Afondamento

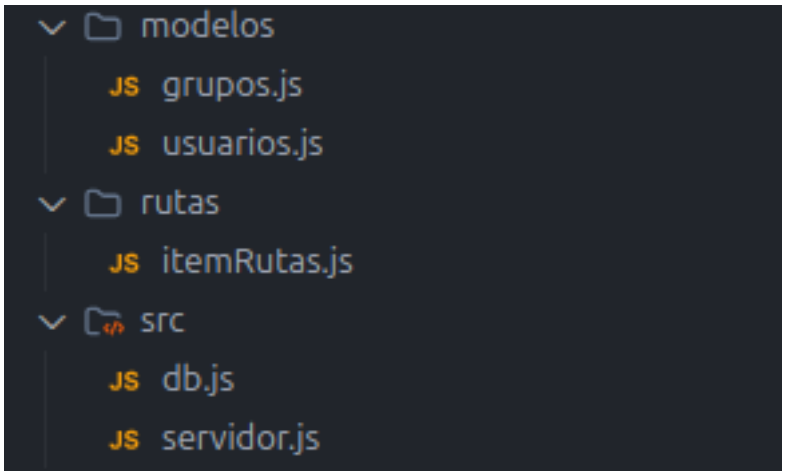
Documentación API Docker

Jacobo Pérez de Torres

Índice

1. Estructura de la API

Para la creación de la API he empleado Express.js con una base de datos local, desplegada a traves de Docker Compose. En ese apartado desglosaré la estrucura seguida en la creación de la API.



1.1. Modelos

Contiene los esquemas de los modelos existentes para la base de datos y la API. En este caso contiene "Gruposz Üsuarios".

```
// Modelo de grupos
import mongoose from "mongoose";

const GrupoEschema = new mongoose.Schema({
    nombreGrupo: { type: String, required: true },
    integrantesGrupo: { type: Array, required: true }
});
```

```
// Modelo de usuarios
import mongoose from "mongoose";

const UsuarioEschema = new mongoose.Schema({
    nombreUsuario: { type: String, required: true },
    apellidoUsuario: { type: String, required: true },
    dniUsuario: { type: String, required: true },
    edadUsuario: {type: Number, required: true }
});

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

1.2. Rutas

En este apartado he empleado router para definir los endpoints de cada método de la API.

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

const router = express.Router();

// Metodos de recibo
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);
});

// Metodos de envio
router.post("/usuarios", async (req, res) => {
  const nuevoUsuario = new Usuario(req.body);
  const usuarioGuardado = await nuevoUsuario.save();
  res.status(201).json(usuarioGuardado);
});

router.post("/grupos", async (req, res) => {
  const nuevoGrupo = new Comanda(req.body);
  const grupoGuardado = await nuevoGrupo.save();
  res.status(201).json(grupoGuardado);
});

// Metodos de actualizacion
router.put("/usuarios/:dniUsuario", async (req, res) => {
  try {
    const usuarioActualizado = await Usuario.findOneAndUpdate(
      { dniUsuario: req.params.numeroMesa },
      req.body,
      { new: true }
    );
    res.json(usuarioActualizado);
    console.log(`Usuario ${dniUsuario} actualizado!`);
  } catch (error) {
    console.log("Error al actualizar usuario!", error);
    res.status(500).json({ message: "Error al actualizar usuario!
      " });
  }
});

router.put("/usuarios/:dniUsuario", async (req, res) => {
  try {
    const usuarioActualizado = await Usuario.findOneAndUpdate(
      { dniUsuario: req.params.numeroMesa },
```

```
      req.body,
      { new: true }
    );
    res.json(usuarioActualizado);
    console.log(`Usuario ${dniUsuario} actualizado!`);
  } catch (error) {
    console.log("Error al actualizar usuario!", error);
    res.status(500).json({ message: "Error al actualizar usuario!" });
  }
});

export default router;
```

1.3. Source

En esta carpeta he guardado los archivos principales de la API.

```
// db.js -> Contiene el codigo de conexion a la base de datos
import mongoose from "mongoose";
import dotenv from "dotenv";
dotenv.config();

const uri = process.env.MONGO_URI;

export async function connectDB() {
  try {
    await mongoose.connect(uri, {});
    console.log("Base de datos MongoDB conectada");
  } catch (err) {
    console.error("Error conectandose a la base de datos MongoDB:", err);
    process.exit(1);
  }
}
```

```
// servidor.js -> Contiene el codigo de inicio del servidor
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import { connectDB } from "./db.js";
import rutas from "../rutas/itemRutas.js"

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

app.use(rutas);

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

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

Los datos referentes a la base de datos local se han guardado en un archivo .env mediante variables de entorno.

