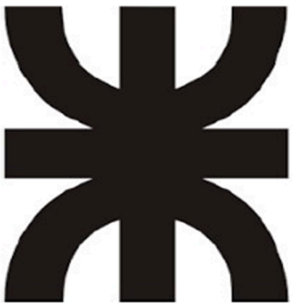
UNIVERSIDAD TECNOLÓGICA NACIONAL



FACULTAD REGIONAL ROSARIO

INFORME - POC Open API/REST/GRAPHQL

2025

ISI - 3K02

MATERIA: Desarrollo de Software

PROFESORES:

- Mario Bressano
- Andrés Otaduy

ALUMNOS:

- Lurati Ignacio 52688
- Olivieri Luca 53001
- Rivero Tomás 51070

Introducción

Qué es una API

Las API son mecanismos que permiten a dos componentes de software comunicarse entre sí sin necesidad de saber cómo están implementados mediante un conjunto de definiciones y protocolos

API significa “interfaz de programación de aplicaciones”. En el contexto de las API, la palabra aplicación se refiere a cualquier software con una función distinta. La interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato define cómo se comunican entre sí mediante solicitudes y respuestas. La documentación de su API contiene información sobre cómo los desarrolladores deben estructurar esas solicitudes y respuestas.

La arquitectura de las API suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor.

Las API pueden funcionar de cuatro maneras diferentes, según el momento y el motivo de su creación.

Qué es REST

La transferencia de estado representacional (REST) es una arquitectura de software que impone condiciones sobre cómo debe funcionar una API.

Es posible utilizar una arquitectura basada en REST para admitir comunicaciones confiables y de alto rendimiento a escala. Puede implementarla y modificarla fácilmente, lo que brinda visibilidad y portabilidad entre plataformas a cualquier sistema de API.

Los desarrolladores de API pueden diseñar API por medio de varias arquitecturas diferentes. Las API que siguen el estilo arquitectónico de REST se llaman API REST. Los servicios web que implementan una arquitectura de REST son llamados servicios web RESTful. El término API RESTful suele referirse a las API web RESTful. Sin embargo, los términos API REST y API RESTful se pueden utilizar de forma intercambiable.

Principios del estilo arquitectónico de REST:

La interfaz uniforme es fundamental para el diseño de cualquier servicio web RESTful. Ella indica que el servidor transfiere información en un formato estándar. El

recurso formateado se denomina representación en REST. Este formato puede ser diferente de la representación interna del recurso en la aplicación del servidor.

La interfaz uniforme impone cuatro limitaciones de arquitectura:

1. Las solicitudes deben identificar los recursos. Lo hacen mediante el uso de un identificador uniforme de recursos.
2. Los clientes tienen información suficiente en la representación del recurso como para modificarlo o eliminarlo si lo desean. El servidor cumple esta condición por medio del envío de los metadatos que describen el recurso con mayor detalle.
3. Los clientes reciben información sobre cómo seguir procesando la representación. El servidor logra esto enviando mensajes autodescriptivos que contienen metadatos sobre cómo el cliente puede utilizarlos de mejor manera.
4. Los clientes reciben información sobre todos los demás recursos relacionados que necesitan para completar una tarea. El servidor logra esto enviando hipervínculos en la representación para que los clientes puedan descubrir dinámicamente más recursos.

Tecnología sin estado

En la arquitectura de REST, la tecnología sin estado se refiere a un método de comunicación en el cual el servidor completa todas las solicitudes del cliente independientemente de todas las solicitudes anteriores. Los clientes pueden solicitar recursos en cualquier orden, y todas las solicitudes son sin estado o están aisladas del resto. Esta limitación del diseño de la API REST implica que el servidor puede comprender y cumplir por completo la solicitud todas las veces.

