

Práctica 4B - Docker y Docker Compose

Docker

1. Utilizando sus palabras, describa qué es Docker y enumere al menos dos beneficios que encuentre para el concepto de contenedores.

Docker es una herramienta que permite empaquetar y ejecutar aplicaciones o procesos en contenedores aislados.

Beneficios de los contenedores:

- **Aislamiento:** Cada contenedor ejecuta su proceso de forma independiente.
- **Portabilidad:** Las aplicaciones pueden ejecutarse en cualquier entorno con Docker instalado.

2. ¿Qué es una imagen? ¿Y un contenedor? ¿Cuál es la principal diferencia entre ambos?

- **Imagen:** Plantilla de solo lectura que define cómo se creará un contenedor (ej: `mysql:5.7`).
- **Contenedor:** Instancia en ejecución de una imagen.

Diferencia principal: Las imágenes son estáticas (como un "molde"), mientras que los contenedores son dinámicos (en ejecución).

3. ¿Qué es Union Filesystem? ¿Cómo lo utiliza Docker?

El **Union Filesystem** permite apilar capas de archivos en una sola estructura, donde cada capa representa cambios incrementales. Docker lo usa para optimizar el almacenamiento: las imágenes se construyen en capas (ej: una capa base + capas de configuración).

4. ¿Qué rango de direcciones IP utilizan los contenedores cuando se crean? ¿De dónde la obtiene?

Por defecto, Docker asigna direcciones IP a los contenedores dentro de la red `172.17.0.0/16` (rango: `172.17.0.1` a `172.17.255.254`).

- **Origen:** Docker crea una red bridge llamada `docker0` con esta subred al instalarse.
- **Asignación dinámica:** Cada contenedor recibe una IP única dentro de este rango al iniciarse.
- **Personalización:** Pueden definirse redes custom con otros rangos usando `docker network create`.

Nota: Esto aplica si no se modifican las configuraciones de red predeterminadas de Docker.

5. ¿De qué manera puede lograrse que los datos sean persistentes en Docker? ¿Qué dos maneras hay de hacerlo? ¿Cuáles son las diferencias entre ellas?

Docker permite persistencia mediante:

1. Volúmenes nombrados (ej: `-v data:/var/lib/mysql`):

- Gestionados por Docker.
- Sobreviven al eliminar contenedores.

2. Bind mounts (montajes directos):

- Vinculan una ruta del host al contenedor.
- Dependen del sistema de archivos del host.

Diferencia: Los volúmenes nombrados son más portables, mientras que los bind mounts dependen de la estructura del host.

(Basado en ejemplos como el de MySQL en páginas 15-16).

Taller

1. Instale Docker CE (Community Edition) en su sistema operativo. Ayuda: seguir las instrucciones de la página de Docker. La instalación más simple para distribuciones de GNU/Linux basadas en Debian es usando los repositorios.

```
apt-get update
```

```
apt-get install ca-certificates curl gnupg
```

```
install -m 0755 -d /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc
```

```
chmod a+r /etc/apt/keyrings/docker.asc
```

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
```

```
https://download.docker.com/linux/debian $(. /etc/os-release && echo "$VERSION_CODENAME")  
stable" | tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
apt-get update
```

```
apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-  
plugin
```

```
sudo docker run hello-world
```

2. Usando las herramientas (comandos) provistas por Docker realice las siguientes tareas:

a. Obtener una imagen de la última versión de Ubuntu disponible. ¿Cuál es el tamaño en disco de la imagen obtenida? ¿Ya puede ser considerada un contenedor? ¿Qué significa lo siguiente: Using default tag: latest?

```
root@so:~# docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
0622fac788ed: Pull complete
Digest: sha256:6015f66923d7afbc53558d7ccffd325d43b4e249f41a6e93eef074c9505d2233
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
root@so:~# docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu latest a0e45e2ce6e6 3 weeks ago 78.1MB
hello-world latest 74cc54e27dc4 4 months ago 10.1kB
root@so:~#
```

La imagen no es un contenedor, es una plantilla para crear contenedores, `using default tag:latest` descarga la ultima imagen posible.

b. De la imagen obtenida en el punto anterior iniciar un contenedor que simplemente ejecute el comando `ls -l`

