

DOCUMENTACIÓN DE LA API DOCKERIZADA

Gestión de Usuarios y Grupos

Eder Martínez Castro

18 de noviembre de 2025

Índice general

1.	Introducción	2
2.	Arquitectura	2
2.1.	Clases	2
2.2.	Patrones de diseño	3
2.3.	Stack empleado	3
3.	Dependencias utilizadas	4
4.	Modelo de datos	4
5.	Endpoints	4
6.	Despliegue dockerizado	10
6.1.	Variables de entorno	10
6.2.	Dockerfile de la API	10
6.3.	docker-compose.yml	11
6.4.	Arranque del proyecto con Docker Compose	11
6.5.	Comprobación del despliegue	12

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 función `createTablesIfNotExists()`, así dejamos la API lista para que se pueda conectar. Toda la lógica la tenemos en un único archivo `server.js`, lo cual simplifica la estructura del proyecto 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 fácilmente** 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.

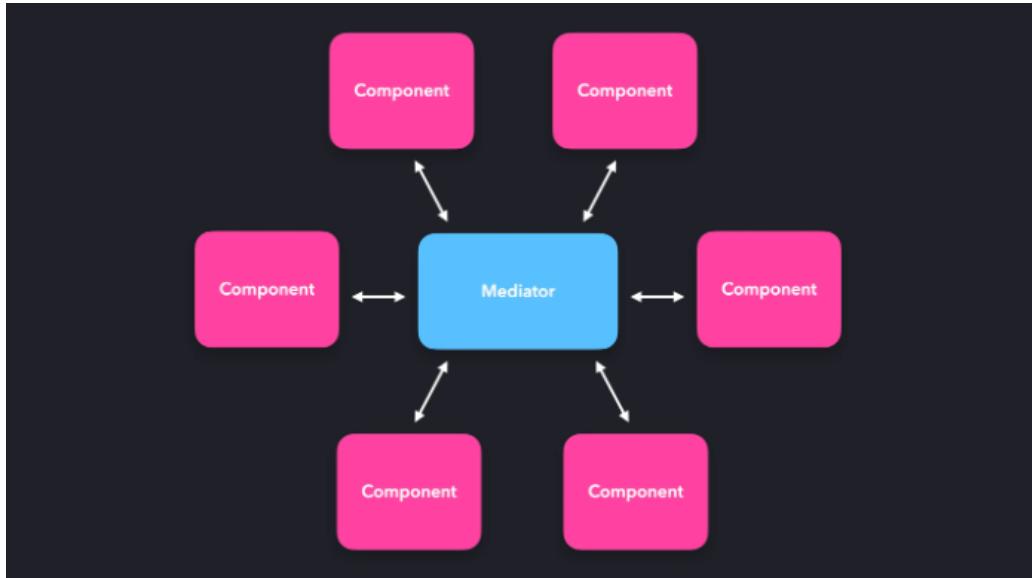


Figura 2: Patrón Mediator/Middleware

2.3. Stack empleado

Para el desarrollo de la API, se utiliza los siguientes tecnologías:

- 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.

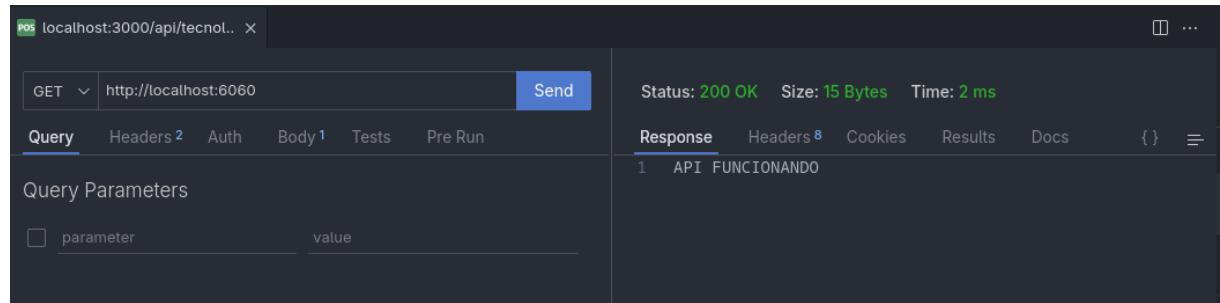
GET / - Comprobar el estado de la API

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

Respuesta

200 - Successful Response

API FUNCIONANDO



The screenshot shows the Thunder Client interface. On the left, there's a query builder with a 'GET' method selected, pointing to 'http://localhost:6060'. Below it, under 'Query Parameters', there are two fields: 'parameter' and 'value'. On the right, the results panel shows the status: 'Status: 200 OK Size: 15 Bytes Time: 2 ms'. Under the 'Response' tab, the text 'API FUNCIONANDO' is displayed.

