DOCUMENTACIÓN DE LA API DOCKERIZADA
Gestión de Usuarios y Grupos

Eder Martínez Castro

5 de diciembre de 2025

Índice general

1. Introducción

En este documento se detalla la API para la **gestión de usuario y grupos de un colegio/centro**, esta fue desarrollada con **Node.js**, **Express** y **MariaDB**.
La API nos permite poder `listar alumnos/usuarios y clases/grupos`, `crear un nuevo usuario/alumno y clase/grupo`, `actualizar la nota de un usuario/alumno` y `borrar un usuario/alumno`.

Este proyecto está **Dockerizado**, utilizando **Docker** y **Docker Compose** para levantar nuestra base de datos **MariaDB** como nuestra API en Express.js.

Puedes ver el código completo de la API en este repositorio: API-Users-Groups.

2. Arquitectura

La API sigue una arquitectura simple de Express accediendo a la base de datos de **MariaDB** usando el cliente **msql2/promise**. Se adopta el patrón de arquitectura **cliente-servidor**, donde el cliente realiza peticiones HTTP al servidor, y este le responde con los datos o mensajes de estado correspondientes.

Nuestro servidor **Node** se conecta a la base de datos en el arranque, y en caso que no existan las tablas `users` y `groups`, se crearán automáticamente usando la funcion `createTablesIfNotExists()`, así dejamos la API lista para que se pueda conectar. Toda la lógica la tenemos en un único archivo `server.js`, lo cúal simplifica la estructura del prouecto al ser una API con solo 5 endpoints.

Figura 1: Patrón de arquitectura cliente-servidor

2.1. Clases

En este caso como bien hemos comentado anteriormente al ser una API muy simple con 5 endpoints no hizo falta crear ninguna clase.

2.2. Patrones de diseño

Ahora vamos a explicar el **patrón de diseño empleado**, que nos permiten estructurar el flujo de ejecución y como interactan. Usar los patrones nos ayuda a mantener el **código modular**, **legible** y **ampliarlo facilmente** en un futuro.

- Patrón Mediator/Middleware

Express ya implementa el **patrón Mediator/Middleware**, este patrón nos permite poder encadenar funciones intermedias entre la solicitud y respuesta, facilitando la **extensión del comportamiento del servidor**, haciendo que el flujo de ejecución sea modular y reutilizable.

En este proyecto se usan:

- `express.json()`: parsea a JSON el "body"de las peticiones.
- `cors()`: permite solicitudes solo desde orígenes conocidos.

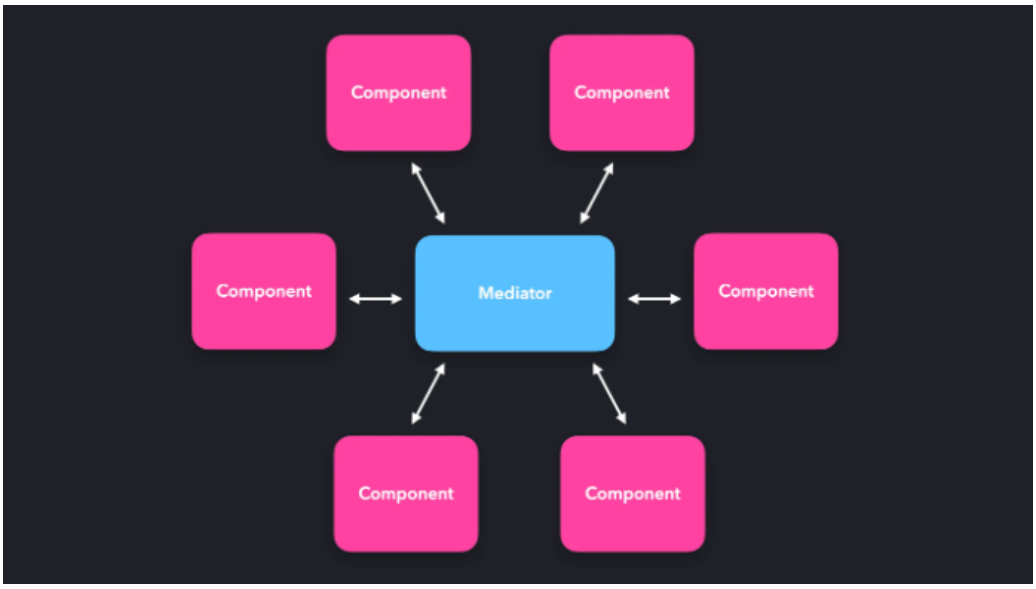

Figura 2: Patrón Mediator/Middleware

2.3. Stack empleado

Para el desarrollo de la API, se utiliza los siguietes tecnológias:

- Utilizamos `Node.js` como **entorno de ejecución** principal.
- Utilizamos `Express.js` para crear la API en JavaScript.
- Utilizamos `MariaDB` como nuestra **base de datos relacional**, levantado en un contenedor Docker.
- Utilizamos `Docker` y `Docker Compose` para orquestar tanto la base de datos como la API.

3. Dependencias utilizadas

Para nuestra API hemos utilizado principalmente **4 dependencias**:

- Utilizamos `express` como **servidor HTTP** para definir rutas y manejar peticiones.
- Utilizamos `mysql2/promise` para conectarnos a la base de datos **MariaDB** usando promesas.
- Utilizamos `cors` para **configurar el CORS** y solo permitir las solicitudes desde orígenes conocidos que definimos.
- Utilizamos `dotenv` para la poder **cargar las variables de entorno** protegidas desde el archivo `.env`.

4. Modelo de datos

En nuestra API tenemos estás dos tablas principales en **MariaDB**:

- **Tabla de grupos** (`TABLE_GROUPS`):
  - `id`: Identificador de nuestro grupo.
  - `group_name`: Nombre del grupo.
- **Tabla de usuarios** (`TABLE_USERS`):
  - `id`: Identificador de nuestro usuario.
  - `name`: Nombre del usuario.
  - `surname`: Apellidos del usuario.
  - `marks`: Nota del usuario.
  - `group_id`: Identificador del grupo al que pertenece el usuario.

Como comentamos anteriormente si no tenemos estas tablas al iniciar nuestra API las creamos con la función `createTablesIfNotExist()`

5. Endpoints

A continuación explicaremos cada uno de los endpoints de nuestra API:

- **GET** `/` - Comprobar que la API funciona.
- **GET** `/api/users-groups` - Obtener todos los usuarios y grupos.
- **POST** `/api/user` - Crear un nuevo usuario.
- **POST** `/api/group` - Crear un nuevo grupo.
- **PUT** `/api/user/:id/marks` - Actualizar la nota de un usuario.
- **DELETE** `/api/user/:id` - Eliminar un usuario.

GET / - Comprobar el estado de la API

Este endpoint nos sirve para verificar rápidamente que la API está desplegada y respondiendo.