Sistema por capas

En una arquitectura de sistema por capas, el cliente puede conectarse con otros intermediarios autorizados entre el cliente y el servidor y todavía recibirá respuestas del servidor. Los servidores también pueden pasar las solicitudes a otros servidores. Es posible diseñar el servicio web RESTful para que se ejecute en varios servidores con múltiples capas, como la seguridad, la aplicación y la lógica empresarial, que trabajan juntas para cumplir las solicitudes de los clientes. Estas capas se mantienen invisibles para el cliente.

Almacenamiento en caché

Los servicios web RESTful admiten el almacenamiento en caché, que es el proceso de almacenar algunas respuestas en la memoria caché del cliente o de un intermediario para mejorar el tiempo de respuesta del servidor.

Código bajo demanda

En el estilo de arquitectura de REST, los servidores pueden extender o personalizar temporalmente la funcionalidad del cliente transfiriendo a este el código de programación del software. Por ejemplo, cuando completa un formulario de inscripción en cualquier sitio web, su navegador resalta de inmediato cualquier error que usted comete, como un número de teléfono incorrecto. El navegador puede hacer esto gracias al código enviado por el servidor.

Qué es una API REST

La API RESTful es una interfaz que dos sistemas de computación utilizan para intercambiar información de manera segura a través de Internet. Es una API que aplica construida en base a la arquitectura REST.

La función básica de una API RESTful es la misma que navegar por Internet. Cuando requiere un recurso, el cliente se pone en contacto con el servidor mediante la API. Los desarrolladores de API explican cómo el cliente debe utilizar la API REST en la documentación de la API de la aplicación del servidor.

A continuación, se indican los **pasos generales para cualquier llamada a la API REST:**

1. El cliente envía una solicitud al servidor. El cliente sigue la documentación de la API para dar formato a la solicitud de una manera que el servidor comprenda.
2. El servidor autentica al cliente y confirma que este tiene el derecho de hacer dicha solicitud.
3. El servidor recibe la solicitud y la procesa internamente.
4. Luego, devuelve una respuesta al cliente. Esta respuesta contiene información que dice al cliente si la solicitud se procesó de manera correcta. La respuesta también incluye cualquier información que el cliente haya solicitado.

¿Qué contiene la solicitud del cliente a la API REST?

Las API RESTful requieren que las solicitudes contengan los siguientes componentes principales:

Identificador único de recursos

El servidor identifica cada recurso con identificadores únicos de recursos. En los servicios REST, el servidor por lo general identifica los recursos mediante el uso de un localizador uniforme de recursos (URL). El URL especifica la ruta hacia el recurso. Un URL es similar a la dirección de un sitio web que se ingresa al navegador para visitar cualquier página web. El URL también se denomina punto de conexión de la solicitud y especifica con claridad al servidor qué requiere el cliente.

Método

Los desarrolladores a menudo implementan API RESTful mediante el uso del protocolo de transferencia de hipertexto (HTTP). Un método de HTTP informa al servidor lo que debe hacer con el recurso. A continuación, se indican cuatro métodos de HTTP comunes:

GET

Los clientes utilizan GET para acceder a los recursos que están ubicados en el URL especificado en el servidor. Pueden almacenar en caché las solicitudes GET y enviar parámetros en la solicitud de la API RESTful para indicar al servidor que filtre los datos antes de enviarlos.

POST

Los clientes usan POST para enviar datos al servidor. Incluyen la representación de los datos con la solicitud.

PUT

Los clientes utilizan PUT para actualizar los recursos existentes en el servidor.

DELETE

Los clientes utilizan la solicitud DELETE para eliminar el recurso. Una solicitud DELETE puede cambiar el estado del servidor. Sin embargo, si el usuario no cuenta con la autenticación adecuada, la solicitud fallará.

Encabezados de HTTP

Los encabezados de solicitudes son los metadatos que se intercambian entre el cliente y el servidor. Por ejemplo, el encabezado de la solicitud indica el formato de la solicitud y la respuesta, proporciona información sobre el estado de la solicitud, etc.

Datos

Las solicitudes de la API REST pueden incluir datos para que los métodos POST, PUT y otros métodos HTTP funcionen de manera correcta.

