

Afundamento

DOCUMENTACIÓN API DOCKER

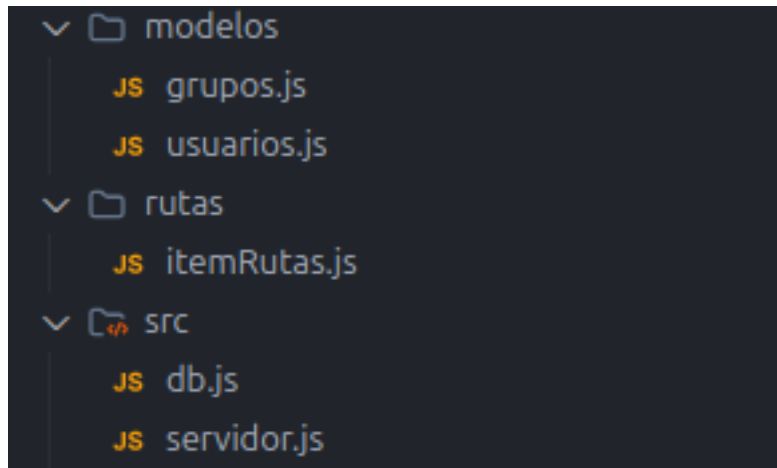
Jacobo Pérez de Torres

Índice

1. Estructura de la API	2
1.1. Modelos	2
1.2. Rutas	2
1.3. Source	4
2. Dockerización de la API	5
3. Configuración del docker-compose.yml	6
4. Subir imagen a DockerHub	6
5. Github actions	7
6. Notificación a telegram con github actions	10

1. Estructura de la API

Para la creación de la API he empleado Express.js con una base de datos local, desplegada a través de Docker Compose. En ese apartado desglosaré la estructura 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 "Grupos" y "Usuarios".

```
1 // Modelo de grupos
2 import mongoose from "mongoose";
3
4 const GrupoEschema = new mongoose.Schema({
5   nombreGrupo: { type: String, required: true },
6   integrantesGrupo: { type: Array, required: true }
7 });
```

```
1 // Modelo de usuarios
2 import mongoose from "mongoose";
3
4 const UsuarioEschema = new mongoose.Schema({
5   nombreUsuario: { type: String, required: true },
6   apellidoUsuario: { type: String, required: true },
7   dniUsuario: { type: String, required: true },
8   edadUsuario: { type: Number, required: true }
9 });
10
11 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.

```

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 // Metodos de recibo
8 router.get("/usuarios", async (req, res) => {
9   const usuarios = await Usuario.find();
10  res.json(usuarios);
11 });
12
13 router.get("/grupos", async (req, res) => {
14   const grupos = await Grupo.find();
15   res.json(grupos);
16 });
17
18 // Metodos de envio
19 router.post("/usuarios", async (req, res) => {
20   const nuevoUsuario = new Usuario(req.body);
21   const usuarioGuardado = await nuevoUsuario.save();
22   res.status(201).json(usuarioGuardado);
23 });
24
25 router.post("/grupos", async (req, res) => {
26   const nuevoGrupo = new Comanda(req.body);
27   const grupoGuardado = await nuevoGrupo.save();
28   res.status(201).json(grupoGuardado);
29 });
30
31 // Metodos de actualizacion
32 router.put("/usuarios/:dniUsuario", async (req, res) => {
33   try {
34     const usuarioActualizado = await Usuario.findOneAndUpdate(
35       { dniUsuario: req.params.numeroMesa },
36       req.body,
37       { new: true }
38     );
39     res.json(usuarioActualizado);
40     console.log(`Usuario ${dniUsuario} actualizado!`);
41   } catch (error) {
42     console.log("Error al actualizar usuario!", error);
43     res.status(500).json({ message: "Error al actualizar usuario!"
44       " });
45   }
46 });
47
48 router.put("/usuarios/:dniUsuario", async (req, res) => {
49   try {
50     const usuarioActualizado = await Usuario.findOneAndUpdate(
51       { dniUsuario: req.params.numeroMesa },

```

```

51     req.body,
52     { new: true }
53   );
54   res.json(usuarioActualizado);
55   console.log('Usuario ${dniUsuario} actualizado!');
56 } catch (error) {
57   console.log("Error al actualizar usuario!", error);
58   res.status(500).json({ message: "Error al actualizar usuario!"
59     " });
60 });
61
62 export default router;

```

1.3. Source

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

```

1  // db.js -> Contiene el código de conexión a la base de datos
2  import mongoose from "mongoose";
3  import dotenv from "dotenv";
4  dotenv.config();
5
6  const uri = process.env.MONGO_URI;
7
8  export async function connectDB() {
9    try {
10      await mongoose.connect(uri, {});
11      console.log("Base de datos MongoDB conectada");
12    } catch (err) {
13      console.error("Error conectandose a la base de datos MongoDB:
14        ", err);
15      process.exit(1);
16    }
17  }

```