Respuesta

`200 - Successful Response`

```
API FUNCIONANDO
```

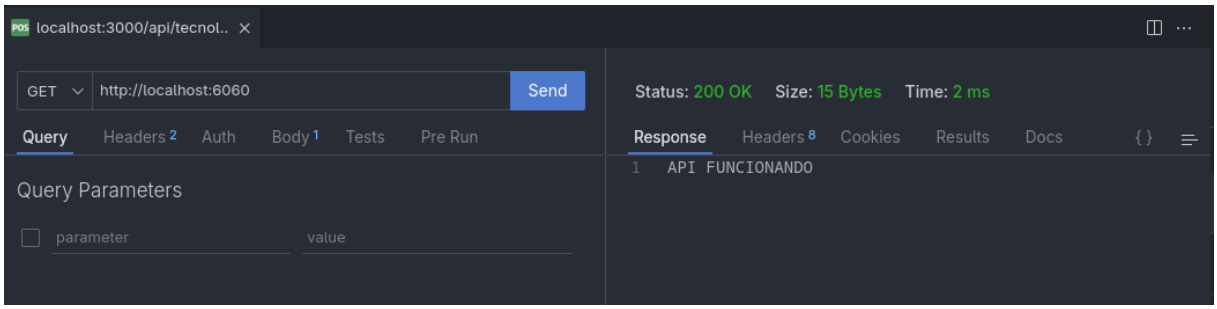

Figura 3: Captura de comprobación del funcionamiento del endpoint usando **Thunder Client**: **GET /**

GET /api/users-groups - Listar usuarios y grupos

Este endpoint nos recupera toda la información de la table usuarios y grupos que tenemos en la base de datos.

Ejemplo de uso

- **GET** - `/api/users-groups`

Respuesta

`200 - Successful Response`

```
{
  "users": [
    {
      "id": 1,
      "name": "Ivan",
      "surname": "Priego",
      "marks": 8,
      "group_id": 1
    }
  ],
  "groups": [
    {
      "id": 1,
      "group_name": "1A"
    }
  ]
}
```

Errores

500 - Internal Server Error

```
{"message": "Error en el servidor", "error": "<detalle>"}
```

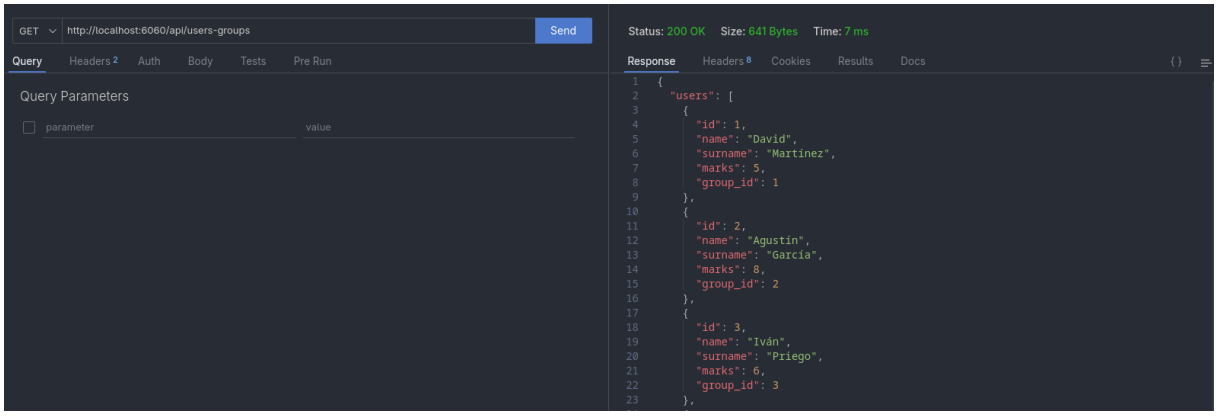

Figura 4: Captura de comprobación del funcionamiento del endpoint usando **Thunder Client**: **GET** `/api/users-groups`

POST `/api/user` - Crear usuario

Este endpoint nos permite crear un nuevo usuario en la base de datos. Recibe los siguientes parámetros `name`, `surname`, `marks` y `groupId`.

Body (JSON)

```
{
  "name": "Ivan",
  "surname": "Priego",
  "marks": 9,
  "groupId": 1
}
```

Respuestas

201 - Created

```
{
  "message": "Usuario creado correctamente",
  "user": {
    "id": 3,
    "name": "Ivan",
    "surname": "Priego",
    "marks": 9,
    "group_id": 1
  }
}
```

Errores

400 - Bad Request

```
{"message": "Falta alguno de los 4 campos requeridos"}
```

400 - Bad Request

```
{"message": "La notado del usario tiene que ser un entero."}
```

500 - Internal Server Error

```
{"message": "Error en el servidor", "error": "<detalle>"}
```

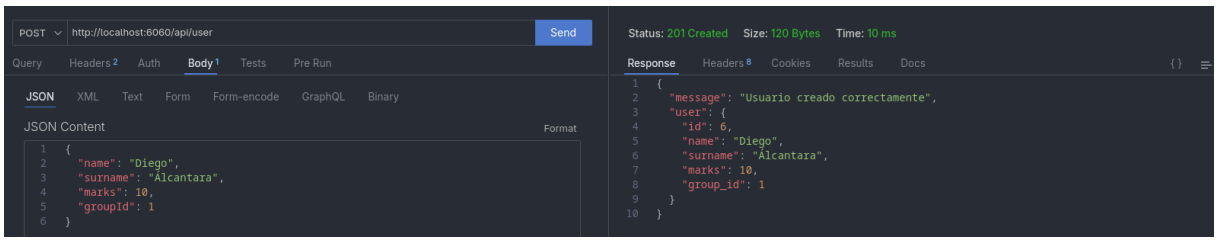

Figura 5: Captura de comprobación del funcionamiento del endpoint usando **Thunder Client**: **POST** `/api/user`

POST `/api/group` - Crear grupo

Este endpoint nos permite crear un nuevo grupo en la base de datos. Recibe el siguiente parámetro `groupName`.

Body (JSON)

```
{
  "groupName": "1A"
}
```

Respuestas

201 - Created

```
{
  "message": "Grupo creado correctamente",
  "group": {
    "id": 1,
    "group_name": "1A"
  }
}
```

Errores

400 - Bad Request

```
{"message": "Falta el campo requerido"}
```

500 - Internal Server Error

```
{"message": "Error en el servidor", "error": "<detalle>"}
```

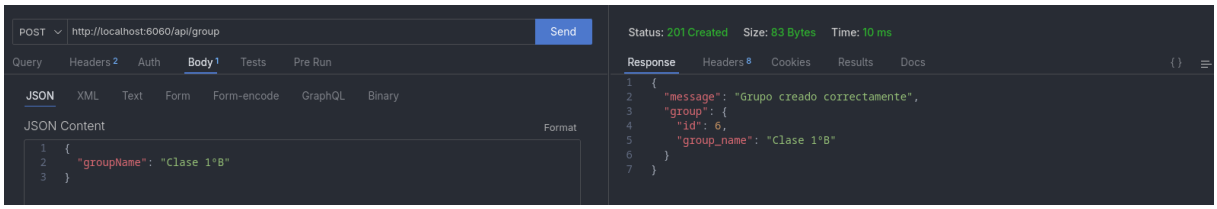

Figura 6: Captura de comprobación del funcionamiento del endpoint usando **Thunder Client**: **POST** `/api/group`

PUT `/api/user/:id/marks` - Actualizar la nota de un usuario

Este endpoint nos permite poder actualizar la nota (`marks`) de un usuario mediante su `id`.

