Arquiteturas de Integração

CAPÍTULO 6. GRAPHQL

PROF. DIOVANI LUIZ MERLO

Arquiteturas de Integração

AULA 6.1. INTRODUÇÃO A GRAPHQL

PROF. DIOVANI LUIZ MERLO

Nesta aula

- Introdução aos conceitos de GraphQL.
- Exemplo de estrutura .
- Exemplo de consulta.
- Princípios de design.
- Conceito de overfetching.

Introdução

Segundo PORCELLO E BANKS (2018), GraphQL é uma linguagem de consulta desenvolvida para uso em APIs, permitindo que os dados sejam manipulados em tempo de execução das APIs de forma agnóstica, tipicamente sobre o protocolo HTTP.

Na visão de FRISENDAL (2018), GraphQL tem tido um interesse muito grande atualmente. Ainda segundo o autor, trata-se de um projeto open-source do Facebook. Pode-se afirmar que GraphQL é independente e não está vinculado a qualquer banco de dados ou mecanismo de armazenamento específico.

Exemplo de estrutura GraphQL

Um serviço baseado em GraphQL é criado a partir da definição de tipos (types) e atributos (fields) associados a funções para cada atributo do respectivo tipo.

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}

function Query_me(request) {
  return request.auth.user;
}

function User_name(user) {
  return user.getName();
}
```

Fonte: GRAPHQL (2020)

Exemplo de consulta com GraphQL

```
query {
  person(personID:5) {
    name
    birthYear
    created
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "19BBY",
      "created": "2014-12-10T15:20:09.791000Z"
    }
  }
}
```

Fonte: PORCELLO, E.; BANKS, A. (2018)

Princípios da linguagem GraphQL

Hierarquia: GraphQL é uma consulta hierárquica, ou seja, os campos são aninhados dentro de outros campos e os dados da consulta são retornados.

Centrado em produto: GraphQL é orientado a dados solicitados dos clientes.

Princípios da linguagem GraphQL

Fortemente tipado: um servidor de GraphQL possui esquemas onde cada dado refere-se a um tipo específico a ser validado.

Consultas especificadas por clientes: um servidor de GraphQL provê somente as funcionalidades permitidas para serem consumidas pelos clientes.

Introspectivo: GraphQL é uma linguagem que permite consultar seus próprios tipos.

Mas qual é o motivo da existência de GraphQL? Com serviços em REST nós já não conseguimos retornar as informações do servidor?

Conceito de “Busca excessiva” (Overfetching)

Uma consulta a uma API para retornar dados de uma pessoa pode trazer uma enorme quantidade de dados que, do ponto de vista do cliente que consome, pode ser considerado desnecessário, pois o interesse seria para um conjunto menor de informações.

Exemplo de Overfetching

Se acionarmos a API de exemplo do link https://swapi.dev/api/people/1/, serão retornados dados de exemplo.


