Búsqueda y Manejo de Imágenes Docker

1. Búsqueda de Imágenes con `docker search`

- **Buscar una imagen en Docker Hub**:

  ```
  docker search node
  ```

  Este comando buscará imágenes relacionadas con Node.js en Docker Hub y mostrará los resultados.
- **Obtener una imagen en Docker Hub**:

  ```
  docker pull node
  ```

  Este comando obtendra la ultima versión de la imagen node.

2. Operaciones sobre Imágenes

- **Inspeccionar una Imagen (`docker inspect`)**:

  ```
  docker inspect node
  ```

  Este comando proporciona detalles sobre la imagen de Node.js, como capas, configuración, etc.
- **Historial de una Imagen (`docker history`)**:

  ```
  docker history node
  ```

  Muestra el historial de la imagen de Node.js, incluyendo los cambios realizados en cada capa.
- **Eliminar una Imagen (`docker rmi`)**:

  ```
  docker rmi [IMAGE_ID]
  ```

  Elimina una imagen específica del sistema local. Reemplaza `[IMAGE_ID]` con el ID de la imagen que deseas eliminar.

Parte 2: Creación y Optimización de Dockerfiles con una Aplicación NodeJS

Código Básico de Node.js

Vamos a crear una aplicación web simple que responda con un mensaje de bienvenida.

- Crear el directorio del proyecto
  En tu terminal, crea un nuevo directorio para tu proyecto y navega a él:

```
mkdir thirdlab
cd thirdlab
```

- Inicializar nuevo proyecto Node.JS
  Inicializa el proyecto con **NPM** para crear un package.json

```
npm init -y
```

- Crea el archivo de la aplicación NodeJS
  Crea un archivo llamado `index.js` con el siguiente contenido:

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('¡Bienvenido a mi aplicación NodeJS en Docker!');
});

app.listen(port, () => {
  console.log(`Aplicación escuchando en http://localhost:${port}`);
});
```

- Añade Express, un framework de servidor web para NodeJS:

```
npm install express --save
```

Creación de un Dockerfile Optimizado

Ahora, creamos un `Dockerfile` que construya esta aplicación en un entorno Docker. Utilizaremos una construcción multietapa para minimizar el tamaño de la imagen final.

Dockerfile:

```
# Etapa de construcción
FROM node:14 AS builder
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .

# Etapa de producción
FROM node:14-alpine
WORKDIR /usr/src/app
COPY --from=builder /usr/src/app .
EXPOSE 3000
USER node
CMD ["node", "index.js"]
```

Explicación del Dockerfile

- **Etapa de Construcción**:
  - Utilizamos una imagen base de Node.js (`node:14`) para instalar las dependencias y construir la aplicación.
  - `WORKDIR` establece el directorio de trabajo en el contenedor.
  - `COPY package*.json ./` copia los archivos `package.json` y `package-lock.json` (si existe) en el contenedor.
  - `RUN npm install` instala las dependencias.
  - `COPY . .` copia todos los archivos restantes de la aplicación al contenedor.
- **Etapa de Producción**:
  - Cambiamos a una imagen base más liviana (`node:14-alpine`) para la etapa de producción.
  - Copiamos los archivos construidos desde la etapa de construcción con `COPY --from=builder /usr/src/app . .`
  - Exponemos el puerto 3000.
  - Cambiamos al usuario `node` , que es un usuario no root proporcionado por la imagen base de Node.js.
  - `CMD ["node", "index.js"]` especifica el comando para ejecutar la aplicación.

> **Nota:**
> **node:14:** Esta imagen se basa en la imagen oficial de Debian (una popular distribución de Linux). Incluye una gran cantidad de paquetes preinstalados, lo que facilita el despliegue de aplicaciones complejas que dependen de varios paquetes y bibliotecas de Linux. Por otro lado, debido a todas estas funcionalidades adicionales, la imagen tiene un tamaño bastante grande.

> **node:14-alpine:** Esta imagen se basa en Alpine Linux, que es una distribución de Linux muy ligera y pequeña. Las imágenes de Alpine son mucho más pequeñas que las imágenes de Debian, lo que puede acelerar el despliegue y reducir los requisitos de almacenamiento y memoria.

Construir y Ejecutar la Imagen

- **Construir la Imagen**:

```
docker build -t lab03 .
```

- **Ejecutar el Contenedor**:

```
docker run -d -p 3000:3000 lab03
```

Para demostrar el uso de un usuario no root en un contenedor, puedes seguir los siguientes pasos. Estos te permitirán verificar que el contenedor está siendo ejecutado con un usuario diferente al root.

Demostración del Uso de Usuario No Root en Contenedor

1. Crear un Dockerfile con Usuario No Root

Primero, necesitas un `Dockerfile` que especifique un usuario no root. Asegúrate de tener tu aplicación Node.js y los archivos necesarios (como `package.json` y `index.js`).

- Crea un nuevo archivo llamado `Dockerfile.nonroot` con el siguiente contenido

Dockerfile:

```
FROM node:14
# Crea un usuario no root y cambia a él
RUN useradd -m nonrootuser
USER nonrootuser
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "index.js"]
```

2. Construir la Imagen y Ejecutar el Contenedor

- **Construir la Imagen**:

```
docker build -t lab03-nonroot -f Dockerfile.nonroot .
```

- **Ejecutar el Contenedor**:

```
docker run -d -p 3000:3000 lab03-nonroot
```

3. Verificar el Usuario Dentro del Contenedor

- **Ingresar al Contenedor**:

```
docker exec -it [CONTAINER_ID] /bin/bash
```

- **Verificar el Usuario Actual**:

```
whoami
```

Este comando debería mostrar `nonrootuser` o el nombre del usuario no root que hayas creado.