Parámetros de ruta

- `id`: id del usuario.

Body (JSON)

```
{
  "marks": 7
}
```

Respuestas

200 - Successful Response

```
{
  "message": "La nota del usario cambio a: 7",
  "user": { "id": 3 }
}
```

Errores

400 - Bad Request

```
{"message": "Falta el id para poder cambiar la nota"}
```

400 - Bad Request

```
{"message": "La notado del usario tiene que ser un entero."}
```

404 - Not Found

```
{"message": "Usuario no encontrado en la base de datos"}
```

500 - Internal Server Error

```
{"message": "Error en el servidor", "error": "<detalle>"}
```

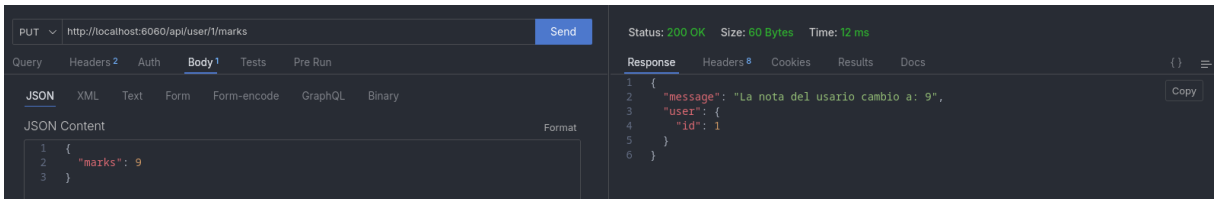

Figura 7: Captura de comprobación del funcionamiento del endpoint usando **Thunder Client**: **PUT** `/api/user/:id/marks`

DELETE `/api/user/:id` - Eliminar usuario

Este endpoint nos permite poder eliminar un usuario de la base de datos a partir de su `id`.

Parámetros de ruta

- `id`: identificador del usuario.

Respuestas

200 - Successful Response

```
{
  "message": "Usuario eliminada correctamente",
  "user": { "id": 3 }
}
```

Errores

400 - Bad Request

```
{"message": "Falta el id para poder eliminar"}
```

404 - Not Found

```
{"message": "Usuario no encontrado en la base de datos"}
```

500 - Internal Server Error

```
{"message": "Error en el servidor", "error": "<detalle>"}
```

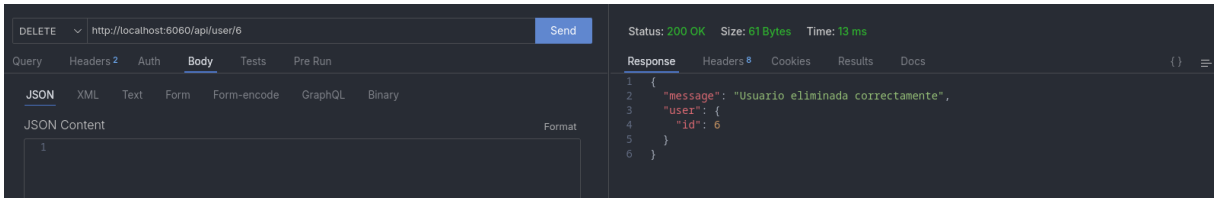

Figura 8: Captura de comprobación del funcionamiento del endpoint usando **Thunder Client**: **DELETE** `/api/user/:id`

6. Despliegue dockerizado

La API se desplegará mediante **Docker** y **Docker Compose**, levantando la base de datos **MariaDB** y la API de Express.

6.1. Variables de entorno

Todas las variables que necesariasestan declaradas en el archivo `.env`. Un ejemplo del contenido que necestariamos en el archivo sería:

Clave	**Valor (ejemplo)**
PORT	`6060`
FRONT_ORIGIN	`http://localhost:4200`
MYSQL_URI	`mysql://user:password@mariadb:3306/db`
TABLE_USERS	`users`
TABLE_GROUPS	`groups`
MYSQL_ROOT_PASSWORD	`rootpassword`
MYSQL_DATABASE	`db`
MYSQL_USER	`user`
MYSQL_PASSWORD	`password`

6.2. Dockerfile de la API

En el `Dockerfile` definimos la imagen para la API en Express.js:

```
# Imagen base
FROM node:20

# Directorio de trabajo en el contenedor
WORKDIR /api

# Copiamos el package.json y el package-lock.json
COPY package*.json ./

# Instalamos las dependencias necesarias
RUN npm install

# Copiar el resto de nuestra API
COPY . .

# Puerto donde estara nuestro contenedor
EXPOSE 6000

# Comando para ejecutar la API
CMD [ "node", "server.js" ]
```

6.3. docker-compose.yml

En el archivo `docker-compose.yml` definimos los servicios para la base de datos y la API:

```
services:
  # Servicio para MariaDB
  mariadb:
    image: mariadb:lts-ubi9
    container_name: mariadb_classroom
    ports:
      - "3307:3306"
    env_file:
      - .env # Cargar las variables del .env
    volumes:
      - ./dbdata:/var/lib/mysql:Z # Persistencia de la base de
         datos

  # Servicio para la API
  api:
    build: .
    container_name: api_classroom
    ports:
      - 6060:6060 # La API estara corriendo en localhost:6060
    env_file:
      - .env # Cargar las variables del .env
    depends_on:
      - mariadb
    restart: unless-stopped
```

6.4. Arranque del proyecto con Docker Compose

Una vez creado el archivo `.env`, el `Dockerfile` y el `docker-compose.yml`, ya podremos probarlo:

1. Construir las imágenes (si lo necesitamos):

   ```
   docker compose build
   ```

2. Levantar los servicios:

   ```
   docker compose up -d
   ```

3. Nuestra API estará corriendo en:

   ```
   http://localhost:6060
   ```

4. Para poder ver los logs de la API:

   ```
   docker compose logs -f api
   ```

6.5. Comprobación del despliegue

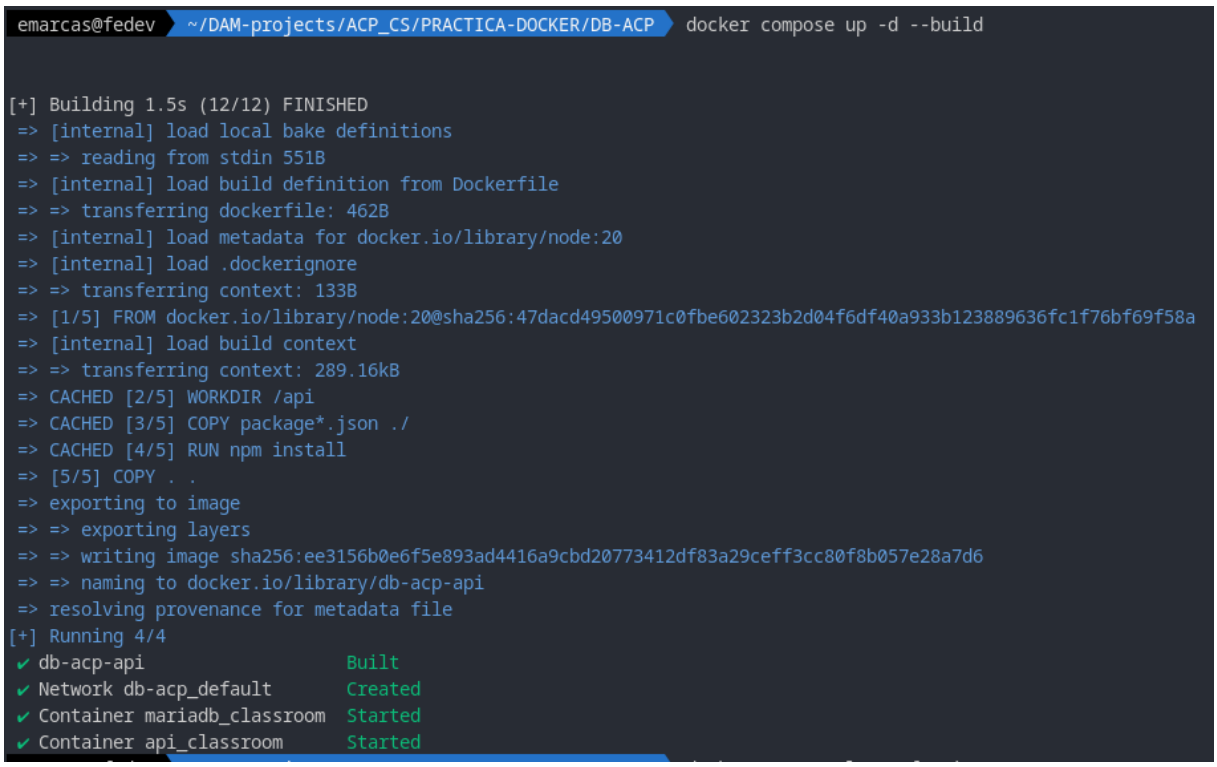

Figura 9: Desplegando y reconstruyendo los servicios mediante el comando **docker compose up -d –build**, a parte este comando no nos mostrará los logs gracias al `-d` para tener la terminal más limpia.

```
docker compose up -d --build
```

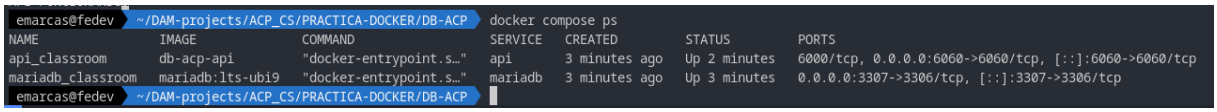

Figura 10: Verificamos el estado de nuestros contenedores tras el despliegue utilizando el comando **docker compose ps**

```
docker compose ps
```

6.6. Subir a Docker Hub

En las siguientes capturas enseñaremos los comandos para poder subir nuestro API a nuestra cuenta de **Docker hub**

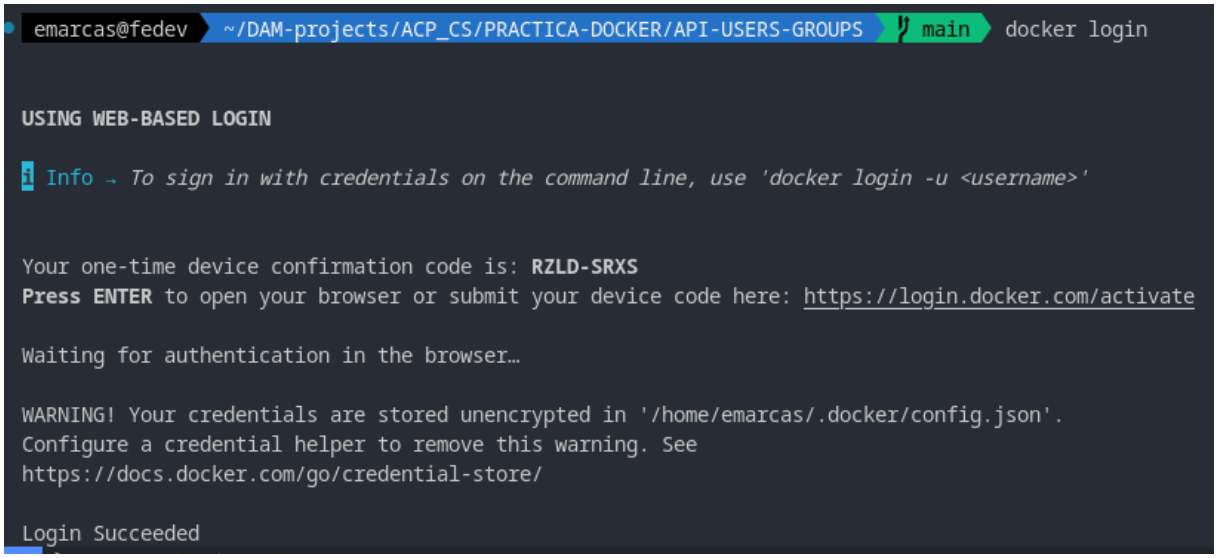

Figura 11: Con este comando podemos iniciar sesión en **Docker hub** desde la terminal utilizando el comando **docker login**

```
docker login
```

