

Documentación de la api Docker

Andrea Sofía Pais Dos Santos

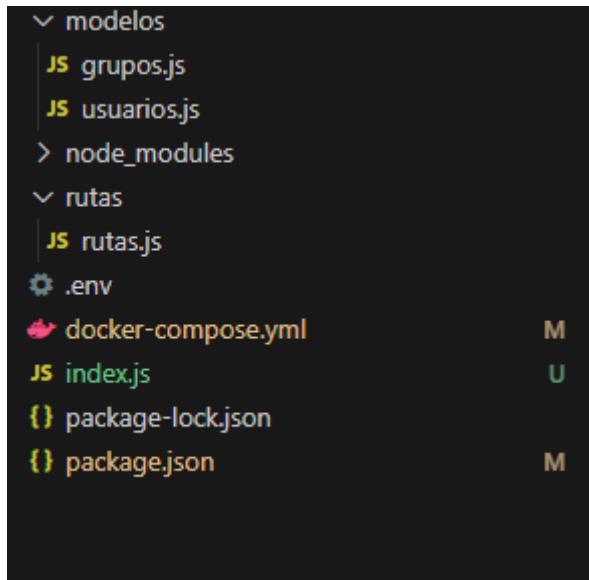
December 4, 2025

1 Índice

- | | |
|---|-----------|
| 2. Creación de la API con base de datos local | página 3 |
| 3. Creación de la imagen | página 6 |
| 4. Subir la imagen a DockerHub | página 8 |
| 5. GitHub Actions | página 9 |
| 6. Notificación Action | página 12 |

2 Creación de la API con base de datos local

Para construir una API con MongoDB Local usando docker, primero vamos a inicializar el proyecto "npm init -y" y crear, por ahora, la siguiente estructura de carpetas y archivos. Luego para la creación de una imagen vamos a tener que crear otros documentos extras.



Desglose de los archivos necesarios junto a su contenido.

- Archivo index.js, contiene las importaciones y la conexión a la base de datos:

```
1 import express from "express";
2 import cors from "cors";
3 import dotenv from "dotenv";
4 import mongoose from "mongoose";
5 import rutas from "./rutas/rutas.js";

6 dotenv.config();
7 const app = express();
8 app.use(cors());
9 app.use(express.json());

11 app.use(rutas);
12 const uri = process.env.MONGO_URI;
13 export async function conectarBD(){
14     try{
15         await mongoose.connect(uri,{});
16         console.log("Base de datos conectada");
17     }catch(error){
18         console.log("Error conectandose a la base de datos", error);
19     }
20 }
21

23 const PORT = process.env.PORT || 3000;
24
25 conectarBD().then(async () => {
26     app.listen(PORT, () => console.log(`API en ${PORT}`));
27 }) ;
```

- Archivo rutas.js, está compuesto por los EndPoints (GET,POST). GET para obtener los usuarios y los grupos y POST para crear esos usuarios y grupos.

```

1 import express from "express";
2 import Usuario from "../modelos/usuarios.js";
3 import Grupo from "../modelos/grupos.js";
4
5 const router = express.Router();
6
7 router.get("/usuarios", async (req, res) => {
8   const usuarios = await Usuario.find();
9   res.json(usuarios);
10 });
11
12 router.get("/grupos", async (req, res) => {
13   const grupos = await Grupo.find();
14   res.json(grupos);
15 });
16
17 router.post("/usuarios", async (req, res) => {
18   const nuevoUsuario = new Usuario(req.body);
19   await nuevoUsuario.save();
20   res.json(nuevoUsuario);
21 });
22
23 router.post("/grupos", async (req, res) => {
24   const nuevoGrupo = new Grupo(req.body);
25   await nuevoGrupo.save();
26   res.json(nuevoGrupo);
27 });
28
29 router.get("/", (req, res) => {
30   res.json({ mensaje: "API funcionando correctamente" });
31 });
32
33 export default router;

