UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Software

Tema:

"COMPARACIÓN DE LA EFICIENCIA ENTRE REST Y GRAPHQL, MEDIANTE EL ANÁLISIS DE TIEMPO DE RESPUESTA Y LA APLICACIÓN DE ASPECTOS DE LA ISO 25023, PARA DETERMINAR LA TECNOLOGÍA MÁS EFICAZ EN EL DESARROLLO DE APIS"

Trabajo de grado previo a la obtención del título de Ingeniero de Software presentado ante la ilustre Universidad Técnica del Norte.

Autor:

Anthony Julian Benavides Paillacho

Director:

PhD. José Antonio Quiña Mera

Ibarra – Ecuador

2025

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN
A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DE CONTACTO			
CÉDULA DE IDENTIDAD:	0401939749		
APELLIDOS Y NOMBRES:	Benavides Paillacho Anthony Julian		
DIRECCIÓN:	Santha Martha de Cuba - Tulcán		
EMAIL:	ajbenavidesp@utn.edu.ec		
TELÉFONO FIJO:	N/A	**TELÉFONO MÓVIL:**	0960296041

DATOS DE LA OBRA	
TÍTULO:	Comparación de la eficiencia entre REST y GraphQL, mediante el análisis de tiempo de respuesta y la aplicación de aspectos de la ISO 25023, para determinar la tecnología más eficaz en el desarrollo de APIs
AUTOR (ES):	Anthony Julian Benavides Paillacho
FECHA DE APROBACIÓN:	03 de septiembre del 2025
PROGRAMA:	PREGRADO
TITULO POR EL QUE OPTA:	Ingeniero en Software
DIRECTOR:	PhD. José Antonio Quiña Mera
ASESOR :	MSc. Alexander Guevara Vega

2. CONSTANCIAS

El autor manifiesta que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto la obra es original y que es el titular de los derechos patrimoniales, por lo que asume la responsabilidad sobre el contenido de la misma y saldrá en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 05 días del mes de septiembre de 2025

EL AUTOR:

Estudiante
Anthony Julian Benavides Paillacho
0401939749

CERTIFICACIÓN DIRECTOR

Ibarra 04 de septiembre del 2025

CERTIFICACIÓN DIRECTOR DEL TRABAJO DE TITULACIÓN

Por medio del presente yo, **José Antonio Quiña Mera**, certifico que el Sr. **Anthony Julian Benavides Paillacho** portador de la cedula de ciudadanía número **0401939749**, ha trabajado en el desarrollo del proyecto de grado **"Comparación de la eficiencia entre REST y GRAPHQL, mediante el análisis de tiempo de respuesta y la aplicación de aspectos de la ISO 25023, para determinar la tecnología más eficaz en el desarrollo de APIs",** previo a la obtención del Título de Ingeniero en Software realizado con interés profesional y responsabilidad que certifico con honor de verdad.

Es todo en cuanto puedo certificar a la verdad

Atentamente

PhD. José Antonio Quiña Mera

DIRECTOR DE TRABAJO DE GRADO

DEDICATORIA

Dedico este trabajo a mis padres, por su apoyo incondicional a la distancia y por preocuparse siempre por mí, enseñándome con su ejemplo la importancia del esfuerzo y la constancia. A mi novia, por ser mi apoyo y animarme a seguir adelante en los momentos difíciles, brindándome ánimo y confianza para no rendirme.

A mis hermanos, por considerarme su apoyo y meta a alcanzar, motivándome a esforzarme cada día más y a superarme continuamente. Y a toda mi familia, que siempre me ha brindado su amor, comprensión y apoyo incondicional, sirviéndome de inspiración para culminar con éxito mis estudios y este trabajo de grado.

AGRADECIMIENTO

Agradezco primeramente a Dios, por ser mi guía, por darme la vida, la salud y la sabiduría necesarias para llegar hasta aquí, y por brindarme la fortaleza para superar cada reto en el camino.

A mi madre, Guadalupe Paillacho, y a mi padre, Wilmer Proaño, por su amor incondicional, por creer siempre en mí, incluso en los momentos más difíciles, y por enseñarme el verdadero valor del esfuerzo, la dedicación y la humildad. Todo lo que he logrado es gracias a ustedes y para ustedes, también a mis hermanos, Matías y Samira, quienes me han visto como un ejemplo y un apoyo; ustedes me inspiran a seguir luchando y a esforzarme cada día más.

A mi novia, Cristina, por ser un pilar fundamental en este proceso, gracias por tu tiempo, por tus consejos, por alentarme a no rendirme y por recordarme siempre la importancia de seguir adelante sin perder el rumbo. Tu compañía ha sido clave para culminar con éxito esta etapa.

Finalmente, a mi tutor de tesis, PhD. Antonio Quiña, por su paciencia, guía y apoyo en este proceso, gracias a sus conocimientos y orientación fue posible la realización de este trabajo.

ÍNDICE DE CONTENIDOS

Contenido

UNIVERSIDAD TÉCNICA DEL NORTE i
Autor: i
Director: i
CERTIFICACIÓN DIRECTOR iv
DEDICATORIA v
AGRADECIMIENTO 1
ÍNDICE DE FIGURAS 4
ÍNDICE DE TABLAS 7
RESUMEN 8
ABSTRACT 9
Introducción 10
Problema 10
Antecedentes: 10
Situación Actual: 10
Prospectiva: 10
Planteamiento del Problema: 11
Objetivos 11
Objetivo General 11
Objetivos Específicos 11
Alcance 12
Justificación 13
Justificación Tecnológica 13
Justificación Práctica 14
I. Capítulo 1: Marco Teórico 14
1.1 Microservicios y Arquitectura de APIs 14
1.1.1 Introducción a los Microservicios 14
1.1.2 Las APIs 15
1.1.3 Arquitectura REST 16
1.1.4 Arquitectura GraphQL 17
1.1.5 Comparativa inicial entre REST y GraphQL 18
1.2 Normativa ISO/IEC 25023 19
1.2.1 Introducción a la norma 19
1.2.2 Métrica de tiempo medio de respuesta 19

1.3 Investigación Bibliográfica 19
1.3.1 Tecnologías más usadas con REST 20
1.3.2 Tecnologías más usadas con GraphQL 23
1.4 Experimentación en la Ingeniería de Software 24
1.5 Herramientas 24
1.5.1 JavaScript 24
1.5.2 Node.js 24
1.5.3 Express.js 25
1.5.4 Python 25
1.5.5 Flask 25
1.5.6 Java 25
1.5.7 Spring Boot 25
1.5.8 Go (Golang) 25
1.5.9 Gin 25
1.5.10 PostgreSQL 25
II. Capítulo 2: Experimento Computacional 26
2.1 Diseño 26
2.2 Desarrollo de la Base de Datos 28
2.2.1 Distribución del Experimento 29
2.3 Desarrollo de las APIs REST 30
2.3.1 API - JavaScript, Express.js y Node.js 30
2.3.2 API - Python y Flask 36
2.3.3 API - Java y SpringBoot 41
2.4 Desarrollo de las APIs GraphQL 50
2.4.1 API - JavaScript, Express y Node.js 50
2.4.2 API - Python y Flask 55
2.4.3 API - Go y Gin 60
2.5 Desarrollo del Cliente 66
2.5.1 Experimento computacional REST 67
2.5.2 Casos de uso REST 72
2.5.3 Experimento computacional GraphQL 74
2.5.4 Casos de uso GraphQL 78
2.5.5 Consulta y Guardado de datos 81
2.5.6 Pruebas de funcionamiento de las APIs 83
2.6 Ejecución del Experimento Computacional 86
III. Capítulo 3: Análisis de los resultados 87

3.1 Datos recolectados de las seis APIs ... 87
3.2 Comparativa de las APIs REST ... 88
3.2.1 Comparación para el Caso de Uso 1 ... 88
3.2.2 Comparación para el Caso de Uso 2 ... 89
3.2.3 Comparación para el Caso de Uso 3 ... 89
3.2.4 Comparación para el Caso de Uso 4 ... 90
3.2.5 Comparación para el Caso de Uso 5 ... 91
3.3 Comparativa de las APIs GraphQL ... 92
3.3.1 Comparación para el Caso de Uso 1 ... 92
3.3.2 Comparación para el Caso de Uso 2 ... 93
3.3.3 Comparación para el Caso de Uso 3 ... 93
3.3.4 Comparación para el Caso de Uso 4 ... 94
3.3.5 Comparación para el Caso de Uso 5 ... 95
3.4 Comparativa de REST con GraphQL ... 98
3.4.1 Comparación para el Caso de Uso 1 ... 98
3.4.2 Comparación para el Caso de Uso 2 ... 99
3.4.3 Comparación para el Caso de Uso 3 ... 100
3.4.4 Comparación para el Caso de Uso 4 ... 101
3.4.5 Comparación para el Caso de Uso 5 ... 102
Conclusiones ... 102
Recomendaciones ... 103
Bibliografía ... 104
Anexos ... 107
Tiempos de Respuesta – REST ... 107
Tiempos de Respuesta – GraphQL ... 119

ÍNDICE DE FIGURAS

Figura 1. Árbol de problemas ... 11
Figura 2. Diagrama del Alcance ... 13
Figura 3 Arquitectura REST ... 17
Figura 4 Arquitectura GraphQL ... 18
Figura 5 Dataset REST y GraphQL ... 20
Figura 6 Estadísticas de Stack Overflow 2024 ... 21
Figura 7 Estadísticas de GitHub 2024 ... 22
Figura 8 Experimento Computacional ... 27
Figura 9 Diagrama Entidad-Relación ... 29
Figura 10 Instanciación del Servidor Express - JS ... 31
Figura 11 Conexión a la base de datos - JS ... 31

Figura 12 Método GET Usuarios - JS 32
Figura 13 Método GET Publicaciones - JS 32
Figura 14 Método GET Comentarios - JS 33
Figura 15 Método GET Respuestas - JS 34
Figura 16 Método GET Reacciones - JS 34
Figura 17 Rutas REST - JS 35
Figura 18 Instanciación del servidor Flask - Py 36
Figura 19 Configuración del servidor y sus rutas HTTP - Py 36
Figura 20 Conexión a la Base de Datos - Py 37
Figura 21 Método GET Usuarios - Py 37
Figura 22 Método GET Publicaciones - Py 38
Figura 23 Método GET Comentarios - Py 38
Figura 24 Método GET Respuestas - Py 39
Figura 25 Método GET Reacciones – Py 39
Figura 26 Ruta GET Usuarios - Py 40
Figura 27 Ruta GET Publicaciones - Py 40
Figura 28 Ruta GET Comentarios - Py 41
Figura 29 Ruta GET Reacciones - Py 41
Figura 30 Página principal de spring initialzr 42
Figura 31 Datos del Proyecto 43
Figura 32 Dependencias del Proyecto 43
Figura 33 Carpetas del Proyecto 44
Figura 34 Carpeta Resources 44
Figura 35 Página de descarga de Maven 45
Figura 36 Carpeta de Maven 45
Figura 37 Pasos para agregar una variable de entorno 46
Figura 38 Pasos para editar la variable Path 46
Figura 39 Comprobación de la instalación de Maven 47
Figura 40 Credenciales de acceso para la base de datos 47
Figura 41 Código de JdbcTemplate 48
Figura 42 Método GET Usuarios - Java 48
Figura 43 Método GET Publicaciones - Java 48
Figura 44 Método GET Comentarios, Respuestas y Reacciones - Java 49
Figura 45 Código de services 49
Figura 46 Código de services 49
Figura 47 Manejo de las rutas HTTP 50
Figura 48 Manejo de las rutas HTTP 50
Figura 49 Manejo de las rutas HTTP 50
Figura 50 Código para el servidor de la API 51
Figura 51 Código para la definición de types 52
Figura 52 Código para la definición de types 52
Figura 53 Definición de las querys 53
Figura 54 Método GET Usuarios – JavaScript GraphQL 53
Figura 55 Conexión a la base de datos 54
Figura 56 Método GET Publicaciones – JavaScript GraphQL 54
Figura 57 Método GET Comentarios – JavaScript GraphQL 54
Figura 58 Método GET Publicaciones y Reacciones – JavaScript GraphQL 55
Figura 59 Instanciación del servidor 56

Figura 60 Clases Usuario y Comentario 57
Figura 61 Clases Respuesta y Reaccion 57
Figura 62 Clase Query 58
Figura 63 Método getUsuarios 58
Figura 64 Método getPublicaciones 59
Figura 65 Métodos get Comentarios y getRespuestas 59
Figura 66 Método getReacciones 59
Figura 67 Instalación de librerías necesarias 60
Figura 68 Código del archivo schema.graphqls 61
Figura 69 Código del archivo schema.graphqls 61
Figura 70 Definición de las querys 62
Figura 71 Código generado automáticamente 62
Figura 72 Código generado automáticamente 63
Figura 73 Código generado automáticamente 63
Figura 74 Instanciación del servidor 64
Figura 75 Método de conexión a la base de datos 64
Figura 76 Método GET Usuarios – Go GraphQL 65
Figura 77 Método GET Publicaciones – Go GraphQL 65
Figura 78 Métodos GET Comentarios, Respuestas y Reacciones – Go GraphQL 66
Figura 79 Método para obtener datos del primer nivel - REST 68
Figura 80 Método para obtener datos del segundo nivel - REST 69
Figura 81 Método para obtener datos del tercer nivel - REST 70
Figura 82 Método para obtener datos del cuarto nivel - REST 71
Figura 83 Método para obtener datos del quinto nivel - REST 71
Figura 84 Código del Caso de uso 1 - REST 72
Figura 85 Código del Caso de uso 2 - REST 73
Figura 86 Código del Caso de uso 3 - REST 73
Figura 87 Código del Caso de uso 4 - REST 74
Figura 88 Código del Caso de uso 5 - REST 74
Figura 89 Método para obtener datos del primer nivel - GraphQL 75
Figura 90 Método para obtener datos del segundo nivel - GraphQL 76
Figura 91 Método para obtener datos del segundo nivel - GraphQL 76
Figura 92 Método para obtener datos del tercer nivel - GraphQL 77
Figura 93 Método para obtener datos del cuarto nivel - GraphQL 77
Figura 94 Método para obtener datos del quinto nivel - GraphQL 78
Figura 95 Código del Caso de uso 1 - GraphQL 79
Figura 96 Código del Caso de uso 2 - GraphQL 79
Figura 97 Código del Caso de uso 3 - GraphQL 80
Figura 98 Código del Caso de uso 4 - GraphQL 80
Figura 99 Código del Caso de uso 5 - GraphQL 80
Figura 100 Código para consultar los datos 81
Figura 101 Datos devueltos por el método he impresos por consola 82
Figura 102 Método del guardado de datos en Excel 83
Figura 103 Datos guardados en Excel 83
Figura 104 Código para ejecutar todos los casos de uso 86
Figura 105 Código para ejecutar todos los casos de uso 86
Figura 106 Resultado en Excel de la ejecución de los casos de uso 87
Figura 107 Resultado en Excel de la ejecución de los casos de uso 87

Figura 108 Tiempo de respuesta de las APIs REST en el Caso de uso 1 88
Figura 109 Tiempo de respuesta de las APIs REST en el Caso de uso 2 89
Figura 110 Tiempo de respuesta de las APIs REST en el Caso de uso 3 90
Figura 111 Tiempo de respuesta de las APIs REST en el Caso de uso 4 90
Figura 112 Tiempo de respuesta de las APIs REST en el Caso de uso 5 91
Figura 113 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 1 92
Figura 114 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 2 93
Figura 115 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 3 93
Figura 116 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 4 94
Figura 117 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 5 95
Figura 118 Desempeño de las APIs GraphQL en el Caso de uso 5... 95
Figura 119 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 1......................... 96
Figura 120 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 2......................... 96
Figura 121 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 3......................... 97
Figura 122 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 4......................... 97
Figura 123 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 1.................. 98
Figura 124 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 2.................. 99
Figura 125 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 3................ 100
Figura 126 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 4................ 101
Figura 127 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 5................ 102

ÍNDICE DE TABLAS

Tabla 1 Comparativa Inicial entre REST y GraphQL.. 19
Tabla 2 Lenguajes de Programación más usados con REST 21
Tabla 3 Frameworks para los lenguajes de REST.. 22
Tabla 4 Lenguajes de Programación más usados con GraphQL 23
Tabla 5 Frameworks para los lenguajes de GraphQL... 23
Tabla 6 Estrellas de GitHub en los Frameworks de Go 24
Tabla 7 Distribución de Datos 30
Tabla 8 Pruebas de funcionamiento de las APIs.. 83

RESUMEN

REST y GraphQL son un lenguajes de consulta que esta implementado en el lado del servidor y está siendo utilizado por las principales empresas de software, estos aparecen como una alternativa ante las estructuras monolíticas, poniendo en juego las nuevas arquitecturas orientadas a microservicios. Tanto REST como GraphQL no han tenido un estudio amplio de la eficiencia del consumo de datos y de su comportamiento frente a bases de datos relacionales y no relacionales.

Es por ello por lo que se construyó una serie de APIs REST y GraphQL para realizar un análisis del consumo de datos entre las diferentes tecnologías que se usan en la construcción de dichas APIs; utilizando las métricas de la eficiencia en el tiempo de respuesta dada por la norma ISO/IEC 25023.

Para empezar con la construcción de las APIs se realizó una investigación de las tecnologías más usadas en desarrollo durante los últimos años (2019 – 2024). Se seleccionó tres tecnologías para trabajar con REST y otras tres para hacerlo con GraphQL. Seguido de esto se procedió con la creación de seis APIs en total, tres bajo la arquitectura de REST y las otras tres bajo GraphQL.

Después del desarrollo de las APIs, había que hacer la comparativa en sus tiempos de respuesta, esto mediante un experimento controlado que agrega validez a esta investigación. El experimento consistió en realizar consultas que estaban definidas en cinco casos de uso, se obtuvo los resultados mediante la fórmula del tiempo de respuesta dada en la norma ISO/IEC 25023, se comparó los resultados y la combinación de tecnologías y arquitectura que obtenía el menor tiempo fue la ganadora.

Palabras clave: GraphQL, API-GraphQL, base de datos, relacional, no relacional, experimento controlado, MongoDB.

ABSTRACT

REST and GraphQL are query languages that are implemented on the server side and are used by the main software companies. They appear as an alternative to monolithic structures, introducing new architectures based on microservices. REST and GraphQL have not had wide studies about the efficiency of data consumption and their behavior with relational and non-relational databases.

For this reason, a set of REST and GraphQL APIs was built to make an analysis of data consumption between the different technologies used in the construction of these APIs, using the efficiency metrics of response time given by the ISO/IEC 25023 standard.

To start the construction of the APIs, research was made about the most used technologies in development during the last years (2019 – 2024). Three technologies were selected to work with REST and other three to work with GraphQL. After that, six APIs were created in total, three with REST architecture and three with GraphQL.

After the development of the APIs, it was necessary to compare their response times. This was made through a controlled experiment that gives validity to this research. The experiment consisted of making queries defined in five use cases. The results were obtained using the response time formula from ISO/IEC 25023. Then, the results were compared, and the combination of technologies and architecture with the lowest response time was the winner.

Keywords: GraphQL, API-GraphQL, database, relational, non-relational, controlled experiment, MongoDB.

Introducción

Problema

Antecedentes:

El empleo de servicios en línea ha experimentado un crecimiento significativo en las últimas décadas, transformándose en una herramienta indispensable para el intercambio de información y la prestación de servicios. Desde la aparición de las primeras arquitecturas de servicios web, como SOAP, hasta la llegada de REST y posteriormente GraphQL, la evolución tecnológica ha estado marcada por la búsqueda de eficiencia en el procesamiento de datos y la mejora de la experiencia del usuario. Sin embargo, a medida que la demanda de servicios en línea ha aumentado, también lo han hecho los desafíos asociados con los tiempos de respuesta y la eficiencia en el Backend (Sayago Heredia & Flores, 2019).

En este contexto, la selección de herramientas y arquitecturas adecuadas para el desarrollo de aplicaciones web ha sido un tema recurrente en la comunidad de desarrolladores, ya que estas decisiones tienen un impacto directo en el rendimiento y la escalabilidad de las soluciones implementadas.

Situación Actual:

En la actualidad, el uso de servicios en línea se ha convertido en el principal medio de interacción entre usuarios y plataformas digitales, este incremento en la demanda ha puesto en evidencia problemas relacionados con los elevados tiempos de respuesta durante el procesamiento de datos en el Backend, lo cual afecta directamente la experiencia del usuario.

Actualmente, REST y GraphQL son las arquitecturas más utilizadas en el desarrollo de servicios web, cada una con características y ventajas particulares. Por un lado está REST, basado en un enfoque de recursos y operaciones HTTP, ha sido ampliamente adoptado por su simplicidad y escalabilidad, y por otra parte GraphQL, con su enfoque en consultas flexibles y eficientes, ha ganado popularidad por su capacidad para reducir el overfetching y underfetching de datos. Sin embargo, aún existe un debate en la comunidad de desarrolladores sobre cuál de estas arquitecturas ofrece un mejor rendimiento en términos de eficiencia y tiempo de respuesta (Andrés & Solano, 2019).

Prospectiva:

A futuro, se espera que la demanda de servicios en línea continúe en aumento, impulsada por la digitalización de más sectores y la necesidad de aplicaciones cada vez más complejas y dinámicas. En este escenario, la elección de la arquitectura adecuada para el desarrollo de servicios web será un factor crítico para garantizar la escalabilidad, el rendimiento y la satisfacción del usuario.

Además, es probable que surjan nuevas tecnologías y enfoques que complementen o compitan con REST y GraphQL, lo que podría redefinir los estándares actuales. Por lo que es fundamental establecer criterios claros para evaluar y comparar estas arquitecturas, con el fin de tomar decisiones informadas que permitan optimizar el procesamiento de datos en el Backend y ofrecer una experiencia de usuario superior.

Planteamiento del Problema:

El problema, como se muestra en la *Figura1*; afecta negativamente la experiencia del usuario y la eficiencia de las aplicaciones web. Dado que REST y GraphQL son las arquitecturas más utilizadas en la actualidad, es necesario determinar cuál de estas prevalece en términos de eficiencia y rendimiento, con el objetivo de ofrecer soluciones que minimicen los tiempos de respuesta y mejoren la experiencia del usuario.

Para abordar este problema, se propone realizar un análisis comparativo entre REST y GraphQL, evaluando su desempeño en el procesamiento de datos y su capacidad para satisfacer las necesidades actuales de los usuarios. Este estudio permitirá establecer criterios claros para la selección de la arquitectura más adecuada en el desarrollo de aplicaciones web, contribuyendo así a la optimización de los servicios en línea y a la mejora de la experiencia del usuario.

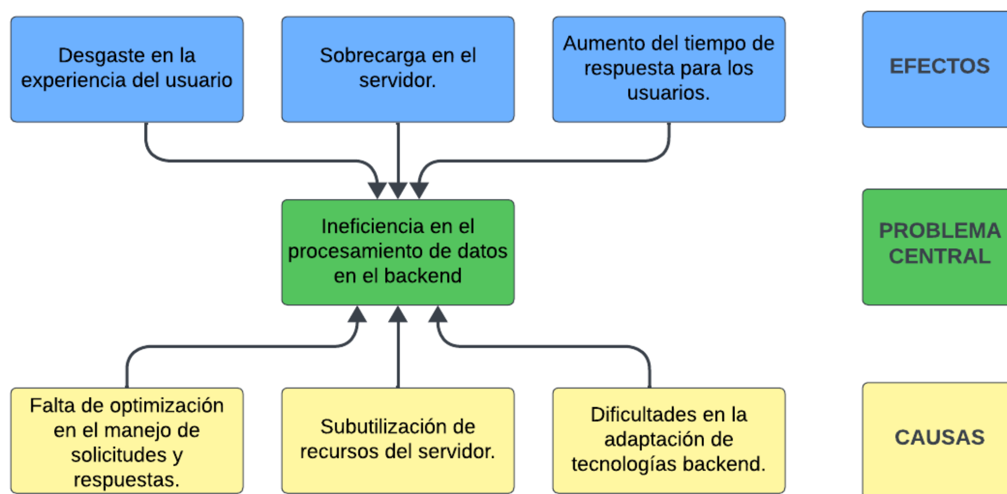

Figura 1. Árbol de problemas

Objetivos

Objetivo General

Comparar la eficiencia de las arquitecturas REST y GraphQL en el procesamiento de datos del Backend, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023.

Objetivos Específicos

- Revisar la literatura y estudios relevantes para identificar las tres tecnologías más utilizadas en el desarrollo Backend para las arquitecturas REST y GraphQL.

- Desarrollar un experimento computacional para comparar la eficiencia del procesamiento de datos Backend con las arquitecturas REST y GraphQL, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023.

- Analizar los resultados obtenidos del experimento computacional para identificar la tecnología y arquitectura más eficiente en el procesamiento de datos en el Backend.

Alcance

La finalidad del presente trabajo es comparar el desempeño de la eficiencia de las tecnologías de REST y GraphQL desarrolladas en los tres principales lenguajes de programación de cada una de estas, para identificar cuál es la arquitectura más eficiente. La evaluación se enfocará de manera primordial en el tiempo de respuesta, una métrica crítica para determinar la eficiencia de ambas tecnologías en la construcción de APIs de alto rendimiento. Esta medición será llevada a cabo siguiendo las pautas y metodologías establecidas por la ISO 25023, asegurando una evaluación objetiva y cuantitativa de la eficiencia de REST y GraphQL, proporcionado una base sólida para las conclusiones del estudio (Jara Córdova et al., 2023).

En una primera etapa, se revisará la literatura disponible, en el proceso de consulta de documentación técnica, así como el análisis de estudios relevantes con el fin de identificar las tres tecnologías más destacadas y ampliamente utilizadas en el desarrollo Backend para REST y GraphQL, proporcionando un contexto esencial que permitirá contextualizar y fundamentar la evaluación comparativa.

Además de la comparación entre REST y GraphQL, el proyecto se propone encontrar la mejor combinación de tecnologías en el Backend mediante el desarrollo de un experimento computacional. Esto implica identificar las tecnologías más utilizadas y determinar cuál es la combinación más eficaz para cada arquitectura, es decir se evaluará la eficiencia de combinaciones como REST + Tecnología 1, REST + Tecnología 2 y REST + Tecnología 3, con el fin de identificar la óptima para el contexto específico. Lo mismo se aplicará a GraphQL + Tecnología 1, GraphQL + Tecnología 2 y GraphQL + Tecnología 3, proporcionando así una guía valiosa para la implementación de soluciones tecnológicas, para la medición de la eficiencia se utilizará la métrica de tiempo medio de respuesta descrita en la ISO/IEC 25023.

Asimismo, es importante señalar qué aspectos no serán abordados en esta investigación; si bien se trabajará con una base de datos, la elección de esta no será objeto de un análisis, sino que se basará en consideraciones prácticas y de compatibilidad con las tecnologías evaluadas, y aunque tradicionalmente se implementaría una capa de Front-End para la interfaz de usuario, se ha determinado que en este contexto, dicha capa resulta innecesaria. Dejando de forma resumida, el alcance del proyecto como se muestra en la *Figura 2* a continuación.

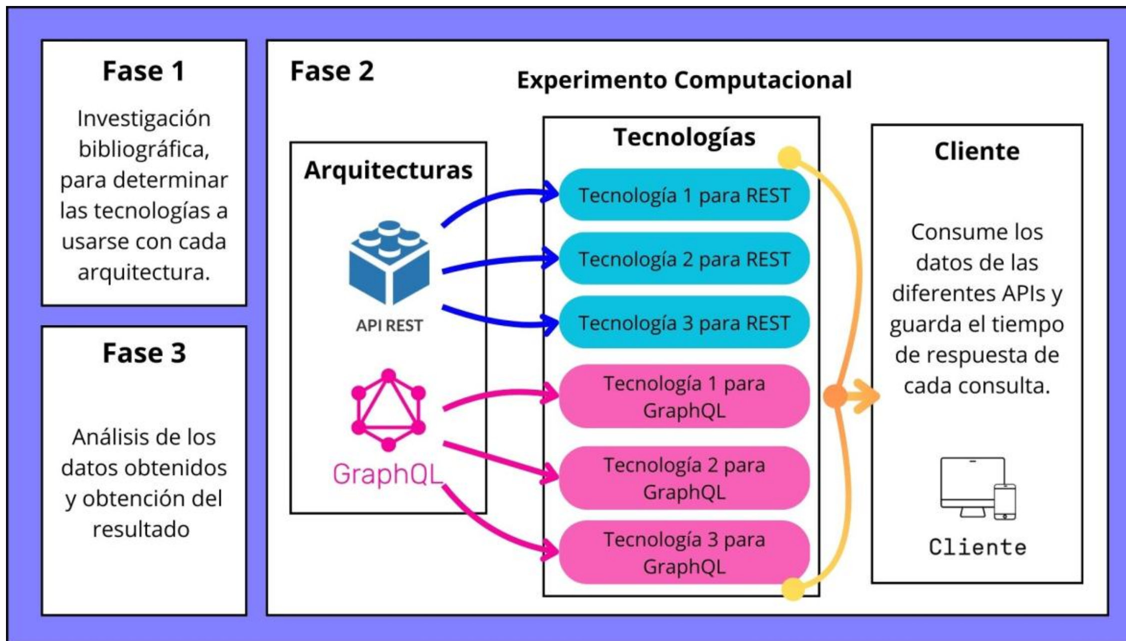

Figura 2. Diagrama del Alcance

Justificación

El presente trabajo, contribuye con el objetivo de desarrollo sostenible planteado por la ONU, número 9: "Industria, Innovación e Infraestructura"; esto debido a que al tener como propósito el evaluar y comparar la eficiencia entre REST y GraphQL para el Backend, se promueve la innovación en el desarrollo de proyectos web, impulsando la utilización de infraestructuras tecnológicas más eficientes y sostenibles (Santías Dema, 2020).

Además de ello, este proyecto concuerda con el "Plan de Creación de Oportunidades 2021-2025" de Ecuador, en el lineamiento estratégico que tiene como objetivo el promover la transformación digital y acceso a tecnologías de la información y comunicación (TIC), al analizar tecnologías vanguardistas como lo son GraphQL y REST, lo que contribuye a actualizar y fortalecer la base tecnológica del país y también promover la implementación de soluciones digitales avanzadas (Mendoza et al., 2021).

Dentro de los beneficiarios directos del proyecto están los docentes de la Universidad Técnica del Norte, profesionales del desarrollo web, ingenieros de software, y demás personal interesado en la optimización del procesamiento de datos en el Backend. De forma indirecta son beneficiarios los estudiantes actuales y futuros de la universidad, usuarios y clientes finales de aplicaciones o sistemas web.

Justificación Tecnológica

En este ámbito, la justificación mencionada se centra en la evaluación y comparación de tecnologías Backend, para mejorar la eficiencia en el procesamiento de datos. Al lograr identificar la tecnología más eficaz, es posible contribuir al desarrollo de infraestructuras digitales más avanzadas y sostenibles, lo que tiene un impacto directo en la economía, la sociedad y el medio ambiente.

Justificación Práctica

Al buscar la mejor alternativa para mejorar los tiempos de respuesta por parte del Backend y las APIs que interactúen allí, con ello conseguir un mejor tiempo de respuesta, se contribuye a resolver problemáticas de tiempos elevados de espera en páginas y servicios web.

I. Capítulo 1: Marco Teórico

1.1 Microservicios y Arquitectura de APIs

1.1.1 Introducción a los Microservicios

Es imposible hablar de APIs, sin antes tratar de por medio los microservicios que representan un enfoque arquitectónico innovador al dividir una aplicación en unidades más pequeñas y autónomas (Red Hat, 2023). Esta descomposición se traduce en microservicios individuales, cada uno encargado de realizar una función específica, este estilo arquitectónico permite una modularidad extrema, ya que cada microservicio opera de manera independiente, facilitando la mejora, actualización o sustitución de componentes sin afectar el funcionamiento global de la aplicación.

Es en la comunicación entre microservicios donde entran las APIs, dado que posibilitan este intercambio, promoviendo una interacción eficiente y fluida. Al adoptar microservicios, las organizaciones pueden agilizar el ciclo de vida de desarrollo, ya que los equipos pueden trabajar de manera aislada en áreas específicas de la aplicación, mejorando la flexibilidad y la velocidad de respuesta a cambios, además, la capacidad de escalar y desplegar microservicios de forma independiente permite optimizar los recursos y adaptarse a las demandas variables (Toapanta, 2024).

Actualmente la arquitectura de microservicios se ha vuelto crucial en entornos donde la agilidad, la escalabilidad y la adaptabilidad son esenciales para mantener la competitividad en el desarrollo de aplicaciones y servicios.

Otra de las ventajas clave es que la arquitectura de microservicios aporta una mayor resiliencia a las aplicaciones, esta capacidad de operar de forma independiente implica que, en caso de fallo en un microservicio los demás pueden seguir funcionando sin interrupciones significativas. Esta resiliencia inherente se traduce en una mayor robustez y confiabilidad del sistema, mitigando los impactos negativos de posibles problemas en uno de los muchos servicios que ofrece un sistema computacional (Chandler, 2022).

Como se mencionó anteriormente la agilidad se presenta como otro pilar fundamental de los microservicios, las características de los microservicios permiten a las empresas innovar de manera más rápida y eficiente, gracias al enfoque descentralizado que facilita la introducción de nuevas funcionalidades, la corrección de errores y la adaptación a cambios en el mercado sin afectar la totalidad de la aplicación, ofreciendo un marco ideal para el desarrollo ágil y la mejora continua (Cassidy et al., 2024).

A pesar de sus beneficios, la adopción de la arquitectura de microservicios no se libra de desafíos con los que debe lidiar, siendo uno de los principales la mayor complejidad asociada, ya que a diferencia de las aplicaciones monolíticas, el diseñar, desarrollar y gestionar microservicios individuales puede requerir un enfoque más elaborado, causando que la coordinación entre

microservicios y la gestión de su ciclo de vida puedan resultar más exigentes, implicando a su vez una mayor demanda de habilidades técnicas junto con una planificación más detallada.

Otro desafío es la posible introducción de mayor latencia en la aplicación debido a la comunicación entre microservicios, dado que aunque la descomposición modular permite una mayor flexibilidad, la interacción entre estos componentes independientes puede generar demoras, lo que afecta potencialmente el rendimiento general de la aplicación. Por ello, la gestión efectiva de la latencia se convierte en un aspecto crítico para garantizar una experiencia de usuario óptima.

Finalmente en términos de seguridad, los microservicios plantean desafíos adicionales por sus múltiples puntos de entrada, incrementando la superficie de ataque para posibles amenazas, esto causa que la seguridad de los microservicios sea más compleja de gestionar en comparación con una aplicación monolítica, ya que cada microservicio debe ser individualmente asegurado, generando además la necesidad de una atención constante a las vulnerabilidades potenciales (Fachat, 2024).

Tanto con sus ventajas como con sus desventajas, los microservicios tienen una relevancia total hoy en día, tal como se menciona en la revista The New Stak: "Los microservicios se están volviendo cada vez más populares a medida que las empresas buscan formas de crear aplicaciones más escalables, resilientes y ágiles" (Humble, 2021).

1.1.2 Las APIs

Una API, por sus siglas en inglés de Application Programming Interface lo que traducido al español significaría: Interfaces de Programación de Aplicaciones; es un componente fundamental en el mundo de la tecnología moderna, ya que es un conjunto de reglas y definiciones que permiten que diferentes aplicaciones se comuniquen entre sí, actuando como intermediarios, y permitiendo que distintos programas informáticos intercambien datos y funcionen de manera conjunta (Goodwin, 2024).

Las API ofrecen una versatilidad sin límites permitiendo realizar una amplia gama de acciones, desde una API de redes sociales que facilita la publicación de mensajes, el intercambio de fotos y la conexión con amigos; hasta una API de pago, esencial para la ejecución eficiente de transacciones financieras mediante tarjetas de crédito o débito (Goodwin, 2024). En ambos casos la finalidad de las APIs es ofrecer una experiencia integrada y personalizada.

Como es mencionado en el blog oficial de Red Hat, las APIs se dividen en diversas categorías, considerando su ámbito de uso (Red Hat, 2018); como se muestra a continuación:

- APIs privadas: Están diseñadas para uso interno dentro de una organización, optimizando procesos entre equipos o sistemas propios.
- APIs públicas: Disponibles para terceros, como las de redes sociales o servicios de pago, que permiten a desarrolladores externos integrar funcionalidades en sus aplicaciones.
- APIs asociadas: Accesibles solo para socios comerciales autorizados, comúnmente utilizadas en acuerdos para compartir datos de forma controlada.

Considerando todo lo expuesto anteriormente, se puede afirmar que las APIs representan una herramienta de gran potencial para la creación de aplicaciones robustas y amigables. Al establecer un formato estandarizado para la comunicación entre diversos programas, las APIs posibilitan a los

desarrolladores crear aplicaciones que se fusionan de manera fluida, proporcionando a los usuarios una experiencia sin inconvenientes.

1.1.3 Arquitectura REST

REST, por sus siglas en inglés de: Representational State Transfer; constituye un sofisticado paradigma arquitectónico diseñado específicamente para sistemas distribuidos, estableciendo un enfoque único que destaca por su simplicidad y eficacia; REST gira en torno a la noción de recursos, entendidos como entidades donde cada entidad se representa a través de un punto final específico (endpoint), estas son identificables y manejables por un sistema, que abarcan desde datos concretos hasta servicios abstractos. La esencia de REST radica en su capacidad para modelar la comunicación a través de la manipulación de dichos recursos mediante operaciones bien definidas (Swanson, 2024).

Entonces, la verdadera innovación de REST viene en sus representaciones de recursos antes mencionadas. Estas representaciones no son más que formas de expresar un recurso en un formato que sea comprensible para el sistema, la versatilidad de estas representaciones permite que los recursos se presenten de manera coherente y, al mismo tiempo, facilita la interoperabilidad entre sistemas.

Para su funcionamiento, REST se vale de un conjunto de métodos HTTP estandarizados para la manipulación eficiente de recursos distribuidos. Estos métodos, también conocidos en inglés como endpoints incluyen GET, POST, PUT y DELETE, cada uno con funciones específicas que delinean las operaciones cruciales en la gestión de información (Gunnell, 2024).

El método GET como primer método, se erige como el medio para recuperar información sobre un recurso identificado, facilitando la consulta de datos existentes. Por otro lado, el método POST tiene como finalidad la creación de nuevos recursos en el sistema, posibilitando la introducción de información novedosa, y abriendo las puertas a la expansión y actualización constante de los recursos disponibles. Por su parte, el método PUT despliega su utilidad al actualizar información en un recurso ya existente, aportando flexibilidad para ajustar y mejorar la información almacenada (Gunnell, 2024).

Y finalmente, el método DELETE asume un papel crucial en el ciclo de vida del recurso al proporcionar la capacidad de eliminar de manera definitiva una entidad específica. Estos métodos, cuidadosamente estructurados, constituyen el andamiaje esencial de REST, permitiendo una manipulación coherente y eficaz de recursos en entornos distribuidos, y destacando la elegancia y practicidad de esta arquitectura (Gunnell, 2024).

De manera simplificada, el proceso que realiza una API REST para cada una de estas acciones (crear, leer, actualizar y borrar datos) se muestra en la siguiente figura.

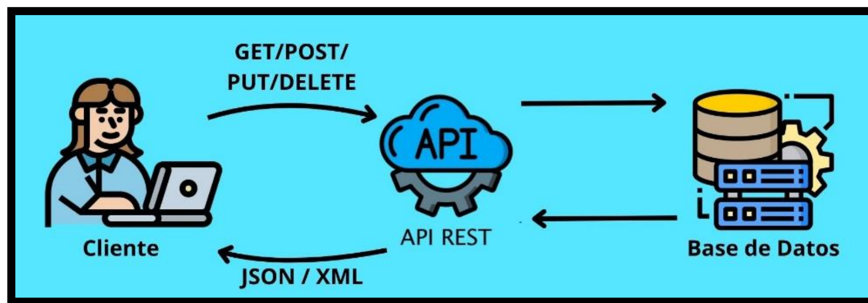

Figura 3 Arquitectura REST

1.1.4 Arquitectura GraphQL

GraphQL por otra parte se distingue como un lenguaje de consulta de datos diseñado para APIs, al proporcionar a los clientes un control preciso sobre la información que desean recuperar, presentando una solución más eficiente en comparación con las APIs convencionales, que suelen devolver conjuntos de datos más extensos de lo necesario para las solicitudes específicas de los clientes (Benjie, 2025).

