

Documentación de la api Docker

Andrea Sofía Pais Dos Santos

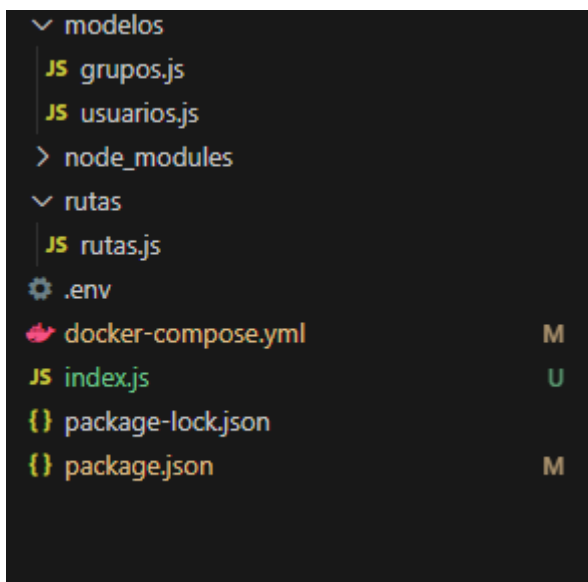
December 4, 2025

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
7 dotenv.config();
8 const app = express();
9 app.use(cors());
10 app.use(express.json());
11
12 app.use(rutas);
13 const uri = process.env.MONGO_URI;
14 export async function conectarBD(){
15     try{
16         await mongoose.connect(uri,{});
17         console.log("Base de datos conectada");
18     }catch(error){
19         console.log("Error conectandose a la base de datos", error);
20     }
21 }
22
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);
```

```

19  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ Archivo usuarios \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
20
21  import mongoose from "mongoose";
22
23  const UsuarioEschema = new mongoose.Schema({
24      nombre_user: {
25          type: String,
26          required: true
27      },
28      apellido_user: {
29          type: String,
30          required: true
31      },
32      edad_user: {
33          type: Number,
34          required: true
35      },
36      dni_user : {
37          type: String,
38          required: true
39      }
40  });
41
42  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.

```

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).

```

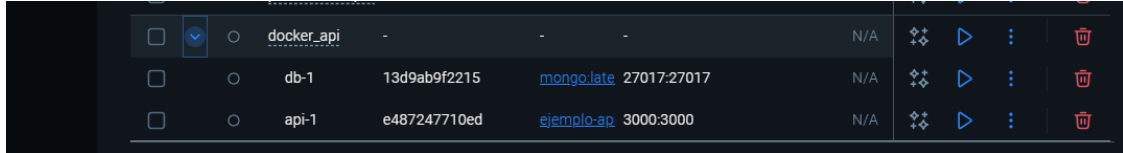
1  version: '3.8'
2
3  services:
4      api:
5          image: ejemplo-api:v1.0.0
6          ports:
7              - "3000:3000"
8          environment:
9              - MONGO_URI=mongodb://nombre:contrasena@db:27017/admin
10         depends_on:
11             - db
12         db:
13             image: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



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 necesitamos"
```

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"]
```

