Taller de Docker: Desplegando y Gestionando una Aplicación Multicontainer con Docker Compose

1. Explicación General del `docker-compose.yml`

Aquí vemos una parte del archivo `docker-compose.yml` proporcionado, que define una aplicación multicontainer con un backend y una base de datos MySQL.

```
version: '3.8'

services:
  backend:
    build:
      context: ./backend/v1
    container_name: backend_container
    environment:
      FLASK_APP: app.py
      DB_HOST: db
      DB_NAME: users
      DB_USER: myapp_user
      DB_PASSWORD: myapp_password
    networks:
      - app_network

  db:
    image: mysql:latest
    container_name: db_container
    environment:
      MYSQL_DATABASE: users
      MYSQL_USER: myapp_user
      MYSQL_PASSWORD: myapp_password
      MYSQL_ROOT_PASSWORD: root_password
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - app_network

networks:
  app_network:
    driver: bridge

volumes:
  db_data:
```

¿Qué hace este `docker-compose.yml`?

- **Define dos servicios:**
  - **backend**: Construye la imagen desde el directorio `./backend/v1`, asigna un nombre personalizado al contenedor y define variables de entorno para la configuración de la aplicación Flask y la conexión a la base de datos.
  - **db**: Usa la imagen oficial de MySQL, configura el nombre del contenedor y define las variables de entorno necesarias para crear la base de datos y gestionar el acceso.
- **networks**: Configura una red llamada `app_network` para que los contenedores puedan

comunicarse.
- **volumes**: Usa un volumen `db_data` para que los datos de la base de datos se mantengan incluso si el contenedor se detiene o elimina.

---

2. Paso a Paso: Ejecutar la Aplicación por Primera Vez

Pasos para ejecutar

1. **Construir y ejecutar la aplicación:**

   ```
   docker compose up --build
   ```

   - **¿Qué hace este comando?**

     - `docker compose up` : Levanta todos los servicios definidos en el archivo `docker-compose.yml` .
     - `--build` : Fuerza la construcción de las imágenes antes de iniciar los contenedores. Es útil cuando hay cambios en el código o en el `Dockerfile` .

Explicación

- **Build**: Al usar `--build` , Docker crea la imagen del backend usando el `Dockerfile` en `./backend/v1` . Esto asegura que la imagen está actualizada con los últimos cambios.
- **Crear y ejecutar contenedores**: Una vez construidas las imágenes, Docker inicia los contenedores y los conecta a la red `app_network` .

Ejercicio: Cambiar el Puerto del Frontend

1. **Problema**: El puerto predeterminado del frontend (80) está en conflicto con otro servicio en tu máquina, y necesitas cambiarlo.
2. **Tu tarea**:

   - Modifica el `docker-compose.yml` para cambiar el puerto del `frontend` al puerto asignado en el taller. Ejemplo: 10002:80

- Ejemplo:

```
  frontend:
    image: nginx:latest
    container_name: frontend_container
    ports:
      - "80:80"
    networks:
      - frontend_network
```

3. **Ejecuta la aplicación**:

```
docker compose up --build
```

4. **Verifica el cambio**: Abre tu navegador y accede al frontend en `http://tidpnube-b.rediris.es/TU_USUARIO/docker`

Pregunta para reflexionar:
¿Por qué puede ser necesario cambiar el puerto en el que se ejecuta un servicio?
Pista: Piensa en conflictos de puertos y en la gestión de múltiples servicios en una misma máquina.

---

4. Ejercicios Prácticos

Ejercicio 1: Modificar el Backend y Reconstruir

1. **Modifica algo en el código del backend** (por ejemplo, cambia un mensaje de bienvenida en `backend/v1/app.py`).
2. **Reconstruir y ejecutar**:

```
docker compose up --build
```

3. **Verificar el cambio**: Comprueba si el cambio se refleja en la aplicación.

Pregunta para reflexionar:
¿Por qué es necesario reconstruir la imagen para ver los cambios en el backend?

