

Arquitectura Aplicacion Movil

Manuel Añel García

7 de diciembre de 2025

1. Introduction

En este documento se va a documentar la dockerizacion de una API de gestion de usuarios y grupos y su subida a docker hub y su automatización con actions de github.

2. Tecnologías utilizadas

- NodeJS con Express: Para el backend de la aplicación (Servidor)
- Mongo: Para la base de datos
- Moongoose: Es un paquete para la conexión entre el backend y la base de datos
- Coors: Para la seguridad de las peticiones HTTP
- Docker: La plataforma con la que vamos a trabajar para subir la API
- Workflow: Para las actions de github

3. Arquitectura de la aplicacion

Arquitectura Backend (API REST) en contenedores

- Servidor (Node.js): Maneja las rutas, realiza operaciones en la base de datos.
- Base de datos (MongoDB): Almacena los datos de los usuarios y grupos.

4. Backend

Ahora se va a explicar el funcionamiento de la API en cada caso

4.1. Conexion a Mongo

El backend se conecta mediante el paquete moongoose en el backend haciendo un moongoose.connect() y dentro de los parentesis la dirección del servidor que tenemos creado

```
1 let direccion = direccion_mongo;
2
3 mongoose.connect(direccion)
4 .then(() => console.log('Conectado a la base de datos'))
5 .catch(err => console.error('Error de conexión a la base de datos:', err));
```

4.2. Servidor Express

En este caso se ve como se crea el servidor mediante express con app = express() y se crean los endpoints que son las peticiones que se van a hacer desde la aplicacion al servidor para que se ejecuten, luego para desplegarlo se usa app.listen(), esto es igual para los grupos

- GET: Es para recojer todos los usuarios que hay en la base de datos
- POST: Mediante el cuerpo de un usuarios (sus parametros: nombre, apellido etc..) se recoje y se guarda el usuario en la base de datos con esos parametros
- PUT: Basicamente mediante un id (el de mongo por defecto) se busca el usuario y mediante un cuerpo como en el anterior que se especifica en la aplicacion de modifica el usuario ya existente
- DELETE: Mediante un id pasado por la aplicacion se elimina el usuario con el que concuerda

```
1 app.get('/usuarios', async (req, res) => {
2   const usuarios = await Usuario.find();
3   res.json(usuarios);
4 });
5
6 app.post('/usuarios', async (req, res) => {
7   const nueva = new Usuario(req.body);
8   await nueva.save();
9   res.json(nueva);
10});
11
12 app.put('/usuarios/:id', async (req, res) => {
13   const actualizada = await Usuario.findOneAndUpdate({ id: Number(req.params.id) },
14     req.body, { new: true });
15   res.json(actualizada);
16 });
17
18 app.delete('/usuarios/:id', async (req, res) => {
19   await Usuario.findOneAndDelete({ id: Number(req.params.id) });
20});
```

4.3. Variables de entorno

Para la seguridad de mi API creé variables de entorno que básicamente la informacion delicada esta en un archivo .env y en donde deberia estar tiene una variable definida en el .env y eso no se sube a docker y luego cuando lo inicies sin eso se puede poner los valores que quieras por consola

Esto es lo que tenemos en el .env

```
1 MONGO_URI='mongodb://localhost:27017/GestionUG'
```

```
1 require('dotenv').config();
2 const direccion_mongo = process.env.MONGO_URI;
```

5. Docker

Vamos ahora a explicar como dockerizar la API

5.1. Crear Dockerfile

Sirve para que siga los pasos para construir la imagen despues

```
1 FROM node:18
2
3 WORKDIR /app
4
5 COPY package*.json ./
6
7 RUN npm install
8
9 COPY . .
10
11 EXPOSE 3000
12
13 CMD ["node", "server.js"]
```

5.2. Crear la imagen

En la carpeta donde está el Dockerfile hay que ejecutar este comando

```
1 docker build -t manu8944/api-docker:v1.0
```

5.3. Subir la imagen a Dockerhub

Ponemos los siguientes comandos para subirla imagen que creamos antes a docker hub

```
1 docker login
2 docker push manu8944/api-docker:v1.0
```

5.4. Docker compose

El docker compose es un archivo para crear el contenedor en nuestro dockerdesktop

```
1 version: "3.9"
2 services:
3   backend:
4     image: ejemplo-api:v1.0
5     ports:
6       - "3000:3000"
7     environment:
8       - MONGO_URI=mongodb://localhost:27017/GestionUG
9     depends_on:
10      - db
11
12   db:
13     image: mongo
14     ports:
15       - "27017:27017"
16     volumes:
17       - mongo_data:/data/db
18
19 volumes:
20   mongo_data:
```

Lo que se menciona en el código son las imágenes para subir, los puertos usados, url de conexión a la bd etc..