Parámetros

Las solicitudes de la API RESTful pueden incluir parámetros que brindan al servidor más detalles sobre lo que se debe hacer. A continuación, se indican algunos tipos de parámetros diferentes:

- Los parámetros de ruta especifican los detalles del URL.

- Los parámetros de consulta solicitan más información acerca del recurso.
- Los parámetros de cookie autentican a los clientes con rapidez.

¿Qué contiene la respuesta del servidor de la API RESTful?

Los principios de REST requieren que la respuesta del servidor contenga los siguientes componentes principales:

Línea de estado

La línea de estado contiene un código de estado de tres dígitos que comunica si la solicitud se procesó de manera correcta o dio error. Por ejemplo, los códigos 2XX indican el procesamiento correcto, pero los códigos 4XX y 5XX indican errores. Los códigos 3XX indican la redirección de URL.

Cuerpo del mensaje

El cuerpo de la respuesta contiene la representación del recurso. El servidor selecciona un formato de representación adecuado en función de lo que contienen los encabezados de la solicitud. Los clientes pueden solicitar información en los formatos XML o JSON, lo que define cómo se escriben los datos en texto sin formato.

Encabezados

La respuesta también contiene encabezados o metadatos acerca de la respuesta. Estos brindan más contexto sobre la respuesta e incluyen información como el servidor, la codificación, la fecha y el tipo de contenido.

Open API

¿Qué es Open API?OpenAPI es un estándar para la descripción de las interfaces de programación(API). La especificación OpenAPI define un formato de descripción abierto e independiente de los fabricantes para los servicios de API. En particular, OpenAPI puede utilizarse para describir, desarrollar, probar y documentar las API compatibles con REST.

La actual especificación OpenAPI surgió del proyecto predecesor Swagger. La empresa de desarrollo SmartBear sometió la especificación existente de Swagger a una licencia abierta y dejó el mantenimiento y desarrollo posterior en manos de la iniciativa OpenAPI. Además de SmartBear, entre los miembros de la iniciativa OpenAPI se encuentran gigantes de la industria como Google, IBM y Microsoft. La Fundación Linux también apoya este proyecto.

Una cuestión que puede causar confusión es la distinción entre OpenAPI y Swagger. OpenAPI es una especificación, es decir, una descripción abstracta que no está ligada a una aplicación técnica concreta. Hasta la versión 2.0, esta especificación todavía se llamaba Swagger y luego fue renombrada como especificación OpenAPI. Sin embargo, las herramientas proporcionadas por SmartBear, la empresa que la desarrolló originalmente, siguen existiendo con el nombre de Swagger.

Con OpenAPI, una API puede describirse de manera uniforme. Esto se conoce como “definición API” y se genera en un formato legible por máquina. En particular, se utilizan dos lenguajes: YAML y JSON.

La especificación OpenAPI define una serie de propiedades que pueden utilizarse para desarrollar una API propia. Estas propiedades se agrupan en los llamados objetos (en inglés, objects). En la actual versión 3.0.3, OpenAPI define la estructura de los siguientes objetos, entre otros:

- Info Object: versión, nombre, etc. de la API.
- Contact Object: datos de contacto del proveedor de la API.
- License Object: licencia bajo la cual la API proporciona sus datos.
- Server Object: nombres del host, estructura del URL y puertos del servidor a través del cual se dirige la API.
- Components Object: componentes encapsulados que pueden utilizarse varias veces dentro de una definición de API.
- Paths Object: rutas relativas a los puntos finales de la API que se utilizan junto con el servidor del objeto.

- Path Item Object: operaciones permitidas para una ruta específica como GET, PUT, POST, DELETE.
- Operation Object: especifica, entre otras cosas, los parámetros y las respuestas del servidor que se esperan de una operación.

