Sumario

[1. REST APIs 2](#_Toc194057177)

[2. GraphQL vs REST API 2](#_Toc194057178)

[3. GraphQL 3](#_Toc194057179)

[**4.** **Ventajas y Desventajas: Overfetching e Underfetching** 3](#_Toc194057180)

[5. SDL (Schema Definition Language)? 7](#_Toc194057181)

[6. GraphQL Query 8](#_Toc194057182)

[7. GraphQL Mutation 9](#_Toc194057183)

[8. GraphQL Resolver 9](#_Toc194057184)

[9. Problema N+1 no GraphQL 11](#_Toc194057185)

[10. Conclusão 13](#_Toc194057186)

[11. Referências 14](#_Toc194057187)

# REST APIs

Las APIs REST son ampliamente adoptadas para integrar sistemas. Su estructura se basa en múltiples endpoints, cada uno representando un recurso específico, como /users, /products, etc.

**Limitaciones de** **REST:**

* **Múltiples endpoints:** cada recurso exige un endpoint específico.
* **Datos fijos:** el cliente recibe un conjunto de información que puede contener mucho más (o menos) de lo que realmente necesita.
* **Overfetching:** el cliente recibe más datos de los que necesita.
* **Underfetching:** el cliente necesita hacer varias llamadas para construir un único objeto.
* **Versionado frecuente para la evolución de la API.**

# GraphQL vs REST API

GraphQL y REST son dos enfoques ampliamente utilizados para la construcción de APIs, cada uno con sus características y ventajas específicas. La siguiente tabla resume las principales diferencias entre ellos:

|  |  |  |
| --- | --- | --- |
| **Característica** | **Rest API** | **GraphQL** |
| Número de endpoints | Múltiples endpoints para diferentes recursos. | Un único endpoint (/graphql) para todas las operaciones. |
| Retorno de datos | Respuestas fijas definidas por el servidor. | El cliente define exactamente qué datos desea recibir. |
| Overfetching | Si, es común recibir más datos de los necesarios. | No, el cliente solicita solo los datos que necesita. |
| Underfetching | Sí, puede requerir múltiples solicitudes para obtener todos los datos. | No, una sola consulta puede recuperar todos los datos necesarios. |
| Metodos HTTP | Usa métodos HTTP estándar: GET, POST, PUT, DELETE, etc. | Generalmente usa solo POST para enviar consultas y mutaciones. |

**Cuándo usar GraphQL en lugar de REST?**

* Si necesitas que los clientes elijan los datos que quieren..
* Cuando se desea reducir múltiples solicitudes en APIs complejas**.**
* Para permitir la evolución sin necesidad de versionamiento.

# GraphQL

​ GraphQL es un lenguaje de consulta y modificación de datos basado en grafos, creado por Facebook en 2012 y abierto al público en 2015. Es una alternativa moderna a REST.

GraphQL es un lenguaje de consulta declarativo que permite al cliente especificar exactamente qué datos desea obtener de una API, lo que hace que la evolución de las API sea más sencilla y sin necesidad de versiones.

**Beneficios:**

* Un único endpoint para todas las operaciones.
* El cliente define qué campos desea recibir.
* Consulta precisa (sin exceso ni falta de datos)
* Evolución sin necesidad de versionamiento.

**GraphQL** **no es**:

* **Una base de dados** – no almacena datos.
* **Un framework** – es una especificación, con implementaciones en varios lenguajes..
* **Exclusivo para HTTP** – Aunque es común con HTTP, se puede utilizar con WebSocket, TCP, etc.

**Operaciones principales:**

1. **Query** → Para obtener datos (equivalente a GET en REST).
2. **Mutation** → Para modificar o eliminar datos (POST/PUT/DELETE).
3. **Subscription** → Para recibir actualizaciones en tiempo real sobre los cambios de datos.
4. **Ventajas y Desventajas: Overfetching e Underfetching**

Una de las principales ventajas de GraphQL frente a REST es cómo permite al cliente seleccionar exactamente qué datos necesita.

Em REST, as respostas são **padronizadas pelo backend**, o que frequentemente gera dois problemas:

* **Overfetching**: o cliente recebe **dados demais**, que não vai usar.
* **Underfetching**: o cliente recebe **dados de menos** e precisa **fazer várias requisições** para completar a informação.

**Overfetching (Exceso de Datos)?**

Ocurre cuando una API devuelve más datos de los necesarios. Esto significa que el cliente recibe información que no solicitó, lo que genera procesamiento innecesario.

