Documentación de la api Docker

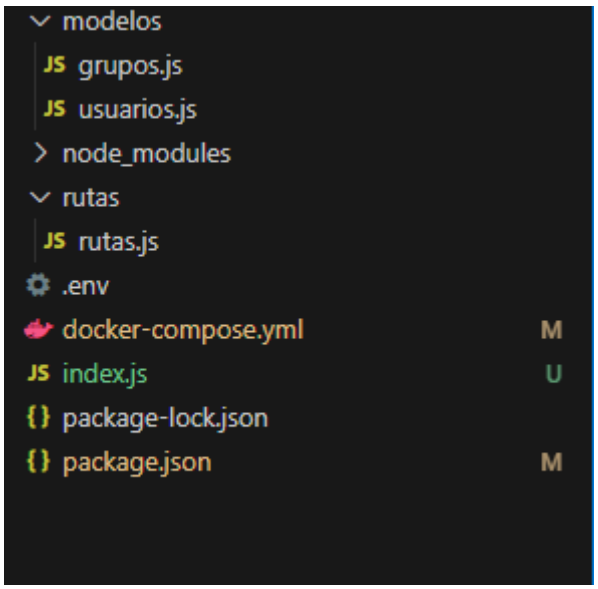
Andrea Sofía Pais Dos Santos

December 4, 2025

1 Índice

2 Creación de la API con base de datos local

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:

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import mongoose from "mongoose";
import rutas from "./rutas/rutas.js";

dotenv.config();
const app = express();
app.use(cors());
app.use(express.json());

app.use(rutas);
const uri = process.env.MONGO_URI;
export async function conectarBD(){
    try{
        await mongoose.connect(uri,{});
        console.log("Base de datos conectada");
    }catch(error){
        console.log("Error conectandose a la base de datos", error);
    }
}

const PORT = process.env.PORT || 3000;

conectarBD().then(async () => {
    app.listen(PORT, () => console.log(`API en ${PORT}`));
});
```

- 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.

```
import express from "express";
import Usuario from "../modelos/usuarios.js";
import Grupo from "../modelos/grupos.js";

const router = express.Router();

router.get("/usuarios", async (req, res) => {
  const usuarios = await Usuario.find();
  res.json(usuarios);
});

router.get("/grupos", async (req, res) => {
  const grupos = await Grupo.find();
  res.json(grupos);
});

router.post("/usuarios", async (req, res) => {
  const nuevoUsuario = new Usuario(req.body);
  await nuevoUsuario.save();
  res.json(nuevoUsuario);
});

router.post("/grupos", async (req, res) => {
  const nuevoGrupo = new Grupo(req.body);
  await nuevoGrupo.save();
  res.json(nuevoGrupo);
});

router.get("/", (req, res) => {
  res.json({ mensaje: "API funcionando correctamente" });
});

export default router;
```

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

```
\\\\\\\\\\\\\\\\\\\\ Archivo grupos \\\\\\\\\\\\\\\\\\\\\\\\\\\\
import mongoose from "mongoose";

const GrupoEschema = new mongoose.Schema({
    nombre_grupo: {
        type: String,
        required: true
    },
    participantes: {
        type: Array,
        required: true
    }
});

export default mongoose.model("Grupo", GrupoEschema);


```

```
\\\\\\\\\\\\\\\\\\\\\\ Archivo usuarios \\\\\\\\\\\\\\\\\\\\\\\\\\\\

import mongoose from "mongoose";

const UsuarioEschema = new mongoose.Schema({
    nombre_user: {
        type: String,
        required: true
    },
    apellido_user: {
        type: String,
        required: true
    },
    edad_user: {
        type: Number,
        required: true
    },
    dni_user : {
        type: String,
        required: true
    }
});

export default mongoose.model("Usuario", UsuarioEschema);
```

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

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

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

```
version: '3.8'

services:
  api:
    image: ejemplo-api:v1.0.0
    ports:
      - "3000:3000"
    environment:
      - MONGO_URI=mongodb://nombre:contrasena@db:27017/admin
    depends_on:
      - db
  db:
    image: mongo:latest
    ports:
      - "27017:27017"
    environment:
      - MONGO_INITDB_ROOT_USERNAME=nombre
      - MONGO_INITDB_ROOT_PASSWORD=contrasena
    volumes:
      - mongo_data:/data/db
    restart: unless-stopped

volumes:
  mongo_data:
```

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

```
docker-compoose up
```

Ahora en Docker deberíamos de ver al API creada.

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

```
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.