```
root@so:~# docker run --rm ubuntu ls -l
total 48
lrwxrwxrwx 1 root root 7 Apr 22 2024 bin -> usr/bin
drwxr-xr-x 2 root root 4096 Apr 22 2024 boot
drwxr-xr-x 5 root root 340 May 22 13:48 dev
drwxr-xr-x 1 root root 4096 May 22 13:48 etc
drwxr-xr-x 3 root root 4096 Apr 15 14:11 home
lrwxrwxrwx 1 root root 7 Apr 22 2024 lib -> usr/lib
lrwxrwxrwx 1 root root 9 Apr 22 2024 lib64 -> usr/lib64
drwxr-xr-x 2 root root 4096 Apr 15 14:04 media
drwxr-xr-x 2 root root 4096 Apr 15 14:04 mnt
drwxr-xr-x 2 root root 4096 Apr 15 14:04 opt
dr-xr-xr-x 164 root root 0 May 22 13:48 proc
drwx----- 2 root root 4096 Apr 15 14:11 root
drwxr-xr-x 4 root root 4096 Apr 15 14:11 run
lrwxrwxrwx 1 root root 8 Apr 22 2024 sbin -> usr/sbin
drwxr-xr-x 2 root root 4096 Apr 15 14:04 srv
dr-xr-xr-x 13 root root 0 May 22 13:48 sys
drwxrwxrwt 2 root root 4096 Apr 15 14:11 tmp
drwxr-xr-x 12 root root 4096 Apr 15 14:04 usr
drwxr-xr-x 11 root root 4096 Apr 15 14:11 var
```

c. ¿Qué sucede si ejecuta el comando `docker [container] run ubuntu /bin/bash 1`? ¿Puede utilizar la shell Bash del contenedor?

Se cierra de una.

i. Modifique el comando utilizado para que el contenedor se inicie con una terminal interactiva y ejecutarlo. ¿Ahora puede utilizar la shell Bash del contenedor? ¿Por qué?

```
docker run -it ubuntu /bin/bash
root@so:~# docker run -it ubuntu /bin/bash
root@ae8af1ffa7b0:/#
```

Si se puede usar, se puede usar porque lo iniciamos con la terminal interactiva y con bash.

ii. ¿Cuál es el PID del proceso bash en el contenedor? ¿Y fuera de éste?

El pid dentro del proceso es 1, fuera usamos un comando para verlo:

```
root@so:~# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ae8af1ffa7b0 ubuntu "/bin/bash" 5 minutes ago Up 5 minutes gracious_ramanujan
c8bbc5baa94f ubuntu "/bin/bash" 6 minutes ago Exited (0) 6 minutes ago competent_beaver
e9f028b06368 hello-world "/hello" About an hour ago Exited (0) About an hour ago
xenodochial_bassi
root@so:~# docker inspect --format '{{.State.Pid}}' gracious_ramanujan
5239
```

iii. Ejecutar el comando `lsns`. ¿Qué puede decir de los namespace?

root@so:~# lsns

```
NS TYPE NPROCS PID USER COMMAND
4026531834 time 112 1 root /sbin/init
4026531835 cgroup 112 1 root /sbin/init
4026531836 pid 111 1 root /sbin/init
4026531837 user 112 1 root /sbin/init
4026531838 uts 108 1 root /sbin/init
4026531839 ipc 111 1 root /sbin/init
4026531840 net 111 1 root /sbin/init
4026531841 mnt 107 1 root /sbin/init
4026532165 mnt 1 245 root └─/lib/systemd/systemd-udev
4026532166 uts 1 245 root └─/lib/systemd/systemd-udev
4026532167 mnt 1 271 systemd-timesync └─/lib/systemd/systemd-timesyncd
4026532208 uts 1 271 systemd-timesync └─/lib/systemd/systemd-timesyncd
4026532266 mnt 1 574 root └─/lib/systemd/systemd-logind
4026532300 uts 1 574 root └─/lib/systemd/systemd-logind
```

```
4026531862 mnt 1 26 root kdevtmpfs
4026532213 mnt 1 5239 root /bin/bash
4026532214 uts 1 5239 root /bin/bash
4026532215 ipc 1 5239 root /bin/bash
4026532216 pid 1 5239 root /bin/bash
4026532217 net 1 5239 root /bin/bash
```

- El contenedor tiene sus propios namespaces (`pid`, `net`, `uts`, etc.), distintos a los del host.

iv. Dentro del contenedor cree un archivo con nombre sistemas-operativos en el directorio raíz del filesystem y luego salga del contenedor (finalice la sesión de Bash utilizando las teclas Ctrl + D o el comando exit).

```
root@so:~# docker ubuntu run /bin/bash
docker: unknown command: docker ubuntu

Run 'docker --help' for more information
root@so:~# docker run ubuntu /bin/bash
root@so:~# docker run -it ubuntu /bin/bash
root@ae8af1ffa7b0:/# echo $$
1
root@ae8af1ffa7b0:/# touch /sistemas-operativos
root@ae8af1ffa7b0:/# exit
exit
root@so:~# docker diff gracious_ramanujan
C /root
A /root/.bash_history
A /sistemas-operativos
```

v. Corrobore si el archivo creado existe en el directorio raíz del sistema operativo anfitrión (host). ¿Existe? ¿Por qué?

- No, porque:
 - El archivo se creó en la **capa escribible** del contenedor (aislada del host).
 - Al salir (sin usar volúmenes o bind mounts), los cambios se pierden al eliminar el contenedor.

d. Vuelva a iniciar el contenedor anterior utilizando el mismo comando (con una terminal interactiva). ¿Existe el archivo creado en el contenedor? ¿Por qué?

No, no esta. El archivo se creo en un sesión efímera de la imagen de ubuntu, esa ultima capa nunca se comiteo así que se borro, porque esta en la ultima capa del proceso.

e. Obtenga el identificador del contenedor (`container_id`) donde se creó el archivo y utilícelo para iniciar con el comando `docker start -ia container_id` el contenedor en el cual se creó el archivo.

i. ¿Cómo obtuvo el `container_id` para este comando?

