ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Análisis y comparación de tecnologías API: REST, GraphQL y gRPC

Analysis and comparison of API technologies: REST, GraphQL, and gRPC

Realizado por
Ignacio Lopezosa Serrano

Tutorizado por
Gerardo Bandera Burgueño

Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2024

Resumen

Este estudio presenta un análisis y comparación exhaustivos de tres tecnologías de APIs: REST, GraphQL y gRPC. A través de pruebas prácticas y análisis teóricos, el proyecto evalúa las fortalezas y debilidades de cada tecnología en términos de rendimiento, escalabilidad y eficiencia de recursos. REST demuestra ser una tecnología robusta y versátil, ofreciendo un rendimiento constante en la mayoría de los escenarios, aunque presenta limitaciones al gestionar operaciones de escritura a gran escala. GraphQL destaca por su flexibilidad en la consulta de datos, pero sufre de una mayor latencia, especialmente con actualizaciones de datos complejas. gRPC ofrece una mayor eficiencia en la transferencia de datos y menor latencia, sobre todo en entornos de microservicios de alto rendimiento, aunque su implementación es más compleja. Los resultados de este estudio proporcionan información clave para elegir la tecnología API adecuada según los requisitos específicos del proyecto.

Palabras clave: API, REST, GraphQL, gRPC, rendimiento

Abstract

This study presents a thorough analysis and comparison of three API technologies: REST, GraphQL, and gRPC. Through practical tests and theoretical insights, the project evaluates the strengths and weaknesses of each technology in terms of performance, scalability, and resource efficiency. REST proves to be a robust and versatile technology, offering consistent performance in most scenarios but showing limitations when handling large-scale write operations. GraphQL shines in its flexibility for querying data but suffers from higher latency, particularly with complex data updates. gRPC demonstrates superior efficiency in data transfer and lower latency, especially in high-throughput microservice environments, but has greater implementation complexity. The results of this study provide critical insights into the best use cases for each API technology, helping developers select the appropriate tool based on project-specific requirements.

Keywords: API, REST, GraphQL, gRPC, performance

Índice

1 Introducción **1**
1.1 Contextualización del Problema **1**
1.2 Motivación y Objetivos **3**
1.3 Estructura de la memoria **4**
1.3.1 Introducción 4
1.3.2 Estado del Arte 4
1.3.3 Metodología 4
1.3.4 Desarrollo del Proyecto 5
1.3.5 Resultados y Pruebas 5
1.3.6 Conclusiones y Líneas Futuras 5
1.3.7 Referencias 6
1.3.8 Apéndices 6
2 Marco Teórico **7**
2.1 REST, GraphQL y gRPC **7**
2.1.1 REST 7
Recursos y Representaciones 8
Operaciones 9
Sin Estado 9
Resumen 10
2.1.2 GraphQL 11
Lenguaje de Consultas 11
Entorno de Ejecución 18
Resumen 19
2.1.2 gRPC 22
Protocol Buffers 22
Servidor y Cliente gRPC 26
Resumen 28
2.2 Comparación de Tecnologías **30**
2.2.1 Revisión de la Literatura 30
2.2.2 Comparación de Criterios 31
Rendimiento 31
Escalabilidad 32
Facilidad de Uso 32
Flexibilidad 33
Seguridad 33
Compatibilidad 33
Capacidad de Tipado 33
2.2.3 Rúbrica de Evaluación 35
3 Metodología **39**
3.1 Objeto de Análisis **39**

3.2 Metodología de Trabajo **41**
3.3 Fases de Trabajo **41**
3.3 Limitaciones Metodológicas y Alcance del Estudio **42**
3.3.1 Limitaciones Metodológicas 42
3.3.2 Alcance del Estudio 43

4 Desarrollo del Proyecto **45**
4.1 Introducción y Estructura General **46**
4.1.1 Contenido 46
4.1.2 Forma 47
4.2 Base de Datos **50**
4.2.1 Tablas y Modelo Entidad-Relación 50
4.2.2 Operaciones CRUD 51
4.2.3 Seeding 52
Inicialización 52
Manejo de Transacciones 52
Creación de Usuarios y Publicaciones 53
Inserción de los Usuarios en la Base de Datos 53
Creación de Chats y Mensajes 54
Inserción de los Chats en la Base de Datos 55
Resumen 55
4.2.4 Modelos, DTOs y Conversión 55
Modelos 55
DTOs 56
Conversión 57
4.3 APIs **58**
4.3.1 REST 58
Estructura 58
Implementación 58
Experiencia de Desarrollo 64
4.3.2 GraphQL 65
Estructura 65
Implementación 66
Experiencia de Desarrollo 68
4.3.3 gRPC 69
Estructura 69
Implementación 69
Experiencia de Desarrollo 74
4.4 Pruebas de Extremo a Extremo con Postman **74**
4.5 Script k6 para Pruebas de Carga y Rendimiento **77**
4.6 Script Python para Análisis Comparativo **79**
4.6.1 Diseño e implementación 79
4.7 Herramientas Accesorias **81**

5 Resultados y Pruebas **83**
5.1 Pruebas de Carga y Rendimiento **83**
5.1.1 REST 84

Prueba General 84
Prueba Sintética de Caché 84
5.1.2 GraphQL 85
Prueba General Datos Mínimos 85
Prueba General Datos Máximos 86
Prueba Sintética de Caché 87
5.1.3 gRPC 87
Prueba General 87
Prueba Sintética de Caché 88
5.1.2 Resumen 89
5.2 Análisis de Resultados 89
5.2.1 Resultados 89
5.2.1 Análisis de las Pruebas Sintéticas 104
5.2.2 Análisis de las Pruebas Generales 105
En Común 105
REST 106
GraphQL Datos Máximos 106
GraphQL Datos Mínimos 107
gRPC 107
5.2.3 Resumen 107
5.3 Hallazgos 108

6 Conclusiones y Líneas Futuras 109
6.1 Conclusiones 109
6.2 Líneas Futuras 111

Referencias 113

Anexo A: Código Fuente, Instalación y Uso 117
A.1 Código Fuente: 117
A.2 Requerimientos: 117
A.3 Obtener el código: 118
A.4 Ejecutar los servidores API: 118
A.5 Ejecutar las pruebas k6: 119
A.6 Ejecutar el script Python de análisis : 120

Anexo B: Evolución del Diseño del Proyecto 121

1

Introducción

1.1 Contextualización del Problema

Primero es necesario definir qué es una API (Interfaz de Programación de Aplicaciones). Como se define en el libro Designing Web APIs: Building APIs That Developers Love: “Una API es una interfaz que un programa de software presenta a otros programas, humanos y, en el caso de las APIs web, a el mundo a través de internet… Las APIs son los pilares que permiten la interoperabilidad entre las principales plataformas empresariales en la web. Las APIs son cómo se crea y se mantiene la identidad en cuentas de software en la nube, desde tu dirección de correo electrónico corporativa hasta el software de diseño colaborativo y las aplicaciones web que te ayudan a pedir pizza. Las API son cómo se comparte la información del pronóstico del tiempo desde una fuente confiable como el Servicio Meteorológico Nacional hasta cientos de aplicaciones de software que se especializan en su presentación. Las API procesan tus tarjetas de crédito y permiten a las empresas cobrar tu dinero sin preocuparse por las minucias de la tecnología financiera y sus correspondientes leyes y regulaciones. [1]”

Las APIs son cada vez más relevantes a nivel global. Según un estudio de Google el 56% de los dirigentes de IT consideran a las APIs activos que ayudan a organizaciones construir mejores experiencias digitales y productos, el 52% afirman que las APIs aceleran la innovación y el 36% ven las APIs como activos

estratégicos para crear valor de negocio [2]. Y por tanto es cada vez más importante conocer en profundidad las diferentes tecnologías y estándares para el desarrollo de APIs, cada uno con sus propias ventajas y desventajas.

Desde la aparición de las primeras APIs web modernas allá a principios de los 2000 [3], han surgido y desaparecido muchas formas de diseñarlas e implementarlas. Según una encuesta realizada por el equipo de Postman en 2023, como se puede ver en la figura 1.1, tres de les enfoques más conocidos y usados a día de hoy son REST, GraphQL y gRPC [4].

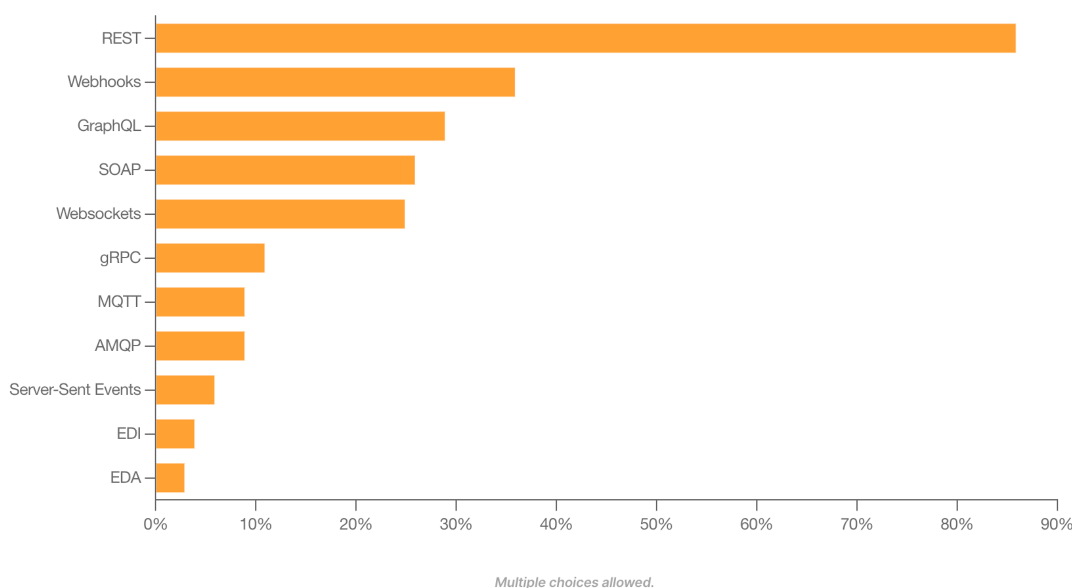


Figura 1.1 Popularidad de estilos arquitecturales de APIs según el *2023 State of the API Report* de Postman [4].

Webhooks y Websockets son estilos de arquitectura enfocados a eventos y *streaming* por lo que no se han tenido en cuenta para este estudio. Igualmente se ha obviado SOAP por su decreciente popularidad y principal relevancia en sistemas heredados, por lo que no se considera interesante para nuevos desarrollos.

En este estudio se analizará y comparará cada una de estas tecnologías para determinar su utilidad, usabilidad y rendimiento. Aunque ya existen buenos estudios que comparan todas o algunas de estas tecnologías como:

- Comparative Review of Selected Internet Communication Protocols [5].
- Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC [6].
- Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC [7].

No llegan a tener un enfoque holístico en el que se analice el rendimiento, escalabilidad y facilidad de uso y desarrollo de estas, y donde se determinen casos de uso específicos para cada una de ellas respaldados con análisis teórico y experimentación.

1.2 Motivación y Objetivos

Dada la creciente relevancia de las APIs en el desarrollo de software actual y la ausencia de un estudio que compare exhaustivamente las tecnologías API REST, GraphQL y gRPC desde una perspectiva holística que integre tanto aspectos teóricos como prácticos, esta investigación pretende llenar ese vacío. El objetivo es ofrecer un estudio de referencia que permita a los desarrolladores e investigadores comparar estas tecnologías en todos los niveles, con un enfoque riguroso en su desempeño en casos de uso reales.

Este estudio está concebido para convertirse en una guía clave para desarrolladores que buscan seleccionar la tecnología más adecuada para sus proyectos, evitando ineficiencias y errores de diseño basados en suposiciones teóricas o en la falta de conocimiento contextual. Además, se espera que esta investigación sirva como base sólida para futuros estudios que deseen profundizar en la comparación, considerando nuevos factores no abordados en este trabajo, o ampliar su alcance incluyendo otras tecnologías emergentes. Para alcanzar estos objetivos, se llevará a cabo un análisis teórico y práctico exhaustivo, con el fin de obtener una visión integral del tema y minimizar al máximo posibles sesgos o puntos ciegos.

En el análisis teórico, se estudiarán y compararán las tecnologías mencionadas y los fundamentos sobre los que se construyen, con el fin de obtener un entendimiento profundo de sus diferencias y de cómo se pueden evaluar en la práctica. Y en el análisis práctico, se diseñará, implementará y evaluará una aplicación que emule escenarios reales, en los cuales cada tecnología presente ventajas y desventajas destacables. La aplicación está diseñada para simular casos de uso que reflejen de manera precisa las posibles implicaciones prácticas de cada tecnología en un entorno comercial. Esta aplicación será sometida a una batería de pruebas, tanto manuales como automatizadas, a través de las cuales se recopilarán múltiples métricas en una variedad de escenarios prácticos. Los resultados serán analizados en profundidad utilizando scripts para el análisis de datos.

Adicionalmente, se incluirá un apartado que abordará la experiencia de desarrollo desde una perspectiva más subjetiva. Aunque este aspecto es

susceptible a variaciones significativas debido a factores externos, se ofrecerán métricas como el tiempo de desarrollo y la cantidad de errores detectados, que pueden servir como un punto de partida. Este apartado también abrirá la posibilidad para futuras investigaciones que exploren en detalle la experiencia de desarrollo con estas tecnologías en diferentes entornos.

1.3 Estructura de la memoria

La presente memoria está organizada de manera que progresa desde aspectos abstractos y generales hacia temas concretos y específicos. Se compone de los siguientes capítulos y secciones: *Introducción*, *Estado del Arte*, *Metodología*, *Desarrollo del Proyecto*, *Resultados y Pruebas*, *Conclusiones y Líneas Futuras*, *Referencias* y *Apéndices* (que incluyen el *Manual de Instalación y Uso*, el *Código Fuente*, y la *Evolución del Diseño del Proyecto*). A continuación, se ofrece una descripción detallada de cada una de estas partes.

1.3.1 Introducción

Este primer capítulo introduce el tema de estudio, presenta la motivación detrás del trabajo y define los objetivos y el alcance de la investigación. Además, se proporciona una visión general de la estructura de la memoria para orientar al lector sobre el contenido de los capítulos siguientes.

1.3.2 Estado del Arte

En este capítulo se presentan los antecedentes teóricos y conceptuales necesarios para comprender las tecnologías comparadas: REST, GraphQL y gRPC. Se divide en dos apartados principales. En el primero, se realiza un análisis exhaustivo de cada tecnología por separado. En el segundo, se comparan sus diferencias y similitudes desde una perspectiva teórica, estableciendo así las bases para el análisis comparativo que se desarrollará en los capítulos posteriores.

1.3.3 Metodología

Aquí se detalla cómo se diseñó y llevó a cabo el estudio. La sección se organiza en cuatro apartados:

- **Objeto de Análisis**: En este apartado se describen las entidades sobre las que se va a realizar el análisis comparativo, incluyendo el diseño general de la API de la aplicación, las técnicas y herramientas utilizadas para el desarrollo, ejecución y análisis de los resultados.

- **Metodología de Trabajo**: Este apartado explica la organización y metodología de desarrollo aplicada durante la implementación de la aplicación, así como las limitaciones metodológicas y el alcance del estudio.
- **Fases de Trabajo**: Aquí se detalla cómo se distribuyó la carga de trabajo en las distintas fases del proyecto.
- **Limitaciones Metodológicas y Alcance del Estudio**: Se recoge de forma clara y concisa todas las limitaciones que presenta el enfoque metodológico elegido y se especifica el alcance exacto del estudio.

1.3.4 Desarrollo del Proyecto

En este capítulo se expone en detalle el análisis, diseño e implementación de la aplicación API, así como las pruebas de integración, carga, rendimiento y los scripts de análisis de datos. El capítulo comienza con una Introducción y Estructura General del Proyecto, que justifica el diseño global de la aplicación y el caso de uso real que replica. A continuación, se detalla el Diseño de la Base de Datos, incluyendo el modelo de entidad-relación, seguido por el Diseño de la API, donde se describen los endpoints y sus funcionalidades, así como las interfaces de entrada y salida. Posteriormente, se presentan los resultados de las Pruebas de Integración realizadas con Postman [8], y se concluye con el diseño y desarrollo de los Scripts para Pruebas de Carga y Rendimiento con k6 [9] y de análisis de datos con Python.

1.3.5 Resultados y Pruebas

Este capítulo presenta, contextualiza y analiza los resultados obtenidos en las pruebas realizadas. Se divide en dos apartados:

- **Pruebas de Carga y Rendimiento**: En este apartado se muestran todas las métricas y datos obtenidos de las pruebas, acompañados de visualizaciones que facilitan su interpretación.
- **Análisis de Resultados**: Aquí se analizan los datos de forma detallada, tanto manualmente como a través del script de análisis de datos en Python, generando gráficos y visualizaciones que ayudan a interpretar correctamente los resultados.

1.3.6 Conclusiones y Líneas Futuras

En este último capítulo se sintetizan e interpretan los resultados del estudio, destacando los hallazgos clave y discutiendo sus implicaciones. Además, se sugieren posibles líneas de investigación futura que podrían expandir o complementar el trabajo realizado.

1.3.7 Referencias

En esta sección se enumeran todas las fuentes bibliográficas y recursos externos utilizados durante la investigación.

1.3.8 Apéndices

Esta sección incluye varios apéndices que proporcionan información complementaria:

- **Manual de Instalación y Uso**: Explica detalladamente cómo instalar y utilizar los programas desarrollados para este estudio en cualquier dispositivo.
- **Código Fuente**: Contiene todo el código desarrollado durante el proyecto, organizado de manera accesible.
- **Evolución del Diseño del Proyecto**: Este apéndice destaca y explica las diferencias entre el diseño inicial presentado en el anteproyecto y el diseño finalmente implementado en este estudio.

2

Marco Teórico

El capítulo aborda los fundamentos teóricos y conceptuales necesarios para entender las tecnologías REST, GraphQL y gRPC. Está estructurado en dos partes principales: primero, se hace un análisis detallado de cada tecnología individualmente; luego, se contrastan sus diferencias y similitudes desde una perspectiva teórica, sentando las bases para el análisis comparativo en capítulos posteriores.

2.1 REST, GraphQL y gRPC

Para realizar una comparación exhaustiva de las tecnologías API objeto de este estudio, es fundamental analizar en profundidad cada una de ellas por separado. Esto proporcionará un contexto teórico sólido que sustentará el análisis comparativo y permitirá derivar conclusiones empíricas basadas en aplicaciones prácticas.

2.1.1 REST

REST (Representational State Transfer) fue introducido en el año 2000 por Roy Thomas Fielding en el capítulo 5 de su tesis doctoral titulada "Architectural Styles and the Design of Network-based Software Architectures."

En su trabajo, Fielding define REST como un estilo de arquitectura para sistemas de hipermedia distribuidos [10].

Como estilo de arquitectura, REST establece principios sobre cómo organizar, transmitir y manipular información a través de una interfaz uniforme y accesible mediante internet. Sus características esenciales se pueden resumir en tres conceptos fundamentales:

Recursos y Representaciones

REST organiza la información en *recursos*. Un *recurso* es cualquier concepto que puede ser identificado y representado, como un documento, una imagen o una colección de otros *recursos*.

> *"Un recurso es un mapeo conceptual hacia un conjunto de entidades, no la entidad a la que corresponde en algún determinado momento [11]." ~ Roy T. Fielding*

Los *recursos* son identificados mediante un identificador de *recurso* (URI), que usualmente corresponde a la URL a la que se realiza la petición. Por ejemplo, en el caso de un documento específico dentro de una colección, el identificador podría ser *http://www.example.com/documents/1*. Aquí, el identificador *"/documents/1"* se añade al final del dominio del servicio web para interactuar con este *recurso*, a esto se lo conoce como el *path*.

Por convención, los identificadores suelen ser sustantivos plurales para colecciones, acompañados de un identificador único para acceder elementos específicos. Las operaciones sobre estos *recursos* no se especifican mediante verbos en el URI, sino que se definen mediante los métodos del protocolo HTTP (Hypertext Transfer Protocol), los cuales se detallan en la siguiente sección.

La *representación* de un recurso es la forma en la que se transmite el estado de ese recurso a través de la red. Una *representación* es una secuencia de bytes que captura el estado actual o deseado de un recurso [12]. También se le suele llamar documento, archivo o *payload*.

Una *representación* está compuesta por datos y metadatos. Los datos indican el estado del *recurso*, mientras que los metadatos, que consisten en pares nombre-valor, proporcionan información adicional sobre los datos o sobre otros metadatos [13]. En el contexto del protocolo HTTP, los datos se encuentran en el

cuerpo del mensaje y los metadatos se ubican en las cabeceras, aunque también pueden formar parte del cuerpo del mensaje.

El formato en el que se presentan los datos se denomina *media type*. Ejemplos comunes incluyen JSON (JavaScript Object Notation), XML (Extensible Markup Language) o HTML (Hypertext Markup Language). Este se especifica como un metadato en la cabecera *Content-Type*.

Operaciones

Además de los *recursos*, identificadores y *representaciones*, REST utiliza *datos de control* para definir el propósito de un mensaje entre componentes, como la acción solicitada o el significado de la respuesta [13].

Los *datos de control* se utilizan principalmente para identificar el tipo de operación que se quiere realizar sobre un *recurso* usando los métodos del protocolo HTTP:

- **GET**: Recupera la *representación* del estado actual del *recurso*.
- **POST**: Crea un nuevo *recurso* usando la *representación* proporcionada en el cuerpo del mensaje.
- **PUT**: Actualiza el estado de un *recurso* con el estado definido por la *representación* proporcionada en el cuerpo.
- **PATCH**: Actualiza parcialmente el estado de un *recurso* con la parte del estado definida en la *representación* del cuerpo del mensaje.
- **DELETE**: Elimina el *recurso* o el estado del *recurso* deseado.
- **OPTIONS**: Obtiene la lista de métodos que el *recurso* soporta.

En casos excepcionales, donde una operación no encaja en ninguno de los métodos HTTP mencionados, se puede utilizar el método POST sobre el recurso con un identificador adicional en la URL para especificar la operación deseada. Por ejemplo, *http://www.example.com/documents/1/send* se podría usar para enviar un documento.

Sin Estado

Una de las características esenciales de REST es que todas las operaciones han de ser *stateless* (sin estado). Lo que significa que cada petición de cliente al servidor debe de contener toda la información necesaria para comprender la petición y no puede hacer uso de ningún contexto almacenado en el servidor. El

estado de la sesión se mantiene enteramente en el cliente [14]. Esta es la única restricción impuesta a nivel de protocolo en el estilo de arquitectura REST.

Resumen

Como se ha analizado, REST no especifica de manera rígida qué protocolos deben de utilizarse para la comunicación (más allá de que la comunicación sea sin estado) ni qué formato de datos debe emplearse para la *representación* de los *recursos*.

Como ya aclaró Roy T. Fielding en su tesis doctoral:

> *"REST no restringe la comunicación a un protocolo en particular, pero sí constriñe la interfaz entre componentes y, por consiguiente, el alcance de las presuposiciones sobre la interacción e implementación que de otra manera podrían hacerse. [15]"*

Dado que no es posible realizar un análisis empírico sobre el estilo arquitectónico REST en sí mismo, se ha optado por analizar REST implementado con las tecnologías más populares, específicamente **HTTP/1.1** y **JSON**.

Al utilizar el protocolo HTTP/1.1 directamente, sin ninguna abstracción adicional, REST debería ser capaz de transmitir información de manera sencilla y eficiente. Sin embargo, debido a la falta de abstracciones que faciliten la realización de consultas complejas, es posible que se requieran múltiples peticiones para realizar operaciones más complejas (*underfetching*) o se obtengan más datos de lo estrictamente necesario (*overfetching*).

En cuanto a JSON, su uso mejora significativamente la usabilidad y la facilidad de desarrollo, ya que es un formato legible por humanos. No obstante, al ser un formato de datos basado en texto, puede presentar inconvenientes al transportar grandes volúmenes de información, debido a que no se serializa en un formato binario más eficiente, lo que aumenta considerablemente el tamaño de los mensajes.

En resumen, en teoría, REST es ideal para realizar operaciones sencillas que implican cantidades reducidas de información. Sin embargo, puede mostrar limitaciones cuando se requiere manejar operaciones complejas o muy específicas o grandes volúmenes de datos.

2.1.2 GraphQL

GraphQL fue desarrollado por Meta (entonces Facebook) para crear una API de recuperación de datos suficientemente potente como parar describir todo Facebook, pero al mismo tiempo simple y sencilla de aprender para que los desarrolladores de producto puedan centrarse en construir cosas rápido. En 2015 lo hicieron de código abierto redactando una especificación y publicando una implementación de referencia [16].

GraphQL se define en su documentación oficial como un lenguaje de consultas para APIs y un entorno de ejecución para procesar consultas usando un sistema de tipos definido por los datos [17]. Véase el lenguaje y el entorno de ejecución por separado.

Lenguaje de Consultas

Esto incluye la sintaxis de las consultas y de las respuestas, cómo están estructuradas y cómo se relacionan entre sí. Es lo que se podría llamar la interfaz de GraphQL. Es la forma en la que se interactúa con la API.

Este lenguaje se compone de tres elementos principales estrechamente relacionados cada uno con sus propios subelementos. Estos son el **esquema**, la **petición** y la **respuesta**. El esquema es la definición de los tipos de datos con los que se opera en la API y de las operaciones que se pueden realizar sobre ellos. La petición es la forma en la que el usuario de la API realiza algún tipo de operación u operaciones en específico. La respuesta es el resultado que manda la API de vuelta al usuario con el resultado de la operación u operaciones que pidió realizar.

Esquema

En el esquema se recoge una definición exhaustiva de la forma que tienen los datos con los que se trata en esa API y. Las formas de estos datos se llaman **objetos**. Un objeto no es más que una entidad identificada por un nombre concreto compuesta de campos, cada uno de ellos con su propio nombre y tipo. Los objetos a su vez representan un nuevo tipo de dato compuesto, i.e., otros campos pueden ser del tipo del objeto.

Existen algunos campos que hacen referencia a un dato especifico y no a un tipo complejo como los que se definen en el esquema, estos son los escalares como las cadenas de caracteres (*String*), los números enteros (*Int*) y decimales

(*Float*), valores booleanos (*Boolean*) y los identificadores (*ID*). A su vez se pueden definir escalares personalizados para otros tipos de datos simples.

Cada uno de estos campos puede ser también una lista de datos del tipo especificado y también se puede indicar si son campos requeridos o no, es decir, si el servidor espera que este campo tenga un valor especificado y si se espera lo mismo en la respuesta.

A continuación, en la figura 2.1, podemos ver un ejemplo de un esquema sencillo en el que se define un objeto *Character* que representa un personaje de la saga de películas Star Wars. Este tiene los campos *name* y *appearsIn* que indican el nombre del personaje y en qué episodios aparece. Ambos campos son requeridos y cada elemento de la lista de episodios a su vez es obligatorio, es decir, ningún elemento de la lista puede ser nulo. Esto se indica mediante el uso de las exclamaciones. Si el tipo de un campo termina en exclamación este campo es requerido. También se reconoce que *appearsIn* es una lista de episodios porque su tipo se encuentra encerrado entre corchetes. Tanto las exclamaciones como los corchetes se les llama **modificadores de tipo**, ya que añaden información extra al tipo base que modifican.

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

Figura 2.1 Ejemplo de esquema GraphQL sencillo tomado de la documentación oficial de GraphQL [17].

Todo campo de un objeto en GraphQL puede tener **argumentos**. Un argumento es un dato específico que se le pasa al campo del objeto para que modifique su comportamiento. Por ejemplo, en el campo *length* del objeto *Starship* que se ve a continuación en la figura 2.2 se le pasa un argumento *unit* de tipo *LengthUnit* con valor por defecto *METER*. El campo *length* en sí resulta en un número decimal. Al pasarle el argumento *unit* el campo devolverá un valor diferente según la unidad de medición escogida.

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

Figura 2.2 Ejemplo de esquema GraphQL con un campo con argumentos tomado de la documentación oficial de GraphQL [17].

En este ejemplo se ve un nuevo tipo de dato, los tipos **enumerados** o **enumeraciones**. También llamados *Enums*, las enumeraciones son un tipo especial de escalar que solo acepta un conjunto de valores en particular.

Volviendo al ejemplo de Star Wars, el tipo *Episode* que representa los episodios de la saga podría ser definido con una enumeración de la siguiente manera. Véase figura 2.3.

```
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
```

Figura 2.3 Ejemplo de un tipo enumerado tomado de la documentación oficial de GraphQL [17].

En este *enum* se definen los posibles valores que *Episode* puede tomar. Estos exclusivamente incluyen identificadores para las tres películas de la trilogía original. Por lo que si se quisiera poner alguna de las películas de la precuela se tendría que primero modificar el tipo enumerado. De otra forma no permite introducir ningún valor que no esté especificado en la enumeración.

Y en los casos en los que el tipo de argumento de un campo es demasiado complejo como para ser representado por ningún escalar. En estos casos se puede definir un tipo especial de objeto que puede ser pasado como argumento. Este tipo se la llama **tipo de entrada** o *input type*. Estos objetos se definen exactamente igual que los objetos normales en GraphQL, pero usando la palabra clave *input* en vez de *type*.

A continuación, figura 2.4, se muestra un ejemplo de tipo *input* para crear una crítica de una película. Este tipo se llama *ReviewInput* y está compuesto de las estrellas otorgadas (*stars*) y un comentario sobre la película (*commentary*).

```
input ReviewInput {
  stars: Int!
  commentary: String
}
```

Figura 2.4 Ejemplo de un tipo de entrada tomado de la documentación oficial de GraphQL [17].

Este tipo de entrada puede luego ser usado como argumento de un campo de un objeto o de una **mutación** o **consulta**.

Las mutaciones y consultas son tipos especiales de objeto. Estos son idénticos a cualquier otro objeto del esquema excepto porque definen el punto de entrada de toda petición GraphQL. Es decir, con estos se define la interfaz que el usuario de la API tiene a su disposición para realizar peticiones y operar con esta API.

Las consultas se usan para consultar información de la API, es decir, para consumir la API. Cada consulta posible se define como un campo dentro del tipo *Query*. La sintaxis viene a reflejar lo que sería una lista de funciones disponibles para realizar las consultas, con los datos de entrada necesarios y la salida de cada consulta. En la figura 2.5 se ve un ejemplo de dos consultas para obtener el héroe de un episodio concreto de Star Wars y un droide específico según su identificador.

```
type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

Figura 2.5 Ejemplo de consultas tomado de la documentación oficial de GraphQL [17].

Las mutaciones se usan para modificar los recursos que ofrece la API, es decir, para producir a través de la API. Al igual que con las consultas, todas las mutaciones posibles se definen dentro del tipo *Mutation*. A continuación, figura 2.6, se muestra cómo se definiría una mutación para crear una crítica de un episodio usando el tipo *ReviewInput* que se vio con anterioridad.

```
type Mutation {
  createReviewForEpisode(ep: Episode!, review: ReviewInput!): Review
}
```

Figura 2.6 Ejemplo de mutación para crear crítica de un episodio.

A su vez, GraphQL hereda de la Programación Orientada a Objetos ciertos elementos como las **interfaces**. Estas sirven como un tipo abstracto que define una serie de campos que otros tipos han de incluir en su definición para poder implementar esta interfaz. Esto permite polimorfismo, ya que cualquier objeto que implemente una interfaz puede ser usado en aquellos campos y argumentos definidos con el tipo de esa interfaz. Las interfaces se definen igual que los tipos de objetos normales, pero usando la palabra clave *interface* en vez de *type*. Véase figura 2.7.

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}
```

Figura 2.7 Ejemplo de interfaz tomado de la documentación oficial de GraphQL [17].

Para que un objeto implemente una interfaz en específico se usa la palabra clave *implements* seguida del nombre de la interfaz a implementar al lado del nombre del objeto. Se puede ver un ejemplo en la figura 2.8.

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

Figura 2.8 Ejemplo de implementación de interfaces tomado de la documentación oficial de GraphQL [17].

Y, por último, están los tipos **unión** que permiten definir tipos de varios tipos, es decir, un tipo que define un conjunto de posibles tipos. Véase la figura 2.9.

```
union SearchResult = Human | Droid | Starship
```

Figura 2.9 Ejemplo de tipo unión tomado de la documentación oficial de GraphQL [17].

Hay muchos más elementos que conforman la especificación técnica de GraphQL, como los alias, fragmentos, directivas, subscripciones, introspección, etc. Pero con estos elementos se tiene una muy buena visión general de lo que es GraphQL y cómo se define un esquema completo. Lo siguiente es ver cómo se pueden realizar peticiones a una API GraphQL para realizar consultas y mutaciones a la API.

Petición

La estructura general de una petición se compone de la operación a realizar, comúnmente identificada por un nombre, junto a los campos de la entidad sobre la que se realiza la operación que se quieren obtener de vuelta. A continuación, se muestra un ejemplo de una petición sencilla en la figura 2.10.

```
1  query Login {
2    viewer {
3      login
4      email
5    }
6  }
7
8  mutation AddComment {
9    addComment(input: {body: "Hi!", subjectId: "I_kwDOI14BHM6U7wxM"}) {
10     commentEdge {
11       node {
12         body
13       }
14     }
15   }
16 }
```

Figura 2.10 Ejemplo de una petición GraphQL sobre la API GraphQL de GitHub.

En este ejemplo se ven dos operaciones distinguidas *Login* y *AddComment*. La primera es una consulta que obtiene el nombre de usuario (*login*) y el email del usuario que está realizando la petición (*viewer*). ¡La segunda es una mutación que añade un comentario a un issue en GitHub identificado por un subjectId con cuerpo de mensaje "Hi!" y se obtiene del comentario resultante su cuerpo.

En cuanto a cómo se suele enviar esta petición a través de la red, aunque GraphQL es agnóstico de protocolos de red, se suele utilizar sobre HTTP. Esto lo aclaran en la propia documentación oficial de GraphQL dónde especifican la forma más común y recomendable de integrar GraphQL con HTTP [18].

Todas las peticiones deben dirigirse a un único *endpoint*, este por convención suele ser */graphql*. Esto ocurre porque, al contrario que con REST, no existen recursos identificados por URIs. En GraphQL solo existe un modelo conceptual de grafos que interconectan las entidades y sus atributos entre sí.

Para las consultas se realizan peticiones HTTP GET especificando el contenido de la petición en un parámetro de búsqueda con nombre *query*. Si la consulta incluyese variables se pueden definir estas en otro parámetro de consulta

llamado *variables* con el valor de las variables en formato JSON. En caso de que haya varias operaciones con nombre, se puede seleccionar cuál ejecutar especificándolo en el parámetro de búsqueda *operationName*. Se muestra un ejemplo en la figura 2.11.

```
https://api.github.com/graphql?query=query Login{viewer{login email}}&operationName=Login
```

Figura 2.11 Ejemplo de una petición de consulta sobre la API GraphQL de GitHub.

