



UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CARRERA DE SOFTWARE

TRABAJO DE INTEGRACIÓN CURRICULAR

TEMA:

“COMPARACIÓN DE EFICIENCIA DE TECNOLOGÍAS FRONTEND EN EL CONSUMO DE DATOS DE APIS CON ARQUITECTURAS REST Y GRAPHQL, UTILIZANDO LA MÉTRICA DE TIEMPO DE RESPUESTA DE LA ISO/IEC 25023”

Trabajo de titulación previo a la obtención del título en Ingeniería de Software

Línea de investigación: Desarrollo, aplicación de software y cyber security (seguridad cibernética)

AUTOR:

Johann Alejandro Quishpe Bolaños

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:	0401726096		
APELLIDOS Y NOMBRES:	QUISHPE BOLAÑOS JOHANN ALEJANDRO		
DIRECCIÓN:	La Victoria – Alfredo Albuja Galindo y pasaje h		
EMAIL:	jaquishpeb@utn.edu.ec		
TELÉFONO FIJO:	062985324	TELÉFONO MÓVIL:	0978689547

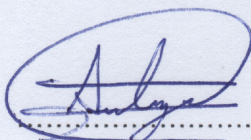
DATOS DE LA OBRA	
TÍTULO:	Comparación de eficiencia de tecnologías Frontend en el consumo de datos de APIs con arquitecturas REST y GraphQL, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023.
AUTOR (ES):	QUISHPE BOLAÑOS JOHANN ALEJANDRO
FECHA: DD/MM/AAAA	04/09/2025
PROGRAMA:	☒ PREGRADO ☐ POSGRADO
TÍTULO POR EL QUE OPTA:	INGENIERÍA EN SOFTWARE
ASESOR /DIRECTOR:	PhD. JOSÉ ANTONIO QUIÑA MERA

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 04 días del mes de septiembre de 2025

EL AUTOR:



.....

Nombre: Johann Alejandro Quishpe Bolaños

CERTIFICADO DEL DIRECTOR DEL TRABAJO DE GRADO



**UNIVERSIDAD TÉCNICA DEL NORTE
FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS**

CERTIFICACIÓN DEL DIRECTOR

Certifico que la Tesis previa a la obtención del título de Ingeniería en Software con el tema: "Comparación de eficiencia de tecnologías Frontend en el consumo de datos de APIs con arquitecturas REST y GraphQL, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023." ha sido desarrollada y terminada en su totalidad por el Sr. Johann Alejandro Quishpe Bolaños, con cédula de identidad Nro. 0401726096 bajo mi supervisión para lo cual firmo mi constancia.

A handwritten signature in blue ink, appearing to read "Antonio Quiña", is written over a horizontal line.

PhD. Antonio Quiña
DIRECTOR DE TESIS

DEDICATORIA

A mi madre y mi abuelita.

Dalia Victua Bolaños Fuel, Gloria Esperanza Fuel Rosero

“Todo lo que soy, y todo lo que espero ser, se lo debo a mi madre.
Su fe en mí fue el primer impulso que me llevó a creer en mí mismo,
incluso cuando el mundo parecía dudar.”

Abraham Lincoln

Johann Alejandro Quishpe Bolaños

AGRADECIMIENTOS

En primer lugar, agradezco a Dios, por darme la vida, la salud y la fortaleza necesaria para seguir adelante en cada momento. Gracias por iluminar mi camino, darme esperanza en la incertidumbre y acompañarme en cada paso de este proceso.

A mi madre, **Dalia Bolaños**, por enseñarme con su ternura, su sabiduría y su fe inquebrantable que los sueños se alcanzan con esfuerzo y corazón.

A mi padre, **Mauricio Quishpe**, por su apoyo y amor, cuyo ejemplo de constancia, esfuerzo y trabajo me inspira a nunca rendirme y a luchar con determinación por lo que deseo.

A mi hermana, **Sofía Quishpe**, gracias por estar siempre a mi lado, por las risas y los chistes que me alegraron en los días difíciles, por darme felicidad en los momentos más grises y por demostrarme un cariño sincero. Tú eres una parte muy importante de mi vida, y valoro profundamente todo lo que compartimos.

También quiero agradecer a mi abuelita, **Gloria Fuel**, quien ha sido como una segunda madre. Su amor incondicional, sus palabras llenas de sabiduría y su abrazo cálido han sido mi refugio constante, recordándome siempre de dónde vengo y lo que realmente importa.

En los momentos en que dudaba, apareció mi mejor amiga, **Carolayne Mejía**, con su energía y alegría, mostrándome el lado positivo de la vida. Su amistad ha sido un regalo invaluable.

No puedo dejar de mencionar a mis amigos **Esteban Revelo** y **Cristian Alemán**, por acompañarme en cada etapa, compartiendo tanto los logros como los desafíos. Gracias por su lealtad, por las bromas que aliviaron las tensiones y por hacer de este camino algo más ameno y divertido.

Finalmente, extendiendo mi gratitud a todos mis demás amigos y familiares que, en diferentes momentos, me brindaron apoyo, ánimo o una palabra de aliento. Cada gesto, grande o pequeño, fue fundamental para llegar hasta aquí.

Cada uno de ustedes ha dejado una huella en esta tesis. No es solo un logro académico, es una historia compartida, tejida con amor, paciencia y amistad. Gracias por caminar conmigo.

TABLA DE CONTENIDOS

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE	2
CERTIFICADO DEL DIRECTOR DEL TRABAJO DE GRADO	4
DEDICATORIA	5
AGRADECIMIENTOS.....	6
RESUMEN.....	13
ABSTRACT	14
INTRODUCCIÓN.....	15
Antecedentes	15
Situación Actual	15
Planteamiento del problema.....	16
Objetivos	17
Objetivo General.....	17
Objetivos Específicos	17
Alcance	17
Metodología	19
Justificación	19
CAPITULO I.....	1
1.1 Fundamentos de tecnologías Frontend.....	1
1.1.1 Evolución del desarrollo frontend	2

1.1.2	Importancia del rendimiento en aplicaciones frontend.....	3
1.2	Arquitecturas REST y GraphQL	4
1.2.1	REST	5
1.2.2	GraphQL.....	7
1.2.3	Comparación teórica entre REST y GraphQL.....	8
1.3	Selección de tecnologías Frontend	10
1.3.1	Definición del mapeo sistemático	10
1.3.2	Identificación de estudios	12
1.3.3	Extracción y análisis de datos.....	13
1.4	ISO/IEC 25023	16
1.4.1	ISO/IEC 25023 métrica de tiempo de respuesta.....	18
CAPITULO II.....		19
2.1	Planificación	19
2.1.1	Definición del experimento computacional	20
2.1.2.	Justificación de los Casos de uso.....	20
2.1.3	Diseño de consultas y estructura de datos	23
2.1.4	Entorno de Pruebas y Configuración Técnica	24
2.2	Desarrollo.....	28
2.2.1.	Flask.....	29
2.2.2	React.....	32
2.2.3	Angular	37

2.3	Obtención de resultados	41
CAPITULO III.....		50
3.1	Análisis de resultados	50
3.1.1	Comparación por tecnología y por tipo de API.	50
3.1.2	¿Qué tecnología se comportó mejor?	59
3.2	Discusión	63
3.2.1	Problemas técnicos	63
3.2.2	Limitaciones del entorno	64
CONCLUSIONES Y RECOMENDACIONES		65
Conclusiones		65
Recomendaciones		67
Bibliografía.....		68

ÍNDICE DE ILUSTRACIONES

Ilustración 1 Árbol de problemas.....	16
Ilustración 2 Descripción del experimento.....	18
Ilustración 3 Proceso de mapeo sistemático	11
Ilustración 4 Distribución de los datos obtenidos	12
Ilustración 5 Complejidad progresiva de los Casos de Uso	22
Ilustración 6 Instalación de React	26
Ilustración 7 Instalación de Angular	27
Ilustración 8 Instalación de Flask	28
Ilustración 9 Distribución de carpetas del proyecto - Flask	29
Ilustración 10 Conexión API - REST	30
Ilustración 11 Conexión API - GraphQL	31
Ilustración 12 Fragmento de código - Consulta en cascada.....	31
Ilustración 13 Fragmento de código - Generar Excel	32
Ilustración 14 Distribución de carpetas del proyecto - React.....	33
Ilustración 15 Conexión - API REST	34
Ilustración 16 Conexión - API GraphQL	34
Ilustración 17 Fragmento de código - Consulta en cascada.....	35
Ilustración 18 Fragmento de código – Obtener resultados.....	36
Ilustración 19 Distribución de carpetas del proyecto - Angular	37
Ilustración 20 Conexión - API REST	38
Ilustración 21 Conexión - API GraphQL	39
Ilustración 22 Fragmento de código - Consulta en cascada.....	40
Ilustración 23 Fragmento de código – Obtener resultados.....	41
Ilustración 24 Flask - Caso de uso N°1	50

Ilustración 25 Flask - Caso de uso N°2	51
Ilustración 26 Flask - Caso de uso N°3	51
Ilustración 27 Flask - Caso de uso N°4	52
Ilustración 28 Flask - Caso de uso N°5	52
Ilustración 29 React - Caso de uso N°1	53
Ilustración 30 React - Caso de uso N°2	54
Ilustración 31 React - Caso de uso N°3	54
Ilustración 32 React - Caso de uso N°4	55
Ilustración 33 React - Caso de uso N°5	55
Ilustración 34 Angular - Caso de uso N°1	56
Ilustración 35 Angular - Caso de uso N°2	57
Ilustración 36 Angular - Caso de uso N°3	57
Ilustración 37 Angular - Caso de uso N°4	58
Ilustración 38 Angular - Caso de uso N°5	59
Ilustración 39 Promedio de tiempos REST	60
Ilustración 40 Promedio de tiempos GraphQL	61
Ilustración 41 Promedio de tiempos REST vs GraphQL	62

ÍNDICE DE TABLAS

Tabla 1 Operaciones REST	6
Tabla 2 REST vs GraphQL	9
Tabla 3 Número de documentos por base de datos	13
Tabla 4 Frecuencia de tecnologías frontend usadas en APIs GraphQL.....	14
Tabla 5 Frecuencia de tecnologías frontend usadas en APIs REST	14
Tabla 6 Características y Sub-características de la ISO/IEC 2023	17
Tabla 7 Promedio de tiempos REST	60
Tabla 8 Promedio de tiempos GraphQL	61
Tabla 10 Promedio de tiempos REST vs GraphQL	62

RESUMEN

La presente investigación tiene como finalidad comparar la eficiencia de diferentes tecnologías frontend en el consumo de datos desde APIs implementadas bajo arquitecturas REST y GraphQL, utilizando como referencia la métrica de tiempo de respuesta definida en la norma ISO/IEC 25023. El estudio surge de la necesidad de identificar qué combinación de arquitectura y tecnología resulta más eficiente en escenarios donde el rendimiento en el manejo de datos es un factor crítico para el desarrollo de aplicaciones modernas. Para ello, se llevó a cabo una revisión sistemática de literatura que permitió seleccionar tres tecnologías frontend representativas: React, Angular y Flask. Posteriormente, se diseñó un experimento computacional compuesto por cinco casos de uso con distintos niveles de complejidad en las consultas, los cuales fueron implementados y ejecutados bajo condiciones controladas que aseguraron la validez de los resultados obtenidos. Dicho experimento midió de forma cuantitativa los tiempos de respuesta de cada tecnología en combinación con ambas arquitecturas, permitiendo realizar una comparación objetiva y fundamentada. Los resultados evidencian que React con REST presentó el mejor desempeño, seguido por React con GraphQL, mientras que Angular mostró un rendimiento intermedio y Flask registró los tiempos más altos, reflejando menor eficiencia. De esta manera, se concluye que la elección de la arquitectura y del framework frontend influye directamente en la eficiencia de las aplicaciones, aportando información relevante para la toma de decisiones en proyectos orientados al consumo intensivo de datos.

Palabras clave: Frontend, comparativa, API, REST, GraphQL, ISO, experimento, tiempo de respuesta, eficiencia, arquitectura, consultas.

ABSTRACT

The purpose of this research is to compare the efficiency of different frontend technologies in data consumption from APIs implemented under REST and GraphQL architectures, using the response time metric defined in the ISO/IEC 25023 standard as a reference. The study arises from the need to identify which combination of architecture and technology is most efficient in scenarios where data management performance is a critical factor for the development of modern applications. To this end, a systematic literature review was conducted, which led to the selection of three representative frontend technologies: React, Angular, and Flask. Subsequently, a computational experiment was designed, consisting of five use cases with different levels of query complexity, which were implemented and executed under controlled conditions to ensure the validity of the results obtained. This experiment quantitatively measured the response times of each technology in combination with both architectures, allowing for an objective and informed comparison. This experiment quantitatively measured the response times of each technology in combination with both architectures, allowing for an objective and informed comparison. The results show that React with REST performed best, followed by React with GraphQL, while Angular showed intermediate performance and Flask recorded the highest times, reflecting lower efficiency. Thus, it can be concluded that the choice of architecture and frontend framework directly influences the efficiency of applications, providing relevant information for decision-making in projects oriented towards intensive data consumption.

Keywords: Frontend, comparison, API, REST, GraphQL, ISO, experiment, response time, efficiency, architecture, queries.

INTRODUCCIÓN

Antecedentes

Debido a su simplicidad, flexibilidad y facilidad de uso, las API REST se han convertido en un estándar para el desarrollo de aplicaciones desde su introducción en el año 2000. Sin embargo, con el aumento de las aplicaciones web y sus demandas de datos complejos, se han observado ciertas limitaciones en REST, como la complejidad de la estructura de intercambio de datos y los problemas de exceso o insuficiencia de datos en las respuestas a las solicitudes. (Eseme, 2023)

GraphQL fue creado por Facebook en 2015 para ofrecer una solución más adaptable y poderosa para la gestión de APIs. A diferencia de REST, GraphQL permite a los clientes especificar exactamente los datos que necesitan, evitando la acumulación de datos, lo cual puede ser esencial para la eficiencia del consumo de datos en aplicaciones web y móviles. (Eseme, 2023)

Situación Actual

En la actualidad, la investigación y la comparación de estas tecnologías son esenciales para guiar la selección de la arquitectura y tecnología Frontend más adecuada para el desarrollo de aplicaciones eficientes y de alto rendimiento debido a la evolución en el manejo de APIs y la necesidad de adaptarse a los requisitos de datos más complejos y cambiantes.

Planteamiento del problema

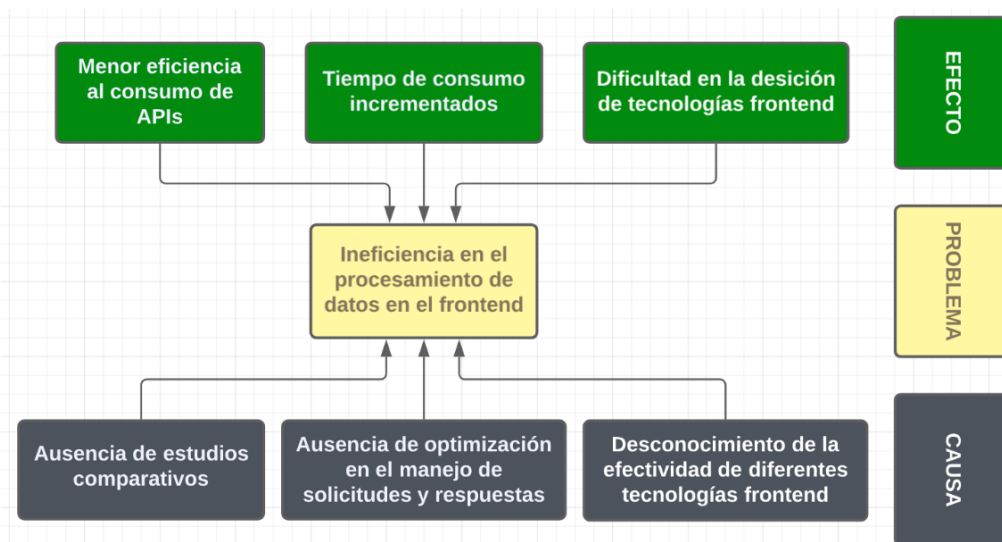
El desarrollo de aplicaciones web y el consumo de APIs se han vuelto fundamentales en el entorno tecnológico actual. Sin embargo, la selección de tecnologías Frontend y la elección de arquitecturas pueden tener un impacto significativo en la eficiencia y el rendimiento de estas aplicaciones. El problema radica en la falta de una evaluación que compare el rendimiento de consumo de datos de las tecnologías Frontend en APIs con las arquitecturas REST y GraphQL. (Guillen-Drija, Quintero, & Kleiman, 2018)

La carencia de esta evaluación hace que los desarrolladores y las organizaciones enfrenten dificultades para tomar decisiones en la selección de tecnologías y arquitecturas, lo que puede dar como resultado aplicaciones menos eficientes.

En los últimos años se ha observado un aumento en el uso de diferentes tecnologías Frontend en combinación con arquitecturas GraphQL y REST. De este modo, se plantea una necesidad de llevar a cabo una evaluación de distintas tecnologías Frontend que utilicen las arquitecturas mencionadas anteriormente. (Guillen-Drija, Quintero, & Kleiman, 2018)

Ilustración 1

Árbol de problemas



Objetivos

Objetivo General

Comparar la eficiencia de tecnologías Frontend en el consumo de datos de APIs con arquitecturas REST y GraphQL, 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 Frontend para el consumo de APIs con las arquitecturas REST y GraphQL.
- Desarrollar un experimento computacional para comparar la eficiencia del consumo de datos Frontend 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 determinar la tecnología y arquitectura más eficiente en el consumo de datos en el Frontend.

Alcance

La finalidad de la investigación consiste en llevar a cabo una comparativa del desempeño de las tecnologías Frontend en el ámbito del consumo de datos. Se utilizará Apis que utilizan arquitecturas GraphQL y REST, priorizando el tiempo de respuesta para determinar su eficacia en su consumo.

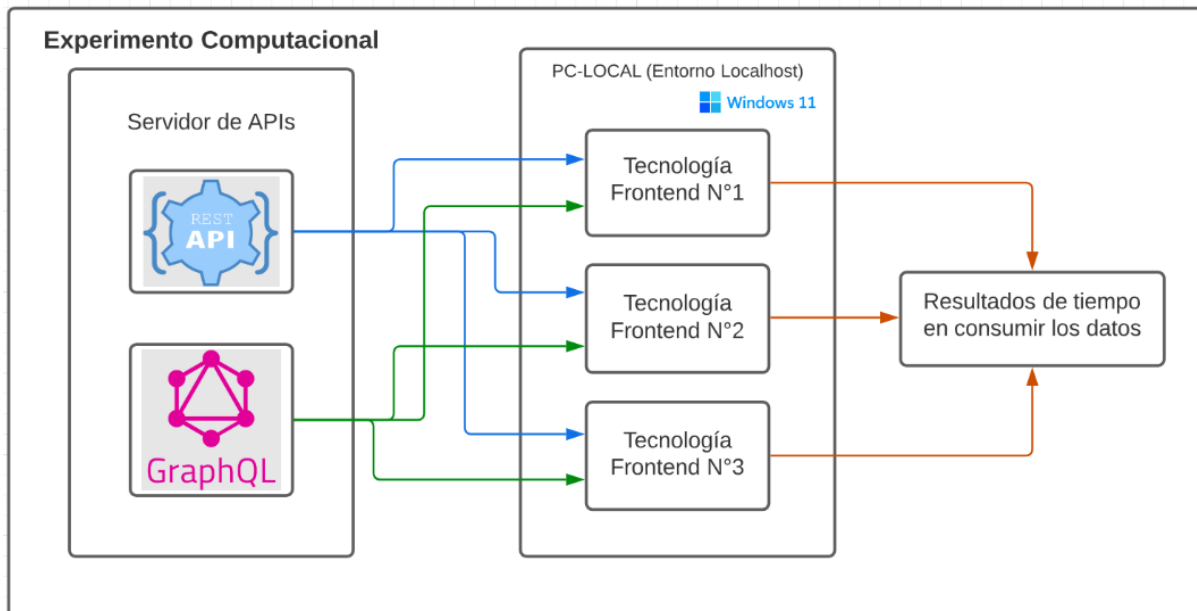
En la fase inicial, se llevará a cabo una revisión de la literatura técnica disponible, involucrando la exploración de documentación especializada y el análisis de estudios pertinentes. El propósito fundamental de esta etapa radica en la identificación de las tres tecnologías más preeminentes y ampliamente empleadas en el contexto del desarrollo Frontend, particularmente en relación con las arquitecturas GraphQL y REST. Este proceso de revisión documental proporcionará el trasfondo esencial necesario para fundamentar y

enmarcar la evaluación comparativa que sigue. Las tecnologías seleccionadas en esta fase servirán como la base sólida desde la cual se realizará la evaluación.

En la segunda fase, se procederá a realizar un experimento computacional como se visualiza en la figura 2 en el cual se evaluará la eficiencia de las tecnologías Frontend en el consumo de datos de APIs con arquitecturas REST y GraphQL. Este experimento se llevará a cabo siguiendo un enfoque cuantitativo, lo que implica la configuración y ejecución de pruebas específicas que permitirán comparar el desempeño de estas tecnologías. Con el propósito de obtener resultados empíricos que respalden la evaluación y que contribuyan de manera sustancial a la investigación. Además, es relevante señalar que las mediciones se llevarán a cabo de acuerdo con la métrica de tiempo de respuesta establecida por la norma ISO/IEC 25023. (iso.org, 2022)

Ilustración 2

Descripción del experimento



Metodología

Para cumplir el primer objetivo en esta investigación, se llevará a cabo una revisión tanto de fuentes primarias (revistas y artículos científicos) y secundarias (foros y documentación menos formal y más empírica). Para ello se utilizarán bases de datos académicas, y literatura especializada en el desarrollo web. Durante toda esta primera fase de investigación se realizarán búsquedas específicas utilizando términos que sean claves y estén relacionados directamente con tecnologías de Frontend y sus tendencias actuales.

Para cumplir el segundo objetivo, se desarrollará un experimento computacional utilizando la guía de Wohlin para comparar el rendimiento, dicha guía se especializa en la realización de experimentos en un entorno controlado, por lo que será de gran ayuda en comparar la eficiencia de las tecnologías Frontend al consumir datos de APIs con las arquitecturas REST y GraphQL, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023.

Para cumplir el tercer objetivo, se tabulará, graficará y analizará los resultados obtenidos del experimento computacional para determinar la tecnología y arquitectura más eficiente en el consumo de datos en el Frontend.

Justificación

La realización de una evaluación comparativa de tecnologías Frontend en el contexto del procesamiento de datos, con un enfoque en las arquitecturas REST y GraphQL, se presenta como una iniciativa relevante y justificada por diversas razones. Esta investigación se alinea con múltiples Objetivos de Desarrollo Sostenible (ODS) y se sustenta en las siguientes fundamentaciones. (Naciones Unidas, 2015).