```
MONGO_URI = mongodb://localhost:27017/backend_docker
PORT = 3000
```

2. Dockerización de la API

Para poder crear una imagen de docker con nuestra API de express deberemos crear el archivo Dockerfile:

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

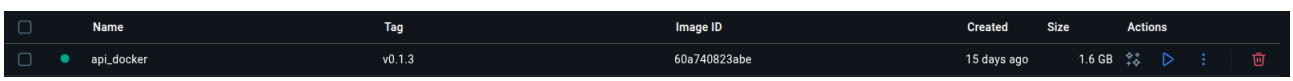
Este archivo quiere decir que desde la versión 20 de Node.js, empleado como directorio de trabajo /app"se copiara todos los paquetes .json, se instalarán los modulos npm de producción, se copiará el contenido a la ruta /appz se abrirá el puerto 3000.
En caso de necesitarlo y querer podemos también crear un archivo ".dockerignore", que funciona prácticamente como un archivo ".gitignore", los archivos o carpetas que determinemos en este archivo serán ignorados por docker y no se añadirán al interior de la ruta.

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

Una vez hayamos finalizado la creación de estos archivos podremos construir la imagen a traves del comando:

```
docker build -t api_docker:v0.1.3
```

Con esto habremos creado la imagen correctamente y si entramos a Docker Desktop podremos comprobar que se ha creado la imagen en imágenes.



3. Configuración del docker-compose.yml

En este apartado mostraré el funcionamiento y composición del docker.compose conteniendo la base de datos local y la imagen creada en el apartado anterior:

```yaml
services:
  mongo:
    image: mongo:7
    container_name: mongo_local
    restart: always
    ports:
      - "27018:27017"
    volumes:
      - ./data:/data/db

  api:
    image: api_docker:v0.1.3
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
      - MONGO_URI=mongodb://mongo:27017/backend_docker
    depends_on:
      - mongo
```

Incluimos ambas imágenes dentro de "services", la primera será la imagen de mongo que se descargará desde dockerhub. Simplemente le damos un nombre al contenedor, seleccionamos los puertos internos y externos y el volumen de datos. La segunda imagen será la de nuestra API, hacemos lo mismo que con la anterior, pero esta ve además deberemos determinar las variables de entorno ya que docker no podrá acceder a nuestro .env y tampoco subiremos este a github.

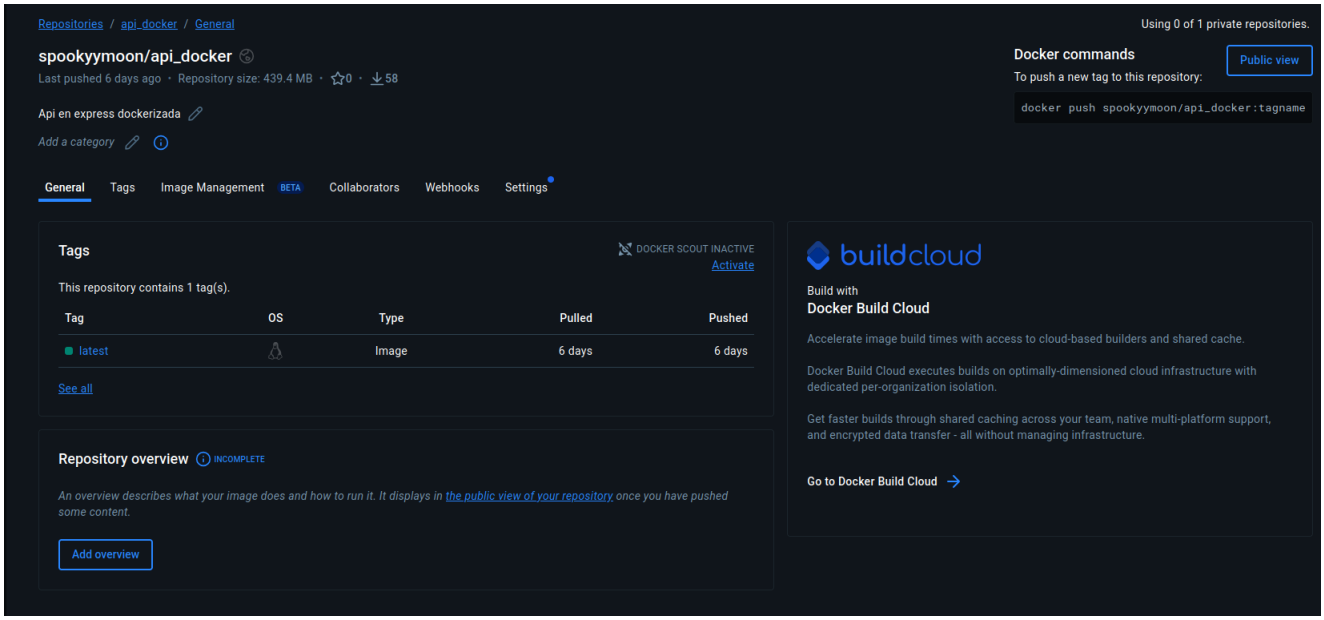
4. Subir imagen a DockerHub

Para subir nuestra imagen a DockerHub simplemente crearemos un repositorio desde el Hub en el que podamos subir la imagen.

Una vez tengamos creado el repositorio deberemos subir la imagen con el siguiente comando:

```
docker push spoolyymoon/api_docker:v0.1.3
```

Cabe destacar que si usamos linux deberemos crear una clave personal a traves del Bash para poder iniciar sesión en DockerHub. Una vez subida la imagen al Hub y si lo hemos hecho correctamente podremos ver la imagen subida en el repositorio.



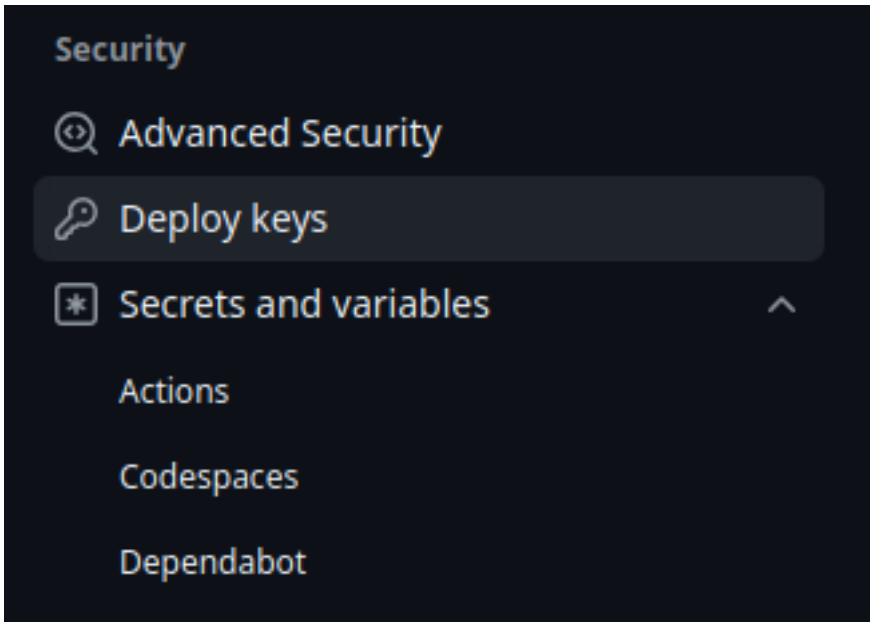
5. Github actions

En esta sección crearemos un workflow de Github que creará una nueva versión de la imagen y la subirá a DockerHub, para comenzar con esto, deberemos crear una nueva carpeta en nuestro espacio de trabajo y la llamaremos ".github", en su interior crearemos una carpeta llamada "workflowsz finalmente, en su interior, crearemos el archivo "docker-push.yml". Dentro de este archivo introduciremos el siguiente código:

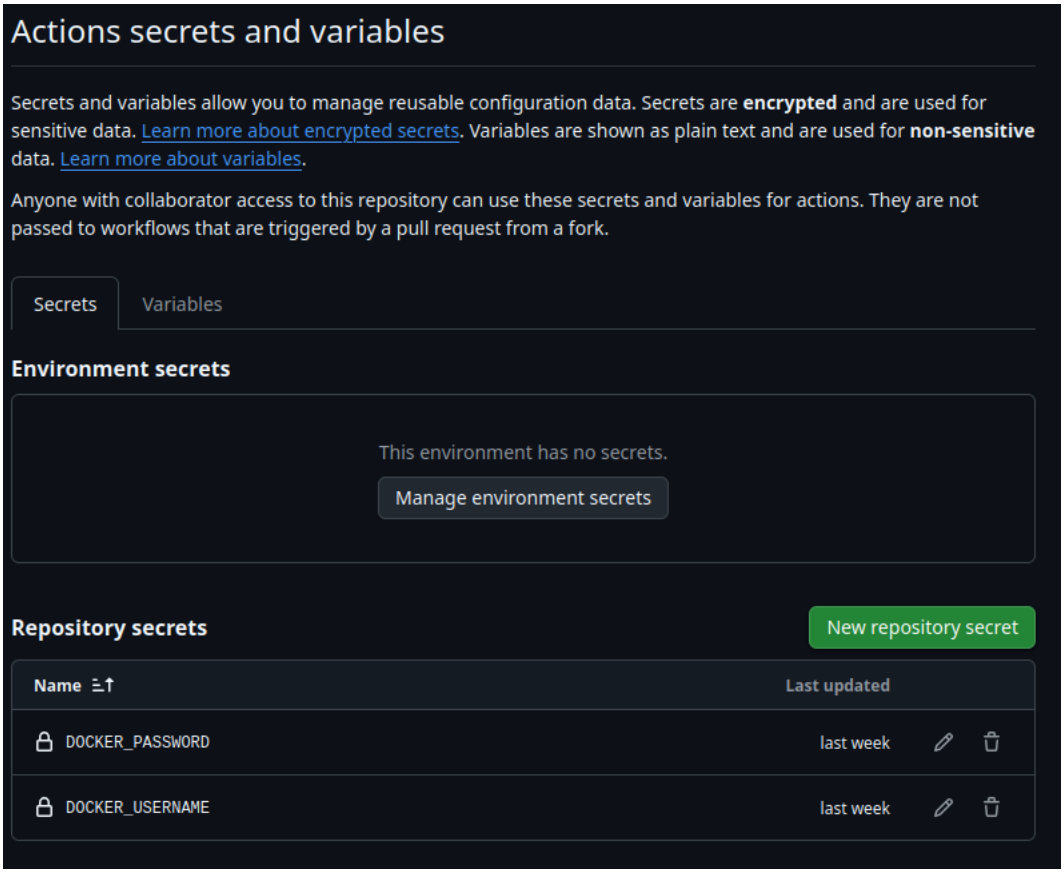
```yaml
name: Build and Push Docker Image klk

