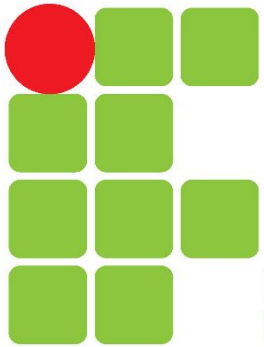
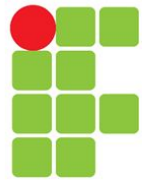




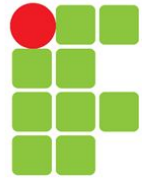
Introdução ao GraphQL com Spring Framework

Desenvolvimento Web Backend (BRADWBK)

Prof. Luiz Gustavo Diniz de Oliveira Véras

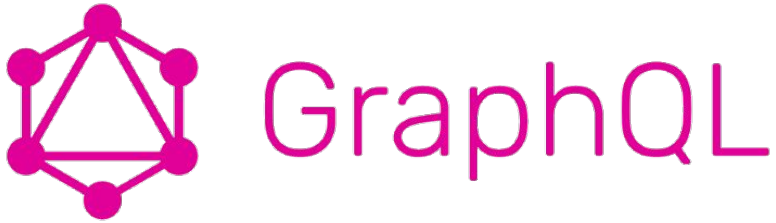
E-mail: gustavo_veras@ifsp.edu.br

Objetivos



- ✔ Introdução ao GraphQL
- ✔ Schemas
- ✔ Consultas
- ✔ Dependências GraphQL para Spring
- ✔ Projeto Spring para GraphQL
- ✔ Definindo o Schema no Projeto Spring
- ✔ Anotações no SpringBoot para GraphQL
- ✔ Testando consultas

Web API



GraphQL

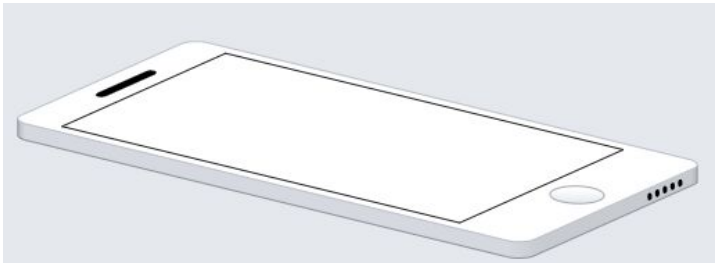
É uma linguagem de consulta (*Query Language*) para APIs que ganhou bastante popularidade recentemente.

Foi desenvolvido internamente pelo Facebook em 2012 antes de ser lançado publicamente em 2015 e foi adotado pela API provedores como GitHub, Yelp e Pinterest.

GraphQL permite que os clientes definam a estrutura dos dados necessários, e o servidor retorna essa estrutura.

Introdução ao GraphQL

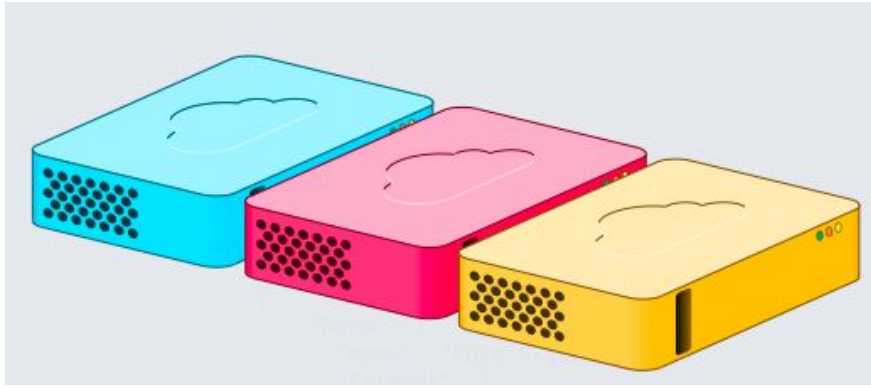
GraphQL



```
{
  user(login: "saurabhsahni") {
    id
    name
    company
    createdAt
  }
}
```

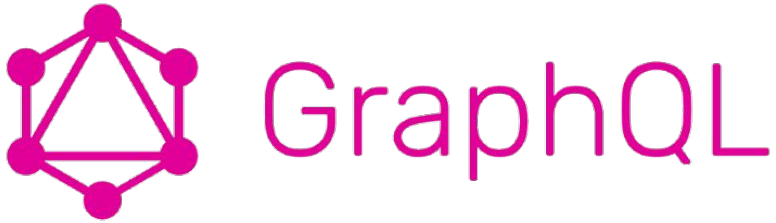
Requisição

Resposta



```
{
  "data": {
    "user": {
      "id": "MDQ6VXNlcjY1MDI5",
      "name": "Saurabh Sahni",
      "company": "Slack",
      "createdAt": "2009-03-19T21:00:06Z"
    }
  }
}
```

Introdução ao GraphQL



GraphQL

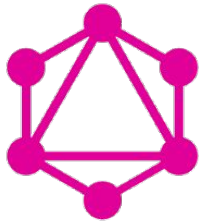
Vantagens

- O GraphQL permite que os clientes aninhem consultas e busquem dados em recursos em uma única solicitação. Com REST seriam necessárias várias.
- Você pode adicionar novos campos e tipos a uma API GraphQL sem afetar as consultas existentes. Com REST essa modificação é mais difícil.
- Com o GraphQL os clientes podem especificar exatamente o que eles precisam, os tamanhos de carga útil podem ser menores. Com REST, às vezes a resposta contém dados desnecessários.
- GraphQL é fortemente tipado, o que reduz a possibilidade de erros no cliente.
- GraphQL possui descoberta nativa. No REST precisamos utilizar alguma ferramenta, como o Swagger.

Desvantagens

- O servidor precisa fazer processamento adicional para analisar consultas complexas e verificar parâmetros.
- Otimizar o desempenho das consultas do GraphQL pode ser difícil, principalmente quando não se sabe quais consultas (querys) usuários externos irão solicitar.

Introdução ao GraphQL



GraphQL

GraphQL

Expert Advice

One of the biggest issues GitHub saw was REST payload creep. Over time, you add additional information to a serializer for, say, a repository. It starts small but as you add additional data (maybe you've added a new feature) that primitive ends up producing more and more data until your API responses are enormous.