```
root@so:~# docker ps -a
```

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

```
7bc49bc4da81 ubuntu "/bin/bash" 6 minutes ago Exited (130) 9 seconds ago suspicious_jepsen
```

```
ae8af1ffa7b0 ubuntu "/bin/bash" 19 minutes ago Exited (0) 9 minutes ago gracious_ramanujan
```

```
c8bbc5baa94f ubuntu "/bin/bash" 21 minutes ago Exited (0) 21 minutes ago competent_beaver
```

```
e9f028b06368 hello-world "/hello" About an hour ago Exited (0) About an hour ago
```

```
xenodochial_bassi
```

ii. Chequee nuevamente si el archivo creado anteriormente existe. ¿Cuál es el resultado en este caso? ¿Puede encontrar el archivo creado?

```
root@so:~# docker start -ia gracious_ramanujan
```

```
root@ae8af1ffa7b0:/# ls
```

```
bin boot dev etc home lib lib64 media mnt opt proc root run sbin sistemas-operativos srv
```

```
sys tmp usr var
```

f. ¿Cuántos contenedores están actualmente en ejecución? ¿En qué estado se encuentra cada uno de los que se han ejecutado hasta el momento?

Ninguno en ejecución, se hallan `exited(0)` hace n minutos.

g. Elimine todos los contenedores creados hasta el momento. Indique el o los comandos utilizados.

```
docker rm -f $(docker ps -aq)
```

3. Creación de una imagen a partir de un contenedor. Siguiendo los pasos indicados a continuación genere una imagen de Docker a partir de un contenedor:

a. Inicie un contenedor a partir de la imagen de Ubuntu descargada anteriormente ejecutando una consola interactiva de Bash.

```
so@so:~$ su -
```

```
Contraseña:
```

```
root@so:~# docker run -it --name mi-contenedor-nginx ubuntu /bin/bash
```

```
root@df920812560b:/# export DEBIAN_FRONTEND=noninteractive
```

```
export TZ=America/Buenos_Aires
```

```
apt update -qq
```

```
apt install -y --no-install-recommends nginx
```

b. Instale el servidor web Nginx, <https://nginx.org/en/>, en el contenedor utilizando los siguientes comandos 2 :

```
export DEBIAN_FRONTEND=noninteractive
```

```
export TZ=America/Buenos_Aires
```

```
apt update -qq
```

```
apt install -y --no-install-recommends nginx
```

c. Salga del contenedor y genere una imagen Docker a partir de éste. ¿Con qué nombre se genera si no se especifica uno?

```
root@so:~# docker commit mi-contenedor-nginx
```

```
sha256:91ea8a35fe64f3f6642d065b7c36c0175d258a927feba93d52a36007310d8c1d
```

El nombre es random si no se especifica.

d. Cambie el nombre de la imagen creada de manera que en la columna Repository aparezca nginx-so y en la columna Tag aparezca v1.

```
root@so:~# docker images
```

```
REPOSITORY TAG IMAGE ID CREATED SIZE
```

```
<none> <none> 91ea8a35fe64 About a minute ago 135MB
```

```
ubuntu latest a0e45e2ce6e6 3 weeks ago 78.1MB
```

```
hello-world latest 74cc54e27dc4 4 months ago 10.1kB
```

```
root@so:~# docker tag 91ea8a35fe64 nginx-so:v1
```

e. Ejecute un contenedor a partir de la imagen nginx-so:v1 que corra el servidor web nginx atendiendo conexiones en el puerto 8080 del host, y sirviendo una página web para corroborar su correcto funcionamiento. Para esto:

I. En el Sistema Operativo anfitrión (host) sobre el cual se ejecuta Docker crear un directorio que se utilizará para este taller. Éste puede ser el directorio nginx-so dentro de su directorio personal o cualquier otro directorio - para los fines de este enunciado haremos referencia a éste como /home/so/nginx-so, por lo que en los lugares donde se mencione esta ruta usted deberá reemplazarla por la ruta absoluta al directorio que haya decidido crear en este paso.

```
mkdir -p /home/so/nginx-so
```

```
cd /home/so/nginx-so
```

II. Dentro de ese directorio, cree un archivo llamado `index.html` que contenga el código HTML de este gist de GitHub:

<https://gist.github.com/ncuesta/5b959fce1c7d2ed4e5a06e84e5a7efc8>.

```
curl -o index.html
```

```
https://gist.githubusercontent.com/ncuesta/5b959fce1c7d2ed4e5a06e84e5a7efc8/raw
```

III. Cree un contenedor a partir de la imagen `nginx-so:v1` montando el directorio del host (`/home/so/nginx-so`) sobre el directorio `/var/www/html` del contenedor, mapeando el puerto 80 del contenedor al puerto 8080 del host, y ejecutando el servidor `nginx` en primer plano 3 . Indique el comando utilizado.

