Sumario

[1. REST APIs 2](#_Toc193981633)

[2. GraphQL vs REST API 2](#_Toc193981634)

[3. GraphQL 3](#_Toc193981635)

[4. Vantagens e Desvantagens: Overfetching e Underfetching 3](#_Toc193981636)

[5. SDL (Schema Definition Language)? 5](#_Toc193981637)

[6. GraphQL Query 7](#_Toc193981638)

[7. GraphQL Mutation 8](#_Toc193981639)

[8. GraphQL Resolver 8](#_Toc193981640)

[9. Problema N+1 no GraphQL 10](#_Toc193981641)

[10. Conclusão 12](#_Toc193981642)

[11. Referências 13](#_Toc193981643)

# REST APIs

As APIs REST são amplamente adotadas para integrar sistemas. Sua estrutura baseia-se em **múltiplos endpoints**, cada um representando um recurso específico, como /users, /products, etc.

**Limitações do REST:**

* **Múltiplos endpoints**: cada recurso exige um endpoint específico.
* **Dados fixos**: o cliente recebe um conjunto de informações que pode conter muito mais (ou menos) do que realmente precisa.
* **Overfetching**: o cliente recebe **mais dados do que precisa**.
* **Underfetching**: o cliente precisa de **várias chamadas** para montar um único objeto.
* **Versionamento frequente** para evolução da API.

# GraphQL vs REST API

GraphQL e REST são duas abordagens amplamente utilizadas para a construção de APIs, cada uma com suas características e vantagens específicas. A tabela a seguir resume as principais diferenças entre elas:​

|  |  |  |
| --- | --- | --- |
| **Característica** | **Rest API** | **GraphQL** |
| Numero de endpoints | Múltiplos endpoints para diferentes recursos. | Um único endpoint (/graphql) para todas as operações. |
| Retorno de dados | Respostas fixas definidas pelo servidor, podendo incluir dados não necessários. | O cliente define exatamente quais dados deseja receber, evitando overfetching. |
| Overfetching | Sim, é comum receber mais dados do que o necessário. | Não, o cliente solicita apenas os dados que precisa. |
| Underfetching | Sim, pode exigir múltiplas requisições para obter todos os dados necessários. | Não, uma única consulta pode recuperar todos os dados necessários. |
| Metodos HTTP | Utiliza métodos HTTP padrão: GET, POST, PUT, DELETE, etc. | Geralmente utiliza apenas POST para enviar consultas (queries) e mutações (mutations). |

**Quando usar GraphQL ao invés de REST?**

* Se você precisa que **clientes escolham os dados** que querem.
* Quando se quer **reduzir múltiplas requisições em APIs complexas.**
* Para permitir **evolução sem versionamento**.

# GraphQL

​O GraphQL é uma linguagem de consulta e postagem de dados baseada em grafos para APIs criada pelo Facebook em 2012 e aberta ao público em 2015. O GraphQL surgiu como uma alternativa às arquiteturas REST tradicionais, visando melhorar a eficiência e a flexibilidade na comunicação entre clientes e servidores

GraphQL é uma linguagem de consulta declarativa que permite ao cliente especificar exatamente quais dados quer obter de uma API, o que torna a evolução das apis mais simples e sem necessidades de versionamento.

**Beneficios:**

* Um único endpoint para todas as operações.
* O cliente define **quais campos** deseja receber.
* **Consulta precisa de dados** (evita excesso ou falta).
* **Evolução da API sem versionamento**.

**GraphQL** **não é**:

* **Um banco de dados** – ele não armazena dados.
* **Um framework** – é uma **especificação**, com implementações em várias linguagens.
* **Exclusivo para HTTP** – apesar de ser comum com HTTP, pode ser usado com WebSocket, TCP, etc.

**Operações Principais:**

1. **Query** → Para buscar dados (**equivalente ao GET em REST**).
2. **Mutation** → Para modificar ou apagar dados (**POST/PUT/DELETE**).
3. **Subscription** → Para receber atualizações em tempo real sobre mudanças nos dados.
4. **Vantagens e Desvantagens: Overfetching e Underfetching**

Um dos maiores diferenciais do GraphQL em relação ao REST é a forma como ele lida com **a seleção e entrega de dados**.

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.

No GraphQL, o cliente decide **exatamente o que quer**, reduzindo chamadas e melhorando a performance.

**Overfetching (Excesso de Dados)?**

Ocorre quando uma API retorna mais dados do que o necessário. Isso significa que o cliente recebe informações que ele não solicitou, resultando em desperdício de largura de banda e processamento desnecessário.