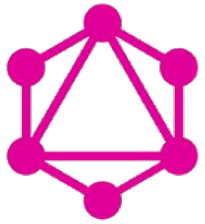
We've tackled that over the years by creating more endpoints, allowing you to specify you'd like the more verbose response, and by adding more and more caching. But, over time, we realized we were returning a ton of data that our integrators didn't even want. That's one of several reasons we've been investing in our GraphQL API. With GraphQL, you specify a query for just the data you want and we return just that data.

—Kyle Daigle, director of ecosystem engineering at GitHub

Introdução ao GraphQL

GraphQL – Consulta e Resposta Simples

Introdução ao GraphQL



GraphQL

GraphQL (Consultas mais complexas)

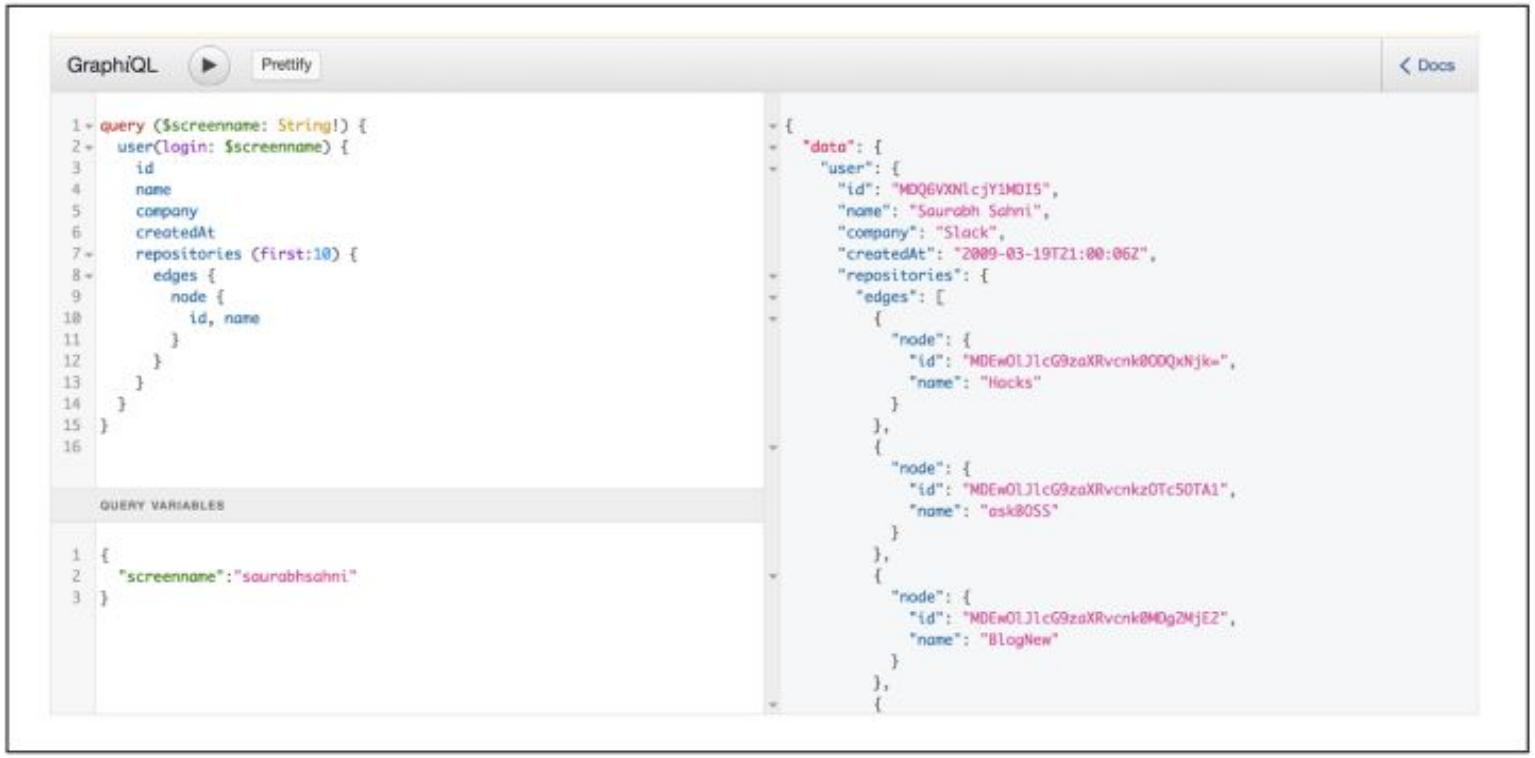


Figure 2-2. GraphiQL: GitHub's GraphQL explorer showing a complex query

Introdução ao GraphQL

Comparando com API REST

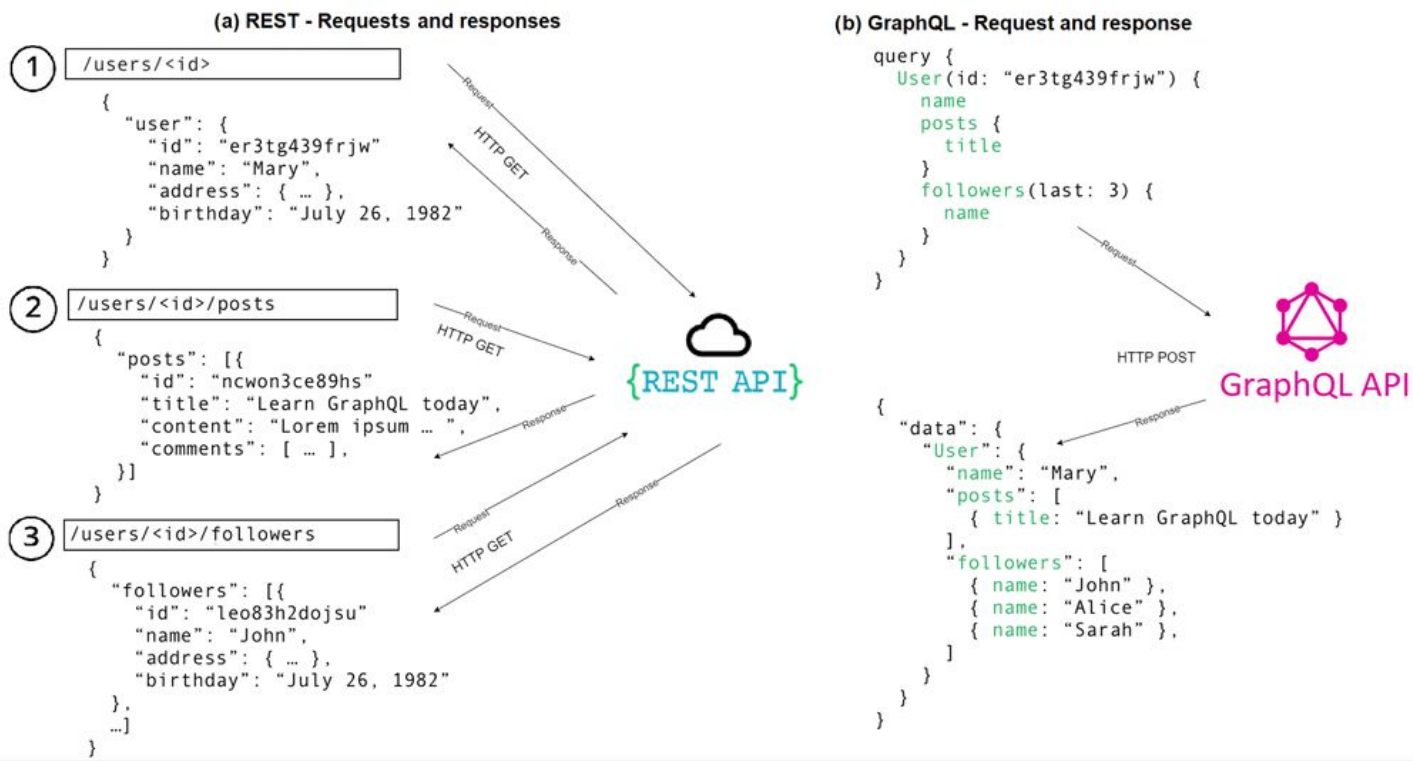
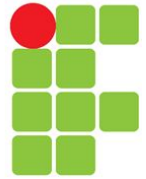


