Programación Multimedia y Dispositivos Móviles

Documentación API dockerizada

Servidor Backend con Express y MongoDB

Autor:

Santi Martínez

7 de diciembre de 2025

Índice

1 Introducción **3**
- 1.1 Contexto del proyecto ... 3
- 1.2 Objetivos de la dockerización ... 3
- 1.3 Tecnologías utilizadas ... 3

2 Fundamentos teóricos utilizados **4**
- 2.1 ¿Qué es Docker? ... 4
  - 2.1.1 Contenedores vs Máquinas Virtuales ... 4
  - 2.1.2 Docker Compose ... 4
  - 2.1.3 Dockerfile ... 4
  - 2.1.4 .dockerignore ... 4
  - 2.1.5 Conceptos clave ... 5
- 2.2 ¿Qué es MongoDB? ... 6
  - 2.2.1 SQL vs NoSQL ... 6
- 2.3 ¿Qué es un .sh? ... 7

3 Arquitectura **8**
- 3.1 Descripción de la API ... 8
- 3.2 Componentes del sistema ... 8

4 Construcción de la imagen **10**
- 4.1 Dockerfile ... 10

5 Docker Compose: Orquestación de servicios **11**
- 5.1 Estructura del archivo docker-compose.yml ... 11
- 5.2 Servicios ... 11
- 5.3 Variables de entorno ... 12
- 5.4 Dependencias entre servicios ... 12

6 Proceso de dockerización **13**
- 6.1 Preparación del entorno ... 13
- 6.2 Construcción de la imagen ... 13
- 6.3 Creación y ejecución de contenedores ... 13
- 6.4 Verificación del funcionamiento ... 13
- 6.5 Recomendación para verificar información ... 14

7 Chatbot en Telegram **15**
- 7.1 Crear y configurar el bot ... 16
- 7.2 Ponerle nombre ... 17
- 7.3 Obtener el ID del bot ... 18
- 7.4 Chatbot funcionando ... 19

8 Actualización de imagen y tag **20**
- 8.1 Imagen Docker Hub ... 20
- 8.2 Tag en GitHub ... 21
- 8.3 Beneficios de la actualización de versiones ... 21

9 Comandos Docker útiles **22**
- 9.1 Comandos básicos ... 22
- 9.2 Gestión de imágenes ... 22

9.3 Gestión de contenedores 22
9.4 Docker Compose CLI 23
9.5 Comandos de depuración 23

10 Pruebas y validación **24**
10.1 Verificación de contenedores activos 24
10.2 Pruebas de conectividad 24
10.3 Pruebas de endpoints 24

11 Despliegue **25**
11.1 Comandos de despliegue 25
11.2 Monitoreo y mantenimiento 25
11.3 Script para el despliegue 25

12 Optimizaciones y mejoras **26**
12.1 Unificación de comandos en .sh 26
12.2 Optimización del Dockerfile 26

13 Seguridad **27**
13.1 Gestión de secretos 27
13.2 Usuarios no privilegiados 27
13.3 Escaneo de vulnerabilidades 27
13.4 Actualización de imágenes base 27

14 Troubleshooting **28**
14.1 Problemas comunes 28
14.2 Errores de conexión 28
14.3 Problemas de permisos 29
14.4 Debugging de contenedores 29

15 Problemas encontrados **30**
15.1 Problema 1: Conexión a MongoDB fallida 30
15.2 Problema 2: Permisos insuficientes en volúmenes 30
15.3 Problema 3: Errores con la actualización de la imagen 30
15.4 Problema 4: Conflictos de puertos 31
15.5 Problema 5: Actualización del tag en GitHub 31

16 Conclusiones **32**
16.1 Ventajas de la dockerización 32
16.2 Resultados obtenidos 32
16.3 Ventajas del .sh 32
16.4 Trabajo adicional 32

17 Referencias **33**

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 utilizados

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 características 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.

2.1.1. 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

2.1.2. 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.

2.1.3. Dockerfile

Un Dockerfile 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.

En este proyecto, se utiliza un Dockerfile para definir cómo construir la imagen de la API.

2.1.4. .dockerignore

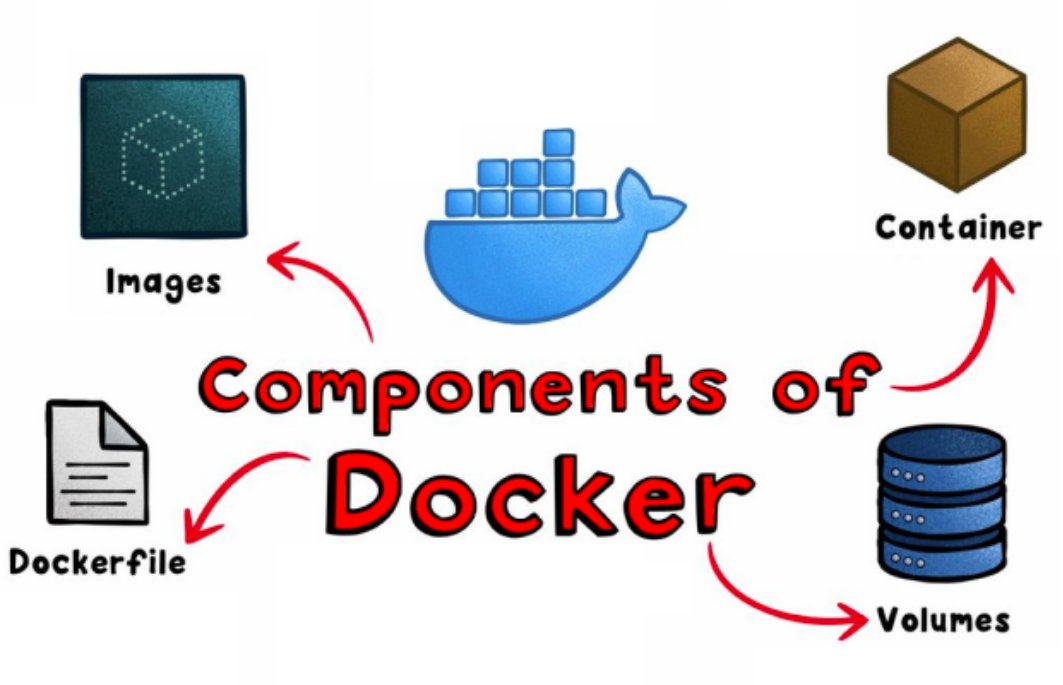
Archivo muy similar al .gitignore, en el que se introduce todo el contenido que no debe ser subido por seguridad u optimización.

La diferencia con .gitignore es que este último es para Git, y el .dockerignore es para Docker; por lo que influye en la construcción de la imagen Docker.

2.1.5. 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 contenedores 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.



2.2. ¿Qué es MongoDB?

MongoDB es una base de datos NoSQL orientada a documentos que almacena datos en formato BSON (Binary JSON).

Para una buena interacción de datos API-MongoDB, se utiliza Mongoose, un ODM (Object Document Mapper) que permite que Node.js se comunique y gestione datos en MongoDB de manera sencilla y eficiente.

Algunas características clave de MongoDB incluyen:

- **Alto rendimiento**: Optimizada para operaciones rápidas de lectura y escritura.
- **Seguridad**: Soporta autenticación, autorización y cifrado de datos.
- **Modelo de datos flexible**: Permite almacenar documentos con estructuras variadas.
- **Escalabilidad horizontal**: Facilita la distribución de datos a través de múltiples servidores.
- **Consultas avanzadas**: Soporta consultas complejas y agregaciones.
- **Alta disponibilidad**: Ofrece replicación y recuperación ante fallos.