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>"}
```

The screenshot shows the Thunder Client interface. At the top, there's a header bar with 'GET' and the URL 'http://localhost:6060/api/users-groups'. Below this is a toolbar with tabs for 'Query', 'Headers 2', 'Auth', 'Body', 'Tests', and 'Pre Run'. A 'Send' button is located at the top right. The main area has two sections: 'Query Parameters' on the left and 'Response' on the right. The 'Response' section shows a JSON array of users:

```
1 {
2   "users": [
3     {
4       "id": 1,
5       "name": "David",
6       "surname": "Martinez",
7       "marks": 5,
8       "group_id": 1
9     },
10    {
11      "id": 2,
12      "name": "Agustin",
13      "surname": "Garcia",
14      "marks": 8,
15      "group_id": 2
16    },
17    {
18      "id": 3,
19      "name": "Iván",
20      "surname": "Priego",
21      "marks": 6,
22      "group_id": 3
23    }
24 }
```

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

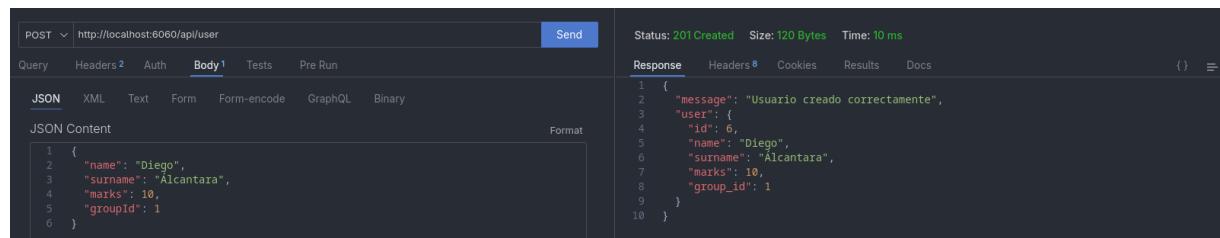
```
{"message": "Falta alguno de los 4 campos requeridos"}
```

400 - Bad Request

```
{"message": "La nota del usuario tiene que ser un entero."}
```

500 - Internal Server Error

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



The screenshot shows a POST request to `http://localhost:6060/api/user`. The Body tab contains the following JSON:

```
1 {
2   "name": "Diego",
3   "surname": "Alcantara",
4   "marks": 10,
5   "groupid": 1
6 }
```

The Response tab shows a successful `201 Created` status with a size of 120 Bytes and a time of 10 ms. The response body is:

```
1 {
2   "message": "Usuario creado correctamente",
3   "user": {
4     "id": 6,
5     "name": "Diego",
6     "surname": "Alcantara",
7     "marks": 10,
8     "groupid": 1
9   }
10 }
```

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>"}
```

The screenshot shows the Thunder Client interface. On the left, there's a request panel with a POST method, URL http://localhost:6060/api/group, and a JSON body containing { "groupName": "Clase 1ºB" }. On the right, the response panel shows a status of 201 Created, size of 83 Bytes, and time of 10 ms. The response body is a JSON object with a message key containing the value "Grupo creado correctamente".

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 usuario 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 usuario 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>"}
```

```

PUT http://localhost:6060/api/user/1/marks
Send
Query Headers 2 Auth Body 1 Tests Pre Run
JSON XML Text Form Form-encode GraphQL Binary
JSON Content Format
1 {
2   "marks": 9
3 }

Response Headers 8 Cookies Results Docs
Status: 200 OK Size: 60 Bytes Time: 12 ms
1 {
2   "message": "La nota del usuario cambio a: 9",
3   "user": {
4     "id": 1
5   }
6 }

```

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>"}
```

```

DELETE http://localhost:6060/api/user/6
Send
Query Headers 2 Auth Body 1 Tests Pre Run
JSON XML Text Form Form-encode GraphQL Binary
JSON Content Format
1

Response Headers 8 Cookies Results Docs
Status: 200 OK Size: 61 Bytes Time: 13 ms
1 {
2   "message": "Usuario eliminada correctamente",
3   "user": {
4     "id": 6
5   }
6 }

```

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 necesarias estan declaradas en el archivo `.env`. Un ejemplo del contenido que necesitariamos en el archivo sería:

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

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

```
emarcas@fedev ~/DAM-projects/ACP_CS/PRACTICA-DOCKER/DB-ACP docker compose up -d --build

[+] Building 1.5s (12/12) FINISHED
=> [internal] load local bake definitions
=> => reading from stdin 551B
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 462B
=> [internal] load metadata for docker.io/library/node:20
=> [internal] load .dockerignore
=> => transferring context: 133B
=> [1/5] FROM docker.io/library/node:20@sha256:47dacd49500971c0fbe602323b2d04f6df40a933b123889636fc1f76bf69f58a
=> [internal] load build context
=> => transferring context: 289.16kB
=> CACHED [2/5] WORKDIR /api
=> CACHED [3/5] COPY package*.json .
=> CACHED [4/5] RUN npm install
=> [5/5] COPY .
=> exporting to image
=> => exporting layers
=> => writing image sha256:ee3156b0e6f5e893ad4416a9cbd20773412df83a29ceff3cc80f8b057e28a7d6
=> => naming to docker.io/library/db-acp-api
=> resolving provenance for metadata file
[+] Running 4/4
✓ db-acp-api           Built
✓ Network db-acp_default Created
✓ Container mariadb_classroom Started
✓ Container api_classroom Started
```

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
```

```
emarcas@fedev ~/DAM-projects/ACP_CS/PRACTICA-DOCKER/DB-ACP docker compose ps
NAME          IMAGE        COMMAND
api_classroom  db-acp-api  "docker-entrypoint.s..."
mariadb_classroom  mariadb:lts-ubi9  "docker-entrypoint.s...
emarcas@fedev ~/DAM-projects/ACP_CS/PRACTICA-DOCKER/DB-ACP
```

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

```
docker compose ps
```