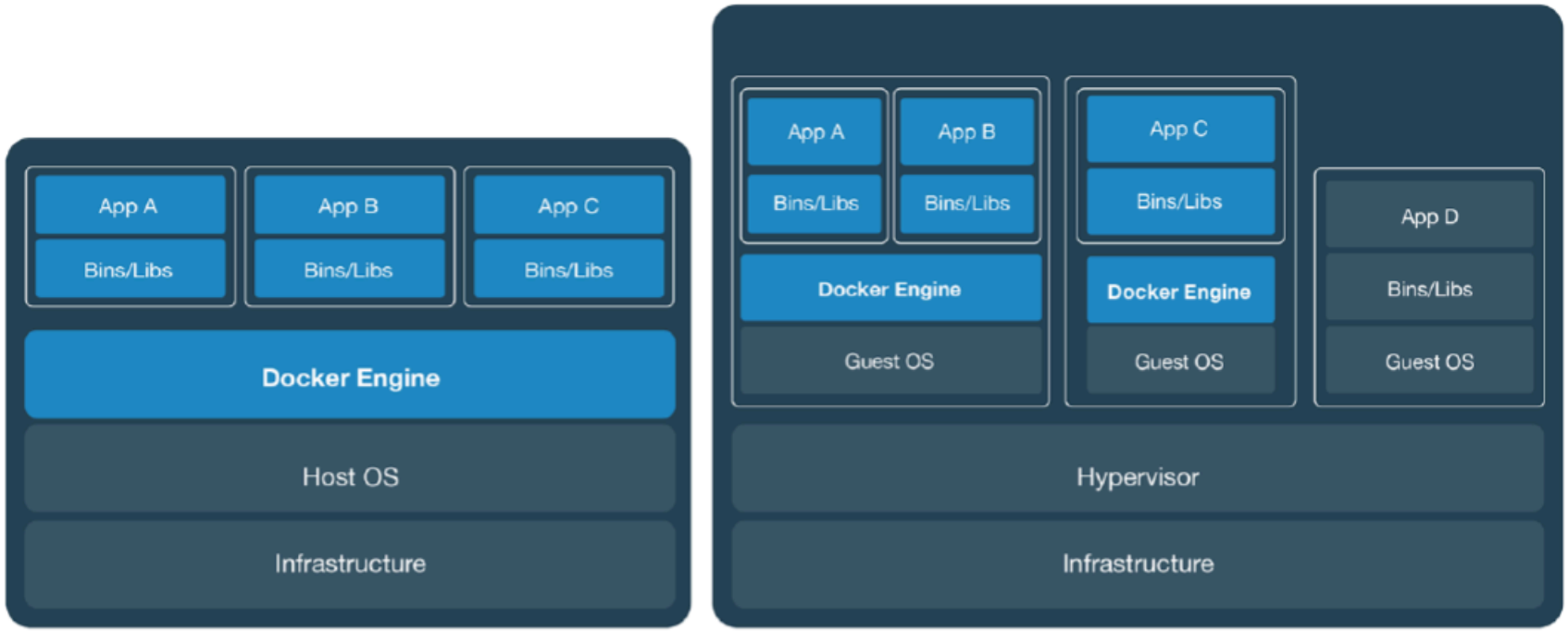
Virtualización

La virtualización permite crear instancias virtuales de un hardware físico, permitiendo ejecutar múltiples sistemas operativos en un solo servidor físico.

- **Virtualización:** Simula hardware completo (CPU, memoria, disco).
- **Docker:** Usa el sistema operativo del host, lo que hace que los contenedores sean más ligeros y rápidos.

Virtualización	Docker
Pesadas	Ligeros
Consumo Recursos	Rápidos

Containers Vs VMs



Ventajas

1. **Portabilidad**
    - Docker empaqueta aplicaciones junto a sus dependencias en contenedores.
    - Esto asegura que las aplicaciones se ejecuten de la misma manera en cualquier entorno.
    - Facilita el despliegue en entornos locales, servidores o la nube sin ajustes adicionales.

2. **Consistencia entre Entornos**
   - Los contenedores permiten que el entorno de desarrollo sea idéntico al de producción.
   - Evita problemas de compatibilidad y errores por diferencias entre entornos.
   - Garantiza que el código funcione igual en desarrollo, pruebas y producción.

