GraphQL

José Ribamar Ferreira Junior
web2ajax@gmail.com

Quer construir APIs como o Facebook? Como o framework do Facebook para criação de APIs, o GraphQL, tornou-se publicamente disponível, essa ambição parece estar ao alcance de muitas empresas. E isso é ótimo. Mas primeiro, vamos aprender o que realmente é o GraphQL e - talvez até mais importante - vamos descobrir como aplicar o GraphQL para criar APIs que os consumidores adoram. Neste livro, usamos uma abordagem prática para aprender GraphQL. Primeiro, exploramos os conceitos das duas linguagens GraphQL usando exemplos. Então começamos a escrever algum código para nossa primeira API GraphQL. Desenvolvemos essa API passo a passo, desde a criação de um esquema e a resolução de consultas, passando por dados de zombaria e conectando fontes de dados até o desenvolvimento de mutações e a configuração de assinaturas de eventos. Seus clientes da API são importantes para você? Este livro mostra como aplicar um processo de design orientado ao consumidor para APIs GraphQL, para que você possa oferecer o que seus clientes realmente desejam: uma API que solucione os problemas e ofereça uma excelente experiência de desenvolvedor. Deseja ativar os consumidores da API para que eles criem ótimos aplicativos? Este livro explica a linguagem de consulta GraphQL, que permite que os consumidores da API recuperem dados, gravem dados e sejam notificados quando os dados forem alterados. Mais importante, você permite que eles decidam quais dados eles realmente precisam da API. Você quer tornar sua API fácil e intuitiva de usar? Este livro mostra como usar a linguagem de esquema GraphQL para definir um sistema de tipos para sua API, que serve como uma documentação de referência e ajuda seus consumidores de API a escrever consultas sintaticamente corretas. Você quer lucrar com o que funcionou para os outros? Este livro fornece uma coleção de práticas recomendadas para o GraphQL que funcionaram para outras.

Introdução ao GraphQL

Aprenda sobre o GraphQL, como ele funciona e como usá-lo nesta série de artigos. Procurando documentação sobre como construir um serviço GraphQL? Existem bibliotecas para ajudá-lo a implementar o GraphQL em muitos idiomas diferentes . Para uma experiência de aprendizado aprofundada com tutoriais práticos, visite o site do tutorial How to GraphQL fullstack.

GraphQL é uma linguagem de consulta para sua API e um tempo de execução do lado do servidor para executar consultas usando um sistema de tipos que você define para seus dados. O GraphQL não está vinculado a nenhum banco de dados ou mecanismo de armazenamento específico e, em vez disso, é respaldado pelo código e pelos dados existentes.

Um serviço GraphQL é criado definindo tipos e campos nesses tipos e, em seguida, fornecendo funções para cada campo em cada tipo. Por exemplo, um serviço GraphQL que informa quem é o usuário logado (`me`) e o nome desse usuário podem ser algo como isto:

```
tipo Query {
  me : User
}

tipo User {
  id : nome da ID
  : String }
```

Junto com funções para cada campo em cada tipo:

```
função Query_me ( request ) {
  return request . auth . usuário ;
}
```

```
function User_name ( user ) {
  retornar usuário . getName ( ) ;
}
```

Uma vez que um serviço GraphQL está sendo executado (geralmente em uma URL em um serviço da web), ele pode enviar consultas GraphQL para validar e executar. Uma consulta recebida é primeiro verificada para garantir que ela se refira apenas aos tipos e campos definidos e, em seguida, executa as funções fornecidas para produzir um resultado.

Por exemplo, a consulta:

```
{
  me {
    nome
  }
}
```

Poderia produzir o resultado JSON:

```
{
  "eu" : {
    "nome" : "Luke Skywalker"
  }
}
```

Saiba mais sobre o GraphQL - a linguagem de consulta, o sistema de tipos, como o serviço GraphQL funciona, bem como as práticas recomendadas para usar o GraphQL para resolver problemas comuns - nos artigos escritos nesta seção.

Nesta página, você aprenderá em detalhes sobre como consultar um servidor GraphQL.

Campos

Em sua forma mais simples, o GraphQL é sobre a solicitação de campos específicos em objetos. Vamos começar olhando para uma consulta muito simples e o resultado que obtemos quando a rodamos:

```
{
 hero {
   nome
 }
}
```

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

Você pode ver imediatamente que a consulta tem exatamente a mesma forma que o resultado. Isso é essencial para o GraphQL, porque você sempre recebe de volta o que espera, e o servidor sabe exatamente quais campos o cliente está pedindo.

O campo `name`retorna um `String`tipo, neste caso o nome do herói principal de Star Wars `"R2-D2"`,.

Ah, mais uma coisa - a consulta acima é *interativa* . Isso significa que você pode mudá-lo como quiser e ver o novo resultado. Tente adicionar um `appearsIn`campo ao `hero`objeto na consulta e veja o novo resultado.

No exemplo anterior, apenas pedimos o nome de nosso herói que retornou uma String, mas os campos também podem se referir a Objects. Nesse caso, você pode fazer uma *sub-seleção* de campos para esse objeto. As consultas GraphQL podem percorrer os objetos relacionados e seus campos, permitindo que os clientes obtenham muitos dados relacionados em uma solicitação, em vez de fazer várias viagens de ida e volta, como seria necessário em uma arquitetura REST clássica.

```
{
 hero {
   nome
   # Consultas podem ter comentários!
   amigos {
     nome
   }
 }
```

```
}

{

 "data" : {

   "hero" : {

     "nome" : "R2-D2" ,

     "amigos" : [

       {

         "nome" : "Luke Skywalker"

       }

       {

         "name" : "Han Solo"

       }

       {

         "nome" : "Leia Organa"

       }

     ]

   }

 }
```

```
}
```

Observe que neste exemplo, o `friends`campo retorna uma matriz de itens. As consultas do GraphQL parecem as mesmas para itens únicos ou listas de itens, no entanto, sabemos qual deles é esperado com base no que é indicado no esquema.

Argumentos

Se a única coisa que poderíamos fazer era atravessar objetos e seus campos, o GraphQL já seria uma linguagem muito útil para a busca de dados. Mas quando você adiciona a capacidade de passar argumentos para campos, as coisas ficam muito mais interessantes.

```
{
 humano ( id : "1000" ) {
   nome
   altura
 }
}
```

```
{
 "data" : {
   "humano" : {
     "nome" : "Luke Skywalker" ,
```

```
    "altura" : 1,72
  }
 }
}
```

Em um sistema como o REST, você só pode passar um único conjunto de argumentos - os parâmetros de consulta e os segmentos de URL em sua solicitação. Mas no GraphQL, cada campo e objeto aninhado pode obter seu próprio conjunto de argumentos, tornando o GraphQL um substituto completo para fazer várias buscas de API. Você pode até mesmo passar argumentos em campos escalares, para implementar transformações de dados uma vez no servidor, em vez de em cada cliente separadamente.

```
{
 humano ( id : "1000" ) {
   nome
   altura ( unidade : FOOT )
 }
}
{
 "data" : {
   "humano" : {
     "nome" : "Luke Skywalker" ,
```

```
      "altura" : 5.6430448
    }
  }
}
```

Argumentos podem ser de muitos tipos diferentes. No exemplo acima, usamos um tipo Enumeration, que representa um de um conjunto finito de opções (neste caso, unidades de comprimento, ou `METER`ou `FOOT`). O GraphQL vem com um conjunto padrão de tipos, mas um servidor GraphQL também pode declarar seus próprios tipos personalizados, desde que eles possam ser serializados em seu formato de transporte.

Leia mais sobre o sistema de tipos GraphQL aqui.

Aliases

Se você tiver um olho aguçado, você deve ter notado que, como os campos do objeto de resultado correspondem ao nome do campo na consulta, mas não incluem argumentos, não é possível consultar diretamente o mesmo campo com argumentos diferentes. É por isso que você precisa de *aliases* - eles permitem renomear o resultado de um campo para qualquer coisa que você queira.

```
{
  impireHero : hero ( episode : EMPIRE ) {
    nome
```

```
 }
 jediHero : hero ( episódio : JEDI ) {
   nome
 }
}
```

```
{
 "data" : {
   "impireHero" : {
     "nome" : "Luke Skywalker"
   }
   "jediHero" : {
     "nome" : "R2-D2"
   }
 }
}
```

No exemplo acima, os dois `hero`campos teriam entrado em conflito, mas como podemos alias-los a nomes diferentes, podemos obter os dois resultados em uma solicitação.

Fragmentos

Digamos que tenhamos uma página relativamente complicada em nosso aplicativo, o que nos permite ver dois heróis lado a lado, junto com os amigos deles. Você pode imaginar que essa consulta poderia se complicar rapidamente, porque precisaríamos repetir os campos pelo menos uma vez - um para cada lado da comparação.

É por isso que o GraphQL inclui unidades reutilizáveis chamadas *fragmentos* . Os fragmentos permitem que você construa conjuntos de campos e os inclua em consultas onde você precisa. Aqui está um exemplo de como você poderia resolver a situação acima usando fragmentos:

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

Detalhes

```
fragment comparisonFields on Character {
```

```
 nome
 aparece em
 amigos {
   nome
 }
}
{
 "data" : {
   "leftComparison" : {
     "nome" : "Luke Skywalker" ,
     "appearIn" : [
       "NEWHOPE" ,
       "EMPIRE" ,
       "JEDI"
     ]
     "amigos" : [
       {
         "name" : "Han Solo"
```

```
    }
    {
      "nome" : "Leia Organa"
    }
    {
      "nome" : "C-3PO"
    }
    {
      "nome" : "R2-D2"
    }
  ]
}
"rightComparison" : {
  "nome" : "R2-D2" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
```

```
      ]
      "amigos" : [
        {
          "nome" : "Luke Skywalker"
        }
        {
          "name" : "Han Solo"
        }
        {
          "nome" : "Leia Organa"
        }
      ]
    }
  }
}
```

Você pode ver como a consulta acima seria bastante repetitiva se os campos fossem repetidos. O conceito de fragmentos é freqüentemente usado para dividir requisitos complicados de dados do aplicativo em partes menores,

especialmente quando você precisa combinar muitos componentes de UI com fragmentos diferentes em uma busca de dados inicial.

Nome da operação

Até agora, temos usado uma sintaxe abreviada onde omitimos a `query`palavra - chave e o nome da consulta, mas em aplicativos de produção é útil usá-los para tornar nosso código menos ambíguo.

Aqui está um exemplo que inclui a palavra-chave `query`como *tipo de operação* e `HeroNameAndFriends`como *nome da operação* :

```
consulta HeroNameAndFriends {
 hero {
   nome
   amigos {
     nome
   }
 }
}
{
 "data" : {
   "hero" : {
```

```
      "nome" : "R2-D2" ,
      "amigos" : [
        {
          "nome" : "Luke Skywalker"
        }
        {
          "name" : "Han Solo"
        }
        {
          "nome" : "Leia Organa"
        }
      ]
    }
  }
}
```

O *tipo de operação* é *consulta* , *mutação* ou *assinatura* e descreve o tipo de operação que você pretende executar. O tipo de operação é necessário, a menos que você esteja usando a sintaxe abreviada de consulta. Nesse caso,

não é possível fornecer um nome ou definições de variáveis para sua operação.

O *nome da operação*é um nome significativo e explícito para sua operação. Ele é requerido somente em documentos multi-operação, mas seu uso é encorajado, pois é muito útil para depuração e registro do lado do servidor. Quando algo dá errado nos seus logs de rede ou no seu servidor GraphQL, é mais fácil identificar uma consulta em sua base de código pelo nome em vez de tentar decifrar o conteúdo. Pense nisso como um nome de função na sua linguagem de programação favorita. Por exemplo, em JavaScript, podemos trabalhar facilmente apenas com funções anônimas, mas quando damos um nome a uma função, é mais fácil rastreá-la, depurar nosso código e registrar quando ela é chamada. Da mesma forma, os nomes de consulta e mutação do GraphQL, juntamente com nomes de fragmentos, podem ser uma ferramenta de depuração útil no lado do servidor para identificar diferentes solicitações do GraphQL.

