



# Práctica 6.1 - Dockerización del despliegue de una aplicación Node.js

## Introducción

En este caso vamos a Dockerizar una aplicación escrita en Node.js que hace uso de una API para gestionar personas en un libro de direcciones que guardará en una base de datos PostgreSQL.

## ¿Por qué *dockerizar*?

Si uno trata de informarse, encontrará [múltiples y variadas razones](#) para dockerizar nuestras aplicaciones y servicios.

Por citar sólo algunas:

- 1. Configuración rápida del entorno en local para el equipo de desarrollo:** si todos los servicios están implementados con contenedores, es muy rápida la configuración de dicho entorno.
- 2. Evita el clásico "en mi máquina funciona":** gran parte de los problemas de desarrollo provienen de la propia configuración que los integrantes del equipo de desarrollo tienen de su entorno. Con los servicios en contenedores, esto queda solucionado en gran medida.
- 3. Despliegues más rápidos**
- 4. Mejor control de versiones:** como ya sabéis, se puede etiquetar (tags), lo que ayuda en el CONTROL DE VERSIONES.
- 5. Rollbacks más fáciles:** puesto que se tienen las cosas mas controladas por la versión, es más fácil revertir el código. A veces, simplemente apuntando a su versión de trabajo anterior.
- 6. Fácil configuración de múltiples entornos:** como hacen la mayoría de los equipos de desarrollo, se establece un entorno local, de integración, de puesta en escena (preprod) y de producción. Esto se hace más fácil cuando los servicios están en contenedores y, la mayoría de las veces, con sólo un cambio de VARIABLES DE ENTORNO.

**7. Apoyo de la comunidad:** existe una fuerte comunidad de ingenieros de software que continuamente contribuyen con grandes imágenes que pueden ser reutilizadas para desarrollar un gran software. ¿Por qué reinventar la rueda, no?

## Despliegue con Docker

En primer lugar, si eliminastéis el repositorio en su momento, debéis volver a clonarlo en vuestra Debian, en caso contrario obviad este paso:

```
$ git clone https://github.com/profeciclo/DAW_practica_Docker_Compose.git
```

Ahora, puesto que la aplicación ya viene con el `Dockerfile` necesario dentro del directorio para construir la imagen y correr el contenedor, vamos a estudiar su contenido.



### "Tarea"

Completa este Dockerfile con las opciones/directivas adecuadas, leed los comentarios y podéis apoyaros en la teoría, en [este cheatsheet](#), en [este otro](#) o en cualquiera que encontréis.

```
_____ node:18.16.0-alpine3.17 #(1)

_____ mkdir -p /opt/app #(3)

_____ /opt/app #(2)

_____ src/package.json src/package-lock.json . #(4)

_____ npm install #(5)

_____ src/ . #(6)

_____ 3000 #(8)

_____ ["npm", "run", "start:dev"] #(9)
```

1. Con \_\_\_\_\_ indicamos que vamos a utilizar la imagen de Docker Hub oficial de Node, en su versión 18.16.0 junto con Alpine, una diminuta distribución Linux
2. \_\_\_\_\_ define el directorio sobre el que se ejecutarán las subsiguientes instrucciones del `Dockerfile`
3. \_\_\_\_\_ ejecuta un comando en una nueva capa de la imagen (podemos tener varios comandos \_\_\_\_\_)
4. \_\_\_\_\_ como su nombre indica, copia los archivos que le indiquemos dentro del contenedor, en este caso `package.json`

Recordemos que `package.json` cumplía ciertas funciones importantes:

- Centraliza la forma de interactuar con la aplicación por medio de definición de scripts (indica comandos que podemos correr dentro de nuestro proyecto, asociándolos a una palabra clave para que npm (o yarn) los reconozca cuando queramos ejecutarlos.)
  - Gestiona de una forma clara y sencilla las dependencias necesarias para que la aplicación pueda funcionar correctamente.
5. Con otro \_\_\_\_\_ ejecutamos el ya conocido comando que nos instala las dependencias que se indican en el archivo que hemos copiado en el paso anterior, el `package.json`
  6. Copiamos todos los archivos de nuestro directorio `/src` al contenedor
  7. Con \_\_\_\_\_ le indicaremos el usuario con el que correrá el contenedor
  8. \_\_\_\_\_ nos permite documentar que puertos están expuestos o a la escucha en el contenedor (sólo será accesible desde otros contenedores)

9. Y finalmente `docker exec` nos permite ejecutar un comando **dentro** del contenedor. En este caso iniciamos la aplicación.
10. Este comando `docker exec` de un `Dockerfile` permite ejecutar comandos dentro del contenedor, en este caso crea los directorios para la aplicación



### "Nota"

En Linux, cuando queremos hacer referencia al directorio actual, lo hacemos con un punto

`.`

Si dentro de nuestro directorio actual tenemos una carpeta llamada `prueba`, podemos hacer referencia a ella como `./prueba`, ya que el `.` hace referencia precisamente al directorio donde nos encontramos

Así pues, tener nuestra aplicación corriendo es cuestión de un par de comandos.