Imagine que um aplicativo precise exibir apenas o nome e o e-mail do utilizador. Se o endpoint /users/{id} retornar **todos os detalhes do usuário**, incluindo endereço, número de telefone e data de nascimento, isso é **Overfetching**.

**Exemplo 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: o cliente **só queria name e email**, mas recebeu muito mais.

**Exemplo GraphQL que evita Overfetching**

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

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

Resultado: **somente os dados necessários são retornados**..

**Underfetching (Falta de Dados)**

Ocorre quando uma API não fornece todos os dados necessários de uma só vez, forçando o cliente a fazer múltiplas requisições encadeadas para obter informações completas.

**Exemplo de underfetching:**

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

Problema: o cliente **precisa combinar múltiplas respostas**.

**Exemplo GraphQL que evita Underfetching**

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

Resultado: **uma única chamada, com estrutura aninhada** e todos os dados desejados.

**Vantagens do GraphQL**

|  |  |
| --- | --- |
| **Vantagem** | **Impacto** |
| Flexibilidade no frontend | Cliente define os campos que quer |
| Documentação e contratos claros | Schema validado, navegável e compatível |
| Redução de Overfetching | Só o necessário é retornado |
| Eliminação de Underfetching | Tudo em uma única consulta |

**Desvantagens do GraphQL**

|  |  |
| --- | --- |
| **Desvantagem** | **Detalhes** |
| Complexidade | Schema + resolvers exigem bom design |
| Caching | Não há URL fixa como em REST |
| Performance ruim se mal implementado | Pode causar Problema N+1 |

# SDL (Schema Definition Language)?

O **SDL** (Schema Definition Language) é a linguagem usada para descrever a estrutura de uma API GraphQL. Ela é independente da linguagem de programação usada no 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 com SDL:**

* **Tipos de dados (**type**)**
* **Campos e seus tipos** (String, Int, ID, Boolean, etc.)
* **Campos obrigatórios ou opcionais**
* **Consultas (Query)**
* **Mutações (Mutation)**
* **Assinaturas em tempo real (Subscription)**
* **Entradas personalizadas (input)**
* **Outros elementos**: enum, interface, union, fragment, entre outros

**Exemplo:**

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

* Type Student define a estrutura de um objeto de estudante.
* O símbolo ! indica que o campo é **obrigatório** (non-nullable).
* Campos sem ! são **opcionais**, como email e street.

**Operações Disponíveis:**

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

* O tipo Query define **consultas de leitura**.
* O tipo Mutation define **operações de escrita** (criação, edição, remoção).
* Os tipos usados nas operações (Student, String, ID!) também devem ser definidos no schema.

# GraphQL Query

Uma **GraphQL Query** é uma solicitação para recuperar dados de um servidor GraphQL. Ela é equivalente a uma requisição **GET** em APIs REST, mas com a vantagem de permitir ao cliente definir exatamente quais dados deseja receber. Isso evita **overfetching** (receber dados desnecessários) e **underfetching** (não receber dados suficientes), tornando as requisições mais eficientes.

**Definição do Schema**: No GraphQL, as queries são definidas no **schema**.

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

**Implementação no Servidor**: O backend deve definir como os dados serão retornados.

No Java, isso pode ser feito com um **Query Mapping**:

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

* **Execução da Query**: O cliente pode solicitar os dados com uma query GraphQL:

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

* **Resposta do Servidor**: Apenas os dados solicitados são retornados:

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

**Vantagens:**

* **Evita overfetching** – Retorna apenas os campos necessários.
* **Maior flexibilidade** – O cliente escolhe exatamente o que quer.
* **Menos requisições** – Pode agrupar múltiplas consultas em uma única chamada.

# GraphQL Mutation

No GraphQL, uma **Mutation** é uma operação utilizada para **criar, atualizar** ou **apagar** dados no servidor, permitindo que os clientes modifiquem o estado dos dados. As mutações são análogas aos métodos **POST, PUT e DELETE** em APIs REST.

As Mutation são definidas no schema GraphQL dentro do tipo Mutation. Cada campo dentro deste tipo representa uma operação que pode ser executada para modificar os dados.  
  
**Exemplo de Schema GraphQL para Mutation (schema.graphqls):**

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

**Implementação de um Resolver de Mutação em Java**

Para que a mutação funcione, é necessário implementar um resolver que trate a lógica correspondente.

**Código Java - MutationMapping**:

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

# ****GraphQL**** Resolver

Um **Resolver** é uma função responsável por fornecer os dados correspondentes a um campo específico no schema.