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

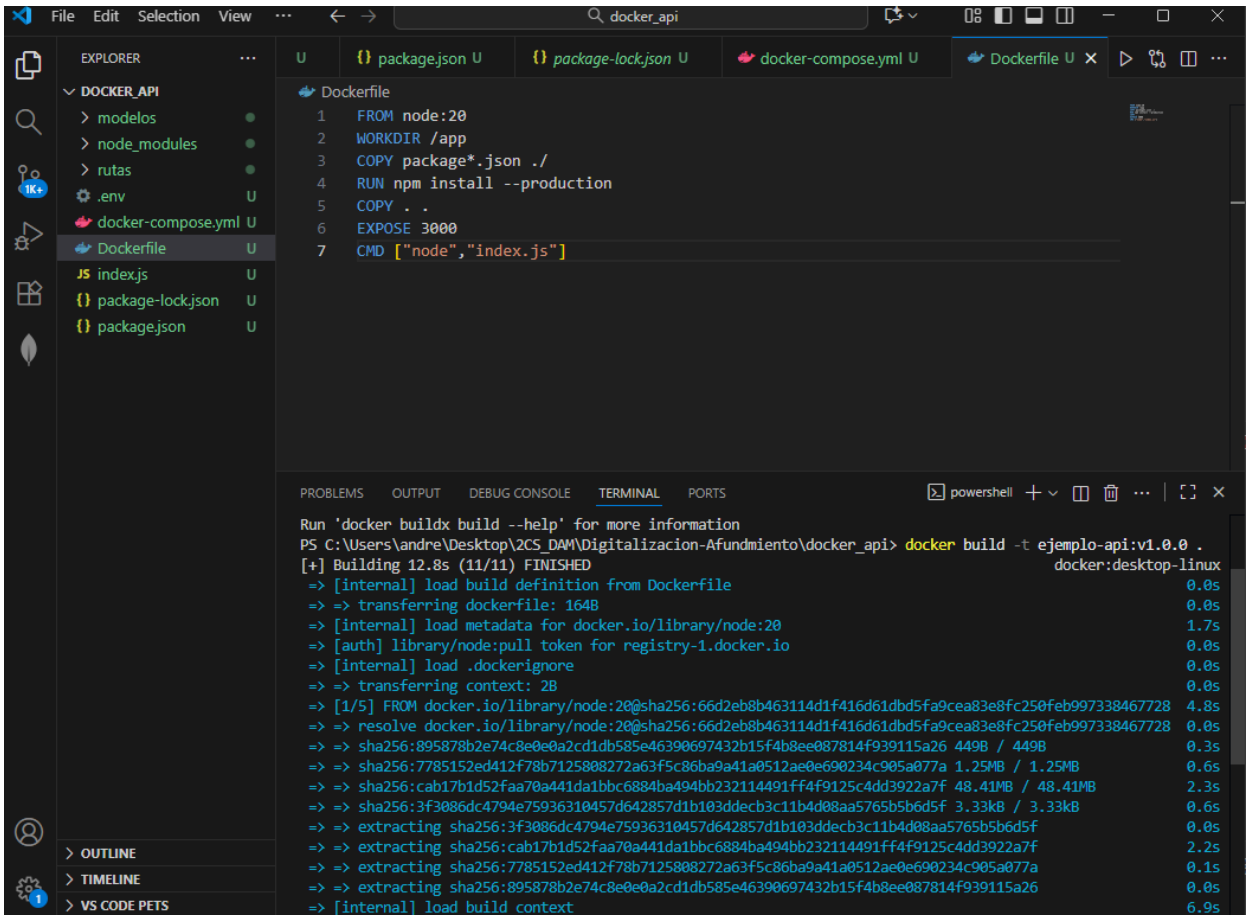
  Lo que nos quiere cedir esto es que desde node vamos a crear "app" que es donde vamos a trabajar desde ahora, se copiará el package*.json, va a ejecutar esa instalación compiando todo en el puerto 300.