Imagine que una aplicación necesita mostrar solo el nombre y el correo electrónico del usuario. Si el endpoint /users/{id} devuelve todos los detalles del usuario, incluida la dirección, el número de teléfono y la fecha de nacimiento, entonces se trata de **Overfetching**.

**Ejemplo REST de overfetching**

|  |
| --- |
| http:  GET /users/1 |

|  |
| --- |
| json:  {  "id": 1,  "name": "Carlos",  "email": "carlos@email.com",  "phone": "123-456-7890",  "address": "Rua A, Bairro B",  "birthDate": "1990-01-01"  } |

Problema: el cliente **solo queria el name e email**, pero recibió mucho más.

**Solución con GraphQL**

|  |
| --- |
| graphql:  query {  user(id: 1) {  name  email  }  } |

|  |
| --- |
| json:  {  "data": {  "user": {  "name": "Carlos",  "email": "carlos@email.com"  }  }  } |

Resultado: solo se devuelven los datos requeridos.

**Underfetching (Falta de Datos)**

Se produce cuando una API no proporciona todos los datos necesarios a la vez, lo que obliga al cliente a realizar múltiples solicitudes encadenadas para obtener información completa.

**Ejemplo REST de underfetching:**

|  |
| --- |
| http:  GET /users/1  GET /users/1/orders |

Problema: el cliente necesita combinar múltiples respuestas.

**Solución con GraphQL**

|  |
| --- |
| graphql:  query {  user(id: 1) {  name  email  orders {  id  total  }  }  } |

Resultado: una única llamada, con estructura anidada y todos los datos deseados.

**Ventajas de GraphQL**

|  |  |
| --- | --- |
| **Ventaja** | **Impacto** |
| Flexibilidad en el frontend | El cliente define los campos que necesita |
| Documentación clara | Schema validado, navegable y compatible |
| Reducción de overfetching | Solo se devuelve lo necesario |
| Eliminación de underfetching | Todo en una sola consulta |

**Desventajas de GraphQL**

|  |  |
| --- | --- |
| **Desventaja** | **Detalles** |
| Complejidad | Schema y resolvers exigen un buen diseño |
| Caching | No hay URL fija como en REST |
| Performance mal implementada | REST y GraphQL Puede generar el problema N+1 cuando overfetching relaciones |

# SDL (Schema Definition Language)?

El **SDL** (Schema Definition Language) es el lenguaje utilizado para describir la estructura de una API GraphQL. Es independiente del lenguaje de programación utilizado en el backend.

Através do SDL definimos o **contrato da API**, incluindo os tipos (type), entradas (input), queries, mutations e mais, especificando claramente quais dados podem ser consultados ou modificados, e como essas operações devem ser realizadas (extensão .graphqls).

**Elementos Definidos con SDL:**

* **Tipos de dados (**type**)**
* **Campos y tipos** (String, Int, ID, Boolean, etc.)
* **Campos obligatorios u opcionales (!)**
* **Consultas (**Query**)**
* **Mutaciones (**Mutation**)**
* **Subscripciones (**Subscription**)**
* **Entradas personalizadas (**input**)**
* **Otros**: enum, interfaz, unión, fragmento, entre otros

**Ejemplo:**

|  |
| --- |
| type Student {  id: ID!  firstName: String!  lastName: String!  email: String  street: String  } |

* Type Student define la estructura de un objeto estudiante.
* El símbolo ! Indica que el campo es obligatorio (no admite valores nulos).
* Campos sin ! son opcionales, como el correo electrónico y la calle.

**Operaciones Disponibles:**

|  |
| --- |
| type Query {  student(id: ID!): Student  allStudents: [Student]  }  type Mutation {  createStudent(firstName: String!, lastName: String!, email: String!): Student  } |

* El tipo Query define **consultas solo de lectura**.
* El tipo Mutation define **operaciones de escrita** (creación, edición, eliminación).
* Los tipos utilizados en las operaciones (Student, String, ID!) también deben estar definidos en el esquema.

# GraphQL Query

Una **GraphQL Query** es una solicitud para obtener datos desde un servidor GraphQL. Equivale a una petición **GET** en REST, pero con más flexibilidad.