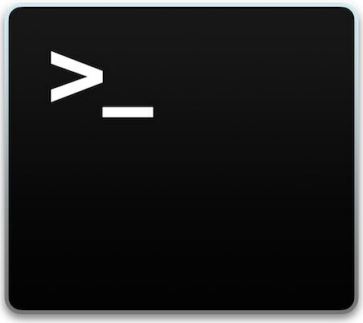
2.2.1. SQL vs NoSQL

Características	SQL (Relacional)	NoSQL (No Relacional)
Estructura de datos	Tablas con filas y columnas	Documentos, clave-valor, grafos, etc.
Esquema	Esquema fijo y predefinido	Esquema flexible y dinámico
Escalabilidad	Escalabilidad vertical	Escalabilidad horizontal
Lenguaje de consulta	SQL (Structured Query Language)	Varios lenguajes específicos
Transacciones	Soporta transacciones ACID	Soporte limitado para transacciones ACID
Ejemplos populares	MySQL, PostgreSQL, Oracle DB	MongoDB, Cassandra, Redis

2.3. ¿Qué es un .sh?

Este apartado está dedicado al encargado de poner en marcha el despliegue. **.sh**.

Se ha utilizado para la simplificación del proceso de dockerización, ya que contiene todos los comandos necesarios para construir y ejecutar los contenedores de forma automática. Eso libra al usuario de tener que escribir varios comandos manualmente, evitando errores y facilitando el proceso.



Un archivo .sh es un script de shell utilizado en sistemas operativos basados en Unix/Linux para automatizar tareas mediante comandos del shell. Estos archivos contienen una serie de instrucciones que se ejecutan secuencialmente cuando se ejecuta el script; de manera similar que el archivo Dockerfile, explicado anteriormente.

Para la ejecución de un archivo .sh, es necesario otorgarle permisos de ejecución utilizando el comando

```
chmod +x nombre_archivo.sh
```

y luego, dentro del repositorio, se puede ejecutar con

```
./nombre_archivo.sh
```

Esto, en un solo comando, ejecuta todas las instrucciones contenidas en el archivo .sh de forma automática. Entre las que se encuentran:

- Construcción de la imagen Docker de la API.
- Levantamiento de los contenedores utilizando Docker Compose.
- Verificación del estado de los contenedores.

Particularmente, se ha utilizado para facilitar el proceso de dockerización del proyecto, haciendo que sea accesible incluso para aquellos con poca experiencia en Docker; evitando así errores comunes y asegurando que todos los pasos necesarios se realicen correctamente.

3. Arquitectura

La arquitectura del proyecto se compone de dos servicios principales: la API REST y la base de datos MongoDB. Estos servicios se ejecutan en contenedores separados, pero se comunican entre sí para proporcionar la funcionalidad completa de la aplicación.

El encargado de orquestar estos contenedores es Docker Compose.

Para ello, se utiliza un archivo **docker-compose.yml** que define los servicios, redes y volúmenes necesarios para la aplicación.

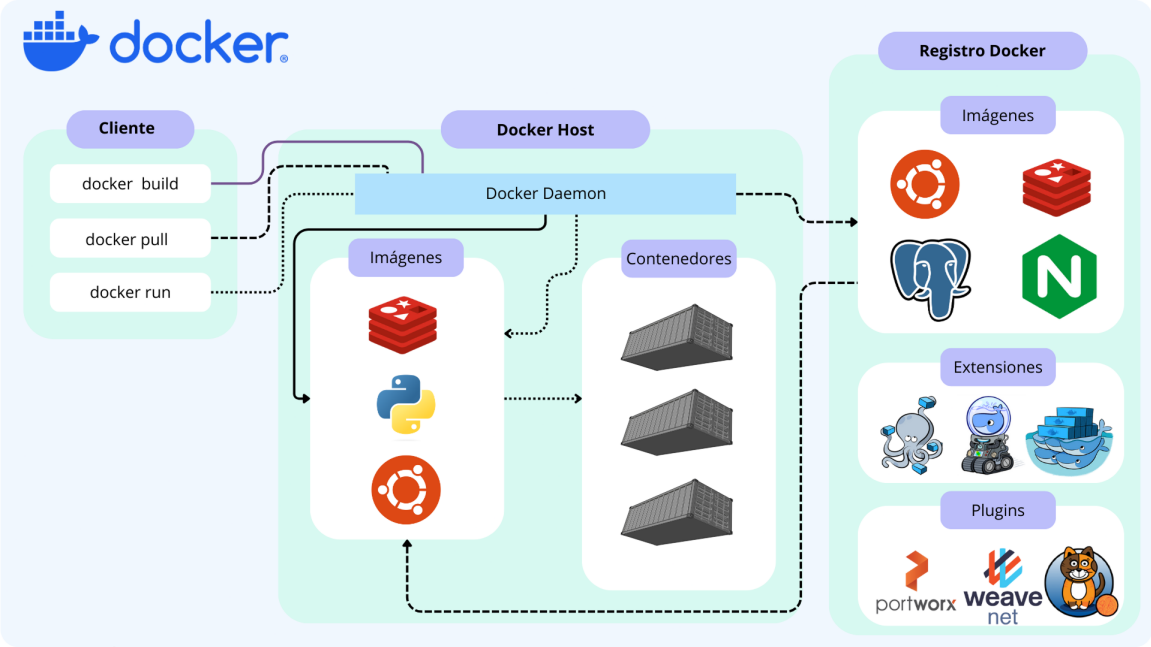


Figura 1: Arquitectura del un sistema dockerizado

3.1. Descripción de la API

Es 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:

- **Servidor de aplicación:** Expone los endpoints REST y maneja la lógica de negocio.
- **Base de datos MongoDB:** Persiste la información de usuarios y grupos.

3.2. Componentes del sistema

- **API REST:** Servidor backend desarrollado con Node.js y Express, que maneja las solicitudes HTTP y la lógica de negocio.
- **MongoDB:** Base de datos NoSQL que almacena los datos de usuarios y grupos.

- **Docker:** Plataforma de contenedorización que encapsula la API y MongoDB en contenedores independientes.
- **Docker Compose:** Herramienta para orquestar los contenedores de la API y MongoDB, definiendo servicios, redes y volúmenes en un archivo YML.

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.

La forma de ejecutar los comandos de su interior es de forma **secuencial**

4.1. Dockerfile

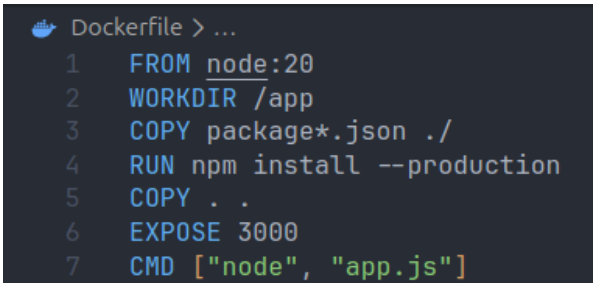


Figura 2: Dockerfile

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.

Uso de **node:20** en lugar de por ejemplo **node:20-alpine**, para evitar problemas de compatibilidad con algunas dependencias nativas de Node.js que pueden no estar disponibles en la versión Alpine, que pese a ser más ligera, puede causar problemas.

WORKDIR /app . Establece el directorio de trabajo dentro del contenedor en /app. Todas las instrucciones posteriores se ejecutarán desde este directorio.

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.

El * en package*.json es un comodín que permite copiar ambos archivos en una sola instrucción.

RUN npm install –production . Ejecuta npm install dentro del contenedor para instalar las dependencias del proyecto, las cuales se pueden instalar gracias al paso anterior.