Cuando se quiere realizar mutaciones, se debe de hacer mediante una petición HTTP POST con un cuerpo de mensaje de tipo *application/json* donde se especifique la petición, el nombre de la operación a usar y las variables. Cada uno de estos elementos se definen en un campo del objeto JSON que se pasa como cuerpo del mensaje. Véase a continuación en la figura 2.12.

```
{
  "query": "mutation AddComment{addComment(input:{body:"Hi!",subjectId:"I_kwDOI14BHM6U7wxM"}){commentEdge{node{body}}}}",
  "operationName": "AddComment",
  "variables": {}
}
```

Figura 2.12 Ejemplo de una petición de mutación sobre la API GraphQL de GitHub.

Aunque últimamente se está usando de forma más frecuente realizar peticiones POST con la petición en el cuerpo del mensaje para las consultas al igual que las mutaciones. Principalmente por la limitación de longitud de las URLs. Limitación que no es tan restringida en el cuerpo del mensaje.

Esto se puede ver en obras tan populares como en el libro *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. En el apartado *fetch Requests* del capítulo 6 se muestra un ejemplo de petición de consulta en el que usan este método [19]. En este estudio se ha optado por adoptar esta nueva forma de realizar las peticiones de consulta por su flexibilidad en el tamaño de las peticiones.

Respuesta

Las respuestas a las peticiones GraphQL no son más que un documento JSON que mantiene la misma forma de los datos solicitados en la petición. Es decir, cuando se realiza una petición de consulta sobre un objeto específico con unos campos en concreto la respuesta es un JSON con ese objeto y esos mismos campos poblados por la información que almacenan.

Las únicas diferencias notables entre la petición y la respuesta en cuanto a forma se encuentran en que la respuesta está dividida en dos secciones, *data* y *errors*, y en que la respuesta no muestra argumentos, directivas y otros añadidos

usados para modificar la consulta. En la primera sección se encuentra la respuesta exitosa a la petición con los datos deseados y en la segunda se encuentra una lista de los errores que hayan ocurrido durante el procesado de la solicitud. Si no han ocurrido errores, la sección *errors* suele omitirse. A continuación, se puede ver en la figura 2.13 la respuesta que se obtiene de realizar la consulta mostrada en la figura 2.11 y en la figura 2.14 la respuesta a la mutación de la figura 2.12.

```
{
  "data": {
    "viewer": {
      "login": "n4xo-dev",
      "email": "me@n4xo.com"
    }
  }
}
```

Figura 2.13 Ejemplo de respuesta a una petición de consulta sobre la API GraphQL de GitHub.

```
{
  "data": {
    "addComment": {
      "commentEdge": {
        "node": {
          "body": "Hi!"
        }
      }
    }
  }
}
```

Figura 2.14 Ejemplo de respuesta una petición de mutación sobre la API GraphQL de GitHub.

Entorno de Ejecución

Hasta ahora se ha visto el apartado léxico, sintáctico y semántico del lenguaje de consultas GraphQL. Pero falta la parte de la "generación del código", es decir, falta aquello que analiza, procesa y resuelve las consultas de las peticiones GraphQL. Eso es lo que se llama el entorno de ejecución.

GraphQL no provee al desarrollador con un entorno de ejecución ya desarrollado ni con herramientas, o lenguajes específicos para desarrollarlos. GraphQL tan solo especifica la estructura de este entorno de ejecución y deja al desarrollador utilizar las herramientas que prefiera para construirlo.

La estructura de este entorno de ejecución se basa en ver todos los campos de objetos como funciones que retornan un valor de un tipo específico según algunos o ningún valor de entrada. Y se aplica esta resolución de funciones sobre todos los campos, incluyendo los de las respuestas retornadas por otros campos, hasta llegar a un valor escalar el cual no puede ser evaluado.

Para que esto funcione a todos los niveles existe una abstracción llamada el **Tipo Raíz** o **Tipo Consulta**. Y de este tipo raíz brotan todos los otros objetos de la consulta como campos de este. Los cuales son evaluados como funciones. Es decir, incluso las operaciones de consulta y mutación son interpretadas como campos del objeto raíz que deben de ser evaluadas mediante su respectiva función.

Estas funciones son los **resolutores**. Según cada implementación puede cambiar un poco su estructura, pero a rasgos generales son funciones que toman el objeto padre al que pertenecen, los argumentos que se hayan proveído en la petición, contexto con información relevante como el usuario haciendo la petición o conexiones a dependencias externas como bases de datos entre otras cosas e información relevante sobre el propio campo del resolutor. Debajo se muestra en la figura 2.15 un ejemplo de implementación de un resolutor usando el lenguaje de programación JavaScript.

```
Query: {
  human(obj, args, context, info) {
    return context.db.loadHumanByID(args.id).then(
      userData => new Human(userData)
    )
  }
}
```

Figura 2.15 Ejemplo de implementación de resolutor tomado de la documentación oficial de GraphQL [17].

Resumen

Como se ha visto, GraphQL es una forma de desarrollar APIs muy compleja. Al contrario que REST, GraphQL es una forma muy estricta y definida de realizar APIs, lo cual otorga ciertas ventajas y desventajas que se comentan a continuación.

Aun así, cabe destacar que tampoco es absolutamente estricto en la forma de implementar esta especificación técnica que es GraphQL. Por eso es necesario

concretar las herramientas y convenciones que se van a usar en este estudio en torno a la implementación de la API con GraphQL.

En este caso, como se aclaró con anterioridad, se va a usar la más reciente forma de hacer peticiones de tipo consulta usando el método HTTP POST junto a la consulta en el cuerpo del mensaje. Esto se hace por el mayor espacio que otorga el cuerpo del mensaje, permitiendo consultas más complejas. Y por supuesto se va a construir toda la aplicación de la API sobre HTTP. Ya que aquí se están comparando APIs Web específicamente.

En cuanto a las ventajas y desventajas que ofrece GraphQL hay varios factores. GraphQL se destaca principalmente por la flexibilidad y la alta maniobrabilidad que ofrece al usuario de la API para realizar operaciones mucho más complejas y obteniendo exactamente los datos que necesita con una sola operación. Esto aumenta mucho la usabilidad de la API una vez el usuario ya ha aprendido a usar este lenguaje de consultas. Pero esto mismo añade dificultad para los novatos que no conocen esta forma de realizar peticiones todavía. Por lo que es muy útil para los usuarios medianamente experimentados, pero supone una barrera de entrada para los más noveles.

El que se puedan realizar operaciones muy complejas de una sola vez hace pensar que se reducirían los tiempos de respuesta debido a un uso más eficiente de la red. Al reducir el número de peticiones HTTP realizadas, se reduce a su vez el tiempo que se dedica a enviar la petición por la red y recibirlo de vuelta. Pero también es posible que, debido a la complejidad añadida por el entorno de ejecución de GraphQL, la petición acabe tardando más tiempo en ser procesada que lo que se ha ahorrado en el envío del paquete HTTP.

Esto indica que probablemente GraphQL sea muy rentable en entornos en los que la velocidad de envío de mensajes a través de la red es muy lenta comparada con la velocidad de procesamiento del servidor resolviendo las peticiones a la API GraphQL.

Esto es algo que habrá que analizar con datos reales para determinar si GraphQL realmente aumenta considerablemente el tiempo de procesado de las peticiones y si, en caso afirmativo, esta diferencia es demasiado grande como para considerarse en cualquier caso en el que la velocidad de respuesta sea crucial incluso en entornos de mala conectividad red.

Una ventaja que sí parece ser innegable para GraphQL es su posibilidad de recibir exactamente los datos que uno requiere. Esta es una de las razones

principales por las que se creó, para minimizar el tamaño de los mensajes que se envían desde el servidor al cliente en el contexto de la tecnología móvil de mediados de la primera década de los 2000. Y es cierto que GraphQL soluciona los problemas de *overfetching* y *underfetching* que otras opciones menos flexibles como REST o gRPC poseen. Pero esta misma capacidad de poder pedir cualquier cantidad de datos junto a la naturaleza relacional que tiene GraphQL ha demostrado presentar sus propios desafíos.

Como Olaf Harting y Jorge Pérez demuestran en su estudio *Semantics and Complexity of GraphQL*. Aunque GraphQL puede ser muy eficiente, muestra un significativo problema en cuanto al tamaño que pueden tomar las respuestas con respecto a sus peticiones. En este estudio se muestra como el tamaño de una respuesta de una API GraphQL pueden llegar a ser exponencialmente mayores que la petición que la originó. Por suerte también muestran un método con complejidad lineal para determinar el tamaño de las respuestas antes de ejecutar la petición. Por lo que este problema puede ser resuelto de forma relativamente satisfactoria, aunque supone un procesamiento extra necesario para garantizar una API estable y confiable [20].

Este problema también fue detectado en el artículo de conferencia *An Empirical Study of GraphQL Schemas* de la decimoséptima conferencia internacional de Computación Orientada a Servicios. Y además en este se encuentra que en muchas APIs GraphQL los sistemas de paginación que implementan no son suficientes para evitar respuestas masivas y se vio que esto podía suponer un vector de ataque que podrían usar agentes maliciosos para realizar ataque de denegación de servicio con mayor facilidad [21]. Por lo que es un punto muy importante para tener en cuenta cuando se desarrolla una API con GraphQL.

Otra ventaja de GraphQL es su sistema de tipos que permite realizar consultas y mutaciones de una forma más segura y confiable, ya que existe un contrato claro al que el cliente y el servidor se adhieren para pasarse información. Además, se han desarrollado muchas herramientas que ayudan a conocer y explorar este contrato en detallo como el IDE GraphiQL [22] y la función de introspección de GraphQL [23].

En resumen, se pueden reducir las ventajas y desventajas de GraphQL en que ofrece una clara interfaz para los clientes de la API y mucha más capacidad para realizar operaciones complejas y muy específicas, pero a costa de una mayor complejidad y potenciales problemas de computación y seguridad.

2.1.2 gRPC

gRPC significa Llamadas de Procedimiento Remoto de gRPC por sus siglas en inglés y el equipo oficial de gRPC lo define de la siguiente manera:

> *"gRPC es una infraestructura de llamada de procedimiento remoto (RPC) de código abierto que puede ser ejecutado en cualquier lugar. Permite comunicarse a aplicaciones de cliente y servidor de forma transparente y facilita la construcción de sistemas conectados [24]."*

En Google hacía más de una década desde que crearon Stubby, una infraestructura RPC de propósito general para conectar sus microservicios. Pero este no seguía ningún estándar y estaba demasiado acoplado a la infraestructura interna de Google como para considerar hacerlo público. Y aprovechando la llegada de nuevos estándares y protocolos Web como el obsoleto SPDY, el precursor a HTTP/3, QUIC, y HTTP/2 [25].

El atractivo principal de gRPC, y de muchas otras herramientas del paradigma RPC, es que el cliente pueda llamar directamente a un método de una aplicación de servidor en otra máquina como su fuera un objeto local. Esto se basa en definir servicios y sus métodos que pueden ser llamados de forma remota con sus parámetros de entrada y productos de salida. En el lado del servidor se implementa esta interfaz y ejecuta un servidor gRPC para procesar las llamadas. En el lado del cliente se usa un *stub* para proveer los métodos del servidor [26].

Aunque se ejecute código remoto en el cliente como si fuese local, no requiere que el cliente y el servidor estén escritos en el mismo lenguaje. Por lo que permite comunicar una gran cantidad de diferentes microservicios entre sí de forma eficiente y sencilla.

Si bien gRPC es "agnóstico al contenido" (*payload agnostic*) y se puede utilizar cualquier formato de datos para transmitir los mensajes a través de gRPC, lo más estandarizado y recomendado es usar los *Protocol Buffers* como mecanismo para serializar datos estructurados.

Protocol Buffers

Como se observa en su documentación oficial: *"es como JSON, excepto que es más pequeño y rápido [27]."* Los *Protocol Buffers* son un mecanismo agnóstico al lenguaje, neutral de plataforma y extensible para serializar datos

estructurados usado en gRPC como el Lenguaje de Definición de Interfaces (IDL) para definir la interfaz de los servicios [28].

Con los *Protocol Buffers* se definen los tipos de los datos de entrada y salida de los métodos, esto es lo que se les llama **mensajes**. Para definirlos se crea un archivo *.proto* en el que se escribe la definición de los mensajes usando la sintaxis específica de los *Protocol Buffers*. A continuación, se muestra en la figura 2.16 un sencillo ejemplo de un mensaje de *Protocol Buffers*.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
}
```

Figura 2.16 Ejemplo de mensaje tomado de la documentación oficial de Protocol Buffers [29].

Simplemente se especifica la versión de la sintaxis que se quiere usar, se indica el comienzo de la definición de un mensaje con la palabra clave *message* seguido de su nombre y entre llaves se especifican los campos del mensaje. Estos campos se definen indicando su tipo, su nombre y su identificador numérico.

El tipo de un campo puede ser escalar, como un número decimal o entero, o una cadena de caracteres; o pueden ser tipos compuestos como enumeraciones, mapas asociativos u otros mensajes. Las enumeraciones son igual que como se vio en GraphQL, son un conjunto fijo de opciones entre las que escoger un valor concreto, véase un ejemplo en la figura 2.17. Los mapas asociativos son una colección de parejas clave-valor de un tipo específico cada uno, en la figura 2.18 se muestra un ejemplo de mapa asociativo con *Protocol Buffers*.

```
enum Corpus {
  CORPUS_UNSPECIFIED = 0;
  CORPUS_UNIVERSAL = 1;
  CORPUS_WEB = 2;
  CORPUS_IMAGES = 3;
  CORPUS_LOCAL = 4;
  CORPUS_NEWS = 5;
  CORPUS_PRODUCTS = 6;
  CORPUS_VIDEO = 7;
}
```

Figura 2.17 Ejemplo de enumeración tomado de la documentación oficial de Protocol Buffers [29].

```
map<string, Project> projects = 3;
```

Figura 2.18 Ejemplo de mapa asociativo tomado de la documentación oficial de Protocol Buffers [29].

Además, se pueden añadir modificadores a los campos de los mensajes para añadir más funcionalidad a estos. Existen los modificadores *optional*, *repeated* y *oneof*. *Optional* permite indicar si un campo puede ser omitido y, por tanto, no ser serializado; este devolvería el valor por defecto de su tipo. *Repeated* convierte al campo en una lista de cero o más elementos del tipo especificado manteniendo el orden. *Oneof* da la posibilidad de que un campo pueda tomar una de varias posibles formas, es decir, que varios campos comparten lugar dentro del mensaje y solo uno de ellos puede ser usado, ahorrando espacio en ciertos casos específicos. Se pueden ver ejemplos del uso de *optional*, *repeated* y *oneof* en las figuras 2.19, 2.20 y 2.21 respectivamente.

```
message Message3 {
  optional Message1 bar = 1;
}
```

Figura 2.19 Ejemplo del uso de *optional* tomado de la documentación oficial de Protocol Buffers [29].

```
message SearchResponse {
  repeated Result results = 1;
}

message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

Figura 2.20 Ejemplo del uso de *repeated* tomado de la documentación oficial de Protocol Buffers [29].

```
message SampleMessage {
  oneof test_oneof {
    string name = 4;
    SubMessage sub_message = 9;
  }
}
```

Figura 2.21 Ejemplo del uso de *oneof* tomado de la documentación oficial de Protocol Buffers [29].

En todos estos ejemplos se ha visto como todo campo de un mensaje o enumeración tiene asignado su propio identificador numérico. Esto es necesario debido a que este identificador es lo que realmente se acaba pasando por la red en el paquete serializado de *Protocol Buffers*. De una forma simplificada, el contenido serializado del mensaje se compone de una serie de etiquetas que identifican el campo del mensaje seguidas de su valor codificado.

La etiqueta está compuesta del identificador numérico del campo y un tipo de transmisión que indica la longitud del valor codificado para saber dónde acaba

este y empieza la etiqueta del siguiente campo. Se muestra un ejemplo visual en la figura 2.22.

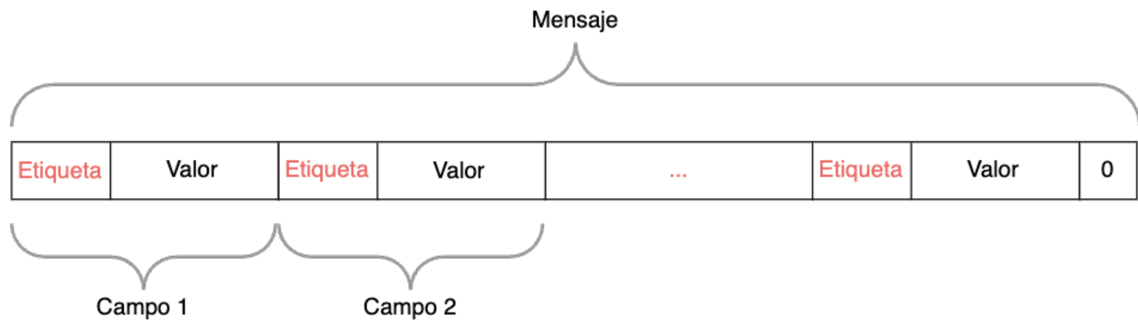


Figura 2.22 Paquete serializado de Protocol Buffers inspirado en ejemplo del libro *gRPC Up & Runnig [30]*.

De esta forma se consigue identificar el campo específico y su tipo usando la cantidad mínima de bytes necesaria. De hecho, se recomienda usar los identificadores numéricos del 1 al 15 para los campos más usados ya que solo ocupan un byte una vez codificados [29].

Una vez se han definido los mensajes se usa el compilador de *Protocol Buffers* (*protoc*) para generar código en el lenguaje de programación deseado que provea con métodos para obtener los datos de los mensajes, extraer valores individuales de los datos, comprobar la existencia de los datos, serializar los datos de vuelta a un mensaje y otras funciones útiles. En la figura 2.23 se puede ver un ejemplo de mensaje definido con *Protocol Buffers*, en la figura 2.24 cómo se usa el código generado en Java para crear un mensaje y enviarlo y en la figura 2.25 cómo se deserializa el mensaje y se obtienen los valores de los campos del mensaje.

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

Figura 2.23 Ejemplo de mensaje tomado de la documentación oficial de Protocol Buffers [29].

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

Figura 2.24 Ejemplo del uso del código generado por *protoc* en Java para enviar un mensaje tomado de la documentación oficial de Protocol Buffers [29].

```
Person john;
fstream input(argv[1], ios::in | ios::binary);
john.ParseFromIstream(&input);
int id = john.id();
std::string name = john.name();
std::string email = john.email();
```

Figura 2.25 Ejemplo del uso del código generado por *protoc* en C++ para leer un mensaje tomado de la documentación oficial de Protocol Buffers [29].

Servidor y Cliente gRPC

En los archivos *.proto* también se definen los servicios que contienen los métodos que componen la API con gRPC. Estos se definen de forma similar a los mensajes, con la palabra clave *service* seguida de su nombre con cada método definido dentro del servicio. Estos se especifican precedidos por la palabra clave *rpc* con su nombre identificador seguido del mensaje de entrada entre paréntesis y la palabra clave *returns* seguida del mensaje de salida. Véase figura 2.26.

```
rpc SayHello(HelloRequest) returns (HelloResponse);
```

Figura 2.26 Ejemplo de definición de servicio tomado de la documentación oficial de gRPC [31]

A su vez los mensajes de entrada y salida pueden ser precedidos por la palabra clave *stream* indicando que este mensaje es una transmisión de varios mensajes de forma secuencial. Si aparece en la entrada, figura 2.27, el cliente manda una transmisión de mensajes para que el servidor los lea. Si aparece en la salida, figura 2.28, el servidor manda al cliente una transmisión de mensajes. Si aparece en ambos, figura 2.29, es una transmisión bidireccional en la que tanto servidor y cliente se mandan y reciben una transmisión de mensajes independientes.

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

Figura 2.27 Ejemplo de definición de servicio con transmisión de cliente tomado de la documentación oficial de gRPC [31].

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

Figura 2.28 Ejemplo de definición de servicio con transmisión de servidor tomado de la documentación oficial de gRPC [31].

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

Figura 2.29 Ejemplo de definición de servicio con transmisión bidireccional tomado de la documentación oficial de gRPC [31].

Y con el uso de un plugin del compilador *protoc* se genera el código en el cliente para realizar la llamada remota pasándole el mensaje esperado y en el servidor para recibir los mensajes, dirigirlos a la función adecuada y aplicar la función definida por el desarrollador con el mensaje recibido y mandar el mensaje resultante de vuelta.

Una limitación relevante de gRPC es el tamaño de sus mensajes que está limitado a 4 Gb. Esto debido a que, en la construcción del mensaje, gRPC añade delante del *Protocol Buffer* una cadena de 5 bytes de los cuales el primero es para indicar si el mensaje está comprimido y los 4 restantes indican la longitud del contenido binario del *Protocol Buffer*, siendo que 4 bytes equivalen a 32 bits el mayor número codificable por 4 bytes es $2^{32} - 1\, bits = 4.294.967.295\, bits \approx 4\, Gb$.

También es importante destacar que gRPC usa HTTP/2 como protocolo de transporte sobre el que construye su protocolo de capa de aplicación. El uso de HTTP/2 es lo que otorga a gRPC con su gran capacidad de manejo de transmisiones en tiempo real. La diferencia principal entre HTTP/1.1 y HTTP/2 es que en el segundo toda comunicación se realiza sobre una sola conexión TCP que puede transportar cualquier número de flujo de bytes bidireccional, que pueden pertenecer a uno o varios mensajes diferentes, permitiendo la multiplexación de mensajes. Esto junto al uso de marcos para dividir grandes mensajes en pequeños trozos más manejables por la red hacen de HTTP/2 un protocolo bastante más eficiente que su predecesor.

La forma en la que gRPC usa HTTP/2 es, una vez el *stub* del cliente ha transformado la petición en el lenguaje del cliente a un mensaje con *Protocol Buffers*, el cliente envía las cabeceras del mensaje entre las que se encuentran el método HTTP (siempre POST) y el camino hacia el método que se quiere llamar (sigue la estructura de *{nombre del servicio}/{nombre del método}*) entre otros, de esta forma inicia la llamada con el servidor. Después el cliente envía los *Protocol Buffers* prefijados con su tamaño como *frames* de HTTP/2. Si el mensaje no cabe en un solo *frame*, este puede ocupar varios *frames*. Cuando no queda más información que transmitir, el cliente envía el último *frame* incluyendo la *flag END_STREAM*, indicando el final del mensaje. Y de forma similar, el servidor manda las cabeceras de respuesta, *frames* con el mensaje de respuesta si existe y el *frame* indicando el fin del mensaje siempre es enviado aparte en el caso de las respuestas. En la figura 2.30 se puede ver un diagrama que resume el funcionamiento de la comunicación cliente-servidor en gRPC. Y en la 2.31 se observa con más detalle la transferencia de *frames* HTTP/2.

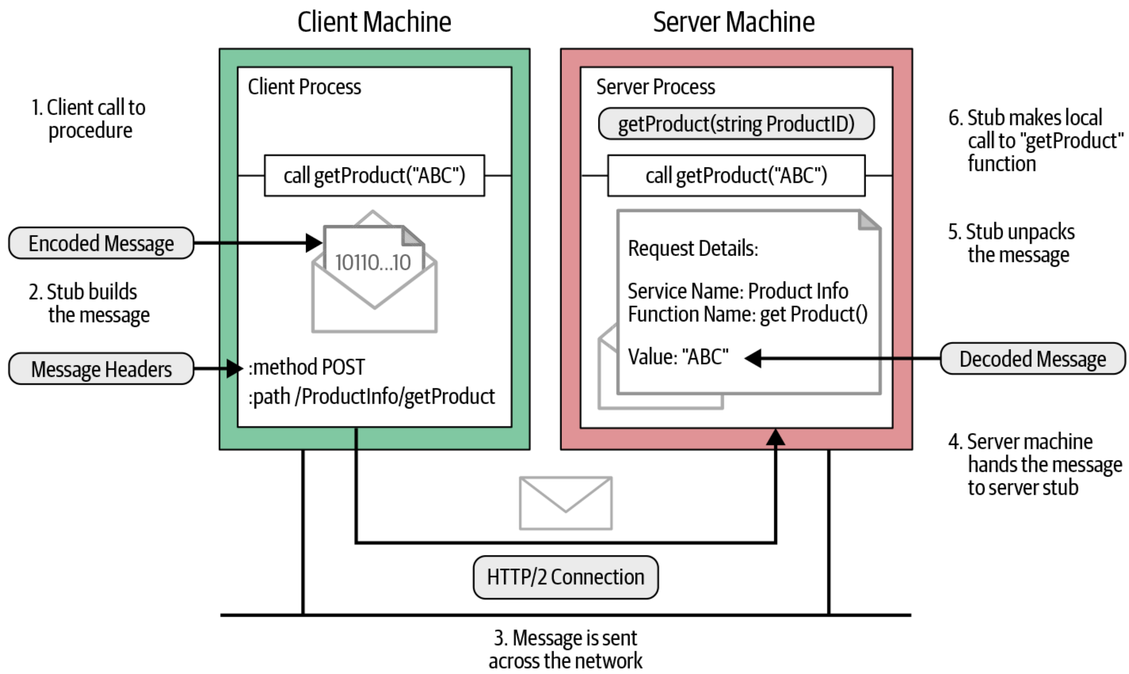


Figura 2.30 Cómo una llamada de procedimiento remoto funciona sobre la red, figura tomada del libro *gRPC Up & Runnig [32]*

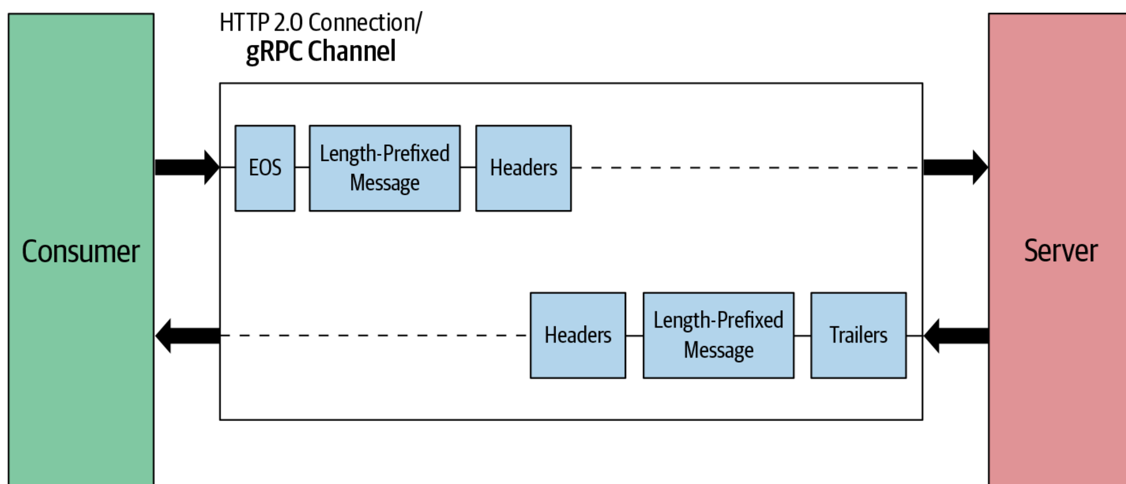


Figura 2.31 RPC simple: flujo de mensajes, figura tomada del libro *gRPC Up & Runnig [33]*.

Resumen

En el caso de gRPC se puede ver que es una forma relativamente simple de construir APIs pero que conlleva una infraestructura muy compleja e intrincada. gRPC se centra mucho más en las herramientas para crear un servicio que use su tecnología y en la manera en la que la comunicación se maneja entre cliente y servidor, dejando la parte de la interfaz mucho más sencilla.

Comparte similitudes con GraphQL, ambos compartiendo una forma de especificar "esquemas" con los que definir las estructuras de datos con las que se realizan las operaciones también definidas aquí de una forma clara y estricta.

Aunque difiere en la forma de organizar la información y la manera de acceder a ella y manipularla. Siendo en esto mucho más parecido a REST, en cuanto a la forma de organizar la información gRPC se podría llegar a definir como REST basado en acciones en vez de recursos.

gRPC ofrece una mezcla de la simplicidad de interfaz y uso de REST con la creación de contratos de interfaz entre cliente y servidor estrictos y rígidos que otorga mucha fiabilidad seguridad. Además, gRPC profundiza aún más en la facilidad de uso permitiendo realizar llamadas al servidor como si de una función interna se tratase, aunque supone un esfuerzo extra inicial en configurar y compilar las definiciones de los archivos *.proto* para obtener acceso a esta funcionalidad.

Pero el factor diferenciador de *gRPC* curiosamente son dos componentes estrechamente relacionados pero independientes, los *Protocol Buffers* y HTTP/2. Estos ofrecen una forma mucho más eficiente de transmitir información por la red, minimizando el tamaño de los mensajes y permitiendo el manejo continuo de la mayor cantidad de mensajes posibles de forma simultánea.

La ventaja que tiene gRPC es que estas dos tecnologías están muy bien integradas consigo gracias a una basta cantidad de herramientas que se han creado entorno a estas tecnologías. Haciendo de gRPC la forma más cómoda, eficiente y efectiva de utilizar los *Protocol Buffers* y HTTP/2 para transmitir los mensajes de una API. Aunque es cierto que estos pueden ser utilizados con REST o GraphQL, esto supondría un gran esfuerzo de desarrollo para integrarlos de forma satisfactoria.

Una gran desventaja es limitarse a HTTP/2, lo cual sopone una barrera de accesibilidad grande para casos en los que se quiere que la API sea accesible al mayor número de usuarios posible. Por esto, entre otras cosas, Martin Nally concluía en su artículo para Google Cloud *"gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design"*, que gRPC es más adecuado para APIs internas que intercomunican servicios privados y para aplicaciones que no estén basadas en navegadores, ya que estos no soportan de forma nativa gRPC. A menos que se requiera priorizar el rendimiento y la eficiencia, se necesite tipado fuerte o se haga un uso intensivo de transmisión de datos. También hay que tener en cuenta que, si la API requiere de intermediación por otros servicios, gRPC suele ser menos adecuado ya que estos servicios tienden a trabajar mejor con APIs REST [34].

En general, Martin Nally resume las ventajas de gRPC en su rendimiento, tipado fuerte, transmisión de datos, generación de código y concurrencia. Mientras

que las desventajas que remarca son su complejidad (especialmente en comparación con REST), el limitado soporte de navegadores, que requiere de software específico, el limitado soporte de *proxis* y la falta de mecanismos por defecto para ciertos casos de actualización de datos.

Revisando la literatura previa se pueden encontrar algunos trabajos como el de Sang Gyun Du, Jong Won Lee y Keecheon Kim, en el que muestran como gRPC muestra menos latencia que REST [35]. O como la tesis de Máster de Ismael Gonzalez, donde concluye que gRPC es una tecnología especialmente convincente en aplicaciones complejas a gran escala [36].

En conclusión, gRPC ofrece una clara interfaz para los clientes de la API sin la capacidad para realizar operaciones complejas y muy específicas, pero ofreciendo mucho rendimiento y una implementación sencilla, especialmente entre servicios internos.

2.2 Comparación de Tecnologías

A partir del análisis individual de cada tecnología API, es posible realizar una comparación teórica que permitirá establecer hipótesis sobre su comportamiento empírico. La revisión de la literatura proporciona un marco para identificar las fortalezas y debilidades de cada tecnología en distintos contextos de uso.

2.2.1 Revisión de la Literatura

Tras una revisión diligente, aunque no exhaustiva de la literatura existente en la que se comparen las tecnologías REST, GraphQL y gRPC se ha notado que no existen muchos estudios en los que se comparen estas tres tecnologías en concreto.

Para este proyecto solo se han encontrado diez estudios previos en los que se comparasen alguna de estas tecnologías entre sí, tampoco se ha hecho una búsqueda exhaustiva ya que no es el objetivo de este trabajo. De estos, cuatro incluían las tres tecnologías en su comparativa, cuatro comparaban REST con GraphQL, dos REST con gRPC y cero GraphQL con gRPC. Siendo que REST aparece en todos los estudios, GraphQL en ocho de ellos y gRPC en seis.

En la figura 2.32 se muestra una tabla de los estudios investigados junto a las tecnologías que cubren. Con este se puede inferir la popularidad de cada tecnología, siendo REST la más popular seguida de GraphQL y con gRPC en el

último puesto. Esto además coincidiría con lo hallado por la encuesta de Postman que se mencionó en la introducción. Véase figura 1.1. Aunque parece que gRPC suscita más interés entre investigadores que entre desarrolladores, ya que esta tecnología aparece en mayor proporción que en la encuesta de Postman.

Estudio	**REST**	**GraphQL**	**gRPC**
Comparative Review of Selected Internet Communication Protocols [5]			
Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC [7]			
Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC [6]			
Popular API Technologies: REST, GraphQL and gRPC [37]			
Comparison of REST and GraphQL web technology performance [38]			
GraphQL vs. REST: A Comparison of Runtime Performance [39]			
Performance comparison of REST vs GraphQL in different web environments: Node.js and Python [40]			
REST or GraphQL?: A Performance Comparative Study [41]			
Benchmarking the request throughput of conventional API calls and gRPC: A Comparative Study of REST and gRPC [43]			
Comparative Study of REST and gRPC for Microservices in Established Software Architectures [43]			

Figura 2.32 Tabla de tecnologías comparadas en la literatura previa, las casillas coloreadas indican que se incluye esta tecnología en el estudio.

Además, también se nota que existe un mayor interés general por comparar REST con GraphQL que con gRPC. Esto podría indicar que GraphQL sería un mejor sustituto para REST que gRPC o que proporciona más diferencias novedosas que gRPC.

2.2.2 Comparación de Criterios

En esta sección se analizan y comparan las tecnologías objeto de análisis de este estudio sobre una serie de criterios esenciales. Se ha limitado el análisis y comparación a esta lista de criterios para simplificar el trabajo y centrarlo en las partes más cruciales de estas tecnologías.

Rendimiento

- **gRPC** supera consistentemente a REST y GraphQL en cuanto a tiempo de respuesta y uso de CPU. Esto se debe principalmente al uso de

HTTP/2, el cual soporta multiplexación, transmisión de datos bidireccional y una serialización binaria eficiente gracias a los *Protocol Buffers*.

- **REST** generalmente otorga un rendimiento adecuado para la mayoría de aplicaciones web, especialmente en casos más sencillos y ligeros. Aunque muestra ser menos eficiente que gRPC cuando se requiere la transmisión y manejo de datos a gran escala o con operaciones más complejas.
- **GraphQL** comúnmente resulta en un alto uso de la CPU y tiempos de respuesta mayores, especialmente con consultas complejas involucrando datos anidados. Esto sugiere que GraphQL es más intensivo con el uso de recursos, particularmente en entornos con datos altamente complejos.