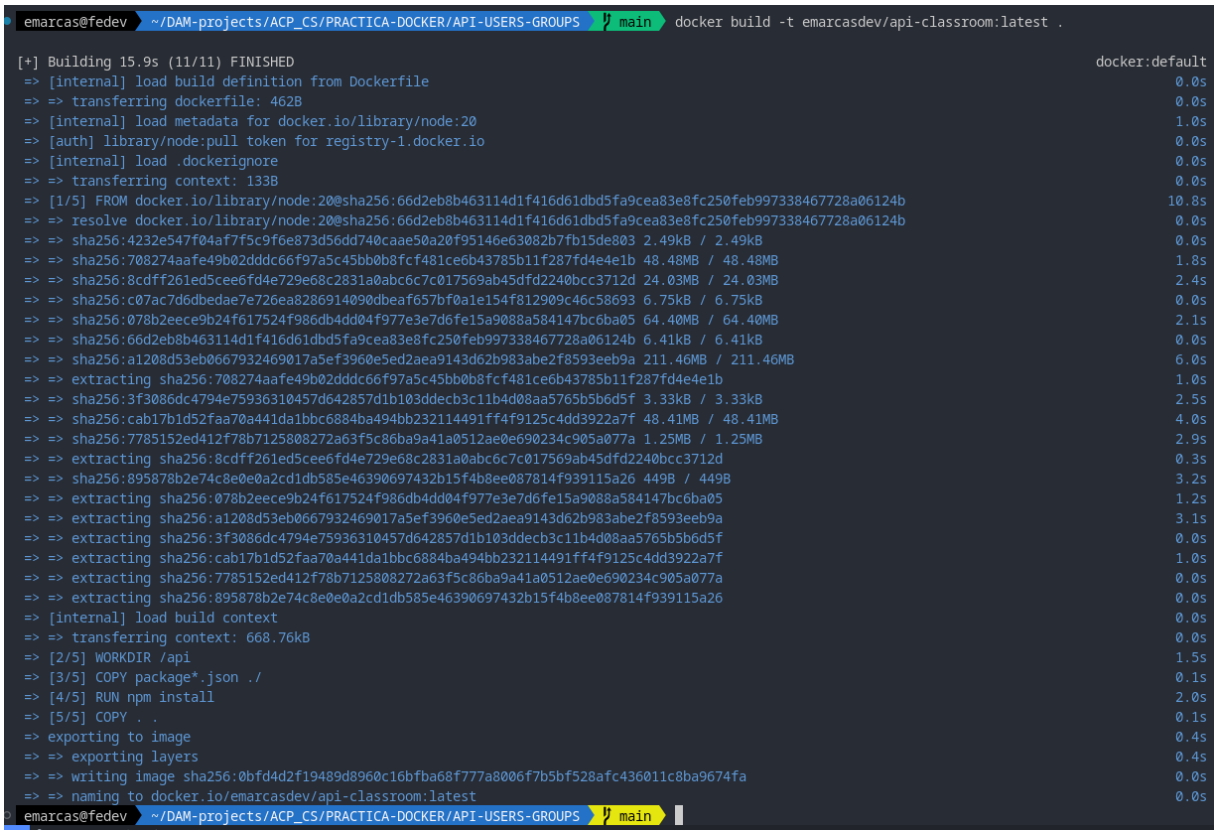

Figura 12: Con el siguiente comando costruimos la `imagen de docker` utilizando el comando **docker build -t <USERNAME>/<NAME>:latest .**

```
docker build -t <USERNAME>/<NAME>:latest .
```

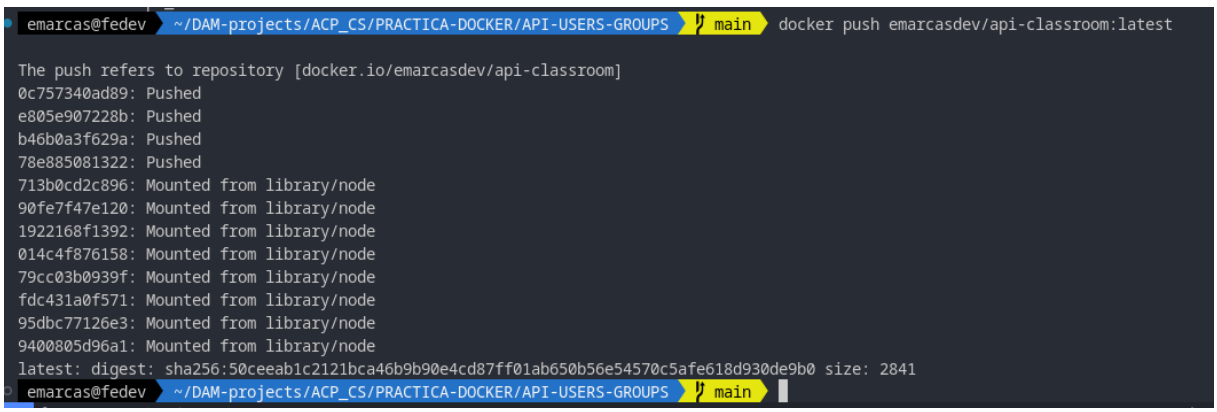

Figura 13: Con este último comando subimos la `imagen de docker` que contruimos con el paso anterior a nuestro repositorio de **Docker hub** utilizando el comando **docker push <USERNAME>/<NAME>:latest**

```
docker push <USERNAME>/<NAME>:latest
```

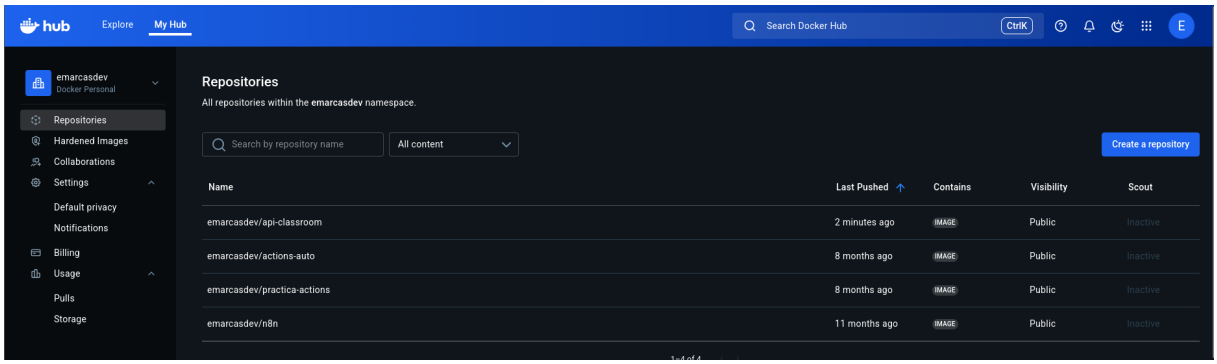

Figura 14: Verificamos que nuestra api que dockerizamos la tenemos subida a **Docker hub**, como podemos ver ya la tenemos subida perfectamente en nuestros repositorios.

6.7. Github Actions

En este apartado configuraremos un **workflow de GitHub Actions** que, que cada vez que hagamos un push a nuestra rama principal, construya y suba automáticamente la imagen de Docker Hub. Además, añadiremos la funcionalidad de enviar una notificación a un canal de Discord informando del resultado del despliegue. Lo primero será crear

la siguiente estructura de carpetas con el siguiente archivo, donde definieremos nuestra **GitHub Action**:

```
.github/
|___ workflows/
    |___ deploy.yml
```

Código del `deploy.yml`:

```yaml
name: Build and Push Docker Image

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

env:
  DOCKER_IMAGE_NAME: ${{ secrets.DOCKER_USERNAME }}/api-classroom
  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_TOKEN }}

      - 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
             }}
          cache-from: type=registry,ref=${{ env.DOCKER_IMAGE_NAME
              }}:buildcache
          cache-to: type=inline

      - name: Mostrar informacion de la imagen
        run: |
          echo "Imagen construida y subida exitosamente:"
          echo "  - Imagen: ${{ env.DOCKER_IMAGE_NAME }}:${{ env.
             DOCKER_TAG }}"
          echo "  - Docker Hub: https://hub.docker.com/r/${{
             secrets.DOCKER_USERNAME }}/api-classroom"
```

Agregar los secrets a GitHub

A continuación, necesitamos ir a nuestro repositorio de GitHub y dirigirnos al apartado de configuración. Dentro de la categoría **security** seleccionamos la opción **secrets and variables** y, entraremos en la sección de **actions**.

En está sección crearemos los **secrets** necesarios (nombre de usuario y el token de Docker Hub), que usaremos para realizar el inicio de sesión en Docker Hub desde el workflow, si no los configuramos la acción fallará.