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

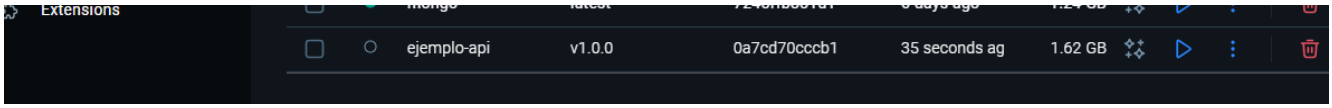
```
node_modules
npm-debug.log
.git
.gitignore
.env
docker-compose.yml
```

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:

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

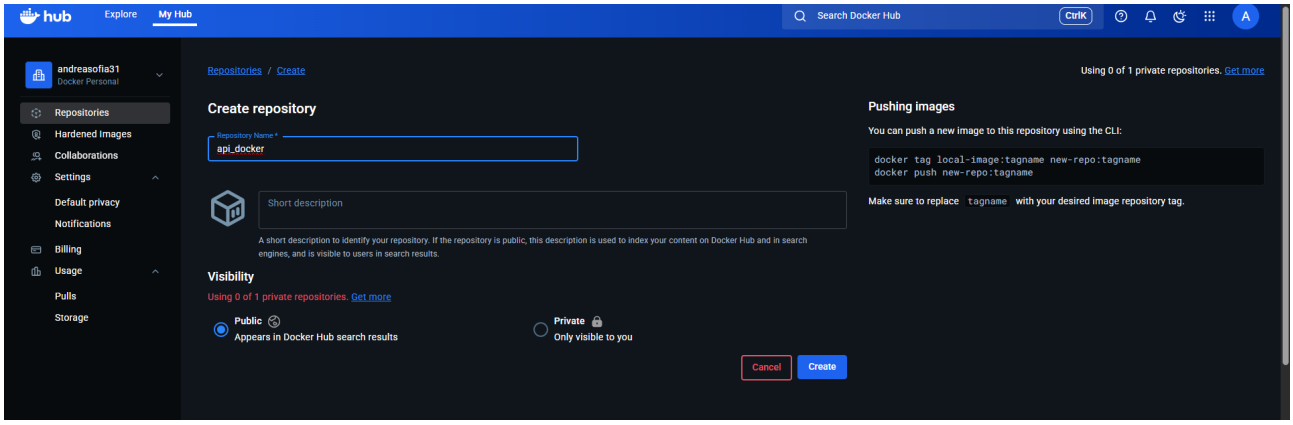


Ahora al acceder a docker, en images tendremos lo siguiente:



4 Subir la imagen a DockerHub

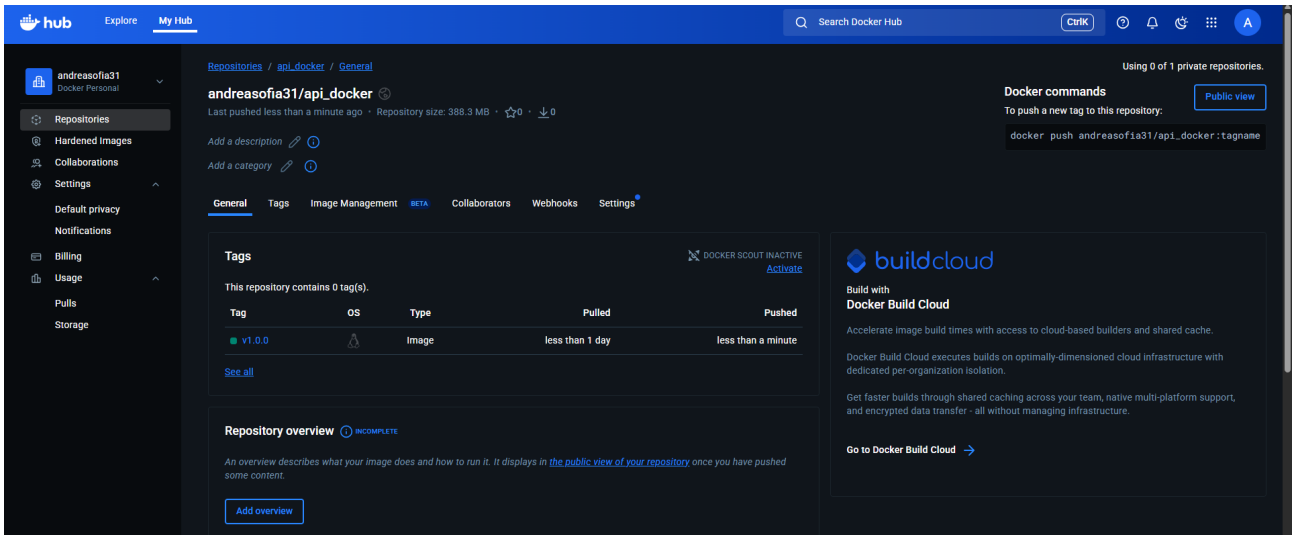
Para subir la imagen a DockerHub, lo primero es tener una cuenta e iniciar sesión. Tras iniciar sesión, creamos un repositorio con el nombre que queramos.



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:



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 repositiorio al pushear cada cambio. Y va a tener la siguiente información:

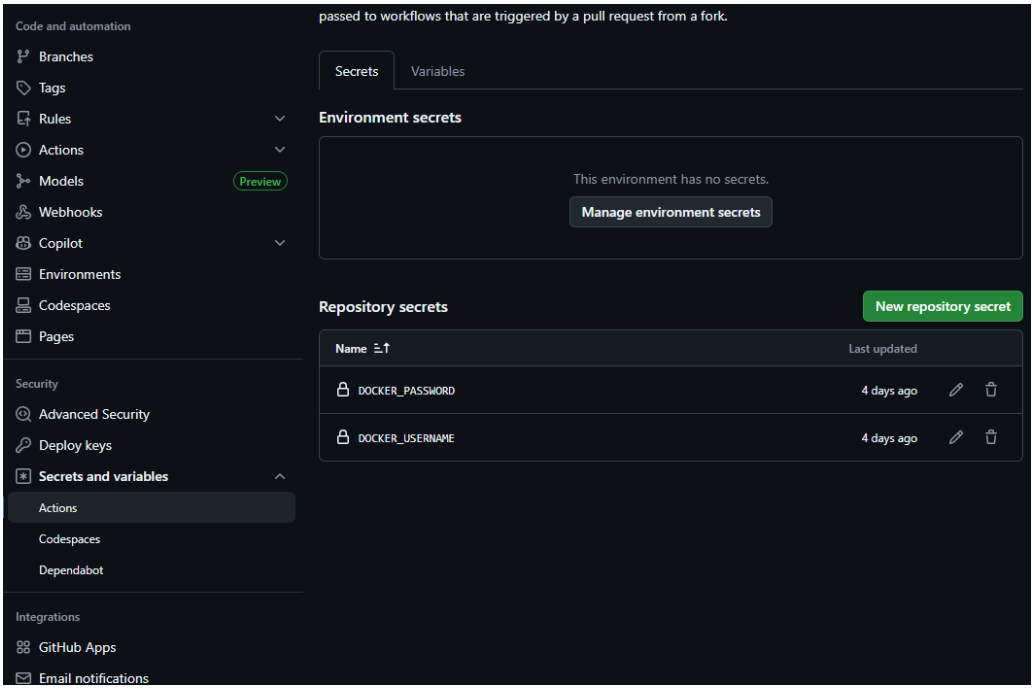
- Push Automático que afecta a la rama u al cualquier cambio en el repositorio.
- Variables de entorno que se almacenan en GitHub Secrets, que necesitaremos "DOCKER USERNAME" y "DOCKER PASSWORD".
- Luego todos los trabajos que va a tener que realizar ejecutando e un servidor Ubuntu.

```yaml
name: Build and Push Docker Image

on:
  push:
    branches:
      - main
      - master
    paths:
      - './**'
      - '.github/workflows/docker-push.yml'
  workflow_dispatch: # Permite ejecutar manualmente

env:
  DOCKER_IMAGE_NAME: ${{ secrets.DOCKER_USERNAME }}/api_docker
  DOCKER_TAG: latest

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout codigo
        uses: actions/checkout@v4

      - name: Configurar Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login a Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Construir y subir imagen Docker
        uses: docker/build-push-action@v5
        with:
          context: ./
          file: ./Dockerfile
          push: true
          tags: ${{ env.DOCKER_IMAGE_NAME }}:${{ env.DOCKER_TAG }}
```

