Sumario

[1. Spring Boot App com REST APIs 2](#_Toc193358757)

[3. GraphQL vs REST API 4](#_Toc193358758)

[4. O que é GraphQL? 4](#_Toc193358759)

[5. O que é GraphQL Schema? 5](#_Toc193358760)

[6. O que é GraphQL Query? 5](#_Toc193358761)

[7. O que é GraphQL Mutation? 6](#_Toc193358762)

[8. O que é GraphQL Resolver? 7](#_Toc193358763)

[9. O que é Spring GraphQL? 7](#_Toc193358764)

[10. GraphQL N+1 Problem 8](#_Toc193358765)

[11. Conclusao 10](#_Toc193358766)

[12. Referencias 11](#_Toc193358767)

# Spring Boot App com REST APIs

**Por que falar de REST antes de GraphQL?**  
Muitas aplicações já utilizam REST e querem migrar para GraphQL.  
REST segue uma abordagem baseada em múltiplos endpoints fixos, enquanto GraphQL tem um único endpoint.  
  
Exemplo de Endpoint REST (Padrão Antigo):

|  |
| --- |
| @RestController  @RequestMapping("/users")  public class UserController {    @GetMapping("/{id}")  public ResponseEntity<User> getUser(@PathVariable Long id) {  return ResponseEntity.ok(userService.getUserById(id));  }  } |

**Limitações do REST:**

* Múltiplos endpoints para diferentes operações.
* Retorna dados fixos, sem personalização para cada cliente.
* Problemas de performance devido ao Overfetching e Underfetching.

1. **Overfetching e Underfetching**

**Overfetching (Excesso de Dados)**

Ocorre quando uma API retorna mais dados do que o necessário para um cliente. 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 resposta de uma REST API que sofre de Overfetching**

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

**Exemplo de Query GraphQL que evita Overfetching**

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

Com GraphQL, o cliente recebe **apenas** os campos solicitados.

**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 para obter informações completas.

Imagine um cenário onde queremos exibir **um usuário e seus pedidos**. Em uma REST API tradicional, precisaríamos:

- Fazer uma requisição para /users/{id} para obter os dados do usuário.

- Fazer outra requisição para /users/{id}/orders para obter seus pedidos.

**GET /users/1**

|  |
| --- |
| {  "id": 1,  "name": "Carlos",  "email": "carlos@email.com"  } |

**GET /users/1/orders**

|  |
| --- |
| [  { "id": 101, "total": 200 },  { "id": 102, "total": 150 }  ] |

**Exemplo de Query GraphQL que evita Underfetching**

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

O cliente recebe **todas** as informações necessárias em **uma única requisição**.

# GraphQL vs REST API

|  |  |  |
| --- | --- | --- |
| **Característica** | **Rest API** | **GraphQL** |
| Numero de endpoints | Múltiplos endpoints | Apenas um endpoint /graphql |
| Retorno de dados | Respostas fixas | O cliente define quais dados deseja |
| Overfetching | Sim | Não |
| Underfetching | Sim | Não |
| Metodos HTTP | GET, POST, PUT, DELETE | Apenas Query e Mutation |

**Quando usar GraphQL ao invés de REST?**  
- Quando há necessidade de flexibilidade na recuperação de dados.  
- Para reduzir múltiplas requisições em APIs complexas.

# O que é GraphQL?

​O GraphQL é uma linguagem de consulta para APIs e um ambiente de execução que permite aos clientes solicitarem exatamente os dados de que necessitam. Desenvolvido pelo Facebook em 2012 e tornado 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.

**Principais Características:**

* **Solicitação Precisa de Dados:** Os clientes podem especificar exatamente quais dados precisam, evitando o retorno de informações excessivas ou insuficientes. ​
* **Uso de um Único Endpoint:** Diferente do REST, que utiliza múltiplos endpoints para diferentes recursos, o GraphQL opera através de um único endpoint para todas as operações.
* **Evolução Sem Versionamento:** Novos campos e tipos podem ser adicionados ao esquema GraphQL sem impactar clientes existentes, eliminando a necessidade de versionamento da API.
* **Sistema de Tipagem Forte:** Significa que **cada dado dentro de uma API GraphQL tem um tipo bem definido**, garantindo que as operações sigam um formato consistente e seguro. Isso ajuda a evitar erros, facilita a validação e melhora a documentação da API.

**Exemplo de Tipagem Forte**

|  |
| --- |
| type User {  id: ID!  name: String!  age: Int  email: String!  } |

* ID! → Representa um identificador único e obrigatório (! indica que não pode ser nulo).
* String! → Representa um texto obrigatório.
* Int → Representa um número inteiro, podendo ser opcional.