3. **Escalabilidad**
    - Docker facilita el escalado horizontal de aplicaciones mediante la creación de múltiples contenedores.
    - Permite el uso de herramientas como Kubernetes para gestionar el escalado automáticamente.
    - Cada servicio puede escalarse de forma independiente en función de la demanda.

4. **Eficiencia en el Uso de Recursos**
    - Los contenedores comparten el núcleo del sistema operativo, siendo más livianos que las máquinas virtuales.
    - Se pueden ejecutar más aplicaciones en el mismo hardware, optimizando recursos.

5. **Velocidad de Desarrollo y Despliegue**
    - Los contenedores se inician en segundos, permitiendo un desarrollo y despliegue rápido.
    - Facilita el uso de CI/CD, reduciendo el tiempo de desarrollo y los ciclos de despliegue.

6. **Seguridad Mejorada**
    - Docker aísla cada contenedor, limitando el acceso entre contenedores y al host.
    - Permite ejecutar aplicaciones de distintos niveles de seguridad en un mismo servidor.

Instalación

- **https://docs.docker.com/get-started/get-docker/**

Instalación en Windows:

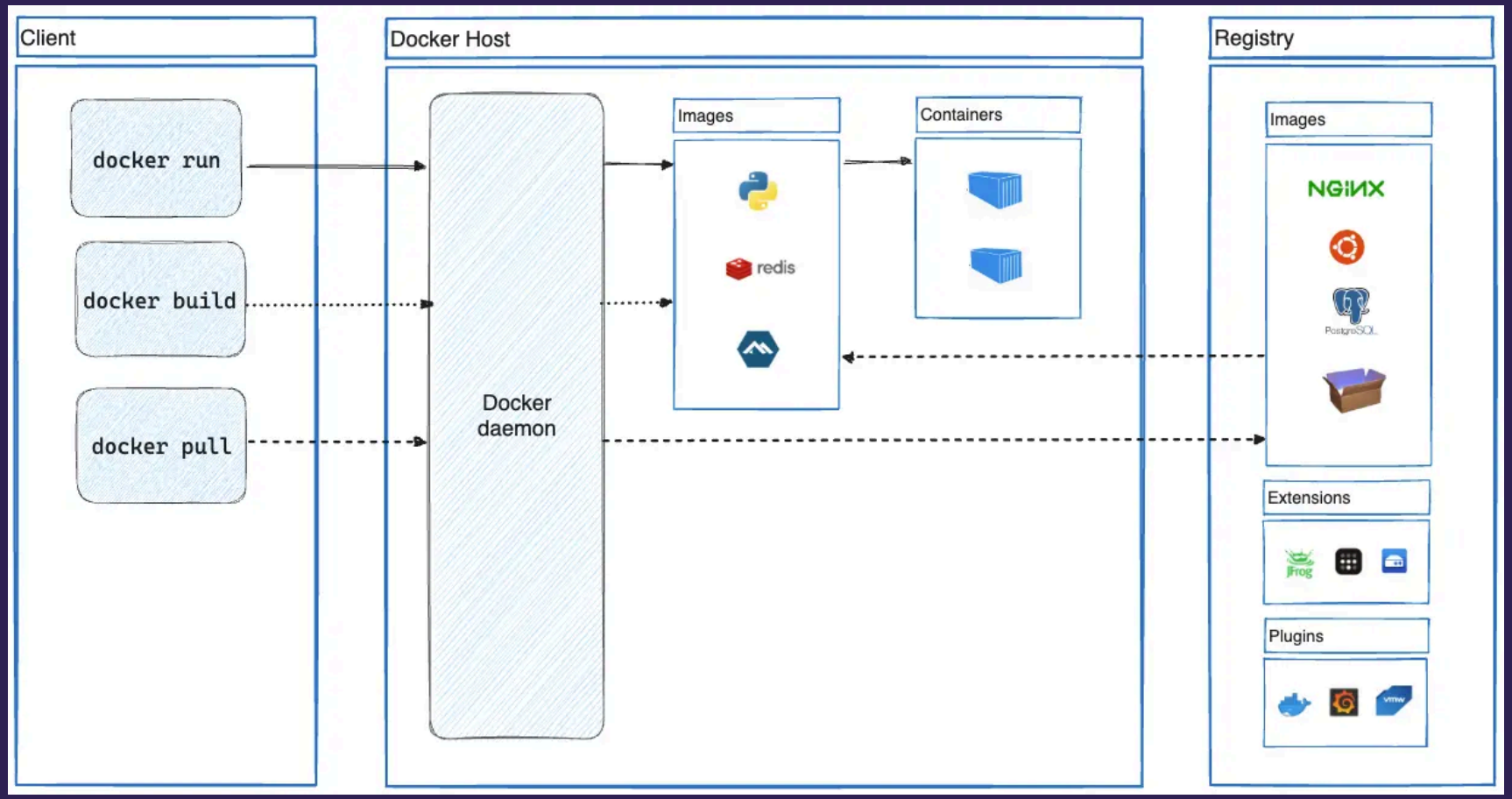
- WSL: Docker se desarrolló inicialmente para Linux.
- Hyper-V: Para contenedores Windows (Licencia Pro/Enterprise)