```

- Carpeta "modelos" con grupos.js y usuarios.js, que contienen el esquema con las propiedades de ambos elementos.

```

1 ////////////////////////////// Archivo grupos /////////////////////
2 import mongoose from "mongoose";
3
4 const GrupoEschema = new mongoose.Schema({
5   nombre_grupo: {
6     type: String,
7     required: true
8   },
9   participantes: {
10     type: Array,
11     required: true
12   }
13 });
14
15 export default mongoose.model("Grupo", GrupoEschema);
16
17
18

```

- Archivo .env es el archivo de configuración que almacena datos delicados y el puerto que usaremos en este caso el 3000.

```
1 MONGO_URI=mongodb://<nombre>:<contrasena>@localhost:27017/mi_base_datos?admin  
2 PORT=3000
```

- Archivo docker-compose.yml contiene unos servicios definidos, servicio "api" (nuestra aplicación) y servicio "db" (la base de datos MongoDB).

```
1 version: '3.8'
2
3 services:
4   api:
5     image: ejemplo-api:v1.0.0
6     ports:
7       - "3000:3000"
8     environment:
9       - MONGO_URI=mongodb://nombre:contrasena@db:27017/admin
10    depends_on:
11      - db
12
13  db:
14    image: mongo:latest
15    ports:
16      - "27017:27017"
17    environment:
18      - MONGO_INITDB_ROOT_USERNAME=nombre
19      - MONGO_INITDB_ROOT_PASSWORD=contrasena
20    volumes:
21      - mongo_data:/data/db
22    restart: unless-stopped
23
24 volumes:
25   mongo_data:
```

Tras tener la API estructurada, la construimos con el siguiente comando:

```
1 docker-compose up
```

Ahora en Docker deberíamos de ver al API creada.

	□	docker_api	-	-	N/A	⋮	⋮
1	□	db-1	13d9ab9f2215	mongolite 27017:27017	N/A	⋮	⋮
2	□	api-1	e487247710ed	ejemplo-ap 3000:3000	N/A	⋮	⋮

Hay que acordarse de que para utilizarlos hay que importar todo lo necesario, con el siguiente comando:

```
1 npm install express mongoose .... "todo lo que necesitemos"
```

3 Creación de la imagen

Para la construcción de la nueva imagen creamos los siguientes archivos y modificamos el docker-compose.yml.

- Archivo Dockerfile va a contener las instrucciones que sirven para crear la imagen.

```
1 FROM node:20
2 WORKDIR /app
3 COPY package*.json .
4 RUN npm install --production
5 COPY .
6 EXPOSE 3000
7 CMD ["node", "index.js"]
```

Lo que nos quiere cedir esto es que desde node vamos a crear "app" que es donde vamos a trabajar desde ahora, se copiará el package*.json, va a ejecutar esa instalación compiendo todo en el puerto 3000.

- Archivo .dockerignore va a ignorar archivos pesados e información sensible.

```
1 node_modules
2 npm-debug.log
3 .git
4 .gitignore
5 .env
6 docker-compose.yml
```

Tras tener esos archivos y hayamos modificado el docker-compose cambiando el nombre de la imagen, vamos a construir la imagen, para ellos usamos el siguiente comando:

```
1 docker build -t ejemplo-api:v1.0.0 .
```

The screenshot shows the VS Code interface with a dark theme. The Explorer sidebar on the left lists files and folders: DOCKER_API (modelos, node_modules, rutas, .env, docker-compose.yml), JS index.js, {} package-lock.json, and {} package.json. The main editor area shows a Dockerfile with the following content:

```

FROM node:20
WORKDIR /app
COPY package*.json .
RUN npm install --production
COPY .
EXPOSE 3000
CMD ["node", "index.js"]

```

The terminal tab at the bottom displays the command `docker build -t ejemplo-api:v1.0.0 .` being run in a powershell window. The output shows the build process, including pulling the Node.js image and copying files into the container.

Ahora al acceder a docker, en images tendremos lo siguiente:



4 Subir la imagen a DockerHub

Para subir la imagen a DockerHub, lo primero es tener una cuenta e iniciar sesión. Tras iniciar sesión, creamos un repositorio con el nombre que queramos.