Variáveis

Até agora, escrevemos todos os nossos argumentos dentro da string de consulta. Mas na maioria dos aplicativos, os argumentos para os campos serão dinâmicos: por exemplo, pode haver um menu suspenso que permita selecionar em qual episódio de Star Wars você está interessado, ou um campo de pesquisa ou um conjunto de filtros.

Não seria uma boa idéia passar esses argumentos dinâmicos diretamente na string de consulta, porque então nosso código do lado do cliente precisaria manipular dinamicamente a string de consulta em tempo de execução e serializá-la em um formato específico do GraphQL. Em vez disso, o GraphQL tem uma maneira de classificar os valores dinâmicos da consulta e passá-los como um dicionário separado. Esses valores são chamados de *variáveis* .

Quando começamos a trabalhar com variáveis, precisamos fazer três coisas:

1. Substitua o valor estático na consulta por `$variableName`
2. Declarar `$variableName`como uma das variáveis aceitas pela consulta
3. Passe `variableName: value`no dicionário de variáveis separado, específico do transporte (geralmente JSON)

Aqui está o que parece tudo junto:

```
consulta HeroNameAndFriends ( $ episode : Episode ) {

 herói ( episódio : $ episode ) {

   nome

   amigos {

     nome

   }

 }

}

{

 "episódio" : "JEDI"

}

{

 "data" : {

   "hero" : {
```

```
      "nome" : "R2-D2" ,
      "amigos" : [
        {
          "nome" : "Luke Skywalker"
        }
        {
          "name" : "Han Solo"
        }
        {
          "nome" : "Leia Organa"
        }
      ]
    }
  }
}
```

Agora, em nosso código de cliente, podemos simplesmente passar uma variável diferente em vez de precisar construir uma consulta inteiramente nova. Isso também é, em geral, uma boa prática para denotar quais argumentos em nossa consulta devem ser dinâmicos - nunca devemos fazer

interpolação de strings para construir consultas a partir de valores fornecidos pelo usuário.

Definições variáveis

As definições de variáveis são a parte que parece `($episode: Episode)`na consulta acima. Funciona exatamente como as definições de argumento para uma função em uma linguagem tipada. Ele lista todas as variáveis, prefixadas por `$`, seguidas por seu tipo, neste caso `Episode`.

Todas as variáveis declaradas devem ser escalares, enums ou tipos de objetos de entrada. Portanto, se você quiser passar um objeto complexo para um campo, precisará saber qual tipo de entrada corresponde ao servidor. Saiba mais sobre os tipos de objeto de entrada na página Esquema.

As definições de variáveis podem ser opcionais ou obrigatórias. No caso acima, como não há um `!`próximo ao `Episode`tipo, é opcional. Mas se o campo em que você está passando a variável requer um argumento não-nulo, então a variável também deve ser requerida.

Para aprender mais sobre a sintaxe dessas definições de variáveis, é útil aprender a linguagem do esquema GraphQL. A linguagem do esquema é explicada em detalhes na página Esquema.

Variáveis Padrão

Valores padrão também podem ser atribuídos às variáveis na consulta, adicionando o valor padrão após a declaração de tipo.

```
consulta HeroNameAndFriends ( $ episode : Episode = JEDI ) {
  hero ( episódio : $ episode ) {
    nome
    amigos {
      nome
    }
```

```
  }
}
```

Quando os valores padrão são fornecidos para todas as variáveis, você pode chamar a consulta sem passar nenhuma variável. Se alguma variável for passada como parte do dicionário de variáveis, elas substituirão os padrões.

Diretivas

Discutimos acima como as variáveis nos permitem evitar a interpolação manual de strings para construir consultas dinâmicas. A passagem de variáveis em argumentos resolve uma classe muito grande desses problemas, mas também podemos precisar de uma maneira de alterar dinamicamente a estrutura e a forma de nossas consultas usando variáveis. Por exemplo, podemos imaginar um componente de UI que tenha uma visão resumida e detalhada, onde um inclui mais campos do que o outro.

Vamos construir uma consulta para esse componente:

```
Herói da consulta ( $ episode : Episode , $ withFriends : Boolean !) {
 herói ( episódio : $ episode ) {
   nome
   amigos @include ( if : $ withFriends ) {
     nome
   }
 }
}
```

```
{

 "episódio" : "JEDI" ,

 "withFriends" : falso

}

{

 "data" : {

   "hero" : {

     "nome" : "R2-D2"

   }

 }

}
```

Tente editar as variáveis acima para em vez passar `true`para `withFriends`, e ver como as mudanças de resultados.

Precisávamos usar um novo recurso no GraphQL chamado *diretiva* . Uma diretiva pode ser anexada a um campo ou inclusão de fragmento e pode afetar a execução da consulta de qualquer maneira que o servidor desejar. A especificação principal do GraphQL inclui exatamente duas diretivas, que devem ser suportadas por qualquer implementação de servidor GraphQL compatível com especificações:

- `@include(if: Boolean)`Apenas inclua este campo no resultado, se o argumento for `true`.

- `@skip(if: Boolean)` Ignore este campo se o argumento for `true`.

As diretivas podem ser úteis para sair de situações em que, de outra forma, você precisaria manipular as strings para adicionar e remover campos em sua consulta. As implementações de servidor também podem adicionar recursos experimentais definindo diretivas completamente novas.

Mutação

A maioria das discussões do GraphQL se concentra na busca de dados, mas qualquer plataforma de dados completa precisa de uma maneira de modificar os dados do lado do servidor também.

No REST, qualquer pedido pode acabar causando alguns efeitos colaterais no servidor, mas por convenção é sugerido que não se use `GET` requisições para modificar dados. O GraphQL é semelhante - tecnicamente, qualquer consulta pode ser implementada para causar uma gravação de dados. No entanto, é útil estabelecer uma convenção de que qualquer operação que cause gravações deve ser enviada explicitamente por meio de uma mutação.

Assim como nas consultas, se o campo de mutação retornar um tipo de objeto, você poderá solicitar campos aninhados. Isso pode ser útil para buscar o novo estado de um objeto após uma atualização. Vamos ver uma simples mutação de exemplo:

```
mutação CreateReviewForEpisode ( $ ep : Episódio ! , $ revisão : ReviewInput !)
{

 createReview ( episódio : $ ep , revisão : $ review ) {

   estrelas

   comentário
```

```
 }
}
{
 "ep" : "JEDI" ,
 "review" : {
   "stars" : 5 ,
   "commentary" : "Este é um ótimo filme!"
 }
}
{
 "data" : {
   "createReview" : {
     "stars" : 5 ,
     "commentary" : "Este é um ótimo filme!"
   }
 }
}
```

Observe como o `createReview`campo retorna os campos `stars`e `commentary`da revisão recém-criada. Isso é especialmente útil ao alterar dados existentes, por exemplo, ao incrementar um campo, já que podemos alterar e consultar o novo valor do campo com uma solicitação.

Você também pode notar que, neste exemplo, a `review`variável que nós passamos não é escalar. É um *tipo de objeto de entrada* , um tipo especial de tipo de objeto que pode ser passado como um argumento. Saiba mais sobre os tipos de entrada na página Esquema.

Múltiplos campos em mutações

Uma mutação pode conter vários campos, assim como uma consulta. Há uma distinção importante entre consultas e mutações, além do nome:

Enquanto os campos de consulta são executados em paralelo, os campos de mutação são executados em série, um após o outro.

Isso significa que, se enviarmos duas `incrementCredits`mutações em uma solicitação, a primeira é garantida para terminar antes do segundo começar, garantindo que não acabemos com uma condição de corrida conosco.

Fragmentos Inline

Como muitos outros sistemas de tipos, os esquemas GraphQL incluem a capacidade de definir interfaces e tipos de união. Aprenda sobre eles no guia de esquema.

Se você está consultando um campo que retorna uma interface ou um tipo de união, você precisará usar *fragmentos embutidos* para acessar dados no tipo concreto subjacente. É mais fácil ver com um exemplo:

```
consulta HeroForEpisode ( $ ep : Episode !) {
```

```
 herói ( episódio : $ ep ) {
   nome
   ... no Droid {
     função primária
   }
   ... on Human {
     altura
   }
 }
}
{
 "ep" : "JEDI"
}
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "primaryFunction" : "Astromech"
```

```
    }
  }
}
```

Nesta consulta, o `hero`campo retorna o tipo `Character`, que pode ser um `Human`ou um `Droid`dependendo do `episode`argumento. Na seleção direta, você pode solicitar apenas campos que existem na `Character`interface, como `name`.

Para solicitar um campo no tipo concreto, você precisa usar um *fragmento embutido* com uma condição de tipo. Como o primeiro fragmento é rotulado como `... on Droid`, o `primaryFunction`campo só será executado se o `Character`retorno `hero`for do `Droid`tipo. Da mesma forma para o `height`campo para o `Human`tipo.

Fragmentos nomeados também podem ser usados da mesma maneira, já que um fragmento nomeado sempre tem um tipo anexado.

Meta Campos

Dado que existem algumas situações em que você não sabe que tipo receberá do serviço GraphQL, você precisa de alguma maneira de determinar como lidar com esses dados no cliente. O GraphQL permite que você solicite `__typename`um campo meta em qualquer ponto de uma consulta para obter o nome do tipo de objeto nesse ponto.

```
{
  search ( text : "an" ) {
    __Digite o nome
```

```
    ... on Human {
      nome
    }
    ... no Droid {
      nome
    }
    ... na nave espacial {
      nome
    }
  }
}
```

```
{
  "data" : {
    "pesquisa" : [
      {
        "__typename" : "Humano" ,
        "name" : "Han Solo"
      }
```

```
      {
        "__typename" : "Humano" ,
        "nome" : "Leia Organa"
      }
      {
        "__typename" : "Nave" ,
        "name" : "TIE Advanced x1"
      }
    ]
  }
}
```

Na consulta acima, `search`retorna um tipo de união que pode ser uma das três opções. Seria impossível distinguir os diferentes tipos do cliente sem o `__typename`campo.

Os serviços GraphQL fornecem alguns meta-campos, os quais são usados para expor o sistema Introspection .

Esquemas e Tipos

Nesta página, você aprenderá tudo o que precisa saber sobre o sistema de tipos GraphQL e como ele descreve quais dados podem ser consultados. Como o GraphQL pode ser usado com qualquer estrutura de back-end ou linguagem de programação, ficaremos longe de detalhes específicos da implementação e falaremos apenas sobre os conceitos.

Digite o sistema

Se você já viu uma consulta GraphQL, sabe que a linguagem de consulta GraphQL é basicamente sobre a seleção de campos em objetos. Então, por exemplo, na seguinte consulta:

```
{
 hero {
   nome
   aparece em
 }
}
```

```
{
 "data" : {
   "hero" : {
```

```
      "nome" : "R2-D2" ,
      "appearIn" : [
        "NEWHOPE" ,
        "EMPIRE" ,
        "JEDI"
      ]
    }
  }
}
```

1. Começamos com um objeto "raiz" especial
2. Nós selecionamos o `hero`campo naquele
3. Para o objeto retornado por `hero`, selecionamos os campos `name`e`appearsIn`

Como a forma de uma consulta GraphQL corresponde ao resultado, você pode prever o que a consulta retornará sem saber muito sobre o servidor. Mas é útil ter uma descrição exata dos dados que podemos solicitar - que campos podemos selecionar? Que tipos de objetos eles podem retornar? Quais campos estão disponíveis nesses sub-objetos? É aí que entra o esquema.

Cada serviço GraphQL define um conjunto de tipos que descrevem completamente o conjunto de dados possíveis que você pode consultar

nesse serviço. Então, quando as consultas chegam, elas são validadas e executadas nesse esquema.