La opción –**production** asegura que solo se instalen las dependencias necesarias para ejecutar la aplicación en producción, excluyendo las dependencias de desarrollo.

COPY . . . Copia todo el código del proyecto al directorio del contenedor

EXPOSE 3000 . Sugiere que el contenedor escuche el puerto 3000

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

La arquitectura definida en el docker-compose.yml consta de dos servicios principales: la API y MongoDB, cada uno en su propio contenedor

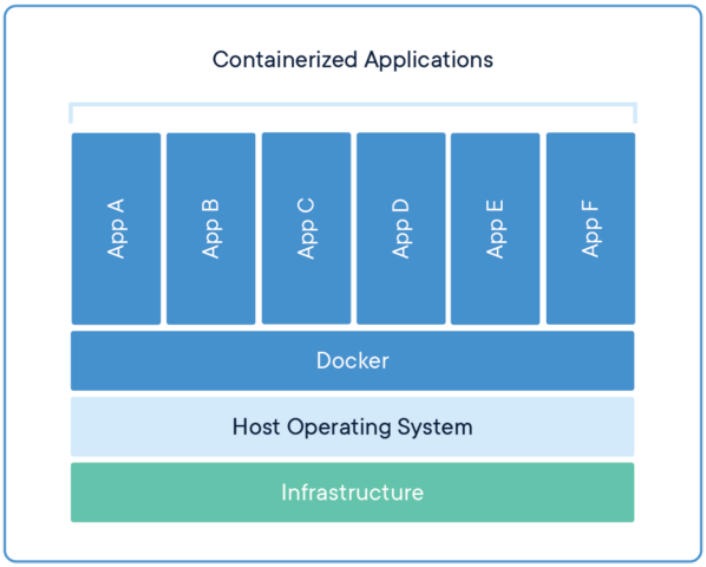


Figura 3: docker-compose.yml

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. Variables de entorno

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

En el repositorio se puede encontrar también una plantilla del mismo llamada **.env.example**, la cual contiene las variables necesarias para el correcto funcionamiento de la API. Solo es necesario copiarla y renombrarla a .env, y rellenar los valores de las variables según las necesidades del usuario.

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.
>
> .dockerignore influye en la construcción de la imagen Docker, mientras que .gitignore es para Git.

En este proyecto, las variables de entorno definidas en el archivo .env son:

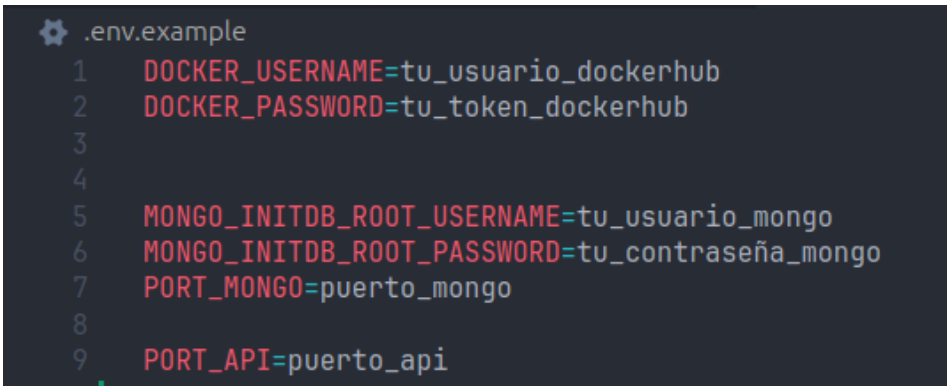


Figura 4: Ejemplo de variables de entorno en .env

5.4. 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

Esta sección detalla los pasos seguidos para dockerizar este proyecto, desde la preparación del entorno hasta la verificación del funcionamiento.

Pasos principales:

1. Preparación del entorno
2. Construcción de la imagen
3. Creación y ejecución de contenedores
4. Verificación del funcionamiento

6.1. Preparación del entorno

- **Instalación de Docker y Docker Compose:** Asegurarse de tener Docker y Docker Compose instalados en el sistema.
- **Estructura del proyecto:** Organizar el código, Dockerfile, docker-compose.yml y otros archivos necesarios.
- **Configuración de variables de entorno:** Crear un archivo .env para almacenar configuraciones sensibles como credenciales de la base de datos.

6.2. Construcción de la imagen

- **Escribir el Dockerfile:** Crear un Dockerfile que defina cómo construir la imagen de la API.
- **Construir la imagen:** Utilizar el comando `docker build -t mi_api:v1.0.0 .` para construir la imagen de la API.
- **Verificar la imagen:** Usar `docker images` para asegurarse de que la imagen se ha creado correctamente.

6.3. Creación y ejecución de contenedores

- **Definir docker-compose.yml:** Crear un archivo docker-compose.yml que describa los servicios de la API y MongoDB, incluyendo redes y volúmenes.
- **Levantar los contenedores:** Ejecutar `docker compose up -d` para iniciar los contenedores en segundo plano.
- **Verificar contenedores activos:** Usar `docker ps` para comprobar que ambos contenedores están en ejecución.

6.4. Verificación del funcionamiento

- **Probar endpoints de la API:** Utilizar herramientas como Postman o curl para enviar solicitudes a los endpoints de la API y verificar las respuestas.
- **Comprobar la conexión a MongoDB:** Asegurarse de que la API puede conectarse a la base de datos y realizar operaciones CRUD.
- **Revisar logs:** Utilizar `docker logs <container_id>` para revisar los logs de ambos contenedores y solucionar posibles problemas.

6.5. Recomendación para verificar información

Para comprobar que la imagen se ha construido correctamente, y que los contenedores están funcionando bien, se puede utilizar una herramienta de open source que desglosa toda la información principal de Docker de manera visual y sencilla.

La herramienta se llama **dockerinfo**, y se puede encontrar en dockerinfo - santimartinezzgb.

```
┌─[santi][santi-ThinkPad][~]
└─▪  dockerinfo

──────────────────────────────────────────────────
📦 IMÁGENES (6)
──────────────────────────────────────────────────

IMAGE                          ID             DISK USAGE   CONTENT SIZE   EXTRA
dockerizacion-api-api:latest   02ce8c5dcfa4       1.63GB          408MB
local/api-docker:latest        b2f7160710bc       1.63GB          408MB
local/api-docker:v1.0.0        b2f7160710bc       1.63GB          408MB
mongo:6                        0a83b2f824b2       1.05GB          270MB
odoo:17                        070bf7985f2c       2.72GB          648MB
postgres:15                    24d6c206bba8        633MB          164MB

──────────────────────────────────────────────────
▶️ CONTENEDORES ACTIVOS (0)
──────────────────────────────────────────────────

  Sin contenedores activos

──────────────────────────────────────────────────
📋 TODOS LOS CONTENEDORES (0)
──────────────────────────────────────────────────

  Sin contenedores

──────────────────────────────────────────────────
💾 VOLÚMENES (2)
──────────────────────────────────────────────────

DRIVER    VOLUME NAME
local     odoo_practica4-santi-martinez_db-data
local     odoo_practica4-santi-martinez_odoo-data

──────────────────────────────────────────────────
🌐 REDES (3)
──────────────────────────────────────────────────

NETWORK ID     NAME      DRIVER    SCOPE
7c147883e0f2   bridge    bridge    local
328eb77985a5   host      host      local
85cc445f6704   none      null      local
```

7. Chatbot en Telegram

Para facilitar la interacción con la API dockerizada, se ha implementado un chatbot en Telegram que permite a los usuarios realizar operaciones básicas a través de mensajes. Este chatbot se comunica con la API para enviar y recibir datos.

