Engineer

GraphQL

lenguaje de consultas donde el cliente elige los campos

Qué es

GraphQL es un lenguaje de consultas para API junto con el entorno de ejecución que las resuelve en el servidor. El cliente envía un documento que nombra los campos exactos que quiere y el servidor responde con datos de esa misma forma. La especificación lo describe como lenguaje y motor de ejecución para aplicaciones cliente-servidor. Registra su creación en 2012, el comienzo del trabajo como estándar abierto en 2015 y su condición, desde 2019, de producto del GraphQL Specification Project bajo la Joint Development Foundation. La última versión es la edición de septiembre de 2025.

Todo parte del esquema publicado por el servicio: tipos con campos con nombre, cada campo con su propio tipo. Los escalares integrados son `Int`, `Float`, `String`, `Boolean` e `ID`; el servicio puede añadir otros, como uno para UUID. Todos los tipos admiten null salvo que se marquen con `!`: `String!` nunca puede ser null. Las listas se escriben entre corchetes: `[LineItem!]!` siempre está presente y nunca contiene null. También se puede consultar el esquema. Los campos de introspección `__schema` y `__type` permiten preguntar al servicio qué ofrece, y sobre ello se construyen herramientas de desarrollo y bibliotecas cliente.

Hay tres clases de operación. Una consulta es, en palabras de la especificación, «una recuperación de solo lectura»; una mutación es una escritura seguida de una recuperación; y una suscripción es una petición de larga duración que entrega datos ante una secuencia de sucesos a lo largo del tiempo. En el servidor, una función, normalmente llamada resolver, produce el valor de cada campo. El servicio valida primero la operación frente al esquema y solo después llama a los resolvers de los campos seleccionados. Los campos de nivel superior de una mutación se ejecutan uno tras otro, no en paralelo: dos escrituras de la misma petición no pueden entrar en una condición de carrera entre sí.

La especificación no impone transporte ni formato de serialización. HTTP es lo más habitual: la API se sirve en una sola URL, normalmente acabada en `/graphql`, y responde en JSON, el formato más común. El cliente envía un POST cuyo cuerpo JSON contiene el texto de la consulta en `query` y, cuando hacen falta, sus `variables`. El servidor también puede aceptar GET para consultas, pero una mutación siempre debe ser POST. La respuesta contiene el resultado en `data` y los problemas en `errors`. Las indicaciones de la GraphQL Foundation siguen una especificación GraphQL-over-HTTP que aún es un borrador.

Utilice el concepto en una decisión

Aplique «¿Qué es GraphQL?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. La parte que el usuario no ve: dónde se guardan los datos, qué reglas se aplican y cómo pueden comunicarse otros sistemas con ella.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Diseño y operación de 6 flujos ETL/ELT — 6 flujos ETL en tres motores de bases de datos

Diseño y operación de 6 flujos ETL/ELT Póngase en contacto →

Por qué importa

En la mayoría de las API HTTP, el servidor decide qué devuelve cada punto de acceso. Una pantalla que necesita el estado de un pedido y el nombre de su cliente recibe el pedido y el cliente completos, más de lo solicitado, o debe hacer una petición por recurso y unir las respuestas. La especificación llama a esos problemas over-fetching y under-fetching. GraphQL invierte la relación: el cliente pide campos individuales y recibe exactamente esos, ninguno más. Una petición también puede seguir relaciones, del pedido a sus líneas y al cliente, porque tiene la forma de los datos que recibirá.

El sistema de tipos aporta dos ventajas. Cada operación se valida frente al esquema antes de ejecutar ningún resolver: un campo mal escrito o un argumento de tipo incorrecto provoca el rechazo de toda la operación antes de ejecutar nada. Las herramientas pueden detectar esos mismos errores mientras se escribe el código. Además, como el cliente solo recibe los campos que eligió, añadir un campo no exige cambios en los clientes existentes. Así resulta práctico evolucionar una API sin números de versión: aparece un campo nuevo junto al antiguo, se marca este con `@deprecated` y un motivo, sigue siendo válido para no romper clientes y se elimina cuando ya no lo usa ninguno.

Los costes son reales y proceden sobre todo de salir de los mecanismos propios de HTTP. Las entidades ya no se identifican mediante URL y todo va a una dirección, normalmente por POST, mientras que son las peticiones GET las que suelen considerarse cacheables de forma predeterminada. Hay que reconstruir deliberadamente la caché que una API REST obtiene de los intermediarios sin esfuerzo. El código de estado ya no cuenta toda la historia: una respuesta con datos y errores sigue siendo un éxito en HTTP. El cliente compone peticiones que el servidor nunca ha visto; una sola puede exigir mucho trabajo y es el servicio quien debe protegerse, no GraphQL quien lo hace por él. Los resolvers por campo son fáciles de escribir, pero una implementación ingenua hace demasiadas llamadas a la base subyacente.