```

1  // servidor.js -> Contiene el código de inicio del servidor
2  import express from "express";
3  import cors from "cors";
4  import dotenv from "dotenv";
5  import { connectDB } from "./db.js";
6  import rutas from "../rutas/itemRutas.js"
7
8  dotenv.config();
9  const app = express();
10 app.use(cors());
11 app.use(express.json());
12
13 app.use(rutas);
14
15 const PORT = process.env.PORT || 3000;

```

```

16 connectDB().then(async () => {
17   app.listen(PORT, () => console.log('API en ${PORT}'));
18 });

```

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:

```

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

```

Este archivo quiere decir que desde la versión 20 de Node.js, empleado como directorio de trabajo /app se copiarán todos los paquetes .json, se instalarán los módulos npm de producción, se copiará el contenido a la ruta /app y 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.

```

1 node_modules
2 documentacion
3 npm-debug.log
4 .git
5 .gitignore
6 .env
7 *.md
8 docker-compose.yml

```

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

```

1 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.

<input type="checkbox"/>	Name	Tag	Image ID	Created	Size	Actions
<input checked="" type="checkbox"/>	api_docker	v0.1.3	60a740823abe	15 days ago	1.6 GB	   

3. Configuración del docker-compose.yml

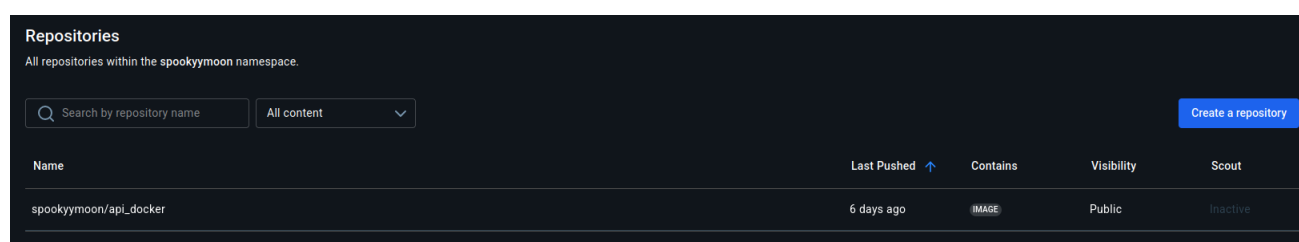
En este apartado mostraré el funcionamiento y composición del docker.compose conteniendo la base de datos local y la imagen creada en el apartado anterior:

```
1 services:
2   mongo:
3     image: mongo:7
4     container_name: mongo_local
5     restart: always
6     ports:
7       - "27018:27017"
8     volumes:
9       - ./data:/data/db
10
11   api:
12     image: api_docker:v0.1.3
13     ports:
14       - "3000:3000"
15     environment:
16       - PORT=3000
17       - MONGO_URI=mongodb://mongo:27017/backend_docker
18     depends_on:
19       - 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 vez 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

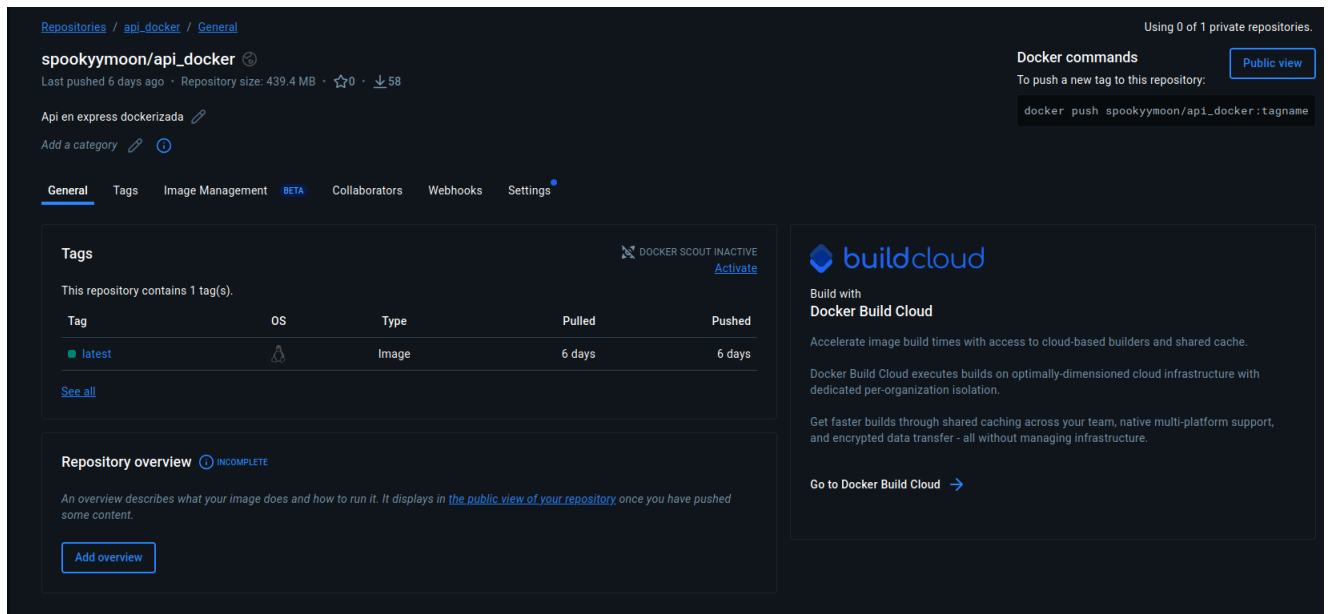
Para subir nuestra imagen a DockerHub simplemente crearemos un repositorio desde el Hub en el que podamos subir la imagen.



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