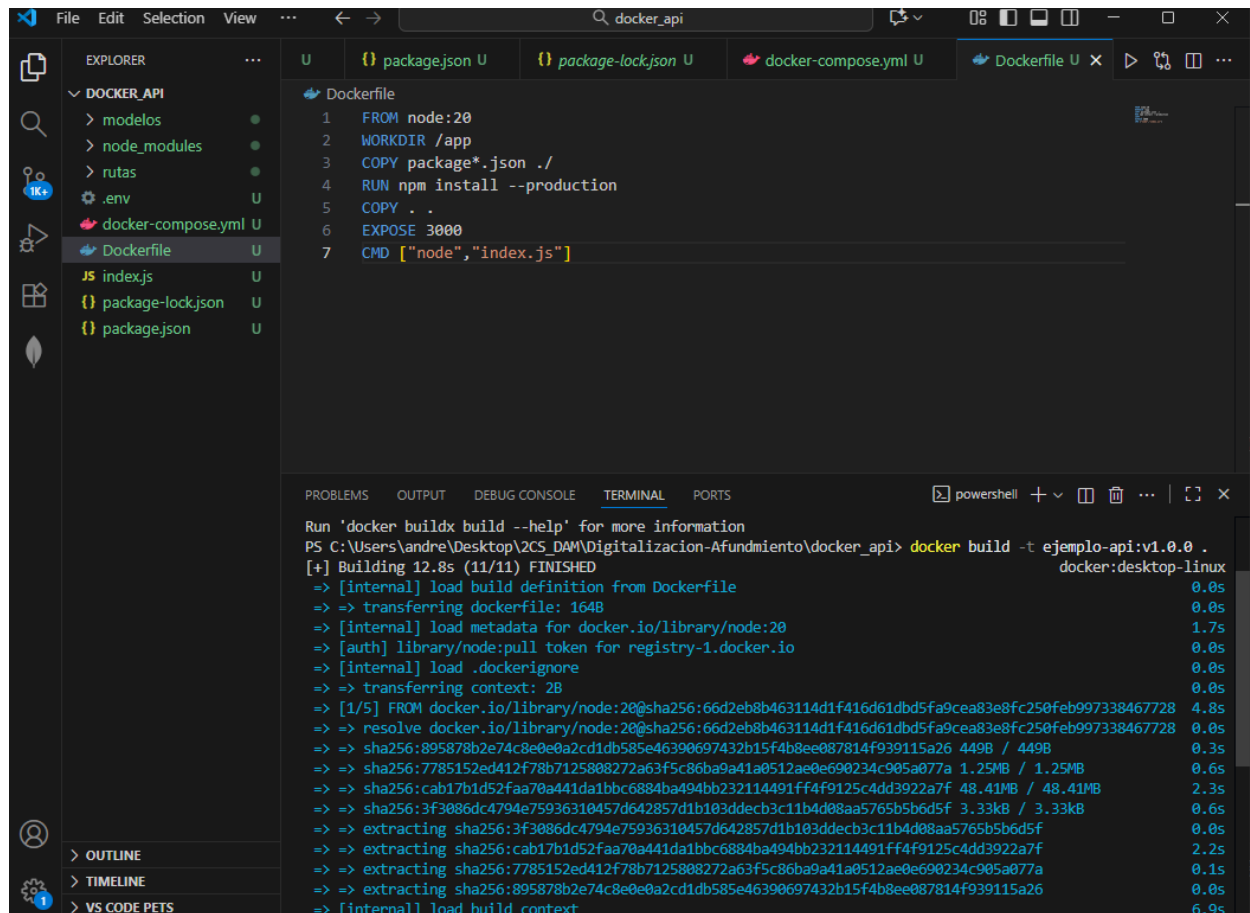
Lo que nos quiere ceder esto es que desde node vamos a crear "app" que es donde vamos a trabajar desde ahora, se copiará el package*.json, va a ejecutar esa instalación copiando todo en el puerto 300.

- 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 .
```



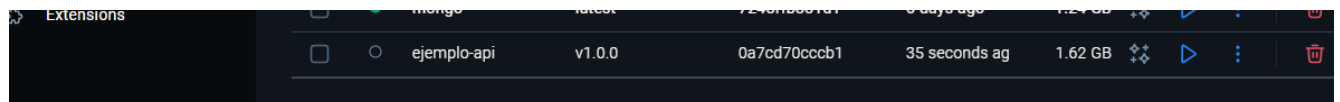
The screenshot shows the Visual Studio Code interface with a project named 'DOCKER_API'. The Explorer sidebar on the left lists files: 'modelos', 'node_modules', 'rutas', '.env', 'docker-compose.yml', 'Dockerfile', 'index.js', 'package-lock.json', and 'package.json'. The 'Dockerfile' is selected and open in the editor. It contains the following instructions:

```
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"]
```

Below the editor, the 'TERMINAL' tab is active, showing the command 'docker build -t ejemplo-api:v1.0.0 .' and its output. The output indicates a successful build of a Docker image named 'ejemplo-api:v1.0.0' based on the 'Dockerfile'.

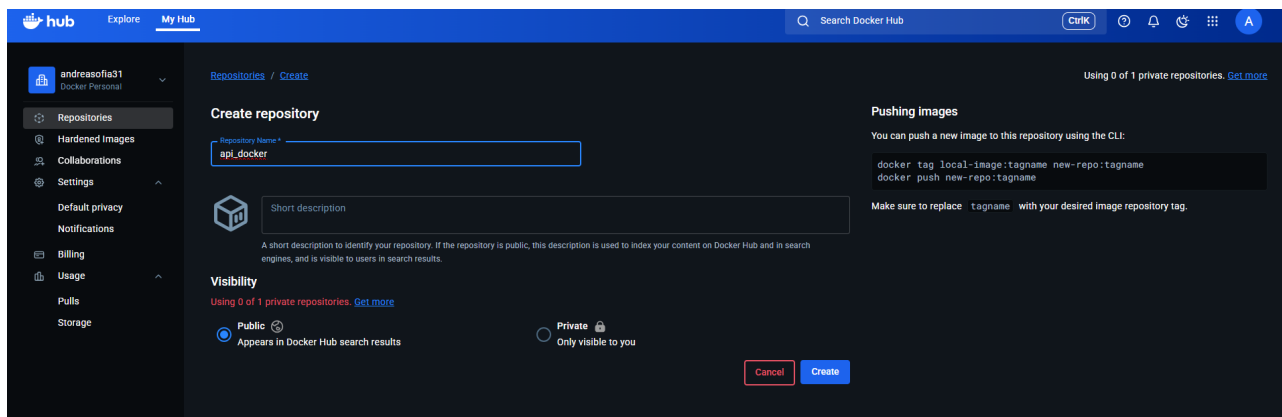
```
Run 'docker buildx build --help' for more information
PS C:\Users\andre\Desktop\2CS_DAM\Digitalizacion-Afundmientto\docker_api> docker build -t ejemplo-api:v1.0.0 .
[+] Building 12.8s (11/11) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 164B                                              0.0s
=> [internal] load metadata for docker.io/library/node:20                       1.7s
=> [auth] library/node:pull token for registry-1.docker.io                     0.0s
=> [internal] load .dockerignore                                                 0.0s
=> => transferring context: 2B                                                  0.0s
=> [1/5] FROM docker.io/library/node:20@sha256:66d2eb8b463114d1f416d61dbd5fa9cea83e8fc250feb997338467728 4.8s
=> => resolve docker.io/library/node:20@sha256:66d2eb8b463114d1f416d61dbd5fa9cea83e8fc250feb997338467728 0.0s