Instalación en MAC:

- Chips M1: Arquitectura ARM

Conceptos

- **Imágenes:** Las imágenes son plantillas de solo lectura que se utilizan para crear contenedores.
- **Contenedores:** Los contenedores son instancias en ejecución de estas imágenes.
- **Registros:** Los registros son almacenes donde se guardan las imágenes Docker.
- **Volúmenes:** Son utilizados para almacenar datos persistentes que sobreviven al ciclo de vida de un contenedor.
- **Networks:** Son utilizadas para conectar por red privadas los distintos contendores.


Client
docker run
docker build
docker pull
Docker Host
Docker daemon
Images
redis
Containers
Registry
Images
NGINX
Extensions
Plugins

Comandos básicos

- Comando básico para invocar al cliente de Docker:

```
docker [OPTIONS] COMMAND
```

- **OPTIONS**: Opciones que modifican el comportamiento del comando, como configuraciones o modos específicos.
- **COMMAND**: La acción que deseas que Docker ejecute, como `run`, `pull`, `build`, entre otros.

- Ejemplos comunes:
  - `docker run -d nginx`: Ejecuta un contenedor en segundo plano con la imagen `nginx`.
  - `docker ps -a`: Lista todos los contenedores, incluyendo los que están detenidos.
- **Nota**: Para ver una lista completa de comandos y opciones, puedes ejecutar:

```
docker --help
docker COMMAND --help
```

Descargar imágenes

```
docker [OPTIONS] pull IMAGE TAG
```

En Docker, las imágenes suelen seguir el esquema de **versionado semántico (SemVer)** en su etiqueta (`<TAG>`), el cual se estructura en el formato `MAJOR.MINOR.PATCH`:

- **MAJOR**: Cambios incompatibles o grandes, como una nueva versión con cambios significativos.
- **MINOR**: Nuevas funcionalidades que mantienen compatibilidad con versiones anteriores.
- **PATCH**: Correcciones de errores o pequeñas mejoras sin cambios en la funcionalidad.

- Ejemplo: `docker pull nginx:1.18.0`
  - Aquí, `1` es la versión principal, `18` es la versión menor y `0` es una corrección menor.
  - Esto permite a los desarrolladores seleccionar versiones específicas o actualizar a nuevas sin romper compatibilidad.

Gestionar imágenes:

```
docker image COMMAND
```

Los comandos de gestión de imágenes permiten realizar varias operaciones útiles sobre las imágenes de Docker:

- **ls**: Lista todas las imágenes en el host.
- **pull <IMAGE>**: Descarga una imagen de un registro.
- **rm <IMAGE>**: Elimina una imagen del host.

- Ejemplo:
  - `docker image ls` - Muestra todas las imágenes disponibles.

Mostrar los contenedores

```
docker ps OPTIONS
```

Este comando permite visualizar los contenedores que están actualmente en ejecución, así como aquellos que han sido detenidos:

- **-a**: Muestra todos los contenedores, incluidos los detenidos.
- **-q**: Muestra solo los IDs de los contenedores.

- Ejemplo:
  - `docker ps -a` - Muestra todos los contenedores, tanto en ejecución como detenidos.

Crear un contenedor

```
docker run OPTIONS IMAGE COMMANDS
```

Este comando se utiliza para crear y ejecutar un nuevo contenedor a partir de una imagen especificada:

- **-d**: Ejecuta el contenedor en segundo plano.
- **-p <HOST_PORT>:<CONTAINER_PORT>**: Publica un puerto del contenedor en el host.
- **--name <NAME>**: Asigna un nombre al contenedor.

- Ejemplo:
  - `docker run -d -p 8080:80 nginx` - Crea y ejecuta un contenedor de Nginx en segundo plano.