Digite o idioma

Os serviços GraphQL podem ser escritos em qualquer idioma. Como não podemos confiar em uma sintaxe específica da linguagem de programação, como JavaScript, para falar sobre os esquemas GraphQL, definiremos nossa própria linguagem simples. Usaremos a "Linguagem de esquema GraphQL" - é semelhante à linguagem de consulta e nos permite falar sobre esquemas GraphQL de maneira independente de linguagem.

Tipos de objeto e campos

Os componentes mais básicos de um esquema GraphQL são tipos de objetos, que representam apenas um tipo de objeto que você pode buscar em seu serviço e quais campos ele possui. Na linguagem do esquema GraphQL, podemos representá-lo assim:

```
tipo Character {
  name : String !
  aparece em : [ episódio ] !
}
```

A linguagem é bastante legível, mas vamos analisá-la para que possamos ter um vocabulário compartilhado:

- `Character`é um *tipo de objeto GraphQL* , o que significa que é um tipo com alguns campos. A maioria dos tipos em seu esquema será de tipos de objetos.
- `name`e `appearsIn`são *campos* no `Character`tipo. Isso significa que `name`e `appearsIn`são os únicos campos que podem aparecer em qualquer parte de uma consulta GraphQL que opera no `Character`tipo.

- `String`é um dos tipos *escalares* incorporados - são tipos que resolvem um único objeto escalar e não podem ter sub-seleções na consulta. Vamos falar sobre os tipos escalares mais tarde.
- `String!`significa que o campo *não* é *anulável* , o que significa que o serviço GraphQL promete sempre dar um valor quando você consulta este campo. Na linguagem de tipos, vamos representar aqueles com um ponto de exclamação.
- `[Episode]!`representa uma *matriz* de `Episode`objetos. Como ele também *não* é *anulável* , você sempre pode esperar uma matriz (com zero ou mais itens) ao consultar o `appearsIn`campo.

Agora você sabe como é um tipo de objeto GraphQL e como ler os fundamentos da linguagem de tipos GraphQL.

Argumentos

Cada campo em um tipo de objeto GraphQL pode ter zero ou mais argumentos, por exemplo, o `length`campo abaixo:

```
tipo Starship {
  id : ID !
  nome : String !
  length ( unidade : LengthUnit = METER ) : Float
}
```

Todos os argumentos são nomeados. Ao contrário de linguagens como JavaScript e Python, onde as funções tomam uma lista de argumentos ordenados, todos os argumentos no GraphQL são passados pelo nome especificamente. Neste caso, o `length`campo tem um argumento definido `unit`,.

Argumentos podem ser obrigatórios ou opcionais. Quando um argumento é opcional, podemos definir um *valor padrão* - se o `unit`argumento não for passado, ele será definido como `METER`padrão.

Os tipos de consulta e mutação

A maioria dos tipos em seu esquema será apenas tipos de objetos normais, mas existem dois tipos especiais em um esquema:

```
schema {
  query : mutação de consulta
  : mutação }
```

Todo serviço GraphQL possui um `query`tipo e pode ou não ter um `mutation`tipo. Esses tipos são iguais a um tipo de objeto regular, mas são especiais porque definem o *ponto* de *entrada* de cada consulta GraphQL. Então, se você ver uma consulta que se parece com:

```
query {

 hero {

   nome

 }

 droid ( id : "2000" ) {

   nome

 }

}

{

 "data" : {
```

```
    "hero" : {
      "nome" : "R2-D2"
    }
    "droid" : {
      "nome" : "C-3PO"
    }
  }
}
```

Isso significa que o serviço GraphQL precisa ter um `Query`tipo com `hero`e `droid`campos:

```
Digite Query {
  hero ( episode : Episode ) : Personagem
  droid ( id : ID ! ) : Droid
}
```

As mutações funcionam de maneira semelhante - você define campos no `Mutation`tipo e esses estão disponíveis como os campos de mutação raiz que você pode chamar em sua consulta.

É importante lembrar que, além do status especial de ser o "ponto de entrada" no esquema, os tipos `Query`e `Mutation`são iguais a qualquer outro tipo de objeto GraphQL, e seus campos funcionam exatamente da mesma maneira.

Tipos escalares

Um tipo de objeto GraphQL possui um nome e campos, mas em algum momento esses campos precisam ser resolvidos para alguns dados concretos. É aí que entram os tipos escalares: eles representam as folhas da consulta.

Na consulta a seguir, os campos `name`e `appearsIn`serão resolvidos para os tipos escalares:

```
{
 hero {
   nome
   aparece em
 }
}
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "appearIn" : [
       "NEWHOPE" ,
       "EMPIRE" ,
       "JEDI"
```

```
      ]

    }

  }

}
```

Sabemos disso porque esses campos não têm subcampos - eles são as folhas da consulta.

O GraphQL vem com um conjunto de tipos escalares padrão prontos para uso:

- `Int`: Um inteiro de 32 bits assinado.
- `Float`: Um valor de ponto flutuante de precisão dupla assinado.
- `String`: Uma seqüência de caracteres UTF-8.
- `Boolean`: `true`Ou `false`.
- `ID`: O tipo escalar de ID representa um identificador exclusivo, geralmente usado para refazer um objeto ou como a chave de um cache. O tipo de ID é serializado da mesma maneira que uma String; no entanto, defini-lo como `ID`significando que não se destina a ser legível por humanos.

Na maioria das implementações de serviços GraphQL, existe também uma maneira de especificar tipos escalares personalizados. Por exemplo, poderíamos definir um `Date`tipo:

```
Data escalar
```

Então cabe à nossa implementação definir como esse tipo deve ser serializado, desserializado e validado. Por exemplo, você pode especificar que o `Date`tipo sempre deve ser serializado em um registro de data e hora inteiro, e seu cliente deve saber esperar esse formato para qualquer campo de data.

Tipos de enumeração

Também chamados de *Enums* , os tipos de enumeração são um tipo especial de escalar que é restrito a um conjunto particular de valores permitidos. Isso permite que você:

1. Valide se algum argumento desse tipo é um dos valores permitidos
2. Comunicar através do sistema de tipos que um campo sempre será um de um conjunto finito de valores

Aqui está o que uma definição enum pode parecer na linguagem do esquema GraphQL:

```
enum Episódio {
  NOVA ESPERANÇA
  IMPÉRIO
  JEDI
}
```

Isto significa que sempre que usar o tipo `Episode`em nosso esquema, esperamos que ele seja exatamente um dos `NEWHOPE`, `EMPIRE`ou `JEDI`.

Observe que as implementações de serviço GraphQL em vários idiomas terão sua própria maneira específica de idioma para lidar com enums. Em idiomas que suportam enums como um cidadão de primeira classe, a implementação pode tirar vantagem disso; em uma linguagem como JavaScript sem suporte a enum, esses valores podem ser mapeados internamente para um conjunto de inteiros. No entanto, esses detalhes não vazam para o cliente, que podem operar inteiramente em termos dos nomes de seqüência de caracteres dos valores enum.

Listas e Não-Nulo

Tipos de objeto, escalares e enums são os únicos tipos de tipos que você pode definir no GraphQL. Mas quando você usa os tipos em outras partes do esquema ou em suas declarações de variáveis de consulta, pode aplicar *modificadores de tipo* adicionais que afetam a validação desses valores. Vamos ver um exemplo:

```
tipo Character {
  name : String !
  aparece em : [ episódio ] !
}
```

Aqui, estamos usando um `String`tipo e marcando como *Non-Null* , adicionando um ponto de exclamação `!`após o nome do tipo. Isso significa que nosso servidor sempre espera retornar um valor não nulo para esse campo e, se ele acabar obtendo um valor nulo que realmente acionará um erro de execução do GraphQL, informando ao cliente que algo deu errado.

O modificador de tipo Não Nulo também pode ser usado ao definir argumentos para um campo, o que fará com que o servidor GraphQL retorne um erro de validação se um valor nulo for passado como esse argumento, seja na cadeia do GraphQL ou nas variáveis.

```
consulta DroidById ( $ id : ID !) {

 droid ( id : $ id ) {

   nome

 }

}

{

 "id" : null
```

```
}

{

 "erros" : [

   {

     "message" : "Variable \" $ id \ "do tipo requerido \" ID! \ "não foi
fornecido." ,

     "localizações" : [

       {

         "line" : 1 ,

         "coluna" : 17

       }

     ]

   }

 ]

}
```

As listas funcionam de maneira semelhante: Podemos usar um modificador de tipo para marcar um tipo como um `List`, o que indica que esse campo retornará uma matriz desse tipo. Na linguagem do esquema, isso é denotado colocando o tipo entre colchetes `[` e `]`. Ele funciona da mesma forma para

argumentos, em que a etapa de validação esperará uma matriz para esse valor.

Os modificadores Non-Null e List podem ser combinados. Por exemplo, você pode ter uma lista de strings não nulos:

```
myField : [ String ! ]
```

Isso significa que a *própria lista* pode ser nula, mas não pode ter nenhum membro nulo. Por exemplo, em JSON:

```
myField : null // válido
myField : [ ] // válido
myField : [ 'a' , 'b' ] // válido
myField : [ 'a' , null , 'b' ] // erro
```

Agora, digamos que definimos uma lista não nula de strings:

```
myField : [ String ] !
```

Isso significa que a lista em si não pode ser nula, mas pode conter valores nulos:

```
myField : null // erro
myField : [ ] // válido
myField : [ 'a' , 'b' ] // válido
myField : [ 'a' , null , 'b' ] // válido
```

Você pode aninhar arbitrariamente qualquer número de modificadores Non-Null e List, de acordo com suas necessidades.

Interfaces

Como muitos sistemas de tipos, o GraphQL suporta interfaces. Uma *Interface* é um tipo abstrato que inclui um determinado conjunto de campos que um tipo deve incluir para implementar a interface.

Por exemplo, você poderia ter uma interface `Character`que representa qualquer personagem da trilogia Guerra nas Estrelas:

```
Caractere de interface {
  id : ID !
  nome : String !
  amigos : [ Personagem ]
  appearIn : [ Episode ] !
}
```

Isso significa que qualquer tipo que *implemente* `Character` precisa ter esses campos exatos, com esses argumentos e tipos de retorno.

Por exemplo, aqui estão alguns tipos que podem ser implementados `Character`:

```
tipo Human implementa Character {
  id : ID !
  nome : String !
  amigos : [ Personagem ]
  appearIn : [ Episode ] !
  naves espaciais : [ Starship ]
  totalCredits : Int
}

Digite Droid implementa Character {
  id : ID !
  nome : String !
  amigos : [ Personagem ]
  appearIn : [ Episode ] !
  primaryFunction : String
}
```

Você pode ver que esses dois tipos de ter todos os campos da `Character`interface, mas também trazer em campos extras, `totalCredits`, `starships`e `primaryFunction`, que são específicos para esse tipo particular de caráter.

Interfaces são úteis quando você deseja retornar um objeto ou conjunto de objetos, mas esses podem ser de vários tipos diferentes.

Por exemplo, observe que a consulta a seguir produz um erro:

```
consulta HeroForEpisode ( $ ep : Episode !) {

 herói ( episódio : $ ep ) {

   nome

   função primária

 }

}

{

 "ep" : "JEDI"

}

{

 "erros" : [

   {

     "message" : "Não foi possível consultar o campo \" primaryFunction \ "no
tipo \" Caracteres \ ". Você quis usar um fragmento in-line em \" Droid \ "?" ,

     "localizações" : [

       {
```

```
          "linha" : 4 ,
          "coluna" : 5
        }
      ]
    }
  ]
}
```

O `hero`campo retorna o tipo `Character`, o que significa que pode ser um `Human`ou um `Droid`dependendo do `episode`argumento. Na consulta acima, você só pode pedir campos que existem na `Character`interface, o que não inclui `primaryFunction`.

Para solicitar um campo em um tipo de objeto específico, você precisa usar um fragmento embutido:

```
consulta HeroForEpisode ( $ ep : Episode !) {
 herói ( episódio : $ ep ) {
   nome
   ... no Droid {
     função primária
   }
```

```
 }
}
{
 "ep" : "JEDI"
}
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "primaryFunction" : "Astromech"
   }
 }
}
```

Saiba mais sobre isso na seção de fragmentos em linha no guia de consulta.

