Pregunta:

Estoy preparando docker para para el entorno de trabajo. No sé si crear una imagen para la api y otra para la aplicación y luego unirlas en un fichero yaml o crear una única imagen única y lanzar ambas partes desde desde ahí, ya que ambas utilizan node 20.19.0. ¿Qué me recomiendas?

Dame también el código del los ficheros Dockerfile y yaml que tendría que usar

Respuesta

¡Excelente pregunta! Es una duda muy común al empezar con Docker y la respuesta tiene implicaciones importantes en la arquitectura de tu aplicación.

**Sin lugar a dudas, la recomendación es crear dos imágenes separadas (una para la API y otra para la aplicación) y luego unirlas usando un fichero docker-compose.yml.**

Aunque ambas usen Node.js, tratarlas como una sola imagen es un antipatrón que va en contra de los principios fundamentales de la contenerización y los microservicios.

### ¿Por Qué Dos Imágenes Separadas es la Mejor Opción?

1. **Separación de Responsabilidades (Single Responsibility Principle):** Cada contenedor debe hacer una sola cosa y hacerla bien. Tu API tiene la responsabilidad de servir datos. Tu aplicación de frontend tiene la responsabilidad de presentar la interfaz de usuario. Mezclarlos en un solo contenedor complica todo.
2. **Escalabilidad Independiente:** Esta es la razón más importante. En el futuro, es muy probable que tu API necesite más recursos que el frontend, o viceversa. Con contenedores separados, puedes escalar los servicios de forma independiente. Por ejemplo, puedes lanzar 5 contenedores de la API para manejar la carga y solo 2 del frontend. Con una sola imagen, tendrías que escalar el monolito completo, lo cual es ineficiente.
3. **Desarrollo y Despliegue Desacoplados:** El equipo de frontend puede hacer cambios, construir su imagen y desplegarla sin afectar para nada al backend. Y viceversa. Esto agiliza enormemente el ciclo de desarrollo e integración continua (CI/CD).
4. **Simplicidad y Mantenimiento:** Tendrás dos Dockerfile mucho más simples y fáciles de entender y mantener que un único Dockerfile complejo que intente construir ambas aplicaciones y luego use un gestor de procesos como pm2 para lanzar ambos servicios.
5. **Optimización del Tamaño de la Imagen:** La imagen del backend no necesita tener las dependencias de React (node\_modules del frontend) y la imagen del frontend no necesita las dependencias del backend (como el driver de MySQL). Esto resulta en imágenes más pequeñas, rápidas y seguras.
6. **Resiliencia:** Si el contenedor del frontend se cae por un error, la API puede seguir funcionando sin problemas, y viceversa.

### Código de los Ficheros Docker

Aquí tienes la estructura de archivos recomendada y el código para cada fichero.

#### **Estructura de Directorios**

Primero, asegúrate de que tu estructura de proyecto se vea así, con los Dockerfile dentro de sus respectivas carpetas y el docker-compose.yml en la raíz.

/mi-proyecto-gestion/  
|-- /mi-api-gestion/ <-- Tu backend NestJS  
| |-- src/  
| |-- Dockerfile <-- Dockerfile para el Backend  
| `-- package.json  
|-- /mi-app-gestion/ <-- Tu frontend Next.js  
| |-- src/  
| |-- Dockerfile <-- Dockerfile para el Frontend  
| `-- package.json  
`-- docker-compose.yml <-- El orquestador que une todo

#### **1. Dockerfile para el Backend (API NestJS)**

Este fichero irá dentro de la carpeta mi-api-gestion. Utiliza una construcción en múltiples etapas para crear una imagen de producción ligera.

**mi-api-gestion/Dockerfile**

Dockerfile

# ---- Etapa 1: Builder ----  
# Usa la imagen completa de Node para instalar dependencias y construir la aplicación  
FROM node:20.19.0-alpine AS builder  
  
WORKDIR /usr/src/app  
  
# Copia los ficheros de dependencias y las instala  
# Esto aprovecha el cache de Docker para no reinstalar si no cambian  
COPY package\*.json ./  
RUN npm install  
  
# Copia el resto del código fuente de la aplicación  
COPY . .  
  
# Construye la aplicación para producción  
RUN npm run build  
  
# ---- Etapa 2: Production ----  
# Usa una imagen base de Node más ligera para la ejecución  
FROM node:20.19.0-alpine  
  
WORKDIR /usr/src/app  
  
# Copia solo las dependencias de producción desde la etapa de builder  
COPY --from=builder /usr/src/app/node\_modules ./node\_modules  
# Copia la aplicación compilada desde la etapa de builder  
COPY --from=builder /usr/src/app/dist ./dist  
  
# Expone el puerto en el que la API va a correr  
EXPOSE 3001  
  
# El comando para iniciar la aplicación en producción  
CMD [ "node", "dist/main" ]

#### **2. Dockerfile para el Frontend (App Next.js)**

Este fichero irá dentro de la carpeta mi-app-gestion. También usa una construcción multi-etapa y aprovecha la función standalone de Next.js para una imagen final súper optimizada.