Esta clase de bots son muy útiles para notificaciones rápidas y para interactuar con la API sin necesidad de una interfaz gráfica compleja.

En este proyecto se ha utilizado el bot **RawDataBot** para crear y gestionar el chatbot en Telegram. El motivo de esta elección es su facilidad de uso y la capacidad de integrarse con APIs externas mediante tokens de acceso, lo cual se consigue de forma sencilla a través de GitHub Actions.

7.1. Crear y configurar el bot

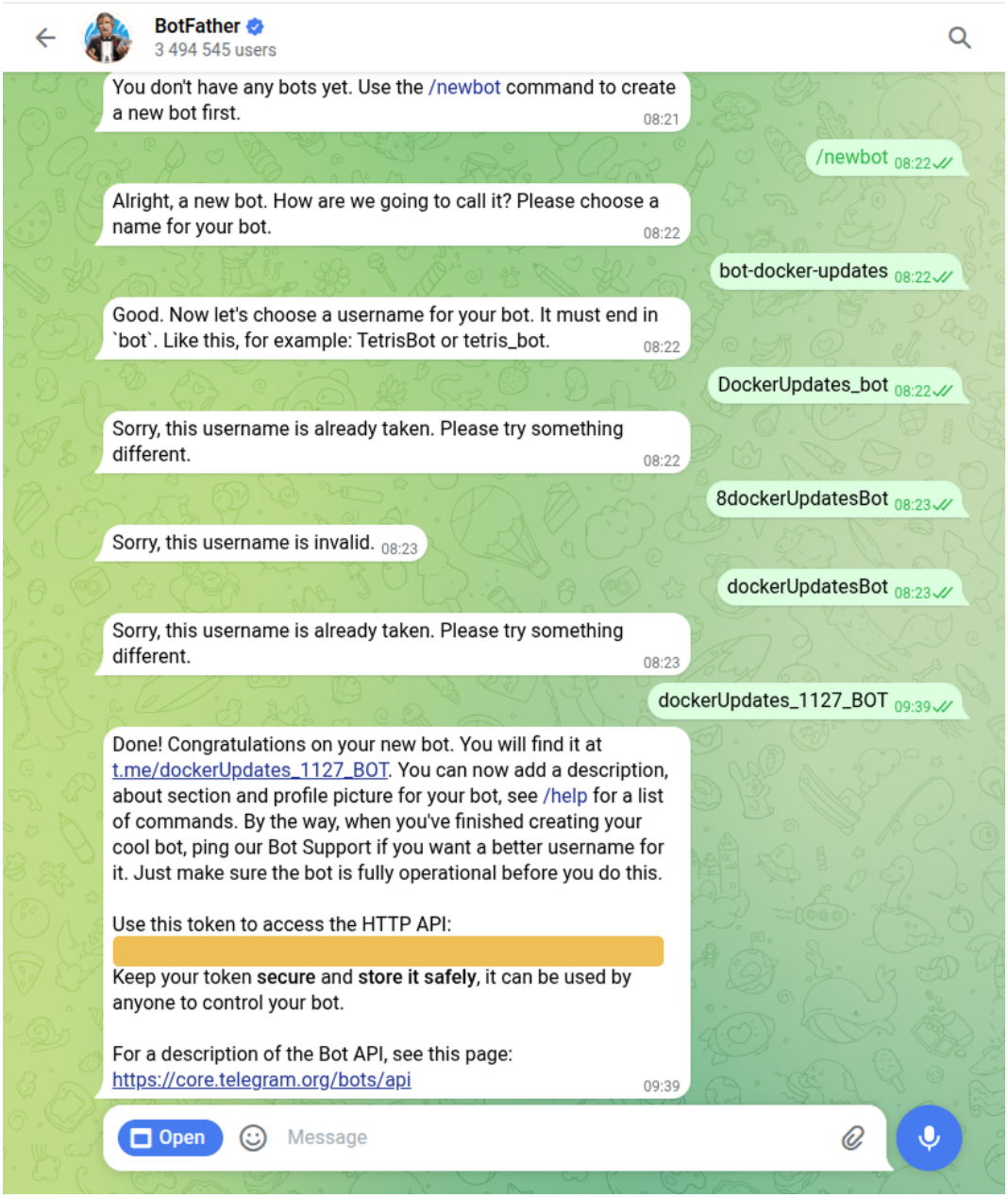
En el buscador de Telegram, se busca "BotFather", el cual es el bot oficial para crear y gestionar otros bots en Telegram.

Al iniciar una conversación con BotFather, se pueden ver varias opciones disponibles para gestionar los bots, como crear uno nuevo, listar los existentes, cambiar configuraciones, entre otras.

7.2. Ponerle nombre

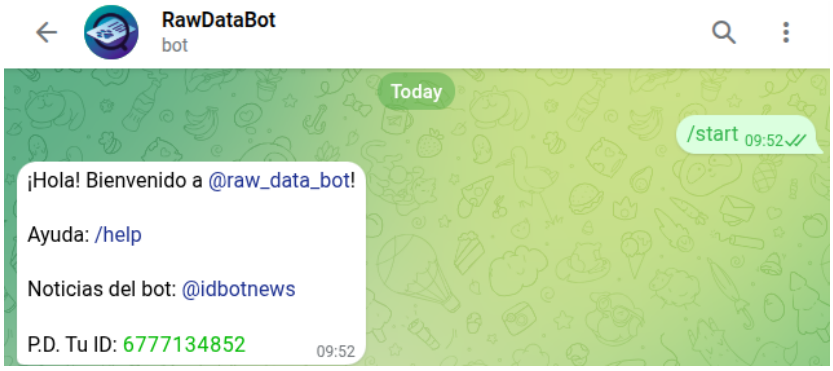
La creación del bot se realiza mediante el comando /newbot, el cual solicita un nombre de usuario para el mismo, el cual, por lo que se ve en la siguiente imagen, tiene que ser atómico y único, sino BotFather no lo aceptará.

Al enviar un nombre válido, el BotFather responde con un mensaje de confirmación y proporciona el token de acceso necesario para la integración con la API (por razones de seguridad, este token se muestra tapado en la imagen).