```
{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77",
  "hair_color": "blond",
  "skin_color": "fair",
  "eye_color": "blue",
  "birth_year": "19BBY",
  "gender": "male",
  "homeworld": "https://swapi.co/api/planets/1/",
  "films": [
    "https://swapi.co/api/films/2/",
    "https://swapi.co/api/films/6/",
    "https://swapi.co/api/films/3/",
    "https://swapi.co/api/films/1/",
    "https://swapi.co/api/films/7/"
  ],
  "species": [
    "https://swapi.co/api/species/1/"
  ],
  "vehicles": [
    "https://swapi.co/api/vehicles/14/",
    "https://swapi.co/api/vehicles/30/"
  ],
  "starships": [
    "https://swapi.co/api/starships/12/",
    "https://swapi.co/api/starships/22/"
  ],
  "created": "2014-12-09T13:50:51.644000Z",
  "edited": "2014-12-20T21:17:56.891000Z",
  "url": "https://swapi.co/api/people/1/"
}
```


Conclusão

- ✓ Em serviços utilizando apenas REST, observa-se um conjunto demasiadamente excessivo, pois neste exemplo, o cliente esperava as informações apenas do nome, altura e peso da pessoa consultada. Conclui-se que o uso de GraphQL pode otimizar drasticamente o uso de APIs no mercado, deixando a cargo do cliente a escolha sobre quais informações ele precisa naquele momento.

Próxima aula

01.

Conceitos da linguagem GraphQL.

Arquiteturas de Integração

AULA 6.2. CONCEITOS GRAPHQL

PROF. DIOVANI LUIZ MERLO

Nesta aula

- Conceitos básicos da linguagem.
- Conceito de Document, Fields, Aliases, Fragments, Variables e Directives.
- Conceitos de Operations e Mutations.

Conceitos da linguagem

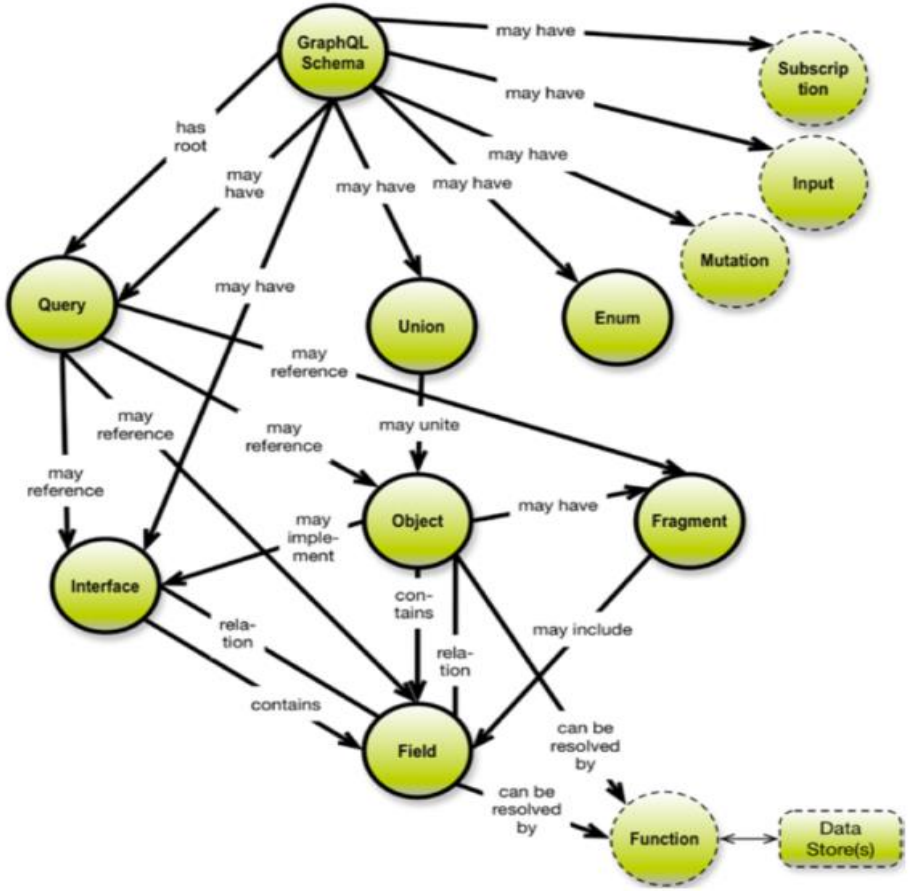
Como GraphQL é uma linguagem, torna-se importante o conhecimento sobre os conceitos relacionados aos termos e elementos utilizados, os quais são vários.

Antes de detalhar cada conceito associado a linguagem GraphQL, é importante destacar que ela se baseia na teoria de grafos, o qual de forma simples é uma representação de coleções de objetos interconectados.

Conceitos existentes em GraphQL

Observa-se a existência de um grafo que descreve relacionamentos nomeados, detalhando seus relacionamentos, que podem ser:

- Um-para-um (sem setas nas extremidades das ligações).
- Um-para-muitos (com uma seta na extremidade de uma ligação).
- Muitos-para-muitos (com setas em ambas as extremidades das ligações).



Fonte: FRISENDAL (2018

Conceitos de *Document*

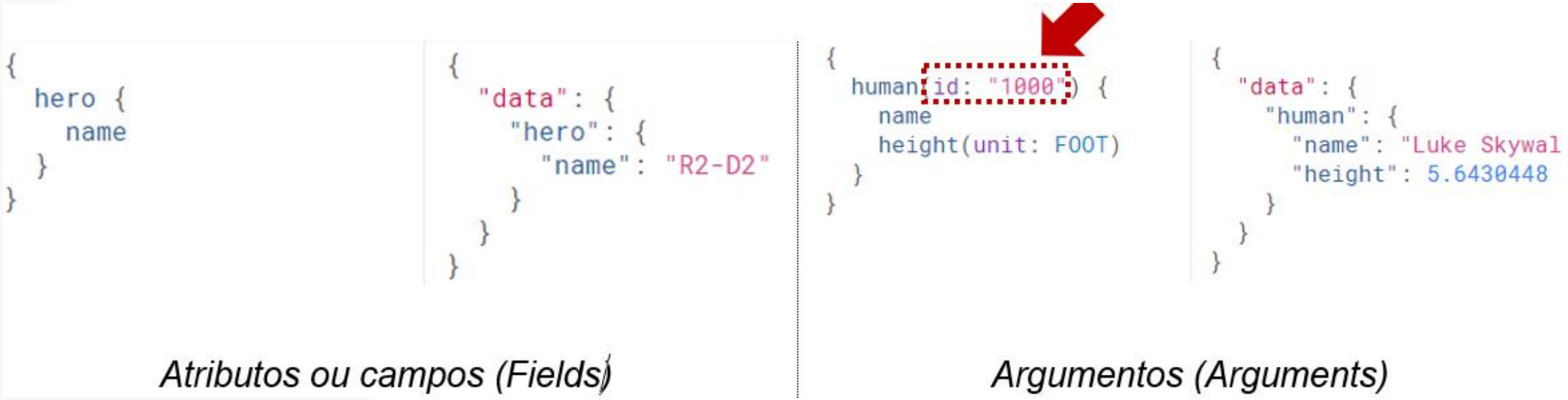
Para comunicar com um serviço que suporte GraphQL, é necessário o envio de um documento escrito na linguagem GraphQL. Esse documento deve conter uma ou mais operações relacionadas a leitura e/ou escrita de dados. Será detalhado mais adiante que as operações de escrita são denominadas como mutações (*mutations*) (BUNA, 2016).