Hacemos un build de la imagen de Docker. Le indicamos que ésta se llama `librodirecciones` y que haga el build con el contexto del directorio actual de trabajo, así como del Dockerfile que hay en él:

```
$ docker build -t librodirecciones .
```

Y por último, iniciamos el contenedor con nuestra aplicación. Ahora sí, con la opción `-p`, le indicamos que escuche conexiones entrantes de cualquier máquina en el puerto 3000 de nuestra máquina anfitrión que haremos coincidir con el puerto 3000 del contenedor (`-p 3000:3000`). Y con la opción `-d` lo haremos correr en modo demonio, en background:

```
$ docker run -p 3000:3000 -d librodirecciones
```

Tras esto sólo queda comprobar que al intentar acceder desde nuestra máquina a la aplicación: `http://IP_Maq_Virtual:3000` se produce un error de conexión.



### "Tarea"

Documenta, incluyendo capturas de pantallas, el proceso que has seguido para realizar el despliegue de esta nueva aplicación, así como el resultado final.

Esto sirve para ilustrar un punto importante de los contenedores: poseen su propia red. La aplicación, por defecto, intenta buscar la base de datos en nuestro localhost pero, técnicamente, está en otro host (su contenedor).

A pesar de que todos los contenedores corren en la misma máquina, cada uno es considerado un host diferente y por eso la aplicación falla al conectar.

Podríamos utilizar los comandos [network](#) de Docker para solucionar el asunto. En lugar de eso, introduciremos el concepto de Docker Compose para administrar contenedores.

## Docker Compose

Docker Compose es una herramienta para gestionar aplicaciones multicontenedor. En Linux tiene que ser instalado por separado, consulta [su documentación](#) para ello.

Docker Compose puede:

- Iniciar y detener múltiples contenedores en secuencia.
- Conectar contenedores utilizando una red virtual.
- Manejar la persistencia de datos usando Docker Volumes.
- Establecer variables de entorno.
- Construir o descargar imágenes de contenedores según sea necesario.

Docker Compose utiliza un archivo de definición YAML para describir toda la aplicación. En nuestro caso:

```
version: "3.9"
services:
  postgres:
    image: postgres:latest
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - '5432:5432'
    volumes:
      - addressbook-db:/var/lib/postgresql/data

  addressbook:
    build:
      context: .
    environment:
      DB_SCHEMA: postgres
      DB_USER: postgres
      DB_PASSWORD: postgres
      DB_HOST: postgres
    depends_on:
      - postgres
    ports:
      - '3000:3000'

volumes:
  addressbook-db:
```

Así las cosas, para levantar nuestra infraestructura basada en contenedores no tenemos más que hacer:

```
$ docker compose run addressbook npm run migrate
```

Esto creará las tablas necesarias en la base de datos.

Y construiremos nuestros contenedores a partir de las imágenes:

```
$ docker compose up --build -d
```

Una vez construidas las imágenes, podemos levantar los contenedores:

Podéis correr unos tests para comprobar que la aplicación funciona correctamente con:

```
$ docker compose run addressbook npm test
```



### "Tarea"

Probad que la aplicación junto con la BBDD funciona correctamente. El funcionamiento de la API es:

- `GET /persons/all` muestra todas las personas en el libro de direcciones
- `GET /persons/1` muestra la persona con el id 1
- `PUT /persons/` añade una persona al libro de direcciones
- `DELETE /persons/1` elimina a la persona con el id 1

Ejemplos:

```
curl -X PUT http://IP_APLICACION:3000/persons -H 'Content-Type: application/json' -d  
  
curl -X GET http://localhost:3000/persons -H 'Content-Type: application/json'
```



### "Recordatorio"

Deberás modificar el grupo de seguridad de tu instancia EC2 de AWS para permitir el tráfico de entrada al puerto 3000, que es el que utiliza por defecto nuestra aplicación de Node.js.

## Referencias

[Dockerización](#)

[Los beneficios de utilizar Docker y contenedores a la hora de programar](#)

Dockerizing

Github