



Universidad Católica
San Pablo

Ciencia de la Computación

Desarrollo Basado en Plataformas

Docente RENZO HERNAN MEDINA ZEBALLOS

GraphQL

Entregado el 15/11/2024

PACSI PONCE, ANGEL FERNANDO

Semestre III

2024-2

"El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo"

GraphQL

1. ¿Qué es?

GraphQL es un lenguaje de consulta para APIs y un entorno de ejecución que facilita las interacciones entre clientes y servidores en la obtención de datos precisos. Desarrollado internamente por Facebook en 2012 y lanzado al público en 2015, GraphQL permite a los clientes pedir exactamente la información que necesitan y nada más, evitando así los problemas de sobrecarga de datos que suelen ocurrir en APIs basadas en REST.

Para el frontend, GraphQL es una herramienta poderosa de consulta de datos que simplifica la comunicación entre el cliente y el servidor. Desde el lado del backend, GraphQL se implementa como una capa de tiempo de ejecución que describe completamente los datos disponibles en la API. Además, GraphQL facilita la agregación de datos de diversas fuentes, y gracias a su sistema de tipos, asegura que las consultas estén estructuradas y validadas correctamente.

Como especificación, GraphQL no está ligado a un lenguaje en particular; se puede implementar en más de 20 lenguajes de programación, lo que lo convierte en una herramienta extremadamente versátil en distintos entornos de desarrollo.

2. Conceptos:

- **Schemas**

En GraphQL, un *schema* o esquema representa una descripción completa y comprensible de los datos que la API puede proporcionar, así como las posibles operaciones que se pueden ejecutar para consultarlos o modificarlos. Los esquemas definen la estructura de datos disponibles, describiendo los objetos, sus campos, relaciones, y métodos (queries y mutaciones) que los clientes pueden utilizar para interactuar con la API. Esta estructura actúa como una guía central que tanto el frontend como el backend pueden seguir, asegurando que todos trabajen sobre una misma base de datos y tipos de datos, facilitando así el desarrollo.

El esquema es el corazón de cualquier implementación de GraphQL. Especifica claramente el conjunto de datos posibles y todas las relaciones entre ellos, validando y ejecutando las consultas en base a esta definición. Los esquemas se construyen mediante el Lenguaje de Definición de Esquema (*SDL*), lo que permite a los desarrolladores comprender fácilmente las operaciones permitidas en la API, mejorando tanto la flexibilidad como la precisión de las consultas.

- **Queries**

Las *queries* o consultas en GraphQL son las solicitudes específicas que el cliente hace para obtener datos de la API. A diferencia de REST, donde usualmente se requieren múltiples endpoints para acceder a distintos conjuntos de datos, en GraphQL una única consulta permite recuperar datos complejos de múltiples fuentes al mismo tiempo. Esto permite que el cliente especifique exactamente los campos que necesita, evitando recibir datos innecesarios y reduciendo la sobrecarga en la transferencia de información.

Cada *query* detalla los campos específicos que deben ser devueltos por el servidor, brindando un control granular sobre los datos que se necesitan en cada solicitud. Esto mejora la eficiencia de las consultas y asegura que el cliente obtenga solo la información relevante, facilitando el rendimiento y la precisión de las aplicaciones.

- **Resolvers**

Un *resolver* es una función esencial en GraphQL que se encarga de proporcionar las respuestas a las consultas (*queries*) y mutaciones realizadas por los clientes. Cada campo definido en un esquema GraphQL tiene un resolver asociado, el cual se encarga de recuperar los datos correspondientes de la base de datos o de cualquier otra fuente de datos disponible.

Cuando un cliente hace una solicitud a la API, GraphQL llama automáticamente al resolver apropiado para obtener los datos solicitados. Esto permite manejar cada campo de manera independiente, asegurando que los datos específicos se procesen y devuelvan correctamente. En esencia, los resolvers son los encargados de "resolver" las consultas y mutaciones, conectando el esquema con la lógica de negocio y las fuentes de datos subyacentes.

3. GraphQL vs. REST APIs

- **Similitudes**

1. **Modelo Cliente-Servidor**

Ambos modelos permiten el intercambio de datos mediante solicitudes de clientes y respuestas de servidores para facilitar la comunicación entre servicios y aplicaciones.

2. **Diseño Basado en Recursos**

Tanto GraphQL como REST utilizan recursos con identificadores únicos y operaciones específicas que permiten manipular estos recursos de manera estructurada.

3. **Intercambio de Datos**

Ambos soportan el formato de datos JSON para la comunicación y permiten la implementación de almacenamiento en caché para optimizar el rendimiento.