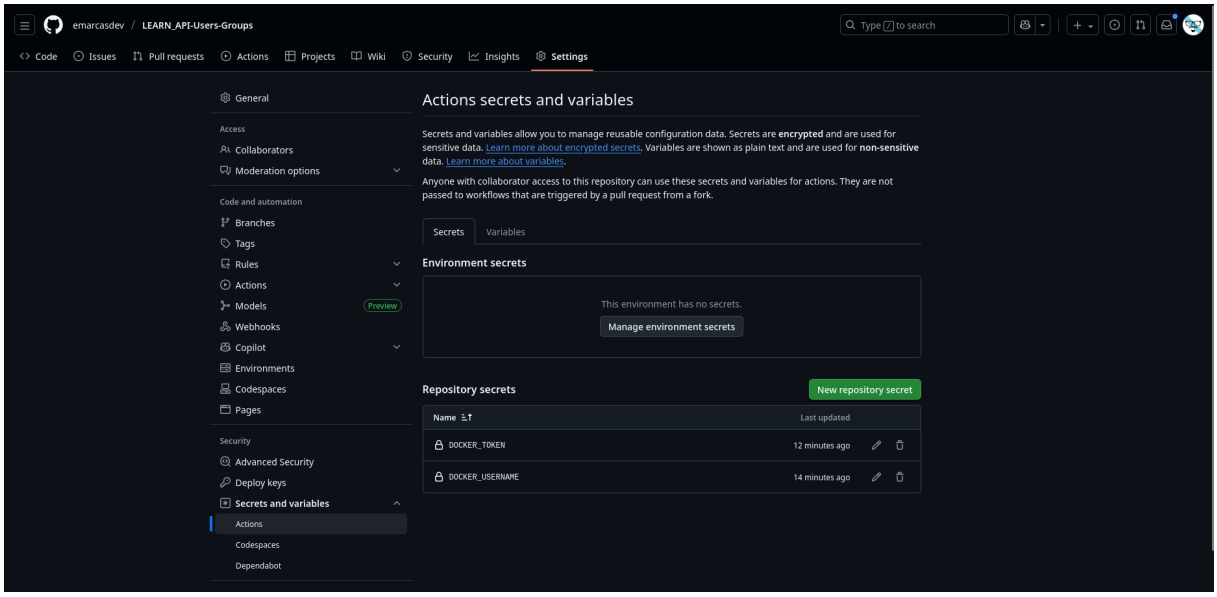

Figura 15: Captura de los **secrets** que tenemos en el repositorio, que son el nombre de usuario y el token de Docker Hub.

Comprobación de que la action

Para poder verificar el resultado en nuestro repositorio nos dirigiremos al apartado de **actions** donde podremos ver el historial de ejecuciones del workflow y comprobar si han pasados correctamente o han fallado.

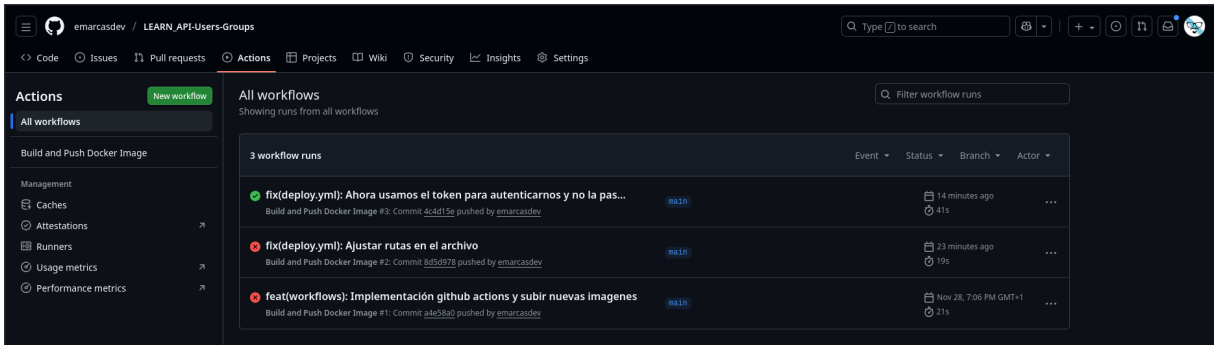

Figura 16: En esta captura podemos ver que tuvimos 3 workflows, los dos primeros nos fallarón, pero resolvimos los fallos de credenciales y rutas, mientras que la última se ejecuto correctamente tras corregir esos errores.

Funcionalidad de notificación en Discord

Ahora para poder implementar la funcionalidad de la notificación en Discord, primero realizaremos una serie de preparaciones en Discord para que nos notifique.

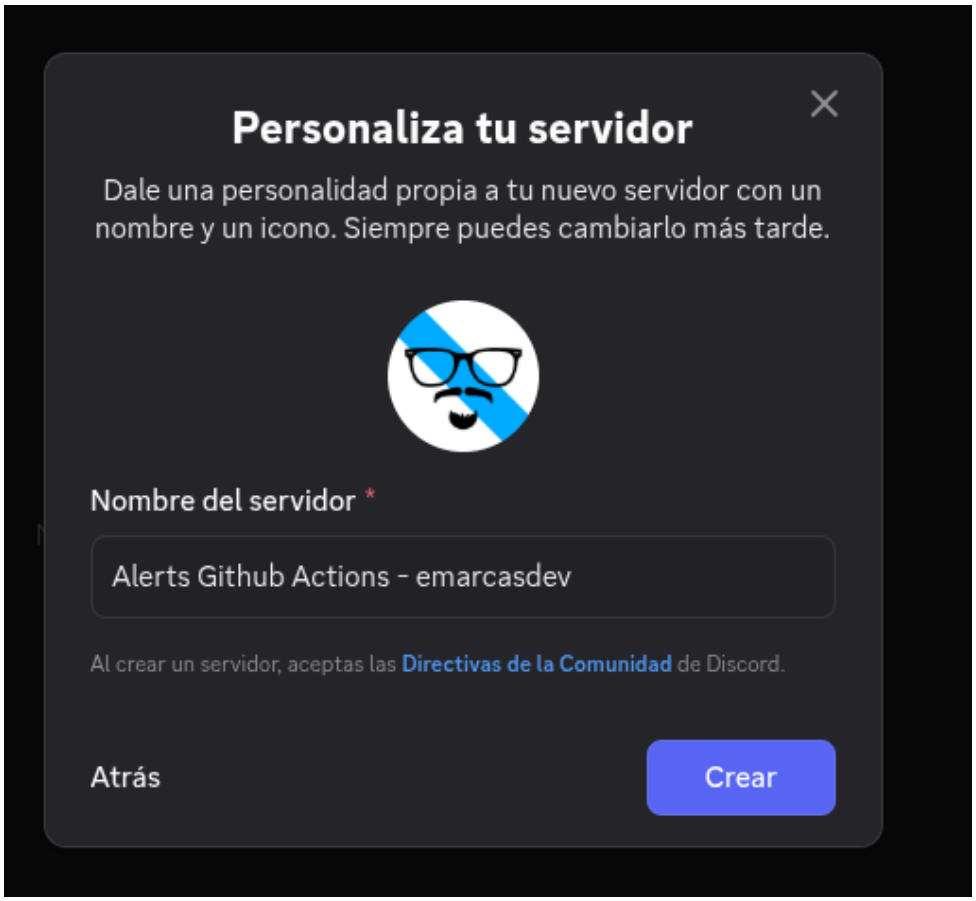

Figura 17: Creación de un servidor en Discord donde recibiremos las notificaciones del despliegue.

