# GraphQL com Spring Boot

## 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?

GraphQL (Graph Query Language) foi desenvolvido pelo Facebook para melhorar a comunicação entre clientes e servidores.  
  
**Principais Características:**  
✔ Permite que o cliente solicite exatamente os dados que precisa.  
✔ Evita overfetching e underfetching, melhorando a performance.  
✔ Usa um único endpoint para todas as operações.  
✔ Suporta múltiplas linguagens: Java, Python, PHP, Ruby, etc.  
  
**Principais Métodos no GraphQL:**  
- Query → Para buscar dados (equivalente ao GET em REST).  
- Mutation → Para modificar dados (Criar, Atualizar, Deletar).

## O que é GraphQL Query?

- Query é utilizada para buscar dados da API (equivalente ao método GET em REST).  
- Retorna apenas os dados solicitados, evitando desperdício de dados.  
  
**Exemplo de Schema GraphQL para Query (schema.graphqls):**

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

**Código Java - Query Resolver (Usa GraphQLQueryResolver):**

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

**Exemplo de Query GraphQL para buscar os dados**:

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

**Resposta esperada:**

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

**Vantagens:**

Evita **overfetching** (retorno excessivo de dados).

Flexibilidade para escolher exatamente os **campos desejados**.

**Reduz o número de requisições** ao backend.

## 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 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 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));  }  } |