```
# Find one article and its list of comments:
query ArticleComments {
  article(articleId: 42) {
    comments {
      commentId
      formattedBody
      timestamp
    }
  }
}
```

Fonte: BUNA (2016)

Conceitos de *Fields*

Representa informação de um objeto e/ou argumento (*arguments*), o qual é um filtro na busca de um objeto ou atributo.



Fonte: GRAPHQL (2020)

Conceitos de *Aliases*

O conceito *aliases* permite que uma consulta seja realizada na API para atributos com nomes iguais, porém com “apelidos” distintos para facilitar o entendimento da resposta do serviço.

É realizada uma consulta dupla para o mesmo atributo hero, porém com aliases distintos associados a empireHero e jediHero, respectivamente.

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

Fonte: GRAPHQL (2020)

Conceitos de *Fragments*

O conceito de fragmento (*fragments*) permite otimizar as consultas, restringindo a parte comum do código de forma unificada.

O fragmento denominado *comparsionFields* que estabelece os atributos a serem exibidos nas duas consultas realizadas com os aliases *leftComparison* e *rightComparison.*

Fonte: GRAPHQL (2020)

Consulta

```
{
  leftComparison: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  rightComparison: hero(episode: JEDI) {
    ...comparisonFields
  }
}

fragment comparisonFields on Character {
  name
  appearsIn
  friends {
    name
  }
}
```

Resposta

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "friends": [
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    },
    "rightComparison": {
      "name": "R2-D2",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        }
      ]
    }
  }
}
```

Conceitos de *Variables*

Acrescenta-se como possibilidade de dinamismo a linguagem, o conceito de variáveis (*variables*).

O termo *$episode* é uma variável que será substituída em tempo de execução da consulta. Uma variável pode ser utilizada também dentro de fragmentos. As variáveis também podem ter valores padrões atribuídos diretamente nas consultas, conforme destacado em negrito no trecho de código abaixo:

query HeroNameAndFriends($episode: ***Episode = JEDI****)*

Consulta

```
query HeroNameAndFriends($episode: Episode) {
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}
```

VARIABLES

```
{
  "episode": "JEDI"
}
```

Resposta

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

Fonte: GRAPHQL (2020)

Conceitos de *Directives*

Estabelece a possibilidade de incluir trechos adicionais de código que serão interpretados pelo servidor (backend) GraphQL para, por exemplo, exibir parte do resultado da consulta baseado em uma variável booleana (verdadeiro/falso).

O termo *@include (if: $withFriends)* é uma diretiva e, além desta, a atual especificação suporta mais a *@skip (if: Boolean)*, ou seja, somente essas duas podem ser utilizadas baseados em filtros booleanos.