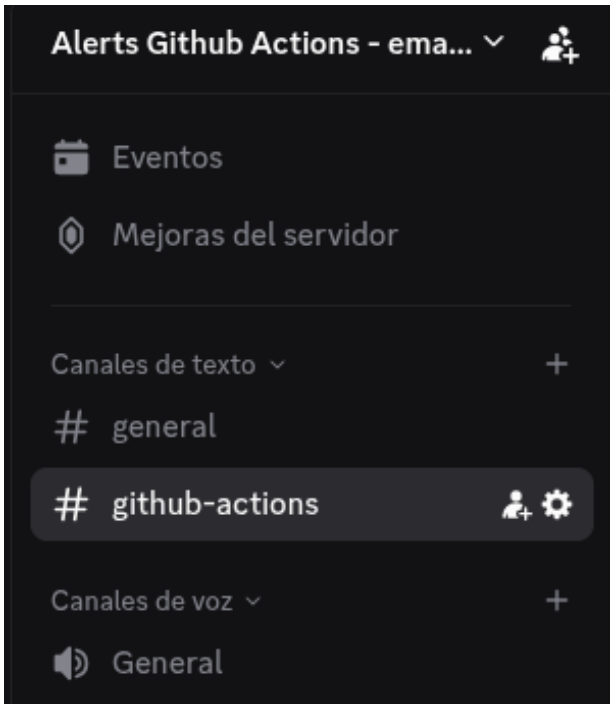

Figura 18: Creación del canal de texto **#github-actions** donde se publicarán las notificaciones.

Figura 19: Menú de edición del canal de texto en Discord. Desde la sección de integraciones crearemos el `webhook`.

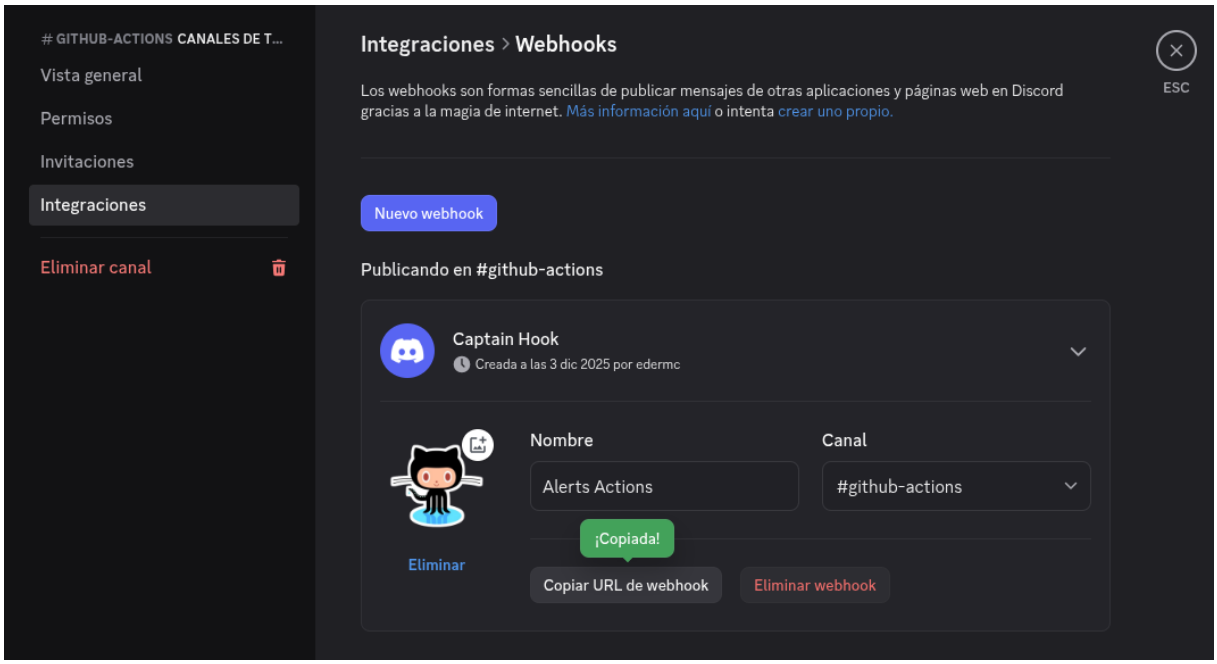

Figura 20: configuración del `webhook`: nombre, canal y una imagen. Lo más importante en este punto es copiar la **URL** de `webhook` que usaremos en el workflow.

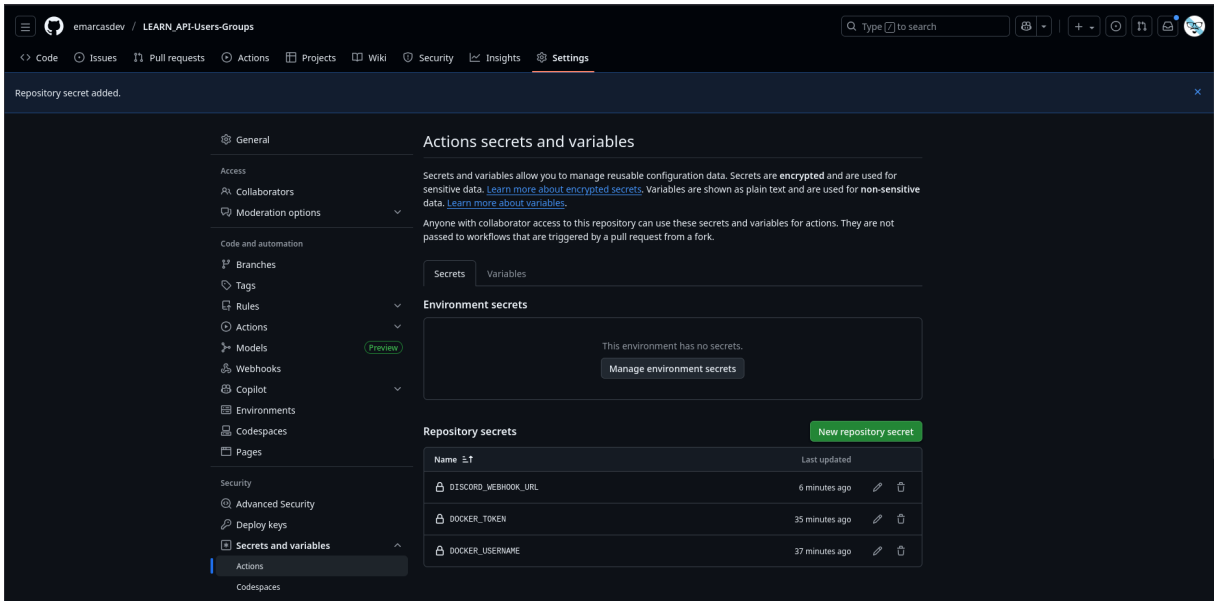

Figura 21: Agregamos la URL del `webhook` como **secret** en el repositorio de GitHub.

