CURSO DE ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL Y BIG DATA

Instalación y configuración de HADOOP

Este documento introduce los conceptos esenciales de Docker mediante ejemplos prácticos, para comprender su funcionamiento y aplicarlo a proyectos reales de desarrollo y despliegue.

JUAN FRANCISCO GARCÍA HINOJOSA

ÍNDICE.

1. Introducción

Docker es una herramienta que permite ejecutar aplicaciones dentro de contenedores. Un contenedor incluye todo lo necesario para que la aplicación funcione igual en cualquier ordenador, sin importar el sistema operativo o las versiones instaladas.

Para comprobar si Docker está instalado correctamente, podemos ejecutar:

```
docker --version          # Muestra la version de Docker instalada
docker run hello-world    # Prueba basica de funcionamiento
```

Si aparece un mensaje de bienvenida, Docker está funcionando correctamente.

¿Por qué usar Docker?

Evita los clásicos problemas de “en mi ordenador sí funciona”, reduce el tiempo de despliegue y facilita la portabilidad de aplicaciones entre distintos entornos.

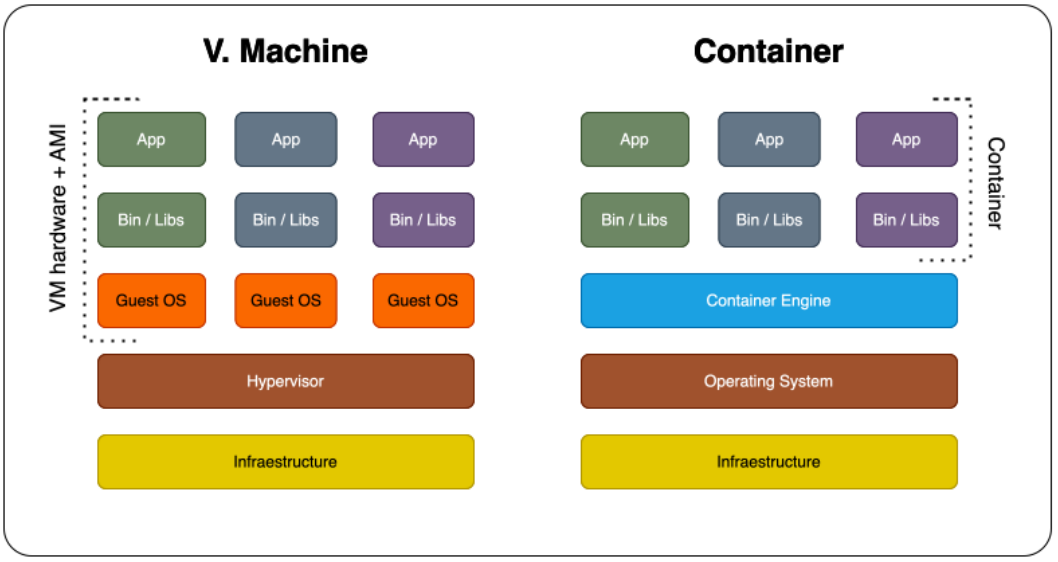

Figura 1: Comparación entre Máquina Virtual y Docker

Una MV es como un ordenador completo dentro de otro. Un contenedor es como una caja con lo necesario para ejecutar una sola aplicación.

2. Conceptos básicos

Antes de empezar a usar Docker, es importante tener muy claros cuatro conceptos fundamentales: **imagen**, **contenedor**, **volumen** y **red**. Cada uno cumple un papel diferente, pero todos trabajan juntos.

- **Imagen**: Una imagen es como una *plantilla* o una *receta* que describe todo lo necesario para ejecutar una aplicación. Contiene el sistema base (por ejemplo, Ubuntu o Alpine), las librerías, las dependencias y las configuraciones necesarias. Las imágenes son *inmutables*, es decir, no cambian una vez creadas. Cuando ejecutamos una imagen, Docker crea una copia temporal de ella (el contenedor).

  Por ejemplo, cuando usamos:

  ```
  docker pull nginx:latest
  docker pull mongo:8.0
  ```

  descargamos las imágenes oficiales con todo lo necesario para ejecutar un servidor web (Nginx) y una base de datos (MongoDB), sin tener que instalarlos en nuestro sistema operativo.

- **Contenedor**: Un contenedor es una instancia en funcionamiento de una imagen. Podemos pensar que la imagen es la receta, y el contenedor es el plato ya cocinado. Cada contenedor está completamente aislado del resto del sistema, lo que evita conflictos entre aplicaciones.