La investigación en tecnologías Frontend y arquitecturas tiene el potencial de generar conocimiento valioso en el campo de la informática y el desarrollo de aplicaciones web. Los resultados de esta investigación pueden beneficiar a la comunidad de desarrolladores y contribuir al ODS 4: Educación de Calidad al proporcionar información actualizada y relevante. Además, la investigación propuesta, encuentra su alineación directa con el Eje Económico y Generación de Empleo del Plan Nacional de Desarrollo (PNBV). Uno de los objetivos clave de este eje es la mejora de la eficiencia y la productividad a nivel nacional. (Secretaría Nacional de Planificación, 2021).

Adicionalmente, esta investigación también tiene el potencial de aportar al Eje Institucional, que busca perfeccionar la eficiencia de las instituciones públicas. Al identificar las tecnologías más eficaces para el desarrollo de consumo de APIs, la investigación podría asistir a las instituciones públicas en la optimización de sus servicios digitales, con un impacto positivo en la eficiencia institucional.

CAPITULO I

Marco Teórico

En este primer capítulo se presenta el marco teórico que sustenta la investigación, proporcionando los conceptos y fundamentos esenciales para comprender los aspectos involucrados en el estudio. Se abordarán temas relevantes como los principios y características de las tecnologías frontend, así como la importancia del consumo eficiente de datos en aplicaciones modernas.

Se realizará una comparación teórica entre las arquitecturas de APIs REST y GraphQL, poniendo énfasis a sus diferencias, para sentar las bases que guían el análisis experimental. También se explicará la métrica de tiempo de respuesta definida por la norma ISO/IEC 25023, la cual se utiliza como criterio principal para evaluar la eficiencia de las tecnologías estudiadas.

Se detallará el proceso y los criterios empleados para la selección de las tecnologías frontend comparadas en este trabajo, fundamentando esta elección en aspectos técnicos y prácticos relevantes para el contexto de la investigación.

1.1 Fundamentos de tecnologías Frontend

El desarrollo Frontend representa una de las áreas más cruciales dentro del ciclo de vida del software moderno, pues es el punto de interacción directa entre el usuario y la aplicación web. Abarca el diseño, construcción de interfaces y el aseguramiento de una experiencia de usuario óptima. Como explican Pérez Ibarra et al. en los Workshops de Investigadores en Ciencias de la Computación (WICC, 2021), “el Frontend trabaja la interfaz visual, y hace que el usuario pueda interactuar con nuestro sitio o sistema. Está orientado al lenguaje de marcas y al lenguaje de programación web de ejecución en equipos clientes”. Esta definición resalta no solo la función visual del Frontend, sino su importancia como intermediario

de la lógica del sistema y la percepción del usuario, lo que lo convierte en una pieza fundamental de la arquitectura web. En otras palabras, el Frontend no es solo el “rostro” de la aplicación, sino también el espacio donde se consolidan principios como la usabilidad, la accesibilidad, el rendimiento y la estética. En el contexto moderno, entender el Frontend implica comprender cómo las tecnologías del lado del cliente evolucionan junto con las expectativas y necesidades de los usuarios.

La introducción de JavaScript y, más específicamente, el surgimiento de frameworks como AngularJS, ReactJS produjeron el verdadero cambio. Estos marcos cambiaron la forma en que se construían las aplicaciones web, permitiendo la creación de aplicaciones dinámicas y de una sola página. AngularJS, desarrollado por Google, y ReactJS, desarrollado por Facebook, proporcionaron técnicas novedosas como el enlace de datos bidireccional y el DOM virtual, lo que facilitó la creación de interfaces de usuario complicadas y mejoró el rendimiento de las aplicaciones web. (Colectiva, 2023)

1.1.1 Evolución del desarrollo frontend

La evolución del frontend en las arquitecturas web modernas ha sido impulsada por la necesidad de construir aplicaciones más eficientes, escalables y centradas en la experiencia del usuario. Esta transformación ha llevado a que el frontend ya no se limite a la presentación visual, sino que participe activamente en la lógica de negocio y en la interacción directa con servicios distribuidos

Lanciaux (2021) propone una arquitectura basada en componentes reutilizables, desarrollados y probados de forma aislada, lo que permite mejorar la organización del código, reducir defectos y acelerar el desarrollo. Esta metodología, inspirada en la filosofía culinaria

de *mise en place*, permite dividir el trabajo en estaciones independientes, facilitando la colaboración entre equipos y la calidad del producto final.

La adopción de frameworks como React, Angular ha sido clave en esta evolución. Según Rathinam (2023), estos entornos ofrecen estructuras robustas, herramientas para la gestión del estado y mecanismos de renderizado eficientes, lo que permite construir aplicaciones modulares y mantenibles. Cada framework presenta ventajas particulares que se adaptan a distintos tipos de proyectos y necesidades técnicas.

El desarrollo de arquitecturas orientadas al futuro, como las que integran tecnologías emergentes en el contexto del Metaverso, evidencia la creciente complejidad del frontend. Ahmed y Ralescu (2023) destacan que el diseño de nuevas arquitecturas de internet requiere una reorganización profunda de los servicios y aplicaciones, donde el frontend juega un papel esencial en la interacción con tecnologías como realidad aumentada, blockchain y redes definidas por software.

1.1.2 Importancia del rendimiento en aplicaciones frontend

El rendimiento en aplicaciones frontend es un factor crítico que influye directamente en la experiencia del usuario, la retención y la eficiencia operativa. Rathinam (2023) señala que la elección del framework afecta significativamente el rendimiento, ya que cada uno ofrece diferentes mecanismos para la gestión del DOM, la optimización de recursos y la estructura del código. Estas diferencias impactan en la capacidad de respuesta de la interfaz y en la carga cognitiva del usuario.

Desde una perspectiva organizacional, el rendimiento también se relaciona con la productividad del equipo de desarrollo. Forsgren et al. (2021) introducen el modelo SPACE, que considera dimensiones como satisfacción, eficiencia, colaboración y flujo de trabajo. Un

frontend bien optimizado contribuye a reducir los tiempos de depuración, mejorar la experiencia del desarrollador y facilitar la entrega continua de valor.

Asimismo, la evaluación del rendimiento requiere modelos integrativos que consideren múltiples dimensiones. Lopes de Araújo et al. (2024) proponen un marco para analizar la efectividad de los sistemas de gestión del rendimiento, el cual puede adaptarse al análisis de métricas clave en aplicaciones frontend, como el *First Contentful Paint* (FCP), el *Time to Interactive* (TTI) y el *Cumulative Layout Shift* (CLS). Este enfoque permite identificar cuellos de botella y oportunidades de mejora desde una perspectiva sistémica.

1.2 Arquitecturas REST y GraphQL

Las interfaces de programación de aplicaciones (APIs) son esenciales para el desarrollo de aplicaciones modernas porque facilitan la comunicación entre varios sistemas de software. Una API funciona como un contrato entre un proveedor de servicios y un consumidor, lo que permite el intercambio de datos y funcionalidades de manera segura y estandarizada. Las APIs han cambiado mucho para satisfacer las demandas de aplicaciones más complejas y experiencias de usuario más diversas.

En el desarrollo de APIs web, la arquitectura REST se ha establecido como un modelo dominante por su simplicidad y escalabilidad, utilizando métodos HTTP estándar para una interacción efectiva entre cliente y servidor, convirtiéndose en la opción preferida en la era de las aplicaciones web y móviles. Sin embargo, con el aumento de la complejidad y la dinámica de las aplicaciones web, surgieron limitaciones en REST, lo que llevó a la introducción de GraphQL como una alternativa innovadora. GraphQL permite a los clientes especificar con precisión los datos requeridos en una consulta, abordando eficientemente los desafíos de rendimiento y optimización de datos. (Vadecha, 2021)

1.2.1 REST

REST (Representational State Transfer) es un estilo arquitectónico para el diseño de servicios web que ha revolucionado el desarrollo de aplicaciones en Internet. Esta arquitectura, concebida por Roy Fielding en su tesis doctoral en el año 2000, se basa en un conjunto de principios y restricciones que facilitan la creación de servicios web escalables, eficientes y fáciles de utilizar. REST utiliza los métodos estándar HTTP, como GET, POST, PUT y DELETE, para interactuar con recursos representados, generalmente en formatos como JSON o XML. (Archip, Herghelegiu, Amarandei, Mironeanu, & Șerban, 2018)

El éxito de REST se debe a su simplicidad y a la forma en que se alinea con los principios básicos de la web. REST facilita la interoperabilidad entre sistemas y simplifica las interacciones cliente-servidor al operar sobre recursos con URLs distintas y aprovechar los verbos HTTP. Como resultado de este método, REST se ha convertido en una opción preferida para la creación de APIs públicas y privadas, lo que ha llevado a su amplia adopción en todo tipo de aplicaciones web y móviles. Además, la escalabilidad y el rendimiento de los servicios web se benefician de la naturaleza stateless de REST, donde cada solicitud contiene toda la información necesaria para su procesamiento. (Williams, 2023)

Las características principales de REST, que lo definen como un estilo arquitectónico clave para el diseño de servicios web, incluyen:

- **Statelessness:** En REST, cada solicitud del cliente al servidor es independiente y autosuficiente. Esto significa que el servidor no necesita recordar el estado del cliente entre diferentes solicitudes, lo que facilita la escalabilidad y simplifica el diseño del servidor. (Cocca, 2023)
- **Cacheability:** REST promueve la caché de respuestas, permitiendo que los clientes almacenen respuestas anteriores para mejorar la eficiencia y reducir la

carga en el servidor. Esta característica es crucial para el rendimiento, ya que minimiza la necesidad de generar la misma respuesta para cada solicitud. (Cocca, 2023)

- **Layered System:** Los sistemas RESTful suelen estar organizados en capas, con cada capa teniendo responsabilidades específicas. Esto promueve la modularidad y mejora la seguridad al aislar componentes del sistema, evitando la propagación de errores a través de las capas. (Cocca, 2023)
- **Uniform Interface:** La interfaz uniforme de REST asegura un conjunto consistente de interacciones, independientemente de los detalles subyacentes de las aplicaciones o bases de datos. Esto simplifica las interacciones y hace que sean intuitivas y predecibles. (Cocca, 2023)
- **Uso de Métodos HTTP para Acciones Específicas:** REST emplea métodos HTTP (GET, POST, PUT, DELETE) para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar), lo que facilita la comprensión y el manejo de las acciones sobre los recursos. (Cocca, 2023)

Tabla 1
Operaciones REST

Acción	HTTP	Descripción
CREATE	POST	Con una solicitud HTTP, crea nuevos recursos.
RETRIEVE	GET	Utiliza una petición HTTP para obtener información de un recurso.

UPDATE	PUT	Actualiza los detalles de un recurso existente mediante una petición HTTP.
DELETE	DELETE	Elimina el recurso mediante una solicitud HTTP.

Fuente: (Galipenso, 2014)

1.2.2 GraphQL

GraphQL es un lenguaje de consulta y manipulación de datos para APIs y un motor de ejecución de consultas en tiempo real en el servidor. GraphQL, que fue lanzado por Facebook en 2015, se ha convertido en un excelente sustituto de las API REST convencionales. La principal ventaja de una API es que permite a los clientes especificar los datos que necesitan en lugar de depender de respuestas fijas del servidor. Esto resulta en APIs más eficientes que pueden satisfacer mejor las necesidades de los clientes con menos tiempo y procesamiento de consultas de datos. Se ha popularizado en una variedad de aplicaciones, desde plataformas de redes sociales hasta soluciones empresariales, gracias a su enfoque en la especificación precisa de los datos solicitados. El enfoque centrado en el cliente, la flexibilidad y la eficiencia lo convierten en una herramienta valiosa para el desarrollo moderno de APIs. (Graphql, 2023)

GraphQL tiene una serie de características clave que lo convierten en un estilo arquitectónico esencial para el diseño de servicios web, incluidas:

- **Obtención flexible de datos:** GraphQL permite a los clientes solicitar los datos precisos que necesitan en una consulta, lo que resuelve los problemas de sobreconsumo y subconsumo de datos que se encuentran en las API tradicionales. (Simha, 2023)
- **Eficiencia y rendimiento:** Las mejoras continuas en los backends de GraphQL están mejorando su eficiencia y rendimiento, con soluciones como Apollo-Router

y Wundergraph liderando en términos de tiempos de respuesta promedio y uso de memoria. (Neuse, 2023)

- **Agregación de varios servicios:** GraphQL facilita la creación de una interfaz más usable y localizada para los desarrolladores frontend al agregar múltiples microservicios en un esquema unificado. (Simha, 2023)
- **Esquemas Definidos:** En GraphQL, un esquema define la estructura de los datos que se pueden consultar en la API. Este esquema se escribe en el lenguaje de definición de esquemas de GraphQL (SDL) y define los tipos disponibles, los campos y las operaciones que se pueden realizar. (Durairaju, 2023).

1.2.3 Comparación teórica entre REST y GraphQL

La elección entre REST y GraphQL como arquitectura de APIs depende de múltiples factores técnicos y contextuales. Mientras REST ha sido el estándar dominante durante años, GraphQL ha ganado popularidad por su flexibilidad y eficiencia en la recuperación de datos. Diversos estudios han analizado estas tecnologías desde perspectivas como el rendimiento, la escalabilidad, la seguridad y la facilidad de mantenimiento. A continuación, se presenta una tabla comparativa basada en literatura científica y técnica que resume las principales diferencias entre ambas tecnologías. (Kanthed, 2023)

Tabla 2*REST vs GraphQL*

Aspecto	REST	GraphQL
Modelo de Datos	Basado en recursos y endpoints específicos.	Basado en esquemas y un único endpoint para consultas personalizadas.
Eficiencia de Datos	Puede provocar sobre-solicitud o sub-solicitud de datos.	Permite solicitar solo los datos necesarios, evitando sobrecarga.
Número de Peticiones	Múltiples peticiones para datos relacionados.	Una sola petición puede obtener datos complejos y relacionados.
Caché	Fácil de implementar con mecanismos HTTP estándar.	Más complejo debido a la naturaleza dinámica de las consultas.
Versionamiento	Requiere nuevos endpoints o cambios estructurales.	Permite evolución sin romper compatibilidad, agregando campos fácilmente.
Seguridad	Basado en autenticación estándar (OAuth, JWT), más maduro.	Requiere control granular de acceso por tipo de consulta.
Escalabilidad	Bien soportado en arquitecturas tradicionales.	Mejor rendimiento en aplicaciones con alta interacción y datos complejos.

Facilidad de Uso	Más simple y ampliamente adoptado.	Mayor curva de aprendizaje, pero más flexible.
Casos de Uso Ideales	Aplicaciones con recursos bien definidos y necesidades simples.	Aplicaciones con relaciones complejas entre datos y evolución constante.

1.3 Selección de tecnologías Frontend

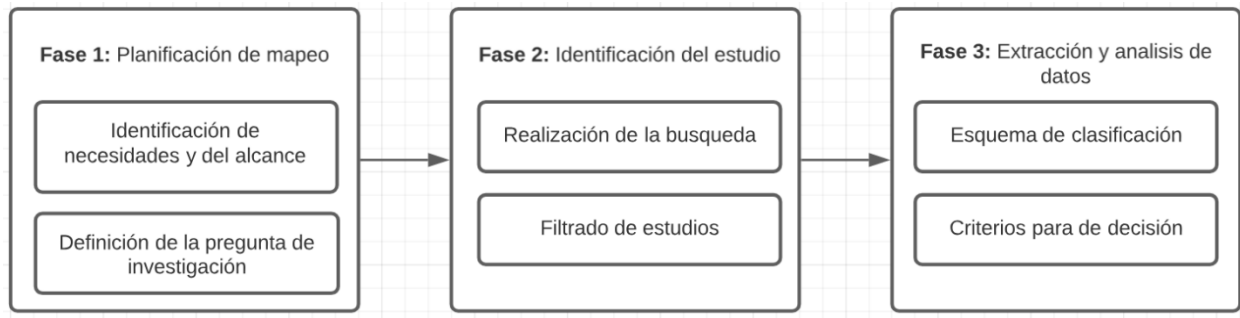
Con el propósito de fundamentar adecuadamente la elección de las tecnologías frontend, se ha realizado una investigación detallada en varios repositorios científicos. Esta investigación tuvo como meta la recopilación de los estudios más actuales que abordan proyectos implementados con las arquitecturas REST y GraphQL. Además, se buscó identificar las tecnologías frontend específicas que se emplean en estos estudios.

1.3.1 Definición del mapeo sistemático

Para identificar las tres tecnologías principales empleadas en el desarrollo de REST y GraphQL, se planea adoptar la metodología delineada en el artículo de investigación del ingeniero Antonio Quiña. Este enfoque metodológico se basa en los principios establecidos en la Guía de Wohlin.

Ilustración 3

Proceso de mapeo sistemático



Es esencial iniciar precisando el fin que persigue la investigación. En este caso el objetivo general es la comparación de la eficiencia de diversas tecnologías Frontend en términos de su consumo de datos de APIs, específicamente al utilizar arquitecturas REST y GraphQL. Así, el punto de partida consiste en formular las preguntas de investigación que orientarán el análisis, en línea con los planteamientos del ingeniero Antonio Quiña (Quiña-Mera, Fernandez, García, & Ruiz-Cortés, 2022). Las interrogantes específicas que abordar son las siguientes:

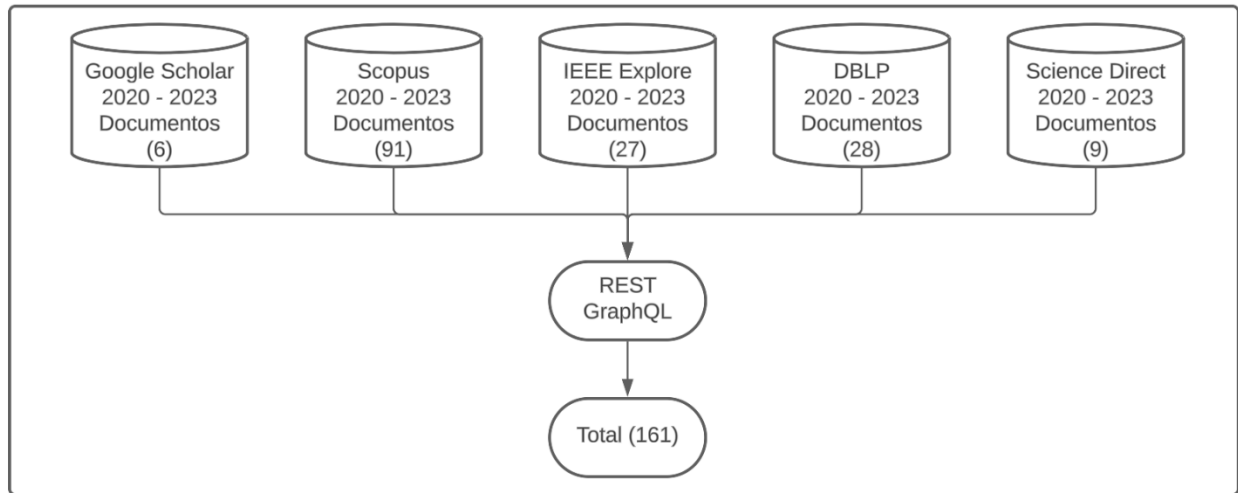
RQ1: ¿Qué tecnologías frontend son las más usadas para el consumo de APIs con REST? El objetivo de esta pregunta es identificar las tecnologías frontend que serán objeto de estudio en relación con REST en etapas posteriores.

RQ2: ¿Qué tecnologías frontend son las más usadas para el consumo de APIs con GraphQL? Así mismo, esta interrogante busca determinar las tecnologías frontend que se utilizarán en conjunción con GraphQL.

A continuación, se ilustra en la Figura 4 como se realizó la indagación en las bases de datos, para luego de forma manual obtener las tecnologías frontend utilizadas en los artículos.

Ilustración 4

Distribución de los datos obtenidos



1.3.2 Identificación de estudios

En esta sección se lleva a cabo la realización de la búsqueda, empleando estrategias sistemáticas para localizar estudios relevantes en bases de datos académicas y otros repositorios científicos. Una vez obtenidos los estudios preliminares, se procede al filtrado de estos, aplicando criterios de inclusión y exclusión previamente establecidos para asegurar la pertinencia y calidad de la información recogida. Esto garantiza que solo los estudios más relevantes sean considerados para la extracción de datos.

Criterios de inclusión:

- Publicaciones de 2020 a 2023
- Publicaciones centradas en uso de las arquitecturas REST y GraphQL
- Publicaciones que contengan objetivos, métodos y procesos establecidos.
- Publicaciones aceptadas en revistas o conferencias.

Criterios de exclusión:

- Libros o capítulos de libros.
- Publicaciones que no sean de acceso abierto.
- Publicaciones sin relación con el área de la tecnología.
- Publicaciones que mencionan las arquitecturas, pero no las aplica.

1.3.3 Extracción y análisis de datos

Esta fase abarca la extracción y el análisis de datos de los estudios seleccionados. Los datos se categorizan según un conjunto de criterios definidos, lo que permite una evaluación y comparación efectivas de las tecnologías frontend identificadas. Esta clasificación facilita la interpretación de los resultados y ayuda a responder las preguntas de investigación planteada. Los resultados obtenidos fueron los siguientes:

Tabla 3

Número de documentos por base de datos

Base de datos bibliográfica	Documentos por arquitectura y base de datos bibliográfica
DBLP	28
GraphQL	17
REST	11
Google Scholar	6
GraphQL	3
REST	3
IEEE Xplore	27

GraphQL	6
REST	21
ScienceDirect	9
GraphQL	6
REST	3
Scopus	91
GraphQL	11
REST	80
Total	161

Tabla 4

Frecuencia de tecnologías frontend usadas en APIs GraphQL

Tecnologías frontend usadas en APIs GraphQL	Frecuencia
Python (Flask)	6
Angular.js	2
Vue.js	1
React.js	4

Tabla 5

Frecuencia de tecnologías frontend usadas en APIs REST

Tecnologías frontend usadas en APIs REST	Frecuencia
Angular.js	3
Vue.js	2
Beetl	1
Hyperlenger Fabric	1

React.js	5
Django	2
JQuery	1

Basándonos en los datos obtenidos, se identifica que para el consumo de APIs con GraphQL, Angular.js y React.js aparecen como las tecnologías más utilizadas. Sin embargo, durante la revisión bibliográfica no se encontró mención directa al framework Flask. A pesar de ello, se observó una alta frecuencia en el uso del lenguaje de programación Python, lo que motivó la búsqueda del framework más representativo asociado a este lenguaje. Según las encuestas de StackOverflow de 2023 y 2024, Flask figura entre los frameworks web más utilizados y deseados por los desarrolladores, mostrando un incremento en su popularidad en 2024 (Stackoverflow, Stackoverflow, 2023) (Stackoverflow, 2024). Por esta razón, se consideró pertinente incluir Flask en la comparativa, no solo por su vínculo con Python, sino también para diversificar el análisis tecnológico, ya que otros frameworks como Vue.js o Next.js también utilizan JavaScript, mientras que React ya ha sido seleccionado como representante para este lenguaje. Por lo tanto, para un análisis comparativo coherente, se seleccionarán Angular.js, Flask y React.js como las tecnologías frontend para realizar el experimento computacional subsiguiente. Esto permitirá determinar cuál de estas tecnologías ofrece una combinación más eficiente en términos de tiempo de respuesta cuando se utiliza con arquitecturas REST y GraphQL.