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

```

```yaml
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 }}
          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/${{ secrets.DOCKER_USERNAME }}/${{ env.DOCKER_IMAGE_NAME }}"
```

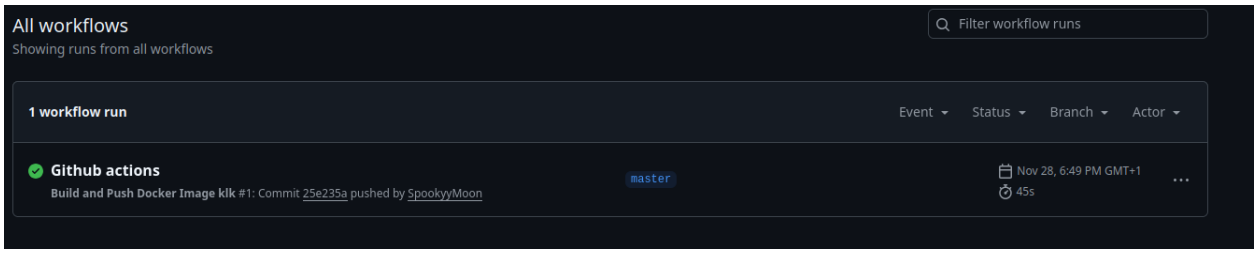
Las variables de entorno que se emplean en este archivo se almacenarán en GitHub Secrets. Para crear secretos simplemente nos dirigimos a github a el repositorio que estemos empleando y dentro de "Settings"debemos localizar "Secrets and variables".



Dentro de este apartado podremos crear los secrets que deseemos como si fuesen variables de entorno.

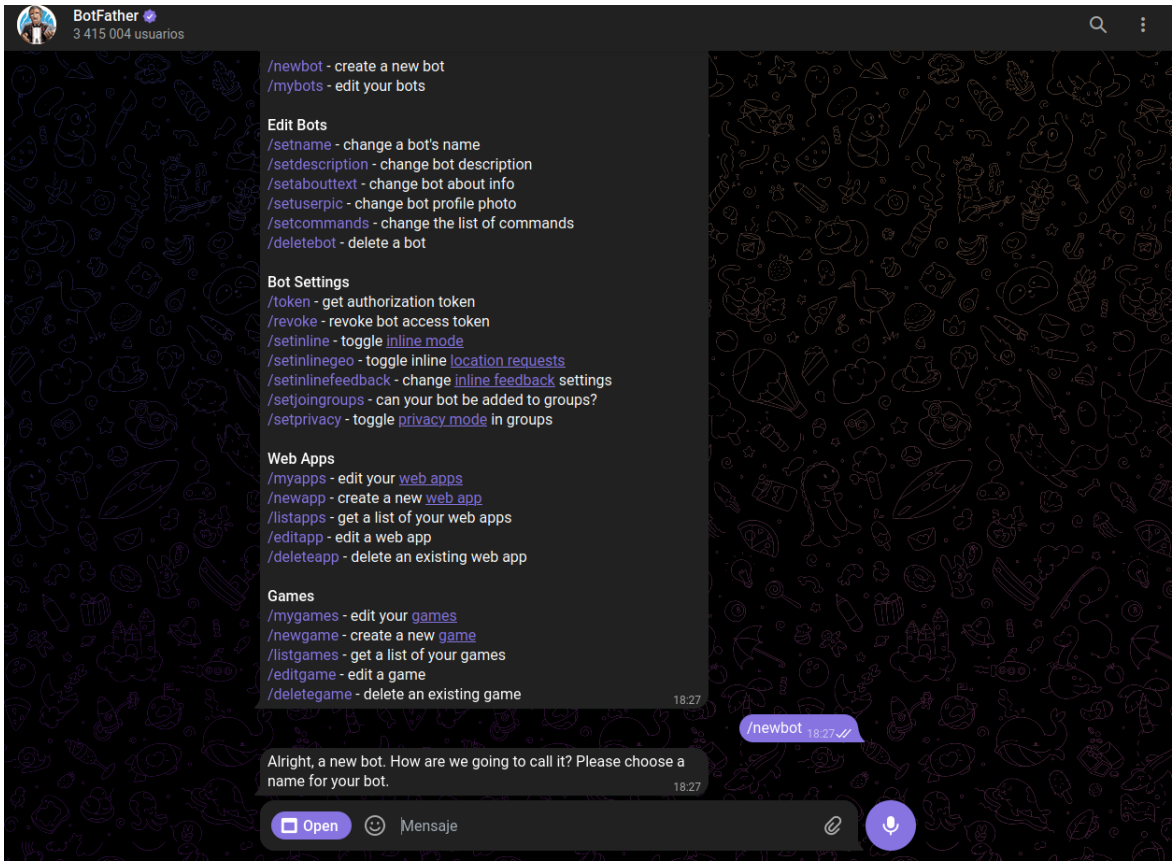


Si después de todo esto accedemos a .^Actions.^en la toolbar del repositorio podremos ver nuestro action:



6. Notificación a telegram con github actions

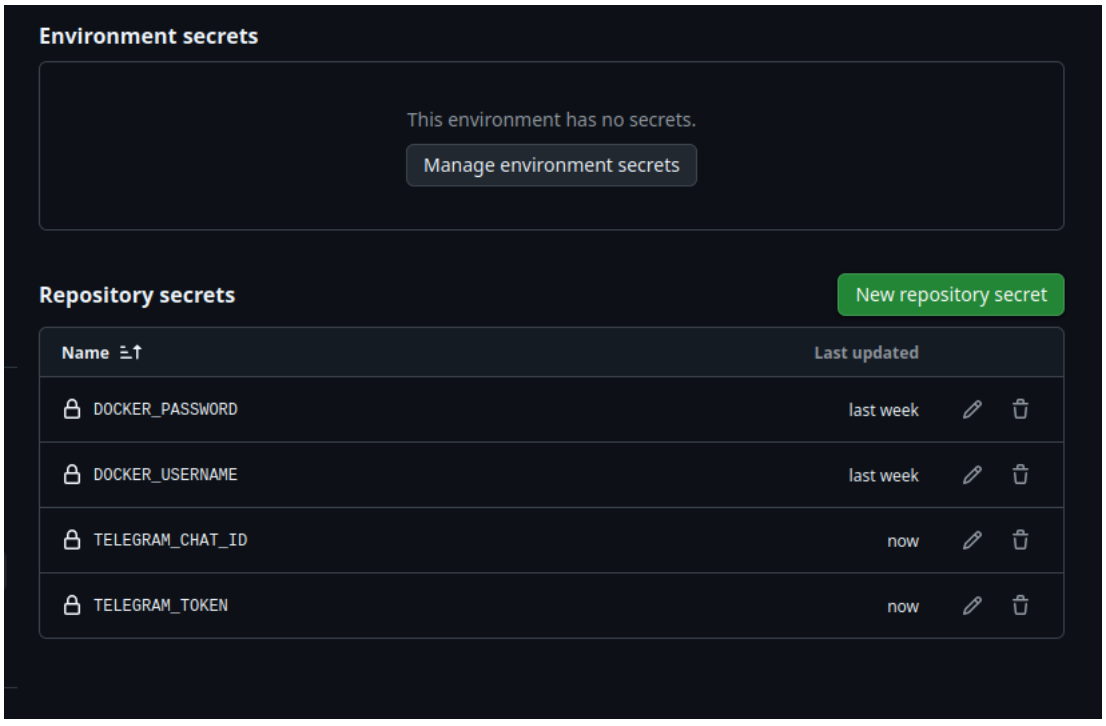
Para comenzar con a registrar los commits y notificarlos en telegram a traves de github actions vamos a seguir el siguiente tutorial: https://github.com/marketplace/actions/telegram-notify
Lo primero que necesitamos hacer es crear un bot de telegram, para ello abrimos la aplicación, buscamos al usuario "@BotFatherz abrimos un chat con él: Una vez hayamos abierto el chat con el escribimos /newbot", a continuación seguimos los pasos rellenando las opciones:



Tras esto buscamos el chat con nuestro bot y lo iniciamos. Una vez iniciado el chat le enviamos cualquier mensaje para activarlo y escribimos el comando çurl -s "https://api.telegram.org/bot[TOK jqçambiando token por el token que hemos recibido al crear el bot.