Escalabilidad

- **gRPC** sobresale en escalabilidad, particularmente para la arquitectura de microservicios, debido a sus eficientes mecanismos de comunicación que reducen la latencia y computo adicional. Su capacidad para manejar varias transmisiones de forma simultánea lo hacen ideal para escenarios de alta concurrencia.
- **REST** es escalable, pero puede sufrir de ineficiencias debido a la naturaleza sin estado del protocolo, llevando a potenciales problemas como el *overfetching* y *underfetching*.
- **GraphQL** ofrece mejor escalabilidad que REST en algunos casos permitiendo a los clientes pedir exactamente la información que requieren, reduciendo el número de peticiones. Aun así, los cuellos de botella que generan su rendimiento pueden limitar la escalabilidad bajo cargas intensivas.

Facilidad de Uso

- **REST** es la tecnología más comúnmente adoptada, con una vasta cantidad de herramientas y un gran soporte por la comunidad, haciéndolo más sencillo de implementar y depurar para la mayoría de los desarrolladores.
- **GraphQL** es más flexible y puede simplificar el desarrollo de lado del cliente permitiendo a los clientes consultar exactamente los datos que necesitan. Pero tiene una mayor curva de aprendizaje y requiere un diseño cuidadoso para evitar problemas de rendimiento y seguridad.
- **gRPC** ofrece capacidades muy potentes, pero puede llegar a ser más complejo de configurar e instalar debido a la necesidad de definir servicios usando *Protocol Buffers* y manejando datos binarios. Es menos conocido para muchos desarrolladores comparado con REST.

Flexibilidad

- **GraphQL** es más flexible en términos de permitir a los clientes definir la estructura de las respuestas. Esto reduce los problemas típicos de *overfetching* y *underfetching* de REST.
- **REST** es flexible dentro de las restricciones que impone, pero puede llegar a ser ineficiente cuando se trata de estructuras de datos complejas o anidadas.
- **gRPC** es flexible en términos de patrones de comunicación, pero requiere de una estructura más rígida definida por *Protocol Buffers*.

Seguridad

- **REST** y **GraphQL** típicamente dependen de HTTP/1.1, el cual puede ser asegurado usando métodos estándar como HTTPS. Sin embargo, la capacidad de GraphQL de exponer consultas complejas podría introducir vulnerabilidades de seguridad como se vio anteriormente si no se manejan de forma adecuada.
- **gRPC** se beneficia de las capacidades de seguridad incorporadas en HTTP/2 y la naturaleza binaria de los *Protocol Buffers*, que pueden ser más seguros frente a cierto tipo de ataques. Sin embargo, su complejidad puede hacer más difícil su correcta implementación.

Compatibilidad

- **REST** es la tecnología más compatible en multitud de plataformas y servicios debido a estar respaldado por el protocolo universalmente soportado HTTP/1.1.
- **gRPC** es altamente compatible dentro de entornos modernos y políglotas. Pero requiere soporte específico para los *Protocol Buffers*, lo que podría no estar disponible en todos los lenguajes o plataformas.
- **GraphQL** también es ampliamente compatible, pero su implementación del lado del servidor puede ser más compleja, especialmente en lenguajes o entornos sin un soporte robusto de GraphQL.

Capacidad de Tipado

- **gRPC** tiene un sistema de tipado fuerte impuesto por los *Protocol Buffers*, que aseguran consistencia y ayudan a prevenir errores en tiempo de compilación.
- **GraphQL** soporta un sistema de tipado que permite al cliente especificar la estructura y tipos de los datos que espera, pero es menos rígido que gRPC.
- **REST** normalmente no impone ningún tipado estricto, delegando esta responsabilidad a los formatos de datos que usa por debajo, como JSON o

XML, lo cual puede conducir a inconsistencias y errores en tiempo de ejecución.

Para resumir la evaluación de REST, GraphQL y gRPC siguiendo estos criterios se ha realizado la tabla de la figura 2.33 en la que se simplifica la evaluación y análisis realizado.

Criterio	REST	GraphQL	gRPC
Rendimiento	**Moderado**, adecuado para aplicaciones sencillas	**Bajo**, alto uso de CPU para consultas complejas	**Alto**, optimizado para comunicación de baja latencia
Escalabilidad	**Moderada**, puede sufrir de ineficiencias	**Buena**, mejor para obtención de datos complejos	**Excelente**, ideal para microservicios con alta concurrencia
Facilidad de Uso	**Alta**, ampliamente adoptado y soportado	**Moderada**, flexible pero con una mayor curva de aprendizaje	**Moderada**, potente pero con compleja configuración inicial
Flexibilidad	**Moderada**, limitado por restricciones de HTTP/1.1	**Alta**, obtención personalizable de datos	**Moderada**, patrones de comunicación flexibles pero estructura rígida
Seguridad	**Buena**, aplica la seguridad estándar de HTTP	**Buena**, pero requiere un manejo cuidadoso de las consultas	**Alta**, se beneficia de las prestaciones de seguridad de HTTP/2
Compatibilidad	**Alta**, universal en todas las plataformas	**Alta**, pero con complejas implementaciones de lado del servidor	**Moderada**, requiere soporte de

			Protocol Buffers
Capacidad de Tipado	**Baja**, depende de formatos como JSON/XML	**Moderada**, sistema de tipos incluido en el lenguaje de consultas	**Alta**, impuesta por los *Protocol Buffers*

Figura 2.33 Tabla comparativa de REST, GraphQL y gRPC basada en criterios fundamentales.

Con esta, junto a las más extensas explicaciones proveídas anteriormente, pueden servir como guía teórica para desarrolladores a la hora de elegir una de estas tecnologías para desarrollar sus APIs, guiando su toma de decisiones basada en requisitos de proyecto específicos, como la necesidad de rendimiento, escalabilidad o facilidad de uso. A su vez también sirve de punto de partida para desarrollar una rúbrica de evaluación con la cual analizar y comparar estas tecnologías en las pruebas realizadas sobre el proyecto real.

2.2.3 Rúbrica de Evaluación

Para realizar una comparación empírica efectiva, se ha diseñado una rúbrica a partir de los criterios de análisis usados en la sección anterior que evaluará cada tecnología en una escala de 1 a 5. Esta rúbrica será aplicada durante la fase de experimentación del estudio, permitiendo una evaluación objetiva y cuantitativa del desempeño de cada API.

En este estudio no se pretende ni se ha podido evaluar las tecnologías API en todos los criterios de la rúbrica debido a limitaciones temporales. Hacerlo conllevaría mucho más tiempo del disponible para la realización de este trabajo. Aquí solo se analizará de forma empírica el rendimiento de cada tecnología. Sin embargo, esta rúbrica puede ser útil para futuros investigadores que quieran profundizar en ella y analizar otros criterios no cubiertos en este proyecto.

Esta rúbrica podría además ser usada por desarrolladores y arquitectos de sistemas para elegir tecnología API de una forma clara y directa sopesando todos los criterios clave que pueden influir en la decisión. Una forma de aplicar esta rúbrica es asignando un multiplicador numérico a cada criterio según su relevancia. Luego, se multiplica la puntuación de la rúbrica por ese valor y se suman los resultados, obteniendo así una nota ponderada para la tecnología API en función de los requisitos específicos.

Con un ejemplo práctico para que se vea más claro. Se supone que un desarrollador quiere realizar una API. Esta API requiere de mucho rendimiento

y escalabilidad, la facilidad de uso es prácticamente irrelevante, la flexibilidad es medianamente importante, la seguridad es clave, la compatibilidad es moderadamente relevante y la capacidad de tipado se aprecia, pero no es necesaria. Asignando una valoración del 1 al 5 para cada criterio en cada tecnología de forma arbitraria, los resultados que saldrían serían los mostrados en la figura 2.34.

Criterios	Requisitos	REST	GraphQL	gRPC	REST ponderado	GraphQL ponderado	gRPC ponderado
Rendimiento	9	4	2	5	36	18	45
Escalabilidad	9	3	4	5	27	36	45
Facilidad de uso	1	5	3	3	5	3	3
Flexibilidad	5	3	5	2	15	25	10
Seguridad	10	4	4	5	40	40	50
Compatibilidad	6	5	4	2	30	24	12
Tipado	4	1	3	5	4	12	20
TOTAL					**157**	**158**	**185**

Figura 2.34 Rúbrica de evaluación de ejemplo aplicada a posible caso real para elegir tecnología.

En este caso, gRPC sería claramente la mejor elección para este caso específico y no habría mucha diferencia entre decantarse por REST o GraphQL.

Para asegurar una evaluación objetiva de cada criterio, se ha dividido cada criterio en varias métricas e indicadores que serán evaluados por separado para determinar la evaluación final de la tecnología sobre este criterio en específico. La rúbrica resultante se muestra en la figura 2.35.

Criterio	*Métrica*	*Descripción*
Rendimiento	**Latencia**	Tiempo que tarda en completarse un ciclo de solicitud-respuesta.
	Tasa de transferencia efectiva (*throughput*)	Número de solicitudes procesadas con éxito dentro de un marco de tiempo específico.
	Utilización de Recursos	Consumo de CPU y memoria durante las operaciones.
	Manejo de Concurrencia	Capacidad para manejar múltiples solicitudes simultáneas sin degradación.
	Tasas de Error	Frecuencia y tipo de errores encontrados durante las operaciones.
	Uso de Ancho de Banda	Cantidad de datos transmitidos por la red para solicitudes y respuestas.
Escalabilidad	**Eficiencia de Escalado Horizontal**	Capacidad para distribuir la carga de manera efectiva entre múltiples servidores o servicios.
	Eficiencia de Escalado Vertical	Capacidad para utilizar recursos adicionales (CPU, memoria) de manera eficiente dentro de un solo servidor.
	Compatibilidad con Balanceo de Carga	Facilidad de integración con balanceadores de carga y distribución uniforme de solicitudes.
	Tolerancia a Fallos	Capacidad para mantener las operaciones a pesar de fallos o errores.
	Manejo de Sesiones	Eficiencia en la gestión y mantenimiento de sesiones de usuario en entornos escalables.

	Mecanismos de Caché	Efectividad en el uso de cachés para mejorar el rendimiento a escala.
Facilidad de Uso	**Curva de Aprendizaje**	Tiempo y esfuerzo necesarios para que los desarrolladores se vuelvan competentes en el protocolo.
	Herramientas y Ecosistema	Disponibilidad y madurez de las herramientas, bibliotecas y frameworks que soportan el protocolo.
	Complejidad de Diseño de API	Esfuerzo requerido para diseñar e implementar APIs usando el protocolo.
	Documentación y Descubribilidad	Facilidad para documentar y entender los puntos de acceso y operaciones del API.
	Implementación del Lado Cliente	Esfuerzo y complejidad involucrados en implementar clientes que consumen las APIs.
	Pruebas y Depuración	Facilidad para escribir pruebas y depurar problemas durante el desarrollo y despliegue.
Flexibilidad	**Capacidades de Consulta**	Capacidad para recuperar datos con diferentes niveles de especificidad y complejidad.
	Evolución del Esquema y Versionado	Soporte para evolucionar modelos de datos y mantener compatibilidad retroactiva.
	Soporte para Comunicación en Tiempo Real	Capacidad para soportar actualizaciones de datos en tiempo real y transmisión de datos.
	Flexibilidad en el Manejo de Errores	Robustez y granularidad en el manejo y reporte de errores.
	Autenticación y Autorización	Soporte y flexibilidad en la implementación de medidas de seguridad y controles de acceso.
	Paginación y Agrupación	Eficiencia y soporte para recuperar grandes conjuntos de datos en partes manejables.
Seguridad	**Seguridad en el Transporte**	Soporte para protocolos de transporte seguro (p. ej., TLS/SSL).
	Mecanismos de Autenticación	Soporte para verificar la identidad del cliente y del servidor.
	Controles de Autorización	Capacidad para hacer cumplir los controles de acceso y permisos.
	Validación y Saneamiento de Datos	Efectividad para prevenir ataques de inyección y garantizar la integridad de los datos.
	Registro de Auditoría y Monitoreo	Capacidad para rastrear y monitorear las actividades del sistema para auditoría de seguridad.
	Resistencia a Ataques Comunes	Robustez contra amenazas de seguridad prevalentes (p. ej., ataques DDoS, ataques de intermediarios).
Compatibilidad e Interoperabilidad	**Soporte Multilenguaje**	Disponibilidad y madurez de las implementaciones del protocolo en varios lenguajes de programación.
	Soporte Multiplataforma	Compatibilidad con diferentes sistemas operativos y entornos de despliegue.
	Integración con Sistemas Legados	Capacidad para integrar o adaptarse a sistemas y protocolos legados existentes.
	Integración con Servicios de Terceros	Facilidad de conexión con servicios externos y APIs.
	Soporte para Proxies y Gateways API	Compatibilidad con proxies y gateways API para gestionar y enrutar solicitudes.
	Soporte para Clientes Móviles y Web	Efectividad en el soporte para varias plataformas cliente y entornos.
	Soporte para Tipado Fuerte	Capacidad para hacer cumplir tipos de datos estrictos en solicitudes y respuestas.

Capacidad de Tipado y Gestión de Esquemas	**Definición y Evolución de Esquemas**	Facilidad para definir, actualizar y gestionar esquemas de datos a lo largo del tiempo.
	Mecanismos de Validación	Soporte incorporado para la validación de datos según los esquemas y reglas definidas.
	Generación de Documentación	Capacidad para autogenerar documentación a partir de esquemas y definiciones.
	Introspección de Esquemas	Capacidad para consultar y explorar los esquemas de manera programática.
	Seguridad en Tipado en Tiempo de Compilación	Soporte para detectar errores de tipo durante el desarrollo en lugar de en tiempo de ejecución.

Figura 2.35 Rúbrica de evaluación completa.

En resumen, mientras que REST ofrece simplicidad y amplia compatibilidad, sufre en rendimiento y flexibilidad en comparación con GraphQL y gRPC. GraphQL sobresale en flexibilidad, pero puede ser más complejo de implementar y optimizar, mientras que gRPC es la opción más robusta en términos de rendimiento y escalabilidad, aunque con limitaciones en su adopción. Estas diferencias teóricas serán puestas a prueba en el análisis empírico que sigue.

3

Metodología

En este capítulo se describe cómo se diseñó y ejecutó el estudio, dividido en cuatro apartados. Primero, se presenta el objeto de análisis, donde se detallan las entidades sobre las que se realizará el análisis comparativo, incluyendo el diseño general de la API de la aplicación y las herramientas y técnicas empleadas para su desarrollo, ejecución y análisis de resultados. Luego, se explica la metodología de trabajo, abordando la organización del desarrollo de la aplicación, las limitaciones metodológicas encontradas y el alcance del estudio. A continuación, se describe cómo se distribuyó el trabajo en las diferentes fases del proyecto. Finalmente, se exponen de manera clara las limitaciones del enfoque metodológico y se especifica el alcance exacto del estudio.

3.1 Objeto de Análisis

Para la realización de la parte práctica de este estudio, se ha decidido replicar la API de una red social ficticia. La arquitectura general de esta aplicación se compone de tres servidores, cada uno implementando una de las tecnologías API estudiadas: REST, GraphQL y gRPC, todos escritos en Go.

Go es un lenguaje de programación compilado, de alto nivel, tipado estático, concurrente, y con recolección de basura, diseñado por Google. Se ha

elegido este lenguaje para el desarrollo del servidor debido a su alto rendimiento y facilidad de uso. El objetivo es minimizar la influencia de factores externos, como la experiencia de desarrollo deficiente asociada con el lenguaje, sin sacrificar el rendimiento. Esta elección es crucial, ya que otras opciones como Node.js, aunque ofrecen una excelente experiencia de desarrollo, presentan un rendimiento inferior que podría sesgar los resultados del estudio al comparar tecnologías API, especialmente si las librerías no están optimizadas. Con Go, se evita este problema al no requerir ni permitir el uso excesivo de abstracciones que podrían ralentizar los programas.

Los servidores se conectan a una base de datos PostgreSQL y a Redis utilizando un Mapeo Objeto-Relacional (ORM) y el cliente oficial de Redis para Go, go-redis. PostgreSQL se ha elegido por su popularidad y alto rendimiento en casos de uso como el de este estudio, aunque otras bases de datos relacionales como MySQL u Oracle DB podrían haberse utilizado sin afectar significativamente los resultados. Redis se ha seleccionado por ser la base de datos en memoria más utilizada a nivel mundial, lo que permite realizar operaciones similares a las de un caché, permitiendo evaluar el comportamiento de cada tecnología API en estos escenarios. Las pruebas de integración se han llevado a cabo utilizando Postman [8], ejecutando scripts que verifican que las respuestas de los servidores son correctas bajo diversas condiciones.

Uno de los aspectos clave del estudio es la obtención de datos de rendimiento bajo diferentes escenarios. Para ello, se han desarrollado varios scripts utilizando Grafana k6, una herramienta de pruebas de carga de código abierto, conocida por ser amigable para el desarrollador y altamente extensible. Esta herramienta permite simular una gran cantidad de usuarios concurrentes bajo diferentes condiciones y comportamientos, evaluando el rendimiento de todos los endpoints [9].

Para analizar mejor los datos obtenidos, se ha creado un script en Python utilizando varias librerías especializadas. Este script procesa los datos y genera gráficas y visualizaciones que facilitan su interpretación en contexto. Python es la elección ideal para esta tarea debido a su extensa comunidad y la disponibilidad de recursos y librerías para análisis de datos, permitiendo realizar análisis detallados con menor esfuerzo.

3.2 Metodología de Trabajo

Para la organización y ejecución del trabajo necesario en este estudio, se ha optado por seguir la metodología ágil SCRUM, principalmente debido a su flexibilidad y adaptabilidad. Dado que este estudio probablemente requiera ajustar el enfoque a medida que se obtienen nuevos datos y se identifican problemas imprevistos, es esencial contar con una metodología que permita adaptarse rápidamente a los cambios en los requisitos o descubrimientos técnicos durante el proyecto.

La implementación de SCRUM en este estudio se realiza mediante sprints en los que se desarrollan versiones incrementales y funcionales de los servidores API. En la primera iteración, se crea una versión de la aplicación que solo implementa un endpoint de la API.

Inicialmente, se planificaron sprints de dos semanas de duración cada uno. Sin embargo, para reducir la carga diaria, se decidió ampliar la duración de los sprints a cuatro semanas, manteniendo constantes las horas totales requeridas para completar el estudio. Durante el desarrollo de los servidores y aplicaciones, se documenta el tiempo consumido para generar cada API y la facilidad de desarrollo con cada una de ellas.

3.3 Fases de Trabajo

Inicialmente, se había planeado desarrollar una API más sencilla con solo un endpoint para cada operación CRUD (Create, Read, Update, Delete). Sin embargo, este enfoque no permitía desarrollar un objeto de análisis lo suficientemente realista o complejo como para derivar conclusiones sólidas. Además, su desarrollo resultaba demasiado simple para cumplir con la extensión necesaria de este estudio.

Finalmente, las fases de trabajo quedaron definidas de la siguiente manera:

1. **Diseño y Creación de la BBDD, Conexión BBDD-Servidor, Seeding, Reset, y Endpoints Iniciales**: Se diseñó e implementó la base de datos de la red social, se desarrollaron scripts para la conexión entre el servidor y la BBDD, la inicialización de la BBDD con datos ficticios

(seeding) y su restablecimiento (reset). Además, se implementaron los endpoints iniciales, como “/redis” y “/healthcheck”.

2. **Implementación del Endpoint "/users"**: Se desarrolló el endpoint “/users”, que constituye la columna vertebral de la aplicación, permitiendo la realización de la mayoría de las operaciones fundamentales.
3. **Implementación de los Endpoints "/posts", "/comments", "/messages" y "/chats"**: Se completaron los endpoints restantes de la API, que soportan las funcionalidades esenciales de la red social.
4. **Pruebas Unitarias y de Integración**: Se diseñaron y ejecutaron pruebas unitarias y de integración para asegurar la correcta implementación de las APIs.
5. **Desarrollo y Ejecución de Scripts para el Análisis Comparativo**: Se crearon y ejecutaron scripts de Grafana k6 para pruebas de rendimiento, así como scripts en Python para el análisis de los datos obtenidos.
6. **Recopilación e Interpretación de Datos**: Se recuperaron y analizaron los resultados obtenidos en las fases anteriores para extraer conclusiones sobre el rendimiento de las tecnologías API.
7. **Redacción de la Memoria**: Se redactó la memoria que documenta todo el trabajo realizado, así como los hallazgos obtenidos.
8. **Revisión Final**: Se revisó la memoria en su totalidad para asegurar su calidad y adecuación antes de su entrega.

3.3 Limitaciones Metodológicas y Alcance del Estudio

3.3.1 Limitaciones Metodológicas

- **Contexto Específico**: El estudio se centra en el caso específico de una red social ficticia. Aunque es un caso completo que permite realizar una gran variedad de operaciones, las conclusiones obtenidas podrían no ser aplicables directamente a otros tipos de aplicaciones con diferentes requisitos o arquitecturas.
- **Selección de Tecnologías y Herramientas**: La elección de Go como lenguaje de programación y PostgreSQL como base de datos, aunque justificada por su rendimiento y popularidad, podría influir en los resultados. Otras tecnologías podrían producir resultados diferentes en términos de rendimiento y facilidad de desarrollo.
- **Simulación de Usuarios**: Las pruebas de rendimiento realizadas con Grafana k6 simulan una gran cantidad de usuarios concurrentes, pero estas simulaciones pueden no capturar completamente la complejidad de un entorno de producción real, donde factores como la red, la carga del servidor, y la variabilidad del tráfico pueden afectar el rendimiento.

- **Alcance de las Pruebas de Rendimiento**: Aunque se han realizado pruebas exhaustivas de carga y rendimiento, estas pruebas no cubren todos los posibles escenarios de uso, como operaciones distribuidas o sistemas de gran escala.
- **Subjetividad en la Evaluación**: La experiencia de desarrollo, aunque se ha documentado objetivamente en términos de tiempo y errores, sigue siendo una evaluación en parte subjetiva. Diferentes desarrolladores con distintos niveles de experiencia podrían tener percepciones y resultados diferentes.
- **Factores No Considerados**: El estudio no aborda aspectos como la seguridad, la compatibilidad con sistemas heredados, la facilidad de mantenimiento a largo plazo, o el soporte de la comunidad, que también podrían ser determinantes en la elección de una tecnología API.
- **Tiempo y Recursos Disponibles**: Las limitaciones de tiempo y recursos pueden haber restringido la profundidad del análisis, así como la cantidad de pruebas que se pudieron realizar. Un análisis más prolongado o con mayores recursos podría haber permitido una exploración más detallada.

3.3.2 Alcance del Estudio

- **Tecnologías Evaluadas**: Este estudio se centra en la comparación de tres tecnologías API específicas: REST, GraphQL y gRPC. No se incluyen otras tecnologías emergentes o alternativas como SOAP, OData, o websockets, lo que delimita el alcance del análisis.
- **API de Red Social Ficticia**: La aplicación analizada es una API de red social ficticia, diseñada para emular casos de uso típicos de una aplicación social. El estudio no explora aplicaciones en otros dominios, como comercio electrónico, IoT, o servicios financieros, lo que limita la aplicabilidad directa de los resultados a otros contextos.
- **Métricas de Evaluación**: El estudio se centra en métricas de rendimiento (tiempo de respuesta, uso de recursos) y en la experiencia de desarrollo (facilidad de implementación, número de errores). Otros aspectos importantes, como la seguridad, la escalabilidad en producción, y la interoperabilidad, están fuera del alcance de este análisis.
- **Expansión del Estudio**: Aunque este trabajo sienta las bases para futuras investigaciones, no se abordan en profundidad temas como la evolución de las tecnologías estudiadas, su adaptación a nuevas arquitecturas (como microservicios o serverless), o su comportamiento en entornos distribuidos a gran escala.

4

Desarrollo del Proyecto

Este capítulo detalla el proceso de análisis, diseño e implementación de la API, junto con las pruebas de integración, carga, rendimiento y los scripts para el análisis de datos. Se inicia con una introducción que presenta la estructura general del proyecto y justifica el diseño global de la aplicación, explicando el caso de uso real que imita. Luego, se describe el diseño de la base de datos, incluyendo el modelo entidad-relación, seguido por el diseño de la API, donde se detallan los endpoints y sus funcionalidades, además de las interfaces de entrada y salida. Más adelante, se presentan los resultados de las pruebas de integración realizadas con Postman, y finalmente se explica el diseño y desarrollo de los scripts para pruebas de carga y rendimiento con k6, así como los scripts de análisis de datos desarrollados en Python.

4.1 Introducción y Estructura General

4.1.1 Contenido

Para realizar este estudio comparativo entre REST, GraphQL y gRPC se ha desarrollado una aplicación con cada una de estas tecnologías sobre las que realizar las pruebas y análisis. Para esto se ha decidido por simular una red social por ser un caso de uso real que realiza un uso intensivo de datos, con varias entidades relacionadas entre sí y en el que las API juegan un papel esencial para integrar la red social con otros microservicios accesorios y para el público general.

En concreto se han desarrollado tres versiones de la misma aplicación de servidor, cada una usando una tecnología de API diferente. Todas ellas se han desarrollado en Go por su simplicidad y gran rendimiento. Los servidores se encuentran conectados a una base de datos relacional PostgreSQLy a una base de datos en memoría Redis.

Para el desarrollo de cada servidor se ha hecho de uso de las librerías más populares y punteras en la tecnología API correspondiente. Esto para intentar dar la mejor oportunidad a cada tecnología de mostrar su capacidad real o lo más cercano a ella. En el caso de gRPC es la implementación oficial del equipo de gRPC. Como no existe un equipo oficial que mantenga REST, ya que este no es una tecnología tan concreta como gRPC, existen muchas librerías para desarrollar APIs REST con Go. Finalmente se ha optado por usar Fiber por su popularidad y por el rendimiento tan elevado que ha muestra en sus propios benchmarks [43].

En el caso de GraphQL, aunque si existe una comunidad que mantiene y da soporte a este tecnología, la librería que se ha acabado escogiendo no es la oficial. Finalmente se ha escogido gqlgen por su popularidad, alta aunque menor que la libreria ofical de GraphQL, pero sobre todo por su rendimiento demostrado en ciertos benchmarks [44] y por su generación de código automático según el esquema, el cual aporta una gran facilidad de desarrollo. También es importante destacar que se ha usado Fiber como interfaz entre la petición HTTP y el entorno de ejecución de GraphQL, recibiendo y redirigiendo todas las peticiones HTTP en el endopoint */graphql* hacia el entorno de ejecución. Por lo que no se podría achacar una diferencia de rendimiento a usar enrutador HTTP diferente. En la figura 4.1 se muestra un diagrama del diseño de la aplicación a grandes rasgos. También se incluye los scripts de prueba y análisis con los que estas aplicaciones se comunicarán.

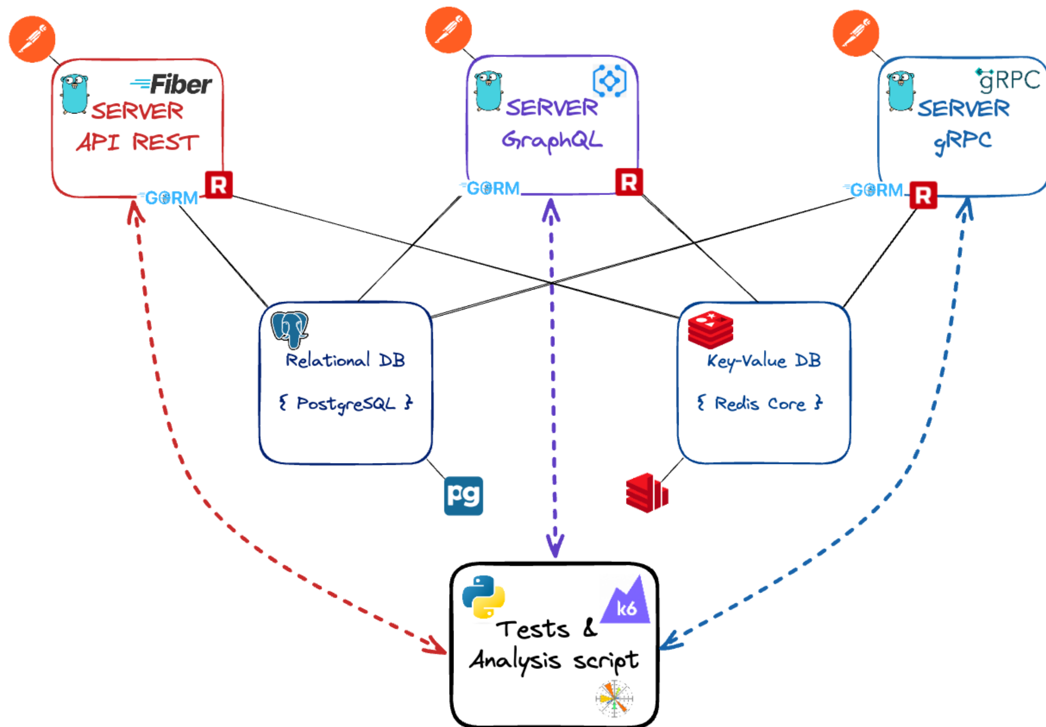


Figura 4.1 Diagrama del diseño general del proyecto.

4.1.2 Forma

El proyecto se ha organizado en un repositorio git siguiendo una estructura Modelo-Vista-Controlador (MVC) adaptada a cada tecnología API. En todos los casos el Modelo es compartido y sirve de interfaz para realizar operaciones con las BBDD a través del ORM y del cliente de Redis. Este se ubica en una carpeta compartida llamada *lib* que contiene librerías que todos los proyectos usan.

Para la Vista en el caso de REST simplemente se traduce el modelo a un subconjunto del mismo denominado DTO (Objeto de Transmisión de Datos por sus siglas en inglés) que solo incluye los campos que se quieren mostrar al usuario. También se podría haber conseguido utilizando la etiqueta *`json:"-"`* para indicar que ese campo no debe de ser codificado, pero suele ser más claro separar estas representaciones. El DTO se codifica en JSON utilizando la librería *encoding/json* incluido en el paquete estándar de Go. Simplemente se añaden anotaciones a los modelos con los nombres de los campos que se desea que tengan en JSON y se manda a codificarlos en JSON con el comando *json.Marshall(objeto)*. Se muestra un ejemplo práctico en la figura 4.2.

```go
package main

import (
	"encoding/json"
	"fmt"
	"os"
)

type ReadUserDTO struct {
	ID        uint64 `json:"id"`
	Name      string `json:"name"`
	Email     string `json:"email"`
	CreatedAt string `json:"createdAt"`
	UpdatedAt string `json:"updatedAt"`
}

func main() {
	user := ReadUserDTO{
		ID:        1,
		Name:      "name",
		Email:     "name@mail.com",
		CreatedAt: "2024-09-01",
		UpdatedAt: "2024-09-01",
	}
	u, err := json.Marshal(user)
	if err != nil {
		fmt.Println("error:", err)
	}
	os.Stdout.Write(u)
}
```

OUTPUT

```
{"id":1,"name":"name","email":"name@mail.com","createdAt":"2024-09-01","updatedAt":"2024-09-01"}
```

Figura 4.2 Ejemplo de codificación de modelo en JSON en Go.

En el caso de GraphQL la Vista es también un JSON, pero esta es montada por el código autogenerado por la librería gqlgen. Por lo que su implementación es una caja negra para el desarrollador.

Algo parecido ocurre con gRPC, cuya Vista no es JSON, son directamente los *Protocol Buffers*, pero que tampoco implementa su conversión de modelo a Vista directamente. De esto se encarga el código autogenerado por el compilador *protoc*.

En cuanto a los Controladores, en GraphQL son los resolvers, en gRPC son los servicios y sus métodos y en REST son funciones definidas directamente por el desarrollador que en este caso se han llamado controladores. Todos y cada uno de ellos son completamente implementados por el desarrollador haciendo uso de la librería compartida del Modelo. Pero en el caso de GraphQL y gRPC, la firma de la función (su nombre parámetros de entrada y tipos de salida) son creados automáticamente por sus respectivas librerías; y no se tiene que montar la Vista en estos ya que de eso también se encargan las librerías. Aunque bien es cierto que en el caso de GraphQL hay que hacer ciertas conversiones adicionales

entre el Modelo y la estructura de datos que espera el código autogenerado por gqlgen.

Todo esto, junto a algunos archivos de configuración, pruebas unitarias y el punto de entrada del programa (archivo *main.go*) reside en un repositorio git con 4 aplicaciones Go independientes con su propio módulo Go (para gestión de dependencias). Estas aplicaciones corresponden a cada implementación para cada tecnología API y la librería compartida de los métodos y funciones para interactuar con las BBDD, es decir, el Modelo.

Para evitar conflictos y gestionar varias aplicaciones Go en un solo repositorio se ha hecho uso de los Go Workspaces que ofrece el lenguaje Go para esta misma tarea. Esto permite la gestión eficiente de dependencias compartidas entre todas las aplicaciones Go e interconectar estas aplicaciones entre sí como dependencias sin necesidad de publicar los módulos.

De forma adyacente se encuentran carpetas para almacenar documentos relevantes, imagenes, los resultados de las pruebas de integración con Postman, los scripts k6 de pruebas de carga y los de análisis de datos, el README.md y un docker-compose.yml para desplegar todas las dependencias con Docker. En la figura 4.3 se puede ver la estructura de carpetas y archivos del repositorio de una forma visual

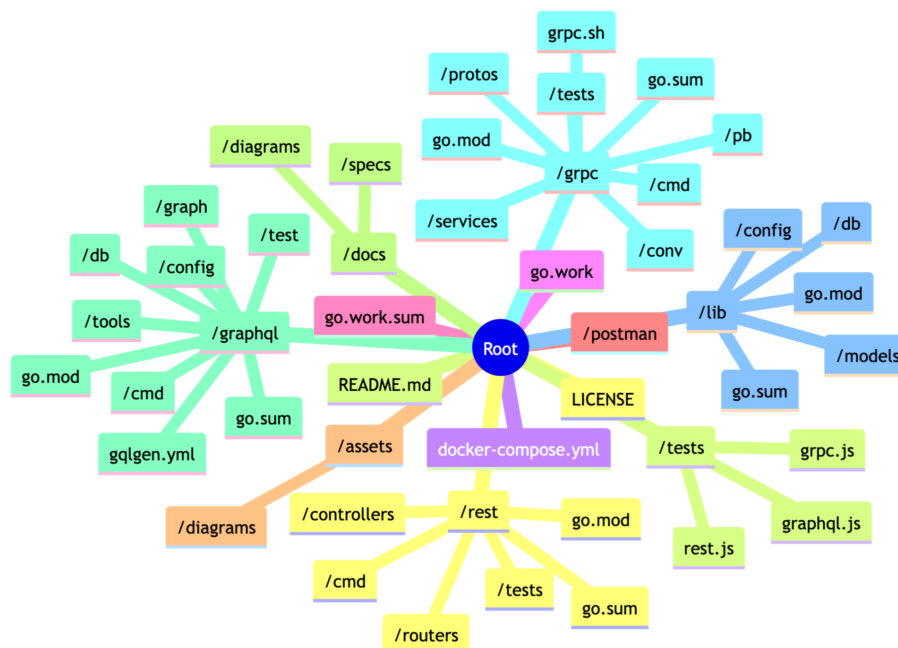


Figura 4.3 Estructura de carpetas del repositorio Git del proyecto creado con Mermaid [45].