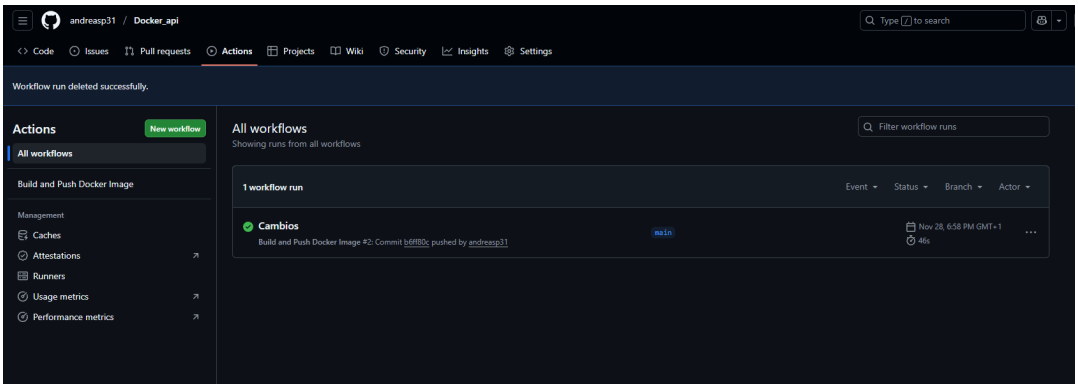
```
41          cache-from: type=registry,ref=${{ env.DOCKER_IMAGE_NAME }}:buildcache
42          cache-to: type=inline
43
44      - name: Mostrar informacion de la imagen
45        run: |
46          echo "Imagen construida y subida exitosamente:"
47          echo "  - Imagen: ${{ env.DOCKER_IMAGE_NAME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

5.1 Requisitos previos

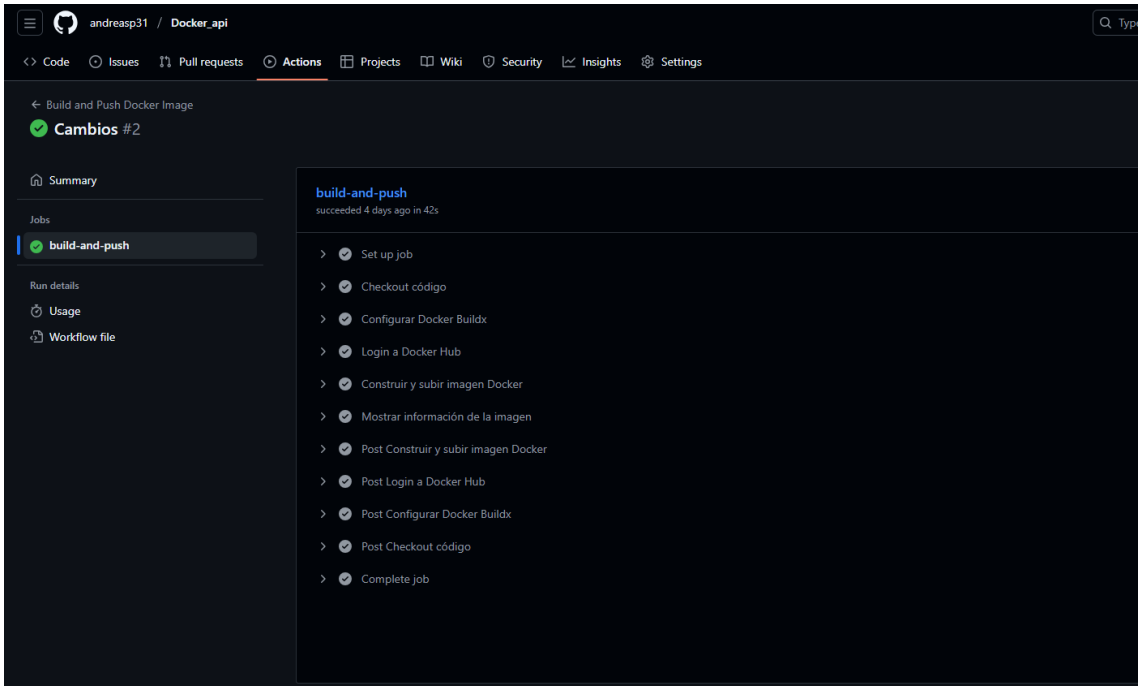
Acceder al repositorio de GitHub donde tengamos el poryecto subido. Dirigirnos configuraciones de repositiorio y buscar el apartado de "Secrets and variables" ahí añadimos el nombre de las variables de entorno con sus declaraciones. En este caso, para usuario ponemos el nuestro de dockerHub y como contraseña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