Tipos de união

Os tipos de união são muito semelhantes às interfaces, mas não especificam campos comuns entre os tipos.

```
union SearchResult = Humano | Droid | Nave estelar
```

Sempre que retornamos um `SearchResult`tipo em nosso esquema, podemos obter um `Human`, um `Droid`ou um `Starship`. Observe que os membros de um tipo de união precisam ser tipos de objetos concretos; você não pode criar um tipo de união de interfaces ou outras uniões.

Nesse caso, se você consultar um campo que retorna o `SearchResult`tipo de união, será necessário usar um fragmento condicional para poder consultar qualquer campo:

```
{
 search ( text : "an" ) {
   ... on Human {
     nome
     altura
   }
   ... no Droid {
     nome
     função primária
   }
   ... na nave espacial {
     nome
```

```
      comprimento
    }
  }
}
{
  "data" : {
    "pesquisa" : [
      {
        "nome" : "Han Solo" ,
        "altura" : 1,8
      }
      {
        "nome" : "Leia Organa" ,
        "altura" : 1,5
      }
      {
        "name" : "TIE Advanced x1" ,
        "comprimento" : 9.2
```

```
      }
    ]
  }
}
```

Tipos de entrada

Até agora, falamos apenas sobre a transmissão de valores escalares, como enums ou strings, como argumentos em um campo. Mas você também pode facilmente passar objetos complexos. Isto é particularmente valioso no caso de mutações, onde você pode querer passar um objeto inteiro para ser criado. Na linguagem do esquema GraphQL, os tipos de entrada são exatamente iguais aos tipos de objetos regulares, mas com a palavra-chave em `input`vez de `type`:

```
input ReviewInput {
  estrelas : Int !
  comentário : String
}
```

Aqui está como você poderia usar o tipo de objeto de entrada em uma mutação:

```
mutação CreateReviewForEpisode ( $ ep : Episódio ! , $ revisão : ReviewInput !)
{
  createReview ( episódio : $ ep , revisão : $ review ) {
    estrelas
    comentário
```

```
 }
}
{
 "ep" : "JEDI" ,
 "review" : {
   "stars" : 5 ,
   "commentary" : "Este é um ótimo filme!"
 }
}
{
 "data" : {
   "createReview" : {
     "stars" : 5 ,
     "commentary" : "Este é um ótimo filme!"
   }
 }
}
```

Os campos em um tipo de objeto de entrada podem se referir a tipos de objetos de entrada, mas você não pode misturar tipos de entrada e saída em seu esquema. Os tipos de objeto de entrada também não podem ter argumentos em seus campos.

Validação

Usando o sistema de tipos, pode ser predeterminado se uma consulta GraphQL é válida ou não. Isso permite que servidores e clientes informem efetivamente os desenvolvedores quando uma consulta inválida foi criada, sem depender de verificações de tempo de execução.

Para nosso exemplo de Star Wars, o arquivo starWarsValidation-test.js contém várias consultas que demonstram várias invalidações e é um arquivo de teste que pode ser executado para exercer o validador da implementação de referência.

Para começar, vamos dar uma consulta válida e complexa. Esta é uma consulta aninhada, semelhante a um exemplo da seção anterior, mas com os campos duplicados fatorados em um fragmento:

```
{
 hero {
   ... NameAndAppearances
   amigos {
     ... NameAndAppearances
     amigos {
       ... NameAndAppearances
     }
```

```
   }
 }
}
```

Detalhes

```
fragment NameAndAppearances on Character {
 nome
 aparece em
}
```

```
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "appearIn" : [
       "NEWHOPE" ,
       "EMPIRE" ,
       "JEDI"
     ]
     "amigos" : [
```

```
{
  "nome" : "Luke Skywalker" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
  ]
  "amigos" : [
    {
      "nome" : "Han Solo" ,
      "appearIn" : [
        "NEWHOPE" ,
        "EMPIRE" ,
        "JEDI"
      ]
    }
    {
      "nome" : "Leia Organa" ,
```

```
    "appearIn" : [
      "NEWHOPE" ,
      "EMPIRE" ,
      "JEDI"
    ]
  }
  {
    "nome" : "C-3PO" ,
    "appearIn" : [
      "NEWHOPE" ,
      "EMPIRE" ,
      "JEDI"
    ]
  }
  {
    "nome" : "R2-D2" ,
    "appearIn" : [
      "NEWHOPE" ,
```

```
        "EMPIRE" ,
        "JEDI"
      ]
    }
  ]
}
{
  "nome" : "Han Solo" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
  ]
  "amigos" : [
    {
      "nome" : "Luke Skywalker" ,
      "appearIn" : [
        "NEWHOPE" ,
```

```
    "EMPIRE" ,
    "JEDI"
  ]
}
{
  "nome" : "Leia Organa" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
  ]
}
{
  "nome" : "R2-D2" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
```

```
      ]
    }
  ]
}
{
  "nome" : "Leia Organa" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
  ]
  "amigos" : [
    {
      "nome" : "Luke Skywalker" ,
      "appearIn" : [
        "NEWHOPE" ,
        "EMPIRE" ,
        "JEDI"
```

```
  ]
}
{
  "nome" : "Han Solo" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
  ]
}
{
  "nome" : "C-3PO" ,
  "appearIn" : [
    "NEWHOPE" ,
    "EMPIRE" ,
    "JEDI"
  ]
}
```

```
            {
              "nome" : "R2-D2" ,
              "appearIn" : [
                "NEWHOPE" ,
                "EMPIRE" ,
                "JEDI"
              ]
            }
          ]
        }
      ]
    }
  }
}
```

E esta consulta é válida. Vamos dar uma olhada em algumas consultas inválidas ...

Um fragmento não pode se referir a si mesmo ou criar um ciclo, pois isso pode resultar em um resultado ilimitado! Aqui está a mesma consulta acima, mas sem os três níveis explícitos de aninhamento:

```
{
 hero {
   ... NameAndAppearancesAndFriends
 }
}
```

Detalhes

```
fragment NameAndAppearancesAndFriends on Character {
 nome
 aparece em
 amigos {
   ... NameAndAppearancesAndFriends
 }
}
```

```
{
 "erros" : [
   {
     "message" : "Não é possível espalhar o fragmento \"
NameAndAppearancesAndFriends \ "dentro de si mesmo." ,
```

```
    "localizações" : [
      {
        "line" : 11 ,
        "coluna" : 5
      }
    ]
  }
 ]
}
```

Quando consultamos campos, temos que consultar um campo que existe no tipo dado. Então, como `hero`retorna a `Character`, temos que consultar um campo `Character`. Esse tipo não tem um `favoriteSpaceship`campo, portanto, essa consulta é inválida:

```
# INVALID: favoriteSpaceship não existe em Character
{
 hero {
   favoriteSpaceship
 }
}
```

```
{

 "erros" : [

   {

     "message" : "Não foi possível consultar campo \" favoriteSpaceship \ "no
tipo \" Caracteres \ "." ,

     "localizações" : [

       {

         "linha" : 4 ,

         "coluna" : 5

       }

     ]

   }

 ]

}
```


Sempre que consultamos um campo e ele retorna algo diferente de um escalar ou um enum, precisamos especificar quais dados queremos recuperar do campo. Herói retorna um `Character`, e nós estamos solicitando campos como `name`e `appearsIn`sobre ele; se omitirmos isso, a consulta não será válida:

```
# INVALID: o herói não é escalar, então campos são necessários
```

```
{

 herói

}

{

 "erros" : [

   {

     "message" : "Field \" hero \ "do tipo \" Character \ "deve ter uma seleção de subcampos. Você quis dizer \" hero {...} \ "?" ,

     "localizações" : [

       {

         "linha" : 3 ,

         "coluna" : 3

       }

     ]

   }

 ]

}
```

Da mesma forma, se um campo é um escalar, não faz sentido consultar campos adicionais nele, e isso tornará a consulta inválida:

```
# INVALID: o nome é um escalar, então campos não são permitidos

{
 hero {
   name {
     firstCharacterOfName
   }
 }
}

{
 "erros" : [
   {
     "message" : "Field \" name \ "não deve ter uma seleção, pois o tipo \" String! \ "não possui subcampos." ,
     "localizações" : [
       {
         "linha" : 4 ,
         "coluna" : 10
       }
```

```
      ]

    }

  ]

}
```

Anteriormente, observou-se que uma consulta só pode consultar campos no tipo em questão; Quando consultamos o `hero`que retorna a `Character`, só podemos consultar os campos em que existem `Character`. O que acontece se quisermos consultar a função primária de R2-D2s?

```
# INVALID: primaryFunction não existe no caractere

{

 hero {

   nome

   função primária

 }

}

{

 "erros" : [

   {

     "message" : "Não foi possível consultar o campo \" primaryFunction \ "no tipo \" Caracteres \ ". Você quis usar um fragmento in-line em \" Droid \ "?" ,
```

```
      "localizações" : [
        {
          "line" : 5 ,
          "coluna" : 5
        }
      ]
    }
  ]
}
```

Essa consulta é inválida porque `primaryFunction`não é um campo `Character`. Queremos alguma maneira de indicar que desejamos buscar `primaryFunction`se o `Character`for `Droid`, e ignorar esse campo de outra forma. Podemos usar os fragmentos que introduzimos anteriormente para fazer isso. Ao configurar um fragmento definido em `Droid`e incluindo-o, garantimos que apenas consultemos `primaryFunction` onde ele está definido.

```
{
  hero {
    nome
    ... DroidFields
```

```
 }
}
```

Detalhes

```
fragmento DroidFields no Droid {
 função primária
}
```

```
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "primaryFunction" : "Astromech"
   }
 }
}
```

Esta consulta é válida, mas é um pouco detalhada; fragmentos nomeados eram valiosos acima quando os usamos várias vezes, mas estamos usando apenas um deles uma vez. Em vez de usar um fragmento nomeado, podemos usar um fragmento in-line; isso ainda nos permite indicar o tipo que estamos consultando, mas sem nomear um fragmento separado:

```
{
 hero {
   nome
   ... no Droid {
     função primária
   }
 }
}
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "primaryFunction" : "Astromech"
   }
 }
}
```

Isso apenas arranhou a superfície do sistema de validação; Existem várias regras de validação para garantir que uma consulta GraphQL seja semanticamente significativa. A especificação entra em mais detalhes sobre

este tópico na seção "Validação", e o diretório de validação em GraphQL.js contém o código que implementa um validador GraphQL compatível com as especificações.

Execução

Após ser validada, uma consulta GraphQL é executada por um servidor GraphQL que retorna um resultado que espelha a forma da consulta solicitada, normalmente como JSON.

O GraphQL não pode executar uma consulta sem um sistema de tipos, vamos usar um sistema de tipo de exemplo para ilustrar a execução de uma consulta. Essa é uma parte do mesmo tipo de sistema usado nos exemplos desses artigos:

```
tipo Consulta {
  humano ( id : ID ! ) : Humano
}

tipo Humano {
  name : String
  aparece Em : [ Episode ]
  naves : [ Starship ]
}

enum Episódio {
  NOVA ESPERANÇA
  IMPÉRIO
  JEDI
}

tipo Starship {
  name : String
}
```

Para descrever o que acontece quando uma consulta é executada, vamos usar um exemplo para percorrer.

```
 {
```

```
 humano ( id : 1002 ) {
   nome
   aparece em
   naves estelares {
     nome
   }
 }
}
```

```
{
 "data" : {
   "humano" : {
     "nome" : "Han Solo" ,
     "appearIn" : [
       "NEWHOPE" ,
       "EMPIRE" ,
       "JEDI"
     ]
     "naves espaciais" : [
```

```
        {
          "nome" : "Millenium Falcon"
        }
        {
          "name" : "shuttle imperial"
        }
      ]
    }
  }
}
```

Você pode pensar em cada campo em uma consulta GraphQL como uma função ou método do tipo anterior, que retorna o próximo tipo. Na verdade, é exatamente assim que o GraphQL funciona. Cada campo em cada tipo é apoiado por uma função chamada *resolvedor,* que é fornecida pelo desenvolvedor do servidor GraphQL. Quando um campo é executado, o *resolvedor* correspondente é chamado para produzir o próximo valor.