Entonces se puede decir que la característica distintiva de GraphQL radica en su capacidad para personalizar las consultas, permitiendo a los clientes especificar los campos y la estructura exactos que desean recibir en la respuesta, evitando así la redundancia de datos innecesarios.

Al adoptar GraphQL, las aplicaciones pueden optimizar significativamente la transferencia de datos entre el cliente y el servidor, ya que la respuesta se ajusta directamente a las necesidades particulares de cada solicitud, con este nivel de flexibilidad no solo se potencia la eficiencia en el intercambio de información, sino que además se contribuye a una experiencia de desarrollo más ágil, ya que la capacidad de definir de manera precisa los datos requeridos agiliza el desarrollo de aplicaciones al reducir el número de solicitudes necesarias y a su vez, esto mejora la eficacia en términos de rendimiento y consumo de recursos (Benjie, 2025).

GraphQL cuenta con un gran abanico de ventajas en comparación con las APIs convencionales, como su eficiencia que se manifiesta en la capacidad de recuperar solo la información necesaria, evitando la transferencia innecesaria de datos, también tiene una flexibilidad que radica en su capacidad para adaptarse a las necesidades específicas de cada solicitud, permitiendo a los clientes definir la estructura de la respuesta según sus requisitos particulares, finalmente también es destacable su capacidad para manejar volúmenes crecientes de datos y solicitudes, ofreciendo así una solución robusta y adaptable a medida que las aplicaciones evolucionan (Sanders & Singh, 2025).

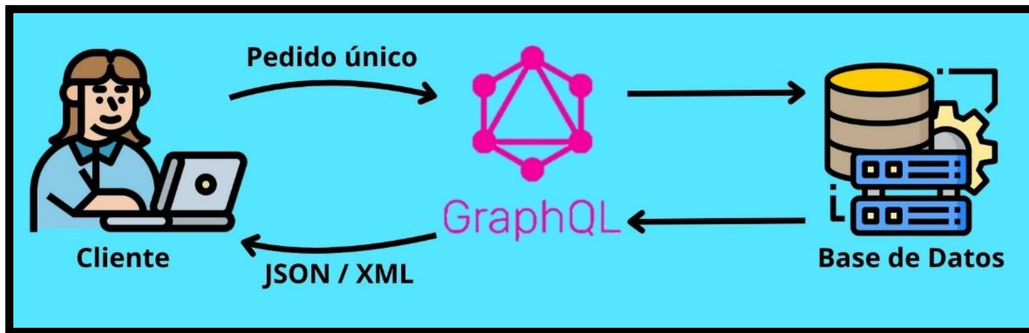

Figura 4 Arquitectura GraphQL

1.1.5 Comparativa inicial entre REST y GraphQL

Con las características presentadas anteriormente de cada arquitectura; una primer comparación superficial comenzando con REST, en este modelo las interacciones con los datos (como lo vimos anteriormente) se realizan mediante métodos HTTP estándar como GET, POST, PUT y DELETE, y cada recurso tiene su propia URL definida (Bello, 2023).

Por su parte, GraphQL propone un modelo más flexible en el que es el cliente quien puede solicitar exactamente los datos que necesite a través de una única consulta, eliminando así la necesidad de múltiples llamadas a diferentes endpoints. Además, en lugar de trabajar con recursos fijos, GraphQL expone un único punto de entrada que permite consultas dinámicas según los requerimientos específicos (Brito & Valente, 2020).

Entonces, la primer diferencia clave radica en la manera en que ambos manejan la estructura de las respuestas; en REST, las respuestas son prediseñadas por el servidor, lo que puede llevar a que los clientes reciban más información de la necesaria o deban realizar múltiples solicitudes para obtener toda la información requerida, mientras que GraphQL proporciona respuestas personalizadas de acuerdo a las solicitudes del cliente, reduciendo tanto el exceso como la falta de datos (Chrystal R, 2024).

Otras diferencias menos fundamentales, pero igualmente relevantes son las que se plantean en otros estudios sobre REST y GraphQL:

REST	GraphQL
Sigue un modelo rígido y predefinido, donde cada endpoint devuelve una estructura fija de datos. Si el cliente necesita información adicional, debe realizar múltiples solicitudes o recibir datos redundantes (Bello, 2023).	Permite al cliente solicitar solo los datos necesarios mediante una única consulta flexible, evitando datos innecesarios y/o la falta de estos (Brito & Valente, 2020).
Utiliza múltiples endpoints (uno por recurso), lo que puede complicar el mantenimiento y escalabilidad en APIs complejas (Chrystal R, 2024).	Opera con un único endpoint, donde las consultas definen la estructura de la respuesta, simplificando la interacción con el backend (Bello, 2023).

Se basa en protocolos estándar HTTP (GET, POST, PUT, DELETE) y formatos como JSON o XML (Quiña-Mera et al., 2025).	Emplea un lenguaje de consulta tipado, que permite a los clientes especificar relaciones anidadas y campos específicos en una sola petición, lo que es ideal para sistemas con datos interconectados (Chrystal R, 2024).
Requiere versionado explícito para introducir cambios sin romper la compatibilidad con clientes existentes (Bello, 2023).	Evita el versionado gracias a su capacidad de evolución incremental, donde nuevos campos pueden añadirse sin afectar consultas antiguas (Brito & Valente, 2020).

Tabla 1 Comparativa Inicial entre REST y GraphQL

1.2 Normativa ISO/IEC 25023

1.2.1 Introducción a la norma

La norma ISO/IEC 25023 forma parte de la serie de estándares ISO/IEC 25000, conocida como SQuaRE (Software Product Quality Requirements and Evaluation), cuyo objetivo principal es definir métodos de medición para evaluar cuantitativamente la calidad de sistemas y productos de software (ISO/IEC, 2016). Esta norma complementa las características y subcaracterísticas de calidad propuestas en la ISO/IEC 25010, proporcionando métricas específicas que permiten medir de manera objetiva distintos aspectos de la calidad (Quiña-Mera et al., 2025).

Dentro del ámbito de eficiencia, que es particularmente relevante para el presente estudio, ya que está directamente relacionada con el desempeño del sistema frente a su uso de recursos y su respuesta en tiempos controlados; la ISO/IEC 25023 establece métricas enfocadas en evaluar el comportamiento temporal, la utilización de recursos y la capacidad del sistema, proporcionando lineamientos claros para medir la calidad del rendimiento bajo condiciones específicas (ISO/IEC, 2016).

1.2.2 Métrica de tiempo medio de respuesta

Dentro de las métricas de eficiencia del rendimiento, la norma ISO/IEC 25023 destaca el tiempo medio de respuesta (mean response time), definido como el intervalo promedio entre una solicitud enviada al sistema y la recepción completa de su respuesta (ISO/IEC, 2016).

$$X = \frac{\sum_{i=1}^{n}(Bi - Ai)}{n}$$

Donde:

- Ai: Tiempo de inicio de la solicitud i.
- Bi: Tiempo de finalización de la solicitud i.
- n: Número de mediciones realizadas.

1.3 Investigación Bibliográfica

Con la finalidad de encontrar las tres tecnologías más comúnmente usadas para el desarrollo de REST y GraphQL como se planteó en el primer objetivo específico del estudio, se pretende seguir

los pasos establecidos en el artículo con temática investigativa del ingeniero Antonio Quiña, quien a su vez toma conceptos de la Guía de Wohlin.

Entonces, se debe comenzar por una revisión bibliográfica orientada a identificar las tecnologías más comúnmente utilizadas en el desarrollo de APIs bajo las arquitecturas REST y GraphQL. Para ello se definen las siguientes preguntas que guían el estudio (Quiña-Mera et al., 2023):

RQ1: ¿Qué tecnologías son las más usadas para desarrollar APIs con REST?

Esta pregunta tiene el propósito de descubrir las tecnologías con las que se trabajará posteriormente para REST.

RQ2: ¿Qué tecnologías son las más usadas para desarrollar APIs con GraphQL?

De la misma forma también se encontrarán las tecnologías para usarse con GraphQL.

El siguiente paso es obtener el conjunto de datos con el que se trabajará en adelante, para lo cual primero se efectuó la búsqueda de documentación en cinco bases de datos académicas reconocidas por su relevancia en el ámbito académico: DBLP, Google Scholar, IEEE Xplore, ScienceDirect y Scopus; en total, se identificaron 118 documentos relacionados con APIs REST y 43 con APIs GraphQL. Cabe aclarar que la cantidad inferior de documentos utilizados para GraphQL en comparación con los de REST se debe principalmente a que su acceso era restringido y la mayoría de pago, por lo que fueron omitidos para este análisis. Dejando la distribución del dataset con el que se trabajará cada arquitectura, de la siguiente forma:

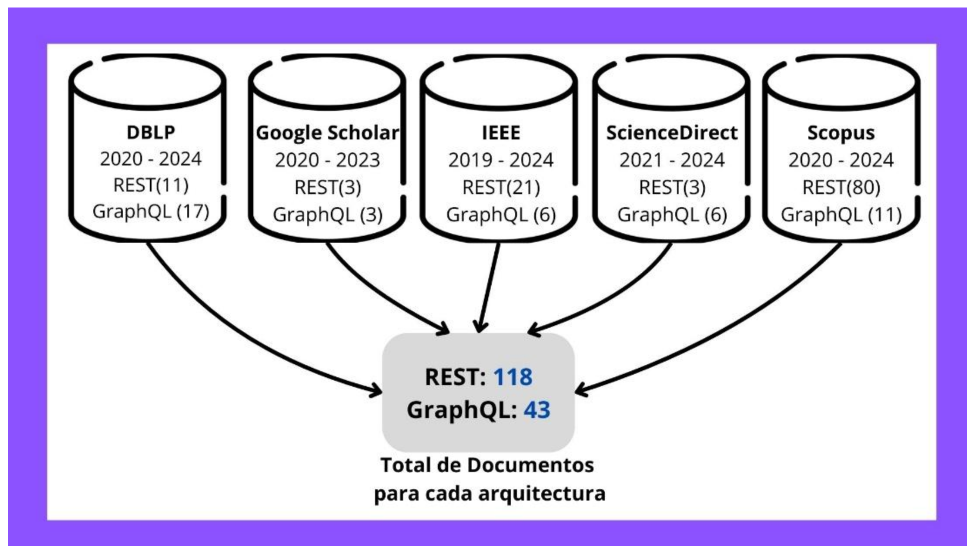

Figura 5 Dataset REST y GraphQL

1.3.1 Tecnologías más usadas con REST

Con el dataset ya obtenido, el siguiente paso es revisar cada documento individualmente en búsqueda de la arquitectura de API, el lenguaje, framework y demás herramientas que se usaron para el desarrollo. En el caso de REST se obtuvo los siguientes resultados:

Lenguaje Backend principal	Veces que se repite
S/T	76
JavaScript	15
Python	13

Java	8
Go	2
Php	2
C#	1
HTML	1

Tabla 2 Lenguajes de Programación más usados con REST

De esta revisión bibliográfica se obtiene que la mayoría de los documentos consultados carecían de información relevante para este estudio, esto posiblemente por la privacidad que querían darle los autores a su obra; pero aun así igualmente se destaca una tendencia en lenguajes de programación y los primeros tres puestos que ocupan este lugar son:

- JavaScript
- Python
- Java

Para corroborar esta tendencia, también se consultaron páginas oficiales orientadas a la programación, como lo son GitHub, y Stack Overflow que también muestran sus estadísticas anuales, y como se observa, también son estos tres lenguajes los que resaltan por sobre los demás:

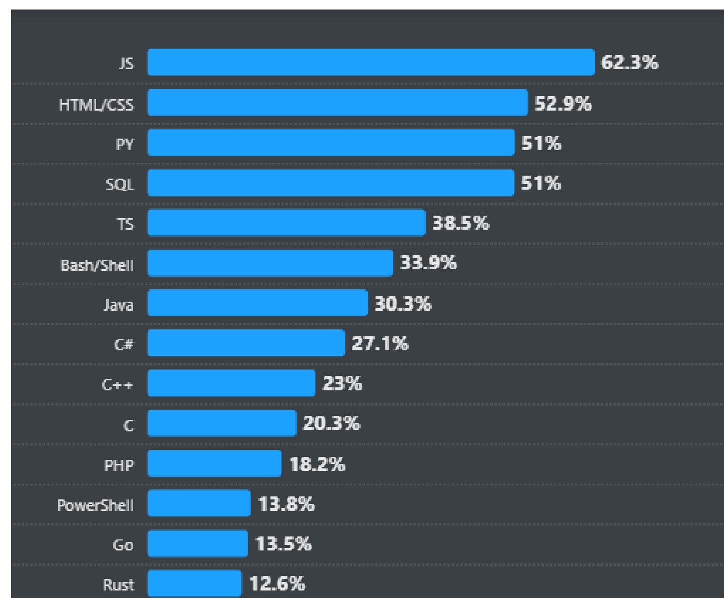

Figura 6 Estadísticas de Stack Overflow 2024

Como se observa en la imagen, se comparten los lenguajes de Backend que comparten los primeros puestos son JavaScript, Python y Java; cabe aclarar que lenguajes como HTML o SQL no son tomados en cuenta ya que no son lenguajes de Backend con los que se pueda crear una API REST.

Por otra parte en las estadísticas del 2024 de GitHub, se tiene un resultado similar:

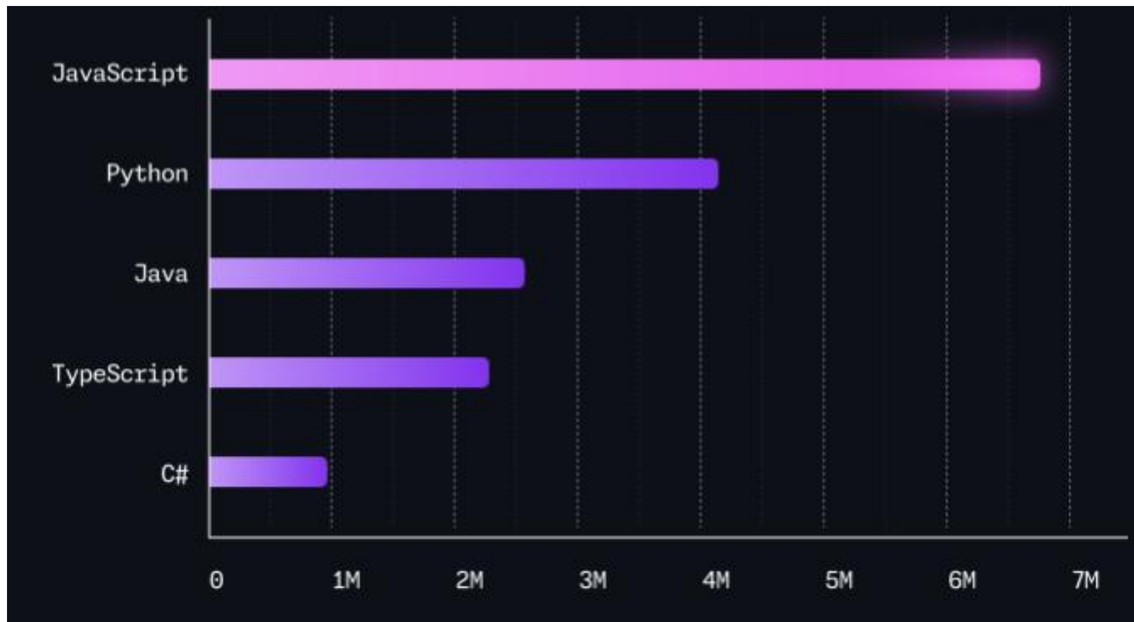

Figura 7 Estadísticas de GitHub 2024

Como se puede ver, los primeros lugares se repiten por lo que se establece a JavaScript, Python y Java como los tres lenguajes con los que se desarrollará el experimento computacional. Tras tener claro los lenguajes de programación que se usarán, el siguiente paso es determinar los frameworks con que facilitarán la creación de las APIs en cada caso establecido, la revisión bibliográfica muestra que:

Lenguaje	Framework	Veces que se repite
JavaScript	S/T	12
	Express	2
	NestJs	1
Python	S/T	6
	Django	2
	FastAPI	2
	Flask	3
Java	S/T	3
	Spring	2
	Spring Boot	3

Tabla 3 Frameworks para los lenguajes de REST

Con esto los frameworks a usarse quedan determinados, dejando la combinación de la siguiente forma:

- JavaScript - Express + NodeJs. En este caso se debe añadir Node, ya que Express debe funcionar obligatoriamente este entorno de ejecución.
- Python – Flask
- Java – Spring Boot

1.3.2 Tecnologías más usadas con GraphQL

Similar a lo realizado con REST, necesitamos establecer las herramientas para las APIs de GraphQL por lo que siguiendo el mismo método anterior, la revisión bibliográfica arroja el siguiente resultado:

Lenguaje Backend principal	Veces que se repite
C#	1
Golang	2
Java	1
JavaScript	7
Python	3
S/T	29
Total general	**43**

Tabla 4 Lenguajes de Programación más usados con GraphQL

En esta tabla se observa que a pesar de nuevamente tener el inconveniente de que la mayoría de los documentos no mencionaban las herramientas usadas, igualmente es posible establecer el ranking de los lenguajes usados para programar APIs GraphQL.

- JavaScript
- Python
- Golang

Ya con los lenguajes establecidos, lo siguiente es determinar el framework de la misma forma que se hizo con REST; la revisión bibliográfica arrojó los siguientes resultados:

Lenguaje	Framework	Veces que se repite
JavaScript	S/T	6
	EvoMaster	1
Python	S/T	1
	Django	1
	Flask	1
Golang	S/T	2

Tabla 5 Frameworks para los lenguajes de GraphQL

Como se puede ver en la tabla, en este caso la información de GraphQL es de mucho menos acceso que la de REST por lo que para el caso de Python donde el ganador no es claro, se optará por elegir de forma arbitraria Flask debido a que es el mismo framework con el que se trabajará para REST por tanto, volver a utilizar dicha herramienta facilitará el desarrollo. Por otra parte, para el caso de Go donde no fue mencionado ningún framework se usará nuevamente las estadísticas de GitHub para encontrar la alternativa más usada, en base a las estrellas que tenga cada repositorio del framework, dado que estas estrellas representan la cantidad de usuarios que han marcado el repositorio como favorito. Por tanto tras acceder a los repositorios de GitHub, los frameworks diseñados para trabajar con Go son:

Framework	Estrellas de GitHub

Gin	82.1k
Fiber	36.3k
Echo	30.9k
Beego	32k
FastHTTP	22.5k

Tabla 6 Estrellas de GitHub en los Frameworks de Go

Como se muestra en la tabla número 6 Gin, supera por mucho la puntuación de los demás frameworks, por lo que esta será la herramienta usada para trabajar con el lenguaje de Go.

1.4 Experimentación en la Ingeniería de Software

La experimentación en ingeniería de software es una estrategia fundamental para analizar y validar resultados empíricos, permitiendo identificar causas y efectos en el desarrollo de herramientas, metodologías o tecnologías (Wohlin et al., 2012). El problema de esta estrategia es que a diferencia de otras disciplinas, la experimentación en el campo del software enfrenta desafíos particulares, como el aplicar métodos rigurosos, tales como las pruebas de hipótesis, y controlar variables relacionadas con sujetos/participantes, objetos/artefactos de software; y herramientas de medición (Brito & Valente, 2020).

En el campo del software destacan los experimentos controlados, ampliamente utilizados en la investigación, estos estudios se basan en la manipulación intencional de uno o más factores bajo condiciones específicas, mientras se mantienen constantes las demás variables (Quiña-Mera et al., 2025). Con esta metodología es posible comparar tratamientos o intervenciones, por ejemplo: evaluar una nueva herramienta frente a un estándar, con el objetivo de obtener conclusiones válidas y reproducibles. O para el caso del experimento presentado en este documento: comparar el tiempo de respuesta entre diferentes APIs creadas bajo las arquitecturas de REST y GraphQL.

1.5 Herramientas

1.5.1 JavaScript

Es un lenguaje de programación interpretado, orientado a objetos y de tipado dinámico, ampliamente utilizado para el desarrollo web tanto en el lado del cliente como del servidor (Rosado, 2024). Dada su naturaleza asincrónica y basada en eventos, JavaScript es especialmente adecuado para manejar múltiples solicitudes simultáneas, siendo por esto una elección popular para aplicaciones que requieren escalabilidad y eficiencia en la comunicación entre cliente y servidor (Rosado, 2024).

1.5.2 Node.js

Node.js es un entorno de ejecución para JavaScript que permite ejecutar código en el lado del servidor, lo cual facilita la creación de aplicaciones Backend, además que con su arquitectura basada en eventos y operaciones no bloqueantes permite manejar gran cantidad de solicitudes concurrentes con un uso eficiente de recursos (Sharma, 2020).

1.5.3 Express.js

Es un framework minimalista para Node.js, diseñado para simplificar la creación de APIs REST, con su middleware modular se facilita la implementación de rutas, manejo de errores y validación de datos, todo ello agiliza el desarrollo de servicios RESTful (Barta, 2015).

1.5.4 Python

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, conocido por su sintaxis clara y su amplia aplicabilidad en áreas como la ciencia de datos, automatización y desarrollo web (Python Software Foundation, 2023). En el ámbito de APIs, Python presenta una facilidad de aprendizaje y un robusto ecosistema de bibliotecas, lo cual permite desarrollar aplicaciones de backend eficientes y mantenibles.

1.5.5 Flask

Flask es un microframework para Python enfocado en el desarrollo de aplicaciones web y APIs de manera rápida y sencilla, al trabajar con Python su ligereza no es inconveniente alguno ya que ofrece extensibilidad mediante módulos, lo que lo hace ideal para proyectos que requieren flexibilidad en su estructura, y en este caso la posibilidad de trabajar tanto con APIs REST, como también las GraphQL (Flask, 2019).

1.5.6 Java

Java es un lenguaje de programación compilado, orientado a objetos y con tipado estático, ampliamente adoptado en sistemas empresariales y aplicaciones de gran escala (Oracle, 2022). Su robustez y portabilidad lo convierten en una opción sólida para el desarrollo de servicios backend.

1.5.7 Spring Boot

Spring Boot es un framework basado en Java que simplifica la configuración y el despliegue de aplicaciones web, al integrar de forma predeterminada componentes del ecosistema Spring, lo que facilita la creación de APIs RESTful mediante el manejo de dependencias y la configuración automática (“Spring Boot,” 2013).

1.5.8 Go (Golang)

Go es un lenguaje de programación desarrollado por Google que combina eficiencia en la ejecución con una sintaxis concisa. Su modelo de concurrencia lo hace ideal para construir servicios web que manejan múltiples conexiones simultáneas. Su simplicidad, junto con su velocidad de compilación y ejecución, lo hacen una opción atractiva para el desarrollo de APIs (*Go*, 2019).

1.5.9 Gin

Gin es un framework web para Go, optimizado para el desarrollo de aplicaciones HTTP de alto rendimiento, con su capacidad para manejar miles de solicitudes por segundo con mínima sobrecarga. se convierte en una herramienta clave para la creación de APIs GraphQL (*Gin Web Framework*, 2025).

1.5.10 PostgreSQL

Sistema de gestión de bases de datos relacional orientado a objetos, de código abierto, reconocido por su robustez y fiabilidad. Su desarrollo inició en 1986 en la Universidad de California en Berkeley, y desde entonces ha mantenido un crecimiento sostenido gracias a una comunidad activa que impulsa mejoras constantes; entre sus principales fortalezas se encuentran su arquitectura

extensible, la integridad de los datos y su compatibilidad con la mayoría de los sistemas operativos (PostgreSQL Global Development Group, 2011).

II. Capítulo 2: Experimento Computacional

2.1 Diseño

Para llevar a cabo el experimento computacional planteado en esta investigación, es necesario iniciar con su respectivo diseño, para poder definir con claridad el procedimiento metodológico del experimento. Por tanto, se presentará una visión general del experimento, la cual será desarrollada posteriormente de forma detallada en cada una de sus fases.

El entorno experimental se compone de una base de datos central, la cual servirá como punto de conexión para un total de seis APIs, que se dividen equitativamente en dos grupos: tres implementadas bajo la arquitectura REST y las otras tres mediante la arquitectura GraphQL. Cada una de estas APIs tendrá la capacidad de ejecutar consultas sobre la base de datos, simulando operaciones reales de consumo de datos.

El siguiente componente esencial es el cliente de pruebas, cuyo rol será consumir las APIs mencionadas y registrar métricas específicas, siendo la principal de ellas el tiempo de respuesta de cada consulta realizada. Además, este cliente estará diseñado para almacenar datos adicionales relevantes para el análisis, los cuales serán descritos con mayor detalle en la sección correspondiente dedicada a su diseño e implementación.

El diseño del experimento computacional queda entonces de la siguiente manera:

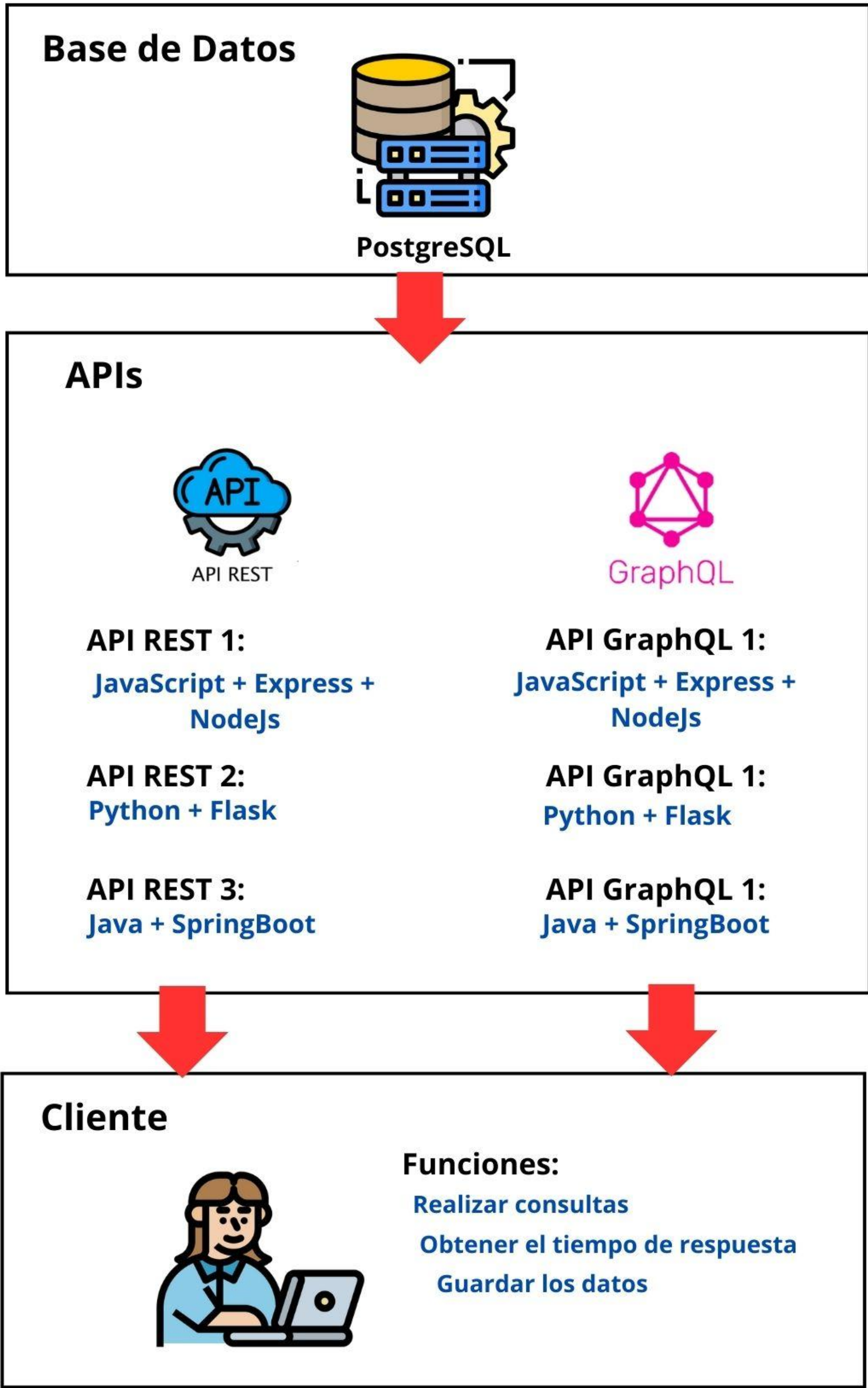

Figura 8 Experimento Computacional

2.2 Desarrollo de la Base de Datos

De acuerdo con el diseño establecido para el experimento computacional, el primer componente a desarrollar es la base de datos. Esta estará estructurada en cinco niveles de profundidad, cada uno representado por una tabla con relaciones jerárquicas entre sí, con el objetivo de simular un entorno de datos que permita evaluar la eficiencia de las consultas ejecutadas por las distintas APIs.

En el primer nivel se encuentra la tabla usuarios, que contiene información básica y de referencia de usuarios registrados en un sistema, esta tabla incluye los siguientes atributos: id_usuario, nombre, correo y clave; a partir de esta tabla se establece una relación de uno a muchos con el segundo nivel, representado por la tabla publicaciones, permitiendo que un usuario pueda tener múltiples publicaciones. Los atributos definidos en la tabla publicaciones son: id_publicacion, id_usuario, titulo, contenido y fecha_publicacion.

El tercer nivel está compuesto por la tabla comentarios, la cual se encuentra vinculada a publicaciones mediante una relación de uno a muchos, dado que una publicación puede recibir múltiples comentarios por parte de diferentes usuarios, además esta tabla incluye los campos: id_comentario, id_publicacion, texto y fecha_comentario.

A continuación, en el cuarto nivel, se encuentra la tabla de respuestas, que representa las respuestas a los comentarios, donde cada comentario puede tener varias respuestas asociadas, continuando con la estructura de relación de uno a muchos; los atributos de esta tabla son: id_respuesta, id_comentario, texto y fecha_respuesta.

Finalmente, el quinto nivel está conformado por la tabla reacciones, donde se relacionan con las respuestas. Cada respuesta puede tener múltiples reacciones, por lo que también se establece una relación de uno a muchos, esta tabla incluye los siguientes campos: id_reaccion, id_respuesta, id_tipo_reaccion y fecha_reaccion.

Adicionalmente, se incorpora una tabla auxiliar denominada reacciones_tipo, que no forma parte de la jerarquía de niveles antes descrita, pero cumple un papel fundamental en la normalización del modelo, esta tabla contiene los posibles tipos de reacción que pueden asignarse a las respuestas, sus atributos son: id_tipo_reaccion y nombre. La inclusión de esta tabla permite evitar redundancias, ya que, aunque solo existen cinco tipos de reacción, estos pueden ser reutilizados en múltiples registros de la tabla reacciones, estableciendo una relación entre reacciones y reacciones_tipo en la que muchas reacciones pueden estar asociadas a un mismo tipo, manteniendo la integridad y eficiencia del modelo.

Todo esto se representa de manera gráfica en el diagrama de entidad-relación de la base de datos:

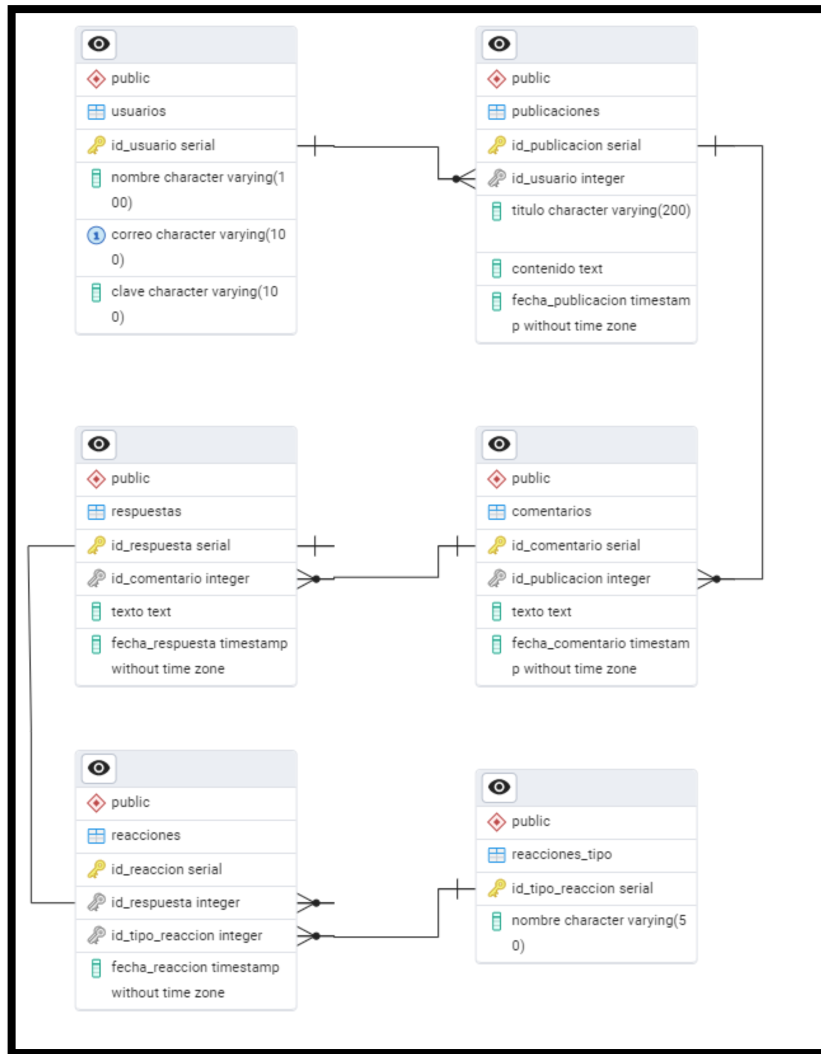

Figura 9 Diagrama Entidad-Relación

2.2.1 Distribución del Experimento

Una vez definida y estructurada la base de datos, el siguiente paso es proceder con su llenado, para lo cual se establece primero la distribución de datos que se empleará durante la ejecución del experimento computacional; esta distribución está directamente relacionada con los diferentes casos de uso diseñados para evaluar el rendimiento de las APIs. En total, se han definido cinco casos de uso, cada uno de los cuales contempla un nivel adicional de profundidad dentro de la base de datos.

En el caso de uso 1, se trabajará exclusivamente con el primer nivel, es decir, la tabla de usuarios, mientras que en el caso de uso 2, se incluirán las tablas del primer y segundo nivel, es decir, usuarios y publicaciones; esta lógica continuará progresivamente hasta llegar al caso de uso 5, donde se utilizarán las cinco tablas jerárquicamente conectadas: usuarios, publicaciones, comentarios, respuestas y reacciones.

Dentro de cada caso de uso, se ejecutarán consultas considerando distintos volúmenes de datos con el objetivo de evaluar el comportamiento de las APIs bajo distintas cargas, los tamaños de muestra

definidos para las consultas son: 1, 10, 100, 1000, 10 000 y 100 000 registros, cabe decir también que medida que se incrementa el número de registros, también se ajusta la cantidad de elementos en las tablas relacionadas, manteniendo la coherencia jerárquica del modelo.

La siguiente tabla ilustra la distribución de datos planificada para cada caso de uso, representando la cantidad de registros involucrados en cada tabla, conforme a la profundidad correspondiente:

Num de Registros	Cu 1	Cu 2	Cu 3	Cu 4	Cu 5
1	1	1	1x1x1	1x1x1x1	1x1x1x1x1
10	10	3x3	2x2x2	2x2x2x1	2x2x1x2x1
100	100	10x10	5x5x4	3x3x3x4	3x3x2x3x2
1000	1000	32x31	10x10x10	6x6x6x5	4x4x4x4x4
10000	10000	100x100	22x22x21	10x10x10x10	6x6x7x6x7
100000	100000	316x316	46x46x47	18x18x18x17	10x10x10x10x10

Tabla 7 Distribución de Datos

En la tabla anterior, cada expresión (por ejemplo, 3×3×2×3×2) representa la cantidad de registros generados por nivel jerárquico, entonces en el caso de uso 5 con 100 registros, se generan 3 usuarios, cada uno con 3 publicaciones, cada publicación con 2 comentarios, cada comentario con 3 respuestas y cada respuesta con 2 reacciones, alcanzando aproximadamente el total deseado de registros.

2.3 Desarrollo de las APIs REST

2.3.1 API - JavaScript, Express.js y Node.js

Una vez completada la definición y estructuración de la base de datos, se da paso al desarrollo de las APIs REST, comenzando por la realizada con JavaScript como lenguaje de programación principal. Esta API utiliza el entorno de ejecución Node.js, mientras que Express facilitará la construcción de las rutas y la gestión de las peticiones HTTP, que caracterizan la arquitectura REST.

Respecto a las versiones de las herramientas empleadas, se utilizó:

- Node.js v22.14.0 (LTS): Esta versión forma parte de la línea *Long Term Support*, lo cual garantiza actualizaciones de seguridad y mantenimiento durante un periodo extendido, lo que es crucial para asegurar estabilidad y compatibilidad a largo plazo durante el desarrollo del experimento.
- Express v4.21.2: Aunque ya se encuentra disponible la versión 5.1.0 del framework, se optó por trabajar con la serie 4.x debido a su madurez, amplia documentación, y extensa compatibilidad con bibliotecas complementarias, dado que la versión 5 aún se encuentra en un estado relativamente reciente y en proceso de adopción.

Con las herramientas y sus versiones claramente definidas, se procedió a la implementación del servidor Express, este servidor constituye la base de la API dado que

permite su ejecución como un servicio local que escucha peticiones en el puerto 5001. Posteriormente todas las solicitudes de consulta de datos se canalizarán a través de este puerto, instanciado como se muestra a continuación:

```
import express from 'express';
import router from './routes/index.js';

const app = express();

app.set('port', process.env.PORT || 5001);
app.use(express.json());
app.use(router);

app.listen(
    app.get('port'), () => {
        console.log(`Servidor escuchando en el puerto ${app.get('port')}`);
    }
);
```

Figura 10 Instanciación del Servidor Express - JS

Tras instanciar el servidor, el siguiente paso consiste en establecer la conexión con la base de datos previamente definida, para lo cual se recurre a la librería oficial de Node.js para PostgreSQL, conocida como pg. Esta biblioteca permite crear una conexión directa con la base de datos solicitando únicamente los parámetros esenciales que son: nombre del host, usuario, contraseña y el nombre de la base de datos que se usará; una vez establecida la conexión, es posible realizar operaciones SQL a través de la API.

```
const conexionBD = new Pool({
    host: 'localhost',
    user: 'postgres',
    password: '100101',
    database: 'Blogs',
});
```

Figura 11 Conexión a la base de datos - JS

Esta conexión se encapsula dentro de lo que se conoce como controladores, que son funciones especializadas encargadas de gestionar las consultas a la base de datos, cada controlador maneja una operación específica, tal como obtener usuarios, publicaciones, o comentarios y actúa como intermediario entre las rutas de la API y la lógica del acceso a datos.

Como se muestra en el fragmento de código incluido a continuación, una vez establecido el servidor y la conexión a la base de datos, se puede proceder con la creación del primer controlador, que será el encargado de manejar las peticiones para consultar una cantidad determinada de usuarios, según se lo solicite desde el cliente.

```
export const getUsuariosCantidad = async(req, res) => {
    const { numUsr } = req.params;
    try {
        let ordenSql;
        if(numUsr <= 320){
            ordenSql = `SELECT * FROM usuarios WHERE id_usuario between 1 and 320 ORDER BY RANDOM() LIMIT ${
        }
        else{
            ordenSql = "SELECT * FROM usuarios ORDER BY RANDOM() LIMIT " + numUsr + ";";
        }
        const respuesta = await conexionBD.query(ordenSql);
        res.status(200).json(respuesta.rows);
    } catch (error) {
        return res.status(500).json({message: error.message});
    }
};
```

Figura 12 Método GET Usuarios - JS

Como se explicó anteriormente, en una API REST se emplean los denominados verbos HTTP, los cuales definen el tipo de operación que se desea realizar sobre los recursos de la base de datos, en el caso del método mostrado en la imagen, se trata de una solicitud de tipo GET, que tiene como propósito recuperar datos desde el servidor, concretamente una cierta cantidad de registros de la tabla de usuarios.

