Educación IT

Trabajo de investigación

GUIA PASO A PASO:
Desarrollar un Build en Docker y Configurar un Entorno Local para Correr una Aplicación Docker-Compose

Curso:

Bootcamp Devops Engineer

Alumno: Ing. Nelson González Escalante

Profesor: Matías Anoniz

Noviembre 18, 2024

Tabla de contenido

Introducción.

En el contexto actual del desarrollo de software, contar con entornos de trabajo consistentes y replicables es fundamental para garantizar la eficiencia y colaboración dentro de los equipos. Este proyecto tiene como objetivo la creación de un entorno de desarrollo local para una aplicación basada en el framework **NestJS**, utilizando tecnologías modernas como **Docker** y **Docker Compose**.

El desarrollo de este entorno permitirá a los miembros del equipo trabajar de manera unificada, eliminando problemas de configuración y asegurando que todos utilicen la misma versión de la aplicación y sus dependencias. Además, al integrar un contenedor para **MongoDB**, se facilita la conexión con la base de datos sin necesidad de configuraciones adicionales.

Por medio de este proyecto, se busca aprender y aplicar conocimientos sobre la creación de contenedores Docker, la escritura de archivos Dockerfile, y la configuración de múltiples servicios mediante Docker Compose y la comunicación entre ellos. Estos conocimientos son esenciales para cualquier estudiante interesado en DevOps y en el desarrollo de software moderno, ya que permiten optimizar tanto el tiempo como los recursos al desarrollar aplicaciones.

Finalmente, este trabajo no solo refuerza habilidades técnicas, sino que también fomenta buenas prácticas de documentación y colaboración, preparando a los participantes para desafíos reales en la industria tecnológica.

Detalles de los Entregables

Para cumplir con los requisitos del proyecto, se han definido los siguientes entregables, cada uno enfocado en asegurar la funcionalidad y la comprensión del entorno desarrollado:

1. Código Fuente

- **Dockerfile**: Archivo que contiene todas las instrucciones necesarias para construir la imagen Docker de la aplicación NestJS solicitad, asegurando que sea ejecutable en cualquier entorno.
- **docker-compose.yml**: Archivo que define la configuración de los servicios para levantar la aplicación NestJS junto con una base de datos MongoDB, facilitando la ejecución del entorno local con un único comando.
- **Configuraciones adicionales**: Cualquier ajuste necesario en los archivos fuente de la aplicación para integrarla correctamente con Docker.

2. Documentación

- **Manual de uso**: Explicación clara y detallada de los pasos para ejecutar el proyecto en un entorno local. Incluye instrucciones sobre cómo construir y ejecutar los contenedores utilizando Docker y Docker Compose.
- **Diagrama de arquitectura**: Representación gráfica de alto nivel que muestra la interacción entre la aplicación NestJS y MongoDB, destacando cómo Docker Compose facilita esta integración.

- **Notas técnicas**: Descripción de las decisiones tomadas durante la implementación (por ejemplo, elección de imágenes base, configuración de puertos, y uso de variables de entorno).

3. Evidencia de Pruebas

- **Capturas de pantalla**: Imágenes que demuestren la ejecución exitosa de los contenedores y el acceso a la aplicación desde un navegador o cliente HTTP.
- **Pruebas de conexión**: Resultados que certifiquen la interacción adecuada entre la aplicación y la base de datos MongoDB.
- **Logs relevantes**: Fragmentos de salida de los comandos **`docker-compose up`** y **`docker logs`** que confirmen la correcta inicialización de los servicios.

4. Repositorio en GitHub

- Un repositorio público o privado con el código fuente, los archivos de configuración y la documentación.
- Debido a que aún no cuento con experiencia en el manejo de **GitHub** para la gestión de repositorios, decidí realizar la subida del proyecto completo en un único paso. Para ello, utilicé la funcionalidad de carga directa que ofrece GitHub, lo que me permitió agregar todos los archivos del proyecto al repositorio de manera manual.

Estos entregables buscan garantizar que el entorno propuesto sea funcional, reproducible y bien documentado, cumpliendo con los estándares esperados en un contexto académico y profesional.