Se um campo produzir um valor escalar como uma string ou um número, a execução será concluída. No entanto, se um campo produzir um valor de objeto, a consulta conterá outra seleção de campos que se aplicam a esse objeto. Isso continua até que os valores escalares sejam atingidos. Consultas GraphQL sempre terminam em valores escalares.

Campos raiz e resolvedores

No nível mais alto de todos os servidores GraphQL, existe um tipo que representa todos os possíveis pontos de entrada na API do GraphQL, geralmente chamado de tipo *Raiz* ou Tipo de *consulta* .

Neste exemplo, nosso tipo de consulta fornece um campo chamado `human`que aceita o argumento `id`. A função de resolução para este campo provavelmente acessa um banco de dados e, em seguida, constrói e retorna um `Human`objeto.

```
Consulta : {
  humano ( obj , args , contexto , info ) {
    contexto de retorno . db . loadHumanByID ( args . id ) . então (
      userData = > new Human ( userData )
    )
  }
}
```

Este exemplo está escrito em JavaScript, no entanto, os servidores GraphQL podem ser construídos em vários idiomas diferentes . Uma função de resolvedor recebe quatro argumentos:

- `obj` O objeto anterior, que para um campo no tipo de consulta raiz, geralmente não é usado.
- `args` Os argumentos fornecidos para o campo na consulta GraphQL.
- `context` Um valor que é fornecido a todos os resolvedores e contém informações contextuais importantes, como o usuário atualmente conectado, ou o acesso a um banco de dados.
- `info`Um valor que contém informações específicas do campo relevantes para a consulta atual, bem como os detalhes do

esquema, também fazem referência ao tipo GraphQLResolveInfo para obter mais detalhes .

Resolvedores Assíncronos

Vamos dar uma olhada mais de perto no que está acontecendo nesta função de resolução.

```
humano ( obj , args , context , info ) {
  contexto de retorno . db . loadHumanByID ( args . id ) . então (
    userData = > new Human ( userData )
  )
}
```

O `context`é usado para fornecer acesso a um banco de dados que é usado para carregar os dados para um usuário pelo `id`fornecido como um argumento na consulta GraphQL. Como o carregamento de um banco de dados é uma operação assíncrona, isso retorna um Promise . Em JavaScript, os Promises são usados para trabalhar com valores assíncronos, mas o mesmo conceito existe em muitos idiomas, geralmente chamados de *Futures* , *Tasks* ou *Deferred* . Quando o banco de dados retorna, podemos construir e retornar um novo `Human`objeto.

Observe que, embora a função de resolução precise estar ciente do Promises, a consulta GraphQL não. Ele simplesmente espera que o `human`campo retorne algo que pode ser solicitado `name`. Durante a execução, o GraphQL esperará que Promises, Futures e Tasks sejam concluídos antes de continuar e o fará com concorrência ideal.

Resolvedores Triviais

Agora que um `Human`objeto está disponível, a execução do GraphQL pode continuar com os campos solicitados nele.

```
Humano : {
  nome ( obj , args , contexto , info ) {
    return obj . name
   }
}
```

Um servidor GraphQL é alimentado por um sistema de tipos que é usado para determinar o que fazer a seguir. Mesmo antes do `human`campo retornar alguma coisa, o GraphQL sabe que o próximo passo será resolver os campos do `Human`tipo, já que o sistema de tipos informa que o `human`campo retornará a `Human`.

Resolver o nome neste caso é muito direto. A função de resolução de nomes é chamada e o `obj`argumento é o `new Human`objeto retornado do campo anterior. Nesse caso, esperamos que o objeto humano tenha uma `name`propriedade que possamos ler e retornar diretamente.

Na verdade, muitas bibliotecas GraphQL permitirão que você omita os resolvedores de maneira simples e apenas presumirá que, se um resolvedor não for fornecido para um campo, uma propriedade com o mesmo nome deverá ser lida e retornada.

Coerção Escalar

Enquanto o `name`campo está sendo resolvido, os campos `appearsIn`e `starships`podem ser resolvidos simultaneamente. O `appearsIn`campo também pode ter um resolvedor trivial, mas vamos dar uma olhada mais de perto:

```
Humano : {
  appearIn ( obj ) {
```

```
    return obj . appearIn // devolve [4, 5, 6]
  }
}
```

Observe que nossas declarações do sistema de tipos `appearsIn`retornarão valores Enum com valores conhecidos, no entanto, essa função está retornando números! De fato, se olharmos para o resultado, veremos que os valores Enum apropriados estão sendo retornados. O que está acontecendo?

Este é um exemplo de coerção escalar. O sistema de tipos sabe o que esperar e converterá os valores retornados por uma função de resolução em algo que mantém o contrato da API. Neste caso, pode haver um Enum definido no nosso servidor que usa números como `4`, `5`e `6`internamente, mas representa-los como valores ENUM em sistema tipo GraphQL.

Lista de resolvedores

Já vimos um pouco do que acontece quando um campo retorna uma lista de itens com o `appearsIn`campo acima. Ele retornou uma *lista* de valores enum e, como esse é o tipo de sistema esperado, cada item da lista foi forçado para o valor enum apropriado. O que acontece quando o `starships`campo é resolvido?

```
Humano : {
  starships ( obj , args , contexto , info ) {
    return obj . starshipIDs . map (
      id = > context . db . loadStarshipByID ( id ) . então (
        shipData = > new Starship ( shipData )
      )
    )
  }
}
```

O resolvedor para este campo não está apenas retornando um Promise, ele está retornando uma *lista* de promessas. O `Human`objeto tinha uma lista de ids

dos `Starships`pilotos, mas precisamos carregar todos esses ids para obter objetos Starship reais.

O GraphQL aguardará todas essas promessas simultaneamente antes de continuar e, quando deixado com uma lista de objetos, continuará simultaneamente a carregar o `name`campo em cada um desses itens.

Produzindo o resultado

À medida que cada campo é resolvido, o valor resultante é colocado em um mapa de valor-chave com o nome do campo (ou alias) como a chave e o valor resolvido como o valor, isso continua nos campos da folha inferior da consulta todo o caminho de volta até o campo original no tipo de consulta raiz. Coletivamente, eles produzem uma estrutura que espelha a consulta original, que pode ser enviada (normalmente como JSON) para o cliente que a solicitou.

Vamos dar uma última olhada na consulta original para ver como todas essas funções de resolução produzem um resultado:

```
{
 humano ( id : 1002 ) {
   nome
   aparece em
   naves estelares {
     nome
```

```
   }
 }
}
{
 "data" : {
   "humano" : {
     "nome" : "Han Solo" ,
     "appearIn" : [
       "NEWHOPE" ,
       "EMPIRE" ,
       "JEDI"
     ]
     "naves espaciais" : [
       {
         "nome" : "Millenium Falcon"
       }
       {
         "name" : "shuttle imperial"
```

```
        }

      ]

    }

  }

}
```

Introspecção

Geralmente, é útil solicitar um esquema GraphQL para obter informações sobre quais consultas ele suporta. O GraphQL nos permite fazer isso usando o sistema de introspecção!

Para nosso exemplo de Star Wars, o arquivo starWarsIntrospection-test.js contém várias consultas que demonstram o sistema de introspecção e é um arquivo de teste que pode ser executado para exercitar o sistema de introspecção da implementação de referência.

Nós projetamos o sistema de tipos, então sabemos quais tipos estão disponíveis, mas se não o fizermos, podemos perguntar ao GraphQL, consultando o `__schema`campo, sempre disponível no tipo de raiz de uma Consulta. Vamos fazer agora e perguntar quais tipos estão disponíveis.

```
{
 __schema {
   types {
     nome
   }
 }
}
```

```
{
 "data" : {
```

```
"__schema" : {
  "tipos" : [
    {
      "nome" : "Consulta"
    }
    {
      "name" : "Episode"
    }
    {
      "nome" : "Personagem"
    }
    {
      "nome" : "ID"
    }
    {
      "name" : "String"
    }
    {
```

```
  "nome" : "Int"
}
{
  "name" : "FriendsConnection"
}
{
  "name" : "FriendsEdge"
}
{
  "name" : "PageInfo"
}
{
  "nome" : "Booleano"
}
{
  "name": "Review"
},
{
```

```
      "name": "SearchResult"
    },
    {
      "name": "Human"
    },
    {
      "name": "LengthUnit"
    },
    {
      "name": "Float"
    },
    {
      "name": "Starship"
    },
    {
      "name": "Droid"
    },
    {
```

```
    "name": "Mutation"
  },
  {
    "name": "ReviewInput"
  },
  {
    "name": "__Schema"
  },
  {
    "name": "__Type"
  },
  {
    "name": "__TypeKind"
  },
  {
    "name" : "__Field"
  }
  {
```

```
        "name" : "__InputValue"
      }
      {
        "name" : "__EnumValue"
      }
      {
        "nome" : "__Diretora"
      }
      {
        "name" : "__DirectiveLocation"
      }
    ]
  }
 }
}
```

Uau, isso é um monte de tipos! O que eles são? Vamos agrupá-los:

- **Consulta, Personagem, Humano, Episódio, Droid** - Estes são os que definimos no nosso sistema de tipos.

- **String, Boolean** - Estes são escalares incorporados que o sistema de tipos forneceu.
- **__Schema, __Tipo, __TypeKind, __Field, __InputValue, __EnumValue, __Directive** - Todos estes são precedidos por um sublinhado duplo, indicando que fazem parte do sistema de introspecção.

Agora, vamos tentar descobrir um bom lugar para começar a explorar quais consultas estão disponíveis. Quando projetamos nosso sistema de tipos, especificamos em que tipo todas as consultas começariam; vamos perguntar ao sistema de introspecção sobre isso!

```
{
 __schema {
   queryType {
     nome
   }
 }
}
```

```
{
 "data" : {
   "__schema" : {
     "queryType" : {
       "nome" : "Consulta"
```

```
      }
    }
  }
}
```

E isso corresponde ao que dissemos na seção do sistema de tipos, que o `Query`tipo é onde vamos começar! Note que a nomeação aqui foi apenas por convenção; nós poderíamos ter nomeado nosso `Query`tipo qualquer outra coisa, e ainda teria sido retornado aqui se tivéssemos especificado que era o tipo inicial para consultas. Nomear isso `Query`, no entanto, é uma convenção útil.

Geralmente é útil examinar um tipo específico. Vamos dar uma olhada no `Droid`tipo:

```
{
  __type ( nome : "Droid" ) {
    nome
  }
}
```

```
{
  "data" : {
    "__type" : {
      "name" : "Droid"
```

```
    }
  }
}
```

E se quisermos saber mais sobre o Droid? Por exemplo, é uma interface ou um objeto?

```
{
  __type ( nome : "Droid" ) {
    nome
    tipo
  }
}
{
  "data" : {
    "__type" : {
      "name" : "Droid" ,
      "kind" : "OBJECT"
    }
  }
```

```
}
```

`kind`retorna um `__TypeKind`enum, um dos valores de quem é `OBJECT`. Se perguntássemos, em `Character`vez disso, descobriríamos que é uma interface:

```
{
 __type ( name : "Character" ) {
   nome
   tipo
 }
}
{
 "data" : {
   "__type" : {
     "nome" : "Personagem" ,
     "tipo" : "INTERFACE"
   }
 }
}
```

É útil para um objeto saber quais campos estão disponíveis, então vamos perguntar ao sistema de introspecção sobre `Droid`:

```
{
 __type ( nome : "Droid" ) {
   nome
   campos {
     nome
     tipo {
       nome
       tipo
     }
   }
 }
}
{
 "data" : {
   "__type" : {
     "name" : "Droid" ,
```

```
"campos" : [
  {
    "nome" : "id" ,
    "type" : {
      "nome" : null ,
      "kind" : "NON_NULL"
    }
  }
  {
    "nome" : "nome" ,
    "type" : {
      "nome" : null ,
      "kind" : "NON_NULL"
    }
  }
  {
    "nome" : "amigos" ,
    "type" : {
```

```
      "nome" : null ,
      "tipo" : "LISTA"
    }
  }
  {
    "name" : "friendsConnection" ,
    "type" : {
      "nome" : null ,
      "kind" : "NON_NULL"
    }
  }
  {
    "name" : "appearIn" ,
    "type" : {
      "nome" : null ,
      "kind" : "NON_NULL"
    }
  }
```

```
        {
          "nome" : "primaryFunction" ,
          "type" : {
            "name" : "String" ,
            "kind" : "SCALAR"
          }
        }
      ]
    }
  }
}
```

Esses são os nossos campos que definimos `Droid`!