Fig. 1. Example of over-fetching and under-fetching in REST API vs. GraphQL API [44].

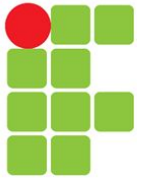
Introdução ao GraphQL



Antes de iniciarmos a implementação no Spring, vamos conhecer alguns conceitos de GraphQL.

- Schema (Estrutura de dados)
- Query (Consulta)
- Mutation (Modificações)

GraphQL Schemas



Um serviço GraphQL é criado definindo tipos e seus campos e, em seguida, escrevendo uma função para cada campo para fornecer os dados necessários.

Por exemplo, um serviço GraphQL que informa o nome de um usuário conectado pode ter a aparência ao lado.

Como a forma de uma consulta GraphQL corresponde ao *Schema*, podemos prever o que a consulta retornará sem saber muito sobre o servidor.

```
type Query {
  me: User
}

type User {
  name: String
}
```

GraphQL Schemas

```
type Query {
  me: User
}

type User {
  name: String
}
```

Exemplo 1: Realizando uma consulta para o Schema anterior (tipo um dentro do outro).

Montagem da estrutura do dado na consulta

```
{
  me {
    name
  }
}
```

Corpo da resposta com a estrutura solicitada.

```
{
  "data": {
    "me": {
      "name": "Luke Skywalker"
    }
  }
}
```

GraphQL Schemas

Exemplo 2: Com base no esquema ao lado **(Hero)**.

```
type Hero{
   name: String
   id: String
}
```

A alteração da consulta ...

```
{
  hero {
    name
    # add additional fields here!
  }
}
```

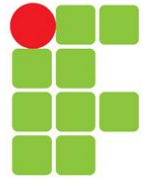
... altera o corpo da resposta com a estrutura solicitada.

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

```
{
  hero {
    name
    id
    # add additional fields here!
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "id": "2001"
    }
  }
}
```

GraphQL Schemas



Tipos de dados para a montagem de Schemas

O esquema descreve os dados e suas relações.

Ele inclui seis tipos principais: **Object, Scalar (Int, Float, String, Boolean, ID), Enum, Interface, Union e Input Object. (Não veremos todos)**

Temos também **Modificadores de Tipo,** que permitem ajustar o comportamento padrão dos tipos, como **List** (para arrays) e **Non-Null** (para valores obrigatórios).

Os esquemas também podem dar suporte a operações, adicionando tipos adicionais e, em seguida, definindo campos nos tipos de operação raiz correspondentes.

- **mutation, subscription, MutationSubscription**

GraphQL Schemas

Cada esquema do GraphQL deve suportar operações. O *ponto de entrada* para esse tipo de operação raiz é um tipo de objeto regular chamado por padrão.

Precisamos de um tipo Query...

```
type Query {
  droid(id: ID!): Droid
}
```

```
type Droid{
  name: String
  id: ID
}
```

... para que a consulta funcione

```
{
  droid(id: "2000") {
    name
  }
}
```

GraphQL Consultas

Query: Numa consulta (*Query*) podem ser consideradas diferentes recursos:

Campos: Em sua forma mais simples, o GraphQL trata de solicitar campos específicos em objetos.

Se o tipo Especificado no Esquema é esse (com Character possuindo "id" e "name")

```
type Query {
  hero: Character
}
```

```
type Character{
  id: String,
  name: String
}
```

... Podemos realizar essa consulta

```
{
  hero {
    id
    name
  }
}
```

GraphQL Consultas

Query: Numa consulta podem ser consideradas diferentes recursos:

Argumentos: Podemos passar argumentos nas consultas.

Human vai receber um ID que não deve ser nulo (o ! de *ID!* Indica isso)

```
type Query {
  human(id: ID!): Human
}
```

```
type Human{
  id: ID
  name: String,
  height: int
}
```

... agora podemos realizar consulta passando argumento

```
{
  human(id: "1000") {
    name
    height
  }
}
```

GraphQL Consultas

Query: Numa consulta podem ser consideradas diferentes recursos:

Argumentos: Podemos passar argumentos nas consultas.

Human vai receber um ID que não deve ser nulo (o ! de *ID!* Indica isso)

```
type Query {
  human(id: ID!): Human
}
```

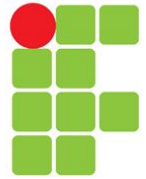
type Human{
id: ID
name: String,
height: int

... agora podemos realizar consulta passando argumento

```
{
  human(id: "1000") {
    name
    height
  }
}
```

! representa que o parâmetro deve ser não-nulo.

GraphQL Consultas



Mutações: operações para modificar dados no servidor.