Requisitos

Primer Paso: Elaboración del Dockerfile

1. **Comprender los requisitos del proyecto** Lo primero que hice fue leer cuidadosamente documento proporcionado para el desafío No. 5 y entender sus necesidades. Identifiqué que el archivo Dockerfile debía contener las instrucciones para instalar dependencias, compilar la aplicación y ejecutarla en un entorno de producción.
2. **Elegir una imagen base adecuada** Decidí usar la imagen oficial de Node.js como base para el contenedor, específicamente la versión 16, porque después de investigar un poco descubrí que es compatible con la mayoría de las aplicaciones modernas y ofrece estabilidad. Esta decisión debvería asegurar que la aplicación funcione sin problemas en cualquier entorno Docker.

```
FROM node:16
```

3. **Definir el directorio de trabajo** Para organizar mejor los archivos en el contenedor, creé un directorio de trabajo llamado `/app`. Esto ayuda a mantener un entorno limpio y estructurado.

```
WORK /app
```

4. **Copiar los archivos necesarios para instalar dependencias** Copié el archivo *package.json* y, si existía, el archivo *package-lock.json* al contenedor. Estos archivos contienen la lista de dependencias necesarias para que la aplicación funcione.

```
COPY package*.json ./
```

5. **Instalar las dependencias** Utilicé el comando *`npm install`* para instalar todas las dependencias definidas en *`package.json`*. Esto asegura que el entorno del contenedor tenga todo lo necesario para ejecutar la aplicación.

```
RUN npm install
```

6. **Copiar el código fuente** Después de instalar las dependencias, copié el resto del código fuente de la aplicación al directorio de trabajo del contenedor. Esto

incluye los archivos de configuración, controladores, servicios, y otros componentes de la aplicación.

```
COPY . .
```

7. **Compilar la aplicación** Como la aplicación NestJS necesita ser compilada antes de ejecutarse, añadí un comando para ejecutar *`npm run build`*. Esto genera los archivos necesarios en el directorio *`dist`*.

```
RUN npm run build
```

8. **Exponer el puerto de la aplicación** La aplicación utiliza por defecto el puerto *`3000`*, por lo que expuse este puerto para que pueda ser accesible desde fuera del contenedor.

```
EXPOSE 3000
```

9. **Definir el comando de inicio** Finalmente, configuré el comando que se ejecutará cuando el contenedor se inicie. En este caso, opté por usar *npm run start:prod* para ejecutar la aplicación en modo producción.

```
CMD [“npm”, “run”, “start:prod”]
```

10. **Guardar el archivo** Guardé el archivo como *Dockerfile* en la raíz del proyecto, asegurándome de que no tuviera ninguna extensión.

Con estos pasos, completé la creación del Dockerfile, asegurándome de que cubra todas las necesidades de la aplicación NestJS y que sea reutilizable para cualquier miembro del equipo.

Tabla 1: Código Fuente del Dockerfile

```
# Usar una imagen de Node.js como base
FROM node:16

# Crear el directorio de trabajo en el contenedor
WORKDIR /app
```

```
# Copiar package.json y package-lock.json para instalar
dependencias
COPY package*.json ./

# Instalar dependencias
RUN npm install

# Copiar el resto del código fuente de la aplicación
COPY . .

# Compilar la aplicación NestJS
RUN npm run build

# Exponer el puerto en el que corre la aplicación (por defecto
3000 en NestJS)
EXPOSE 3000

# Comando para iniciar la aplicación
CMD ["npm", "run", "start:prod"]
```

Fuente: Trabajo “Desafío No. 5”

Segundo Paso: Elaboración del docker-compose

1. **Comprender la estructura del proyecto**

   Antes de iniciar, analicé que la aplicación requiere dos servicios principales: la aplicación NestJS y una base de datos MongoDB. Esto me llevó a identificar las

configuraciones necesarias para que ambos servicios funcionen correctamente y puedan comunicarse entre sí.

2. **Definir la versión del archivo Docker Compose**
   Empecé mi archivo especificando la versión de Docker Compose que iba a usar. Elegí la versión 3 porque es ampliamente compatible y estable para este tipo de configuraciones.

```
version: '3'
```

3. **Configurar los servicios**

   Creé una sección llamada *`services`*, que es donde definí los detalles para cada uno de los contenedores necesarios en el proyecto.