En general, OpenAPI se utiliza para describir API REST de manera uniforme. Como esta descripción, es decir, la definición API, está disponible en un formato legible por máquina, se pueden generar automáticamente diversos artefactos virtuales a partir de ella. En concreto, estos incluyen:

- **Creación de documentación API:** La documentación basada en HTML se genera automáticamente a partir de la definición API legible por máquina. Esta sirve como material de consulta para los desarrolladores que acceden a los servicios API. Si la definición API cambia, la documentación se vuelve a generar para que ambas concuerden.
- **Creación de conexiones en diferentes lenguajes de programación:** Con las herramientas apropiadas, se puede crear una biblioteca de software adecuada del lado del cliente a partir de la definición API en un lenguaje de programación compatible. Esto permite a los programadores de todo tipo acceder a la API. La biblioteca de software se incorpora de manera convencional. Por lo tanto, el acceso a los servicios de API tiene lugar, por ejemplo, mediante el acceso a las funciones dentro del mismo entorno de programación.
- **Elaboración de casos de prueba:** Cada componente de un software debe someterse a diversas pruebas para asegurar su funcionalidad. En concreto, es preciso volver a probar un componente de software cada vez que se cambia el código subyacente. A partir de la definición API, se pueden generar estos casos de prueba automáticamente para poder comprobar la funcionalidad de los componentes del software en todo momento.

En última instancia, cabe señalar que no todas las API pueden representarse utilizando OpenAPI. Sin embargo, las API REST son compatibles sin duda.

Ventajas

En general, la ventaja de OpenAPI es que la puesta en marcha, documentación y prueba de una API son coherentes y constantes durante el desarrollo y el mantenimiento. Además, el uso de la especificación OpenAPI permite una mejor coordinación del desarrollo de la API entre los equipos de backend y frontend. En ambos equipos, los componentes del código pueden generarse a partir de la definición API para que tanto en backend como en frontend puedan desarrollarlos y probarlos sin tener que esperar al otro.

Además de estas ventajas generales, OpenAPI se utiliza en particular como estándar de base para el desarrollo de API REST. Esto es atractivo, porque desarrollar una API compatible con REST de forma manual no es una trivialidad.

Además, el uso de OpenAPI ofrece las siguientes ventajas:

- Definir las API HTTP independientemente de un lenguaje de programación específico.
- Generar código de servidor para una API definida en OpenAPI.
- Generar bibliotecas del lado del cliente para una API compatible con OpenAPI en más de 40 lenguajes de programación.
- Procesar una definición OpenAPI con las herramientas apropiadas.
- Crear documentación interactiva de API.
- Permitir que las personas y las máquinas descubran y entiendan las capacidades de un servicio sin tener que mirar el código fuente o la documentación adicional.
- Acceder a los servicios de API con un gasto mínimo de puesta en marcha.
- Utilizar herramientas que nos provee para crear sanitizadores, crear clientes axios, realizar testing, y demás funcionalidades útiles para el desarrollo de nuestra aplicación.

Desventajas

Curva de aprendizaje inicial: su configuración y la correcta documentación de cada endpoint puede resultar laboriosa para quienes recién comienzan a utilizarla. Sin embargo, si se integra desde las primeras etapas del desarrollo, este proceso se vuelve más ágil con el tiempo.

Alcance limitado: OpenAPI está orientado principalmente a describir APIs HTTP con arquitectura REST. Su aplicación a otros tipos de APIs (por ejemplo, basadas en RPC, SOAP o protocolos distintos de HTTP) es limitada o requiere adaptaciones adicionales.

Limitaciones en la documentación automática: para APIs ya desarrolladas se pueden emplear herramientas como *swagger-autogen* para generar documentación automáticamente. No obstante, dichas herramientas presentan limitaciones:

El problema con esta herramienta son los siguientes:

- Los parámetros y propiedades suelen reconocerse como tipo “any” en lugar de identificar correctamente los tipos.
- Su correcto funcionamiento depende de frameworks con estructuras bien definidas (por ejemplo NestJS), lo que puede restringir su uso en otros entornos.