Con esta selección, abordamos la primera pregunta de investigación (RQ1), concluyendo que seleccionarán Angular.js, Flask y React.js son las tecnologías de mayor prevalencia para el desarrollo con REST. Del mismo modo, al responder la segunda pregunta (RQ2), establecemos que estas mismas tecnologías son igualmente representativas para el desarrollo con GraphQL.

1.4 ISO/IEC 25023

La norma ISO/IEC 25023, integrante de la serie Systems and Software Quality Requirements and Evaluation (SQuaRE), establece un enfoque para la medición cuantitativa de la calidad en productos y sistemas de software. Adoptada en 2016, esta normativa es clave para definir medidas de calidad que permiten una evaluación sistemática y detallada del software. (iso.org, 2022)

Este estándar incluye:

- Un conjunto específico de medidas de calidad, abarcando diversos aspectos del software.
- Directrices sobre cómo implementar estas medidas para evaluar la calidad de los productos y sistemas de software.

Además, la norma contiene anexos informativos que ofrecen directrices adicionales sobre el uso de estas medidas de calidad, los elementos utilizados para definir estas medidas, y una explicación detallada de los diferentes tipos de medición. Es importante destacar que la norma no prescribe rangos específicos de valores para las medidas, ya que estos dependen de la naturaleza particular del sistema o producto evaluado, así como de factores como la categoría del software, el nivel de integridad y las necesidades específicas de los usuarios (iso.org, 2022). El propósito principal de las medidas de calidad en la ISO/IEC 25023 es su uso en la garantía y mejora de la calidad de productos y sistemas de software a lo largo o después de su ciclo de vida de desarrollo. Los principales beneficiarios de esta norma son los profesionales encargados de la especificación y evaluación de requisitos de calidad en diferentes etapas, incluyendo el desarrollo, gestión de calidad, suministro, adquisición y mantenimiento de software o sistemas. (iso.org, 2022)

Tabla 6

Características y Sub-características de la ISO/IEC 2023

CARÁCTERÍSTICA	SUB-CARÁCTERÍSTICAS
Funcionalidad	Adecuación: capacidad del software para proporcionar las funciones requeridas por el usuario.
	Exactitud: precisión y corrección de los resultados y salidas del software
	Interoperabilidad: capacidad del software para interactuar y funcionar con otros sistemas
	Seguridad de acceso: protección de la información y los datos del software contra accesos no autorizados.
	Madurez: capacidad del software para evitar fallos o errores.
Confiabilidad	Tolerancia a fallos: capacidad del software para mantener un nivel de rendimiento aceptable incluso en caso de fallos.
	Capacidad de recuperación: capacidad del software para recuperarse rápidamente después de un fallo.
	Cumplimiento: capacidad del software para cumplir con los estándares, regulaciones y requisitos legales aplicables.
	Comprensibilidad: facilidad de comprensión del software para los usuarios.
Usabilidad	Aprendizaje: facilidad de aprendizaje del software para los usuarios.
	Operabilidad: facilidad de operación del software para los usuarios.
	Atractivo visual: diseño atractivo y agradable del software.
	Comportamiento en el tiempo: rendimiento y tiempo de respuesta del software.
Eficiencia	Utilización de recursos: eficiencia en el uso de los recursos del sistema, como CPU, memoria y espacio en disco
	Capacidad: capacidad del software para manejar grandes volúmenes de datos o transacciones.
Mantenibilidad	Analizabilidad: facilidad de comprensión y análisis del software para el mantenimiento y la modificación.

	Cambiabilidad: facilidad de realizar cambios y modificaciones en el software.
	Estabilidad: capacidad del software para evitar efectos no deseados al realizar cambios.
	Testabilidad: facilidad para realizar pruebas y validar el software.
	Adaptabilidad: capacidad del software para adaptarse a diferentes entornos y plataformas.
Portabilidad	Instalabilidad: facilidad de instalación del software en diferentes entornos.
	Reemplazabilidad: facilidad de reemplazar el software existente por otro software.

Fuente: (Valencia, 2023)

1.4.1 ISO/IEC 25023 métrica de tiempo de respuesta

La métrica de tiempo, dentro de la característica de eficiencia, evalúa aspectos críticos como el tiempo de respuesta y el tiempo de procesamiento de un sistema de software. Es una medida cuantitativa que determina la rapidez con la que un software realiza sus funciones necesarias bajo condiciones específicas. Esta métrica es especialmente relevante en aplicaciones y sistemas donde el rendimiento temporal es un factor crítico, como en sistemas de procesamiento de transacciones en tiempo real, aplicaciones interactivas, o plataformas que requieren alta capacidad de respuesta. La importancia de esta métrica radica en su capacidad para proporcionar una evaluación objetiva y estandarizada de la eficiencia temporal del software. Proporciona una forma de medir cuán efectivamente un producto o sistema de software cumple con los requisitos de rendimiento en términos de velocidad de procesamiento y tiempo de respuesta. Esta evaluación es crucial para determinar la viabilidad y eficacia de un sistema de software en entornos donde la velocidad y la eficiencia son esenciales para el éxito de la operación. (iso.org, 2022)

CAPITULO II

En este capítulo se describe el diseño y desarrollo del experimento computacional que permite evaluar la eficiencia de diferentes tecnologías frontend en el consumo de datos desde APIs basadas en arquitecturas REST y GraphQL. El objetivo principal de este apartado es detallar el proceso seguido para garantizar la validez y confiabilidad de los resultados obtenidos.

Se inicia con la definición y configuración de los casos de uso que representan distintos niveles de complejidad en las consultas, y que sirven como base para la comparación entre las tecnologías seleccionadas. Luego, se explica el diseño de las consultas ajustadas a los requerimientos de cada caso de uso.

Asimismo, se presenta la configuración del entorno de pruebas, incluyendo las herramientas utilizadas, las instrucciones de instalación del experimento computacional y los pasos necesarios para su correcta ejecución. Finalmente, se explica el procedimiento seguido para la recolección de los datos, asegurando condiciones controladas y homogéneas durante toda la experimentación.

2.1 Planificación

Con el objetivo de evaluar cuál tecnología Frontend y API resulta más eficiente bajo distintos tipos de carga y estructuras de consulta, se planteó la necesidad de diseñar un experimento computacional controlado, considerando diferentes escenarios de consulta y volúmenes de datos.

Para garantizar que el experimento refleje condiciones diversas y realistas, se propone construir una arquitectura de datos relacional jerárquica que permita aplicar tanto consultas simples como consultas en cascada. Estas últimas permiten simular situaciones donde una

solicitud de información implica acceder a múltiples entidades asociadas, algo común en aplicaciones modernas.

El diseño de estos casos permite observar cómo se comportan las tecnologías frontend cuando aumenta la profundidad y cantidad de datos consultados, revelando las ventajas o limitaciones de cada enfoque según el tipo de consulta que se realice.

2.1.1 Definición del experimento computacional

El experimento se diseña bajo criterios de equidad tecnológica, garantizando que tanto REST como GraphQL accedan a la misma estructura de datos y sean evaluados en igualdad de condiciones.

Para cumplir con el objetivo del experimento computacional, se establecieron una serie de requerimientos. Estos incluyeron el consumo de APIs equivalentes en REST y GraphQL con una base de datos relacional jerárquica con cinco niveles de entidades relacionadas, y la definición de cargas crecientes que permitan analizar el impacto del volumen de datos en el rendimiento. Asimismo, se repitieron las pruebas múltiples veces por escenario, y se organizaron los experimentos en casos de uso con distintos niveles de profundidad, lo que permite evaluar tanto la eficiencia de las tecnologías como la forma en que estas escalan ante estructuras de datos más complejas.

2.1.2. Justificación de los Casos de uso

Los casos de uso representan distintos grados de complejidad en la obtención de información al consumir la API. Cada uno fue diseñado para probar un nivel de profundidad en la estructura jerárquica, lo cual permite evaluar el rendimiento progresivo de las tecnologías bajo prueba.

Caso de Uso N°1 – Consulta a un nivel

Este caso representa la consulta más simple, en la que se accede a una sola tabla (por ejemplo, obtener una lista de usuarios). Su propósito es establecer una línea base del rendimiento de cada tecnología sin la carga añadida de relaciones. La distribución de carga ([1, 10, 100, 1000, 10000, 100000]) permite observar cómo se comportan las tecnologías al escalar en cantidad, pero sin profundidad.

Caso de Uso N°2 – Consulta a dos niveles

Aquí se incluye una relación uno a muchos entre dos entidades (usuarios → publicaciones). Este caso evalúa cómo cada tecnología gestiona la recuperación de datos relacionados, agregando una capa más de complejidad. La distribución ([1x1, 3x3, 10x10, 32x31, 100x100, 316x316]) fue elegida para generar incrementos realistas y medibles de volumen en ambas entidades relacionadas.

Caso de Uso N°3 – Consulta a tres niveles

Este escenario añade una tercera entidad relacionada (usuarios → publicaciones → comentarios), simulando una estructura habitual en muchas aplicaciones. La distribución ([1x1x1, 2x2x2, 5x5x4, 10x10x10, 22x22x21, 46x46x47]) busca representar combinaciones con crecimiento moderado y progresivo, permitiendo evaluar el impacto de las consultas anidadas de mayor profundidad.

Caso de Uso N°4 – Consulta a cuatro niveles

Con una jerarquía aún más profunda (usuarios → publicaciones → comentarios → respuestas), este caso representa situaciones donde se requiere extraer información más interconectada y con múltiples dependencias. La carga ([1x1x1x1, 2x2x2x1, 3x3x3x4, 6x6x6x5, 10x10x10x10,

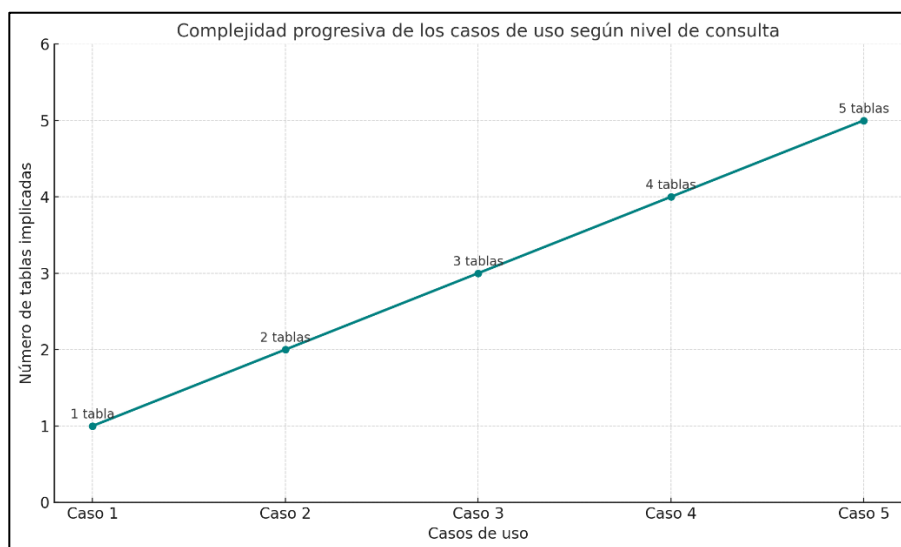
18x18x18x17]) aumenta la complejidad sin generar combinaciones artificiales, permitiendo un análisis más detallado del rendimiento frente a consultas encadenadas.

Caso de Uso N°5 – Consulta a cinco niveles

Este es el caso más exigente, en el que se evalúa la eficiencia de las tecnologías al manejar cinco niveles de relaciones (usuarios → publicaciones → comentarios → respuestas → reacciones). Esta estructura es ideal para analizar el rendimiento en escenarios con datos altamente jerárquicos y de múltiples dependencias. La distribución ([1×1×1×1×1, 2×2×1×2×1, 3×3×2×3×2, 4×4×4×4×4, 6×6×7×6×7, 10×10×10×10×10]) permite observar tanto escenarios moderados como condiciones de carga intensiva.

Ilustración 5

Complejidad progresiva de los Casos de Uso



Como se puede observar en la **Ilustración 5**

Complejidad progresiva de los Casos de Uso se logra el objetivo de aumentar la complejidad según el caso de uso.

2.1.3 Diseño de consultas y estructura de datos

Para garantizar la validez y la replicabilidad del experimento computacional, es esencial establecer desde el inicio un diseño claro tanto de la estructura de datos que las APIs a consumir debería de ofrecer. El enfoque adoptado considera una arquitectura organizada y jerárquica que permita simular distintas profundidades de acceso a los datos, de acuerdo con los cinco casos de uso definidos previamente.

Estructura jerárquica de datos

La base de datos relacional que deben utilizar las APIs debe estar conformada por cinco entidades principales con relaciones uno a muchos entre ellas.

Cada entidad estará relacionada con la anterior de manera jerárquica, por ejemplo: un usuario puede tener múltiples publicaciones, cada publicación múltiples comentarios, cada comentario múltiples respuestas, y cada respuesta múltiples reacciones. Esta organización facilita la construcción de consultas escalonadas que permiten medir el rendimiento en escenarios de creciente complejidad.

Estructura general de consulta

Cada consulta se diseñará para adaptarse dinámicamente a la distribución de carga definida en los casos de uso. Para ello, se utilizará un sistema de control que permita seleccionar el caso a evaluar y, de acuerdo con él, iniciar una serie de peticiones estructuradas. Estas peticiones deben ser capaces de:

- Solicitar la cantidad exacta de registros especificados para cada entidad.

- Iterar sobre las entidades padre para construir dinámicamente las solicitudes a sus entidades hijas.
- Repetir cada consulta al menos tres veces para calcular un tiempo de respuesta promedio y mitigar posibles desviaciones.

Estructura del proyecto

Para llevar a cabo estas consultas, se necesita que el sistema frontend tenga una estructura modular y reutilizable.

- Un controlador central que gestione la lógica de ejecución del experimento, incluyendo la selección del caso, la iteración por cada distribución de carga y la ejecución repetida de cada escenario.
- Un conjunto de funciones encargadas de realizar llamadas específicas a la API REST o GraphQL, según la arquitectura utilizada, para recuperar datos de cada entidad.
- Un sistema de medición de tiempo que registre la duración total de cada consulta desde su inicio hasta su finalización.
- Un mecanismo para almacenar los resultados obtenidos, y permitir su descarga en formato Excel para facilitar el análisis comparativo.

2.1.4 Entorno de Pruebas y Configuración Técnica

Características del equipo de pruebas

Para la ejecución del experimento computacional se utilizó un equipo de escritorio con las siguientes especificaciones técnicas:

- ❖ **Procesador:** 13th Gen Intel(R) Core(TM) i7-13620H (2.40 GHz)
- ❖ **Memoria RAM:** 16 GB DDR4
- ❖ **Unidad de almacenamiento:** SSD de 512 GB
- ❖ **Sistema operativo:** Windows 11 Pro (64 bits)
- ❖ **Gráficos:** Integrados (Intel UHD Graphics)

Configuración del entorno

El entorno de pruebas fue configurado de manera local en el equipo que se describió anteriormente. Estas fueron las herramientas y los elementos utilizados:

Backend: Servidor Node.js con Express (para REST) y Apollo Server (para GraphQL)

Frontend: Aplicaciones desarrolladas en:

- React
- Angular
- Flask con Jinja2

Gestión de datos: Datos simulados generados dinámicamente en el Backend, emulando estructuras anidadas (usuarios, publicaciones, comentarios, respuestas y reacciones)

Herramientas de medición:

La función que se utiliza en JavaScript (React) y TypeScript (Angular) es `performance.now()` y `time.time()` en Python (Flask) para medir los tiempos de respuesta del lado del cliente.

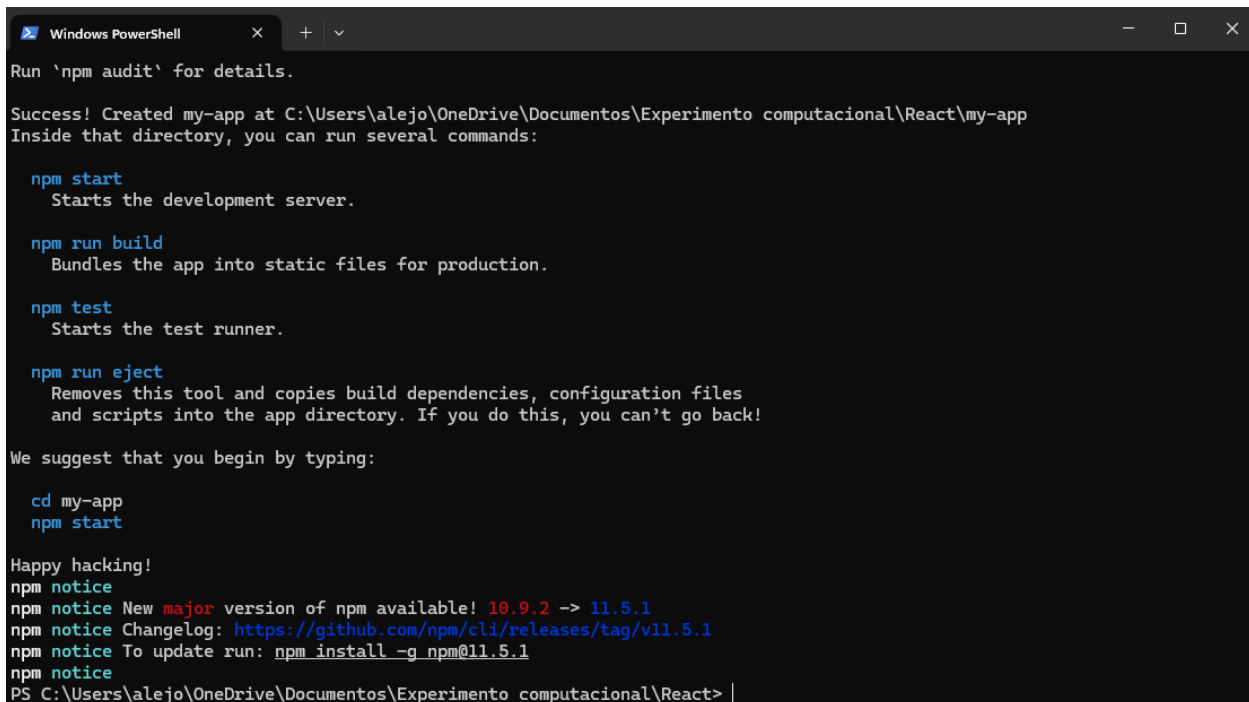
Todos los resultados fueron registrados en archivos Excel, lo que permitió analizarlos posteriormente.

Servidor de pruebas: Localhost en puerto 5003 para GraphQL y 5001 para REST

Instalación del entorno

Ilustración 6

Instalación de React



```
Windows PowerShell
Run 'npm audit' for details.

Success! Created my-app at C:\Users\alejo\OneDrive\Documentos\Experimento computacional\React\my-app
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd my-app
  npm start

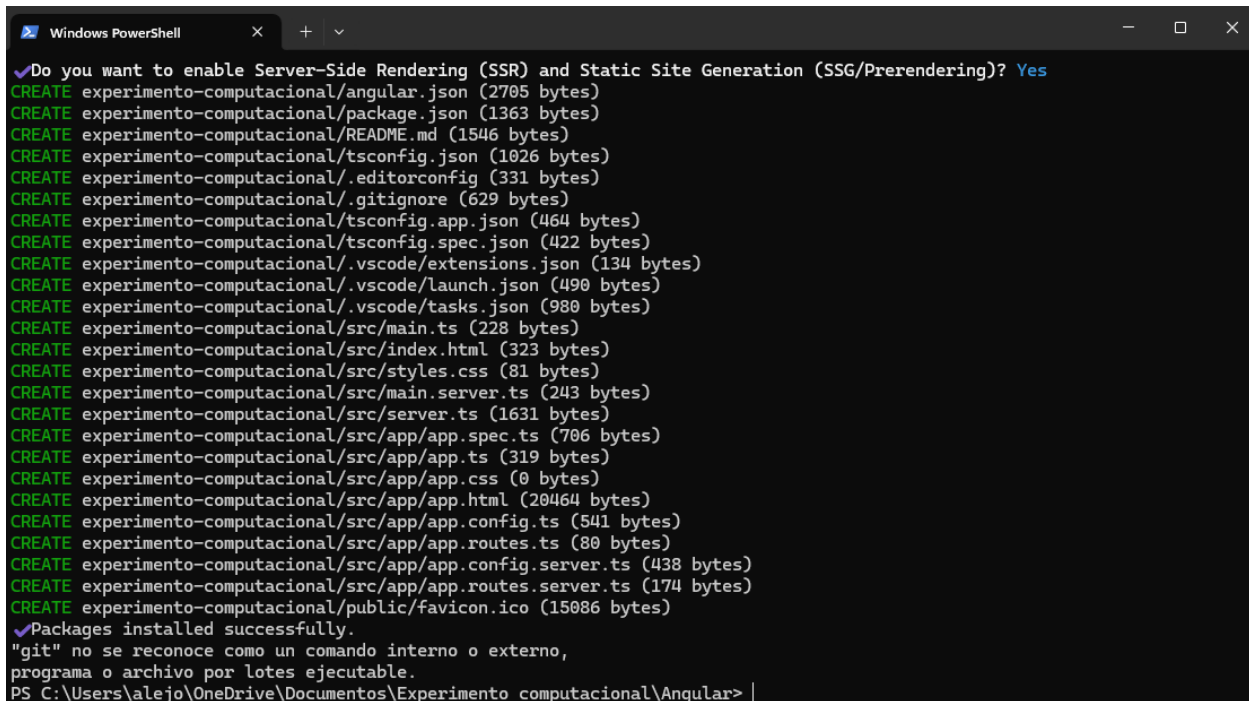
Happy hacking!
npm notice
npm notice New major version of npm available! 10.9.2 -> 11.5.1
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.5.1
npm notice To update run: npm install -g npm@11.5.1
npm notice
PS C:\Users\alejo\OneDrive\Documentos\Experimento computacional\React> |
```