Consulta

```
query Hero($episode: Episode, $withFriends:
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

VARIABLES

```
{
  "episode": "JEDI",
  "withFriends": false
}
```

Resposta

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

Fonte: GRAPHQL (2020)

Conceitos de *Operations*

O termo que vem logo após a palavra-chave *query* da linguagem é o nome da operação que foi definida e sua respectiva estrutura de dados, que será retornada quando a operação for realizada no backend.

Consulta

```
query HeroNameAndFriends {
  hero {
    name
    friends {
      name
    }
  }
}
```

Resposta

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

Fonte: GRAPHQL (2020)

Conceitos de *Mutations*

Soluções de API também podem oferecer serviços que modificam informações no servidor (backend). Então, tem-se o conceito de mutação (*mutations*), o qual permite que uma operação de escrita de dados seja realizada.

Operação

```
mutation CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
  createReview(episode: $ep, review: $review) {
    stars
    commentary
  }
}
```

VARIABLES

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

Resposta

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

Fonte: GRAPHQL (2020)

Conclusão

- ✓ Ao conhecer os conceitos da linguagem GraphQL, percebe-se sua importância e as possibilidades que são abertas para dar robustez as APIs no que tange a forma de interação com os clientes.

Próxima aula

01.

GraphQL – Design do Esquema.

Arquiteturas de Integração

AULA 6.3. DESIGN DO ESQUEMA

PROF. DIOVANI LUIZ MERLO

Nesta aula

- Introdução ao esquema de tipos.

Introdução sobre esquema em GraphQL

Na visão de PORCELLO e BANKS (2018), GraphQL vem justamente para mudar o processo de design de uma API, onde esta deixa de ser apenas um conjunto de endereços (endpoints) REST para ser uma coleção de tipos.

Essa coleção de tipos é denominada esquema (*schema*). **É importante destacar que o time de desenvolvimento precisa levar essa visão em consideração para a construção de uma nova API que utilize GraphQL.**

Introdução sobre esquema em GraphQL

Schema First é uma metodologia que permite que os times de desenvolvimento (*frontend e backend*) se alinhem em relação a todos os tipos que compõem uma solução.

O time de backend tem um entendimento claro sobre as informações, que precisam manter e disponibilizar.

O time de frontend tem um entendimento claro sobre as telas, que precisa entregar para o cliente final.

Introdução sobre esquema em GraphQL

Portanto, o entendimento de um esquema possibilita a todos um vocabulário claro, que é utilizado como ferramenta de comunicação sobre a solução que está sendo construída (PORCELLO, E.; BANKS, A., 2018).

Além disso, é importante destacar que GraphQL pode ser utilizado com qualquer framework ou linguagem de programação, para servidor (backend). Ou seja, um serviço baseado em GraphQL pode ser escrito em qualquer linguagem (GRAPHQL, 2020).

Esquema: conceito de *Type*

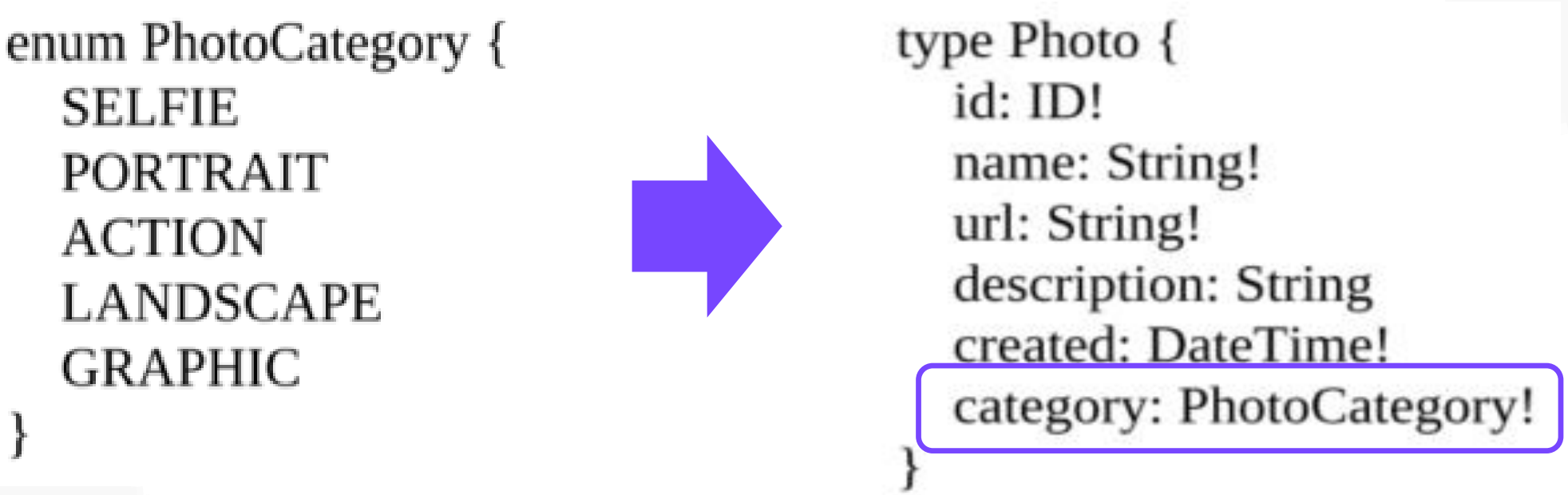
Representa um objeto customizado e como este descreve as suas funcionalidades. Por exemplo, uma aplicação comum para rede social consiste em usuários e seus posts.

Outro exemplo, uma aplicação de blog a qual é composta por categorias e artigos. Todos esses tipos (Types) representam os dados da aplicação.