- Puede mostrar comportamientos inconsistentes cuando existen middlewares complejos, tokens o respuestas dinámicas.
- Las estructuras de respuestas generadas automáticamente pueden no reflejar con precisión la implementación real.

Algunos lenguajes cuentan con soluciones más maduras y robustas, como **Swashbuckle para ASP.NET**, que permite generar documentación de forma más precisa y con menos intervención manual. En otros entornos, estas herramientas pueden ser menos completas o requerir mayor configuración.

Conclusión

OpenAPI surge como una herramienta clave para estandarizar la definición, documentación y prueba de APIs REST que se convirtió en un estándar en los últimos años. Su mayor valor está en garantizar coherencia en el desarrollo, facilitar la colaboración entre equipos y automatizar tareas como la generación de clientes, servidores y documentación.

Esta tecnología cuenta con bastos beneficios, como simplificar la documentación o proveer una forma rápida de probar los endpoints. No obstante, OpenAPI tiene limitaciones: una curva de aprendizaje inicial, su enfoque casi exclusivo en REST y las restricciones de las herramientas automáticas en entornos complejos.

En definitiva, la combinación de REST como arquitectura y OpenAPI como especificación de documentación ofrece una base sólida, estandarizada y práctica para el desarrollo de APIs modernas, siempre que se evalúe su alcance frente a necesidades más allá del paradigma REST.

GraphQL

1.1. Contexto y origen de GraphQL

En 2012, Meta (entonces Facebook) necesitaba una API de obtención de datos que fuera lo suficientemente potente para sus aplicaciones móviles de Facebook . Liderado por Lee Byron, Facebook desarrolló GraphQL como una forma de simplificar la obtención de datos, específicamente desde la perspectiva de los diseñadores y desarrolladores de productos. GraphQL fue utilizado por primera vez internamente por Facebook, y luego lanzado públicamente y convertido en código abierto en 2015. Siguiendo la trayectoria de muchos otros proyectos de código abierto, en 2019 el proyecto GraphQL se trasladó a su propia GraphQL Foundation , alojada por la Linux Foundation .

1.2. Definición y propósito

GraphQL constituye un lenguaje de consulta para interfaces de programación de aplicaciones (APIs), acompañado de un entorno de ejecución en el servidor que facilita la obtención de datos de manera precisa y estructurada. Desarrollado inicialmente por Facebook en el año 2012 y publicado como proyecto de código abierto en 2015, GraphQL se distingue de los modelos tradicionales basados en múltiples endpoints, como REST, al proveer un único punto de acceso mediante el cual los clientes pueden especificar con exactitud qué información requieren

El propósito fundamental de GraphQL es optimizar la interacción entre clientes y servidores en el consumo de datos, proponiendo un mecanismo más eficiente, flexible y predecible. En este sentido, busca:

- Minimizar el sobre-fetching y el under-fetching de información, garantizando que los clientes reciban únicamente los campos solicitados.

- Unificar el acceso a múltiples fuentes de datos bajo un único esquema coherente y centralizado.

- Otorgar mayor autonomía a los clientes, al permitir que definan la forma y extensión de los datos requeridos.

- Favorecer la experiencia de desarrollo, gracias a su tipado estricto y a la autogeneración de documentación a partir del esquema definido.

2. Conceptos Fundamentales

GraphQL se estructura en torno a un conjunto de conceptos que determinan su funcionamiento y lo diferencian de otros enfoques de diseño de APIs. Estos elementos permiten comprender cómo se definen, solicitan y manipulan los datos dentro de este paradigma.

2.1. Esquema (*Schema*)

El esquema constituye el núcleo de una implementación de GraphQL. Define los tipos de datos disponibles, las relaciones entre ellos y las operaciones que pueden ejecutarse. Funciona como un contrato entre el servidor y el cliente, garantizando que ambas partes compartan una comprensión común de la estructura de la información.