4.2 Base de Datos

4.2.1 Tablas y Modelo Entidad-Relación

La base de datos consiste en las siguientes entidades:

- **User**: estos son los usuarios de la red social. Tienen nombre, email, publicaciones que realizan, mensajes que envían en chats, comentarios que dejan en publicaciones y chats en los que participan.
- **Post**: son las publicaciones que realizan los usuarios. Están formadas por un título y un contenido, y tienen asociado el usuario que creó la publicación y comentarios que realizan otros.
- **Comment**: refiere a los comentarios que dejan los usuarios en las publicaciones. Almacenan el contenido de texto del comentario y el ID del usuario que lo escribió y de la publicación a la que refiere.
- **Chat**: son los espacios en los que unos usuarios específicos intercambien mensajes entre sí. Guarda estos mensajes y los participantes del chat.
- **Message**: son los mensajes de texto que usuarios envían a través de un chat determinado. Está compuesto de su contenido de texto y el ID del usuario que lo redactó y el chat en el que se envió.
- **Participant**: es una tabla intermedia entre usuarios y chats que permite que un usuario pueda estar en varios chats y un chat pueda tener varios usuarios. Básicamente mapea un usuario en concreto a un chat en específico.

En la figura 4.4 se muestra el diagrama de Entidad-Relación de la base de datos:

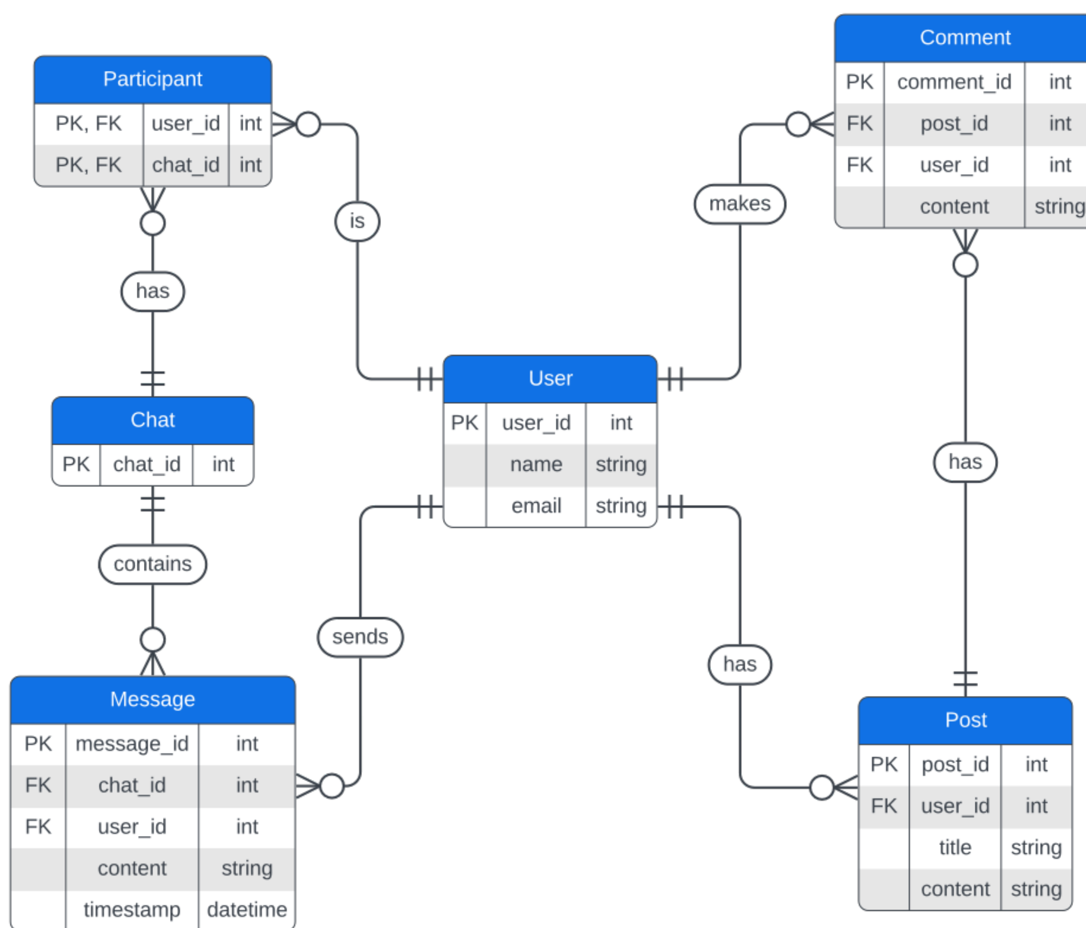


Figura 4.4 Diagrama de Entidad-Relación de la base de datos.

4.2.2 Operaciones CRUD

Como se ha adelantado con anterioridad en el apartado 3.1, se ha usado un ORM para conectar la base de datos PostgreSQL con las aplicaciones de servidor en Go. Esto ayuda a realizar operaciones sobre la base de datos de una forma más intuitiva operando sobre estructuras de datos del propio lenguaje de programación sin la necesidad de realizar consultas SQL. Suelen ser bastante estándar en la industria y por eso entre otras razones se ha optado por incluirlo en el desarrollo.

Parar estandarizar las comunicaciones con la base de datos se define bajo la carpeta *lib/db* un archivo Go para cada tabla con las operaciones que se pueden realizar sobre estas. Por ejemplo, se encuentra el archivo *users.go* en el que se definen operaciones para actualizar o crear un usuario nuevo (*upsert* [46]), obtener los datos de un usuario, actualizar parcial o completamente un usuario, borrarlo, listar todos los usuarios y encontrar un usuario según su email. Este tipo de operaciones es lo que se le suele llamar CRUD (Crear, Leer, Actualizar y Borrar por sus siglas en inglés).

4.2.3 Seeding

También se encuentra en esta misma carpeta un archivo *seed.go* que permite poblar la base de datos con datos ficticios (*seeding*) y eliminar todas las filas de todas las tablas de la base de datos (*reset*). El algoritmo de *seeding* hace uso de la librería *jaswdr/faker* [47], para generar datos verosímiles falsos de usuarios, publicaciones, mensajes, etc. Para configurar el alcance y la extensión del algoritmo de *seeding* se pueden usar variables de entorno como *NUM_OF_USERS* para definir el número de usuarios que se quieren crear. Estas se guardan en un objeto de configuración al que luego accede el algoritmo de *seeding*. Si no se configuran estas variables de entorno se usa un valor por defecto. El algoritmo de *seeding* se ha implementado en una función de Go basada en una transacción SQL para garantizar la integridad de los datos y prevenir inconsistencias si ocurre un error en el proceso.

Inicialización

La función comienza obteniendo la configuración del sistema y creando el generador de datos ficticios de *jaswdr/faker* como se ve en la figura 4.5.

```go
conf := config.GetConfig()
fake := faker.New()
```

Figura 4.5 Código de inicialización del algoritmo de *seeding* de la base de datos.

Manejo de Transacciones

Se utiliza una transacción para garantizar que los datos se escriban de forma atómica. Si se produce un error en cualquier parte del proceso, la transacción se revierte (rollback) para evitar que la base de datos quede en un estado inconsistente. Se puede ver el código responsable en la figura 4.6. Este bloque asegura que cualquier *panic* o error sea manejado de forma adecuada y se realice un *rollback* si es necesario.

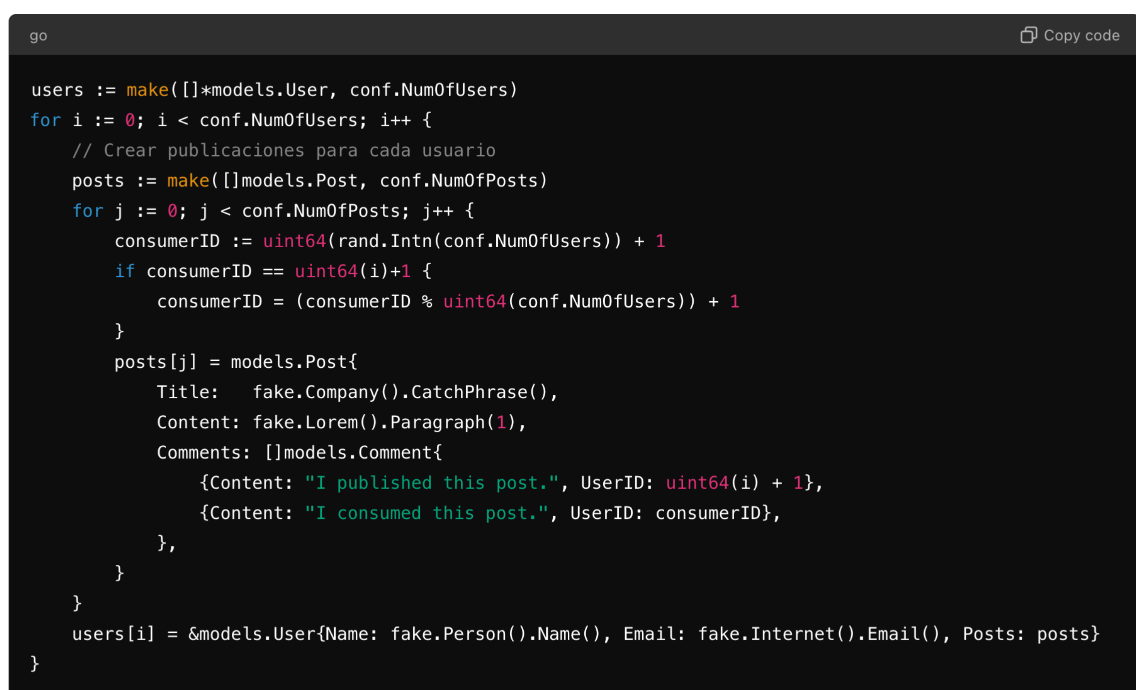
```go
tx := DBClient.Begin()

defer func() {
    if r := recover(); r != nil {
        tx.Rollback()
        log.Println("SEED ERROR", r)
    }
}()
```

Figura 4.6 Código para revertir la transacción si ocurre un error.

Creación de Usuarios y Publicaciones

El siguiente bloque de código, véase figura 4.7, crea una lista de usuarios junto con sus posts y comentarios. Para cada usuario, se generan un conjunto de publicaciones que incluyen comentarios tanto del usuario que publica como de un consumidor aleatorio.

- **Usuarios**: Se genera un nombre y un correo electrónico ficticio para cada usuario.
- **Publicaciones**: Cada usuario tiene un número de publicaciones, que incluyen un título, contenido y dos comentarios (uno del autor y otro de un usuario aleatorio).



```go
users := make([]*models.User, conf.NumOfUsers)
for i := 0; i < conf.NumOfUsers; i++ {
    // Crear publicaciones para cada usuario
    posts := make([]models.Post, conf.NumOfPosts)
    for j := 0; j < conf.NumOfPosts; j++ {
        consumerID := uint64(rand.Intn(conf.NumOfUsers)) + 1
        if consumerID == uint64(i)+1 {
            consumerID = (consumerID % uint64(conf.NumOfUsers)) + 1
        }
        posts[j] = models.Post{
            Title:   fake.Company().CatchPhrase(),
            Content: fake.Lorem().Paragraph(1),
            Comments: []models.Comment{
                {Content: "I published this post.", UserID: uint64(i) + 1},
                {Content: "I consumed this post.", UserID: consumerID},
            },
        }
    }
    users[i] = &models.User{Name: fake.Person().Name(), Email: fake.Internet().Email(), Posts: posts}
}
```

Figura 4.7 Código para la generación de los usuarios y sus publicaciones con comentarios.

Esta parte es clave porque simula interacciones sociales realistas (como comentarios de diferentes usuarios), lo que es útil para probar las APIs en un entorno más auténtico.

Inserción de los Usuarios en la Base de Datos

Una vez que los usuarios y sus publicaciones han sido creados, se insertan en la base de datos dentro de la transacción. Si ocurre algún error durante la creación, la transacción se revierte. Se muestra el código en la figura 4.8.

```go
if result := tx.Create(users); result.Error != nil {
    tx.Rollback()
    log.Fatal("SEED ERROR", result.Error)
}
```

Figura 4.8 Código para insertar los usuarios con sus publicaciones y revertir la transacción en caso de error.

Creación de Chats y Mensajes

Después de crear los usuarios, el siguiente paso (figura 4.9) es generar chats grupales con un conjunto de participantes seleccionados aleatoriamente. Luego, se generan mensajes dentro de estos chats:

- **Participantes de los Chats**: Se selecciona un número aleatorio entre 1 y el número máximo de participantes para cada chat, y estos participantes son seleccionados de entre los usuarios disponibles.
- **Mensajes**: Para cada chat, se generan mensajes aleatorios enviados por los participantes seleccionados.

```go
chats := make([]*models.Chat, conf.NumOfChats)
for i := 0; i < conf.NumOfChats; i++ {
    unusedUsers := make([]uint64, conf.NumOfUsers)
    for j := 0; j < conf.NumOfUsers; j++ {
        unusedUsers[j] = uint64(j) + 1
    }
    numOfParticipants := rand.Intn(conf.MaxNumOfParticipants) + 1
    participants := make([]*models.User, numOfParticipants)
    for j := 0; j < numOfParticipants; j++ {
        unusedUserIndex := rand.Intn(len(unusedUsers))
        participantID := unusedUsers[unusedUserIndex]
        participants[j] = &models.User{ID: participantID}
        unusedUsers = slices.Delete(unusedUsers, unusedUserIndex, unusedUserIndex)
    }
    numOfMessages := rand.Intn(conf.MaxNumOfMessages) + 1
    messages := make([]models.Message, numOfMessages)
    for j := 0; j < numOfMessages; j++ {
        usrIndex := uint64(rand.Intn(len(participants)))
        messages[j] = models.Message{
            Content: fake.Lorem().Sentence(rand.Intn(conf.MaxNumOfWords) + 1),
            UserID:  participants[usrIndex].ID,
        }
    }
    chats[i] = &models.Chat{Messages: messages, Participants: participants}
}
```

Figura 4.9 Código para la generación de los chats con sus participantes y mensajes.

Inserción de los Chats en la Base de Datos

Finalmente, al igual que con los usuarios, los chats y los mensajes generados se insertan en la base de datos dentro de la misma transacción. Si hay un error, se revierte la transacción. El código se encuentra en la figura 4.10.

Figura 4.10 Código para insertar los chats con sus participantes y mensajes y revertir la transacción en caso de error.

Resumen

Esta función de *seeding* crea un conjunto complejo de relaciones entre los usuarios, publicaciones, comentarios, chats y mensajes. Se utiliza una transacción para garantizar la consistencia y se manejan posibles errores con un mecanismo de *rollback*, lo que asegura que la base de datos no se quede en un estado incoherente en caso de fallos. Este proceso de *seeding* es esencial para poder realizar pruebas de rendimiento y funcionalidad en un entorno que simule condiciones realistas.

4.2.4 Modelos, DTOs y Conversión

Bajo la carpeta */lib/models* se ubican los archivos dedicados a definir las estructuras de datos que representan las entidades de la base de datos, la vista que se muestra al usuario y los métodos de conversión entre estas. Respectivamente, el código correspondiente a cada uno de estos ámbitos se encuentra en los archivos *models.go, dtos.go* y *conv.go.*

Modelos

Para definir los modelos que a su vez definen las tablas de la base de datos simplemente se tienen que crear un *struct* para cada tabla, siendo cada campo de este las columnas de la tabla en SQL. Gracias al uso de GORM, los tipos de cada campo de los *struct* es traducido a un tipo equivalente en la base de datos usada. Y, el caso de que la configuración por defecto no fuera suficiente, se pueden añadir etiquetas *gorm* a cada campo para indicar cómo debe de interpretar y convertir GORM ese campo a la columna de la tabla; por ejemplo: *gorm:"primaryKey"* para indicar que ese campo es la clave principal de la tabla. En la figura 4.11 se puede ver un ejemplo de cómo se define el Modelo para la entidad Usuario.

```go
package models

import (
    "time"

    "gorm.io/gorm"
)

type User struct {
    ID        uint64         `gorm:"primaryKey" json:"id"`
    Name      string         `json:"name"`
    Email     string         `gorm:"unique;not null;index" json:"email"`
    Posts     []Post         `json:"posts"`
    Messages  []Message      `json:"messages"`
    Comments  []Comment      `json:"comments"`
    Chats     []*Chat        `gorm:"many2many:participants;" json:"chats"`
    CreatedAt time.Time      `json:"createdAt"`
    UpdatedAt time.Time      `json:"updatedAt"`
    DeletedAt gorm.DeletedAt `json:"deletedAt"`
}
```

Figura 4.11 Código para definir el Modelo de la entidad Usuario y su tabla en PostgreSQL.

DTOs

Los DTOs, al ser una aplicación de servidor API, son la Vista del sistema. Son la forma en la que el usuario ve e interactúa con los Modelos subyacentes.

Se definen igual que los Modelos, *structs* con campos. Pero en este caso solo es necesario indicar con etiquetas *json* el nombre que se quiere que tengan los campos en el JSON resultante. Esto es solo necesario para las APIs REST y GraphQL que están basadas en JSON, ya que gRPC usa los Protocol Buffers. Pero las anotaciones también se mantienen en la aplicación gRPC ya que simplemente son ignoradas.

Ya que el usuario no interactúa igual con las entidades para consumirlas que para producirlas, se define un DTO para cada uno de estos casos en cada entidad. Es decir, que para devolver un usuario se usa el DTO *ReadUserDTO* mientras que para crearlo se usa *WriteUserDTO*, véase la figura 4.12.

```go
go                                                           Copy code

package models

type ReadUserDTO struct {
    ID        uint64 `json:"id"`
    Name      string `json:"name"`
    Email     string `json:"email"`
    CreatedAt string `json:"createdAt"`
    UpdatedAt string `json:"updatedAt"`
}

type WriteUserDTO struct {
    Name  string `json:"name"`
    Email string `json:"email"`
}
```

Figura 4.12 Código para definir el los DTO de entrada y salida de la entidad Usuario.

Conversión

Como es necesario convertir los Modelos a Vistas y las Vistas a Modelos, se ha añadido a cada *struct* una función para convertirlo en el *struct* correspondiente. Esta función simplemente crea un objeto del tipo deseado mapeando los campos del objeto existente al nuevo.

Solo se definen funciones de conversión de Modelo a DTO de lectura y de DTO de escritura a Modelo, ya que no existe ningún caso en el que el Modelo se tenga que convertir a un DTO de escritura (el Modelo ya está creado o modificado), o en el que se necesite transformar un DTO de Lectura en un Modelo (el DTO de Lectura no añadiría o modificaría información al Modelo). Se muestra un ejemplo de funciones de conversión en la figura 4.13.

```go
go                                                           Copy code

package models

func (u *WriteUserDTO) ToUser() User {
    return User{
        Name:  u.Name,
        Email: u.Email,
    }
}

func (u *User) ToReadUserDTO() ReadUserDTO {
    return ReadUserDTO{
        ID:        u.ID,
        Name:      u.Name,
        Email:     u.Email,
        CreatedAt: u.CreatedAt.String(),
        UpdatedAt: u.UpdatedAt.String(),
    }
}
```

Figura 4.13 Código para convertir los DTO y el Modelo del Usuario entre sí.

4.3 APIs

En todas las aplicaciones de cada tecnología API existen elementos en común y otros que los diferencian. En este apartado se compara cada aplicación a nivel de estructura, implementación y experiencia de desarrollo.

4.3.1 REST

Estructura

Todas las aplicaciones comparten una estructura muy similar que parte de adaptar la arquitectura MVC a cada tecnología y sus herramientas. Todas tienen su punto de entrada en el archivo */cmd/main.go*, todas tienen unos archivos encargados de llevar la petición al lugar oportuno para que sea procesada y todos tienen una sección para las funciones que procesan las peticiones. Además, todos tienen sus propias pruebas específicas para asegurar el correcto funcionamiento de la aplicación.

En el caso de REST, los archivos encargados de dirigir la petición al lugar de procesamiento indicado son los enrutadores. Se encuentran en la carpeta */routers*, la cual tiene un archivo principal *routes.go* que deriva la petición a un archivo específico para el recurso al que se quiere acceder. Y es cada uno de estos ficheros específicos que deciden que función se ejecuta para procesar la petición. Estas funciones son los controladores. Se encuentran bajo la carpeta */controllers* y existe uno para cada recurso.

También, como todas las aplicaciones, tiene sus archivos *go.mod* y *go.sum* para la gestión de dependencias y un archivo de configuración de *Air*. Se explica en el apartado 4.3.7 más en detalle la naturaleza y funcionamiento de esta herramienta.

Implementación

cmd/main.go

La implementación del *cmd/main.go* es prácticamente idéntica en todas las APIs. La única diferencia es la forma de iniciar el servidor que va a resolver las peticiones del usuario. El propósito de este archivo es inicializar la aplicación, manejar pruebas y opciones específicas que se pasan como argumentos de línea de comandos, y levantar el servidor API correspondiente.

Primero, se manda mostrar por pantalla el uso de memoria mediante el uso de la función auxiliar *PrintMemUsage()* (figura 4.15) antes de finalizar la ejecución del programa. Se crea un canal por el cual se notifica al programa de señales del sistema de SIGTERM o SIGINT para cerrar el servidor de manera segura. Se cargan las variables de entorno definidas en el *.env*. Y se configuran las opciones de línea de comandos que se pueden usar para realizar diferentes operaciones con la aplicación, siendo estas *tests* para ejecutar pruebas específicas, *reset* para reiniciar y poblar la base de datos y *server* para iniciar el servidor de la API. El código se encuentra en la figura 4.14. Estas opciones permiten ejecutar comandos flexibles como en la figura 4.16.

```go
defer PrintMemUsage()

sigs := make(chan os.Signal, 1)
signal.Notify(sigs, os.Interrupt, syscall.SIGTERM)

// Initialize the environment variables
err := godotenv.Load()
if err != nil {
    log.Fatal("Error loading .env file")
}

// Initialize command line arguments and flags
testsPtr := flag.String("tests", "", "Tests to run")
resetDB := flag.Bool("reset", false, "Reset the database")
server := flag.Bool("server", false, "Run the server")
flag.Parse()
```

Figura 4.14 Primera sección del *cmd/main.go*.

```go
func PrintMemUsage() {
    var m runtime.MemStats
    runtime.ReadMemStats(&m)
    fmt.Printf("Alloc = %v MiB", m.Alloc/1024/1024)
    fmt.Printf("\tTotalAlloc = %v MiB", m.TotalAlloc/1024/1024)
    fmt.Printf("\tSys = %v MiB", m.Sys/1024/1024)
    fmt.Printf("\tNumGC = %v\n", m.NumGC)
}
```

Figura 4.15 Función auxiliar para mostrar el uso de memoria del programa.

```bash
$ go run src/cmd/main.go -tests=users -server
```

Figura 4.16 Ejemplo de ejecución del programa con opciones de línea de comandos.

Después se conecta a las bases de datos, tanto PostgreSQL como Redis, haciendo uso de las funciones definidas para ello en la librería compartida */lib*.

Además, se asegura que se cierren las conexiones al terminar la ejecución del programa con el uso de *defer* como se muestra en la figura 4.17.

```go
db.Connect()
defer db.Disconnect()
db.RedisConnect()
defer db.RedisDisconnect()
```

Figura 4.17 Código de *cmd/main.go* para conectarse a las bases de datos y cerrar la conexión al final de ejecución.

Luego, en la figura 4.18, se pasa a resetear y repoblar la base de datos si se ha pasado la opción *-reset*, a ejecutar las pruebas que se la hayan pasado a la opción *-tests* (si se le ha pasado alguna) y ejecutar el servidor de la API.

```go
// Reset and seed the database
if *resetDB {
    fmt.Println("\nResetting the database...")
    db.Reset()
    fmt.Println("\nSeeding database...")
    db.Seed()
}

// Run the tests
if *testsPtr != "" {
    if err := validateFlags(testsPtr); err == nil {
        tests.Run(testsPtr)
    } else {
        fmt.Println("\nERROR:", err)
    }
} else {
    fmt.Println("\nNo tests to run")
}
```

Figura 4.18 Código de *cmd/main.go* para resetear la base de datos y ejecutar las pruebas seleccionadas.

La ejecución del servidor de la API es la única parte del archivo que difiere en implementación para cada tecnología. En el caso de REST, se inicia un servidor HTTP utilizando el framework Fiber. Se configura un middleware de *logging* para registrar las peticiones entrantes, y luego se definen las rutas mediante el paquete *routers*. Véase figura 4.19.

```go
if *server {
    app := fiber.New()
    app.Use(logger.New(logger.Config{
        Format: "[${time}] ${ip} ${status} - ${latency} ${method} ${path} ? ${queryParams} : ${body} ! ${error}\n",
    }))
    routers.Setup(app)
    go func() {
        app.Listen(":8080")
    }()
    _ = <-sigs
    app.Shutdown()
}
```

Figura 4.19 Código de *cmd/main.go* para configurar e iniciar el servidor HTTP con Fiber.

Después de iniciar el servidor HTTP en una función asíncrona usando la instrucción *go*, se espera a que el canal de notificación de señales del sistema reciba alguna señal. Después de recibirla, ejecuta la función para apagar el servidor de la manera adecuada. Este comportamiento es común a todas las implementaciones de los servidores.

/routers

Para configurar el servidor HTTP con Fiber se ejecuta la función *routers.Setup(app)* (figura 4.20), pasando la aplicación del servidor HTTP Fiber como argumento. Esta función proviene del paquete *routers* ubicado en la carpeta homónima.

Más concretamente, la función está definida en el archivo *routers/routes.go*. Aquí la función recibe la aplicación Fiber en la que registra todas las rutas de todos los recursos agrupadas bajo la dirección de URL */rest*. De esta forma todos los identificadores de los recursos están prefijados con esta dirección.

```go
func Setup(app *fiber.App) {
    api := app.Group("/rest")

    UsersRouter(&api)
    PostsRouter(&api)
    CommentsRouter(&api)
    MessagesRouter(&api)
    ChatsRouter(&api)
    HealthcheckRouter(&api)
    RedisRouter(&api)
}
```

Figura 4.20 Código de *routers/routes.go* para configurar las rutas de todos los recursos de la API REST bajo el prefijo */rest*.

Para definir todas las rutas de las posibles operaciones sobre cada recurso se delega esta responsabilidad a otra función en un archivo con el nombre del recurso que mapea. Por ejemplo, véase como se define la función *UsersRouter(api *fiber.Router)* del archivo *routers/users.go* en la figura 4.21.

```go
func UsersRouter(api *fiber.Router) {
    u := (*api).Group("/users")
    u.Get("/", controllers.UserList)
    u.Get("/:id", controllers.UserRead) // Can use email as id
    u.Post("/", controllers.UserCreate)
    u.Delete("/:id", controllers.UserDelete)
    u.Put("/:id", controllers.UserUpdate)
    u.Patch("/:id", controllers.UserPatch)
    u.Get("/:id/posts", controllers.UserPosts)
    u.Get("/:id/comments", controllers.UserComments)
    u.Get("/:id/messages", controllers.UserMessages)
    u.Get("/:userId/chat/:chatId/messages", controllers.UserChatMessages)
}
```

Figura 4.21 Código de *routers/users.go* para configurar todas las rutas sobre el recurso usuario y asociarlas a un controlador específico.

De forma similar al *routers/routes.go*, se agrupan todos los endpoints del recurso usuario bajo el prefijo */users* que lo identifica como recurso. Luego se especifica cada una de las operaciones posibles usando los métodos provistos por Fiber; especificando el método HTTP de la petición, el *path* de la URL y la función del controlador que la procesa. Se permite pasar parámetros en el *path* precediéndolo con dos puntos (":").

Las funciones que procesan la petición son parte del paquete *controllers* bajo la carpeta de mismo nombre. Es aquí donde se específica el comportamiento de cada endpoint, donde se realizan todas las consultas y operaciones necesarias, y donde se monta la respuesta usando la vista adecuada para enviársela al usuario.

Aquí también se dividen las funciones por recurso, cada uno en un archivo diferente para mantener orden y coherencia. En cada uno de estos crean funciones que toman un objeto *fiber.Ctx* (contexto de Fiber), el cual contiene información sobre la petición como parámetros del *path,* de consulta o el cuerpo de la petición HTTP entre otros; y devuelve un posible error en caso de que ocurra para que Fiber pueda manejarlo de la manera adecuada.

Para enviar la respuesta al usuario también se usa el objeto *fiber.Ctx*. Existen muchas funciones para devolver diferentes datos e información al usuario. Las más comunes son *Status(status int)* y *JSON(data interface{})*, que asigna el código de estatus HTTP y rellena el cuerpo de la respuesta con un objeto en

formato JSON. Al finalizar la ejecución de la función Fiber envía la respuesta al usuario. Para simplificar el ejemplo en la figura 4.22 se muestra solo el controlador para obtener los datos de un usuario específico:

```go
func UserRead(c *fiber.Ctx) error {
    id, err := strconv.ParseUint(c.Params("id"), 10, 64)

    if err != nil {
        return c.Status(400).JSON(fiber.Map{
            "error": "id is required",
        })
    }
    if id < 1 {
        return c.Status(400).JSON(fiber.Map{
            "error": "id must be greater than 0",
        })
    }

    u, err := db.UserRead(id)

    if err == nil {
        return c.JSON(u)
    }

    if errors.Is(err, gorm.ErrRecordNotFound) {
        return c.Status(404).JSON(fiber.Map{
            "error": "user not found",
        })
    }

    return c.Status(500).JSON(fiber.Map{
        "error": err.Error(),
    })
}
```

Figura 4.22 Código de *controllers/users.go* para obtener los datos de un usuario específico.

Como se puede observar, se utiliza el objeto *c* de tipo *fiber.Ctx* pasado por referencia (de forma que es compartido por toda la aplicación Fiber) para obtener el ID especificado en la dirección URL y para devolver respuestas con un status HTTP y cuerpo de mensaje JSON específicos.

La lógica general de este controlador es:

1. Se obtiene el ID del usuario deseado y se intenta convertir de cadena de caracteres a un número entero (*uint* de *unsigned integer*).
2. Se comprueba que la conversión haya sido exitosa. Si no lo ha sido, se devuelve una respuesta con status 400 y un JSON en el cuerpo indicando la razón del error: “se requiere ID.”

3. Se asegura que el numero sea mayor que cero, ya que no se permite como valor de ID de usuario. Si no lo es se devuelve otro error 400: "el ID debe de ser mayor que 0."
4. Se intenta leer el usuario desde la base de datos usando el ID como patrón de búsqueda.
5. Si no hay ningún error, se devuelven los datos recuperados de la base de datos en formato JSON.
6. En caso contrario, se comprueba si el error es del tipo *gorm.ErrRecordNotFound* (error registro no encontrado) que proporciona el ORM GORM para realizar este tipo de comprobaciones. Y de así serlo, se envía al usuario un error 404 con el mensaje "usuario no encontrado."
7. Si no fuese ese el error, se devuelve un error 500 con el mensaje del error recibido de la base de datos.

Todos los controladores realizan operaciones similares a las mostradas en este ejemplo adaptadas a su recurso y la operación en específico.

Experiencia de Desarrollo

Como se ha podido ver, la implementación es bastante sencilla e intuitiva. Para la mayoría de los desarrolladores, que suelen tener más experiencia con APIs REST, este servidor es extremadamente sencillo de implementar.

Además, la herramienta principal, Fiber, se inspira fuertemente un probablemente la librería de creación de APIs REST más popular del mundo actualmente, Express.js; con más de 29 millones de descargas semanales y cerca de 65 mil estrellas en GitHub. Esto hace que sea increíblemente accesible para un enorme número de desarrolladores, incluso aunque no conozcan Go en profundidad.

Sí es cierto que requiere que el desarrollador escriba absolutamente todo el código necesario para hacer funcionar la API. No como gRPC o GraphQL (en el caso de gqlgen) que generan gran parte del código de forma automática. Pero es tan sencillo y su interfaz es tan clara que requiere de muy poco código para hacer que funcione.

Esto da a su vez más flexibilidad al desarrollador para que estructure su código de muchas más maneras sin tener que preocuparse en cómo adaptar el código autogenerado a esta. Que junto a la gran cantidad de *plugins* que Fiber ofrece, hacen del desarrollo de APIs REST una experiencia muy gratificante.

En cuanto a horas de desarrollo, se estima que se han requerido de 10 horas para completar la implementación de las partes diferenciadoras de REST. El tiempo requerido en código compartido entre las implementaciones como la mayoría de *cmd/main.go* no se tiene en cuenta ya que no ayuda a comparar las tecnologías API entre sí.

4.3.2 GraphQL

Estructura

La estructura del proyecto GraphQL es prácticamente idéntica a la de REST con una diferencia: en vez de tener enrutadores y controladores tiene una carpeta */graph* donde se encuentra todo el código relacionado a enrutar y controlar la petición además de otras cosas.

Esta carpeta se compone de archivos autogenerados y la definición del esquema GraphQL definida por el usuario que usa para generar el código. También se puede configurar para que almacene los archivos generados en diferentes lugares, pero este es el funcionamiento por defecto.

Genera una carpeta */model* con un archivo *models_gen.go* en la que define los modelos específicos para GraphQL. A gqlgen se le indica dónde están las definiciones de los modelos y si encuentra algunos que encajen con los definidos en el esquema GraphQL, los usa en vez de crear nuevos. En este caso ha creado los tipos de entrada de las mutaciones y se han modificado algunas para que hagan uso de los DTO de la librería compartida.

También se crean los siguientes archivos:

- *generated.go*: Este archivo contiene el entorno de ejecución de GraphQL implementado en Go. Es el equivalente a los enrutadores, ya que se encarga de dirigir las peticiones a los resolutores indicados para satisfacer la petición. Pero, además, monta la vista que envía de respuesta al usuario.
- *resolver.go*: Es el tipo raíz de todos los resolutores. Sirve como gestor centralizado de inyección de dependencias y del estado de la aplicación.
- *schema.resolver.go*: Aquí es donde se crean los resolutores para procesar las peticiones y generar los datos deseados para cada campo de la petición.

Por lo demás es idéntico en estructura a al proyecto REST.

Implementación

Para la implementación del servidor GraphQL a priori solo se necesitaría definir el esquema, configurar bien el generador de código e implementar algunos resolutores. Por desgracia no ha sido tan fácil.

Se puede ver el esquema GraphQL completo en el código fuente del repositorio en GitHub. A continuación, en la figura 4.23, se muestra un extracto de las secciones relacionadas con la creación, lectura y actualización de usuarios:

```graphql
type User {
  id: ID!
  name: String!
  email: String!
  posts: [Post!]!
  messages: [Message!]!
  comments: [Comment!]!
  chats: [Chat!]!
  createdAt: String!
  updatedAt: String
  deletedAt: String
}

type Query {
  users: [User!]!
  user(id: ID!): User!
}

input NewUser {
  name: String!
  email: String!
}

input UpdateUser {
  name: String
  email: String
}

type Deletion {
  id: ID!
  msg: String!
}

type Mutation {
  createUser(input: NewUser!): User!
  updateUser(id: ID!, input: UpdateUser!): User!
  deleteUser(id: ID!): Deletion!
}
```

Figura 4.23 Extracto del esquema de la aplicación API GraphQL.

