TALLER – APIREST HEROES

ALMACENAMIENTO DE DATOS

DEYTON RIASCOS ORTIZ
2246208
SAMUEL IZQUIERDO BONILLA
2246993

UNIVERSIDAD AUTÓNOMA DE OCCIDENTE
FACULTAD DE INGENIERÍA

PROGRAMA INGENIERÍA DE DATOS E INTELIGENCIA ARTIFICIAL

SANTIAGO DE CALI
2025

Octubre 1 de 2025

DOCUMENTACIÓN PROYECTO BACKEND APIREST HEROES

1. MIGRACIÓN Y ESTÁNDARES DE DESARROLLO

1.1 Migración a Módulos ES (import/export)

El proyecto ha sido configurado para utilizar la sintaxis moderna de módulos de JavaScript, abandonando la estructura clásica de require().

Característica	Sintaxis Antigua (CommonJS)	Sintaxis Actual (ES Modules)	Archivos Clave
Importación	const express = require('express');	import express from 'express';	Múltiples, ej: server.js.
Exportación	module.exports = { miFuncion };	export const miFuncion = ...; o export default ...;	*Middlewares*, *Helpers*, *Routes*.
Rutas Dinámicas	Se utiliza (await import(...)).default para cargar rutas dinámicamente en server.js.		

1.2 Configuración de Variables de Entorno (.env)

El archivo principal (app.js) carga las variables de entorno al iniciar el servidor usando dotenv.

- JavaScript

```
//
app.js
import dotenv from 'dotenv'; dotenv.config(); // Carga las variables del archivo .env
// ...
```

Estas variables son utilizadas para la **conexión a la base de datos** y la **generación/validación de JWT**.

Variable de Entorno	Propósito	Uso en Archivo
PORT	Puerto de escucha del servidor (ej. 8080).	server.js.
SECRETORPRIVATEKEY	Clave secreta para firmar y verificar los JSON Web Tokens (JWT).	generar-jwt.js, validar-jwt.js.
DB_LOCAL_*	Credenciales y configuración para la conexión a la base de datos **local**.	database/connection.js.
DB_REMOTE_*	Credenciales y configuración para la conexión a la base de datos **en la nube** (actualmente comentada en server.js).	

2. BASE DE DATOS Y MODELOS (SEQUELIZE)

2.1 Conexión a Base de Datos

El archivo database/connection.js define dos instancias de conexión de Sequelize, aunque solo la conexión **local (bdmysql)** se autentica actualmente en server.js.

- JavaScript

```
//
database/connection.jsexport const bdmysql = new Sequelize(    process.env.DB_LOCAL_NAME,
process.env.DB_LOCAL_USER,    process.env.DB_LOCAL_PASSWORD,    // ... host, port, dialect ... );
// ...
```

- **Conexión Activa:** bdmysql (Base de datos local).

- **Convenciones Sequelize:** Todos los modelos usan freezeTableName: true, createdAt: false y updatedAt: false.

2.2 Esquema de Modelos y Relaciones

Modelo	Tabla (_ds en BD)	Campos Clave / Tipo de Datos	Relaciones / Notas
Usuarios	usuarios_ds	id (PK, AI), correo (UNIQUE), password (HASH), rol (ENUM: 'ADMIN_ROLE', 'USER_ROLE').	Gestión de autenticación.
Heroes	heroes_ds	id (PK), nombre, bio, casa, aparicion.	Entidad principal.
Peliculas	peliculas_ds	id (PK, AI), nombre.	Entidad principal.
Multimedias	multimedias_ds	idmultimedia (PK, AI), nombre, url, tipo.	Recursos (Imágenes, Videos).
Protagonistas	protagonistas_ds	heroes_id (FK), peliculas_id (FK), papel, fecha_participacion.	Tabla pivote N:M (Héroe - Película).
MultimediasHeroes	multimedias_heroes_ds	heroes_id (FK), idmultimedia (FK).	Tabla pivote N:M (Héroe - Multimedia).

3. DOCUMENTACIÓN DE API (ENDPOINTS Y VALIDACIONES)

Todos los *endpoints* se sirven bajo el prefijo **/api**.

3.1. Rutas de Usuarios (/api/usuarios)

Método	Path	Seguridad	Validaciones de Entrada (express-validator + Helpers)
POST	/ (Registro)	**Ninguna**	nombre (Obligatorio); password (Mínimo 6 chars); correo (Formato email + custom(existeEmail)); rol (isIn('ADMIN_ROLE', 'USER_ROLE')).
POST	/login	**Ninguna**	correo (Formato email + custom(noExisteEmail)); password (Obligatoria + Mínimo 6 chars).
GET	/	**validarJWT + esAdminRole**	Ninguna (Solo Auth/Auth)

3.2. Rutas CRUD Generales

Esta tabla describe las entidades de la API, los prefijos de las rutas, las operaciones permitidas y las **validaciones/middlewares** implementados en el orden de ejecución.

Entidad	Prefijo API	Método	Path	Operación	Validaciones (Ruta)	Lógica Adicional (Controller)
Héroes	/api/heroes	**GET**	/, /:id, /como/:termino	Listar, Detalle, Búsqueda	Ninguna.	CRUD estándar de Sequelize. La Búsqueda (/como/:termino) utiliza un operador LIKE.
Héroes	/api/heroes	**POST**	/	Crear	1. **validarJWT** (Requiere token) 2. **esAdminRole** (Requiere ADMIN_ROLE) 3. **check** (Validación de campos body) 4. **validarCampos**	CRUD estándar de Sequelize.
Héroes	/api/heroes	**PUT/DELETE**	/:id	Actualizar, Eliminar	1. **validarJWT** (Requiere token) 2. **esAdminRole** (Requiere ADMIN_ROLE) 3. **check('id')** (Validación de ID) 4. **validarCampos**	CRUD estándar de Sequelize.
Películas	/api/peliculas	**GET**	/, /:id	Listar, Detalle	Ninguna.	CRUD estándar de Sequelize.
Películas	/api/peliculas	**POST**	/	Crear	1. **validarJWT** (Requiere token) 2. **esAdminRole** (Requiere	Valida que el **nombre no**