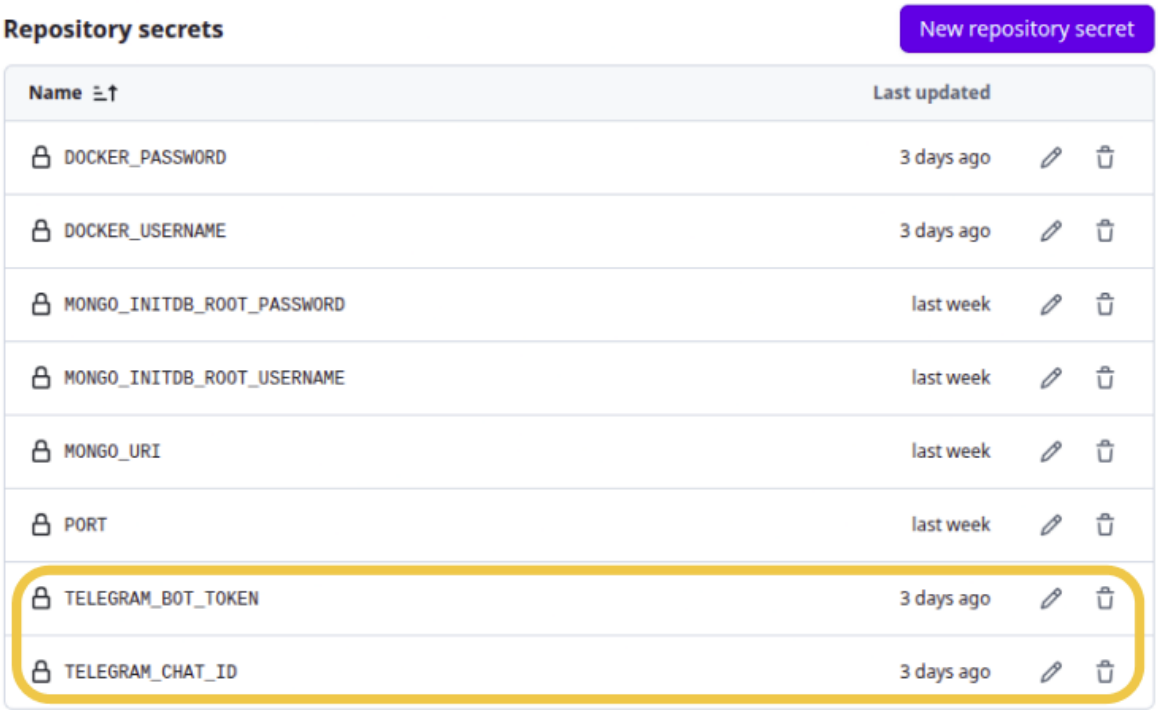
7.3. Obtener el ID del bot

Para obtener el ID del bot, basta con enviar el mensaje /start.ª RawDataBot en Telegram, el cual responderá con un mensaje que incluye el ID del bot.



Con este ID, y el token obtenido anteriormente, se puede configurar la API para que el bot pueda interactuar con ella.

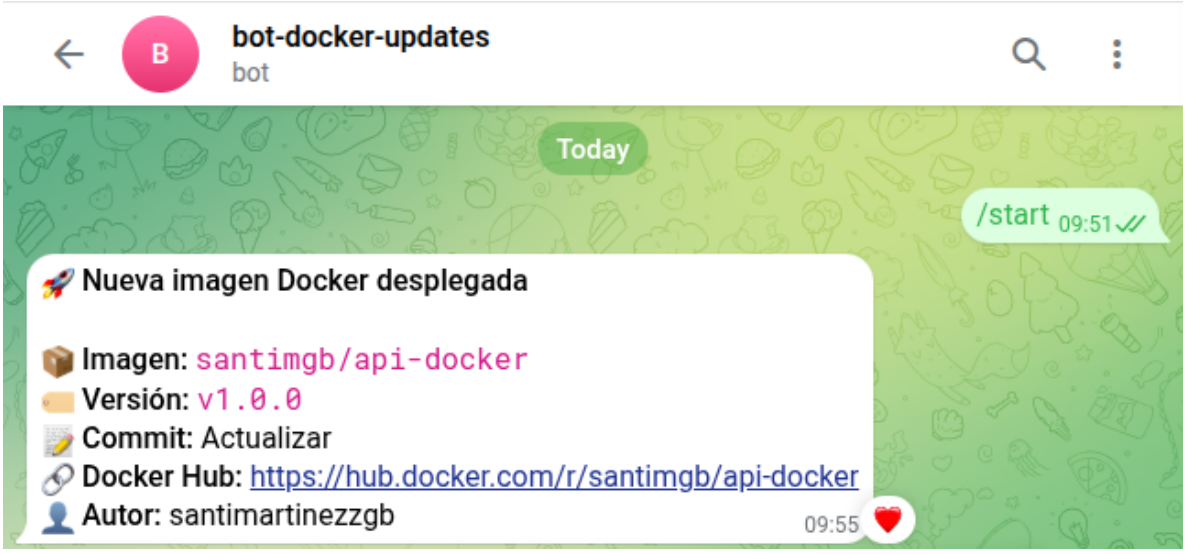
Para ello se entrará en secrets ->actions en GitHub, y se añadirán dos nuevos secretos con los nombres **TELEGRAM_BOT_TOKEN** y **TELEGRAM_CHAT_ID**, con los valores del token y el ID respectivamente.



7.4. Chatbot funcionando

Una vez configurado el bot y obtenido el ID, se puede interactuar con él enviando comandos específicos que la API reconoce y procesa.

El mensaje de confirmación del bot, se enviará al pushear los cambios a la rama main del repositorio, lo que desencadenará el workflow de GitHub Actions que construye y despliega la API dockerizada.

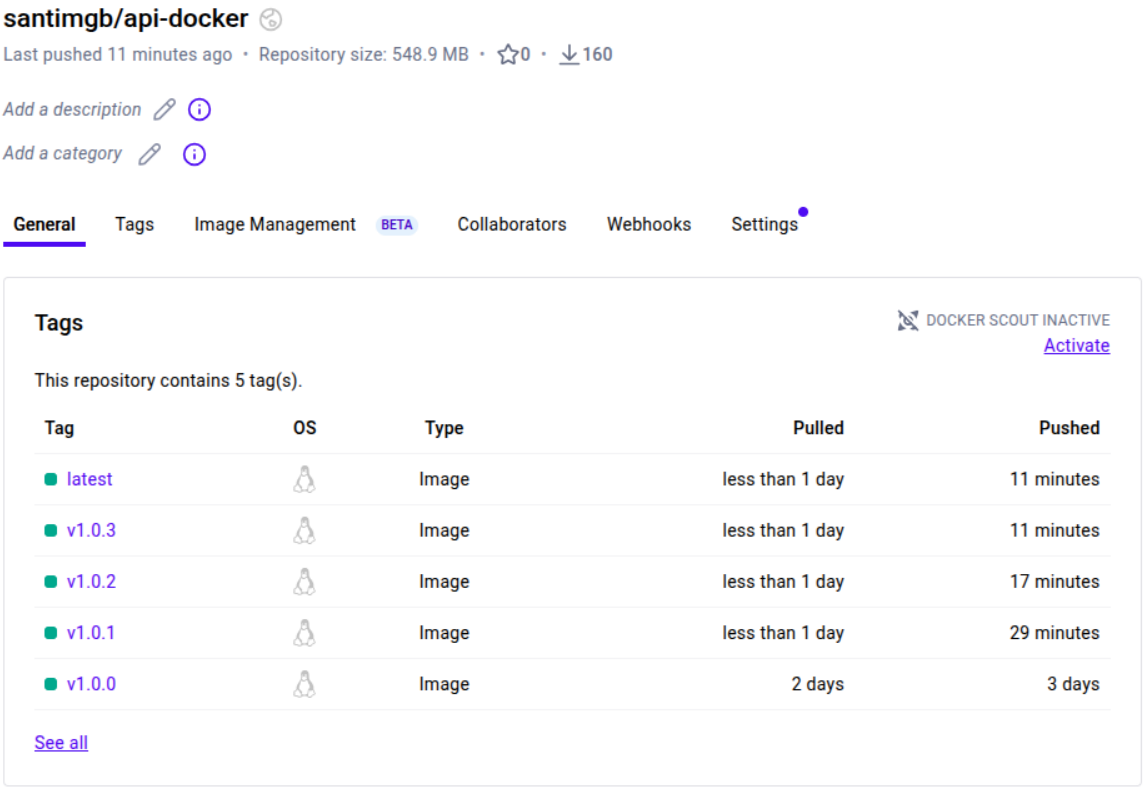


8. Actualización de imagen y tag

Tanto la imagen de Docker como su tag en GitHub, se actualizan en el momento que se pushean cambios a la rama main del repositorio. Esto se logra mediante un workflow de GitHub Actions que se activa con cada push a main.