Dentro del cuerpo del controlador se observa una condición particular: si el número de usuarios solicitados es menor a 320, la consulta se limitará a seleccionar aleatoriamente usuarios cuyo id_usuario se encuentre en el rango de 1 a 320, la razón detrás de esta restricción radica en un criterio de eficiencia.

Solo a los primeros 320 usuarios se les han asignado publicaciones, ya que incrementar innecesariamente la cantidad de publicaciones para todos los 100.000 usuarios generaría una base de datos excesivamente pesada y poco manejable, dado que para el Caso de Uso 2 (CU2) solo se requiere que existan publicaciones asociadas a un máximo de 316 usuarios, por lo que se determinó que solo ese subconjunto necesitaba datos adicionales en niveles inferiores; esta misma lógica se aplicará posteriormente para las publicaciones y su relación con los comentarios.

Además de esta lógica condicional, el controlador cuenta con un bloque de manejo de excepciones (try-catch), estructura que es fundamental para asegurar la estabilidad del servicio, ya que si se produjera un error no controlado durante la ejecución de la API, este podría comprometer su funcionamiento general, por lo que capturando los errores de forma explícita, se logra que la API pueda seguir operativa y brindar una respuesta controlada, incluso si una petición específica falla, en este caso la respuesta incluirá un código de estado 200 para indicar una solicitud exitosa o 500 en caso de que se presente un error interno del servidor.

El siguiente método desarrollado permite obtener las publicaciones asociadas a los usuarios antes mencionados, su implementación se presenta a continuación:

Figura 13 Método GET Publicaciones - JS

```
export const getPublicacionesCantidad = async(req, res) => {
    const { idUsr, numPst } = req.params;
    let ordenSql;
    try {
        if(numPst <= 60){
            ordenSql = `SELECT * FROM publicaciones where id_usuario=${idUsr} AND fecha_publicacion IS NOT
        }else{
            ordenSql = `SELECT * FROM publicaciones where id_usuario=${idUsr} ORDER BY RANDOM() LIMIT ${nu
        }
        const respuesta = await conexionBD.query(ordenSql);
        res.status(200).json(respuesta.rows);
    } catch (error) {
        return res.status(500).json({message: error.message});
    }
};
```

Este controlador sigue el mismo patrón que el anterior: se trata también de una solicitud de tipo GET, implementada con una estructura de manejo de errores similar, sin embargo la lógica de la consulta SQL cambia ligeramente porque en este caso, la consulta está condicionada según la cantidad de publicaciones solicitadas. Si se solicitan 60 o menos publicaciones, la consulta se limitará a retornar aquellas que posean un valor en el campo fecha_publicacion, mientras que si se requieren más de 60 publicaciones la selección será aleatoria, sin importar si este campo está presente o no.

El campo fecha_publicacion antes mencionado, cumple una función clave dentro de esta estructura de datos jerárquica al actuar como un indicador de que la publicación tiene asociada una o más entradas en el siguiente nivel, que corresponde a los comentarios. Las publicaciones que tienen asignada una fecha poseen comentarios, mientras que las que no lo tienen, simplemente no cuentan con estos registros relacionados, esta decisión de diseño responde nuevamente a criterios de optimización, al evitar poblar innecesariamente toda la base de datos con datos que no serán utilizados durante el experimento y solamente harán la base de datos más lenta y pesada de usar.

Fuera de esta condición lógica, el resto del proceso sigue siendo consistente, primero construyendo y ejecutando una sentencia SQL, y luego retornando los datos resultantes al cliente que realiza la solicitud.

El siguiente método a desarrollar es el encargado de consultar los comentarios asociados a una determinada publicación:

```
export const getComentariosCantidad = async(req, res) => {
    const { idPub, numCmt } = req.params;
    try {
        const ordenSql = `SELECT * FROM comentarios where id_publicacion=${idPub} ORDER BY RANDOM() LIMIT ${
        const respuesta = await conexionBD.query(ordenSql);
        res.status(200).json(respuesta.rows);
    } catch (error) {
        return res.status(500).json({message: error.message});
    }
};
```

Figura 14 Método GET Comentarios - JS

A diferencia de los métodos anteriores, en este punto ya se cuenta con una estructura de datos suficientemente optimizada y acotada, por lo que no se considera necesario establecer condiciones adicionales en la consulta, esto quiere decir que todos los comentarios

recuperados en esta etapa ya cuentan con respuestas asociadas, en concordancia con la cantidad definida para el experimento, con lo que es posible ejecutar directamente la consulta sin condicionales adicionales, y manteniendo la coherencia en la progresión de niveles de la base de datos.

El mismo caso ocurre con el método que da acceso al penúltimo nivel, y es el de obtener las respuestas a cada comentario:

```
export const getRespuestasCantidad = async(req, res) => {
    const { idCmt, numRes } = req.params;
    try {
        const ordenSql = `SELECT * FROM respuestas where id_comentario=${idCmt} ORDER BY RANDOM() LIMIT ${nu
        const respuesta = await conexionBD.query(ordenSql);
        res.status(200).json(respuesta.rows);
    } catch (error) {
        return res.status(500).json({message: error.message});
    }
};
```

Figura 15 Método GET Respuestas - JS

Al igual que en el método anterior, aquí no se implementa ninguna condición adicional en la consulta, lo que implica que cada comentario registrado cuenta con respuestas asociadas, y a su vez cada una de estas respuestas contiene sus correspondientes reacciones, adicionalmente este método también es de tipo GET, manteniendo la coherencia con la estructura general de los controladores REST antes desarrollados: manejo de excepciones mediante try-catch, ejecución de una consulta SQL y devolución de los resultados al cliente que realiza la solicitud.

Con esto, se llega al último método de consulta de esta API, correspondiente al nivel más profundo de la jerarquía de datos, que es la obtención de reacciones.

```
export const getReaccionesCantidad = async(req, res) => {
    const { idRes, numReac } = req.params;
    try {
        const ordenSql = `SELECT * FROM reacciones where id_respuesta=${idRes} ORDER BY RANDOM() LIMIT ${num
        const respuesta = await conexionBD.query(ordenSql);
        res.status(200).json(respuesta.rows);
    } catch (error) {
        return res.status(500).json({message: error.message});
    }
};
```

Figura 16 Método GET Reacciones - JS

Como en los casos anteriores, también se cuenta con la estructura general, mientras que en la consulta SQL se limita el número de datos que se desean obtener.

Es importante resaltar que, a diferencia del nivel uno (usuarios), donde basta con indicar la cantidad de registros a obtener, a partir del segundo nivel en adelante (publicaciones, comentarios, respuestas y reacciones) se requiere una dependencia jerárquica del nivel anterior. Es decir:

- Para obtener un número específico de publicaciones, es necesario proporcionar el ID del usuario (id_usuario) al que pertenecen.

- Para obtener comentarios, se debe indicar el ID de la publicación.
- Para obtener respuestas, se debe especificar el ID del comentario.
- Y finalmente, para obtener reacciones, se necesita el ID de la respuesta.

Con todos los controladores definidos y funcionales, el último paso en la construcción de esta API REST consiste en establecer las rutas HTTP, las cuales permiten que el servidor exponga sus funcionalidades y pueda ser consultado desde distintos clientes, ya sea navegadores web, scripts, o herramientas especializadas como Postman.

```
const router = Router();

router.get('/usuarios/:numUsr', getUsuariosCantidad);
router.get('/publicaciones/:idUsr/:numPst', getPublicacionesCantidad);
router.get('/comentarios/:idPub/:numCmt', getComentariosCantidad);
router.get('/respuestas/:idCmt/:numRes', getRespuestasCantidad);
router.get('/reacciones/:idRes/:numReac', getReaccionesCantidad);


export default router;
```

Figura 17 Rutas REST - JS

El fragmento de código anterior muestra claramente cómo se realiza esta configuración, definiendo un enrutador a través de la función Router() de Express, tras lo cual se establecen las rutas que serán accesibles públicamente, junto con los métodos GET correspondientes que apuntan a sus respectivos controladores:

- /usuarios/:numUsr: retorna una cantidad específica de usuarios, según el parámetro numUsr.
- /publicaciones/:idUsr/:numPst: permite obtener un número determinado de publicaciones (numPst) asociadas al usuario identificado por idUsr.
- /comentarios/:idPub/:numCmt: devuelve comentarios asociados a una publicación específica.
- /respuestas/:idCmt/:numRes: obtiene respuestas relacionadas a un comentario dado.
- /reacciones/:idRes/:numReac: retorna las reacciones correspondientes a una respuesta concreta.

Con todo esto se puede dar por concluido el desarrollo de la primer API REST, por lo que la siguiente API es la que usa Python como su lenguaje de programación y Flask como framework principal para el desarrollo de la API.

2.3.2 API - Python y Flask

Dado que la API REST desarrollada con JavaScript fue la primera en ser implementada y validada para el experimento, se busca mantener una estructura coherente en las siguientes APIs REST, es por ello que esta API de Python con Flask sigue el mismo diseño general, salvo en aquellos aspectos donde la arquitectura del lenguaje o del framework requiera modificaciones, las cuales serán explicadas en su respectiva sección.

El primer paso, al igual que en la API de JavaScript, es la instanciación del servidor, lo que para este caso se muestra a continuación:

```python
from src import initApp

app = initApp()

if __name__ == "__main__":
    app.run(port=5002, debug=True)
```

Figura 18 Instanciación del servidor Flask - Py

```python
from flask import Flask
from src.routes.routes import api_bp

def initApp():
    app = Flask(__name__)
    app.config['JSON_SORT_KEYS'] = False

    # Registro de las rutas HTTP
    app.register_blueprint(api_bp)

    return app
```

Figura 19 Configuración del servidor y sus rutas HTTP - Py

A diferencia de la API construida con JavaScript, en este caso la inicialización del servidor se realiza en dos archivos distintos: uno que ejecuta la aplicación (main.py) y otro que encapsula la configuración de Flask (src/__init__.py); esta separación se hace con fines de modularización y mantenibilidad del código, lo cual es una práctica común en aplicaciones Flask ya que permite centralizar la configuración de la aplicación en un único lugar y facilita el crecimiento del proyecto en caso de que se agreguen nuevas funcionalidades o configuraciones. Adicionalmente, se ha definido el puerto 5002 para evitar conflictos con la API anterior, que se ejecuta en el 5001.

Para conectar la aplicación con la base de datos PostgreSQL, se hace uso de la librería psycopg2, la cual cumple una función similar a la librería pg en el entorno de Node.js, como se observa en el código, primero se definen los parámetros necesarios para establecer la conexión con la base de datos, los mismos utilizados en la API anterior. También está la función conexionBD() que devuelve una instancia de conexión activa, que será utilizada posteriormente por los controladores para realizar las distintas consultas SQL.

```
import psycopg2
from psycopg2.extras import RealDictCursor

DB_CONFIG = {
    "host": "localhost",
    "database": "Blogs",
    "user": "postgres",
    "password": "100101"
}

def conexionBD():
    return psycopg2.connect(**DB_CONFIG)
```

Figura 20 Conexión a la Base de Datos - Py

Una vez establecida la conexión con la base de datos, se procede a implementar los métodos de consulta, en términos generales estos métodos replican la lógica ya aplicada en la API anterior desarrollada con JavaScript, lo cual permite mantener una arquitectura coherente en toda la solución, aunque la principal diferencia radica en la sintaxis propia del lenguaje y del framework utilizados.

```
def getUsuariosCantidad(numUsr):
    if(numUsr <= 320):
        query = f"SELECT * FROM usuarios WHERE id_usuario between 1 and 320 ORDER BY RANDOM() LIMIT {numUsr}
    else:
        query = f"SELECT * FROM usuarios ORDER BY RANDOM() LIMIT {numUsr};"
    with conexionBD() as conn:
        with conn.cursor(cursor_factory=RealDictCursor) as cursor:
            cursor.execute(query)
            return cursor.fetchall()
```

Figura 21 Método GET Usuarios - Py

Este método conserva tanto el nombre como la lógica del controlador equivalente en JavaScript, su función es obtener una cantidad específica de usuarios de forma aleatoria, utilizando una cláusula LIMIT en la consulta SQL y la condición if que permite distinguir entre los casos en que se solicita un número de usuarios menor o igual a 320, y los casos en los que se solicita una cantidad mayor, adaptando la consulta a cada situación.

El siguiente método es el de obtener publicaciones, y en este también encontramos un patrón general:

```python
def getPublicacionesCantidad(idUsr, numPst):
    if(numPst <= 60):
        query = f"""
            SELECT * FROM publicaciones
            WHERE id_usuario = {idUsr} AND fecha_publicacion IS NOT NULL
            ORDER BY RANDOM() LIMIT {numPst};
        """
    else:
        query = f"SELECT * FROM publicaciones where id_usuario = {idUsr} ORDER BY RANDOM() LIMIT {numPst};"
    with conexionBD() as conn:
        with conn.cursor(cursor_factory = RealDictCursor) as cursor:
            cursor.execute(query)
            return cursor.fetchall()
```

Figura 22 Método GET Publicaciones - Py

Este método también replica la lógica del establecido en JavaScript, incluyendo el condicional que diferencia entre solicitudes de hasta 60 publicaciones y solicitudes mayores, siendo en este caso que si se solicita un número menor o igual a 60, se filtran las publicaciones sin fecha nula (fecha_publicacion IS NOT NULL).

También es notable que a diferencia de la API anterior, en esta implementación no se utiliza el bloque try-catch dentro del controlador, ya que el manejo de errores se realiza en una capa posterior, específicamente en las rutas.

El siguiente método a tratar es el de obtener los comentarios, que como se mencionó anteriormente, ya no lleva un condicional y solamente ejecuta la orden SQL buscando los comentarios correspondientes a una publicación:

```python
def getComentariosCantidad(idPub, numCmt):
    query = f"""
        SELECT * FROM comentarios
        WHERE id_publicacion = {idPub}
        ORDER BY RANDOM() LIMIT {numCmt};
    """
    with conexionBD() as conn:
        with conn.cursor(cursor_factory = RealDictCursor) as cursor:
            cursor.execute(query)
            return cursor.fetchall()
```

Figura 23 Método GET Comentarios - Py

Este método se encarga de obtener un número determinado de comentarios asociados a una publicación específica, similar a la API previa, no incluye una condición para variar la consulta, sino que se ejecuta directamente en base al ID de la publicación y la cantidad solicitada.

El siguiente método es el de obtener las respuestas correspondientes a un comentario:

```python
def getRespuestasCantidad(idCmt, numRes):
    query = f"""
        SELECT * FROM respuestas
        WHERE id_comentario = {idCmt}
        ORDER BY RANDOM() LIMIT {numRes};
    """
    with conexionBD() as conn:
        with conn.cursor(cursor_factory = RealDictCursor) as cursor:
            cursor.execute(query)
            return cursor.fetchall()
```

Figura 24 Método GET Respuestas - Py

Y finalmente el método de 5to nivel que obtiene las reacciones asociadas a una respuesta:

```python
def getReaccionesCantidad(idRes, numReac):
    query = f"""
        SELECT * FROM reacciones
        WHERE id_respuesta = {idRes}
        ORDER BY RANDOM() LIMIT {numReac};
    """
    with conexionBD() as conn:
        with conn.cursor(cursor_factory = RealDictCursor) as cursor:
            cursor.execute(query)
            return cursor.fetchall()
```

Figura 25 Método GET Reacciones – Py

Una de las diferencias más notorias entre esta API y la desarrollada previamente en JavaScript radica en la gestión de las rutas HTTP, ya que mientras que en la API construida con Express.js las rutas se limitaban a definir la dirección URL y delegaban la lógica al controlador, en el caso de Flask, las rutas cumplen una función más integral.

Esta nueva función radica en especificar la URL y el método HTTP correspondiente, cada ruta en Flask actúa como una función decorada que invoca al controlador, maneja posibles excepciones y formatea la respuesta a JSON antes de enviarla al cliente, para tener más claro estas diferencias, a continuación se muestra la ruta del primer método:

```python
api_bp = Blueprint('api', __name__)

@api_bp.route('/usuarios/<int:num_usr>', methods=['GET'])
def usuarios_cantidad(num_usr):
    try:
        data = getUsuariosCantidad(num_usr)
        return jsonify(data), 200
    except Exception as e:
        return jsonify({"message": str(e)}), 500
```

Figura 26 Ruta GET Usuarios - Py

Como muestra la imagen, en esta rut, se recibe el parámetro num_usr desde la URL, se invoca el método getUsuariosCantidad definido previamente y se devuelve el resultado en formato JSON, controlando también que si ocurre algún error durante la ejecución, se captura mediante un bloque try-except y se responde con un mensaje de error y código de estado 500.

Esto se repite en las demás rutas, la ruta de publicaciones queda de la siguiente manera:

```python
@api_bp.route('/publicaciones/<int:id_usr>/<int:num_pst>', methods=['GET'])
def publicaciones_cantidad(id_usr, num_pst):
    try:
        data = getPublicacionesCantidad(id_usr, num_pst)
        return jsonify(data), 200
    except Exception as e:
        return jsonify({"message": str(e)}), 500
```

Figura 27 Ruta GET Publicaciones - Py

La ruta de comentarios y respuestas son los siguiente métodos:

```python
@api_bp.route('/comentarios/<int:id_pub>/<int:num_cmt>', methods=['GET'])
def comentarios_cantidad(id_pub, num_cmt):
    try:
        data = getComentariosCantidad(id_pub, num_cmt)
        return jsonify(data), 200
    except Exception as e:
        return jsonify({"message": str(e)}), 500

@api_bp.route('/respuestas/<int:id_cmt>/<int:num_res>', methods=['GET'])
def respuestas_cantidad(id_cmt, num_res):
    try:
        data = getRespuestasCantidad(id_cmt, num_res)
        return jsonify(data), 200
    except Exception as e:
        return jsonify({"message": str(e)}), 500
```

Figura 28 Ruta GET Comentarios - Py

Y finalmente la ruta de reacciones que permite recuperar una cantidad determinada de reacciones vinculadas a una respuesta específica, al igual que en las anteriores se maneja de forma segura mediante la captura de excepciones para evitar que un error en la consulta afecte el funcionamiento completo de la API.

```python
@api_bp.route('/reacciones/<int:id_res>/<int:num_reac>', methods=['GET'])
def reacciones_cantidad(id_res, num_reac):
    try:
        data = getReaccionesCantidad(id_res, num_reac)
        return jsonify(data), 200
    except Exception as e:
        return jsonify({"message": str(e)}), 500
```

Figura 29 Ruta GET Reacciones - Py

Hasta aquí llega la API REST desarrollada con Python y Flask, que si bien presenta diferencias técnicas propias del lenguaje y framework, mantiene una estructura homogénea respecto a la API en JavaScript, con este enfoque se permite que el consumo y comprensión de los servicios REST sea independiente del lenguaje en que estén implementados, además las rutas para cada tipo de recurso (usuarios, publicaciones, comentarios, respuestas y reacciones) siguen un patrón común en cuanto a su definición, uso de parámetros, control de errores y formato de respuesta; esto no solo agiliza el desarrollo y las pruebas a realizarse en el cliente, sino que también estandariza el comportamiento general de las APIs REST a lo largo del experimento.

2.3.3 API - Java y SpringBoot

La tercer y ultima API desarrollada bajo la arquitectura de REST es la API que usa Java como su lenguaje principal y SpringBoot como el framework para la API; es un poco especial ya que para usar SpringBoot es necesario valerse de la herramienta spring initializr la cual ayudará a crear un

paquete que cuente con todas las herramientas y dependencias necesarias para comenzar con la codificación de la API.

La herramienta antes mencionada tiene la siguiente interfaz web:

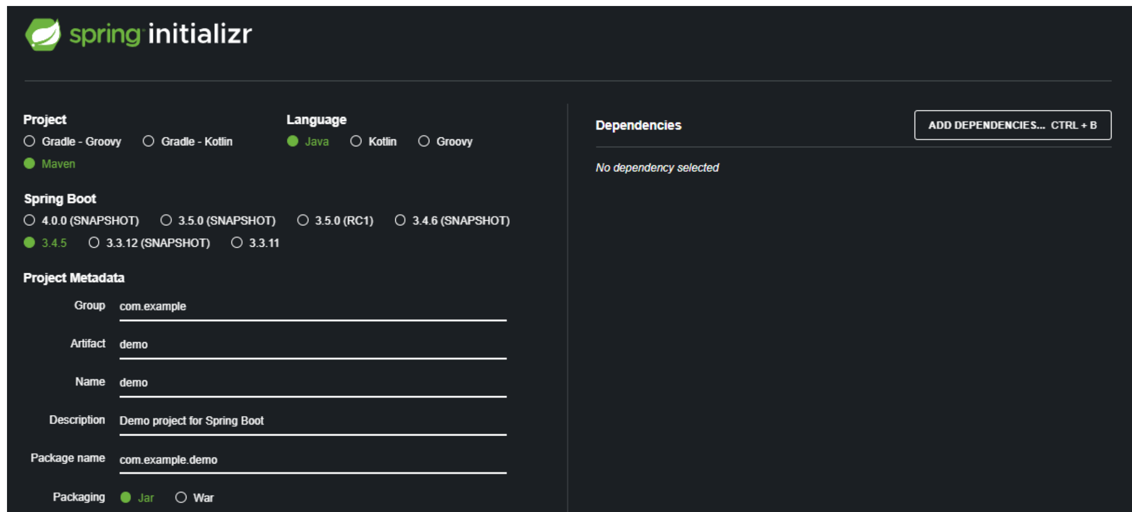

Figura 30 Página principal de spring initialzr

Aquí se pueden ver las configuraciones que se deben escoger para generar nuestro proyecto, comenzando con el gestor de dependencias que queramos usar para nuestro proyecto, para el desarrollo de esta API el gestor escogido es Maven, luego el lenguaje de programación obviamente será Java y para la versión de SpringBoot se omiten las versiones en desarrollo activo, es decir las versiones que tengan la etiqueta de SNAPSHOT; y también las versiones que son candidatas a lanzarse etiquetadas con RC. Tras todos estos filtros las versiones que quedan son: 3.4.5 y 3.3.11, de entre las dos se escoge la más actual y estable, siendo esta la 3.4.5; luego es necesario llenar los diversos campos que se solicitan, como se muestra a continuación:

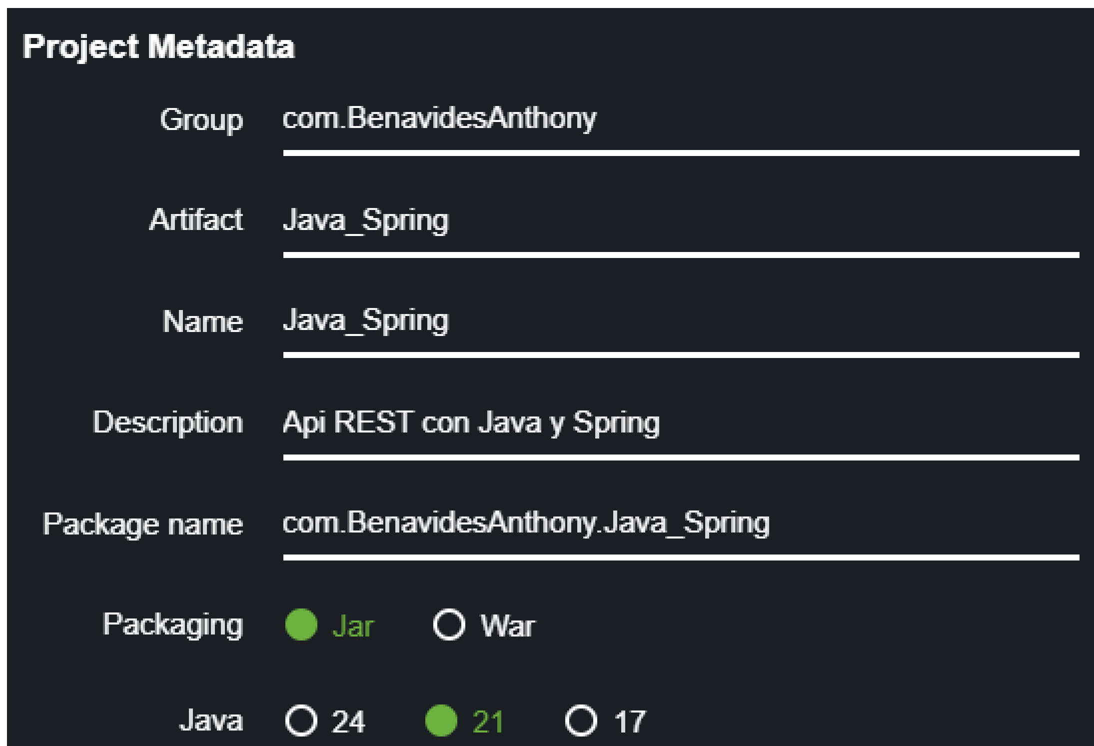

Figura 31 Datos del Proyecto

Adicionalmente a la información por llenar, es necesario escoger si el paquete se descargará en formato Jar o War según las preferencias del usuario, de la misma manera se escoge la versión de Java que en este caso será la 21, para no escoger la más reciente (24) ni la más antigua (17) disponible.

Una vez definidos estos parámetros, es necesario incluir las dependencias que serán utilizadas en el desarrollo de la API, para esta implementación específica, se añadieron las que se muestran a continuación:

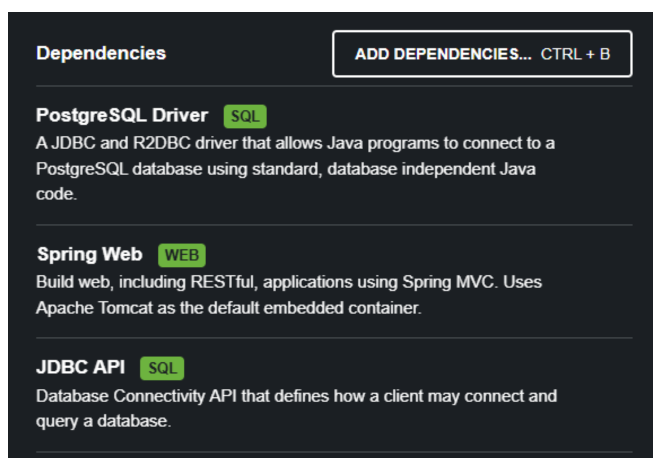

Figura 32 Dependencias del Proyecto

La primer dependencia es el Driver correspondiente que permitirá la conexión a la base de datos de PostgreSQL como se ha hecho en las APIs anteriores, la segunda dependencia es Spring Web que facilita la creación de controladores y servicios REST. Finalmente la tercer dependencia es un

JDBC (Java Database Connectivity), que habilita la interacción con la base de datos a través de consultas SQL de manera directa, simplificando el acceso a datos.

Con estas dependencias y configuraciones básicas definidas, el proyecto está listo para ser generado y descargado. Al descomprimir el paquete generado y abrirlo en el entorno de desarrollo, se puede observar que este ya cuenta con una estructura de carpetas predeterminada, tal como se muestra en la siguiente imagen:

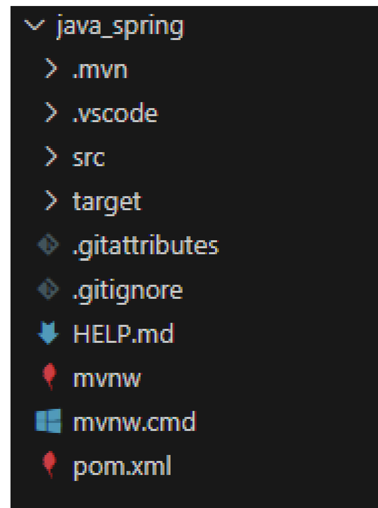

Figura 33 Carpetas del Proyecto

De todas estas carpetas generadas para el proyecto, las más interesantes se encuentran dentro de src, ya que aquí estará la carpeta main, donde se debe colocar toda la API, y la carpeta resources, que es donde se coloca principalmente las credenciales de la base de datos. Estas carpetas se muestran a continuación:

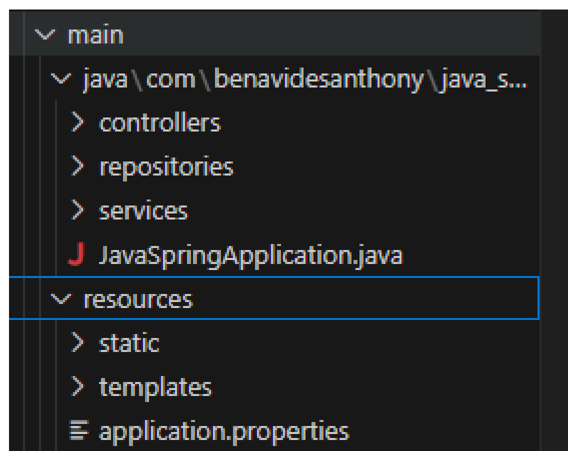

Figura 34 Carpeta Resources

Antes de continuar con el desarrollo de la API, es importante aclarar que durante la configuración inicial del proyecto se seleccionó **Maven** como gestor de dependencias, esta herramienta no se encuentra instalada por defecto en todos los entornos, por lo que si se intenta compilar o ejecutar el proyecto sin tener Maven correctamente instalado y configurado, se presentarán

errores indicando que no se pueden resolver las dependencias necesarias. Por este motivo, a continuación se describirá brevemente cómo se configura Maven en el entorno local, antes de proceder con la implementación de la API.

Configuración de Maven

Para comenzar necesitamos descargar Maven de su página oficial, ya que es una herramienta totalmente gratuita:

Figura 35 Página de descarga de Maven

Como vemos la versión más reciente y estable de Maven es la 3.9.9, por lo que esta es la que será usada, tras tener la descarga completada, necesitamos tener el archivo en formato .zip y moverlo dentro de una capeta en nuestro sistema que sea fácilmente identificable, en este caso se creó la carpeta Maven y se colocó el archivo dentro y se descomprimió:

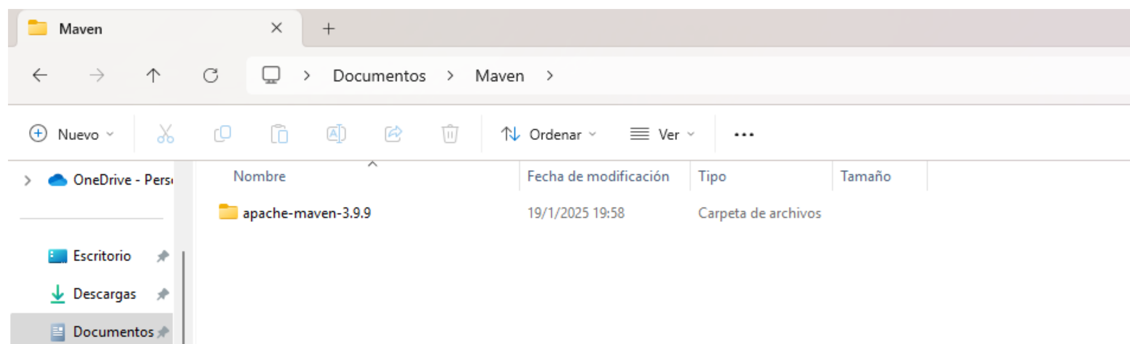

Figura 36 Carpeta de Maven

Luego de descomprimir el archivo, es necesario configurar Maven como una variable de entorno en el sistema, esto se realiza con el propósito de que el sistema operativo reconozca el comando mvn desde cualquier terminal o línea de comandos, lo cual permite compilar y gestionar el proyecto junto con todas las librerías que incluimos.

Por eso tras entrar en las variables de entorno del sistema hay que añadir una nueva y le colocarles la ruta de la carpeta creada anteriormente y el nombre de “MAVEN_HOME”:

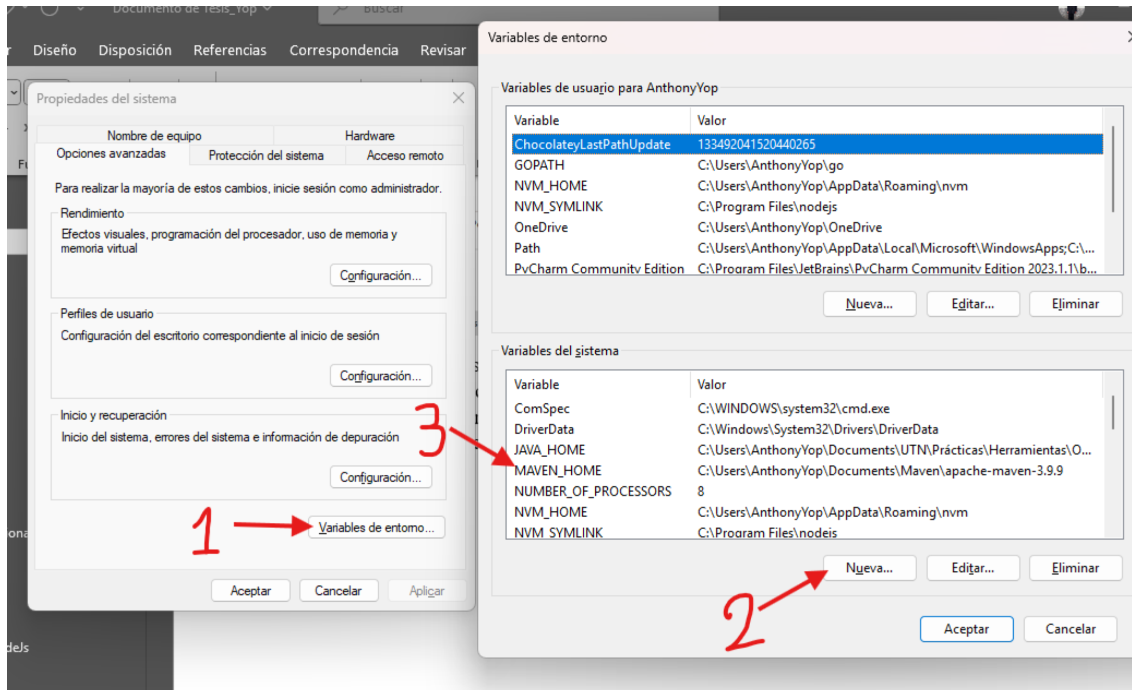

Figura 37 Pasos para agregar una variable de entorno

Tal como se indica en la imagen, los pasos a seguir son sencillos. Tras tener la nueva variable de entorno ya creada se da clic en aceptar para ambas ventanas, con la finalidad de guardar estos cambios, pero aún queda otro paso a seguir y es que se necesita volver a abrir la ventana de las variables de entorno pero ahora toca dirigirse a editar la variable de entorno llamada "Path" y a esta se le añade una característica más con el nombre de "%MAVEN_HOME\bin%", tal como se muestra:

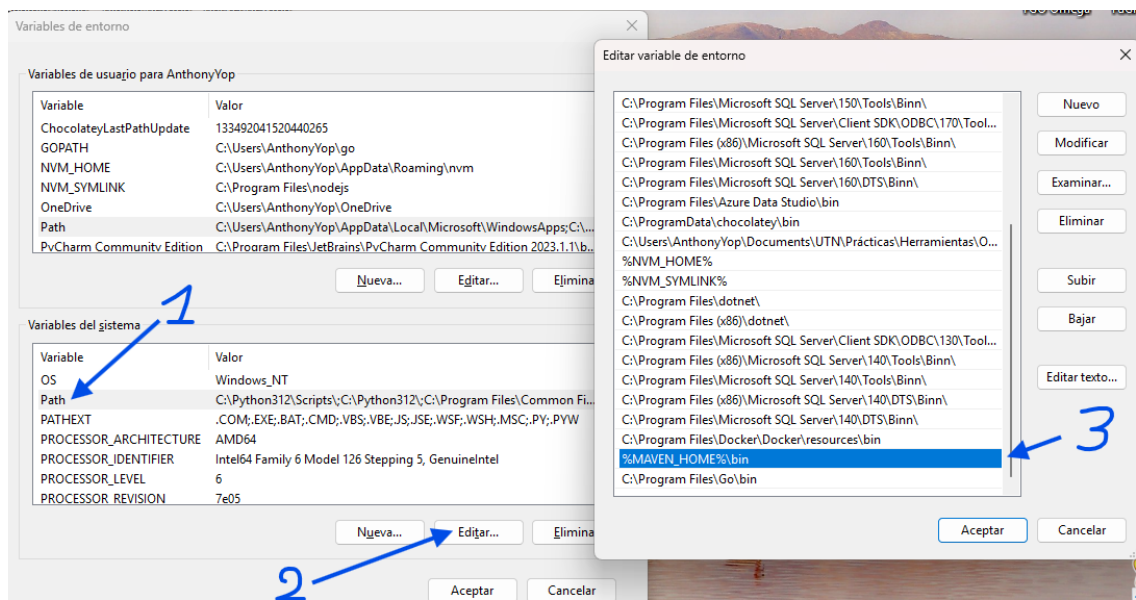

Figura 38 Pasos para editar la variable Path

Con esto queda terminada la instalación de Maven para trabajar con SpringBoot, aunque como recomendación, es preferible reiniciar el computador para que no haya problemas que eviten que se reconozca la variable de entorno creada.

Entonces tras reiniciar la maquina se puede comprobar que Maven está funcionando correctamente desde una terminal del sistema, consultando "mvn -v", con lo que deberíamos obtener algo similar a la siguiente imagen:

```
C:\Users\AnthonyYop>mvn -v
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: C:\Users\AnthonyYop\Documents\Maven\apache-maven-3.9.9
```

Figura 39 Comprobación de la instalación de Maven

Desarrollo de la API

Con Maven ya funcional en nuestro sistema, se puede seguir con el desarrollo de la API, para esto las 3 principales capetas con las que se trabajará serán: controllers, repositories, y services, distribución que se asemeja bastante a con las de las 2 APIs anteriores. Por lo que se tratará su código brevemente, primero a diferencia de las anteriores, en SpringBoot es necesario establecer las claves de acceso a la base de datos en un archivo aparte denominado "application.properties", en lugar de hacerlo directamente donde se realizan las consultas, esto es debido a que centralizar la configuración en un solo archivo permite separar la lógica de negocio de la configuración, facilita el mantenimiento, y mejora la seguridad al evitar que datos sensibles estén distribuidos en múltiples archivos, por lo que se colocan las credenciales de acceso como se muestra a continuación:

```
application.properties
java_spring > src > main > resources > application.properties
1  spring.datasource.url=jdbc:postgresql://localhost:5432/Blogs
2  spring.datasource.username=postgres
3  spring.datasource.password=100101
4  spring.datasource.driver-class-name=org.postgresql.Driver
5
```

Figura 40 Credenciales de acceso para la base de datos

Con la conexión a la base de datos ya hecha, se procede con la modificación de los métodos que realicen las consultas a la base de datos, esto hay que hacerlo en el archivo que se encuentra en "repositories", y esto es porque el patrón Repository permite abstraer el acceso a datos, encapsulando la lógica de persistencia y facilitando la reutilización y el testeo del código. Lo primero que se realiza en cada API es la creación de una variable que actúe como puente de conexión con la base de datos. En este caso, Spring Boot ofrece la clase JdbcTemplate, que simplifica la ejecución de consultas SQL y el manejo de resultados.