```
1 docker push spookyymoon/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 "workflows" finalmente, en su interior, crearemos el archivo "docker-push.yml". Dentro de este archivo introduciremos el siguiente código:

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

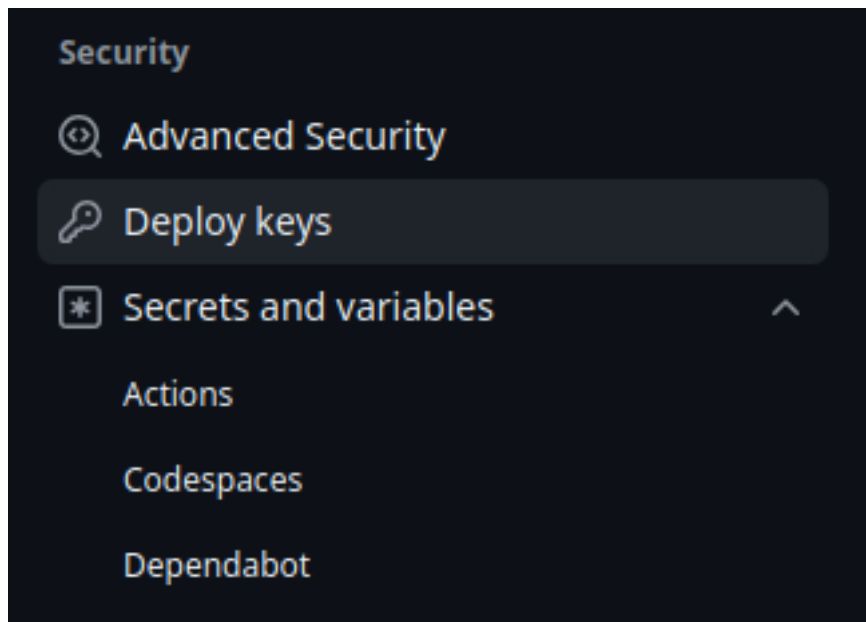


```

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 }
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/${ secrets.DOCKER_USERNAME }/${ 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.

Actions secrets and variables

Secrets and variables allow you to manage reusable configuration data. Secrets are **encrypted** and are used for sensitive data. [Learn more about encrypted secrets](#). Variables are shown as plain text and are used for **non-sensitive** data. [Learn more about variables](#).

Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.

Secrets Variables

Environment secrets

This environment has no secrets.

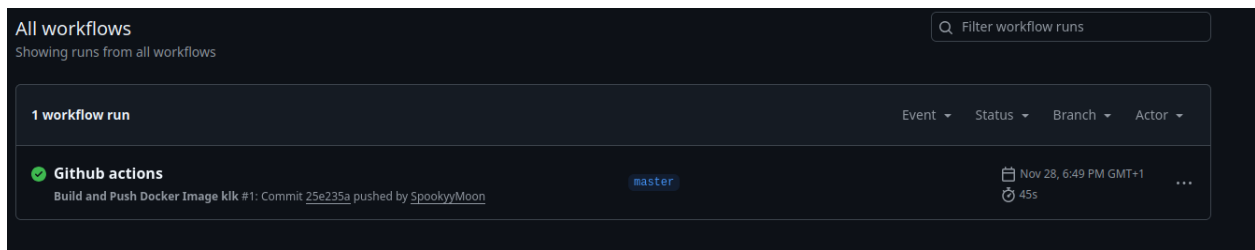
[Manage environment secrets](#)

Repository secrets

[New repository secret](#)

Name	Last updated
DOCKER_PASSWORD	last week
DOCKER_USERNAME	last week

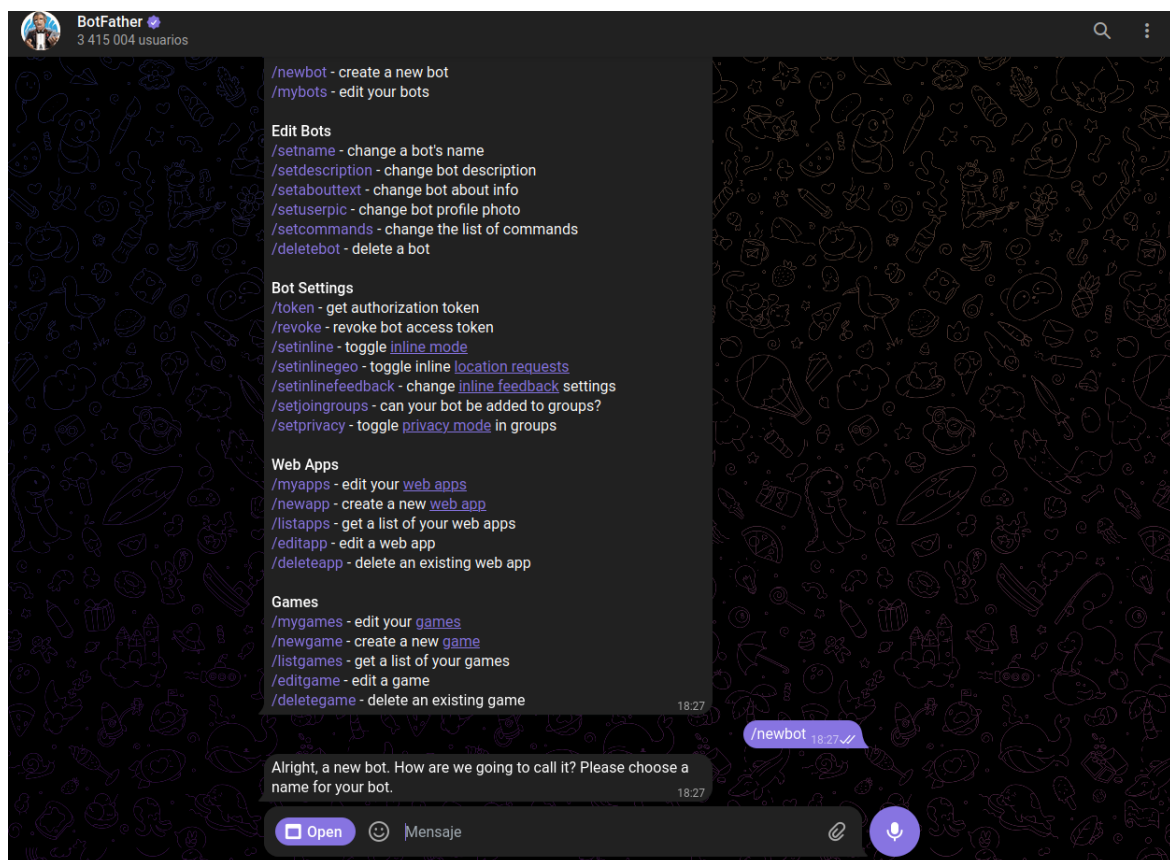
Si después de todo esto accedemos a `.Actions` en la toolbar del repositorio podremos ver nuestro action:



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 `curl -s "https://api.telegram.org/bot[TOKEN]sendMessage?text=any_message"` cambiando token por el token que hemos recibido al crear el bot.

```
mn0VBilPdIjnJs4zseM70Y/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.

Environment secrets

This environment has no secrets.

Manage environment secrets

Repository secrets

New repository secret

Name	Last updated	
DOCKER_PASSWORD	last week	
DOCKER_USERNAME	last week	
TELEGRAM_CHAT_ID	now	
TELEGRAM_TOKEN	now	

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

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

```

3 jobs:
4
5   build:
6     name: Build
7     runs-on: ubuntu-latest
8     steps:
9       - name: send telegram message on push
10         uses: appleboy/telegram-action@master
11         with:
12           to: ${ secrets.TELEGRAM_CHAT_ID }
13           token: ${ secrets.TELEGRAM_TOKEN }
14           message: |
15             ${ github.actor } created commit:
16             Commit message: ${ github.event.commits[0].message
17               }
18
19             Repository: ${ github.repository }
20
21             See changes: https://github.com/${ github.repository
22               }/commit/${ github.sha }

```

Por último hacemos commit y push de nuestros cambios para añadirlos a github y probamos que todo funcione.