2.2. Consultas (*Queries*)

Las consultas representan las operaciones de lectura en GraphQL. A través de ellas, el cliente especifica de manera declarativa qué campos y subcampos necesita obtener, evitando la transferencia de datos superfluos. Una consulta puede abarcar múltiples tipos de datos relacionados, obtenidos en una sola solicitud al servidor.

2.3. Mutaciones (*Mutations*)

Las mutaciones constituyen las operaciones de escritura o modificación de datos. Mediante ellas, los clientes pueden crear, actualizar o eliminar información. Al igual que en las consultas, las mutaciones permiten definir qué datos deben ser devueltos tras la ejecución de la operación, facilitando la confirmación inmediata del cambio realizado.

2.4. Suscripciones (*Subscriptions*)

Las suscripciones introducen la capacidad de recibir datos en tiempo real. Se basan generalmente en el uso de WebSockets u otros mecanismos de comunicación persistente, y permiten que el servidor envíe actualizaciones automáticas a los clientes cuando ocurre un evento determinado, sin necesidad de que estos realicen solicitudes reiteradas.

2.5. Resolutores (*Resolvers*)

Los resolutores son funciones que implementan la lógica de obtención de datos para cada campo del esquema. Actúan como intermediarios entre la consulta del cliente y las fuentes de información (bases de datos, servicios externos o

sistemas internos), garantizando que la respuesta cumpla con la estructura definida en el esquema.

3. Ventajas y Desventajas de GraphQL

3.1 Ventajas

- **Eficiencia en la transferencia de datos**: evita la sobrecarga de información al permitir que el cliente solicite únicamente los campos que necesita.

- **Flexibilidad en el consumo de información**: un único endpoint centralizado satisface múltiples necesidades de consulta.

- **Tipado fuerte**: garantiza coherencia y reduce errores durante el desarrollo.

- **Experiencia de desarrollo mejorada**: el esquema actúa como documentación, facilitando la comprensión de la API.

3.2 Desventajas

- **Mayor carga en el servidor**: consultas mal diseñadas pueden generar sobrecarga de procesamiento.

- **Gestión del caché más compleja**: a diferencia de REST, donde el almacenamiento en caché se simplifica a nivel de endpoints, en GraphQL requiere estrategias más sofisticadas.

- **Problemas de seguridad potenciales**: la posibilidad de realizar consultas muy profundas o costosas obliga a implementar medidas de control adicionales.

4. Comparación con REST

4.1 Arquitectura y endpoints

En REST, cada recurso se accede a través de un endpoint específico, lo que puede derivar en múltiples solicitudes para obtener información relacionada. En cambio, GraphQL unifica todas las operaciones bajo un único endpoint, desde el cual el cliente define la estructura de los datos requeridos.

4.2 Ejemplo práctico comparativo

- **REST**: para obtener los datos de un usuario junto con sus publicaciones, se necesitan dos solicitudes distintas, cada una con una estructura de respuesta predefinida.

- **GraphQL**: la misma información puede obtenerse en una única consulta, especificando los campos de interés.

4.3 Situaciones de aplicabilidad

REST resulta adecuado en escenarios donde la simplicidad y la estandarización son prioritarias. GraphQL, en cambio, ofrece ventajas en contextos donde los clientes poseen requerimientos de datos variables, como en aplicaciones móviles o sistemas con interfaces complejas.

5. Casos de Uso

5.1 Aplicaciones móviles

En dispositivos móviles, la eficiencia en la transferencia de datos es crucial. GraphQL permite reducir el consumo de ancho de banda al limitar la información transmitida únicamente a la solicitada.

5.2 Aplicaciones web modernas

Frameworks como React, Angular o Vue se benefician de la flexibilidad de GraphQL, ya que pueden adaptar las consultas a los componentes específicos de la interfaz.

5.3 APIs públicas y empresas

Diversas compañías han adoptado GraphQL en sus APIs públicas, entre ellas GitHub, Shopify y Twitter. Estas implementaciones permiten a los desarrolladores integrar servicios de forma más eficiente y personalizada.