					ADMIN_ROLE) 3. **check('nombre')** (Validación de nombre) 4. **validarCampos**	**exista** antes de crear.
Películas	/api/peliculas	**PUT/DELETE**	/:id	Actualizar, Eliminar	1. **validarJWT** (Requiere token) 2. **esAdminRole** (Requiere ADMIN_ROLE) 3. **check('id')** (Validación de ID) 4. **validarCampos**	CRUD estándar de Sequelize.
Protagonistas	/api/protagonistas	**GET**	/, /:id	Listar, Detalle	Ninguna.	CRUD estándar de Sequelize para tabla pivote.
Protagonistas	/api/protagonistas	**POST**	/	Crear	1. **validarJWT** (Requiere token) 2. **check** (Validación de campos body: IDs) 3. **validarCampos**	CRUD estándar de Sequelize para tabla pivote.
Protagonistas	/api/protagonistas	**PUT/DELETE**	/:id	Actualizar, Eliminar	1. **validarJWT** (Requiere token) 2. **check('id')** (Validación de ID) 3. **validarCampos**	CRUD estándar de Sequelize para tabla pivote.
Multimedias	/api/multimedias	**GET**	/, /:id	Listar, Detalle	Ninguna.	CRUD estándar de Sequelize.
Multimedias	/api/multimedias	**POST**	/	Crear	1. **validarJWT** (Requiere token) 2. **check** (Validación de campos body) 3. **validarCampos**	CRUD estándar de Sequelize.
Multimedias	/api/multimedias	**PUT/DELETE**	/:id	Actualizar, Eliminar	1. **validarJWT** (Requiere token) 2. **check('id')** (Validación de ID) 3. **validarCampos**	CRUD estándar de Sequelize.
MultimediasHeroes	/api/multimediasHeroes	**GET**	/, /:id	Listar, Detalle	Ninguna.	CRUD estándar de Sequelize para tabla pivote.
MultimediasHeroes	/api/multimediasHeroes	**POST**	/	Crear	1. **validarJWT** (Requiere token) 2. **check** (Validación de campos body: IDs) 3. **validarCampos**	CRUD estándar de Sequelize para tabla pivote.
MultimediasHeroes	/api/multimediasHeroes	**PUT/DELETE**	/:id	Actualizar, Eliminar	1. **validarJWT** (Requiere token) 2. **check('id')** (Validación de ID) 3. **validarCampos**	CRUD estándar de Sequelize para tabla pivote.
Usuarios	/api/usuarios	**POST**	/	Crear Usuario	1. **check** (nombre, password, rol) 2. **check('correo').custom(existeEmail)** (Valida correo único) 3. **validarCampos**	Encriptación de contraseña. Generación y retorno de **JWT**.
Usuarios	/api/usuarios	**POST**	/login	Iniciar Sesión	1. **check** (correo, password, longitud) 2. **check('correo').custom(noExisteEmail)** (Valida que el correo exista) 3. **validarCampos**	**Autenticación**: Verifica estado y contraseña. Genera y retorna **JWT**.
Usuarios	/api/usuarios	**GET**	/	Listar Usuarios	1. **validarJWT** (Requiere token) 2. **esAdminRole** (Requiere ADMIN_ROLE)	Listado estándar de Sequelize.

3.3. Autenticación y Autorización

Este documento describe los módulos de seguridad (JWT y Roles) y las validaciones implementadas para la gestión y autenticación de usuarios en la API.

3.3.1 Middlewares de Seguridad y Autorización

Los middlewares son funciones que se ejecutan secuencialmente antes de que un controlador maneje la petición, permitiendo verificar credenciales y permisos.

validarJWT (../middlewares/validar-jwt.js)

Propósito: Verificar la autenticidad del usuario a través de un **JSON Web Token (JWT)**.

Header Requerido	Descripción
x-token	El JWT generado tras un inicio de sesión exitoso. Debe enviarse en los headers de la petición.

Flujo:

1. Verifica la existencia del token en el header x-token.
2. Decodifica y valida el token usando la clave secreta (SECRETORPRIVATEKEY).
3. Utiliza el uid (ID de usuario) del payload del token para buscar el usuario en la base de datos.
4. Verifica que el usuario exista y que su estado sea true.
5. Si es exitoso, adjunta el objeto del usuario a la petición: **req.usuario**.

3.3.2. esAdminRole y tieneRole (../middlewares/validar-roles.js)

Estos middlewares se utilizan para la **Autorización** (verificar si el usuario tiene permiso para realizar una acción).

Middleware	Propósito	Dependencia
esAdminRole	Restringe el acceso únicamente a usuarios con el rol **ADMIN_ROLE**.	Requiere que validarJWT se haya ejecutado previamente para asegurar que req.usuario exista.
tieneRole(...roles)	Genera un middleware que restringe el acceso a usuarios que posean **al menos uno** de los roles pasados como argumento (ej: tieneRole('ADMIN_ROLE', 'USER_ROLE')).	Requiere que validarJWT se haya ejecutado previamente.

Rutas y Endpoints de Usuarios

Las rutas de usuarios (../routes/usuarios.route.js) definen los endpoints para la creación y autenticación.

3.3.4. Registro de Usuario (POST /api/usuarios)

Método	URL	Función	Descripción
POST	/api/usuarios	usuariosPost	Crea un nuevo usuario en la base de datos.

Validaciones Requeridas (en orden):

1. **nombre**: Obligatorio.
2. **password**: Obligatorio y mínimo de 6 caracteres. (Se hashea con bcryptjs antes de guardar).
3. **correo**: Obligatorio, formato de email válido y debe ser **único** (usa el helper existeEmail).
4. **rol**: Debe ser uno de los roles permitidos (ej: ADMIN_ROLE, USER_ROLE).
5. **validarCampos**: Recolecta y devuelve todos los errores de validación de express-validator si los hay.

3.3.5. Inicio de Sesio´n (POST /api/usuarios/login)

Método	URL	Función	Descripción
POST	/api/usuarios/login	login	Autentica al usuario y genera un JWT.

Validaciones Requeridas:

1. **correo**: Obligatorio, formato de email válido y debe **existir** en BD (usa el helper noExisteEmail).
2. **password**: Obligatorio y mínimo de 6 caracteres.

Respuesta Exitosa: Si el correo y la contraseña son correctos, el endpoint retorna el objeto del usuario y el JWT:

```
{
  "ok": true,
  "msg": "Login OK",
  "usuario": {
     "id": 1,
     "nombre": "Jane Doe",
     "correo": "jane.doe@test.com",
     "rol": "USER_ROLE",
     // ... otros campos
  },
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOjEsImlhdCI6MTYzNjUwNTY3MywiZXhwIjoxNjM2NTIwMDczfQ.exampleToken"
```

}

Funciones Auxiliares (Helpers)

3.3.6. generarJWT (../helpers/generar-jwt.js)

Función asíncrona que crea un nuevo JWT para el usuario especificado.

Parámetro	Descripción
uid	El ID del usuario que se incluirá en el payload del token.

El token generado tiene una caducidad de 4 **horas**.

2. Validadores de Base de Datos (../helpers/db-validators.js)

Función	Propósito	Uso en Ruta
existeEmail(correo)	Lanza un error si el correo ya existe. Garantiza que no se registren correos duplicados.	POST /api/usuarios (Registro)
noExisteEmail(correo)	Lanza un error si el correo no existe. Garantiza que solo se intente iniciar sesión con usuarios registrados.	POST /api/usuarios/login (Login)

Ejemplo de Aplicacio´n de Middlewares

Para proteger el endpoint de listado de usuarios (solo para Administradores):

```
// En /routes/usuarios.route.js

router.get('/',

   validarJWT,   // 1. Verifica que el usuario esté autenticado.

   esAdminRole,  // 2. Verifica que el usuario autenticado sea ADMIN_ROLE.

   usuariosGet   // 3. Si pasa, se ejecuta el controlador.

);
```

Controladores (Controllers)

Se añadieron y modificaron las siguientes funciones en ../controllers/usuarios.controller.js:

Función	Descripción
usuariosPost	Recibe datos del cuerpo, hashea la contraseña usando bcryptjs y guarda el nuevo usuario en la BD.
login	Busca el usuario por correo, compara la contraseña (compareSync), verifica el estado del usuario y genera un JWT si las credenciales son correctas.
usuariosGet	Lista todos los usuarios (rutas protegidas con validarJWT y esAdminRole).