Gracias a esta herramienta, el código queda estructurado de la siguiente forma:

```java
import org.springframework.jdbc.core.JdbcTemplate;

@org.springframework.stereotype.Repository
public class Repository {

    private final JdbcTemplate jdbcTemplate;

    public Repository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
```

Figura 41 Código de JdbcTemplate

Siguiendo la misma lógica que en los proyectos anteriores, el primer método desarrollado es getUsuarios, el cual permite consultar una cantidad determinada de usuarios, este método incluye una estructura condicional basada en el número de usuarios a recuperar, tal como se había implementado previamente:

```java
public List<?> getUsuariosCantidad(int numUsr) {
    String sql;
    if(numUsr <= 320){
        sql = "SELECT * FROM usuarios WHERE id_usuario between 1 and 320 ORDER BY RANDOM() LIMIT ?";
    }
    else{
        sql = "SELECT * FROM usuarios ORDER BY RANDOM() LIMIT ?";
    }
    return jdbcTemplate.queryForList(sql, numUsr);
}
```

Figura 42 Método GET Usuarios - Java

Dado que la lógica de todos estos métodos es idéntica solo variando el lenguaje de programación, no es necesario abordar a detalle cada método a menos que haya algo destacable que mencionar, dicho esto la siguiente consulta es la de publicaciones:

```java
public List<?> getPublicacionesCantidad(int idUsr, int numPst) {
    String sql;
    if(numPst <= 60){
        sql = "SELECT * FROM publicaciones WHERE id_usuario = ? AND fecha_publicacion IS NOT NULL ORDER
    }else{
        sql = "SELECT * FROM publicaciones where id_usuario = ? ORDER BY RANDOM() LIMIT ?";
    }
    return jdbcTemplate.queryForList(sql, idUsr, numPst);
}
```

Figura 43 Método GET Publicaciones - Java

De la consulta de publicaciones la que sigue es la que consulta los comentarios pertenecientes a una publicación, seguido de la que consulta las respuestas de un comentario y finalmente la que obtiene las reacciones de una respuesta:

```java
public List<?> getComentariosCantidad(int idPub, int numCmt) {
    String sql = "SELECT * FROM comentarios WHERE id_publicacion = ? ORDER BY RANDOM() LIMIT ?";
    return jdbcTemplate.queryForList(sql, idPub, numCmt);
}

public List<?> getRespuestasCantidad(int idCmt, int numRes) {
    String sql = "SELECT * FROM respuestas WHERE id_comentario = ? ORDER BY RANDOM() LIMIT ?";
    return jdbcTemplate.queryForList(sql, idCmt, numRes);
}

public List<?> getReaccionesCantidad(int idRes, int numReac) {
    String sql = "SELECT * FROM reacciones WHERE id_respuesta = ? ORDER BY RANDOM() LIMIT ?";
    return jdbcTemplate.queryForList(sql, idRes, numReac);
}
```

Figura 44 Método GET Comentarios, Respuestas y Reacciones - Java

Como se mencionó anteriormente, la lógica es la misma y lo único que varía son las características inherentes al lenguaje de Java.

Lo que sigue son los métodos que se encuentran en services, cuya función es actuar como intermediarios entre los repositories y los controllers. Su responsabilidad principal es contener la lógica de negocio que transforma los datos antes de ser enviados al controlador o luego de ser recuperados desde el repositorio, permitiendo mantener un diseño limpio y desacoplado. Se tiene uno por cada método de petición a la base de datos, de la siguiente forma:

```java
public List<?> getUsuariosCantidad(int numUsr) {
    return repository.getUsuariosCantidad(numUsr);
}

public List<?> getPublicacionesCantidad(int idUsr, int numPst) {
    return repository.getPublicacionesCantidad(idUsr, numPst);
}

public List<?> getComentariosCantidad(int idPub, int numCmt) {
    return repository.getComentariosCantidad(idPub, numCmt);
}

public List<?> getRespuestasCantidad(int idCmt, int numRes) {
    return repository.getRespuestasCantidad(idCmt, numRes);
}
```

Figura 45 Código de services

```java
public List<?> getReaccionesCantidad(int idRes, int numReac) {
    return repository.getReaccionesCantidad(idRes, numReac);
}
```

Figura 46 Código de services

Como se puede ver en cada uno de los métodos, estos reciben los datos que luego será, usados en la consulta a la base de datos, finalmente vamos con el código de la clase

"Controller.java", es en esta parte en donde la arquitectura varía con respecto a las 2 APIs anteriores, ya que aquí se maneja la ruta HTTP sobre el método, de la siguiente manera:

```
@GetMapping("/usuarios/{numUsr}")
public ResponseEntity<List<?>> getUsuariosCantidad(@PathVariable int numUsr) {
    return ResponseEntity.ok(service.getUsuariosCantidad(numUsr));
}
```

Figura 47 Manejo de las rutas HTTP

Es aquí donde se asigna una ruta HPPT para cada consulta definida, este código pertenece al método que obtiene una cantidad de usuarios solicitada, pero lo mismo pasa en los demás casos, como se muestra en la imagen a continuación:

```
@GetMapping("/publicaciones/{idUsr}/{numPst}")
public ResponseEntity<List<?>> getPublicacionesCantidad(@PathVariable int idUsr, @PathVariable int numPs
    return ResponseEntity.ok(service.getPublicacionesCantidad(idUsr, numPst));
}

@GetMapping("/comentarios/{idPub}/{numCmt}")
public ResponseEntity<List<?>> getComentariosCantidad(@PathVariable int idPub, @PathVariable int numCmt)
    return ResponseEntity.ok(service.getComentariosCantidad(idPub, numCmt));
}

@GetMapping("/respuestas/{idCmt}/{numRes}")
public ResponseEntity<List<?>> getRespuestasCantidad(@PathVariable int idCmt, @PathVariable int numRes)
    return ResponseEntity.ok(service.getRespuestasCantidad(idCmt, numRes));
}
```

Figura 48 Manejo de las rutas HTTP

```
@GetMapping("/reacciones/{idRes}/{numReac}")
public ResponseEntity<List<?>> getReaccionesCantidad(@PathVariable int idRes, @PathVariable int numReac)
    return ResponseEntity.ok(service.getReaccionesCantidad(idRes, numReac));
}
```

Figura 49 Manejo de las rutas HTTP

Con las rutas ya asignadas está todo listo para ejecutar esta API junto con las 2 anteriores, pero aún hacen falta las 3 APIs que trabajan con GraphQL, por lo que antes de realizar pruebas hace falta ver como estas se construyeron.

2.4 Desarrollo de las APIs GraphQL

2.4.1 API - JavaScript, Express y Node.js

La primer API a desarrollarse con la arquitectura de GraphQL, es (similar al caso anterior) la que usa como lenguaje JavaScript y el framework de Express, en este caso también se usará la librería de "pg" para crear la conexión a la base de datos pero a esta librería se le sumará la de "graphql" que será la que ayude a manejar esta arquitectura y finalmente la librería "graphql-scalars" que será necesaria para trabajar con tipos de datos especiales y que no los incluye la librería anterior, eso se tratará en el código más adelante. Finalmente la última librería será la de "apollo-server-express", esta librería incrementa las funcionalidades de Express.js, permitiéndole trabajar con GraphQL, y que de forma base esto no es posible.

Como es costumbre en este proyecto, lo primero que debe hacerse es instanciar el servidor que albergará la API, ya que anteriormente se habló tanto del framework como del lenguaje, la instanciación es idéntica a la vez anterior:

```
const app = express();

module.exports = app;

async function iniciarServidor(){
    const apolloServer = new ApolloServer({
        typeDefs,
        resolvers
    });
    await apolloServer.start();
    apolloServer.applyMiddleware({app});

    app.listen(5003, () => {
        console.log('Servidor iniciado en el puerto:' , 5003);
    })
};

iniciarServidor();
```

Figura 50 Código para el servidor de la API

El código presentado mantiene una estructura similar al de la primera API; sin embargo, en este caso se incorpora dentro de un método en el que además de utilizar Express, se inicializa el servidor de Apollo, que se integra como middleware en la aplicación de Express y, posteriormente, se procede a su ejecución. Debido a que se implementa como una función, resulta necesario invocarla explícitamente para que el servidor pueda ponerse en marcha.

Una vez iniciado el servidor, se introduce una característica fundamental de GraphQL: la definición de los types. Estos elementos permiten estructurar y gestionar los datos que serán consultados, facilitando su organización y manipulación, para este trabajo se definieron los siguientes types:

```
const {gql} = require('apollo-server-express');
const { GraphQLDateTime } = require('graphql-scalars');

const typeDefs = gql`
    scalar DateTime

    type Usuario{
        id_usuario: Int
        nombre: String
        correo: String
        clave: String
    }
    type Publicacion{
        id_publicacion: Int
        id_usuario: Int
        titulo: String
        contenido: String
        fecha_publicacion: DateTime
    }
```

Figura 51 Código para la definición de types

```
type Comentario{
    id_comentario: Int
    id_publicacion: Int
    texto: String
    fecha_comentario: DateTime
}
type Respuesta{
    id_respuesta: Int
    id_comentario: Int
    texto: String
    fecha_respuesta: DateTime
}
type Reaccion{
    id_reaccion: Int
    id_respuesta: Int
    id_tipo_reaccion: Int
    fecha_reaccion: DateTime
}
```

Figura 52 Código para la definición de types

En las imágenes puede observarse que cada type refleja de manera casi exacta los atributos de su tabla correspondiente en la base de datos, esto se realiza porque en GraphQL es indispensable definir la estructura de los datos a nivel del esquema antes de poder realizar cualquier consulta o mutación, puesto que dicho esquema funciona como un contrato que especifica qué información está disponible, qué operaciones pueden ejecutarse y qué formato tendrán los resultados, lo cual garantiza consistencia y control en la interacción entre cliente y servidor.

De manera complementaria, es necesario definir las querys que aceptará la API; en los enfoques tradicionales basados en REST, estas consultas eran sustituidas por rutas HTTP enviadas desde el cliente hacia la API mediante una URL, en cambio en GraphQL este mecanismo se transforma en métodos denominados querys que reciben parámetros y devuelven la información solicitada. Además, cada query debe indicar explícitamente el tipo (type) de dato que retornará, lo que otorga mayor precisión en el manejo de la información. A continuación se muestran las queries implementadas en esta API:

```
type Query{
    getUsuariosCantidad(numUsr: Int): [Usuario]
    getPublicacionesCantidad(idUsr: Int, numPst: Int): [Publicacion]
    getComentariosCantidad(idPub: Int, numCmt: Int): [Comentario]
    getRespuestasCantidad(idCmt: Int, numRes:Int): [Respuesta]
    getReaccionesCantidad(idRes: Int, numReac: Int): [Reaccion]
}
```

Figura 53 Definición de las querys

Una vez definida la forma en que el cliente solicita la información, es necesario describir cómo estos métodos se comunican con la base de datos para obtener lo requerido. Los componentes responsables de esta tarea se denominan resolvers, ya que se encargan de "resolver" cada consulta o mutación especificada en el esquema de GraphQL. En otras palabras, un resolver es una función que implementa la lógica para acceder a la fuente de datos correspondiente, procesar la petición y retornar el resultado en el formato definido por el type asociado.

Para ilustrar lo anterior, se presenta a continuación el código correspondiente al primer método, el cual tiene como finalidad consultar un número específico de usuarios:

```
const resolvers = {
    DateTime: GraphQLDateTime,
    Query: {
        getUsuariosCantidad: async(_, args) =>{
            const {numUsr} = args;
            try {
                let ordenSql;
                if(numUsr <= 320){
                    ordenSql = `SELECT * FROM usuarios WHERE id_usuario between 1 and 320 ORDER BY RANDOM()
                }
                else{
                    ordenSql = "SELECT * FROM usuarios ORDER BY RANDOM() LIMIT " + numUsr + ";";
                }
                const respuesta = await conexionBD.query(ordenSql);
                return respuesta.rows;
            } catch (error) {
                console.log(error.message);
            }
        },
```

Figura 54 Método GET Usuarios – JavaScript GraphQL

En la imagen se observa que los resolvers se agrupan dentro de una única variable, lo que permite exportarla y utilizarla fuera de la clase principal. Dentro de esta estructura se encuentra la sección denominada Query, en la cual se implementa la lógica correspondiente a las operaciones que en este caso son métodos que realizan la extracción de información desde la base de datos, valiéndose de la conexión establecida anteriormente de la siguiente manera:

```
const {Pool} = require('pg');
const { GraphQLDateTime } = require('graphql-scalars');

const conexionBD = new Pool({
    host: 'localhost',
    user: 'postgres',
    password: '100101',
    database: 'Blogs',
});
```

Figura 55 Conexión a la base de datos

La conexión a la base de datos se establece de manera similar a la utilizada en la primera API REST, dado que se emplea la misma librería de conexión (pg). En cuanto a los métodos, se incluye, el procedimiento para obtener una cantidad determinada de publicaciones asociadas a un usuario, el cual mantiene el mismo patrón de implementación, esto se debe principalmente a que el código se desarrolla en JavaScript, lo que genera una API con estructura comparable a la primera desarrollada bajo REST, diferenciándose únicamente en los elementos específicos necesarios para adaptarse a la arquitectura GraphQL. A continuación, se presentan de manera resumida los métodos restantes, dado que no incorporan características adicionales relevantes que no hayan sido previamente discutidas:

```
getPublicacionesCantidad: async(_, args) =>{
    const {idUsr, numPst} = args;
    try {
        let ordenSql;
        if(numPst <= 60){
            ordenSql = `SELECT * FROM publicaciones where id_usuario=${idUsr} AND fecha_publicacion
        }else{
            ordenSql = `SELECT * FROM publicaciones where id_usuario=${idUsr} ORDER BY RANDOM() LIMI
        }
        const respuesta = await conexionBD.query(ordenSql);
        return respuesta.rows;
    } catch (error) {
        console.log(error.message);
    }
},
```

Figura 56 Método GET Publicaciones – JavaScript GraphQL

```
getComentariosCantidad: async(_, args) =>{
    const {idPub, numCmt} = args;
    try {
        const ordenSql = `SELECT * FROM comentarios where id_publicacion=${idPub} ORDER BY RANDOM()
        const respuesta = await conexionBD.query(ordenSql);
        return respuesta.rows;
    } catch (error) {
        console.log(error.message);
    }
},
```

Figura 57 Método GET Comentarios – JavaScript GraphQL

```
getRespuestasCantidad: async(_, args) =>{
    const {idCmt, numRes} = args;
    try {
        const ordenSql = `SELECT * FROM respuestas where id_comentario=${idCmt} ORDER BY RANDOM() LIMIT ${numRes};`;
        const respuesta = await conexionBD.query(ordenSql);
        return respuesta.rows;
    } catch (error) {
        console.log(error.message);
    }
},
getReaccionesCantidad: async(_, args) =>{
    const {idRes, numReac} = args;
    try {
        const ordenSql = `SELECT * FROM reacciones where id_respuesta=${idRes} ORDER BY RANDOM() LIMIT ${numReac};`;
        const respuesta = await conexionBD.query(ordenSql);
        return respuesta.rows;
    } catch (error) {
        console.log(error.message);
    }
},
```

Figura 58 Método GET Publicaciones y Reacciones – JavaScript GraphQL

Con toda esta arquitectura ya establecida y sabiendo el funcionamiento de GraphQL, queda terminada la revisión de la API construida bajo la arquitectura de GraphQL, con el lenguaje de JavaScript, y el Framework de Express.

2.4.2 API - Python y Flask

La siguiente API a tratar es la que está construida con Python, en este caso además del lenguaje también se repite el framework ya que Flask da la posibilidad de extender su funcionalidad y trabajar no solamente con arquitectura REST sino también con GraphQL, y para esto únicamente es necesario instalar las librería correspondientes antes de empezar el proyecto, estas librerías son:

- flask
- flask-graphql
- graphene
- psycopg2
- flask-cors

La librería principal continúa siendo Flask, mientras que flask-graphql se utiliza para integrar soporte de GraphQL en la aplicación, por otra parte Graphene permite definir esquemas, tipos y resolvers de GraphQL de manera estructurada y eficiente, facilitando la construcción de consultas y mutaciones. Psycopg2 se emplea nuevamente para la interacción con la base de datos PostgreSQL, y flask-cors habilita el control de accesos desde diferentes orígenes, permitiendo solicitudes HTTP entre dominios de manera segura.

Con las librerías instaladas, el siguiente paso consiste en crear el servidor que ejecutará la API, proceso que se realiza de la siguiente manera:

```python
from flask import Flask
from flask_graphql import GraphQLView
from flask_cors import CORS
from resolvers import schema

app = Flask(__name__)
CORS(app)

app.add_url_rule(
    "/graphql",
    view_func=GraphQLView.as_view(
        "graphql",
        schema=schema,
        graphiql=True
    )
)

if __name__ == "__main__":
    app.run(port=5004, debug=True)
```

Figura 59 Instanciación del servidor

A diferencia de la API REST, en la cual no fue necesario utilizar CORS debido a que las solicitudes se realizaban desde el mismo dominio que el servidor, en este caso sí se requiere habilitarlo, ya que la API basada en GraphQL puede recibir peticiones desde distintos orígenes, lo que demanda un control de acceso adecuado para garantizar el correcto funcionamiento de las interacciones. Posteriormente, es necesario integrar al servidor de Flask el paquete GraphQLView, permitiendo que la aplicación soporte consultas y mutaciones propias de GraphQL, tras esto y una vez establecido el servidor, el siguiente paso consiste en definir los esquemas de la API siguiendo la misma metodología empleada en la API desarrollada en JavaScript; a continuación se presentan las clases correspondientes:

```python
class Usuario(graphene.ObjectType):
    id_usuario = graphene.Int()
    nombre = graphene.String()
    correo = graphene.String()
    clave = graphene.String()

class Publicacion(graphene.ObjectType):
    id_publicacion = graphene.Int()
    id_usuario = graphene.Int()
    titulo = graphene.String()
    contenido = graphene.String()
    fecha_publicacion = DateTime()

class Comentario(graphene.ObjectType):
    id_comentario = graphene.Int()
    id_publicacion = graphene.Int()
    texto = graphene.String()
    fecha_comentario = DateTime()
```

Figura 60 Clases Usuario y Comentario

```python
class Respuesta(graphene.ObjectType):
    id_respuesta = graphene.Int()
    id_comentario = graphene.Int()
    texto = graphene.String()
    fecha_respuesta = DateTime()

class Reaccion(graphene.ObjectType):
    id_reaccion = graphene.Int()
    id_respuesta = graphene.Int()
    id_tipo_reaccion = graphene.Int()
    fecha_reaccion = DateTime()
```

Figura 61 Clases Respuesta y Reaccion

La definición de las clases corresponde nuevamente a las tablas de la base de datos, lo que permite mantener una estructura coherente entre la capa de persistencia y la lógica de la API, además esta correspondencia facilita la implementación de resolvers, garantiza la integridad de los datos y simplifica la gestión de relaciones entre entidades, optimizando el desarrollo de consultas complejas. Con los esquemas definidos y listos para su uso, es posible continuar con la configuración de las consultas (querys) que el servidor aceptará y que serán redireccionadas hacia los métodos de consulta correspondientes:

```
class Query(graphene.ObjectType):
    getUsuariosCantidad = graphene.List(Usuario, numUsr = graphene.Int())
    getPublicacionesCantidad = graphene.List(Publicacion, idUsr = graphene.Int(), numPst = graphene.Int())
    getComentariosCantidad = graphene.List(Comentario, idPub = graphene.Int(), numCmt = graphene.Int())
    getRespuestasCantidad = graphene.List(Respuesta, idCmt = graphene.Int(), numRes = graphene.Int())
    getReaccionesCantidad = graphene.List(Reaccion, idRes = graphene.Int(), numReac = graphene.Int())
```

Figura 62 Clase Query

Los métodos de esta API mantienen una estructura similar a la de la API anterior, con el objetivo de preservar la coherencia en el diseño y la organización del código. A continuación, se presentan los resolvers responsables de realizar las consultas a la base de datos, comenzando por aquel encargado de recuperar la información correspondiente a los usuarios:

```
def resolve_getUsuariosCantidad(self, info, numUsr):
    conn = getConexionBD()
    cur = conn.cursor()
    if numUsr <= 320:
        query = f"""SELECT * FROM usuarios
            WHERE id_usuario BETWEEN 1 AND 320
            ORDER BY RANDOM()
            LIMIT {numUsr};"""
    else:
        query = f"SELECT * FROM usuarios ORDER BY RANDOM() LIMIT {numUsr};"
    cur.execute(query)
    rows = cur.fetchall()
    cur.close()
    conn.close()
    return [Usuario(*row) for row in rows]
```

Figura 63 Método getUsuarios

En esta sección se retoman las consultas a la base de datos; fuera del uso de la librería específica correspondiente, la lógica subyacente se mantiene consistente con la implementada en las APIs anteriores.

Una vez comprendido el funcionamiento general de dichas consultas, las siguientes se presentan de manera resumida, dado que, más allá de las particularidades impuestas por el lenguaje y la arquitectura utilizada, la lógica permanece inalterada, como se ilustra a continuación:

```python
def resolve_getPublicacionesCantidad(self, info, idUsr, numPst):
    conn = getConexionBD()
    cur = conn.cursor()
    if numPst <= 60:
        query = f"""
        SELECT * FROM publicaciones
        WHERE id_usuario = {idUsr}
        AND fecha_publicacion IS NOT NULL
        ORDER BY RANDOM() LIMIT {numPst};"""
    else:
        query = f"""SELECT * FROM publicaciones WHERE id_usuario = {idUsr}
            ORDER BY RANDOM() LIMIT {numPst};"""
    cur.execute(query)
    rows = cur.fetchall()
    cur.close()
    conn.close()
    return [Publicacion(*row) for row in rows]
```

Figura 64 Método getPublicaciones

Lo mismo ocurre en los siguientes métodos:

```python
def resolve_getComentariosCantidad(self, info, idPub, numCmt):
    conn = getConexionBD()
    cur = conn.cursor()
    cur.execute(f"SELECT * FROM comentarios WHERE id_publicacion={idPub} ORDER BY RANDOM() LIMIT {numCmt
    rows = cur.fetchall()
    cur.close()
    conn.close()
    return [Comentario(*row) for row in rows]

def resolve_getRespuestasCantidad(self, info, idCmt, numRes):
    conn = getConexionBD()
    cur = conn.cursor()
    cur.execute(f"SELECT * FROM respuestas WHERE id_comentario={idCmt} ORDER BY RANDOM() LIMIT {numRes};
    rows = cur.fetchall()
    cur.close()
    conn.close()
    return [Respuesta(*row) for row in rows]
```

Figura 65 Métodos get Comentarios y getRespuestas

Es notable que tanto la lógica como las sentencias SQL usadas para cada consulta son prácticamente idénticas, únicamente adaptando cada sentencia para que sea capaz de ejecutarse en un lenguaje y arquitectura diferentes.

Finalmente el método de obtener los datos del quinto nivel, que en este caso corresponde a las reacciones a un comentario:

```python
def resolve_getReaccionesCantidad(self, info, idRes, numReac):
    conn = getConexionBD()
    cur = conn.cursor()
    cur.execute(f"SELECT * FROM reacciones WHERE id_respuesta={idRes} ORDER BY RANDOM() LIMIT {numReac};
    rows = cur.fetchall()
    cur.close()
    conn.close()
    return [Reaccion(*row) for row in rows]
```

Figura 66 Método getReacciones

Un detalle que puede pasar desapercibido es la presencia del parámetro info en los métodos, el cual no se utiliza directamente en la lógica interna de los mismos, este parámetro es propio de la implementación de GraphQL y proporciona información contextual sobre la ejecución de la consulta, como el campo solicitado, el esquema o el resolver padre, siendo incluido automáticamente en todos los métodos por la librería utilizada. A pesar de que info permite acceder a esta información, en el presente caso su uso no resulta necesario, por lo que se encuentra instanciado únicamente debido a su carácter obligatorio en la definición de los resolvers, sin afectar la funcionalidad de los métodos por conveniencia de implementación.

2.4.3 API - Go y Gin

La última API a analizar está desarrollada con el lenguaje Go y el framework Gin (Gin-Gonic), al igual que en los casos anteriores será necesario emplear diversas librerías para extender las funcionalidades del lenguaje y permitir el trabajo con las herramientas establecidas previamente. Al tratarse de un lenguaje diferente y relativamente reciente, Go presenta ciertas características específicas que se detallarán a medida que se describa el desarrollo paso a paso de la API.

Las librerías utilizadas para esta API son las siguientes:

- glqgen: permite implementar la arquitectura GraphQL de manera estructurada, generando automáticamente los resolvers y esquemas a partir de definiciones de tipo.
- gin: es el framework para usarse en este caso, ligero y de alto rendimiento para construir servidores web y manejar rutas de forma eficiente.
- gqlparser: proporciona funcionalidades para analizar y validar esquemas y consultas GraphQL, facilitando la interpretación de la estructura de los querys.
- postgres: librería que habilita la conexión con la base de datos PostgreSQL, de manera similar a los casos anteriores.
- gorm: simplifica la interacción con la base de datos, permitiendo realizar operaciones CRUD y manejar relaciones entre entidades de forma estructurada.

Antes de realizar la instalación de las librerías, es necesario crear el proyecto, similar a lo que se hace en JavaScript con el comando "npm init", en este caso el comando que se usa es "go mod init nombre_del_proyecto", con esto se crea la estructura de carpetas necesarias para trabajar en el proyecto. Con el proyecto ya iniciado, ahora si es posible instalar las librerías antes mencionadas, como se muestra a continuación:

```
go get github.com/gin-gonic/gin
go get github.com/99designs/gqlgen
go get gorm.io/gorm
go get gorm.io/driver/postgres
```

Figura 67 Instalación de librerías necesarias

Con las librerías previamente instaladas, el siguiente paso consiste en ejecutar el comando que genera un archivo base, el cual contendrá los types, que representan los modelos de la base de datos. El comando empleado para este propósito es "go run github.com/99designs/gqlgen init", que una vez ejecutado genera un archivo denominado schema.graphqls. Con este archivo disponible, se procede a modificarlo para adaptarlo a los requerimientos específicos del caso actual, dejándolo configurado de la siguiente manera:

```
scalar DateTime

type Usuario {
  id_usuario: Int
  nombre: String
  correo: String
  clave: String
}

type Publicacion {
  id_publicacion: Int
  id_usuario: Int
  titulo: String
  contenido: String
  fecha_publicacion: DateTime
}
```

Figura 68 Código del archivo schema.graphqls

```
type Comentario {
  id_comentario: Int
  id_publicacion: Int
  texto: String
  fecha_comentario: DateTime
}

type Respuesta {
  id_respuesta: Int
  id_comentario: Int
  texto: String
  fecha_respuesta: DateTime
}

type Reaccion {
  id_reaccion: Int
  id_respuesta: Int
  id_tipo_reaccion: Int
  fecha_reaccion: DateTime
}
```

Figura 69 Código del archivo schema.graphqls

Así mismo es en este archivo que definimos las querys que aceptará nuestra API, de forma similar a como se ha realizado en los casos anteriores:

```
type Query {
  getUsuariosCantidad(numUsr: Int): [Usuario]
  getPublicacionesCantidad(idUsr: Int, numPst: Int): [Publicacion]
  getComentariosCantidad(idPub: Int, numCmt: Int): [Comentario]
  getRespuestasCantidad(idCmt: Int, numRes: Int): [Respuesta]
  getReaccionesCantidad(idRes: Int, numReac: Int): [Reaccion]
}
```

Figura 70 Definición de las querys

Tras tener el código personalizado al proyecto, la librería glqgen facilita la generación de código basándose en el esquema que acabamos de definir, para esto solamente hay que ejecutar el siguiente comando:

- go run github.com/99designs/gqlgen generate

Con esto se crearán varios archivos que ayudarán a que la API funcione correctamente, para saber si estos archivos fueron generados automáticamente por la biblioteca, basta mirar el primer comentario al inicio de cada uno, ya que aquí se establece que dicho código fue generado, como se ve en la siguiente imagen:

```
models_gen.go ×
graph > model > models_gen.go > Comentario > IDPublicacion
1   // Code generated by github.com/99designs/gqlgen, DO NOT EDIT.
2
3   package model
4
5   type Comentario struct {
6       IDComentario     *int32  `json:"id_comentario,omitempty"`
7       IDPublicacion    *int32  `json:"id_publicacion,omitempty"`
8       Texto            *string `json:"texto,omitempty"`
9       FechaComentario  *string `json:"fecha_comentario,omitempty"`
10  }
```

Figura 71 Código generado automáticamente

```go
generated.go 9+ ×
graph > generated.go > parsedSchema
1  // Code generated by github.com/99designs/gqlgen, DO NOT EDIT.
2
3  package graph
4
5  import (
6      "Go_Gin_GQL/graph/model"
7      "bytes"
8      "context"
9      "embed"
10     "errors"
11     "fmt"
12     "strconv"
13     "sync"
14     "sync/atomic"
```

Figura 72 Código generado automáticamente

```go
schema.resolvers.go ×
graph > schema.resolvers.go > ...
1  package graph
2
3  // This file will be automatically regenerated based on the schema, any resolver implementations
4  // will be copied through when generating and any unknown code will be moved to the end.
5  // Code generated by github.com/99designs/gqlgen version v0.17.66
6
7  import (
8      "Go_Gin_GQL/graph/generated"
9      "Go_Gin_GQL/graph/model"
10     "context"
11 )
```

Figura 73 Código generado automáticamente

Con esto las bases del proyecto quedan establecidas, por lo que lo siguiente es establecer el servidor que es facilitado por el Framework de Gin, esto se logra mediante el siguiente código:

```go
func main() {
    r := gin.Default()

    database.ConectarBD()

    srv := handler.NewDefaultServer(
        generated.NewExecutableSchema(generated.Config{Resolvers: &graph.Resolver{}}))

    r.POST("/query", func(c *gin.Context) {
        srv.ServeHTTP(c.Writer, c.Request)
    })

    r.GET("/", func(c *gin.Context) {
        playground.Handler("GraphQL", "/query").ServeHTTP(c.Writer, c.Request)
    })

    log.Println("Servidor corriendo en http://localhost:5005/")
    r.Run(":5005")
}
```

Figura 74 Instanciación del servidor

En este caso, el servidor se configura para hospedarse en el puerto 5005, evitando posibles conflictos con otras APIs en ejecución. Antes de confirmar que el servidor funciona correctamente, se realiza una petición GET con el objetivo de verificar que la instancia del servidor está activa y responde adecuadamente a solicitudes externas, funcionando como una comprobación inicial de disponibilidad. Además, se observa un método externo encargado de establecer la conexión con la base de datos, denominado database.ConectarBD, este método se ubica en una clase separada para mantener organizada la estructura del proyecto y evitar interferencias con la configuración del servidor, el código utilizado para realizar la conexión con la base de datos es el siguiente:

```go
import (
    "log"

    "gorm.io/driver/postgres"
    "gorm.io/gorm"
)

var DB *gorm.DB

func ConectarBD() {
    var err error
    dsn := "host=localhost user=postgres password=100101 dbname=Blogs port=5432 sslmode=disable"
    DB, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})
    if err != nil {
        log.Fatal("Error al conectar a la base de datos:", err)
    }
}
```

Figura 75 Método de conexión a la base de datos

En esta sección se utiliza la librería postgres para establecer la comunicación con la base de datos, mientras que gorm facilita la manipulación de las tablas, la ejecución de operaciones CRUD y el manejo de relaciones entre entidades de manera estructurada y eficiente, tal como se mencionó anteriormente de forma general.

Como uno de los pasos finales, se procede a crear la lógica correspondiente a cada consulta, siguiendo un enfoque similar al empleado en las APIs anteriores, adaptando únicamente ciertas variables y sintaxis propias del lenguaje Go. A continuación, se presenta el primer método, encargado de consultar una cantidad determinada de usuarios:

```go
func (r *Resolver) GetUsuariosCantidad(ctx context.Context, numUsr int) ([]*model.Usuario, error) {
    var usuarios []*model.Usuario
    var query string

    if database.DB == nil {
        log.Println("Error: La conexión a la base de datos no está inicializada")
    }

    if numUsr <= 320 {
        query = fmt.Sprintf("SELECT * FROM usuarios WHERE id_usuario BETWEEN 1 AND 320 ORDER BY RANDOM()
    } else {
        query = fmt.Sprintf("SELECT * FROM usuarios ORDER BY RANDOM() LIMIT %d;", numUsr)
    }

    err := database.DB.Raw(query).Scan(&usuarios).Error
    return usuarios, err
}
```

Figura 76 Método GET Usuarios – Go GraphQL

La estructura de la consulta SQL y el condicional empleado se mantiene consistente con las implementaciones previas incluso desde las APIs REST, por lo que no es necesario profundizar en su descripción. De manera general, la lógica consiste en preparar un arreglo para almacenar los datos, verificar la existencia de una conexión activa con la base de datos, determinar la consulta a ejecutar en función del número de usuarios solicitados, enviar la consulta a la base de datos y finalmente procesar el resultado antes de devolverlo como respuesta.

A continuación, se presentan casos similares correspondientes a los métodos de consulta para los demás niveles de la base de datos:

```go
func (r *Resolver) GetPublicacionesCantidad(ctx context.Context, idUsr, numPst int) ([]*model.Publicacion,
    var publicaciones []*model.Publicacion
    var query string

    if numPst <= 60 {
        query = fmt.Sprintf("SELECT * FROM publicaciones WHERE id_usuario = %d AND fecha_publicacion IS NO
    } else {
        query = fmt.Sprintf("SELECT * FROM publicaciones WHERE id_usuario = %d ORDER BY RANDOM() LIMIT %d;
    }

    err := database.DB.Raw(query).Scan(&publicaciones).Error
    return publicaciones, err
}
```

Figura 77 Método GET Publicaciones – Go GraphQL

```
func (r *Resolver) GetComentariosCantidad(ctx context.Context, idPub, numCmt int) ([]*model.Comentario,
    var comentarios []*model.Comentario
    var query = fmt.Sprintf("SELECT * FROM comentarios where id_publicacion=%d ORDER BY RANDOM() LIMIT %
    err := database.DB.Raw(query).Scan(&comentarios).Error
    return comentarios, err
}

func (r *Resolver) GetRespuestasCantidad(ctx context.Context, idCmt, numRes int) ([]*model.Respuesta, er
    var respuestas []*model.Respuesta
    var query = fmt.Sprintf("SELECT * FROM respuestas where id_comentario=%d ORDER BY RANDOM() LIMIT %d;
    err := database.DB.Raw(query).Scan(&respuestas).Error
    return respuestas, err
}

func (r *Resolver) GetReaccionesCantidad(ctx context.Context, idRes, numReac int) ([]*model.Reaccion, er
    var reacciones []*model.Reaccion
    var query = fmt.Sprintf("SELECT * FROM reacciones where id_respuesta=%d ORDER BY RANDOM() LIMIT %d;"
    err := database.DB.Raw(query).Scan(&reacciones).Error
    return reacciones, err
}
```

Figura 78 Métodos GET Comentarios, Respuestas y Reacciones – Go GraphQL

Recalcando nuevamente sin intentar caer en la redundancia, la lógica general de los resolvers es muy similar a las APIs anteriores, compartiendo con exactitud el código SQL, y únicamente variando en las características del lenguaje, en este caso Go.

Con la revisión de esta última API, finalmente se han tratado las seis que serán usadas y consumidas por un cliente que será el encargado de llevar a cabo el experimento computacional.

2.5 Desarrollo del Cliente

Tal como se menciona en el alcance del proyecto, la finalidad del cliente es consumir cada una de las seis diferentes APIs desarrolladas, tomar el tiempo de respuesta de cada consulta realizada a cada API, y almacenar esta información. Fuera de estas funcionalidades las herramientas que se usaron para su construcción fueron decididas por recomendación del PhD. Antonio Quiña, estableciendo que el cliente será meramente Backend y los resultados y demás información importante que muestre sea por la terminal de Visual Studio Code, y el lenguaje que se usará es JavaScript dada su flexibilidad para consumir APIs.

Adicionalmente ya que durante el experimento se recopilarán una gran cantidad de datos, que se guardarán en un documento de Excel que posteriormente será analizado en búsqueda de resultados; hacer este proceso de forma manual implicaría un gasto excesivo de tiempo, por lo que además de consumir las APIs según el caso de uso, el cliente también optimizará el proceso de guardado, colocando automáticamente los resultados en un mismo documento de Excel sin necesidad de intervención, esto se mostrará a detalle más adelante.

En primer lugar, se revisan las bibliotecas de Node.js necesarias para este subproyecto:

- node-fetch: permite realizar solicitudes HTTP de manera sencilla y manejar las respuestas de manera asincrónica, facilitando la interacción con servicios externos.
- graphql-request: será la encargada de enviar las solicitudes a las APIs con arquitectura GraphQL, puesto que estas reciben las peticiones a menara de “querys” en lugar de URLs como se hace en REST

- xlsx-populate: esta librería será la que maneje el guardado automático de datos en un único documento de Excel.

Con las bibliotecas establecidas, el siguiente paso consiste en definir la estructura completa del cliente, la cual se organizará de la siguiente manera:

1. Primero una clase contendrá los métodos necesarios para hacer las consultas a cada nivel de la base de datos a través de la API.
2. Luego como se describe el experimento computacional, cada consulta se debe ejecutar tres veces por tanto y por motivos de optimización, en lugar de ejecutar manualmente cada consulta, se considera mejor manejar las tres consultas automáticamente en una clase propia que ejecute cada caso de uso con solo una llamada.
3. Finalmente el proceso termina con el guardado del tiempo de ejecución en el documento de Excel, junto con más datos relevantes tales como:

- Id del caso de uso: ID que identifica el caso de uso ejecutado por ejemplo el caso de uso 1 se identifica como CU1, el caso de uso 2 como CU2, y así sucesivamente.
- Arquitectura: Es la arquitectura de la API consultada, puede ser REST o GraphQL.
- Tecnología: Hace en este campo se especificará tanto el lenguaje como el framework que se usó para la construcción de la API.
- Número de registros: Es el total de datos obtenidos de lavase de datos, este es prácticamente el producto de la distribución.
- Distribución: Refleja la distribución antes establecida para cada caso de uso.
- Número de iteración: Indica si los datos guardados corresponden a la primer, segunda o tercer iteración.
- Tiempo (ms): El tiempo de ejecución registrado y guardado en milisegundos.
- Registro de Iteraciones: Muestra los datos que fueron consultados en cada nivel es decir, si trabajamos con el caso de uso 3, la distribución es de 10x10x10 y por ende el total de datos obtenidos es 1000, en esta casilla se registrará que los datos obtenidos del primer nivel fueron 10, del segundo nivel fueron 100, y finalmente del tercer nivel fueron 1000 (10-100-1000).

Si la ejecución del caso de uso fue correcta y los datos se pudieron guardar sin inconvenientes, se mostrará un mensaje en la terminal de que el proceso ha finalizado. Con el funcionamiento general del cliente ya explicado, veamos a detalle el código de cada parte.

2.5.1 Experimento computacional REST

La clase responsable de ejecutar el experimento computacional está compuesta por varios métodos, los cuales se encargan de consultar a la API los datos correspondientes a cada tabla, organizados según su nivel como se ilustra a continuación:

```
export const obtenerDatosPrimerNivel = async (urlBase, numDatosPrimerNivel) => {
  const urlConsulta = urlBase + '/usuarios/' + numDatosPrimerNivel;
  try {
    let usuariosConsultados = [];
    const tInicial = new Date();

    const respuestaAPI = await fetch(urlConsulta);
    const tFinal = new Date() - tInicial;

    const datosEnJson = await respuestaAPI.json();
    usuariosConsultados = [...usuariosConsultados, ...datosEnJson];

    const cantidadDatosUsuarios = usuariosConsultados.length + "";
    console.log('R Usrs =====>', usuariosConsultados.length);

    return {
      datos: usuariosConsultados.filter((d) => d !== null),
      tiempo: tFinal,
      cadenaDatos: cantidadDatosUsuarios
    };
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Figura 79 Método para obtener datos del primer nivel - REST

Como indica su nombre, este método se encarga de obtener los datos correspondientes al primer nivel de la base de datos, es decir, realiza la consulta de los usuarios en la tabla con el mismo nombre, el método además se define como asíncrono, debido a la naturaleza de las solicitudes realizadas a la API.

Cabe destacar que la estructura general de este método se mantiene en los siguientes, por lo que se detallará su funcionamiento únicamente en este caso para evitar redundancias.

En primer lugar, se construye la URL de consulta a partir de una URL base, que sirve como elemento reutilizable para todos los métodos, a la cual se agregan los parámetros específicos de la consulta, a continuación y dentro de un bloque de control de excepciones (try-catch), se crea la variable destinada a almacenar los datos consultados y se registra el tiempo inicial de ejecución. Este registro se utiliza para calcular el tiempo de respuesta, restando posteriormente el tiempo final, de manera que la medición refleje con precisión únicamente la duración de la consulta, minimizando la influencia del procesamiento posterior de los datos.

Una vez obtenidos los datos, se transforman a formato JSON para ser devueltos como respuesta. El resto del código tiene fines principalmente visuales, mostrando la cantidad de usuarios consultados para mantener un registro en caso de errores o datos faltantes, este conteo también se incluye en la respuesta, permitiendo completar la casilla de Registro de Iteraciones mencionada previamente.

A continuación, se procede con el método correspondiente al segundo nivel:

```
export const obtenerDatosSegundoNivel = async (urlBase, numDatosPrimerNivel, numDatosSegundoNivel) => {
  try {
    let publicacionesConsultadas = [];
    const tInicial = new Date();

    const { datos: usuarios, cadenaDatos: cadenaUsrs } = await obtenerDatosPrimerNivel(urlBase, numDatosPrim
    const idsUsuarios = usuarios.map((res) => res.id_usuario);

    for (const id of idsUsuarios) {
      const urlConsulta = urlBase + '/publicaciones/' + id + '/' + numDatosSegundoNivel;

      const publicaciones = await fetch(urlConsulta);
      const datosEnJson = await publicaciones.json();

      publicacionesConsultadas = [...publicacionesConsultadas, ...datosEnJson];
    }

    const tFinal = new Date() - tInicial;

    const cadenaPubls = cadenaUsrs + " - " + publicacionesConsultadas.length;
    console.log('R Publs =====>', publicacionesConsultadas.length);

    return {
      datos: publicacionesConsultadas.filter((d) => d !== null),
      tiempo: tFinal,
      cadenaDatos: cadenaPubls
    };
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Figura 80 Método para obtener datos del segundo nivel - REST

En este caso el método recibe un mayor número de parámetros, ya que al corresponder al segundo nivel, debe también realizar la consulta de su nivel anterior. Para ello se emplea un enfoque anidado: para obtener las publicaciones asociadas a un usuario, primero es necesario contar con el Id de dicho usuario. Este se obtiene llamando al método del primer nivel, que devuelve los Ids requeridos, posteriormente se recorre la respuesta construyendo la URL de consulta para cada Id, lo que permite obtener la cantidad de publicaciones solicitadas.