```
type Photo {
    id: ID!
    name: String!
    url: String!
    description: String
}
```

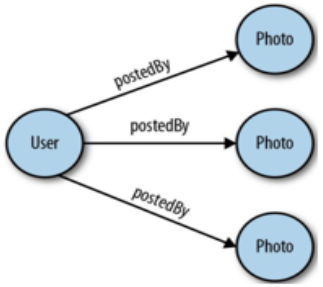
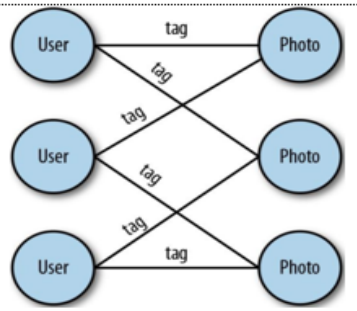
Fonte: PORCELLO, E.; BANKS, A. (2018)

Esquema: conceito de *Enum Types*



Fonte: PORCELLO, E.; BANKS, A. (2018)

Relacionamentos entre tipos

Um para um	*Um para muitos*	*Muitos para muitos*
		

type User {
githubLogin: ID!
name: String
avatar: String
}
 	
type Photo {
id: ID!
name: String!
url: String!
description: String
created: DateTime!
category: PhotoCategory!
postedBy: User!
}
	type User {
	githubLogin: ID!
	name: String
	avatar: String
	postedPhotos: [Photo!]!
	}
		type User {
		...
		inPhotos: [Photo!]!
		}
		
		type Photo {
		...
		taggedUsers: [User!]!
		}

Fonte: PORCELLO, E.; BANKS, A. (2018)

Esquema: conceito de *interface*

GraphQL também oferece um tipo (Type) denominado interface (Interface), o qual define um tipo abstrato, porém diferentemente de outras linguagens de programação, permite que sejam definidos conjuntos de atributos.

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

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

Fonte: PORCELLO, E.; BANKS, A. (2018)

União entre Tipos

Consulta

```
union SearchResult = Human | Droid | Starship
```

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
```

Resposta


```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo",
        "height": 1.8
      },
      {
        "__typename": "Human",
        "name": "Leia Organa",
        "height": 1.5
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced x1",
        "length": 9.2
      }
    ]
  }
}
```


Fonte: PORCELLO, E.; BANKS, A. (2018)

Esquema: conceito de *Input Types*

Representam uma forma melhor para organizar os argumentos (Arguments) para permitir o dinamismo na execução das consultas e mutações da linguagem GraphQL.

```
input PostPhotoInput {
  name: String!
  description: String
  category: PhotoCategory=PORTRAIT
}

type Mutation {
   postPhoto(input: PostPhotoInput!): Photo!
}
```

```
mutation newPhoto($input: PostPhotoInput!) {
   postPhoto(input: $input) {
      id
      url
      created
   }
}
```

Fonte: PORCELLO, E.; BANKS, A. (2018)

Conclusão

- ✓ Entender o esquema de GraphQL é importante para que uma API possa ser representada através de um mapeamento de linguagens e seus tipos, permitindo um vocabulário comum para os clientes da solução.

Próxima aula

01.

Boas práticas de GraphQL.