=> => sha256:895878b2e74c8e0e0a2cd1db585e46390697432b15f4b8ee087814f939115a26 449B / 449B 0.3s
=> => sha256:7785152ed412f78b7125808272a63f5c86ba9a41a0512ae0e690234c905a077a 1.25MB / 1.25MB 0.6s
=> => sha256:cab17b1d52faa70a441da1bbc6884ba494bb232114491ff4f9125c4dd3922a7f 48.41MB / 48.41MB 2.3s
=> => sha256:3f3086dc4794e75936310457d642857d1b103dddec3c11b4d08aa5765b5b6d5f 3.33kB / 3.33kB 0.6s
=> => extracting sha256:3f3086dc4794e75936310457d642857d1b103dddec3c11b4d08aa5765b5b6d5f 0.0s
=> => extracting sha256:cab17b1d52faa70a441da1bbc6884ba494bb232114491ff4f9125c4dd3922a7f 2.2s
=> => extracting sha256:7785152ed412f78b7125808272a63f5c86ba9a41a0512ae0e690234c905a077a 0.1s
=> => extracting sha256:895878b2e74c8e0e0a2cd1db585e46390697432b15f4b8ee087814f939115a26 0.0s
=> [internal] load build context                                                6.9s
```

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



4 Subir la imagen a DockerHub

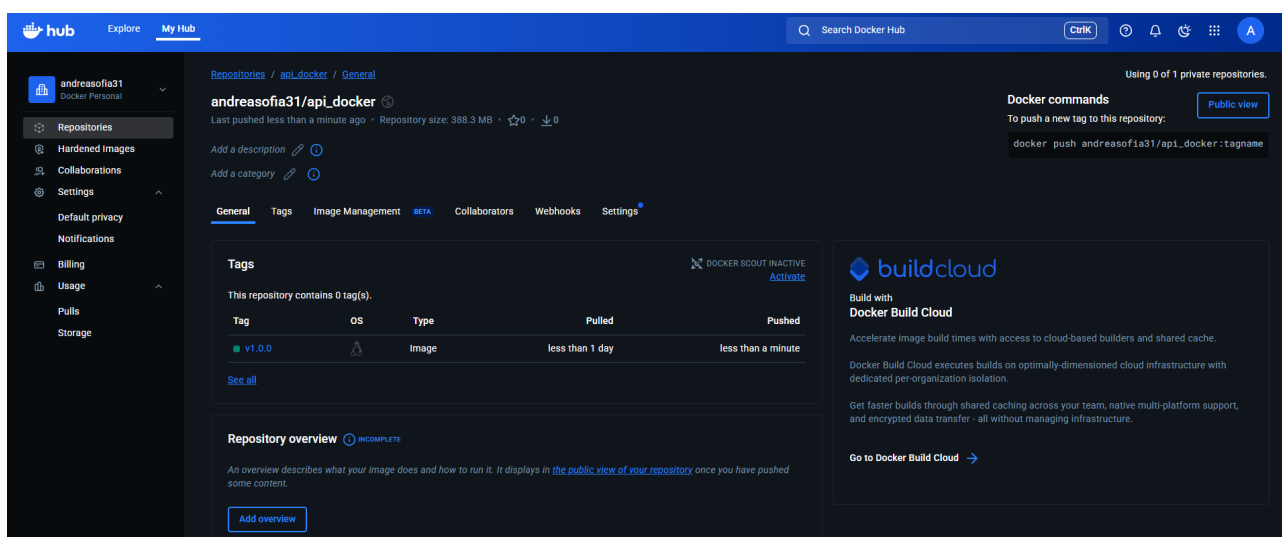
Para subir la imagen a DockerHub, lo primero es tener una cuenta e iniciar sesión. Tras iniciar sesión, creamos un repositorio con el nombre que queramos.



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 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 }
```