4. USO DE APIREST HEROES / Ejemplo de uso

creación de un nuevo usuario (POST /api/usuarios) como ejemplo, ya que incluye varias validaciones:

1. Validación de campos obligatorios (nombre, password, correo).
2. Validación de longitud de password.
3. Validación de formato de correo.
4. Validación personalizada de correo existente (existeEmail).
5. Validación de rol permitido (rol).

Guía para usar Thunder Client: POST /api/usuarios

1. Iniciar Thunder Client

1. Abre Visual Studio Code.
2. Haz clic en el icono de **Thunder Client** en la barra lateral izquierda (o presiona Ctrl+Alt+Z).
3. Haz clic en el botón **"New Request"** para crear una nueva solicitud.

2. Configurar la Solicitud (Request)

A. Método y URL

1. **Método:** Selecciona **POST** en el menú desplegable.
2. **URL:** Ingresa la URL base de tu API y el *endpoint* para crear usuarios. Asumiremos que tu servidor se ejecuta en el puerto $3000 (el puerto predeterminado en el archivo server.js es process.env.PORT):
3. http://localhost:3000/api/usuarios

B. Cuerpo de la Solicitud (Body)

1. Ve a la pestaña **Body**.
2. Selecciona el tipo **JSON**.
3. Ingresa el cuerpo de la solicitud con los datos que deseas enviar. **Asegúrate de incluir todos los campos requeridos** según tu archivo usuarios.route.js: nombre, password, correo, img, y rol.

Ejemplo de Petición Exitosa:

JSON

{ "nombre": "Tony Stark", "correo": "tony.stark@starkindustries.com", "password": "unpasswordseguro123", "img": "no-image.jpg", "rol": "USER_ROLE", "google": false }

3. Ejecución y Validación

Ahora probaremos el *endpoint* y cómo responde a las diferentes validaciones implementadas en tus *middlewares* (validar-campos.js, db-validators.js y usuarios.route.js).

Caso 1: Petición Exitosa (Status 200/201)

1. Utiliza el **Ejemplo de Petición Exitosa** de arriba.
2. Haz clic en **"Send"**.
3. **Resultado Esperado:**
    - **Status:** 200 OK (o 201 Created, si tu controlador lo retorna así, pero tu controlador usuariosPost parece retornar 200 OK por defecto).
    - **Body:** Un objeto JSON que confirma la creación del usuario, posiblemente mostrando el objeto del usuario y ok: true.
    -

Caso 2: Validación de Campos Obligatorios (validarCampos - Status 400)

Si omites un campo obligatorio (como nombre o password), el *middleware* validarCampos que utiliza express-validator interceptará la solicitud.

1. **Modifica el Body:** Elimina la línea nombre.
2. JSON
3. { "correo": "fallo@ejemplo.com", "password": "password123", "img": "no-image.jpg", "rol": "USER_ROLE", "google": false }
4. Haz clic en **"Send"**.
5. **Resultado Esperado:**
    - **Status:** 400 Bad Request.
    - **Body:** Un objeto JSON de express-validator detallando los errores, incluyendo el mensaje que definiste en usuarios.route.js (e.g., "El nombre es obligatorio").

Caso 3: Validación de Formato de Correo (isEmail - Status 400)

Si el formato del correo es incorrecto.

1. **Modifica el Body:** Usa un correo inválido.
2. JSON
3. { "nombre": "Fallo Email", "correo": "correoinvalido", "password": "password123", "img": "no-image.jpg", "rol": "USER_ROLE", "google": false }
4. Haz clic en **"Send"**.
5. **Resultado Esperado:**
    - **Status:** 400 Bad Request.
    - **Body:** El error de validación para correo (e.g., "El correo es obligatorio", aunque debería ser más específico al usar isEmail()).

Caso 4: Validación de Correo Existente (existeEmail - Status 400)
Esta validación personalizada se encuentra en db-validators.js y usuarios.route.js. Para probarla, primero debes crear un usuario exitosamente (Caso 1), y luego intentar crearlo de nuevo con el **mismo correo**.

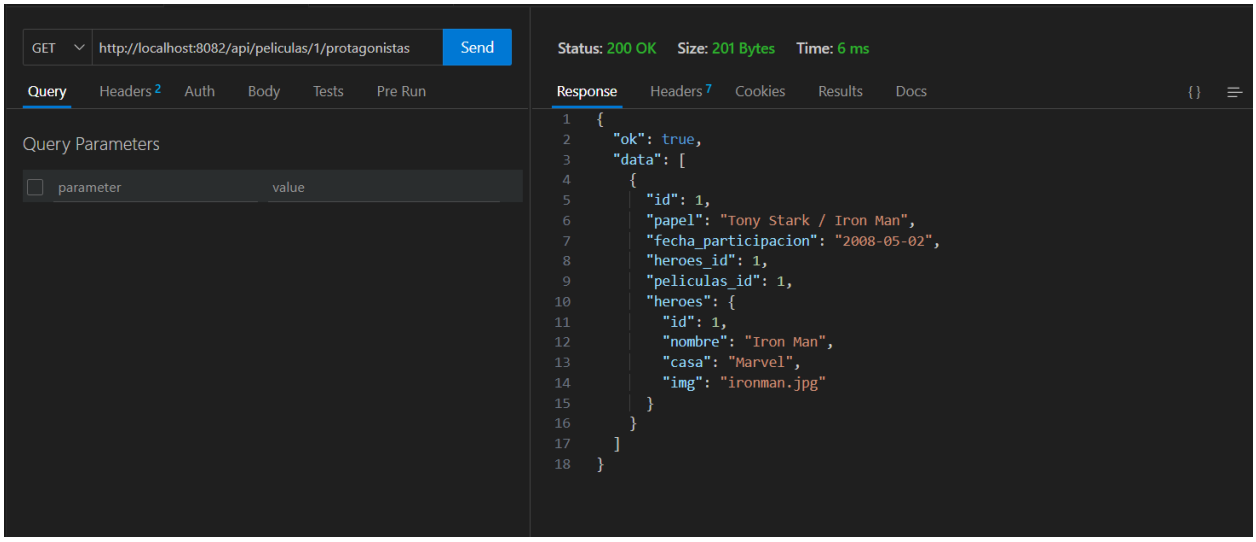
1. **Modifica el Body:** Usa el correo que acabas de registrar en el Caso 1 (tony.stark@starkindustries.com).
2. JSON
3. { "nombre": "Otro Usuario", "correo": "tony.stark@starkindustries.com", // El correo ya existe en la BD "password": "otrapassword", "img": "no-image.jpg", "rol": "USER_ROLE", "google": false }
4. Haz clic en **"Send"**.
5. **Resultado Esperado:**
    - **Status:** 400 Bad Request.
    - **Body:** El mensaje de error personalizado que configuraste en db-validators.js: "El email tony.stark@starkindustries.com ya existe en la Base de Datos...".

Caso 5: Validación de Rol (Role) (isIn - Status 400)
Si intentas asignar un rol que no está definido en tu ruta (ADMIN_ROLE o USER_ROLE).

1. **Modifica el Body:** Usa un rol inválido.
2. JSON
3. { "nombre": "Goku", "correo": "goku@saiyan.com", "password": "password123456", "img": "no-image.jpg", "rol": "GOD_ROLE", // Rol no permitido "google": false }
4. Haz clic en **"Send"**.
5. **Resultado Esperado:**
    - **Status:** 400 Bad Request.
    - **Body:** El error de validación para rol con el mensaje: "No es un rol valido".

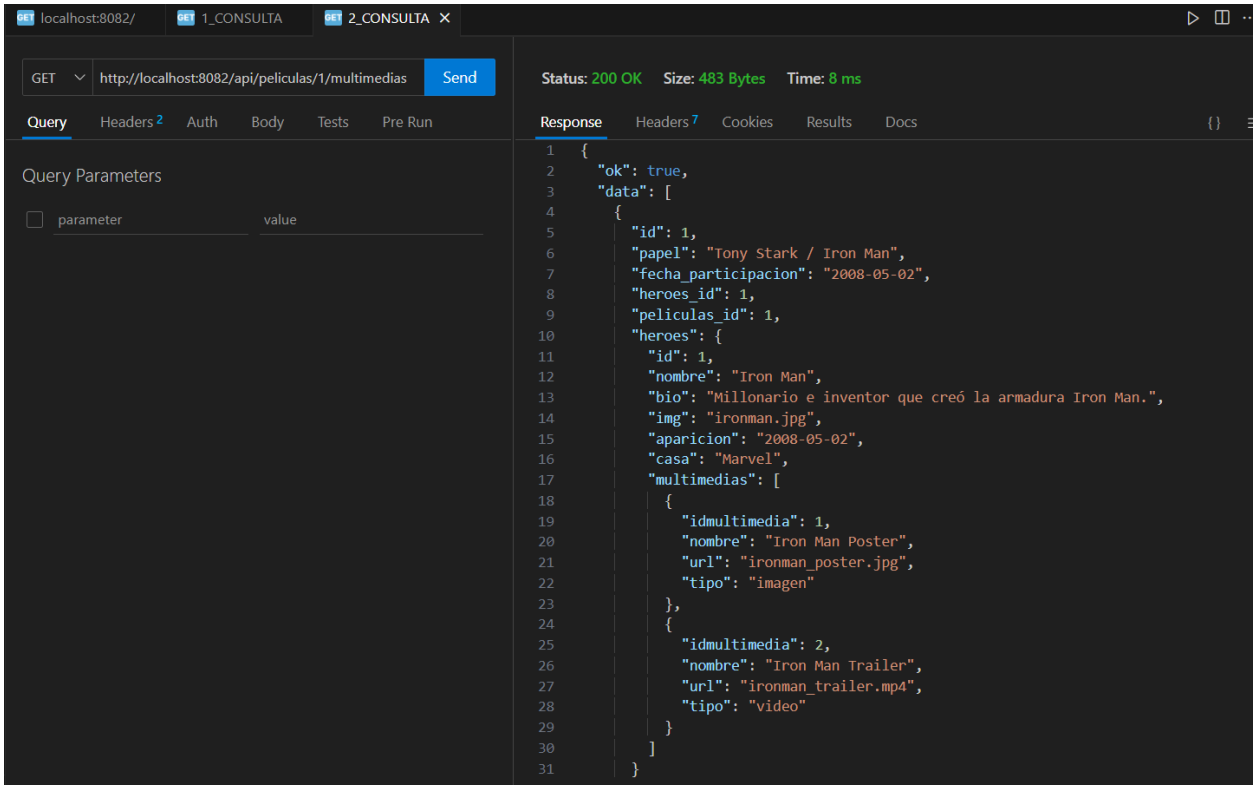
5. EVIDENCIAS

1. Por la película conocer cuáles son los protagonistas de esta, y el papel desempeñado en ella.



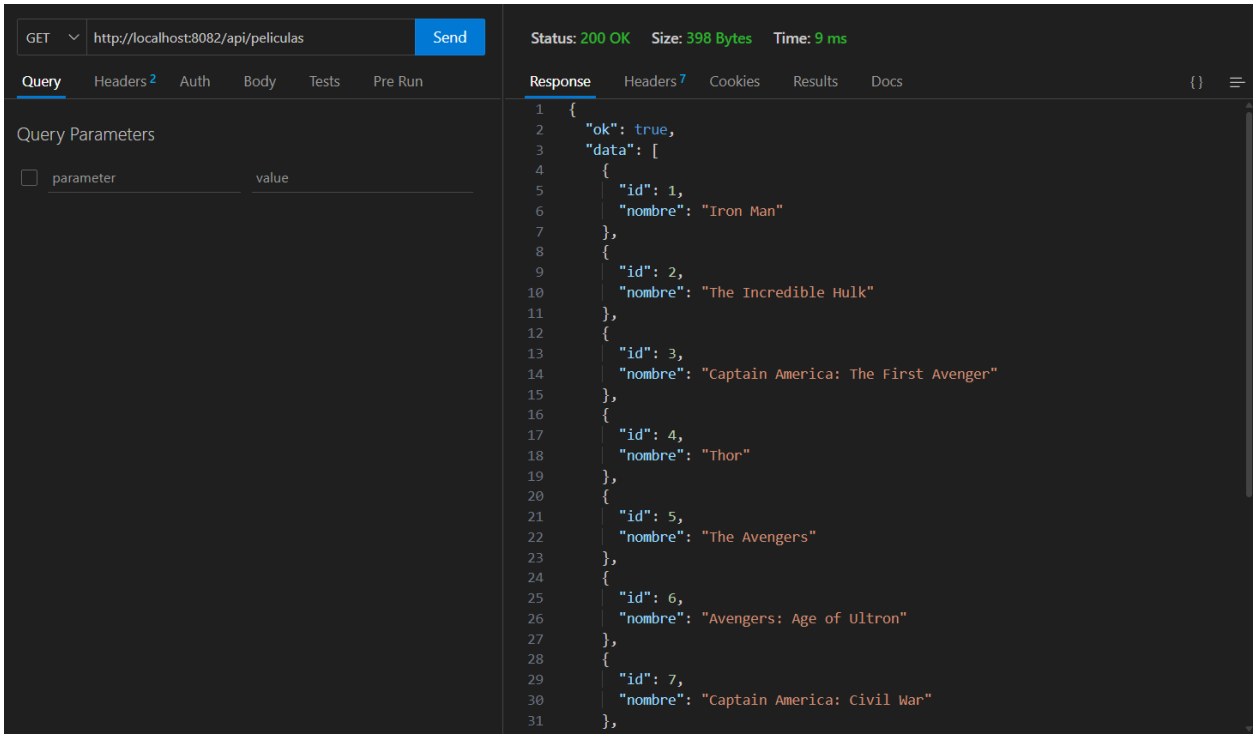
https://localhost:8082/api/peliculas/"#id"/protagonistas

2. Por película conocer cuáles son los elementos multimedia que tiene la película a través de la asignación de estos a través del héroe.

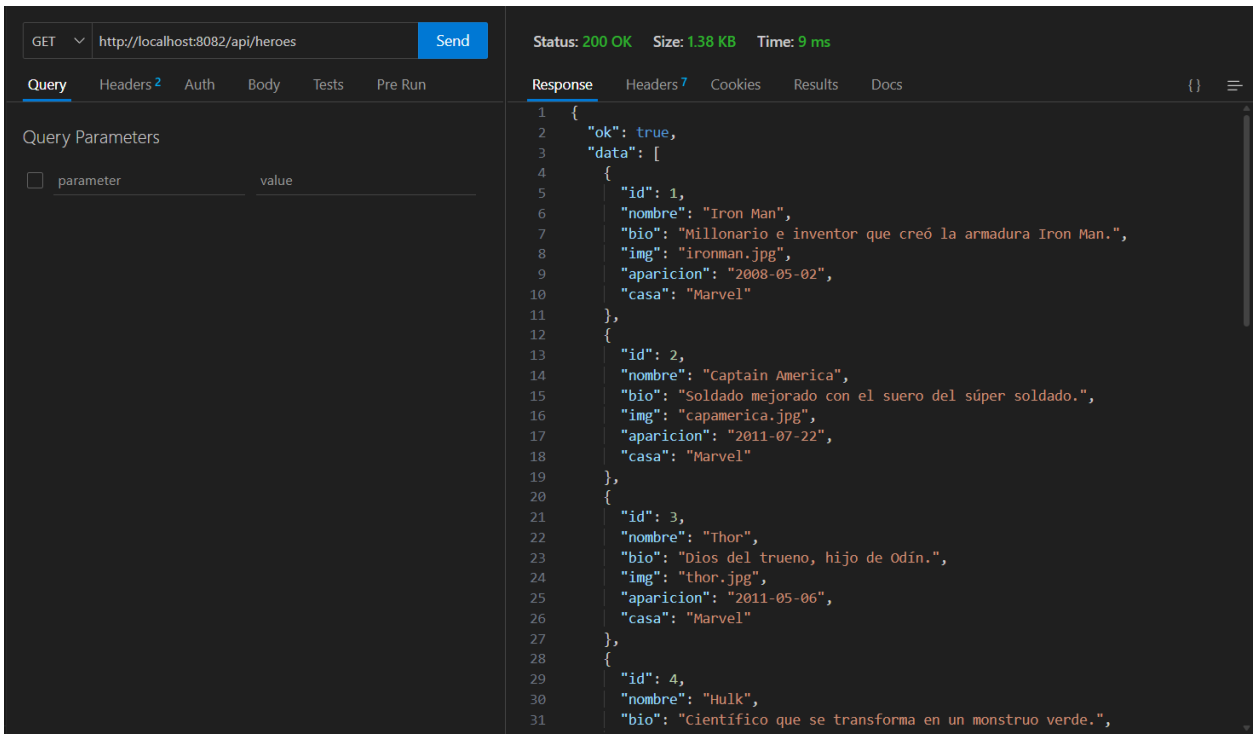


https://localhost:8082/api/peliculas/"#id"/multimedias

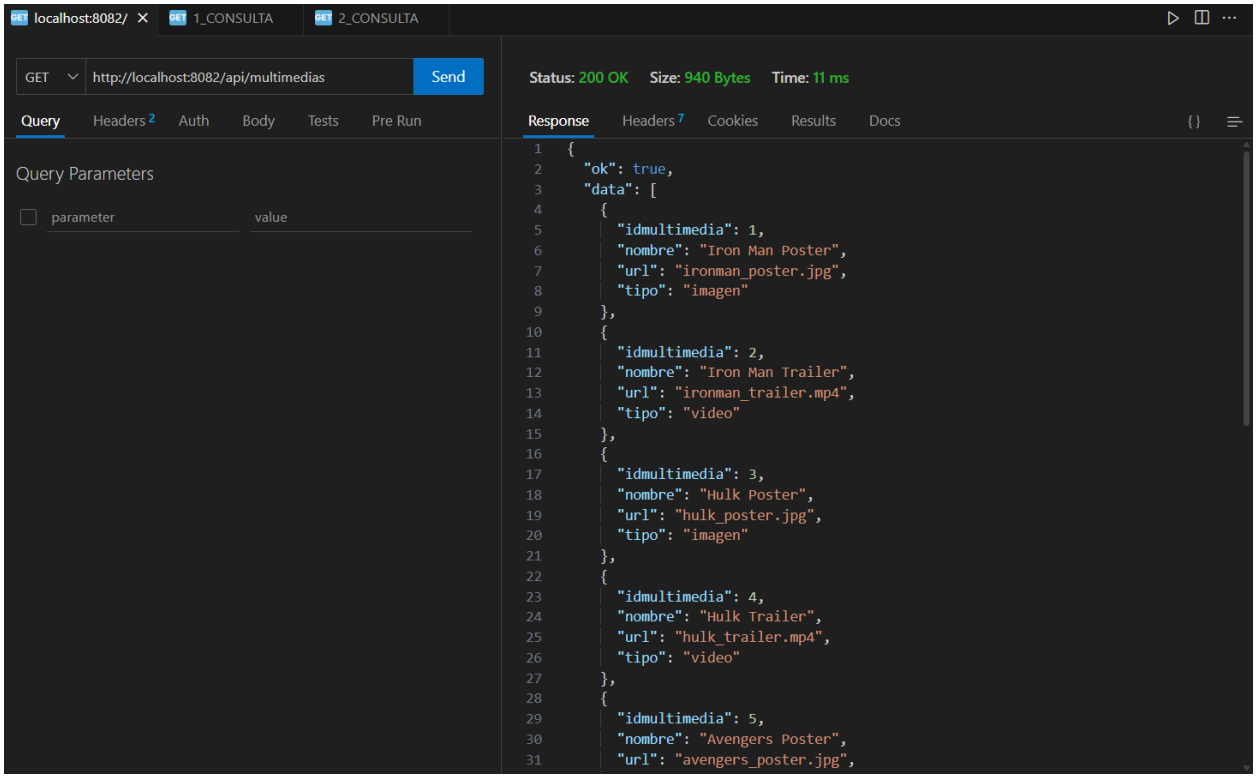
3. Películas



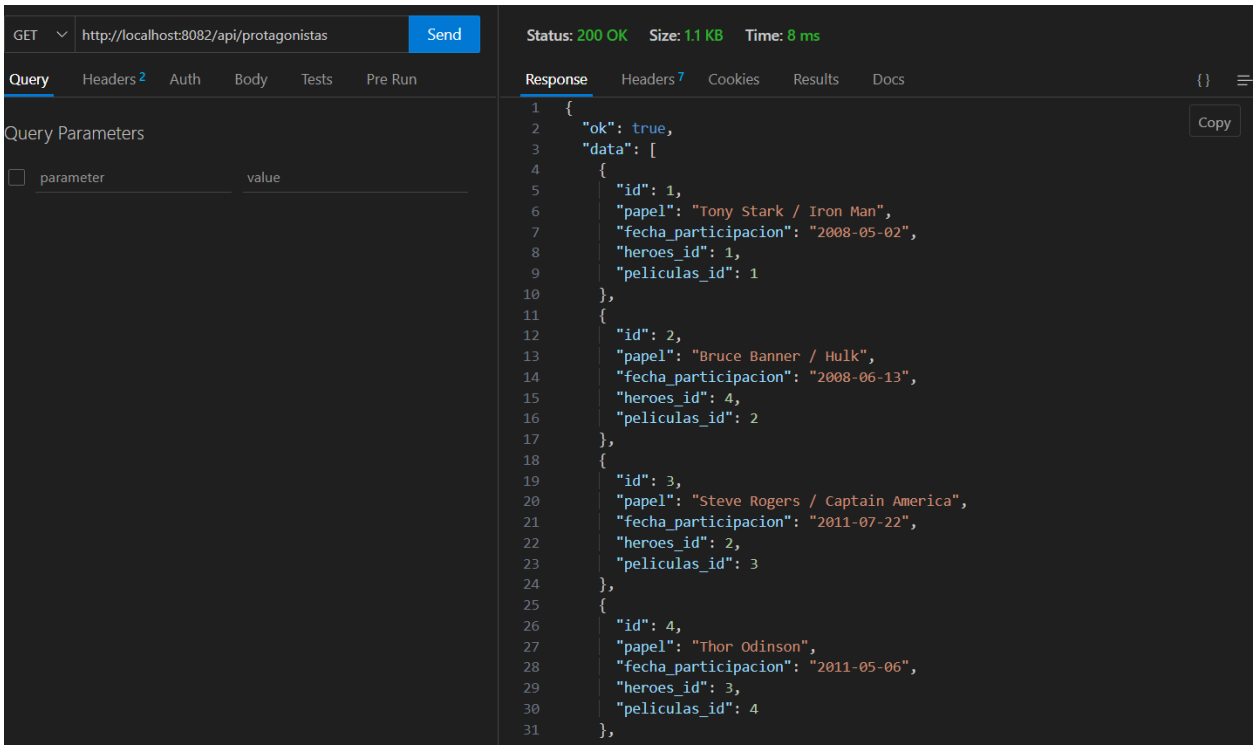
4. Héroes



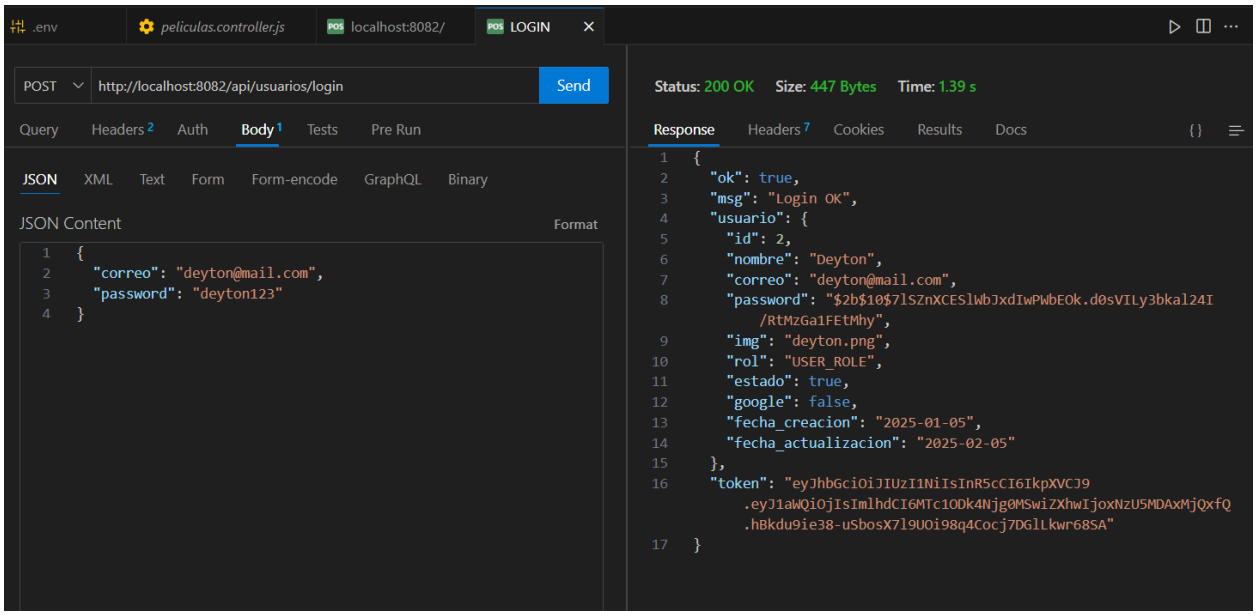
5. Multimedia



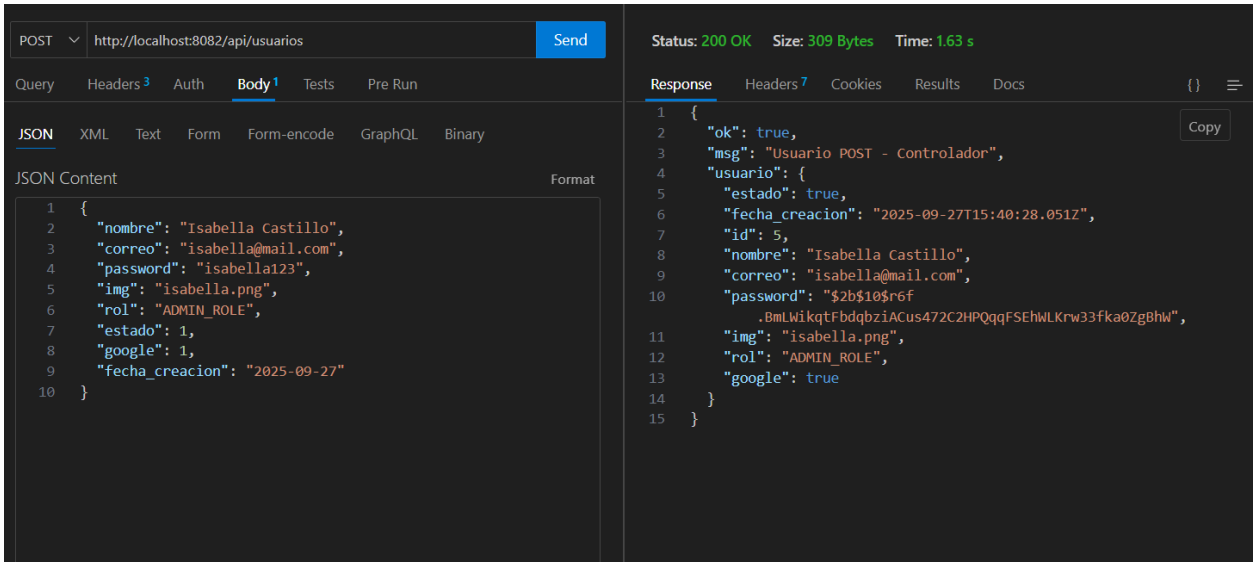
6. Protagonistas



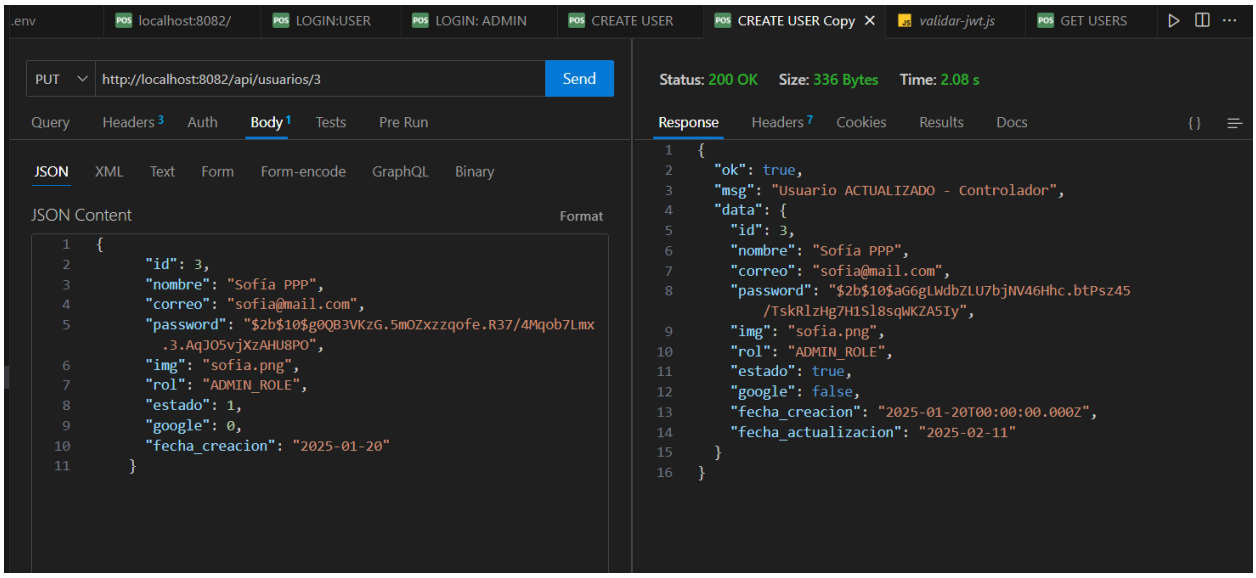
7. Login



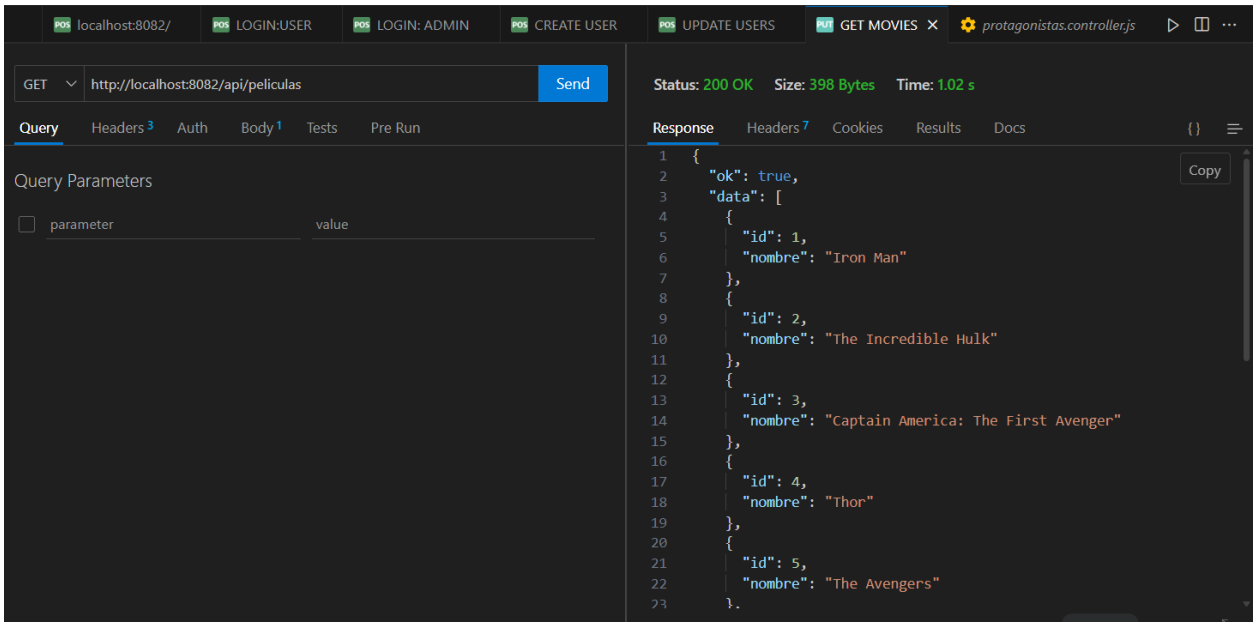
8. Creación de usuario



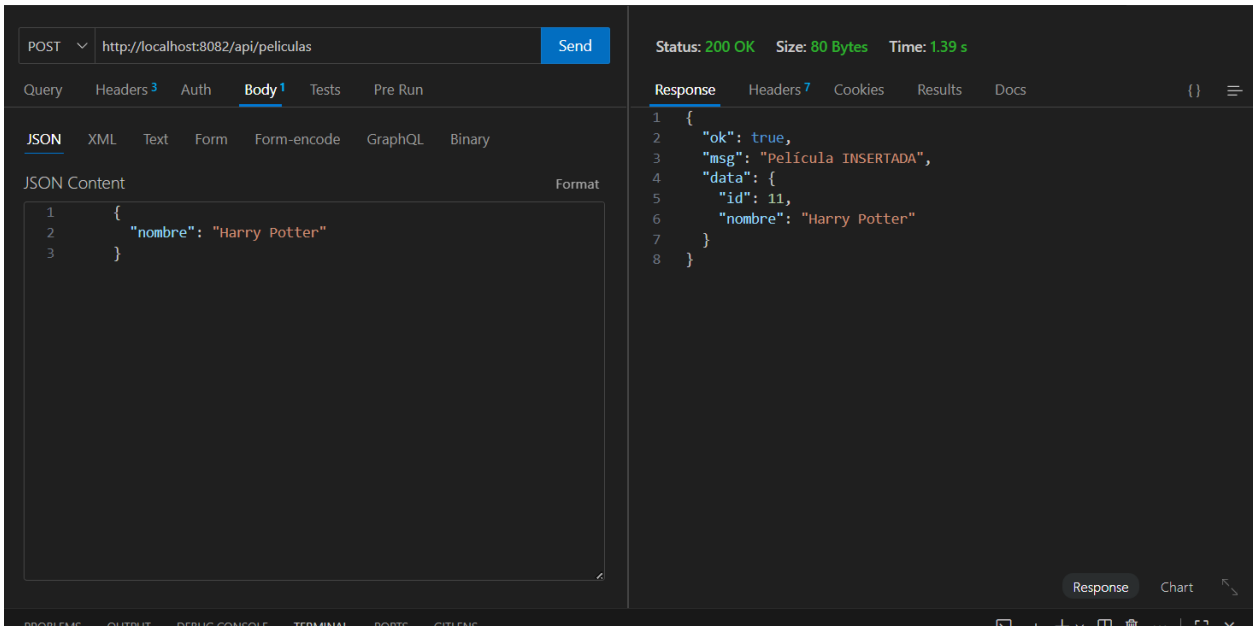
9. Actualizar usuario



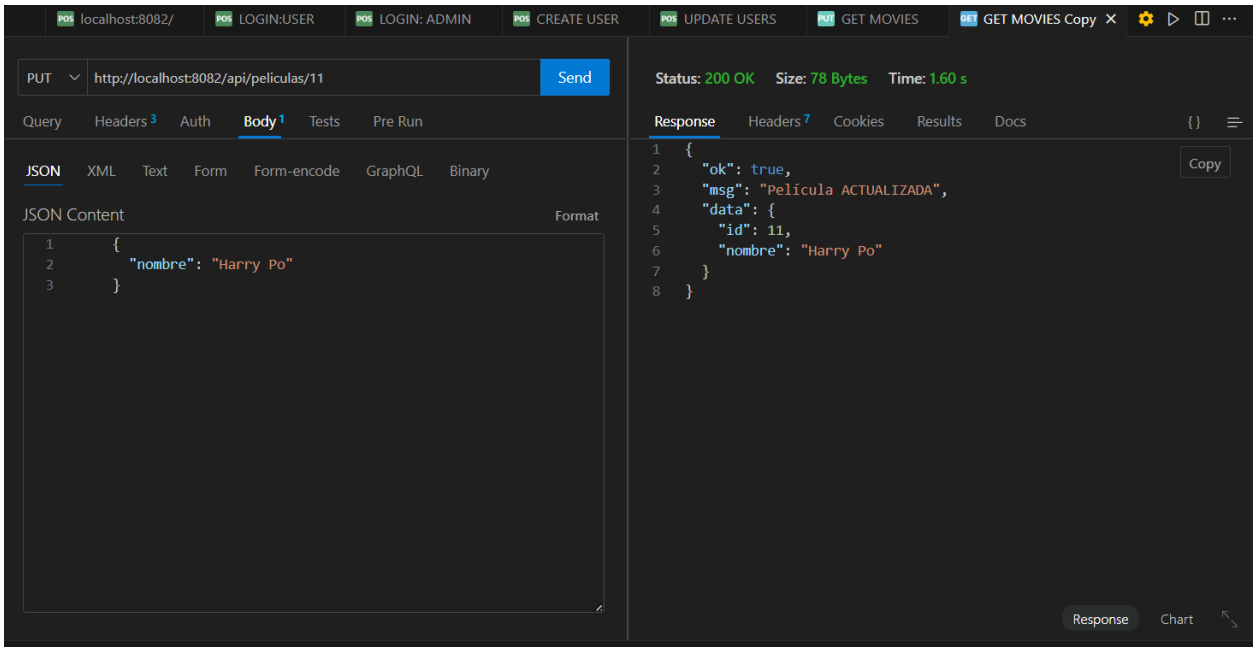