Elas permitem:

- **Adicionar Dados:** *Mutations* permitem criar novos dados, como adicionar uma avaliação a um filme.
- **Atualizar Dados:** É possível modificar dados existentes, como alterar o nome de um personagem.
- **Remover Dados:** *Mutations* também podem ser usadas para deletar dados, como excluir uma nave espacial.
- **Execução Serial**: Diferente das queries, os campos de mutations são executados em série, garantindo que uma operação termine antes da próxima começar.

GraphQL Consultas

Adicionar Dados

```
enum Episode {
    NEWHOPE
    EMPIRE
    JEDI
}


input ReviewInput {
     stars: Int!
     commentary: String
}


type Mutation {
     createReview(episode: Episode, review: ReviewInput!):Review
}
```

Enum é um tipo do esquema.

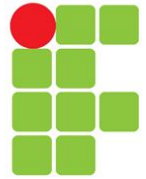
ReviewInput define um tipo composto para ser usada argumento pela *mutation*.

createReview é a *mutation*.

Ela tem como argumentos o Enum e o ReviewInput.

Ele irá salvar uma Review.

GraphQL Consultas



Adicionar Dados

```
enum Episode {
    NEWHOPE
    EMPIRE
    JEDI
}


input ReviewInput {
     stars: Int!
     commentary: String
}


type Mutation {
´     createReview(episode: Episode, review: ReviewInput!):Review
}
```

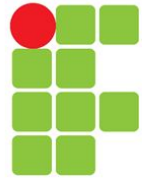
CONSULTA

```
mutation {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

VARIÁVEL (Tudo que está referenciado com $ na consultas)

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

GraphQL Consultas



Mutações: operações para modificar dados no servidor.

Atualizar Dados e Remover Dados é idêntico à **Adicionar Dados,** basta definir as mutações próprias para tal.

Atualizar Dados

```
type Mutation { updateHumanName(id: ID!, name: String!): Human}
```

Remover Dados

```
type Mutation { deleteStarship(id: ID!): ID!}
```

Dependências GraphQL para Spring

O GraphQL não está vinculado a nenhum **banco de dados** ou mecanismo de armazenamento específico — ele é apoiado por seu código e dados existentes. Portanto podemos utilizar qualquer **linguagem de programação** que lhe dê suporte. Vamos utilizar o **Spring**

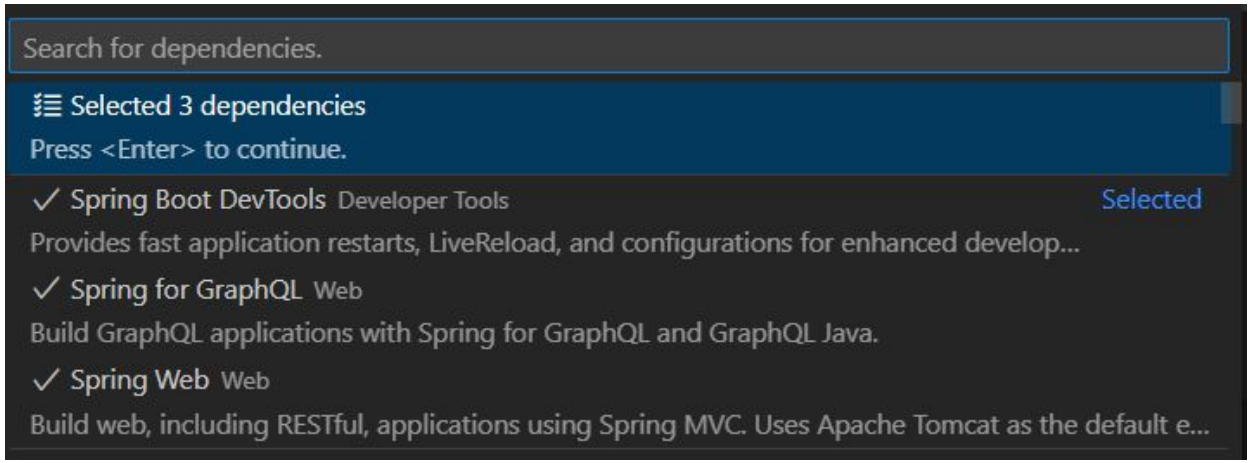
GraphQL é independente do meio de transporte utilizado para comunicação (*transport-agnostic*) . Isso significa que ele não está limitado a um protocolo específico, como HTTP, e pode ser implementado em diferentes tipos de transporte, como WebSocket ou WebFlux (processamento reativo que é **assíncrono e não bloqueante**).

Utilizaremos o **HTTP**. Mas se quiser adicionar outra forma de transporte, então a sua dependência deve ser adicionada.

Starter	Transport	Implementation
`spring-boot-starter-web`	HTTP	Spring MVC
`spring-boot-starter-websocket`	WebSocket	WebSocket for Servlet apps
`spring-boot-starter-webflux`	HTTP, WebSocket	Spring WebFlux
`spring-boot-starter-rsocket`	TCP, WebSocket	Spring WebFlux on Reactor Netty

Dependências GraphQL para Spring

Para um projeto novo, selecionar dependências com o Spring Init

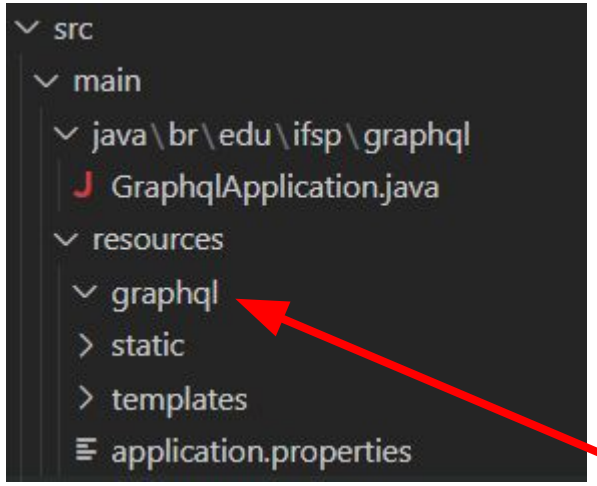


Pode-se adicionar diretamente no *pom.xml*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

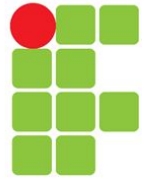
Dependências GraphQL para Spring

Em um projeto temos algumas pastas específicas.



Uma aplicação Spring GraphQL precisa de um esquema definido no momento da inicialização. Por padrão, você pode criar arquivos de esquema com as extensões **".graphqls"** ou **".gqls"** e colocá-los no diretório **src/main/resources/graphql/**. O Spring Boot detectará esses arquivos automaticamente e os usará para configurar a API GraphQL.

Definindo o Schema no Projeto Spring



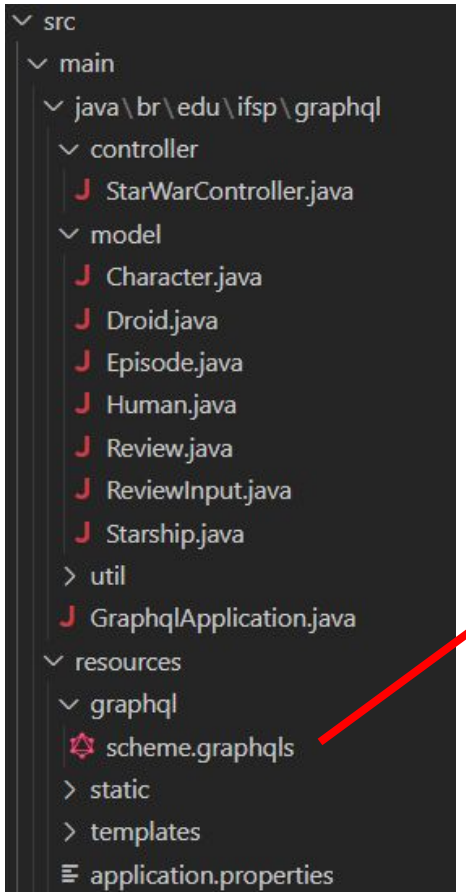
1. Ao criar um *documento* GraphQL, sempre começamos com um tipo de operação raiz (o tipo Object para este exemplo) porque ele serve como um ponto de entrada para a API.

2. A partir daí, devemos especificar o *conjunto de seleção* de campos nos quais estamos interessados, até seus valores folha, que serão do tipo Escalar ou Enum.

Assim, as consultas do GraphQL poderão atravessar objetos relacionados e seus campos, permitindo que os clientes busquem muitos dados relacionados em uma solicitação, em vez de fazer várias viagens de ida e volta, como seria necessário em uma arquitetura REST clássica.

Definindo o Schema n

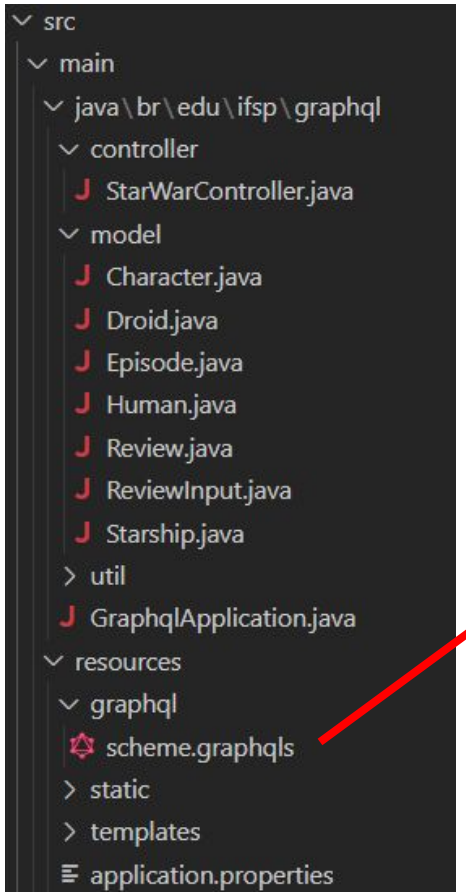
No nosso projeto de exemplo, utilizaremos classes de filmes do StarWars