8.1. Imagen Docker Hub

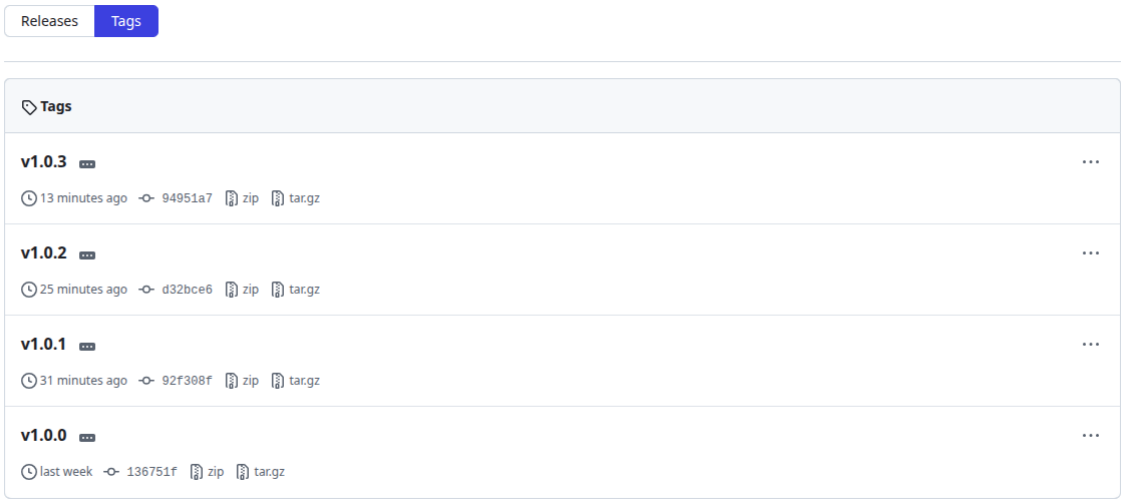
A medida que se realizan cambios en la API, es importante mantener la imagen de Docker actualizada para reflejar estos cambios. Esto implica reconstruir la imagen y asignarle un nuevo tag que indique la versión actualizada.



Como se puede observar en la imagen anterior, cada vez que se realiza un push a main, se crea una nueva versión de la imagen en Docker Hub con un tag que refleja la versión actualizada (por ejemplo, v1.0.1, v1.0.2, etc.).

8.2. Tag en GitHub

Similarmente, el tag en GitHub también se actualiza con cada push a main. Esto permite llevar un control de versiones claro y organizado del código fuente de la API.



En la imagen anterior, se puede ver cómo los tags en GitHub reflejan las versiones de la API, facilitando la identificación de cambios y mejoras en cada versión.

8.3. Beneficios de la actualización de versiones

Mantener la imagen de Docker y los tags en GitHub actualizados ofrece varios beneficios clave:

- **Control de versiones:** Permite rastrear cambios y revertir a versiones anteriores si es necesario.
- **Despliegue consistente:** Asegura que las implementaciones utilicen la versión correcta de la API.
- **Facilita la colaboración:** Los desarrolladores pueden trabajar con versiones específicas del código y la imagen Docker.

9. Comandos Docker útiles

Para poder trabajar correctamente con Docker y Docker Compose, es necesario conocer una serie de comandos básicos que son de gran ayuda en la gestión de imágenes y contenedores, así como en la depuración de problemas de los mismos.

Pese a la explicación de cada uno de los principales comandos para poder seguir la línea de una correcta dockerización, en este esa labor la realiza el archivo **setup.sh** que se encuentra en la raíz del proyecto, el cual contiene todos los comandos necesarios para construir y ejecutar los contenedores de forma automática.

9.1. Comandos básicos

Son los comandos esenciales para construir, ejecutar y gestionar contenedores e imágenes de Docker.

- **docker build -t <nombre_imagen>:<tag>.**: Construye una imagen de Docker a partir del Dockerfile en el directorio actual.
- **docker images**: Lista todas las imágenes de Docker disponibles localmente.
- **docker run -d -p <host>:<contenedor>–name <contenedor><imagen>:<version>** Crea y ejecuta un contenedor en segundo plano, mapeando puertos y asignando un nombre.

  En este caso: `docker run -d -p 3000:3000 -name api api-docker:v1.0.0`
- **docker ps**: Muestra los contenedores en ejecución.
- **docker stop <nombre_ contenedor>**: Detiene un contenedor en ejecución.
- **docker rm <nombre_contenedor>**: Elimina un contenedor detenido.
- **docker rmi <nombre_imagen>:<tag>**: Elimina una imagen de Docker. Cabe mencionar que no se puede eliminar una imagen si hay contenedores con esa imagen, tanto en ejecución como detenidos.

9.2. Gestión de imágenes

Manejo de imágenes de Docker, incluyendo el pull, etiquetado y subida a repositorios.

- **docker pull <nombre_imagen>:<tag>**: Descarga una imagen de Docker desde un repositorio (como Docker Hub).
- **docker tag <imagen_id><nuevo_nombre_imagen>:<nuevo_tag>**: Etiqueta una imagen existente con un nuevo nombre y tag.
- **docker push <nombre_imagen>:<tag>**: Sube una imagen de Docker a un repositorio.

9.3. Gestión de contenedores

Gestión avanzada de contenedores en ejecución.

- **docker exec -it <nombre_contenedor>/bin/bash**: Accede a la terminal de un contenedor en ejecución.
- **docker logs <nombre_contenedor>**: Muestra los logs de un contenedor.
- **docker inspect <nombre_contenedor>**: Proporciona detalles técnicos sobre un contenedor.

9.4. Docker Compose CLI

Comandos específicos para gestionar aplicaciones definidas con Docker Compose.

- **docker compose up -d**: Inicia los servicios definidos en docker-compose.yml en segundo plano.
- **docker compose down**: Detiene y elimina los contenedores, redes y volúmenes creados por docker-compose up.
- **docker compose ps**: Muestra el estado de los servicios definidos en docker-compose.yml.
- **docker compose logs <servicio>**: Muestra los logs de un servicio específico definido en docker-compose.yml.

9.5. Comandos de depuración

Son útiles para monitorear y diagnosticar problemas en contenedores en ejecución.

- **docker stats**: Muestra estadísticas en tiempo real del uso de recursos de los contenedores en ejecución.
- **docker network ls**: Lista todas las redes de Docker.
- **docker volume ls**: Lista todos los volúmenes de Docker.

10. Pruebas y validación

Después de dockerizar la API y MongoDB, es fundamental realizar pruebas para asegurar que ambos servicios funcionan correctamente y se comunican entre sí como se espera.

10.1. Verificación de contenedores activos

Utilizar el comando `docker ps` para comprobar que ambos contenedores (API y MongoDB) están en ejecución y funcionando correctamente.

El comando utilizado es:

```
docker ps
```

10.2. Pruebas de conectividad

Asegurarse de que la API puede conectarse a MongoDB utilizando la cadena de conexión definida en las variables de entorno. Esto se puede verificar revisando los logs del contenedor de la API para confirmar que la conexión se ha establecido correctamente.

Las pruebas de conectividad se han realizado revisando los logs del contenedor de la API con el comando:

```
docker logs <nombre_contenedor_api>
```

10.3. Pruebas de endpoints

Utilizar herramientas como Postman o curl para enviar solicitudes a los endpoints de la API y verificar que las respuestas son correctas. Esto incluye probar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para usuarios y grupos.

En el caso de este proyecto, se ha probado curl con los siguientes comandos:

```
curl -X GET http://localhost:mi-puerto/api/users
```

11. Despliegue

El despliegue de la aplicación dockerizada implica poner en marcha los contenedores en un entorno de desarrollo, asegurando que estén accesibles y funcionando correctamente.