Arquiteturas de Integração

AULA 6.4. BOAS PRÁTICAS PARA GRAPHQL

PROF. DIOVANI LUIZ MERLO

Nesta aula

- Boas práticas de HTTP.
- Boas práticas de HTTP.
- Boas práticas de autorização e versionamento.
- Boas práticas de paginação.

Introdução

A especificação de GraphQL abstrai questões como rede, autorização e paginação, porém não significa que não existem soluções para essas questões, que são implementadas fora da própria especificação.

Boas práticas: HTTP

GraphQL é diferente de uma API REST tradicional, pois expõe apenas uma URL para acesso a API em detrimento de vários outros endpoints, como acontece com REST. Normalmente, a URL a ser acessada em uma API com GraphQL é:

- “http://<host-da-api>/graphql?query”
- As requisições POST devem ser utilizada com *application/json* no *content type* e incluir no corpo (*body*) da requisição *JSON-encoded*.
- As respostas da API serão geradas em forma JSON.

Boas práticas: autorização e versionamento

GraphQL delega o mecanismo de autorização para a camada de negócio.

GraphQL somente retorna os dados baseados em requisições explícitas utilizando a própria linguagem, ou seja, novas funcionalidades (atributos e operações) não afetam a execução da API.

Boas práticas: paginação

Os relacionamentos um para muitos ou muitos para muitos podem retornar uma lista de valores para um determinado atributo.

Em GraphQL o cliente pode utilizar uma técnica tradicional e enviar no documento de consulta o número de elementos a serem retornados, por exemplo, friends(first:2), onde first determina que retornará os 2 primeiros;

Para manter uma relação da consulta com a paginação, pode-se utilizar um objeto do tipo sistema denominado:

```
pageInfo {
        endCursor
        hasNextPage
}
```

Conclusão

- ✓ Conhecer as boas práticas de uso da linguagem GraphQL permite a criação de APIs que suportam mudanças sem impactos aos seus clientes (versionamento), e a eliminação de conhecimento de várias URIs de serviços REST tradicionais.

Próxima aula

01.

GraphQL na prática

Arquiteturas de Integração

AULA 6.5. GRAPHQL NA PRÁTICA

PROF. DIOVANI LUIZ MERLO

Nesta aula

- GraphQL na prática.
- Ferramentas e bibliotecas para linguagens de programação.
- Desafios para GraphQL.
- Hans On com GraphQL.

Introdução

GraphQL é utilizado por uma variedade de empresas para potencializar seus aplicativos, sites e APIs. Como exemplo, tem-se o GitHub, que passou a utilizar a partir da versão 4 de sua API, The New York Times, IBM, Twitter e Yelp (PORCELLO, E.; BANKS, A., 2018).

Ferramentas e bibliotecas GraphQL

Bibliotecas para linguagens de programação:

- https://graphql.org/code/

Ferramentas:

GraphiQL:

- https://github.com/graphql/graphiql
- https://graphql.org/swapi-graphql

GraphQL Playground:

- https://www.graphqlbin.com/v2/new
- https://github.com/prisma-labs/graphql-playground

Ferramentas e bibliotecas GraphQL

Neo4j e GraphQL:

- https://neo4j.com/developer/graphql

GitHub API:

- https://developer.github.com/v4/explorer/

Outras ferramentas/APIs:

- https://github.com/APIs-guru/graphql-apis

Desafios na prática para GraphQL

Provavelmente, após o entendimento dos conceitos sobre a linguagem GraphQL e suas aplicabilidades para interagir com dados de uma API, torna-se necessário colocar em prática os conceitos.

No mundo real, muitas aplicações já expõem suas APIs a partir de lógicas de negócio e estrutura de dados legadas.

São muitos os desafios que devem ser enfrentados pelos times de desenvolvimento para conseguir implementar GraphQL em aplicações legadas, tendo em vista a necessidade de mapeamento dos tipos (Types), atributos (Fields), relacionamentos e operações (Operations) a partir de um vocabulário que muitas das vezes não é preciso e encontra-se redundante em aplicações legadas.

Desafios na prática para GraphQL

Fonte: FRISENDAL (2018)

Hands On com GraphQL

Conclusão

- ✓ O uso de GraphQL para nova soluções de APIs pode ser implementada com uma complexidade muito menor, se forem adotados os conceitos da linguagem para representação das informações, através de seu esquema e tipos. Contudo, em soluções legadas de API existem vários desafios para colocar em prática essa linguagem.