

# Potenciando GitHub-GQL con Neo4j

Javier Fernández Castillo, Manuel Otero Barbasán

25 de junio de 2024

## Resumen

Este documento describe el desarrollo y la implementación de una API REST que interactúa con la base de datos Neo4j y la API de GraphQL de GitHub para extraer, almacenar y procesar datos de GitHub. La API permite una interacción robusta y eficiente, facilitando la extracción de datos de usuarios y repositorios a través de diversos endpoints POST y GET. Estos endpoints no solo permiten la inserción de datos en la base de datos sino que también facilitan la recuperación y el análisis avanzado de la información. Se describen las funciones específicas utilizadas para la extracción de datos, la población de la base de datos, y el procesamiento avanzado de la información para realizar análisis detallados. El documento también incluye una guía de instalación y configuración del sistema, así como ejemplos prácticos de uso de la API para facilitar la interacción con los datos almacenados. Obtenemos como resultado un sistema capaz de extraer información procesada que no podría obtenerse de una forma directa a través de la API de GraphQL de GitHub.

**Palabras clave:** API REST, Neo4j, GraphQL de GitHub, Extracción de datos, Población de base de datos, Análisis de datos, Interacción con base de datos.

## 1. Introducción

### 1.1. Contexto del proyecto

En el contexto actual, el desarrollo de aplicaciones web y servicios de API se ha vuelto fundamental para la interacción entre sistemas y el manejo de datos. Con la creciente popularidad de las plataformas de desarrollo colaborativo como GitHub, la necesidad de gestionar y analizar datos de manera eficiente se ha convertido en una prioridad para muchas organizaciones y desarrolladores.

El proyecto que presentamos se centra en la creación de una API utilizando ExpressJS, Node.js y Neo4j, junto con la integración de la API de GraphQL de GitHub. La finalidad es extraer, procesar y almacenar información relevante sobre perfiles de usuarios, repositorios, issues, seguidores, colaboradores y otras entidades relacionadas con proyectos alojados en GitHub.

GitHub proporciona una API de GraphQL que permite acceder a datos detallados sobre los repositorios, usuarios y actividades en la plataforma. Sin embargo, a veces es necesario realizar procesamiento adicional sobre estos datos para obtener información específica o realizar análisis complejos que no son directamente proporcionados por la API de GitHub.

El objetivo principal de nuestro proyecto es ofrecer una interfaz de programación que permita a los usuarios acceder a información detallada sobre los proyectos y perfiles de GitHub, así como realizar consultas avanzadas y análisis sobre estos datos. Además, la integración con Neo4j como base de datos permite almacenar relaciones entre entidades y realizar consultas de manera eficiente, lo que facilita la recuperación y el análisis de datos relacionales.

### 1.2. Objetivos

El proyecto tiene como objetivo principal adquirir conocimientos y habilidades en diversas tecnologías y conceptos, así como desarrollar una API robusta y funcional que permita interactuar con la API de GraphQL de GitHub y la base de datos Neo4j. Los objetivos específicos del proyecto son los siguientes:

- **Conocer el funcionamiento de Neo4j:** Entender los fundamentos de las bases de datos de grafos y aprender a utilizar Neo4j para almacenar y consultar datos relacionales de manera eficiente.
- **Aprender el funcionamiento de una API de GraphQL:** Familiarizarse con el modelo de datos de GraphQL y comprender cómo interactuar con la API de GitHub para obtener datos detallados sobre perfiles de usuarios, repositorios, issues, etc.
- **Desarrollar endpoints de una API con Express y Node:** Implementar endpoints de tipo POST y GET utilizando ExpressJS y Node.js para poblar la base de datos con información de GitHub y permitir consultas avanzadas sobre estos datos.
- **Utilizar Swagger como interfaz de llamadas a la API:** Utilizar Swagger para documentar y diseñar la API, proporcionando una interfaz clara y fácil de usar para los usuarios que deseen interactuar con ella.
- **Aprovechar el enfoque de grafos de Neo4j para procesar datos extraídos de una API relacional como GraphQL:** Aprovechar las capacidades de Neo4j para almacenar y procesar datos relacionales de forma similar a como lo hace GraphQL, permitiendo consultas flexibles y eficientes sobre la información almacenada.

Estos objetivos se centran en el aprendizaje y la aplicación práctica de tecnologías clave, así como en el desarrollo de una solución que combine de manera efectiva las funcionalidades de Neo4j, GraphQL y ExpressJS para proporcionar una API útil y versátil para el análisis de datos de GitHub.

### 1.3. Alcance

El proyecto abarca desde el diseño y desarrollo de una API con ExpressJS y Node.js, hasta la integración con la base de datos Neo4j y la API de GraphQL de GitHub. El alcance del proyecto incluye lo siguiente:

- **Diseño y desarrollo de la API:** Creación de endpoints con ExpressJS para interactuar con la API de GraphQL de GitHub y gestionar las consultas de los usuarios. Se desarrollarán endpoints de tipo POST para poblar la base de datos y endpoints de tipo GET para realizar consultas avanzadas.
- **Integración con Neo4j:** Conexión y manipulación de la base de datos Neo4j para almacenar y recuperar datos relacionados con perfiles de usuarios, repositorios, issues, etc. La integración incluye la creación de nodos y relaciones de acuerdo con la estructura de datos obtenida de la API de GraphQL de GitHub.
- **Utilización de Swagger para documentación:** Empleo de Swagger para documentar la API, proporcionando descripciones claras de cada endpoint, los parámetros esperados y las respuestas correspondientes. Esto permitirá a los usuarios comprender fácilmente cómo utilizar la API.
- **Desarrollo de funcionalidades avanzadas:** Implementación de funcionalidades que permitan a los usuarios realizar consultas complejas sobre los datos almacenados en Neo4j.
- **Documentación y presentación del proyecto:** Elaboración de una memoria detallada que documente el diseño, la implementación, el uso y las conclusiones del proyecto. Además, se incluirán manuales de usuario y de instalación.
- **Entrega del código en un repositorio GitHub:** Subida del código fuente del proyecto a un repositorio GitHub público.

El alcance del proyecto se centra en desarrollar una API funcional que permita acceder y analizar datos de GitHub de manera eficiente, cumpliendo con los objetivos establecidos y los requisitos de la institución educativa. Además, se busca proporcionar una solución que sirva como base para futuras mejoras y extensiones.

## 1.4. Estructura del documento

El documento está estructurado de la siguiente manera:

- **Tecnologías Utilizadas:** En esta sección se describen las tecnologías utilizadas en el proyecto, desde el framework de desarrollo de la API hasta la base de datos y las herramientas de documentación y consulta de la API de GitHub.
- **Diseño y Arquitectura:** Esta sección aborda la arquitectura general del sistema, el diseño de la base de datos en Neo4j y la estructura de la API REST utilizando ExpressJS y Swagger para la documentación.
- **Implementación:** Aquí se detallan los endpoints implementados en la API, el proceso de población de la base de datos con datos de GitHub y el procesamiento avanzado de la información para consultas complejas.
- **Manual de Instalación:** En esta sección se proporciona un manual detallado para la instalación del proyecto, incluyendo los requisitos previos, la instalación de las dependencias necesarias y la configuración del entorno de desarrollo.
- **Manual de Usuario:** aquí se encuentran las instrucciones para el uso de la API, junto con ejemplos de consultas que los usuarios pueden realizar para obtener información específica de GitHub.
- **Validación:** se pone a prueba el sistema, demostrando el correcto funcionamiento con los ejemplos especificados en el manual de usuario.
- **Conclusiones:** Finalmente, se presentan las conclusiones del proyecto, incluyendo los logros alcanzados, las limitaciones identificadas y posibles mejoras, así como el impacto potencial del proyecto en su contexto.

## 2. Tecnologías Utilizadas

### 2.1. ExpressJS

ExpressJS es un framework web minimalista y flexible"[1] (web de ExpressJS) para Node.js que proporciona una capa de abstracción sobre el servidor HTTP de Node.js. Está diseñado para crear aplicaciones web y APIs de manera rápida y sencilla, proporcionando una serie de utilidades y funciones para manejar solicitudes HTTP, definir rutas y gestionar middleware.

- **Definición de rutas y endpoints:** ExpressJS se utiliza para definir las rutas de la API REST. Esto implica especificar cómo responderá la aplicación a las solicitudes HTTP entrantes, ya sea para recuperar, crear, actualizar o eliminar recursos.
- **Gestión de middleware:** ExpressJS ofrece un sistema de middleware que permite ejecutar funciones intermedias antes de que una solicitud alcance su destino final. Esto se usa para realizar tareas como la autenticación, la validación de datos, la compresión de respuestas, entre otros.
- **Manejo de solicitudes y respuestas:** ExpressJS maneja las solicitudes HTTP entrantes y las respuestas asociadas. Esto incluye el análisis de las solicitudes para extraer datos relevantes, como parámetros de ruta o cuerpo de la solicitud, y enviar las respuestas adecuadas, ya sea en formato JSON, HTML u otros.

### 2.2. Node.js

Node.js<sup>1</sup> es un entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome. Permite ejecutar código JavaScript en el lado del servidor, lo que lo hace ideal para construir aplicaciones web escalables y de alto rendimiento. Node.js utiliza un modelo de entrada y salida sin bloqueo y orientado a eventos, lo que lo hace eficiente para aplicaciones de red y E/S intensivas.

- **Motor del servidor:** Node.js actúa como el motor del servidor que ejecuta la lógica de la aplicación y gestiona las operaciones de E/S, como la lectura y escritura de archivos, el acceso a bases de datos, y las solicitudes y respuestas HTTP.
- **Gestión de paquetes y dependencias:** Node.js utiliza el gestor de paquetes npm para instalar, compartir y gestionar dependencias de proyectos. Esto es fundamental para integrar bibliotecas y herramientas adicionales en la aplicación.
- **Conexión con la base de datos:** Node.js facilita la conexión y la comunicación con la base de datos Neo4j, permitiendo realizar consultas y actualizaciones de datos desde la aplicación.

### 2.3. Neo4j

Neo4j<sup>2</sup> es un sistema de gestión de bases de datos de grafo altamente escalable y flexible. Está diseñado específicamente para almacenar y consultar datos relacionales de manera eficiente, lo que lo hace ideal para modelar y analizar redes complejas de datos. Utiliza el lenguaje de consulta Cypher para interactuar con los datos almacenados en forma de grafo.

- **Almacenamiento de datos:** Neo4j se utiliza como la base de datos principal para almacenar la información extraída de la API de GraphQL de GitHub. Los perfiles de usuarios, repositorios, colaboradores, issues y sus relaciones se representan como nodos y relaciones en el grafo de Neo4j.
- **Consultas y análisis:** Neo4j proporciona una interfaz poderosa para realizar consultas complejas sobre datos de grafo. Esto permite realizar análisis avanzados, como encontrar caminos más cortos entre nodos, identificar comunidades de usuarios, y descubrir patrones de colaboración en repositorios.
- **Integración con Node.js:** Neo4j cuenta con un cliente oficial para Node.js que facilita la conexión y la ejecución de consultas desde la aplicación.

---

<sup>1</sup>Enlace a la documentación de Node.js: <https://nodejs.org/docs/latest/api/>

<sup>2</sup>Enlace a la documentación de Neo4j: <https://neo4j.com/docs/getting-started/>

## 2.4. Swagger

Swagger es un conjunto de herramientas para diseñar, construir, documentar y consumir APIs REST de manera fácil y eficiente. Permite describir la estructura de la API en un formato legible por humanos y máquinas, lo que facilita la comprensión y el uso de la API tanto para desarrolladores como para consumidores.

- **Documentación de la API:** Swagger se utiliza para documentar la API REST creada con ExpressJS. Esto incluye descripciones detalladas de los endpoints, parámetros, cuerpos de solicitud y respuestas esperadas.
- **Diseño de la API:** Swagger proporciona herramientas para diseñar la estructura de la API de manera clara y coherente. Esto ayuda a establecer una interfaz consistente y fácil de entender para los usuarios de la API.
- **Pruebas de la API:** Swagger permite probar los endpoints de la API directamente desde la interfaz de documentación, lo que facilita la verificación de su funcionamiento y la validación de los resultados.

## 2.5. API GraphQL de GitHub

GraphQL es un lenguaje de consulta y manipulación de datos desarrollado por Facebook que permite a los clientes solicitar únicamente los datos que necesitan, en la forma exacta en que los necesitan. Proporciona una manera eficiente de interactuar con una API al permitir a los clientes especificar de forma precisa qué datos quieren obtener, evitando así la sobrecarga de datos y minimizando la cantidad de solicitudes.

La API de GitHub ofrece una API GraphQL que proporciona acceso a la mayoría de los datos disponibles en la plataforma de GitHub. Utilizar GraphQL permite obtener información de perfiles de usuarios, repositorios, issues y otras entidades de GitHub de manera más eficiente y flexible que utilizando la API REST tradicional.

- **Extracción de datos:** La API de GraphQL de GitHub se utiliza para extraer información detallada sobre perfiles de usuarios, repositorios, colaboradores, issues y otras entidades relevantes de GitHub.
- **Flexibilidad en las consultas:** GraphQL permite a la aplicación solicitar solo los campos necesarios para cada consulta, lo que reduce el ancho de banda y mejora el rendimiento de la aplicación.
- **Especificación de esquemas:** GitHub proporciona un esquema GraphQL completo que describe los tipos de datos disponibles y las relaciones entre ellos, lo que facilita la comprensión y la construcción de consultas.

# 3. Diseño y Arquitectura

## 3.1. Arquitectura del sistema

La arquitectura del sistema está diseñada para garantizar la escalabilidad, la modularidad y el rendimiento óptimo de la aplicación. Se compone de varios componentes que interactúan entre sí para proporcionar la funcionalidad completa del proyecto.

### 3.1.1. Componentes principales

- **ExpressJS y Node.js**
  - **Descripción:** ExpressJS actúa como el framework principal para el servidor web, mientras que Node.js sirve como el entorno de ejecución del lado del servidor.
  - **Funciones:**
    - Define y gestiona los endpoints de la API REST.

- Maneja solicitudes HTTP entrantes y genera respuestas adecuadas.
- Integra middleware para procesamiento adicional de solicitudes.

#### ■ Neo4j

- **Descripción:** Neo4j se utiliza como la base de datos principal para almacenar y consultar datos relacionales en forma de grafo.
- **Funciones:**
  - Almacena información sobre perfiles de usuarios, repositorios, colaboradores, issues y relaciones entre ellos.
  - Proporciona una interfaz para realizar consultas complejas sobre el grafo de datos.

#### ■ API de GraphQL de GitHub

- **Descripción:** La API de GraphQL de GitHub permite acceder a información detallada de GitHub, como perfiles de usuarios, repositorios e issues.
- **Funciones:**
  - Extrae datos específicos de GitHub mediante consultas GraphQL personalizadas.
  - Proporciona información adicional que puede no estar disponible fácilmente a través de la API REST tradicional de GitHub.

#### ■ Swagger

- **Descripción:** Swagger se utiliza para documentar y diseñar la API REST, proporcionando una interfaz interactiva para explorar y probar los endpoints.
- **Funciones:**
  - Genera documentación detallada de la API, incluyendo descripciones de endpoints, parámetros y respuestas esperadas.
  - Permite probar los endpoints directamente desde la interfaz de usuario.

### 3.1.2. Interacción entre componentes

**Solicitud de datos:** Cuando un cliente hace una solicitud a la API REST a través de ExpressJS, ExpressJs solicita a Node la información de la solicitud. Esta solicitud se procesa. En primer lugar, se comprueba que la información solicitada existe en la base de datos. Si no existe, se solicita la información a la API de GraphQL de GitHub y se almacena en la base de datos.

**Consulta a la base de datos Neo4j:** Neo4j ejecuta la consulta solicitada y devuelve los resultados al servidor Node.js.

**Respuesta al cliente:** El servidor Node.js procesa los datos recibidos de Neo4j y, y genera una respuesta adecuada, que se devuelve al cliente a través de ExpressJS.

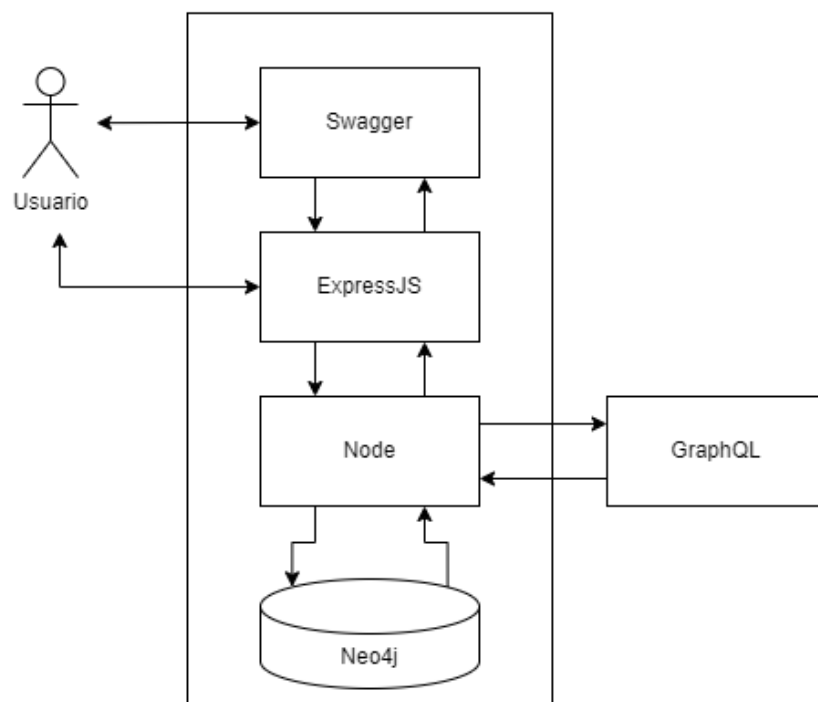


Figura 1: Diagrama de arquitectura de la aplicación

Esta arquitectura permite una comunicación eficiente entre los componentes, garantizando un rendimiento óptimo y una experiencia de usuario fluida. Los datos almacenados en Neo4j se pueden acceder y manipular a través de la API REST documentada en Swagger, y en caso de necesitar información adicional de GitHub, el servidor se encargará de solicitarla y almacenarla.

## 3.2. Diseño de la base de datos en Neo4j

El diseño de la base de datos en Neo4j se centra en representar de manera eficiente la relación entre usuarios, repositorios e issues. Se utilizan nodos para representar entidades como usuarios, repositorios e issues, y relaciones para definir las interacciones entre ellas.

### 3.2.1. Nodos

#### ■ Usuarios:

- Representan los usuarios de GitHub.
- Cada nodo de usuario tiene propiedades como el nombre de usuario, nombre completo o el número de seguidores y seguidos.

#### ■ Repositorios:

- Representan los repositorios de GitHub.
- Cada nodo de repositorio tiene propiedades como el nombre del repositorio, la descripción y la fecha de creación.

#### ■ Issues:

- Representan las issues de GitHub.
- Cada nodo de issue tiene propiedades como el título, el estado (abierto/cerrado) y la fecha de cierre.

### 3.2.2. Relaciones

#### ■ Sigue a:

- Relación entre nodos de usuario que indica que un usuario sigue a otro usuario en GitHub.
- Puede tener propiedades adicionales como la fecha en que se sigue al otro usuario.

#### ■ Trabaja en:

- Relación entre nodos de usuario y nodos de repositorio que indica que un usuario está contribuyendo en un repositorio.
- Puede tener propiedades como la fecha en que comenzó a trabajar en el repositorio y el rol del usuario en el proyecto.

#### ■ Asignado a:

- Relación entre nodos de usuario y nodos de issue que indica que un usuario está asignado a una issue.
- Puede tener propiedades como la fecha en que se asignó la issue al usuario y el estado de la asignación.



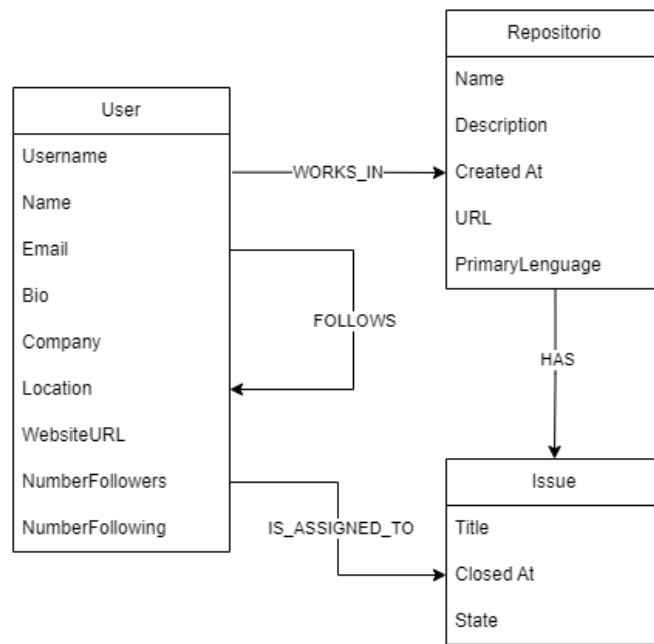


Figura 2: Modelo de datos de Neo4j

Este diseño de base de datos en Neo4j permite una representación eficiente y flexible de la información de GitHub, facilitando consultas y análisis complejos sobre la red de usuarios, repositorios e issues.

### 3.3. Diseño de la API REST con ExpressJS y Swagger

El diseño de la API REST con ExpressJS y Swagger se centra en definir endpoints claros y bien documentados que permitan a los clientes interactuar de manera eficiente con los datos almacenados en la base de datos Neo4j, así como con la información adicional obtenida de la API de GraphQL de GitHub.

- Endpoints de la API:
  - Endpoints **POST** para poblar la base de datos:
    - Se utilizan varios endpoints POST para crear nodos y relaciones en la base de datos Neo4j a partir de los datos obtenidos de la API de GitHub.
    - Estos endpoints son fundamentales para inicializar la base de datos y mantenerla actualizada con la información más reciente.
  - Endpoints **GET** para consultar la información almacenada:
    - Se utilizan para obtener información almacenada en la base de datos Neo4j.
    - Estos endpoints permiten a los usuarios acceder a la información almacenada de manera eficiente.

#### Estrategia de población de la base de datos:

Cuando un cliente realiza una solicitud a un endpoint GET y la base de datos no tiene la información suficiente para responder, se utiliza una estrategia auxiliar para poblar la base de datos mediante llamadas a los endpoints POST correspondientes de forma automática, sin necesidad de que el usuario realice ninguna consulta previa de POST.

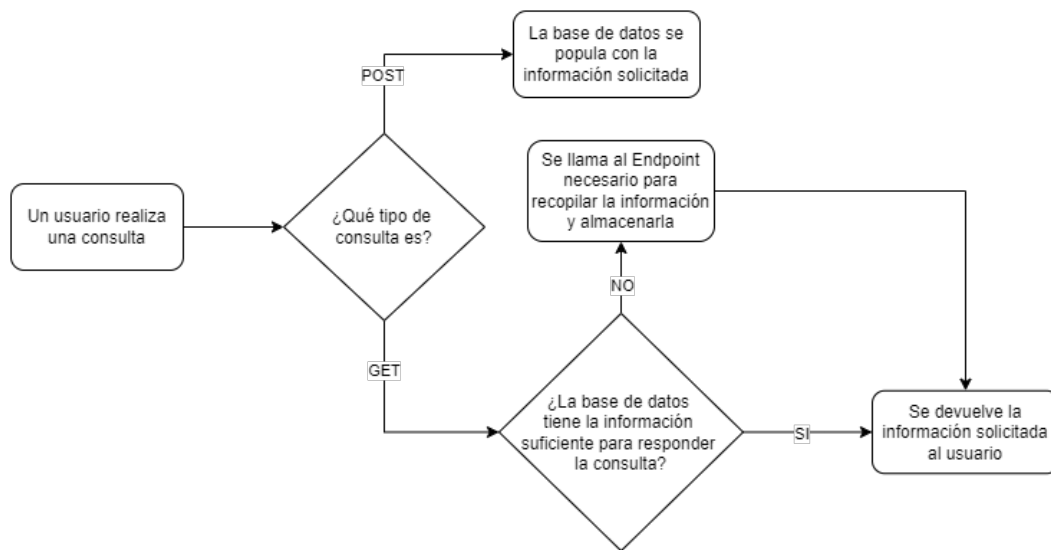


Figura 3: Diagrama de Flujo de la API REST

Los endpoints GET permiten a los usuarios acceder a la información almacenada de manera eficiente, incluso cuando la base de datos no ha sido poblada previamente mediante los endpoints POST.

Este diseño de API proporciona una forma robusta y bien estructurada de interactuar con la base de datos Neo4j y la API de GraphQL de GitHub, garantizando un rendimiento óptimo y una experiencia fluida para los usuarios finales.

## 4. Implementación

### 4.1. Extracción de datos de la API de GraphQL de GitHub

Para obtener la información, se realizan consultas a la API de GraphQL de GitHub. Para obtener la información, se realizan consultas a la API de GraphQL de GitHub, comenzando con detalles de los usuarios mediante la función `getUserInfo`. Esta función recoge información sobre el nombre de usuario, email, biografía, ubicación, y las listas de seguidores y seguidos. Estos datos permiten también establecer relaciones de seguimiento entre usuarios en la base de datos.

Posteriormente, para obtener datos detallados de los repositorios de un usuario, se utiliza la función `getRepositoriesFromUser`, que envía una consulta GraphQL específicamente diseñada para recoger detalles del repositorio como nombre, descripción, fecha de creación, URL, lenguaje principal, otros lenguajes utilizados, colaboradores y detalles sobre las issues abiertas y sus asignados.

Finalmente, la función `fetchGithubData` en el módulo `fetcher` gestiona las interacciones con la API de GitHub utilizando `GQLPaginator` para manejar la paginación y el volumen de datos eficientemente. `GQL-Paginator` utiliza internamente `axios` para hacer llamadas a la API de GitHub.

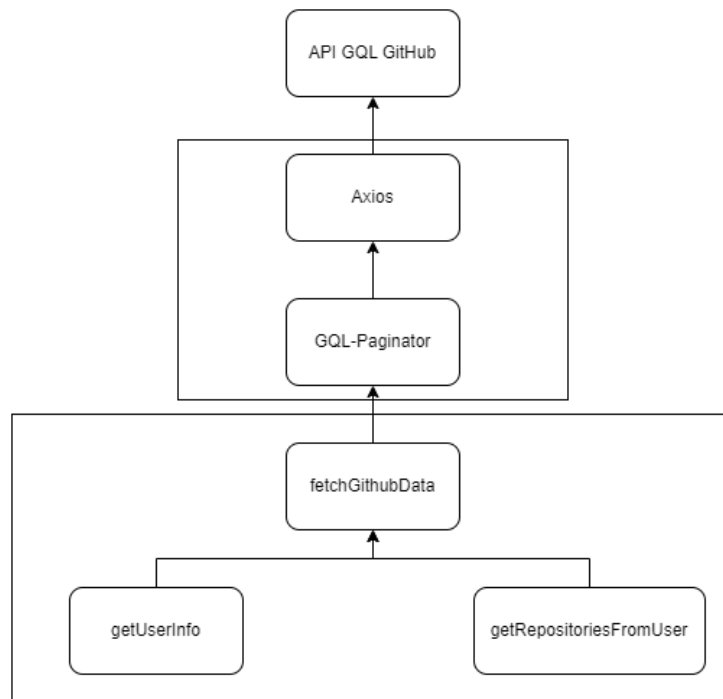


Figura 4: Diagrama de extracción de datos de la API de GraphQL de GitHub

Este conjunto de funciones asegura que el sistema pueda recuperar y almacenar datos de GitHub para posteriormente realizar consultas avanzadas.

### 4.2. Población de la base de datos

La población de la base de datos es un proceso indispensable en nuestro sistema, diseñado para asegurar que los datos extraídos de la API de GraphQL de GitHub se almacenen correctamente en Neo4j, formando una representación estructurada que soporte consultas complejas y análisis profundos. Este proceso se realiza mediante una serie de endpoints POST.

La estrategia de población de la base de datos se centra en la inserción y actualización dinámica de datos obtenidos directamente desde la API de GraphQL de GitHub. Esta estrategia se implementa a través de varios endpoints, cada uno destinado a manejar diferentes tipos de datos.

Endpoint	Método	Descripción	Implementación
/user	POST	Crea un usuario en la base de datos Neo4j utilizando datos provenientes de GitHub.	El endpoint recibe datos del usuario a través del cuerpo de la solicitud. Primero verifica si el usuario ya existe en la base de datos mediante una consulta. Si no existe, obtiene los datos del usuario desde GitHub y los inserta en la base de datos Neo4j.
/user/followers/following	POST	Crea relaciones de seguidores y seguidos entre usuarios en la base de datos Neo4j.	Este endpoint también recibe datos a través del cuerpo de la solicitud. Después de verificar o crear el usuario principal, obtiene la lista de seguidores y seguidos desde GitHub. Para cada seguidor y cada seguido, verifica si ya están en la base de datos (si no, los crea) y establece las relaciones FOLLOWS adecuadas en la base de datos.
/repositories	POST	Crea repositorios en la base de datos Neo4j a partir de datos de GitHub.	Este endpoint recibe datos del usuario a través del cuerpo de la solicitud. Verifica si el usuario existe en la base de datos. Si no existe, lo crea. Luego obtiene los repositorios del usuario desde GitHub y los inserta en la base de datos Neo4j.
/repositories/collaborators	POST	Crea repositorios y colaboradores relacionados en la base de datos Neo4j a partir de datos de GitHub.	Este endpoint también recibe datos a través del cuerpo de la solicitud. Verifica o crea el usuario principal en la base de datos. Luego obtiene los repositorios y colaboradores del usuario desde GitHub y los inserta en la base de datos Neo4j.
/repositories/issues/assignees	POST	Crea las issues y asigna usuarios a dichas issues en repositorios específicos en la base de datos Neo4j a partir de datos de GitHub.	Este endpoint recibe datos del usuario a través del cuerpo de la solicitud. Verifica o crea el usuario principal en la base de datos. Luego obtiene los repositorios, issues y asignados del usuario desde GitHub y los inserta en la base de datos Neo4j.

### 4.3. Procesamiento y obtención avanzada de la información

La API ofrece endpoints de tipo GET que permiten extraer datos de la base de datos de Neo4j. Estos endpoints permiten realizar análisis más profundos sobre la actividad de los usuarios y los repositorios en GitHub que no se podrían hacer sobre la propia API de GraphQL de GitHub.

Endpoint	Método	Descripción	Implementación
/findAssigneesInCommonIssues/:username	GET	Encuentra todos los miembros asignados a una issue en la que participa un usuario	Este endpoint verifica si el usuario existe en la base de datos. Luego realiza una consulta en la base de datos para encontrar miembros asignados a issues en las que él también participa.
/listSharedFollowers/:username1/:username2	GET	Lista los seguidores compartidos entre dos usuarios.	Este endpoint verifica si los usuarios existen en la base de datos. Luego realiza una consulta para encontrar los seguidores compartidos entre los dos usuarios.
/findSharedIssues/:username1/:username2	GET	Encuentra issues compartidas entre dos usuarios.	Este endpoint verifica si los usuarios existen en la base de datos. Luego realiza una consulta para encontrar issues compartidas entre los dos usuarios.
/findNonUserAssignedIssues/:username/:repositoryName	GET	Encuentra issues no asignadas a un usuario específico en un repositorio específico.	Este endpoint verifica si el usuario y el repositorio existen en la base de datos. Luego realiza una consulta para encontrar issues no asignadas al usuario en el repositorio especificado.
/findAllUsersContributingInRepositoriesContributedBy/:username	GET	Encuentra todos los usuarios que contribuyen en repositorios en el que ha trabajado un usuario específico.	Este endpoint verifica si el usuario existe en la base de datos. Luego realiza una consulta para encontrar usuarios que contribuyen en repositorios en los que ha contribuido un usuario especificado.
/findUsersFollowedByTheFollowsOfTheUser/:username	GET	Encuentra los usuarios a los que siguen los perfiles a los que tu sigues	Este endpoint verifica si el usuario existe en la base de datos. Luego realiza una consulta para encontrar los usuarios a los que siguen los perfiles a los que tu sigues
/findIssuesAssignedToUserFollows/:username	GET	Encuentra issues asignadas a los seguidores de un usuario.	Este endpoint verifica si el usuario existe en la base de datos. Luego realiza una consulta para encontrar issues asignadas a los seguidores del usuario especificado, excluyendo las issues asignadas al usuario.
/findRepositoriesWithUserOpenIssues/:username	GET	Encuentra repositorios con issues abiertas asignadas a un usuario específico.	Este endpoint verifica si el usuario existe en la base de datos. Luego realiza una consulta para encontrar repositorios con issues abiertas asignadas al usuario especificado.
/findUsersWithSimilarContributions/:username	GET	Encuentra usuarios con contribuciones similares a un usuario específico (al menos 2 repositorios en común).	Este endpoint verifica si el usuario existe en la base de datos. Luego realiza una consulta para encontrar usuarios con los que ha trabajado en 2 o más repositorios.

## 5. Manual de Instalación

Para instalar y ejecutar el proyecto, siga los siguientes pasos:

### 1. Instalar Neo4j:

- Descargue e instale Neo4j Desktop desde el [sitio web oficial](#).
- Una vez instalado, cree un nuevo proyecto y agregue una nueva base de datos. La contraseña puede ser la que desee ya que configuraremos un usuario y contraseña personalizados.
- Inicie la base de datos pulsando el botón de **Start** y se abrirá la interfaz de Neo4j Browser.
- Dentro de la interfaz de Neo4j Browser, acceda al panel izquierdo y seleccione el primer icono (una base de datos) que le dará acceso al panel de información de la base de datos.
- En la sección **Connected as**, haga clic en el botón **Server user Add**
- Cree un nuevo usuario con el nombre de usuario **githubdata** y la contraseña **githubdata**. Es importante que sean estos valores ya que la aplicación está configurada para utilizar estos valores por defecto.
- Asigne los roles de **admin** y **PUBLIC** al usuario y haga clic en el botón **Add User**.

Recuerde que debe mantener la base de datos de Neo4j en ejecución para que la aplicación pueda conectarse a ella.

### 2. Clonar el repositorio:

```
git clone https://github.com/JaviFdez7/Github-data-with-Neo4j-nodejs-and-expressJS
```

El enlace del repositorio es el siguiente: [repositorio](#)

### 3. Instalar las dependencias:

```
cd Github-data-with-Neo4j-nodejs-and-expressJS
npm install
```

### 4. Configurar las variables de entorno:

Cambie el nombre del archivo `.env.example` a `.env` y configure las siguientes variables de entorno:

- **GH\_TOKEN**: Debe proporcionar un token de autenticación de GitHub para acceder a la API de GitHub. Puede obtener un token de acceso personal en la [configuración](#) de su cuenta de GitHub.

### 5. Iniciar la aplicación:

```
npm start
```

En caso de no tener instalado el paquete **nodemon**, puede instalarlo de manera global con el siguiente comando:

```
npm install -g nodemon
```

O bien, puede ejecutar la aplicación con el siguiente comando:

```
node app.js
```

La aplicación se iniciará en el puerto 3000 por defecto. Puede acceder a la interfaz de swagger en <http://localhost:3000/api-docs> para realizar consultas y pruebas.

## 6. Manual de Usuario

### 6.1. Uso de la API

Esta API es accesible desde cualquier cliente HTTP, como Postman, o mediante interfaces interactivas como Swagger, que proporcionamos para facilitar la visualización y el manejo de las solicitudes y respuestas. La API está diseñada para interactuar con datos de GitHub y almacenarlos en una base de datos Neo4j, permitiendo análisis avanzados y consultas específicas.

#### 6.1.1. Organización de endpoints

La API ofrece una serie de endpoints que se dividen en operaciones de tipo POST y GET:

- **Operaciones POST:** Permiten crear y almacenar información en la base de datos Neo4j a partir de datos extraídos de GitHub.
- **Operaciones GET:** Facilitan la recuperación de información avanzada y análisis específicos a partir de los datos almacenados. Además, internamente, estos endpoints realizan consultas a la API de GitHub en caso de que no se disponga de la información necesaria en la base de datos.

#### 6.1.2. Listado detallado de endpoints

A continuación, se detallan los endpoints disponibles en la API, junto con una breve descripción de su funcionalidad:

**POST /user** Este endpoint permite la creación de un usuario en la base de datos Neo4j utilizando como entrada el nombre de usuario de GitHub.

**POST /user/followers/following** Registra en la base de datos no solo al usuario especificado, sino también a todos los perfiles que siguen a este usuario. Entonces crea una relación de FOLLOWS bidireccional.

**POST /repositories** Almacena información detallada sobre todos los repositorios públicos de un usuario de GitHub.

**POST /repositories/collaborators** Similar al anterior, pero además de guardar los repositorios del usuario, también almacena información sobre otros usuarios que colaboran en esos repositorios.

**POST /repositories/issues/assignees** Este endpoint captura no solo los repositorios del usuario y sus issues asociadas, sino también los usuarios asignados a cada issue.

A continuación se detallan los endpoints de tipo GET, que proporcionan información que desde la API de GitHub no se puede obtener directamente, pero que puede ser de utilidad a la hora de analizar los perfiles y proyectos de GitHub.

**GET /query/findAssigneesInCommonIssues/username** : Devuelve una lista de todos los usuarios que están asignados a las mismas issues que el usuario especificado. Esto es útil para identificar colaboraciones y dependencias en equipos de trabajo.

**GET /query/listSharedFollowers/username1/username2** : Muestra una lista de seguidores que son comunes entre dos usuarios especificados. Este endpoint es valioso ver las conexiones entre los usuarios.

**GET /query/findSharedIssues/username1/username2** : Encuentra las issues a las cuales ambos usuarios especificados están asignados, facilitando la identificación de áreas de trabajo en común o colaboración.

**GET /query/findNonUserAssignedIssues/username/repositoryName** : Obtiene una lista de issues en un repositorio específico que no están asignadas al usuario dado, es decir, las tareas que no están bajo su responsabilidad.

GET /query/findAllUsersContributingInRepositoriesContributedBy/username : Este endpoint proporciona una lista de todos los usuarios que contribuyen en los mismos repositorios que el usuario especificado, es decir, devuelve los usuarios que han trabajado junto al usuario.

GET /query/findUsersFollowedByTheFollowsOfTheUser/username : Devuelve los usuarios que son seguidos por aquellos que el usuario dado sigue. Es útil para explorar cómo se extienden las redes de influencia a través de conexiones indirectas.

GET /query/findIssuesAssignedToUserFollows/username : Recupera las issues asignadas a los usuarios que son seguidos por el usuario especificado, proporcionando una vista de la carga de trabajo y las responsabilidades los perfiles que sigue.

GET /query/findRepositoriesWithUserOpenIssues/username : Lista todos los repositorios que tienen issues abiertas asignadas al usuario dado, lo cual da una idea de los proyectos en los que está involucrado en los que quedan tareas pendientes.

GET /query/findUsersWithSimilarContributions/username : Busca usuarios que tienen contribuciones en al menos dos repositorios en común con el usuario dado, facilitando la identificación de patrones de colaboración y especialización en áreas específicas de desarrollo.

## 6.2. Ejemplos de consultas

A continuación, se proporcionan ejemplos de consultas para cada uno de los endpoints mencionados anteriormente. Estos ejemplos ilustran cómo se deben estructurar las solicitudes para interactuar efectivamente con la API.

```
POST /user {
  "username": "JaviFdez7"
}

POST /user/followers/following {
  "username": "JaviFdez7"
}

POST /repositories {
  "username": "JaviFdez7"
}

POST /repositories/collaborators {
  "username": "JaviFdez7"
}

POST /repositories/issues/assignees {
  "username": "JaviFdez7"
}

GET /query/findAssigneesInCommonIssues/motero2k

GET /query/listSharedFollowers/motero2k/JaviFdez7

GET /query/findSharedIssues/motero2k/JaviFdez7

GET /query/findNonUserAssignedIssues/motero2k/ISPP-G1-Talent

GET /query/findAllUsersContributingInRepositoriesContributedBy/motero2k

GET /query/findUsersFollowedByTheFollowsOfTheUser/motero2kv2

GET /query/findIssuesAssignedToUserFollows/motero2k

GET /query/findRepositoriesWithUserOpenIssues/motero2k

GET /query/findUsersWithSimilarContributions/motero2k
```

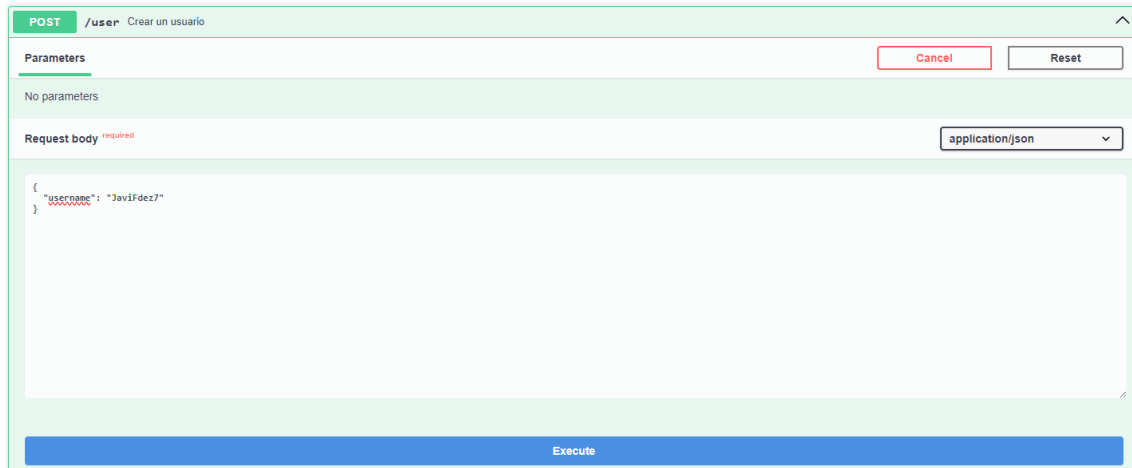


## 7. Validación

A continuación, se llevó a cabo la validación del sistema mediante pruebas del uso de los endpoints, donde se verificó su correcto funcionamiento y la integridad de los datos. Se adjuntan capturas de los resultados obtenidos durante estas pruebas. Se han utilizado los ejemplos de consulta descritos en el manual de usuario.

Para los endpoints de GET, se han validado todos, pero a continuación se muestran algunos ejemplos con los resultados obtenidos.

Para todos los endpoints de POST se utiliza el siguiente body con JaviFdez7 como username:



POST /user Crear un usuario

Parameters

No parameters

Request body required

application/json

```
{  
  "username": "JaviFdez7"  
}
```

Execute

Figura 5: Llamada al endpoint de POST

### 7.1. POST /user

Visualización del grafo en la base de datos de Neo4j:

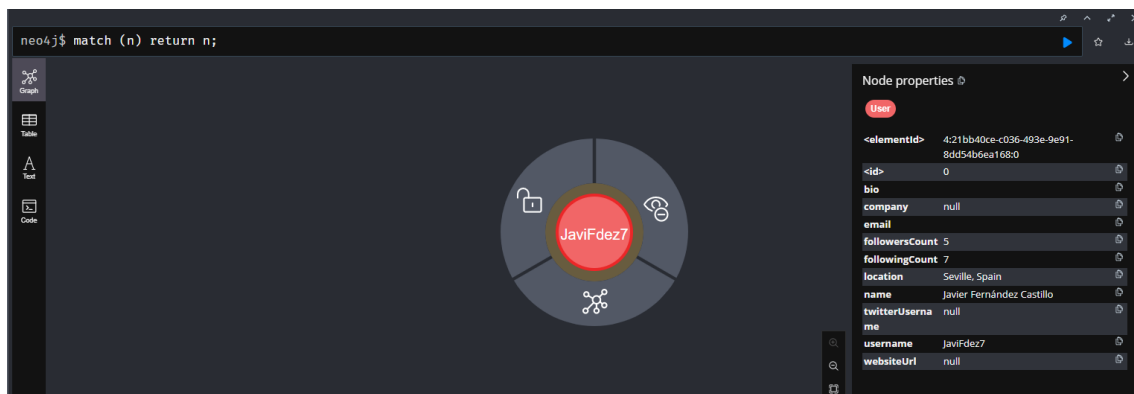


Figura 6: Resultado del endpoint de POST /user

## 7.2. POST /user/followers/following

Visualización del grafo en la base de datos de Neo4j:



Figura 7: Resultado del endpoint de POST /user/followers/following

## 7.3. POST /repositories

Visualización del grafo en la base de datos de Neo4j:

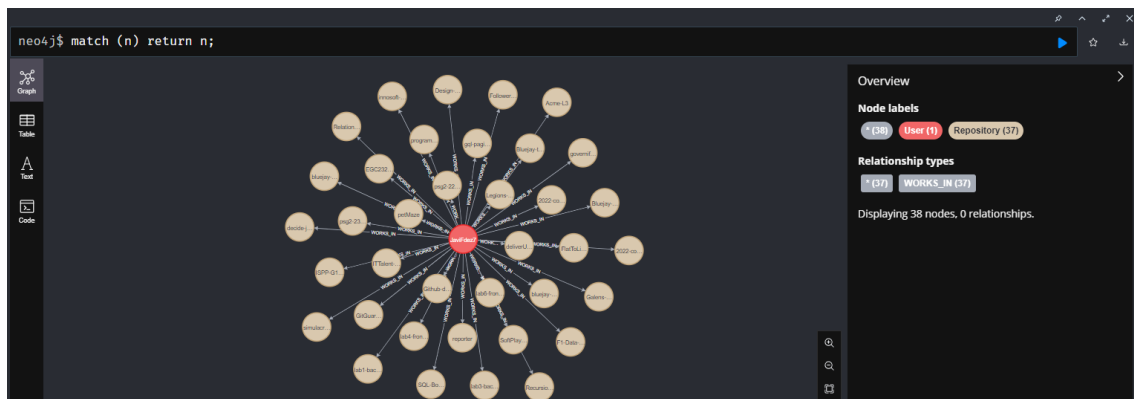


Figura 8: Resultado del endpoint de POST /repositories

## 7.4. POST /repositories/collaborators

Visualización del grafo en la base de datos de Neo4j:

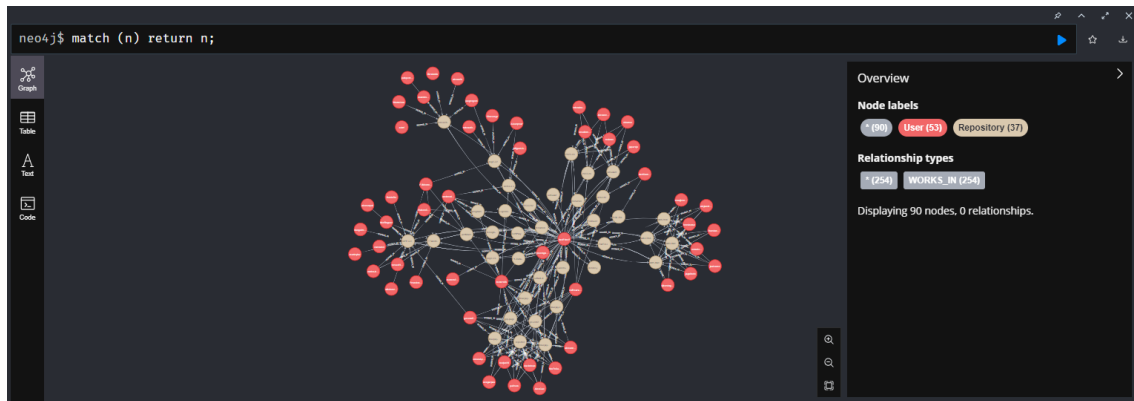


Figura 9: Resultado del endpoint de POST /repositories/collaborators

## 7.5. POST /repositories/issues/assignees

Visualización del grafo en la base de datos de Neo4j:

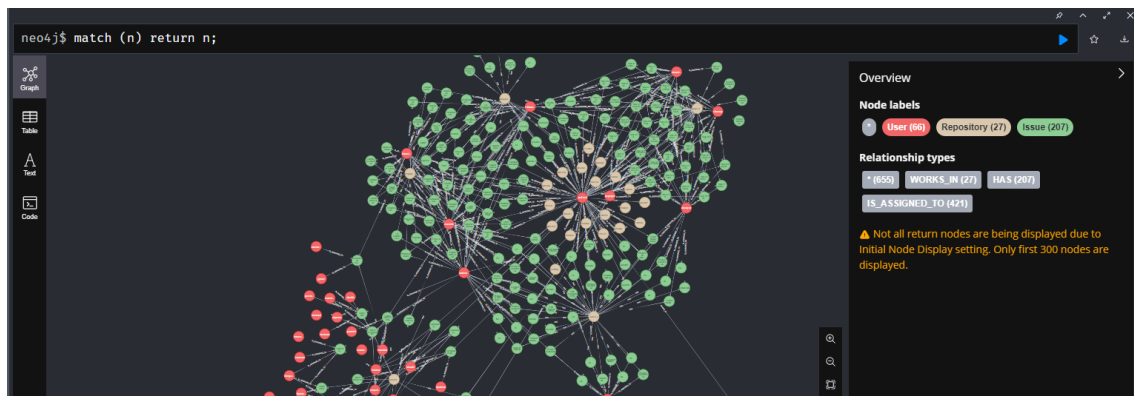
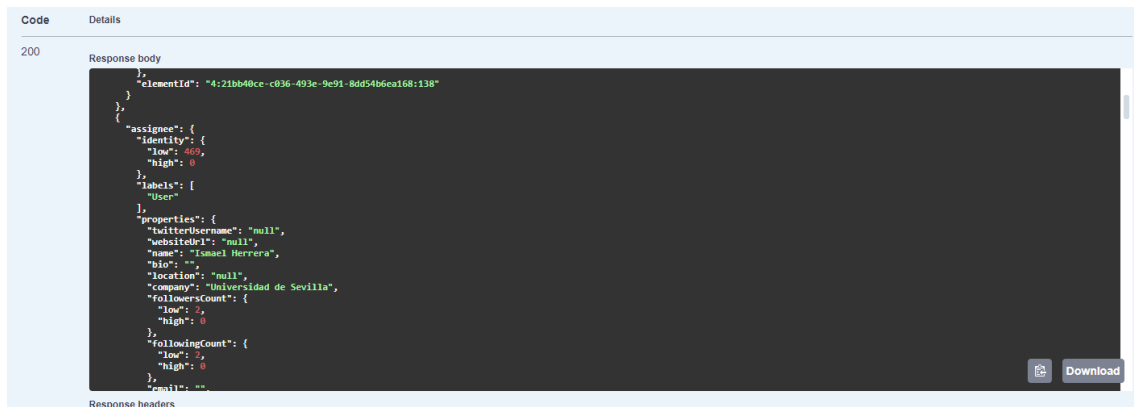


Figura 10: Resultado del endpoint de POST /repositories/issues/assignees

## 7.6. GET /query/findAssigneesInCommonIssues/motero2k

Visualización del JSON resultante de la base de datos:

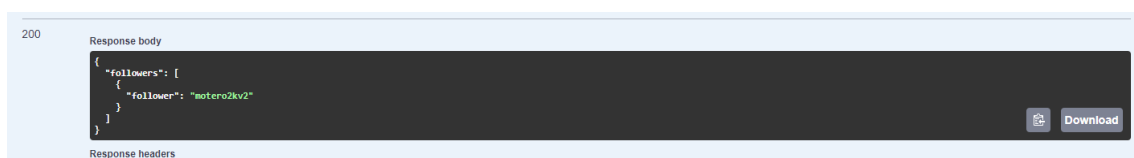


```
200
Response body
{
  "elementId": "4:21bb40ce-c036-493e-9e91-8dd54b6ea168:138"
},
{
  "assignee": {
    "identity": {
      "loc": 0,
      "high": 0
    },
    "labels": [
      "User"
    ],
    "properties": {
      "twitterUsername": "null",
      "websiteUrl": "null",
      "name": "Ismael Herrera",
      "bio": "",
      "location": "null",
      "company": "Universidad de Sevilla",
      "followersCount": {
        "low": 2,
        "high": 0
      },
      "followingCount": {
        "low": 2,
        "high": 0
      },
      "email": ""
    }
  }
}
```

Figura 11: Resultado del endpoint de GET /query/findAssigneesInCommonIssues/motero2k

## 7.7. GET /query/listSharedFollowers/motero2k/JaviFdez7

Visualización del JSON resultante de la base de datos:



```
200
Response body
{
  "followers": [
    {
      "follower": "motero2k2"
    }
  ]
}
```

Figura 12: Resultado del endpoint de GET /query/listSharedFollowers/motero2k/JaviFdez7

## 7.8. GET /query/findUsersWithSimilarContributions/motero2k

Visualización del JSON resultante de la base de datos:



```
200
Response body
{
  "similarUsers": [
    {
      "similarContributor": "motero2k"
    },
    {
      "similarContributor": "JaviFdez7"
    },
    {
      "similarContributor": "RubenCasal"
    },
    {
      "similarContributor": "carlos-bermejo"
    },
    {
      "similarContributor": "andresdominguezruiz"
    },
    {
      "similarContributor": "FJMonteroInformatica"
    },
    {
      "similarContributor": "alemerpal"
    }
  ]
}
```

Figura 13: Resultado del endpoint de GET /query/findAssigneesInCommonIssues/motero2k

## 8. Conclusiones

### 8.1. Logros alcanzados

La utilización de una base de datos de grafos, como Neo4j, junto con una API REST diseñada específicamente para interactuar con esta, proporciona capacidades superiores para el manejo y análisis de relaciones complejas entre datos que no serían posibles mediante el uso exclusivo de una API de GraphQL. Al almacenar los datos extraídos de GitHub en Neo4j, la API facilita consultas avanzadas y análisis en profundidad que aprovechan la estructura intrínsecamente relacional de una base de datos de grafos. Esto permite realizar consultas más complejas y obtener estadísticas más detalladas sobre las interacciones y relaciones entre usuarios y repositorios. En consecuencia, el uso de Neo4j amplía considerablemente las posibilidades de manipulación y consulta de datos frente a las limitaciones que presenta una API de GraphQL cuando se trata de consultas que requieren un alto grado de análisis detallado.

La elección de Neo4j en lugar de otras bases de datos como MongoDB o SQL se basa en las características específicas de Neo4j que lo hacen especialmente adecuado para manejar datos relacionales y consultas complejas.

Neo4j está diseñado específicamente para trabajar con datos estructurados en forma de grafos. Esto hace que sea ideal para representar y almacenar relaciones complejas entre entidades, como las que se encuentran en GitHub, donde los usuarios siguen a otros usuarios, contribuyen a repositorios, y las issues están asignadas a múltiples usuarios, entre otras relaciones. Además cobra mucho más sentido cuando estamos usando la API de GraphQL de GitHub, que a su vez, está orientada a grafos.

Aunque tradicionalmente las bases de datos de grafos pueden enfrentar desafíos de escalabilidad en comparación con otras bases de datos, Neo4j ha mejorado significativamente su capacidad de escalar horizontalmente, lo que lo hace viable para aplicaciones con grandes volúmenes de datos.

Si bien bases de datos como MongoDB y SQL también son utilizadas en diferentes contextos, Neo4j destaca cuando se trata de modelar y consultar datos altamente relacionales, como los que se encuentran en GitHub.

### 8.2. Limitaciones y futuras mejoras

Aunque la integración de Neo4j ha mejorado significativamente la capacidad de la API para manejar datos complejos, aún existen algunas limitaciones y áreas para futuras mejoras. Entre estas se incluyen:

- **Escalabilidad:** A medida que la cantidad de datos y usuarios aumente, podría ser necesario optimizar aún más el rendimiento y la escalabilidad del sistema para garantizar un tiempo de respuesta aceptable.
- **Seguridad:** Se deben implementar medidas adicionales de seguridad para proteger la integridad de los datos almacenados en la base de datos y garantizar que solo los usuarios autorizados puedan acceder a la API.
- **Implementar una interfaz de usuario:** Aunque la funcionalidad de la API está sólidamente implementada, se podría realizar una interfaz de usuario para hacerla más intuitiva y fácil de usar.
- **Implementación de nuevas consultas:** Debido a que las consultas están hechas a medida, sería necesario conocer el sistema para poder ampliarlo con nuevas consultas interesantes.

### 8.3. Impacto del proyecto

El proyecto ha tenido un impacto significativo en la forma en que se manejan y analizan los datos de GitHub. Al proporcionar una forma más eficiente de procesar y consultar datos relacionales, se ha facilitado el análisis de los datos de la API de GraphQL de GitHub. Esto puede tener un impacto positivo en diversos campos, como el desarrollo de software, la investigación académica y la toma de decisiones en empresas basadas en tecnología. Podría ser un aliciente para realizar una herramienta que analice los perfiles de los usuarios con objetivo de medir su capacidad de colaboración, su nivel de implicación en los proyecto o las buenas prácticas en sus repositorios.

## 9. Uso de la IA

Durante este proyecto se ha hecho uso de IAs como ChatGPT o copilot.

Fundamentalmente, se ha utilizado copilot para el autocompletado de código para así agilizar el desarrollo.

Respecto a ChatGPT, se ha utilizado con el objetivo de reescribir ciertas partes de la memoria para que quedarán más concisas, con mayor claridad y más robustas.

Aquí queda un ejemplo de como hemos usado ChatGPQ en la memoria:

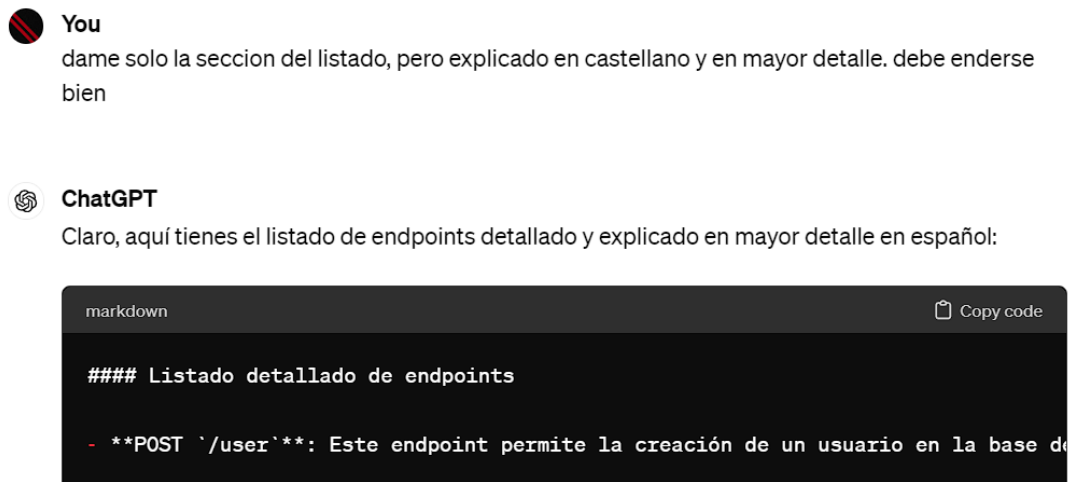


Figura 14: Uso de ChatGPT en la memoria

## Referencias

- [1] Web de ExpressJS <https://expressjs.com>