11.1. Comandos de despliegue

- **docker compose up -d**: Levanta los contenedores, y el -d indica que se ejecuten en segundo plano.
- **docker compose down**: Detiene y elimina los contenedores, redes y volúmenes creados.

11.2. Monitoreo y mantenimiento

Implementar herramientas de monitoreo para supervisar el rendimiento y la salud de los contenedores, así como establecer procedimientos de mantenimiento regular para actualizar imágenes y gestionar datos persistentes.

Ejemplos de herramientas de monitoreo:

- **Prometheus**: Sistema de monitoreo y alerta.
- **cAdvisor**: Herramienta para monitorear el uso de recursos de los contenedores.
- **Grafana**: Plataforma de visualización de datos.

11.3. Script para el despliegue

Para facilitar el proceso de despliegue, se ha creado un script llamado **setup.sh** que automatiza la construcción de la imagen y el levantamiento de los contenedores.

El contenido del script es el siguiente:

>_ setup.sh

```
1  #!/bin/bash
2  set -e
3
4  # Cargar variables de entorno desde el archivo .env si existe
5  [ -f .env ] && export $(grep -v '^#' .env | xargs)
6
7  # Construir y levantar los servicios con Docker Compose
8  echo "Levantando servicios..."
9
10 # 2>/dev/null para evitar errores si no hay servicios corriendo
11 docker compose down 2>/dev/null || true
12 docker compose up -d --build
13
14 # Esperar unos segundos para que los servicios se inicien
   correctamente
15 echo "Esperando servicios..."
16 sleep 5
17
18 # Mostrar mensaje de éxito
19 echo "API disponible en http://localhost:${PORT_API:-3000}"
```

Este script permite ejecutar todo el proceso de dockerización con un solo comando, simplificando así el despliegue de la aplicación.

12. Optimizaciones y mejoras

Para mejorar el rendimiento y la eficiencia de la imagen Docker y los contenedores, se pueden implementar varias optimizaciones durante la construcción del Dockerfile y la configuración de Docker Compose.

12.1. Unificación de comandos en .sh

Crear un script .sh que contenga todos los comandos necesarios para construir y ejecutar los contenedores, facilitando así el proceso de despliegue con un solo comando.

En el caso de este proyecto, se ha creado el archivo **setup.sh** que contiene los comandos para construir la imagen y levantar los contenedores.

Es un archivo sencillo que se puede realizar con cuidado mediante ayuda de IA y que facilita mucho el proceso de dockerización, además de evitar errores al escribir varios de los comandos manualmente.

12.2. Optimización del Dockerfile

- **Uso de imágenes base ligeras:** Elegir imágenes base que sean lo más pequeñas posible para reducir el tamaño final de la imagen.
- **Minimización de capas:** Combinar múltiples comandos RUN en una sola instrucción para reducir el número de capas en la imagen.
- **Eliminación de archivos innecesarios:** Utilizar .dockerignore para excluir archivos y directorios que no son necesarios en la imagen final.

13. Seguridad

Esta es una de las secciones que considero más importantes a la hora de dockerizar una aplicación, ya que si no se tienen en cuenta ciertos aspectos de seguridad, la aplicación puede quedar expuesta a ataques y vulnerabilidades.

Para garantizar la integridad y confidencialidad de los datos, resulta fundamental implementar buenas prácticas de seguridad en la configuración y gestión de los contenedores Docker.

Como por ejemplo, asegurarse de que el archivo .env no se suba a ningún repositorio público, ya que contiene información sensible como credenciales de la base de datos.

13.1. Gestión de secretos

Utilización de herramientas como Docker Secrets o Github Actions Secrets para gestionar credenciales y datos sensibles de forma segura.

13.2. Usuarios no privilegiados

Configurar los contenedores para que se ejecuten con usuarios no privilegiados, minimizando el riesgo de escalada de privilegios en caso de una brecha de seguridad.

13.3. Escaneo de vulnerabilidades

Implementar herramientas de escaneo de vulnerabilidades para identificar y mitigar posibles riesgos en las imágenes Docker utilizadas.

Algunas herramientas populares incluyen Trivy, Clair y Anchore.

- **Trivy**: Es una herramienta de escaneo de vulnerabilidades de código abierto que detecta vulnerabilidades en imágenes de contenedores, sistemas de archivos y repositorios de código.
- **Clair**: Es una herramienta de análisis estático de vulnerabilidades para contenedores.
- **Anchore**: Es una plataforma de análisis de seguridad de contenedores que ofrece escaneo de vulnerabilidades, cumplimiento de políticas y gestión de imágenes.

13.4. Actualización de imágenes base

Mantener las imágenes base actualizadas para beneficiarse de las últimas correcciones de seguridad y mejoras.

Para ello, se ha implementado la opción de que cada vez que se haga un push de este proyecto a GitHub, se construya y suba automáticamente la imagen Docker actualizada a Docker Hub mediante GitHub Actions; y a su vez, se envía un mensaje por telegram notificando que la imagen ha sido actualizada correctamente.

14. Troubleshooting

Durante el proceso de dockerización, pueden surgir diversos problemas que afectan al correcto funcionamiento de los contenedores. Esta sección aborda algunos de los problemas más comunes y sus soluciones.

Troubleshooting es el proceso de identificar, diagnosticar y resolver problemas técnicos en sistemas y aplicaciones. En el contexto de Docker, implica analizar errores en contenedores, imágenes o redes para restaurar su funcionamiento adecuado.

14.1. Problemas comunes

- **Contenedor no inicia:** Verificar los logs del contenedor para identificar errores específicos.
- **Problemas de conexión a la base de datos:** Asegurarse de que la cadena de conexión es correcta y que el contenedor de MongoDB está en ejecución.
- **Errores de permisos:** Comprobar que los volúmenes y archivos tienen los permisos adecuados para ser accedidos por los contenedores.
- **Problemas de red:** Verificar la configuración de redes en Docker Compose y asegurarse de que los contenedores pueden comunicarse entre sí.
- **Problemas de dependencias:** Asegurarse de que todas las dependencias necesarias están instaladas en la imagen Docker.
- **Errores en el Dockerfile:** Revisar la sintaxis y las instrucciones del Dockerfile para asegurarse de que son correctas.
- **Problemas con Docker Compose:** Verificar la configuración del archivo docker-compose.yml para asegurarse de que todos los servicios están definidos correctamente.
- **Problemas con credenciales:** Asegurarse de que las variables de entorno están configuradas correctamente y que las credenciales son válidas.

14.2. Errores de conexión

1. **Verificar estado de contenedores:** Usar `docker ps` para asegurarse de que ambos contenedores (API y MongoDB) están en ejecución.
2. **Revisar cadena de conexión:** Confirmar que la cadena de conexión a MongoDB utiliza el nombre del servicio correcto (por ejemplo, "mongo") y el puerto adecuado (27017).
3. **Comprobar logs:** Utilizar `docker logs <nombre_contenedor>` para revisar los logs de la API y MongoDB en busca de errores relacionados con la conexión.
4. **Verificar redes:** Asegurarse de que ambos contenedores están en la misma red de Docker Compose y pueden comunicarse entre sí.
5. **Probar conectividad manualmente:** Acceder al contenedor de la API usando `docker exec -it <nombre_contenedor>/bin/bash` y utilizar herramientas como `ping` para probar la conexión a MongoDB.
6. **Revisar configuración de firewall:** Asegurarse de que no hay reglas de firewall que bloqueen la comunicación entre los contenedores.

14.3. Problemas de permisos

1. **Verificar permisos de volúmenes:** Asegurarse de que los volúmenes montados tienen los permisos adecuados para ser accedidos por los contenedores.