  Al ejecutar una imagen, Docker crea automáticamente un contenedor:

  ```
  # Contenedor de Nginx accesible desde el navegador
  docker run -d --name mi_nginx -p 8080:80 nginx

  # Contenedor de MongoDB con contrasea de root
  docker run -d --name mi_mongo \
  -e MONGO_INITDB_ROOT_USERNAME=admin \
  -e MONGO_INITDB_ROOT_PASSWORD=1234 mongo
  ```

  El primer comando ejecuta un servidor web Nginx al que podemos acceder desde `http://localhost:8080`. El segundo crea un contenedor con MongoDB y establece un usuario administrador y su contraseña.

- **Volumen**: Los volúmenes sirven para **guardar datos de forma persistente**. Por defecto, si eliminamos un contenedor, todos los datos internos desaparecen. Los volúmenes permiten que esa información se mantenga incluso cuando los contenedores se eliminan o actualizan.

  Podemos crear un volumen y usarlo así:

```
# Crear un volumen para MongoDB
docker volume create datos_mongo

# Usarlo al ejecutar el contenedor
docker run -d --name mi_mongo -v datos_mongo:/data/db \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=1234 mongo
```

Aquí, la carpeta `/data/db` dentro del contenedor estará conectada al volumen `datos_mongo` en el sistema local. Aunque el contenedor se borre, los datos almacenados en la base de datos se conservarán.

- **Red**: Las redes en Docker permiten que los contenedores se comuniquen entre sí de forma controlada y segura. Por ejemplo, una aplicación web (Nginx) y una base de datos (MongoDB) pueden estar en contenedores distintos, pero hablar entre ellos por medio de una red interna sin exponerse al exterior.

  Ejemplo:

```
# Crear una red interna
docker network create mi_red

# Ejecutar MongoDB en esa red
docker run -d --name mongo_db --network mi_red \
-v datos_mongo:/data/db \
-e MONGO_INITDB_ROOT_USERNAME=admin \
-e MONGO_INITDB_ROOT_PASSWORD=1234 mongo

# Ejecutar Nginx en la misma red
docker run -d --name web_nginx --network mi_red -p 8080:80 nginx
```

  Ambos contenedores están en la red `mi_red` y pueden comunicarse entre sí usando el nombre del contenedor (por ejemplo, `mongo_db`) como si fuera una dirección. Así, Nginx podría actuar como servidor frontal mientras MongoDB gestiona los datos en segundo plano.

Estos cuatro elementos son la base de todo lo que haremos con Docker. Las imágenes nos dan la base, los contenedores ejecutan, los volúmenes guardan los datos y las redes conectan los servicios.

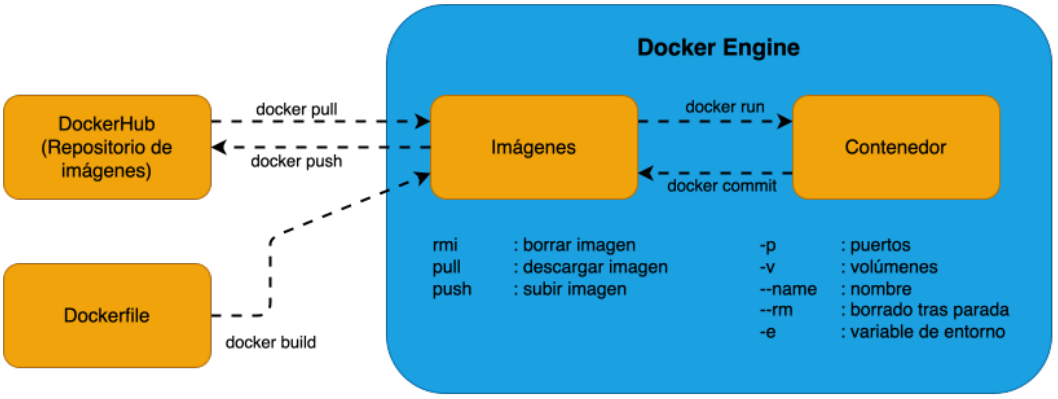

Figura 2: Relación entre Imagen, Contenedor, Volumen y Red

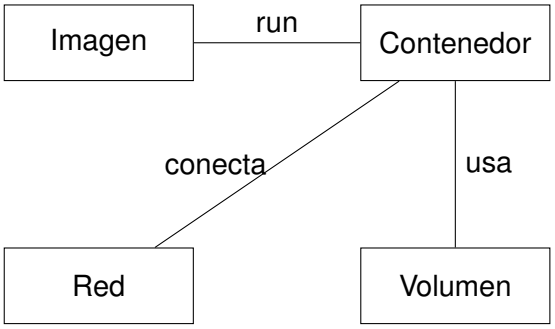

Figura 3: Esquema de funcionamiento básico de Docker

3. Creación y ejecución de contenedores

Podemos ejecutar un contenedor en segundo plano usando:

```
docker run -d --name web -p 8080:80 nginx
# Ejecuta un contenedor llamado "web" que usa nginx
# Mapea el puerto 80 del contenedor al 8080 del host
```