```
mn0VBiLpdIjnJs4zseM7OY/getUpdates" | jq
{
  "ok": true,
  "result": [
    {
      "update_id": 482273579,
      "message": {
        "message_id": 5,
        "from": {
          "id": 7920708362,
          "is_bot": false,
          "first_name": "Spooky",
          "language_code": "es"
        },
        "chat": {
          "id": 7920708362,
          "first_name": "Spooky",
          "type": "private"
        },
        "date": 1765042477,
        "text": "hola"
      }
    }
  ]
```


Del json que hemos recibido debemos quedarnos con el ID del chat y nos dirigimos de nuevo a github secrets, donde crearemos dos nuevos secretos, uno contenido el token del bot y otro la ID del chat.



Dentro de la carpeta ".github/workflowsçreamos un archivo .yml que contendrá nuestro action (Que sacaremos del tutorial):

```
1 name: telegram message
2 on: [push]
```

```
jobs:

  build:
    name: Build
    runs-on: ubuntu-latest
    steps:
      - name: send telegram message on push
        uses: appleboy/telegram-action@master
        with:
          to: ${{ secrets.TELEGRAM_CHAT_ID }}
          token: ${{ secrets.TELEGRAM_TOKEN }}
          message: |
            ${{ github.actor }} created commit:
            Commit message: ${{ github.event.commits[0].message
               }}

            Repository: ${{ github.repository }}

            See changes: https://github.com/${{ github.repository
                }}/commit/${{github.sha}}
```

Por último hacemos commit y push de nuestros cambios para añadirlos a github y probamos que todo funcione haciendo un commit y push cualquiera. Si todo ha ido bien el bot debería habernos mandado un mensaje: