UNIVERSIDAD EAFIT

COMPARATIVA DE TIEMPOS DE RESPUESTA: GRAPHQL VS REST CON GRANDES VOLÚMENES DE DATOS

Miguel Fernando Ramos García

Tesis

Asesor:

Daniel Correa Botero

UNIVERSIDAD EAFIT
ESCUELA DE CIENCIAS APLICADAS E INGENIERÍA
MAESTRÍA EN INGENIERÍA
MEDELLÍN
2024

CONTENIDO

pág.

LISTA DE TABLAS

pág.

LISTA DE FIGURAS

pág.

Resumen

En el desarrollo de aplicaciones, la eficiencia en el rendimiento de los servicios web es esencial debido a su impacto directo en la velocidad de carga, la escalabilidad y la experiencia del usuario final. Investigaciones previas han explorado tecnologías de comunicación de servicios web tales como REST y GraphQL, sin llegar a un consenso sobre cuál maneja de manera más eficiente la transferencia de grandes volúmenes de datos. El desafío consiste en determinar cuál de estas dos tecnologías de comunicación de servicios web, REST o GraphQL proporciona tiempos de respuesta más rápidos en solicitudes de grandes volúmenes de datos. Se propone comparar ambas tecnologías bajo condiciones controladas para identificar cuál ofrece un mejor rendimiento. Los resultados de este estudio buscan orientar a quienes diseñan y desarrollan aplicaciones, permitiéndoles elegir la tecnología de comunicación de servicios web que mejor se adapte a sus necesidades y mejore la experiencia de sus usuarios.

Palabras clave: *Aplicación Web, Big Data, GraphQL, Rendimiento, REST, Servicio web.*

Abstract

In application development, the performance efficiency of web services is essential due to its direct impact on loading speed, scalability and end-user experience. Previous research has explored web service communication technologies such as REST and GraphQL, without reaching a consensus on which one handles large data transfers more efficiently. The challenge is to determine which of these two web services communication technologies, REST or GraphQL provide faster response times for large data volume requests. To this end, it is proposed to compare both technologies under controlled conditions to identify which one offers better performance. The results of this study aim to provide clear guidance to those designing and developing applications, allowing them to choose the web services communication technology that best suits their needs and improves their users' experience.

Keywords: *Big Data, GraphQL, Performance, REST, Web Application, Web Service.*

1. INTRODUCCIÓN

En el desarrollo de software, las aplicaciones se componen de diversos servicios web que interactúan entre sí para cumplir funciones específicas, siendo la comunicación entre estos servicios web un pilar fundamental para su operatividad y eficiencia. En este sentido, es fundamental considerar diferentes tecnologías de comunicación de servicios web, tales como REST y GraphQL, ya que determinan la rapidez y eficacia con la que los datos se intercambian entre las distintas partes de una aplicación, influyendo directamente en la velocidad de carga, la escalabilidad y la experiencia del usuario final.

Estudios previos han analizado diferentes tecnologías de comunicación de servicios web, especialmente REST y GraphQL, explorando cómo cada una gestiona las peticiones y los datos, mencionando sus ventajas y limitaciones en contextos específicos [1][2]. Sin embargo, no se profundiza en cuál de estas tecnologías de comunicación de servicios web gestiona de manera más eficiente los grandes volúmenes de datos, un aspecto crítico ante el creciente aumento en la generación y consumo de datos en un mundo cada vez más digital.

El desafío central de este estudio es determinar si REST o GraphQL ofrece tiempos de respuesta más rápidos al procesar solicitudes de grandes volúmenes de datos. Identificar cuál de estas dos tecnologías de comunicación de servicios web proporciona tiempos de respuesta más rápidos en condiciones de alta demanda de datos es crucial para optimizar la eficiencia y la escalabilidad de las aplicaciones web. Para ello, se propone comparar ambas tecnologías bajo condiciones controladas mediante el desarrollo de una aplicación adaptable a ambas. Esta aplicación permitirá medir los tiempos de respuesta de REST y GraphQL durante solicitudes de grandes volúmenes de datos, permitiendo una evaluación precisa y directa de su rendimiento en escenarios comparables.

El propósito de este estudio es enriquecer los criterios para elegir entre las distintas tecnologías de comunicación de servicios web. La investigación analiza cómo estas tecnologías gestionan los datos, desde pequeños hasta grandes volúmenes, para entender mejor sus fortalezas y limitaciones. Esto permitirá a desarrolladores y arquitectos de software tomar decisiones más informadas al seleccionar la tecnología de comunicación de servicios web más adecuada para sus proyectos.

Este documento está organizado con la siguiente estructura: La sección 2 detalla el planteamiento del problema. La sección 3 expone la justificación del estudio. La sección 4 presenta los objetivos del

estudio, tanto generales como específicos. La sección 5 presenta el marco teórico, abordando las definiciones clave y herramientas utilizadas. En la sección 6 se indica la metodología empleada, incluyendo las etapas del diseño y desarrollo del estudio. La sección 7 destaca la relevancia de REST y GraphQL en las aplicaciones y se explica brevemente su funcionamiento. La sección 8 presenta el desarrollo del trabajo, donde se describe una "Aplicación Base" implementada en las tecnologías de comunicación de servicios web REST y GraphQL. La sección 9 explica la estrategia de población que se realizará para la toma de datos en la aplicación REST y la aplicación GraphQL junto con la medición de sus respectivos tiempos de respuesta para con los diferentes volúmenes de datos. En la sección 10 se analizan los resultados obtenidos en las pruebas de rendimiento de ambas tecnologías. Finalmente, la sección 11 presenta las conclusiones y propone recomendaciones para trabajos futuros.

2. PLANTEAMIENTO DEL PROBLEMA

A pesar de las investigaciones que examinan la eficiencia de GraphQL frente a REST, existe una brecha significativa en el análisis del impacto de los grandes volúmenes de datos en los tiempos de respuesta de estas tecnologías de comunicación de servicios web para una única solicitud. Esta brecha es especialmente relevante dado que la escalabilidad se ha convertido en un requisito esencial para las aplicaciones, debido al creciente volumen de datos que deben transferir. Brito, Mombach y Valente [1] han demostrado las ventajas de GraphQL en la reducción de la sobrecarga causada por la transferencia de columnas innecesarias en las consultas de datos. Sin embargo, su investigación no aborda directamente el comportamiento ante grandes cargas de datos, comunes en las aplicaciones web actuales. Por otro lado, Margański y Pańczyk señalan la eficiencia de REST bajo condiciones de múltiples solicitudes de datos, pero no profundizan en cómo ambas tecnologías gestionan el aumento de los datos solicitados en una sola *request* (solicitud) [2].

3. JUSTIFICACIÓN

El análisis comparativo de los tiempos de respuesta de REST y GraphQL en el manejo de grandes volúmenes de datos proporcionará evidencia para guiar a los desarrolladores y arquitectos en la selección de la tecnología de comunicación de servicios más adecuada para sus aplicaciones. Además, este estudio genera valor a la comunidad académica al presentar documentación y guías sobre el uso de tecnologías de comunicación de servicios para el manejo de grandes volúmenes de datos.

4. OBJETIVOS

4.1. OBJETIVO GENERAL

Comparar los tiempos de respuesta de las tecnologías de comunicación de servicios REST y GraphQL en el manejo de grandes volúmenes de datos para determinar la eficiencia en distintos escenarios de consulta de datos.

4.2. OBJETIVO ESPECÍFICOS

- Analizar REST y GraphQL para identificar características en la gestión de diferentes volúmenes de datos.
- Diseñar una aplicación web base que pueda acoplarse tanto a tecnologías de comunicación de servicios REST como GraphQL.
- Desarrollar una aplicación web utilizando la tecnología REST, siguiendo el diseño de la aplicación base.
- Desarrollar una aplicación web utilizando la tecnología GraphQL, siguiendo el diseño de la aplicación base.
- Poblar progresivamente una base de datos con distintos volúmenes de datos para simular diferentes escenarios de prueba para las aplicaciones web implementadas con REST y GraphQL.
- Evaluar los tiempos de respuesta de ambas aplicaciones web (REST y GraphQL) bajo distintos volúmenes de datos para determinar su rendimiento.

5. MARCO TEÓRICO

En el desarrollo de aplicaciones web, la eficiencia en la comunicación entre servicios es clave, ya que el tiempo de respuesta en la comunicación de servicios influye directamente en el rendimiento y la experiencia del usuario final. Elegir la tecnología adecuada para manejar grandes volúmenes de datos es crucial, ya que una mala elección puede generar demoras y afectar la eficiencia del sistema. Este marco teórico explora tecnologías como REST y GraphQL, que facilitan la interacción y el intercambio de datos entre servicios. Además, se incluyen herramientas como Django, MySQL, JSON, Postman y GitHub, que apoyan el desarrollo y la gestión de aplicaciones web robustas y escalables.

- **API (Interfaz de Programación de Aplicaciones / *Application Programming Interface*):** Un conjunto de definiciones y protocolos para construir y utilizar software de aplicación, que permite la comunicación entre productos de software o servicios [3].

- **Django:** Un *framework* (marco de trabajo) de alto nivel para el desarrollo rápido de aplicaciones web seguras y mantenibles [4]. Django se integra perfectamente con tecnologías de comunicación de servicios web como REST y GraphQL, facilitando la creación de aplicaciones robustas y escalables.

- **GitHub:** Una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git [5].

- **GraphQL:** Es una tecnología de comunicación de servicios que proporciona un entorno de ejecución del lado del servidor para procesar consultas, permitiendo así la interacción entre el cliente y el servidor [6].

- **JSON (Notación de Objetos de JavaScript / *JavaScript Object Notation*):** Un formato de texto ligero diseñado para el intercambio de datos entre aplicaciones web, que es fácil de leer para los humanos y sencillo de interpretar y generar para las máquinas [7].

- **MySQL:** Un sistema de gestión de bases de datos relacional basado en SQL (Lenguaje de Consulta Estructurado / *Structured Query Language*), ampliamente en la industria para almacenar datos estructurados de manera eficiente y organizada [8].

- **Postman:** Cliente de APIs que permite a los desarrolladores automatizar pruebas y realizar peticiones a diferentes servicios web [9]. Postman es una herramienta esencial para validar y asegurar la correcta implementación de tecnologías de comunicación de servicios web, ofreciendo un entorno de prueba.

- **REST (Transferencia de Estado Representacional / *Representational State Transfer*):** Un estilo arquitectónico para servicios web que utiliza métodos HTTP (Protocolo de Transferencia de Hipertexto / *Hypertext Transfer Protocol*) sin estado para manipular representaciones de recursos web [10]. Como implementación de tecnología de comunicación de servicios web, es ampliamente adoptado por su simplicidad y eficiencia en la transmisión de datos a través de la web.

- **Tecnología de Comunicación de Servicios:** Conjunto de métodos y protocolos que permiten la interacción y comunicación entre aplicaciones independientes a través de una red, asegurando la interoperabilidad y la eficiencia en el intercambio de datos [11].

6. METODOLOGÍA

El presente estudio se llevó a cabo utilizando una estrategia de investigación basada en la metodología de ciencia del diseño (o *design science*) propuesta por Peffers [12]. Esta metodología se compone de seis etapas principales (ver Figura 1) que se describen a continuación.

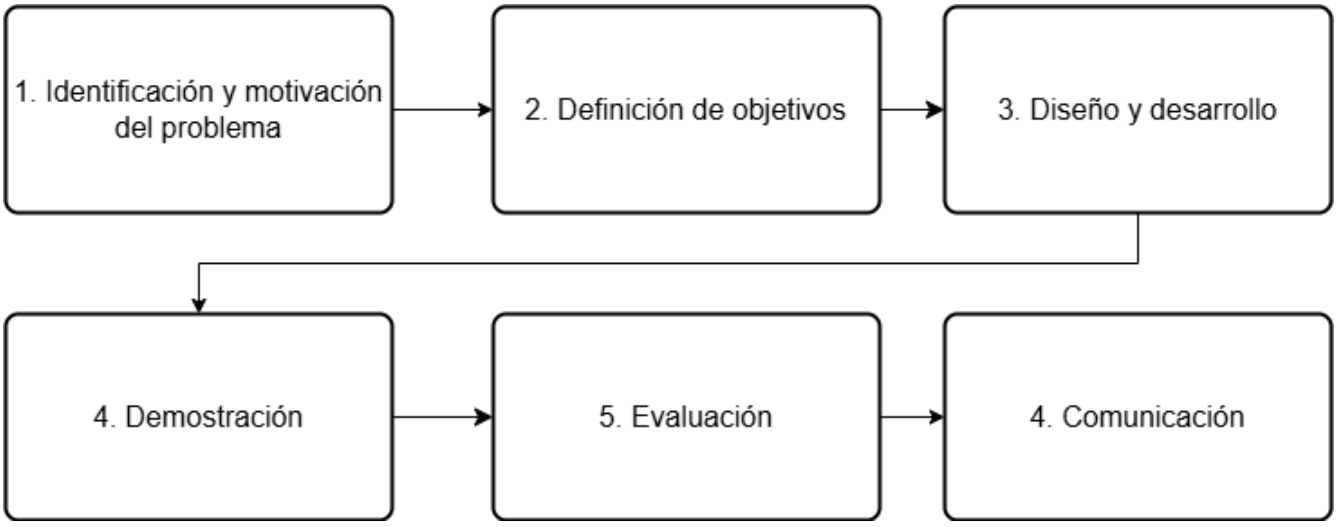

Figura 1. Metodología del trabajo

6.1. IDENTIFICACIÓN Y MOTIVACIÓN DEL PROBLEMA

Esta etapa implica definir el problema de investigación y justificar el valor de una solución [12], evaluando si REST o GraphQL es la mejor tecnología de comunicación de servicios para manejar grandes volúmenes de datos en aplicaciones web. Para más detalles, ver Capítulo 1.

6.2. DEFINICIÓN DE OBJETIVOS

Se definen los objetivos de la solución a partir de la identificación del problema y del conocimiento de lo que es posible y factible [12]. En este caso los objetivos se centran en comparar si REST o GraphQL proporciona tiempos de respuesta más rápidos al manejar grandes volúmenes de datos en diversos escenarios. Para más detalles, ver Capítulo 2.

6.3. DISEÑO Y DESARROLLO

Consiste en la creación del artefacto, que puede ser un constructo, modelo, método o instancia. Esta actividad incluye determinar la funcionalidad deseada del artefacto y su arquitectura, y luego crear el artefacto [12]. Durante la fase de diseño y desarrollo, se crearon dos aplicaciones web utilizando un diseño común, al que denominaremos artefacto o "Aplicación Base". Una de estas versiones implementad REST, mientras que la otra utiliza GraphQL. Para más detalles, ver Capítulo 5.

6.4. DEMOSTRACIÓN

En esta etapa se demuestra el uso del artefacto para resolver una o más instancias del problema. Esto podría involucrar su uso en experimentación, simulación, estudio de caso, prueba u otra actividad apropiada [12]. En este caso, las dos aplicaciones desarrolladas serán utilizadas para realizar una serie de pruebas. Cada tecnología de comunicación de servicios web será evaluada en su capacidad para manejar solicitudes de datos de tamaño creciente, midiendo tiempos de respuesta. Esta etapa se detalla en el Capítulo 5.

6.5. EVALUACIÓN

Consiste en la evaluación del artefacto al medir su eficacia para resolver un problema, comparando los objetivos planeados con los resultados reales obtenidos en su demostración. La forma de evaluación varía según el contexto y puede incluir métricas de rendimiento cuantificables [12]. En este caso los datos recolectados durante las pruebas se analizan para comparar los tiempos de respuesta de las tecnologías REST y GraphQL bajo diversos escenarios. Para más detalles, ver Capítulo 6.

6.6. COMUNICACIÓN

En esta etapa se comunica el problema, su importancia, el artefacto, su utilidad y novedad, la rigurosidad de su diseño y su efectividad a investigadores y otras audiencias relevantes, como profesionales en práctica [12]. En este caso, los resultados del estudio serán detallados en este documento con la finalidad de proporcionar una evaluación que sirva de guía para la selección de tecnologías de comunicación de servicios web, destacando los contextos en los cuales cada tecnología puede ser más beneficiosa.

7. REST Y GRAPHQL EN APLICACIONES

En este capítulo, abordaremos la relevancia de las tecnologías REST y GraphQL en el desarrollo de aplicaciones. Iniciaremos con una revisión de la arquitectura de microservicios y su comunicación basada en el patrón cliente-servidor. Exploraremos cómo estas comunicaciones pueden ser implementadas mediante REST o GraphQL, destacando la importancia de estas tecnologías de comunicación de servicios web en las aplicaciones. Finalmente, profundizaremos en el funcionamiento de cada una de estas tecnologías de comunicación de servicios web, ilustrándolo con una demostración práctica mediante una pequeña API.

7.1. IMPORTANCIA DE LAS TECNOLOGÍAS DE COMUNICACIÓN DE SERVICIOS WEB

En la actualidad, la arquitectura de microservicios se ha convertido en una metodología cada vez más popular y en auge, ofreciendo flexibilidad en las conexiones entre servicios que pueden implementarse tanto utilizando REST como GraphQL [13]. Los microservicios se refieren a servicios pequeños e independientes que trabajan en conjunto [14]. Cada servicio tiene su propia funcionalidad específica, y la práctica ideal es hacer que los servicios estén poco acoplados, lo que significa minimizar las dependencias entre ellos [15].

En una arquitectura de microservicios, al examinar la interacción entre dos servicios distintos, se evidencia un modelo cliente-servidor: El servicio solicitante actúa como cliente y el servicio que proporciona los datos funciona como servidor. Este ciclo de solicitud y respuesta se repite continuamente, creando una red de microservicios interconectados.

El tiempo de respuesta de una aplicación es la suma de todos los tiempos de comunicación entre los servicios involucrados en una solicitud, por lo que es crucial que estas comunicaciones sean rápidas y eficientes. Este estudio se enfoca en la importancia de elegir la tecnología adecuada, ya que, para grandes volúmenes de datos, decidir entre REST o GraphQL puede tener un impacto significativo en el rendimiento y la experiencia del usuario final.

REST y GraphQL son dos tecnologías de comunicación de servicios web ampliamente utilizadas en la mayoría de las aplicaciones modernas [16]. La elección entre ellas es fundamental para una gestión

eficiente de datos. En las siguientes secciones, se explorarán en profundidad estas dos tecnologías, analizando su funcionamiento y gestión de datos a través de ejemplos simples y reproducibles.

7.2. INTRODUCCIÓN A REST

REST es una tecnología estándar de comunicación de servicios web para APIs que ha estado en uso desde principios de la década de 2000 [10]. Esta tecnología se basa en un modelo cliente-servidor y utiliza URLs fijas. En este modelo, la aplicación expone un servicio a través de una API, proporcionando acceso a datos y operaciones mediante *endpoints* (puntos de acceso). Cada *endpoint* permite a los clientes realizar solicitudes específicas para acceder a recursos, y cada recurso cuenta con un conjunto predefinido de campos [18][19].

A continuación, se presenta un ejemplo de una API REST que gestiona información sobre personas, con dos *endpoints*, uno para obtener personas de género masculino y otro para obtener personas de género femenino. La API fue desarrollada utilizando Django, un *framework* de alto nivel para el desarrollo rápido de aplicaciones web seguras y mantenibles, escrito en Python [4]. El código fuente completo y la documentación de ejecución de esta API se encuentra disponible en el siguiente repositorio de GitHub: https://github.com/mframosg/basic-rest-django.

Al interactuar con la API REST, realizamos una petición al siguiente *endpoint*: `GET http://127.0.0.1:8000/users/men/` y obtendremos la siguiente respuesta en formato JSON, que contiene los nombres de los hombres disponibles en la aplicación.

```
{
    "men": [
        "Juan",
        "Pedro",
        "Pablo",
        "Jose"
    ]
}
```

En la Figura 2, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

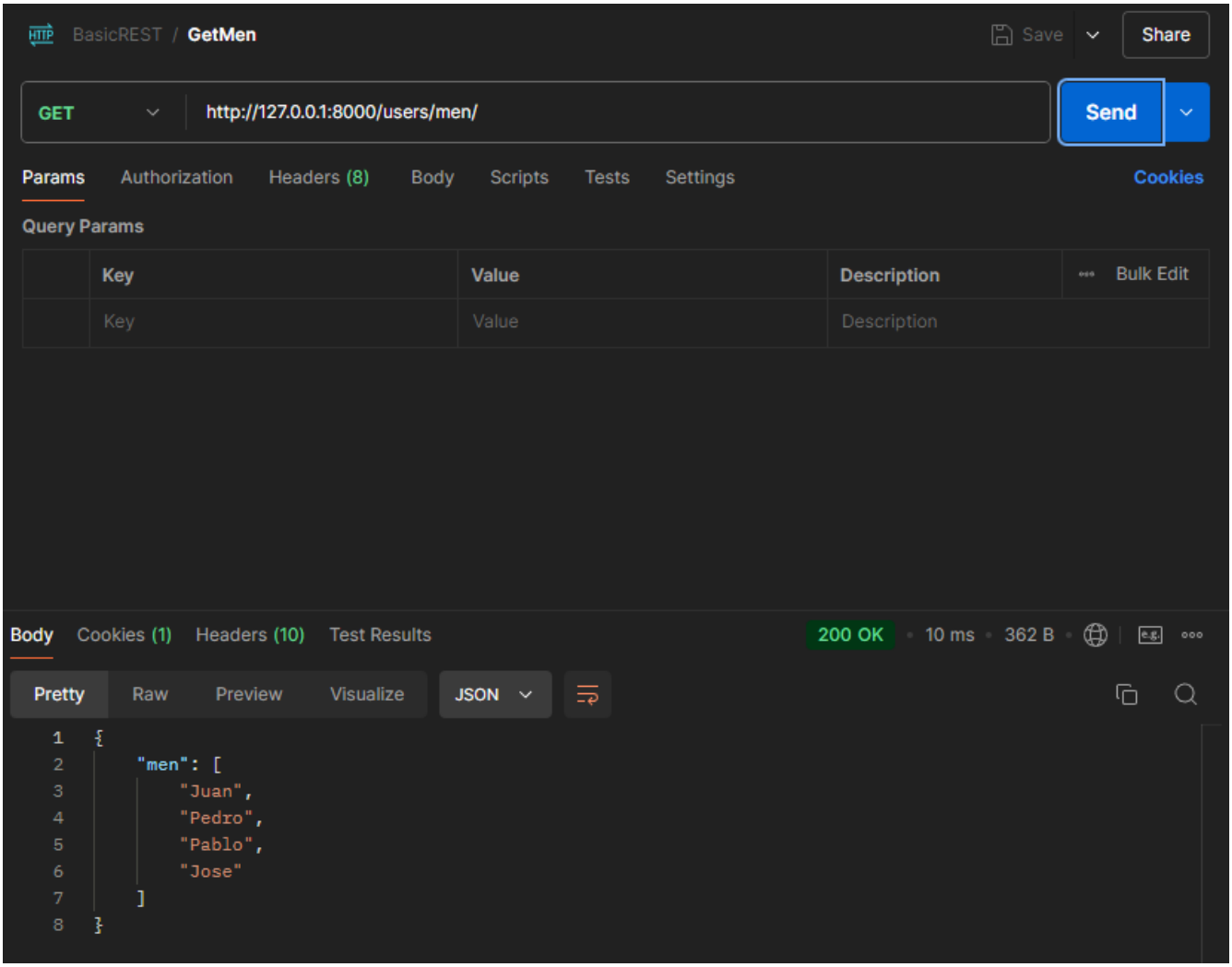

Figura 2. Respuesta API REST para obtener nombres de personas de género masculino

De manera similar, al realizar una petición al siguiente *endpoint*: `GET http://127.0.0.1:8000/users/woman/` obtendremos el siguiente JSON de respuesta, el cual contiene los nombres de las mujeres disponibles en la aplicación.

```
{
    "women": [
        "Maria"
    ]
}
```

En la Figura 3, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

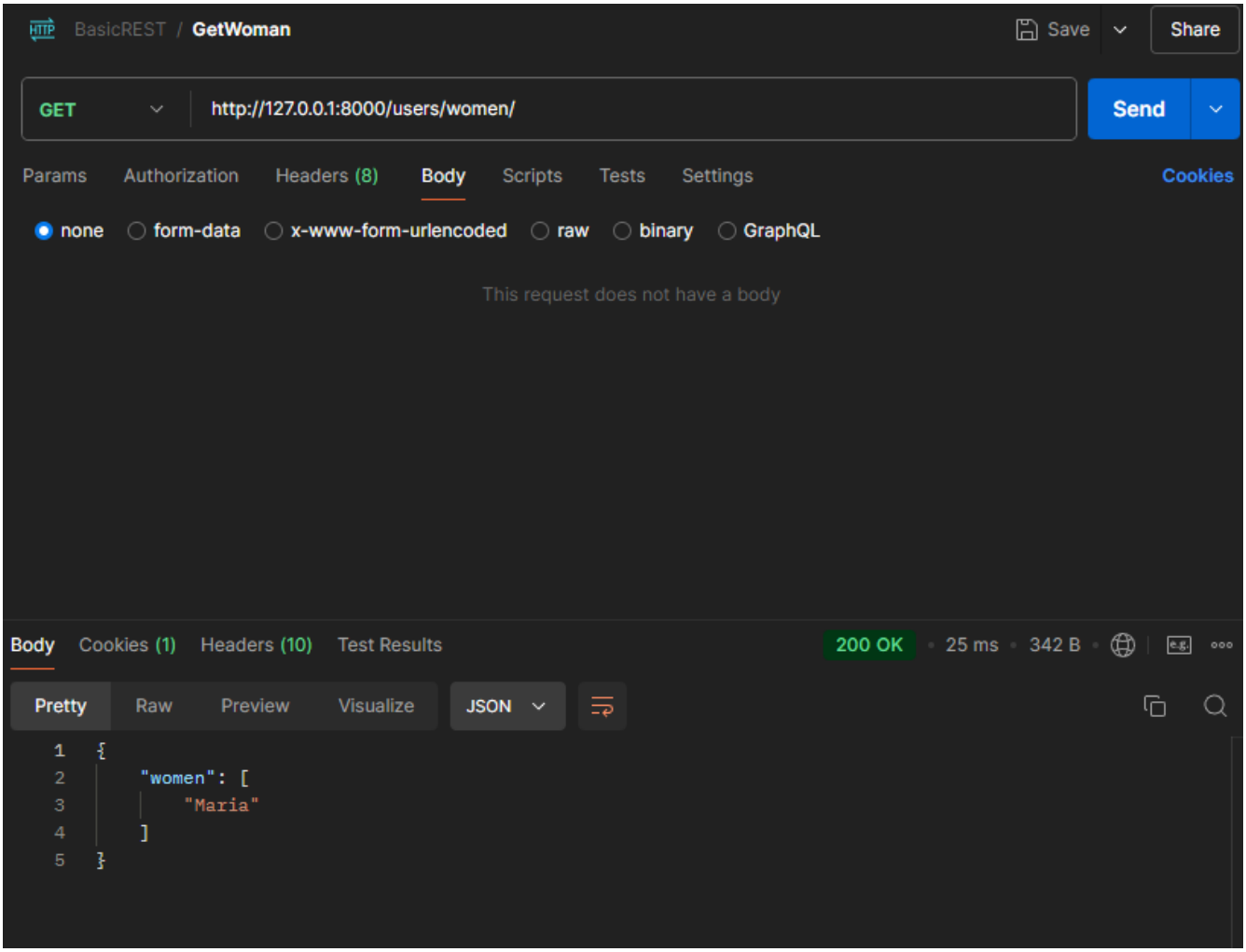

Figura 3. Respuesta API REST para obtener nombres de personas de género femenino

Las API REST utilizan múltiples *endpoints* para estructurar y gestionar eficientemente la interacción con los datos. Si se requiere consultar otro tipo de datos, deberán crearse nuevos *endpoints* específicos para acceder a la información deseada.

7.3. INTRODUCCIÓN A GRAPHQL

GraphQL, desarrollado por Facebook en 2015, permite realizar consultas dinámicas, lo que ayuda a los clientes a especificar de manera más eficiente los datos que necesitan [6]. Este enfoque se basa en un modelo cliente-servidor en el que los datos se representan como un grafo, definido mediante un esquema. Cada nodo del grafo representa objetos y contiene varios campos. Los clientes acceden al servicio

GraphQL a través de un único *endpoint*, que se utiliza para enviar múltiples consultas en una sola petición [6].

A continuación, se presenta un ejemplo de una API GraphQL que gestiona información sobre personas. Esta API permite obtener nombres de personas de género masculino y femenino a través de una única consulta. La API fue desarrollada utilizando Django, que facilita la implementación de la tecnología de comunicación de servicios GraphQL. El código fuente completo y la documentación de ejecución de esta API se encuentra disponible en el siguiente repositorio de GitHub: https://github.com/mframosg/basic-graphql-django.

Al interactuar con la API GraphQL, las peticiones al único *endpoint* existente, deben estar acompañadas de un cuerpo que contine la *query* (consulta) solicitada de la siguiente manera: `POST http://127.0.0.1:8000/graphql/` junto con la siguiente *query* en el *body* (cuerpo).

```
{
  men
}
```

Luego de realizar la solicitud, se obtiene la siguiente respuesta en formato JSON, que contiene los nombres de los hombres disponibles en la aplicación.

```
"data": {
        "men": [
            "Juan",
            "Pedro",
            "Pablo",
            "Jose"
        ]
    }
```

En la Figura 4, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

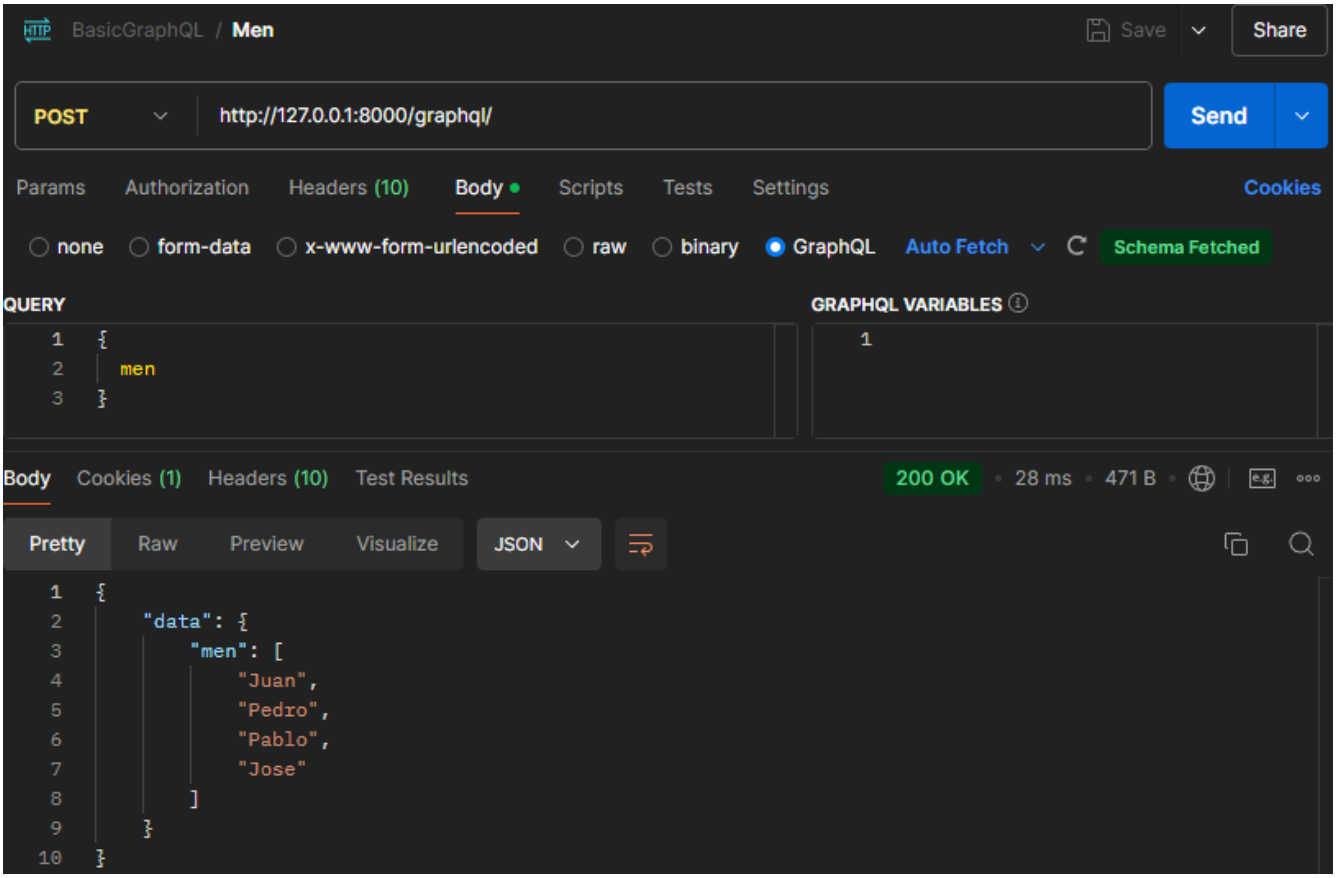

Figura 4. Respuesta API GraphQL para obtener nombres de personas de género masculino

De manera similar, el mismo *endpoint* funcionará si en el *body* se solicita por las mujeres de la siguiente manera: `POST http://127.0.0.1:8000/graphql/` junto con la siguiente *query* en el *body*.

```
{
  woman
}
```

Luego de realizar la solicitud, se obtiene la siguiente respuesta en formato JSON, que contiene los nombres de las mujeres disponibles en la aplicación.

```
{
    "data": {
        "women": [
            "Maria"
        ]
    }
}
```

En la Figura 5, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

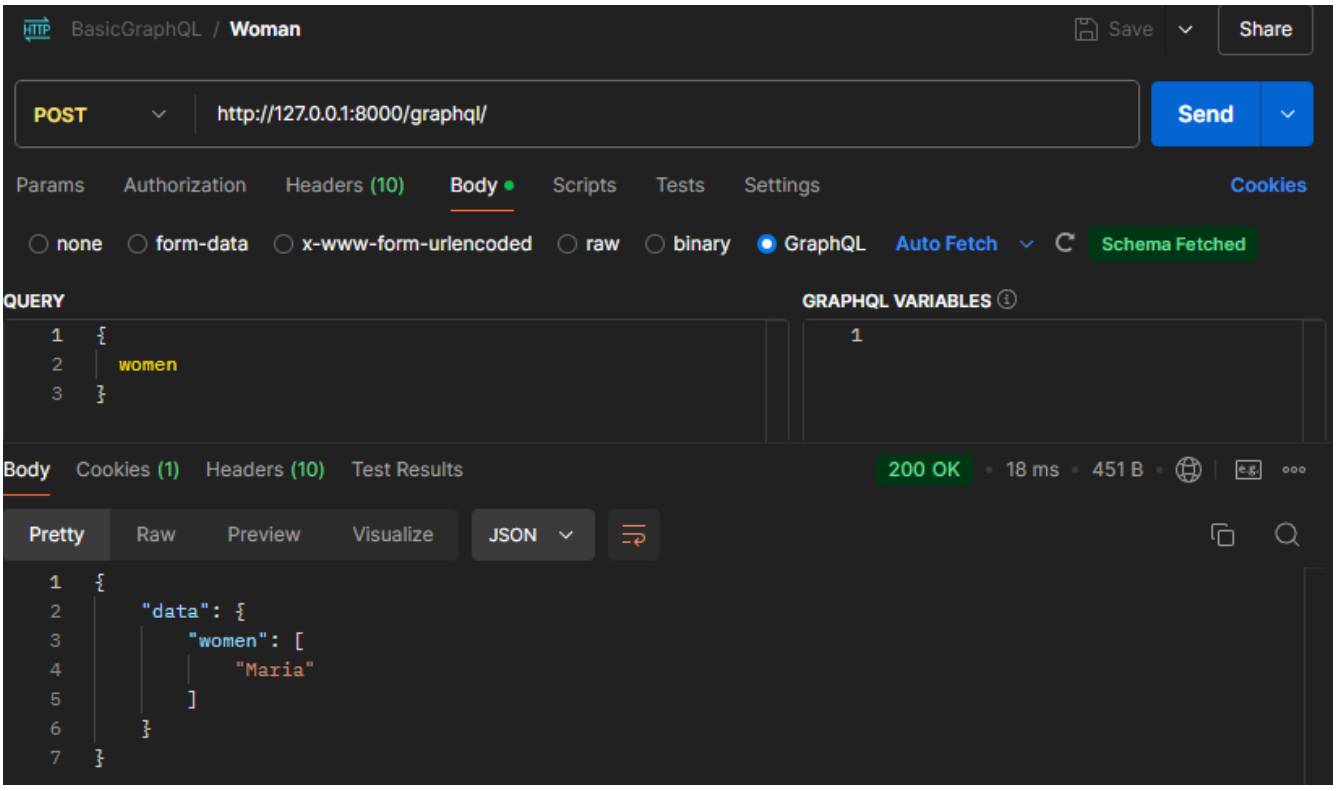

Figura 5. Respuesta API GraphQL para obtener nombres de personas de género femenino

Lo interesante de GraphQL no solo radica en su único *endpoint* para responder a cada consulta, sino también en la posibilidad de personalizar las *queries*, incluyendo únicamente los campos que se consideren necesarios. Volviendo al ejemplo, si se requiere ver información de los hombres y mujeres,

en una API REST debería crearse un nuevo *endpoint*, pero en GraphQL solo es necesario modificar un poco la *query* y solicitar al mismo *endpoint* de la siguiente forma: `POST http://127.0.0.1:8000/graphql/` junto con la siguiente *query* en el *body*.

```
{
  women, men
}
```

Luego de realizar la solicitud, se obtiene la siguiente respuesta en formato JSON, que contiene los nombres de las mujeres y hombres disponibles en la aplicación.

```
{
    "data": {
        "women": [
            "Maria"
        ],
        "men": [
            "Juan",
            "Pedro",
            "Pablo",
            "Jose"
        ]
    }
}
```

En la Figura 6, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

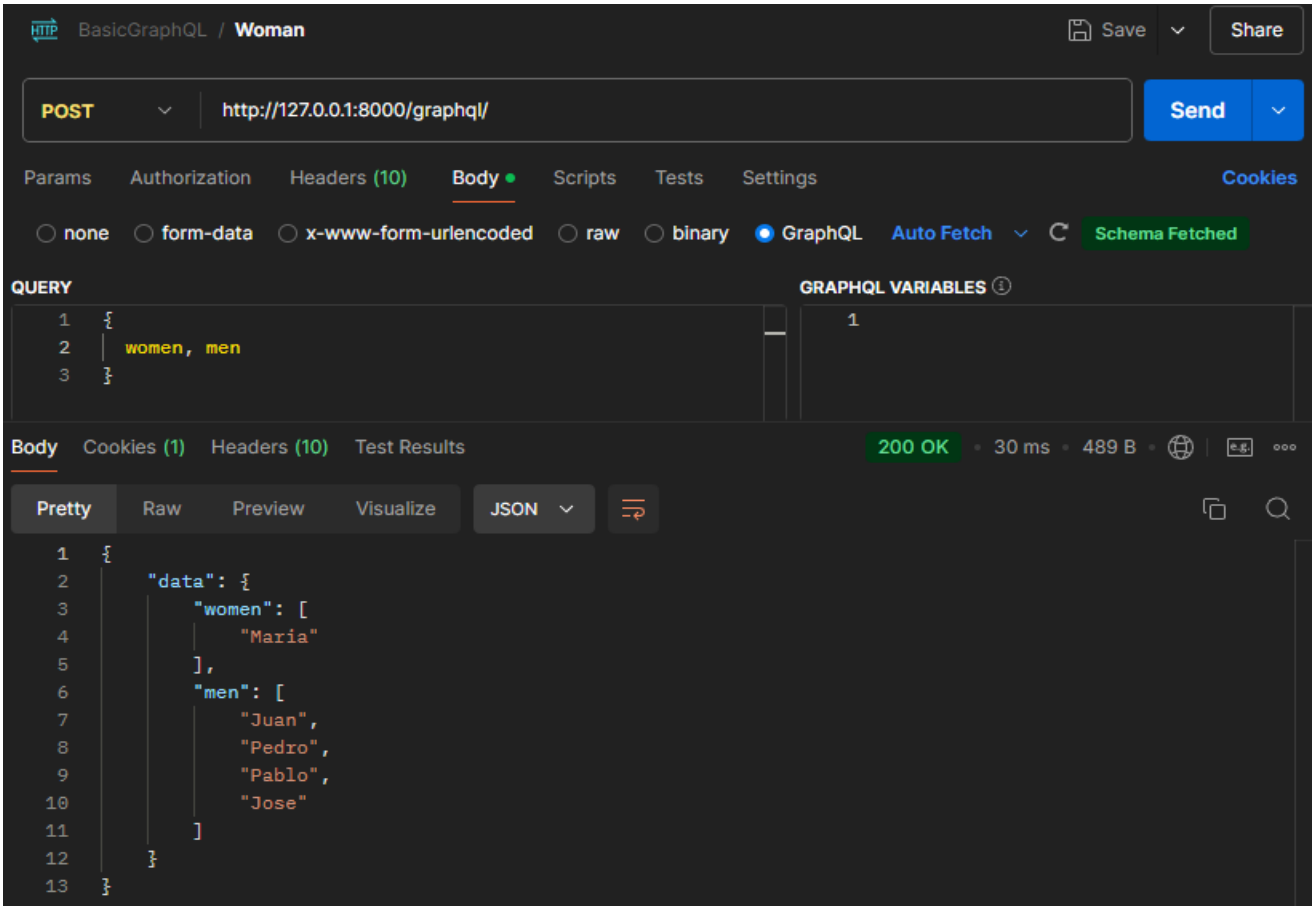

Figura 6. Respuesta API GraphQL para obtener nombres de personas de género masculino y femenino

La información de todas las personas de la aplicación se extrajo sin la necesidad de crear un nuevo *endpoint*. GraphQL utiliza *queries* para estructurar y gestionar eficientemente la interacción con los datos. Los clientes pueden especificar exactamente qué campos desean obtener, lo que permite una mayor flexibilidad y eficiencia en comparación con los enfoques tradicionales basados en REST.

8. DESARROLLO DEL TRABAJO

En este capítulo se detalla el diseño de una "Aplicación Base" adaptable a las tecnologías de comunicación de servicios web REST y GraphQL. Posteriormente, se profundiza en las tres operaciones principales para interactuar con los datos de los usuarios, analizando el volumen de datos a manejar y la implementación específica para cada tecnología, destacando las diferencias entre una aplicación basada en REST y otra en GraphQL. Finalmente, los resultados de las pruebas se utilizarán para comparar el rendimiento de ambas tecnologías en diferentes escenarios, lo que permitirá extraer conclusiones claras sobre su eficiencia.

A continuación, se definirá el alcance de la "Aplicación Base", se establecerá el tamaño de los volúmenes de datos a evaluar, se describirá el diseño de la "Aplicación Base", y finalmente, se abordará el *stack* tecnológico, diseño e implementación de la "Aplicación REST" y "Aplicación GraphQL" basados en la "Aplicación Base".

8.1. DEFINICIÓN DEL ALCANCE DE LA APLICACIÓN BASE

En este trabajo decidimos diseñar una "Aplicación Base" para gestionar información de usuarios y realizar varios tipos de operaciones sobre un conjunto de datos de usuarios, de los cuales se tendrá nombre, edad y género. A partir del diseño agnóstico a la tecnología de la "Aplicación Base", se llevarán a cabo dos implementaciones: una utilizando REST y otra con GraphQL.

La Figura 7 muestra la relación entre la "Aplicación Base", con sus dos implementaciones: una con REST ("Aplicación REST") y la otra con GraphQL ("Aplicación GraphQL"). Ambas implementaciones comparten una estructura común heredada de la "Aplicación Base", lo que significa que tienen los mismos métodos y funcionalidades. La única diferencia entre ellas radica en la tecnología de comunicación de servicios web utilizada: REST en una implementación y GraphQL en la otra.

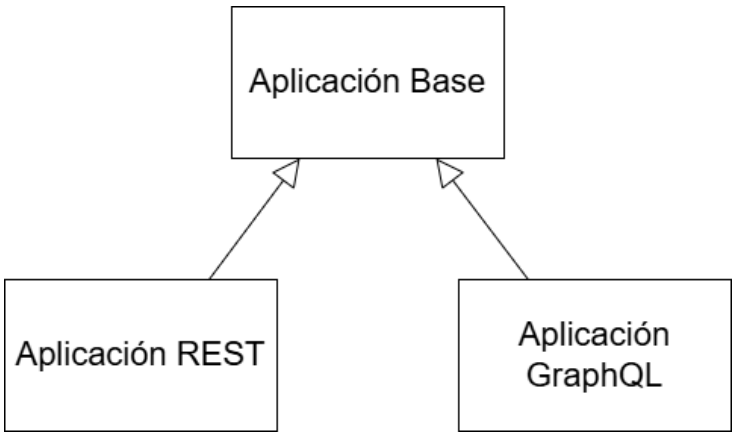

Figura 7. Diagrama de relación entre las aplicaciones

A continuación, se listan las operaciones de la “Aplicación Base”.

- **Obtener todos los usuarios:** Recupera una lista completa de todos los usuarios almacenados en la base de datos.
- **Filtrar usuarios por género:** Permite la recuperación de usuarios basándose en un parámetro de género específico (masculino o femenino).
- **Obtener usuarios por género y condición de edad:** Recupera usuarios que cumplen con una condición específica de edad (por ejemplo, mayores de 30 años) y que pertenecen a un género determinado.

8.2. TAMAÑOS DE DATOS A EVALUAR

En el contexto de esta aplicación, es esencial evaluar cómo ambas tecnologías manejan diferentes tamaños de datos. Mientras que REST puede ser más sencillo de implementar debido a su estructura basada en recursos y sus URLs predefinidas, GraphQL ofrece la ventaja de permitir consultas precisas que pueden reducir la cantidad de datos transferidos. Esta flexibilidad puede resultar en un rendimiento mejorado cuando se manejan grandes conjuntos de datos, pero también podría implicar una mayor complejidad en la implementación y optimización de las consultas.

Para evaluar el rendimiento de las tecnologías de comunicación de servicios web REST y GraphQL, se definirá un conjunto de tamaños de datos sobre los cuales se realizarán las pruebas. Estos tamaños son.

- **Pequeño:** Conjunto de datos con 1000 usuarios.
- **Mediano:** Conjunto de datos con 100000 usuarios.
- **Grande:** Conjunto de datos con 500000 usuarios.
- **Muy grande:** Conjunto de datos con 1000000 de usuarios.

Estas pruebas permitirán observar cómo cada tecnología maneja la carga de datos en diferentes escalas y proporcionar una evaluación comparativa del rendimiento en términos de tiempos de respuesta.

8.3. DISEÑO DE LA APLICACIÓN BASE

Una vez entendido cómo funcionan REST y GraphQL, diseñamos una aplicación web que sea adaptable a ambas tecnologías de comunicación de servicios. La Figura 8 ilustra el diseño de la "Aplicación Base" agnóstico a la tecnología.

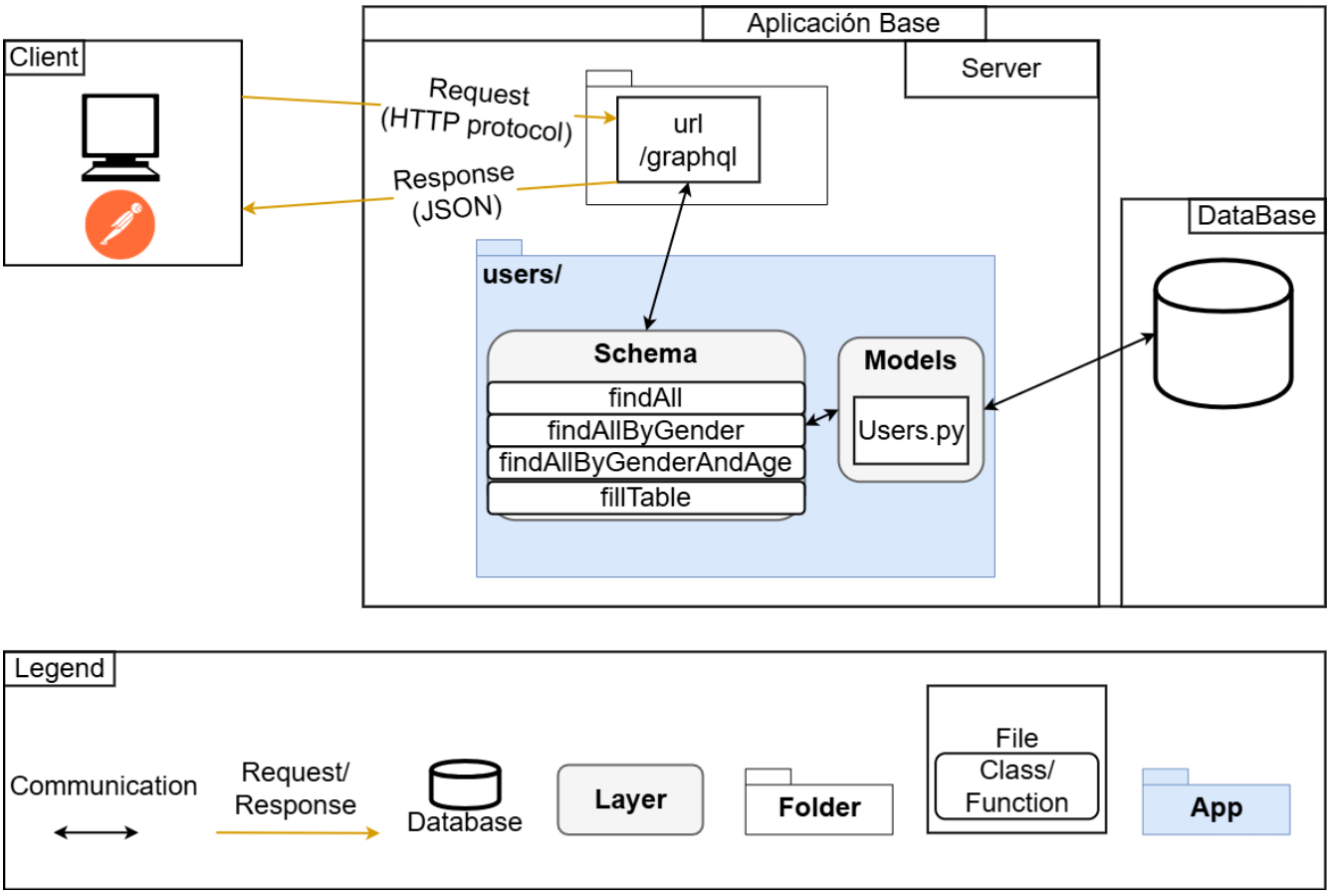

Figura 8. Diseño de la "Aplicación Base" agnóstico a la tecnología

- **Client:** Usuario que envía peticiones a la API y maneja las respuestas en formato JSON para actualizar la interfaz.
- **Server:** El *server* (servidor) actúa como intermediario entre el *client* (cliente) y la *database* (base de datos), manejando las solicitudes del cliente. Puede ser implementado en tecnologías como REST (con varios *endpoints*) o GraphQL (con un único *endpoint*). Este *server* es crucial para la seguridad y transaccionalidad de la aplicación, ya que protege la base de datos de accesos no autorizados.
- **Database:** Almacena todos los datos de la aplicación y es únicamente consultada y actualizada por el *server* según las peticiones del *client*.

8.4. SELECCIÓN DE *STACK* TECNOLÓGICO PARA LA APLICACIÓN REST Y LA APLICACIÓN GRAPHQL

En esta sección, se detallan los criterios que orientaron la selección de las herramientas tecnológicas de la "Aplicación REST" y la "Aplicación GraphQL", enfocándose en la selección del lenguaje de programación, su *framework* asociado y el sistema de gestión de bases de datos. Estas decisiones se basaron en factores como la popularidad, soporte técnico, facilidad de uso y adaptabilidad. Al seleccionar Python y su *framework* Django se tuvieron los siguientes criterios.

- **Popularidad y Soporte:** Python figura como uno de los lenguajes de programación más populares en GitHub, con una comunidad activa y un soporte extensivo. Esto garantiza acceso a un amplio conjunto de librerías para resolver problemas y adoptar las mejores prácticas en el desarrollo [17].
- **Facilidad de Uso y Adaptabilidad:** Python y Django simplifican significativamente la implementación de APIs, tanto REST como GraphQL, gracias a sus herramientas y librerías integradas, lo que permite una adaptabilidad superior en el desarrollo de aplicaciones web [4].
- **Compatibilidad:** La capacidad de integrarse fácilmente con las bases de datos y otros sistemas existentes.

La base de datos seleccionada fue **MySQL** por su amplia aceptación dentro de la industria y su capacidad eficiente para manejar datos relacionales en aplicaciones web. Debido a su uso generalizado, ofrece excelente compatibilidad con diversos lenguajes de programación y es de uso abierto [8].

8.5. DISEÑO E IMPLEMENTACIÓN DE LA APLICACIÓN REST

Para implementar la “Aplicación REST”, utilizaremos el stack tecnológico seleccionado. La Figura 9 ilustra el diseño de la “Aplicación REST”.

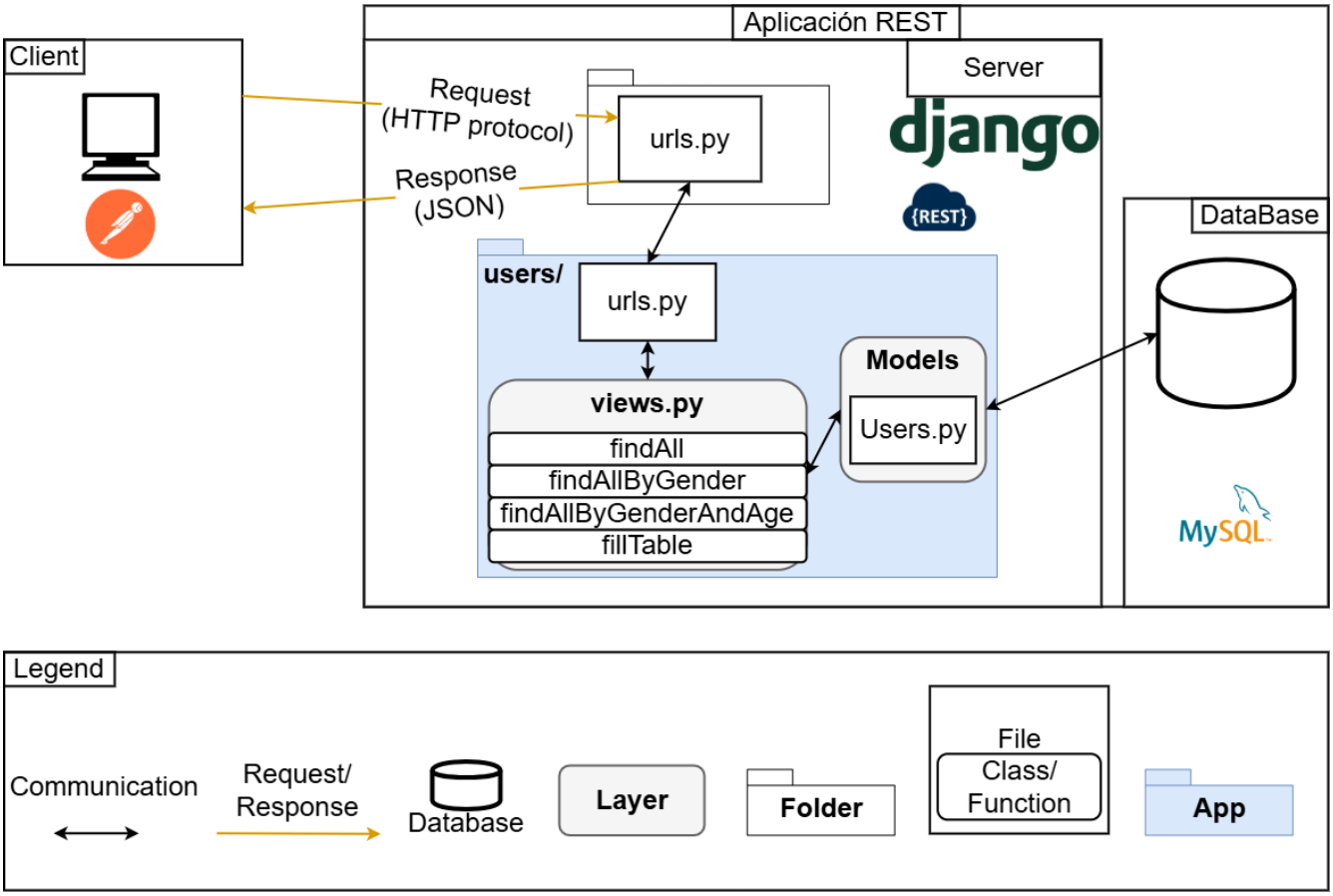

Figura 9. Diseño de la “Aplicación REST”

- **Client:** Utilizado para enviar las diferentes solicitudes a la “Aplicación Base” y verificar las respuestas en formato JSON.
- **Server:** El *server* implementado con Django maneja las solicitudes del *client* y actúa como intermediario entre el *client* y la *database* con múltiples *endpoints*.
- **Database:** Almacena la información requerida por la aplicación, es consultada y actualizada por la API REST según las peticiones del cliente. Esta es implementada con MySQL, un sistema de gestión de bases de datos relacional basado en SQL de código abierto [8].

8.6. IMPLEMENTACIÓN DE LA APLICACIÓN REST

A continuación, se presentará una breve descripción de los componentes involucrados en la implementación de la "Aplicación REST". El código fuente de la aplicación está disponible en el siguiente repositorio: https://github.com/mframosg/rest-api. Se abordarán los siguientes componentes: el modelo de datos, vistas y las URLs.

a) Definición del modelo de datos en **users/models.py**

En este archivo, definimos el modelo de los usuarios e indicamos el nombre de la tabla donde se almacenarán (revisar documentación del repositorio para configuración de la base de datos).

```
from django.db import models

class User(models.Model):
    id = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    gender = models.CharField(max_length=1, choices=[('M', 'Male'), ('F',
'Female')])
    class Meta:
        db_table = 'user'

    def to_dict(self):
        return {
            'id': self.idusuario,
            'name': self.name,
            'age': self.age,
            'gender': self.gender,
        }
```

b) Creación de vistas en **users/views.py**

Aquí, creamos las vistas que manejan las peticiones para obtener los usuarios. Para este fragmento de código, solo está la operación de retornar la totalidad de usuarios, pero aquí van todas las operaciones.

```
def users_api(request):
    users = User.objects.values('name', 'age', 'gender')
    return JsonResponse(list(users), safe=False)
```

c) Definición de URLs en **api/urls.py**

En este archivo, definimos las rutas o *endpoints* para cada operación definida en la vista.

```
from django.contrib import admin
from django.urls import path
from users.views import users_api, fill_table

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/users/', users_api)
]
```

Para cada operación definida en la vista, se tendrá un *endpoint* distinto, resultando en cada operación asociada a una URL.

Finalmente, al ejecutar la aplicación ya utilizando el siguiente *endpoint*: `GET http://127.0.0.1:8000/api/users/` obtendremos la siguiente respuesta en formato JSON, que contiene una selección de todos los usuarios disponibles en la "Aplicación REST".

```
[
    {
        "name": "Natasha Gilbert",
        "age": "62",
        "gender": "F"
    },
    {
        "name": "Jennifer Hernandez",
        "age": "45",
        "gender": "F"
    }
]
```


En la Figura 10 se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

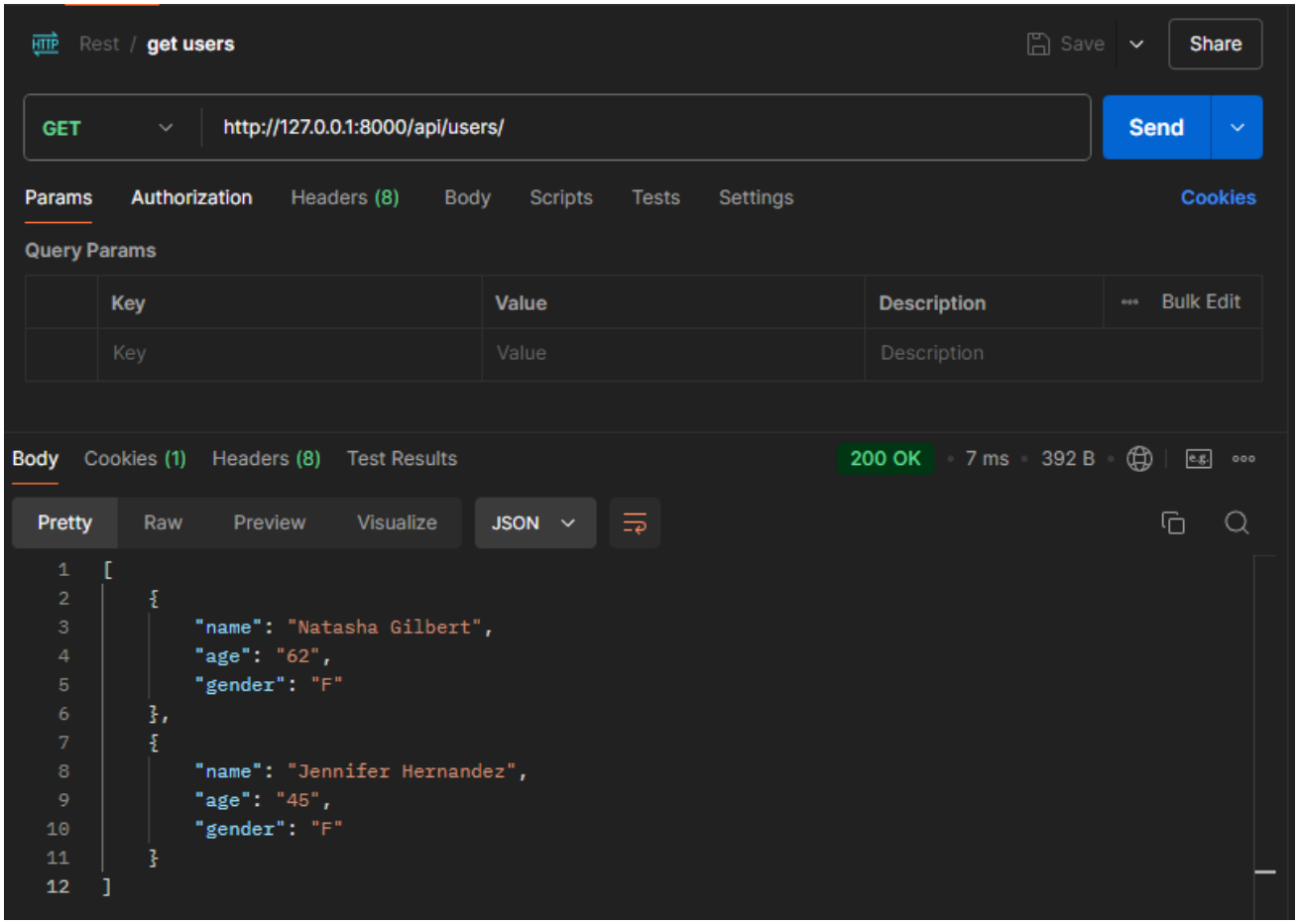

Figura 10. Respuesta de la “Aplicación REST”

8.7. DISEÑO E IMPLEMENTACIÓN DE LA APLICACIÓN GRAPHQL

Para implementar la "Aplicación GraphQL", utilizaremos el *stack* tecnológico seleccionado. La Figura11 ilustra el diseño de la "Aplicación GraphQL".

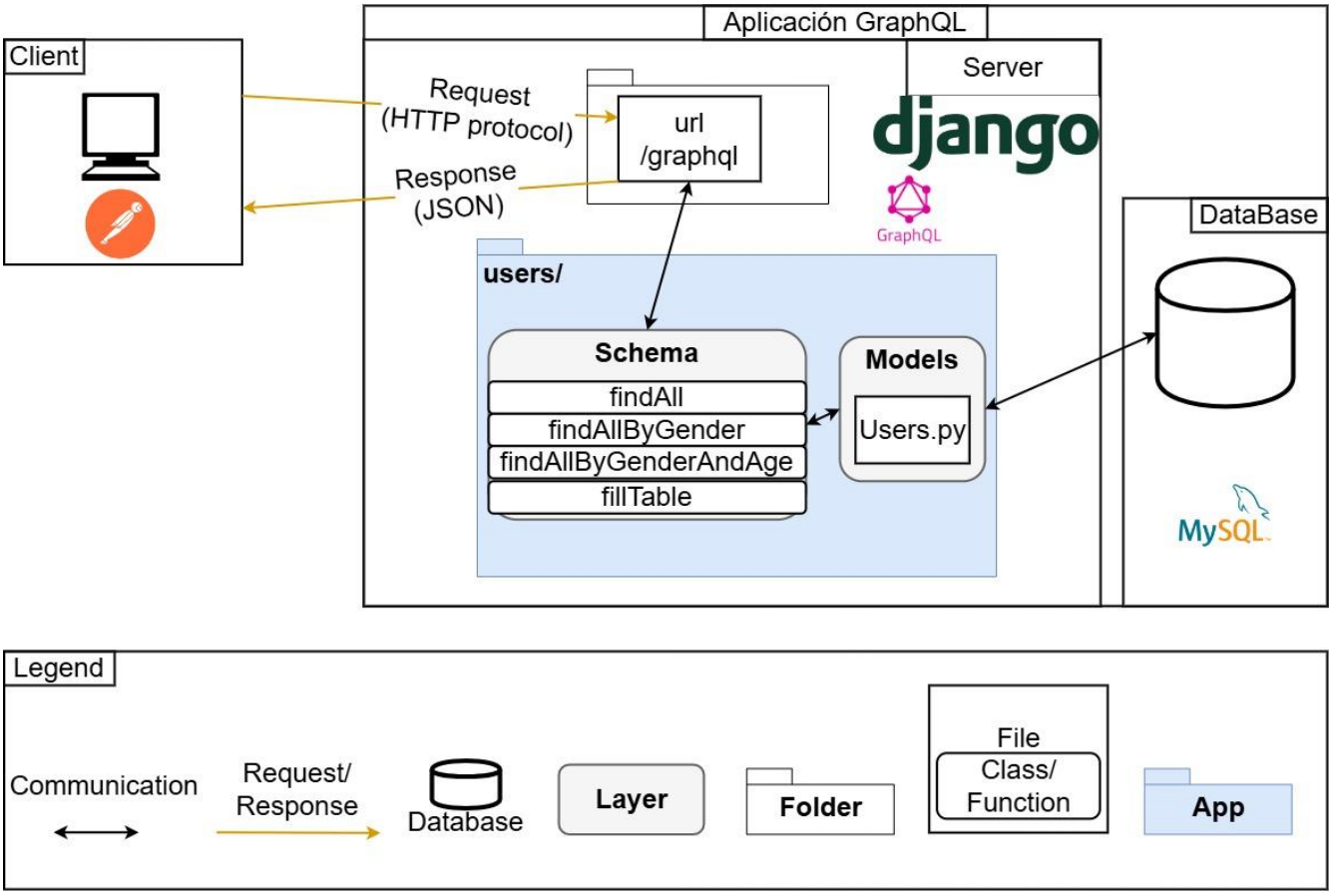

Figura 11. Diseño de la "Aplicación GraphQL"

- **Client:** Utilizado para enviar las diferentes solicitudes a la API GraphQL y verificar las respuestas en formato JSON.
- **Server:** El *server* implementado con Django que maneja las solicitudes del cliente y actúa como intermediario entre el *client* y la *database* con único *endpoint*.
- **Base de Datos:** Almacena la información requerida por la aplicación, es consultada y actualizada por la API GraphQL según las peticiones del *client*. Esta es implementada con MySQL, un sistema de gestión de bases de datos relacional basado en SQL de código abierto [8].

8.8. IMPLEMENTACIÓN DE LA APLICACIÓN GRAPHQL

A continuación, se presentará una breve descripción de los componentes involucrados en la implementación de la "Aplicación GraphQL". El código fuente de la aplicación está disponible en el siguiente repositorio: https://github.com/mframosg/graphql-api. Se abordarán los siguientes componentes: el modelo de datos, esquema y las URLs.

a) Definición del modelo de datos en **users/models.py**

En este archivo, definimos el modelo de los usuarios e indicamos el nombre de la tabla donde se almacenarán (revisar la documentación del repositorio para la configuración de la base de datos).

```
from django.db import models

class User(models.Model):
    id = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    gender = models.CharField(max_length=1, choices=[('M', 'Male'), ('F',
'Female')])
    class Meta:
        db_table = 'user'
```

b) Configuración de GraphQL en **users/schema.py**

Aquí creamos el esquema GraphQL que manejará las peticiones para obtener los usuarios. Definimos la *query* para gestionar las peticiones necesarias para las operaciones de consultas y la lógica necesaria para aplicar en el conjunto de datos.

```
import graphene
from graphene_django.types import DjangoObjectType
from .models import User

class UserType(DjangoObjectType):
```

```
    class Meta:
        model = User
        fields = ('name', 'age', 'gender')

class Query(graphene.ObjectType):
    users = graphene.List(UserType)

    def resolve_users(self, info):
        return User.objects.all()

class Mutation(graphene.ObjectType):
    fill_table = FillTable.Field()

schema = graphene.Schema(query=Query, mutation=Mutation)
```

c) Definición de URLs en **graphql_api/urls.py**

En este archivo, definimos la ruta para el único *endpoint* de GraphQL que recibirá los *queries* para interactuar con las operaciones definidas en el **schema.py**.

```
from django.contrib import admin
from django.urls import path
from graphene_django.views import GraphQLView
from users.schema import schema



urlpatterns = [
   path('admin/', admin.site.urls),
   path("graphql/", GraphQLView.as_view(graphiql=True, schema=schema)),
]
```

Finalmente, al ejecutar la aplicación, se podrá interactuar con ella utilizando el siguiente *endpoint*: `POST http://127.0.0.1:8000/graphql/` junto con la siguiente *query* en el *body*.

```
{
  users {
    name
    age
    gender
  }
}
```

Luego de realizar la solicitud, se obtiene la siguiente respuesta en formato JSON, que contiene una selección de todos los usuarios disponibles en la "Aplicación GraphQL".

```
{
  "data": {
    "users": [
      {
        "name": "Natasha Gilbert",
        "age": 62,
        "gender": "F"
      },
      {
        "name": "Jennifer Hernandez",
        "age": 45,
        "gender": "F"
      }
    ]
  }
}
```

En la Figura 12, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

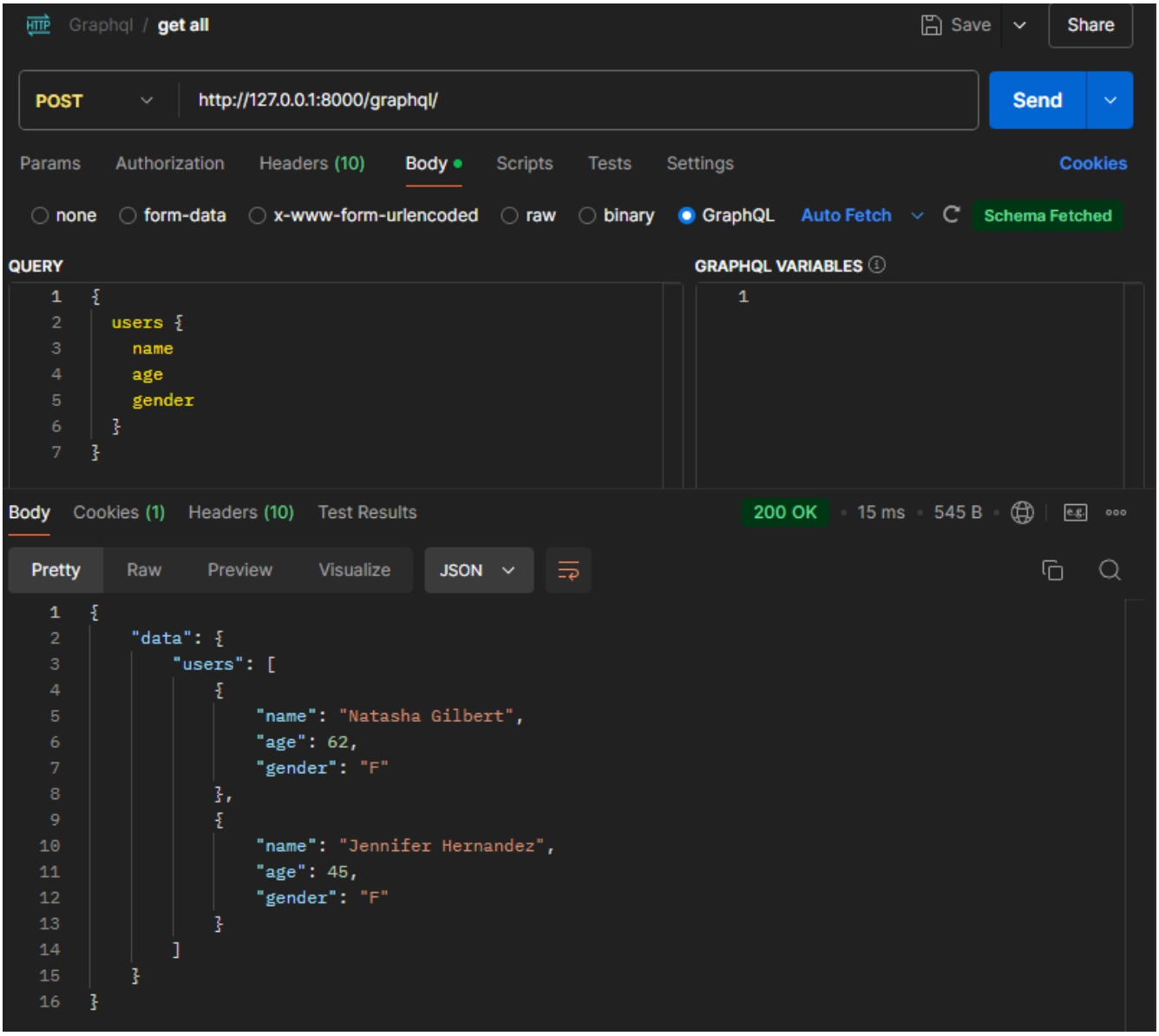

Figura 12. Respuesta de la “Aplicación GraphQL”

9. POBLACIÓN Y PRUEBA DE DATOS PARA MEDICIÓN DE TIEMPOS DE RESPUESTA

Para evaluar y comparar el rendimiento de las tecnologías de comunicación de servicios web REST y GraphQL, es necesario poblar la base de datos con los diversos volúmenes de datos definidos (para más detalles, ver Capítulo 8.2) y medir los tiempos de respuesta de las operaciones específicas.

Este proceso incluye la generación y carga de datos en la base de datos. Las siguientes secciones describen el procedimiento para poblar la base de datos y realizar pruebas de tiempos de respuesta para ambas tecnologías.

9.1. POBLACIÓN DE BASE DE DATOS

A continuación, se explicará el proceso de población de base datos y de los *endpoints* utilizados para poblar la "Aplicación REST" y la "Aplicación GraphQL".

9.1.1. POBLACIÓN DE DATOS DE LA APLICACIÓN REST

Para poblar la base de datos utilizando la "Aplicación REST", se utilizará el siguiente *endpoint*: `POST http://127.0.0.1:8000/api/fill_table/` el cual permite especificar el número de usuarios que se desean agregar a la base de datos mediante un cuerpo de solicitud en formato JSON.

```
{
    "num_entries": "CANTIDAD_USUARIOS_DESEADA"
}
```

La variable "num_entries" indica la cantidad de usuarios deseados en la base de datos para hacer pruebas. Al enviar una solicitud para añadir datos, los datos existentes se eliminarán en su totalidad para ingresar los nuevos datos.

9.1.2. PRUEBA DE *ENDPOINT* DE LA APLICACIÓN REST

El siguiente ejemplo muestra cómo agregar 20000 usuarios enviando una petición al siguiente *endpoint*: `POST http://127.0.0.1:8000/api/fill_table/` junto con el siguiente *body*.

```
{
    "num_entries": "20000"
}
```

Al enviar esta solicitud, la “Aplicación REST” procesa la petición eliminando todos los registros existentes y añade 20000 registros creados aleatoriamente a la base de datos. Estos registros se generan utilizando la librería Faker [20], que permite crear datos ficticios realistas como nombres, direcciones, correos electrónicos, entre otros. Esta librería es especialmente útil para crear conjuntos de datos de prueba de manera rápida y sencilla. Una vez que la base de datos esté poblada, se recibirá el siguiente JSON como respuesta.

```
{
    "message": "Se han insertado 20000
usuarios en la base de datos."
}
```

En la Figura 13, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

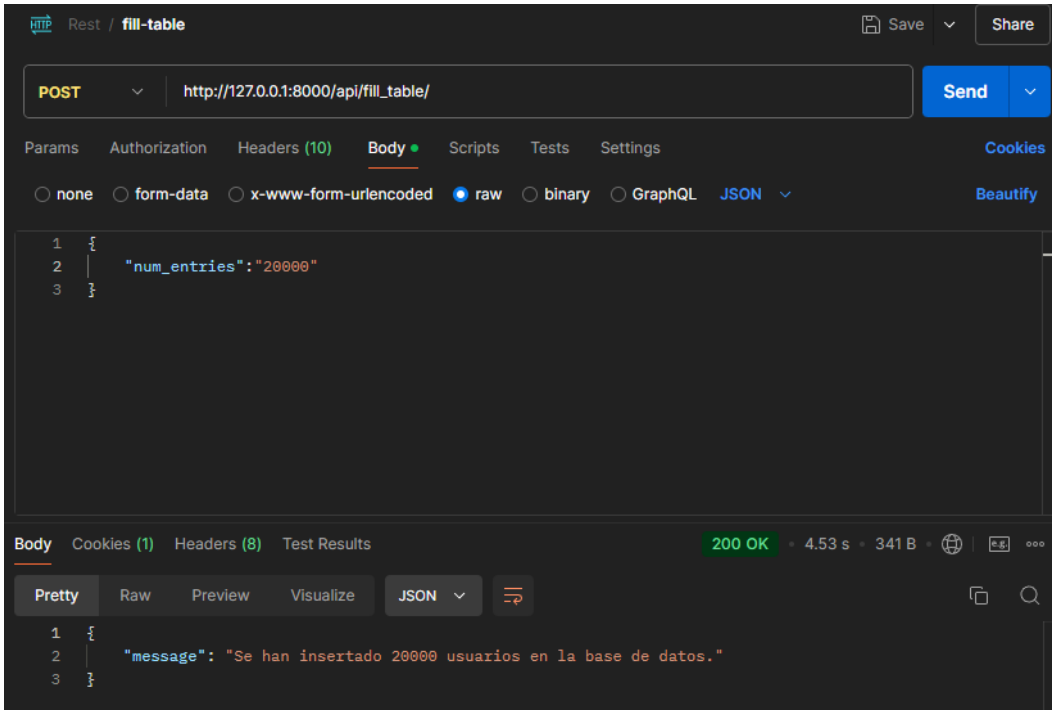

Figura 13. Prueba de *endpoint* para ingresar registros a la base de datos con REST

En la Figura 14, se observa con claridad que, al consultar la cantidad de registros en la tabla de usuarios, esta coincide exactamente con el número de usuarios que hemos agregado anteriormente y en la Figura 15 se observa una pequeña muestra de los usuarios ingresados a la base de datos.

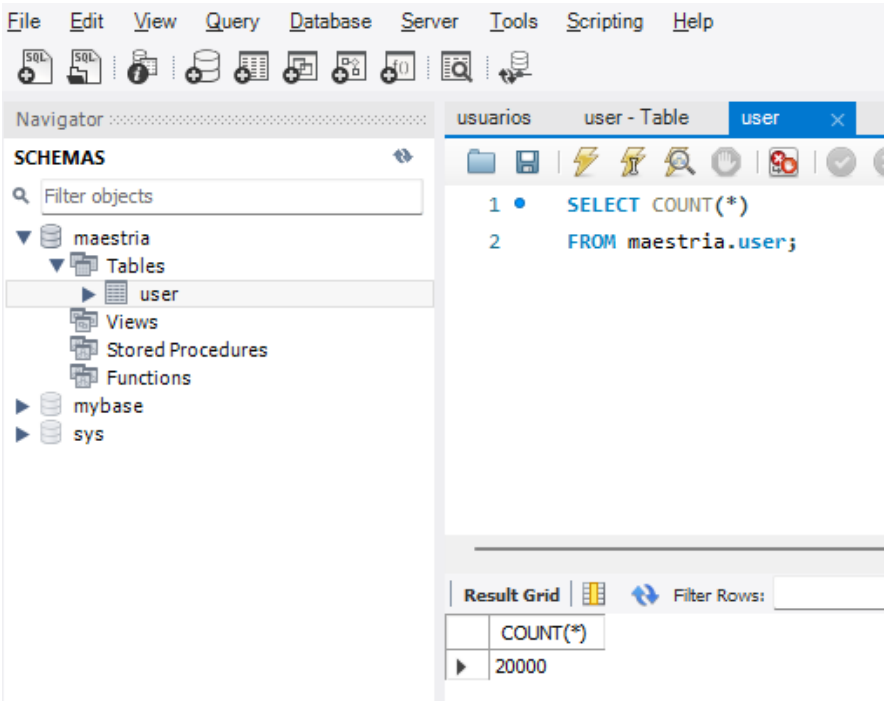

Figura 14. Base de datos con el número de registros ingresados

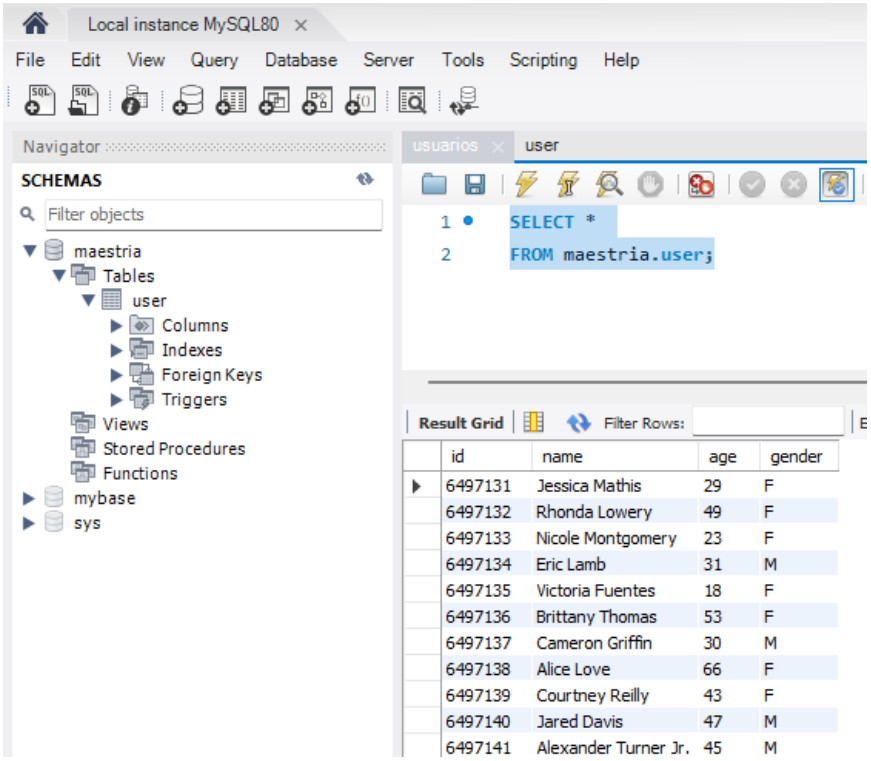

Figura 15. Muestra de los registros ingresados

9.1.3. POBLACIÓN DE DATOS DE LA APLICACIÓN GRAPHQL

Para poblar la base de datos utilizando la "Aplicación GraphQL", se utilizará el siguiente *endpoint*: `POST http://127.0.0.1:8000/graphql/` el cual permite especificar el número de usuarios que se desean agregar a la base de datos mediante una *query* en el *body* de solicitud.

```
mutation {
  fillTable(numEntries: "CANTIDAD_USUARIOS_DESEADA") {
    success
  }
}
```

La variable "numEntries" indica la cantidad de usuarios deseados en la base de datos para hacer pruebas y el campo *success* (exitoso) luego confirmará si la operación se realizó con éxito. Al enviar una solicitud para añadir datos, los datos existentes se eliminarán en su totalidad para ingresar los nuevos datos.

9.1.4. PRUEBA DE *ENDPOINT* DE LA APLICACIÓN GRAPHQL

El siguiente ejemplo muestra cómo agregar 20000 usuarios enviando una petición al siguiente *endpoint*: `POST http://127.0.0.1:8000/graphql/` junto con la siguiente query en el *body*.

```
mutation {
  fillTable(numEntries: 10000) {
    success
  }
}
```

Al enviar esta solicitud, la "Aplicación GraphQL" procesa la petición eliminando todos los registros existentes y añade 10000 registros creados aleatoriamente con la misma librería Faker [20] usada previamente en la "Aplicación REST" para ingresar registros a la base de datos.

Una vez que la base de datos esté poblada, se recibirá el siguiente JSON como respuesta.

```
{
    "data": {
        "fillTable": {
            "success": true
        }
    }
}
```

En la Figura 16, se puede observar con más detalle la solicitud enviada y su respuesta utilizando Postman.

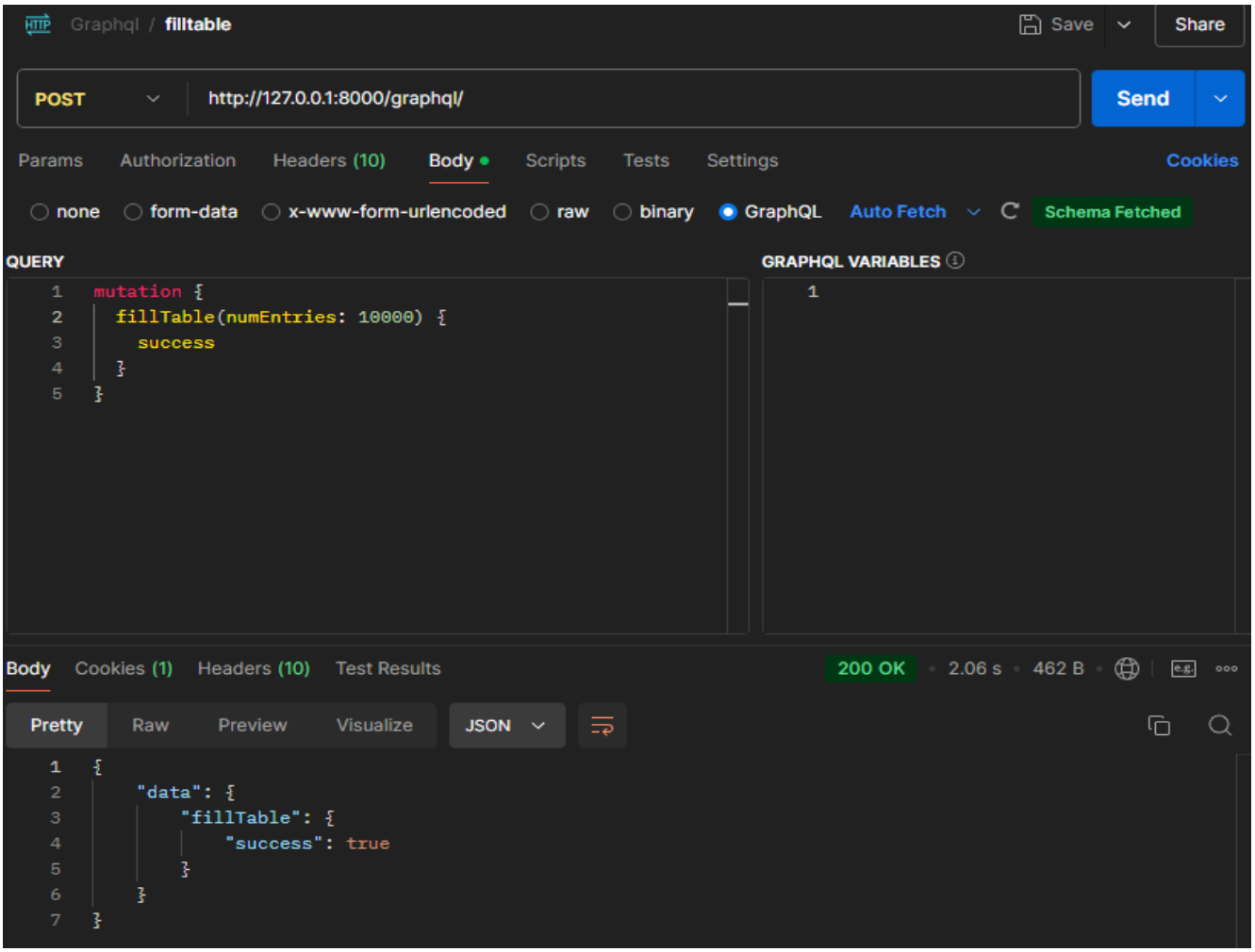

Figura 16. Prueba de *endpoint* para ingresar registros a la base de datos con GraphQL

En la Figura 17, se observa con claridad que, al consultar la cantidad de registros en la tabla de usuarios, esta coincide exactamente con el número de usuarios que hemos agregado anteriormente y en la Figura 18 se observa una pequeña muestra de los usuarios ingresados a la base de datos.

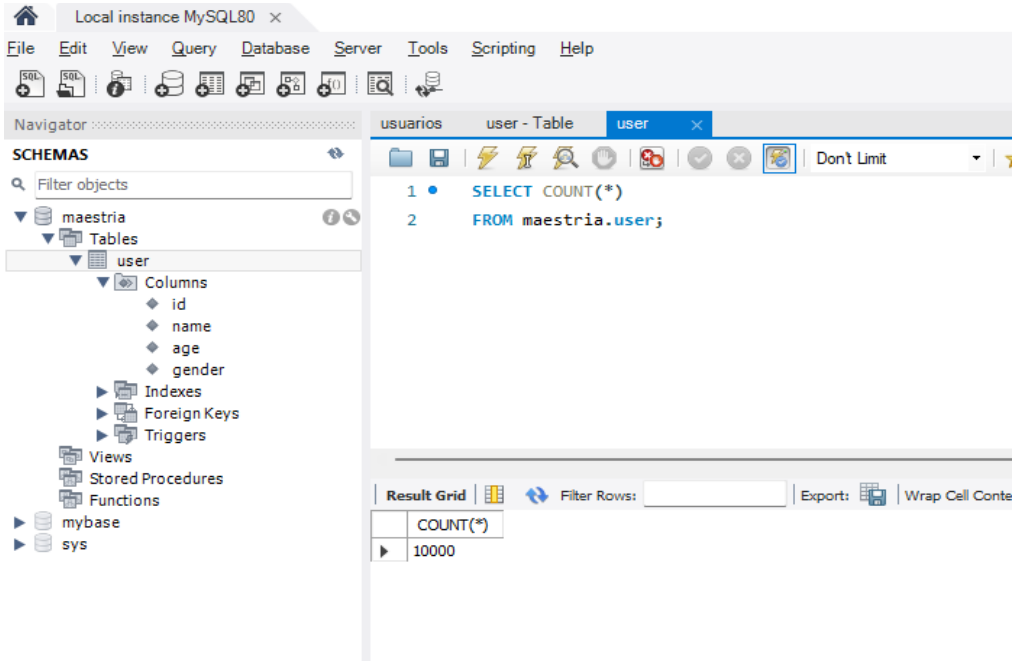

Figura 17. Base de datos con el número de registros ingresados

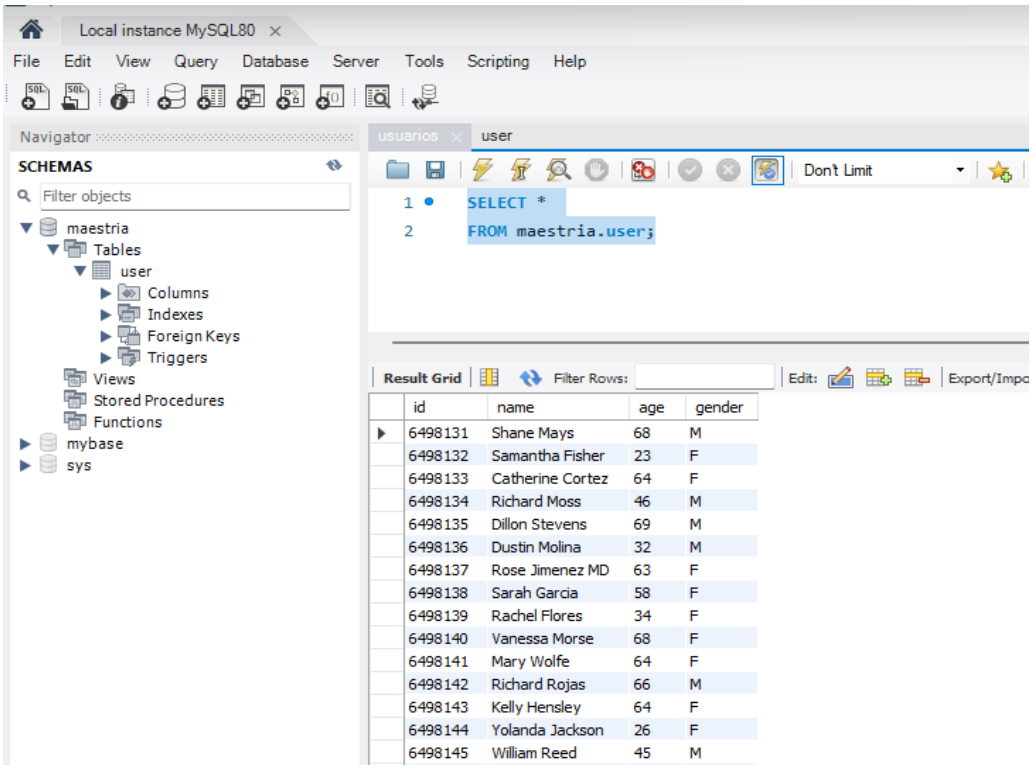

Figura 18. Muestra de los registros ingresados

9.2. MEDICIÓN DE TIEMPOS DE RESPUESTA PARA APLICACIÓN REST Y APLICACIÓN GRAPHQL

El tiempo de respuesta es uno de los requisitos no funcionales para medir el rendimiento de las aplicaciones web. Esta sección detalla la definición de las especificaciones del equipo de prueba, los casos de prueba a analizar, los resultados obtenidos para cada tipo de consulta en ambas tecnologías y dos gráficas para realizar la comparación de resultados. Las unidades de medida para el tiempo se expresarán en milisegundos, mientras que el peso se medirá en kilobytes.

9.2.1. ESPECIFICACIONES DEL EQUIPO DE PRUEBA

Para replicar las pruebas y obtener resultados similares, es importante utilizar un equipo con especificaciones similares a las empleadas en este proyecto. Las especificaciones relevantes para el análisis incluyen.

Componente	**Descripción**
Procesador	Intel Core i7 - 11370H a 3.3 GHz
Memoria RAM	24 GB
Disco Duro	M.2 SSD 500 GB
Sistema Operativo	Windows 11
Software	Python 3.8, Django 3.14

Tabla 1. Especificaciones técnicas del equipo de prueba

9.2.2. CASOS DE PRUEBA PARA ANALIZAR

La Tabla 2 presenta los casos de prueba, donde cada caso está representado por un identificador, la tecnología usada, el tipo de consulta a realizar y el volumen de los datos a enviar.

ID del Caso de Prueba	Tecnología Utilizada	Tipo de Consulta	Volumen de Datos
1	REST	Obtener todos los usuarios	Pequeño (1000 registros)
2	REST	Obtener todos los usuarios	Mediano (100000 registros)
3	REST	Obtener todos los usuarios	Grande (500000 registros)
4	REST	Obtener todos los usuarios	Muy grande (1000000 de registros)
5	REST	Filtrar usuarios por género	Pequeño (1000 registros)
6	REST	Filtrar usuarios por género	Mediano (100000 registros)
7	REST	Filtrar usuarios por género	Grande (500000 registros)
8	REST	Filtrar usuarios por género	Muy grande (1000000 de registros)
9	REST	Usuarios por género y edad	Pequeño (1000 registros)
10	REST	Usuarios por género y edad	Mediano (100000 registros)
11	REST	Usuarios por género y edad	Grande (500000 registros)
12	REST	Usuarios por género y edad	Muy grande (1000000 de registros)

13	GraphQL	Obtener todos los usuarios	Pequeño (1000 registros)
14	GraphQL	Obtener todos los usuarios	Mediano (100000 registros)
15	GraphQL	Obtener todos los usuarios	Grande (500000 registros)
16	GraphQL	Obtener todos los usuarios	Muy grande (1000000 de registros)
17	GraphQL	Filtrar usuarios por género	Pequeño (1000 registros)
18	GraphQL	Filtrar usuarios por género	Mediano (100000 registros)
19	GraphQL	Filtrar usuarios por género	Grande (500000 registros)
20	GraphQL	Filtrar usuarios por género	Muy grande (1000000 de registros)
21	GraphQL	Usuarios por género y edad	Pequeño (1000 registros)
22	GraphQL	Usuarios por género y edad	Mediano (100000 registros)
23	GraphQL	Usuarios por género y edad	Grande (500000 registros)
24	GraphQL	Usuarios por género y edad	Muy grande (1000000 de registros)

Tabla 2. Casos de prueba para la "Aplicación REST" y "Aplicación GraphQL"

Cada vez que se llena la base de datos con una nueva cantidad de usuarios deseada, la diversidad de hombres, mujeres y edades es aleatoria, por lo cual para estandarizar las pruebas y que traigan siempre la misma cantidad de usuarios para comparaciones justas se realizará de la siguiente manera.

1. Se poblará la base de datos con 1000 usuarios.
2. Se realizará la consulta de traer todos los usuarios con la "Aplicación REST".
3. Se realizará un filtro para traer solo los hombres con la "Aplicación REST".
4. Se realizará un filtro para traer solo los hombres de 30 años con "Aplicación REST".
5. Se realizará la consulta de traer todos los usuarios con la "Aplicación GraphQL".
6. Se realizará un filtro para traer solo los hombres con la "Aplicación GraphQL".
7. Se realizará un filtro para traer solo los hombres de 30 años con la "Aplicación GraphQL".
8. Finalmente se poblará la base de datos con 100000 usuarios y repite nuevamente el ciclo para tener unas pruebas estandarizadas con cada volumen de dato.

Los *endpoints* para realizar cada una de las consultas están detallados en la documentación ubicada en el GitHub de la "Aplicación REST" (https://github.com/mframosg/rest-api) y "Aplicación GraphQL" (https://github.com/mframosg/graphql-api).

9.2.3. MEDICIÓN DE TIEMPOS DE RESPUESTA DE LA APLICACIÓN REST

La Tabla 3 presenta los resultados de la prueba para la "Aplicación REST", donde cada caso está relacionado al identificador de la tabla anterior, tipo de consulta realizada, el volumen de datos a solicitar, el tiempo de consulta obtenido y el peso del JSON recibido.

ID del Caso de Prueba	Tipo de Consulta	Volumen de Datos	Tiempo obtenido	Peso del JSON de respuesta
1	Obtener todos los usuarios	Pequeño (1000 registros)	8 ms	54,29 KB
2	Obtener todos los usuarios	Mediano (100000 registros)	162 ms	5270 KB

3	Obtener todos los usuarios	Grande (500000 registros)	860 ms	26350 KB
4	Obtener todos los usuarios	Muy grande (1000000 de registros)	1788 ms	52710 KB
5	Filtrar usuarios por género	Pequeño (1000 registros)	8 ms	25,66 KB
6	Filtrar usuarios por género	Mediano (100000 registros)	90 ms	2630 KB
7	Filtrar usuarios por género	Grande (500000 registros)	498 ms	13150 KB
8	Filtrar usuarios por género	Muy grande (1000000 de registros)	1106 ms	26280 KB
9	Usuarios por género y edad	Pequeño (1000 registros)	6 ms	0,663 KB
10	Usuarios por género y edad	Mediano (100000 registros)	37 ms	50,5 KB
11	Usuarios por género y edad	Grande (500000 registros)	161 ms	256,21 KB
12	Usuarios por género y edad	Muy grande (1000000 de registros)	365 ms	506,17 KB

Tabla 3. Mediciones de tiempos de respuesta en la "Aplicación REST"

9.2.4. MEDICIÓN DE TIEMPOS DE RESPUESTA DE LA APLICACIÓN GRAPHQL

La Tabla 4 presenta los resultados de la prueba para la "Aplicación GraphQL", donde cada caso está relacionado al identificador de la tabla anterior, tipo de consulta realizada, el volumen de datos a solicitar, el tiempo de consulta obtenido y el peso del JSON recibido.

ID del Caso de Prueba	Tipo de Consulta	Volumen de Datos	Tiempo obtenido	Peso del JSON de respuesta
13	Obtener todos los usuarios	Pequeño (1000 registros)	29 ms	33,95 KB
14	Obtener todos los usuarios	Mediano (100000 registros)	1643 ms	3270 KB
15	Obtener todos los usuarios	Grande (500000 registros)	8140 ms	16340 KB
16	Obtener todos los usuarios	Muy grande (1000000 de registros)	16840 ms	32690 KB
17	Filtrar usuarios por género	Pequeño (1000 registros)	25 ms	22,14 KB
18	Filtrar usuarios por género	Mediano (100000 registros)	1178 ms	2250 KB
19	Filtrar usuarios por género	Grande (500000 registros)	5790 ms	11240 KB
20	Filtrar usuarios por género	Muy grande (1000000 de registros)	22850 ms	22470 KB
21	Usuarios por género y edad	Pequeño (1000 registros)	16 ms	0,776 B

22	Usuarios por género y edad	Mediano (100000 registros)	66 ms	43,39 KB
23	Usuarios por género y edad	Grande (500000 registros)	258 ms	219,21 KB
24	Usuarios por género y edad	Muy grande (1000000 de registros)	556 ms	432,89 KB

Tabla 4. Mediciones de tiempos de respuesta en la "Aplicación GraphQL"

9.2.5. COMPARACIÓN DE LOS RESULTADOS DE LOS TIEMPOS DE RESPUESTAS DE LAS APLICACIONES ANALIZADAS

En la Figura 19 se presentan gráficamente los resultados de los tiempos de respuesta obtenidos. La primera gráfica ilustra los tiempos de respuesta de la "Aplicación REST" en milisegundos (ms) a través de distintos volúmenes de datos definidos, considerando todas las operaciones evaluadas, lo que permite un análisis detallado de su rendimiento. En la parte inferior, se muestra la misma información para la "Aplicación GraphQL", facilitando así una comparación directa y precisa entre ambas tecnologías.

Posteriormente en la Figura 20 se presentan los resultados correspondientes al peso de las respuestas en formato JSON para las mismas operaciones de la Figura 19, medido en kilobytes (KB). La gráfica superior muestra los datos para la "Aplicación REST" y la gráfica inferior refleja los resultados para GraphQL, posibilitando así un análisis comparativo del peso de las respuestas generadas por cada aplicación.

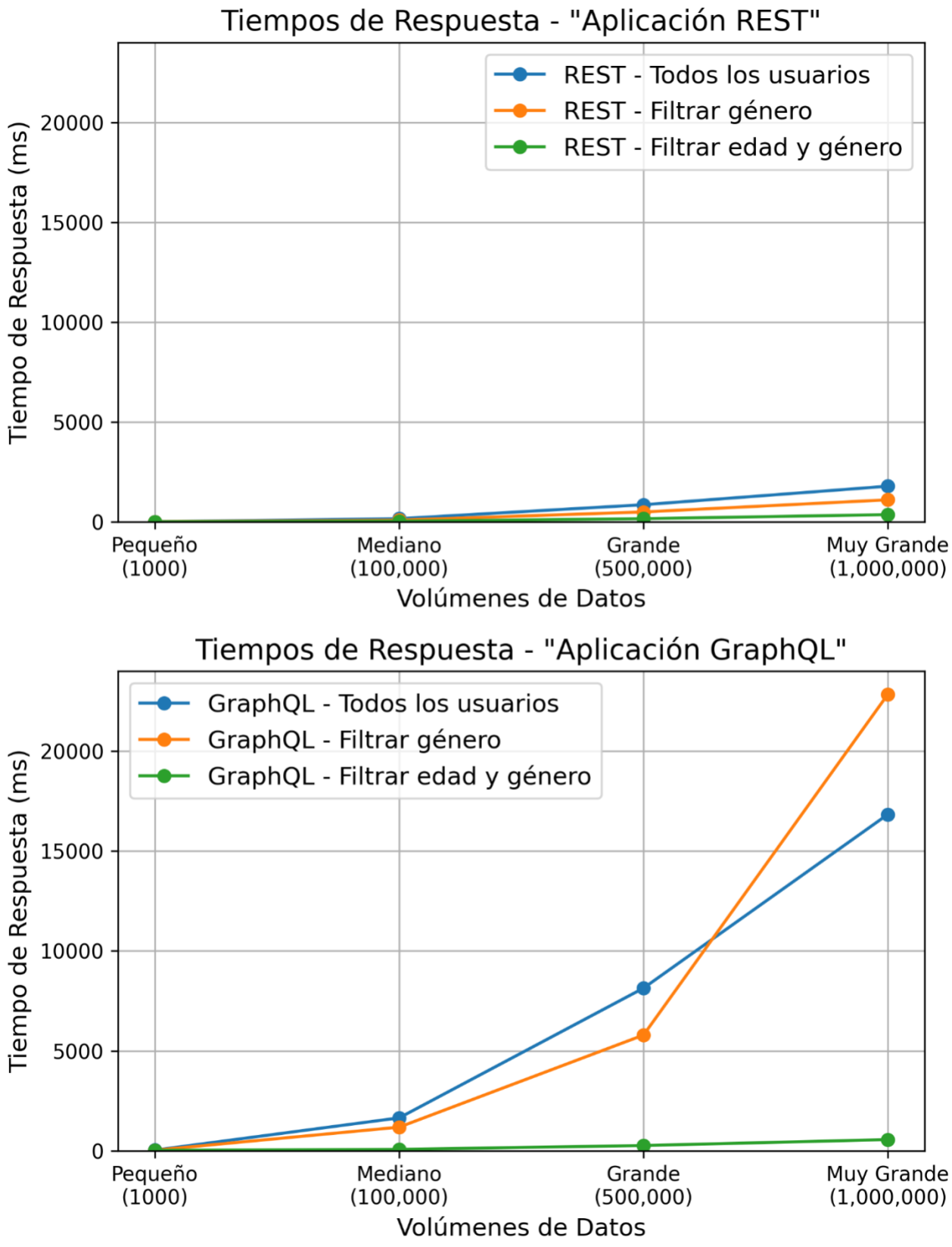

Figura 19. Comparativa REST y GraphQL en tiempos de respuesta en ms

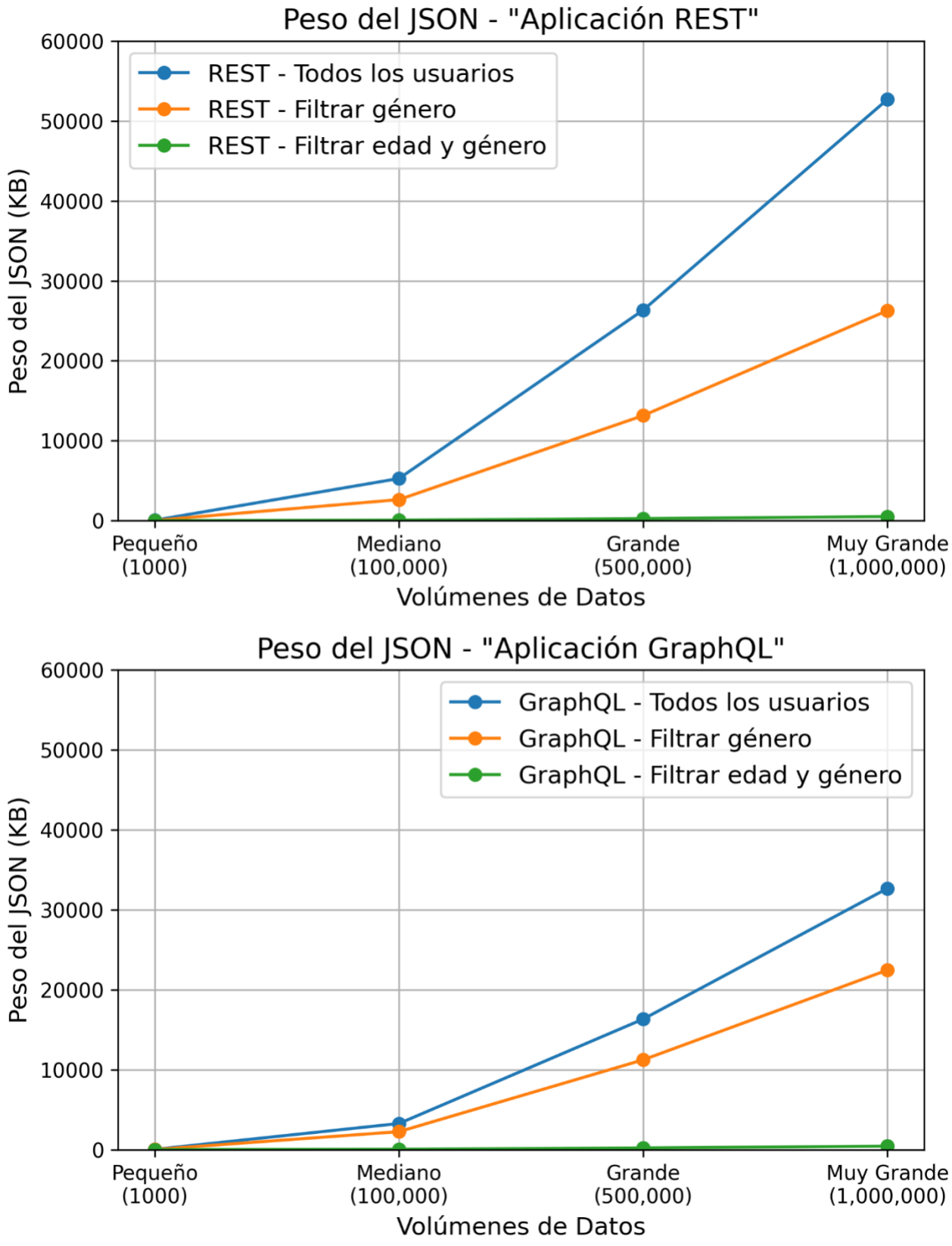

Figura 20. Comparativa REST y GraphQL en peso de la respuesta en KB

10. RESULTADOS

En este capítulo se analizan las gráficas mostradas en el capítulo anterior, que ilustran las diferencias en rendimiento entre las tecnologías REST y GraphQL. Estas visualizaciones permiten evaluar de manera clara y efectiva los puntos fuertes de cada enfoque, proporcionando información valiosa para la toma de decisiones en función de las necesidades específicas del proyecto y del volumen de datos a manejar.

Un hallazgo significativo es el punto de inflexión que se observa a partir de 100000 datos, donde se evidencia un aumento notable en el tamaño de la respuesta en formato JSON de REST, así como en los tiempos de respuesta de GraphQL. Esta tendencia respalda las conclusiones de Brito, Mombach y Valente, quienes destacan que GraphQL puede reducir significativamente el tamaño de los documentos JSON, resultando en una transferencia de datos más eficiente, aunque esto conlleva un incremento en los tiempos de respuesta [1]. A continuación, se realiza una comparación de los tiempos de respuesta de ambas aplicaciones, y luego se realiza una discusión acerca de los hallazgos de la tesis.

10.1. COMPARACIÓN DE LOS TIEMPOS DE RESPUESTA

La comparación de los tiempos de respuesta entre las aplicaciones REST y GraphQL nos permite destacar la siguiente información.

REST:

- En general, las consultas REST tienen tiempos de respuesta más rápidos, especialmente con grandes volúmenes de datos.
- Al obtener todos los usuarios con un volumen muy grande (1000000 de registros) toma 1788 ms en REST, mientras que la misma consulta en GraphQL toma 16840 ms.

GraphQL:

- GraphQL tiende a tener tiempos de respuesta más largos en comparación con REST, especialmente con grandes volúmenes de datos ya que requiere procesar la lógica para interpretar consultas específicas y resolver relaciones entre datos, lo que puede aumentar el tiempo de

respuesta en el servidor. Sin embargo, en consultas más específicas como filtrar usuarios por género y edad, GraphQL sigue mostrando tiempos de respuesta cortos para volúmenes pequeños y medianos.

Aunque GraphQL ofrece flexibilidad en las consultas, REST demuestra un rendimiento superior en términos de tiempo de respuesta, particularmente cuando se manejan grandes volúmenes de datos.

10.2. DISCUSIÓN

¿ES MEJOR REST EN TODOS LOS ESCENARIOS?

Para determinar en qué intervalos de datos una tecnología se destaca sobre la otra, es crucial examinar tanto los tiempos de respuesta como el peso del JSON en diferentes volúmenes de datos. A continuación, se detallan los resultados según el tamaño de los datos.

Pequeños Volúmenes de Datos (1000 registros):

- REST muestra tiempos de respuesta más rápidos.
- GraphQL presenta un JSON de respuesta ligeramente más liviano.

Medianos Volúmenes de Datos (100000 registros):

- REST continúa destacando en tiempos de respuesta.
- GraphQL tiene ventajas en el peso del JSON, aunque sus tiempos de respuesta son considerablemente más altos.

Grandes Volúmenes de Datos (500000 registros):

- REST es notablemente más rápido.
- GraphQL mantiene una ventaja en el peso del JSON.

Muy Grandes Volúmenes de Datos (1000000 registros):

- REST tiene una ventaja significativa en tiempos de respuesta.
- GraphQL continúa siendo más eficiente en el peso del JSON.

¿CÓMO DECIDIR CUÁL TECNOLOGÍA DE COMUNICACIÓN DE SERVICIOS WEB UTILIZAR EN MI PROYECTO?

Para aplicaciones donde el tiempo de respuesta es crítico, REST es la tecnología ideal, especialmente en volúmenes de datos medianos a muy grandes. Para aplicaciones donde la eficiencia del uso de datos y la flexibilidad de las consultas son importantes, GraphQL ofrece ventajas, especialmente en términos de peso del JSON, siendo más notable en volúmenes de datos grandes a muy grandes.

Vohra y Manuaba [21] destacaron que, aunque REST y GraphQL pueden ofrecer un rendimiento similar en general, las diferencias se vuelven notables al manejar consultas GET, lo que es crucial para seleccionar la tecnología más adecuada según el contexto. En apoyo a esto, la investigación de Vogel, Weber y Zirpins muestra que GraphQL ofrece ventajas en la simplificación de las interfaces de API y en la mejora del rendimiento en tiempo de ejecución [22]. No obstante, también se debe considerar el costo asociado con las consultas en GraphQL. En este contexto, Mavroudeas y colaboradores subrayan los desafíos para estimar el costo de las consultas debido a la compleja estructura anidada de GraphQL [23].

Vadlamani et al. [24] examinaron GraphQL y REST desde una perspectiva amplia, considerando tanto aspectos cuantitativos como cualitativos. Su investigación indica que ambas tecnologías poseen conjuntos distintos de ventajas, adaptándose a contextos específicos de cada aplicación, sin que una supere absolutamente a la otra. Por lo tanto, la elección entre REST y GraphQL debe depender de los requisitos particulares del proyecto. Como se pudo evidenciar en este estudio REST puede ofrecer tiempos de respuesta más rápidos para grandes volúmenes de datos, mientras que GraphQL, aunque más lento, proporciona mayor flexibilidad en las consultas y eficiencia en el tamaño de las respuestas.

11. CONCLUSIONES

En este capítulo se presentan las conclusiones del estudio comparativo entre REST y GraphQL, así como las propuestas para investigaciones futuras. A lo largo de este trabajo, se compararon los tiempos de respuesta de ambas tecnologías de comunicación mediante el diseño agnóstico a la tecnología de una "Aplicación Base", que fue implementada en dos versiones: una utilizando REST y otra con GraphQL. Se llevaron a cabo pruebas de rendimiento con volúmenes de datos crecientes, lo que permitió evidenciar las fortalezas y limitaciones de cada tecnología en distintos escenarios de uso. A partir de los resultados obtenidos, se presentan las conclusiones del estudio y unas recomendaciones a trabajos futuros.

- **Comparación general de REST y GraphQL:** Los resultados obtenidos demuestran que REST ofrece tiempos de respuesta significativamente más rápidos en escenarios con grandes volúmenes de datos en comparación con GraphQL. Esto lo convierte en una opción más adecuada cuando la velocidad es un factor crítico en aplicaciones que manejan grandes cantidades de información.

- **Escalabilidad y rendimiento:** REST mostró un mejor rendimiento en todos los escenarios probados, particularmente cuando se incrementaron los volúmenes de datos. Lo que afecta la viabilidad de GraphQL en aplicaciones con demandas de datos masivas.

- **Flexibilidad de consultas:** A pesar de los tiempos de respuesta más largos, GraphQL se destacó por su capacidad de manejar consultas complejas con un solo *endpoint*, lo que reduce la sobrecarga en términos de *endpoints* múltiples que suelen caracterizar a REST. Esto es especialmente útil en aplicaciones que requieren una alta flexibilidad en las consultas y una optimización del tamaño de los datos transferidos.

- **Impacto del volumen de datos:** A medida que se incrementó el volumen de datos, REST mantuvo su ventaja en términos de tiempos de respuesta, siendo notablemente más eficiente en la gestión de grandes solicitudes (hasta 1 millón de registros). Sin embargo, en volúmenes más pequeños, la diferencia de rendimiento entre REST y GraphQL fue menos significativa.

- **Peso de la respuesta:** Si bien los tiempos de respuesta fueron más largos en GraphQL, el tamaño de los archivos JSON generados por las consultas fue menor en comparación con REST. Esto

sugiere que GraphQL podría ser más ventajoso en aplicaciones donde el peso de la respuesta tiene un impacto directo en el ancho de banda o en la eficiencia de la red.

- **Selección tecnológica basada en casos de uso:** La elección entre REST y GraphQL debe basarse en los requisitos específicos del proyecto. REST es ideal para aplicaciones que priorizan el rendimiento y manejan grandes volúmenes de datos, mientras que GraphQL es preferible cuando la flexibilidad en las consultas y la optimización del tamaño de los datos es más importante que la velocidad de respuesta.

- **Consideraciones Futuras:** El estudio demuestra que REST sigue siendo más eficiente en términos de rendimiento general, pero futuras investigaciones podrían centrarse en optimizar aún más el rendimiento de GraphQL con técnicas como la paginación, la cual no se abordaron en este trabajo.

Es importante mencionar que las variables de tiempo y peso de la respuesta no son los únicos requisitos no funcionales para considerar en proyectos reales. Existen técnicas adicionales que pueden mejorar el rendimiento de estas tecnologías de comunicación de servicios web, tales como la optimización y particionamiento de consultas, el uso de bases de datos distribuidas y NoSQL, la paginación en las respuestas, entre otras que no fueron tenidas en cuenta en este estudio. Sin embargo, este estudio se enfocó en llevar a REST y GraphQL a sus límites sin considerar estas técnicas.

Como continuación de esta investigación, sería relevante explorar la implementación de técnicas como la paginación o el uso de caché, lo que podría generar mejoras en el desempeño al reducir la cantidad de datos procesados por cada solicitud. Además, la integración de bases de datos NoSQL o distribuidas podría ser otra línea de investigación, ya que estas arquitecturas son particularmente eficientes en el manejo de grandes volúmenes de datos y podrían optimizar aún más las consultas en ambos enfoques.

REFERENCIAS

[1] G. Brito, T. Mombach, y M. T. Valente, "Migrating to GraphQL: A practical assessment," 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019.

[2] P. Margański y B. Pańczyk, "Comparative analysis between REST and GraphQL," Journal of Computer Sciences Institute, vol. 19, 2021.

[3] Amazon Web Services, Inc., "¿Qué es una API? - Explicación de interfaz de programación de aplicaciones," Amazon Web Services, Inc., Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://aws.amazon.com/es/what-is/api/.

[4] Django Software Foundation. "Django Overview". Django Software Foundation. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://docs.djangoproject.com/en/3.2/intro/overview/.

[5] GitHub, Inc. "About GitHub". GitHub, Inc. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://docs.github.com/en/github/getting-started-with-github/about-github.

[6] Facebook Inc. "GraphQL specification (draft)". Facebook Inc. Accedido: Mar. 28, 2024. [En línea]. Disponible en: https://facebook.github.io/graphql/draft/.

[7] JSON.org. "Introducing JSON". JSON.org. Accedido: Mar. 24, 2024. [En línea]. Disponible en: http://json.org/.

[8] MySQL. "Introduction to MySQL". MySQL Documentation. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://dev.mysql.com/doc/refman/8.0/en/introduction.html.

[9] Postman, Inc. "Postman Learning Center". Postman, Inc. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://learning.postman.com/.

[10] Red Hat, Inc., "¿Qué es una API y cómo funciona?," Red Hat, Inc., Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces.

[11] IBM, "Web Services," IBM Documentation. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://www.ibm.com/docs/es/was/9.0.5?topic=services-web.

[12] K. Peffers, T. Tuunanen, M. A. Rothenberger, y S. Chatterjee, “A Design Science Research Methodology for Information Systems Research,” Journal of Management Information Systems, vol. 24, no. 3, pp. 45–77, 2008

[13] N. Dragoni, S. Giallorenzo, A.L. Lafuente, et al., “Microservices: yesterday, today, and tomorrow,” Present Ulterior Softw Eng., pp. 195–216, 2017. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://doi.org/10.1007/978-3-319-67425-4_12.

[14] S. Newman, Building Microservices: Designing Fine-Grained Systems. Sebastopol, CA: O’Reilly, Inc., 2014. [En línea]. Disponible en: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/

[15] Nadareishvili, R. Mitra, M. McLarty y M. Amundsen, Microservice Architecture: Aligning Principles, Practices and Culture, 1ª ed. Sebastopol, CA: O’Reilly Media, Inc., p. 67, 2016. En línea]. Disponible en: https://www.oreilly.com/library/view/microservice-architecture/9781491956328/

[16] Amazon Web Services, Inc., “Comparing GraphQL and REST,” Amazon Web Services, Inc. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://aws.amazon.com/compare/the-difference-between-graphql-and-rest/.

[17] GitHub, Inc., “Octoverse 2022: Top Programming Languages,” GitHub, Inc. Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://octoverse.github.com/2022/top-programming-languages.

[18] R. T. Fielding y R. N. Taylor, “Principled design of the modern web architecture,” 22nd International Conference on Software Engineering (ICSE), pp. 407-416, 2000.

[19] G. Brito y M. T. Valente, “REST vs GraphQL: A Controlled Experiment,” 2020 IEEE International Conference on Software Architecture (ICSA), pp. 81-91, 2020.

[20] Faker, “Faker: A Python package that generates fake data,” Python Package Index, Accedido: Mar. 24, 2024. [En línea]. Disponible en: https://pypi.org/project/Faker/.

[21] N. Vohra y I. B. K. Manuaba, “Implementation of REST API vs. GraphQL in Microservice Architecture,” Proc. of the International Conference 2022 on Management and Information Technology (ICIMTech), 2022.

[22] M. Vogel, S.Weber, C. Zirpins, “Experiences in migrating web services to GraphQL,” International Conference on Service-Oriented Computing, 2017.

[23] G. Mavroudeas, G. Baudart, A. Cha, M. Hirzel, et al., “Learning GraphQL *Query* Cost,” 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1146-1150, 2021. [En línea]. Disponible en: https://doi.org/10.1109/ASE51524.2021.9678513.

[24] S. L. Vadlamani, B. Emdon, J. Arts y O. Baysal, “Can GraphQL replace REST? A study of their efficiency and viability,” Proc. of the IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), 2021.