Los métodos correspondientes a los niveles superiores comparten esta característica anidada, cada método consulta los datos del nivel inmediatamente anterior y tras procesar la información obtenida, realiza la consulta de su propio nivel. De este modo el quinto nivel llama al cuarto, el cuarto al tercero, el tercero al segundo y este último al primero; conforme se devuelven los datos, estos se van consolidando hasta que el quinto nivel retorna el resultado final.

Aplicando lo dicho a este método, primero se reciben tanto los datos solicitados del primer nivel como los correspondientes al segundo nivel, a continuación se crea la variable destinada a almacenar el resultado de la consulta a la API y se registra el tiempo inicial de ejecución. Seguidamente se llama al método del nivel anterior, es decir al del primer nivel, obteniendo de este la cantidad de usuarios y el conteo de los que se devuelven como respuesta. Posteriormente, se consultan las publicaciones asociadas a cada usuario, utilizando su Id como clave primaria, y se almacenan los datos obtenidos y finalmente, se calcula el tiempo de ejecución, se imprime el resultado y se devuelve la información consultada, junto con los registros correspondientes al primer y segundo nivel.

El siguiente método a revisar es el encargado de obtener los datos de tercer nivel:

```
export const obtenerDatosTercerNivel = async (urlBase, numDatosPrimerNivel, numDatosSegundoNivel, numD
  try {
    let comentariosConsultados = [];
    const tInicial = new Date();

    const { datos: publicaciones, cadenaDatos: cadenaPubls } = await obtenerDatosSegundoNivel(urlBase,
    const idsPublicaciones = publicaciones.map((res) => res.id_publicacion);

    for (const id of idsPublicaciones) {
      const urlConsulta = urlBase + '/comentarios/' + id + '/' + numDatosTercerNivel;

      const publicaciones = await fetch(urlConsulta);
      const datosEnJson = await publicaciones.json();

      comentariosConsultados = [...comentariosConsultados, ...datosEnJson];
    }

    const tFinal = new Date() - tInicial;

    const cadenaCmts = cadenaPubls + " - " + comentariosConsultados.length;
    console.log('R Cmts =======>', comentariosConsultados.length);

    return {
      datos: comentariosConsultados.filter((d) => d !== null),
      tiempo: tFinal,
      cadenaDatos: cadenaCmts
    };
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Figura 81 Método para obtener datos del tercer nivel - REST

A partir de este punto, no se presentan novedades significativas ya que la lógica general de los métodos sigue el mismo patrón establecido en el segundo nivel, aplicable a los niveles subsecuentes. De manera general, cada método realiza los siguientes pasos: se crea la variable destinada a almacenar los datos, se inicia la medición del tiempo de ejecución, se consultan los datos del nivel anterior y a partir de los resultados, se obtienen las Ids necesarias y la cadena de datos por nivel; a continuación, se ejecuta la consulta correspondiente a cada Id obtenida, los datos se procesan en formato JSON; finalmente, tras imprimir un mensaje con la información consultada del nivel actual, se devuelven los datos junto con los registros acumulados de todos los niveles anteriores, incluyendo el nivel actual.

```
export const obtenerDatosCuartoNivel = async (urlBase, numDatosPrimerNivel, numDatosSegundoNivel, numDatosTe
  try {
    let respuestasConsultadas = [];
    const tInicial = new Date();

    const { datos: comentarios, cadenaDatos: cadenaCmts } = await obtenerDatosTercerNivel(urlBase, numDatosP
    const idsComentarios = comentarios.map((res) => res.id_comentario);

    for (const id of idsComentarios) {
      const urlConsulta = urlBase + '/respuestas/' + id + '/' + numDatosCuartoNivel;
      const comentarios = await fetch(urlConsulta);
      const datosEnJson = await comentarios.json();

      respuestasConsultadas = [...respuestasConsultadas, ...datosEnJson];
    }

    const tFinal = new Date() - tInicial;

    const cadenaResps = cadenaCmts + " - " + respuestasConsultadas.length;
    console.log('R Resps =======>', respuestasConsultadas.length);

    return {
      datos: respuestasConsultadas.filter((d) => d !== null),
      tiempo: tFinal,
      cadenaDatos: cadenaResps
    };
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Figura 82 Método para obtener datos del cuarto nivel - REST

```
export const obtenerDatosQuintoNivel = async (urlBase, numDatosPrimerNivel, numDatosSegundoNivel, numDatosTe
  try {
    let reaccionesConsultadas = [];
    const tInicial = new Date();

    const { datos: respuestas, cadenaDatos: cadenaResps } = await obtenerDatosCuartoNivel(urlBase, numDatosP
    const idsRespuestas = respuestas.map((res) => res.id_respuesta);

    for (const id of idsRespuestas) {
      const urlConsulta = urlBase + '/reacciones/' + id + '/' + numDatosQuintoNivel;

      const respuestas = await fetch(urlConsulta);
      const datosEnJson = await respuestas.json();

      reaccionesConsultadas = [...reaccionesConsultadas, ...datosEnJson];
    }

    const tFinal = new Date() - tInicial;

    const cadenaReacs = cadenaResps + " - " + reaccionesConsultadas.length;
    console.log('R Reacs ========>', reaccionesConsultadas.length);

    return {
      datos: reaccionesConsultadas.filter((d) => d !== null),
      tiempo: tFinal,
      cadenaDatos: cadenaReacs
    };
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Figura 83 Método para obtener datos del quinto nivel - REST

Con los métodos de consulta a la API que se use, ya tratados, lo siguiente es revisar el desarrollo de los casos de uso.

2.5.2 Casos de uso REST

De forma similar a los casos anteriores, la forma en que se desarrolló el método para el caso de uso 1, sirvió de plantilla para los demás, por lo que el primero se tratará a detalle y el resto se discutirá brevemente. Para comenzar entonces lo primero es importar todos los métodos desarrollados en la clase anterior, tras esto se puede desarrollar el primer caso de uso, como se muestra a continuación:

```
import {
    obtenerDatosCuartoNivel,
    obtenerDatosPrimerNivel,
    obtenerDatosQuintoNivel,
    obtenerDatosSegundoNivel,
    obtenerDatosTercerNivel
} from './experimetoComputacional_REST.js';

export async function CasodeUso1R(urlBase, numdeUsuarios, arquiterctura, tecnologia) {
    let datosObtenidosLv1;
    let arregloResultado = new Array(3);
    let distribucion = '' + numdeUsuarios;
    arregloResultado[0] = ['Cu1', arquiterctura, tecnologia, numdeUsuarios, distribucion];
    arregloResultado[1] = ['Cu1', arquiterctura, tecnologia, numdeUsuarios, distribucion];
    arregloResultado[2] = ['Cu1', arquiterctura, tecnologia, numdeUsuarios, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv1 = await obtenerDatosPrimerNivel(urlBase, numdeUsuarios);
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv1.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv1.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 84 Código del Caso de uso 1 - REST

Tras la importación de los métodos, se procede a crear las variables que se utilizarán a lo largo del proceso, a continuación se genera un arreglo bidimensional para almacenar los datos, ya que la librería empleada para guardar la información en Excel requiere esta estructura; en este esquema, cada fila del arreglo corresponde a una fila en el documento y cada elemento del arreglo se interpreta como una columna.

Seguido de introducir los datos del arreglo y dentro de un bloque de control de excepciones, se implementa un bucle que se ejecuta tres veces, correspondiendo a la cantidad de iteraciones necesarias para aplicar la fórmula de tiempo medio de respuesta una vez completadas todas las consultas. Durante cada iteración se obtienen los datos del nivel correspondiente, en este caso del primer nivel; y a partir de los resultados se registran el tiempo de ejecución y la cadena de datos, que refleja los elementos consultados por nivel.

Al finalizar el método, se retorna el arreglo construido listo para ser guardado. Las características de este primer caso se repiten prácticamente de manera idéntica en los demás casos de uso, variando únicamente en la especificación del caso y los parámetros que recibe cada método, a continuación se presentan los demás casos de uso:

```
export async function CasodeUso2R(urlBase, numdeUsuarios, numdePosts, arquiterctura, tecnologia) {
    let datosObtenidosLv2;
    let numdeRegistros = numdeUsuarios * numdePosts;
    let distribucion = numdeUsuarios + 'x' + numdePosts;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu2', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu2', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu2', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv2 = await obtenerDatosSegundoNivel(urlBase, numdeUsuarios, numdePosts);
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv2.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv2.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 85 Código del Caso de uso 2 - REST

```
export async function CasodeUso3R(urlBase, numdeUsuarios, numdePosts, numdeComentarios, arquiterctura,
    tecnologia) {
    let datosObtenidosLv3;
    let let numdeRegistros: number + 'x' + numdePosts + 'x' + numdeComentarios;
    let numdeRegistros = numdeUsuarios * numdePosts * numdeComentarios;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu3', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu3', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu3', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv3 = await obtenerDatosTercerNivel(urlBase, numdeUsuarios, numdePosts, numdeComer
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv3.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv3.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 86 Código del Caso de uso 3 - REST

```
export async function CasodeUso4R(urlBase, numdeUsuarios, numdePosts, numdeComentarios, numdeRespuestas,
    arquiterctura, tecnologia) {
    let datosObtenidosLv4;
    let distribucion = numdeUsuarios + 'x' + numdePosts + 'x' + numdeComentarios + 'x' + numdeRespuestas;
    let numdeRegistros = numdeUsuarios * numdePosts * numdeComentarios * numdeRespuestas;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu4', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu4', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu4', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv4 = await obtenerDatosCuartoNivel(urlBase, numdeUsuarios, numdePosts, numdeComen
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv4.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv4.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 87 Código del Caso de uso 4 - REST

```
export async function CasodeUso5R(urlBase, numdeUsuarios, numdePosts, numdeComentarios, numdeRespuestas,
    numdeReacciones, arquiterctura, tecnologia) {
    let datosObtenidosLv5;
    let distribucion = numdeUsuarios + 'x' + numdePosts + 'x' + numdeComentarios + 'x' + numdeRespuestas + '
    let numdeRegistros = numdeUsuarios * numdePosts * numdeComentarios * numdeRespuestas * numdeReacciones;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu5', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu5', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu5', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv5 = await obtenerDatosQuintoNivel(urlBase, numdeUsuarios, numdePosts, numdeComen
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv5.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv5.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 88 Código del Caso de uso 5 - REST

2.5.3 Experimento computacional GraphQL

Las consultas de GraphQL son diferentes a las de REST ya que en lugar de enviar una URL que devuelve los datos, en este caso se envía una Query completa, por lo que la forma de consumir las tres APIs de esta arquitectura es diferente en la forma de armar la consulta, pero tras la obtención de los datos el resto del método se asimila mucho a los vistos anteriormente, a continuación se muestra el método para obtener los datos del primer nivel:

```
export const obtenerDatosPrimerNivel = async(urlBase, numDatosPrimerNivel) => {
    try {
        const query = gql`
        query GetUsuariosCantidad($numUsr: Int!) {
            getUsuariosCantidad(numUsr: $numUsr) {
                id_usuario
                nombre
                correo
                clave
            }
        }
        `;

        const datosConsulta = { numUsr: numDatosPrimerNivel };
        const tInicial = new Date();
        let usuariosConsultados = await request(urlBase, query, datosConsulta);
        const tFinal = new Date() - tInicial;
        usuariosConsultados = usuariosConsultados.getUsuariosCantidad;

        const cantidadDatosUsuarios = usuariosConsultados.length + "";
        console.log('G Usrs =====>', usuariosConsultados.length)

        return {
            datos: usuariosConsultados,
            tiempo: tFinal,
            cadenaDatos: cantidadDatosUsuarios
        };
    } catch (error) {
        console.error('Error:', error);
    }
}
```

Figura 89 Método para obtener datos del primer nivel - GraphQL

Primero el método recibe la URL base vista anteriormente y también la cantidad de usuarios que se desea consultar, después de esto y dentro de la sección de manejo de excepciones correspondiente se procede a armar la query, y guardar en una variable los datos que se usarán en la consulta, tras esto es necesario iniciar la medición del tiempo puesto que lo siguiente es realizar la consulta, con los datos obtenidos y el tiempo de respuesta guardado ya se puede hacer lo que se hizo en los métodos anteriores, que es imprimir la cantidad de datos consultados y finalmente devolver los datos consultados, junto con el tiempo de respuesta y la cantidad de datos consultados.

Como se ve en la imagen, en primer lugar el método recibe la URL base, ya descrita previamente; así como la cantidad de usuarios que se desea consultar, a continuación, y dentro del bloque de manejo de excepciones correspondiente, se construye la query y se almacenan en una variable los datos necesarios para la consulta, tras esto se inicia la medición del tiempo de ejecución, ya que lo siguiente consiste en realizar la consulta a la API. Con los datos obtenidos y el tiempo de respuesta registrado, se procede a imprimir la cantidad de datos consultados y finalmente a devolver la información recopilada junto con el tiempo de respuesta y el conteo de elementos consultados.

A continuación, se presenta el método correspondiente al segundo nivel, el cual sigue un enfoque anidado: antes de consultar los datos de este nivel, se llama al método encargado de obtener la información del primer nivel, como se mostrará a continuación:

```
export const obtenerDatosSegundoNivel = async(urlBase, numDatosPrimerNivel, numDatosSegundoNivel) => {
    let publicacionesConsultadas = [];

    try {
        const query = gql`
            query GetPublicacionesCantidad($idUsr: Int!, $numPst: Int!) {
                getPublicacionesCantidad(idUsr: $idUsr, numPst: $numPst) {
                    id_publicacion
                    id_usuario
                    titulo
                    contenido
                    fecha_publicacion
                }
            }
        `;

        const tInicial = new Date();
        const { datos: usuarios, cadenaDatos: cadenaUsrs } = await obtenerDatosPrimerNivel(urlBase, numDatos
        const idsUsuarios = usuarios.map((res) => res.id_usuario);

        for (const id of idsUsuarios) {
            const datosConsulta = { idUsr: id,  numPst: numDatosSegundoNivel};
            let respuesta = await request(urlBase, query, datosConsulta);
            respuesta = respuesta.getPublicacionesCantidad;
            publicacionesConsultadas = [...publicacionesConsultadas, ...respuesta];
        }

        const tFinal = new Date() - tInicial;

        const cadenaPubls = cadenaUsrs + " - " + publicacionesConsultadas.length;
```

Figura 90 Método para obtener datos del segundo nivel - GraphQL

```
        console.log('G Publs =====>', publicacionesConsultadas.length)

        return {
            datos: publicacionesConsultadas,
            tiempo: tFinal,
            cadenaDatos: cadenaPubls
        };
    } catch (error) {
        console.error('Error:', error);
    }
}
```

Figura 91 Método para obtener datos del segundo nivel - GraphQL

El método comienza nuevamente recibiendo los parámetros, tanto del primer como del segundo nivel, ya que aunque primero se arma nuevamente la query , en la consulta de datos primer se llama al que permite consultar los de primer nivel, una vez obtenido tanto el resultado de la consulta como el número de usuarios devueltos; es hasta esta parte que apenas se puede realizar la consulta de los datos del segundo nivel, obteniendo una consulta por cada Id de usuario solicitado, tras esto se termina el cálculo del tiempo de respuesta y se devuelven los mismos datos que en el primer caso.

El resto de los métodos para obtener datos del tercer al quinto nivel, son sumamente similares al del segundo nivel, por lo que solamente se mostrará la primera parte que es la query, puesto que esta es la única que varía en todos los métodos, como se muestra a continuación:

```
export const obtenerDatosTercerNivel = async(urlBase, numDatosPrimerNivel, numDatosSegundoNivel, numDatosTe
    let comentariosConsultados = [];

    try {
        const query = gql`
            query GetComentariosCantidad($idPub: Int!, $numCmt: Int!) {
                getComentariosCantidad(idPub: $idPub, numCmt: $numCmt) {
                    id_comentario
                    id_publicacion
                    texto
                    fecha_comentario
                }
            }
        `;

        const tInicial = new Date();
        const { datos: publicaciones, cadenaDatos: cadenaPubls } = await obtenerDatosSegundoNivel(urlBase,
        const idsPublicaciones = publicaciones.map((res) => res.id_publicacion);

        for (const id of idsPublicaciones) {
            const datosConsulta = { idPub: id,  numCmt: numDatosTercerNivel};
            let respuesta = await request(urlBase, query, datosConsulta);
            respuesta = respuesta.getComentariosCantidad;
            comentariosConsultados = [...comentariosConsultados, ...respuesta];
        }

        const tFinal = new Date() - tInicial;

        const cadenaCmts = cadenaPubls + " - " + comentariosConsultados.length;
```

Figura 92 Método para obtener datos del tercer nivel - GraphQL

```
export const obtenerDatosCuartoNivel = async(urlBase, numDatosPrimerNivel, numDatosSegundoNivel, numDat
    let respuestasConsultadas = [];

    try {
        const query = gql`
            query GetRespuestasCantidad($idCmt: Int!, $numRes: Int!) {
                getRespuestasCantidad(idCmt: $idCmt, numRes: $numRes) {
                    id_respuesta
                    id_comentario
                    texto
                    fecha_respuesta
                }
            }
        `;

        const tInicial = new Date();
        const { datos: comentarios, cadenaDatos: cadenaCmts } = await obtenerDatosTercerNivel(urlBase, 
        const idsComentarios = comentarios.map((res) => res.id_comentario);

        for (const id of idsComentarios) {
            const datosConsulta = { idCmt: id,  numRes: numDatosCuartoNivel};
            let respuesta = await request(urlBase, query, datosConsulta);
            respuesta = respuesta.getRespuestasCantidad;
            respuestasConsultadas = [...respuestasConsultadas, ...respuesta];
        }

        const tFinal = new Date() - tInicial;

        const cadenaResps = cadenaCmts + " - " + respuestasConsultadas.length;
```

Figura 93 Método para obtener datos del cuarto nivel - GraphQL

```
export const obtenerDatosQuintoNivel = async(urlBase, numDatosPrimerNivel, numDatosSegundoNivel, numDatosTer
    let reaccionesConsultadas = [];

    try {
        const query = gql`
            query GetReaccionesCantidad($idRes: Int!, $numReac: Int!) {
                getReaccionesCantidad(idRes: $idRes, numReac: $numReac) {
                    id_reaccion
                    id_respuesta
                    id_tipo_reaccion
                    fecha_reaccion
                }
            }
        `;

        const tInicial = new Date();
        const { datos: respuestas, cadenaDatos: cadenaResps } = await obtenerDatosCuartoNivel(urlBase, numDa
        const idsRespuestas = respuestas.map((res) => res.id_respuesta);

        for (con
            cons let respuestaConsulta: any d,  numReac: numDatosQuintoNivel};
            let respuestaConsulta = await request(urlBase, query, datosConsulta);
            respuestaConsulta = respuestaConsulta.getReaccionesCantidad;
            reaccionesConsultadas = [...reaccionesConsultadas, ...respuestaConsulta];
        }

        const tFinal = new Date() - tInicial;

        const cadenaReacs = cadenaResps + " - " + reaccionesConsultadas.length;
        console.log('G Reacs ========>', reaccionesConsultadas.length)
```

Figura 94 Método para obtener datos del quinto nivel - GraphQL

Como se puede observar, la estructura general de los métodos es similar entre estos, he incluso es similar a la forma de consultar las APIs REST. Con esto queda claro cómo es que se obtienen los datos de cada API GraphQL.

2.5.4 Casos de uso GraphQL

Los casos de uso de esta arquitectura al funcionar solamente como intermediarios para realizar las actividades de consulta repetida y organización para el guardado de datos; sus métodos son prácticamente idénticos a los casos de uso realizados para REST, por lo que no se explicará a detalle su código, solamente se tratarán con brevedad a continuación:

```
export async function CasodeUso1G(urlBase, numdeUsuarios, arquiterctura, tecnologia) {
    let datosObtenidosLv1;
    let arregloResultado = new Array(3);
    let distribucion = '' + numdeUsuarios;
    arregloResultado[0] = ['Cu1', arquiterctura, tecnologia, numdeUsuarios, distribucion];
    arregloResultado[1] = ['Cu1', arquiterctura, tecnologia, numdeUsuarios, distribucion];
    arregloResultado[2] = ['Cu1', arquiterctura, tecnologia, numdeUsuarios, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv1 = await obtenerDatosPrimerNivel(urlBase, numdeUsuarios);
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv1.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv1.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 95 Código del Caso de uso 1 - GraphQL

```
export async function CasodeUso2G(urlBase, numdeUsuarios, numdePosts, arquiterctura, tecnologia) {
    let datosObtenidosLv2;
    let numdeRegistros = numdeUsuarios * numdePosts;
    let distribucion = numdeUsuarios + 'x' + numdePosts;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu2', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu2', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu2', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv2 = await obtenerDatosSegundoNivel(urlBase, numdeUsuarios, numdePosts);
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv2.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv2.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 96 Código del Caso de uso 2 - GraphQL

```
export async function CasodeUso3G(urlBase, numdeUsuarios, numdePosts, numdeComentarios, arquiterctura,
    tecnologia) {
    let datosObtenidosLv3;
    let distribucion = numdeUsuarios + 'x' + numdePosts + 'x' + numdeComentarios;
    let numdeRegistros = numdeUsuarios * numdePosts * numdeComentarios;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu3', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu3', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu3', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv3 = await obtenerDatosTercerNivel(urlBase, numdeUsuarios, numdePosts, numdeCome
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv3.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv3.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 97 Código del Caso de uso 3 - GraphQL

```
export async function CasodeUso4G(urlBase, numdeUsuarios, numdePosts, numdeComentarios, numdeRespuestas,
    arquiterctura, tecnologia) {
    let datosObtenidosLv4;
    let distribucion = numdeUsuarios + 'x' + numdePosts + 'x' + numdeComentarios + 'x' + numdeRespuestas;
    let numdeRegistros = numdeUsuarios * numdePosts * numdeComentarios * numdeRespuestas;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu4', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu4', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu4', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv4 = await obtenerDatosCuartoNivel(urlBase, numdeUsuarios, numdePosts, numdeCome
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv4.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv4.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 98 Código del Caso de uso 4 - GraphQL

```
export async function CasodeUso5G(urlBase, numdeUsuarios, numdePosts, numdeComentarios, numdeRespuestas,
    numdeReacciones, arquiterctura, tecnologia) {
    let datosObtenidosLv5;
    let distribucion = numdeUsuarios + 'x' + numdePosts + 'x' + numdeComentarios + 'x' + numdeRespuestas +
    let numdeRegistros = numdeUsuarios * numdePosts * numdeComentarios * numdeRespuestas * numdeReacciones;
    let arregloResultado = new Array(3);
    arregloResultado[0] = ['Cu5', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[1] = ['Cu5', arquiterctura, tecnologia, numdeRegistros, distribucion];
    arregloResultado[2] = ['Cu5', arquiterctura, tecnologia, numdeRegistros, distribucion];
    try {
        for(let i = 0; i < 3; i++){
            datosObtenidosLv5 = await obtenerDatosQuintoNivel(urlBase, numdeUsuarios, numdePosts, numdeComer
            arregloResultado[i] = arregloResultado[i].concat(i + 1);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv5.tiempo);
            arregloResultado[i] = arregloResultado[i].concat(datosObtenidosLv5.cadenaDatos);
        }
        return arregloResultado;
    } catch (error) {
        console.log('Error'. error);
    }
};
```

Figura 99 Código del Caso de uso 5 - GraphQL

Fuera del nombre del método que sirve para identificar entre los casos de uso REST y los GraphQL, el resto del código es sumamente similar al de los casos de uso REST, por lo que aparte de mostrar el código, es irrelevante explicarlo a detalle ya que esto se hizo anteriormente.

2.5.5 Consulta y Guardado de datos

Con toda las estructura de consulta terminada ya es posible utilizar estos métodos para obtener los datos de la API, para ello únicamente es necesario invocar el método del caso de uso necesario y enviar los datos que este solicita, tras lo cual también se debe recibir el resultado en una variable que posteriormente se puede mostrar o enviar a guardarse en el documento. El código para consultar los datos es el siguiente:

```
43    console.log('---------------------------------------------');
44    const urlEnUso = urlPyRst;
45    const tecnologiaEnUso = 'Python y Flask';
46
47    let pruebaFuncionamiento = await CasodeUso5R(urlEnUso, 1, 1, 1, 1, 1, 'REST', tecnologiaEnUso)
48    console.log(pruebaFuncionamiento)
49
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   PORTS   TERMINAL   AZURE

---------------------------------------------
R Usrs =====> 1
R Publs =====> 1
R Cmts =======> 1
R Resps =======> 1
R Reacs ========> 1
R Usrs =====> 1
R Publs =====> 1
R Cmts =======> 1
R Resps =======> 1
R Reacs ========> 1
R Usrs =====> 1
R Publs =====> 1
R Cmts =======> 1
R Resps =======> 1
R Reacs ========> 1
```

Figura 100 Código para consultar los datos

Adicionalmente al llamar el caso de uso se muestra en la terminal los datos de cada nivel que fueron consultados, y esto se ve tres veces porque como está planificado una sola ejecución imprime las tres veces que se debe ejecutar cada consulta. En el código también se imprimen los datos por consola, estos datos son los que devuelve el método y están organizados para ser guardados posteriormente, en la consola el mensaje se ve de la siguiente forma:

```
[
  'Cu5',
  'REST',
  'Python y Flask',
  1,
  '1x1x1x1x1',
  1,
  500,
  '1 - 1 - 1 - 1 - 1'
],
[
  'Cu5',
  'REST',
  'Python y Flask',
  1,
  '1x1x1x1x1',
  2,
  430,
  '1 - 1 - 1 - 1 - 1'
],
[
  'Cu5',
  'REST',
  'Python y Flask',
  1,
  '1x1x1x1x1',
  3,
  510,
  '1 - 1 - 1 - 1 - 1'
]
]
---- Ejecución Completa ----
```

Figura 101 Datos devueltos por el método he impresos por consola

Por otro lado, el método de guardado resulta bastante sencillo gracias a las funcionalidades que ofrece la librería XlsxPopulate, la cual simplifica el trabajo con hojas de cálculo. De manera general el código lee el archivo de Excel establecido, selecciona la hoja de trabajo principal e inserta en ella el arreglo bidimensional previamente generado, donde cada subarreglo representa una fila y cada elemento de este una columna. Finalmente el archivo es guardado en la ruta indicada, quedando disponible con todos los datos consultados y listos para su análisis posterior.

Todo esto se realiza con el siguiente código:

```javascript
async function guardarDatos(datosParaGuardar){
    try {
        const documento = await XlsxPopulate.fromFileAsync('./src/Matriz_TiemposRespuesta.xlsx');
        let hojaExcel;
        let celdaRegistro;

        if(datosParaGuardar[1][1] == 'REST'){
            hojaExcel = 'REST';
            celdaRegistro = 'B1';

        }else if(datosParaGuardar[1][1] == 'GraphQL'){
            hojaExcel = 'GraphQL';
            celdaRegistro = 'B2';

        }else{
            throw new Error('¡¡Arquitectura no válida, por favor verifique ese parámetro!!');
        }

        let registro = documento.sheet('NumRF').cell(celdaRegistro).value();
        let celdaVacia = 'A' + registro;
        documento.sheet(hojaExcel).cell(celdaVacia).value(datosParaGuardar);
        registro = registro + 3;
        documento.sheet('NumRF').cell(celdaRegistro).value(registro);
        await documento.toFileAsync('./src/Matriz_TiemposRespuesta.xlsx');
        console.log('Registros', hojaExcel ,'guardados exitosamente!!')

    } catch (error) {
        console.log('Error', error);
    }
};
```

Figura 102 Método del guardado de datos en Excel

Al terminar el método guarda los datos en el documento de Excel, los datos guardados quedan de la siguiente forma:

ID Caso de Uso	Arquitectura	Tecnología	Número de Registros	Distribución	Número de Iteración	Tiempo (ms)	Registro de Iteraciones
Cu1	REST	Python y Flask	1	1	1	80	1
Cu1	REST	Python y Flask	1	1	2	66	1
Cu1	REST	Python y Flask	1	1	3	66	1

Figura 103 Datos guardados en Excel

Con esto ya están listas todas las partes necesarias para realizar la ejecución del experimento computacional, por lo que resta comprobar el funcionamiento de las APIs, que es lo que se realizará a continuación.

2.5.6 Pruebas de funcionamiento de las APIs

Dado que como se explicó anteriormente los métodos de consulta son anidados, en lugar de probar cada caso de uso por individual, la mejor alternativa es solamente probar el caso de uso cinco, ya que con esto se probarán todos los anteriores casos, dicho esto se muestra la prueba de funcionamiento del caso de uso cinco para cada una de las seis APIs, con la distribución 1x1x1x1x1. A continuación se muestra la ejecución de todas las APIs:

Tabla 8 Pruebas de funcionamiento de las APIs

Lenguaje y Framework	Arquitectura	Ejecución

JavaScript y Express	REST	43 console.log('---'); 44 const urlEnUso = urlJSRst; 45 const tecnologiaEnUso = 'JavaScript y Express'; 46 47 await CasodeUso5R(urlEnUso, 1, 1, 1, 1, 1, 'REST', tecnologiaEnUso) 48 49 console.log('---- Ejecución Completa ----') PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL AZURE --- R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 ---- Ejecución Completa ----
Python y Flask	REST	43 console.log('---'); 44 const urlEnUso = urlPyRst; 45 const tecnologiaEnUso = 'Python y Flask'; 46 47 await CasodeUso5R(urlEnUso, 1, 1, 1, 1, 1, 'REST', tecnologiaEnUso) 48 49 console.log('---- Ejecución Completa ----') PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL AZURE --- R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 ---- Ejecución Completa ----
Java y SpringBoot	REST	43 console.log('---'); 44 const urlEnUso = urlJvRst; 45 const tecnologiaEnUso = 'Java y Spring'; 46 47 await CasodeUso5R(urlEnUso, 1, 1, 1, 1, 1, 'REST', tecnologiaEnUso) 48 49 console.log('---- Ejecución Completa ----') PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL AZURE --- R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 R Usrs =====> 1 R Publs =====> 1 R Cmts =======> 1 R Resps =======> 1 R Reacs ========> 1 ---- Ejecución Completa ----

JavaScript y Express	GraphQL	43 console.log('--'); 44 const urlEnUso = urlJSGQL; 45 const tecnologiaEnUso = 'JavaScript y Express'; 46 47 await CasodeUso5G(urlEnUso, 1, 1, 1, 1, 1, 'GraphQL', tecnologiaEnUso) 48 49 console.log('---- Ejecución Completa ----') PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL AZURE -- G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 ---- Ejecución Completa ----
Python y Flask	GraphQL	43 console.log('--'); 44 const urlEnUso = urlPyGQL; 45 const tecnologiaEnUso = 'Python y Flask'; 46 47 await CasodeUso5G(urlEnUso, 1, 1, 1, 1, 1, 'GraphQL', tecnologiaEnUso) 48 49 console.log('---- Ejecución Completa ----') PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL AZURE -- G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 ---- Ejecución Completa ----
Go y Gin	GraphQL	43 console.log('--'); 44 const urlEnUso = urlGoGQL; 45 const tecnologiaEnUso = 'Go y Gin'; 46 47 await CasodeUso5G(urlEnUso, 1, 1, 1, 1, 1, 'GraphQL', tecnologiaEnUso) 48 49 console.log('---- Ejecución Completa ----') PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL AZURE -- G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 G Usrs =====> 1 G Publs =====> 1 G Cmts =======> 1 G Resps =======> 1 G Reacs ========> 1 ---- Ejecución Completa ----

Se puede comprobar que se ejecuta cada consulta sin problemas, por lo que lo siguiente por hacer es ejecutar todos los casos de uso para cada API y con ello recolectar los datos a analizarse.

2.6 Ejecución del Experimento Computacional

Con el cliente ya finalizado el paso restante es ejecutar todos los casos de uso para cada una de las seis APIs, esto se puede realizar de la siguiente manera:

```
console.log('-------------------------------------------');
const urlEnUso = urlGoGQL;
const tecnologiaEnUso = 'Go y Gin';

console.log('########## CASO DE USO 1 ##########')
await guardarDatos(await CasodeUso1G(urlEnUso, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso1G(urlEnUso, 10, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso1G(urlEnUso, 100, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso1G(urlEnUso, 1000, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso1G(urlEnUso, 10000, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso1G(urlEnUso, 100000, 'GraphQL', tecnologiaEnUso));

console.log('########## CASO DE USO 2 ##########')
await guardarDatos(await CasodeUso2G(urlEnUso, 1, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso2G(urlEnUso, 3, 3, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso2G(urlEnUso, 10, 10, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso2G(urlEnUso, 32, 31, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso2G(urlEnUso, 100, 100, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso2G(urlEnUso, 316, 316, 'GraphQL', tecnologiaEnUso));

console.log('########## CASO DE USO 3 ##########')
await guardarDatos(await CasodeUso3G(urlEnUso, 1, 1, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso3G(urlEnUso, 2, 2, 2, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso3G(urlEnUso, 5, 5, 4, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso3G(urlEnUso, 10, 10, 10, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso3G(urlEnUso, 22, 22, 21, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso3G(urlEnUso, 46, 46, 47, 'GraphQL', tecnologiaEnUso));
```

Figura 104 Código para ejecutar todos los casos de uso

```
console.log('########## CASO DE USO 4 ##########')
await guardarDatos(await CasodeUso4G(urlEnUso, 1, 1, 1, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso4G(urlEnUso, 2, 2, 2, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso4G(urlEnUso, 3, 3, 3, 4, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso4G(urlEnUso, 6, 6, 6, 5, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso4G(urlEnUso, 10, 10, 10, 10, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso4G(urlEnUso, 18, 18, 18, 17, 'GraphQL', tecnologiaEnUso));

console.log('########## CASO DE USO 5 ##########')
await guardarDatos(await CasodeUso5G(urlEnUso, 1, 1, 1, 1, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso5G(urlEnUso, 2, 2, 1, 2, 1, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso5G(urlEnUso, 3, 3, 2, 3, 2, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso5G(urlEnUso, 4, 4, 4, 4, 4, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso5G(urlEnUso, 6, 6, 7, 6, 7, 'GraphQL', tecnologiaEnUso));
await guardarDatos(await CasodeUso5G(urlEnUso, 10, 10, 10, 10, 10, 'GraphQL', tecnologiaEnUso));

console.log('---- Ejecución Completa ----')
```

Figura 105 Código para ejecutar todos los casos de uso

El procedimiento para obtener los datos de los cinco casos de uso, junto con todas las distribuciones de estos en una sola ejecución es sumamente sencillo ya que solamente basta con llamar al caso de uso correspondiente dentro del método para guardar los datos, asegurando así que la información se registre en el documento de Excel tras obtenerse. Esta ejecución se deberá repetir

para las seis APIs variando únicamente la URL, Arquitectura y Tecnologías en uso, sin olvidar también cambiar el método dependiendo de si el caso de uso es REST o GraphQL.

Tras la ejecución de lo mencionado, el documento de Excel tendrá los datos guardados de la siguiente forma:

ID Caso de Uso	Arquitectura	Tecnología	Número de Registros	Distribución	Número de Iteración	Tiempo (ms)	Registro de Iteraciones
Cu1	GraphQL	Go y Gin	1	1	1	36	1
Cu1	GraphQL	Go y Gin	1	1	2	10	1
Cu1	GraphQL	Go y Gin	1	1	3	5	1
Cu1	GraphQL	Go y Gin	10	10	1	12	10
Cu1	GraphQL	Go y Gin	10	10	2	11	10
Cu1	GraphQL	Go y Gin	10	10	3	6	10
Cu1	GraphQL	Go y Gin	100	100	1	8	100
Cu1	GraphQL	Go y Gin	100	100	2	11	100
Cu1	GraphQL	Go y Gin	100	100	3	7	100
Cu1	GraphQL	Go y Gin	1000	1000	1	73	1000
Cu1	GraphQL	Go y Gin	1000	1000	2	81	1000
Cu1	GraphQL	Go y Gin	1000	1000	3	65	1000
Cu1	GraphQL	Go y Gin	10000	10000	1	118	10000
Cu1	GraphQL	Go y Gin	10000	10000	2	116	10000
Cu1	GraphQL	Go y Gin	10000	10000	3	90	10000
Cu1	GraphQL	Go y Gin	100000	100000	1	1060	100000

Figura 106 Resultado en Excel de la ejecución de los casos de uso

Cu5	REST	Java y Srping	1	1x1x1x1x1	1	60	1 - 1 - 1 - 1 - 1
Cu5	REST	Java y Srping	1	1x1x1x1x1	2	21	1 - 1 - 1 - 1 - 1
Cu5	REST	Java y Srping	1	1x1x1x1x1	3	20	1 - 1 - 1 - 1 - 1
Cu5	REST	Java y Srping	8	2x2x1x2x1	1	61	2 - 4 - 4 - 8 - 8
Cu5	REST	Java y Srping	8	2x2x1x2x1	2	58	2 - 4 - 4 - 8 - 8
Cu5	REST	Java y Srping	8	2x2x1x2x1	3	49	2 - 4 - 4 - 8 - 8
Cu5	REST	Java y Srping	108	3x3x2x3x3	1	194	3 - 9 - 18 - 54 - 108
Cu5	REST	Java y Srping	108	3x3x2x3x3	2	196	3 - 9 - 18 - 54 - 108
Cu5	REST	Java y Srping	108	3x3x2x3x3	3	222	3 - 9 - 18 - 54 - 108
Cu5	REST	Java y Srping	1.024	4x4x4x4x4	1	861	4 - 16 - 64 - 256 - 1024
Cu5	REST	Java y Srping	1.024	4x4x4x4x4	2	808	4 - 16 - 64 - 256 - 1024
Cu5	REST	Java y Srping	1.024	4x4x4x4x4	3	875	4 - 16 - 64 - 256 - 1024
Cu5	REST	Java y Srping	10.584	6x6x7x6x6	1	4.263	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Java y Srping	10.584	6x6x7x6x6	2	3.915	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Java y Srping	10.584	6x6x7x6x6	3	4.175	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Java y Srping	100.000	10x10x10x10x10	1	30.672	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	Java y Srping	100.000	10x10x10x10x10	2	29.899	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	Java y Srping	100.000	10x10x10x10x10	3	30.438	10 - 100 - 1000 - 10000 - 100000

Figura 107 Resultado en Excel de la ejecución de los casos de uso

En la imagen se ver el resultado de la ejecución de los casos de uno 1 y 5, el primero con la arquitectura GraphQL y el segundo con REST, por tanto el resultado se repetirá para obtener el resto de los datos en cada API, que posteriormente deben ser analizados como se muestra en el siguiente capítulo.