```
docker run -d \
```

```
--name nginx-container \
```

```
-p 8080:80 \
```

```
-v /home/so/nginx-so:/var/www/html \
```

```
nginx-so:v1 \
```

```
nginx -g "daemon off;"
```

- `-d` : Ejecuta en segundo plano (detached).
- `-p 8080:80` : Mapea el puerto 80 del contenedor al 8080 del host.
- `-v /home/so/nginx-so:/var/www/html` : Monta el directorio del host en la carpeta donde Nginx sirve archivos estáticos.
- `nginx -g "daemon off;"` : Ejecuta Nginx en primer plano (requerido para contenedores Docker).

f. Verifique que el contenedor esté ejecutándose correctamente abriendo un navegador web y visitando la URL `http://localhost:8080`.

```
curl http://localhost:8080
```

g. Modifique el archivo `index.html` agregándole un párrafo con su nombre y número de alumno. ¿Es necesario reiniciar el contenedor para ver los cambios?

No, aparece solo el cambio.

h. Analice: ¿por qué es necesario que el proceso `nginx` se ejecute en primer plano? ¿Qué ocurre si lo ejecuta sin `-g 'daemon off;'`?

- Docker está diseñado para manejar **procesos en primer plano**. Si el proceso principal (en este caso, Nginx) termina, Docker considera que el contenedor "completó su tarea" y lo detiene.
- Desactiva el modo demonio de Nginx (que normalmente corre en segundo plano).
- Mantiene el proceso `nginx` en primer plano, evitando que el contenedor se cierre.

4. Creación de una imagen Docker a partir de un archivo Dockerfile. Siguiendo los pasos indicados a continuación, genere una nueva imagen a partir de los pasos descritos en un Dockerfile.

a. En el directorio del host creado en el punto anterior (/home/so/nginx-so), cree un archivo Dockerfile que realice los siguientes pasos:

i. Comenzar en base a la imagen oficial de Ubuntu.

ii. Exponer el puerto 80 del contenedor.

iii. Instalar el servidor web nginx.

iv. Copiar el archivo index.html del mismo directorio del host al directorio /var/www/html de la imagen.

v. Indicar el comando que se utilizará cuando se inicie un contenedor a partir de esta imagen para ejecutar el servidor nginx en primer plano: nginx -g 'daemon off;'. Use la forma exec 4 para definir el comando, de manera que todas las señales que reciba el contenedor sean enviadas directamente al proceso de nginx. Ayuda: las instrucciones necesarias para definir los pasos en el Dockerfile son FROM, EXPOSE, RUN, COPY y CMD.

```
# i. Usar la imagen oficial de Ubuntu como base
FROM ubuntu:latest

# ii. Exponer el puerto 80 del contenedor
EXPOSE 80

# iii. Instalar Nginx (configurando variables para evitar prompts interactivos)
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update -qq && \
    apt-get install -y --no-install-recommends nginx && \
    rm -rf /var/lib/apt/lists/*

# iv. Copiar el archivo index.html del host al contenedor
COPY index.html /var/www/html/

# v. Ejecutar Nginx en primer plano (con exec para manejo adecuado de señales)
CMD ["nginx", "-g", "daemon off;"]
```

b. Utilizando el Dockerfile que generó en el punto anterior construya una nueva imagen Docker guardándola localmente con el nombre nginx-so:v2.

```
docker build -t nginx-so:v2 -f /home/so/nginx-so/Dockerfile /home/so/nginx-so/
```

c. Ejecute un contenedor a partir de la nueva imagen creada con las opciones adecuadas para que pueda acceder desde su navegador web a la página a través del puerto 8090 del host. Verifique que puede visualizar correctamente la página accediendo a <http://localhost:8090>.

```
docker run -d --name nginx-v2 -p 8090:80 nginx-so:v2
```

d. Modifique el archivo index.html del host agregando un párrafo con la fecha actual y recargue la página en su navegador web. ¿Se ven reflejados los cambios que hizo en el archivo? ¿Por qué?

No no se ven porque la imagen copio el html en su construcción.

e. Termine el contenedor iniciado antes y cree uno nuevo utilizando el mismo comando. Recargue la página en su navegador web. ¿Se ven ahora reflejados los cambios realizados en el archivo HTML? ¿Por qué?

```
docker stop nginx-v2 && docker rm nginx-v2
```

f. Vuelva a construir una imagen Docker a partir del Dockerfile creado anteriormente, pero esta vez dándole el nombre nginx-so:v3. Cree un contenedor a partir de ésta y acceda a la página en su navegador web. ¿Se ven reflejados los cambios realizados en el archivo HTML? ¿Por qué?

```
docker build -t nginx-so:v3 -f /home/so/nginx-so/Dockerfile /home/so/nginx-so/
```