```
# :::::::::::::::: Tipos da API ::::::::::::::::::::

enum Episode{
    NEWHOPE
    EMPIRE
    JEDI
}

interface Character{
    id: ID! # tipo que representa um ID não nulo
    name: String! # aceita string não null
    appearsIn: [Episode]! # aceita uma lista de Episodes não nulos
    friends: [Character] # aceita lista de Characters. Podem ser nulos
}

type Droid implements Character{
    id: ID!
    name: String!
    appearsIn: [Episode]!
    friends: [Character]
    primaryFunction: String
}

type Human implements Character {
  id: ID!
  name: String!
  appearsIn: [Episode]!
  friends: [Character]
  height: Float
}

type Starship {
  id: ID!
  name: String!
  length: Float
}

type Review {
  stars: Int!
  commentary: String
}
```

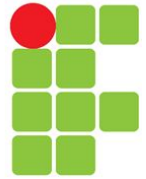
Definindo o Schema n

No nosso projeto de exemplo, utilizaremos classes de filmes do StarWars




```
# :::::::::::::::: Tipo para Inputs ::::::::::::::::::::

input ReviewInput{
    stars: Int!
    commentary: String
}

# :::::::::::::::: Querys ::::::::::::::::::::

type Query {
  hero(episode: Episode): Character #Será o mesmo nome no
método Java
  droid(id: ID!): Droid # Será o mesmo nome no método Java
  search(text: String!): [SearchResult!]! # Será o mesmo
nome no método Java
}

# :::::::::::::::: Mutations ::::::::::::::::::::

type Mutation {
    createReview(episode: Episode!, review: ReviewInput!):
Review # Será o mesmo nome no método Java
}

# :::::::::::::::: Union (Consulta combinada)
::::::::::::::::::::

union SearchResult = Human | Droid | Starship
```

Anotações no SpringBoot



O Spring GraphQL utiliza um modelo baseado em anotações para simplificar a criação de APIs GraphQL. As anotações ajudam a mapear métodos e classes para os campos e tipos definidos no esquema GraphQL.

Utilizaremos:

- @**Controller** => Já a conhecimentos do próprio spring
- @**QueryMapping** => Define o mapeamento de uma consulta para um método Java.
- @**Argument** => Define o mapeamento de um argumento para um parâmetro em um método Java.
- @**MutationMapping** => Define o mapeamento de uma mutation para um método Java que irá receber dados.

Definindo o Schema no Projeto Spring

No nosso projeto de exemplo, utilizaremos classes de filmes do StarWars