Toda vez que uma query ou mutation é executada, o GraphQL **mapeia cada campo solicitado para um resolver** que retorna os dados correspondentes.

**Como funcionam os Resolvers?**

Quando uma consulta GraphQL é executada, cada campo invoca seu resolver associado. Esses resolvers podem acessar bancos de dados, chamar APIs externas ou realizar cálculos para obter os dados necessários. Eles podem retornar valores, objetos ou até mesmo listas de objetos, conforme definido no 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();  }  } |

Aqui, o resolver busca **as matérias que o estudante está aprendendo**, evitando que múltiplas requisições sejam feitas separadamente.

**Tipos de Resolvers no GraphQL**

1. **Query Resolver** → Retorna dados para queries (Query).
2. **Mutation Resolver** → Manipula dados (criação, atualização, remoção) dentro das mutations (Mutation).
3. **Field Resolver** → Resolve dados complexos de um objeto, como relacionamentos entre entidades.

**Por que usar resolvers?**

* **Separação de responsabilidades**: cada campo tem sua lógica própria.
* **Execução condicional**: o backend só executa resolvers para os campos realmente solicitados.
* **Facilidade de manutenção e teste**: cada resolver é independente.
* Evitam underfetching e overfetching, melhorando a eficiência.

**Cuidados com resolvers**

* Evitar lógica pesada em resolvers simples.
* **Evitar o problema N+1**: resolvers aninhados que acessam o banco em loop.
* Usar **DataLoader** ou agregações eficientes.
* Manter responsabilidade clara: não misturar resolvers com lógica de negócio densa.

# Problema N+1 no GraphQL

O problema N+1 ocorre quando uma aplicação realiza uma consulta inicial para obter uma lista de itens (N) e, em seguida, executa N consultas adicionais para buscar dados relacionados a cada item individualmente. Isso resulta em **N+1 consultas** ao banco de dados, causando ineficiências e impactando negativamente o desempenho da aplicação.

**Exemplo do problema N+1:**

Considere uma aplicação que precisa listar autores e seus respetivos livros:​

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

Se houver 100 autores, isso resultará em 101 consultas ao banco de dados, caracterizando o problema N+1.

**Soluções para o problema N+1:**

* Fetch Join (Eager Loading).
* Batch Loading: Agrupa múltiplas requisições em uma única consulta ao banco de dados, reduzindo o número total de consultas.
* Uso de DataLoader para agrupar as consultas: Uma biblioteca que implementa o padrão de carregamento em lote e caching, permitindo que consultas sejam agrupadas e otimizadas.  
    
  Exemplo de um **DataLoader** para buscar **posts de vários Utilizadores em lote** ao invés de um por um.

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

**O que esse código faz?**

* Agrupa **todas** as solicitações de posts de usuários **em uma única consulta** SQL.
* Usa CompletableFuture para processar as consultas **de forma assíncrona** e eficiente.

**Implementando o Resolver no Spring GraphQL**

Agora, no **resolver GraphQL**, usamos o **DataLoader** para buscar os posts corretamente.

|  |
| --- |
| @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());  }  } |

**O que esse código faz?**

* Usa o **DataLoader** para buscar **todos os posts de uma só vez** em vez de fazer várias consultas individuais.
* **Evita o problema N+1**, tornando a API muito mais rápida.

# Conclus****ã****o

GraphQL e REST são abordagens distintas para APIs, cada uma com suas vantagens e desvantagens.

No REST, cada recurso é identificado por um URL específico, e a estrutura dos dados retornados é definida pelo servidor. Por exemplo, ao solicitar /books/1, o servidor retorna todos os detalhes do livro com ID 1, independentemente de quais dados o cliente realmente necessita. Isso pode levar ao problema de **overfetching**, onde mais dados do que os necessários são transferidos, ou **underfetching**, onde dados insuficientes são fornecidos, exigindo múltiplas chamadas à API.

Por outro lado, o GraphQL permite que o cliente especifique exatamente quais dados deseja através de consultas personalizadas. Por exemplo, um cliente pode solicitar apenas o título e o autor de um livro específico, recebendo exatamente essas informações e nada mais. Essa flexibilidade reduz o overfetching e o underfetching, tornando as operações mais eficientes.

Enquanto o REST usa múltiplos endpoints fixos, cada um correspondendo a um recurso ou ação específica, o que pode levar a transferências ineficientes de dados, por outro lado o GraphQL permite consultas por meio de um único endpoint, tornando a recuperação de dados mais eficiente. Essa abordagem centralizada simplifica a interação com a API e pode melhorar a eficiência na recuperação de dados.

# Referências

* **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>