Programación Multimedia y Dispositivos Móviles

Documentación API dockerizada

Servidor Backend con Express y MongoDB

Autor:

Santi Martínez

26 de noviembre de 2025

Índice

1. Introducción

1.1. Contexto del proyecto

Este proyecto aborda la dockerización de una API REST desarrollada con Node.js y Express, que utiliza MongoDB como sistema de gestión de base de datos. La API implementa funcionalidades de gestión de usuarios y grupos.

La aplicación se compone de dos servicios principales que deben funcionar de manera coordinada: el servidor de aplicación que expone los endpoints REST y la base de datos MongoDB que persiste la información.

La dockerización de estos componentes permite encapsular cada servicio en contenedores independientes, facilitando su gestión, despliegue y mantenimiento.

1.2. Objetivos de la dockerización

Los principales objetivos para realizar la dockerización de la API son los siguientes:

- **Consistencia:** Permite la ejecución del entorno en cualquier ordenador preparado para ejecutar Docker; es decir, funciona de manera autónoma, llevando todo lo necesario en su interior y operando desde ahí, de forma similar a un caballo de Troya.
- **Aislamiento de componentes:** Cada servicio (API y MongoDB) se ejecuta en su propio contenedor con su propio sistema de archivos, procesos y red, evitando conflictos de dependencias.
- **Portabilidad:** Permitir que la aplicación se ejecute de manera consistente en cualquier sistema operativo (Windows, macOS, Linux).
- **Simplificación del despliegue:** Reducir el proceso de instalación y configuración.
- **Escalabilidad:** Permite crear varias copias de un servicio para que se pueda ejecutar en varios lugares al mismo tiempo.
- **Gestión de dependencias:** Encapsular todas las dependencias que usa la API (mongoose, express, dotenv, etc ...) de la aplicación dentro de la imagen Docker, garantizando que siempre se mantengan correctas.

1.3. Tecnologías utilizadas

El conjunto de tecnologías utilizado es el siguiente:

- **Node.js 20:** Entorno de ejecución de JavaScript del lado del servidor.
- **Express.js:** Framework de Node para la realización de servidores.
- **MongoDB 6:** Base de datos NoSQL orientada a documentos.
- **Mongoose:** ODM (Object Document Mapper) que permite que Node.js se comunique y gestione datos en MongoDB.
- **Docker:** Plataforma de contenedorización que permite empaquetar aplicaciones con todas sus dependencias en contenedores estandarizados. Proporciona un aislamiento ligero y eficiente.
- **Docker Compose:** Herramienta de docker para orquestar varios contenedores a la vez. Permite configurar todos los servicios, redes y volúmenes de la aplicación en un único archivo YML (docker-compose.yml).
- **Variables de entorno:** Variables de configuración del programa, guardadas de forma privada en un .env.

La combinación de estas tecnologías crea un ecosistema robusto, escalable y fácil de mantener.

2. Fundamentos teóricos

2.1. ¿Qué es Docker?

Es una plataforma que ejecuta aplicaciones en contenedores, asegurando que su funcionamiento sea el correcto en cualquier sistema.

Docker corre principalmente sobre Linux, porque utiliza cacterísticas del kernel de Linux para los contenedores.

- En **Linux**, se ejecuta de forma nativa.
- En **Windows** o **Mac**, usa una máquina virtual ligera con Linux para poder correr contenedores.

2.2. Contenedores vs Máquinas Virtuales

Características	Contenedor	Máquina Virtual (VM)
Sistema operativo	Comparte el SO del host	Cada VM tiene su propio SO completo
Peso	Ligero, rápido de iniciar	Pesado, tarda más en iniciar
Recursos	Usa solo lo necesario	Consume más recursos, reserva CPU/RAM
Aislamiento	Aislado a nivel de procesos	Aislado a nivel de hardware virtual
Portabilidad	Muy portable	Menos portable

Tabla 1: Comparación entre contenedores y máquinas virtuales

2.3. Conceptos clave

Para una mayor comprensión de la utilización de Docker, hay que analizar también unos conceptos principales; estos son de vital importancia:

- **Imágenes**: Plantilla que contiene todo lo necesario para ejecutar una aplicación perfectamente (Mongo, Ubuntu, Odoo, ...)
- **Contenedores**: Instancia de una imagen, la cual se encuentra aislada.
- **Volúmenes**: Almacenamiento persistente que conserva datos fuera del contenedor. Básicamente, en el caso de eliminar el contenedor por lo que sea, los datos guardados en el volumen siguen existiendo en el host. Se utiliza para datos importantes.
- **Redes**: Medio que permite la comunicación entre contederores y el exterior. Por ejemplo a la hora de realizar manualmente ejecutarse a un contenedor, la parte del comando `-p 3000:3000`, el primer puerto es del host y el segundo del contenedor, eso hace que ambos se comuniquen.