4. **Configurar el servicio para la aplicación NestJS**
   Dentro de *`services`*, añadí el servicio *`app`* para la aplicación NestJS. Aquí están los pasos que seguí para configurarlo:
   - **Especificar la construcción**: Indiqué que el servicio *`app`* se construiría desde el Dockerfile ubicado en la carpeta actual (*`context: .`*) y que usaría el archivo *`Dockerfile`*.

```
app:
```

```
build:
  context: .
  dockerfile: Dockerfile
```

- **Mapear los puertos**: Configuré los puertos para que el puerto *3000* del contenedor se mapee al puerto *3000* de mi máquina local.

```
ports:
  - "3000:3000"
```

- **Establecer dependencias**: Añadí la dependencia de MongoDB para que Docker Compose asegure que el servicio *mongo* se inicie antes que la aplicación.

```
depends_on:
  - mongo
```

- **Configurar las variables de entorno**: Añadí la variable *MONGO_URI* para definir la conexión a MongoDB desde la aplicación.

```
environment:
  - MONGO_URI=mongodb://mongo:27017/nestdb
```

5. **Configurar el servicio para MongoDB**

   Creé un segundo servicio llamado *mongo* para configurar la base de datos MongoDB:

   - **Especificar la imagen de MongoDB**: Utilicé la imagen oficial *mongo:latest* del Docker Hub para asegurarme de que tenga la versión más reciente.

```
mongo:
  image: mongo:latest
```

   - **Mapear los puertos**: Configuré los puertos para que el puerto 27017 del contenedor MongoDB se mapee al puerto *27017* de mi máquina local.

```
ports:
  - "27017:27017"
```

- **Definir la base de datos inicial**: Añadí una variable de entorno para configurar automáticamente una base de datos llamada *nestdb* al iniciar el contenedor.

```
  environment:
  MONGO_INITDB_DATABASE: nestdb
```

6. **Guardar el archivo**
   Guardé el archivo como *docker-compose.yaml* en la raíz del proyecto, asegurándome de que no tuviera errores de indentación, ya que YAML es sensible al formato.
7. **Verificar el funcionamiento**
   Probé la configuración ejecutando el siguiente comando en la terminal:

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

Esto levantó ambos servicios, y validé que la aplicación NestJS estuviera disponible en http://localhost:3000 y que MongoDB funcionara correctamente en el puerto 27017.

Siguiendo estos pasos, completé la elaboración del archivo *docker-compose.yaml*, asegurándome de que sea funcional y cumpla con las necesidades del entorno de desarrollo.

Tabla 2: Código Fuente del docker-compose.yaml

```
version: '3'
services:
  app:
    Build:
      # La carpeta actual
      context: .
      dockerfile: Dockerfile
    Ports:
      # Mapea el puerto de la app a localhost
      - "3000:3000"
    depends_on:
      - mongo
    Environment:
      # Conexión a MongoDB en el contenedor
      - MONGO_URI=mongodb://mongo:27017/nestdb

  mongo:
    image: mongo:latest
    ports:
      - "27017:27017"
    Environment:
```

```
    # Base de datos predeterminada
    MONGO_INITDB_DATABASE: nestdb
```

Tercer Paso: Pruebas exitosas

Configuración de la Instancia en Multipass

1. **Instalación de Multipass**
   Instalé Multipass en mi máquina con Windows siguiendo las instrucciones oficiales. Esto me permitió crear y gestionar máquinas virtuales de Ubuntu de forma sencilla.
2. **Creación de una instancia de Ubuntu**
   Utilicé el siguiente comando para crear una nueva instancia de Ubuntu con Multipass, asignándole un nombre descriptivo (`docker-env`) y especificando recursos básicos (2 CPU, 4 GB de RAM y 10 GB de almacenamiento):

```
multipass launch --name docker-env --cpus 2 --mem 4G --disk 10G
```

3. **Conexión a la instancia**
   Me conecté a la instancia utilizando Multipass para instalar las herramientas necesarias para Docker y Docker Compose:

```
multipass shell docker-env
```

4. **Instalación de Docker y Docker Compose**
   Dentro de la instancia, instalé Docker y Docker Compose para garantizar que el entorno estuviera listo para las pruebas:

```
sudo apt update && sudo apt install -y docker.io docker-compose
```

5. **Creación de una carpeta compartida**
   Para facilitar el trabajo entre mi máquina con Windows y la instancia de Ubuntu, monté una carpeta compartida entre ambas. Esto me permitió editar los archivos del proyecto en **Visual Studio Code** desde Windows y probarlos directamente en la máquina virtual.

   En Windows, seleccioné una carpeta de trabajo (`C:\desafio5`).

   Monté esta carpeta en la instancia de Multipass con el siguiente comando:

```
multipass mount /desafio5 docker-env
```

6. **Acceso a los archivos desde Ubuntu**
   En la instancia de Ubuntu, accedí a la carpeta compartida montada en `/desafio5`, donde ya estaban disponibles los archivos del proyecto (`Dockerfile` y `docker-compose.yaml`).