Una vez configurado y generado el código, se tuvieron que modificar ciertos modelos para intentar aprovechar la librería compartida y que todos los proyectos tuvieran las mismas vistas. En concreto los tipos de entrada para la creación de

entidades se separaron en un archivo *news.go* y se modificaron para que usaran directamente los DTOs del modelo como se muestra en la figura 4.24.

```go
package model

import "github.com/n4xo-dev/api-wars/lib/models"

type NewUser struct {
    models.WriteUserDTO
}

type NewPost struct {
    models.WritePostDTO
}

type NewComment struct {
    models.WriteCommentDTO
}

type NewMessage struct {
    models.WriteMessageDTO
}

type NewChat struct {
    Participants []uint64
}

type NewRedisRecord struct {
    Key   string `json:"key"`
    Value string `json:"value"`
}
```

Figura 4.24 Código de *graph/model/news.go* para usar los DTOs en los tipos de entrada GraphQL.

En cuanto al *resolver.go*, como solo se usa la librería compartida para gestionar las llamadas con la base de datos y no se requiere mantener estado en la aplicación, este archivo se queda esencialmente vacío. Pero a la hora de implementar los resolutores es cierto que ha habido muchas complicaciones.

Por defecto, gqlgen intenta asociar el resolutor del campo con algún campo de los modelos que tiene a su disposición para implementar el resolutor. En la mayoría de los campos escalares suele tener éxito. Pero, cuando no lo tiene deja en las manos del desarrollador implementar ese resolutor en concreto. Aunque implementar los resolutores no suele ser demasiado complicado. Por ejemplo, véase el resolutor para obtener un usuario en particular en la figura 4.25.

```go
// User is the resolver for the user field.
func (r *queryResolver) User(ctx context.Context, id string) (*models.User, error) {
	uID, err := strconv.ParseUint(id, 10, 64)
	if err != nil {
		return nil, err
	}

	user, err := db.FullUserRead(uID)
	if err != nil {
		return nil, err
	}

	return &user, nil
}
```

Figura 4.25 Código del resolutor de usuario de *graph/schema.resolvers.go*.

Simplemente se obtiene el ID como parámetro de la consulta, se intenta convertir a un número, se usa para buscar ese usuario en la base de datos haciendo uso de los métodos de la librería compartida y si se encuentra se devuelve.

Experiencia de Desarrollo

En este caso la experiencia de desarrollo ha sido buena para un servicio tan complejo como un servidor GraphQL. Pero con ciertas complicaciones engorrosas.

Hubo muchos resolutores que no pudieron ser auto-implementados cuando podrían haberlo sido. Pero el mayor problema vino de intentar usar los métodos compartidos para consultar listas de entidades en la base de datos. Ya que los resolutores esperan una lista de referencias a los objetos, no una lista de objetos en sí, que es como estaban implementados los métodos para el servidor REST. Por suerte, es un cambio sencillo, pero muy tedioso de hacer en todas las entidades.

Una vez resueltos estos problemas es cierto que funciona perfectamente. Para la complejidad que tiene desarrollar un servidor GraphQL desde cero, probablemente la librería gqlgen es la que más trabajo ahorra al desarrollador. Y si se es capaz de configurar perfectamente la generación de código, muchos de estos problemas no se habrían ocasionado.

En general, la dificultad de implementar el servidor no ha sido tan alta como se esperaba tras el análisis teórico realizado con anterioridad. Pero de todas formas ha sido bastante más largo y complejo que REST. Se estima que el tiempo de desarrollo se aproxima a las 17 horas.

4.3.3 gRPC

Estructura

Al igual que con GraphQL, la aplicación gRPC comparte prácticamente toda su estructura con la aplicación REST. Pero, en este caso, reemplaza los controladores por los servicios y las rutas por los "archivos pb." Además, se añaden las definiciones de los *Protocol Buffers* de mensajes y servicios en los archivos *.proto* bajo la carpeta */protos.*

Los servicios se encargan de recibir, procesar y resolver las peticiones de los usuarios. Los "archivos pb" son archivos autogenerados por el compilador de *Protocol Buffers*, existen 2 tipos:

- Archivos *.pb.go*: sirven para serializar, popular y recibir tipos de mensaje de entrada y salida.
- Archivos *_grpc.pb.go*: contienen interfaces para que los clientes realicen peticiones con los métodos definidos en los archivos *.proto* y para que los servidores implementen estos métodos.

También se ha incluido un archivo *conv/conv.go* para almacenar las funciones para convertir los modelos y DTOs de la librería compartida en los mensajes de *Protocol Buffers.*

Implementación

Gracias al código autogenerado por el compilador de *Protocol Buffers*, lo único que ha sido necesario implementar la definición de los mensajes y servicios con *Protocol Buffers,* los servicios con sus métodos para resolver las peticiones a la API y unas funciones auxiliares que convierten entre estructuras de la librería compartida y los mensajes de *Protocol Buffers.*

Las funciones de conversión simplemente transforman cada tipo de la librería compartida en un su tipo *Protocol Buffer* correspondiente. Estas funciones generalmente son un mapeo uno a uno entre campos idénticos de diferentes estructuras. Véase la figura 4.26.

```graphql
func UserDTOToPb(u models.ReadUserDTO) *pb.UserDTO {
    return &pb.UserDTO{
        Id:        u.ID,
        Name:      u.Name,
        Email:     u.Email,
        CreatedAt: u.CreatedAt,
        UpdatedAt: u.UpdatedAt,
    }
}
```

Figura 4.26 Código para la conversión de un *ReadUserDTO* al tipo equivalente en *Protocol Buffers*.

Aunque en algunos casos excepcionales, como cuando se convierten arreglos de estructuras de datos, hace falta hacer conversiones algo más complejas. A continuación, como ejemplo, la conversión del modelo *Chat* al *Protocol Buffer* respectivo en la figura 4.27.

```go
func ChatToPb(c models.Chat) *pb.Chat {
    pbMessages := make([]*pb.MessageDTO, len(c.Messages))
    for i, m := range c.Messages {
        pbMessages[i] = MessageToPb(m)
    }

    pbParticipants := make([]*pb.UserDTO, len(c.Participants))
    for i, u := range c.Participants {
        pbParticipants[i] = UserToPb(*u)
    }

    return &pb.Chat{
        Id:           c.ID,
        Messages:     pbMessages,
        Participants: pbParticipants,
        CreatedAt:    c.CreatedAt.String(),
        UpdatedAt:    c.UpdatedAt.String(),
        DeletedAt:    c.DeletedAt.Time.String(),
    }
}
```

Figura 4.27 Código para la conversión de un *Chat* al tipo equivalente en *Protocol Buffers*.

Se puede ver como se debe de iterar sobre cada mensaje y participante del chat, y a su vez convertir estos a su tipo de *Protocol Buffers*, antes de poder crear el objeto *pb.Chat* equivalente.

En cuanto a la implementación de los servicios, viene a ser lo mismo que en el caso de GraphQL, pero sin ningún problema con los modelos gracias a las funciones de conversión. Por ejemplo, véase en la figura 4.28 cómo se obtiene un usuario en específico.

```go
func (u *UserServiceServer) GetUser(ctx context.Context, getReq *pb.GetUserRequest) (*pb.GetUserResponse, error) {
    if getReq.Id < 1 {
        return nil, status.Errorf(codes.InvalidArgument, "invalid id")
    }

    user, err := db.UserRead(getReq.Id)
    if err == nil {
        return &pb.GetUserResponse{User: conv.UserDTOToPb(user)}, nil
    }
    if errors.Is(err, gorm.ErrRecordNotFound) {
        return nil, status.Errorf(codes.NotFound, "user not found")
    }

    return nil, status.Errorf(codes.Internal, "error getting user: %v", err)
}
```

Figura 4.28 Código implementando el método *GetUser* del servicio *UserService*.

Como se observa, simplemente se obtiene un objeto *getReq* el cual contiene toda la información pasada por el cliente en la petición (en este caso solo el ID del usuario). De ahí se obtiene el ID del usuario a buscar, se realizan las comprobaciones oportunas, se busca en usuario en la base de datos, si no hay errores se devuelve el usuario convertido a su mensaje de *Protocol Buffers* y en caso contrario se devuelve un error específico de entre los ya definidos por gRPC utilizando su librería *codes* y *status*.

Cabe destacar que el método del servidor está asociado a un tipo *UserServiceServer*, el cual incluye un tipo definido por el compilador *protoc* para que el desarrollador implemente los métodos necesarios a modo de interfaz. Además de que incluye otros métodos necesarios para manejar las peticiones adecuadamente.

En cuanto a la especificación de los archivos *.proto* cabe destacar que siguen un patrón normalmente inusual en otros entornos, que a priori parece erróneo, pero tiene una razón de ser importante. Este patrón es muy sencillo, simplemente cada método de todo servicio ha de tener su propio tipo de mensaje de entrada y salida particular incluso aunque sea idéntico al de otros métodos.

Esto suele ir en contra de uno de los principios de la programación eficiente más básico conocido por sus siglas en inglés *DRY* (no te repitas). Ya que hay muchos tipos de mensajes que son idénticos y podrían todos ser representados por un único tipo común.

Pero esto causaría a futuro problemas de compatibilidad. Ya que, en el caso de que se quiera modificar el tipo de alguno de los métodos que lo comparten, esto requeriría crear un nuevo tipo diferente y todos los clientes que no hubiesen actualizado sus *stubs* dejarían de poder usar ese método. Esto ocurre porque,

aunque el tipo sea idéntico en forma a excepción de un campo (aunque sea este opcional), se registra como un tipo diferente. Es decir, los *Protocol Buffers* solo tienen en cuenta el identificador del tipo para determinar si son el mismo tipo o no.

Si se usa un tipo diferente para cada método, en el futuro solo habrá que añadir los nuevos campos a este tipo (no se recomienda eliminar campos para mantener retrocompatibilidad). Y en el caso de los clientes que no tengan su *stub* actualizado, simplemente enviaran o recibir un mensaje con esos nuevos campos vacíos o con su valor por defecto. De forma que, aunque pierdan información, pueden seguir usando ese servicio parcialmente.

Un ejemplo de cómo se ve un archivo *.proto* siguiendo este patrón se puede ver en un extracto del archivo *users.proto* en el que solo se incluyen operaciones de CRUD del usuario en la figura 4.30.

También se ha añadido un minúsculo script de terminal *grpc.sh* en el que se recoge el comando exacto que hace falta ejecutar para compilar todos los archivos *.proto* y almacenar los resultantes en las carpetas deseadas. Véase el comando en la figura 4.29.

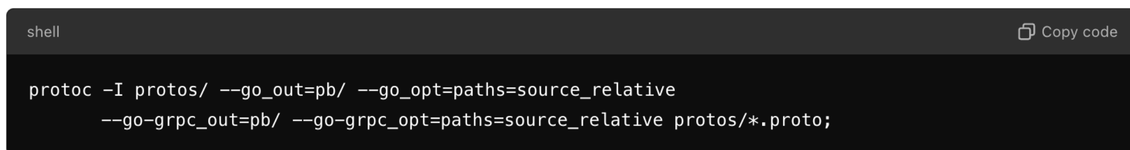


Figura 4.29 Comando de *protoc* para compilar los archivos *.porto* de la forma deseada.

```
syntax = "proto3";

import "posts.proto";
import "comments.proto";
import "messages.proto";

option go_package = "github.com/n4xo-dev/api-wars/grpc/pb";

service UserService {
  rpc ListUsers(ListUsersRequest) returns (ListUsersResponse);
  rpc GetUser(GetUserRequest) returns (GetUserResponse);
  rpc CreateUser(CreateUserRequest) returns (CreateUserResponse);
  rpc DeleteUser(DeleteUserRequest) returns (DeleteUserResponse);
  rpc UpdateUser(UpdateUserRequest) returns (UpdateUserResponse);
}

message UserDTO {
  uint64 id = 1;
  string name = 2;
  string email = 3;
  string created_at = 4;
  string updated_at = 5;
}

message ListUsersRequest {}

message ListUsersResponse {
  repeated UserDTO users = 1;
}

message GetUserRequest {
  uint64 id = 1;
}

message GetUserResponse {
  UserDTO user = 1;
}

message CreateUserRequest {
  string name = 1;
  string email = 2;
}

message CreateUserResponse {
  UserDTO user = 1;
}

message DeleteUserRequest {
  uint64 id = 1;
}

message DeleteUserResponse {
  bool deleted = 1;
}

message UpdateUserRequest {
  uint64 id = 1;
  string name = 2;
  string email = 3;
}

message UpdateUserResponse {
  UserDTO user = 1;
}
```

Figura 4.30 Extracto del archivo *users.proto*.

Experiencia de Desarrollo

Aunque gRPC es más rígido y estricto que GraphQL, la experiencia de desarrollo no se ha visto afectada negativamente por ello. Al contrario, ha sido mejor en líneas generales que con GraphQL.

Si bien es cierto que ha hecho falta crear los métodos auxiliares y la definición de los archivos *.proto* ha sido más engorrosa para seguir los patrones recomendados. En general ha sido muy sencillo desarrollar todo el código necesario para implementar el servidor gRPC gracias a su simple forma. Al ser mucho más sencillo en estructura, no ha requerido tiempo extra en entenderlo. Y tampoco ha causado problemas con el uso de los modelos de la librería compartida al requerir por defecto su conversión a un tipo específico para los métodos gRPC. Aun requiriendo más esfuerzo que REST, ha resultado ser marginalmente más difícil que este. Con unas horas de desarrollo estimadas en 13 horas.

4.4 Pruebas de Extremo a Extremo con Postman

Para la realización de pruebas que simulan el uso de las APIs por usuarios reales (pruebas de extremo a extremo) se ha hecho uso de Postman. Postman es una herramienta para hacer pruebas de APIs (interfaces de programación de aplicaciones) que permite enviar solicitudes HTTP como GET, POST, PUT, DELETE, y ver las respuestas del servidor. Además de soportar REST, también permite hacer consultas GraphQL y pruebas de APIs basadas en gRPC. También ofrece funciones como la gestión de entornos, colecciones de solicitudes, pruebas automatizadas y la integración con pipelines de CI/CD [8].

Para la realización de las pruebas se ha creado una colección para cada API que recoje todas las posibles peticiones que se le puede hacer a cada una y se han ejecutado individualmente con diferentes parámetros y mensajes para asegurar su debido funcionamiento. A continuación se muestra la creación de un usuario con REST (figura 4.31), GraphQL (figura 4.32) y gRPC (figura 4.33) usando Postman.

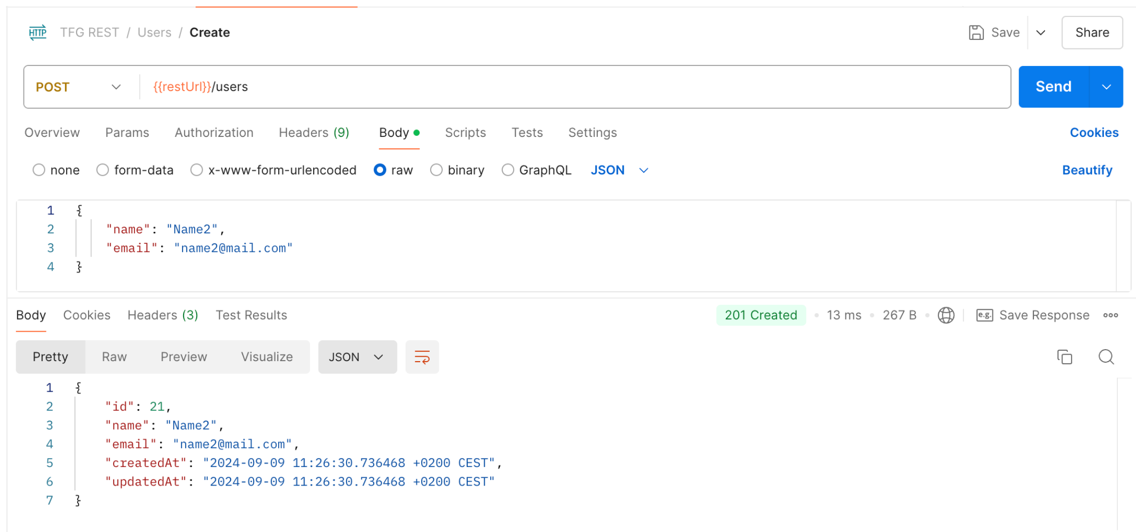


Figura 4.31 Ejemplo de consulta REST para crear un usuario con Postman.

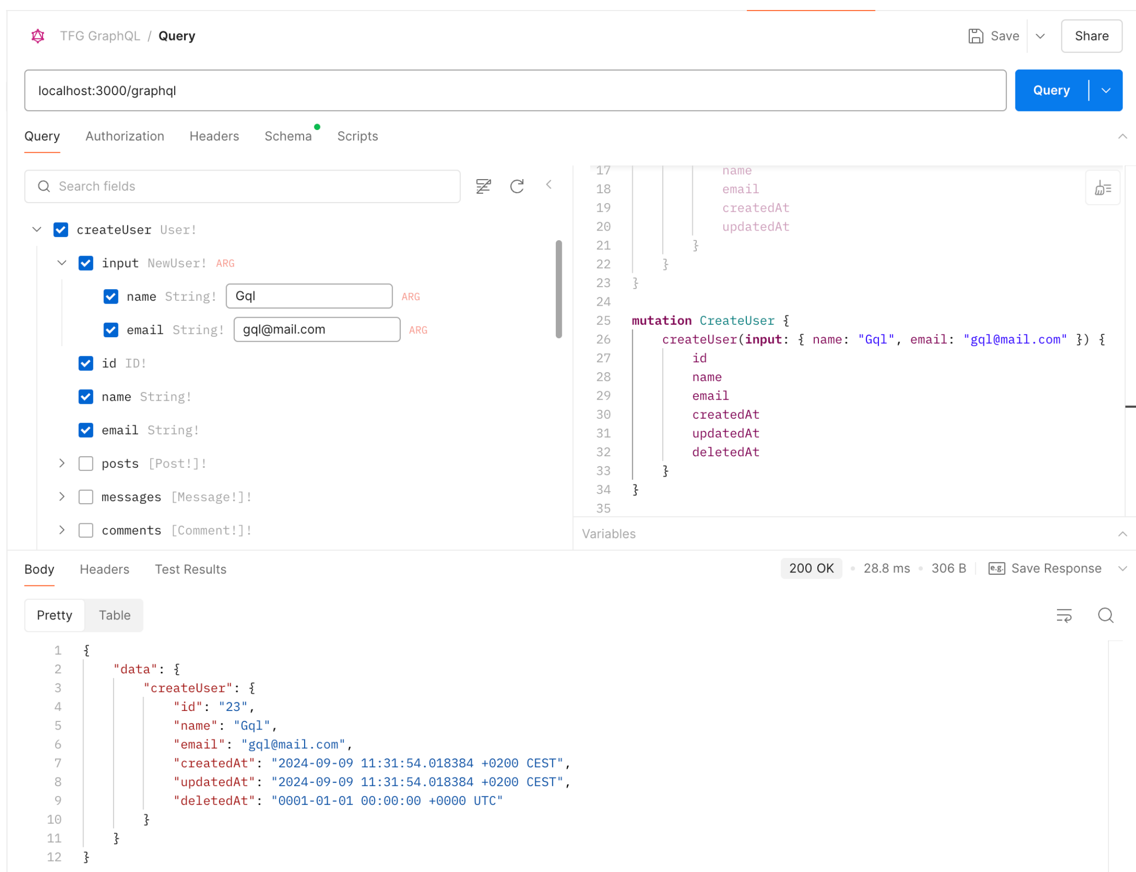


Figura 4.32 Ejemplo de consulta GraphQL para crear un usuario con Postman.

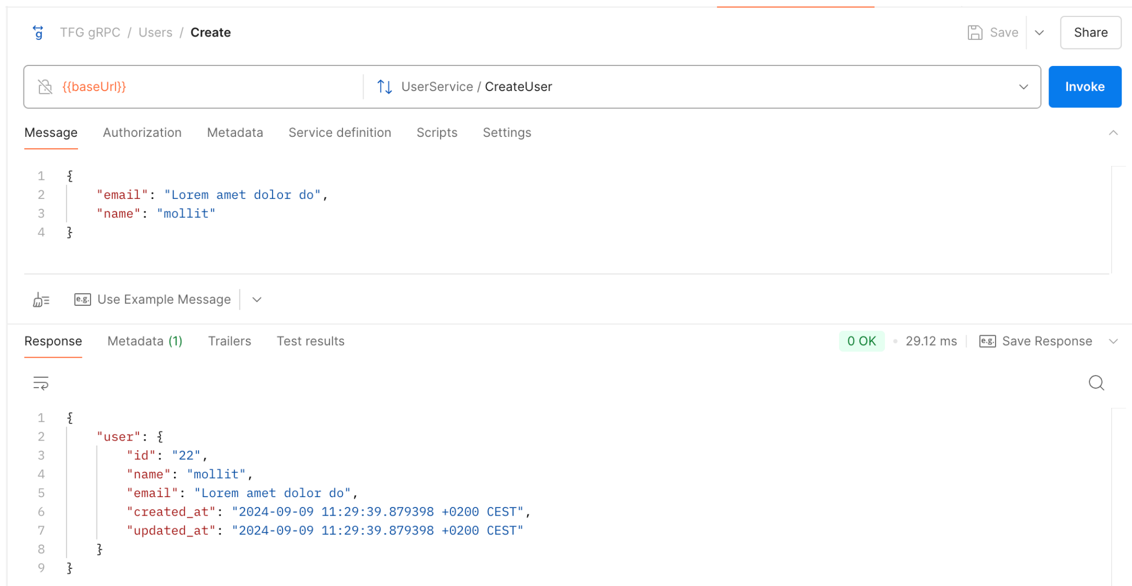


Figura 4.33 Ejemplo de consulta gRPC para crear un usuario con Postman.

Estas colecciones se encuentran públicamente accesibles a través de las siguientes direcciones URL:

- REST: https://www.postman.com/interstellar-trinity-921035/workspace/public/collection/21900620-700765de-a7c5-4651-bff7-57feb81f9583?action=share&creator=21900620
- GraphQL: https://www.postman.com/interstellar-trinity-921035/workspace/public/collection/64c6c928c6af1dc7b1373852?action=share&creator=21900620
- gRPC: https://www.postman.com/interstellar-trinity-921035/workspace/public/collection/64c6c851161d7c9ed2441d13?action=share&creator=21900620

Además, Postman permite realizar pruebas de rendimiento sobre estas colecciones. Aunque son demasiado limitadas para sacar conclusiones en este estudio, sirven para hacerse una idea general del rendimiento de las APIs sin necesidad de crear scripts de pruebas complicados.

A continuación, en la figura 4.34, se muestra los resultados de la ejecución de una prueba de rendimiento sobre la API REST. No es una prueba bien diseñada, por lo que sus resultados no son concluyentes ni representativos, sirve simplement para mostrar la utilidad de la herramienta. También se adjuntan los resultados detallados en el pdf *TFG-REST-performance-report.pdf*.

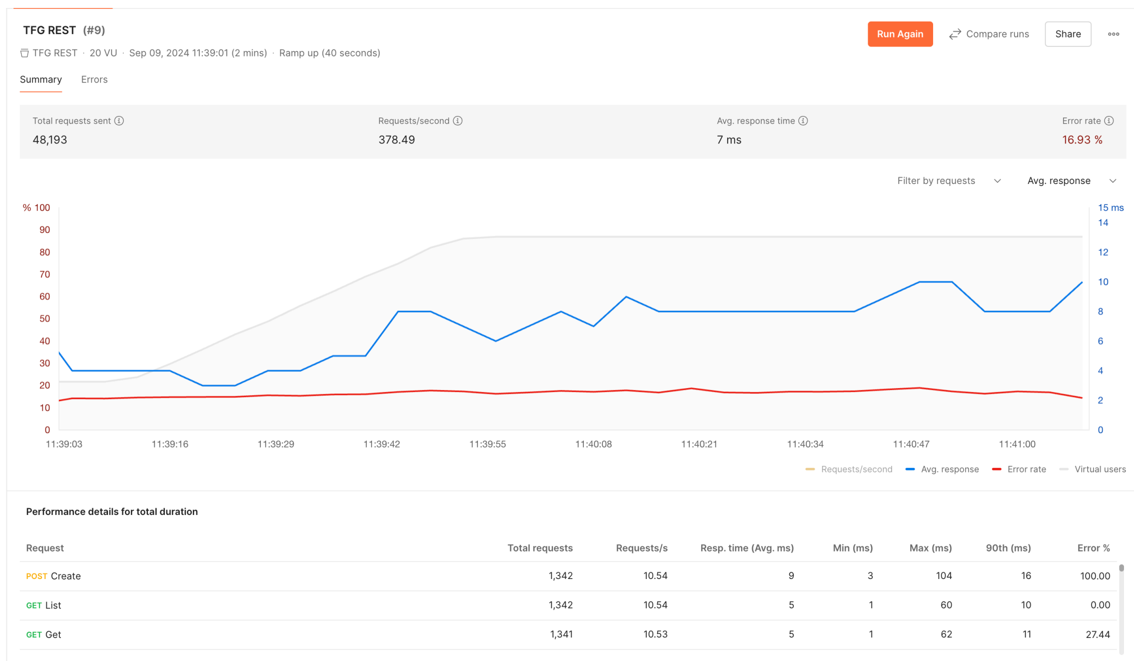


Figura 4.34 Resultados de prueba de rendimiento sobre la colección TFG REST en Postman.

4.5 Script k6 para Pruebas de Carga y Rendimiento

Para la realización de los scripts de pruebas de carga y rendimiento se ha diseñado el mismo test para los 3 servidore y luego se han adaptado a la API que ofrece cada uno. Se ha empezado con la prueba del servidor REST, ya que k6 está pensado principalmente para servidores HTTP tradicionales [9], y esta se ha modificado para usar y aprovechar las funcionalidades de GraphQL y gRPC.

La estructura general de la prueba está dividida en escenarios que simulan un comportamiento específico de los usuarios y aumenta la cantidad de estos gradualmente durante 1 minuto hasta llegar a 1000 usuarios concurrentes. Luego se mantienen esos 1000 usuarios simultáneos constantemente durante 30 segundos. Y finalmente se reducen la cantidad de usuarios durante 30 segundos hasta llegar a cero.

Se han diseñado 4 escenarios diferentes para modelar el comportamiento de distintos tipos de usuarios. La intención no es tanto simular la actividad de un usuario real, como la de poner a prueba todas las capacidades de los servidores sin caer en pruebas demasiado sintéticas que no se puedan traducir tan directamente a los casos de uso reales. Los escenarios son:

- **Consumer**: Consume todos los endpoints en los que se obtiene información de la aplicación siguiendo un comportamiento verosímil. Primero lista todos los usuarios, escoge uno de forma aleatoria y obtiene

sus detalles. Luego lista todas las publicaciones del usuario, toma una aleatoriamente y recupera sus detalles. Después hace lo mismo con los comentarios del usuario, los chats del usuario y los mensajes del usuario. Y finalmente elige al azar un participante del chat anterior y pide todos sus mensajes en ese chat.

- **Producer**: Crea nuevas entidades de todos los tipos posibles, también siguiendo un comportamiento relativamente creíble. Empieza creando un usuario. Luego crea una publicación con ese usuario. Escribe un comentario en la publicación. Crea un chat y añade el nuevo usuario al chat. Y acaba escribiendo un mensaje en ese chat con el mismo usuario.
- **Updater**: Realiza exactamente las mismas operaciones que el Producer, pero realizando actualizaciones parciales en vez de crear entidades enteras desde cero. Por ejemplo, en vez de crear un usuario, le cambia el correo electrónico a otro. Y en vez de crear mensajes, comentarios y publicaciones, simplemente les cambia el contenido.
- **Cache**: Esta es la única prueba considerada sintética. En esta se prueba el rendimiento de la base de datos en memoria Redis mediante un *ping* (excepto en GraphQL que no tiene función ping), asignando un valor aleatorio a una clave al azar e inmediatamente recuperando el valor de esta clave. En este caso hay 10.000 usuarios concurrentes y no se espera nada entre operaciones. Además, se ha ejecutado solo durante 15 segundos (más 30 de *ramp down*) para que el archivo de los resultados no fuese demasiado grande. Con 2 minutos y medio de duración se obtenían archivos de 10 Gbs. Se ha separado del resto para no enturbiar los resultados generales de las otras pruebas.

Entre cada acción el usuario espera 1 segundo antes de realizar la siguiente acción. Esto para reflejar mejor la actividad de un usuario real, pero permitiendo cargar el servidor con muchas peticiones.

Cada escenario se ejecuta secuencialmente una única vez en orden excepto por el escenario Consumer que se ejecuta una segunda vez después del escenario Producer para simular la consulta de datos masivos. Ya que el Producer se ha encargado de generar muchas nuevas entidades que ahora Consumer puede consultar. Esto muestra la resiliencia de las APIs ante peticiones que requieren transportar muchos datos de una sola vez. A este escenario se le llama Big Consumer.

Para las pruebas de la API REST es suficiente con las métricas por defecto HTTP de k6. Principalmente se hace uso de *http_reqs* (peticiones realizadas), *http_req_duration* (latencia) y *http_req_failed* (tasa de error).

En el caso de GraphQL se ha requerido de una métrica personalizada para medir de forma correcta la tasa de error, ya que los errores en GraphQL no se devuelven con un estatus HTTP de error (códigos 4xx y 5xx), estos siempre devuelven estatus 200 y si hay errores incluyen una lista de estos en el mensaje de respuesta. Esta métrica se ha denominado *graphql_error_rate*.

Sobre las pruebas gRPC se han usado las métricas gRPC incluidas en k6, principalmente *grpc_req_duration*) y se han añadido 2 adicionales: *grpc_error_rate* (tasa de error) y *grpc_requests* (peticiones realizadas). Los archivos se encuentran en la carpeta */tests* de su respectivo proyecto, es decir: *rests/tests*, *graphql/tests* y *grpc/tests*. Los resultados de ejecución de los scripts se encuentran en la carpeta */tests* del repositorio, donde también está el script de análisis de datos de Python y las gráficas que este genera.

Las pruebas y los servidores se han ejecutado ambos localmente en un ordenador Linux Ubuntu 22.04.4 LTS con 16 Gb de RAM y procesador Intel Core i5-12600K de 16 núcleos. Para una ejecución más sencilla de las pruebas se ha creado un script Bash que busca y ejecuta todos los scripts k6 del proyecto junto al servido correspondiente, reseteando la base de datos y reiniciando el servidor para cada prueba. Este se encuentra en la carpeta raíz del repositorio en el archivo *runTests.sh*. El uso y ejecución de este y todos los programas y scripts se detalla en el Apéndice A: Manual de Instalación y Uso.

4.6 Script Python para Análisis Comparativo

En este apartado se detalla el diseño e implementación del script Python ubicado en la carpeta */tests* del proyecto. Este script se utiliza para el análisis y la visualización de los resultados obtenidos de las pruebas de rendimiento ejecutadas con k6. Los resultados generados por el script también se encuentran en dicha carpeta.

4.6.1 Diseño e implementación

El script está diseñado para procesar y analizar los resultados de las pruebas de rendimiento realizadas sobre las APIs de REST, GraphQL, y gRPC. Los datos provienen de archivos JSON generados por k6, una herramienta de pruebas de carga de código abierto. A partir de esos datos, el script realiza un análisis comparativo, destacando métricas clave como tasas de solicitud, latencia y tasas de error.

El script sigue una estructura modular y clara, organizada en varias funciones que manejan diferentes aspectos del análisis:

Aspectos considerados:

- **Carga e Interpretación de Datos**: Se procesan los archivos JSON generados por k6 para extraer las métricas de interés.
- **Análisis de Métricas**: Se calculan estadísticas resumidas para cada una de las APIs.
- **Comparación entre APIs**: Se comparan las tres tecnologías en función de las métricas analizadas.
- **Visualización**: Se generan gráficos (barras, gráficas apiladas y series temporales) que permiten una visualización clara de los resultados.

Funciones implementadas:

- *load_json_file()*: Esta función es responsable de cargar e interpretar los archivos JSON de salida de k6. Asegura que los datos sean interpretados correctamente para su posterior análisis. Como k6 devuelve los resultados en formato JSONL (en el que cada línea del archivo es un objeto JSON valido e independiente), ha requerido una implementación más compleja de lo habitual.
- *analyze_metrics()*: Calcula estadísticas resumidas como promedio, percentiles y tasas de error para las diferentes métricas, como latencia y transferencia de datos.
- *compare_apis()*: Compara los resultados obtenidos de las tres tecnologías API y destaca los puntos fuertes y débiles de cada una.
- *plot_bar_chart()*, *plot_stacked_bar_chart()*, *plot_detailed_time_series()*: Generan diferentes tipos de visualizaciones que permiten observar la evolución de las métricas a lo largo del tiempo, así como comparaciones globales entre las APIs.

El script se enfoca en analizar varias métricas clave para comparar las tecnologías:

- **Tasa de Solicitudes**: Mide cuántas peticiones por segundo puede manejar cada tecnología.
- **Latencia**: El tiempo que tarda cada petición en ser procesada y recibir una respuesta.
- **Tasa de Errores**: Registra la cantidad de peticiones fallidas durante las pruebas.
- **Transferencia de Datos**: Analiza los datos enviados y recibidos por las APIs.

El script genera varias visualizaciones para facilitar la interpretación de los datos:

- **Gráficos de Barras**: Comparan globalmente las métricas clave entre las tres tecnologías en cada prueba k6.
- **Gráficos Apilados**: Representan la transferencia de datos (enviados y recibidos) de manera clara. Aunque la cantidad de datos enviados en insignificante en comparación con la de datos recibidos, por lo que acaba no siendo tan útil para sacar conclusiones.
- **Series Temporales**: Muestran la evolución de las métricas a lo largo del tiempo para cada una de las pruebas. Las métricas consideradas son peticiones por segundo, latencia, tasa de error, datos enviados y datos recibidos.

4.7 Herramientas Accesorias

Para facilitar el desarrollo del proyecto se han utilizado 2 herramientas del entorno de Go que no son directamente relevantes para la implementación de los servidores API.

La primera herramienta es Air. Esta herramienta aprovecha la velocidad de compilación de Go para implementar un sistema de "refresco en vivo" del programa. Air compila y ejecuta el programa Go de la misma forma que el propio compilador de Go lo hace. Pero, además, vigila los archivos de Go en busca de modificaciones y, en cuanto detecta un cambio, detiene el programa actual, lo recompila con el nuevo código y lo ejecuta. De esta forma se "refresca" el programa con la nueva implementación de forma automática cada vez que se guarda un cambio.

La segunda es GoDotEnv, la cual permite la gestión de variables de entorno para modificar la configuración del programa mediante el uso de archivos .env. Esta es una práctica estándar en muchos lenguajes de programación, como en JavaScript y en Ruby. Y permite incluir datos sensibles o altamente configurables al programa sin necesidad de incluirlos en el código, el cual puede ser filtrado y requiere un alto esfuerzo para modificarlo.

5

Resultados y Pruebas

Este capítulo expone, contextualiza y analiza los resultados obtenidos a partir de las pruebas realizadas. En la primera parte, se presentan las métricas y datos recogidos durante las pruebas de carga y rendimiento, apoyados por visualizaciones que facilitan su interpretación. Posteriormente, se lleva a cabo un análisis detallado de esos datos, tanto de forma manual como utilizando el script de análisis de datos en Python, lo que permite generar gráficos y visualizaciones para una correcta interpretación de los resultados.

5.1 Pruebas de Carga y Rendimiento

Las pruebas de carga y rendimiento con k6 producen dos conjuntos de datos como resultado. El primero es un resumen del rendimiento general de todas las peticiones y pruebas realizadas durante la ejecución del script. Este aparece directamente en la terminal al terminar la ejecución, pero con el uso del script bash *runTests.sh* se han redirigido un archivo TXT para preservar el resultado.

Aunque estos resultados no van a ser usados para el análisis de datos, sirve para hacerse una idea general del rendimiento de las APIs en cada prueba.

A continuación se muestra el resumen de cada prueba junto a una breve explicación e interpretación de las mismas. Si se quiere ver un análisis más detallado de cada prueba donde se lleguen a conclusiones sustantivas, véase el apartado 5.2 de esta capítulo.

5.1.1 REST

Prueba General

La figura 5.1 muestra los resultados de la prueba general de k6 sobre la API REST (*rest/tests/rest.js*). Esta prueba evaluó la latencia, la tasa de errores y el rendimiento en términos de solicitudes por segundo (RPS). Los resultados indicaron:

- **Tasa de solicitudes (RPS)**: Un valor promedio de 671,02 solicitudes por segundo.
- **Latencia**: La latencia promedio fue del 159,75 ms. Y el 95% de las solicitudes se completaron en menos de 1,11 s.
- **Datos transferidos**: Se recibieron aproximadamente 61,95 KB/req y se enviaron 144,56 B/req.
- **Tasa de errores**: El 14,03 % de las solicitudes fallaron.

```
data_received..................: 18 GB  41 MB/s
data_sent......................: 42 MB  93 kB/s
group_duration.................: avg=3.97s    min=25.08ms  med=4.03s    max=25.71s   p(90)=6.04s    p(95)=6.28s
http_req_blocked...............: avg=3.56µs   min=317ns    med=2.66µs   max=41.71ms  p(90)=4.3µs    p(95)=5.19µs
http_req_connecting............: avg=495ns    min=0s       med=0s       max=41.68ms  p(90)=0s       p(95)=0s
http_req_duration..............: avg=159.75ms min=39.73µs  med=3.61ms   max=6.49s    p(90)=395.24ms p(95)=1.11s
  { expected_response:true }...: avg=78.74ms  min=222.53µs med=3.24ms   max=6.49s    p(90)=18.59ms  p(95)=153.19ms
http_req_failed................: 14.03% ✓ 42752      ✗ 261900
http_req_receiving.............: avg=92.04µs  min=4.44µs   med=27.16µs  max=129.15ms p(90)=46.98µs  p(95)=63.41µs
http_req_sending...............: avg=12.71µs  min=1.67µs   med=10.45µs  max=96.48ms  p(90)=17.77µs  p(95)=21.32µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=159.64ms min=21.98µs  med=3.56ms   max=6.49s    p(90)=394.94ms p(95)=1.11s
http_reqs......................: 304652 671.023663/s
iteration_duration.............: avg=3.97s    min=25.11ms  med=4.03s    max=25.71s   p(90)=6.04s    p(95)=6.28s
iterations.....................: 63042  138.855723/s
vus............................: 6      min=0         max=1000
vus_max........................: 1000   min=1000      max=1000
```

Figura 5.1 Resultados resumidos de la prueba general de k6 de REST, *rest/tests/rest.js*.

Prueba Sintética de Caché

En la figura 5.2 se presentan los resultados de la prueba de caché sintética (*rest/tests/restCache.js*). Esta prueba simula un escenario en el que la API REST usa caché para mejorar el rendimiento:

- **Tasa de solicitudes (RPS)**: La tasa de solicitudes fue mayor que en la prueba general, alcanzando un promedio de 36.376,86 solicitudes por segundo.
- **Latencia**: La latencia promedio fue de 227,55 ms, con un 95% de las solicitudes completadas en menos de 444,01 ms.
- **Datos transferidos**: Se recibieron y enviaron aproximadamente 392,38 B/req de datos cada uno.
- **Tasa de errores**: Ninguna petición falló.

```
data_received..................: 70 MB  4.5 MB/s
data_sent......................: 70 MB  4.6 MB/s
group_duration.................: avg=683.15ms min=180.63ms med=609.93ms max=1.34s    p(90)=892.8ms  p(95)=985.75ms
http_req_blocked...............: avg=11.61µs  min=237ns    med=951ns    max=48.36ms  p(90)=1.99µs   p(95)=2.97µs
http_req_connecting............: avg=8.53µs   min=0s       med=0s       max=48.33ms  p(90)=0s       p(95)=0s
http_req_duration..............: avg=227.55ms min=7.07ms   med=197.82ms max=713.21ms p(90)=351.98ms p(95)=444.01ms
  { expected_response:true }...: avg=227.55ms min=7.07ms   med=197.82ms max=713.21ms p(90)=351.98ms p(95)=444.01ms
http_req_failed................: 0.00%  ✓ 0           ✗ 561192
http_req_receiving.............: avg=33.67µs  min=4.93µs   med=12.64µs  max=32.84ms  p(90)=27.93µs  p(95)=70.47µs
http_req_sending...............: avg=11.12µs  min=1.5µs    med=4.95µs   max=25.58ms  p(90)=9.64µs   p(95)=17.55µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=227.5ms  min=6.9ms    med=197.79ms max=713.19ms p(90)=351.95ms p(95)=443.95ms
http_reqs......................: 561192 36376.859745/s
iteration_duration.............: avg=683.17ms min=180.65ms med=609.95ms max=1.34s    p(90)=892.83ms p(95)=985.76ms
iterations.....................: 187064 12125.619915/s
vus............................: 10000  min=1571      max=10000
vus_max........................: 10000  min=10000     max=10000
```

Figura 5.2 Resultados resumidos de la prueba sintética de caché de k6 de REST, *rest/tests/restCache.js*.

5.1.2 GraphQL

Prueba General Datos Mínimos

En la figura 5.3 se muestran los resultados de la prueba de GraphQL solicitando la cantidad mínima de datos (*graphql/tests/graphqlMin.js*):

- **Tasa de solicitudes (RPS)**: Se observó un valor de 434,60 solicitudes por segundo.
- **Latencia**: La latencia promedio fue de 444,22 ms, con el 95% de las solicitudes completadas en menos de 2,39 s.
- **Transferencia de datos**: La cantidad de datos transferidos fue mínima. Especialmente en la cantidad de datos recibida con solo 6,38 KB/req, aunque la cantidad enviada aumenta con 356,12 B/req enviados.
- **Tasa de errores**: Alrededor del 17,62% de las consultas GraphQL fallaron o presentaron algún error.

```
data_received..................: 1.2 GB 2.7 MB/s
data_sent......................: 67 MB  147 kB/s
graphql_error_rate.............: 17.62% ✓ 34762      × 162518
group_duration.................: avg=3.1s     min=2.06ms   med=1.91s    max=30.33s  p(90)=8.01s   p(95)=8.02s
http_req_blocked...............: avg=4.21µs   min=297ns    med=2.98µs   max=31.63ms p(90)=5.01µs  p(95)=5.95µs
http_req_connecting............: avg=688ns    min=0s       med=0s       max=31.57ms p(90)=0s      p(95)=0s
http_req_duration..............: avg=444.22ms min=279.93µs med=8.02ms   max=29.33s  p(90)=1.62s   p(95)=2.39s
  { expected_response:true }...: avg=444.22ms min=279.93µs med=8.02ms   max=29.33s  p(90)=1.62s   p(95)=2.39s
http_req_failed................: 0.00%  ✓ 0          × 197280
http_req_receiving.............: avg=52.08µs  min=4.79µs   med=30.17µs max=70.93ms p(90)=63.62µs p(95)=86.4µs
http_req_sending...............: avg=19.2µs   min=1.86µs   med=13.95µs max=168.3ms p(90)=25.23µs p(95)=29.83µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s      p(95)=0s
http_req_waiting...............: avg=444.15ms min=244.22µs med=7.97ms   max=29.33s  p(90)=1.62s   p(95)=2.39s
http_reqs......................: 197280 434.60454/s
iteration_duration.............: avg=3.1s     min=2.08ms   med=1.91s    max=30.33s  p(90)=8.01s   p(95)=8.02s
iterations.....................: 80644  177.657383/s
vus............................: 17     min=0        max=1000
vus_max........................: 1000   min=1000     max=1000
```

Figura 5.3 Resultados resumidos de la prueba general consultando la cantidad mínima de datos de k6 de GraphQL, *graphql/tests/graphqlMin.js*.

Prueba General Datos Máximos

La figura 5.4 presenta los resultados de la prueba solicitando la cantidad máxima de datos (*graphql/tests/graphqlMax.js*). Los resultados fueron:

- **Tasa de solicitudes (RPS)**: La tasa de solicitudes fue significativamente menor, con un promedio de 398,74 solicitudes por segundo.
- **Latencia**: La latencia promedio fue de 524,02 ms, pero con una latencia de menos de 3,36 s en el 95% de los casos debido a la mayor cantidad de datos transferidos.
- **Datos transferidos**: La cantidad de datos recibidos se multiplica por más de 5 con 38,28 KB/req y la enviada aumenta marginalmente con 502,77 B/req.
- **Tasa de errores**: La tasa de errores se reduce ligeramente hasta el 14,04%.

```
data_received..................: 6.7 GB 15 MB/s
data_sent......................: 88 MB  192 kB/s
graphql_error_rate.............: 14.04% ✓ 25783      × 157748
group_duration.................: avg=3.49s    min=2.91ms   med=2.25s    max=50.51s   p(90)=8.01s   p(95)=8.07s
http_req_blocked...............: avg=4.76µs   min=308ns    med=3.09µs   max=20.79ms  p(90)=5.13µs  p(95)=6.05µs
http_req_connecting............: avg=628ns    min=0s       med=0s       max=20.76ms  p(90)=0s      p(95)=0s
http_req_duration..............: avg=524.02ms min=319.28µs med=8.05ms   max=32.99s   p(90)=1.75s   p(95)=3.36s
  { expected_response:true }...: avg=524.02ms min=319.28µs med=8.05ms   max=32.99s   p(90)=1.75s   p(95)=3.36s
http_req_failed................: 0.00%  ✓ 0          × 183531
http_req_receiving.............: avg=108.42µs min=5.58µs   med=31.7µs   max=217.28ms p(90)=62.54µs p(95)=81.3µs
http_req_sending...............: avg=19.69µs  min=2.38µs   med=14.63µs  max=61.91ms  p(90)=25.98µs p(95)=30.78µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s      p(95)=0s
http_req_waiting...............: avg=523.89ms min=295.99µs med=7.99ms   max=32.99s   p(90)=1.75s   p(95)=3.36s
http_reqs......................: 183531 398.738713/s
iteration_duration.............: avg=3.49s    min=2.93ms   med=2.25s    max=50.51s   p(90)=8.01s   p(95)=8.07s
iterations.....................: 72869  158.314897/s
vus............................: 28     min=0        max=1000
vus_max........................: 1000   min=1000     max=1000
```

Figura 5.4 Resultados resumidos de la prueba general consultando la cantidad máxima de datos de k6 de GraphQL, *graphql/tests/graphqlMax.js*.

Prueba Sintética de Caché

En la figura 5.5 se muestran los resultados de la prueba de caché sintética de GraphQL (*graphql/tests/graphqlCache.js*). Esta prueba, similar a la de REST, mostró una mejora en el rendimiento:

- **Tasa de solicitudes (RPS)**: Alcanza 34.229,83 solicitudes por segundo siendo ligeramente menor a REST.
- **Latencia**: La latencia promedio se encuentra en los 241,12 ms y con el 95% de las peticiones por debajo de 395,81 ms. Siendo la media negligiblemente mayor que REST y el percentil 95 ligeramente inferior.
- **Transferencia de datos**: Se han recibido menos de la mitad de datos por petición y enviado significativament menos que en REST con 163,95 B/req y 283,92 B/req respectivamente.
- **Tasa de errores**: Tampoco ocurrió ningún error en ninguna consulta.

```
data_received..................: 82 MB  5.4 MB/s
data_sent......................: 143 MB 9.3 MB/s
graphql_error_rate.............: 0.00%  ✓ 0           × 524436
group_duration.................: avg=483.12ms min=4.57ms med=484.85ms max=819.83ms p(90)=687.2ms  p(95)=726.31ms
http_req_blocked...............: avg=65.96µs  min=242ns  med=1.03µs   max=183.02ms p(90)=2.06µs   p(95)=2.76µs
http_req_connecting............: avg=51.26µs  min=0s     med=0s       max=182.69ms p(90)=0s       p(95)=0s
http_req_duration..............: avg=241.12ms min=1.02ms med=232.26ms max=500.24ms p(90)=367.29ms p(95)=395.81ms
  { expected_response:true }...: avg=241.12ms min=1.02ms med=232.26ms max=500.24ms p(90)=367.29ms p(95)=395.81ms
http_req_failed................: 0.00%  ✓ 0           × 524436
http_req_receiving.............: avg=42.52µs  min=4.4µs  med=13.08µs  max=92.74ms  p(90)=26.67µs  p(95)=55.98µs
http_req_sending...............: avg=25.68µs  min=1.9µs  med=5.92µs   max=51.07ms  p(90)=10.6µs   p(95)=17.34µs
http_req_tls_handshaking.......: avg=0s       min=0s     med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=241.06ms min=1.02ms med=232.23ms max=500.12ms p(90)=367.25ms p(95)=395.78ms
http_reqs......................: 524436 34229.825971/s
iteration_duration.............: avg=483.18ms min=4.63ms med=484.87ms max=819.84ms p(90)=687.21ms p(95)=726.33ms
iterations.....................: 262218 17114.912985/s
vus............................: 10000  min=1561      max=10000
vus_max........................: 10000  min=10000     max=10000
```

Figura 5.5 Resultados resumidos de la prueba sintética de caché de k6 de GraphQL, *graphql/tests/graphqlCache.js*.

5.1.3 gRPC

Prueba General

La figura 5.6 presenta los resultados de la prueba general de gRPC (*grpc/tests/grpc.js*). Los resultados indican que gRPC ofrece el mejor rendimiento en términos de latencia:

- **Tasa de solicitudes (RPS)**: Un promedio de 610,61 solicitudes por segundo, ligeramente inferior a REST.
- **Latencia**: La latencia promedio fue de solo 226,14 ms, con un 95% de las solicitudes completadas en menos de 1,56 s. También algo inferior en rendimiento a REST.

- **Transferencia de datos**: Con 41,61 KB/req recibidos y 177,77 B/req enviados, es considerablemente más eficiente que REST, pero no llega a la eficiencia de consulta de GraphQL.
- **Tasa de errores**: Destaca cortando en la mitad la mejor tasa de errores hasta ahora con 7,46% de errores.

```
data_received........: 11 GB  24 MB/s
data_sent............: 47 MB  103 kB/s
group_duration.......: avg=4.33s    min=1.89ms   med=4.05s max=26.49s p(90)=6.37s    p(95)=6.82s
grpc_error_rate......: 7.46%  ✓ 20697      × 256529
grpc_req_duration....: avg=226.14ms min=263.46µs med=8.2ms max=10.25s p(90)=126.47ms p(95)=1.56s
grpc_requests........: 277226 610.605963/s
iteration_duration...: avg=4.33s    min=2.2ms    med=4.05s max=26.49s p(90)=6.38s    p(95)=6.83s
iterations...........: 58449  128.737232/s
vus..................: 68     min=0        max=1000
vus_max..............: 1000   min=1000     max=1000
```

Figura 5.6 Resultados resumidos de la prueba general de k6 de gRPC, grpc*/tests/grpc.js*.

Prueba Sintética de Caché

Finalmente, en la figura 5.7 se presentan los resultados de la prueba de caché sintética de gRPC (*grpc/tests/grpcCache.js*). Similar a las otras tecnologías, se observó una mejora en la tasa de solicitudes y la latencia:

- **Tasa de solicitudes (RPS)**: 19.913,20 solicitudes por segundo, considerablemente inferior al resto.
- **Latencia**: La latencia promedio fue de 396,12 ms, con un 95% de las solicitudes completadas en menos de 813,05 ms. Poniendo a gRPC como la API más lenta en el uso de caché.
- **Transferencia de datos**: Enviando 115,64 B/req y recibiendo 173,46 B/req, gRPC es la tecnología que menos uso hace del ancho de banda con operaciones con caché.

```
data_received........: 34 MB  2.2 MB/s
data_sent............: 51 MB  3.3 MB/s
group_duration.......: avg=1.19s    min=15.97ms med=1.2s    max=2.79s p(90)=1.77s    p(95)=2.05s
grpc_error_rate......: 0.00%  ✓ 0           × 308298
grpc_req_duration....: avg=396.12ms min=2.45ms  med=359.4ms max=1.53s p(90)=700.79ms p(95)=813.05ms
grpc_requests........: 308298 19913.202001/s
iteration_duration...: avg=1.23s    min=18.76ms med=1.25s   max=2.8s  p(90)=1.87s    p(95)=2.11s
iterations...........: 102766 6637.734/s
vus..................: 3642   min=0          max=10000
vus_max..............: 10000  min=3924       max=10000
```

Figura 5.7 Resultados resumidos de la prueba sintética de caché de k6 de gRPC, grpc*/tests/grpc.js*.

5.1.2 Resumen

Las pruebas de carga y rendimiento han proporcionado una visión general del comportamiento de cada tecnología API (REST, GraphQL, gRPC) bajo diferentes condiciones de carga y uso de caché. REST mostró una mayor tasa de solicitudes en escenarios sin caché, pero gRPC demostró ser más eficiente en términos de latencia y transferencia de datos. Por su parte, GraphQL presentó una menor tasa de solicitudes pero mostró una mayor eficiencia en la transferencia de datos en escenarios de consultas más pesadas.

En general, los resultados sugieren que cada tecnología tiene sus fortalezas en diferentes aspectos, siendo REST la más eficiente en latencia, GraphQL la más eficiente con el tamaño de los mensajes y gRPC la más confiable. Sin embargo, estos resultados requieren un análisis más detallado para extraer conclusiones definitivas sobre la comparación de rendimiento entre las tecnologías.

5.2 Análisis de Resultados

Para obtener una visión más profunda y sustantiva de los resultados obtenidos en estas pruebas, en este apartado se presenta un análisis detallado de los datos producidos por las pruebas de k6. En este análisis se utilizan las herramientas de visualización y análisis desarrolladas en Python, las cuales procesan los datos generados por k6 y permiten una comparación exhaustiva del rendimiento de las APIs. A continuación, se presentan los resultados visuales y estadísticos que guiarán las conclusiones finales sobre el rendimiento de las diferentes tecnologías API.

5.2.1 Resultados

El script de Python genera varias visualizaciones entre las que se encuentra un gráfico de barras indicando el total de peticiones realizadas en cada prueba k6 a su respectiva API (figura 5.8). Esto sirve para hacerse una idea superficial de la capacidad de procesamiento de cada tecnología en cada caso.

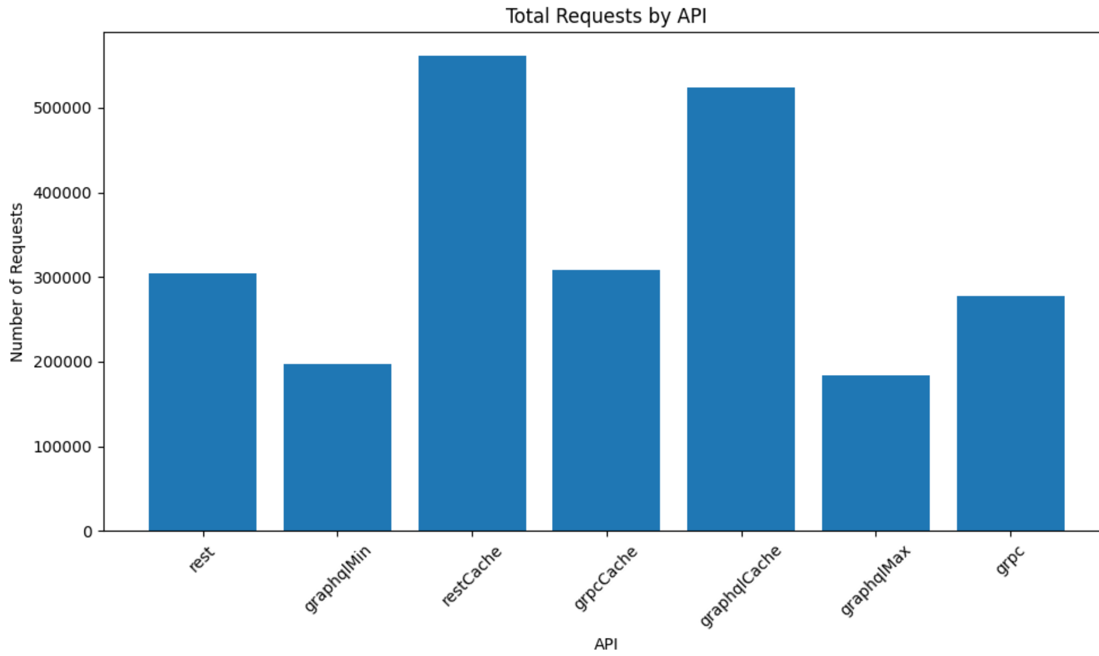


Figura 5.8 Peticiones totales realizadas en cada prueba k6.

También genera otra gráfica de barras mostrando el tiempo de respuesta promedio de cada petición a la API correspondiente en cada prueba k6 en la figura 5.9. Es útil para estimar el tiempo que suele tardar en procesar una petición cada tecnología. Es inversamente proporcional a la cantidad de peticiones realizadas en total y por segundo.

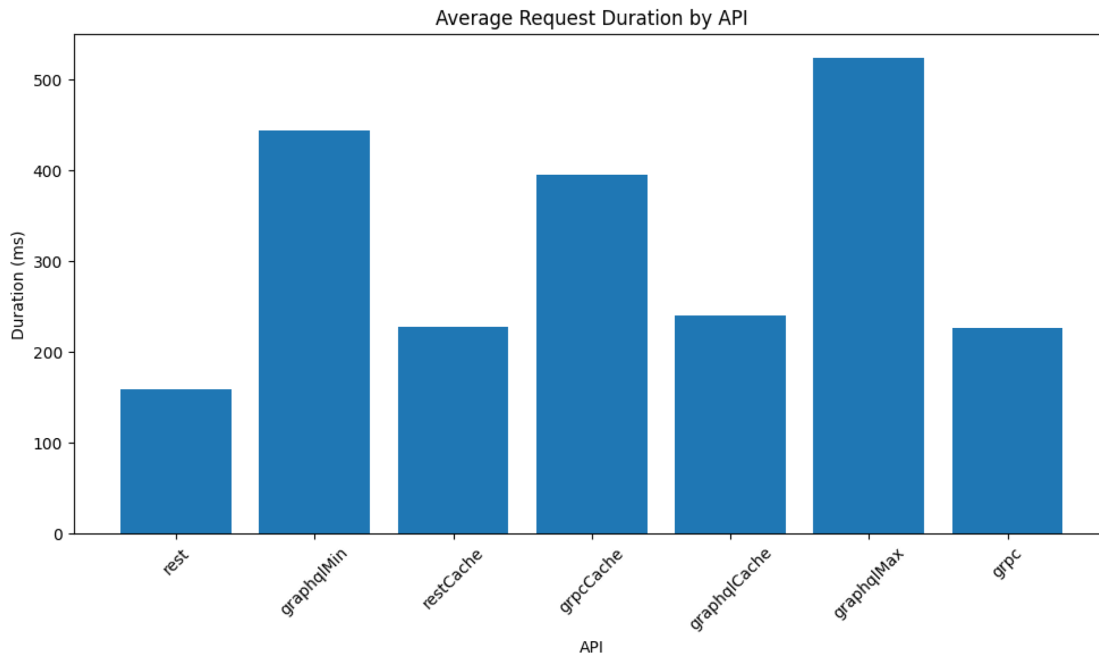


Figura 5.9 Latencia promedio en cada prueba k6.

En la figura 5.10 se muestra otro de los gráficos de barras enseñando la cantidad total de datos recibidos por todas las peticiones realizadas durante cada prueba k6 por su respectiva API en bytes. Por sí sola no aporta mucha

información relevante, ya que se podría estar recibiendo mucha información por procesar muchas más peticiones o porque cada petición optimiza menos el uso del espacio.

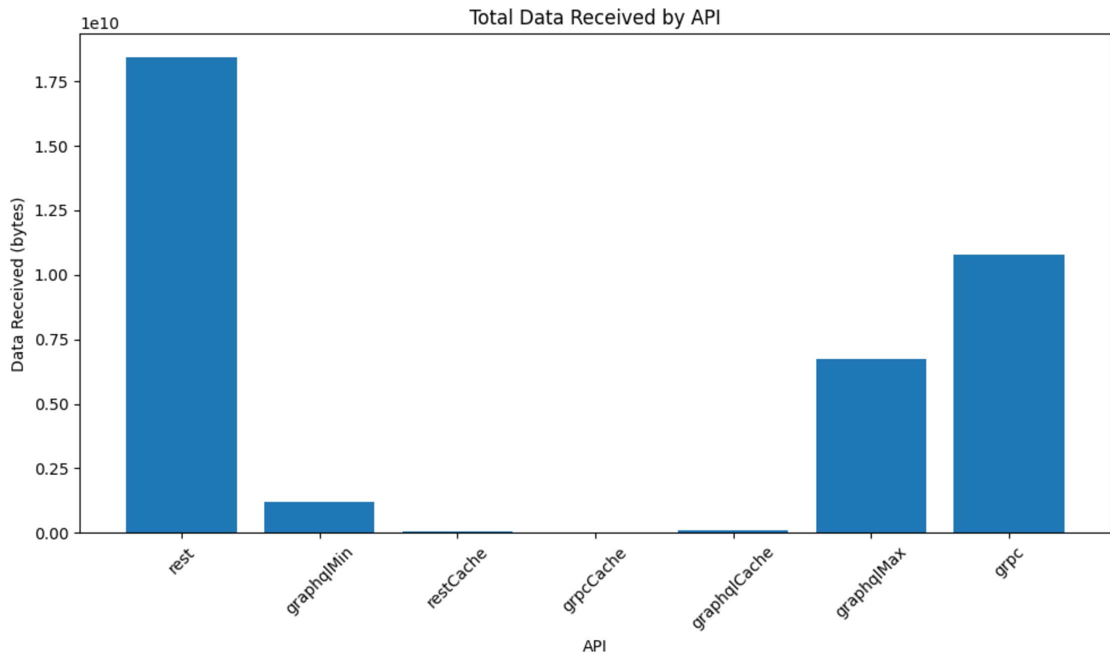


Figura 5.10 Cantidad total de datos recibidos en cada prueba k6 en bytes (1e10 B ≈ 10 GB).

Junto a este se encuentra en la figura 5.11 otro equivalente con la cantidad total de datos enviados por todas las peticiones de cada prueba k6 a su respectiva API en bytes. Al igual que con los datos recibidos, no es muy concluyente para inferir eficiencia y rendimiento de las tecnologías. Una vez se compara con las peticiones realizadas, obteniendo el tamaño medio de las peticiones, es entonces cuando se vuelve información muy útil.

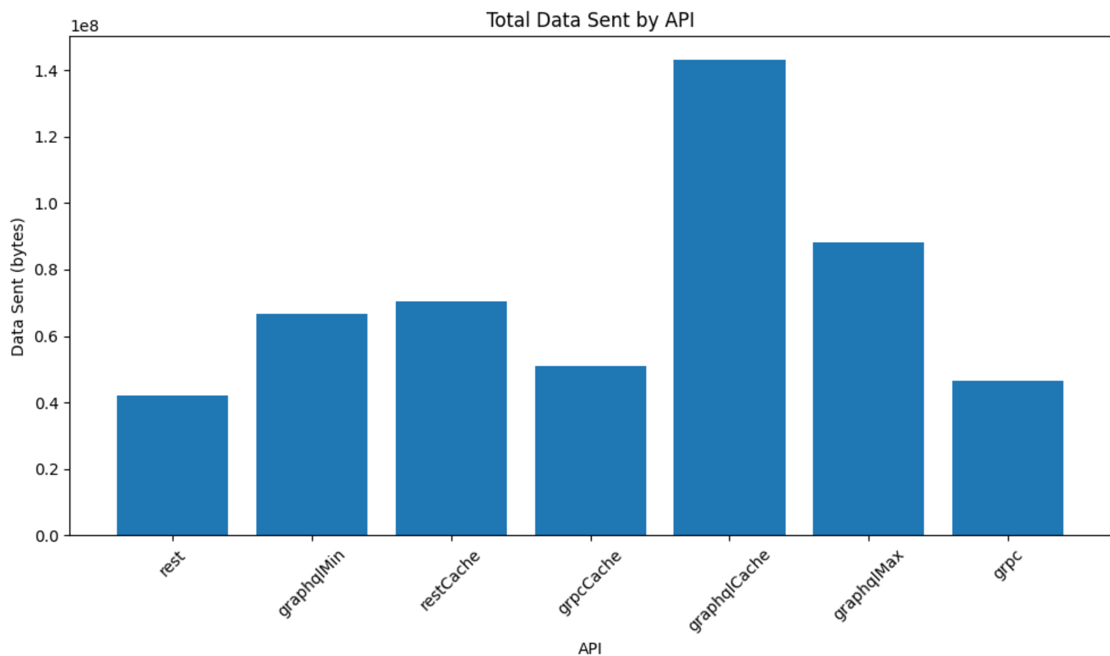


Figura 5.11 Cantidad total de datos enviados en cada prueba k6 en bytes (1e8 B ≈ 100 MB).

Y derivado de estas dos previas métricas se genera el tamaño de petición (figura 5.12) y respuesta (figura 5.13) promedio. El tamaño medio de las peticiones realizadas en cada prueba k6 a su respectiva API en bytes es imprescindible para poder analizar la eficiencia en el uso del espacio de cada tecnología.

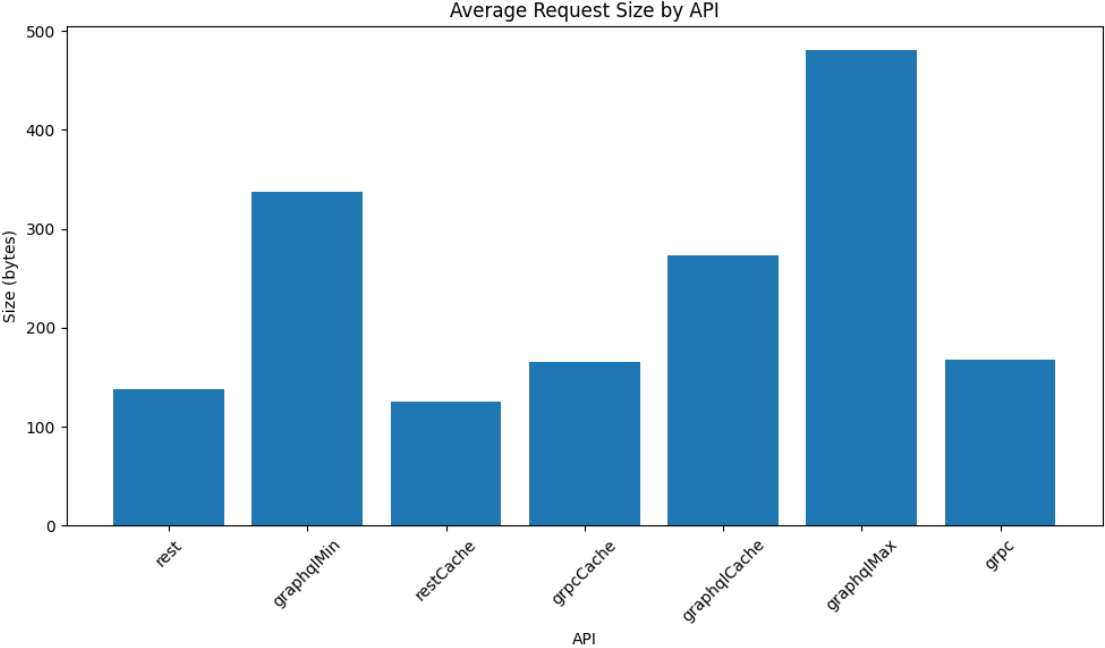


Figura 5.12 Tamaño promedio de las peticiones de cada prueba k6 en bytes.

Y el tamaño medio de las respuestas recibidas por la API correspondiente en cada prueba k6 medido en bytes es especialmente útil para comparar la mejora de uso del ancho de banda al usar eficientemente tecnologías de consulta más sofisticadas (GraphQL).

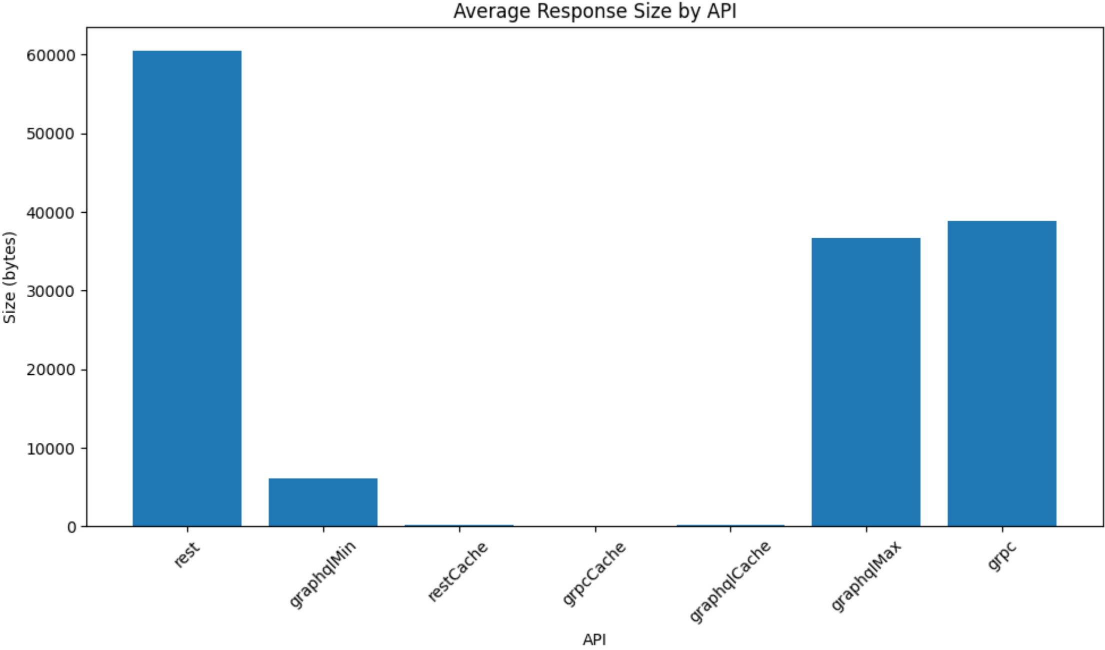


Figura 5.13 Tamaño promedio de las respuestas de cada prueba k6 en bytes.

Juntando la cantidad de datos enviados y recibidos se obtiene la figura 5.14, mostrando la cantidad total de datos transferidos en cada prueba k6 en bytes. En azul la enviada, en naranja la recibida. Con esto se puede observar que el tamaño de las peticiones es completamente negligible en comparación con el tamaño de las respuestas. Por tanto, cuando se quiera optimizar el uso de la red, mirar al tamaño de las peticiones es inútil excepto en el caso de datos atómicos como con las cachés.

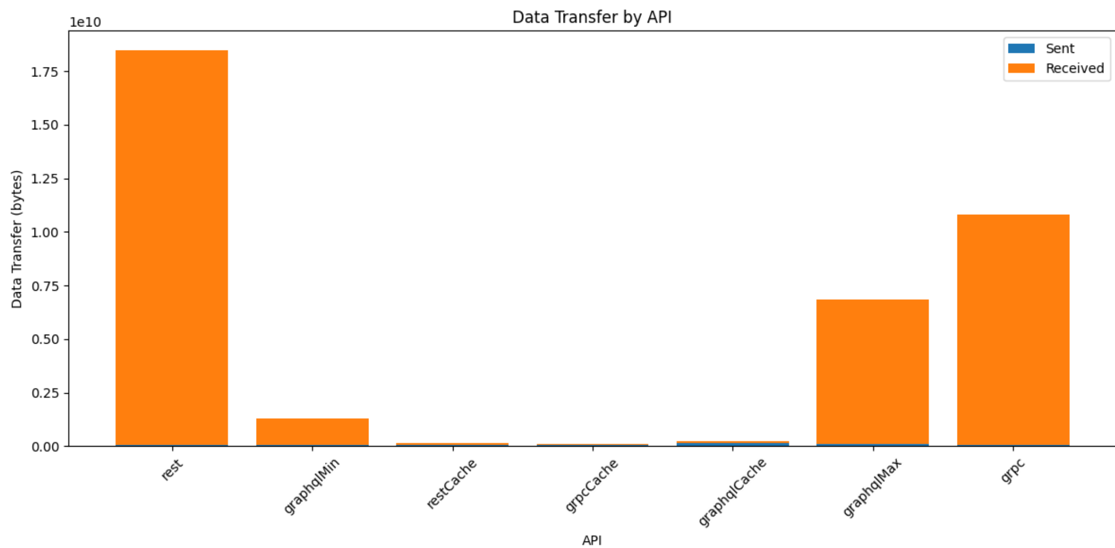


Figura 5.14 Cantidad total de datos transferidos en cada prueba k6 en bytes (1e10 B ≈ 10 GB)

Finalmente, en la última gráfica de barras generada por el script Python (figura 5.15) se muestra el porcentaje de peticiones que acabaron en un error en cada prueba k6. De gran utilidad para comprobar la resiliencia y la tolerancia de las tecnologías API.

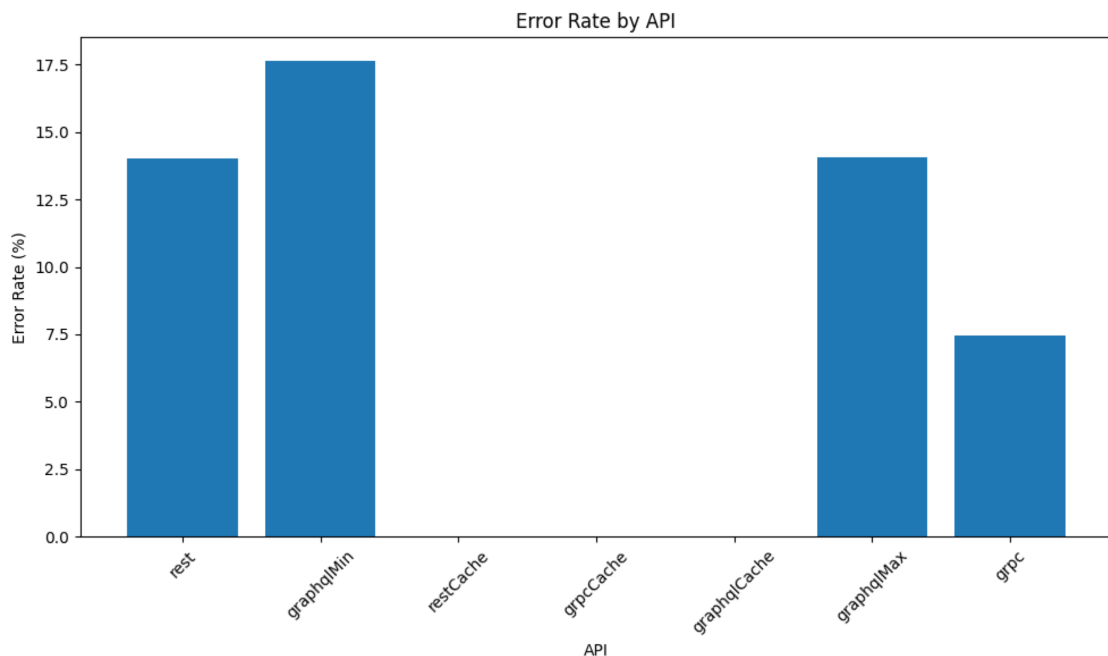


Figura 5.15 Tasa de errores en cada prueba k6.

Todas las figuras de la 5.16 a la 5.22 comparten una estructura idéntica relativamente compleja. Así que se va a explicar esta estructura común antes de entrar en detalle qué muestran cada una.

Estas son cuatro gráficas compartiendo eje de abscisas. Este eje marca el momento exacto en el que ocurrió la recogida de la métrica medida. Las cuatro gráficas de las que se componen muestran la evolución de cuatro métricas medidas a lo largo de la ejecución de una prueba k6 en concreto. Estas cuatro métricas son: peticiones por segundo, latencia en milisegundos, tasa de errores, cantidad de datos enviados y cantidad de datos recibidos (ambas en bytes).

Con esto se puede ver de una forma sencilla y visual la forma en la que las métricas se ven afectadas a lo largo del tiempo por las diferentes partes de las pruebas y como cada métrica se correlaciona o no con el resto. A continuación, se explican un poco más en detalle las figuras.

En la figura 5.16 se muestra las series temporales de cuatro métricas clave recolectadas durante una prueba general de la API REST utilizando k6. Se visualiza la evolución de las peticiones por segundo, la latencia en milisegundos, la tasa de errores, y la cantidad de datos transferidos (tanto enviados como recibidos en bytes). Estas métricas permiten entender cómo varía el rendimiento de la API REST a medida que avanza la prueba, proporcionando una visión clara de la estabilidad y eficiencia del sistema bajo cargas variables.

Junto a esta se muestra la figura 5.17, ilustrando el comportamiento de las mismas métricas (peticiones por segundo, latencia, tasa de errores y cantidad de datos transferidos) durante una prueba sintética que simula el uso intensivo de una caché Redis en la API REST. La utilización de Redis introduce mejoras significativas en el rendimiento al reducir la latencia y la cantidad de datos procesados, lo que se refleja en los gráficos. La figura ofrece un análisis detallado de cómo la caché afecta al rendimiento de REST.

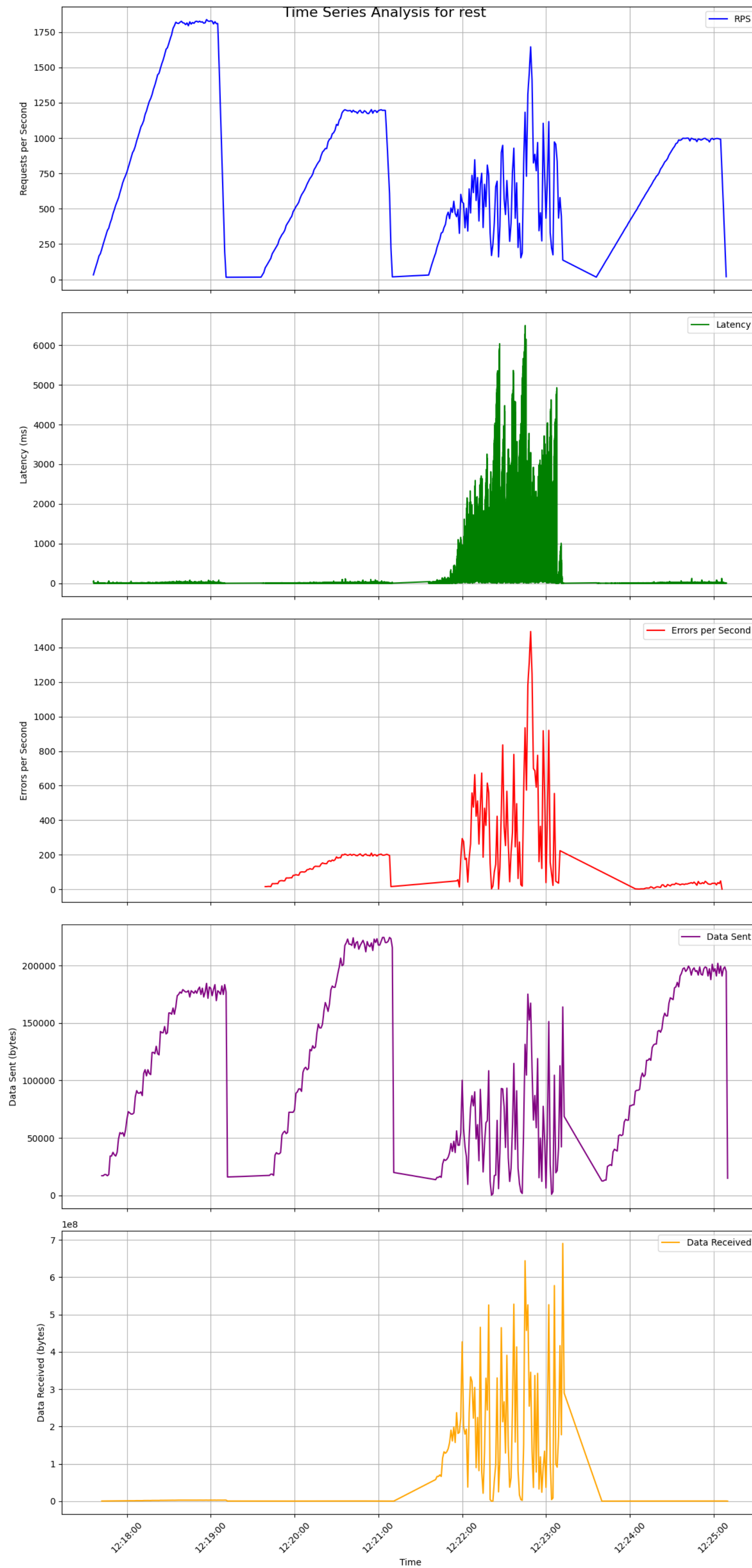


Figura 5.16 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba general de k6 para REST.

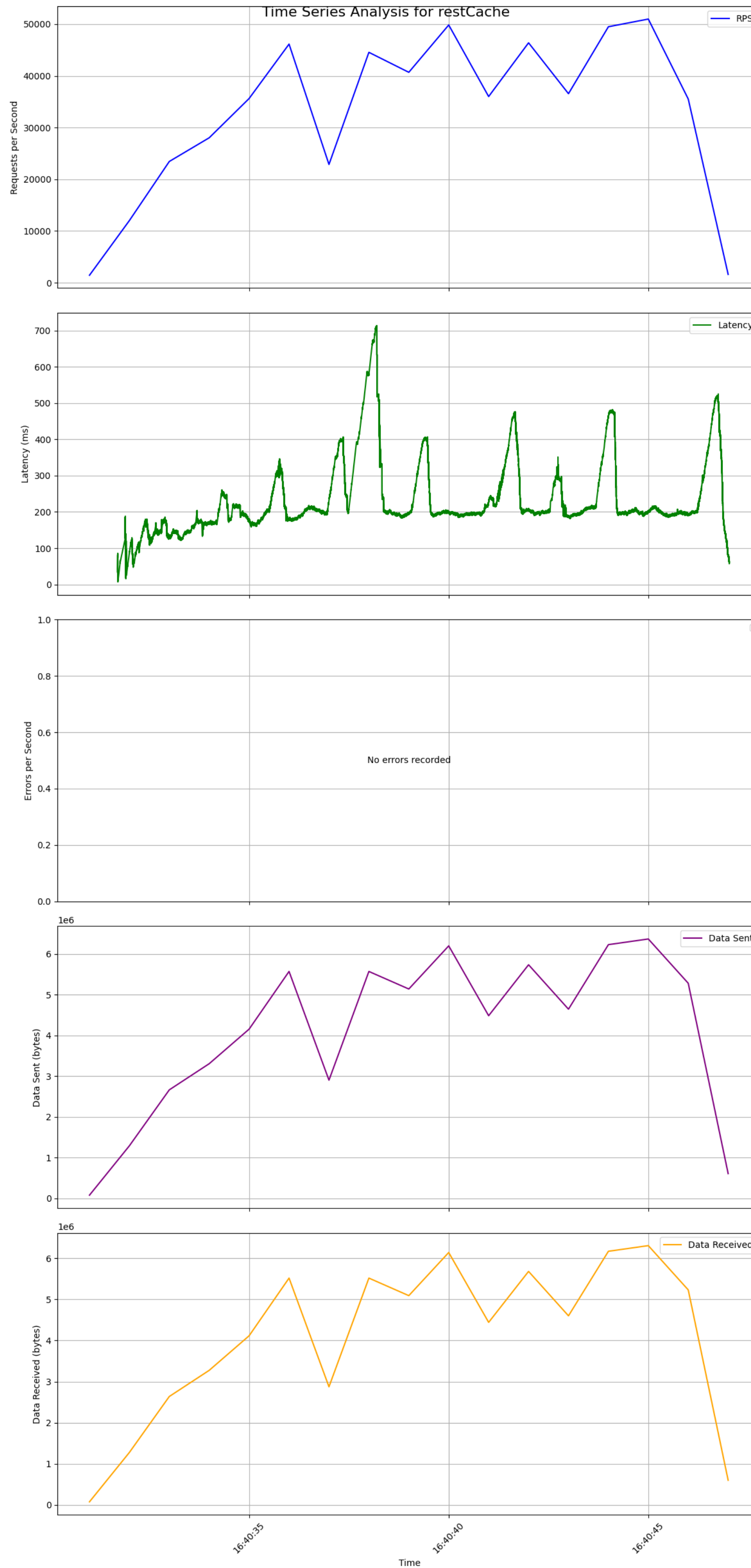


Figura 5.17 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba de caché de k6 para REST.

Pasando a los resultados sobre la API Graphql, en la figura 5.18, se observan las mismas métricas que en las anteriores (peticiones por segundo, latencia, tasa de errores y datos transferidos) aplicadas a una prueba sintética sobre la API GraphQL con caché Redis. La caché mejora el rendimiento de GraphQL de manera similar a REST, reduciendo la latencia y optimizando las solicitudes. Los gráficos permiten ver cómo la caché influye en la eficiencia de las consultas.

En la figura 5.19, se presentan las series temporales de las métricas para una prueba general, en la que se consulta una cantidad de datos similar a la que se usa en las pruebas de REST y gRPC. Los gráficos permiten comparar el rendimiento de GraphQL frente a REST y gRPC bajo condiciones equivalentes, proporcionando información valiosa sobre la eficiencia de las consultas en términos de tiempo de respuesta, tasa de errores y uso de datos.

Y en el último conjunto de gráficas de la API GraphQL (figura 5.20) se muestra la evolución de las métricas en una prueba general donde se solicita la cantidad mínima de datos necesarios para completar las operaciones. A través de los gráficos, es posible analizar cómo se comporta la API bajo la mínima carga de datos, proporcionando una visión clara de la optimización en el uso de ancho de banda y la eficiencia en la ejecución de consultas simples.

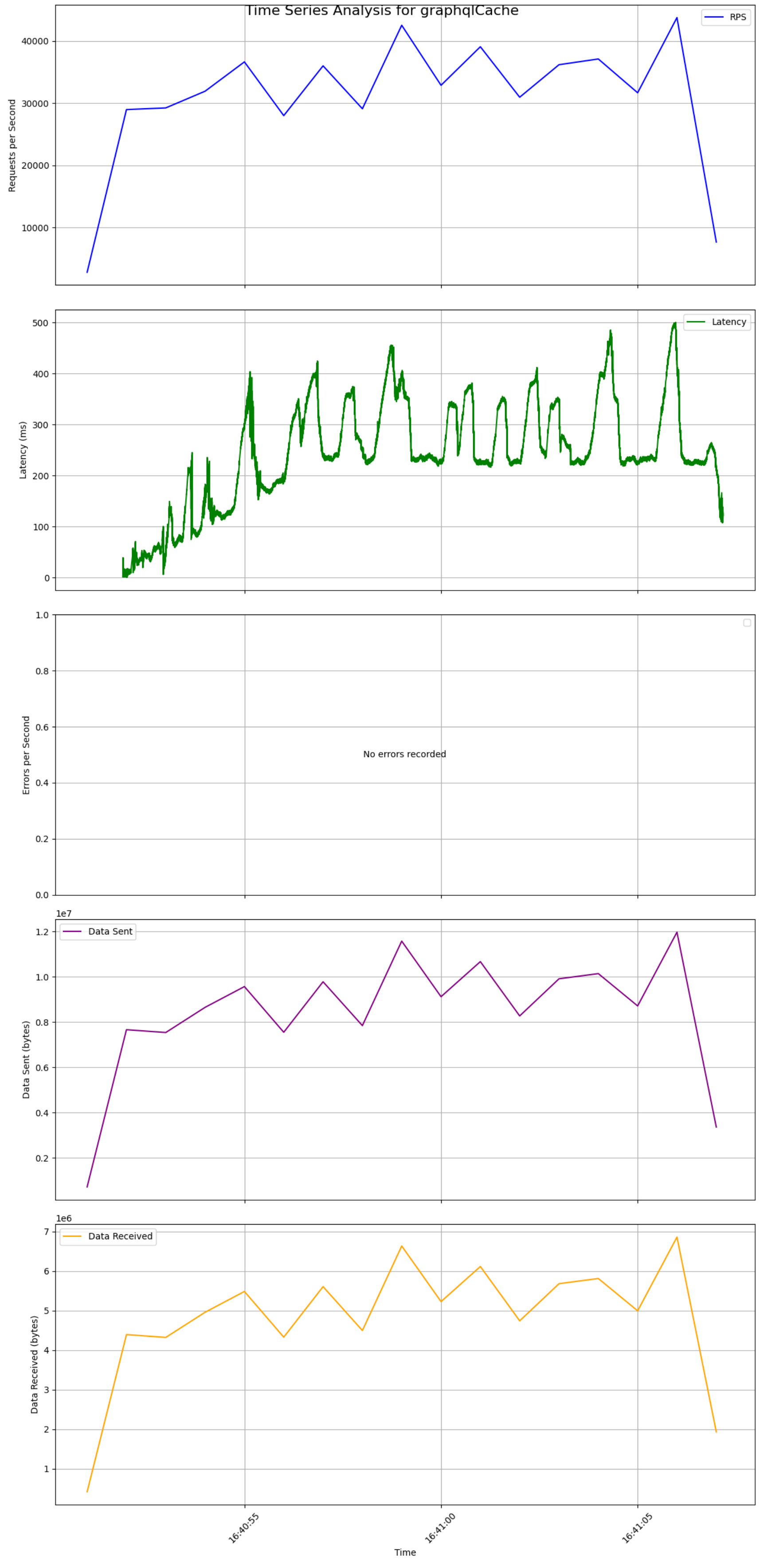


Figura 5.18 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba de caché de k6 para GraphQL.

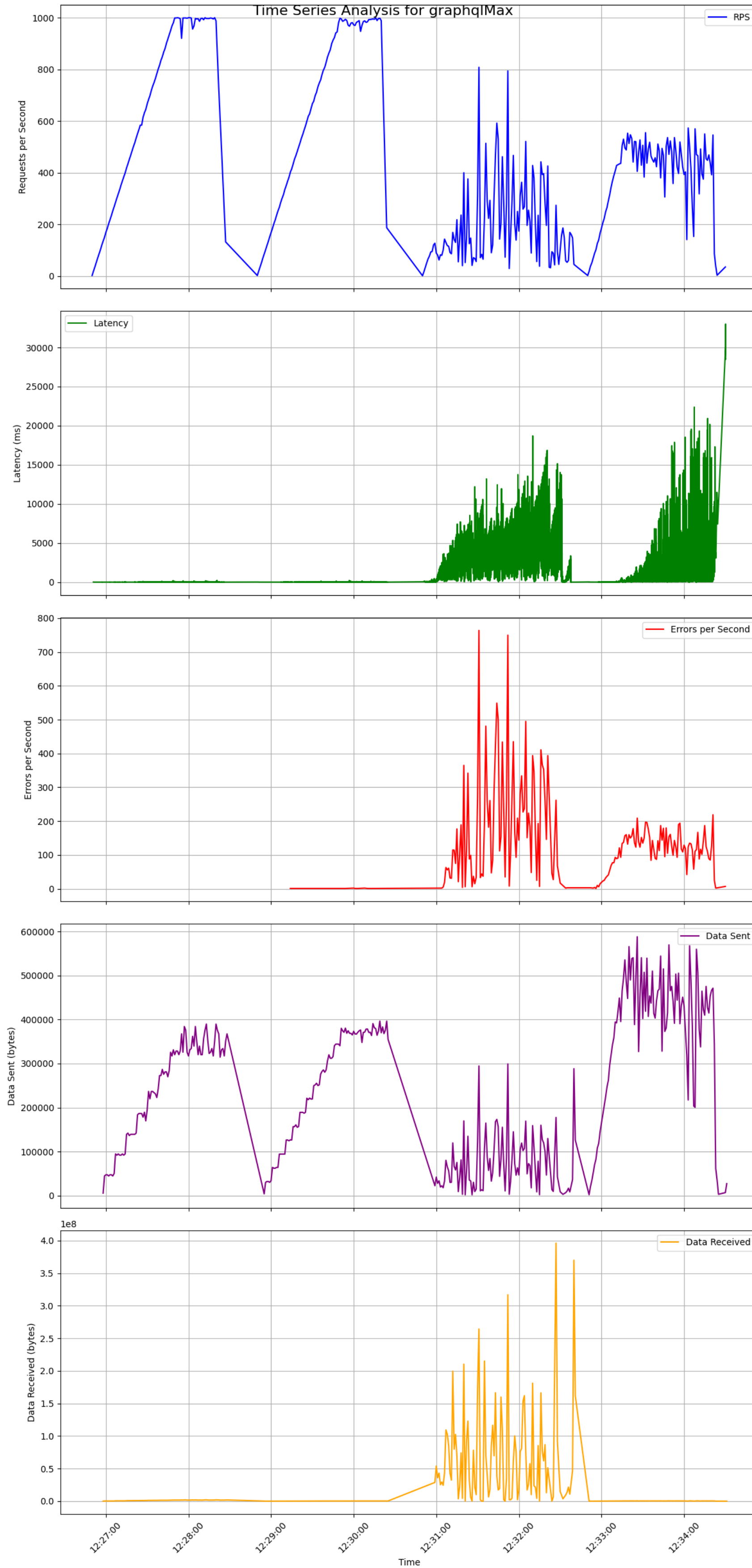


Figura 5.19 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba general de máximos datos de k6 para GraphQL.

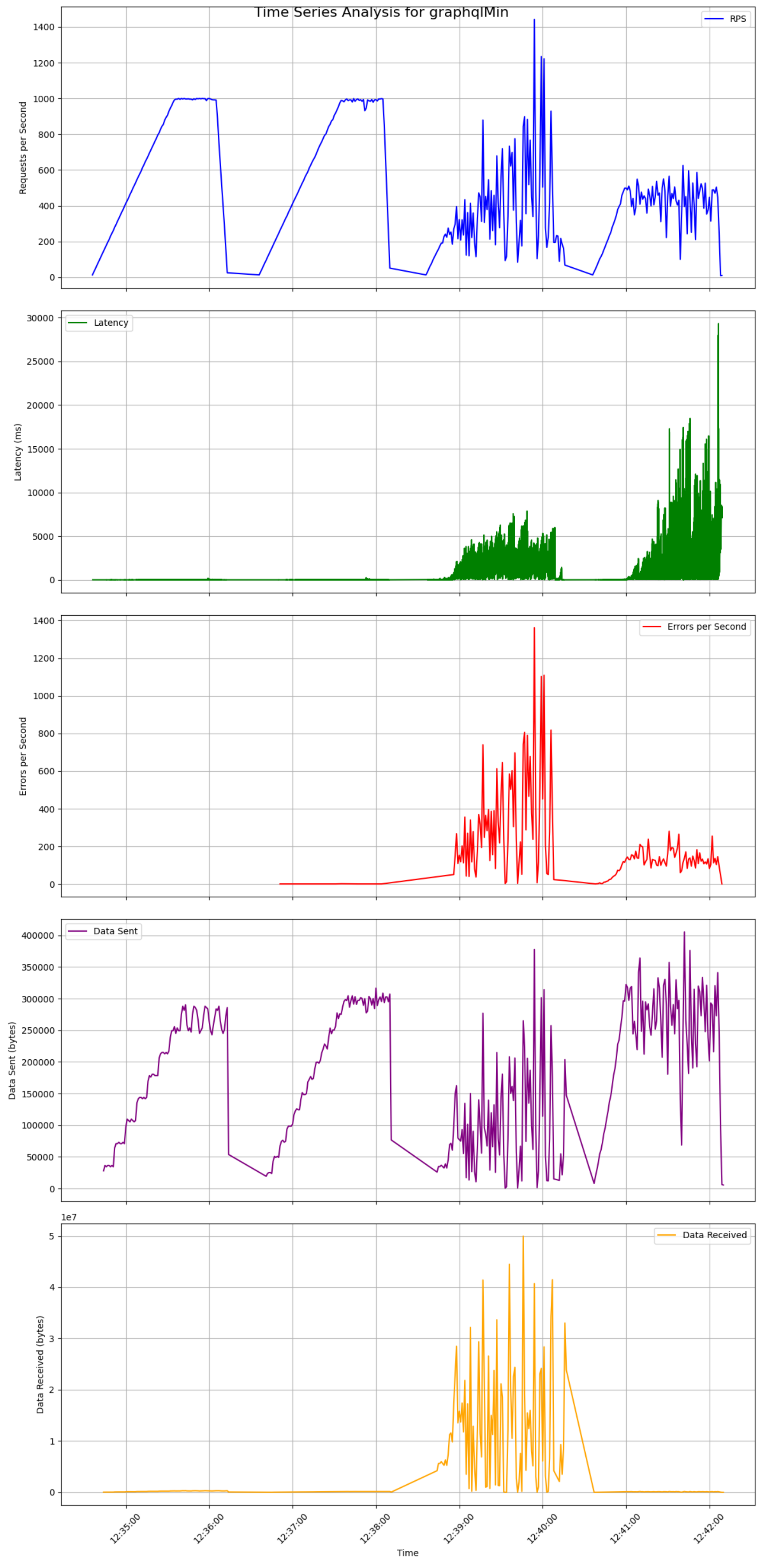


Figura 5.20 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba general de datos mínimos de k6 para GraphQL.

Las dos últimas imágenes generadas por el script de análisis de datos son las de la API gRPC. En la primera (figura 5.21), se observa el comportamiento de las mismas métricas en una prueba general. Aquí se destaca la eficiencia de gRPC, especialmente en términos de latencia y la tasa de errores, mostrando una menor variabilidad en comparación con REST y GraphQL. Los gráficos también revelan cómo gRPC maneja las solicitudes con mayor estabilidad a lo largo del tiempo, manteniendo un rendimiento óptimo incluso con una carga de datos creciente.

Esta última figura (figura 5.22) ilustra el impacto de la caché Redis en la API gRPC mediante una prueba sintética. Similar a las pruebas anteriores con REST y GraphQL, la caché optimiza significativamente el rendimiento de gRPC, reduciendo la latencia y la cantidad de datos transferidos. La figura permite comparar directamente cómo la caché afecta de manera diferente a cada una de las tecnologías API.

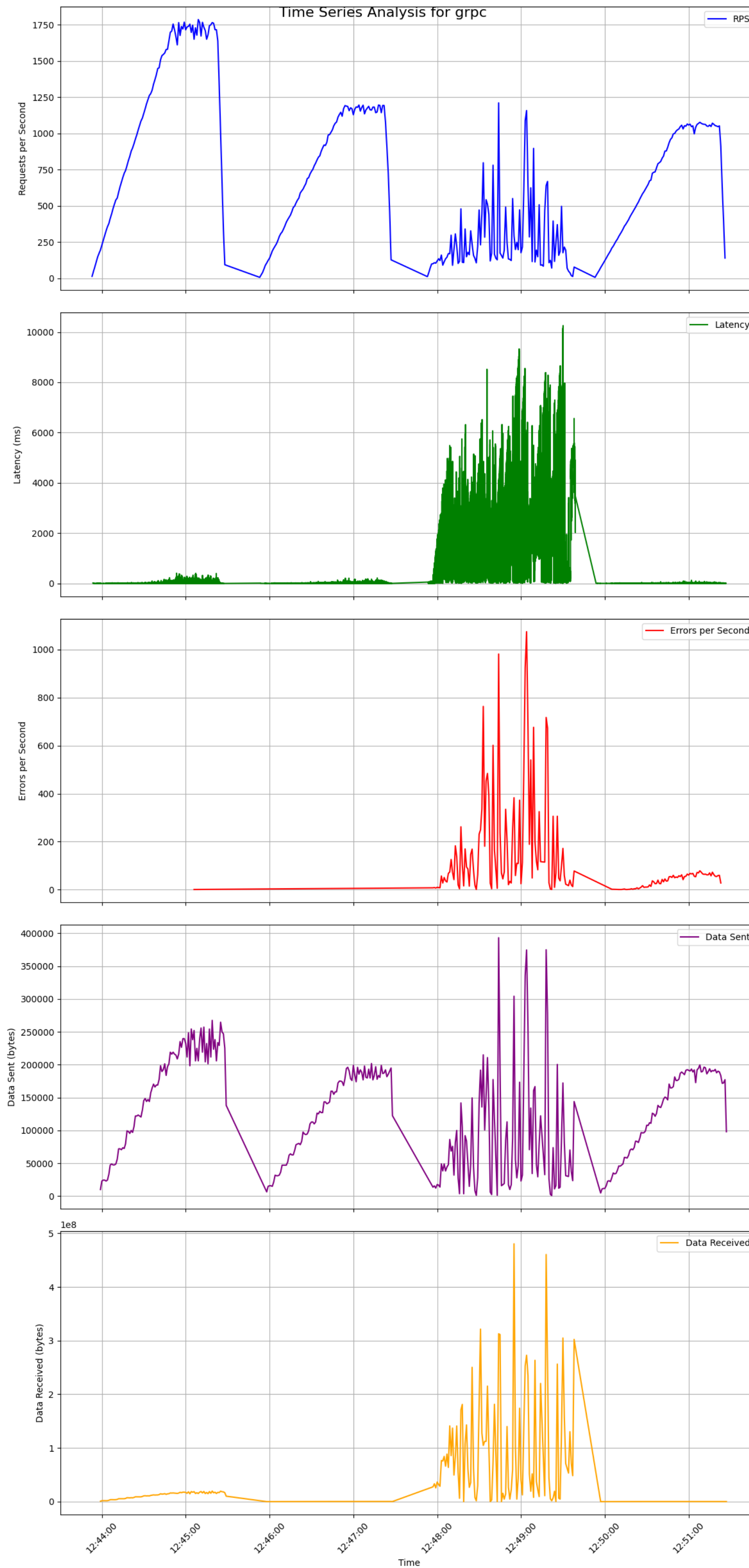


Figura 5.21 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba general de k6 para gRPC.

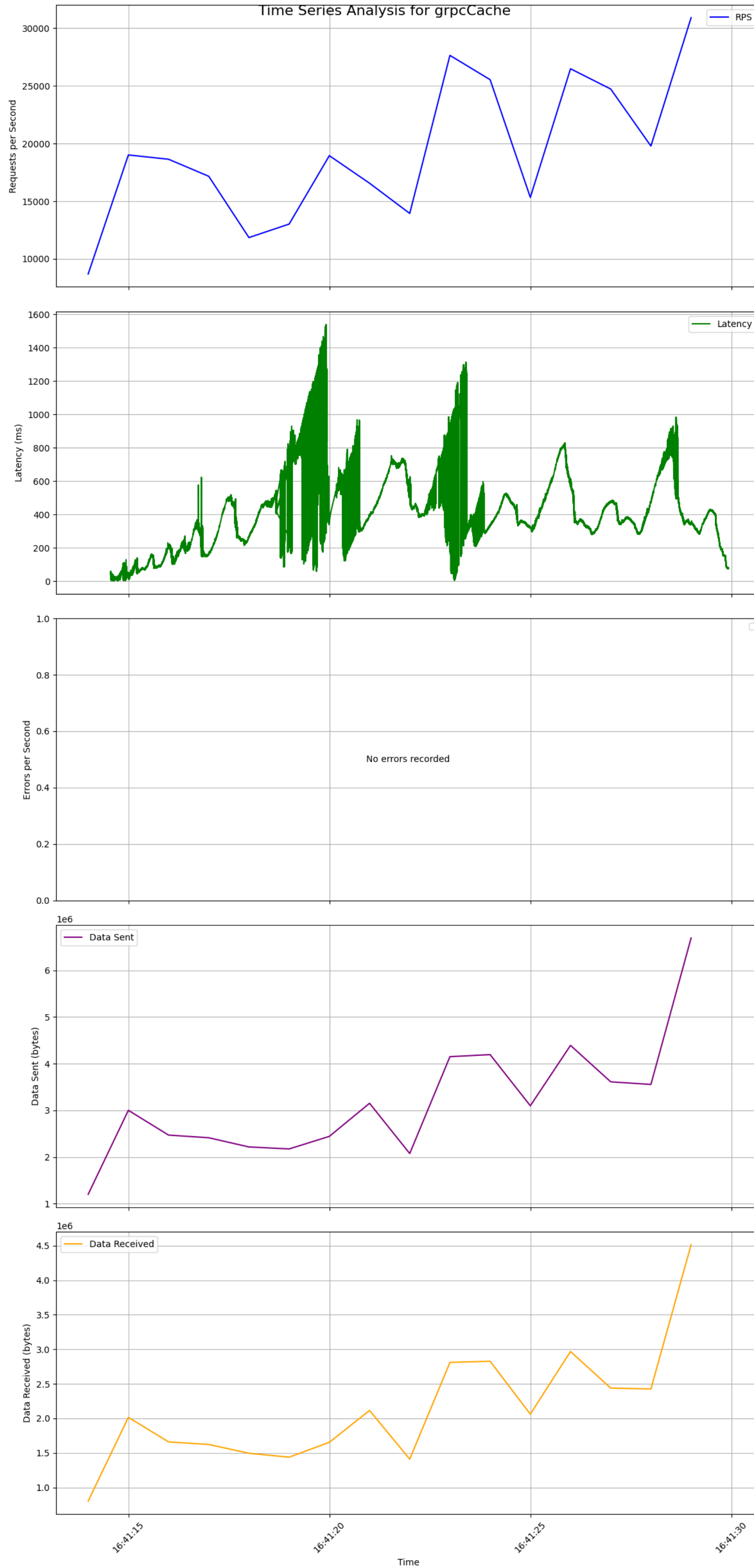


Figura 5.22 Evolución de las peticiones por segundo, latencia, tasa de errores, cantidad de datos enviados y recibidos en la prueba de caché de k6 para gRPC.

De las figuras 5.8 a la 5.15 inclusive se pueden sacar las mismas conclusiones que en el apartado 5.1. Ya que recogen los mismos datos de una forma visual. En este apartado es más interesante analizar las series temporales (figuras de la 5.15 a la 5.22) donde se puede ver el comportamiento de cada métrica según se va incrementando el número de usuarios virtuales y en cada uno de los escenarios de las pruebas generales.

5.2.1 Análisis de las Pruebas Sintéticas

Estas pruebas muestran un comportamiento casi idéntico en todas las tecnologías API.

Como en esta prueba las peticiones son siempre idénticas, la cantidad de datos enviados y recibidos siempre es directamente proporcional a la cantidad de peticiones realizadas. Por lo que las métricas de datos transferidos son irrelevantes. En ninguna prueba ocurrió un solo error. Por lo que esta métrica también puede ser ignorada.

En cuanto a latencia, se puede observar que los picos de latencia se correlacionan con los picos de peticiones por segundo de una forma bastante cercana. Con más variabilidad en la latencia y mostrando una tendencia a aumentar con el paso del tiempo, aunque las peticiones por segundo no lo hagan.

Esto podría indicar, como sería de esperar, que a más tiempo se mantenga un tráfico elevado sobre las APIs más ineficientes se vuelve la resolución de peticiones. Probablemente ya que se estaría consumiendo más memoria que no puede ser liberada lo suficientemente rápido por el recolector de basura de Go. Aunque en todos ellos la tendencia parece reducirse con el paso del tiempo.

Tanto la gráfica de peticiones por segundo de REST como de GraphQL muestran estar directamente movidas por la cantidad de tráfico actual, viendo como incrementa en los primeros 15 segundos y después se mantiene relativamente estable hasta el *ramp down* de la prueba. Pero en gRPC se muestra como durante toda la ejecución de la prueba se ha aumentado sostenidamente la cantidad de peticiones procesadas por segundo sin incrementar la latencia. Por lo que gRPC podría ser capaz de mejorar su rendimiento a más tiempo pase.

Esto no se puede concluir con certeza con estos datos, pero podría ser un caso interesante por estudiar. Quizás la implementación de gRPC haga optimizaciones a peticiones comunes de forma similar a como el motor de JavaScript V8 optimiza partes del código de JavaScript que son ejecutadas con más frecuencia [48].

Por lo general se puede concluir que para una API que requiera realizar muchas pequeñas operaciones de escritura y lectura de la forma más rápida posible, en general la mejor opción es REST. Debido a su simplicidad, facilidad de uso y de implementación y que, generalmente, no se requiere de operaciones de consulta complejas como en GraphQL. A menos que requieras un rendimiento similar, pero con tamaños de petición y respuesta más pequeños, en cuyo caso GraphQL sería mejor. Y en el caso extremo de que solo sea relevante minimizar el tamaño de las peticiones y respuestas a cualquier precio, gRPC sería la mejor opción.

5.2.2 Análisis de las Pruebas Generales

En estas pruebas sí se pueden observar diferencias considerables en cuanto al comportamiento de las APIs según se va incrementando la cantidad de peticiones concurrentes y en cada escenario de las pruebas. Antes de entrar a analizar cada caso por separado conviene detallar las partes y comportamientos comunes a todas las pruebas.

En Común

Para empezar, si se observa las gráficas de peticiones por segundo, se pueden observar una serie de trapecios con una pendiente ascendiente de izquierda a derecha, una meseta relativamente corta, una caída drástica y otra meseta antes del siguiente trapecio. Esto es ocasionado por los escenarios explicados en el apartado 4.5: Consumer, Producer, Big Consumer y Updater.

En todas las pruebas se observa que el escenario Big Consumer es el que muestra la menor y más inestable latencia de todos los escenarios. A su vez, este escenario es el que más errores causa con diferencia en todos los casos. Y es cuando más cantidad de datos son recibidos como es de esperar.

Cabe destacar que durante la ejecución de este escenario siempre se veía en los logs de los servidores una muy mayor ocurrencia de errores en la base de datos PostgreSQL, por lo que puede ser que esta estuviese haciendo cuello de botella a las aplicaciones API. Sería interesante volver a realizar estas pruebas con una base de datos mucho más veloz como Redis o almacenando la información directamente en memoria.

En todos los casos, la cantidad de datos enviados son directamente proporcionales a la cantidad de peticiones realizadas. Y también se comparte el fenómeno de actualizaciones de datos más lentas que la creación de nuevos datos.

Aunque el tamaño de las peticiones de actualización es considerablemente más pequeño, incluso en GraphQL que realiza todas las actualizaciones en una sola petición gracias a su lenguaje de consultas. Ahora se pasa a analizar las diferencias y comportamientos específicos que muestra cada prueba.

REST

En REST destaca que, junto con gRPC, tiene su mayor pico de peticiones por segundo en el escenario Consumer. Indicando que es más eficiente consumiendo información que produciéndola o actualizándola siempre que la cantidad de información no sea masiva (Big Consumer).

REST no muestra degradación de los servicios en las operaciones de actualización, como gRPC, cuando en GraphQL sí ocurre esto. También parece ser la aplicación que mejor resiste a la degradación de servicio en el caso de consultas masivas de información, siendo la que más consultas por segundo aguantó antes de aumentar la latencia y tasa de errores en torno a las 500 peticiones por segundo.

Aunque muestra una cantidad baja pero inusual de errores en el escenario Producer. Por lo que puede ser que REST sea menos capaz de procesar muchas operaciones de escritura de forma satisfactorio. Sin embargo, estos errores no aumentaron de forma apreciable el tiempo de respuesta del servicio.

GraphQL Datos Máximos

Como se ha indicado en el anterior segmento. GraphQL parece mostrar serios problemas para procesar las solicitudes de actualización de datos. Aun pudiendo realizar todas las operaciones en una sola consulta, parece que el entorno de ejecución de GraphQL sufre mucho para poder aplicar todas las actualizaciones de forma eficiente. Por lo que se muestra que la capacidad de realizar consultas complejas puede no ser una ventaja a nivel de rendimiento.

No muestra problemas a la hora de crear nuevos datos. De hecho, es prácticamente igual de eficiente creando información que consumiéndola, aunque la capacidad de creación de REST y gRPC siguen siendo superiores.

Sorprendentemente el tamaño de las peticiones y consultas no afecta significativamente a la calidad del servicio. Sería interesante realizar pruebas con consultas fuertemente anidadas, ya que en estas pruebas no se ha profundizado más allá de 2 niveles.

GraphQL Datos Mínimos

Como indicado anteriormente. Resulta equivalente en comportamiento al caso con datos máximos de su misma tecnología API.

gRPC

Teniendo un comportamiento bastante similar a REST, pero con un rendimiento generalmente inferior. Cabe destacar que, aunque el tamaño de las peticiones suele ser por lo general mayor que REST (y menor que GraphQL), muestra un comportamiento inusual a la hora de crear entidades. Ya que, al contrario que REST, donde la cantidad de datos enviados aumenta (aun cuando se reducen las peticiones por segundo), en gRPC la cantidad de datos enviados se reduce sin que se reduzca más las peticiones por segundo que REST. Esto parece indicar que, para el envío de grandes cantidades de datos, gRPC es considerablemente más eficiente que REST en el uso del ancho de banda.

5.2.3 Resumen

El análisis de las pruebas generales muestra que las tres tecnologías API (REST, GraphQL y gRPC) presentan comportamientos diferenciados según el escenario de uso. REST destaca en su capacidad para manejar operaciones de lectura, mostrando un buen rendimiento y resistencia frente a la degradación del servicio, incluso bajo cargas altas. Sin embargo, REST tiene una mayor tasa de errores en escenarios de escritura intensiva, lo que podría indicar una menor eficiencia en la gestión de operaciones de escritura.

GraphQL, por su parte, presenta un desempeño mixto. Aunque es eficiente en la creación de nuevos datos y en la ejecución de consultas más pequeñas, las pruebas demuestran que su capacidad de actualización se ve afectada negativamente en comparación con las otras tecnologías. Esta ineficiencia sugiere que la ventaja de poder realizar consultas complejas y estructuradas no siempre se traduce en un mejor rendimiento, especialmente en escenarios donde las actualizaciones de datos son frecuentes.

gRPC, aunque similar a REST en muchos aspectos, muestra una mayor eficiencia en la transferencia de grandes volúmenes de datos. En particular, destaca por su capacidad para manejar transmisiones de datos más optimizadas, lo que lo convierte en una opción interesante para escenarios en los que la eficiencia del uso del ancho de banda es prioritaria. Sin embargo, su rendimiento

en operaciones de escritura es algo inferior a REST, lo que sugiere que podría no ser la mejor opción en todos los casos.

En resumen, REST sigue siendo una opción sólida para aplicaciones generales que requieren de buen rendimiento en la mayoría de los escenarios. GraphQL es útil cuando se requiere flexibilidad en las consultas, pero presenta dificultades en la actualización de datos a gran escala. gRPC, aunque es eficiente en la transmisión de datos, necesita mejorar su rendimiento en operaciones de escritura.

5.3 Hallazgos

Las pruebas y análisis realizados en este capítulo demuestran que cada una de las tecnologías API evaluadas tiene sus propias fortalezas y debilidades, lo que las hace más o menos adecuadas para diferentes tipos de aplicaciones y casos de uso.

REST destaca por su simplicidad, rendimiento equilibrado y escalabilidad, lo que lo convierte en una opción versátil para aplicaciones web convencionales que no requieren optimizaciones extremas en la transmisión de datos. Sin embargo, su alta tasa de errores en escenarios de escritura intensiva sugiere que puede no ser la opción ideal cuando las operaciones de escritura son críticas.

GraphQL se posiciona como una tecnología potente para escenarios donde se requiere la personalización de las consultas y la reducción del overfetching. No obstante, sus problemas en la actualización de datos y su latencia relativamente alta en comparación con REST y gRPC limitan su aplicabilidad en entornos donde el rendimiento es crucial.

gRPC, por su parte, destaca por su eficiencia en la transferencia de datos y baja latencia, haciéndolo ideal para entornos de microservicios o aplicaciones donde el ancho de banda es un recurso limitado. No obstante, su rendimiento en operaciones de escritura y su mayor complejidad de implementación en comparación con REST lo hacen menos accesible para algunos desarrolladores.

En conclusión, la elección de la tecnología API más adecuada dependerá en gran medida de las necesidades específicas del proyecto. REST sigue siendo la opción más equilibrada para la mayoría de los casos, mientras que GraphQL y gRPC se destacan en nichos específicos, donde sus características particulares ofrecen ventajas significativas.

6

Conclusiones y Líneas Futuras

Este capítulo final resume e interpreta los resultados del estudio, resaltando los hallazgos más importantes y examinando sus implicaciones. Asimismo, se proponen posibles direcciones para futuras investigaciones que podrían ampliar o complementar el trabajo realizado.

6.1 Conclusiones

El presente trabajo ha comparado exhaustivamente tres tecnologías de APIs ampliamente utilizadas en la industria: REST, GraphQL y gRPC. Cada una de estas tecnologías presenta características únicas que las hacen adecuadas para diferentes contextos y necesidades.

REST sigue destacándose como una solución versátil y equilibrada. Su simplicidad y compatibilidad con múltiples plataformas lo convierten en una opción ideal para aplicaciones web tradicionales. Aunque presenta una tasa de

errores ligeramente superior en escenarios de alta carga de escritura, su capacidad de manejar operaciones de lectura lo posiciona como una tecnología robusta para la mayoría de los proyectos.

Por otro lado, GraphQL ha demostrado ser muy eficiente en la personalización de consultas, minimizando la sobrecarga de datos (overfetching). Sin embargo, esta ventaja se ve limitada por una mayor latencia y dificultades en la actualización de datos masivos, lo que lo hace menos eficiente en escenarios donde el rendimiento es crítico.

Finalmente, gRPC se presenta como la opción más eficiente en términos de latencia y transferencia de datos, especialmente en entornos de microservicios o aplicaciones que requieren optimización del ancho de banda. A pesar de su complejidad de implementación, gRPC ha demostrado ser extremadamente eficaz en la transmisión de grandes volúmenes de datos, aunque su rendimiento en operaciones de escritura podría mejorar.

A partir de los resultados, se ha completado parcialmente la rúbrica de evaluación propuesta en el apartado 2.2.3. Las métricas evaluadas con datos empíricos y teóricos se muestran a continuación, incluyendo un asterisco (*) en aquellas métricas en las que el análisis se ha basado principalmente en la experiencia de desarrollo y con dos asteríscos (**) aquellos basados en la revisión teórica.

Criterio	***Métrica***	***REST***	***GraphQL***	***gRPC***
Rendimiento	**Latencia**	5	2	3
	Tasa de transferencia efectiva (*throughput*)	5	2	3
	Utilización de Recursos	5	2	3
	Manejo de Concurrencia	5	2	4
	Tasas de Error	3	2	5
	Uso de Ancho de Banda	5	3	4
Escalabilidad	**Eficiencia de Escalado Horizontal**	-	-	-
	Eficiencia de Escalado Vertical	-	-	-
	Compatibilidad con Balanceo de Carga	-	-	-
	Tolerancia a Fallos	5	3	4
	Manejo de Sesiones	-	-	-
	Mecanismos de Caché	-	-	-
Facilidad de Uso	**Curva de Aprendizaje**	5*	1*	3*
	Herramientas y Ecosistema	5*	3*	2*
	Complejidad de Diseño de API	5*	2*	4*
	Documentación y Descubribilidad	3*	4*	4*
	Implementación del Lado Cliente	-	-	-
	Pruebas y Depuración	5*	2*	3*
Flexibilidad	**Capacidades de Consulta**	2**	5**	2**
	Evolución del Esquema y Versionado	5**	4**	3**

	Soporte para Comunicación en Tiempo Real	3**	1**	5**
	Flexibilidad en el Manejo de Errores	5**	3**	3**
	Autenticación y Autorización	-	-	-
	Paginación y Agrupación	-	-	-
Seguridad	**Seguridad en el Transporte**	4**	4**	5**
	Mecanismos de Autenticación	-	-	-
	Controles de Autorización	-	-	-
	Validación y Saneamiento de Datos	-	-	-
	Registro de Auditoría y Monitoreo	-	-	-
	Resistencia a Ataques Comunes	4**	3**	5**
Compatibilidad e Interoperabilidad	**Soporte Multilenguaje**	5**	4**	3**
	Soporte Multiplataforma	5**	5**	3**
	Integración con Sistemas Legados	5**	4**	3**
	Integración con Servicios de Terceros	5**	4**	2**
	Soporte para Proxies y Gateways API	5**	5**	2**
	Soporte para Clientes Móviles y Web	5**	5**	2**
Capacidad de Tipado y Gestión de Esquemas	**Soporte para Tipado Fuerte**	0**	3**	5**
	Definición y Evolución de Esquemas	0**	5**	3**
	Mecanismos de Validación	0**	3**	5**
	Generación de Documentación	-	-	-
	Introspección de Esquemas	0**	5**	2**
	Seguridad en Tipado en Tiempo de Compilación	0**	1**	5**

En conclusión, la elección de la tecnología API más adecuada depende del contexto específico del proyecto. REST sigue siendo la opción más balanceada para aplicaciones generales, mientras que GraphQL y gRPC son más apropiadas para nichos específicos donde sus características particulares ofrecen ventajas significativas.

6.2 Líneas Futuras

Este estudio abre varias líneas futuras de investigación y desarrollo. En primer lugar, sería interesante explorar cómo se comportan estas tecnologías en un entorno distribuido a gran escala, donde se incluyan elementos como la replicación de bases de datos y el manejo de datos en tiempo real. Además, se podría investigar más a fondo cómo estas APIs se desempeñan en aplicaciones móviles y entornos con restricciones de ancho de banda, ya que gRPC podría tener un rendimiento superior en estas áreas.

Otra línea de investigación podría centrarse en la mejora de la eficiencia de GraphQL en la actualización de datos y la reducción de su latencia. El estudio de nuevas técnicas o algoritmos que optimicen estas operaciones podría hacer que GraphQL sea más competitivo en entornos donde el rendimiento es esencial.

También hay espectos cubiertos en este estudio que se podrían profundizar o mejorar para tener conclusiones más claras y confiables al respecto. Principalmente investigar la posible mejora de rendimiento de gRPC en entornos de prubeas sintéticos de alta concurrencia con mensajes de tamaño mínimo como se apreció en el apartado 5.2.1, reemplazar la base de datos PostgreSQL por una más rápida o por mantener los datos en memoria para evitar posibles cuellos de botella como se vió en la sección 5.2.2.1, y realizar consultas con alta anidación para comprobar el rendimiento comparativo de GraphQL como se adelantó en el punto 5.2.2.3.

Finalmente, sería valioso analizar el impacto de estas tecnologías en términos de seguridad, facilidad de mantenimiento y adopción por parte de la comunidad de desarrolladores. Esto permitiría obtener una visión más completa y holística sobre cuándo y cómo implementar estas soluciones en distintos proyectos.

Referencias

[1] Jin, B., Shevat, A., & Sahni, S. (2018). Chapter 1 What's an API? In Designing web apis: Building apis that developers love. (p. 1). O'Reilly Media.

[2] State of API economy 2021 report. (p. 6). https://www.itp.net/public/Apigee_StateOfAPIS_eBook_2020_0.pdf

[3] Lane, K. (2024, July 10). Intro to apis: History of apis. Postman Blog. https://blog.postman.com/intro-to-apis-history-of-apis/

[4] 2023 state of the API report: API Technologies. Postman API Platform. (n.d.). https://www.postman.com/state-of-api/api-technologies/#architectural-style-soap-slips

[5] Kamiński, L., Kozłowski, M., Sporysz, D., Wolska, K., Zaniewski, P., & Roszczyk, R. (2023). Comparative review of selected internet communication protocols. Foundations of Computing and Decision Sciences, 48(1), 39–56. https://doi.org/10.2478/fcds-2023-0003

[6] Niswar, M., Arisandy Safruddin, R., Bustamin, A., & Aswad, I. (2024). Performance evaluation of microservices communication with rest, GraphQL, and grpc. International Journal of Electronics and Telecommunications, 429–436. https://doi.org/10.24425/ijet.2024.149562

[7] Śliwa, M., & Pańczyk, B. (2021). Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. Journal of Computer Sciences Institute, 21, 356–361. https://doi.org/10.35784/jcsi.2744

[8] Postman api platform. Postman API Platform. (n.d.-b). https://www.postman.com/

[9] Grafana K6: Grafana K6 documentation. Grafana Labs. (n.d.). https://grafana.com/docs/k6/latest/

[10] Fielding, R. T. (2000). (thesis). Architectural styles and the design of network-based software architectures. (p. 76) .

[11] Fielding, R. T. (2000). (thesis). Architectural styles and the design of network-based software architectures. (p. 88) .

[12] Fielding, R. T. (2000). (thesis). Architectural styles and the design of network-based software architectures. (p. 90) .

[13] Fielding, R. T. (2000). (thesis). Architectural styles and the design of network-based software architectures. (p. 91) .

[14] Fielding, R. T. (2000). (thesis). Architectural styles and the design of network-based software architectures. (pp. 78-79) .

[15] Fielding, R. T. (2000). (thesis). Architectural styles and the design of network-based software architectures. (p. 100) .
[16] A data query language. GraphQL. (n.d.). https://graphql.org/blog/2015-09-14-graphql/#:~:text=When%20we%20built,graphql.org.
[17] Introduction to graphql. GraphQL. (n.d.-b). https://graphql.org/learn/
[18] Serving over HTTP. GraphQL. (n.d.-c). https://graphql.org/learn/serving-over-http/
[19] Porcello, E., & Banks, A. (2018). Learning graphql: Declarative data fetching for modern web apps. O'Reilly. (p. 123)
[20] Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference (WWW '18). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1155–1164. https://doi.org/10.1145/3178876.3186014
[21] Wittern, E., Cha, A., Davis, J.C., Baudart, G., Mandel, L. (2019). An Empirical Study of GraphQL Schemas. In: Yangui, S., Bouassida Rodriguez, I., Drira, K., Tari, Z. (eds) Service-Oriented Computing. ICSOC 2019. Lecture Notes in Computer Science(), vol 11895. Springer, Cham. https://doi.org/10.1007/978-3-030-33702-5_1
[22] Graphql. (n.d.). Graphiql/packages/graphiql at main · GRAPHQL/graphiql. GitHub. https://github.com/graphql/graphiql/tree/main/packages/graphiql#readme
[23] Introspection. GraphQL. (n.d.-c). https://graphql.org/learn/introspection
[24] FAQ. gRPC. (2023, September 13). https://grpc.io/docs/what-is-grpc/faq
[25] Ryan, L. (2015, September 8). GRPC motivation and design principles. gRPC. https://grpc.io/blog/principles/
[26] Introduction to grpc. gRPC. (2024, July 25). https://grpc.io/docs/what-is-grpc/introduction/
[27] Overview. Protocol Buffers Documentation. (n.d.). https://protobuf.dev/overview/
[28] Indrasiri, K., & Kuruppu, D. (2020). GRPC: Up and running: Building cloud native applications with go and Java for Docker and Kubernetes. O'Reilly Media, Inc. (pp. 4)
[29] Language Guide (proto 3). Protocol Buffers Documentation. (n.d.-a). https://protobuf.dev/programming-guides/proto3/
[30] Indrasiri, K., & Kuruppu, D. (2020). GRPC: Up and running: Building cloud native applications with go and Java for Docker and Kubernetes. O'Reilly Media, Inc. (pp. 65)
[31] Core Concepts, architecture and Lifecycle. gRPC. (2024a, July 25). https://grpc.io/docs/what-is-grpc/core-concepts/

[32] Indrasiri, K., & Kuruppu, D. (2020). GRPC: Up and running: Building cloud native applications with go and Java for Docker and Kubernetes. O'Reilly Media, Inc. (pp. 62)

[33] Indrasiri, K., & Kuruppu, D. (2020). GRPC: Up and running: Building cloud native applications with go and Java for Docker and Kubernetes. O'Reilly Media, Inc. (pp. 77)

[34] Nally, M. (n.d.). GRPC vs rest: Understanding grpc, openapi and rest and when to use them in API design | google cloud blog. Google. https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them

[35] Sang Gyun Du, Jong Won Lee, and Keecheon Kim. 2018. Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN. In Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication (IMCOM '18). Association for Computing Machinery, New York, NY, USA, Article 68, 1–6. https://doi.org/10.1145/3164541.3164563

[36] Gonzalez, I. (2020). Building microservice apis using GRPC (thesis). Building microservice APIs using GRPC. California State University, Northridge, Northridge, CA.

[37] Ali, O. (2024). Popular API Technologies: REST, GraphQL, and gRPC.

[38] Mikuła, M., & Dzieńkowski, M. (2020). Comparison of rest and GraphQL Web Technology Performance. Journal of Computer Sciences Institute, 16, 309–316. https://doi.org/10.35784/jcsi.2077

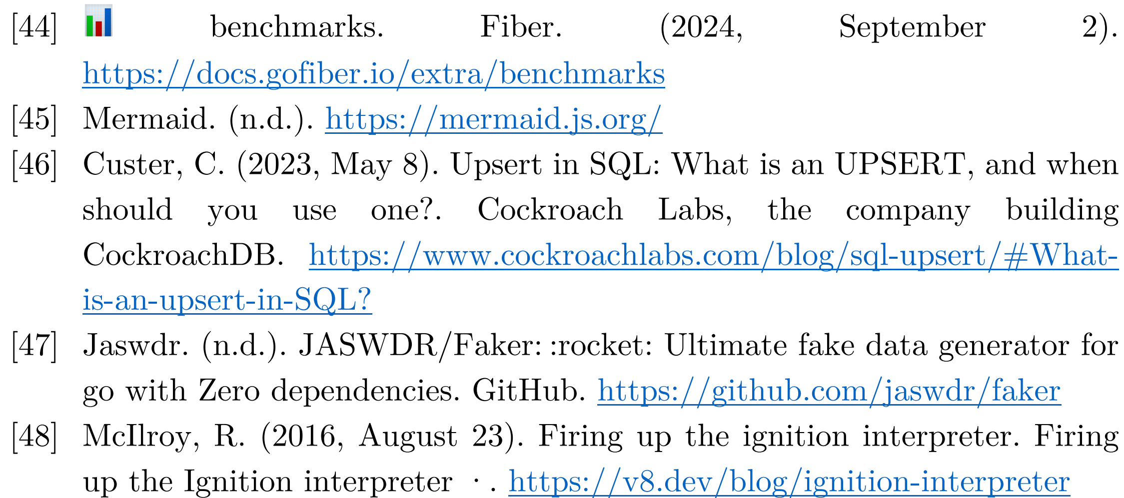
[39] Frigård, E. (2022). GraphQL vs. REST : A Comparison of Runtime Performance (Dissertation). Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-113917

[40] Nilsson, E., & Demir, D. (2023). Performance comparison of REST vs GraphQL in different web environments : Node.js and Python (Dissertation). Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-121877

[41] Seabra, M., Nazário, M. F., & Pinto, G. (2019). Rest or graphql? Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse. https://doi.org/10.1145/3357141.3357149

[42] Berg, J., & Mebrahtu Redi, D. (2023). Benchmarking the request throughput of conventional API calls and gRPC : A Comparative Study of REST and gRPC (Dissertation, KTH Royal Institute of Technology). Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-334990

[43] Johansson, M., & Isabella, O. (2023). Comparative Study of REST and gRPC for Microservices in Established Software Architectures (Dissertation). Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-195563

[44]  benchmarks. Fiber. (2024, September 2). https://docs.gofiber.io/extra/benchmarks

[45] Mermaid. (n.d.). https://mermaid.js.org/

[46] Custer, C. (2023, May 8). Upsert in SQL: What is an UPSERT, and when should you use one?. Cockroach Labs, the company building CockroachDB. https://www.cockroachlabs.com/blog/sql-upsert/#What-is-an-upsert-in-SQL?

[47] Jaswdr. (n.d.). JASWDR/Faker: :rocket: Ultimate fake data generator for go with Zero dependencies. GitHub. https://github.com/jaswdr/faker

[48] McIlroy, R. (2016, August 23). Firing up the ignition interpreter. Firing up the Ignition interpreter · . https://v8.dev/blog/ignition-interpreter

Apéndice A

Código Fuente, Instalación y Uso

A.1 Código Fuente:

Todo el código desarrollado a lo largo de este trabajo, los archivos generados por este código y cualquier otro documento se puede consultar en el repositorio público de GitHub: https://github.com/n4xo-dev/api-wars.

A.2 Requerimientos:

Para poder ejecutar todos los programas y pruebas creados en este proyecto es necesario tener cierto software instalado. A continuación, se listan todos los programas necesarios junto a enlaces donde encontrar instrucciones para instalarlos:

- Git: https://git-scm.com/downloads.
- Go: https://go.dev/doc/install.
- Python 3: https://wiki.python.org/moin/BeginnersGuide/Download. A su vez, el script de Python requiere de ciertas dependencias para funcionar debidamente. A continuación, se listan estas dependencias con el comando de pip para instalarlas:
  - Matplotlib: `pip install matplotlib`.
  - Numpy: `pip install numpy`.
- Grafana k6: https://grafana.com/docs/k6/latest/set-up/install-k6/.
- WSL (Windows Subsystem for Linux): https://learn.microsoft.com/en-us/windows/wsl/install. Necesario para ejecutar los scripts de Bash en Windows. Aunque no es necesario el uso de estos scripts, ya que se pueden ejecutar los comandos que automatizan de forma manual.
- Docker desktop: https://www.docker.com/products/docker-desktop/.
- Postman (opcional): https://www.postman.com/downloads/.

A.3 Obtener el código:

Si no se tiene el código fuente de este proyecto se puede obtener clonando su repositorio de GitHub en https://github.com/n4xo-dev/api-wars. Esto se puede hacer con cualquiera de los dos siguientes comandos en la terminal de comandos:

- `git clone https://github.com/n4xo-dev/api-wars.git`
- `git clone git@github.com:n4xo-dev/api-wars.git`

Recuerde ejecutar el comando desde la carpeta en la que quiera clonar el repositorio.

A.4 Ejecutar los servidores API:

Para ejecutar los servidores antes hace falta configurar las variables de entorno. Para esto se proveen de archivos *.sample.env* que pueden usarse de referencia. Simplemente modifique cualquiera de estos archivos parar que tenga la configuración deseada y cámbieles el nombre a *.env* o copie su contenido en un archivo *.env*.

Incluso aunque usted no quiera cambiar la configuración por defecto, asegúrese que tiene un archivo *.env* válido en la carpeta desde la que ejecute los programas Go.

También es necesario tener instaladas y ejecutándose todos los programas de los que los servidores dependen. Siendo estos PostgreSQL y Redis. Para facilitar esto, en la carpeta raíz del repositorio se encuentra un archivo *docker-compose.yml*. Con este documento puede ejecutar estas bases de datos contenerizadas en Docker. Para esto asegúrese de tener Docker Desktop abierto y simplemente ejecute el siguiente comando el la terminal desde la misma carpeta raíz del repositorio:

- `docker compose up -d`

Con esto ya tendrá en Docker PostgreSQL, Redis y pgAdmin (para poder consultar y gestionar la base de datos con una GUI agradable) corriendo.

Ahora solo hace falta compilar y ejecutar el archivo principal *cmd/main.*go del servidor deseado. Para esto puede primero compilar el código a un binario y luego ejecutar este binario, o puede ejecutar el programa directamente compilandolo a un archivo temporal con el comando `go run`. Si tiene pensado ejecutar varias veces

el mismo programa y prefiere ahorrarse el tiempo de compilación haga lo siguiente:

1. Compile el código deseado. Suponiendo que se está en la carpeta raíz del repositorio y se quiere ejecutar el servidor de API REST sería:
   a. `git build rest/cmd/main.go -o bin`
2. Ahora ejecute el binario creado:
   a. En Linux y Mac: `./bin -server`
   b. En Windows: `./bin.exe -server`

Si se quiere ejecutar el código sin compilación y no es relevante esperar unos segundos más para que se inicie:

- `git run rest/cmd/main.go -server`

Si se quieren ejecutar las pruebas unitarias o resetear la base de datos se pasan las opciones de terminal respectivas. Por ejemplo:

- `git run rest/cmd/main.go -tests users -reset`

A.5 Ejecutar las pruebas k6:

Para ejecutar las pruebas individualmente simplemente se ha de ejecutar el script k6 deseado usando la herramienta de terminal de k6. Si se quiere información detallada de cómo usarla puede ir aquí: https://grafana.com/docs/k6/latest/get-started/running-k6/. Es necesario que esté ejecutando el servidor sobre el que se va a hacer la prueba.

Si se quiere ejecutar todas las pruebas se recomienda el uso del script *runTests.sh*. Se encuentra en la carpeta raíz del repositorio y se explica cómo funciona en el punto 4.5.

Es necesario usar un sistema operativo basado en UNIX o WSL en Windows. Parar ejecutar este script simplemente haga:

- `bash runTests.sh`

Asegúrese de tener todos los servidores APIs apagados para evitar conflictos.

A.6 Ejecutar el script Python de análisis :

Para ejecutar este script simplemente hay que ubicarse en la carpeta */tests* del respositorio y ejectuar:

- `python analysis.py` o `python3 analysis.py`

Es necesario haber ejecutado todas las prubas de k6 antes para tener los datos que analizar.

Apéndice C

Evolución del Diseño del Proyecto

El diseño inicial del proyecto, presentado en el anteproyecto, proponía un enfoque general para la comparación de tecnologías API: REST, GraphQL y gRPC. Se planteaba el desarrollo de tres servidores con las tecnologías mencionadas se planificaba la implementación de una aplicación CRUD para cada una de ellas, junto con pruebas de carga y análisis de rendimiento para medir métricas clave como la latencia y escalabilidad.

Cambios en el Diseño Final

El diseño final del proyecto mantuvo los objetivos principales del anteproyecto, pero con varias modificaciones clave para asegurar una implementación más robusta y ajustada a las necesidades del estudio comparativo:

1. **Complejidad y Alcance Ampliado**: En el diseño inicial, el proyecto solo contemplaba una estructura simple de API CRUD. Sin embargo, durante el desarrollo, se concluyó que una API más compleja y realista, que simulara una red social con varias entidades interrelacionadas (usuarios, publicaciones, comentarios, mensajes y chats), proporcionaría mejores datos para la comparación . Este cambio permitió evaluar más exhaustivamente las fortalezas y debilidades de cada tecnología, no solo en operaciones CRUD simples sino también en escenarios más demandantes.
2. **Añadido de PostgreSQL como Base de Datos de Soporte**: Inicialmente, el diseño no mencionaba el uso de una base de datos relacional. No obstante, en el diseño final se decidió incluir PostgreSQL junto a Redis, ya que permite simular escenarios más realistas donde las APIs interactuan con grandes bases de datos SQL.

3. **Automatización de Pruebas y Scripts**: Aunque en el anteproyecto se proponía el uso de Postman para pruebas de integración y Python para las pruebas de carga y scripts de análisis, en la versión final se optó por usar Postman para pruebas sencillas de extremo a extremo, k6 para las pruebas de carga y Python para generar análisis comparativos de rendimiento. Principalmente ya que Postman no aportaba herramientas robustas para extraer información detallada y relevante, mientras que el rendimiento de Python no llegaba a ser suficiente para llegar a saturar los servidores desarrollados con Go.

En resumen, el diseño final del proyecto amplió significativamente la complejidad y el alcance respecto al plan inicial, y eliminó partes accesorias que no aportaban al objetivo del estudio. Permitiendo una evaluación más rica y detallada de las tecnologías API en escenarios de uso más complejos y realistas manteniéndose relativamente dentro del marco temporal previsto.