Cómo se aplica en la práctica

Una pequeña parte de un esquema de pedidos, cada uno con líneas y cliente:

```
type Query {
  order(id: ID!): Order
}

type Order {
  id: ID!
  status: String!
  items: [LineItem!]!
  customer: Customer
}

type LineItem {
  name: String!
  quantity: Int!
}

type Customer {
  name: String!
}
```

Una pantalla que muestra el estado, las líneas y el nombre del cliente envía esta operación a `/graphql` en el cuerpo de un POST, con `{"id": "7421"}` como variables:

```
query OrderSummary($id: ID!) {
  order(id: $id) {
    status
```

```
    items {
      name
      quantity
    }
    customer {
      name
    }
  }
}
```

Supongamos que se resuelven el pedido y sus líneas, pero el servicio de `customer` está caído. La respuesta aún contiene todo lo que se pudo producir. El campo fallido vuelve como null y una entrada de `errors` explica qué falló, dónde ocurrió en el documento —la línea 8, columna 5 es `customer` — y qué ruta de la respuesta afecta:

```
{
  "data": {
    "order": {
      "status": "open",
      "items": [{ "name": "notebook", "quantity": 2 }],
      "customer": null
    }
  },
  "errors": [
    {
      "message": "Customer service unavailable",
      "locations": [{ "line": 8, "column": 5 }],
      "path": ["order", "customer"]
    }
  ]
}
```

Como `customer` admite null, el daño se limita a ese campo. Si fuera `Customer!` , el error se propagaría al padre y todo el pedido volvería como null. Además, como `data` no es null, el servidor responde con un estado 2xx pese al error: HTTP no tiene código para el éxito parcial. Un cliente que solo mire el código tratará esa respuesta como completa.

El diseño contrario, donde el servidor decide la forma de cada respuesta, es el que construyó un proyecto del portfolio de este sitio. En su paso directo de JSON de Postgres a la API, las funciones PostgreSQL ensamblan la respuesta completa como JSON y el manejador Go la reenvía intacta. Cambiar lo que devuelve una función cambia lo que recibe el cliente sin editar una línea del manejador. Allí el

frontend depende directamente de la forma que le da la base. En GraphQL el cliente nombra los campos y el servidor debe poder resolver cualquier combinación permitida por el esquema.

Dónde surgen los problemas

Un campo, mil llamadas a la base. Los resolvers son funciones pequeñas, una por campo, y un servicio ingenuo los llama objeto a objeto. Una lista de pedidos con el cliente de cada uno cuesta una llamada a la base para la lista y otra por pedido. La GraphQL Foundation lo llama problema N+1 y señala el procesamiento por lotes: se reúnen las peticiones de datos realizadas en un intervalo corto y se envían a la base como una sola petición, con una herramienta como DataLoader de Facebook.

Una petición lo pide todo. Como el cliente escribe la operación, un documento puede anidar amigos de amigos a muchos niveles, repetir un campo costoso bajo cien alias o agrupar muchas operaciones en una petición. Todos son válidos en GraphQL y pueden mantener ocupado el servidor. Las defensas corresponden al servicio: paginar campos de lista, rechazar antes de ejecutar las operaciones que excedan una profundidad o amplitud máxima, limitar alias y tamaño de lote y, para afinar más, asignar pesos a los campos y rechazar operaciones cuyo coste estimado supere el presupuesto. Es el análisis de complejidad de consultas, para el que la propia especificación no da reglas. Una API que solo atiende a sus propios clientes puede ir más lejos con documentos de confianza: únicamente se permiten las operaciones almacenadas de antemano y buscadas por hash.

200 OK oculta el fallo. Una respuesta con errores puede seguir siendo 2xx, como en el ejemplo. Con el tipo de medio antiguo `application/json`, las indicaciones van más allá: se recomienda 2xx incluso si una petición válida no puede ejecutarse, y algunos servidores antiguos devuelven 2xx también ante errores de validación. El cliente y su monitorización deben leer `errors`, no solo el estado. El tipo más reciente `application/graphql-response+json` vuelve a dar significado al estado: una petición válida que no puede ejecutarse recibe entonces 4xx o 5xx. Los clientes deberían pedirlo en la cabecera `Accept`.

La caché deja de funcionar. Cada operación es un POST a la misma URL: las cachés entre cliente y servidor, incluida una CDN, ven una dirección y un método cuyas respuestas no almacenan de forma predeterminada. Enviar consultas mediante GET ayuda, pero un documento largo puede exceder la longitud de URL admitida por navegadores y CDN. Las consultas persistidas resuelven ambos

problemas: el cliente envía un hash y el servidor busca el documento completo antes de validarlo y ejecutarlo. Cuando se almacenan respuestas en caché HTTP, las sensibles deben guardarse en una caché privada o no guardarse en caché.

Desaparece un campo. Los clientes eligen campos individualmente: eliminar o renombrar uno rompe todos los que aún lo seleccionan, y de forma brusca. Una operación que nombra un campo desconocido se rechaza por completo durante la validación, sin devolver datos. La especificación mantiene válidos los campos obsoletos precisamente para no romper clientes existentes. Añada primero el sustituto, marque el antiguo con `@deprecated` y un motivo que apunte al nuevo y elimínelo solo cuando ya nada lo seleccione.

Lecturas adicionales

- Versiones de la especificación GraphQL — Todas las ediciones, desde julio de 2015 a septiembre de 2025, y el borrador de trabajo: lenguaje, tipos, validación, ejecución y errores.
- Servicio mediante HTTP — El punto de acceso único, POST y GET, los dos tipos de medio de respuesta y las reglas de estado de cada uno.
- Rendimiento — GET para consultas, consultas persistidas, problema N+1 y procesamiento por lotes.
- Seguridad — Documentos de confianza, límites de profundidad, amplitud y lotes, análisis de complejidad, introspección y nivel de detalle de los errores en producción.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es GraphQL?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/graphql/