6. Ecosistema y Herramientas

6.1 Lado servidor

Entre las principales implementaciones se encuentran Apollo Server, GraphQL Yoga, Hot Chocolate y Sangria. Estas herramientas facilitan la creación de servidores compatibles con GraphQL en distintos lenguajes de programación.

6.2 Lado cliente

Existen librerías que simplifican la integración de GraphQL en las aplicaciones cliente, tales como Apollo Client y Relay, las cuales gestionan consultas, mutaciones y el almacenamiento en caché de manera eficiente.

6.3 Herramientas de exploración

GraphQL ofrece entornos interactivos para probar consultas y explorar esquemas, como **GraphiQL** y **GraphQL Playground**, los cuales mejoran la productividad de los desarrolladores y actúan como documentación dinámica.

7. Ejemplo Práctico

7.1 Definición de un esquema básico

Un esquema en GraphQL define los tipos de datos disponibles y las operaciones que pueden realizarse sobre ellos. El siguiente esquema simplifica la gestión de usuarios y publicaciones:

```
type Usuario {
  id: ID!
  nombre: String!
  email: String!
  publicaciones: [Publicacion!]!
}

type Publicacion {
  id: ID!
  titulo: String!
  contenido: String!
  fecha: String!
}

type Query {
  usuario(id: ID!): Usuario
  publicaciones: [Publicacion!]!
}
```

```
type Mutation {
  crearPublicacion(
    titulo: String!,
    contenido: String!
  ): Publicacion
}
```

En este esquema se definen dos tipos principales: Usuario y Publicación, junto con dos operaciones: una consulta (*Query*) y una mutación (*Mutation*).

7.2 Consulta de ejemplo (*Query*)

Una aplicación cliente que desee obtener información sobre un usuario y sus publicaciones podría realizar la siguiente consulta:

```
{
  usuario(id: 1) {
    nombre
    email
    publicaciones {
      titulo
      fecha
    }
  }
}
```

La respuesta del servidor tendría únicamente la información solicitada:

```
{
  "data": {
    "usuario": {
      "nombre": "Juan Perez",
      "email": "juanperez@example.com",
      "publicaciones": [
        {
          "titulo": "Introducción a GraphQL",
          "fecha": "2025-09-01"
```

```
        },
        {
          "titulo": "Comparación entre REST y GraphQL",
          "fecha": "2025-09-10"
        }
      ]
    }
  }
}
```

7.3 Ejemplo de mutación (*Mutation*)

Para crear una nueva publicación, el cliente puede ejecutar la siguiente mutación:

```
mutation {
  crearPublicacion(
    titulo: "Nuevas tendencias en GraphQL",
    contenido: "Análisis de suscripciones en tiempo real."
  ) {
    id
    titulo
    fecha
  }
}
```

El servidor devuelve como resultado la publicación recién creada, confirmando la operación:

```
{
  "data": {
    "crearPublicacion": {
      "id": "3",
      "titulo": "Nuevas tendencias en GraphQL",
      "fecha": "2025-09-30"
    }
  }
}
```

8. Futuro y Tendencias

8.1 Federation y microservicios

La adopción de arquitecturas basadas en microservicios ha motivado el desarrollo de GraphQL Federation, un mecanismo que permite combinar múltiples esquemas en una única interfaz coherente.

8.2 Tiempo real y suscripciones

El creciente interés en aplicaciones interactivas impulsa la expansión de las suscripciones en GraphQL, las cuales permiten recibir datos en tiempo real mediante tecnologías como *WebSockets*.

8.3 Convivencia con otras tecnologías

Lejos de reemplazar completamente a REST, GraphQL puede coexistir con este y con otras herramientas como gRPC, siendo elegido en aquellos contextos donde sea beneficioso.