**mi-app-gestion/Dockerfile**

Dockerfile

# ---- Etapa 1: Builder ----  
FROM node:20.19.0-alpine AS builder  
  
# Habilitar la salida standalone en Next.js  
ENV NEXT\_TELEMETRY\_DISABLED 1  
  
WORKDIR /app  
  
COPY package\*.json ./  
RUN npm install  
  
COPY . .  
  
# Construye la aplicación Next.js para producción  
RUN npm run build  
  
# ---- Etapa 2: Runner (Producción) ----  
FROM node:20.19.0-alpine  
  
WORKDIR /app  
  
# Habilitar la salida standalone en Next.js  
ENV NEXT\_TELEMETRY\_DISABLED 1  
ENV PORT 3000  
  
# Copia la salida optimizada 'standalone' de la etapa de builder  
COPY --from=builder /app/.next/standalone ./  
# Copia los assets estáticos  
COPY --from=builder /app/.next/static ./.next/static  
# Copia los assets públicos  
COPY --from=builder /app/public ./public  
  
# Expone el puerto en el que la App va a correr  
EXPOSE 3000  
  
# Comando para iniciar el servidor de Next.js optimizado  
CMD ["node", "server.js"]

#### **3. Fichero docker-compose.yml (El Orquestador)**

Este fichero irá en la **raíz de tu proyecto** (mi-proyecto-gestion/). Define los tres servicios: la base de datos, el backend y el frontend, y cómo se comunican entre sí.

Este docker-compose.yml está **configurado para un entorno de desarrollo**, utilizando volúmenes para el hot-reloading (recarga en caliente) del código.

**docker-compose.yml**

YAML

version: '3.8'  
  
services:  
 # Servicio de la Base de Datos MySQL  
 mysql-db:  
 image: mysql:8.0  
 container\_name: mysql-db-container  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: 'tu\_contraseña\_de\_mysql' # ¡Usa la misma que en el .env del backend!  
 MYSQL\_DATABASE: 'nombre\_de\_tu\_bd\_existente' # ¡Usa el mismo que en el .env!  
 ports:  
 - "3306:3306" # Expone el puerto de MySQL a tu máquina local  
 volumes:  
 - db-data:/var/lib/mysql # Persiste los datos de la BD  
  
 # Servicio del Backend (API NestJS)  
 backend-api:  
 container\_name: backend-api-container  
 build:  
 context: ./mi-api-gestion # Ruta al Dockerfile del backend  
 dockerfile: Dockerfile  
 ports:  
 - "3001:3001"  
 volumes:  
 - ./mi-api-gestion:/usr/src/app # Sincroniza el código para desarrollo  
 - /usr/src/app/node\_modules # Evita que el node\_modules local sobreescriba el del contenedor  
 command: npm run start:dev # Ejecuta en modo desarrollo con hot-reload  
 depends\_on:  
 - mysql-db # No arrancar hasta que la BD esté lista  
 env\_file:  
 - ./mi-api-gestion/.env # Carga las variables de entorno desde el fichero .env  
  
 # Servicio del Frontend (App Next.js)  
 frontend-app:  
 container\_name: frontend-app-container  
 build:  
 context: ./mi-app-gestion # Ruta al Dockerfile del frontend  
 dockerfile: Dockerfile  
 ports:  
 - "3000:3000"  
 volumes:  
 - ./mi-app-gestion:/app  
 - /app/node\_modules  
 command: npm run dev  
 depends\_on:  
 - backend-api # Opcional, pero bueno para asegurar el orden de arranque  
  
volumes:  
 db-data: # Define el volumen para la persistencia de datos

### **Instrucciones Finales y Ajustes Clave**

1. ¡IMPORTANTE! Actualiza la conexión a la base de datos:  
   Dentro de la red de Docker, los contenedores se comunican usando el nombre del servicio. Por lo tanto, debes hacer un cambio en el fichero .env de tu API.  
   En mi-api-gestion/.env, cambia DB\_HOST:  
   Fragmento de código  
   # Cambiar 'localhost' por el nombre del servicio de la BD en docker-compose.yml  
   DB\_HOST=mysql-db
2. Para levantar todo el entorno:  
   Abre una terminal en la raíz de tu proyecto (mi-proyecto-gestion) y ejecuta:  
   Bash  
   docker-compose up --build  
   * --build fuerza la reconstrucción de las imágenes si has hecho cambios en los Dockerfile.
   * La primera vez tardará un poco en descargar las imágenes base y construir todo. Las siguientes veces será mucho más rápido.

Verás los logs de los tres contenedores (base de datos, backend y frontend) en tu terminal. Una vez que todo esté en marcha, podrás acceder:

* **Frontend:** http://localhost:3000
* **Backend (API Docs):** http://localhost:3001/api-docs
* **Base de Datos:** Accesible desde tu máquina local en el puerto 3306 con cualquier cliente de MySQL.