Con React, se utilizó Create React App, una herramienta que permite inicializar proyectos React de manera rápida y con una configuración predeterminada. Esta herramienta facilita la creación de una estructura de proyecto organizada. La instalación se realizó con el siguiente comando en PowerShell, una vez instalado Node.js y npm:

`npx create-react-app my-app`

Ilustración 7

Instalación de Angular



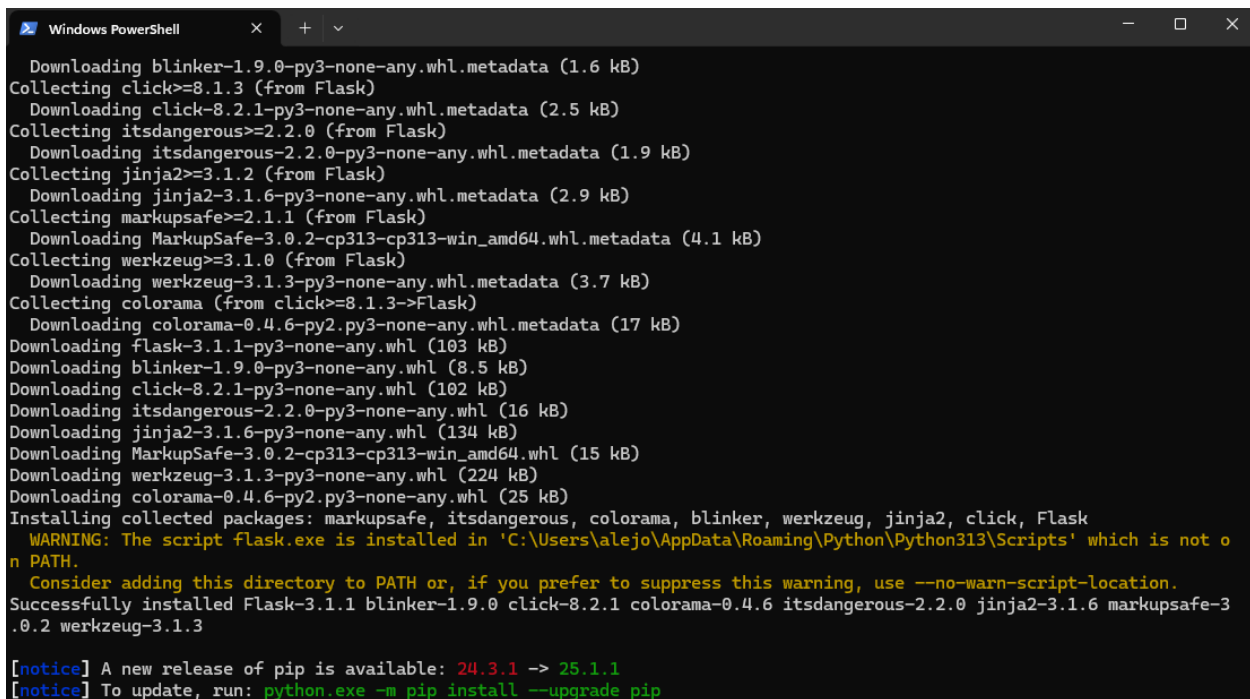
```
Windows PowerShell
✔Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? Yes
CREATE experimento-computacional/angular.json (2705 bytes)
CREATE experimento-computacional/package.json (1363 bytes)
CREATE experimento-computacional/README.md (1546 bytes)
CREATE experimento-computacional/tsconfig.json (1026 bytes)
CREATE experimento-computacional/.editorconfig (331 bytes)
CREATE experimento-computacional/.gitignore (629 bytes)
CREATE experimento-computacional/tsconfig.app.json (464 bytes)
CREATE experimento-computacional/tsconfig.spec.json (422 bytes)
CREATE experimento-computacional/.vscode/extensions.json (134 bytes)
CREATE experimento-computacional/.vscode/launch.json (490 bytes)
CREATE experimento-computacional/.vscode/tasks.json (980 bytes)
CREATE experimento-computacional/src/main.ts (228 bytes)
CREATE experimento-computacional/src/index.html (323 bytes)
CREATE experimento-computacional/src/styles.css (81 bytes)
CREATE experimento-computacional/src/main.server.ts (243 bytes)
CREATE experimento-computacional/src/server.ts (1631 bytes)
CREATE experimento-computacional/src/app/app.spec.ts (706 bytes)
CREATE experimento-computacional/src/app/app.ts (319 bytes)
CREATE experimento-computacional/src/app/app.css (0 bytes)
CREATE experimento-computacional/src/app/app.html (20464 bytes)
CREATE experimento-computacional/src/app/app.config.ts (541 bytes)
CREATE experimento-computacional/src/app/app.routes.ts (80 bytes)
CREATE experimento-computacional/src/app/app.config.server.ts (438 bytes)
CREATE experimento-computacional/src/app/app.routes.server.ts (174 bytes)
CREATE experimento-computacional/public/favicon.ico (15086 bytes)
✔Packages installed successfully.
"git" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
PS C:\Users\alejo\OneDrive\Documentos\Experimento computacional\Angular>
```

Para implementar el experimento computacional utilizando Angular, se utilizó Angular CLI, una herramienta que facilita la creación, configuración y gestión de aplicaciones Angular. Esta herramienta permite generar proyectos estructurados, compilar aplicaciones y gestionar sus componentes. Anteriormente, fue necesario instalar Angular CLI globalmente a través de npm. La creación del proyecto se realizó mediante el siguiente comando:

ng new experimento-computacional

Ilustración 8

Instalación de Flask



```
Windows PowerShell
Downloading blinker-1.9.0-py3-none-any.whl.metadata (1.6 kB)
Collecting click>=8.1.3 (from Flask)
  Downloading click-8.2.1-py3-none-any.whl.metadata (2.5 kB)
Collecting itsdangerous>=2.2.0 (from Flask)
  Downloading itsdangerous-2.2.0-py3-none-any.whl.metadata (1.9 kB)
Collecting jinja2>=3.1.2 (from Flask)
  Downloading jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting markupsafe>=2.1.1 (from Flask)
  Downloading MarkupSafe-3.0.2-cp313-cp313-win_amd64.whl.metadata (4.1 kB)
Collecting werkzeug>=3.1.0 (from Flask)
  Downloading werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Collecting colorama (from click>=8.1.3->Flask)
  Downloading colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Downloading flask-3.1.1-py3-none-any.whl (103 kB)
Downloading blinker-1.9.0-py3-none-any.whl (8.5 kB)
Downloading click-8.2.1-py3-none-any.whl (102 kB)
Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Downloading jinja2-3.1.6-py3-none-any.whl (134 kB)
Downloading MarkupSafe-3.0.2-cp313-cp313-win_amd64.whl (15 kB)
Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Installing collected packages: markupsafe, itsdangerous, colorama, blinker, werkzeug, jinja2, click, Flask
WARNING: The script flask.exe is installed in 'C:\Users\alejo\AppData\Roaming\Python\Python313\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed Flask-3.1.1 blinker-1.9.0 click-8.2.1 colorama-0.4.6 itsdangerous-2.2.0 jinja2-3.1.6 markupsafe-3.0.2 werkzeug-3.1.3

[notice] A new release of pip is available: 24.3.1 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Para la creación del experimento computacional en Flask, se realizó en un entorno virtual de Python. El siguiente comando se utilizó para instalar Flask, el cual incluye por defecto el soporte para Jinja2:

```
pip install Flask
```

2.2 Desarrollo

Para el desarrollo del experimento computacional se utilizó como entorno principal el editor de código Visual Studio Code, debido a su versatilidad, compatibilidad con múltiples lenguajes de programación y su amplio ecosistema de extensiones. En este entorno se implementaron y crearon los tres frameworks: Angular, React y Flask, cada uno configurado con sus respectivas versiones y dependencias.

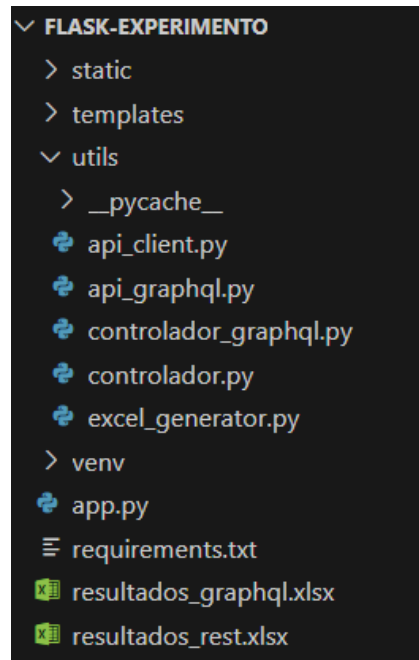
Todos los proyectos fueron desarrollados siguiendo una estructura similar y utilizando el mismo método para la obtención de datos. Por esta razón, tanto la forma de obtener como de procesar los datos puede parecer repetitiva. Es posible que exista cierta redundancia al explicar estos aspectos en cada uno de los proyectos.

2.2.1. Flask

Estructura del proyecto

Ilustración 9

Distribución de carpetas del proyecto - Flask



El proyecto Flask se organizó en módulos los cuales se pueden visualizar en la

Ilustración 9

Distribución de carpetas del proyecto - Flask, separando la lógica de consumo de API (api_client.py, api_graphql.py), el controlador del experimento (controlador_graphql.py,

controlador.py). También se creó un módulo en donde se va a encargar exclusivamente para crear el documento Excel con los datos obtenidos (excel_generador.py).

Conexión a las APIs

Ilustración 10

Conexión API - REST

```
utils > api_client.py > --
1 import requests
2
3 class APIClient:
4     BASE_URL = "http://localhost:5001" # URL DE API REST
5
6     def get_usuarios(self, num_usr):
7         url = f"{self.BASE_URL}/usuarios/{num_usr}"
8         response = requests.get(url)
9         return response.json()
10
11     def get_usuarios_con_publicaciones(self, num_usr):
12         url = f"{self.BASE_URL}/usuarios/{num_usr}"
13         response = requests.get(url)
14         return response.json()
15
16     def get_publicaciones(self, id_usr, num_pst):
17         url = f"{self.BASE_URL}/publicaciones/{id_usr}/{num_pst}"
18         response = requests.get(url)
19         return response.json()
20
21     def get_comentarios(self, id_pub, num_cmt):
22         url = f"{self.BASE_URL}/comentarios/{id_pub}/{num_cmt}"
23         response = requests.get(url)
24         return response.json()
25
26     def get_respuestas(self, id_cmt, num_res):
27         url = f"{self.BASE_URL}/respuestas/{id_cmt}/{num_res}"
28         response = requests.get(url)
29         return response.json()
30
```

La Ilustración 10

Conexión API - REST muestra el módulo encargado de obtener los datos desde la API REST.

En esta sección únicamente se realiza la obtención de los datos, sin aplicar ningún tipo de procesamiento sobre ellos.

Ilustración 11

Conexión API - GraphQL

```
utils > epi.graphql.py > APIGraphQL
1 from gql import gql, Client
2 from gql.transport.requests import RequestsHTTPTransport
3
4 class APIGraphQL:
5     BASE_URL = "http://localhost:5003/graphql" # URL API GraphQL
6
7     def __init__(self):
8         transport = RequestsHTTPTransport(url=self.BASE_URL)
9         self.client = Client(transport=transport, fetch_schema_from_transport=True)
10
11     def get_usuarios(self, num_usr):
12         query = gql("""
13             query GetUsuariosCantidad($numUsr: Int!) {
14                 getUsuariosCantidad(numUsr: $numUsr) {
15                     id_usuario
16                     nombre
17                     correo
18                     clave
19                 }
20             }
21         """)
22         result = self.client.execute(query, variable_values={"numUsr": num_usr})
23         return result.get("getUsuariosCantidad", [])
```

La Ilustración 11

Conexión API - GraphQL muestra al módulo en donde se obtienen los datos desde la API GraphQL.

Controlador del experimento

Ilustración 12

Fragmento de código - Consulta en cascada

```
utils > controlador.py > ControladorExperimento > consultar_datos
6 class ControladorExperimento:
7
8     def consultar_datos(self, caso, distribucion):
9         partes = distribucion.split('x')
10         niveles = len(partes)
11         datos = {}
12
13         # Consultar usuarios
14         cantidad_usuarios = int(partes[0])
15
16         # Determinar qué método usar para obtener usuarios
17         if caso == "Cul":
18             usuarios = self.api_client.get_usuarios(cantidad_usuarios) # Usa la URL específica para Cul
19         else:
20             usuarios = self.api_client.get_usuarios_con_publicaciones(cantidad_usuarios) # Usa la URL para otros casos
21
22         if not isinstance(usuarios, list) or not usuarios:
23             return {"error": "No se encontraron usuarios"}
24         datos["usuarios"] = usuarios
25
26         # Procesar niveles adicionales (publicaciones, comentarios, respuestas, reacciones)
27         for usuario in usuarios[:cantidad_usuarios]:
28             usuario_id = usuario.get('id_usuario', None)
29             if not usuario_id:
30                 continue
31
32             datos_usuario = self._procesar_niveles(usuario_id, partes[1:], niveles)
33             datos.setdefault("detalles", []).append(datos_usuario)
34
35         return datos
```

El fragmento de código que se observa en la Ilustración 12

Fragmento de código - Consulta en cascada podemos comprobar que el experimento

computacional fue configurado para realizar las consultas en cascada y poder cumplir todos los casos de uso.

Exportación de resultados

Ilustración 13

Fragmento de código - Generar Excel

```
import openpyxl

class ExcelGenerator:
    def __init__(self, api_type, filename_prefix="resultados"):
        # Genera un nombre de archivo basado en el tipo de API
        self.filename = f"{filename_prefix}_{api_type}.xlsx"
        self.workbook = openpyxl.Workbook()
        self.sheet = self.workbook.active
        # Encabezados actualizados
        self.sheet.append([
            "Caso de uso",
            "Arquitectura",
            "Tecnología",
            "Distribución",
            "Numero de Registros", # Nueva columna
            "Tiempo(ms)" # Ya no incluimos "Promedio de tiempo"
        ])
    })
```

El código que se visualiza en la **Ilustración 13**

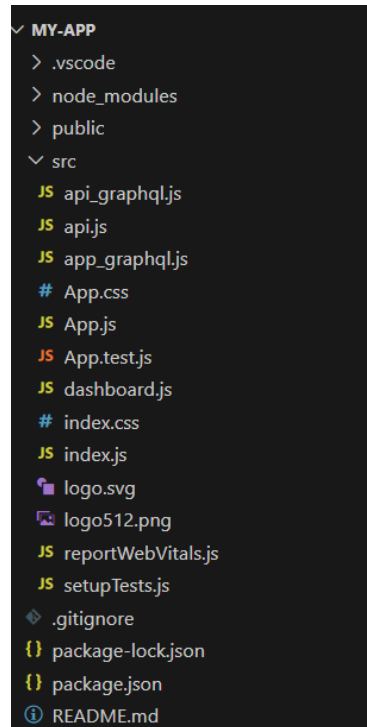
Fragmento de código - Generar Excel es el encargado de generar el archivo Excel con los resultados del experimento computacional

2.2.2 React

Estructura del proyecto

Ilustración 14

Distribución de carpetas del proyecto - React



En la Ilustración 14

Distribución de carpetas del proyecto - React se observa la distribución de carpetas y módulos del proyecto React, en el cual es importante destacar la lógica de acceso a la API (`api_client.js`, `api.js`), la gestión del experimento (`app_graphql.js`, `app.js`) y el diseño visual (`dashboard.js`).

Conexión a las APIs

Ilustración 15

Conexión - API REST

```
src > # apijs > ...
1 import axios from "axios";
2
3 const BASE_URL = "http://localhost:5001";
4
5 export default {
6   fetchUsuarios: async (numUsuarios) => {
7     try {
8       const response = await axios.get(`${BASE_URL}/usuarios/${numUsuarios}`);
9       return response.data;
10    } catch (error) {
11      console.error('Error al obtener usuarios:', error);
12      throw error;
13    }
14  },
15   fetchUsuariosconPublicacion: async(numUsuarios) => {
16     try {
17       const response = await axios.get(`${BASE_URL}/usuarios/${numUsuarios}`);
18       return response.data;
19     } catch (error) {
20       console.error('Error al obtener usuarios:', error);
21       throw error;
22     }
23  },
24   fetchPublicaciones: async (idUserio, numPublicaciones) => {
25     try {
26       const response = await axios.get(`${BASE_URL}/publicaciones/${idUserio}/${numPublicaciones}`);
27       return response.data;
28     } catch (error) {
29       console.error('Error al obtener publicaciones${idUserio} ? ` para el usuario ${idUserio} ` : ""', error);
30       throw error;
31     }
32  },
33 }
```

La Ilustración 15

Conexión - API REST muestra la obtención de datos desde la API REST mediante el uso de Axios. En esta sección únicamente se realiza la solicitud y recepción de los datos, sin que se aplique ningún tipo de procesamiento sobre ellos.

Ilustración 16

Conexión - API GraphQL

```
src > # api_graphqljs > @ default > @ fetchUsuarios
1 import { ApolloClient, InMemoryCache, gql } from '@apollo/client';
2
3 // Configuración del cliente Apollo
4 const client = new ApolloClient({
5   uri: 'http://localhost:5003/graphql', // URL de la API GraphQL
6   cache: new InMemoryCache(),
7 });
8
9 export default {
10   fetchUsuarios: async (numUsuarios) => {
11     try {
12       const query = gql`
13         query GetUsuariosCantidad($numUsr: Int!) {
14           getUsuariosCantidad(numUsr: $numUsr) {
15             id_usuario
16             nombre
17             correo
18             clave
19           }
20         }
21       `;
22       const { data } = await client.query({
23         query,
24         variables: { numUsr: parseInt(numUsuarios, 10) },
25         fetchPolicy: 'no-cache'
26       });
27       return data.getUsuariosCantidad;
28     } catch (error) {
29       console.error('Error al obtener usuarios:', error);
30       throw error;
31     }
32   },
33 }
```

La Ilustración 16

Conexión - API GraphQL muestra cómo se realiza la obtención de datos desde la API GraphQL.

En esta sección es importante destacar que se configura Apollo Server para que no utilice la caché durante las consultas.

Controlador del experimento

Ilustración 17

Fragmento de código - Consulta en cascada

```
75 // Otros casos de uso (lógica existente)
76 const partesDistribucion = distribucion.split('x').map(Number);
77 const [usuarios, publicaciones, comentarios, respuestas, reacciones] = partesDistribucion;
78
79 // Imprime la distribución antes de medir el tiempo
80 console.log("Distribución actual: ${distribucion}");
81 console.log("Usuarios: ${usuarios}, Publicaciones: ${publicaciones}, Comentarios: ${comentarios}, Respuestas: ${respuestas}, Reacciones: ${reacciones}");
82
83 let usuariosData, publicacionesData, comentariosData, respuestasData, reaccionesData;
84
85 const inicioConsulta = performance.now();
86 try {
87   usuariosData = await DataController.fetchUsuarios(usuarios);
88
89   await Promise.all(
90     usuariosData.map(async (usuario) => {
91       if (publicaciones) {
92         publicacionesData = await DataController.fetchPublicaciones(usuario.id_usuario, publicaciones);
93
94         await Promise.all(
95           publicacionesData.map(async (publicacion) => {
96             if (comentarios) {
97               comentariosData = await DataController.fetchComentarios(publicacion.id_publicacion, comentarios);
98
99               await Promise.all(
100                 comentariosData.map(async (comentario) => {
101                   if (respuestas) {
102                     respuestasData = await DataController.fetchRespuestas(comentario.id_comentario, respuestas);
103
104                     await Promise.all(
105                       respuestasData.map(async (respuesta) => {
106                         if (reacciones) {
107                           reaccionesData = await DataController.fetchReacciones(respuesta.id_respuesta, reacciones);
108                         }
109                       })
110                     );
111                   }
112                 })
113               );
114             }
115           })
116         );
117       }
118     })
119   );
120 }
```

El código que se visualiza en la **Ilustración 17**

Fragmento de código - Consulta en cascada demuestra como se controlan los datos y se los consulta en cascada para poder cumplir con cada caso de uso.

Exportación de resultados

Ilustración 18

Fragmento de código – Obtener resultados

```
128 // Imprime los resultados después de medir el tiempo
129 console.log('Usuarios consultados:', usuariosData);
130 if (publicacionesData) console.log('Publicaciones:', publicacionesData);
131 if (comentariosData) console.log('Comentarios:', comentariosData);
132 if (respuestasData) console.log('Respuestas:', respuestasData);
133 if (reaccionesData) console.log('Reacciones:', reaccionesData);
134
135 const nuevoRegistro = {
136   "Caso de Uso": selectedCase,
137   "Arquitectura": "REST",
138   "Tecnología": "JavaScript y React JS",
139   "Distribución": distribucion,
140   "Número de Registros": numeroRegistros,
141   "Iteración": iteracion + 1,
142   "Tiempo (ms)": tiempoConsulta.toFixed(2),
143 };
144
145 setDatosExcel(prevDatos => [...prevDatos, nuevoRegistro]);
```

La Ilustración 18

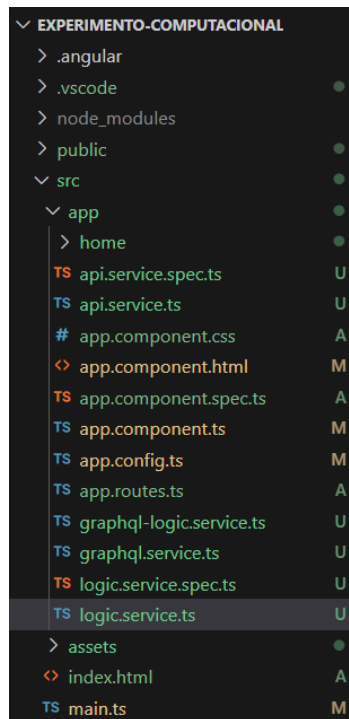
Fragmento de código – Obtener resultados muestra el fragmento de código en donde se guardan los datos para luego poder crear un archivo Excel con los resultados del experimento.

2.2.3 Angular

Estructura del proyecto

Ilustración 19

Distribución de carpetas del proyecto - Angular



En la Ilustración 19

Distribución de carpetas del proyecto - Angular se visualiza la distribución de carpetas y módulos del proyecto Angular, el cual contiene la lógica de acceso a la API (`api.service.ts`, `graphql.service.ts`), la gestión del experimento (`logic.service.ts`, `graphql-logic.service.ts`) y el diseño visual (`app.component.ts`).

Conexión a las APIs

Ilustración 20

Conexión - API REST