Permite al cliente definir **exactamente qué datos necesita.**

**Definición del Schema**: en GraphQL, las consultas se definen en el **schema**.

|  |
| --- |
| type Query {  firstQuery: String  secondQuery: String  } |

**Una consulta GraphQL se realiza con Query Mapping**:

|  |
| --- |
| @Controller  public class Query {  @QueryMapping  public String firstQuery() {  return "First Query";  }  @QueryMapping  public String secondQuery() {  return "Second Query";  }  } |

* **Query**: El cliente puede solicitar los datos con una consulta GraphQL:

|  |
| --- |
| query {  firstQuery  secondQuery  } |

* **Respuesta**: Sólo se devuelven los datos solicitados:

|  |
| --- |
| {  "data": {  "firstQuery": "First Query",  "secondQuery": "Second Query"  }  } |

**~~Vantajas:~~**

* **~~Evita overfetching~~** ~~– Devuelve solo los campos obligatorios~~
* **~~Mayor flexibilidad~~** ~~– El cliente elige exactamente lo que quiere.~~
* **~~Menos peticiones~~** ~~– Puede agrupar varias consultas en una sola llamada.~~

# GraphQL Mutation

Una **Mutation** es una operación para crear, actualizar o eliminar datos. Equivale a los métodos **POST**, **PUT** o **DELETE** en REST.

Las mutaciones se definen en el schema dentro del tipo Mutation, y cada campo representa una operación que puede modificar datos.  
  
**Ejemplo de schema (schema.graphqls):**

|  |
| --- |
| type Mutation {  : StudentResponse  } |

**Una Mutation GraphQL se realiza con MutationMapping**:

|  |
| --- |
| @Controller  public class Mutation {  @Autowired  StudentService studentService;  @MutationMapping  public StudentResponse createStudent(@Argument CreateStudentRequest createStudentRequest) {  return new StudentResponse(studentService.createStudent(createStudentRequest));  }  } |

# ****GraphQL**** Resolver

Un **Resolver** es una función responsable de obtener los datos de un campo específico del schema. Cada vez que se ejecuta una query o mutation, GraphQL llama al resolver correspondiente.

**Como funcionan?**

Cuando se ejecuta una consulta GraphQL, cada campo invoca su Resolver asociado. Estos Resolvers pueden acceder a bases de datos, llamar a API externas o realizar cálculos para obtener los datos requeridos. Pueden devolver valores, objetos o incluso listas de objetos tal como se definen en el esquema GraphQL.

|  |
| --- |
| graphql:  query {  getStudent(id: 1) {  firstName  lastName  fullName  }  } |

**Query Resolver**

|  |
| --- |
| @Controller  public class Query {  @QueryMapping  public StudentResponse getStudent(@Argument Long id) {  return studentService.findById(id);  }  **}** |

**Field Resolver**

|  |
| --- |
| @Controller  public class StudentResponseResolver {  @SchemaMapping(typeName = "StudentResponse", field = "fullName")  public String getFullName(StudentResponse studentResponse) {  return studentResponse.getFirstName() + " " + studentResponse.getLastName();  }  } |

Aquí, el Resolver el nombre de la persona, evitando que se realicen múltiples solicitudes por separado.

**Tipos de Resolvers no GraphQL**

1. **Query Resolver** → Devuelve datos para consultas (Query).
2. **Mutation Resolver** → Para cambios de datos, como creación, actualización, eliminación (Mutation).
3. **Field Resolver** → Para campos complejos o relaciones entre entidades.

**Por qué utilizar resolvers?**

* **Separación de responsabilidades: cada campo tiene su propia lógica**.
* **Ejecución condicional: el backend sólo ejecuta solucionadores para los campos realmente solicitados.**
* **Facilidad de mantenimiento y pruebas**: cada resolver es independiente.
* Evite underfetching e overfetching.

**Buenas práticas**

* Evitar lógica pesada en resolvers.
* Cuidado con el **problema N+1**.
* Usar **DataLoader** o estratégias de agregacion.
* Mantener clara la responsabilidad de cada resolver.

# Problema N+1 en GraphQL

El problema N+1 ocurre cuando se realiza una consulta principal para obtener una lista (N) y luego N consultas adicionales para obtener los datos relacionados.

Esto resulta en **N+1 consultas a la base de datos**, lo cual puede afectar gravemente la **performance**.

**Ejemplo:**

|  |
| --- |
| SELECT \* FROM authors;  SELECT \* FROM books WHERE author\_id = 1;  SELECT \* FROM books WHERE author\_id = 2;  … |

Si hay 100 autores, se hacen **101 consultas**. Esto es el problema N+1.

**Soluciones para o problema N+1:**

* Fetch Join (Eager Loading).
* Batch Loading: agrupar consultas relacionadas.
* Uso de DataLoader: para agrupar consultas: una biblioteca que implementa el patrón de carga por lotes y almacenamiento en caché, lo que permite agrupar y optimizar las consultas.  
    