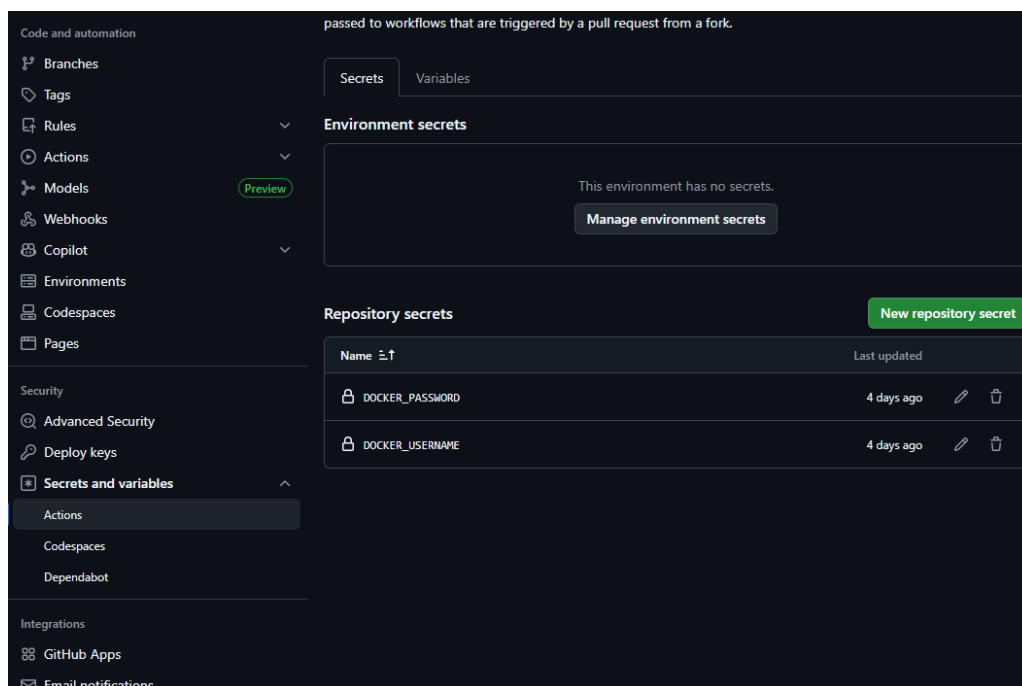
```

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"

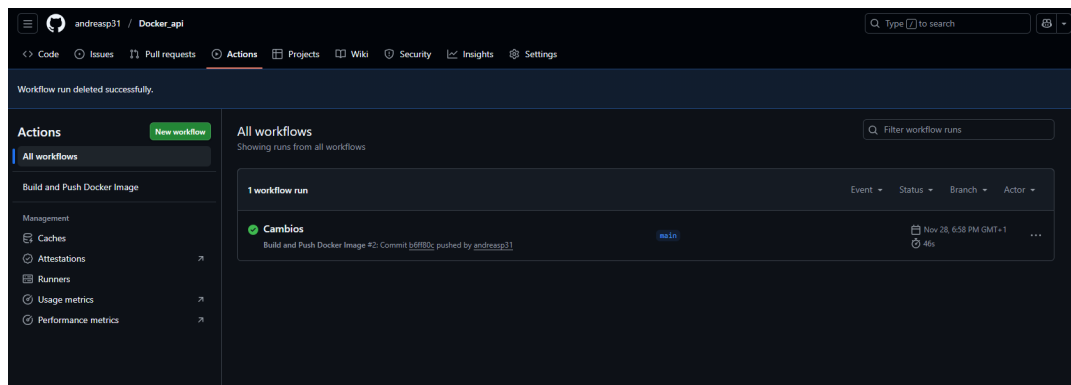
```