4. **Neutralidad del Lenguaje y Base de Datos**

Ambos pueden funcionar con cualquier lenguaje de programación y estructura de base de datos, lo que facilita su integración en diversas plataformas.

- **Diferencias Clave**

1. **Enfoque Arquitectónico**

- **REST:** Es un estilo arquitectónico que utiliza verbos HTTP (GET, POST, PUT, DELETE) y múltiples endpoints para interactuar con distintos recursos.
- **GraphQL:** Es un lenguaje de consulta que permite al cliente especificar exactamente qué datos necesita, todo a través de un único endpoint.

2. **Solicitud del Cliente**

- **REST:** Utiliza URL fijas y verbos HTTP que identifican recursos específicos, limitando la flexibilidad en las consultas.
- **GraphQL:** Emplea consultas para leer datos, mutaciones para modificarlos y suscripciones para actualizaciones en tiempo real, permitiendo solicitudes más flexibles y específicas en cuanto a los campos requeridos.

3. Datos Devueltos al Cliente

- **REST:** Responde con datos en una estructura completa definida por el servidor.
- **GraphQL:** Retorna únicamente los datos solicitados por el cliente, en la estructura especificada en la consulta.

4. Esquema del Servidor

- **REST:** No necesita un esquema del servidor, aunque puede definirse opcionalmente.
- **GraphQL:** Utiliza un esquema detallado en el Lenguaje de Definición de Esquemas (SDL) que define todos los datos y servicios disponibles.

5. Control de Versiones

- **REST:** Usa control de versiones en la URL (por ejemplo, /api/v1/recurso).
- **GraphQL:** Evita el control de versiones y maneja la compatibilidad hacia atrás, eliminando campos obsoletos con advertencias para los desarrolladores.

6. Manejo de Errores

- **REST:** Permite flexibilidad en el manejo de errores, pero requiere una integración explícita en el código.
- **GraphQL:** Gracias a su tipado estricto, permite identificar errores de solicitud automáticamente, facilitando la identificación de problemas en la consulta.

4. Ventajas y Desventajas de GraphQL

Ventajas de GraphQL

1. Precisión en la Recuperación de Datos

Los clientes pueden solicitar únicamente los datos específicos que necesitan, lo que reduce el uso innecesario de ancho de banda y mejora el rendimiento en aplicaciones móviles o con conexiones limitadas.

2. Flexibilidad en las Consultas

GraphQL permite estructurar las consultas para obtener los datos en el formato

deseado, lo cual es útil para aplicaciones complejas o cuando se necesita personalización en las respuestas.

3. **Evolución de la API**

Facilita la adición de nuevos campos y tipos sin romper las consultas existentes. Esta compatibilidad hacia atrás permite a los desarrolladores hacer mejoras sin afectar a las versiones anteriores.

4. **Unificación de Múltiples Fuentes de Datos**

GraphQL puede combinar datos de diversas bases de datos y servicios en una sola consulta, consolidando la información de varias fuentes en una respuesta unificada.

5. **Menos Peticiones**

Con una única solicitud, GraphQL puede devolver todos los datos necesarios, reemplazando múltiples llamadas a la API en REST y reduciendo así la cantidad de peticiones realizadas al servidor.

Desventajas de GraphQL

1. **Complejidad del Cliente**

Los clientes deben manejar la estructura de sus consultas y gestionar la respuesta, lo que puede aumentar la complejidad en la implementación del cliente y requerir conocimientos específicos.

2. **Seguridad y Control de Acceso**

La flexibilidad de GraphQL en las consultas presenta desafíos adicionales en la implementación de medidas de seguridad adecuadas, como el control de acceso y la validación de datos, ya que cada campo puede necesitar permisos específicos.

3. **Problemas de Rendimiento (N+1 Problem)**

Las consultas anidadas pueden llevar a múltiples solicitudes a la base de datos, lo cual puede degradar el rendimiento. Sin embargo, este problema se puede mitigar con herramientas como *DataLoader*, que permite optimizar la carga de datos en consultas anidadas.

4. **Almacenamiento en Caché**

GraphQL generalmente utiliza métodos POST para las consultas, y al no utilizar identificadores de recursos en las URLs, se dificulta el almacenamiento

en caché nativo. Implementar el almacenamiento en caché en GraphQL requiere soluciones personalizadas.

5. **Sobrecarga en el Servidor**

Las consultas de gran tamaño o altamente anidadas pueden generar una sobrecarga en el servidor, ya que GraphQL procesa y estructura la respuesta según la consulta, aumentando el uso de recursos en comparación con REST.