Comprobamos los contenedores activos:

```
docker ps        # Lista contenedores en ejecucion
docker stop web  # Detiene el contenedor "web"
docker rm web    # Elimina el contenedor
```

4. Creación de imágenes con Dockerfile

Supongamos que queremos crear una aplicación Python con FastAPI. Primero escribimos un pequeño programa llamado `main.py`:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home():
    return {"mensaje": "Hola desde FastAPI en Docker!"}
```

Creamos un `Dockerfile` en el mismo directorio:

```
FROM python:3.10-slim          # Imagen base ligera con Python
WORKDIR /app                   # Carpeta dentro del contenedor
COPY main.py .                 # Copiamos el codigo
RUN pip install fastapi uvicorn  # Instalamos dependencias
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

Construimos y ejecutamos la imagen:

```
docker build -t fastapi-demo .   # Construye la imagen
docker run -d -p 8080:80 fastapi-demo  # Ejecuta la app
```

Al abrir `http://localhost:8080`, veremos el mensaje de FastAPI.

5. Persistencia de datos con volúmenes

Cuando trabajamos con bases de datos, los datos deben persistir aunque el contenedor se elimine. Para ello usamos volúmenes:

```
docker volume create mongo_data
docker run -d --name mongo \
  -v mongo_data:/data/db \
  -p 27017:27017 mongo
# Creamos un volumen persistente llamado mongo_data
# y lo montamos dentro del contenedor MongoDB
```

Podemos inspeccionar el volumen:

```
docker volume inspect mongo_data
```

El directorio donde Docker guarda los volúmenes en Linux suele ser: `/var/lib/docker/volumes/`

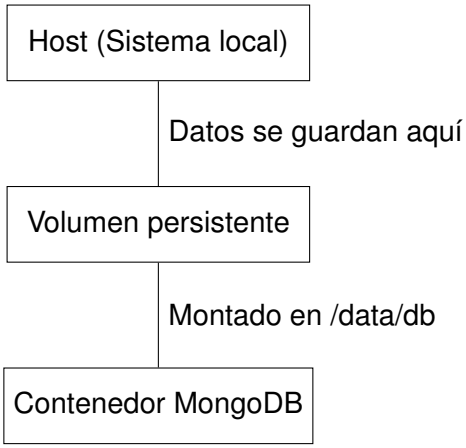

Figura 4: Persistencia de datos con volúmenes

6. Redes y comunicación entre contenedores

Docker permite que los contenedores se comuniquen entre sí usando redes personalizadas:

```
docker network create mi_red
docker run -d --name mongo --network mi_red mongo
docker run -d --name api --network mi_red fastapi-demo
# Ambos contenedores pueden comunicarse por el nombre del contenedor
```

Podemos comprobarlo con:

```
docker exec -it api ping mongo
```

Importante: instalar la utilidad `ping`

Cuando queremos comprobar la conexión entre contenedores usando el comando `ping`, puede ocurrir que aparezca un error diciendo que el comando no se encuentra.

Esto no significa que la red no funcione, sino que el contenedor no tiene instalada la herramienta `ping`. Muchas imágenes oficiales de Docker son muy ligeras y no incluyen utilidades básicas del sistema.

Para solucionarlo, podemos acceder al contenedor e instalar el paquete necesario.

Por ejemplo, si tenemos un contenedor basado en Nginx llamado `my_nginx`:

```
docker exec -it api /bin/bash
apt-get update
apt-get install inetutils-ping
```

Después de esto, podremos ejecutar `ping` sin problemas:

```
docker exec -it api ping mongo
```

7. Servidor Apache y web estática

Para servir una página web con Apache, basta con montar una carpeta local con el contenido HTML:

```
docker run -d --name web \
  -p 8080:80 \
  -v $(pwd)/html:/usr/local/apache2/htdocs \
  httpd
```

Donde la carpeta `html` contiene, por ejemplo:

```
<!DOCTYPE html>
<html>
  <head><title>Mi web en Docker</title></head>
  <body><h1>Hola desde Apache en Docker!</h1></body>
</html>
```

Antes de usar Docker Compose: comprobar la estructura de carpetas

Antes de crear y ejecutar nuestro archivo `docker-compose.yml`, es importante comprobar dónde hemos guardado los archivos de los ejercicios anteriores. Docker necesita saber exactamente en qué carpetas se encuentran el `Dockerfile`, los archivos de la aplicación y los contenidos web que vamos a montar en los contenedores.