```
docker run -d --name nginx-v3 -p 8100:80 nginx-so:v3
```

Sí, los cambios hechos al html se visualizan ahora.

Docker Compose

1. Utilizando sus palabras describa, ¿qué es docker compose?

Docker Compose es una herramienta que permite definir y gestionar aplicaciones compuestas por **múltiples contenedores** (como servicios, bases de datos, APIs, etc.) mediante un archivo de configuración (`docker-compose.yml`). Simplifica el despliegue, la comunicación entre contenedores y la replicación de entornos.

Beneficios clave:

- **Centralización:** Todo se define en un solo archivo (servicios, redes, volúmenes).
- **Automatización:** Comandos como `docker compose up` inician todos los servicios a la vez.
- **Reproducibilidad:** Compartir el archivo permite replicar el entorno en cualquier sistema con Docker.

(Basado en el documento, págs. 4-5 y 13).

2. ¿Qué es el archivo compose y cual es su función? ¿Cuál es el “lenguaje” del archivo?

¿Qué es?

Es un archivo (generalmente llamado `docker-compose.yml` o `compose.yaml`) que describe:

- **Servicios:** Contenedores a desplegar (ej: `nginx` , `mysql`).
- **Redes:** Comunicación entre contenedores.
- **Volúmenes:** Almacenamiento persistente.
- **YAML** (Yet Another Markup Language): Formato legible y estructurado con indentación.

3. ¿Cuáles son las versiones existentes del archivo docker-compose.yaml existentes y qué características aporta cada una? ¿Son compatibles entre sí? ¿Por qué?

Versión	Características	Compatibilidad
1.x	Obsoleta. Sintaxis antigua sin soporte para redes/volúmenes personalizados.	✗ No compatible con V2/V3.
2.x	Introduce redes/volúmenes definibles. Soporta múltiples contenedores en un servicio (escalado).	✓ Compatible con V3 (pero no con V1).
3.x	La más usada. Optimizada para Docker Swarm y Kubernetes. Mejora en manejo de recursos (CPU/memoria).	✓ Compatible con V2 (excepto opciones específicas de Swarm).

4. Investigue y describa la estructura de un archivo compose. Desarrolle al menos sobre los siguientes bloques indicando para qué se usan:

a. services

Función: Define los contenedores (servicios) que componen la aplicación. Cada servicio puede ser una base de datos, un backend, un frontend, etc.

```
services:
  web:
```

```
image: nginx
db:
  image: mysql
```

b. build

Función: Especifica cómo construir una imagen desde un `Dockerfile` en lugar de usar una imagen preexistente.

```
services:
  app:
    build:
      context: ./directorio_dockerfile # Ruta al Dockerfile
      dockerfile: Dockerfile.dev      # Nombre personalizado del Dockerfile
```

c. image

Función: Indica la imagen Docker que se usará para el servicio (puede ser de Docker Hub o un registro privado).

```
services:
  redis:
    image: redis:alpine # Versión específica
```

d. volumes

Función: Persiste datos o comparte directorios entre el host y el contenedor, o entre contenedores.

```
services:
  db:
    volumes:
      - db_data:/var/lib/mysql
volumes:
  db_data: # Declaración del volumen nombrado
```

e. restart

Función: Define la política de reinicio del contenedor si falla.

- `no`: No reiniciar (default).
- `always`: Siempre reiniciar.
- `on-failure`: Reiniciar solo si falla (con código de error).
- `unless-stopped`: Reiniciar siempre, a menos que se detenga manualmente.

```
services:
  web:
    restart: always
```

f. depends_on

Función: Establece dependencias de inicio entre servicios (espera a que otro servicio esté "en ejecución", no necesariamente listo).

```
services:
  web:
    depends_on:
      - db # Inicia después de `db`
```

g. environment

Función: Configura variables de entorno para el servicio.

```
services:
  db:
    environment:
      - MYSQL_ROOT_PASSWORD=1234
      - MYSQL_DATABASE=app
```

h. ports

Función: Mapea puertos del contenedor al host (`HOST:CONTAINER`).

```
services:
  web:
    ports:
      - "8080:80" # Host:8080 → Contenedor:80
```

i. expose

Función: Expone puertos solo para otros servicios en la misma red (no los publica en el host).

```
services:
  api:
    expose:
      - "3000" # Accesible solo para otros contenedores
```

j. networks

Función: Define redes personalizadas para aislar o conectar servicios.

```
services:
  web:
    networks:
      - frontend
  db:
    networks:
      - backend
networks:
  frontend:
  backend:
```

5. Conceptualmente: ¿Cómo se podrían usar los bloques “healthcheck” y “depends_on” para ejecutar una aplicación Web dónde el backend debería ejecutarse si y sólo si la base de datos ya está ejecutándose y lista?

Para garantizar que un **backend** se inicie solo cuando la base de datos esté ejecutándose y lista para **aceptar conexiones**, se combinan:

1. **healthcheck**: Verifica el estado real de la base de datos.
2. **depends_on** + **condition: service_healthy**: Define la dependencia basada en el resultado del healthcheck.

```
version: "3.9"
services:
  db:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: ejemplo
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5
    ports:
      - "5432:5432"
    volumes:
```

```

- db_data:/var/lib/postgresql/data

backend:
  build: ./backend
  depends_on:
    db:
      condition: service_healthy # Espera hasta que db pase el healthcheck
  ports:
    - "3000:3000"

volumes:
  db_data:

```

6. Indique qué hacen y cuáles son las diferencias entre los siguientes comandos:

- docker compose create y docker compose up
- docker compose stop y docker compose down
- docker compose run y docker compose exec
- docker compose ps
- docker compose logs

a. docker compose create vs docker compose up

Comando	Función	Diferencia
<code>docker compose create</code>	Crea los contenedores definidos en el archivo <code>docker-compose.yml</code> sin iniciarlos.	Solo prepara los contenedores (útil para inspeccionar configuraciones antes de arrancar).
<code>docker compose up</code>	Crea y inicia los contenedores. Además, reconstruye imágenes si hay cambios en el <code>Dockerfile</code> .	Ejecuta todo el stack de servicios en primer plano (usar <code>-d</code> para modo detached).

Ejemplo:

```

docker compose create # Solo crea contenedores
docker compose up -d  # Crea, inicia y ejecuta en segundo plano

```

b. docker compose stop vs docker compose down

Comando	Función	Diferencia
<code>docker compose stop</code>	Detiene los contenedores sin eliminarlos . Los recursos (volúmenes, redes) se mantienen.	Útil para pausar servicios y retomarlos luego con <code>docker compose start</code> .
<code>docker compose down</code>	Detiene y elimina contenedores, redes y (opcionalmente) volúmenes.	Limpia completamente el entorno. Usar <code>-v</code> para borrar volúmenes: <code>docker compose down -v</code> .

Ejemplo:

```
docker compose stop      # Pausa servicios
docker compose down      # Elimina todo el stack
```

c. docker compose run vs docker compose exec

Comando	Función	Diferencia
<code>docker compose run</code>	Crea y ejecuta un nuevo contenedor temporal para un servicio (incluso si el servicio no está corriendo).	Útil para tareas únicas (ej: migraciones de bases de datos).
<code>docker compose exec</code>	Ejecuta un comando en un contenedor ya existente .	Ideal para debuggear o interactuar con servicios en ejecución.

Ejemplo:

```
docker compose run web python manage.py migrate # Nuevo contenedor para migración
docker compose exec db psql -U postgres         # Ejecuta psql en el contenedor de DB existente
```

d. docker compose ps

Función: Lista los contenedores asociados al proyecto Compose, mostrando:

- Nombres de contenedores.
- Estado (`Up`, `Exited`, etc.).
- Puertos mapeados.

Ejemplo:

Salida típica:

NAME	COMMAND	STATUS	PORTS
web	"nginx -g 'daemon...'"	Up	0.0.0.0:8080->80/tcp
db	"docker-entrypoint.s..."	Up	5432/tcp

e. docker compose logs

Función: Muestra los logs (salida estándar y errores) de los servicios.

Opciones útiles:

- `-f`: Sigue los logs en tiempo real (como `tail -f`).
- `--tail=N`: Muestra las últimas `N` líneas.
- `service_name`: Filtra logs de un servicio específico.

Ejemplo:

```
docker compose logs -f web # Muestra logs del servicio "web" en tiempo real
```

Resumen visual

Comando	Acción	Escenario típico
<code>create</code>	Crea contenedores sin iniciar	Pre-check de configuración.
<code>up</code>	Inicia todo el stack	Despliegue inicial.
<code>stop</code>	Pausa contenedores	Mantener datos para reinicio rápido.
<code>down</code>	Elimina contenedores y recursos	Limpieza completa.
<code>run</code>	Ejecuta comando en contenedor nuevo	Tareas puntuales (migraciones).
<code>exec</code>	Ejecuta comando en contenedor existente	Debug o administración.
<code>ps</code>	Lista contenedores	Ver estado del proyecto.
<code>logs</code>	Muestra salida de servicios	Diagnóstico de errores.

7. ¿Qué tipo de volúmenes puede utilizar con docker compose?
¿Cómo se declara cada tipo en el archivo compose?

En Docker Compose, existen **tres tipos principales de volúmenes** para manejar datos persistentes o compartidos. Cada tipo se declara de forma distinta en el archivo `docker-compose.yml`:

1. Volúmenes Nombrados (*Named Volumes*)

Uso:

- Almacenamiento persistente gestionado por Docker (ideal para bases de datos).
- Los datos sobreviven a la eliminación de contenedores.

Declaración:

```
services:
  db:
    image: mysql
    volumes:
      - db_data:/var/lib/mysql # Monta el volumen en el contenedor

volumes:
  db_data: # Define el volumen nombrado (Docker lo crea automáticamente)
  driver: local # Opcional: especifica el driver (por defecto 'local')
```

Características:

- Ubicación en el host: `/var/lib/docker/volumes/<nombre_volumen>`.
 - Docker gestiona su ciclo de vida (pueden borrarse con `docker volume rm`).
-

2. Bind Mounts

Uso:

- Compartir directorios específicos del host con el contenedor (ej: código fuente).
- Los cambios en el host o contenedor son inmediatos.

Declaración:

```
services:
  web:
    image: nginx
    volumes:
      - ./app:/usr/share/nginx/html # Ruta absoluta o relativa del host
```

Características:

- El host debe tener la ruta especificada (`./app` debe existir).
 - Útil para desarrollo (edición en caliente del código).
-

3. Volúmenes Temporales (*tmpfs*)

Uso:

- Almacenamiento en memoria RAM (no persiste después de reiniciar el contenedor).
- Ideal para datos sensibles o temporales (evita escritura en disco).

Declaración:

```
services:
  cache:
    image: redis
    tmpfs:
      - /tmp # Monta un volumen temporal en /tmp
```



Características:

- Máxima velocidad (RAM), pero los datos se pierden al detener el contenedor.




8. ¿Qué sucede si en lugar de usar el comando “docker compose down” utilizo “docker compose down -v/--volumes”?

La principal diferencia radica en el manejo de los volúmenes nombrados:

1. `docker compose down` (sin `-v`):

-  Elimina contenedores y redes creadas por Compose
-  **Mantiene intactos los volúmenes nombrados**
- Los datos persistentes (como bases de datos) se conservan para futuros contenedores
- Ejemplo típico: reiniciar servicios sin perder información

2. `docker compose down -v` (con `--volumes`):

-  Elimina contenedores y redes
-  **Elimina también los volúmenes nombrados declarados en el compose**
-  Todos los datos persistentes se pierden permanentemente
- Ejemplo típico: resetear completamente el entorno de desarrollo

Ejemplo visual con PostgreSQL

```
# docker-compose.yml
services:
  db:
    image: postgres
    volumes:
      - pg_data:/var/lib/postgresql/data
volumes:
  pg_data:
```

- Con `down` normal:
 - La próxima vez que hagas `up`, tu base de datos conservará todos los datos
- Con `down -v`:
 - La base de datos comenzará desde cero como nueva

Casos de uso recomendados

1. Usar SIN `-v` cuando:
 - Estás en producción
 - Quieres reiniciar servicios manteniendo datos
 - Estás desarrollando y necesitas conservar información entre sesiones
2. Usar CON `-v` cuando:
 - Necesitas empezar con datos limpios (ej: pruebas)
 - Quieres eliminar completamente rastros de un proyecto
 - Estás experimentando y quieres garantizar un estado inicial fresco

Notas importantes

1. Los **bind mounts** (rutas del host) NUNCA se eliminan, independientemente del uso de `-v`
2. Para eliminar volúmenes no declarados en el compose, debes usar `docker volume prune`
3. Siempre haz backup antes de usar `-v` en entornos importantes

Comandos relacionados útiles

```
# Ver volúmenes existentes
docker volume ls

# Eliminar volúmenes no utilizados (no asociados a contenedores)
docker volume prune

# Inspeccionar un volumen específico
docker volume inspect pg_data
```

Instalación de docker compose

Instalalo:

```
curl -L "https://github.com/docker/compose/releases/download/v2.24.7/docker-compose-$(uname
-s)-$(uname -m)" -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

Ejercicio guiado - Instanciando un Wordpress y una Base de Datos.

```

version: "3.9"

services:
  db:
    image: mysql:5.7
    networks:
      - wordpress
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    networks:
      - wordpress
    volumes:
      - ${PWD}:/data
      - wordpress_data:/var/www/html
    ports:
      - "127.0.0.1:8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress

volumes:
  db_data: {}
  wordpress_data: {}

networks:
  wordpress: {}

```

- ¿Cuántos contenedores se instancian?

2

- ¿Por qué no se necesitan Dockerfiles?

En este caso no se necesitan Dockerfiles porque el archivo `docker-compose.yml` utiliza imágenes preconstruidas directamente desde Docker Hub (`mysql:5.7` y `wordpress:latest`).

• ¿Por qué el servicio identificado como “wordpress” tiene la siguiente línea?

```
depends_on:  
- db
```

Wordpress solo se crea si la db se crea correctamente.

• ¿Qué volúmenes y de qué tipo tendrá asociado cada contenedor?

En tu archivo `docker-compose.yml`, cada contenedor tiene volúmenes asociados para **persistir datos** y **compartir archivos**. Aquí el desglose detallado:

1. Contenedor `db` (MySQL)

Volúmenes asociados:

- `db_data:/var/lib/mysql`
 - **Tipo:** Volumen con nombre (named volume).
 - **Propósito:** Persistir la base de datos de MySQL.
 - **Ubicación física:**
 - Docker gestiona el almacenamiento en `/var/lib/docker/volumes/` (en el host).
 - Puedes ubicarlo con:

```
docker volume inspect db_data
```

2. Contenedor `wordpress` (WordPress)

Volúmenes asociados:

- `wordpress_data:/var/www/html`
 - **Tipo:** Volumen con nombre (named volume).
 - **Propósito:** Persistir archivos de WordPress (plugins, temas, uploads).
- `${PWD}:/data`
 - **Tipo:** Bind mount (montaje de host).
 - **Propósito:** Compartir archivos entre el host y el contenedor.
 - **Ubicación física:**

- Usa el directorio actual del host (desde donde ejecutas `docker-compose up`).
- Ejemplo: Si `${PWD}` es `/home/usuario/proyecto`, los archivos del host se reflejarán en `/data` del contenedor.

• ¿Por que uso el volumen nombrado

```
volumes:  
- db_data:/var/lib/mysql
```

para el servicio db en lugar de dejar que se instancie un volumen anónimo con el contenedor?

Puedes identificarlo y gestionarlo fácilmente con comandos como:

```
docker volume ls  
docker volume inspect db_data  
docker volume prune # (no lo borrará si está en uso)
```

Un volumen anónimo:

- Se genera con un ID aleatorio (ej: `a1b2c3d4...`), lo que dificulta su identificación.
- Se elimina automáticamente al borrar el contenedor (a menos que uses `docker-compose down -v`).