III. Capítulo 3: Análisis de los resultados

3.1 Datos recolectados de las seis APIs

Tras la ejecución de los seis casos de uso bajo condiciones controladas (con el equipo conectado a la corriente eléctrica y únicamente con la API y el cliente en funcionamiento), se procede al análisis de los resultados obtenidos a partir de los tiempos de respuesta registrados, estos resultados se presentan en forma de gráficos construidos con los datos recopilados, mientras que la totalidad de los valores se adjunta en el apartado de anexos para su consulta detallada.

El proceso de análisis se organiza de la siguiente manera: en primer lugar, se comparan las tres APIs de la arquitectura REST caso de uso por caso de uso, con el fin de identificar cuál de ellas presenta un mejor desempeño general. Posteriormente, se repite la misma dinámica con las tres APIs

de la arquitectura GraphQL y finalmente las APIs que mostraron el mejor rendimiento dentro de cada arquitectura se enfrentan nuevamente entre sí, también caso de uso por caso de uso, para determinar cuál de las dos arquitecturas resulta más eficiente en el contexto del experimento realizado.

3.2 Comparativa de las APIs REST

Ahora que se cuenta con el desempeño individual de cada API, se posible proceder con la comparación entre estas, esto como se indica en la metodología se realizará tomando en cuenta el tiempo de ejecución promedio por lo que las tablas mostradas a continuación mostrarán únicamente el promedio de las tres iteraciones para cada distribución por caso de uso. Con esto claro a continuación se procederá a analizar las tablas de resultado.

3.2.1 Comparación para el Caso de Uso 1

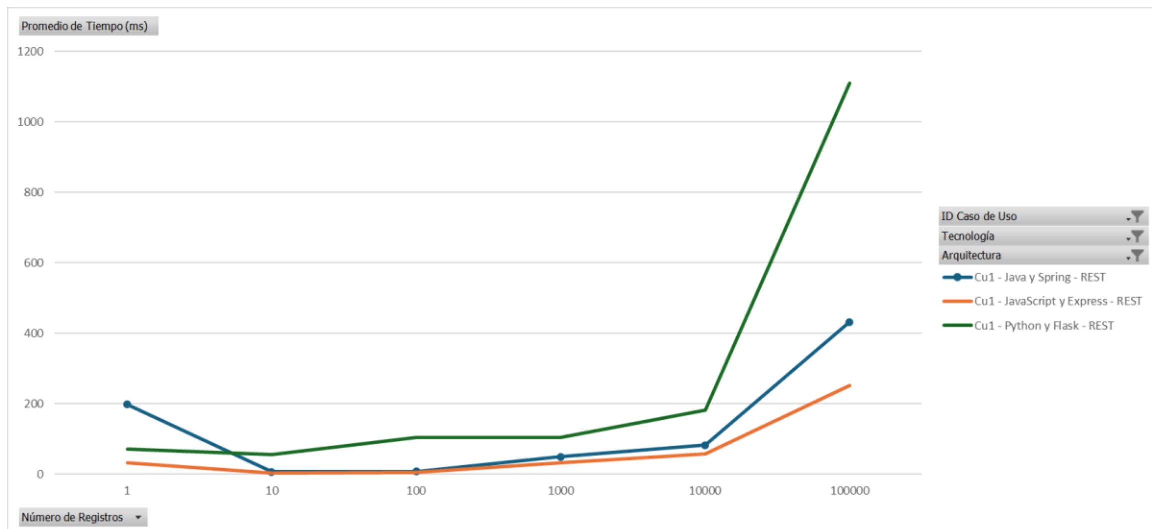

Figura 108 Tiempo de respuesta de las APIs REST en el Caso de uso 1

A simple vista se puede notar que la API desarrollada en Python con Flask presenta un mayor tiempo de respuesta en comparación con las otras dos alternativas, si bien la diferencia es poco significativa en los primeros volúmenes de datos, esta se incrementa conforme aumenta la cantidad de registros consultados. También se observa que en el primer volumen de datos, la API de Java muestra cierta dificultad inicial para alcanzar un rendimiento óptimo, aunque posteriormente compensa con buenos tiempos en volúmenes de datos medianos y grandes.

Al analizar con mayor detalle la última distribución (donde la diferencia es más notoria) y convirtiendo los resultados de milisegundos a segundos, se obtiene lo siguiente: la API de Java tarda aproximadamente 0,252 segundos, la API de JavaScript alrededor de 0,432 segundos, mientras que la API de Python requiere cerca de 1,109 segundos para completar la consulta, este último valor la convierte en la única que supera el segundo de ejecución en el primer caso de uso, lo que la posiciona como la principal candidata a ser descartada. Sin embargo, antes de llegar a una conclusión definitiva, es necesario verificar si este patrón de menor rendimiento se mantiene en el segundo caso de uso.

3.2.2 Comparación para el Caso de Uso 2

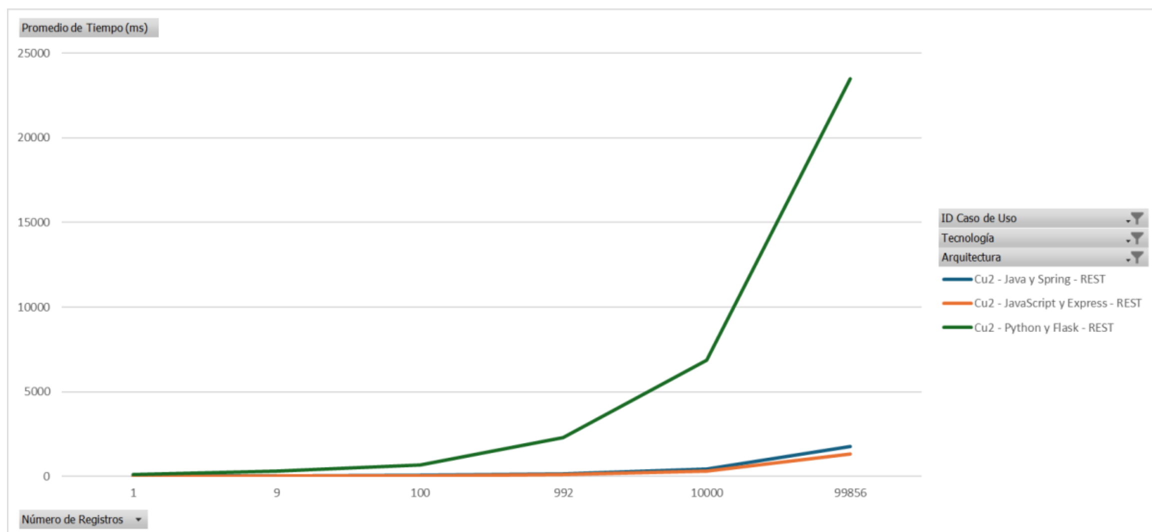

Figura 109 Tiempo de respuesta de las APIs REST en el Caso de uso 2

En un primer vistazo se confirma que la API de Python vuelve a mostrar el menor rendimiento, incluso con un desempeño más desfavorable que en el primer caso de uso lo puede atribuirse a que en este escenario se consultan dos niveles de la base de datos en lugar de uno, lo que incrementa la complejidad del proceso.

Entre las tecnologías con mejores resultados, la API de Java se posiciona como la segunda mejor opción, mientras que JavaScript lidera la comparación, aunque la diferencia respecto a Java aún no es demasiado amplia.

Al analizar con mayor detalle los tiempos registrados en la tabla, se observa que Python presenta un rendimiento aproximadamente 17 veces inferior en consultas con grandes volúmenes de datos. Esto se refleja claramente en su tiempo de ejecución: alrededor de 2306 milisegundos (2,306 segundos), una cifra considerablemente más alta si la comparamos con los 1,759 segundos de Java o los 1,348 segundos de JavaScript.

3.2.3 Comparación para el Caso de Uso 3

Antes de descartar definitivamente a la API de Python con Flask de la comparación, se muestra su desempeño en el tercer caso de uso:

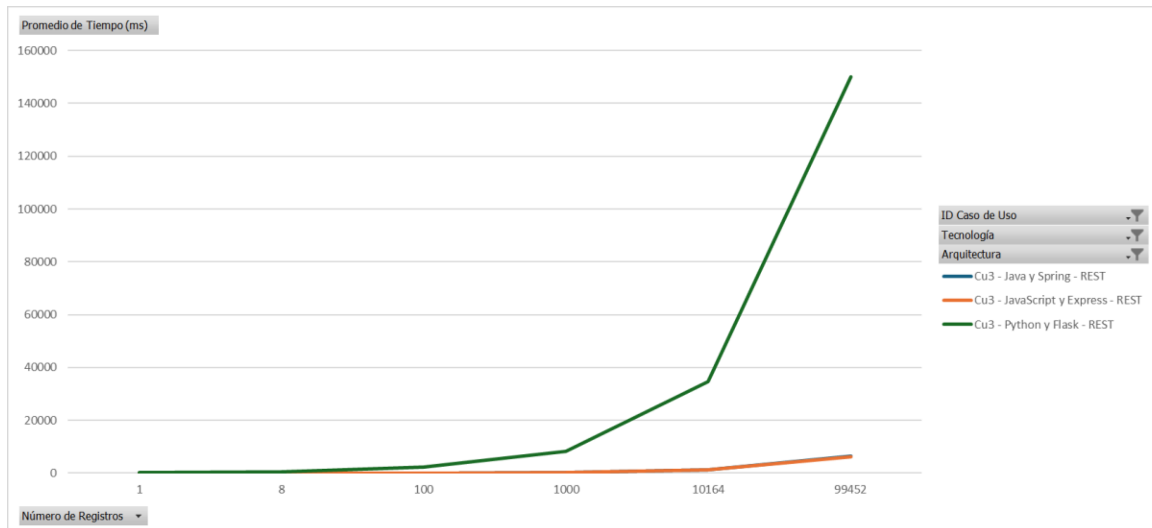

Figura 110 Tiempo de respuesta de las APIs REST en el Caso de uso 3

Como es posible notar, la API de Python con Flask continúa mostrando un rendimiento deficiente en comparación con las APIs de Java y JavaScript, en este caso resulta aproximadamente 23 veces menos eficiente que JavaScript, ya que requiere alrededor de 149 976 milisegundos (≈149,9 segundos) para completar la consulta de 100 000 registros, este tiempo elevado se explica también porque la consulta involucra tres de los cinco niveles de la base de datos.

En contraste, tanto la API de Java como la de JavaScript logran ejecutar la misma tarea en aproximadamente 6 segundos, evidenciando una diferencia sustancial.

Debido a este bajo rendimiento, la API de Python con Flask dejará de ser considerada en los dos casos de uso siguientes.

3.2.4 Comparación para el Caso de Uso 4

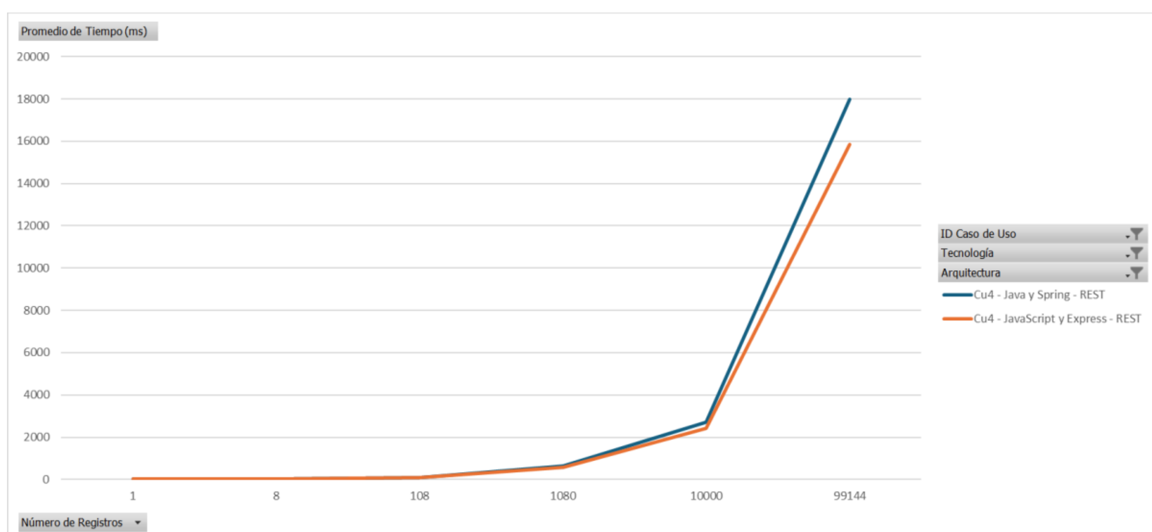

Figura 111 Tiempo de respuesta de las APIs REST en el Caso de uso 4

Con la eliminación de la API de Python, la diferencia en el rendimiento entre Java y JavaScript es más notoria, pero aun así en lo que respecta a pequeños volúmenes de datos, están igualadas.

A partir de los datos representados en el gráfico, es posible deducir que en la distribución final de datos, es decir, en la consulta con el mayor volumen de registros, la API de JavaScript con Express presenta un rendimiento aproximadamente 12% superior al de la API de Java con Spring. Este resultado se evidencia en el tiempo de ejecución, ya que la API de JavaScript procesa 99 144 registros en alrededor de 15 857 milisegundos, mientras que la API de Java requiere aproximadamente 17 988 milisegundos para completar la misma consulta. Si bien esta diferencia no alcanza la magnitud observada en la API de Python, confirma la ventaja sostenida de JavaScript en escenarios de alta carga de datos.

3.2.5 Comparación para el Caso de Uso 5

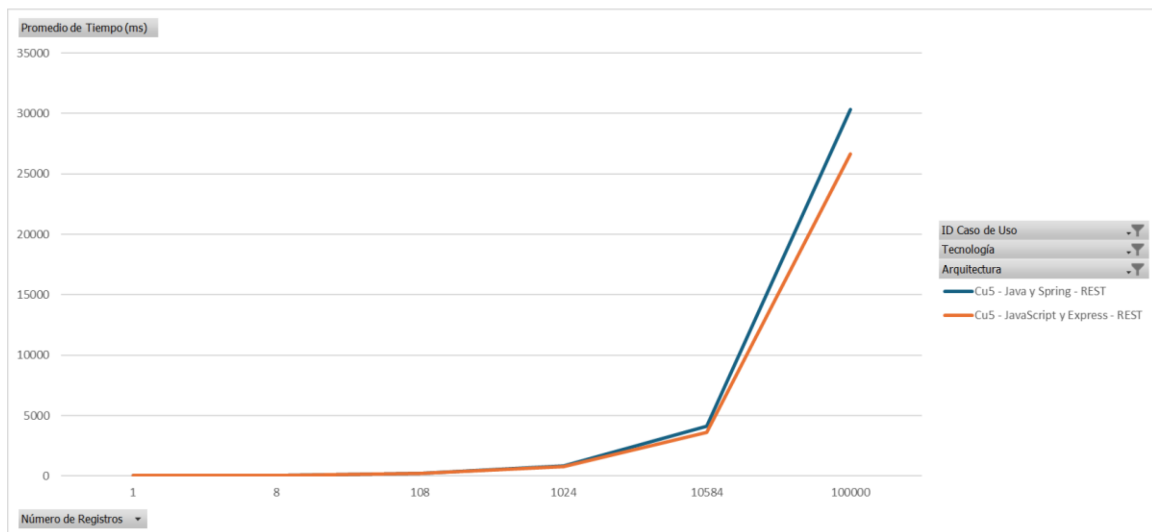

Figura 112 Tiempo de respuesta de las APIs REST en el Caso de uso 5

Nuevamente se observa que la diferencia de rendimiento entre las APIs es mínima en volúmenes reducidos de datos; sin embargo, esta se vuelve más evidente conforme aumenta la cantidad de registros procesados, a partir de la visualización de los gráficos es posible identificar a la API de JavaScript con Express como la de mejor desempeño, aunque para corroborar esta conclusión resulta necesario analizar los tiempos de ejecución.

El caso de uso 5 constituye el de mayor complejidad, ya que implica la extracción de información en cada uno de los cinco niveles de la base de datos, en este escenario al procesar 100 000 registros, la API de JavaScript completa la tarea en aproximadamente 26,684 segundos, mientras que la API de Java lo hace en 30,336 segundos. Esta diferencia de 3,652 segundos implica que Java con Spring resulta aproximadamente 13,7% más lento que JavaScript en este volumen de datos.

En consecuencia, se determina que la tecnología más eficiente para el desarrollo de APIs REST es JavaScript con Express, al demostrar un rendimiento consistentemente superior en condiciones de alta carga.

3.3 Comparativa de las APIs GraphQL

Ahora es momento de analizar el rendimiento de las APIs GraphQL para encontrar la más eficiente entre las tres y compararla con la API ganadora de REST. La comparación se llevará de la misma manera que se hizo en REST, comparando cada caso de uso y luego eliminando la tecnología menos eficiente para en el caso de uso 5 encontrar la mejor.

3.3.1 Comparación para el Caso de Uso 1

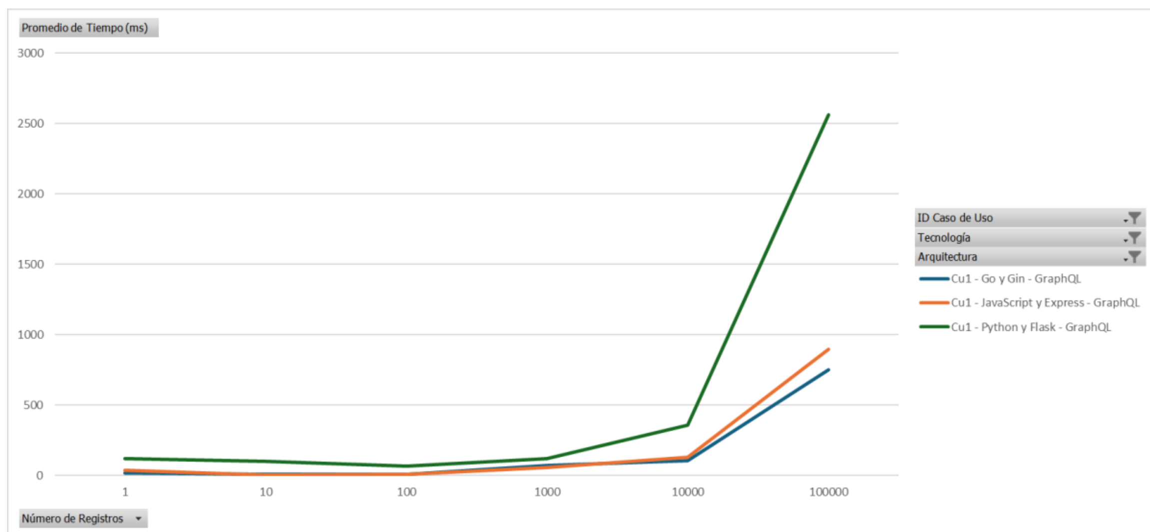

Figura 113 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 1

La gráfica del rendimiento en el primer caso de uso muestra un comportamiento muy similar al observado en la comparación de las APIs REST. Una vez más, la API de Python no logra destacar en términos de eficiencia, mientras que en esta ocasión la API de Go con Gin obtiene una ventaja al registrar un menor tiempo de respuesta. Este primer análisis del rendimiento de las APIs con arquitectura GraphQL permite anticipar cuál podría ser la combinación tecnológica con mejor desempeño, sin embargo para confirmar dicha hipótesis es necesario examinar el comportamiento en el resto de los casos de uso.

3.3.2 Comparación para el Caso de Uso 2

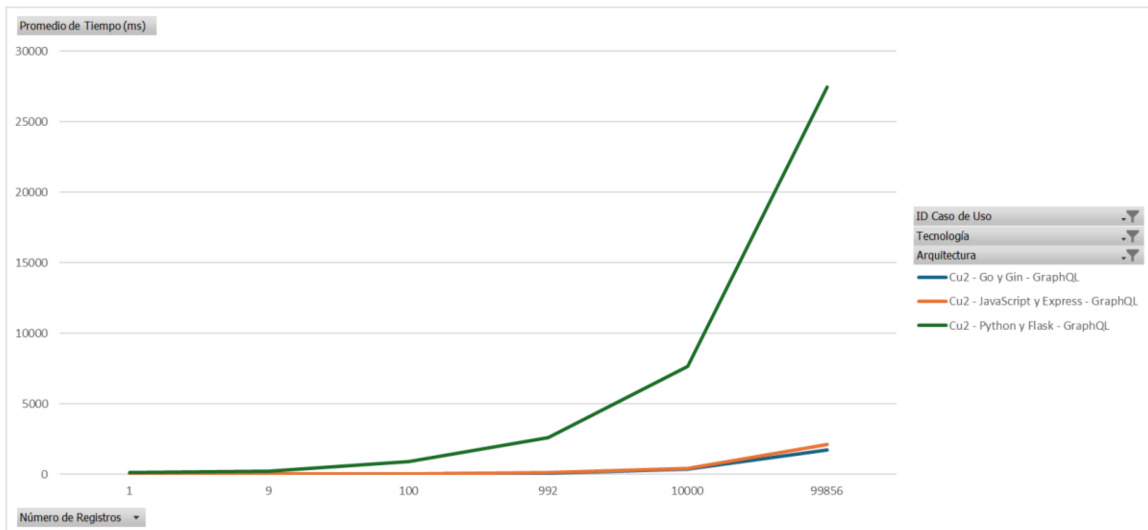

Figura 114 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 2

La tendencia observada en el caso de uso anterior se mantiene en este escenario, aunque con una diferencia aún más marcada en el desempeño de la API de Python que presenta un rendimiento significativamente inferior. En contraste, las otras dos APIs muestran resultados mucho más cercanos entre sí, para corroborar lo representado en el gráfico se recurre al análisis de los tiempos de ejecución: en el volumen de datos más alto, la API de Go con Gin alcanza el mejor desempeño con aproximadamente 1 723 milisegundos, seguida por la API de JavaScript con Express, con un tiempo promedio de 2 131 milisegundos y finalmente la API de Python con Flask obtiene el peor resultado, con un tiempo de ejecución cercano a 27 476 milisegundos.

Esta diferencia es considerable, ya que implica que la API más lenta (Python) es aproximadamente 16 veces menos eficiente que la más rápida (Go), confirmando así la marcada brecha de rendimiento entre ambas tecnologías.

3.3.3 Comparación para el Caso de Uso 3

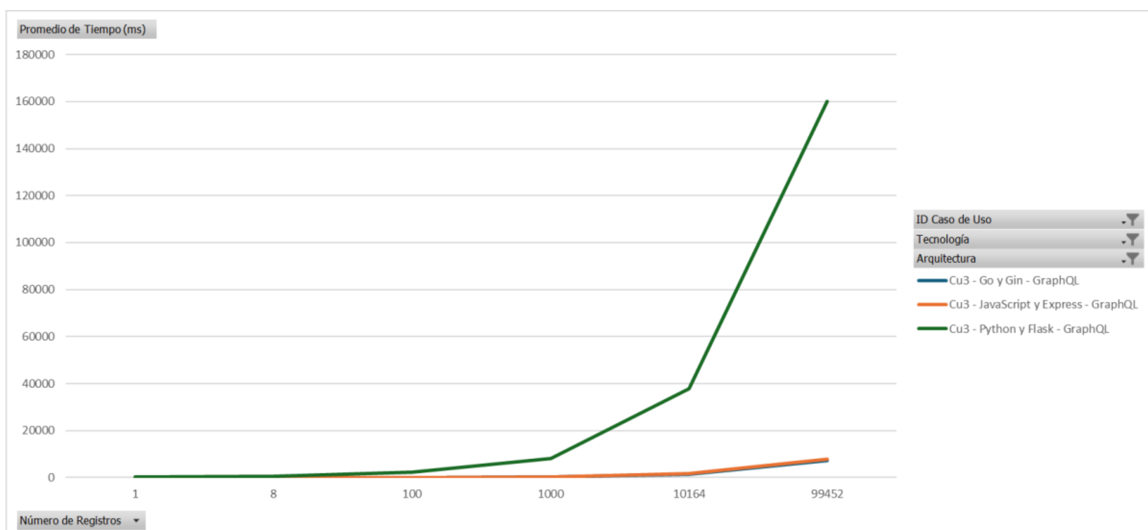

Figura 115 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 3

Una vez más, la API de Python con Flask demuestra ser la de menor rendimiento y tomando en cuenta los datos; la API de Go con Gin continúa siendo la más eficiente, completando la consulta del mayor volumen de datos en aproximadamente 7 102 milisegundos, en comparación la API de JavaScript con Express requiere alrededor de 7 810 milisegundos, lo que significaría un 10% más de tiempo de ejecución que Go. Por su parte, Python tarda aproximadamente 160 087 milisegundos, lo que equivale a cerca de 2,66 minutos, demostrando ser 22,5 veces menos eficiente que Go.

Dado su bajo rendimiento, se decide excluir la API de Python con Flask de las comparaciones en los casos de uso restantes.

3.3.4 Comparación para el Caso de Uso 4

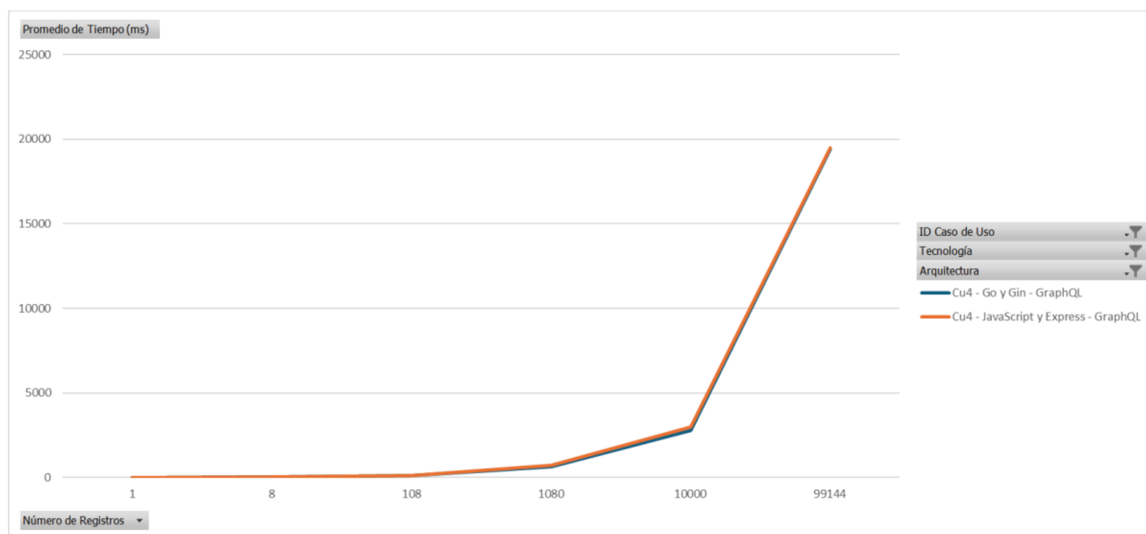

Figura 116 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 4

Con la API de Python fuera de la comparación, las diferencias de rendimiento entre la API de Go con Gin y la API de JavaScript con Express son prácticamente imperceptibles, ya que la gráfica muestra una curva que aparenta ser una única línea. No obstante, es posible realizar un análisis detallado al comparar los tiempos de ejecución en la consulta de mayor volumen de datos, aquí se observa que la API de Go completa la operación en aproximadamente 19 411 milisegundos, mientras que la API de JavaScript lo hace en 19 507 milisegundos, lo que representa una diferencia de apenas 96 milisegundos, este valor muestra que Go resulta aproximadamente 0,5% más eficiente que JavaScript en este escenario.

3.3.5 Comparación para el Caso de Uso 5

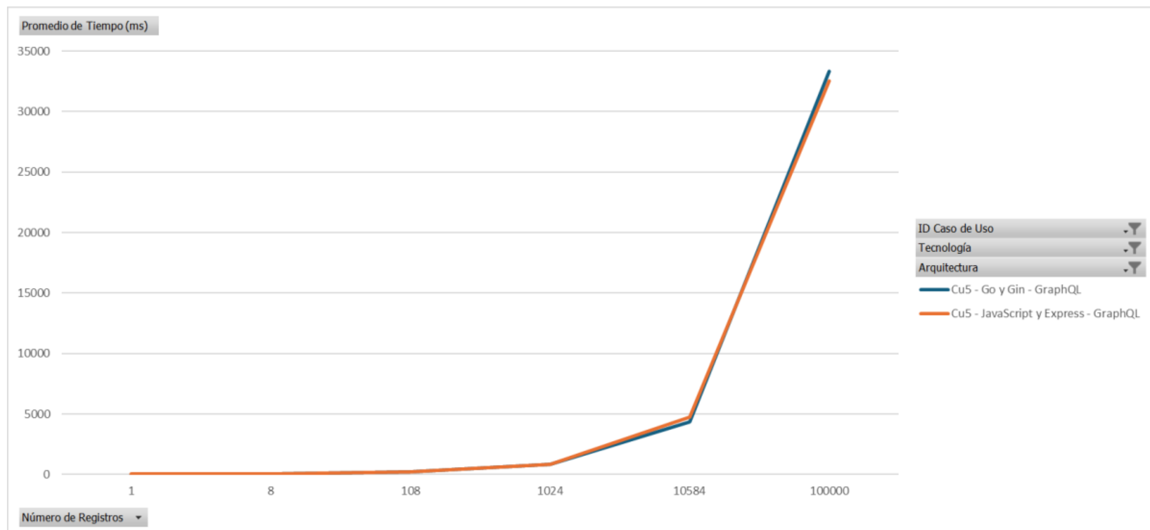

Figura 117 Tiempo de respuesta de las APIs GraphQL en el Caso de uso 5

A simple vista, se evidencia que la competencia entre ambas APIs es muy reñida. Sin embargo, al analizar con mayor detalle la gráfica, se puede distinguir que la línea correspondiente al desempeño de la API de Go con Gin (color azul) se sitúa ligeramente por encima de la línea de la API de JavaScript con Express, lo que indica una pequeña desventaja en términos de rendimiento respecto a JavaScript, esta diferencia se traduce en un tiempo de respuesta adicional de 774 milisegundos, ya que Go registra 33 330 milisegundos, mientras que JavaScript completa la misma consulta en 32 556 milisegundos.

Para examinar con mayor precisión el origen de esta ligera caída de rendimiento, se presenta a continuación un gráfico que detalla las tres iteraciones de la consulta, permitiendo observar las variaciones en cada ejecución.

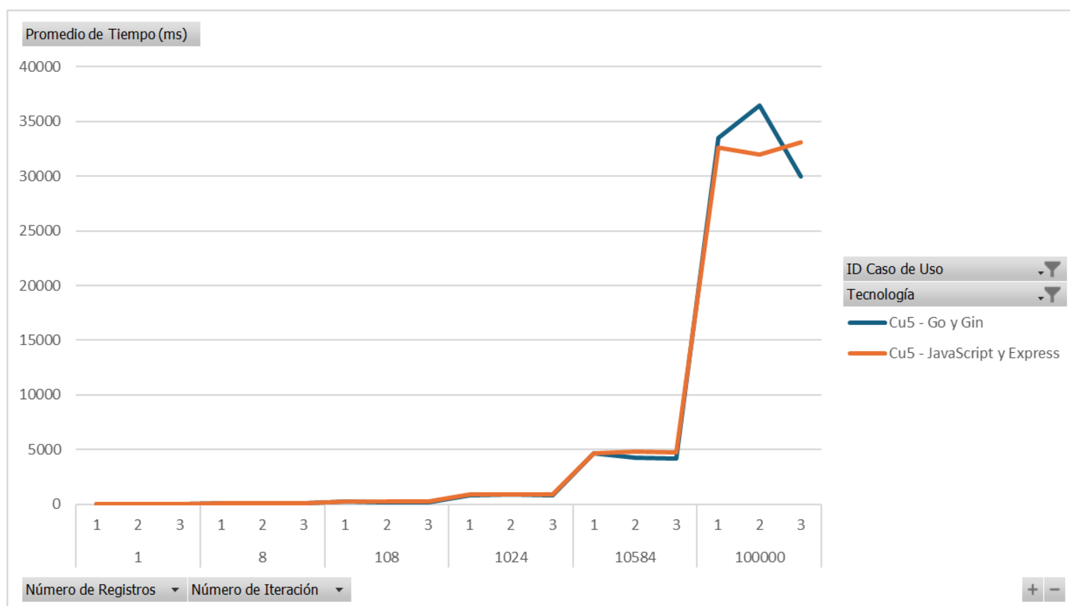

Figura 118 Desempeño de las APIs GraphQL en el Caso de uso 5

En esta gráfica se observa que durante la segunda iteración de la consulta con 100 000 registros, se produce un aumento repentino en el tiempo de respuesta, esto provoca que a pesar de que en la tercera iteración el tiempo disminuya considerablemente, el promedio de las tres iteraciones no permita que la API de Go con Gin recupere la ventaja que había mostrado frente a la API de JavaScript con Express en los casos de uso anteriores.

Asimismo, al analizar la comparación de ambas APIs desde el primer caso de uso, se puede identificar una tendencia clara: en dicha instancia, la diferencia entre los tiempos de ejecución es más significativa, posicionando a Go como la API de mejor rendimiento en ese escenario inicial.

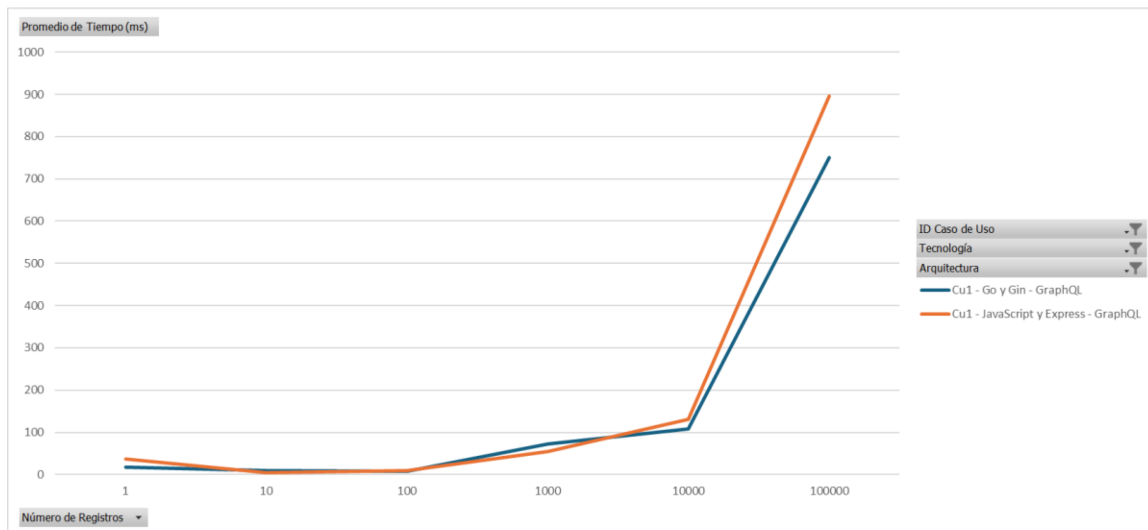

Figura 119 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 1

Luego, en el caso de uso siguiente aún no hay un gran cambio, puesto que Go aún es más eficiente para la consulta de datos y de hecho hasta gana un poco más de ventaja:

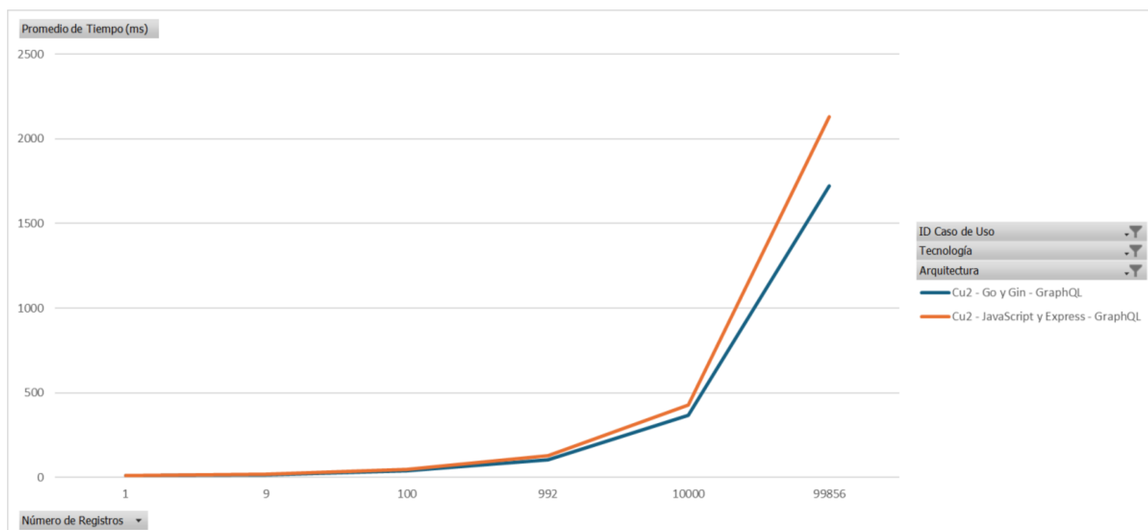

Figura 120 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 2

Pero el punto de inflexión ocurre en el tercer caso de uso, que es donde se comienza a cerrar la brecha de rendimiento entre ambas tecnologías:

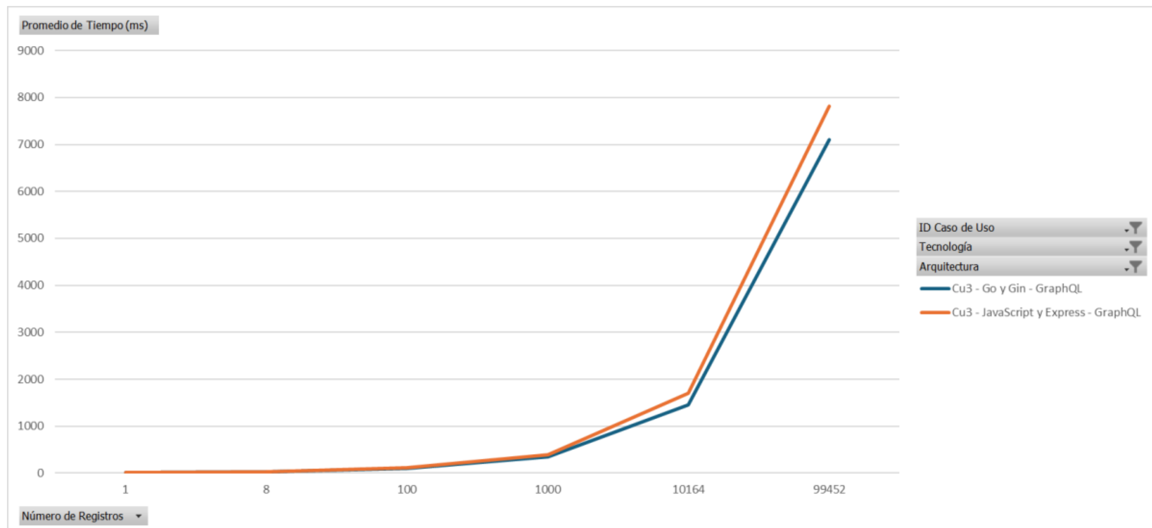

Figura 121 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 3

Y al pasar al cuarto caso de uso, esta brecha de rendimiento que separaba a Go de JavaScript desaparece, puesto que como muestra el gráfico ambas líneas de rendimiento están prácticamente sobrepuestas:

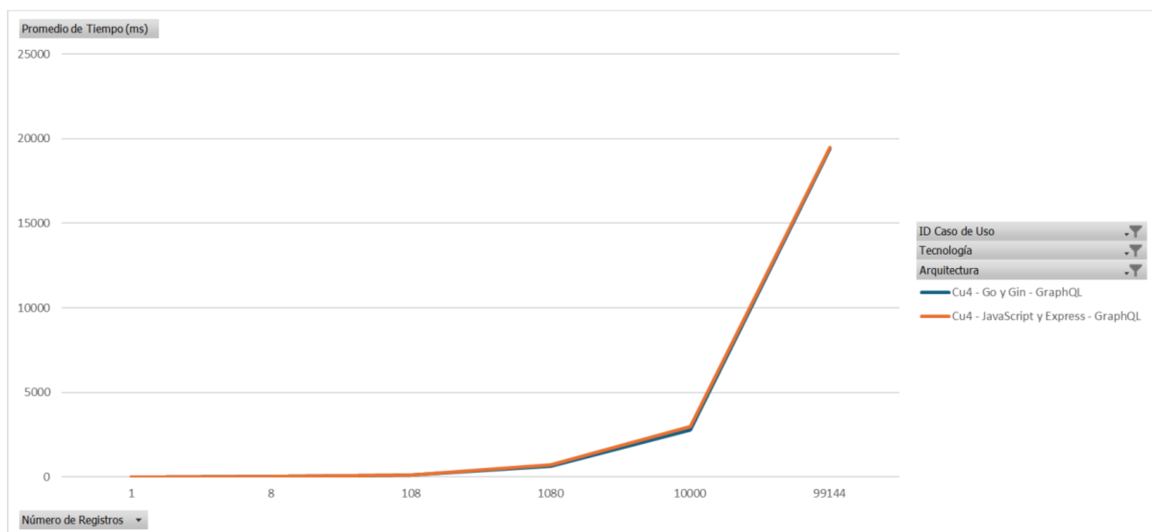

Figura 122 Tiempo de respuesta de las APIs de Go y JavaScript en el Caso de uso 4

A partir de este gráfico, se puede deducir que a medida que aumenta la cantidad de datos, la API de Go con Gin pierde eficiencia y es superada por la API de JavaScript con Express. Esta tendencia se evidencia de manera más clara en el último caso de uso, donde JavaScript logra una ligera ventaja sobre Go, a pesar de que esta última había liderado en los casos iniciales.