5.5. Construir contenedor con DockerCompose

Es para crear el contenedor localmente

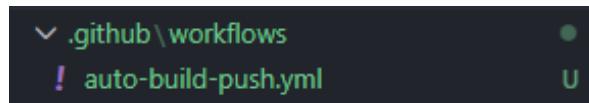
```
1 docker compose up -d
```

6. Automatizar subida

Ahora vamos a ver como se hace para que al hacer push en nuestro repositorio de github automáticamente se haga un push de la imagen en docker hub

6.1. Crear archivo

Creamos esta estructura de carpetas con el archivo



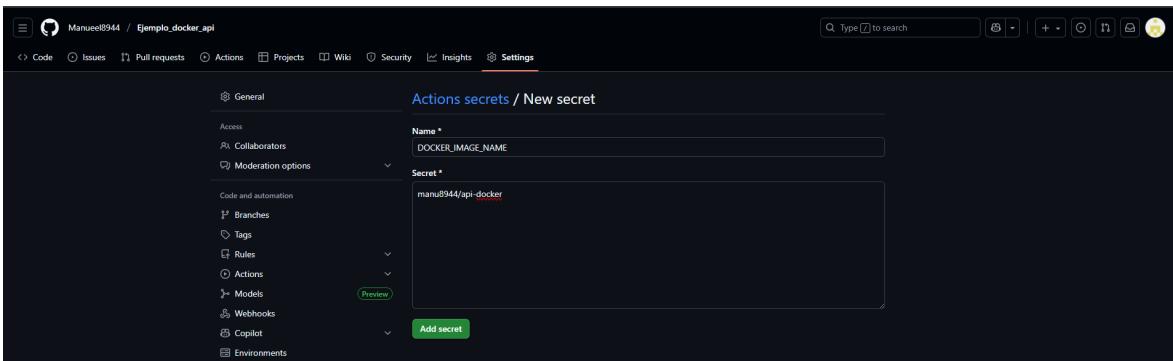
Ponemos esto para que se haga el push a DH cuando se haga un commit

```
1 name: auto-build-push
2
3 on:
4   push:
5     branches:
6       - main
7     workflow_dispatch:
8
9   jobs:
10    build-and-push:
11      runs-on: ubuntu-latest
12
13    steps:
14      - name: Checkout código
15        uses: actions/checkout@v4
16
17      - name: Configurar Docker Buildx
18        uses: docker/setup-buildx-action@v3
19
20      - name: Login a Docker Hub
21        uses: docker/login-action@v3
22        with:
23          username: ${{ secrets.DOCKERHUB_USERNAME }}
24          password: ${{ secrets.DOCKER_PASSWORD }}
25
26      - name: Construir y subir imagen Docker
27        uses: docker/build-push-action@v5
28        with:
29          context: .
30          file: ./Dockerfile
31          push: true
32          tags: |
33            ${{ secrets.DOCKER_IMAGE_NAME }}:latest
```

6.2. Secrets de github

Ahora en el repositorio hay que poner los secrets que son básicamente las variables de entorno que vimos en el código anterior.

Así se hacen los secrets:



Así es como quedan todos los secrets:

Name	Last updated		
🔒 DOCKERHUB_USERNAME	1 minute ago	Edit	Delete
🔒 DOCKER_IMAGE_NAME	now	Edit	Delete
🔒 DOCKER_PASSWORD	1 minute ago	Edit	Delete
🔒 DOCKER_TAG	now	Edit	Delete

6.3. Hacer commit

Ahora hacemos el commit para que funcione la subida de imagen, aquí vemos como ha funcionado

7. Mensaje telegram

Ahora vamos a añadirle que envie un mensaje por telegram cada vez que hagamos un se suba la imagen, para ello hay que crear un bot de telegram con el BotFather y cojer el token y el chatID y lo guardamos en secrets

Ahora le añadimos esto a nuestro yml

```

1 - name: Notificar por Telegram
2   if: success()
3   run: |
4     MESSAGE="Nueva imagen subida a Docker Hub\nRepo: ${{ github.repository }}\
5       nCommit: ${{ github.sha }}\nImagen: ${{ secrets.DOCKER_IMAGE_NAME }}:
6         latest"
7     curl -s -X POST "https://api.telegram.org/bot${{ secrets.TELEGRAM_BOT_TOKEN
8       }}/sendMessage" \
9       -d chat_id="${{ secrets.TELEGRAM_CHAT_ID }}" \
10      -d text="$MESSAGE"

```

Vemos como funciona aunque los saltos de línea no era con /n