Ejecutar dentro de un contenedor

```
docker exec [OPTIONS] CONTAINER COMMAND
```

Permite ejecutar un comando específico dentro de un contenedor en ejecución:

- **-it**: Permite la interacción con el contenedor.
- Ejemplo:
  - `docker exec -it my_container /bin/bash` - Inicia una sesión de shell dentro del contenedor.

Parar un contenedor

```
docker stop OPTIONS CONTAINER
```

Este comando detiene un contenedor en ejecución de forma controlada:

- **-t <SECONDS>**: Espera un número específico de segundos antes de forzar la detención.

- Ejemplo:
  - `docker stop my_container` - Detiene el contenedor especificado.

Gestionar redes

```
docker network COMMAND
```

Permite realizar operaciones sobre las redes de Docker:

- **ls**: Lista todas las redes disponibles.
- **create <NETWORK>**: Crea una nueva red.
- **rm <NETWORK>**: Elimina una red especificada.

- Ejemplo:
  - `docker network ls` - Muestra todas las redes disponibles en el host.

Gestionar volúmenes

```
docker volume COMMAND
```

Este comando permite gestionar volúmenes en Docker, que son utilizados para persistir datos:

- **ls**: Lista todos los volúmenes existentes.
- **create <VOLUME>**: Crea un nuevo volumen.
- **rm <VOLUME>**: Elimina un volumen especificado.

- Ejemplo:
  - `docker volume ls` - Muestra todos los volúmenes disponibles.

Copiar contenido entre contenedor y host

```
docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH
```

Permite transferir archivos entre un contenedor y el sistema host:

- Esta operación es útil para recuperar logs, datos, o para transferir archivos hacia el contenedor.

- Ejemplo:
  - `docker cp my_container:/data/file.txt ./file.txt` - Copia un archivo del contenedor al host.

Ver los registros de un contenedor

```
docker logs [OPTIONS] CONTAINER
```

Este comando muestra los registros generados por un contenedor:

- **-f**: Permite seguir los registros en tiempo real.
- **--tail <N>**: Muestra las últimas N líneas de los registros.

- Ejemplo:
  - `docker logs -f my_container` - Sigue los registros del contenedor en tiempo real.

Eliminar un contenedor

```
docker rm OPTIONS CONTAINER
```

Este comando se utiliza para eliminar un contenedor que ha sido detenido:

- **-f**: Fuerza la eliminación de un contenedor en ejecución.

- Ejemplo:
  - `docker rm my_container` - Elimina el contenedor especificado.

Contenedores en Segundo Plano

Los contenedores en segundo plano (detached mode) se ejecutan en *background*, lo que permite que el terminal permanezca libre para otras tareas. Esta funcionalidad es especialmente útil en entornos de servidor, donde se pueden ejecutar múltiples servicios sin necesidad de mantener el terminal ocupado.

```
docker run -d --name=nginx-test nginx
```

Ventajas del modo en segundo plano:

- Mantiene el terminal libre para ejecutar otros comandos.
- Permite gestionar múltiples contenedores simultáneamente.
- Ideal para aplicaciones de larga duración o en producción.

Modo Interactivo

Para acceder a un contenedor que se está ejecutando en segundo plano, es necesario utilizar el comando `docker exec` para lanzar una terminal interactiva.

```
docker exec -it nginx-test /bin/bash
```

Este comando nos permitirá acceder al contenedor y realizar diversas tareas, tales como:

- **Instalación de software:** Instalar aplicaciones o herramientas necesarias dentro del contenedor.
- **Pruebas y depuración:** Ejecutar comandos para comprobar el estado del contenedor y solucionar problemas.
- **Exploración del sistema de archivos:** Navegar por la estructura de directorios y archivos del contenedor.

Puertos

Los puertos son esenciales para conectar y comunicar un contenedor con el mundo exterior, permitiendo el acceso a los servicios que se ejecutan dentro de él, como servidores web, bases de datos o APIs. Al exponer un puerto, Docker facilita la interacción entre aplicaciones externas, otros contenedores y el host.

```
docker run -d -p HOST CONTAINER IMAGE
docker run -d -p 8080:80 --name nginx-port nginx
```

> Asegúrate de que el puerto en el host no esté en uso por otra aplicación.
> Puedes exponer múltiples puertos usando varias opciones `-p` en un solo comando.