Por ejemplo, podemos tener la siguiente estructura en nuestro ordenador:

```
Escritorio/

 docker-compose.yml

 programa_python/
    Dockerfile
    main.py

 web/
     index.html
```

En este caso:

- La carpeta `programa_python` contiene el `Dockerfile` y el programa en Python (por ejemplo, con FastAPI).
- La carpeta `web` contiene la página web estática que servirá el contenedor Apache.
- El archivo `docker-compose.yml` está en el Escritorio, al mismo nivel que ambas carpetas.

Es muy importante mantener esta estructura, ya que el archivo Compose necesita acceder correctamente a las rutas de cada servicio. Si los archivos estuvieran en otras carpetas, habría que ajustar las rutas en el bloque `build` y en los `volumes` para que Docker pueda encontrarlos.

Por ejemplo, cuando indiquemos en el `docker-compose.yml` que la API se construye a partir de la carpeta `programa_python`, lo haremos con:

```
build:
  context: ./programa_python
```

De esta forma, Docker sabrá dónde buscar el `Dockerfile` para construir la imagen y podrá montar correctamente los archivos de la aplicación.

8. Creación de Docker Compose

Docker Compose nos permite definir varios servicios en un solo archivo. Creamos un archivo llamado `docker-compose.yml`:

```
version: "3"
services:
  web:
    image: httpd
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/local/apache2/htdocs

  api:
    build:
        context: ./programa_python
    ports:
      - "8000:80"
    depends_on:
      - mongo
    networks:
      - mi_red

  mongo:
    image: mongo
    volumes:
      - mongo_data:/data/db
    networks:
      - mi_red

volumes:
  mongo_data:

networks:
  mi_red:
```

Ejecutamos todo el conjunto:

```
docker compose up -d
```

9. Ejercicios prácticos entrega en Evex

Estos ejercicios están pensados para seguir practicando lo visto en clase. Lo importante es que pruebes cosas distintas y entiendas bien cómo funciona Docker.

1. Usa una base de datos diferente

Crea un contenedor de `MySQL` o `PostgreSQL`. Configura usuario, contraseña y nombre de base de datos con variables de entorno. Comprueba que el contenedor arranca correctamente y que puedes conectarte desde una herramienta externa como **TablePlus**.

2. Explora el contenido de un contenedor

Ejecuta un contenedor de `nginx` o `python`, entra dentro de él con `docker exec -it` y mira qué hay en su sistema de archivos. Localiza los archivos principales y comprueba si hay alguna carpeta donde puedas crear un archivo temporal.

3. Prueba a modificar un volumen

Crea un contenedor con un volumen montado. Abre el contenedor, modifica o añade un archivo dentro del volumen y luego elimina el contenedor. Después, crea otro contenedor que use el mismo volumen y comprueba si el archivo sigue estando ahí.

4. Red entre dos contenedores

Crea una red con `docker network create`. Lanza dos contenedores dentro de esa red (por ejemplo, uno de `nginx` y otro de `python`). Comprueba si pueden comunicarse entre sí. Si necesitas hacer pruebas de conexión, instala el comando `ping` dentro de uno de ellos.

5. Página web simple con volumen local

Crea una carpeta local con un archivo `index.html`. Ejecuta un contenedor de `nginx` montando esa carpeta como volumen. Modifica el HTML desde tu ordenador y comprueba en el navegador cómo se actualiza automáticamente sin reiniciar el contenedor.

6. Crea una pequeña API con FastAPI

Haz una aplicación sencilla en Python con FastAPI que tenga dos rutas. Empaquétala con un `Dockerfile` y ejecuta la imagen para acceder a la API desde el navegador. No hace falta que tenga base de datos, solo que responda correctamente a las peticiones.

7. Usa Docker Compose para juntar dos servicios

Crea un archivo `docker-compose.yml` que levante una API y una base de datos (por ejemplo, `FastAPI + MongoDB`). Comprueba que ambos servicios arrancan y que están conectados a la misma red. Luego detenlos y elimínalos correctamente.

8. Limpieza del entorno

Has creado varios contenedores, redes y volúmenes, por lo tanto una vez finalizada la práctica limpia todo con los comandos de eliminación. Comprueba con `docker ps -a`, `docker images` y `docker volume ls` qué elementos quedan después de cada limpieza.