`id`parece um pouco estranho lá, não tem nome para o tipo. Isso é porque é um tipo de "wrapper" `NON_NULL`. Se consultássemos `ofType`o tipo desse campo, encontraríamos o `ID`tipo lá, nos informando que esse é um ID não nulo.

Da mesma forma, ambos `friends`e `appearsIn`não têm nome, pois são o `LIST`tipo de wrapper. Podemos consultar `ofType`sobre esses tipos, que nos dirão quais são essas listas.

```
{
```

```
 __type ( nome : "Droid" ) {
   nome
   campos {
     nome
     tipo {
       nome
       tipo
       ofType {
         nome
         tipo
       }
     }
   }
 }
}
{
 "data" : {
   "__type" : {
```

```
"name" : "Droid" ,

"campos" : [

  {

    "nome" : "id" ,

    "type" : {

      "nome" : null ,

      "tipo" : "NON_NULL" ,

      "ofType" : {

        "nome" : "ID" ,

        "kind" : "SCALAR"

      }

    }

  }

  {

    "nome" : "nome" ,

    "type" : {

      "nome" : null ,

      "tipo" : "NON_NULL" ,
```

```
      "ofType" : {
        "name" : "String" ,
        "kind" : "SCALAR"
      }
    }
  }
}
{
  "nome" : "amigos" ,
  "type" : {
    "nome" : null ,
    "kind": "LIST",
    "ofType": {
      "name": "Character",
      "kind": "INTERFACE"
    }
  }
},
{
```

```
    "name": "friendsConnection",
    "type": {
      "name": null,
      "kind": "NON_NULL",
      "ofType": {
        "name": "FriendsConnection",
        "kind": "OBJECT"
      }
    }
  },
  {
    "name": "appearsIn",
    "type": {
      "name": null,
      "kind": "NON_NULL",
      "ofType": {
        "name": null,
        "tipo" : "LISTA"
```

```
            }
          }
        }
        {
          "nome" : "primaryFunction" ,
          "type" : {
            "name" : "String" ,
            "kind" : "SCALAR" ,
            "ofType" : null
          }
        }
      ]
    }
  }
}
```

Vamos terminar com um recurso do sistema de introspecção particularmente útil para ferramentas; vamos pedir ao sistema documentação!

```
{
```

```
 __type ( nome : "Droid" ) {
   nome
   descrição
 }
}
{
 "data" : {
   "__type" : {
     "name" : "Droid" ,
     "description" : "Um personagem mecânico autônomo no universo de Star Wars"
   }
 }
}
```

Assim, podemos acessar a documentação sobre o sistema de tipos usando introspecção e criar navegadores de documentação ou experiências ricas em IDE.

Isso apenas arranhou a superfície do sistema de introspecção; Podemos consultar os valores enum, quais interfaces um tipo implementa e mais. Podemos até introspectar o próprio sistema de introspecção. A especificação entra em mais detalhes sobre este tópico na seção "Introspection", e o arquivo de introspecção no GraphQL.js contém o código que implementa um

sistema de introspecção de consulta GraphQL compatível com as especificações.

Práticas recomendadas do GraphQL

A especificação GraphQL é intencionalmente silenciosa em um punhado de questões importantes enfrentadas por APIs, como lidar com a rede, autorização e paginação. Isso não significa que não haja soluções para esses problemas ao usar o GraphQL, apenas que eles estão fora da descrição sobre o que *é* o GraphQL e, em vez disso, são apenas práticas comuns.

Os artigos desta seção não devem ser tomados como evangelho e, em alguns casos, podem ser legitimamente ignorados em favor de alguma outra abordagem. Alguns artigos introduzem algumas das filosofias desenvolvidas no Facebook em torno da criação e implementação de serviços GraphQL, enquanto outras são sugestões mais táticas para resolver problemas comuns como servir por HTTP e executar autorização.

A seguir, são apresentadas breves descrições de algumas das melhores práticas e posturas mais comuns mantidas pelos serviços GraphQL, no entanto, cada artigo desta seção aprofundará esses e outros tópicos.

HTTP

GraphQL é normalmente servido via HTTP através de um único ponto de extremidade que expressa o conjunto completo de recursos do serviço. Isso está em contraste com as APIs REST, que expõem um conjunto de URLs, cada uma das quais expõe um único recurso. Embora o GraphQL possa ser

usado junto com um conjunto de URLs de recursos, isso pode dificultar o uso com ferramentas como o GraphiQL.

Leia mais sobre isso em Exibição por HTTP .

JSON (com GZIP)

Os serviços GraphQL geralmente respondem usando JSON, mas a especificação GraphQL não exige isso . O JSON pode parecer uma escolha estranha para uma camada de API que promete melhor desempenho de rede, no entanto, como é principalmente texto, ele é compactado excepcionalmente bem com o GZIP.

É encorajado que qualquer serviço GraphQL de produção permita o GZIP e encoraje seus clientes a enviar o cabeçalho:

```
Aceitar - Codificação : gzip
```

O JSON também é muito familiar para desenvolvedores de clientes e APIs e é fácil de ler e depurar. Na verdade, a sintaxe GraphQL é parcialmente inspirada na sintaxe JSON.

Versão

Embora não exista nada que impeça que um serviço GraphQL seja versionado como qualquer outra API REST, o GraphQL tem uma opinião forte sobre como evitar o controle de versões, fornecendo as ferramentas para a evolução contínua de um esquema GraphQL.

Por que a maioria das APIs versão? Quando há controle limitado sobre os dados que são retornados de um endpoint da API, *qualquer alteração* pode ser considerada uma alteração significativa e a interrupção de alterações exige uma nova versão. Se a adição de novos recursos a uma API exigir uma nova versão, surge uma compensação entre liberar com frequência e ter muitas

versões incrementais versus a capacidade de compreensão e manutenção da API.

Por outro lado, o GraphQL retorna apenas os dados explicitamente solicitados, para que novos recursos possam ser adicionados por meio de novos tipos e novos campos nesses tipos sem criar uma alteração significativa. Isso levou a uma prática comum de evitar sempre a quebra de alterações e a exibição de uma API sem versão.

Nulabilidade

A maioria dos sistemas de tipos que reconhecem "null" fornece o tipo comum e a versão *anulável* desse tipo, em que, por padrão, os tipos não incluem "null", a menos que explicitamente declarados. No entanto, em um sistema de tipos GraphQL, todos os campos são *anuláveis* por padrão. Isso ocorre porque há muitas coisas que podem dar errado em um serviço de rede apoiado por bancos de dados e outros serviços. Um banco de dados poderia ficar inativo, uma ação assíncrona poderia falhar, uma exceção poderia ser lançada. Além das falhas do sistema, a autorização pode geralmente ser granular, em que campos individuais em uma solicitação podem ter regras de autorização diferentes.

Por padrão, cada campo pode ser *anulado* , qualquer um desses motivos pode resultar em apenas o campo retornado como "nulo", em vez de ter uma falha completa na solicitação. Em vez disso, o GraphQL fornece variantes não nulas de tipos que garantem aos clientes que, se solicitado, o campo nunca retornará "nulo". Em vez disso, se ocorrer um erro, o campo pai anterior será "nulo".

Ao projetar um esquema GraphQL, é importante ter em mente todos os problemas que poderiam dar errado e se "nulo" for um valor apropriado para um campo com falha. Normalmente é, mas ocasionalmente, não é. Nesses casos, use tipos não nulos para garantir isso.

Paginação

O sistema de tipos GraphQL permite que alguns campos retornem listas de valores , mas deixa a paginação de listas maiores de valores até o designer da API. Há uma ampla variedade de possíveis designs de API para paginação, cada um com vantagens e desvantagens.

Normalmente campos que poderia retornar listas longas aceitar argumentos "primeiros" e "depois" para permitir a especificação de uma região específica de uma lista, onde "depois" é um identificador exclusivo de cada um dos valores da lista.

Por fim, projetar APIs com paginação rica em recursos levou a um padrão de práticas recomendadas chamado "Conexões". Algumas ferramentas do cliente para GraphQL, como o Relay , conhecem o padrão Connections e podem fornecer automaticamente suporte automático para paginação do lado do cliente quando uma API do GraphQL emprega esse padrão.

Leia mais sobre isso no artigo sobre paginação .

Batching e Cache do Lado do Servidor

O GraphQL é projetado de forma a permitir que você escreva código limpo no servidor, onde cada campo em cada tipo tem uma função focalizada de propósito único para resolver esse valor. No entanto, sem consideração adicional, um serviço GraphQL ingênuo pode ser muito "tagarela" ou carregar repetidamente dados de seus bancos de dados.

Isso é comumente resolvido por uma técnica de lotes, em que várias solicitações de dados de um back-end são coletadas em um curto período de tempo e enviadas em um único pedido para um banco de dados ou

microsserviço subjacente usando uma ferramenta como o DataLoader do Facebook .

Práticas recomendadas do GraphQL

A especificação GraphQL é intencionalmente silenciosa em um punhado de questões importantes enfrentadas por APIs, como lidar com a rede, autorização e paginação. Isso não significa que não haja soluções para esses problemas ao usar o GraphQL, apenas que eles estão fora da descrição sobre o que *é* o GraphQL e, em vez disso, são apenas práticas comuns.

Os artigos desta seção não devem ser tomados como evangelho e, em alguns casos, podem ser legitimamente ignorados em favor de alguma outra abordagem. Alguns artigos introduzem algumas das filosofias desenvolvidas no Facebook em torno da criação e implementação de serviços GraphQL, enquanto outras são sugestões mais táticas para resolver problemas comuns como servir por HTTP e executar autorização.

A seguir, são apresentadas breves descrições de algumas das melhores práticas e posturas mais comuns mantidas pelos serviços GraphQL, no entanto, cada artigo desta seção aprofundará esses e outros tópicos.

HTTP

GraphQL é normalmente servido via HTTP através de um único ponto de extremidade que expressa o conjunto completo de recursos do serviço. Isso está em contraste com as APIs REST, que expõem um conjunto de URLs,

cada uma das quais expõe um único recurso. Embora o GraphQL possa ser usado junto com um conjunto de URLs de recursos, isso pode dificultar o uso com ferramentas como o GraphiQL.

Leia mais sobre isso em Exibição por HTTP .

JSON (com GZIP)

Os serviços GraphQL geralmente respondem usando JSON, mas a especificação GraphQL não exige isso . O JSON pode parecer uma escolha estranha para uma camada de API que promete melhor desempenho de rede, no entanto, como é principalmente texto, ele é compactado excepcionalmente bem com o GZIP.

É encorajado que qualquer serviço GraphQL de produção permita o GZIP e encoraje seus clientes a enviar o cabeçalho:

```
Aceitar - Codificação : gzip
```

O JSON também é muito familiar para desenvolvedores de clientes e APIs e é fácil de ler e depurar. Na verdade, a sintaxe GraphQL é parcialmente inspirada na sintaxe JSON.