• ¿Qué genera la línea

```
volumes:  
- ${PWD}:/data
```

en la definición de wordpress?

La definición del servicio `wordpress` genera un `bind mount` (montaje de host) con características específicas.

• ¿Qué representa la información que estoy definiendo en el bloque `environment` de cada servicio? ¿Cómo se “mapean” al instanciar los contenedores?

1. Para el servicio `db` (MySQL)

```
environment:  
  MYSQL_ROOT_PASSWORD: somewordpress
```

```
MYSQL_DATABASE: wordpress
MYSQL_USER: wordpress
MYSQL_PASSWORD: wordpress
```

Qué representa:

- Configuración inicial de la base de datos MySQL:
 - `MYSQL_ROOT_PASSWORD`: Contraseña del usuario root (administrador).
 - `MYSQL_DATABASE`: Nombre de la base de datos que se crea automáticamente.
 - `MYSQL_USER` y `MYSQL_PASSWORD`: Usuario y contraseña con permisos sobre la base de datos.

Cómo se mapea:

- Estas variables son leídas por la imagen oficial de MySQL al iniciar el contenedor.
- El contenedor ejecuta scripts internos que:
 1. Configuran el usuario root con la contraseña especificada.
 2. Crean la base de datos `wordpress`.
 3. Crean el usuario `wordpress` con acceso a esa base de datos.
- 2. Para el servicio `wordpress`

```
environment:
  WORDPRESS_DB_HOST: db
  WORDPRESS_DB_USER: wordpress
  WORDPRESS_DB_PASSWORD: wordpress
  WORDPRESS_DB_NAME: wordpress
```

Qué representa

- Configuración de conexión a la base de datos:
 - `WORDPRESS_DB_HOST`: Nombre del servicio de la base de datos (`db` , gracias a Docker DNS).
 - `WORDPRESS_DB_USER` y `WORDPRESS_DB_PASSWORD`: Credenciales para conectarse a MySQL (deben coincidir con las definidas en el servicio `db`).
 - `WORDPRESS_DB_NAME`: Nombre de la base de datos a usar (debe ser el mismo que `MYSQL_DATABASE`).

Cómo se mapea:

- WordPress (al iniciar) lee estas variables para generar su archivo de configuración `wp-config.php` .
- El valor `db` en `WORDPRESS_DB_HOST` funciona porque Docker Compose crea una red interna (`wordpress`) donde los nombres de servicio son resueltos como DNS.

• ¿Qué sucede si cambio los valores de alguna de las variables definidas en bloque “environment” en solo uno de los

contenedores y hago que sean diferentes? (Por ej: cambio SOLO en la definición de wordpress la variable WORDPRESS_DB_NAME)

Si modificas solo en WordPress la variable `WORDPRESS_DB_NAME` (ej: cambiando `wordpress` → `wp_custom`) sin actualizar el `MYSQL_DATABASE` correspondiente en MySQL, ocurrirá que el servicio wordpress va a fallar recurrentemente.

• ¿Cómo sabe comunicarse el contenedor “wordpress” con el contenedor “db” si nunca doy información de direccionamiento?

En tu archivo `docker-compose.yml`, ambos servicios están conectados a la misma red llamada `wordpress`

- Docker asigna un **nombre de host interno** a cada servicio (el nombre definido en `services:`).
 - El servicio `db` es accesible desde WordPress simplemente usando el nombre `db` (como en `WORDPRESS_DB_HOST: db`).
 - Esto funciona porque Docker incluye un **servidor DNS interno** que resuelve los nombres de los servicios.
- No es necesario especificar el puerto (`3306`) en la conexión porque:
 - WordPress usa el puerto **predeterminado** de MySQL (3306).
 - Los contenedores en la misma red pueden comunicarse directamente por sus puertos expuestos **sin mapeo al host**.

• ¿Qué puertos expone cada contenedor según su Dockerfile? (pista: navegue el sitio https://hub.docker.com/_/wordpress y https://hub.docker.com/_/mysql para acceder a los Dockerfiles que generaron esas imágenes y responder esta pregunta.)

1. Contenedor `wordpress` (imagen oficial)

- Puerto expuesto: `80/tcp`
 - Razón: WordPress funciona sobre HTTP (puerto 80 por defecto).
 - Dockerfile oficial:
La imagen hereda de `php:apache`, que expone el puerto 80 para el servidor web Apache.
- En el compose:
Mapeas `80` del contenedor al puerto `8000` del host:
ports:
- "127.0.0.1:8000:80" # Host:8000 → Contenedor:80

2. Contenedor `db` (MySQL 5.7)

- Puerto expuesto: `3306/tcp` y `33060/tcp`
 - `3306`: Puerto estándar para conexiones MySQL.
 - `33060`: Puerto para MySQL X Protocol (usado por herramientas como MySQL Shell).

• ¿Qué servicio se “publica” para ser accedido desde el exterior y en qué puerto? ¿Es necesario publicar el otro servicio? ¿Por qué?

El servicio wordpress se publica por el puerto 8000 en el host y 80 en container.

Instanciando

Seguí los pasos no hay nada que hacer literal, solamente los creas, probas el bash dentro y después los bajas.