  Ejemplo de **DataLoader**.

|  |
| --- |
| @Component  public class PostDataLoader {    @Autowired  private PostRepository postRepository;  public DataLoader<Long, List<Post>> getLoader() {  return DataLoader.newMappedDataLoader(userIds ->  CompletableFuture.supplyAsync(() ->  postRepository.findByUserIds(userIds)  )  );  }  } |

**Qué hace este código?**

* Agrupa todas las solicitudes de publicaciones de los usuarios en una única consulta SQL.

**Implementación del Resolver**

Ahora, en el resolver GraphQL, utilizamos DataLoader para obtener las publicaciones correctamente.

|  |
| --- |
| @Component  public class UserResolver {  private final PostDataLoader postDataLoader;  public UserResolver(PostDataLoader postDataLoader) {  this.postDataLoader = postDataLoader;  }  @SchemaMapping(typeName = "User", field = "posts")  public CompletableFuture<List<Post>> getPosts(User user, DataFetchingEnvironment env) {  DataLoader<Long, List<Post>> dataLoader = env.getDataLoader(PostDataLoader.class.getSimpleName());  return dataLoader.load(user.getId());  }  } |

**Qué hace este código?**

* Utiliza **DataLoader** para obtener **todas las publicaciones de una sola vez** en lugar de realizar múltiples consultas individuales.
* **Evita el problema N+1**, haciendo que la API sea mucho más rápida.

# Conclus****i****ón

GraphQL y REST son enfoques distintos de las API, cada uno con sus propias ventajas y desventajas.

En REST, cada recurso se identifica mediante una URL específica y la estructura de los datos devueltos la define el servidor. Por ejemplo, al solicitar /books/1, el servidor devuelve todos los detalles del libro con ID 1, independientemente de qué datos necesite realmente el cliente. Esto puede generar problemas de **overfetching**, donde se transfieren más datos de los necesarios, o de **underfetching**, donde no se proporcionan suficientes datos, lo que requiere múltiples llamadas API.

Por su parte, GraphQL permite al cliente especificar exactamente qué datos desea a través de consultas personalizadas. Por ejemplo, un cliente podría solicitar sólo el título y el autor de un libro, recibiendo exactamente esa información y nada más.

~~Mientras que REST utiliza múltiples puntos finales fijos, cada uno correspondiente a un recurso o acción específicos, lo que puede generar transferencias de datos ineficientes,~~ ~~GraphQL, por otro lado, permite realizar consultas a través de un único punto final, lo que hace que la recuperación de datos sea más eficiente. Este enfoque centralizado simplifica la interacción con la API y puede mejorar la eficiencia en la recuperación de datos.~~

A diferencia de REST, que expone múltiples endpoints según los recursos y acciones disponibles, GraphQL utiliza un único punto de entrada. Lo que realmente mejora la eficiencia en la recuperación de datos es la capacidad de GraphQL para permitir consultas personalizadas, donde el cliente especifica exactamente qué información necesita, evitando así tanto el **overfetching** como el **underfetching**.

# Referencias

* **Documentação Oficial do GraphQL** – <https://graphql.org/>
* **Spring for GraphQL** – <https://spring.io/projects/spring-graphql>
* **Getting started with GraphQL and Spring Boot**  – <https://www.baeldung.com/spring-graphql>
* **GraphQL vs REST** – <https://www.apollographql.com/blog/graphql-vs-rest/>
* **GraphQL API with Java Spring Boot & Spring Framework** – <https://www.udemy.com/course/graphql-api-with-java-spring-boot-and-spring-framework/>
* **Understanding the N+1 Problem in GraphQL and How to Solve It** – <https://engrmuhammadusman108.medium.com/understanding-the-n-1-problem-in-graphql-and-how-to-solve-it-1799e928066a>
* **GraphQL N+1 Problem - Hygraph Blog** – <https://hygraph.com/blog/graphql-n-1-problem>
* **Handling the N+1 Problem in GraphQL - Apollo GraphQL** – <https://www.apollographql.com/docs/graphos/schema-design/guides/handling-n-plus-one>
* **GraphQL N+1 Problem - YouTube** – <https://www.youtube.com/watch?v=uCbFMZYQbxE>
* **Spring GraphQL - API Deprecation List** – <https://docs.spring.io/spring-graphql/docs/current/api/deprecated-list.html>
* **GraphQL Schema - Documentação Oficial** – <https://graphql.org/learn/schema/>
* **Spring GraphQL Reference Guide** – <https://docs.spring.io/spring-graphql/reference/index.html>
* **GraphQL Mutation - Documentação Oficial** – <https://graphql.org/learn/mutations/>  
  **Spring Boot & GraphQL: Guia Completo – YouTube** – <https://www.youtube.com/watch?v=uNB2N_w_ypo>