

O'REILLY®



Introdução ao GraphQL

BUSCA DE DADOS COM ABORDAGEM DECLARATIVA PARA APLICAÇÕES WEB MODERNAS

novatec

Eve Porcello & Alex Banks

Eve Porcello e Alex Banks

Novatec

Authorized Portuguese translation of the English edition of Learning GraphQL ISBN 9781492030713 © 2018 Alex Banks and Eve Porcello. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to sell the same.

Tradução em português autorizada da edição em inglês da obra da Learning GraphQL ISBN 9781492030713 © 2018 Alex Banks and Eve Porcello. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., que detém ou controla todos os direitos para publicação e venda desta obra.

Copyright © 2018 da Novatec Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Smirna Cavalheiro

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-710-7

Histórico de edições impressas:

Setembro/2018 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

[Agradecimentos](#)

[Convenções usadas neste livro](#)

[Uso de exemplos de código de acordo com a política da O'Reilly](#)

Capítulo 1 ■ Bem-vindo ao GraphQL

[O que é GraphQL?](#)

[Especificação GraphQL](#)

[Princípios de design do GraphQL](#)

[Origens do GraphQL](#)

[História do transporte de dados](#)

[Remote Procedure Call \(RPC\)](#)

[Simple Object Access Protocol \(SOAP\)](#)

[REST](#)

[Desvantagens do REST](#)

[Overfetching](#)

[Underfetching](#)

[Administrando endpoints REST](#)

[GraphQL no mundo real](#)

[Clientes GraphQL](#)

Capítulo 2 ■ Teoria dos grafos

[Vocabulário da teoria dos grafos](#)

[História da teoria dos grafos](#)

[Árvores e grafos](#)

[Grafos no mundo real](#)

Capítulo 3 ■ A linguagem de consulta GraphQL

[Ferramentas para APIs GraphQL](#)

[GraphiQL](#)

[GraphQL Playground](#)

[API GraphQL públicas](#)

[A consulta GraphQL](#)

[Arestas e conexões](#)

[Fragmentos](#)

[Mutações](#)

[Usando variáveis de consulta](#)

[Subscriptions](#)

[Introspecção](#)

[Árvores sintáticas abstratas](#)

Capítulo 4 ■ **Design de um esquema**

[Definindo tipos](#)

[Tipos](#)

[Tipos escalares](#)

[Enumerados](#)

[Conexões e listas](#)

[Conexões de um para um](#)

[Conexões de um para muitos](#)

[Conexões de muitos para muitos](#)

[Lista de tipos diferentes](#)

[Argumentos](#)

[Filtrando dados](#)

[Mutações](#)

[Tipos input](#)

[Tipo de retorno](#)

[Subscriptions](#)

[Documentação dos esquemas](#)

Capítulo 5 ■ **Criando uma API GraphQL**

[Configuração do projeto](#)

[Resolvers](#)

[Resolvers raiz](#)

[Resolvers de tipo](#)

[Usando entradas e enumerados](#)

[Arestas e conexões](#)

[Escalares personalizados](#)

[apollo-server-express](#)

[Contexto](#)

[Instalando o Mongo](#)

[Adicionando um banco de dados no contexto](#)

[Autorização do GitHub](#)

[Configurando o OAuth do GitHub](#)

[Processo de autorização](#)

[Mutação githubAuth](#)

[Autenticando usuários](#)

[Conclusão](#)

Capítulo 6 ■ **Clientes GraphQL**

[Usando uma API GraphQL](#)

- [Requisições fetch](#)
- [graphql-request](#)
- [Apollo Client](#)
- [Apollo Client com o React](#)
- [Configuração do projeto](#)
- [Configuração do Apollo Client](#)
- [O componente Query](#)
- [Componente Mutation](#)
- [Autorização](#)
- [Autorizando o usuário](#)
- [Identificando o usuário](#)
- [Trabalhando com cache](#)
- [Políticas de busca](#)
- [Persistência do cache](#)
- [Atualizando o cache](#)

Capítulo 7 - GraphQL no mundo real

- [Subscriptions](#)
- [Trabalhando com subscriptions](#)
- [Consumindo subscriptions](#)
- [Upload de arquivos](#)
- [Tratando uploads no servidor](#)
- [Postando uma nova foto com o Apollo Client](#)
- [Segurança](#)
- [Timeout de requisições](#)
- [Limitações nos dados](#)
- [Limitando a profundidade da consulta](#)
- [Limitando a complexidade das consultas](#)
- [Apollo Engine](#)
- [Próximos passos](#)
- [Migração gradual](#)
- [A estratégia de desenvolvimento Schema-First](#)
- [Eventos associados ao GraphQL](#)
- [Comunidade](#)
- [Canais Slack da comunidade](#)

Prefácio

Agradecimentos

Este livro não teria sido publicado sem a ajuda de muitas pessoas incríveis. Tudo começou com a ideia de Ally MacDonald, nossa editora para o livro *Learning React*, que nos incentivou a escrever *Introdução ao GraphQL*. Tivemos muita sorte então em poder trabalhar com Alicia Young, que conduziu o livro até a sua publicação. Agradecemos a Justin Billing, Melanie Yarbrough e Chris Edwards, que apararam todas as arestas durante a produção gráfica extremamente minuciosa.