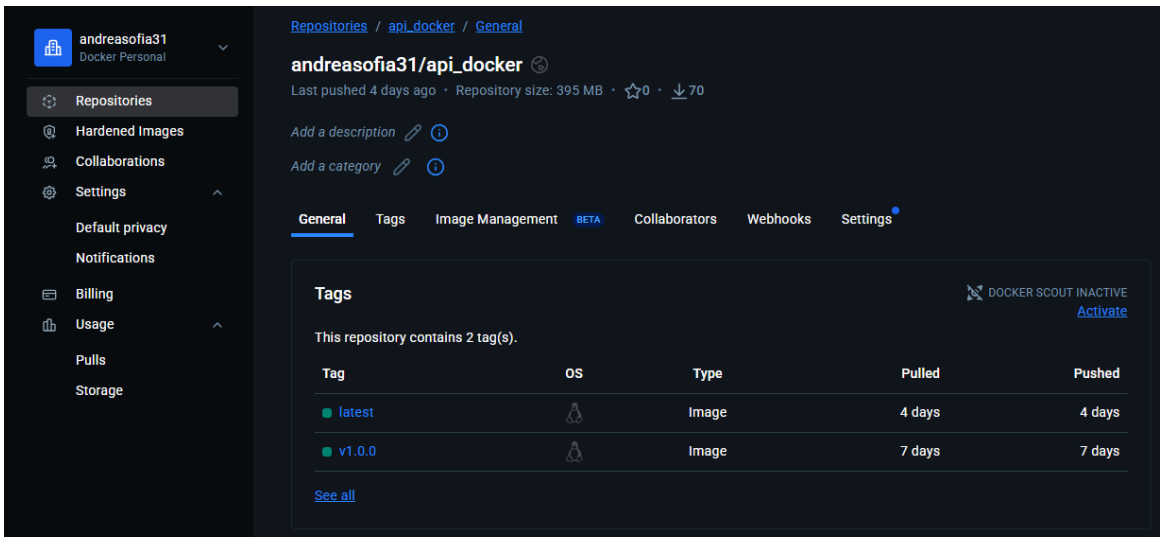
Si están correctamente creadas y el archivo está correcto, al pushear los nuevos cambios debería aparecer en el repositorio de GitHub un apartado llamado Actions. Y en el vemos que se ejecutó el workflow.



Y dentro vemos todos los servicios realizados con éxito.



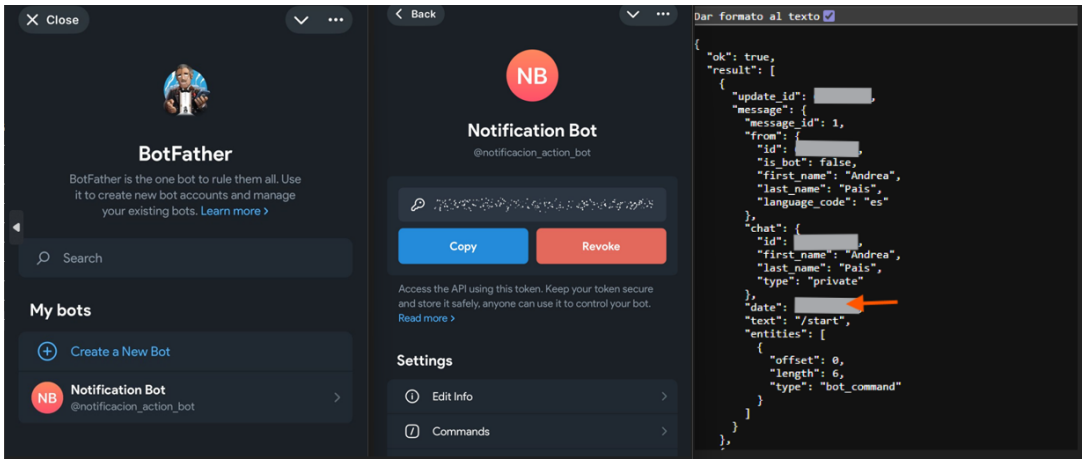
Por último para ver que todo está correcto, vamos a DockerHub y debería salir la imagen actualizada.



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 también el id del chat para mandar las notificación. Para ello le mandamos a nuestro bot cualquier mensaje. Y si nos dirigimos a hrefhttps://api.telegram.org/bot<miToken>/getUpdates nos sale un JSON y obtenemos el id de ahí.



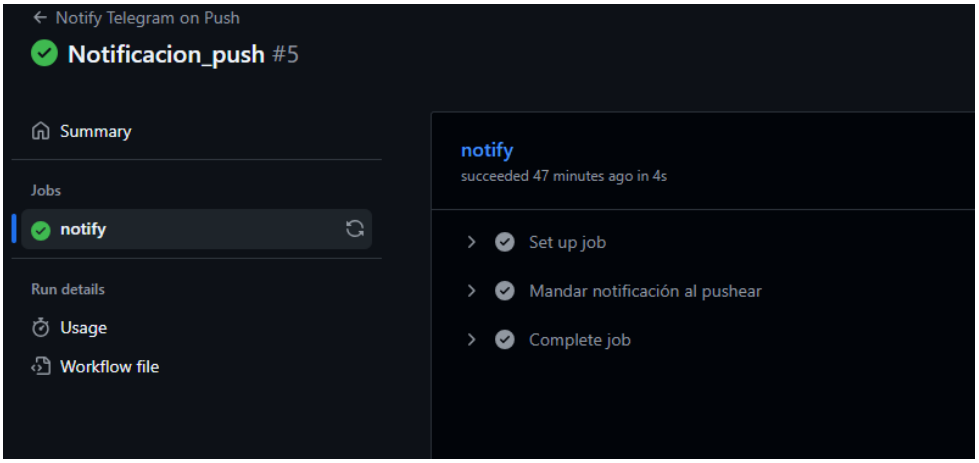
- Ahora en los ajustes del repositorio de GitHub creamos nuevas variables de entorno, el id y el token que es mediante lo que va a acceder el bot de telegram.

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

```
name: Notify Telegram on Push

on:
  push:
    branches:
      - main
      - master

jobs:
  notify:
    runs-on: ubuntu-latest
    steps:
      - name: Mandar notificacin al pushear
        run: |
          curl -s -X POST https://api.telegram.org/bot${{secrets.TELEGRAM_TOKEN}}
          /sendMessage\
          -d chat_id=${{ secrets.TELEGRAM_CHAT_ID }} \
          -d text=" Aqui va todo lo que nos interesa mostrar
          (no lo puse porque no me deja latex ponerlo comentado) \
```

```
20          -d parse_mode=HTML
```

- 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.