Formato del comando:

- `<HOST>:<CONTAINER>` asigna puertos entre el host y el contenedor.
- En el ejemplo `8080:80`, el puerto `8080` del host se mapea al puerto `80` del contenedor, lo que permite acceder al servicio del contenedor a través del puerto `8080` en el host.

Logs

Los registros (logs) permiten monitorear y depurar el comportamiento de las aplicaciones dentro de los contenedores, facilitando la comprensión de su ejecución y la detección de problemas o excepciones.

Para ver los logs de un contenedor en ejecución, se utiliza el siguiente comando:

```
docker logs [OPTIONS] CONTAINER
```

Ejemplo: Para obtener los logs de un contenedor llamado `nginx-port`:

```
docker logs nginx-port
```

Opciones útiles:

- `-f`: Muestra los logs en tiempo real (follow).
- `--tail`: Muestra solo las últimas líneas de los logs.

Variables de Entorno

Las variables de entorno son valores configurables que se pueden definir para un contenedor, proporcionando un medio flexible para configurar aplicaciones sin modificar su código fuente. Esto permite que las aplicaciones se comporten de manera diferente según el entorno en el que se ejecutan.

Es recomendable no incluir información sensible, como contraseñas, directamente en los comandos. Considera usar archivos de configuración o herramientas de gestión de secretos.
Puedes acceder a las variables de entorno dentro del contenedor a través del código de la aplicación, facilitando la personalización.

Ejemplo 1: Configurar credenciales para una base de datos MongoDB:

```
docker run -P -e MONGO_INITDB_ROOT_USERNAME=root \
           -e MONGO_INITDB_ROOT_PASSWORD=toor \
           --name mongo-dev -d mongo
```

Contenedores Sin Servicios

Los contenedores sin servicios en Docker son aquellos que no ejecutan un servicio de larga duración (como un servidor web o una base de datos). En su lugar, realizan tareas puntuales y luego se detienen. Estos contenedores son ideales para ejecutar tareas únicas, scripts o comandos que se completan rápidamente.

Ejemplo 1: Ejecutar un comando en un contenedor Ubuntu:

```
docker run --rm ubuntu echo "Hola desde Docker"
```

Ejemplo 2: Ejecutar un script de Python:

```
docker run --rm python:3.8 python -c "print('Hello from Docker!')"
```

Ejemplo 3: Crear un archivo comprimido utilizando tar:

```
docker run --rm -v $(pwd):/data ubuntu tar -czvf /data/mi_archivo.tar.gz /data
```

Nota: Usar la opción `--rm` permite eliminar automáticamente el contenedor una vez que ha terminado de ejecutar el comando, lo que ayuda a mantener el entorno limpio.

¿Qué es una IP?

- Una **IP** (Protocolo de Internet) es como la dirección de una casa, pero en una red.
- Permite identificar de forma única a cada dispositivo conectado a una red (ordenador, móvil, servidor, etc).
- Sin una IP, los dispositivos no podrían comunicarse entre sí.

Ejemplo de dirección IP

- Una dirección IP típica tiene este aspecto: `192.168.1.10`
- Está formada por cuatro números separados por puntos, cada uno entre 0 y 255.
- Ejemplo visual:

Dispositivo	Dirección IP
Ordenador	192.168.1.10
Impresora	192.168.1.20
Móvil	192.168.1.30

¿Cómo funciona una IP?

- Cuando un dispositivo quiere comunicarse con otro, utiliza la IP de destino.
- Es como enviar una carta: necesitas la dirección del destinatario.
- En redes locales (como en casa o en una oficina), las IP suelen empezar por `192.168.x.x` o `10.x.x.x`.
- En Internet, las IP pueden ser públicas y únicas en todo el mundo.

¿Por qué es importante la IP en Docker?

- Docker asigna una IP a cada contenedor para que puedan comunicarse entre sí.
- Puedes ver y configurar estas IPs al crear redes personalizadas.
- Entender las IPs ayuda a diagnosticar problemas de conexión entre contenedores.

¿Qué es una máscara de red?

- Una **máscara de red** es un número que define qué parte de una dirección IP corresponde a la red y qué parte a los dispositivos (hosts) dentro de esa red.
- Ayuda a los dispositivos a saber si otro está en su misma red local o si debe comunicarse a través de un router.