10. Get películas



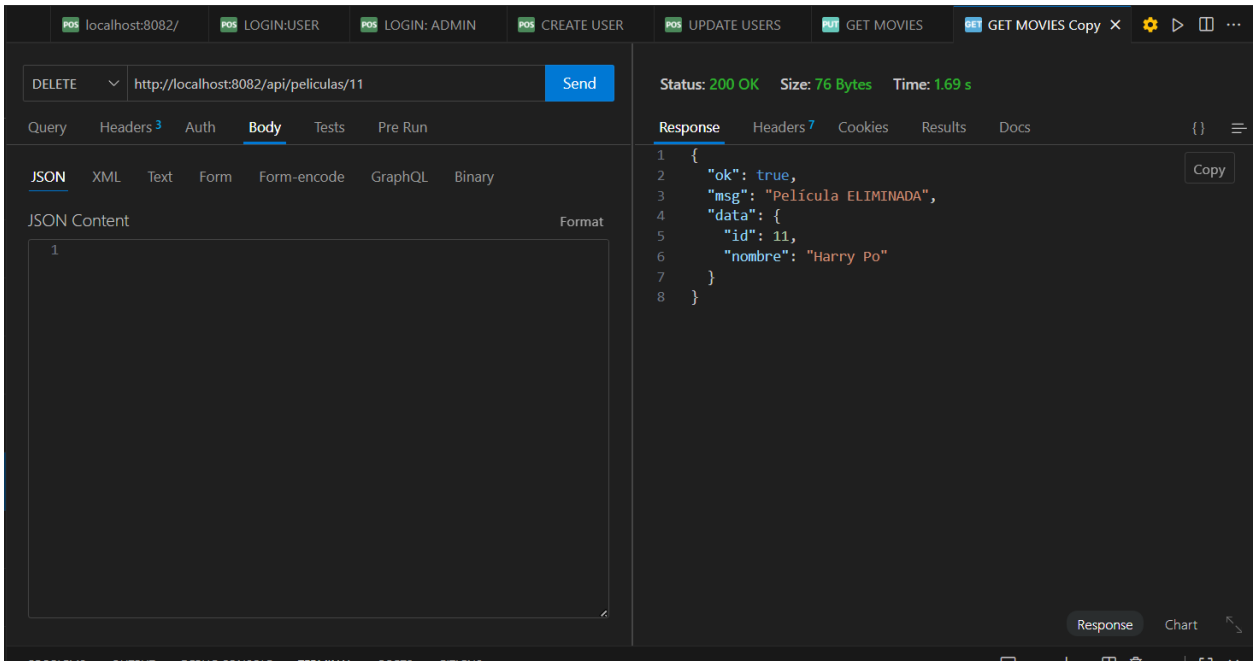
11. Creación de películas



12. Actualizar película



13. Eliminación películas


localhost:8082/
LOGIN:USER
LOGIN: ADMIN
CREATE USER
UPDATE USERS
GET MOVIES
GET MOVIES Copy
DELETE
http://localhost:8082/api/peliculas/11
Send
Query
Headers 3
Auth
Body
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
1
Status: 200 OK
Size: 76 Bytes
Time: 1.69 s
Response
Headers 7
Cookies
Results
Docs
Copy
{
"ok": true,
"msg": "Película ELIMINADA",
"data": {
"id": 11,
"nombre": "Harry Po"
}
}
Response
Chart