```java
@Controller
public class StarWarController {

    /*
     * Mapeado no resources/graphql/scheme.graphqls
     *
     * type Query {
     *     hero(episode: Episode): Character
     * }
     */
    @QueryMapping
    public Character hero(@Argument Episode episode) {
        return new Droid(
                "2001",
                "R2-D2",
                List.of(Episode.NEWHOPE, Episode.EMPIRE, Episode.JEDI),
                List.of(),
                "Astromech"
        );
    }
```

Definindo o Schema no Projeto Spring

No nosso projeto de exemplo, utilizaremos classes de filmes do StarWars

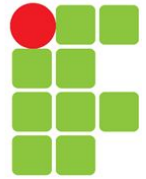

```java
/*
     * Mapeado no resources/graphql/scheme.graphqls
     *
     * type Query {
     *     droid(id: ID!): Droid
     * }
     */
    @QueryMapping
    public Droid droid(@Argument String id) {
        return new Droid(
            id,
            "R2-D2",
            List.of(Episode.NEWHOPE, Episode.EMPIRE, Episode.JEDI),
            List.of(),
            "Astromech"
        );
    }

    /*
     * Mapeado no resources/graphql/scheme.graphqls
     *
     * type Query {
     *      search(text: String!): [SearchResult!]!
     * }
     */
    @QueryMapping
    public List<Object> search(@Argument String text) {
        return List.of(
                new Droid("2001", "R2-D2", List.of(), List.of(), "Astromech"),
                new Human("1001", "Luke", List.of(), List.of(), 1.72f),
                new Starship(3000, "Millenium Falcon", 1000));
    }
```

Definindo o Schema no Projeto Spring

No nosso projeto de exemplo, utilizaremos classes de filmes do StarWars

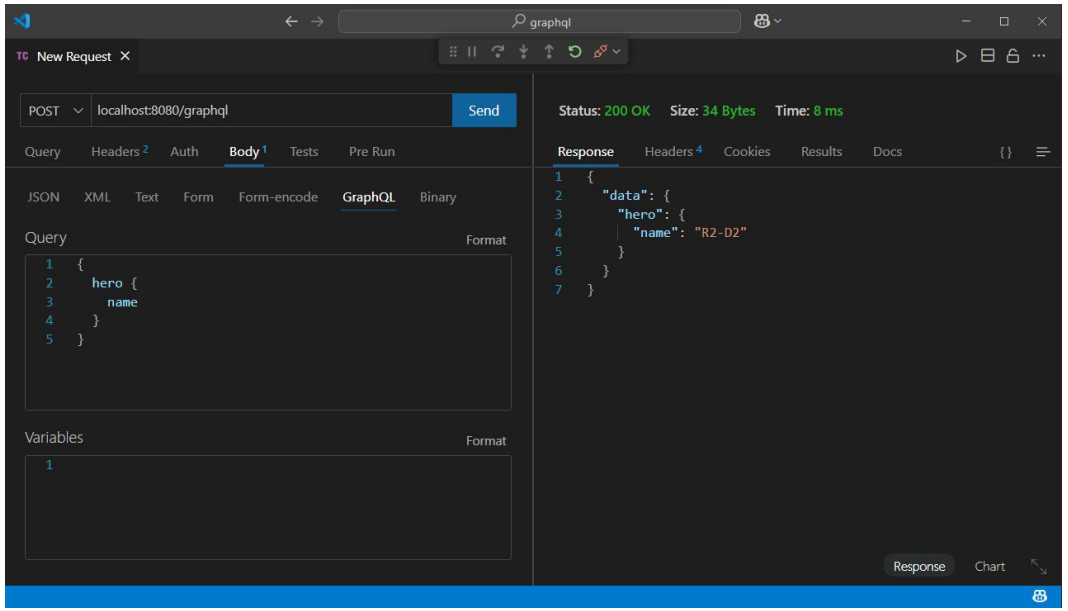