**Principais Operações no GraphQL**

1. **Query** → Para buscar dados (**equivalente ao GET em REST**).
2. **Mutation** → Para modificar dados (**criar, atualizar, deletar**).
3. **Subscription** → Para receber atualizações em tempo real sobre mudanças nos dados.

# O que é GraphQL Schema?

- Define quais atributos existem e seus tipos de dados.  
- Utiliza a extensão .graphqls.  
- Estabelece um contrato entre cliente e servidor.  
  
**Benefícios do Schema-First:**

* Contratos Claros: O schema garante consistência entre backend e frontend.
* Validação Automática: Queries e mutations são validadas antes da execução.

**Exemplo de Schema GraphQL**:

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

# O que é 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** (.graphql).

|  |
| --- |
| 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 Resolver (GraphQLQueryResolver)**:

|  |
| --- |
| @Component  public class Query implements GraphQLQueryResolver {  public String firstQuery() {  return "First Query";  }  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.

# O que é GraphQL Mutation?

- Mutation é usada para criar, atualizar ou deletar dados.  
- Equivalente aos métodos POST, PUT e DELETE em REST.  
  
**Exemplo de Schema GraphQL para Mutation (schema.graphqls):**

|  |
| --- |
| type Mutation {  createUser(name: String!, email: String!): User  } |

**Código Java - Mutation Resolver (Usa GraphQLMutationResolver)**:

|  |
| --- |
| @Service  public class Mutation implements GraphQLMutationResolver {  @Autowired  StudentService studentService;  public StudentResponse createStudent(CreateStudentRequest request) {  return new StudentResponse(studentService.createStudent(request));  }  } |

# O que é GraphQL Resolver?

- Um GraphQL Resolver é responsável por processar e retornar os dados para cada campo consultado.  
- Ele atua como um intermediário entre a API GraphQL e o banco de dados.  
  
**Exemplo de Resolver para evitar Underfetching (Falta de Dados):**

|  |
| --- |
| @Service  public class StudentResponseResolver implements GraphQLResolver<StudentResponse> {  public List<SubjectResponse> getLearningSubjects(StudentResponse studentResponse) {  List<SubjectResponse> learningSubjects = new ArrayList<>();  if (studentResponse.getStudent().getLearningSubjects() != null) {  for (Subject subject : studentResponse.getStudent().getLearningSubjects()) {  learningSubjects.add(new SubjectResponse(subject));  }  }  return learningSubjects;  }  } |

# O que é Spring GraphQL?

O Spring GraphQL é um projeto oficial do Spring que facilita a criação de APIs GraphQL.  
  
**Os GraphQL Schemas são essenciais no Spring GraphQL porque:**  
- Definem os tipos e operações da API.  
- Garantem contratos claros entre Backend e Frontend.  
- Permitem validação automática e documentação embutida.  
  
**Exemplo de Query no Spring GraphQL:**

|  |
| --- |
| @Controller public class UserController {  @QueryMapping  public List<User> listUsers() {  return List.of(new User(1L, "Alice"), new User(2L, "Bob"));  } } |

**Exemplo de Mutation no Spring GraphQL**:

|  |
| --- |
| @Controller  public class UserController {  @MutationMapping  public User createUser(@Argument String name, @Argument String email) {  return new User(name, email);  }  } |

**Exemplo de Query Resolver no Spring GraphQL**:

|  |
| --- |
| @Component  public class StudentResponseResolver {  @SchemaMapping(typeName = "StudentResponse", field = "learningSubjects")  public List<SubjectResponse> getLearningSubjects(StudentResponse studentResponse) {  return studentResponse.getStudent().getLearningSubjects();  }  } |

# GraphQL N+1 Problem

O que é o problema N+1?

- Ocorre quando uma **query inicial busca uma lista de itens (N)** e, depois, realiza **N consultas adicionais** para cada item.

- Isso resulta em **múltiplas consultas desnecessárias** e **impacta a performance**.  
  
**Soluções para o problema N+1:**

* Fetch Join (Eager Loading).
* Batch Loading.
* Uso de DataLoader para agrupar as consultas.  
    
  **Exemplo com DataLoader:**

|  |
| --- |
| @Component  public class CommentDataLoader {  @Autowired  private CommentRepository commentRepository;  public CompletableFuture<List<Comment>> load(Long postId) {  return CompletableFuture.supplyAsync(() -> commentRepository.findByPostId(postId));  }  } |

# Conclusao

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.

# Referencias

* **Documentação Oficial do GraphQL** – [https://graphql.org/](https://graphql.org/" \t "_new)
* **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** – [Udemy](https://www.udemy.com/course/graphql-with-java-spring-boot-query-mutation-schema-resolver-edge-jpa/)