The screenshot shows the DockerHub interface for creating a new repository. On the left, there's a sidebar with options like Repositories, Hardened Images, Collaborations, Settings, Billing, Usage, Pulls, and Storage. The main area is titled 'Create repository' with a field for 'Repository Name' containing 'api.docker'. Below it is a 'Short description' input field and a note about visibility: 'A short description to identify your repository. If the repository is public, this description is used to index your content on Docker Hub and in search engines, and is visible to users in search results.' Under 'Visibility', there are two options: 'Public' (selected) and 'Private' (radio button). A note says 'Appears in Docker Hub search results' for Public and 'Only visible to you' for Private. At the bottom are 'Cancel' and 'Create' buttons.

Y desde la terminal, ejecutamos el siguiente comando para que suba la imagen:

```
1 docker push andreasofia31/api_docker:v1.0.0
```

Si está correctamente subida, en el repositorio de DockerHub debería salir la imagen que hemos creada:

The screenshot shows the DockerHub repository page for 'andreasofia31/api_docker'. The sidebar on the left is identical to the previous screenshot. The main page shows the repository details: 'andreasofia31/api_docker' (Public), last pushed less than a minute ago, repository size: 388.3 MB, stars: 0, forks: 0. Below this is a 'General' tab with sub-options: Tags, Image Management (Beta), Collaborators, Webhooks, and Settings. The 'Tags' section shows one tag: 'v1.0.0'. To the right, there's a 'Docker commands' section with a 'Public view' button and a command: 'docker push andreasofia31/api_docker:tagname'. Further down, there's a 'buildcloud' section with a 'Build with Docker Build Cloud' button and a note about accelerating build times. At the bottom, there's a 'Repository overview' section with an 'Incomplete' status and a 'Add overview' button.

5 GitHub Actions

Vamos a crear un workflow de GitHub Actions que automatiza la construcción y publicación de imágenes Docker. Para ellos vamos a añadir una carpeta ".github" con una subcarpeta llamada "workflows" en ella vamos crear el archivo "docker-push.yml".

En el archivo que creamos vamos a añadir el siguiente código que va a ejecutarse desde el repositorio al pushear cada cambio. Y va a tener la siguiente información:

- Push Automático que afecta a la rama u al cualquier cambio en el repositorio.
- Variables de entorno que se almacenan en GitHub Secrets, que necesitaremos "DOCKER_USERNAME" y "DOCKER_PASSWORD".
- Luego todos los trabajos que va a tener que realizar ejecutando e un servidor Ubuntu.

```
1 name: Build and Push Docker Image
2
3 on:
4   push:
5     branches:
6       - main
7       - master
8     paths:
9       - './**'
10      - '.github/workflows/docker-push.yml'
11   workflow_dispatch: # Permite ejecutar manualmente
12
13 env:
14   DOCKER_IMAGE_NAME: ${{ secrets.DOCKER_USERNAME }}/api_docker
15   DOCKER_TAG: latest
16
17 jobs:
18   build-and-push:
19     runs-on: ubuntu-latest
20
21     steps:
22       - name: Checkout codigo
23         uses: actions/checkout@v4
24
25       - name: Configurar Docker Buildx
26         uses: docker/setup-buildx-action@v3
27
28       - name: Login a Docker Hub
29         uses: docker/login-action@v3
30         with:
31           username: ${{ secrets.DOCKER_USERNAME }}
32           password: ${{ secrets.DOCKER_PASSWORD }}
33
34       - name: Construir y subir imagen Docker
35         uses: docker/build-push-action@v5
36         with:
37           context: .
38           file: ./Dockerfile
39           push: true
40           tags: ${{ env.DOCKER_IMAGE_NAME }}:${{ env.DOCKER_TAG }}
```

```

41 cache-from: type=registry,ref=${{ env.DOCKER_IMAGE_NAME }}:buildcache
42 cache-to: type=inline
43
44 - name: Mostrar informacion de la imagen
45   run: |
46     echo "Imagen construida y subida exitosamente:"
47     echo " - Imagen: ${{ env.DOCKER_IMAGE_NAME }}:${{ env.DOCKER_TAG }}"
48     echo " - Docker Hub: https://hub.docker.com/r/{{ secrets.DOCKER_USERNAME
}}/api_docker"

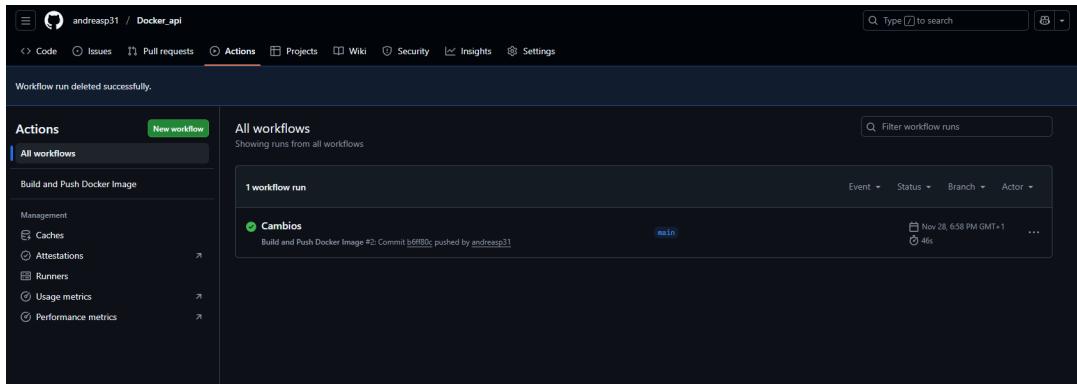
```