```
src > app > ts - api.service.ts > ApiService
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Observable } from 'rxjs';
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class ApiService {
9   private baseUrl = 'http://localhost:5001'; // URL de tu API
10
11   constructor(private http: HttpClient) {}
12
13   getUsuariosCantidad(numUsr: number): Observable<any> {
14     return this.http.get(`${this.baseUrl}/usuarios/${numUsr}`);
15   }
16
17   getPublicacionesCantidad(idUsr: number, numPst: number): Observable<any> {
18     return this.http.get(`${this.baseUrl}/publicaciones/${idUsr}/${numPst}`);
19   }
20
21   getComentariosCantidad(idPub: number, numCmt: number): Observable<any> {
22     return this.http.get(`${this.baseUrl}/comentarios/${idPub}/${numCmt}`);
23   }
24
25   getRespuestasCantidad(idCmt: number, numRes: number): Observable<any> {
26     return this.http.get(`${this.baseUrl}/respuestas/${idCmt}/${numRes}`);
27   }
28
29   getReaccionesCantidad(idRes: number, numReac: number): Observable<any> {
30     return this.http.get(`${this.baseUrl}/reacciones/${idRes}/${numReac}`);
31   }
32 }
```

En la Ilustración 20

Conexión - API REST se observa el código para obtener los datos de la API REST, se utiliza la librería HttpClient para poder realizar estas consultas.

Ilustración 21

Conexión - API GraphQL

```
src > app > TS graphqlService.ts >
4
5 @Injectable({
6   providedIn: 'root',
7 })
8 export class GraphQLService {
9   private graphqlUrl = 'http://localhost:5003/graphql'; // URL de tu API GraphQL
10
11   constructor(private http: HttpClient) { }
12
13   executeQuery(query: string, variables: any = {}): Observable<any> {
14     return this.http.post(this.graphqlUrl, {
15       query,
16       variables
17     });
18   }
19
20   getUsuariosCantidad(numUsr: number): Observable<any> {
21     const query = `
22       query GetUsuariosCantidad($numUsr: Int!) {
23         getUsuariosCantidad(numUsr: $numUsr) {
24           id_usuario
25           nombre
26           correo
27           clave
28         }
29       }
30     `;
31     return this.executeQuery(query, { numUsr });
32   }
33 }
```

La Ilustración 21

Conexión - API GraphQL contiene un fragmento de código en el cual demuestra como se está consultado los datos desde la API GraphQL utilizando la librería HttpClient.

Controlador del experimento

Ilustración 22

Fragmento de código - Consulta en cascada

```
8 export class LogicService {
68 private async consultarDistribucion(distribucion: string) {
69     const niveles = distribucion.split('x').map(Number);
70
71     if (niveles.length === 1) {
72         //console.log('Consultando ${niveles[0]} usuarios...');
73         await this.apiService.getUsuariosCantidad(niveles[0]).toPromise();
74     } else {
75         //console.log('Consultando ${niveles[0]} usuarios con publicaciones...');
76         let usuarios = await this.apiService.getUsuariosCantidad(niveles[0]).toPromise();
77         //console.log('Usuarios obtenidos:', usuarios);
78
79         for (let i = 1; i < niveles.length; i++) {
80             const nivel = niveles[i];
81             switch (i) {
82                 case 1:
83                 //console.log('Obteniendo ${nivel} publicaciones por usuario...');
84                 usuarios = await Promise.all(
85                     usuarios.map(async (usr: any) => {
86                         if (!usr || !usr.id_usuario) {
87                             //console.error('Error: Usuario inválido:', usr);
88                             return [];
89                         }
90                         //console.log('Consultando publicaciones del usuario ${usr.id_usuario}...');
91                         const publicaciones = await this.apiService.getPublicacionesCantidad(usr.id_usuario, nivel).toPromise();
92                         //console.log('Publicaciones obtenidas para el usuario ${usr.id_usuario}:', publicaciones);
93                         return publicaciones;
94                     })
95                 );
96                 break;
97
98                 case 2:
99                 //console.log('Obteniendo ${nivel} comentarios por publicación...');
100                 usuarios = await Promise.all(
101                     usuarios.flat().map(async (pub: any) => {
102                         if (!pub || !pub.id_publicacion) {
103                             console.error('Error: Publicación inválida:', pub);

```

En la Ilustración 22

Fragmento de código - Consulta en cascada se observa la forma en cómo se procesan los datos, se demuestra que se utiliza una consulta en cascada. Varios console.log están comentados ya que los mismos afectan el rendimiento al momento de ejecutar los casos de uso (en este proyecto). Solo se los utilizó para comprobar que se estén obteniendo bien los datos.

Exportación de resultados

Ilustración 23

Fragmento de código – Obtener resultados

```
for (let iteracion = 1; iteracion <= 3; iteracion++) {
  console.log(`Iteración ${iteracion} para distribución ${distribucion}...`);

  const inicio = performance.now();
  await this.consultarDistribucion(distribucion);
  const fin = performance.now();
  const tiempo = fin - inicio;

  console.log(`Tiempo medido: ${tiempo.toFixed(2)} ms`);

  // Agregar datos al Excel
  this.excelData.push({
    CasoDeUso: casoUso,
    Arquitectura: 'REST',
    Tecnología: 'TypeScript y Angular',
    Distribución: distribucion,
    NumeroDeRegistros: numeroDeRegistros, // Usa el número de registros fijo
    Iteración: iteracion,
    Tiempo: tiempo.toFixed(2)
  });

  resultados.push(`Distribución ${distribucion}, Iteración ${iteracion}, Tiempo: ${tiempo.toFixed(2)} ms`);
}

console.log(`Finalizado procesamiento para distribución: ${distribucion}\n`);

// Generar o actualizar el archivo Excel
this.generarOActualizarExcel();

return resultados;
}
```

La Ilustración 23

Fragmento de código – Obtener resultados contiene un fragmento de código el cual tiene la funcionalidad de guardar los datos obtenidos de los resultados para luego crear un archivo Excel.

2.3 Obtención de resultados

2.3.1 Estandarización del entorno

Se toman varias medidas para estandarizar el entorno de pruebas y garantizar la validez de los resultados. Primero, todo el experimento fue ejecutado en la misma máquina. Cada aplicación fue evaluada utilizando el navegador Firefox, sin complementos instalados y con el almacenamiento en caché desactivado. Además, se refrescó el navegador al finalizar la

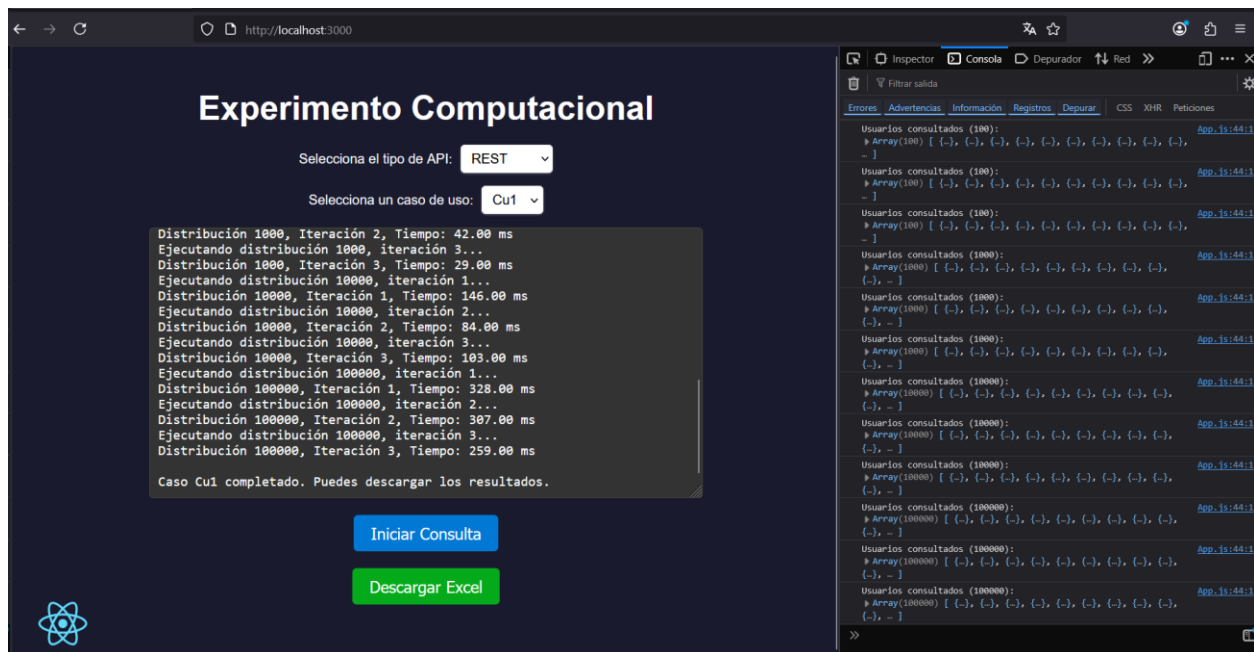
ejecución del Caso N°3, y se utilizó una API a la vez, evitando así interferencias que pudieran alterar el rendimiento del experimento computacional.

2.3.2 Ejecución del experimento computacional

Una vez estandarizado el entorno de pruebas para todos los proyectos se procede con la ejecución de un proyecto a la vez siguiendo las directrices.

Ilustración 24

Ejecución experimento computacional - React



Para tener una visión más clara del funcionamiento del experimento computacional, la

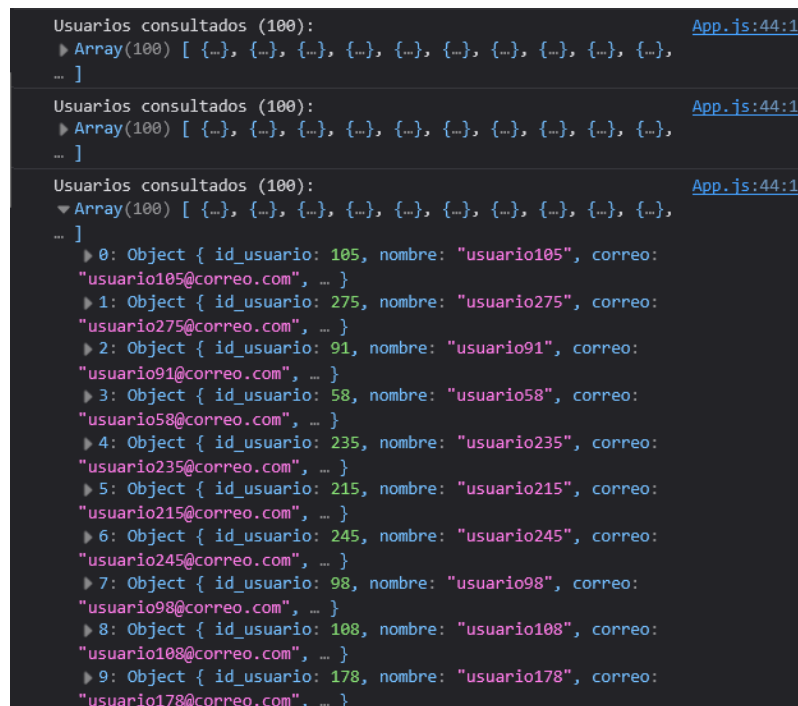
Ilustración 24

Ejecución experimento computacional - React muestra la ejecución del Caso N.º 1 utilizando React. En esta interfaz, se emplea un primer *combo box* para seleccionar la API a consumir (REST o GraphQL) y un segundo *combo box* para elegir el caso de uso a ejecutar. Los tiempos

de ejecución de cada distribución se visualizan en un campo de tipo *text area*, mientras que los datos obtenidos se imprimen en la consola para verificar que la consulta se haya realizado correctamente. Adicionalmente tenemos un botón de “Descargar Excel” que tiene la funcionalidad de guardar los resultados de los casos de uso ejecutados en un archivo Excel.

Ilustración 25

Datos obtenidos en consola - React



```
Usuarios consultados (100): App.js:44:1
  ▶ Array(100) [ {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
  ... ]

Usuarios consultados (100): App.js:44:1
  ▶ Array(100) [ {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
  ... ]

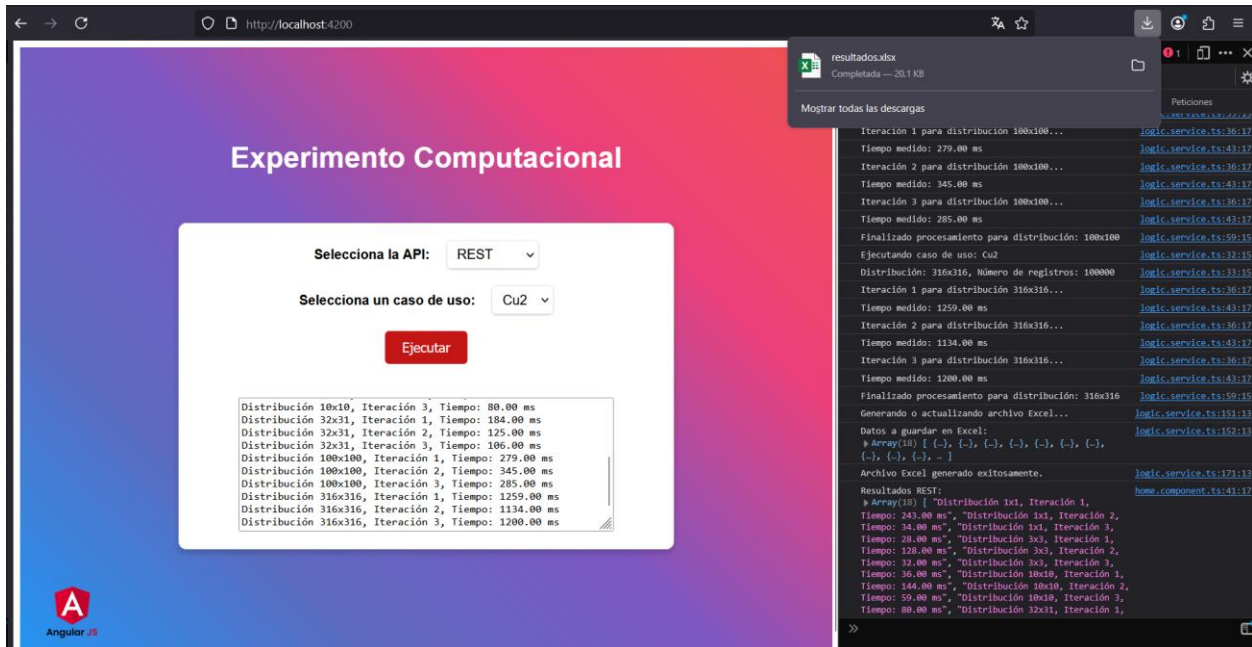
Usuarios consultados (100): App.js:44:1
  ▼ Array(100) [ {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
  ... ]
    ▶ 0: Object { id_usuario: 105, nombre: "usuario105", correo:
      "usuario105@correo.com", ... }
    ▶ 1: Object { id_usuario: 275, nombre: "usuario275", correo:
      "usuario275@correo.com", ... }
    ▶ 2: Object { id_usuario: 91, nombre: "usuario91", correo:
      "usuario91@correo.com", ... }
    ▶ 3: Object { id_usuario: 58, nombre: "usuario58", correo:
      "usuario58@correo.com", ... }
    ▶ 4: Object { id_usuario: 235, nombre: "usuario235", correo:
      "usuario235@correo.com", ... }
    ▶ 5: Object { id_usuario: 215, nombre: "usuario215", correo:
      "usuario215@correo.com", ... }
    ▶ 6: Object { id_usuario: 245, nombre: "usuario245", correo:
      "usuario245@correo.com", ... }
    ▶ 7: Object { id_usuario: 98, nombre: "usuario98", correo:
      "usuario98@correo.com", ... }
    ▶ 8: Object { id_usuario: 108, nombre: "usuario108", correo:
      "usuario108@correo.com", ... }
    ▶ 9: Object { id_usuario: 178, nombre: "usuario178", correo:
      "usuario178@correo.com", ... }
```

La Ilustración 25

Datos obtenidos en consola - React presenta el resultado de los datos obtenidos en el Caso de Uso N.º 1 para la distribución de 100 datos. Dado que cada distribución se ejecuta tres veces, en la consola se visualizan tres arreglos (*arrays*). Al revisar estos arreglos, se puede verificar si las consultas están devolviendo los datos correctamente, conforme a la definición establecida para cada caso de uso.

Ilustración 26

Ejecución experimento computacional - Angular



La ejecución del experimento computacional en Angular la podemos observar en la

Ilustración 26

Ejecución experimento computacional - Angular, aquí se muestra la ejecución del Caso de uso N°2. En esta interfaz tenemos un *combo box* para elegir que API consumir (REST o GraphQL) y un *combo box* para elegir el caso de uso a ejecutar. Los tiempos de ejecución de cada distribución se visualizan en un campo de tipo *text area*, sin embargo, a diferencia del proyecto en React, en este caso no se imprimen los datos obtenidos en la consola. Esto se debe a que, en Angular, la impresión de datos afecta directamente al tiempo de ejecución. Por lo tanto, si se desea visualizar los datos consultados (como se hace en React), es necesario descomentar las líneas de código correspondientes, las cuales se muestran en la **Ilustración 22**

Fragmento de código - Consulta en cascada Adicionalmente, en este proyecto después de

ejecutar un caso de uso automáticamente descarga el documento Excel con los resultados obtenidos.

Ilustración 27

Datos obtenidos en consola - Angular

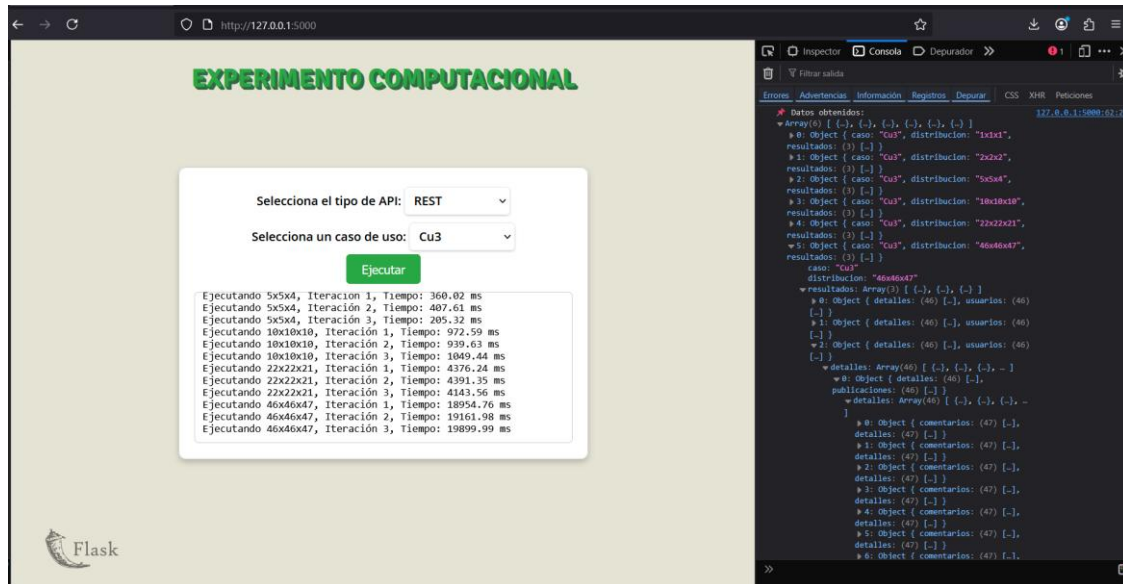
```
Iteración 1 para distribución 316x316... logic.service.ts:36:17
Tiempo medido: 1259.00 ms logic.service.ts:43:17
Iteración 2 para distribución 316x316... logic.service.ts:36:17
Tiempo medido: 1134.00 ms logic.service.ts:43:17
Iteración 3 para distribución 316x316... logic.service.ts:36:17
Tiempo medido: 1200.00 ms logic.service.ts:43:17
Finalizado procesamiento para distribución: 316x316 logic.service.ts:59:15
Generando o actualizando archivo Excel... logic.service.ts:151:13
Datos a guardar en Excel: logic.service.ts:152:13
  » Array(18) [ {...}, {...}, {...}, {...}, {...}, {...}, {...},
  {...}, {...}, {...}, ... ]
Archivo Excel generado exitosamente. logic.service.ts:171:13
```

En el proyecto desarrollado con Angular, la consola del navegador cumple una función informativa. Su propósito es mostrar, durante la ejecución del experimento, detalles como la iteración actual, la distribución que se está procesando y los tiempos correspondientes. Al finalizar, también indica los resultados que han sido almacenados en el archivo Excel. Esta funcionalidad se puede observar en la **Ilustración 27**

Datos obtenidos en consola - Angular

Ilustración 28

Ejecución experimento computacional - Flask

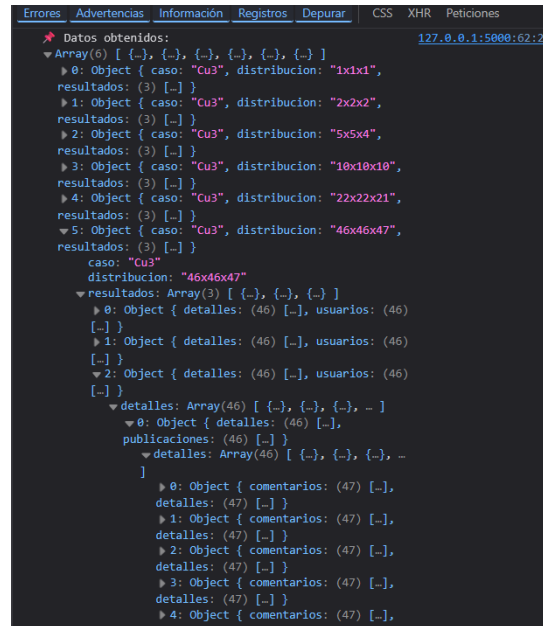


En la Ilustración 28

Ejecución experimento computacional - Flask podemos observar la ejecución del experimento computacional Flask, aquí se muestra la ejecución del Caso de uso N°3. La interfaz incluye un *combo box* para elegir que API consumir (REST o GraphQL) y un *combo box* para elegir el caso de uso a ejecutar. Los tiempos de ejecución de cada distribución se visualizan en un campo de tipo *text area*, en este caso si se imprimen los datos obtenidos en la consola ya que estos no afectan al tiempo de ejecución porque se imprimen después de ejecutarse el caso de uso. El archivo Excel se genera automáticamente en la carpeta del proyecto con los resultados ejecutados.

Ilustración 29

Datos obtenidos en consola - Flask