```java
/*
     * Mapeado no resources/graphql/scheme.graphqls
     *
     *  type Mutation {
     *      createReview(episode: Episode!, review: ReviewInput!): Review
     *  }
     */
    @MutationMapping
    public Review createReview(@Argument Episode episode, @Argument ReviewInput
review) {
        return new Review(review.getStars(), review.getCommentary());
    }
}
```

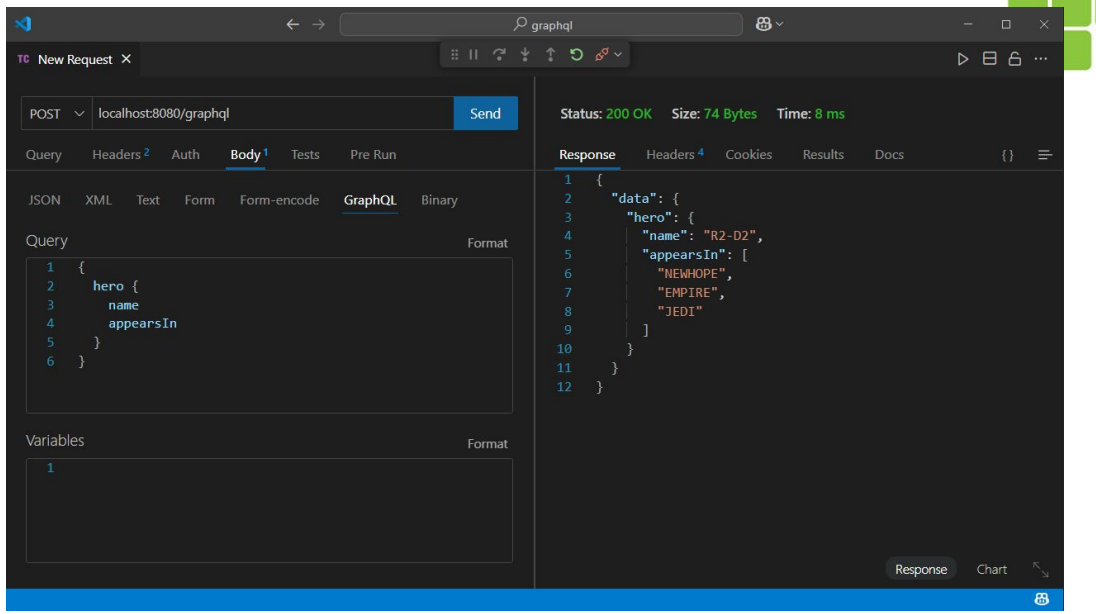
Anotações no SpringBoot

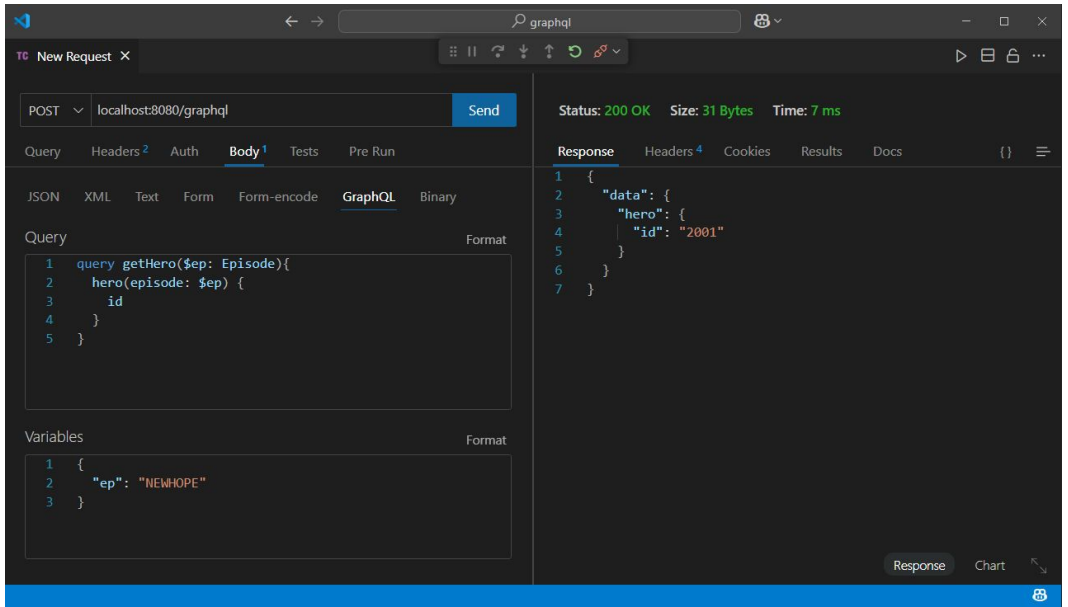


Vamos ao código!

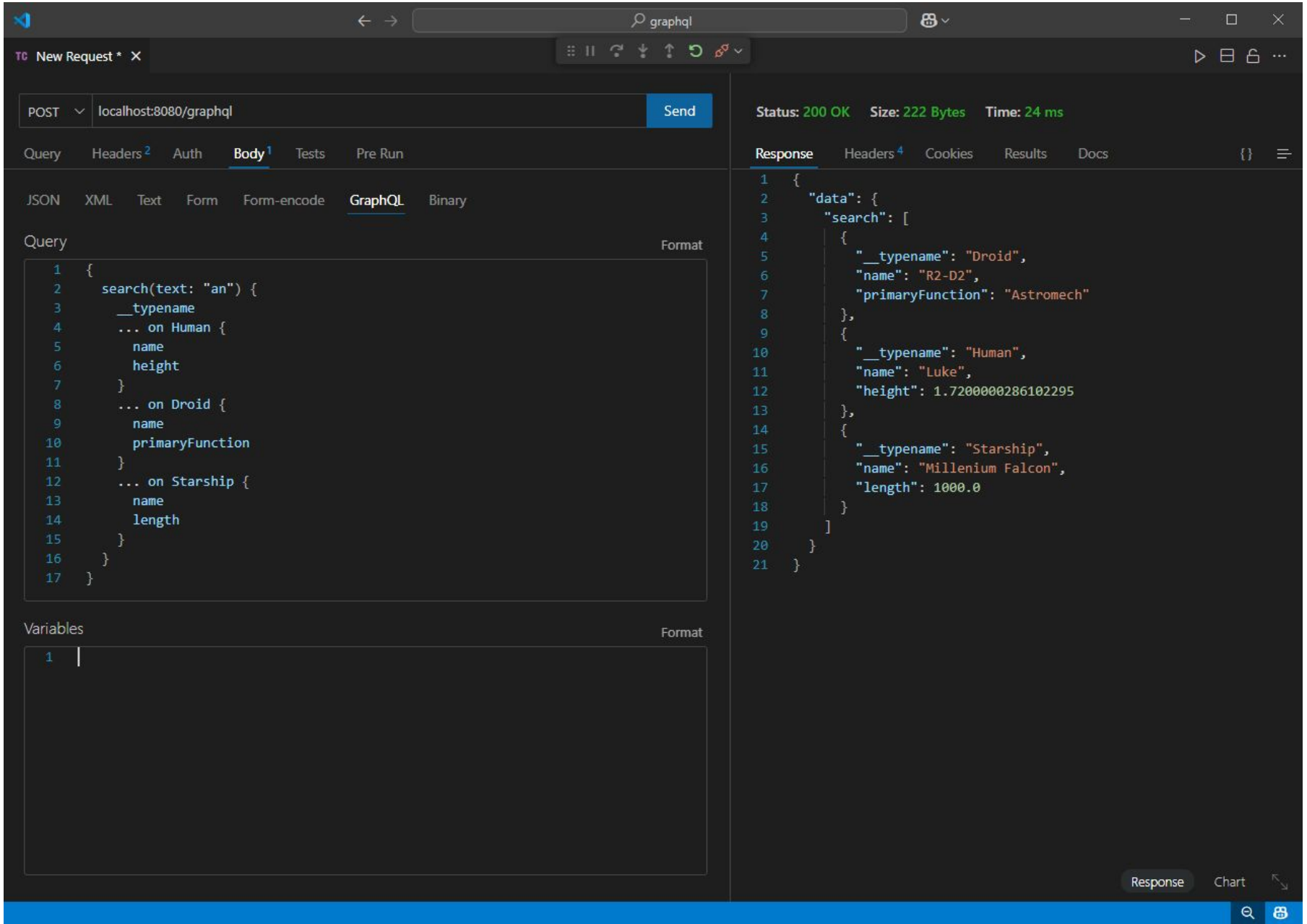
Testando consultas (Fazer no Thunder Client – Usa-se somente POST)



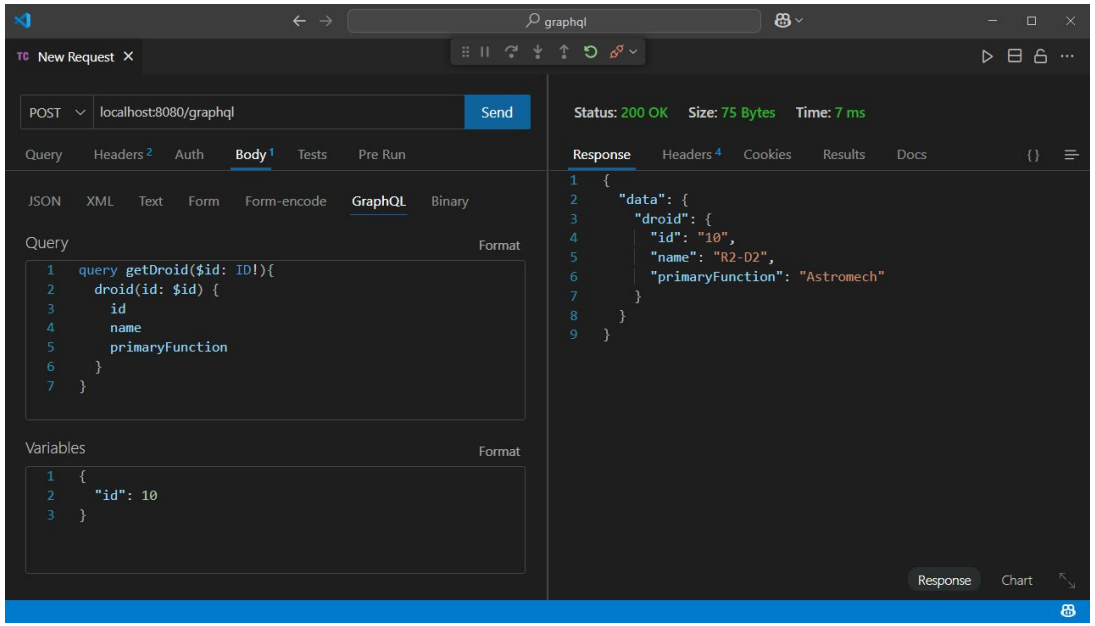