Trabajo con Visual Studio Code

1. **Edición de archivos**
   Utilicé Visual Studio Code en mi máquina con Windows para editar los archivos del proyecto dentro de la carpeta compartida. Esta integración me permitió aprovechar herramientas como la sintaxis resaltada y extensiones específicas para Docker.

2. **Pruebas en Ubuntu**
   Una vez editados los archivos, ejecuté las pruebas directamente desde la instancia de Ubuntu para garantizar que los contenedores funcionaran correctamente.

Reto: Problema de Permisos al Ejecutar Docker Compose

Durante el desarrollo del proyecto, enfrenté un problema técnico que inicialmente no había previsto: al intentar ejecutar el comando `docker-compose up --build` para levantar los servicios definidos, me encontré con un error relacionado con los permisos. El mensaje indicaba que no se podía acceder al socket de Docker debido a una restricción de permisos (`PermissionError: [Errno 13] Permission denied`).

Este inconveniente surgió porque mi usuario no tenía los privilegios necesarios para comunicarse con el daemon de Docker. Aunque inicialmente intenté resolverlo ejecutando el comando con `sudo`, entendí que esta no era una solución adecuada a largo plazo, ya que afectaría la fluidez del flujo de trabajo y podría ocasionar problemas de permisos con los archivos generados por Docker.

Solución Implementada

Después de investigar y analizar el problema, descubrí que la solución más efectiva era agregar mi usuario al grupo `docker`. Este grupo permite a los usuarios ejecutar comandos de Docker sin necesidad de privilegios de superusuario. A continuación, detallo los pasos que seguí para resolver el reto:

1. **Verifiqué si el grupo `docker` existía en mi sistema**
   Confirmé que el grupo estaba configurado correctamente, lo cual es fundamental para asignar permisos al socket de Docker.
2. **Agregué mi usuario al grupo `docker`**
   Utilicé el siguiente comando:

```
sudo usermod -aG docker nelson
```

   Esto permitió que mi usuario tuviera acceso directo al daemon de Docker.

3. **Cerré sesión y la inicié nuevamente**
   Este paso fue necesario para que los cambios de permisos surtieran efecto.

4. **Verifiqué los permisos del socket de Docker**

   Me aseguré de que el archivo `/var/run/docker.sock` estuviera configurado para permitir acceso al grupo `docker`:

```
ls -l /var/run/docker.sock
```

Confirmé que los permisos eran correctos, ya que se mostraba algo similar a:

```
srw-rw---- 1 root docker ...
```