Ejercicio 2: Usar un Archivo `.env` para Variables de Entorno

1. **Crear un archivo** `.env` en la raíz del proyecto con el siguiente contenido (en el mismo nivel que docker-compose.yaml):

```
FLASK_APP=app.py
DB_HOST=db
DB_NAME=users
DB_USER=myapp_user
DB_PASSWORD=myapp_password
MYSQL_ROOT_PASSWORD=root_password
MYSQL_USER=myapp_user
MYSQL_PASSWORD=myapp_password
EXTERNAL_API_URL=https://api.example.com
```

2. **Modificar** `docker-compose.yml` para usar estas variable: Quitamos toda la parte en enviroment por esta otra

```
env_file:
- .env
```

3. **Ejecutar la aplicación**:

```
docker compose down
docker compose up
```

4. **Explicación**: Ahora, las variables de entorno se gestionan de forma centralizada en el archivo `.env`, lo que facilita la configuración y el despliegue en diferentes entornos (desarrollo, producción, etc.).

Pregunta para reflexionar:
¿Por qué es mejor gestionar las variables de entorno en un archivo separado?

Ejercicio 3: Comprobación de Persistencia de Datos

1. **Ejecutar la aplicación**:

```
docker compose up
```

2. **Crear algunos datos** en la base de datos (puedes acceder a la ruta del taller de tu usuario: **https:/tidpnube-b.rediris.es/TU_USUARIO/docker/usuario**).

3. Comprueba el usuario en: **https:/tidpnube-b.rediris.es/TU_USUARIO/docker/data**

4. **Detener y eliminar los contenedores**:

```
docker compose down
```

5. **Volver a iniciar la aplicación**:

```
docker compose up
```

6. **Comprobar los datos**: Verifica que los datos que creaste siguen estando en la base de datos.

Pregunta para reflexionar:
¿Por qué los datos persisten incluso después de detener y eliminar los contenedores?

---

4. Configuración con Múltiples Redes

Ejemplo Completo del `docker-compose.yml` con Múltiples Redes

```
version: '3.8'

services:
  backend1:
    build:
      context: ./backend/v1
    container_name: backend1_container
    environment:
      FLASK_APP: app.py
      DB_HOST: db
      DB_NAME: users
      DB_USER: myapp_user
      DB_PASSWORD: myapp_password
    networks:
      - app_network
      - frontend_network

  backend2:
    build:
      context: ./backend/v2
    container_name: backend2_container
    environment:
      FLASK_APP: app.py
      DB_HOST: db
      DB_NAME: users
      DB_USER: myapp_user
      DB_PASSWORD: myapp_password
    networks:
      - app_network

  db:
    image: mysql:latest
    container_name: db_container
    environment:
```

```
      MYSQL_DATABASE: users
      MYSQL_USER: myapp_user
      MYSQL_PASSWORD: myapp_password
      MYSQL_ROOT_PASSWORD: root_password
    volumes:
      - db_data:/var/lib/mysql
    networks:
      - app_network

  frontend:
    image: nginx:latest
    container_name: frontend_container
    ports:
      - "80:80"
    networks:
      - frontend_network

networks:
  app_network:
    driver: bridge
  frontend_network:
    driver: bridge

volumes:
  db_data:
```

Explicación

- **Dos backends**: `backend1` y `backend2`, conectados a redes diferentes.
- **frontend**: Está en `frontend_network`, lo que significa que no puede comunicarse directamente con `backend2` sin hacer algunos cambios.

Ejercicio 4: Cambios para Conectividad

Previo Añadimos la red fronted al docker-compose:

```
networks:
  app_network:
  frontend_network:
```

Y ahora al servicio de front le dejamos solo con la red de frontend:

```
frontend:
    build:
      context: ./nginx
    container_name: nginx_container
    environment:
      BACKEND_NAME: backend
    ports:
      - "8080:80"
    depends_on:
      - backend
    networks:
      - frontend_network
```

1. **Problema**: El `frontend` no puede comunicarse con `backend2` porque están en redes diferentes.
2. **Tu tarea**:
    - Modifica `docker-compose.yml` para conectar `frontend` con `backend`.
    - **Ejemplo**:

      ```
      backend:
      build:
      context: ./backend/v1
      container_name: backend_container
      env_file:
      - .env
      networks:
      - app_network
      - frontend_network
      ```

3. **Prueba la conectividad**: Comprueba si el `frontend` ahora puede comunicarse con ambos backends.

Desafío Adicional

1. Cambia la configuración del `frontend` en la variable de entorno para que use `backend2` en lugar de `backend`.
2. Reflexiona sobre lo que sucede y por qué.

Solución y Resultados

Resultado Esperado

- Después de conectar el `frontend` a ambas redes, debería poder comunicarse tanto con `backend1` como con `backend2`.
- Tambien podemos tener algunos de los backends en la red de frontend
- La persistencia de datos debe funcionar correctamente gracias al volumen `db_data`.