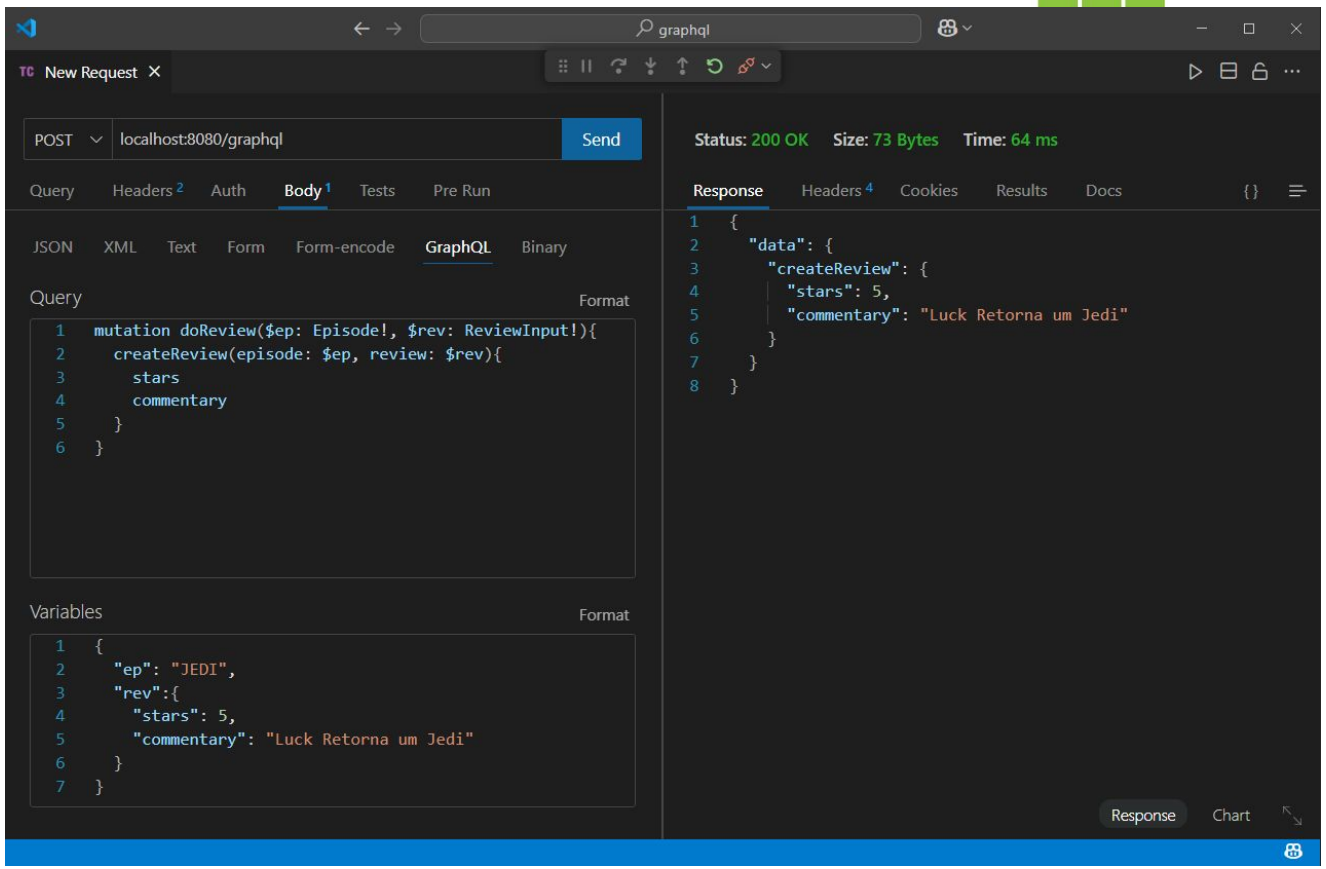


Testando consultas (Fazer no Thunder Client – Usa-se somente POST)



Testando consultas (Fazer no Thunder Client – Usa-se somente POST)





Fontes e Links

JIN, Brenda; SAHNI, Saurabh; SHEVAT, Amir. **Designing Web APIs: Building APIs That Developers Love**. " O'Reilly Media, Inc.", 2018.

QUIÑA-MERA, Antonio et al. Graphql: a systematic mapping study. **ACM Computing Surveys**, v. 55, n. 10, p. 1-35, 2023.

https://graphql.org/

https://graphql.org/learn/

https://graphql.org/learn/schema/

https://graphql.org/learn/queries/

https://spring.io/guides/gs/graphql-server

https://dl.acm.org/doi/pdf/10.1145/3561818