9. Conclusiones

GraphQL representa una evolución en el diseño de interfaces de programación de aplicaciones, orientada a satisfacer las demandas de eficiencia, flexibilidad y escalabilidad que caracterizan a las aplicaciones modernas. Su capacidad para reducir la transferencia innecesaria de datos, unificar múltiples fuentes en un único endpoint y proporcionar un esquema tipado lo convierte en una herramienta valiosa tanto para desarrolladores como para organizaciones.

No obstante, su implementación conlleva desafíos, entre ellos la necesidad de un diseño cuidadoso de las consultas y la gestión avanzada del rendimiento y la seguridad en el servidor. En este sentido, GraphQL no debe considerarse un reemplazo absoluto de REST, sino una alternativa complementaria que aporta ventajas particulares en contextos donde la precisión y personalización del consumo de datos resultan críticas.

Conclusión conjunta / Comparación de tecnologías

Ambas tecnologías nacieron para resolver el mismo problema, el fetching de datos de parte del cliente. REST es la solución más sencilla de implementar, en la mayoría de casos es suficiente. Sin embargo, GraphQL es la tecnología más compleja y completa de las dos, permitiendo resolver el problema de under y overfetching que tanto caracteriza a las API Rest.

REST es un modelo probado y ampliamente adoptado, que organiza los recursos mediante múltiples endpoints y utiliza los métodos HTTP para acceder y modificar datos. Su principal fortaleza es la simplicidad y la estandarización, mientras que su mayor desafío surge cuando se requieren múltiples llamadas para obtener datos relacionados o se recibe información innecesaria.

GraphQL, por su parte, ofrece un enfoque más flexible y centrado en el cliente, permitiendo que este solicite exactamente los datos que necesita mediante un único endpoint. La existencia del esquema centralizado y los resolvers asegura que las consultas sean predecibles y coherentes, mientras que las mutaciones permiten modificar datos de manera controlada y las suscripciones facilitan la actualización en tiempo real.

En síntesis, REST es ideal para escenarios donde la simplicidad y la estandarización son prioritarias, mientras que GraphQL sobresale en aplicaciones que requieren eficiencia en la transferencia de datos y flexibilidad en las consultas. Ambos enfoques pueden coexistir y complementarse, dependiendo de las necesidades del proyecto y el cliente.

Tema	REST	GraphQL
Modelo de datos	Endpoints fijos (ej: /users/1/posts)	Un solo endpoint (/graphql) y queries flexibles
Overfetching / Underfetching	Problema común: Devuelve más o menos información de la necesario	Se pide exactamente lo necesario.
Versionado	Normal tener /v1/, /v2/	Generalmente no hace falta, el esquema evoluciona
Tipado	JSON (estandar) / YAML	Esquema fuertemente tipado (GraphQL Schema)
Performance	Muchas veces hay que hacer varias requests para juntar datos	Una sola query puede traer todo lo necesario
Caching	Sencillo (porque es RESTful y usa GETs predecibles)	Más complejo, hay que implementar cache a nivel cliente (ej: Apollo Client)
Errores	Basado en códigos HTTP (404, 500, 401, etc.)	Siempre 200 OK, los errores vienen en la respuesta

Soporte	Nativo en todos los frameworks y servidores HTTP	Se necesita instalar un runtime de GraphQL

Bibliografía

- Amazon Web Services. (s. f.). *What is RESTful API?* Recuperado de https://aws.amazon.com/what-is/restful-api/

- Equipo editorial de IONOS. (18 febrero 2021). *¿Qué es OpenAPI?* IONOS Digital Guide. Recuperado de https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/que-es-openapi/

- Facebook Open Source. GraphQL Specification. Disponible en: https://graphql.org

- GitHub. GraphQL API. Disponible en: https://docs.github.com/graphql

- Apollo GraphQL. Introduction to GraphQL. Disponible en: https://www.apollographql.com

- F5. ¿Qué es GraphQL?. Disponible en: https://www.f5.com/es_es/glossary/graphql

- Red Hat. ¿Qué es GraphQL?. Disponible en: https://www.redhat.com/es/topics/api/what-is-graphql