```
Errores  Advertencias  Información  Registros  Depurar  CSS  XHR  Peticiones
★ Datos obtenidos: 127.0.0.1:5000:62:29
▼ Array(6) [ {}, {}, {}, {}, {}, {} ]
  ▶ 0: Object { caso: "Cu3", distribucion: "1x1x1", resultados: (3) [ - ] }
  ▶ 1: Object { caso: "Cu3", distribucion: "2x2x2", resultados: (3) [ - ] }
  ▶ 2: Object { caso: "Cu3", distribucion: "5x5x4", resultados: (3) [ - ] }
  ▶ 3: Object { caso: "Cu3", distribucion: "10x10x10", resultados: (3) [ - ] }
  ▶ 4: Object { caso: "Cu3", distribucion: "22x22x21", resultados: (3) [ - ] }
  ▶ 5: Object { caso: "Cu3", distribucion: "46x46x47", resultados: (3) [ - ] }
    caso: "Cu3"
    distribucion: "46x46x47"
    ▼ resultados: Array(3) [ {}, {}, {} ]
      ▶ 0: Object { detalles: (46) [ - ], usuarios: (46) [ - ] }
      ▶ 1: Object { detalles: (46) [ - ], usuarios: (46) [ - ] }
      ▶ 2: Object { detalles: (46) [ - ], usuarios: (46) [ - ] }
        detalles: Array(46) [ {}, {}, {}, ..., {} ]
          ▶ 0: Object { detalles: (46) [ - ], publicaciones: (46) [ - ] }
            detalles: Array(46) [ {}, {}, {}, ..., {} ]
              ▶ 0: Object { comentarios: (47) [ - ], detalles: (47) [ - ] }
              ▶ 1: Object { comentarios: (47) [ - ], detalles: (47) [ - ] }
              ▶ 2: Object { comentarios: (47) [ - ], detalles: (47) [ - ] }
              ▶ 3: Object { comentarios: (47) [ - ], detalles: (47) [ - ] }
              ▶ 4: Object { comentarios: (47) [ - ], detalles: (47) [ - ] }
```

Los datos impresos en la consola del proyecto Flask se pueden observar en la

Ilustración 29

Datos obtenidos en consola - Flask. En esta sección, los datos se almacenan en arreglos y se subdividen según la distribución correspondiente, lo que permite visualizar de forma detallada la información consultada en cada ejecución.

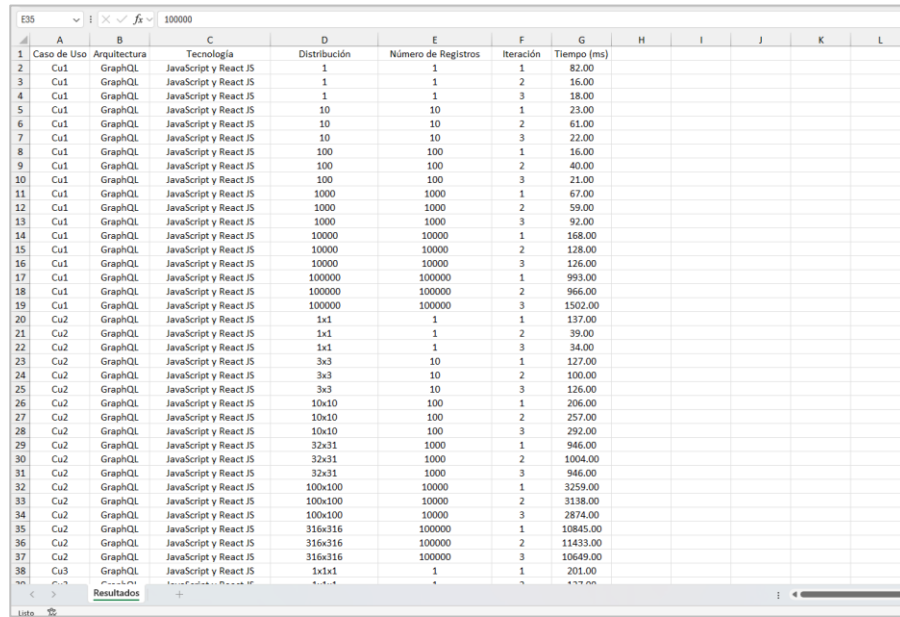
2.3.3 Resultados

Los resultados de todos los proyectos se almacenan en un archivo Excel, lo que facilita su análisis posterior y permite identificar cuáles fueron los tiempos de consulta más eficientes.

El documento que se genera en cada proyecto tiene la misma estructura, la diferencia es la información que se guarda en la columna de arquitectura y tecnología ya que esta varía dependiendo de que API se está consumiendo y desde que frontend.

Ilustración 30

Documento Excel de los resultados - GraphQL



#	A	B	C	D	E	F	G	H	I	J	K	L
1	Caso de Uso	Arquitectura	Tecnología	Distribución	Número de Registros	Iteración	Tiempo (ms)					
2	Cu1	GraphQL	JavaScript y React JS	1	1	1	82.00					
3	Cu1	GraphQL	JavaScript y React JS	1	1	2	16.00					
4	Cu1	GraphQL	JavaScript y React JS	1	1	3	18.00					
5	Cu1	GraphQL	JavaScript y React JS	10	10	1	23.00					
6	Cu1	GraphQL	JavaScript y React JS	10	10	2	61.00					
7	Cu1	GraphQL	JavaScript y React JS	10	10	3	22.00					
8	Cu1	GraphQL	JavaScript y React JS	100	100	1	16.00					
9	Cu1	GraphQL	JavaScript y React JS	100	100	2	40.00					
10	Cu1	GraphQL	JavaScript y React JS	100	100	3	21.00					
11	Cu1	GraphQL	JavaScript y React JS	1000	1000	1	67.00					
12	Cu1	GraphQL	JavaScript y React JS	1000	1000	2	59.00					
13	Cu1	GraphQL	JavaScript y React JS	1000	1000	3	92.00					
14	Cu1	GraphQL	JavaScript y React JS	10000	10000	1	168.00					
15	Cu1	GraphQL	JavaScript y React JS	10000	10000	2	128.00					
16	Cu1	GraphQL	JavaScript y React JS	10000	10000	3	126.00					
17	Cu1	GraphQL	JavaScript y React JS	100000	100000	1	993.00					
18	Cu1	GraphQL	JavaScript y React JS	100000	100000	2	966.00					
19	Cu1	GraphQL	JavaScript y React JS	100000	100000	3	1502.00					
20	Cu2	GraphQL	JavaScript y React JS	1x1	1	1	137.00					
21	Cu2	GraphQL	JavaScript y React JS	1x1	1	2	39.00					
22	Cu2	GraphQL	JavaScript y React JS	1x1	1	3	34.00					
23	Cu2	GraphQL	JavaScript y React JS	3x3	10	1	127.00					
24	Cu2	GraphQL	JavaScript y React JS	3x3	10	2	100.00					
25	Cu2	GraphQL	JavaScript y React JS	3x3	10	3	126.00					
26	Cu2	GraphQL	JavaScript y React JS	10x10	100	1	206.00					
27	Cu2	GraphQL	JavaScript y React JS	10x10	100	2	257.00					
28	Cu2	GraphQL	JavaScript y React JS	10x10	100	3	292.00					
29	Cu2	GraphQL	JavaScript y React JS	32x31	1000	1	946.00					
30	Cu2	GraphQL	JavaScript y React JS	32x31	1000	2	1004.00					
31	Cu2	GraphQL	JavaScript y React JS	32x31	1000	3	946.00					
32	Cu2	GraphQL	JavaScript y React JS	100x100	10000	1	3259.00					
33	Cu2	GraphQL	JavaScript y React JS	100x100	10000	2	3138.00					
34	Cu2	GraphQL	JavaScript y React JS	100x100	10000	3	2874.00					
35	Cu2	GraphQL	JavaScript y React JS	316x316	100000	1	10845.00					
36	Cu2	GraphQL	JavaScript y React JS	316x316	100000	2	11433.00					
37	Cu2	GraphQL	JavaScript y React JS	316x316	100000	3	10649.00					
38	Cu3	GraphQL	JavaScript y React JS	1x1x1	1	1	201.00					

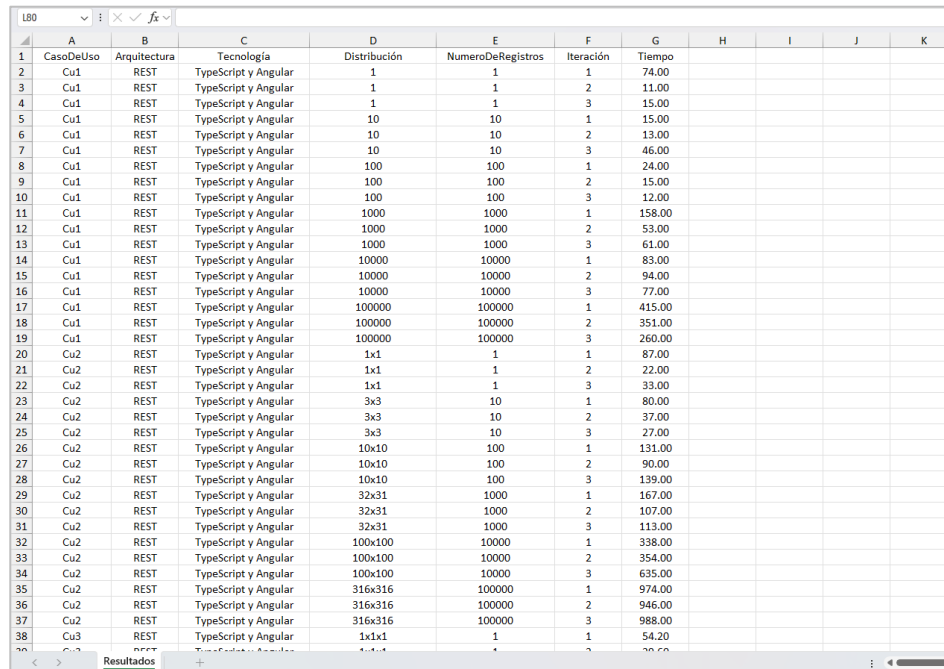
El documento generado por el experimento computacional lo podemos ver en la

Ilustración 30

Documento Excel de los resultados - . Este archivo proporciona información detallada por caso de uso y distribución. En este caso genera un documento Excel por la arquitectura GraphQL y la tecnología JavaScript y React.

Ilustración 31

Documento Excel de los resultados - REST



	A	B	C	D	E	F	G	H	I	J	K
1	CasoDeUso	Arquitectura	Tecnología	Distribución	NumeroDeRegistros	Iteración	Tiempo				
2	Cu1	REST	TypeScript y Angular	1	1	1	74.00				
3	Cu1	REST	TypeScript y Angular	1	1	2	11.00				
4	Cu1	REST	TypeScript y Angular	1	1	3	15.00				
5	Cu1	REST	TypeScript y Angular	10	10	1	15.00				
6	Cu1	REST	TypeScript y Angular	10	10	2	13.00				
7	Cu1	REST	TypeScript y Angular	10	10	3	46.00				
8	Cu1	REST	TypeScript y Angular	100	100	1	24.00				
9	Cu1	REST	TypeScript y Angular	100	100	2	15.00				
10	Cu1	REST	TypeScript y Angular	100	100	3	12.00				
11	Cu1	REST	TypeScript y Angular	1000	1000	1	158.00				
12	Cu1	REST	TypeScript y Angular	1000	1000	2	53.00				
13	Cu1	REST	TypeScript y Angular	1000	1000	3	61.00				
14	Cu1	REST	TypeScript y Angular	10000	10000	1	83.00				
15	Cu1	REST	TypeScript y Angular	10000	10000	2	94.00				
16	Cu1	REST	TypeScript y Angular	10000	10000	3	77.00				
17	Cu1	REST	TypeScript y Angular	100000	100000	1	415.00				
18	Cu1	REST	TypeScript y Angular	100000	100000	2	351.00				
19	Cu1	REST	TypeScript y Angular	100000	100000	3	260.00				
20	Cu2	REST	TypeScript y Angular	1x1	1	1	87.00				
21	Cu2	REST	TypeScript y Angular	1x1	1	2	22.00				
22	Cu2	REST	TypeScript y Angular	1x1	1	3	33.00				
23	Cu2	REST	TypeScript y Angular	3x3	10	1	80.00				
24	Cu2	REST	TypeScript y Angular	3x3	10	2	37.00				
25	Cu2	REST	TypeScript y Angular	3x3	10	3	27.00				
26	Cu2	REST	TypeScript y Angular	10x10	100	1	131.00				
27	Cu2	REST	TypeScript y Angular	10x10	100	2	90.00				
28	Cu2	REST	TypeScript y Angular	10x10	100	3	139.00				
29	Cu2	REST	TypeScript y Angular	32x31	1000	1	167.00				
30	Cu2	REST	TypeScript y Angular	32x31	1000	2	107.00				
31	Cu2	REST	TypeScript y Angular	32x31	1000	3	113.00				
32	Cu2	REST	TypeScript y Angular	100x100	10000	1	338.00				
33	Cu2	REST	TypeScript y Angular	100x100	10000	2	354.00				
34	Cu2	REST	TypeScript y Angular	100x100	10000	3	635.00				
35	Cu2	REST	TypeScript y Angular	316x316	100000	1	974.00				
36	Cu2	REST	TypeScript y Angular	316x316	100000	2	946.00				
37	Cu2	REST	TypeScript y Angular	316x316	100000	3	988.00				
38	Cu3	REST	TypeScript y Angular	1x1x1	1	1	54.20				

La Ilustración 31

Documento Excel de los resultados - muestra el documento generado por el experimento computacional. El archivo brinda información por caso de uso. En este ejemplo se genera un documento Excel con arquitectura REST y la tecnología TypeScript y Angular.

CAPITULO III

En este capítulo, se presenta el análisis de los resultados obtenidos a partir del experimento computacional, el cual fue diseñado siguiendo las directrices metodológicas propuestas por Wohlin (2012) en *Experimentation in Software Engineering*, con el fin de responder a la pregunta de investigación: ¿Qué combinación de arquitectura (GraphQL o REST) y tecnología frontend (React, Angular o Flask) ofrece un mejor desempeño en términos de eficiencia en el consumo de datos? El experimento se enfocó en comparar distintas combinaciones tecnológicas bajo escenarios controlados, utilizando casos de uso previamente definidos. La métrica de evaluación principal fue el tiempo de respuesta, en conformidad con la norma ISO/IEC 25023. El análisis de resultados se sustenta en tablas y representaciones gráficas que permiten observar y contrastar los datos recolectados, facilitando así una interpretación precisa y fundamentada.

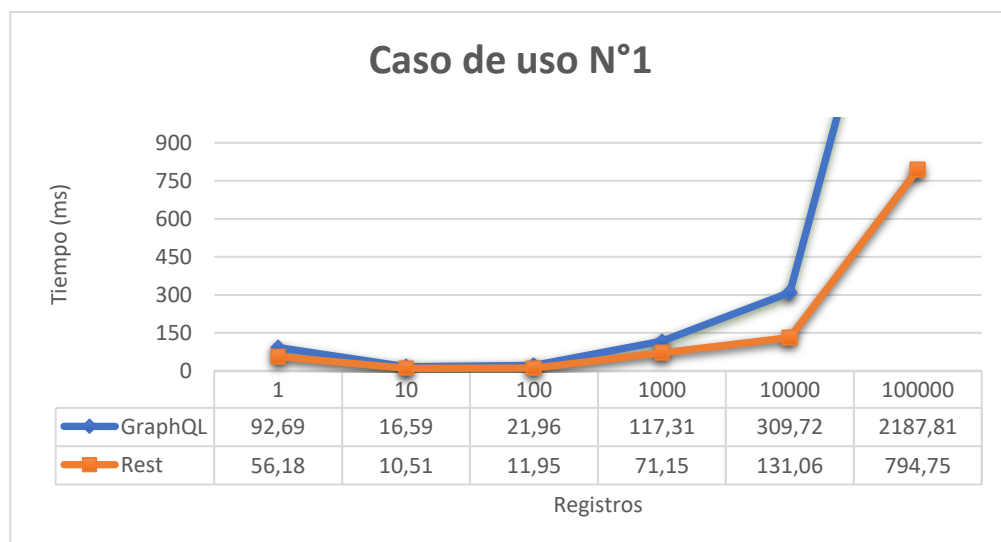
3.1 Análisis de resultados

3.1.1 Comparación por tecnología y por tipo de API.

Flask

Ilustración 32

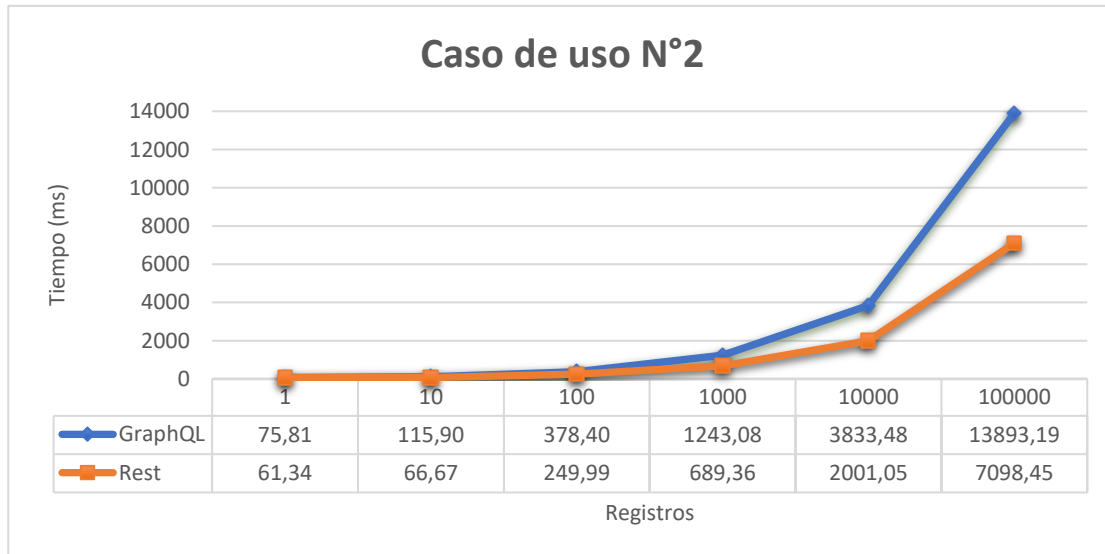
Flask - Caso de uso N°1



La **ilustración 33** muestra la gráfica de tiempos para el Caso de uso N°1

Ilustración 33

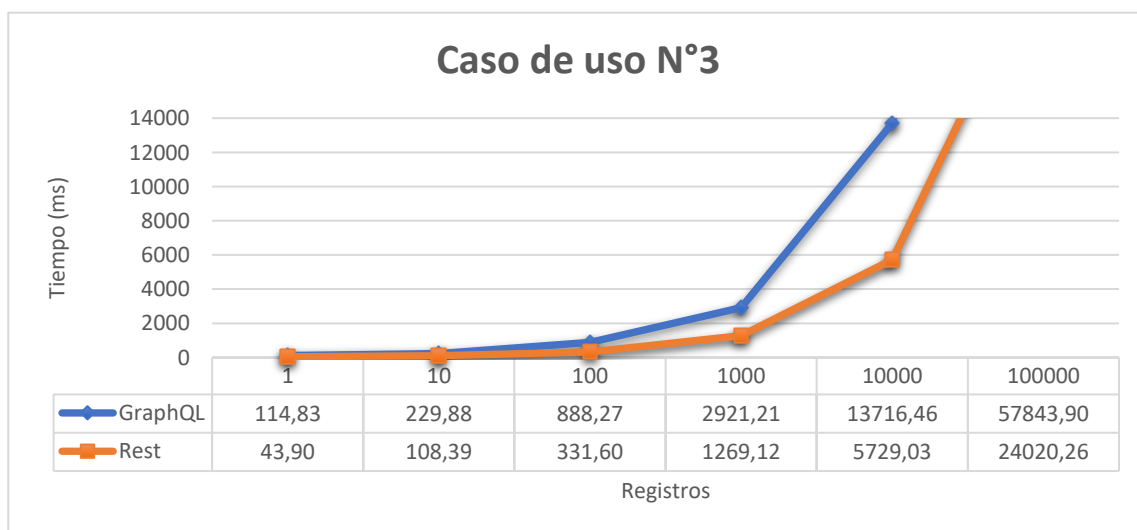
Flask - Caso de uso N°2



La **ilustración 34** muestra la gráfica de tiempos para el Caso de uso N°2