Versão

Embora não exista nada que impeça que um serviço GraphQL seja versionado como qualquer outra API REST, o GraphQL tem uma opinião forte sobre como evitar o controle de versões, fornecendo as ferramentas para a evolução contínua de um esquema GraphQL.

Por que a maioria das APIs versão? Quando há controle limitado sobre os dados que são retornados de um endpoint da API, *qualquer alteração* pode ser considerada uma alteração significativa e a interrupção de alterações exige uma nova versão. Se a adição de novos recursos a uma API exigir uma nova

versão, surge uma compensação entre liberar com frequência e ter muitas versões incrementais versus a capacidade de compreensão e manutenção da API.

Por outro lado, o GraphQL retorna apenas os dados explicitamente solicitados, para que novos recursos possam ser adicionados por meio de novos tipos e novos campos nesses tipos sem criar uma alteração significativa. Isso levou a uma prática comum de evitar sempre a quebra de alterações e a exibição de uma API sem versão.

Nulabilidade

A maioria dos sistemas de tipos que reconhecem "null" fornece o tipo comum e a versão *anulável* desse tipo, em que, por padrão, os tipos não incluem "null", a menos que explicitamente declarados. No entanto, em um sistema de tipos GraphQL, todos os campos são *anuláveis* por padrão. Isso ocorre porque há muitas coisas que podem dar errado em um serviço de rede apoiado por bancos de dados e outros serviços. Um banco de dados poderia ficar inativo, uma ação assíncrona poderia falhar, uma exceção poderia ser lançada. Além das falhas do sistema, a autorização pode geralmente ser granular, em que campos individuais em uma solicitação podem ter regras de autorização diferentes.

Por padrão, cada campo pode ser *anulado* , qualquer um desses motivos pode resultar em apenas o campo retornado como "nulo", em vez de ter uma falha completa na solicitação. Em vez disso, o GraphQL fornece variantes não nulas de tipos que garantem aos clientes que, se solicitado, o campo nunca retornará "nulo". Em vez disso, se ocorrer um erro, o campo pai anterior será "nulo".

Ao projetar um esquema GraphQL, é importante ter em mente todos os problemas que poderiam dar errado e se "nulo" for um valor apropriado para

um campo com falha. Normalmente é, mas ocasionalmente, não é. Nesses casos, use tipos não nulos para garantir isso.

Paginação

O sistema de tipos GraphQL permite que alguns campos retornem listas de valores , mas deixa a paginação de listas maiores de valores até o designer da API. Há uma ampla variedade de possíveis designs de API para paginação, cada um com vantagens e desvantagens.

Normalmente campos que poderia retornar listas longas aceitar argumentos "primeiros" e "depois" para permitir a especificação de uma região específica de uma lista, onde "depois" é um identificador exclusivo de cada um dos valores da lista.

Por fim, projetar APIs com paginação rica em recursos levou a um padrão de práticas recomendadas chamado "Conexões". Algumas ferramentas do cliente para GraphQL, como o Relay , conhecem o padrão Connections e podem fornecer automaticamente suporte automático para paginação do lado do cliente quando uma API do GraphQL emprega esse padrão.

Leia mais sobre isso no artigo sobre paginação .

Batching e Cache do Lado do Servidor

O GraphQL é projetado de forma a permitir que você escreva código limpo no servidor, onde cada campo em cada tipo tem uma função focalizada de propósito único para resolver esse valor. No entanto, sem consideração adicional, um serviço GraphQL ingênuo pode ser muito "tagarela" ou carregar repetidamente dados de seus bancos de dados.

Isso é comumente resolvido por uma técnica de lotes, em que várias solicitações de dados de um back-end são coletadas em um curto período de

tempo e enviadas em um único pedido para um banco de dados ou microsserviço subjacente usando uma ferramenta como o DataLoader do Facebook .

Pensando em Gráficos

É Gráficos todo o caminho *

Com o GraphQL, você modela o domínio do seu negócio como um gráfico

Os gráficos são ferramentas poderosas para modelar muitos fenômenos do mundo real porque se assemelham a nossos modelos mentais naturais e descrições verbais do processo subjacente. Com o GraphQL, você modela seu domínio de negócios como um gráfico definindo um esquema; dentro de seu esquema, você define diferentes tipos de nós e como eles se conectam / se relacionam uns com os outros. No cliente, isso cria um padrão semelhante à programação orientada a objeto: tipos que fazem referência a outros tipos. No servidor, como o GraphQL apenas define a interface, você tem a liberdade de usá-lo com qualquer backend (novo ou legado!).

Idioma compartilhado

Nomear as coisas é uma parte difícil, mas importante, da criação de APIs intuitivas

Pense no seu esquema GraphQL como uma linguagem compartilhada expressiva para sua equipe e seus usuários. Para criar um bom esquema, examine a linguagem cotidiana que você usa para descrever sua empresa.

Por exemplo, vamos tentar descrever um aplicativo de e-mail em inglês simples:

- Um usuário pode ter várias contas de email
- Cada conta de e-mail tem um endereço, caixa de entrada, rascunhos, itens excluídos e itens enviados
- Cada email tem remetente, data de recebimento, assunto e corpo
- Os usuários não podem enviar um email sem um endereço de destinatário

Nomear as coisas é uma parte difícil, mas importante, da criação de APIs intuitivas, portanto, reserve um tempo para pensar cuidadosamente sobre o que faz sentido para o domínio do problema e para os usuários. Sua equipe deve desenvolver um entendimento compartilhado e um consenso dessas regras de domínio de negócios, porque você precisará escolher nomes intuitivos e duráveis para nós e relacionamentos no esquema GraphQL. Tente imaginar algumas das consultas que você deseja executar:

Buscar o número de e-mails não lidos na minha caixa de entrada para todas as minhas contas

```
{
  accounts {
    inbox {
      unreadEmailCount
    }
  }
}
```

Buscar as "informações de visualização" dos primeiros 20 rascunhos na conta principal

```
{
  mainAccount {
    rascunhos ( primeiro : 20 ) {
      ... previewInfo
     }
```

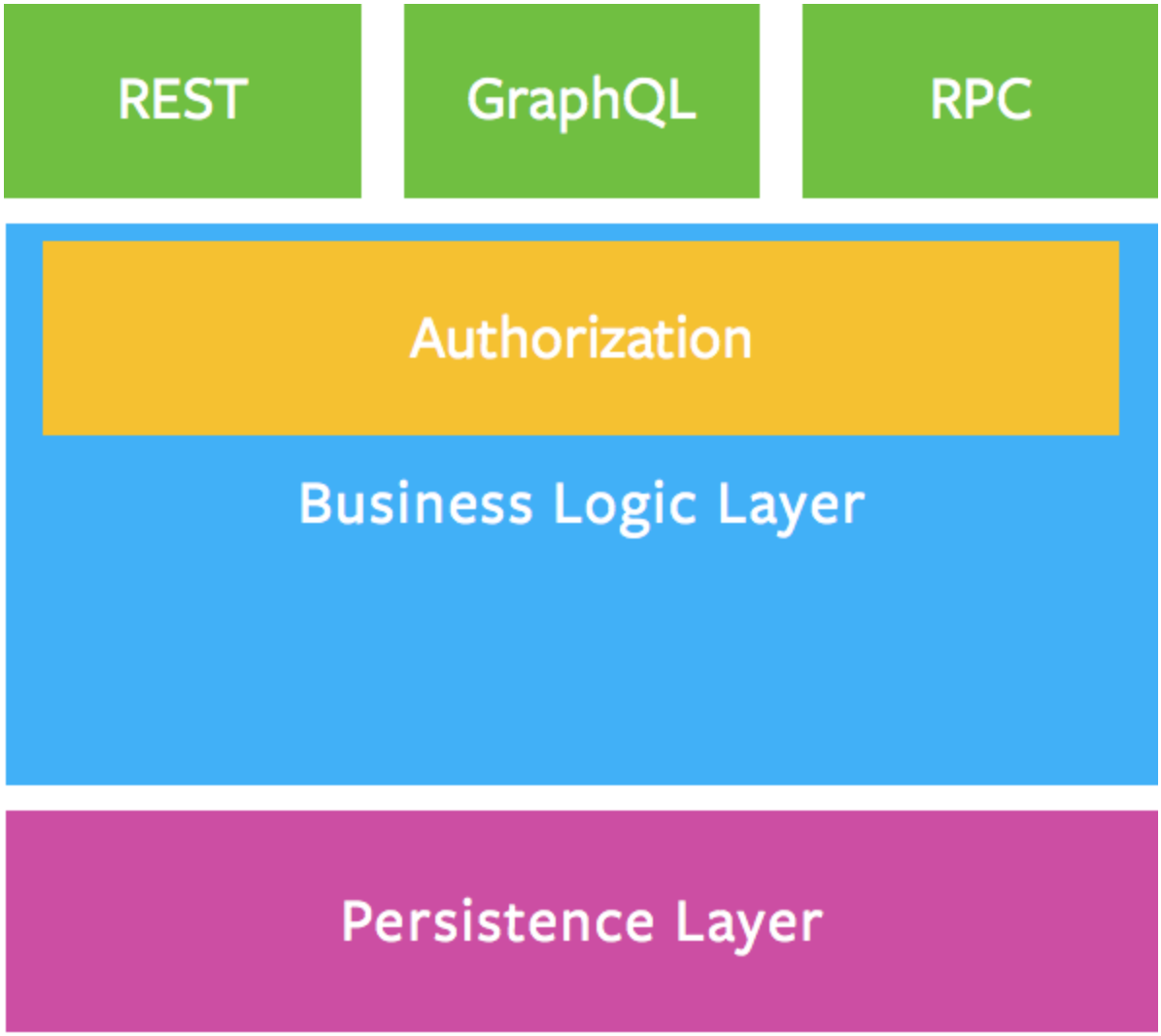
```
  }
}

fragment previewInfo no email {
  sujeito
  bodyPreviewSentence
}
```

Camada de lógica de negócios

Sua camada de lógica de negócios deve agir como a única fonte de verdade para impor regras de domínio de negócios

Onde você deve definir a lógica de negócios real? Onde você deve realizar verificações de validação e autorização? A resposta: dentro de uma camada dedicada de lógica de negócios. Sua camada de lógica de negócios deve agir como a única fonte de verdade para impor regras de domínio de negócios.



No diagrama acima, todos os pontos de entrada (REST, GraphQL e RPC) no sistema serão processados com as mesmas regras de validação, autorização e tratamento de erros.

Trabalhando com dados herdados

Prefira construir um esquema GraphQL que descreva como os clientes usam os dados, em vez de espelhar o esquema do banco de dados legado.

Às vezes, você se verá trabalhando com fontes de dados herdadas que não refletem perfeitamente como os clientes consomem os dados. Nesses casos, prefira criar um esquema GraphQL que descreva como os clientes usam os dados, em vez de espelhar o esquema de banco de dados herdado.

Crie seu esquema GraphQL para expressar "o que" em vez de "como". Então você pode melhorar seus detalhes de implementação sem quebrar a interface com clientes mais antigos.

Um passo de cada vez

Obtenha validação e feedback com mais frequência

Não tente modelar todo o seu domínio comercial de uma só vez. Em vez disso, crie apenas a parte do esquema necessária para um cenário de cada vez. Ao expandir gradualmente o esquema, você obterá validação e feedback com mais frequência para orientá-lo na construção da solução correta.

Servindo por HTTP

O HTTP é a opção mais comum para o protocolo cliente-servidor ao usar o GraphQL devido à sua onipresença. Aqui estão algumas diretrizes para configurar um servidor GraphQL para operar sobre HTTP.

Pipeline de Pedido da Web

A maioria dos frameworks web modernos usa um modelo de pipeline onde os pedidos são passados através de uma pilha de middlewares (filtros / plugins da AKA). À medida que a solicitação flui pelo pipeline, ela pode ser

inspecionada, transformada, modificada ou finalizada com uma resposta. O GraphQL deve ser colocado após todo o middleware de autenticação, para que você tenha acesso à mesma sessão e às informações do usuário que faria nos manipuladores de terminal HTTP.