5.1 Requisitos previos

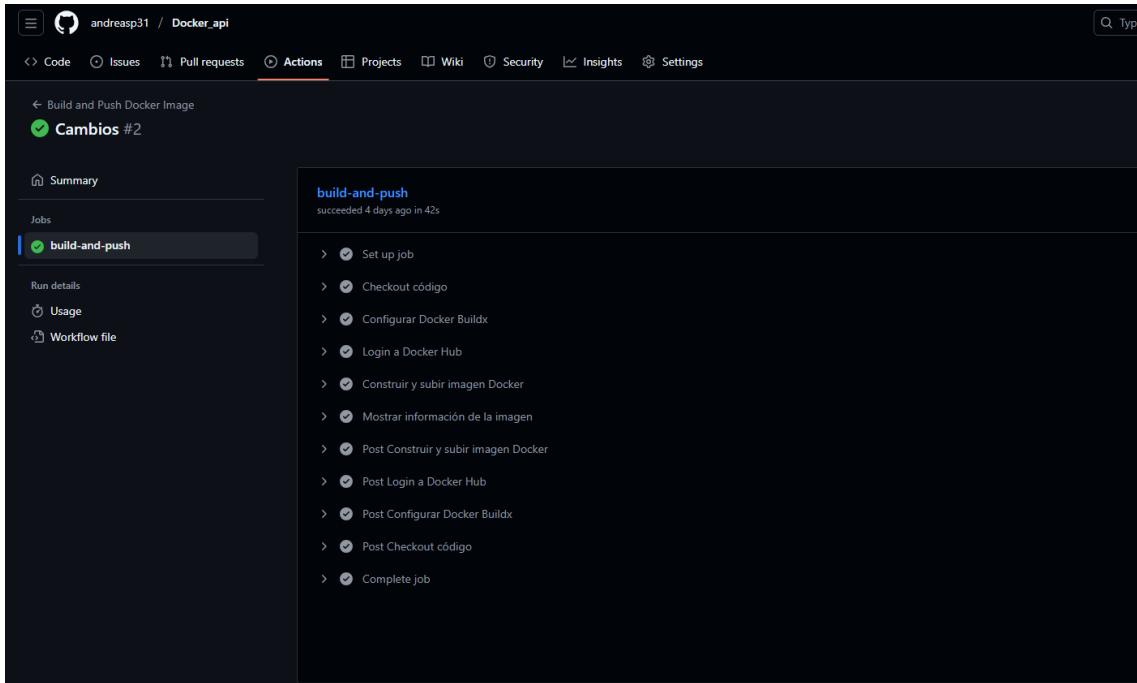
Acceder al repositorio de GitHub donde tengamos el proyecto subido. Dirigirnos configuraciones de repositorio y buscar el apartado de "Secrets and variables" ahí añadimos el nombre de las variables de entorno con sus declaraciones. En este caso, para usuario ponemos el nuestro de dockerHub y como contraseña usamos un token creado en DockerHub para poder acceder.