2. **Revisar usuario del contenedor:** Confirmar que el contenedor se está ejecutando con un usuario que tiene los permisos necesarios para acceder a los archivos y directorios.

3. **Ajustar permisos en el host:** Si es necesario, cambiar los permisos de los archivos y directorios en el host utilizando comandos como `chmod` o `chown`.

4. **Comprobar SELinux/AppArmor:** Si se utilizan sistemas de seguridad como SELinux o AppArmor, asegurarse de que no están bloqueando el acceso a los volúmenes. (No es el caso en este proyecto, pero es importante tenerlo en cuenta en otros entornos como Red Hat o CentOS, que son otras distribuciones de Linux con políticas de seguridad más estrictas).

14.4. Debugging de contenedores

El Debugging de contenedores es el proceso de identificar y solucionar problemas dentro de contenedores Docker. Implica revisar logs, inspeccionar configuraciones y ejecutar comandos para diagnosticar fallos y restaurar el funcionamiento adecuado de los contenedores.

Es fundamental para mantener la estabilidad y rendimiento de aplicaciones que se ejecutan en entornos de contenedores.

Para ello, se pueden utilizar herramientas y comandos como **docker logs**, **docker exec** y **docker inspect** para obtener información detallada sobre el estado y comportamiento de los contenedores.

Por ejemplo, en este proyecto ha sido muy utilizado el comando

```
docker logs <nombre_contenedor>
```

para revisar los logs de ambos contenedores y solucionar posibles problemas.

15. Problemas encontrados

Durante el proceso de dockerización, se han encontrado varios problemas que han requerido soluciones específicas para garantizar el correcto funcionamiento de la aplicación.

La mayoría de los problemas fueron solventados individualmente, pero al ir avanzando entre días en el proyecto, o me saltaba pasos sin darme cuenta, o cometia pequeños errores a la hora de introducir por terminal los comandos necesarios.

Así que para facilitar el proceso y evitar estos errores, se creó un script **setup.sh** que contiene todos los comandos necesarios para construir y ejecutar los contenedores, facilitando así el proceso de despliegue con un solo comando. Por lo que lo único que hay que hacer es ejecutar el script y todo se realiza automáticamente.

15.1. Problema 1: Conexión a MongoDB fallida

Descripción: La API no podía conectarse a la base de datos MongoDB, lo que resultaba en errores al intentar realizar operaciones CRUD.

Causa: La api se quedaba colgada, debido a que el contenedor de la base de datos no se estaba levantando correctamente, y por tanto, la API no podía establecer la conexión.

Solución: Se corrigió la cadena de conexión en las variables de entorno, asegurándose de utilizar el nombre del servicio "mongoz el puerto correcto (27017). Además, se verificó que el contenedor de MongoDB estuviera en ejecución antes de iniciar la API, utilizando la opción `depends_on` en el archivo docker-compose.yml.

Uso del **healthcheck**: Esto garantiza que la base de datos esté lista para aceptar conexiones antes de que la API intente conectarse a ella. En el docker-compose.yml, se añadió la siguiente configuración al servicio de MongoDB:

```
mongo:
    image: mongo:latest
    container_name: mongo
    volumes:
    - mongo_data:/data/db
    healthcheck:
    test: ["CMD", "mongo", "--eval", "db.adminCommand('ping')"]
    interval: 5s
    timeout: 3s
    retries: 5
```

15.2. Problema 2: Permisos insuficientes en volúmenes

Descripción: El contenedor de MongoDB no podía acceder al volumen montado, lo que impedía la persistencia de datos.

Causa: Los permisos del directorio estaban partidos entre superusuario y usuario normal, lo que causaba conflictos al intentar acceder a los datos.

Solución: Se ajustaron los permisos del directorio en el host utilizando el comando `chmod` para asegurar la accesibilidad adecuada por parte del contenedor.

15.3. Problema 3: Errores con la actualización de la imagen

Descripción: La imagen Docker no se actualizaba correctamente al realizar cambios en el código.

Causa: Existia una mala estructuración del Dockerfile que impedía la correcta reconstrucción de la imagen. En el caso de este proyecto, el error concreto fue en el apartado CMD, que apuntaba a un script que no existía.

Solución: Se revisó y corrigió el Dockerfile para asegurar que todas las instrucciones fueran correctas y que la imagen se construyera adecuadamente. Además, se implementó un script de automatización para facilitar la reconstrucción y despliegue de la imagen.

15.4. Problema 4: Conflictos de puertos

Descripción: Al intentar iniciar los contenedores, se producían errores debido a conflictos de puertos en el host.

Causa: Otro servicio en el host ya estaba utilizando los puertos asignados a los contenedores, en este caso el puerto 3000 para la API.

Solución: Se modificaron los mapeos de puertos en el otro servicio que los tenía abiertos para evitar conflictos.disponibles.

15.5. Problema 5: Actualización del tag en GitHub

Descripción: Al realizar un push a GitHub, la imagen Docker no se actualizaba correctamente en Docker Hub.

Causa: El tag de la imagen no se estaba actualizando correctamente en el workflow de GitHub Actions. Esto era debido a que la versión del tag estaba hardcodeada en el archivo del workflow, en el que ponía siempre v1.0.0.

Solución: Se modificó el workflow para incluir un paso que actualizara el tag de la imagen con cada push, asegurando que la última versión de la imagen estuviera siempre disponible en Docker Hub.

16. Conclusiones

16.1. Ventajas de la dockerización

Este proyecto me ha permitido comprender las ventajas de la dockerización, tales como la portabilidad, escalabilidad y consistencia en los entornos de desarrollo y producción.

Es una herramienta que me está pareciendo fundamental para sobre todo la escalabilidad de aplicaciones, ya que permite desplegar múltiples instancias de una aplicación de manera sencilla y eficiente. Además, la seguridad que te aporta el hecho de aislar las aplicaciones en contenedores es un punto muy a favor.

Lo fundamental en mi caso ha sido el entender correctamente el fin que tiene cada parte y su orden en la dockerización.

16.2. Resultados obtenidos

Pese a que en un principio pueda parecer un proceso complejo, una vez se entienden los conceptos básicos y se sigue un proceso estructurado, la dockerización de aplicaciones se vuelve mucho más manejable.

En un principio, tuve algunos fallos que me hicieron perder algo de tiempo, pero al ir entendiendo por mi cuenta las ventajas anteriormente mencionadas, y al ir corrigiendo los errores, me picó mucho la curiosidad por seguir aprendiendo sobre Docker y sus aplicaciones en el desarrollo de software.

Finalmente, he conseguido dockerizar correctamente una API con MongoDB, y he implementado un chatbot en Telegram para interactuar con la API.

16.3. Ventajas del .sh

La creación del script **setup.sh** ha sido una de las mejores decisiones que he tomado en este proyecto, ya que ha permitido automatizar el proceso de construcción y despliegue de los contenedores, evitando errores manuales y facilitando la repetición del proceso en el futuro.

Este archivo es sencillo de crear y puede ser muy útil en proyectos futuros donde se requiera dockerización.

16.4. Trabajo adicional

De forma adicional, por entretenimiento y para practicar más con Docker, he creado un repositorio público en GitHub donde he creado otro script que muestra información relevante de Docker de manera visual y sencilla, llamado **dockerinfo**.

Aquí lo dejo por si alguien quiere echarle un vistazo: dockerinfo - santimartinezzgb.

17. Referencias

1. **Documentación oficial de Docker:** https://docs.docker.com/
2. **Documentación oficial de Docker Compose:** https://docs.docker.com/compose/
3. **Documentación oficial de Telegram Bots:** https://core.telegram.org/bots
4. **GitHub Actions Documentation:** https://docs.github.com/en/actions