Dado que, a diferencia de las APIs REST no existe una API con un desempeño consistentemente diferenciador en todos los casos de uso y considerando que la tendencia indica que, con volúmenes aún mayores de datos, la API de JavaScript probablemente mantendría la ventaja sobre Go, mientras que en volúmenes pequeños Go es más eficiente, se decide incluir ambas APIs en la comparación final. De esta manera, la comparación final no se limitará a dos tecnologías, sino que se evaluarán tres alternativas en total.

3.4 Comparativa de REST con GraphQL

Con las tecnologías determinadas, el siguiente paso consiste en realizar el análisis de desempeño siguiendo la misma metodología aplicada en los casos anteriores, comenzando con el primer caso de uso.

3.4.1 Comparación para el Caso de Uso 1

Esta vez el gráfico muestra un tiempo de respuesta muy similar entre las tres APIs, para la consulta de datos con el menos volumen, pero a partir de los 100 datos ya se visualizan diferencias notables que se hacen aún más evidentes en la consulta de mayor cantidad de datos.

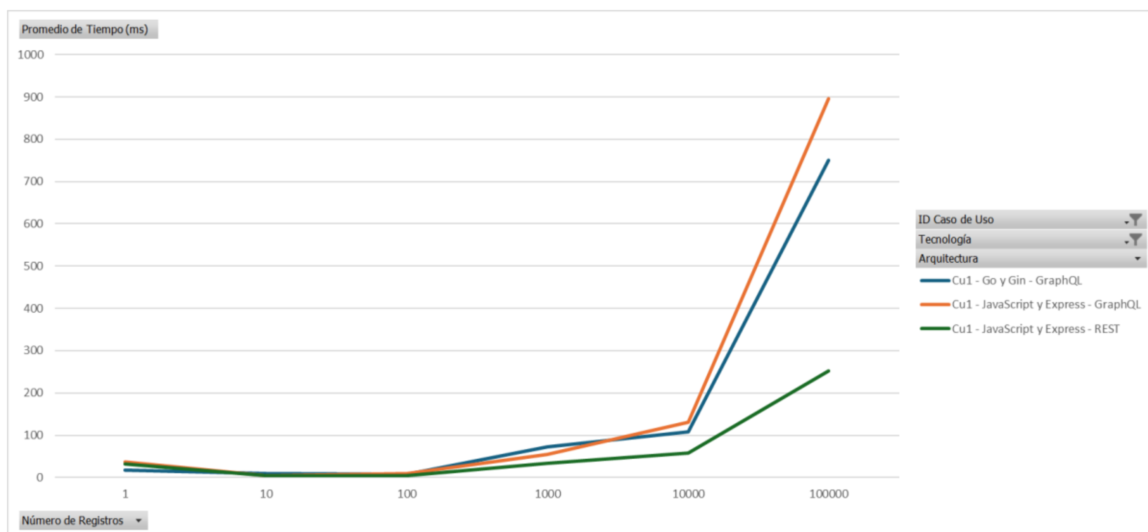

Figura 123 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 1

Al analizar con mayor detalle la consulta de 100 000 datos, donde se presenta la mayor diferencia entre las APIs, se observa que la API más eficiente es la construida con JavaScript y Express bajo arquitectura REST, seguida por la API desarrollada con Go y Gin y finalmente, en tercer lugar, se encuentra la API del mismo lenguaje y framework que la primera, pero desarrollada con arquitectura GraphQL.

En términos de tiempo de ejecución para completar la consulta con el mayor volumen de datos, la API REST de JavaScript requiere aproximadamente 251,67 ms, la API GraphQL de JavaScript tarda 896,33 ms, y la API de Go completa la tarea en 750,33 ms. De esta manera, se deduce que la API de Go es aproximadamente 3 veces más lenta que la API REST de JavaScript, mientras que la API GraphQL de JavaScript es aproximadamente 3,6 veces más lenta que la misma referencia.

Un dato adicional que se tendrá en cuenta para este análisis es el "costo promedio por registro adicional", este representa en milisegundos lo que le tomaría a la API procesar un registro más y se calculará aplicando la fórmula de la pendiente:

$$m = \frac{\Delta Y}{\Delta X} = \frac{y2 - y1}{x2 - x1}$$

Como ejemplo, la pendiente que existe en el gráfico, entre los 10 mil datos y los 1000 mil datos para la API de Go se calcula de la siguiente forma:

$$m = \frac{\Delta Y}{\Delta X} = \frac{750{,}33 - 108}{100000 - 10000} = 0{,}007137\ ms$$

Esto indica entonces que consultar un solo dato más significaría aumentar el tiempo de respuesta de la API para este caso de uso en 0,0071 ms aproximadamente, este mismo valor calculado para las demás APIs refleja lo siguiente:

- API REST - JavaScript con Express: 0,0025 ms
- API GraphQL - JavaScript con Express: 0,0089 ms
- API GraphQL - Go con Gin: 0,0071 ms

En este caso se puede observar que el costo en milisegundos por registro adicional también varía entre tecnologías, siendo la API de JavaScript con Express y bajo la arquitectura REST, la que mantiene este coste menor.

3.4.2 Comparación para el Caso de Uso 2

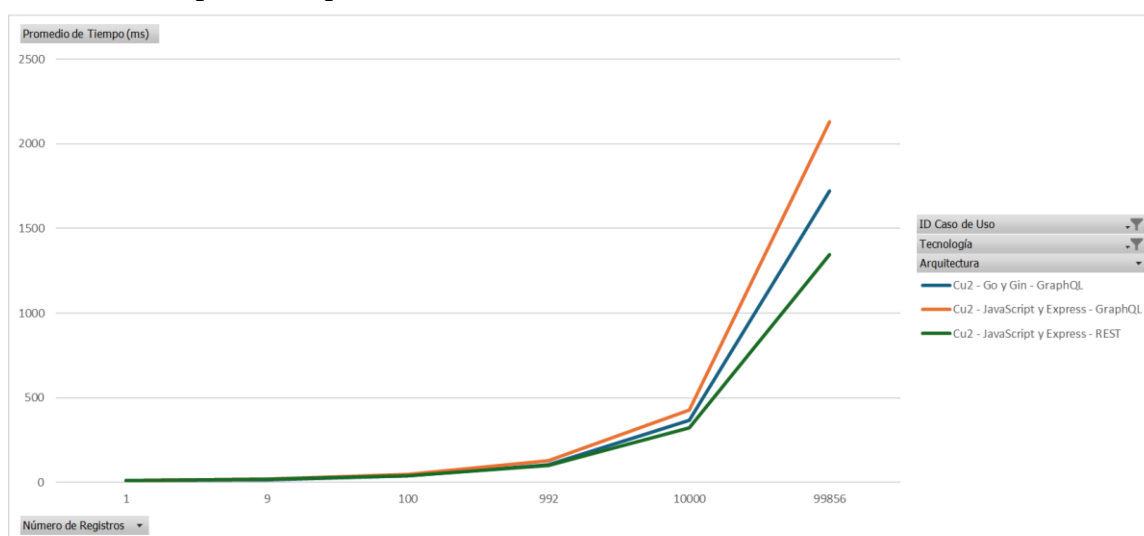

Figura 124 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 2

En la gráfica del segundo caso de uso se observa que la diferencia en los tiempos de respuesta entre las APIs se reduce con respecto al primer caso, para los dos primeros volúmenes de datos prácticamente no existe una diferencia apreciable; sin embargo, en los volúmenes mayores, el orden de rendimiento se mantiene igual al observado anteriormente.

Ordenando las APIs de mayor a menor rendimiento, se obtiene lo siguiente: la API de JavaScript con Express (REST) ocupa el primer lugar, con un tiempo de respuesta de aproximadamente 1 347,67 ms para la consulta de 99 856 registros; le sigue la API de Go con Gin, con un tiempo de 1 722,67 ms en la misma consulta; y finalmente, la API de JavaScript con GraphQL, con un tiempo de 2 130,67 ms.

Con el cálculo de la pendiente para el mayor volumen de datos se obtienen los siguientes datos:

- API REST - JavaScript con Express: 0,012 ms
- API GraphQL - JavaScript con Express: 0,019 ms

- API GraphQL - Go con Gin: 0,015 ms

Entre las dos APIs construidas con JavaScript, se puede ver que la API con arquitectura REST es 1.58 veces más eficiente que la construida con GraphQL.

3.4.3 Comparación para el Caso de Uso 3

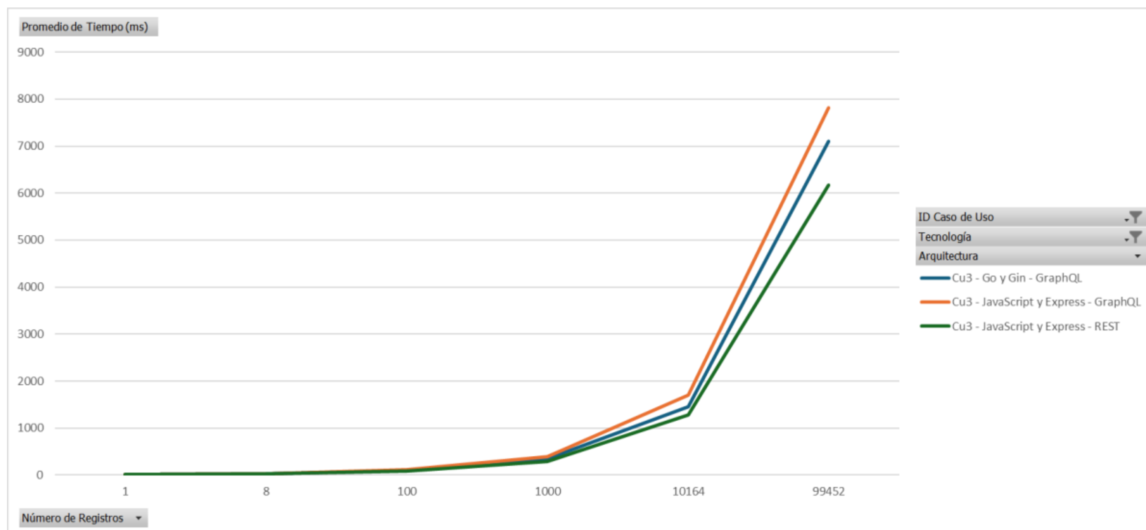

Figura 125 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 3

En este tercer caso de uso, la diferencia en los tiempos de respuesta para el mayor volumen de datos se reduce aún más, lo que podría indicar una tendencia que se mantenga en consultas más complejas, esta hipótesis podrá verificarse en los casos de uso siguientes. Al analizar los datos del gráfico, se observa que el orden de las APIs respecto al tiempo de respuesta se mantiene: la API de JavaScript con Express (REST) continúa siendo la más eficiente, con un tiempo de 6 174 ms, seguida por la API de Go con Gin, con 7 101,67 ms, y finalmente, la API de JavaScript con GraphQL, que registra un tiempo de 7 810,33 ms.

Adicionalmente las pendientes registradas en este caso de uso en la consulta de mayor cantidad de datos son las siguientes:

- API REST - JavaScript con Express: 0,054 ms
- API GraphQL - JavaScript con Express: 0,068 ms
- API GraphQL - Go con Gin: 0,063 ms

Esta vez las APIs de JavaScript reflejan un rendimiento de 1,25 veces entre la de REST y la de GraphQL.

3.4.4 Comparación para el Caso de Uso 4

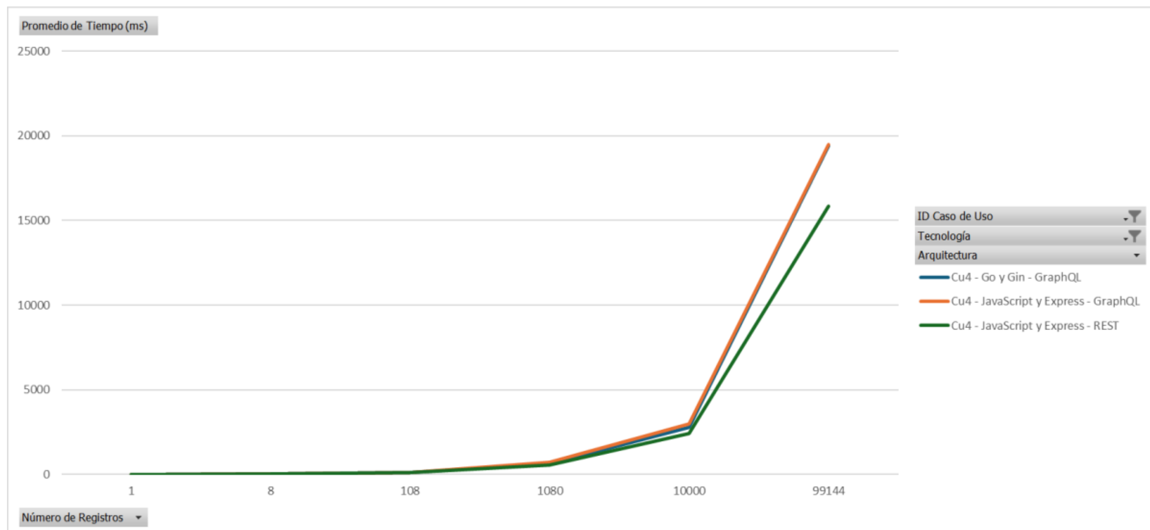

Figura 126 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 4

En este caso de uso ya se puede notar de mejor manera la tendencia que se venía gestando desde los casos de uso anteriores, se observa a simple vista que el rendimiento de las dos APIs GraphQL es idéntico, esta tendencia fue establecida cuando se analizó el rendimiento de las tres APIs GraphQL, por otra parte la API de REST continúa con un rendimiento mucho más optimo. Tomando en cuenta que nuevamente en cantidades pequeñas de datos el rendimiento es sumamente similar, es mejor analizar las diferencias donde las hay, lo que resulta en que el rendimiento de las tres APIs con el mayor número de datos consultados sea el siguiente:

1. JS-Express/REST: 15 857.33 ms
2. Go/Gin/GraphQL: 19 411 ms (es 1.22 veces más lento que la primer API)
3. JS-Express/GraphQL: 19 507.00 ms (es 1.23 veces más lento que la primer API)

En el tiempo de ejecución mostrado se observa que la segunda y tercera API comparten un rendimiento sumamente similar, pero que en este caso es superado por la primer API. Ahora en el caso del coste por dato extra consultado, la primer API vuelve a ser la mejor en rendimiento, mientras que al tener una diferencia tan ínfima las otras dos APIs tienen un coste casi idéntico.

- API REST - JavaScript con Express: 0,1509 ms
- API GraphQL - JavaScript con Express: 0,1853 ms (es 1.23 veces menos eficiente que la primer API)
- API GraphQL - Go con Gin: 0,1866 ms (es 1.24 veces menos eficiente que la primer API)

Como se puede ver el coste por consultar un dato adicional entre las dos APIs de GraphQL es sumamente similar, he incluso si aproximamos los valores, ambas APIs tendrían un coste de 0,19 ms por registro, esto no era así ates puesto que Go rendía de manera más eficiente, pero a medida que aumenta la complejidad de las consultas parece haber sido alcanzada por la API de JavaScript (GraphQL).

3.4.5 Comparación para el Caso de Uso 5

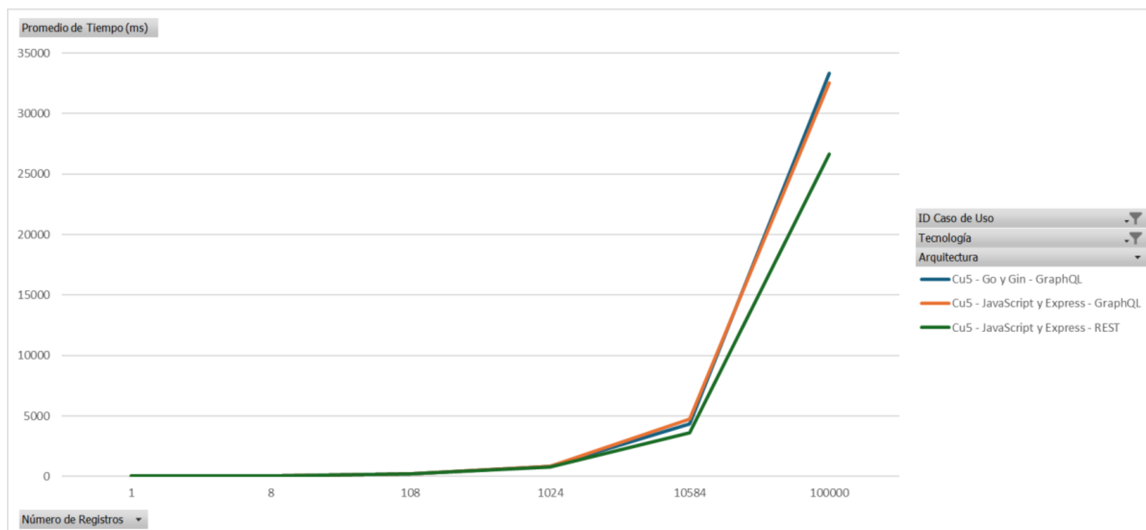

Figura 127 Comparativa del tiempo de respuesta entre REST y GraphQL - Caso de Uso 5

Finalmente en el quinto caso de uso se puede observar que la API de JavaScript con REST sigue liderando con un menor tiempo de ejecución, mientras que la API de Go finalmente fue superada por la segunda API de JavaScript, esto se corrobora al analizar los datos obtenidos de este gráfico. Comenzando con el tiempo de respuesta que queda de la siguiente manera:

1. JS-Express/REST: 26 684 ms
2. JS-Express/GraphQL: 32 556,33 ms
3. Go/Gin/GraphQL: 33 330 ms

A partir de estos resultados, se deduce que la API de Go pierde su segundo lugar, siendo reemplazada por la API de JavaScript con GraphQL, también al calcular la diferencia relativa en los tiempos de ejecución, se observa que la segunda API es aproximadamente 22% más lenta que la primera, y la tercera API es alrededor de 25% más lenta que la primera.

Seguido está la pendiente que arroja los siguientes datos:

- API REST - JavaScript/Express: 0,2580 ms/registro
- API GraphQL - JavaScript/Express: 0,3110 ms/registro
- API GraphQL - Go/Gin: 0,3240 ms/registro

De esto se deduce que la API REST de JavaScript es 1,2 veces más eficiente que la API GraphQL de JavaScript y así mismo es 1,26 veces más eficiente que la API de Go. Aunque entre las dos APIs GraphQL la más eficiente pasó a ser en esta ocasión la de JavaScript con un 1,04 más de eficiencia que la de Go, lo que representa un aproximado del 4% en la diferencia de desempeño.

Conclusiones

- Con el desarrollo del marco teórico se definió la base conceptual científica que sirvió para determinar la arquitectura tecnológica, las tecnologías para el desarrollo de las APIs tanto REST como también las GraphQL; el diseño de la base de datos y el diseño del experimento computacional para realizar la comparativa de la eficiencia con respecto al

tiempo de respuesta de dichas APIs. También se pudo definir la base conceptual de la norma ISO/IEC 25023 para a evaluar la calidad del software.

- Mediante el diseño de un experimento computacional, usando las diferentes APIs desarrolladas se logró comparar la eficiencia del consumo de datos entre las diferentes tecnologías con las que fueron creadas, basando los resultados en las métricas ISO / IEC 25023. En donde la ejecución del experimento mostro que las variables dependientes se ven afectadas por las variables independientes, es decir que el tiempo de respuesta es afectado por las tecnologías que usa la API que se conecta a la base de datos.
- La API REST desarrollada con JavaScript y Express demostró ser la más eficiente en términos de tiempo de respuesta, manteniendo un desempeño estable y escalable en consultas con volúmenes de datos pequeños, medianos y grandes. Por su parte, las APIs de GraphQL, aunque funcionales, mostraron tiempos de respuesta superiores, siendo menos eficientes especialmente en consultas con grandes volúmenes de datos o niveles de complejidad elevados.
- Las APIs desarrolladas en Python con Flask demostraron un rendimiento consistentemente inferior en todos los casos de uso, llegando a ser hasta 23 veces más lentas que las alternativas en JavaScript y Go en consultas de gran volumen. Este bajo desempeño hizo que fueran descartadas en los análisis de los casos de uso más complejos.
- La API construida en Go con Gin presentó un rendimiento competitivo, especialmente en consultas de volúmenes bajos o medios, pero su ventaja disminuyó a medida que se incrementaba la complejidad de las consultas, siendo finalmente superada por la API de JavaScript con GraphQL en el último caso de uso. Esto evidencia que Go es eficiente, pero su desempeño puede verse afectado en operaciones anidadas complejas.

Recomendaciones

- Se recomienda incorporar métricas adicionales de rendimiento, como latencia por operación y coste por registro procesado, para obtener un análisis más integral de la eficiencia de las APIs bajo distintas arquitecturas y volúmenes de datos. Esto con la finalidad de identificar cuellos de botella y posibles optimizaciones en escenarios más complejos.
- Se sugiere incluir en futuras pruebas operaciones CRUD completas (crear, actualizar y eliminar), además de las consultas de lectura. Esto permitiría estudiar diferencias de rendimiento y escalabilidad de REST y GraphQL en un contexto más amplio y representativo de aplicaciones reales ya que el presente trabajo solo abarca las consultas GET.
- Se sugiere replicar los experimentos utilizando otras tecnologías de almacenamiento, como bases de datos NoSQL o sistemas distribuidos, para evaluar cómo la arquitectura de la base de datos impacta en el desempeño de REST y GraphQL.
- Se recomienda aumentar tanto la cantidad de registros consultados como el número de niveles de la base de datos en futuras pruebas, con el objetivo de corroborar la tendencia de rendimiento observada en la API de Go con Gin. Esto permitiría validar si el comportamiento detectado, donde Go pierde eficiencia relativa frente a JavaScript en consultas muy complejas, se mantiene con volúmenes de datos mayores y estructuras jerárquicas más profundas.

Bibliografía

Andrés, C., & Solano, R. (2019). *ANÁLISIS COMPARATIVO ENTRE LOS ESTÁNDARES ORIENTADO A SERVICIOS WEB SOAP, REST Y GRAPHQL*. https://repositorio.puce.edu.ec/items/fc83d222-4fde-4d17-a89d-b090d0b83c52

Barta, P. (2015, March 15). *Express*. Https://Expressjs.Com/. https://expressjs.com/

Bello, G. (2023, December 7). *GraphQL vs. REST | Postman Blog*. Https://Blog.Postman.Com/Graphql-vs-Rest/.

Benjie. (2025, March 6). *Introduction to GraphQL | GraphQL*. Https://Graphql.Org/. https://graphql.org/learn/

Brito, G., & Valente, M. T. (2020). REST vs GraphQL: A controlled experiment. *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020*, 81–91. https://doi.org/10.1109/ICSA47634.2020.00016

Cassidy, T., Arya, H., & Coulter, D. (2024, August 22). *Introduction to microservices on Azure - Azure Service Fabric*. Https://Www.Microsoft.Com/Es-Ec/. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices

Chandler, H. (2022, May 21). *Arquitectura de microservicios | Atlassian*. Https://Www.Atlassian.Com/Es/. https://www.atlassian.com/es/microservices/cloud-computing/advantages-of-microservices

Chrystal R. (2024, March 29). *GraphQL vs REST: ¿Cuál es la diferencia? | IBM*. Https://Www.Ibm.Com/Mx-Es. https://www.ibm.com/mx-es/think/topics/graphql-vs-rest-api

Fachat, A. (2024, January 30). *Challenges and benefits of the microservice architectural style, Part 1 - IBM Developer*. Https://Developer.Ibm.Com/. https://developer.ibm.com/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-1/

Flask. (2019, July 2). *Welcome to Flask*. Https://Flask.Palletsprojects.Com/En/Stable/.

Gin Web Framework. (2025, April 6). Https://Gin-Gonic.Com/Es/.

Go. (2019, October 27). Https://Go.Dev/.

Goodwin, M. (2024, April 9). *What is an Application Programming Interface (API)? | IBM*. https://www.ibm.com/mx-es. https://www.ibm.com/topics/api

Gunnell, M. (2024, March 26). *¿Qué es una llamada API? Tipos, ejemplos y usos*. Https://Www.Techopedia.Com/Es/. https://www.techopedia.com/es/definicion/llamada-api

Humble, C. (2021, July 27). *The Future of Microservices? More Abstractions - The New Stack*. Https://Thenewstack.Io/. https://thenewstack.io/microservices/the-future-of-microservices-more-abstractions/

ISO/IEC. (2016). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality*. Https://Www.Iso.Org/Es/Home. https://www.iso.org/obp/ui/es/#iso:std:iso-iec:25023:ed-1:v1:en

Jara Córdova, O., Arias Gonzáles, J., Antonio Flores Limo, F., & Anthony Balmaceda Flores, C. (2023). *Métodos mixtos de investigación para principiantes Timoteo Cueva Luza*. https://doi.org/10.35622/inudi.b.106

Mendoza, G., Correa, I., Merchán, J., Álaba, P., & Bermeo, J. (2021, September 20). *Plan de Creación de Oportunidades 2021-2025*. Https://Www.Planificacion.Gob.Ec/. https://www.planificacion.gob.ec/wp-content/uploads/2021/09/Plan-de-Creacio%CC%81n-de-Oportunidades-2021-2025-Aprobado.pdf

Oracle. (2022, June 6). *¿Qué es Java y por qué lo necesito?* Https://Www.Java.Com/Es/Download/Help/Whatis_java.Html.

PostgreSQL Global Development Group. (2011, September 12). *¿Qué es PostgreSQL?* Https://Www.Postgresql.Org/About/.

Python Software Foundation. (2023, May 12). *El tutorial de Python*. Https://Docs.Python.Org/Es/3.13/Tutorial/Index.Html.

Quiña-Mera, A., Fernandez, P., García, M., & Ruiz-Cortés, A. (2023). *GraphQL: A Systematic Mapping Study*. https://doi.org/10.1145/3561818

Quiña-Mera, A., Guitarra de la Cruz, Z. M., & Guevara-Vega, C. (2025). Efficiency study of GraphQL and REST Microservices in Docker containers: A computational experiment. *Data and Metadata*, *4*. https://doi.org/10.56294/DM2025199

Red Hat. (2018, June 29). *El concepto de las interfaces de programación de aplicaciones*. Https://Www.Redhat.Com/Es/Topics/Api.

Red Hat. (2023, May 1). *What are microservices?* Https://Www.Redhat.Com/En. https://www.redhat.com/en/topics/microservices/what-are-microservices

Rosado, G. L. (2024, July). *Reporte Técnico: Proyecto de JavaScript*. Https://Www.Researchgate.Net/Publication/382047310_Reporte_Tecnico_Proyecto_de_JavaScript.

Sanders, S., & Singh, S. (2025, April 27). *Schemas and Types | GraphQL*. Https://Graphql.Org/. https://graphql.org/learn/schema/

Santías Dema, I. (2020, August 17). *17 Objetivos de Desarrollo Sostenible de la ONU*. Https://Www.Ecologiaverde.Com/. https://www.ecologiaverde.com/17-objetivos-de-desarrollo-sostenible-de-la-onu-2991.html

Sayago Heredia, J., & Flores, E. (2019). *Análisis Comparativo entre los Estándares Orientados a Servicios Web SOAP, REST y GRAPHQL*. https://doi.org/10.5281/zenodo.3592004

Sharma, B. P. (2020). *NodeJS Quick Learning*. https://doi.org/10.13140/RG.2.2.34610.43203

Spring Boot. (2013). *Https://Spring.Io/Projects/Spring-Boot*.

Swanson, N. (2024, May). *The Evolution of API Architectures; REST & GraphQL*. Https://Scholarsbank.Uoregon.Edu/Home. https://scholarsbank.uoregon.edu/server/api/core/bitstreams/e21656fa-73f8-46c0-bfc6-cb573bdeee2b/content

Toapanta, K. (2024, November 13). *Microservicios: la arquitectura clave para el desarrollo de software*. Https://Itsqmet.Edu.Ec/. https://itsqmet.edu.ec/microservicios/

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in Software Engineering. In *Experimentation in Software Engineering* (pp. 123–151). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-29044-2_10

Anexos

Tiempos de Respuesta – REST

ID Caso de Uso	Arquitectura	Tecnología	Número de Registros	Distribución	Número de Iteración	Tiempo (ms)	Registro de Iteraciones
Cu1	REST	Python y Flask	1	1	1	80	1
Cu1	REST	Python y Flask	1	1	2	66	1
Cu1	REST	Python y Flask	1	1	3	66	1
Cu1	REST	Python y Flask	10	10	1	52	10
Cu1	REST	Python y Flask	10	10	2	56	10
Cu1	REST	Python y Flask	10	10	3	59	10
Cu1	REST	Python y Flask	100	100	1	185	100
Cu1	REST	Python y Flask	100	100	2	62	100
Cu1	REST	Python y Flask	100	100	3	64	100
Cu1	REST	Python y Flask	1.000	1.000	1	112	1000
Cu1	REST	Python y Flask	1.000	1.000	2	96	1000
Cu1	REST	Python y Flask	1.000	1.000	3	105	1000
Cu1	REST	Python y Flask	10.000	10.000	1	193	10000
Cu1	REST	Python y Flask	10.000	10.000	2	176	10000
Cu1	REST	Python y Flask	10.000	10.000	3	180	10000
Cu1	REST	Python y Flask	100.000	100.000	1	1.207	100000
Cu1	REST	Python y Flask	100.000	100.000	2	1.101	100000
Cu1	REST	Python y Flask	100.000	100.000	3	1.020	100000
Cu2	REST	Python y Flask	1	1x1	1	119	1 - 1
Cu2	REST	Python y Flask	1	1x1	2	113	1 - 1
Cu2	REST	Python y Flask	1	1x1	3	119	1 - 1
Cu2	REST	Python y Flask	9	3x3	1	442	3 - 9
Cu2	REST	Python y Flask	9	3x3	2	242	3 - 9

Cu2	REST	Python y Flask	9	3x3	3	225	3 - 9
Cu2	REST	Python y Flask	100	10x10	1	620	10 - 100
Cu2	REST	Python y Flask	100	10x10	2	756	10 - 100
Cu2	REST	Python y Flask	100	10x10	3	648	10 - 100
Cu2	REST	Python y Flask	992	32x31	1	2.432	32 - 992
Cu2	REST	Python y Flask	992	32x31	2	1.975	32 - 992
Cu2	REST	Python y Flask	992	32x31	3	2.506	32 - 992
Cu2	REST	Python y Flask	10.000	100x100	1	6.667	100 - 10000
Cu2	REST	Python y Flask	10.000	100x100	2	6.771	100 - 10000
Cu2	REST	Python y Flask	10.000	100x100	3	7.135	100 - 10000
Cu2	REST	Python y Flask	99.856	316x316	1	23.476	316 - 99856
Cu2	REST	Python y Flask	99.856	316x316	2	23.466	316 - 99856
Cu2	REST	Python y Flask	99.856	316x316	3	23.576	316 - 99856
Cu3	REST	Python y Flask	1	1x1x1	1	186	1 - 1 - 1
Cu3	REST	Python y Flask	1	1x1x1	2	190	1 - 1 - 1
Cu3	REST	Python y Flask	1	1x1x1	3	192	1 - 1 - 1
Cu3	REST	Python y Flask	8	2x2x2	1	397	2 - 4 - 8
Cu3	REST	Python y Flask	8	2x2x2	2	419	2 - 4 - 8
Cu3	REST	Python y Flask	8	2x2x2	3	509	2 - 4 - 8
Cu3	REST	Python y Flask	100	5x5x4	1	2.278	5 - 25 - 100
Cu3	REST	Python y Flask	100	5x5x4	2	2.360	5 - 25 - 100
Cu3	REST	Python y Flask	100	5x5x4	3	2.083	5 - 25 - 100
Cu3	REST	Python y Flask	1.000	10x10x10	1	7.802	10 - 100 - 1000
Cu3	REST	Python y Flask	1.000	10x10x10	2	8.215	10 - 100 - 1000
Cu3	REST	Python y Flask	1.000	10x10x10	3	8.475	10 - 100 - 1000

Cu3	REST	Python y Flask	10.164	22x22x21	1	35.648	22 - 484 - 10164
Cu3	REST	Python y Flask	10.164	22x22x21	2	34.456	22 - 484 - 10164
Cu3	REST	Python y Flask	10.164	22x22x21	3	33.615	22 - 484 - 10164
Cu3	REST	Python y Flask	99.452	46x46x47	1	149.847	46 - 2116 - 99452
Cu3	REST	Python y Flask	99.452	46x46x47	2	150.463	46 - 2116 - 99452
Cu3	REST	Python y Flask	99.452	46x46x47	3	149.618	46 - 2116 - 99452
Cu4	REST	Python y Flask	1	1x1x1x1	1	348	1 - 1 - 1 - 1
Cu4	REST	Python y Flask	1	1x1x1x1	2	218	1 - 1 - 1 - 1
Cu4	REST	Python y Flask	1	1x1x1x1	3	223	1 - 1 - 1 - 1
Cu4	REST	Python y Flask	8	2x2x2x1	1	1.227	2 - 4 - 8 - 8
Cu4	REST	Python y Flask	8	2x2x2x1	2	1.291	2 - 4 - 8 - 8
Cu4	REST	Python y Flask	8	2x2x2x1	3	986	2 - 4 - 8 - 8
Cu4	REST	Python y Flask	108	3x3x3x4	1	2.678	3 - 9 - 27 - 108
Cu4	REST	Python y Flask	108	3x3x3x4	2	2.801	3 - 9 - 27 - 108
Cu4	REST	Python y Flask	108	3x3x3x4	3	2.695	3 - 9 - 27 - 108
Cu4	REST	Python y Flask	1.080	6x6x6x5	1	17.395	6 - 36 - 216 - 1080
Cu4	REST	Python y Flask	1.080	6x6x6x5	2	18.140	6 - 36 - 216 - 1080
Cu4	REST	Python y Flask	1.080	6x6x6x5	3	16.999	6 - 36 - 216 - 1080
Cu4	REST	Python y Flask	10.000	10x10x10x10	1	75.014	10 - 100 - 1000 - 10000
Cu4	REST	Python y Flask	10.000	10x10x10x10	2	77.604	10 - 100 - 1000 - 10000
Cu4	REST	Python y Flask	10.000	10x10x10x10	3	75.595	10 - 100 - 1000 - 10000
Cu4	REST	Python y Flask	99.144	18x18x18x17	1	423.321	18 - 324 - 5832 - 99144
Cu4	REST	Python y Flask	99.144	18x18x18x17	2	425.758	18 - 324 - 5832 - 99144
Cu4	REST	Python y Flask	99.144	18x18x18x17	3	424.380	18 - 324 - 5832 - 99144
Cu5	REST	Python y Flask	1	1x1x1x1x1	1	448	1 - 1 - 1 - 1 - 1

Cu5	REST	Python y Flask	1	1x1x1x1x1	2	333	1 - 1 - 1 - 1 - 1
Cu5	REST	Python y Flask	1	1x1x1x1x1	3	464	1 - 1 - 1 - 1 - 1
Cu5	REST	Python y Flask	8	2x2x1x2x1	1	1.357	2 - 4 - 4 - 8 - 8
Cu5	REST	Python y Flask	8	2x2x1x2x1	2	1.895	2 - 4 - 4 - 8 - 8
Cu5	REST	Python y Flask	8	2x2x1x2x1	3	1.554	2 - 4 - 4 - 8 - 8
Cu5	REST	Python y Flask	108	3x3x2x3x3	1	6.879	3 - 9 - 18 - 54 - 108
Cu5	REST	Python y Flask	108	3x3x2x3x3	2	7.322	3 - 9 - 18 - 54 - 108
Cu5	REST	Python y Flask	108	3x3x2x3x3	3	8.164	3 - 9 - 18 - 54 - 108
Cu5	REST	Python y Flask	1.024	4x4x4x4x4	1	28.367	4 - 16 - 64 - 256 - 1024
Cu5	REST	Python y Flask	1.024	4x4x4x4x4	2	29.657	4 - 16 - 64 - 256 - 1024
Cu5	REST	Python y Flask	1.024	4x4x4x4x4	3	29.110	4 - 16 - 64 - 256 - 1024
Cu5	REST	Python y Flask	10.584	6x6x7x6x6	1	156.969	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Python y Flask	10.584	6x6x7x6x6	2	157.268	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Python y Flask	10.584	6x6x7x6x6	3	157.146	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Python y Flask	100.000	10x10x10x10x10	1	985.669	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	Python y Flask	100.000	10x10x10x10x10	2	980.449	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	Python y Flask	100.000	10x10x10x10x10	3	986.821	10 - 100 - 1000 - 10000 - 100000
Cu1	REST	JavaScript y Express	1	1	1	89	1
Cu1	REST	JavaScript y Express	1	1	2	4	1
Cu1	REST	JavaScript y Express	1	1	3	3	1
Cu1	REST	JavaScript y Express	10	10	1	6	10
Cu1	REST	JavaScript y Express	10	10	2	3	10
Cu1	REST	JavaScript y Express	10	10	3	3	10
Cu1	REST	JavaScript y Express	100	100	1	6	100

Cu1	REST	JavaScript y Express	100	100	2	5	100
Cu1	REST	JavaScript y Express	100	100	3	5	100
Cu1	REST	JavaScript y Express	1.000	1.000	1	45	1000
Cu1	REST	JavaScript y Express	1.000	1.000	2	27	1000
Cu1	REST	JavaScript y Express	1.000	1.000	3	29	1000
Cu1	REST	JavaScript y Express	10.000	10.000	1	61	10000
Cu1	REST	JavaScript y Express	10.000	10.000	2	56	10000
Cu1	REST	JavaScript y Express	10.000	10.000	3	57	10000
Cu1	REST	JavaScript y Express	100.000	100.000	1	253	100000
Cu1	REST	JavaScript y Express	100.000	100.000	2	261	100000
Cu1	REST	JavaScript y Express	100.000	100.000	3	241	100000
Cu2	REST	JavaScript y Express	1	1x1	1	13	1 - 1
Cu2	REST	JavaScript y Express	1	1x1	2	9	1 - 1
Cu2	REST	JavaScript y Express	1	1x1	3	8	1 - 1
Cu2	REST	JavaScript y Express	9	3x3	1	24	3 - 9
Cu2	REST	JavaScript y Express	9	3x3	2	19	3 - 9
Cu2	REST	JavaScript y Express	9	3x3	3	15	3 - 9
Cu2	REST	JavaScript y Express	100	10x10	1	34	10 - 100
Cu2	REST	JavaScript y Express	100	10x10	2	43	10 - 100
Cu2	REST	JavaScript y Express	100	10x10	3	38	10 - 100
Cu2	REST	JavaScript y Express	992	32x31	1	104	32 - 992
Cu2	REST	JavaScript y Express	992	32x31	2	99	32 - 992
Cu2	REST	JavaScript y Express	992	32x31	3	102	32 - 992
Cu2	REST	JavaScript y Express	10.000	100x100	1	346	100 - 10000
Cu2	REST	JavaScript y Express	10.000	100x100	2	321	100 - 10000