Name	Last updated
DOCKER_PASSWORD	4 days ago
DOCKER_USERNAME	4 days ago

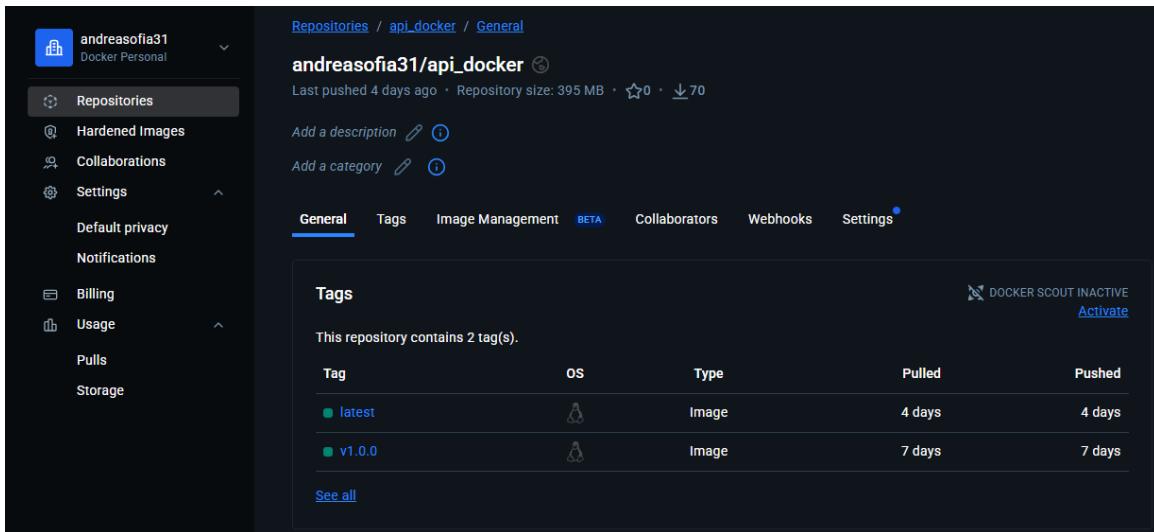
Si están correctamente creadas y el archivo está correcto, al pushear los nuevos cambios debería aparecer en el repositorio de GitHub un apartado llamado Actions. Y en el vemos que se ejecutó el workflow.



Y dentro vemos todos los servicios realizados con éxito.



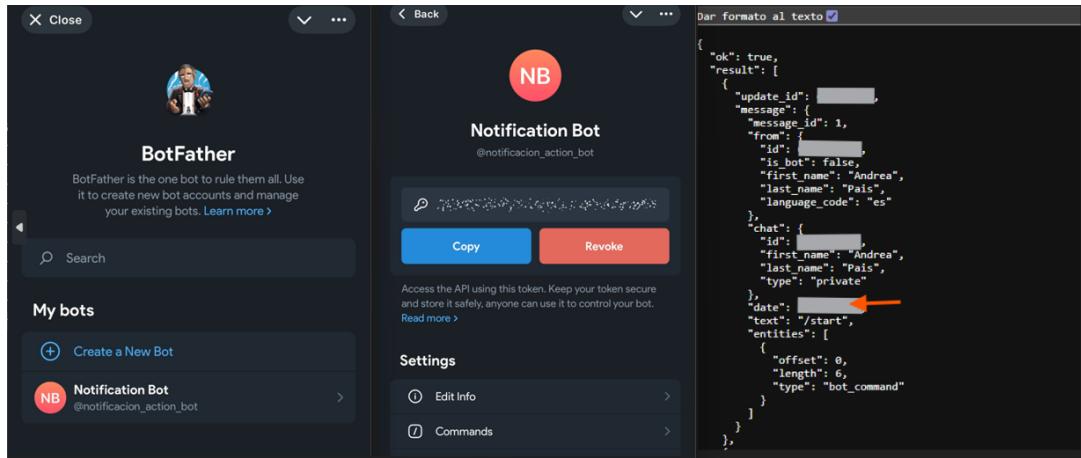
Por último para ver que todo está correcto, vamos a DockerHub y debería salir la imagen actualizada.