REPOSITORIO:

<https://github.com/FerPonce42/TAREAS-DBEP/tree/main/TAREAS/GRAPHQL>

DEMOSTRACIONES:

app.py:

```
app.py > ...
1  from ariadne import QueryType, make_executable_schema
2  from ariadne.asgi import GraphQL
3  import uvicorn
4
5  from ariadne import load_schema_from_path
6  type_defs = load_schema_from_path("schema.graphql")
7
8  query = QueryType()
9
10 from resolvers import resolver_hola, resolver_saludo, resolver_obtenerPersona
11
12 query.set_field("hola", resolver_hola)
13 query.set_field("saludo", resolver_saludo)
14 query.set_field("obtenerPersona", resolver_obtenerPersona)
15
16 esquema = make_executable_schema(type_defs, query)
17
18 app = GraphQL(esquema)
19
20 if __name__ == "__main__":
21     uvicorn.run(app, host="127.0.0.1", port=8000)
22
```

resolvers.py:

```
resolvers.py > resolver_hola
1  def resolver_hola(obj, info):
2      return "¡Hola, DBEP!"
3
4  def resolver_saludo(obj, info, nombre):
5      return f"¡Hola, {nombre}!"
6
7  personas_db = {
8      "1": {"id": "1", "nombre": "Fernando", "edad": 19},
9      "2": {"id": "2", "nombre": "Ángel", "edad": 15}
10 }
11
12 def resolver_obtenerPersona(obj, info, id):
13     return personas_db.get(id)
14
```


schema.graphql:

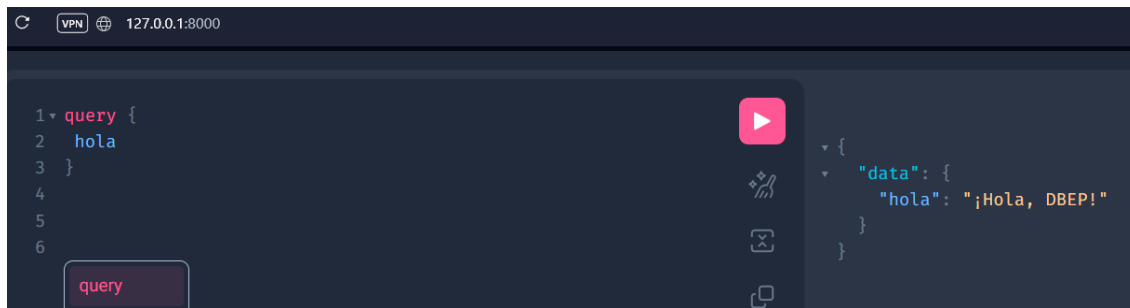
```
schema.graphql
1  type Query {
2      hola: String!
3      saludo(nombre: String!): String!
4      obtenerPersona(id: ID!): Persona
5  }
6
7  type Persona {
8      id: ID!
9      nombre: String!
10     edad: Int!
11 }
12
```

Ejecutar el servidor desde la terminal:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
* History restored

PS C:\Users\Fernando Ponce\Documents\TAREAS-DBEP\TAREAS\GRAPHQL> py app.py
INFO:      Started server process [17548]
INFO:      Waiting for application startup.
INFO:      ASGI 'lifespan' protocol appears unsupported.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

INTERFAZ:



A screenshot of a GraphQL IDE interface. The top bar shows a VPN icon and the address 127.0.0.1:8000. The left pane contains a query:

```
1 query {  
2   hola  
3 }  
4  
5  
6
```

 Below the query is a text input field containing the word "query". The right pane shows the JSON response:

```
{  
  "data": {  
    "hola": "¡Hola, DBEP!"  
  }  
}
```

 Between the panes are icons for running the query, toggling syntax highlighting, and copying the text.

CONSULTAS:

1)



A screenshot of a GraphQL IDE interface. The left pane contains a query:

```
1 query {  
2   saludo(nombre: "Fernando")  
3 }  
4  
5  
6  
7
```

 The right pane shows the JSON response:

```
{  
  "data": {  
    "saludo": "¡Hola, Fernando!"  
  }  
}
```

2)



A screenshot of a GraphQL IDE interface. The left pane contains a query:

```
1 query {  
2   obtenerPersona(id: "1") {  
3     id  
4     nombre  
5     edad  
6   }  
7 }  
8  
9  
10  
11
```

 The right pane shows the JSON response:

```
{  
  "data": {  
    "obtenerPersona": {  
      "id": "1",  
      "nombre": "Fernando",  
      "edad": 19  
    }  
  }  
}
```

3)



A screenshot of a GraphQL IDE interface. The left pane contains a query:

```
1 query {  
2   obtenerPersona(id: "3") {  
3     id  
4     nombre  
5     edad  
6   }  
7 }  
8
```

 The right pane shows the JSON response:

```
{  
  "data": {  
    "obtenerPersona": null  
  }  
}
```