2.4. Docker Compose

Docker Compose es una herramienta que permite orquestar varios contenedores mediante un archivo de configuración (**docker-compose.yml**). Facilita levantar, detener y administrar varios contenedores juntos, incluyendo servicios, redes y volúmenes.

3. Arquitectura

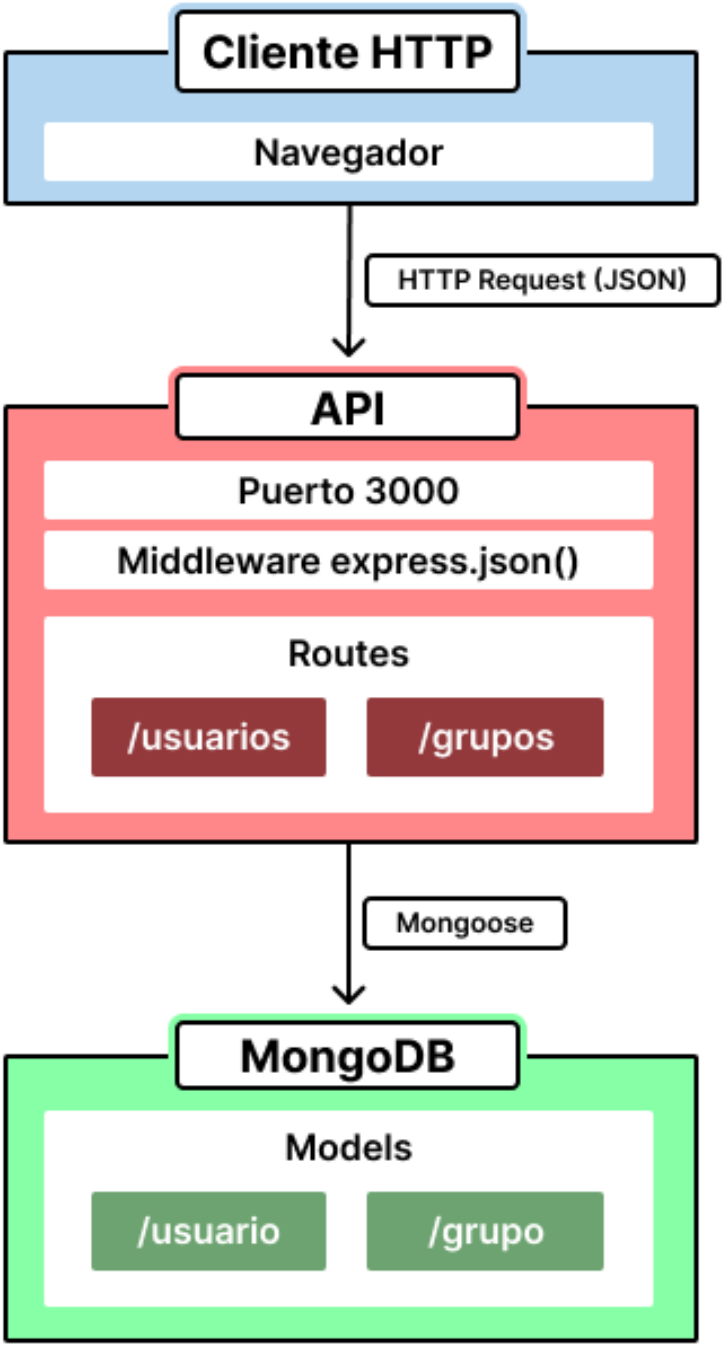

Figura 1: Dockerfile

4. Construcción de la imagen

Es un archivo de texto que contiene instrucciones para construir una imagen de Docker. Define la base del sistema, dependencias, configuraciones y comandos necesarios para que la aplicación se ejecute de forma consistente en cualquier contenedor.

4.1. Dockerfile

```
FROM node:20
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Figura 2: Dockerfile

4.2. Análisis del dockerfile

1. **FROM node:20**. Indica la imagen base para construir un contenedor. En este Dockerfile se utiliza la version 20, la cual incluye ya Node y npm instalados.
2. **WORKDIR /app**. Establece el directorio de trabajo dentro del contenedor en /app. Todos los comandos que vienen a continuación se ejecutarán dentro de esta carpeta.
3. **COPY package*.json ./**. Copia los archivos package.json y package-lock.json desde el directorio del host al directorio del contenedor. Así se instalarán las dependencias necesarias en el contenedor.
4. **RUN npm install**. Ejecuta npm install dentro del contenedor para instalar las dependencias del proyecto, las cuales se pueden instalar gracias al paso anterior.
5. **COPY . .**. Copia todo el código del proyecto al directorio del contenedor
6. **EXPOSE 3000**. Sugiere que el contenedor escuche el puerto 3000
7. **CMD ["npm","start"]**. Comando que se va a ejecutar cuando se inicia el contenedor, iniciando así la aplicación.

5. Docker Compose: Orquestación de servicios

Es una herramienta que permite definir y ejecutar aplicaciones que usan múltiples contenedores mediante un archivo de configuración (docker-compose.yml). Describe servicios, redes y volúmenes, facilitando desplegar y administrar toda la aplicación de manera consistente.

5.1. Estructura del archivo docker-compose.yml