Durante o processo, tivemos a sorte de receber feedback de Peggy Rayzis e Sashko Stubailo da equipe da Apollo, que compartilharam seus insights e deram dicas muito interessantes sobre os recursos mais recentes. Agradecemos também a Adam Rackis, Garrett McCullough e Shivi Singh, que foram excelentes revisores técnicos.

Escrevemos este livro sobre GraphQL porque amamos o GraphQL. Achamos que você também vai amá-lo.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados,

variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma observação geral.



Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O'Reilly

Materiais suplementares (exemplos de código, exercícios etc.) estão disponíveis para download em <https://github.com/moonhighway/learning-graphql/>.

Este livro está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você poderá usar o código em seus programas e em sua documentação. Você não precisa nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. Porém vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição

geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Learning GraphQL* de Eve Porcello e Alex Banks (O’Reilly). Copyright 2018 Eve Porcello e Alex Banks, 978-1-492-03071-3”.

Se você achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade para nos contatar em permissions@oreilly.com.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora pelo email: novatec@novatec.com.br.

Temos uma página web para este livro, na qual incluimos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição traduzida para o português

<https://www.novatec.com.br/livros/introducao-graphql>

- Página da edição original em inglês

<http://bit.ly/learning-graphql-orm>

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: <http://www.novatec.com.br>.

CAPÍTULO 1

Bem-vindo ao GraphQL

Antes de a rainha da Inglaterra conceder-lhe o título de cavaleiro, Tim Berners-Lee era um programador. Trabalhou no CERN, o laboratório europeu de física de partículas na Suíça, e vivia cercado por um grupo grande de pesquisadores talentosos. Berners-Lee queria ajudar seus colegas a compartilhar suas ideias, então decidiu criar uma rede na qual os cientistas pudessem postar e atualizar informações. Futuramente o projeto se transformou no primeiro servidor e cliente web, e o navegador “WorldWideWeb” (renomeado depois para “Nexus”) foi implantado no CERN (<https://www.w3.org/People/Berners-Lee/Longer.html>) em dezembro de 1990.

Com seu projeto Berners-Lee tornou possível aos pesquisadores visualizar e atualizar conteúdo web em seus próprios computadores. O “WorldWideWeb” era composto de HTML, URLs, um navegador e uma interface WYSIWYG na qual o conteúdo era atualizado.

Atualmente a internet não inclui apenas HTML e um navegador. Ela inclui notebooks. Relógios de pulso. Smartphones. Um chip de RFID (Radio-Frequency Identification, ou Identificação por Radiofrequência) em seu bilhete de teleférico. Um robô que alimenta seu gato enquanto você está viajando.

Os clientes são mais numerosos hoje em dia, mas ainda estamos nos esforçando para fazer a mesma tarefa: carregar dados em algum lugar o mais rápido possível. Queremos que nossas aplicações tenham um bom desempenho porque nossos usuários consideram que temos um alto padrão. Eles esperam que nossos aplicativos funcionem bem em qualquer condição: com 2G em

feature phones¹ ou em uma internet ultrarrápida por fibra, usando computadores desktop com telas enormes. Aplicações rápidas facilitam que mais pessoas interajam com o nosso conteúdo. Aplicações rápidas deixam nossos usuários satisfeitos. E, sim, aplicações rápidas nos permitem ganhar dinheiro.

Fazer com que os dados sejam transmitidos de um servidor para o cliente de forma rápida e previsível é a história da web, no passado, no presente e no futuro. Embora este livro muitas vezes recorra a fatos do passado para apresentar um contexto, estamos aqui para falar de soluções modernas. Estamos aqui para falar do futuro. Estamos aqui para falar do GraphQL.

O que é GraphQL?

GraphQL (<https://www.graphql.org/>) é uma linguagem de consulta (query language) para suas APIs. É também um runtime para atender a consultas com seus dados. O serviço GraphQL é independente do meio de transporte, mas, em geral, é servido sobre HTTP.

Para demonstrar uma consulta GraphQL e sua resposta, vamos observar a SWAPI (<https://graphql.org/swapi-graphql/>), a API Star Wars. A SWAPI é uma API REST (Representational State Transfer, ou Transferência de Estado Representativo) encapsulada com o GraphQL. Podemos usá-la para enviar consultas e receber dados.