Ilustración 34

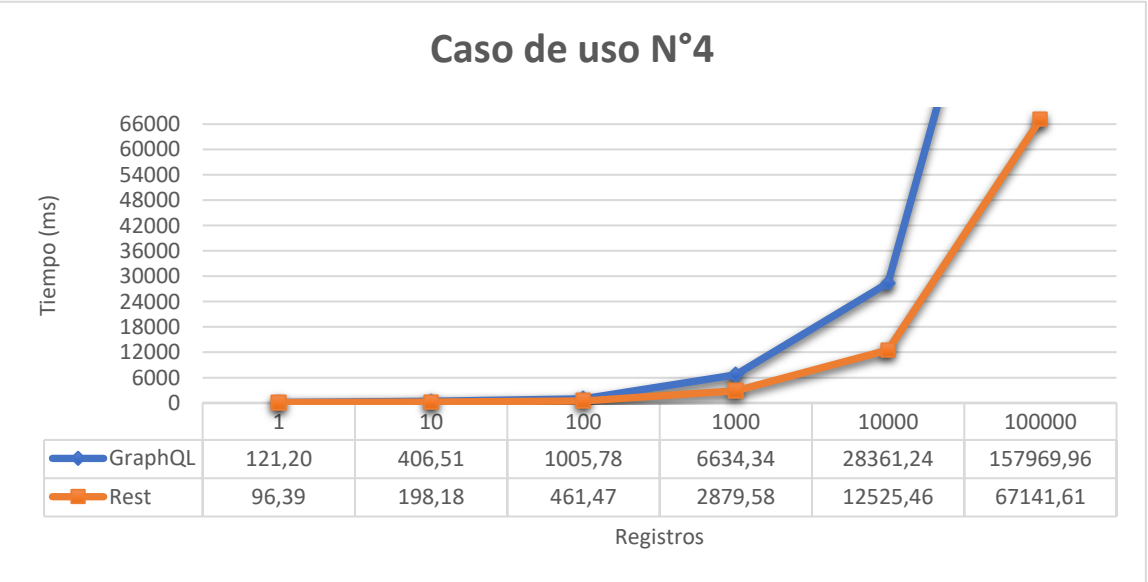
Flask - Caso de uso N°3



La **ilustración 35** muestra la gráfica de tiempos para el Caso de uso N°3

Ilustración 35

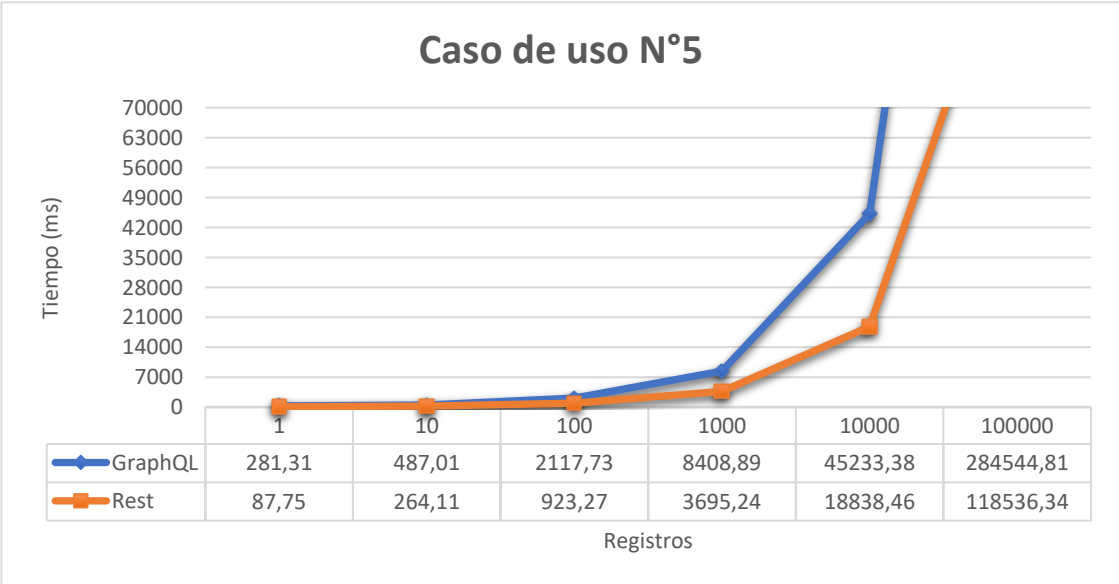
Flask - Caso de uso N°4



La **ilustración 36** muestra la gráfica de tiempos para el Caso de uso N°4

Ilustración 36

Flask - Caso de uso N°5



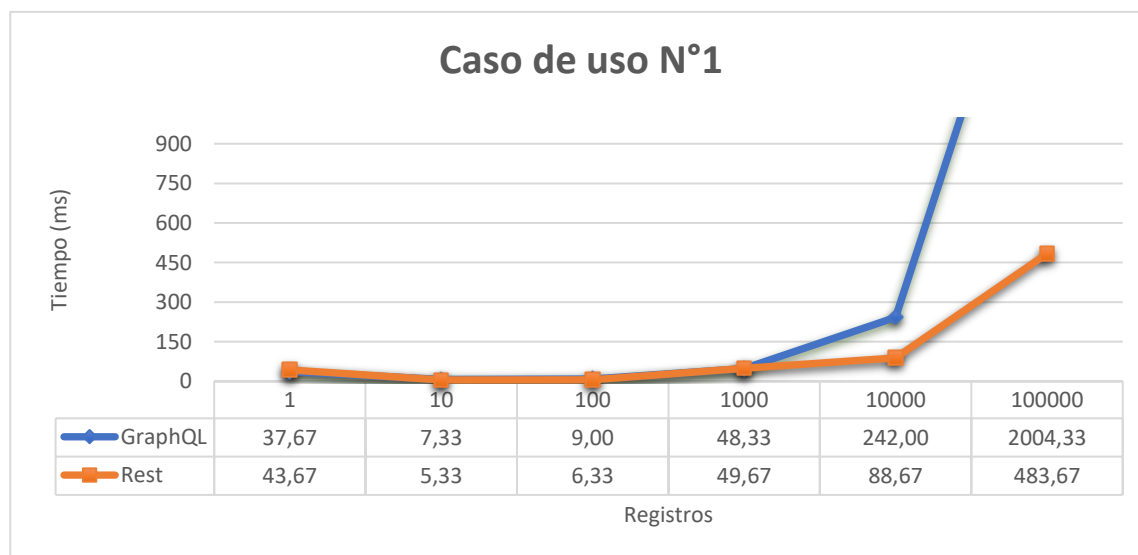
La **ilustración 37** muestra la gráfica de tiempos para el Caso de uso N°5

Según los resultados obtenidos en Flask, GraphQL comienza con tiempos de respuesta bajos, pero su rendimiento se ve afectado significativamente al aumentar la cantidad de datos y la consulta de la jerarquía de tablas. En cambio, REST se mantiene con tiempos de respuesta más bajos y en todos los niveles, lo que le da una ventaja evidente en esta situación.

React

Ilustración 37

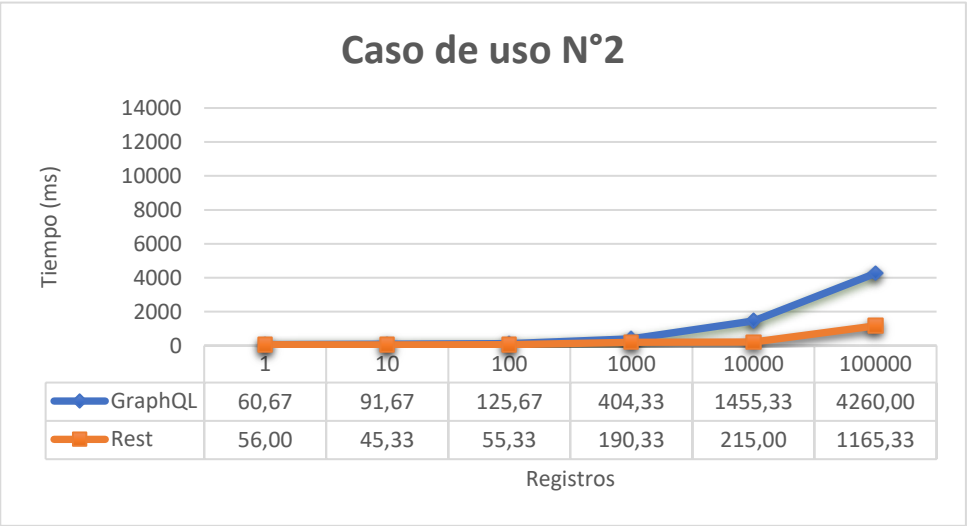
React - Caso de uso N°1



La **ilustración 38** muestra la gráfica de tiempos para el Caso de uso N°1

Ilustración 38

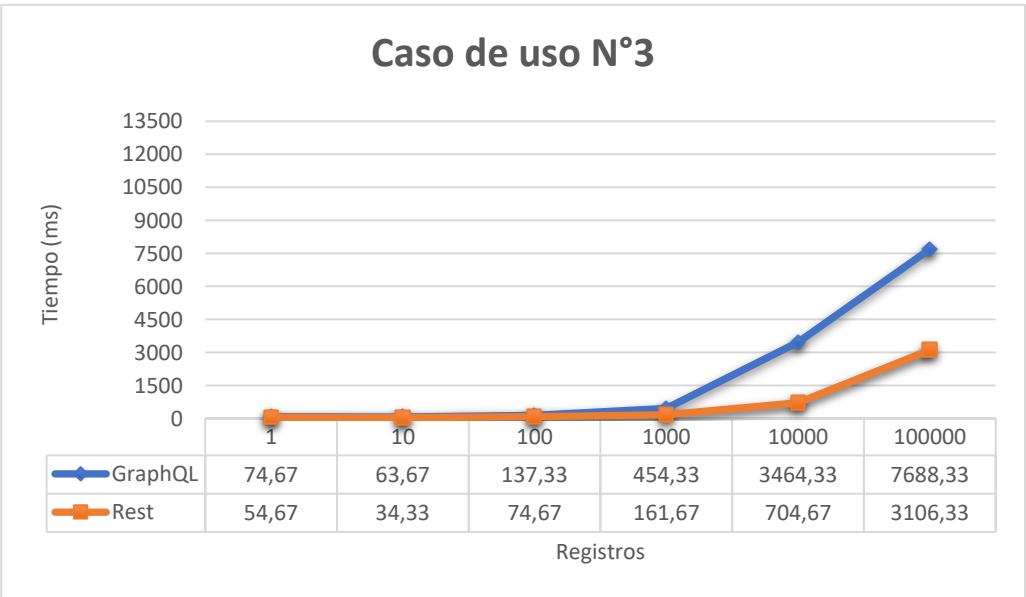
React - Caso de uso N°2



La **ilustración 39** muestra la gráfica de tiempos para el Caso de uso N°2

Ilustración 39

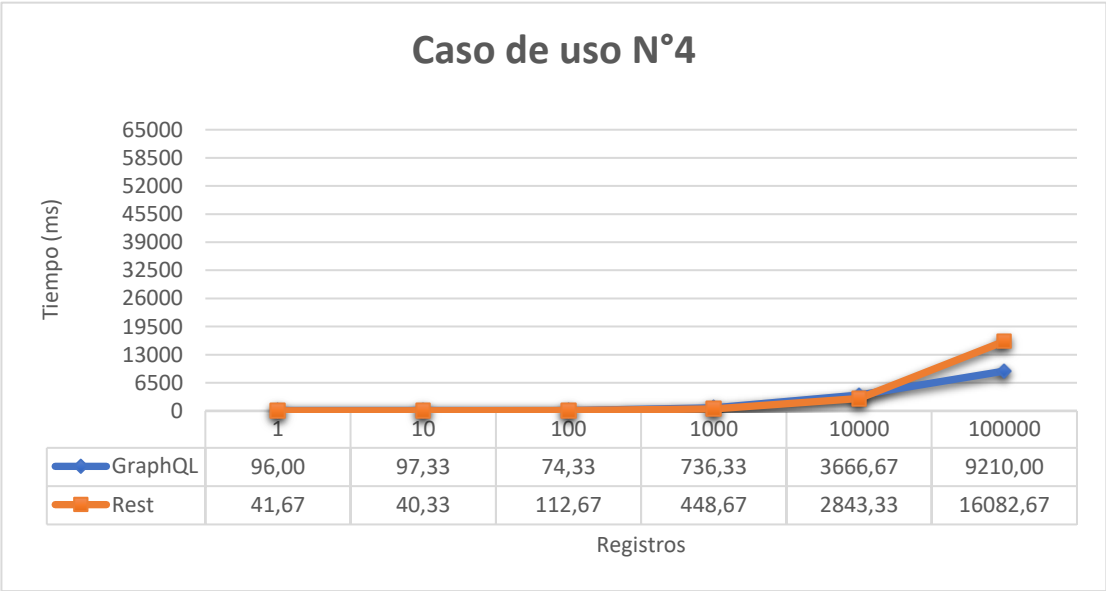
React - Caso de uso N°3



La **ilustración 40** muestra la gráfica de tiempos para el Caso de uso N°3

Ilustración 40

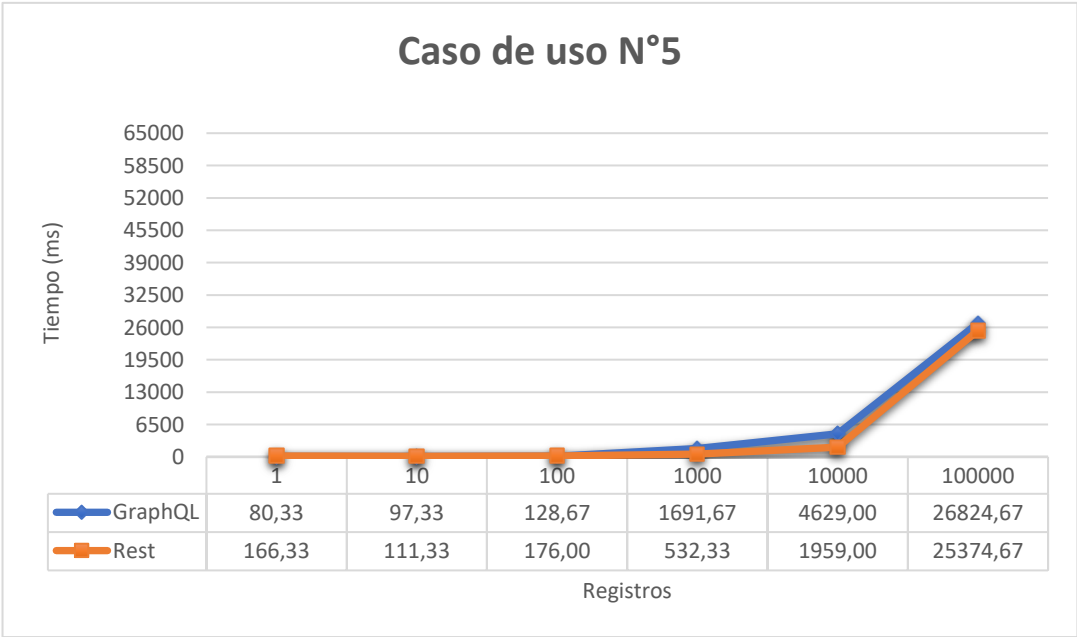
React - Caso de uso N°4



La **ilustración 41** muestra la gráfica de tiempos para el Caso de uso N°4

Ilustración 41

React - Caso de uso N°5



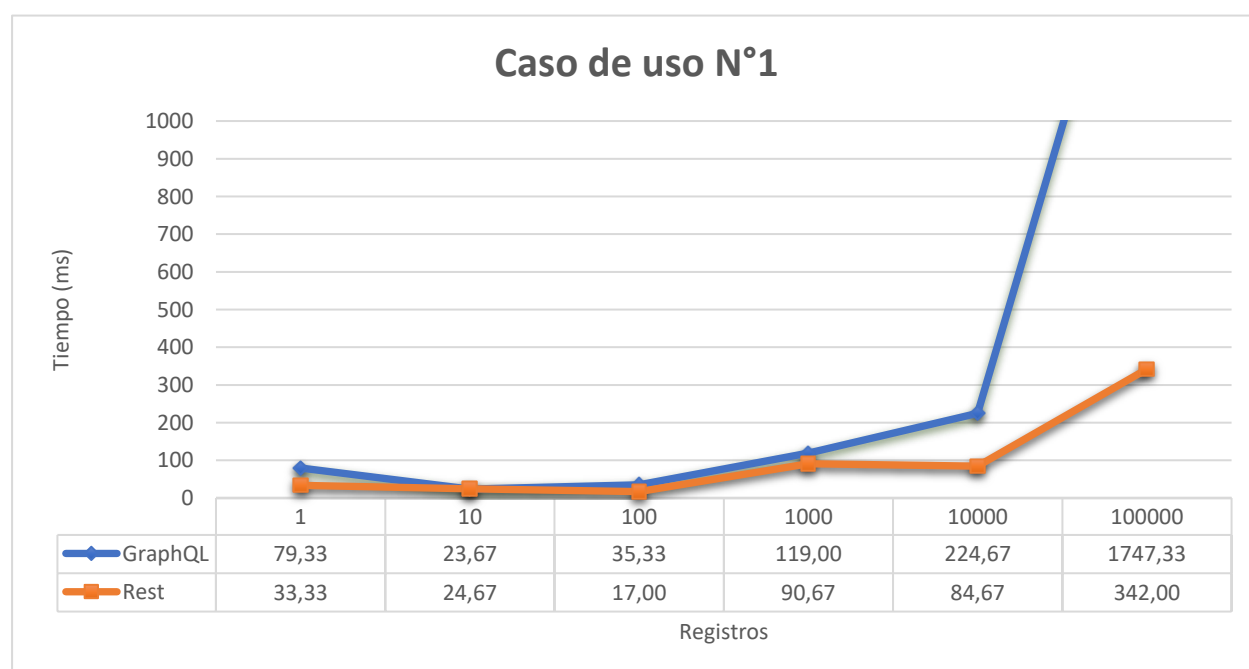
La **ilustración 42** muestra la gráfica de tiempos para el Caso de uso N°5

En React, los tiempos de respuesta entre REST y GraphQL son bastante parejos en la mayoría de los niveles. REST mantiene una ligera ventaja cuando las consultas no son tan altas, a excepción del Caso de uso N°5 que se puede visualizar en la **Ilustración 41** React - Caso de uso N°5 donde en los primeros registros GraphQL es más eficiente. Sin embargo, cuando los registros aumentan la cantidad de datos a consultar hay una tendencia a que REST presente un mejor rendimiento. Un caso particular sucede en la **Ilustración 40** React - Caso de uso N°4, donde GraphQL logra superar a REST con un menor tiempo de respuesta al manejar una consulta más grande, siendo la única excepción en esta serie de pruebas.

Angular

Ilustración 42

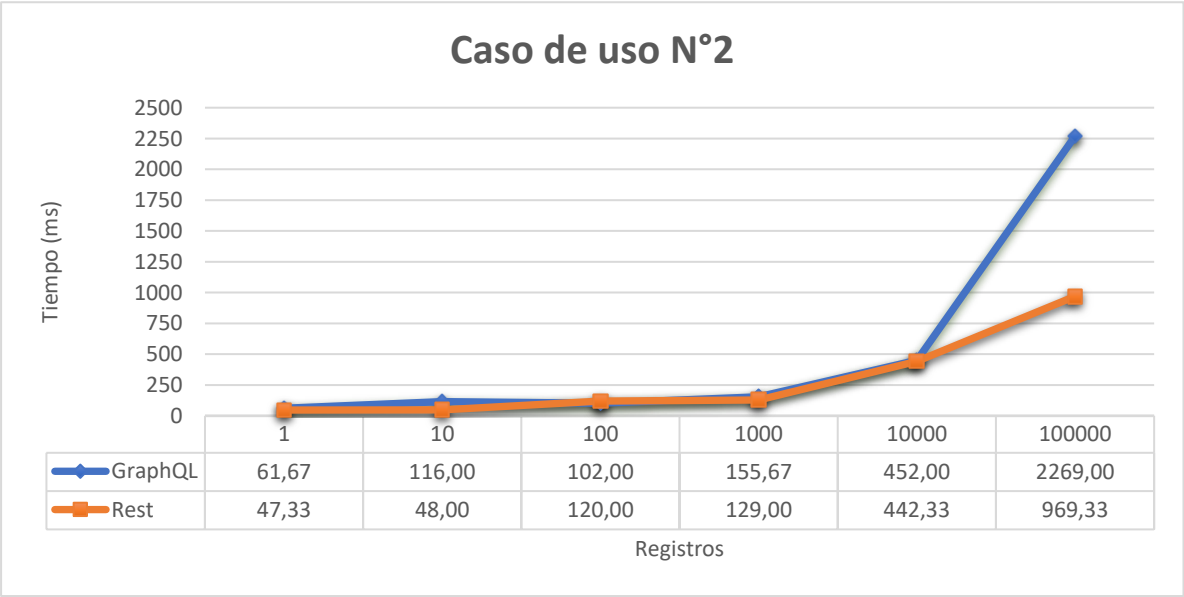
Angular - Caso de uso N°1



La **ilustración 43** muestra la gráfica de tiempos para el Caso de uso N°1

Ilustración 43

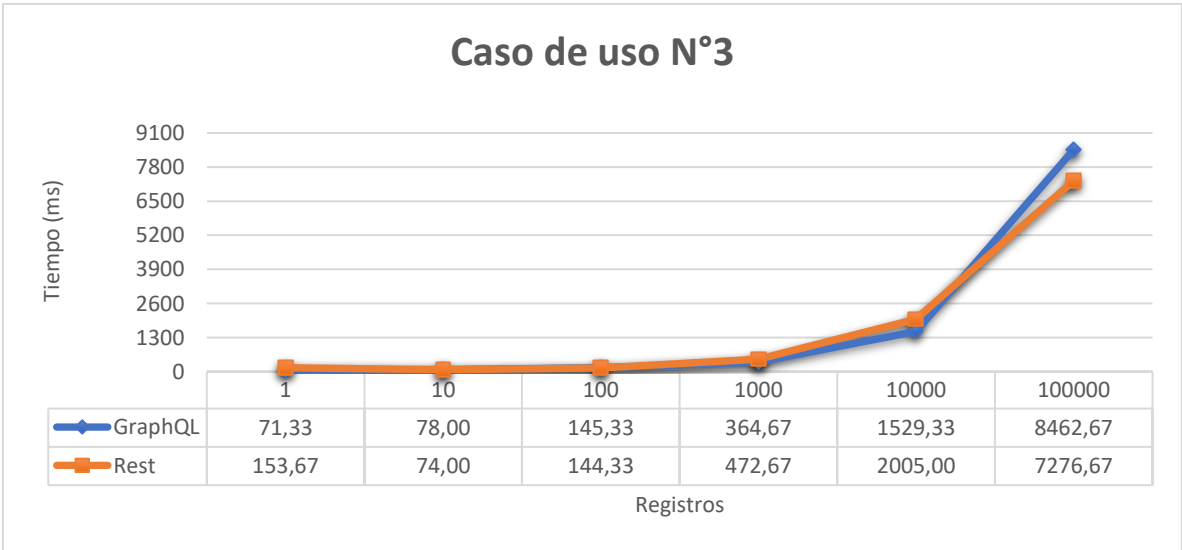
Angular - Caso de uso N°2



La **ilustración 44** muestra la gráfica de tiempos para el Caso de uso N°2

Ilustración 44

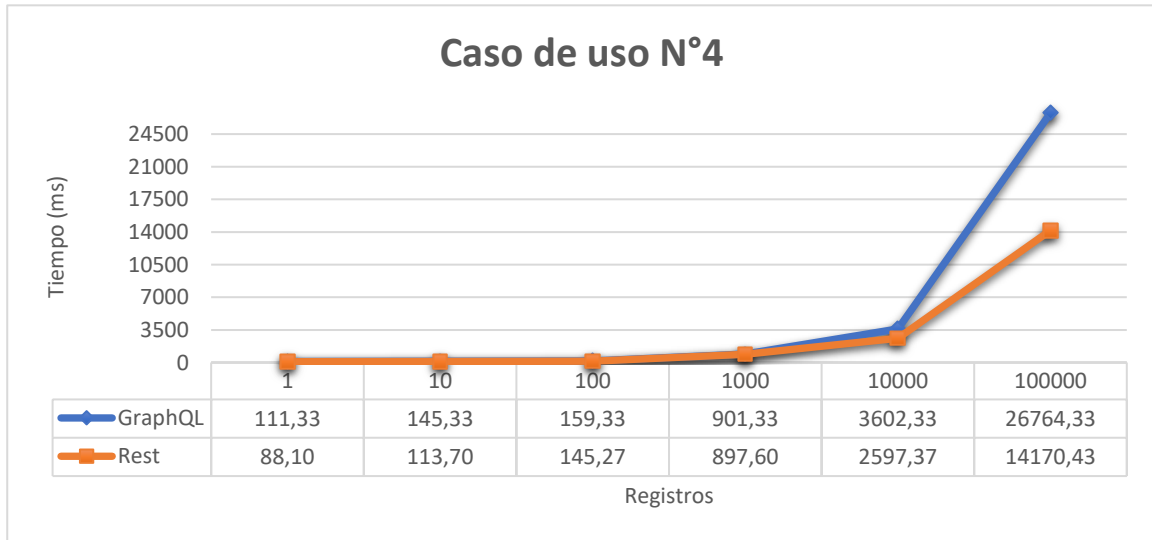
Angular - Caso de uso N°3



La **ilustración 45** muestra la gráfica de tiempos para el Caso de uso N°3

Ilustración 45

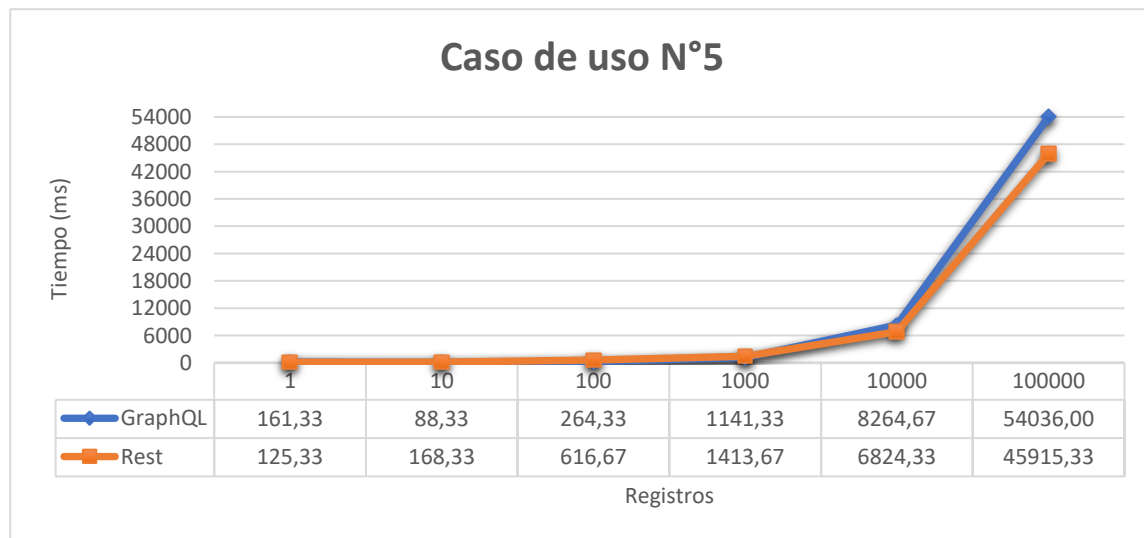
Angular - Caso de uso N°4



La **ilustración 46** muestra la gráfica de tiempos para el Caso de uso N°4

Ilustración 46

Angular - Caso de uso N°5



La **ilustración 47** muestra la gráfica de tiempos para el Caso de uso N°5

En Angular, REST mantiene tiempos de respuesta más bajos en la mayoría de las distribuciones. Sin embargo, en la **ilustración 44**

Angular - Caso de uso N°3 se puede observar una excepción interesante, donde GraphQL logra superar a REST al manejar mejor la consulta de 10000 datos. A pesar de este resultado, la tendencia general favorece a REST en cuanto a rendimiento.

3.1.2 ¿Qué tecnología se comportó mejor?

Con el fin de obtener una visión más clara sobre cuál tecnología se desempeñó mejor y logró menores tiempos de respuesta, se procederá a calcular el promedio de los tiempos registrados en los distintos casos de uso.

Resultados REST

Ilustración 47

Promedio de tiempos REST

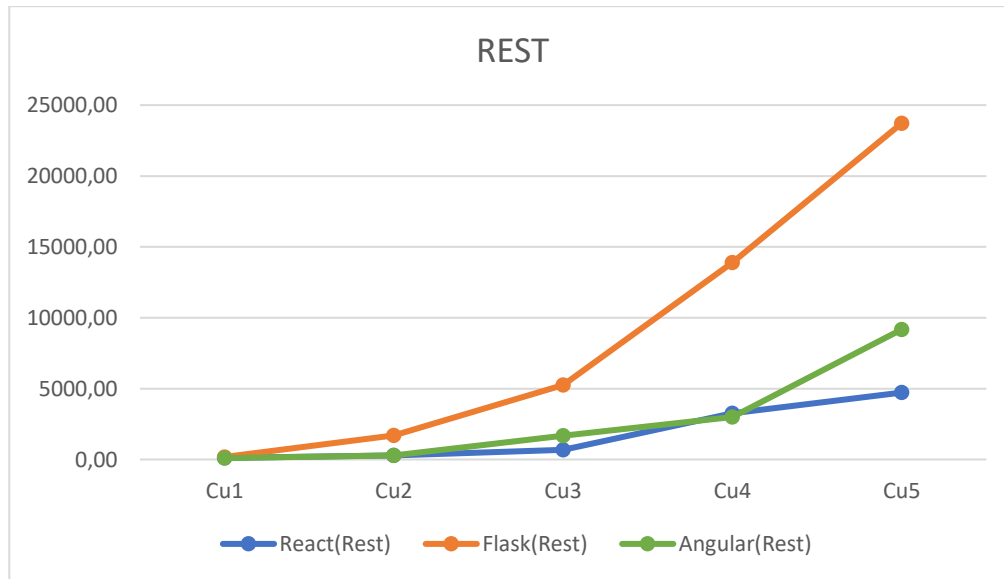


Tabla 7

Promedio de tiempos REST

	Cu1	Cu2	Cu3	Cu4	Cu5	Promedio
React	112,89	287,89	689,39	3261,56	4719,94	1814,33
Flask	179,27	1694,48	5250,39	13883,78	23724,20	8946,42
Angular	98,72	292,67	1687,72	3002,08	9177,28	2851,69

En la

Tabla 7

Promedio de tiempos REST se compara los promedios de tiempos de REST con las diferentes tecnologías. Se puede observar que existe una tendencia en la que React tiene mejor promedio de tiempos en los diferentes casos, con una excepción en el **Cu1** en el cual Angular logra un

mejor rendimiento. En contraste, Flask presenta los tiempos más elevados frente a las demás opciones.

Resultados GraphQL

Ilustración 48

Promedio de tiempos GraphQL

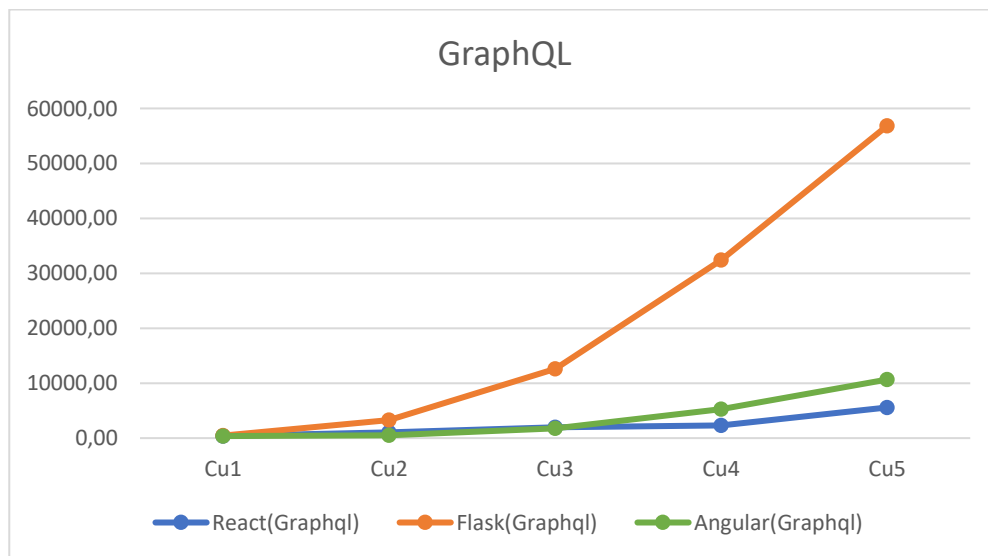


Tabla 8

Promedio de tiempos GraphQL

	Cu1	Cu2	Cu3	Cu4	Cu5	Promedio
React	391,44	1066,28	1980,44	2313,44	5575,28	2265,38
Flask	457,68	3256,64	12619,09	32416,51	56845,52	21119,09
Angular	371,56	526,06	1775,22	5280,67	10659,33	3722,57

La Tabla 8

Promedio de tiempos GraphQL presenta los promedios de tiempo utilizando GraphQL. Se puede observar que Angular obtiene mejores tiempos en los casos Cu1, Cu2 y Cu3, mientras

que React se destaca en Cu4 y Cu5. A pesar de esta distribución, el promedio general posiciona a React como la tecnología más eficiente. Flask, por su parte, muestra los tiempos más altos en comparación con las otras alternativas.

REST vs GraphQL

Ilustración 49

Promedio de tiempos REST vs GraphQL

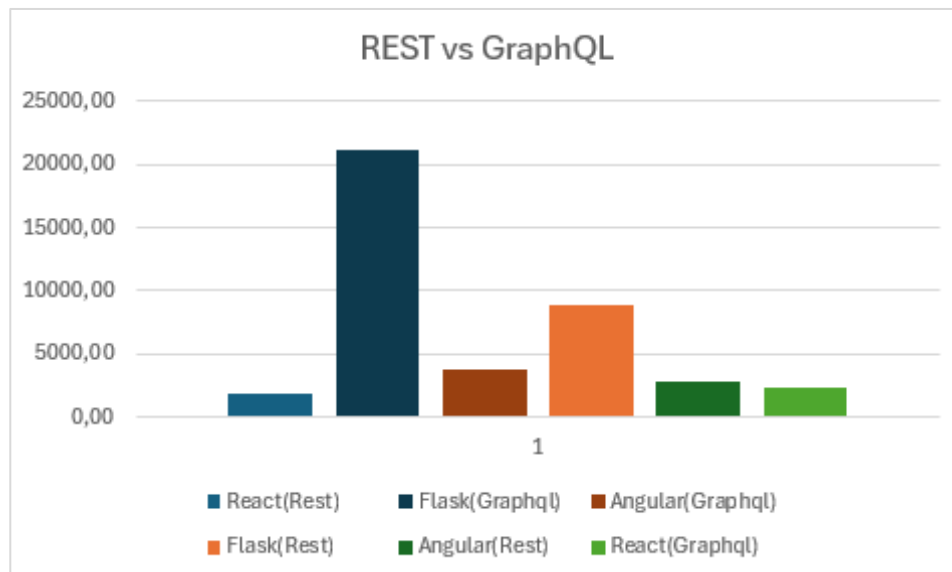


Tabla 9

Promedio de tiempos REST vs GraphQL

	REST	GraphQL	Porcentaje
React	1814,33	2265,38	19,91%
Flask	8946,42	21119,09	57,64%
Angular	2851,69	3722,57	23,39%

En la **Tabla 9**

Promedio de tiempos REST vs GraphQL se visualiza que React es la tecnología con los

tiempos de respuesta más bajos en ambas estructuras, REST y GraphQL, siendo REST el que ofrece el mejor desempeño para esta tecnología. Angular ocupa la segunda posición, con valores algo más altos. Por otro lado, Flask registra los tiempos más elevados en ambos casos, lo que refleja un rendimiento notablemente inferior en comparación con las otras tecnologías.

3.2 Discusión

3.2.1 Problemas técnicos

Durante la ejecución del experimento computacional, fue necesario considerar ciertos aspectos técnicos que influían directamente en los resultados. Uno de los más relevantes fue la variación en los tiempos de consulta cuando la memoria caché se encontraba activada, especialmente notoria en Angular con GraphQL. Esta diferencia podía generar una ventaja en el rendimiento, por lo que se optó por desactivar el caché en todas las tecnologías evaluadas, garantizando así condiciones equitativas para la comparación.

Asimismo, tras realizar múltiples ejecuciones del experimento, se identificó que el modo en que se procesaban los datos también tenía un impacto considerable en los tiempos de respuesta. Inicialmente, se empleó una estructura de bucles for para gestionar las consultas de acuerdo con los diferentes casos de uso definidos. No obstante, se buscó la alternativa de utilizar métodos asincrónicos como `async` y `await`, lo que permitió evidenciar una mejora significativa en el procesamiento de datos en comparación con el enfoque secuencial tradicional. Debido a esto, se decidió adoptar los métodos asincrónicos en todas las tecnologías, con la excepción de Flask, donde dicha implementación no era aplicable de la misma manera.

Otro inconveniente detectado durante la ejecución del experimento computacional estuvo relacionado con el lenguaje de programación JavaScript. Dado que el objetivo del

experimento era obtener mediciones de tiempo lo más precisas posible, era fundamental considerar únicamente el tiempo real que tomaba realizar las consultas, sin incluir procesos adicionales.

Sin embargo, JavaScript tiene una particularidad, si los datos obtenidos no son utilizados explícitamente, por ejemplo, si no se imprimen o no se muestran en el frontend, el motor del lenguaje puede omitir dichas operaciones, optimizando automáticamente el código. Esta optimización afecta a las mediciones, ya que ignora el tiempo que tomaría procesar o recuperar esos datos, generando resultados inexactos.

Para solucionar este inconveniente, fue necesario utilizar los datos consultados de una forma que garantizara su procesamiento efectivo, pero sin impactar el tiempo total de la medición. Esto permitió obtener resultados representativos del rendimiento real de las tecnologías evaluadas, evitando que el lenguaje excluyera operaciones relevantes del cálculo del tiempo de respuesta.

3.2.2 Limitaciones del entorno

El experimento computacional requiere una cantidad considerable de recursos, dado que realiza un alto consumo de datos según el caso de uso ejecutado. Uno de los inconvenientes más frecuentes fue la limitación en la cantidad de solicitudes HTTP que pueden manejar algunos navegadores web, particularmente Google Chrome y Microsoft Edge. En estos navegadores, el experimento no podía ejecutarse con normalidad, ya que en los casos Cu4 y Cu5 tendía a colapsar, arrojando mensajes de error relacionados con la restricción de recursos.