Ejemplo de máscara de red

- La máscara más común en redes domésticas es `255.255.255.0` (también se puede ver como `/24`).
- Ejemplo:
  - IP: `192.168.1.10`
  - Máscara: `255.255.255.0`
  - Esto significa que todos los dispositivos con IP `192.168.1.x` están en la misma red local.
- Si la máscara fuera `255.255.0.0` (`/16`), entonces todos los dispositivos con IP `192.168.x.x` estarían en la misma red.

¿Por qué es importante la máscara de red?

- Permite dividir redes grandes en redes más pequeñas (subredes).
- Ayuda a organizar y aislar dispositivos dentro de una red.
- En Docker, al crear redes personalizadas, puedes definir la máscara de red para controlar cuántos contenedores pueden estar en la misma red.

Redes

Introducción a Redes en Docker

- Las redes en Docker permiten la comunicación entre contenedores.
- Proporcionan aislamiento y control sobre cómo se comunican los contenedores.
- Las redes pueden persistir más allá de la vida de los contenedores.

Tipos de Redes:

bridge (predeterminada):

- Red privada para los contenedores que se ejecutan en el mismo host.
- Ideal para entornos de desarrollo donde los contenedores necesitan comunicarse entre sí.
- Los contenedores pueden acceder al exterior a través del gateway, pero están aislados de otros contenedores.

- El rango de IPs predeterminado para la red `bridge` es `172.17.0.0/16`.
- Docker automáticamente asigna una IP dentro de este rango cuando un contenedor se ejecuta en esta red.
- Modificar el rango por defecto del bridge: **https://docs.docker.com/engine/network/drivers/bridge**

Comandos de Ejemplo:

```
docker network create --subnet 172.20.0.0/16 my_network
docker network create --driver bridge --subnet 172.19.0.0/16 my_network_2
docker network create my_network_3
docker network create --subnet 172.20.0.0/16 my_network_4
docker network inspect NETWORK_NAME|NETWORK_ID
```

host:

- El contenedor comparte la red del host, eliminando la capa de aislamiento.
- Utiliza la red del sistema directamente, mejorando el rendimiento en aplicaciones que requieren baja latencia.
- Sin embargo, esto sacrifica el aislamiento entre contenedores.

Comando de Ejemplo:

```
docker run --network host -d nginx
```

none:

- No se asigna ninguna red al contenedor, dejándolo completamente aislado.
- Útil para pruebas de seguridad o para aplicaciones que no necesitan comunicación de red.

Comando de Ejemplo:

```
docker run --network none -d busybox top docker exec -it container_id ping google.com
```

Comparación de Tipos de Redes en Docker

Características	Aislamiento	Acceso a la Red Externa	Usos Comunes
bridge	Aislado entre contenedores y host.	A través de NAT y gateway.	Desarrollo local, múltiples contenedores en un mismo host.
host	No hay aislamiento, comparte la red del host.	Directo, sin NAT.	Contenedores de alto rendimiento, aplicaciones que requieren baja latencia.
none	Totalmente aislado, sin acceso a la red.	No tiene acceso a la	Pruebas de seguridad, contenedores que no

Volúmenes

¿Qué son los volúmenes en Docker?

Los **volúmenes** en Docker son un mecanismo para almacenar y persistir datos.

- **Persistir datos**: Los datos no se pierden al detener o eliminar un contenedor.
- **Compartir datos entre contenedores**: Permiten que varios contenedores accedan al mismo conjunto de datos.

Independientes del contenedor: Los volúmenes existen de forma separada, por lo que los datos permanecen incluso si el contenedor se elimina.

Ejemplo práctico: Persistencia de datos en MongoDB

- Si ejecutamos un contenedor de MongoDB sin un volumen, **todos los datos se perderán** si el contenedor se elimina.
- Al usar un volumen, los datos se almacenan de forma persistente en el sistema del host.

```
docker run -d -p 27017:27017 --name mongodb -v mongodb_data:/data/db mongo
```

- En este ejemplo, el volumen `mongodb_data` almacena los datos persistentes de MongoDB.
- Incluso si el contenedor se elimina, los datos quedan guardados en el volumen y pueden reutilizarse en un nuevo contenedor.

Tipos de Volúmenes en Docker

Volúmenes gestionados por Docker (Named Volumes)

- Son volúmenes creados y gestionados automáticamente por Docker, sin necesidad de configurarlos manualmente.
- Se almacenan en una ubicación predeterminada dentro del sistema de archivos del host (generalmente en `/var/lib/docker/volumes`).
- Docker asegura la persistencia de los datos, incluso si el contenedor se detiene o elimina.