```
Run All Services
services:
  Run Service
  api:
    build: .
    container_name: mi_api
    ports:
      - "3000:3000"
    environment:
      MONGO_URI: ${URI}
    depends_on:
      - mongo

  Run Service
  mongo:
    image: mongo:6
    container_name: mongodb
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${MONGO_INITDB_ROOT_USERNAME}
      MONGO_INITDB_ROOT_PASSWORD: ${MONGO_INITDB_ROOT_PASSWORD}
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Figura 3: docker-compose.yml

5.2. Servicios

- **build**. Define la ruta o Dockerfile para construir una imagen personalizada del servicio antes de ejecutarlo.

  En este proyecto se utiliza para: `docker build -t mi_api:v1.0.0 .`

- **image**. Especifica la imagen de Docker que se va a usar para el servicio; puede ser desde Docker Hub o de forma local.

  En este proyecto se hará de forma local.

- **container_name**. Le da un nombre al contenedor, sino lo hace automáticamente.

- **ports**. Mapea puertos del contenedor a puertos del host, permitiendo que los servicios sean accesibles desde el exterior.
- **environment**. Define variables de entorno que se pasarán al contenedor para configurarlo.
- **depends_on**. Indica que el servicio en el que se sitúa depende de otros, lo que asegura que estos otros se inicien antes.
- **volumes**. Permite montar directorios/archivos del host dentro del contenedor, así se persisten los datos y se comparte información entre ambos.

5.3. Configuración de volúmenes

Figura 4: Volumen en el docker-compose.yml

En el docker-compose.yml se declara el volumen **mongo_data**, y este garantiza que los datos de Mongo persistan y no se pierdan al eliminar o reiniciar el contenedor.

5.4. Variables de entorno

Variables de configuración del programa, guardadas de forma privada en un .env.

Posteriormente el .env en el que se encuentran estas variables será guardado en .gitignore y en .dockerignore.

> **.dockerignore** y **.gitignore** son archivos en los que se introduce todo el contenido que no debe ser subido por seguridad u optimización.
>
> Ejemplo: .env por seguridad y node_modules porque pesan mucho y no es necesaria su carga.

5.5. Dependencias entre servicios

En el **docker-compose.yml**, el servicio **api** depende del servicio **mongo**, lo que significa que Docker Compose inicia primero el contenedor de Mongo antes de levantar la API. Así se asegura que la base de datos esté en ejecución cuando la API intente conectarse y no se quede colgada.

El **depends_on** garantiza el orden de inicio de los contenedores.

6. Proceso de dockerización

6.1. Preparación del entorno

6.2. Construcción de la imagen

6.3. Creación y ejecución de contenedores

6.4. Verificación del funcionamiento

7. Comandos Docker útiles

7.1. Comandos básicos

7.2. Gestión de imágenes

7.3. Gestión de contenedores

7.4. Docker Compose CLI

7.5. Comandos de depuración

8. Redes y comunicación

8.1. Red por defecto de Docker Compose

8.2. Resolución de nombres DNS

8.3. Comunicación entre contenedores

9. Persistencia de datos

9.1. Volúmenes en Docker

9.2. Volumen mongo_data

9.3. Backup y recuperación

10. Pruebas y validación

10.1. Verificación de contenedores activos

10.2. Pruebas de conectividad

10.3. Pruebas de endpoints

10.4. Logs y monitorización

11. Optimizaciones y mejoras

11.1. Optimización del Dockerfile

11.2. Multi-stage builds

11.3. Reducción del tamaño de la imagen

11.4. Cache de capas

12. Seguridad

12.1. Gestión de secretos

12.2. Usuarios no privilegiados

12.3. Escaneo de vulnerabilidades

12.4. Actualización de imágenes base

13. Despliegue en producción

13.1. Diferencias con desarrollo

13.2. Configuración para producción

13.3. Escalabilidad

13.4. Herramientas de orquestación

14. Troubleshooting

14.1. Problemas comunes

14.2. Errores de conexión

14.3. Problemas de permisos

14.4. Debugging de contenedores

15. Conclusiones

15.1. Ventajas de la dockerización

15.2. Resultados obtenidos

15.3. Trabajo futuro

16. Referencias

16.1. Documentación oficial

16.2. Recursos adicionales