URIs, rotas

O HTTP é comumente associado ao REST, que usa "recursos" como seu conceito principal. Em contraste, o modelo conceitual do GraphQL é um gráfico de entidade. Como resultado, as entidades no GraphQL não são identificadas por URLs. Em vez disso, um servidor GraphQL opera em uma única URL / terminal, geralmente `/graphql`, e todas as solicitações GraphQL para um determinado serviço devem ser direcionadas a esse terminal.

Métodos HTTP, cabeçalhos e corpo

Seu servidor HTTP GraphQL deve manipular os métodos HTTP GET e POST.

Solicitação GET

Ao receber uma solicitação HTTP GET, a consulta GraphQL deve ser especificada na string de consulta "query". Por exemplo, se quiséssemos executar a seguinte consulta GraphQL:

```
{
  me {
    nome
  }
}
```

Este pedido pode ser enviado via HTTP GET assim:

```
http : / / MyAPI / graphql ? query = { me { name } }
```

As variáveis de consulta podem ser enviadas como uma string codificada em JSON em um parâmetro de consulta adicional chamado `variables`. Se a consulta contiver várias operações nomeadas, um `operationName`parâmetro de consulta poderá ser usado para controlar qual delas deve ser executada.

Pedido de POST

Uma solicitação POST GraphQL padrão deve usar o `application/json`tipo de conteúdo e incluir um corpo codificado por JSON da seguinte forma:

```
{
  "Consulta" : "..." ,
  "operationName" : "..." ,
  "variáveis" : { "myVariable" : "someValue" , . . . }
}
```

`operationName`e `variables`são campos opcionais. `operationName`é necessário apenas se várias operações estiverem presentes na consulta.

Além do acima, recomendamos o suporte a dois casos adicionais:

- Se o parâmetro de string de consulta "query" estiver presente (como no exemplo GET acima), ele deverá ser analisado e tratado da mesma maneira que o caso HTTP GET.
- Se o cabeçalho "Content / Type" do "application / graphql" estiver presente, trate o conteúdo do corpo HTTP POST como a string de consulta GraphQL.

Se você estiver usando express-graphql, você já terá esses comportamentos de graça.

Resposta

Independentemente do método pelo qual a consulta e as variáveis foram enviadas, a resposta deve ser retornada no corpo da solicitação no formato JSON. Como mencionado na especificação, uma consulta pode resultar em alguns dados e alguns erros, e esses devem ser retornados em um objeto JSON do formulário:

```
{
  "dados" : { . . . } ,
  "errors" : [ . . . ]
}
```

Se não houver erros retornados, o `"errors"`campo não deve estar presente na resposta. Se nenhum dado for retornado, de acordo com a especificação GraphQL , o `"data"`campo só deve ser incluído se o erro ocorrer durante a execução.

GraphiQL

O GraphiQL é útil durante o teste e desenvolvimento, mas deve ser desativado na produção por padrão. Se você estiver usando express-graphql, você pode alterná-lo com base na variável de ambiente NODE_ENV:

```
app . use ( '/ graphql' , graphqlHTTP ( {
  esquema : MySessionAwareGraphQLSchema ,
  graphiql : process . env . NODE_ENV === 'desenvolvimento' ,
} ) ) ;
```

Nó

Se você estiver usando o NodeJS, recomendamos usar express-graphql ou graphql-server .

Autorização

Delegar a lógica de autorização à camada de lógica de negócios

Autorização é um tipo de lógica de negócios que descreve se um determinado usuário / sessão / contexto tem permissão para executar uma ação ou ver uma parte dos dados. Por exemplo:

“Somente autores podem ver seus rascunhos”

Impor esse tipo de comportamento deve acontecer na camada de lógica de negócios . É tentador colocar a lógica de autorização na camada GraphQL assim:

```
var postType = new GraphQLObjectType ( {
  nome : 'Post' ,
  campos : {
    body : {
      type : GraphQLString ,
      resolução : ( post , args , contexto , { rootValue } ) = > {
        // retorna o corpo da postagem apenas se o usuário for o autor da
postagem
        if ( contexto . usuário && ( contexto .usuário . id === postar .
authorId ) ) {
          postagem de retorno . corpo ;
        }
        retornar null ;
      }
    }
  }
} ) ;
```

Observe que definimos "autor possui um post" verificando se o `authorId`campo da postagem é igual ao usuário atual `id`. Você pode identificar o problema? Seria necessário duplicar esse código para cada ponto de entrada no serviço. Então, se a lógica de autorização não for mantida perfeitamente em sincronia, os usuários podem ver dados diferentes dependendo de qual API eles usam, mas podemos evitar isso tendo uma única fonte de verdade para autorização.

Definindo a lógica de autorização dentro do resolvedor é bom quando aprender GraphQL ou prototipagem. No entanto, para uma base de código de produção, delegue a lógica de autorização à camada de lógica de negócios. Aqui está um exemplo:

```
// A lógica de autorização reside dentro de postRepository
var postRepository = require ( 'postRepository' ) ;

var postType = new GraphQLObjectType ( {
  nome : 'Post' ,
  campos : {
    corpo : {
      Tipo : GraphQLString ,
      resolver : ( pós , args , contexto , { rootValue } ) = > {
        retornar postRepository . getBody ( contexto . utilizador , pós ) ;
      }
    }
  }
} ) ;
```

No exemplo acima, vemos que a camada de lógica de negócios requer que o chamador forneça um objeto de usuário. Se você estiver usando GraphQL.js, o objeto User deve ser preenchido no `context`argumento ou `rootValue`no quarto argumento do resolvedor.

Recomendamos transmitir um objeto Usuário totalmente hidratado em vez de um token opaco ou uma chave de API para sua camada de lógica comercial. Dessa forma, podemos lidar com as distintas preocupações de autenticação

e autorização em diferentes estágios do pipeline de processamento de solicitações.

Paginação

Modelos de paginação diferentes permitem diferentes capacidades do cliente

Um caso de uso comum no GraphQL é atravessar o relacionamento entre conjuntos de objetos. Há várias maneiras diferentes pelas quais esses relacionamentos podem ser expostos no GraphQL, fornecendo um conjunto variado de recursos para o desenvolvedor do cliente.

Plurais

A maneira mais simples de expor uma conexão entre objetos é com um campo que retorna um tipo plural. Por exemplo, se quiséssemos obter uma lista de amigos do R2-D2, poderíamos pedir por todos eles:

```
{
 hero {
   nome
   amigos {
     nome
   }
```

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

```
  }

}
```

Cortando

Rapidamente, porém, percebemos que existem comportamentos adicionais que um cliente pode querer. Um cliente pode querer especificar quantos amigos deseja buscar; talvez eles só queiram os dois primeiros. Então, gostaríamos de expor algo como:

```
{
  hero {
    nome
    amigos ( primeiro : 2 ) {
      nome
    }
  }
}
```

Mas, se acabássemos de buscar os dois primeiros, poderíamos querer também folhear a lista; uma vez que o cliente busca os dois primeiros amigos, eles podem querer enviar um segundo pedido para pedir os próximos dois amigos. Como podemos ativar esse comportamento?

Paginação e Bordas

Existem várias maneiras de fazer paginação:

- Poderíamos fazer algo como `friends(first:2 offset:2)`pedir os próximos dois da lista.

- Poderíamos fazer algo como `friends(first:2 after:$friendId)`pedir os próximos dois depois do último amigo que buscamos.
- Poderíamos fazer algo como `friends(first:2 after:$friendCursor)`, onde pegamos um cursor do último item e o usamos para paginar.

Em geral, descobrimos que **a paginação baseada em cursor** é a mais poderosa das projetadas. Especialmente se os cursores são opacos, deslocamento ou paginação baseada em ID pode ser implementada usando paginação baseada em cursor (fazendo o cursor o deslocamento ou o ID) e usando cursores dá flexibilidade adicional se o modelo de paginação for alterado no futuro. Como um lembrete de que os cursores são opacos e que seu formato não deve ser invocado, sugerimos codificá-los na base64.

Isso nos leva a um problema; Apesar; Como podemos obter o cursor do objeto? Nós não queremos que o cursor viva no `User`tipo; é uma propriedade da conexão, não do objeto. Então, poderíamos querer introduzir uma nova camada de indireção; nosso `friends`campo deve nos fornecer uma lista de arestas e uma aresta tem um cursor e o nó subjacente:

```
{
  hero {
    nome
    amigos ( primeiro : 2 ) {
      arestas {
        nó {
          nome
        }
        cursor
      }
    }
  }
}
```

O conceito de uma borda também é útil se houver informações específicas da borda, em vez de um dos objetos. Por exemplo, se quiséssemos expor "tempo de amizade" na API, tê-lo ao vivo no limite é um lugar natural para colocá-lo.

Fim de lista, contagens e conexões

Agora temos a capacidade de paginar através da conexão usando cursores, mas como sabemos quando chegamos ao fim da conexão? Temos que continuar consultando até recebermos uma lista vazia, mas gostaríamos que a conexão nos informasse quando chegamos ao fim, para que não precisemos dessa solicitação adicional. Da mesma forma, e se quisermos saber informações adicionais sobre a conexão em si; Por exemplo, quantos amigos totais o R2-D2 possui?

Para resolver esses dois problemas, nosso `friends`campo pode retornar um objeto de conexão. O objeto de conexão terá então um campo para as arestas, além de outras informações (como contagem total e informações sobre a existência de uma próxima página). Então, nossa consulta final pode parecer mais:

```
{
  hero {
    nome
    amigos ( primeiro : 2 ) {
      totalCount
      bordas {
        node {
          nome
        }
        cursor
      }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
```

```
  }
}
```

Observe que também podemos incluir `endCursor`e `startCursor`neste `PageInfo`objeto. Dessa forma, se não precisarmos de nenhuma das informações adicionais contidas na borda, não precisamos consultar as bordas, pois recebemos os cursores necessários para a paginação `pageInfo`. Isso leva a uma melhoria potencial de usabilidade para conexões; em vez de apenas expor a `edges`lista, poderíamos também expor uma lista dedicada de apenas os nós, para evitar uma camada de indireção.

Modelo completo de conexão

Claramente, isso é mais complexo do que o nosso projeto original de ter apenas um plural! Mas ao adotar esse design, nós desbloqueamos vários recursos para o cliente:

- A capacidade de paginar pela lista.
- A capacidade de solicitar informações sobre a conexão em si, como `totalCount`ou `pageInfo`.
- A capacidade de pedir informações sobre a própria borda, como `cursor`ou `friendshipTime`.
- A capacidade de alterar a forma como o nosso backend faz paginação, uma vez que o usuário usa apenas cursores opacos.

Para ver isso em ação, há um campo adicional no esquema de exemplo chamado `friendsConnection`, que expõe todos esses conceitos. Você pode verificá-lo na consulta de exemplo. Tente remover o `after`parâmetro para `friendsConnection`ver como a paginação será afetada. Além disso, tente substituir o `edges`campo pelo campo auxiliar `friends`na conexão, o que permite que você chegue diretamente à lista de amigos sem a camada de borda adicional de indireção, quando isso é apropriado para os clientes.

```
{
 hero {
   nome
   friendsConnection ( primeiro : 2 depois de : "Y3Vyc29yMQ ==" ) {
     totalCount
     bordas {
       node {
         nome
       }
       cursor
     }
     pageInfo {
       endCursor
       hasNextPage
     }
   }
 }
}
```

```
{
 "data" : {
   "hero" : {
     "nome" : "R2-D2" ,
     "friendsConnection" : {
       "totalCount" : 3 ,
       "arestas" : [
         {
           "nó" : {
             "name" : "Han Solo"
           }
           "cursor" : "Y3Vyc29yMg =="
         }
         {
           "nó" : {
             "nome" : "Leia Organa"
           }
           "cursor" : "Y3Vyc29yMw =="
```

```
        }
      ]
      "pageInfo" : {
        "endCursor" : "Y3Vyc29yMw ==" ,
        "hasNextPage" : false
      }
    }
  }
}
}
```