- **Ventaja principal:** Los datos persisten independientemente del ciclo de vida del contenedor.
- **Uso común:** Almacenar bases de datos y otros datos que deben mantenerse a través de múltiples ciclos de vida de contenedores.

```
docker volume create volumen_nombre
docker run -d -v volumen_nombre:/data/db mongo
```

Montajes de directorios del host (Bind Mounts)

- Los **bind mounts** permiten usar directorios del host directamente dentro del contenedor.
- El contenedor puede **leer y escribir** en estos directorios, permitiendo acceso directo a los archivos del host.

- **Ventajas:**
  - Permite compartir archivos entre el host y el contenedor, útil para logs, configuraciones, bases de datos, etc.
  - Cambios realizados en el host se reflejan inmediatamente en el contenedor y viceversa.

- **Desventajas:**
  - Riesgo de corrupción de datos si el contenedor y el host no están bien sincronizados.
  - Dependencia de la estructura del sistema de archivos del host.

- **Uso común:** En entornos de desarrollo, donde se necesitan cambios inmediatos entre el código del host y el contenedor.

```
docker run -d -v ${PWD}/index.html:/usr/share/nginx/html/index.html nginx
```

Volúmenes temporales (Anonymous Volumes)

- Volúmenes creados automáticamente por Docker sin un nombre explícito.
- Estos volúmenes no tienen un nombre y **persisten** después de que el contenedor se elimina, aunque no se pueden acceder fácilmente sin un nombre.

- **Ventajas:** Útiles para almacenar datos temporales generados por un contenedor, como archivos de logs o datos temporales.
- **Desventajas:** No se pueden gestionar fácilmente y pueden acumularse si no se eliminan explícitamente.

- **Uso común:** Datos que solo deben existir durante la vida del contenedor, como archivos temporales generados durante su ejecución.

```
docker run -d -v /data/db mongo
```

Imágenes

¿Qué son las Imágenes de Docker?

- Las imágenes de Docker son plantillas de solo lectura utilizadas para crear contenedores.
- Contienen el sistema de archivos y todas las dependencias necesarias para ejecutar una aplicación.
- Se pueden compartir y distribuir fácilmente a través de registros como Docker Hub.
- Para crear imágenes personalizadas, se utiliza un `Dockerfile`, que es un archivo de texto con instrucciones para construir la imagen.

¿Qué es un Dockerfile?

- Un Dockerfile es un archivo de texto que define los pasos necesarios para construir una imagen de Docker, como instalar dependencias, copiar archivos y configurar el entorno.
- Se utiliza con el comando `docker build` para generar una imagen personalizada y reproducible.
- Un Dockerfile se construye usando una serie de instrucciones o palabras reservadas, cada una de las cuales genera una nueva capa en la imagen.
- Las capas son almacenadas de manera eficiente, y Docker reutiliza las capas que no han cambiado, lo que acelera las construcciones subsecuentes.
- La capa final de la imagen es el contenedor que se ejecutará, proporcionando el entorno listo para ejecutar la aplicación.

Instrucciones Comunes en Dockerfile

- `FROM`: Define la imagen base a partir de la cual se construye la nueva imagen.

```
FROM nginx:alpine
```

- `WORKDIR`: Cambia el directorio donde se ejecutarán los siguientes comandos.
- `ENV`: Define las variables de entorno.
- `RUN`: Ejecuta un comando dentro de la imagen durante su construcción.

```
RUN apk update && apk add --no-cache iputils nano
```

- `VOLUME`: Crea un punto de montaje para volúmenes.

```
VOLUME ["/etc/nginx/conf.d"]
```

- `COPY` / `ADD`: Copia archivos o directorios del host al contenedor.

```
COPY ./index.html /usr/share/nginx/html/index.html
```

- `EXPOSE`: Informa a Docker que el contenedor escucha en un puerto específico. No mapea automáticamente el puerto al host.

```
EXPOSE 80
```

- `CMD`: Especifica el comando que se ejecutará cuando se inicie el contenedor. Es esencial para definir el comportamiento del contenedor.

```
CMD ["nginx", "-g", "daemon off;"]
```

Más información: **https://docs.docker.com/reference/dockerfile/**

.dockerignore

- El archivo `.dockerignore` especifica qué archivos o directorios no deben ser copiados a la imagen de Docker.
- Similar al `.gitignore`, ayuda a evitar archivos innecesarios en la imagen.
- Mejora la eficiencia al reducir el tamaño de la imagen y acelera el proceso de construcción.
- Protege la seguridad al evitar incluir archivos sensibles en la imagen.