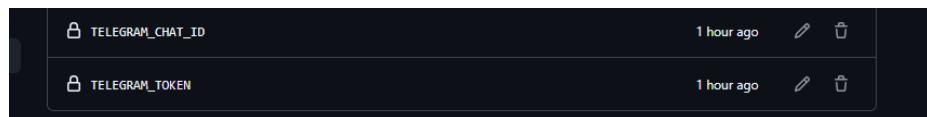
6 Notificación Action

Para crear un GitHub Actions que nos notifique cada vez que realizamos un push en el repositorio por ejemplo vía Telegram, tenemos que realizar los siguientes pasos:

- Creamos nuestro Bot en telegram, para ello tenemos que buscar @BotFather y le damos un nombre al bot y un nombre de usuario. Al crearlo nos genera un Token y necesitamos también el id del chat para mandar las notificación. Para ello le mandamos a nuestro bot cualquier mensaje. Y si nos dirigimos a <https://api.telegram.org/bot<miToken>/getUpdates> nos sale un JSON y obtenemos el id de ahí.



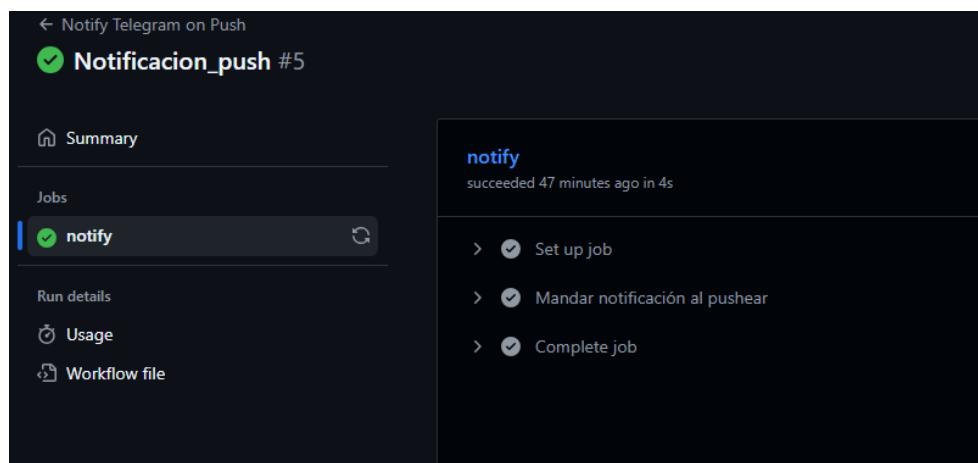
- Ahora en los ajustes del repositorio de GitHub creamos nuevas variables de entorno, el id y el token que es mediante lo que va a acceder el bot de telegram.



- Luego creamos un nuevo archivo .yml en nuestra carpeta workflows llamado "telegram-notify.yml". Dentro del archivo vamos a añadir lo siguiente:

```
1 name: Notify Telegram on Push
2
3 on:
4   push:
5     branches:
6       - main
7       - master
8
9 jobs:
10   notify:
11     runs-on: ubuntu-latest
12     steps:
13       - name: Mandar notificación al pushear
14         run:
15           curl -s -X POST https://api.telegram.org/bot${{secrets.TELEGRAM_TOKEN}}
16             /sendMessage\
17             -d chat_id=${{ secrets.TELEGRAM_CHAT_ID }} \
18             -d text=" Aquí va todo lo que nos interesa mostrar
19             (no lo puse porque no me deja latex ponerlo comentado) \\"
```

- Hacemos un commit y un push del nuevo archivo. Si todo está correcto, en el apartado de Actions se ejecutó automáticamente el workflow que creamos.



- Al mismo tiempo, ya recibimos a través del Bot un mensaje de telegram indicando que se ha pusheado algo en el repositorio.