Para evitar este problema, se optó por utilizar el navegador Firefox, el cual permite gestionar una mayor cantidad de llamadas HTTP sin bloquearlas o rechazarlas, lo que permitió llevar a cabo el experimento de forma más estable.

Adicionalmente, se identificó que a partir del caso Cu3, independientemente de la tecnología utilizada, era necesario refrescar la página del navegador antes de ejecutar los casos siguientes. Esto se debía a que las consultas previas quedaban almacenadas en la memoria del navegador, lo que influía negativamente en los tiempos de respuesta para Cu4 y Cu5. Al recargar la página, se logró obtener mediciones más precisas y representativas.

Es fundamental que, al momento de ejecutar el experimento, no existan procesos innecesarios activos en el sistema operativo, ya que estos pueden afectar en el rendimiento general. El experimento demanda un uso intensivo de la memoria RAM, por lo que liberar recursos previamente es ideal para mantener la estabilidad y consistencia de los resultados obtenidos.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

Mediante una revisión de fuentes bibliográficas, se construyó un marco teórico sólido que permitió comprender los elementos involucrados en esta investigación. Esta base conceptual brindó el sustento necesario para entender las arquitecturas REST y GraphQL, sus características, ventajas y limitaciones, así como las tecnologías frontend seleccionadas para el estudio: React, Angular y Flask. La teoría consultada permitió fundamentar el diseño del experimento computacional, establecer los parámetros de evaluación y aplicar correctamente la

métrica de tiempo de respuesta definida por la norma ISO/IEC 25023. De esta manera, la revisión teórica aportó claridad y respaldo metodológico a todas las etapas del trabajo.

Con base en los conocimientos adquiridos, se diseñó e implementó un experimento computacional orientado a comparar la eficiencia de diferentes tecnologías frontend en el consumo de datos desde APIs desarrolladas bajo arquitecturas REST y GraphQL. Para ello, se configuraron seis combinaciones tecnológicas (React con REST, React con GraphQL, Angular con REST, Angular con GraphQL, Flask con REST y Flask con GraphQL) y se definieron cinco casos de uso con distintos niveles de complejidad, los cuales permitieron simular escenarios reales de consulta de datos. El experimento fue configurado para garantizar la neutralidad en las condiciones de prueba, desactivando elementos como el caché y controlando el entorno de ejecución, lo cual aseguró la validez y confiabilidad de los resultados obtenidos.

En respuesta a la pregunta de investigación planteada *¿Qué combinación de arquitectura (GraphQL o REST) y tecnología frontend (React, Angular o Flask) ofrece un mejor desempeño en términos de eficiencia en el consumo de datos?*, los resultados obtenidos a partir del experimento computacional indican que la combinación de React con la arquitectura REST fue la más eficiente en cuanto a tiempos de respuesta con un 19,91% que React con GraphQL. Angular con REST y Angular con GraphQL ocuparon el tercer y cuarto lugar, respectivamente. Las combinaciones que presentaron el menor rendimiento fueron Flask con REST y Flask con GraphQL. Estos resultados evidencian que, aunque GraphQL puede presentar ventajas en ciertos escenarios específicos, REST mantiene un desempeño superior en la mayoría de los casos evaluados, particularmente cuando se utiliza con tecnologías frontend modernas como React. Asimismo, se comprueba que la elección tanto de la arquitectura como de la tecnología frontend influye directamente en la eficiencia del sistema, por lo que es fundamental considerar estas variables al diseñar soluciones donde el tiempo de respuesta sea un criterio crítico.

Recomendaciones

El experimento se realizó en un entorno local controlado. Para obtener resultados en entornos de producción, se recomienda desplegar las aplicaciones en servidores remotos o servicios en la nube, donde influyen factores como la latencia de red, la carga distribuida y la disponibilidad de recursos.

Se sugiere replicar el experimento utilizando tecnologías móviles como React Native o Flutter, con el fin de evaluar cómo varía la eficiencia de REST y GraphQL en entornos móviles. Esto permitiría obtener una visión más completa del rendimiento de ambas arquitecturas en diferentes plataformas y contextos de uso.

Bibliografía

Archip, A., Herghelegiu, Amarandei, Mironeanu, C., & Șerban, E. (1 de Octubre de 2018). RESTful Web Services – A Question of Standards. *International Conference on System Theory, Control and Computing*. Obtenido de <https://www.semanticscholar.org/paper/RESTful-Web-Services—A-Question-of-Standards-Archip-Amarandei/5909bd3bf97e253ed6fa3b8fe9bc2e6cea635729>

Cerro, P. V. (2020). Frameworks y lenguajes de desarrollo para Frontend. *Dialnet*. Obtenido de <https://dialnet.unirioja.es/servlet/articulo?codigo=7220422>

Cocca, G. (27 de Abril de 2023). *The REST API Handbook – How to Build, Test, Consume, and Document REST APIs*. Obtenido de freeCodeCamp: <https://www.freecodecamp.org/news/build-consume-and-document-a-rest-api/>

Colectiva, N. (27 de Julio de 2023). *Que es un framework, historia y más detalles*. Obtenido de Blog de Programación y Desarrollo - Nube Colectiva.: <https://blog.nubecolectiva.com/que-es-un-framework-historia-y-mas-detalles/>

Durairaju, P. (7 de Octubre de 2023). *GraphQL backends becoming more efficient and performant: Takeaways from GraphQL Conf 2023*. Obtenido de Hasura GraphQL Engine Blog.: <https://hasura.io/blog/graphql-conf-2023-recap-and-the-future-of-graphql/>

Eseme, S. (29 de mayo de 2023). *GraphQL vs REST: todo lo que necesitas saber*. Obtenido de Kinsta®: <https://kinsta.com/es/blog/graphql-vs-rest/#qu-es-una-api-restful>

Galipenso. (Diciembre de 2014). *Retrieved from Experto JAVA*. Obtenido de Servicios Rest.: <http://expertojava.ua.es/experto/restringido/2014-15/rest/rest.html>

GraphQL. (2023). *Introduction to GraphQL*. Obtenido de <https://graphql.org/learn/>

Guillen-Drija, C., Quintero, R., & Kleiman, A. (2018). *GraphQL vs REST: una comparación desde la perspectiva de eficiencia de desempeño*. ResearchGate. Obtenido de <https://doi.org/10.13140/RG.2.2.25221.19680>

iso.org. (2022). ISO/IEC 25023. Obtenido de <https://www.iso.org/standard/35747.html>

Kanthed, S. (2023). Rest vs. GraphQL: Comparative Analysis of API Design Approaches. *International Journal of Multidisciplinary Research and Growth Evaluation*, 8.

Lawson, E. (8 de Diciembre de 2023). *GraphQL APIs and the future of Front-End development*. Obtenido de DEV Community.: <https://dev.to/hackmamba/graphql-apis-and-the-future-of-front-end-development-4jbm>

Naciones Unidas. (2015). *Objetivos de Desarrollo Sostenible*. Obtenido de <https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/>

Neuse, J. (1 de Marzo de 2023). *Is GraphQL dying? 2023 Trend Analysis of REST, GraphQL, OpenAPI, SOAP, gRPC and tRPC*. Obtenido de WunderGraph: https://wundergraph.com/blog/graphql_rest_openapi_trend_analysis_2023

Quiña-Mera, A., Fernandez, P., García, J., & Ruiz-Cortés, A. (09 de Septiembre de 2022). GraphQL: A Systematic Mapping Study. *ACM Computing Surveys*. Obtenido de https://www.researchgate.net/publication/363419960_GraphQL_A_Systematic_Mapping_Study

Secretaría Nacional de Planificación. (2021). *Plan Nacional de Desarrollo "Plan de Creación de Oportunidades 2021-2025"*. Obtenido de <https://siteal.iiep.unesco.org/bdnp/3676/plan-nacional-desarrollo-plan-creacion-oportunidades-2021-2025>

Simha, D. (13 de Septiembre de 2023). *State of GraphQL Gateways in 2023*. Obtenido de The Guild: <https://the-guild.dev/blog/state-of-graphql-gateways-in-2023>

Stackoverflow. (2023). *Stackoverflow*. Obtenido de <https://survey.stackoverflow.co/2023/#section-admired-and-desired-web-frameworks-and-technologies>

Stackoverflow. (2024). Obtenido de <https://survey.stackoverflow.co/2024/technology>

Vadecha, A. (25 de Octubre de 2021). *GraphQL vs REST API – Which Should You Use for Back End API Development?* Obtenido de freecodecamp: <https://www.freecodecamp.org/news/graphql-vs-rest-api/>

Valencia, N. D. (2023). Desarrollo de un envoltorio del API-REST de Napster utilizando el lenguaje de consultas GraphQL, para mejorar la eficiencia del consumo de los datos basado en la característica de eficiencia de la norma ISO/IEC 25023. *Tesis de pregrado, Universidad Técnica del Norte*. doi:<http://repositorio.utn.edu.ec/handle/123456789/14818>

Williams, A. (25 de Octubre de 2023). *REST API Standards: A Comprehensive Guide*. Obtenido de Integrate: <https://www.integrate.io/blog/rest-api-standards/#:~:text=Standardized%20REST%20APIs%20serve%20as,evolve%2C%20their%20technological%20needs%20change>.