Cuando ya tenemos configurado el servidor, el canal y el `webhook` en Discord, y hemos agregado a los secrets de nuestro repositorio la URL del `webhook`, modificaremos el archivo `deploy.yml` agregando estás lineas al final del documento:

```
  # Mandar notificacion de exito en discord
  - name: Exito de la Action (Discord)
    if: success()
    run: |
      curl -X POST \
        -H "Content-Type: application/json" \
        -d "{\"content\":\"DESPLIEGUE EXITOSO en \`${{ github.
           repository }}\` (branch \`${{ github.ref_name }}\`).
           Imagen: \`${{ env.DOCKER_IMAGE_NAME }}:${{ env.
           DOCKER_TAG }}\`\"}" \
        ${{ secrets.DISCORD_WEBHOOK_URL }}

  # Mandar notificacion de que fallo en discord
  - name: Fallo de la Action (Discord)
    if: failure()
    run: |
      curl -X POST \
        -H "Content-Type: application/json" \
        -d "{\"content\":\"FALLO EL DESPLIEGUE en \`${{ github.
           repository }}\` (branch \`${{ github.ref_name }}\`).
           Revisa logs: https://github.com/${{ github.repository
           }}/actions/runs/${{ github.run_id }}\"}" \
        ${{ secrets.DISCORD_WEBHOOK_URL }}
```

Y a continuación se mostrarán las verificaciones de su funcionamiento:

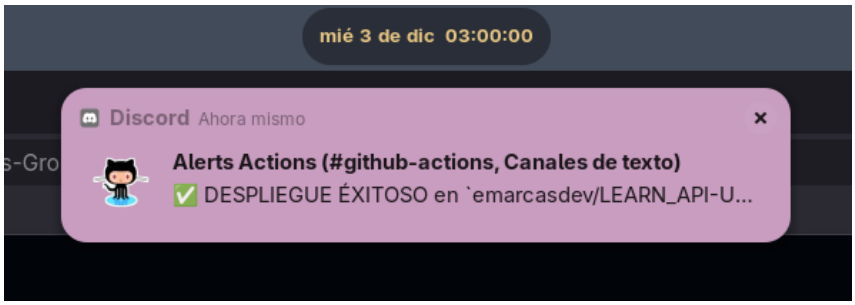

Figura 22: Captura de la notificación de éxito en Discord, verificando que se implemento correctamente la funcionalidad.

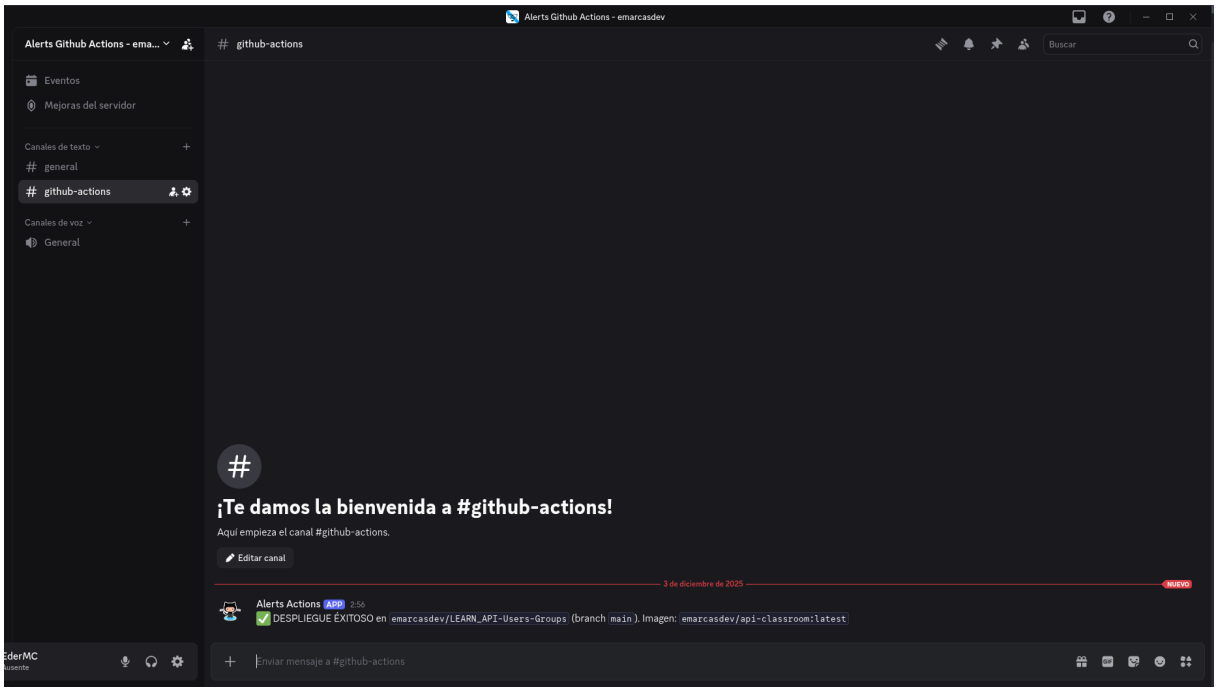

Figura 23: En esta captura vemos como en el canal de texto **#github-actions** de nuestro servidor se registran todos los mensajes sobre los despliegues.

Archivo `deploy.yml` completo

```
  name: Build and Push Docker Image

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

env:
  DOCKER_IMAGE_NAME: ${{ secrets.DOCKER_USERNAME }}/api-classroom
  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_TOKEN }}

      - 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
             }}
          cache-from: type=registry,ref=${{ env.DOCKER_IMAGE_NAME
              }}:buildcache
          cache-to: type=inline

      - name: Mostrar informacion de la imagen
        run: |
          echo "Imagen construida y subida exitosamente:"
          echo "  - Imagen: ${{ env.DOCKER_IMAGE_NAME }}:${{ env.
             DOCKER_TAG }}"
          echo "  - Docker Hub: https://hub.docker.com/r/${{
             secrets.DOCKER_USERNAME }}/api-classroom"

      # Mandar notificacion de exito en discord
      - name: Exito de la Action (Discord)
        if: success()
        run: |
          curl -X POST \
            -H "Content-Type: application/json" \
            -d "{\"content\":\"DESPLIEGUE EXITOSO en \`${{ github
               .repository }}\` (branch \`${{ github.ref_name
               }}\`). Imagen: \`${{ env.DOCKER_IMAGE_NAME }}:${{
               env.DOCKER_TAG }}\`\"}" \
            ${{ secrets.DISCORD_WEBHOOK_URL }}

      # Mandar notificacion de que fallo en discord
      - name: Fallo de la Action (Discord)
        if: failure()
        run: |
          curl -X POST \
            -H "Content-Type: application/json" \
            -d "{\"content\":\"FALLO EL DESPLIEGUE en \`${{
               github.repository }}\` (branch \`${{ github.
               ref_name }}\`). Revisa logs: https://github.com/${{
                github.repository }}/actions/runs/${{ github.
               run_id }}\"}" \
```

```
            ${{ secrets.DISCORD_WEBHOOK_URL }}
```