Cu2	REST	JavaScript y Express	10.000	100x100	3	297	100 - 10000
Cu2	REST	JavaScript y Express	99.856	316x316	1	1.394	316 - 99856
Cu2	REST	JavaScript y Express	99.856	316x316	2	1.331	316 - 99856
Cu2	REST	JavaScript y Express	99.856	316x316	3	1.318	316 - 99856
Cu3	REST	JavaScript y Express	1	1x1x1	1	14	1 - 1 - 1
Cu3	REST	JavaScript y Express	1	1x1x1	2	11	1 - 1 - 1
Cu3	REST	JavaScript y Express	1	1x1x1	3	11	1 - 1 - 1
Cu3	REST	JavaScript y Express	8	2x2x2	1	23	2 - 4 - 8
Cu3	REST	JavaScript y Express	8	2x2x2	2	25	2 - 4 - 8
Cu3	REST	JavaScript y Express	8	2x2x2	3	21	2 - 4 - 8
Cu3	REST	JavaScript y Express	100	5x5x4	1	83	5 - 25 - 100
Cu3	REST	JavaScript y Express	100	5x5x4	2	84	5 - 25 - 100
Cu3	REST	JavaScript y Express	100	5x5x4	3	72	5 - 25 - 100
Cu3	REST	JavaScript y Express	1.000	10x10x10	1	275	10 - 100 - 1000
Cu3	REST	JavaScript y Express	1.000	10x10x10	2	295	10 - 100 - 1000
Cu3	REST	JavaScript y Express	1.000	10x10x10	3	285	10 - 100 - 1000
Cu3	REST	JavaScript y Express	10.164	22x22x21	1	1.278	22 - 484 - 10164
Cu3	REST	JavaScript y Express	10.164	22x22x21	2	1.280	22 - 484 - 10164
Cu3	REST	JavaScript y Express	10.164	22x22x21	3	1.267	22 - 484 - 10164
Cu3	REST	JavaScript y Express	99.452	46x46x47	1	6.482	46 - 2116 - 99452
Cu3	REST	JavaScript y Express	99.452	46x46x47	2	6.002	46 - 2116 - 99452
Cu3	REST	JavaScript y Express	99.452	46x46x47	3	6.038	46 - 2116 - 99452
Cu4	REST	JavaScript y Express	1	1x1x1x1	1	15	1 - 1 - 1 - 1
Cu4	REST	JavaScript y Express	1	1x1x1x1	2	12	1 - 1 - 1 - 1
Cu4	REST	JavaScript y Express	1	1x1x1x1	3	11	1 - 1 - 1 - 1

Cu4	REST	JavaScript y Express	8	2x2x2x1	1	36	2 - 4 - 8 - 8
Cu4	REST	JavaScript y Express	8	2x2x2x1	2	37	2 - 4 - 8 - 8
Cu4	REST	JavaScript y Express	8	2x2x2x1	3	35	2 - 4 - 8 - 8
Cu4	REST	JavaScript y Express	108	3x3x3x4	1	98	3 - 9 - 27 - 108
Cu4	REST	JavaScript y Express	108	3x3x3x4	2	97	3 - 9 - 27 - 108
Cu4	REST	JavaScript y Express	108	3x3x3x4	3	95	3 - 9 - 27 - 108
Cu4	REST	JavaScript y Express	1.080	6x6x6x5	1	547	6 - 36 - 216 - 1080
Cu4	REST	JavaScript y Express	1.080	6x6x6x5	2	588	6 - 36 - 216 - 1080
Cu4	REST	JavaScript y Express	1.080	6x6x6x5	3	572	6 - 36 - 216 - 1080
Cu4	REST	JavaScript y Express	10.000	10x10x10x10	1	2.411	10 - 100 - 1000 - 10000
Cu4	REST	JavaScript y Express	10.000	10x10x10x10	2	2.418	10 - 100 - 1000 - 10000
Cu4	REST	JavaScript y Express	10.000	10x10x10x10	3	2.397	10 - 100 - 1000 - 10000
Cu4	REST	JavaScript y Express	99.144	18x18x18x17	1	15.818	18 - 324 - 5832 - 99144
Cu4	REST	JavaScript y Express	99.144	18x18x18x17	2	15.924	18 - 324 - 5832 - 99144
Cu4	REST	JavaScript y Express	99.144	18x18x18x17	3	15.830	18 - 324 - 5832 - 99144
Cu5	REST	JavaScript y Express	1	1x1x1x1x1	1	48	1 - 1 - 1 - 1 - 1
Cu5	REST	JavaScript y Express	1	1x1x1x1x1	2	16	1 - 1 - 1 - 1 - 1
Cu5	REST	JavaScript y Express	1	1x1x1x1x1	3	18	1 - 1 - 1 - 1 - 1
Cu5	REST	JavaScript y Express	8	2x2x1x2x1	1	49	2 - 4 - 4 - 8 - 8
Cu5	REST	JavaScript y Express	8	2x2x1x2x1	2	49	2 - 4 - 4 - 8 - 8
Cu5	REST	JavaScript y Express	8	2x2x1x2x1	3	53	2 - 4 - 4 - 8 - 8
Cu5	REST	JavaScript y Express	108	3x3x2x3x3	1	198	3 - 9 - 18 - 54 - 108
Cu5	REST	JavaScript y Express	108	3x3x2x3x3	2	207	3 - 9 - 18 - 54 - 108
Cu5	REST	JavaScript y Express	108	3x3x2x3x3	3	211	3 - 9 - 18 - 54 - 108
Cu5	REST	JavaScript y Express	1.024	4x4x4x4x4	1	773	4 - 16 - 64 - 256 - 1024

Cu5	REST	JavaScript y Express	1.024	4x4x4x4x4	2	781	4 - 16 - 64 - 256 - 1024
Cu5	REST	JavaScript y Express	1.024	4x4x4x4x4	3	736	4 - 16 - 64 - 256 - 1024
Cu5	REST	JavaScript y Express	10.584	6x6x7x6x6	1	3.698	6 - 36 - 252 - 1512 - 10584
Cu5	REST	JavaScript y Express	10.584	6x6x7x6x6	2	3.539	6 - 36 - 252 - 1512 - 10584
Cu5	REST	JavaScript y Express	10.584	6x6x7x6x6	3	3.625	6 - 36 - 252 - 1512 - 10584
Cu5	REST	JavaScript y Express	100.000	10x10x10x10x10	1	26.096	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	JavaScript y Express	100.000	10x10x10x10x10	2	27.088	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	JavaScript y Express	100.000	10x10x10x10x10	3	26.868	10 - 100 - 1000 - 10000 - 100000
Cu1	REST	Java y Spring	1	1	1	584	1
Cu1	REST	Java y Spring	1	1	2	7	1
Cu1	REST	Java y Spring	1	1	3	5	1
Cu1	REST	Java y Spring	10	10	1	6	10
Cu1	REST	Java y Spring	10	10	2	9	10
Cu1	REST	Java y Spring	10	10	3	5	10
Cu1	REST	Java y Spring	100	100	1	11	100
Cu1	REST	Java y Spring	100	100	2	5	100
Cu1	REST	Java y Spring	100	100	3	9	100
Cu1	REST	Java y Spring	1.000	1.000	1	74	1000
Cu1	REST	Java y Spring	1.000	1.000	2	33	1000
Cu1	REST	Java y Spring	1.000	1.000	3	42	1000
Cu1	REST	Java y Spring	10.000	10.000	1	84	10000
Cu1	REST	Java y Spring	10.000	10.000	2	89	10000
Cu1	REST	Java y Spring	10.000	10.000	3	75	10000
Cu1	REST	Java y Spring	100.000	100.000	1	463	100000

Cu1	REST	Java y Spring	100.000	100.000	2	398	100000
Cu1	REST	Java y Spring	100.000	100.000	3	434	100000
Cu2	REST	Java y Spring	1	1x1	1	40	1 - 1
Cu2	REST	Java y Spring	1	1x1	2	16	1 - 1
Cu2	REST	Java y Spring	1	1x1	3	18	1 - 1
Cu2	REST	Java y Spring	9	3x3	1	27	3 - 9
Cu2	REST	Java y Spring	9	3x3	2	25	3 - 9
Cu2	REST	Java y Spring	9	3x3	3	23	3 - 9
Cu2	REST	Java y Spring	100	10x10	1	98	10 - 100
Cu2	REST	Java y Spring	100	10x10	2	87	10 - 100
Cu2	REST	Java y Spring	100	10x10	3	88	10 - 100
Cu2	REST	Java y Spring	992	32x31	1	158	32 - 992
Cu2	REST	Java y Spring	992	32x31	2	172	32 - 992
Cu2	REST	Java y Spring	992	32x31	3	185	32 - 992
Cu2	REST	Java y Spring	10.000	100x100	1	551	100 - 10000
Cu2	REST	Java y Spring	10.000	100x100	2	398	100 - 10000
Cu2	REST	Java y Spring	10.000	100x100	3	340	100 - 10000
Cu2	REST	Java y Spring	99.856	316x316	1	1.882	316 - 99856
Cu2	REST	Java y Spring	99.856	316x316	2	1.794	316 - 99856
Cu2	REST	Java y Spring	99.856	316x316	3	1.602	316 - 99856
Cu3	REST	Java y Spring	1	1x1x1	1	19	1 - 1 - 1
Cu3	REST	Java y Spring	1	1x1x1	2	7	1 - 1 - 1
Cu3	REST	Java y Spring	1	1x1x1	3	8	1 - 1 - 1
Cu3	REST	Java y Spring	8	2x2x2	1	20	2 - 4 - 8
Cu3	REST	Java y Spring	8	2x2x2	2	18	2 - 4 - 8

Cu3	REST	Java y Spring	8	2x2x2	3	20	2 - 4 - 8
Cu3	REST	Java y Spring	100	5x5x4	1	93	5 - 25 - 100
Cu3	REST	Java y Spring	100	5x5x4	2	101	5 - 25 - 100
Cu3	REST	Java y Spring	100	5x5x4	3	101	5 - 25 - 100
Cu3	REST	Java y Spring	1.000	10x10x10	1	312	10 - 100 - 1000
Cu3	REST	Java y Spring	1.000	10x10x10	2	276	10 - 100 - 1000
Cu3	REST	Java y Spring	1.000	10x10x10	3	301	10 - 100 - 1000
Cu3	REST	Java y Spring	10.164	22x22x21	1	1.380	22 - 484 - 10164
Cu3	REST	Java y Spring	10.164	22x22x21	2	1.365	22 - 484 - 10164
Cu3	REST	Java y Spring	10.164	22x22x21	3	1.372	22 - 484 - 10164
Cu3	REST	Java y Spring	99.452	46x46x47	1	6.767	46 - 2116 - 99452
Cu3	REST	Java y Spring	99.452	46x46x47	2	6.488	46 - 2116 - 99452
Cu3	REST	Java y Spring	99.452	46x46x47	3	6.365	46 - 2116 - 99452
Cu4	REST	Java y Spring	1	1x1x1x1	1	23	1 - 1 - 1 - 1
Cu4	REST	Java y Spring	1	1x1x1x1	2	14	1 - 1 - 1 - 1
Cu4	REST	Java y Spring	1	1x1x1x1	3	14	1 - 1 - 1 - 1
Cu4	REST	Java y Spring	8	2x2x2x1	1	46	2 - 4 - 8 - 8
Cu4	REST	Java y Spring	8	2x2x2x1	2	38	2 - 4 - 8 - 8
Cu4	REST	Java y Spring	8	2x2x2x1	3	38	2 - 4 - 8 - 8
Cu4	REST	Java y Spring	108	3x3x3x4	1	105	3 - 9 - 27 - 108
Cu4	REST	Java y Spring	108	3x3x3x4	2	115	3 - 9 - 27 - 108
Cu4	REST	Java y Spring	108	3x3x3x4	3	98	3 - 9 - 27 - 108
Cu4	REST	Java y Spring	1.080	6x6x6x5	1	622	6 - 36 - 216 - 1080
Cu4	REST	Java y Spring	1.080	6x6x6x5	2	653	6 - 36 - 216 - 1080
Cu4	REST	Java y Spring	1.080	6x6x6x5	3	700	6 - 36 - 216 - 1080

Cu4	REST	Java y Spring	10.000	10x10x10x10	1	2.802	10 - 100 - 1000 - 10000
Cu4	REST	Java y Spring	10.000	10x10x10x10	2	2.703	10 - 100 - 1000 - 10000
Cu4	REST	Java y Spring	10.000	10x10x10x10	3	2.640	10 - 100 - 1000 - 10000
Cu4	REST	Java y Spring	99.144	18x18x18x17	1	18.227	18 - 324 - 5832 - 99144
Cu4	REST	Java y Spring	99.144	18x18x18x17	2	17.989	18 - 324 - 5832 - 99144
Cu4	REST	Java y Spring	99.144	18x18x18x17	3	17.748	18 - 324 - 5832 - 99144
Cu5	REST	Java y Spring	1	1x1x1x1x1	1	60	1 - 1 - 1 - 1 - 1
Cu5	REST	Java y Spring	1	1x1x1x1x1	2	21	1 - 1 - 1 - 1 - 1
Cu5	REST	Java y Spring	1	1x1x1x1x1	3	20	1 - 1 - 1 - 1 - 1
Cu5	REST	Java y Spring	8	2x2x1x2x1	1	61	2 - 4 - 4 - 8 - 8
Cu5	REST	Java y Spring	8	2x2x1x2x1	2	58	2 - 4 - 4 - 8 - 8
Cu5	REST	Java y Spring	8	2x2x1x2x1	3	49	2 - 4 - 4 - 8 - 8
Cu5	REST	Java y Spring	108	3x3x2x3x3	1	194	3 - 9 - 18 - 54 - 108
Cu5	REST	Java y Spring	108	3x3x2x3x3	2	196	3 - 9 - 18 - 54 - 108
Cu5	REST	Java y Spring	108	3x3x2x3x3	3	222	3 - 9 - 18 - 54 - 108
Cu5	REST	Java y Spring	1.024	4x4x4x4x4	1	861	4 - 16 - 64 - 256 - 1024
Cu5	REST	Java y Spring	1.024	4x4x4x4x4	2	808	4 - 16 - 64 - 256 - 1024
Cu5	REST	Java y Spring	1.024	4x4x4x4x4	3	875	4 - 16 - 64 - 256 - 1024
Cu5	REST	Java y Spring	10.584	6x6x7x6x6	1	4.263	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Java y Spring	10.584	6x6x7x6x6	2	3.915	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Java y Spring	10.584	6x6x7x6x6	3	4.175	6 - 36 - 252 - 1512 - 10584
Cu5	REST	Java y Spring	100.000	10x10x10x10x10	1	30.672	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	Java y Spring	100.000	10x10x10x10x10	2	29.899	10 - 100 - 1000 - 10000 - 100000
Cu5	REST	Java y Spring	100.000	10x10x10x10x10	3	30.438	10 - 100 - 1000 - 10000 - 100000

Tiempos de Respuesta – GraphQL

ID Caso de Uso	Arquitectura	Tecnología	Número de Registros	Distribución	Número de Iteración	Tiempo (ms)	Registro de Iteraciones
Cu1	GraphQL	Go y Gin	1	1	1	36	1
Cu1	GraphQL	Go y Gin	1	1	2	10	1
Cu1	GraphQL	Go y Gin	1	1	3	5	1
Cu1	GraphQL	Go y Gin	10	10	1	12	10
Cu1	GraphQL	Go y Gin	10	10	2	11	10
Cu1	GraphQL	Go y Gin	10	10	3	6	10
Cu1	GraphQL	Go y Gin	100	100	1	8	100
Cu1	GraphQL	Go y Gin	100	100	2	11	100
Cu1	GraphQL	Go y Gin	100	100	3	7	100
Cu1	GraphQL	Go y Gin	1000	1000	1	73	1000
Cu1	GraphQL	Go y Gin	1000	1000	2	81	1000
Cu1	GraphQL	Go y Gin	1000	1000	3	65	1000
Cu1	GraphQL	Go y Gin	10000	10000	1	118	10000
Cu1	GraphQL	Go y Gin	10000	10000	2	116	10000
Cu1	GraphQL	Go y Gin	10000	10000	3	90	10000
Cu1	GraphQL	Go y Gin	100000	100000	1	1060	100000
Cu1	GraphQL	Go y Gin	100000	100000	2	600	100000
Cu1	GraphQL	Go y Gin	100000	100000	3	591	100000
Cu2	GraphQL	Go y Gin	1	1x1	1	9	1 - 1
Cu2	GraphQL	Go y Gin	1	1x1	2	10	1 - 1
Cu2	GraphQL	Go y Gin	1	1x1	3	11	1 - 1
Cu2	GraphQL	Go y Gin	9	3x3	1	16	3 - 9
Cu2	GraphQL	Go y Gin	9	3x3	2	17	3 - 9
Cu2	GraphQL	Go y Gin	9	3x3	3	16	3 - 9
Cu2	GraphQL	Go y Gin	100	10x10	1	45	10 - 100
Cu2	GraphQL	Go y Gin	100	10x10	2	35	10 - 100
Cu2	GraphQL	Go y Gin	100	10x10	3	36	10 - 100
Cu2	GraphQL	Go y Gin	992	32x31	1	112	32 - 992
Cu2	GraphQL	Go y Gin	992	32x31	2	105	32 - 992
Cu2	GraphQL	Go y Gin	992	32x31	3	94	32 - 992
Cu2	GraphQL	Go y Gin	10000	100x100	1	373	100 - 10000
Cu2	GraphQL	Go y Gin	10000	100x100	2	385	100 - 10000
Cu2	GraphQL	Go y Gin	10000	100x100	3	341	100 - 10000
Cu2	GraphQL	Go y Gin	99856	316x316	1	1921	316 - 99856
Cu2	GraphQL	Go y Gin	99856	316x316	2	1618	316 - 99856
Cu2	GraphQL	Go y Gin	99856	316x316	3	1629	316 - 99856
Cu3	GraphQL	Go y Gin	1	1x1x1	1	14	1 - 1 - 1
Cu3	GraphQL	Go y Gin	1	1x1x1	2	11	1 - 1 - 1
Cu3	GraphQL	Go y Gin	1	1x1x1	3	9	1 - 1 - 1

Cu3	GraphQL	Go y Gin	8	2x2x2	1	26	2 - 4 - 8
Cu3	GraphQL	Go y Gin	8	2x2x2	2	20	2 - 4 - 8
Cu3	GraphQL	Go y Gin	8	2x2x2	3	20	2 - 4 - 8
Cu3	GraphQL	Go y Gin	100	5x5x4	1	98	5 - 25 - 100
Cu3	GraphQL	Go y Gin	100	5x5x4	2	90	5 - 25 - 100
Cu3	GraphQL	Go y Gin	100	5x5x4	3	92	5 - 25 - 100
Cu3	GraphQL	Go y Gin	1000	10x10x10	1	326	10 - 100 - 1000
Cu3	GraphQL	Go y Gin	1000	10x10x10	2	416	10 - 100 - 1000
Cu3	GraphQL	Go y Gin	1000	10x10x10	3	287	10 - 100 - 1000
Cu3	GraphQL	Go y Gin	10164	22x22x21	1	1605	22 - 484 - 10164
Cu3	GraphQL	Go y Gin	10164	22x22x21	2	1371	22 - 484 - 10164
Cu3	GraphQL	Go y Gin	10164	22x22x21	3	1390	22 - 484 - 10164
Cu3	GraphQL	Go y Gin	99452	46x46x47	1	7356	46 - 2116 - 99452
Cu3	GraphQL	Go y Gin	99452	46x46x47	2	7110	46 - 2116 - 99452
Cu3	GraphQL	Go y Gin	99452	46x46x47	3	6839	46 - 2116 - 99452
Cu4	GraphQL	Go y Gin	1	1x1x1x1	1	17	1 - 1 - 1 - 1
Cu4	GraphQL	Go y Gin	1	1x1x1x1	2	15	1 - 1 - 1 - 1
Cu4	GraphQL	Go y Gin	1	1x1x1x1	3	13	1 - 1 - 1 - 1
Cu4	GraphQL	Go y Gin	8	2x2x2x1	1	77	2 - 4 - 8 - 8
Cu4	GraphQL	Go y Gin	8	2x2x2x1	2	45	2 - 4 - 8 - 8
Cu4	GraphQL	Go y Gin	8	2x2x2x1	3	39	2 - 4 - 8 - 8
Cu4	GraphQL	Go y Gin	108	3x3x3x4	1	114	3 - 9 - 27 - 108
Cu4	GraphQL	Go y Gin	108	3x3x3x4	2	113	3 - 9 - 27 - 108
Cu4	GraphQL	Go y Gin	108	3x3x3x4	3	104	3 - 9 - 27 - 108
Cu4	GraphQL	Go y Gin	1080	6x6x6x5	1	636	6 - 36 - 216 - 1080
Cu4	GraphQL	Go y Gin	1080	6x6x6x5	2	687	6 - 36 - 216 - 1080
Cu4	GraphQL	Go y Gin	1080	6x6x6x5	3	557	6 - 36 - 216 - 1080
Cu4	GraphQL	Go y Gin	10000	10x10x10x10	1	2600	10 - 100 - 1000 - 10000
Cu4	GraphQL	Go y Gin	10000	10x10x10x10	2	2846	10 - 100 - 1000 - 10000
Cu4	GraphQL	Go y Gin	10000	10x10x10x10	3	2875	10 - 100 - 1000 - 10000

Cu4	GraphQL	Go y Gin	99144	18x18x18x17	1	18717	18 - 324 - 5832 - 99144
Cu4	GraphQL	Go y Gin	99144	18x18x18x17	2	18959	18 - 324 - 5832 - 99144
Cu4	GraphQL	Go y Gin	99144	18x18x18x17	3	20557	18 - 324 - 5832 - 99144
Cu5	GraphQL	Go y Gin	1	1x1x1x1x1	1	19	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	Go y Gin	1	1x1x1x1x1	2	17	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	Go y Gin	1	1x1x1x1x1	3	19	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	Go y Gin	8	2x2x1x2x1	1	84	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	Go y Gin	8	2x2x1x2x1	2	67	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	Go y Gin	8	2x2x1x2x1	3	68	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	Go y Gin	108	3x3x2x3x3	1	228	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	Go y Gin	108	3x3x2x3x3	2	194	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	Go y Gin	108	3x3x2x3x3	3	204	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	Go y Gin	1024	4x4x4x4x4	1	785	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	Go y Gin	1024	4x4x4x4x4	2	865	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	Go y Gin	1024	4x4x4x4x4	3	817	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	Go y Gin	10584	6x6x7x6x6	1	4632	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	Go y Gin	10584	6x6x7x6x6	2	4265	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	Go y Gin	10584	6x6x7x6x6	3	4156	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	Go y Gin	100000	10x10x10x10x10	1	33526	10 - 100 - 1000 - 10000 - 100000
Cu5	GraphQL	Go y Gin	100000	10x10x10x10x10	2	36492	10 - 100 - 1000 - 10000 - 100000
Cu5	GraphQL	Go y Gin	100000	10x10x10x10x10	3	29972	10 - 100 - 1000 - 10000 - 100000
Cu1	GraphQL	Python y Flask	1	1	1	222	1
Cu1	GraphQL	Python y Flask	1	1	2	73	1
Cu1	GraphQL	Python y Flask	1	1	3	70	1

Cu1	GraphQL	Python y Flask	10	10	1	161	10
Cu1	GraphQL	Python y Flask	10	10	2	68	10
Cu1	GraphQL	Python y Flask	10	10	3	69	10
Cu1	GraphQL	Python y Flask	100	100	1	65	100
Cu1	GraphQL	Python y Flask	100	100	2	64	100
Cu1	GraphQL	Python y Flask	100	100	3	67	100
Cu1	GraphQL	Python y Flask	1000	1000	1	149	1000
Cu1	GraphQL	Python y Flask	1000	1000	2	106	1000
Cu1	GraphQL	Python y Flask	1000	1000	3	112	1000
Cu1	GraphQL	Python y Flask	10000	10000	1	364	10000
Cu1	GraphQL	Python y Flask	10000	10000	2	383	10000
Cu1	GraphQL	Python y Flask	10000	10000	3	331	10000
Cu1	GraphQL	Python y Flask	100000	100000	1	2580	100000
Cu1	GraphQL	Python y Flask	100000	100000	2	2465	100000
Cu1	GraphQL	Python y Flask	100000	100000	3	2647	100000
Cu2	GraphQL	Python y Flask	1	1x1	1	127	1 - 1
Cu2	GraphQL	Python y Flask	1	1x1	2	139	1 - 1
Cu2	GraphQL	Python y Flask	1	1x1	3	141	1 - 1
Cu2	GraphQL	Python y Flask	9	3x3	1	252	3 - 9
Cu2	GraphQL	Python y Flask	9	3x3	2	232	3 - 9
Cu2	GraphQL	Python y Flask	9	3x3	3	246	3 - 9
Cu2	GraphQL	Python y Flask	100	10x10	1	1079	10 - 100
Cu2	GraphQL	Python y Flask	100	10x10	2	742	10 - 100
Cu2	GraphQL	Python y Flask	100	10x10	3	1000	10 - 100
Cu2	GraphQL	Python y Flask	992	32x31	1	2596	32 - 992

Cu2	GraphQL	Python y Flask	992	32x31	2	2644	32 - 992
Cu2	GraphQL	Python y Flask	992	32x31	3	2677	32 - 992
Cu2	GraphQL	Python y Flask	10000	100x100	1	7994	100 - 10000
Cu2	GraphQL	Python y Flask	10000	100x100	2	7670	100 - 10000
Cu2	GraphQL	Python y Flask	10000	100x100	3	7297	100 - 10000
Cu2	GraphQL	Python y Flask	99856	316x316	1	26295	316 - 99856
Cu2	GraphQL	Python y Flask	99856	316x316	2	28703	316 - 99856
Cu2	GraphQL	Python y Flask	99856	316x316	3	27430	316 - 99856
Cu3	GraphQL	Python y Flask	1	1x1x1	1	194	1 - 1 - 1
Cu3	GraphQL	Python y Flask	1	1x1x1	2	188	1 - 1 - 1
Cu3	GraphQL	Python y Flask	1	1x1x1	3	192	1 - 1 - 1
Cu3	GraphQL	Python y Flask	8	2x2x2	1	565	2 - 4 - 8
Cu3	GraphQL	Python y Flask	8	2x2x2	2	530	2 - 4 - 8
Cu3	GraphQL	Python y Flask	8	2x2x2	3	439	2 - 4 - 8
Cu3	GraphQL	Python y Flask	100	5x5x4	1	2150	5 - 25 - 100
Cu3	GraphQL	Python y Flask	100	5x5x4	2	2193	5 - 25 - 100
Cu3	GraphQL	Python y Flask	100	5x5x4	3	2241	5 - 25 - 100
Cu3	GraphQL	Python y Flask	1000	10x10x10	1	7585	10 - 100 - 1000
Cu3	GraphQL	Python y Flask	1000	10x10x10	2	8799	10 - 100 - 1000
Cu3	GraphQL	Python y Flask	1000	10x10x10	3	7848	10 - 100 - 1000
Cu3	GraphQL	Python y Flask	10164	22x22x21	1	38602	22 - 484 - 10164
Cu3	GraphQL	Python y Flask	10164	22x22x21	2	38573	22 - 484 - 10164
Cu3	GraphQL	Python y Flask	10164	22x22x21	3	36216	22 - 484 - 10164
Cu3	GraphQL	Python y Flask	99452	46x46x47	1	160176	46 - 2116 - 99452
Cu3	GraphQL	Python y Flask	99452	46x46x47	2	159876	46 - 2116 - 99452

Cu3	GraphQL	Python y Flask	99452	46x46x47	3	160209	46 - 2116 - 99452
Cu4	GraphQL	Python y Flask	1	1x1x1x1	1	253	1 - 1 - 1 - 1
Cu4	GraphQL	Python y Flask	1	1x1x1x1	2	259	1 - 1 - 1 - 1
Cu4	GraphQL	Python y Flask	1	1x1x1x1	3	252	1 - 1 - 1 - 1
Cu4	GraphQL	Python y Flask	8	2x2x2x1	1	1047	2 - 4 - 8 - 8
Cu4	GraphQL	Python y Flask	8	2x2x2x1	2	1000	2 - 4 - 8 - 8
Cu4	GraphQL	Python y Flask	8	2x2x2x1	3	1223	2 - 4 - 8 - 8
Cu4	GraphQL	Python y Flask	108	3x3x3x4	1	3125	3 - 9 - 27 - 108
Cu4	GraphQL	Python y Flask	108	3x3x3x4	2	2866	3 - 9 - 27 - 108
Cu4	GraphQL	Python y Flask	108	3x3x3x4	3	3138	3 - 9 - 27 - 108
Cu4	GraphQL	Python y Flask	1080	6x6x6x5	1	20293	6 - 36 - 216 - 1080
Cu4	GraphQL	Python y Flask	1080	6x6x6x5	2	19368	6 - 36 - 216 - 1080
Cu4	GraphQL	Python y Flask	1080	6x6x6x5	3	19688	6 - 36 - 216 - 1080
Cu4	GraphQL	Python y Flask	10000	10x10x10x10	1	83232	10 - 100 - 1000 - 10000
Cu4	GraphQL	Python y Flask	10000	10x10x10x10	2	79782	10 - 100 - 1000 - 10000
Cu4	GraphQL	Python y Flask	10000	10x10x10x10	3	82158	10 - 100 - 1000 - 10000
Cu4	GraphQL	Python y Flask	99144	18x18x18x17	1	456863	18 - 324 - 5832 - 99144
Cu4	GraphQL	Python y Flask	99144	18x18x18x17	2	454049	18 - 324 - 5832 - 99144
Cu4	GraphQL	Python y Flask	99144	18x18x18x17	3	459787	18 - 324 - 5832 - 99144
Cu5	GraphQL	Python y Flask	1	1x1x1x1x1	1	531	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	Python y Flask	1	1x1x1x1x1	2	469	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	Python y Flask	1	1x1x1x1x1	3	497	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	Python y Flask	8	2x2x1x2x2	1	2075	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	Python y Flask	8	2x2x1x2x2	2	1874	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	Python y Flask	8	2x2x1x2x2	3	2142	2 - 4 - 4 - 8 - 8

Cu5	GraphQL	Python y Flask	108	3x3x2x3x3	1	8713	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	Python y Flask	108	3x3x2x3x3	2	9290	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	Python y Flask	108	3x3x2x3x3	3	8339	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	Python y Flask	1024	4x4x4x4x4	1	32615	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	Python y Flask	1024	4x4x4x4x4	2	33022	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	Python y Flask	1024	4x4x4x4x4	3	32781	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	Python y Flask	10584	6x6x7x6x6	1	179618	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	Python y Flask	10584	6x6x7x6x6	2	192078	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	Python y Flask	10584	6x6x7x6x6	3	192072	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	Python y Flask	100000	10x10x10x10x10	1	1153893	10 - 100 - 1000 - 10000 - 100000
Cu5	GraphQL	Python y Flask	100000	10x10x10x10x10	2	1151200	10 - 100 - 1000 - 10000 - 100000
Cu5	GraphQL	Python y Flask	100000	10x10x10x10x10	3	1125424	10 - 100 - 1000 - 10000 - 100000
Cu1	GraphQL	JavaScript y Express	1	1	1	99	1
Cu1	GraphQL	JavaScript y Express	1	1	2	8	1
Cu1	GraphQL	JavaScript y Express	1	1	3	5	1
Cu1	GraphQL	JavaScript y Express	10	10	1	5	10
Cu1	GraphQL	JavaScript y Express	10	10	2	5	10
Cu1	GraphQL	JavaScript y Express	10	10	3	5	10
Cu1	GraphQL	JavaScript y Express	100	100	1	8	100
Cu1	GraphQL	JavaScript y Express	100	100	2	10	100
Cu1	GraphQL	JavaScript y Express	100	100	3	10	100
Cu1	GraphQL	JavaScript y Express	1000	1000	1	89	1000
Cu1	GraphQL	JavaScript y Express	1000	1000	2	38	1000
Cu1	GraphQL	JavaScript y Express	1000	1000	3	39	1000

Cu1	GraphQL	JavaScript y Express	10000	10000	1	141	10000
Cu1	GraphQL	JavaScript y Express	10000	10000	2	128	10000
Cu1	GraphQL	JavaScript y Express	10000	10000	3	125	10000
Cu1	GraphQL	JavaScript y Express	100000	100000	1	964	100000
Cu1	GraphQL	JavaScript y Express	100000	100000	2	878	100000
Cu1	GraphQL	JavaScript y Express	100000	100000	3	847	100000
Cu2	GraphQL	JavaScript y Express	1	1x1	1	12	1 - 1
Cu2	GraphQL	JavaScript y Express	1	1x1	2	11	1 - 1
Cu2	GraphQL	JavaScript y Express	1	1x1	3	10	1 - 1
Cu2	GraphQL	JavaScript y Express	9	3x3	1	18	3 - 9
Cu2	GraphQL	JavaScript y Express	9	3x3	2	18	3 - 9
Cu2	GraphQL	JavaScript y Express	9	3x3	3	20	3 - 9
Cu2	GraphQL	JavaScript y Express	100	10x10	1	47	10 - 100
Cu2	GraphQL	JavaScript y Express	100	10x10	2	50	10 - 100
Cu2	GraphQL	JavaScript y Express	100	10x10	3	49	10 - 100
Cu2	GraphQL	JavaScript y Express	992	32x31	1	127	32 - 992
Cu2	GraphQL	JavaScript y Express	992	32x31	2	119	32 - 992
Cu2	GraphQL	JavaScript y Express	992	32x31	3	137	32 - 992
Cu2	GraphQL	JavaScript y Express	10000	100x100	1	416	100 - 10000
Cu2	GraphQL	JavaScript y Express	10000	100x100	2	442	100 - 10000
Cu2	GraphQL	JavaScript y Express	10000	100x100	3	422	100 - 10000
Cu2	GraphQL	JavaScript y Express	99856	316x316	1	2190	316 - 99856
Cu2	GraphQL	JavaScript y Express	99856	316x316	2	2089	316 - 99856
Cu2	GraphQL	JavaScript y Express	99856	316x316	3	2113	316 - 99856
Cu3	GraphQL	JavaScript y Express	1	1x1x1	1	16	1 - 1 - 1

Cu3	GraphQL	JavaScript y Express	1	1x1x1	2	13	1 - 1 - 1
Cu3	GraphQL	JavaScript y Express	1	1x1x1	3	11	1 - 1 - 1
Cu3	GraphQL	JavaScript y Express	8	2x2x2	1	33	2 - 4 - 8
Cu3	GraphQL	JavaScript y Express	8	2x2x2	2	28	2 - 4 - 8
Cu3	GraphQL	JavaScript y Express	8	2x2x2	3	26	2 - 4 - 8
Cu3	GraphQL	JavaScript y Express	100	5x5x4	1	104	5 - 25 - 100
Cu3	GraphQL	JavaScript y Express	100	5x5x4	2	114	5 - 25 - 100
Cu3	GraphQL	JavaScript y Express	100	5x5x4	3	115	5 - 25 - 100
Cu3	GraphQL	JavaScript y Express	1000	10x10x10	1	397	10 - 100 - 1000
Cu3	GraphQL	JavaScript y Express	1000	10x10x10	2	390	10 - 100 - 1000
Cu3	GraphQL	JavaScript y Express	1000	10x10x10	3	370	10 - 100 - 1000
Cu3	GraphQL	JavaScript y Express	10164	22x22x21	1	1735	22 - 484 - 10164
Cu3	GraphQL	JavaScript y Express	10164	22x22x21	2	1724	22 - 484 - 10164
Cu3	GraphQL	JavaScript y Express	10164	22x22x21	3	1659	22 - 484 - 10164
Cu3	GraphQL	JavaScript y Express	99452	46x46x47	1	8171	46 - 2116 - 99452
Cu3	GraphQL	JavaScript y Express	99452	46x46x47	2	7831	46 - 2116 - 99452
Cu3	GraphQL	JavaScript y Express	99452	46x46x47	3	7429	46 - 2116 - 99452
Cu4	GraphQL	JavaScript y Express	1	1x1x1x1	1	19	1 - 1 - 1 - 1
Cu4	GraphQL	JavaScript y Express	1	1x1x1x1	2	14	1 - 1 - 1 - 1
Cu4	GraphQL	JavaScript y Express	1	1x1x1x1	3	13	1 - 1 - 1 - 1
Cu4	GraphQL	JavaScript y Express	8	2x2x2x1	1	51	2 - 4 - 8 - 8
Cu4	GraphQL	JavaScript y Express	8	2x2x2x1	2	46	2 - 4 - 8 - 8
Cu4	GraphQL	JavaScript y Express	8	2x2x2x1	3	49	2 - 4 - 8 - 8
Cu4	GraphQL	JavaScript y Express	108	3x3x3x4	1	125	3 - 9 - 27 - 108
Cu4	GraphQL	JavaScript y Express	108	3x3x3x4	2	113	3 - 9 - 27 - 108

Cu4	GraphQL	JavaScript y Express	108	3x3x3x4	3	118	3 - 9 - 27 - 108
Cu4	GraphQL	JavaScript y Express	1080	6x6x6x5	1	750	6 - 36 - 216 - 1080
Cu4	GraphQL	JavaScript y Express	1080	6x6x6x5	2	715	6 - 36 - 216 - 1080
Cu4	GraphQL	JavaScript y Express	1080	6x6x6x5	3	703	6 - 36 - 216 - 1080
Cu4	GraphQL	JavaScript y Express	10000	10x10x10x10	1	3008	10 - 100 - 1000 - 10000
Cu4	GraphQL	JavaScript y Express	10000	10x10x10x10	2	2903	10 - 100 - 1000 - 10000
Cu4	GraphQL	JavaScript y Express	10000	10x10x10x10	3	3044	10 - 100 - 1000 - 10000
Cu4	GraphQL	JavaScript y Express	99144	18x18x18x17	1	19794	18 - 324 - 5832 - 99144
Cu4	GraphQL	JavaScript y Express	99144	18x18x18x17	2	19658	18 - 324 - 5832 - 99144
Cu4	GraphQL	JavaScript y Express	99144	18x18x18x17	3	19069	18 - 324 - 5832 - 99144
Cu5	GraphQL	JavaScript y Express	1	1x1x1x1x1	1	41	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	JavaScript y Express	1	1x1x1x1x1	2	16	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	JavaScript y Express	1	1x1x1x1x1	3	18	1 - 1 - 1 - 1 - 1
Cu5	GraphQL	JavaScript y Express	8	2x2x1x2x2	1	65	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	JavaScript y Express	8	2x2x1x2x2	2	58	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	JavaScript y Express	8	2x2x1x2x2	3	62	2 - 4 - 4 - 8 - 8
Cu5	GraphQL	JavaScript y Express	108	3x3x2x3x3	1	215	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	JavaScript y Express	108	3x3x2x3x3	2	249	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	JavaScript y Express	108	3x3x2x3x3	3	235	3 - 9 - 18 - 54 - 108
Cu5	GraphQL	JavaScript y Express	1024	4x4x4x4x4	1	889	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	JavaScript y Express	1024	4x4x4x4x4	2	855	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	JavaScript y Express	1024	4x4x4x4x4	3	846	4 - 16 - 64 - 256 - 1024
Cu5	GraphQL	JavaScript y Express	10584	6x6x7x6x6	1	4692	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	JavaScript y Express	10584	6x6x7x6x6	2	4800	6 - 36 - 252 - 1512 - 10584
Cu5	GraphQL	JavaScript y Express	10584	6x6x7x6x6	3	4726	6 - 36 - 252 - 1512 - 10584

Cu5	GraphQL	JavaScript y Express	100000	10x10x10x10x10	1	32589	10 - 100 - 1000 - 10000 - 100000
Cu5	GraphQL	JavaScript y Express	100000	10x10x10x10x10	2	31948	10 - 100 - 1000 - 10000 - 100000
Cu5	GraphQL	JavaScript y Express	100000	10x10x10x10x10	3	33132	10 - 100 - 1000 - 10000 - 100000