¿Cómo crear una imagen Docker?

- Para crear una imagen Docker solo necesitas un archivo llamado Dockerfile con las instrucciones para construir el entorno y la aplicación.
- Utiliza el comando docker build para generar la imagen a partir del Dockerfile y el contexto, que es la carpeta donde están los archivos necesarios.

```
docker build -t nombre-imagen:tag .
```

Opciones avanzadas de `docker build`

- Puedes usar **varios** `-t` para etiquetar la imagen con diferentes nombres/tags:

```
docker build -t miapp:latest -t miapp:v1.0 .
```

- Para indicar un `Dockerfile` con otro nombre o en otra ruta, usa `-f`:

```
docker build -f Dockerfile.dev -t miapp:dev .
```

- Puedes construir imágenes desde cualquier carpeta como contexto:

```
docker build -t miapp:prod ./directorio-con-el-dockerfile
```

Docker Compose

¿Qué es Docker Compose?

Es una herramienta para definir y ejecutar aplicaciones de múltiples contenedores a partir de un único archivo de configuración YAML.

- **Ventajas**: Simplifica la gestión de servicios, redes y volúmenes en un solo archivo, facilitando el despliegue de tu stack de aplicaciones de forma eficiente.

- **Uso**: Con un solo comando, puedes crear y arrancar todos los servicios configurados en el archivo `docker-compose.yml`, ideal para entornos de producción, desarrollo, testing y CI/CD.

¿Cómo Funciona Docker Compose?

- Docker Compose utiliza un archivo YAML (`compose.yaml`) para definir la configuración de tu aplicación y sus servicios.
- Sigue las reglas establecidas por la *Compose Specification* (**especificación completa**).
- También se soportan archivos `docker-compose.yml` para compatibilidad con versiones anteriores.

Configuración compose

En un archivo `compose.yaml`, las claves principales son:

- **services**: Define los servicios (contenedores) que forman tu aplicación. Cada servicio puede tener su propia imagen, variables de entorno, puertos, volúmenes, etc.
- **image**: Especifica la imagen de Docker que se usará para crear el contenedor del servicio. Puede ser una imagen pública, privada o una construida localmente.
- **environment**: Permite establecer variables de entorno para los servicios.

- **container_name**: Permite asignar un nombre personalizado al contenedor que se crea para el servicio, facilitando
- **ports**: Expone y mapea puertos del contenedor al host.
- **depends_on**: Indica dependencias de arranque entre servicios.
- **volumes**: Define volúmenes persistentes para almacenar datos fuera del ciclo de vida de los contenedores.
- **networks**: Permite definir redes personalizadas para que los servicios se comuniquen entre sí de forma aislada o con el exterior.

Comandos Comunes de Docker Compose

`docker compose` (**referencia**) para gestionar aplicaciones multicontenedor.

- `docker compose up`: Inicia todos los servicios definidos en el archivo.
- `docker compose down`: Detiene y elimina todos los contenedores, redes y volúmenes creados por `up`.
- `docker compose logs`: Muestra los logs de todos los servicios.
- `docker compose ps`: Lista todos los contenedores que se están ejecutando.

Puedes invocar contenedores de manera individual usando `docker compose <command> <nombre del servicio>`.

Creando un archivo `compose.yaml` básico

Creación de un contenedor con `docker run`

```
docker run \
-d  \
-p 3000:3000 \
--network course-network \
--hostname course-backend \
--name cb \
-e USE_DB=true \
-e DB_HOST=course-database \
-e DB_PORT=5432 \
-e DB_USER=postgres \
-e DB_PASS=12345678 \
-e DB_NAME=postgres \
course-backend
```

Convertir la configuración a un compose

```yaml
services:
  course-backend:
    image: course-backend
    container_name: cb
    hostname: course-backend
    networks:
      - course-network
    ports:
      - "3000:3000"
    environment:
      - USE_DB=true
      - DB_HOST=course-database
      - DB_PORT=5432
      - DB_USER=postgres
      - DB_PASS=12345678
      - DB_NAME=postgres

networks:
  course-network:
    external: true
```