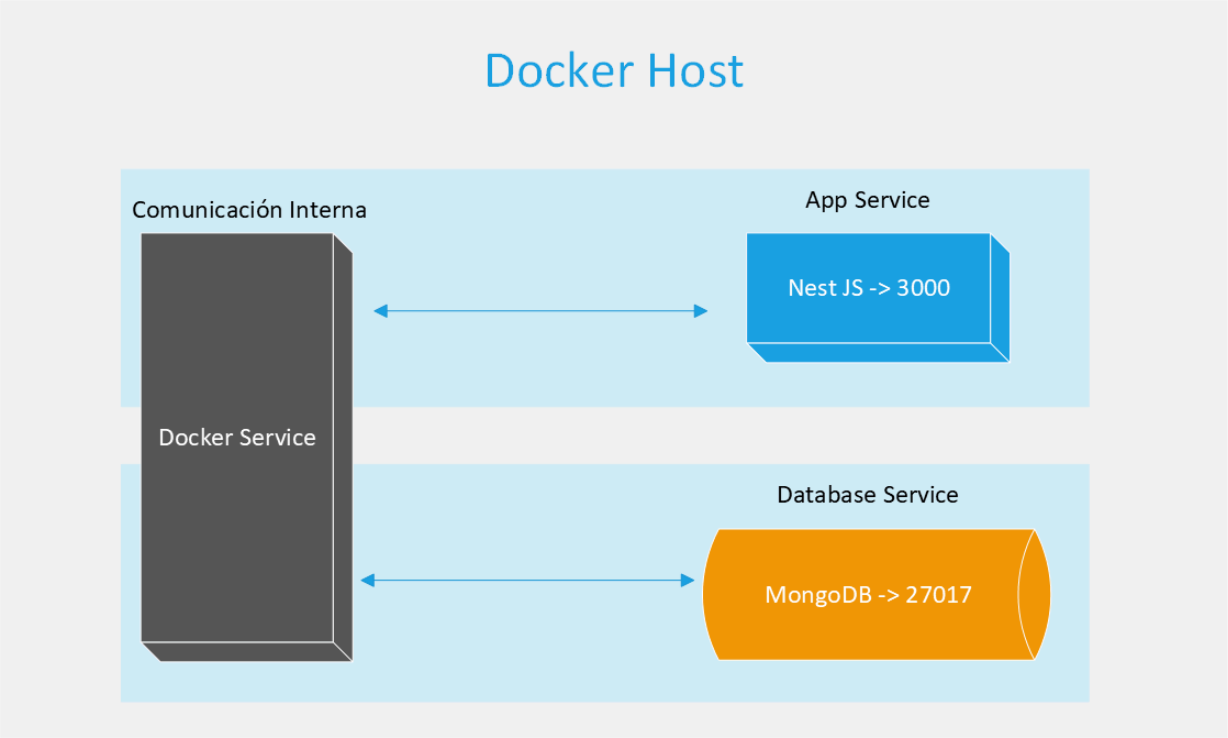
5. **Probé nuevamente el comando**

   Finalmente, volví a ejecutar `docker-compose up --build`, y esta vez los servicios se levantaron correctamente sin necesidad de usar `sudo`.

Resolver este problema fue una experiencia enriquecedora, ya que me permitió entender cómo Docker gestiona los permisos en el sistema y la importancia de configurar correctamente el entorno de trabajo. Si bien este desafío representó un contratiempo, también fue una oportunidad para aprender sobre aspectos fundamentales de la administración de sistemas, como la gestión de grupos y permisos. Este conocimiento será valioso en futuros proyectos y en la resolución de problemas similares en contextos más avanzados.

Diagrama

Figura 7: Diagrama de alto nivel



El diagrama representa un sistema basado en contenedores Docker que utiliza un servicio de aplicación (*App Service*) y un servicio de base de datos (*Database Service*) configurados dentro de un host Docker. A continuación, detallo los elementos del diagrama y su función:

1. Docker Host

- Representado como el contenedor principal que incluye todos los servicios Docker.
- Actúa como la máquina anfitriona que ejecuta los contenedores dentro de un ambiente aislado

2. Docker Service

- Es el motor de Docker que gestiona la ejecución y comunicación de los contenedores dentro del host.
- Representado como un bloque gris, permite la coordinación de los servicios internos mediante una red virtual.

3. App Service

- Representado por un bloque azul y etiquetado como **NestJS -> 3000**.
- Este contenedor ejecuta una aplicación basada en NestJS y expone el puerto `3000`, que es donde la aplicación está accesible.
- Tiene una flecha de doble dirección hacia el **Docker Service**, indicando comunicación interna mediante la red de Docker.

4. Database Service

- Representado por un bloque naranja y etiquetado como **MongoDB -> 27017**.
- Este contenedor ejecuta MongoDB como base de datos, que expone el puerto `27017`.
- También tiene una flecha de doble dirección hacia el **Docker Service**, lo que indica que está configurado para comunicarse con el servicio de aplicación a través de la red interna.

5. Comunicación Interna

- Representada por las flechas bidireccionales entre los servicios.
- Utiliza la red interna configurada en `docker-compose.yml`, lo que asegura que ambos contenedores puedan intercambiar datos sin exponer puertos al exterior del host.
- Esto permite una interacción segura y eficiente entre la aplicación y la base de datos.

Relación entre los Componentes

- **App Service** depende de **Database Service** para operar correctamente, ya que necesita conectarse a MongoDB para almacenar y recuperar datos.
- Ambos servicios están orquestados mediante Docker Compose, asegurando que se levanten en el orden correcto y utilicen la misma red virtual.

Conclusión

El desarrollo de este proyecto representó un desafío significativo y una valiosa oportunidad de aprendizaje. Desde la configuración inicial del entorno Docker hasta la integración de múltiples servicios con Docker Compose, cada etapa me permitió profundizar en habilidades técnicas esenciales para el desarrollo de software moderno. Además, enfrentar y resolver problemas, como el manejo de permisos en Docker, reforzó mi capacidad para investigar y aplicar soluciones prácticas.

Crear un entorno reproducible y funcional para una aplicación NestJS conectada a MongoDB no solo me ayudó a comprender la importancia de la portabilidad en entornos de desarrollo, sino también a valorar las herramientas y tecnologías que simplifican estos procesos. Ahora reconozco el impacto que tiene un entorno bien configurado en la colaboración y productividad de los equipos de desarrollo.

En general, este proyecto consolidó mi interés por la gestión de contenedores y la infraestructura como código. Estoy seguro de que las lecciones aprendidas aquí serán fundamentales para abordar proyectos más complejos en el futuro y continuar desarrollándome como profesional en el área de tecnología.