5.1 Requisitos previos

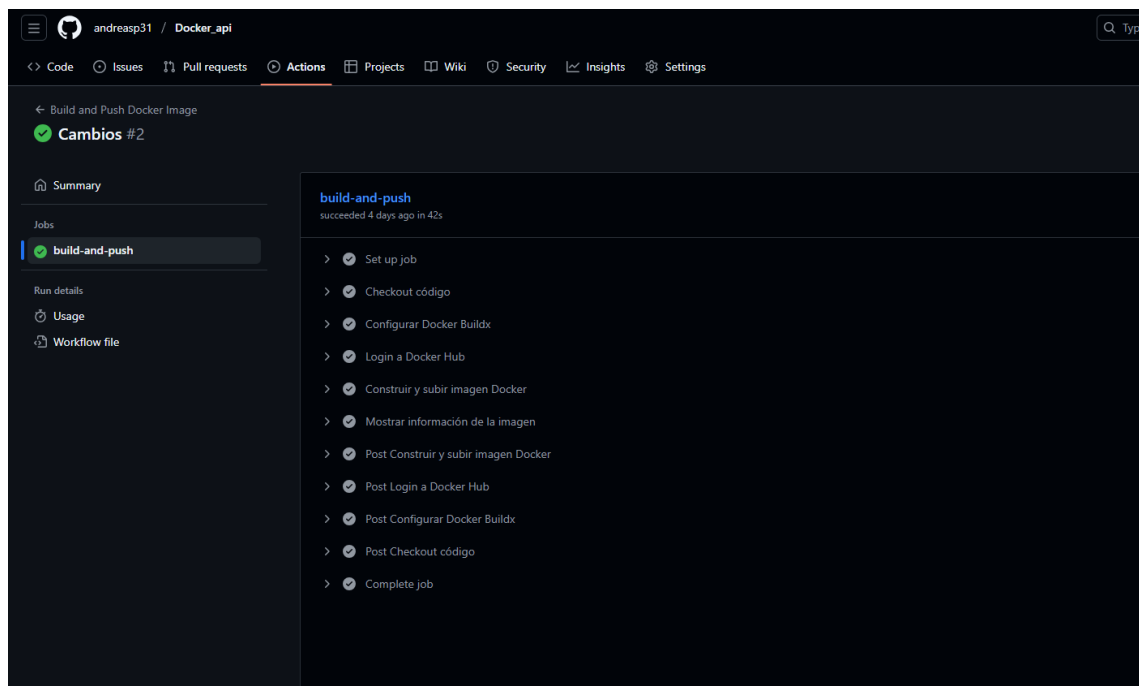
Acceder al repositorio de GitHub donde tengamos el proyecto subido. Dirigirnos configuraciones de repositorio y buscar el apartado de "Secrets and variables" ahí añadimos el nombre de las variables de entorno con sus declaraciones. En este caso, para usuario ponemos el nuestro de dockerHub y como contraseña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



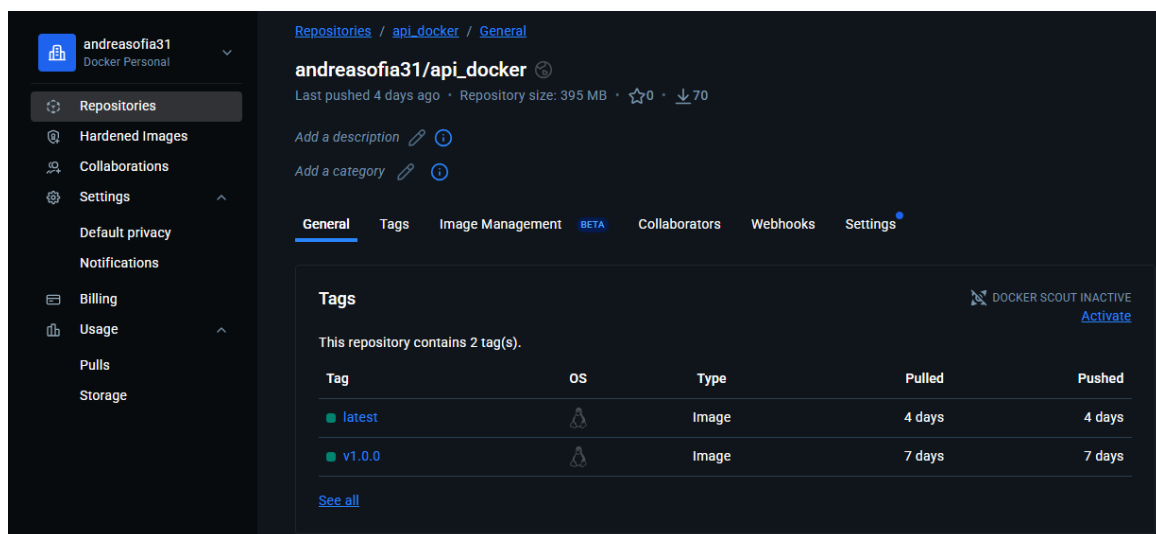
Si están correctamente creadas y el archivo está correcto, al pushear los nuevos cambios debería aparecer en el repositorio de GitHub un apartado llamado Actions. Y en el vemos que se ejecutó el workflow.



Y dentro vemos todos los servicios realizados con éxito.



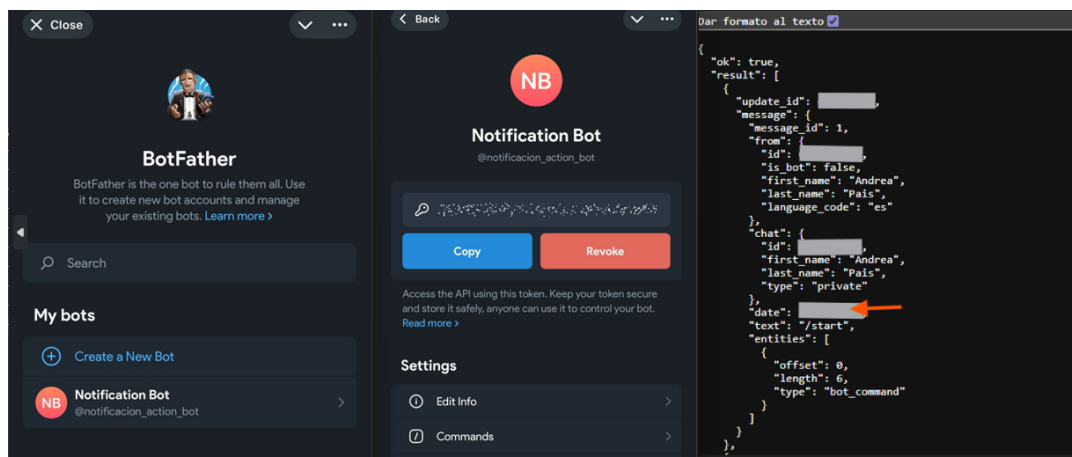
Por último para ver que todo está correcto, vamos a DockerHub y debería salir la imagen actualizada.



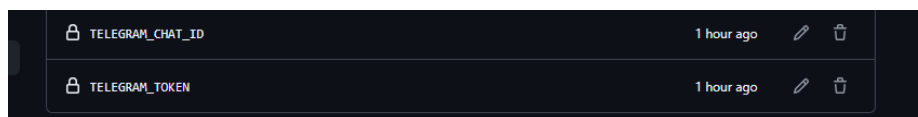
6 Notificación Action

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.



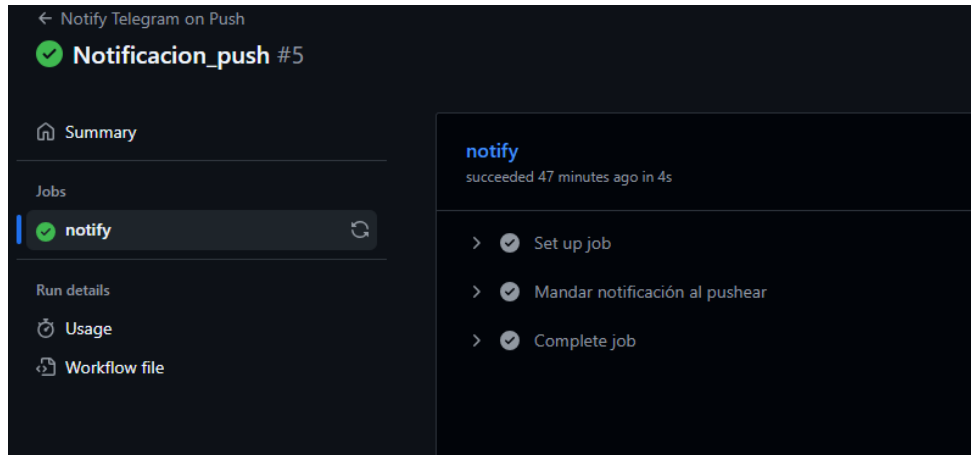
- Ahora en los ajustes del repositorio de GitHub creamos nuevas variables de entorno, el id y el token que es mediante lo que va a acceder el bot de telegram.



- 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 notificacin 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=" Aqui 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.