Uma consulta GraphQL pede somente os dados de que necessita. A Figura 1.1 é um exemplo de uma consulta GraphQL. A consulta está à esquerda. Pedimos os dados sobre uma pessoa, a Princesa Leia. Obtivemos o registro de Leia Organa porque especificamos que queríamos a quinta pessoa (`personID:5`). Em seguida, pedimos três campos de dados: `name`, `birthYear` e `created`. À direita está a nossa resposta: dados JSON formatados para que estejam de acordo com o formato de nossa consulta. Essa resposta contém somente os dados de que precisamos.

```

1 query {
2   person(personID:5) {
3     name
4     birthYear
5     created
6   }
7 }

```

```

{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "1988Y",
      "created": "2014-12-10T15:20:09.791000Z"
    }
  }
}

```

Figura 1.1 – Consulta de pessoa para a API Star Wars.

Podemos então fazer ajustes na consulta, pois as consultas são interativas. É possível alterá-la e ver um novo resultado. Se adicionarmos o campo `filmConnection`, podemos pedir o título de cada um dos filmes da Princesa Leia, como mostra a Figura 1.2.

```

1 query {
2   person(personID:5) {
3     name
4     birthYear
5     created
6     filmConnection {
7       films {
8         title
9       }
10    }
11  }
12 }

```

```

{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "1988Y",
      "created": "2014-12-10T15:20:09.791000Z",
      "filmConnection": {
        "films": [
          {
            "title": "A New Hope"
          },
          {
            "title": "The Empire Strikes Back"
          },
          {
            "title": "Return of the Jedi"
          },
          {
            "title": "Revenge of the Sith"
          },
          {
            "title": "The Force Awakens"
          }
        ]
      }
    }
  }
}

```

Figura 1.2 – Consulta de conexões.

A consulta é aninhada, e quando é executada pode percorrer objetos relacionados. Isso nos permite fazer uma única requisição

HTTP para dois tipos de dados. Não precisamos fazer várias viagens de ida e volta para explorar diversos objetos. Não recebemos dados adicionais indesejados sobre esses tipos. Com o GraphQL, nossos clientes podem obter todos os dados de que precisam com uma só requisição.

Sempre que uma consulta for executada em um servidor GraphQL, ela será validada em relação a um sistema de tipos. Todo serviço GraphQL define tipos em um esquema GraphQL. Podemos pensar em um sistema de tipos como um modelo para os dados de sua API, com suporte de uma lista de objetos que você definir. Por exemplo, a consulta a uma pessoa feita antes tem um objeto `Person` como base.

```
type Person {  
  id: ID!  
  name: String  
  birthYear: String  
  eyeColor: String  
  gender: String  
  hairColor: String  
  height: Int  
  mass: Float  
  skinColor: String  
  homeworld: Planet  
  species: Species  
  filmConnection: PersonFilmsConnection  
  starshipConnection: PersonStarshipConnection  
  vehicleConnection: PersonVehiclesConnection  
  created: String  
  edited: String  
}
```

O tipo `Person` define todos os campos, juntamente com seus tipos, disponíveis a uma consulta sobre a Princesa Leia. No Capítulo 3, exploraremos melhor o esquema e o sistema de tipos de GraphQL.

O GraphQL muitas vezes é referenciado como uma linguagem *declarativa* de busca de dados. Com isso queremos dizer que os desenvolvedores listarão seus requisitos de dados, informando *quais* dados precisam, sem focar em *como* irão obtê-los. Há

bibliotecas de servidores GraphQL em muitas linguagens diferentes, incluindo C#, Clojure, Elixir, Erlang, Go, Groovy, Java, JavaScript, .NET, PHP, Python, Scala e Ruby².

Neste livro, nosso foco estará na implementação de serviços GraphQL com JavaScript. Todas as técnicas que discutiremos aqui serão aplicáveis ao GraphQL em qualquer linguagem.

Especificação GraphQL

O GraphQL é uma especificação (spec) para comunicação cliente-servidor. O que é uma especificação? Uma especificação descreve os recursos e as características de uma linguagem. Nós nos beneficiamos com as especificações de linguagem porque elas oferecem um vocabulário comum e as melhores práticas para o uso da linguagem pela comunidade.

Um bom exemplo de uma especificação de software que merece destaque é a ECMAScript. De vez em quando, grupos de representantes de empresas de navegadores, de empresas de tecnologia e a comunidade em geral se reúnem e planejam o que deve ser incluído (e o que deve ser deixado de fora) da especificação ECMAScript. O mesmo vale para o GraphQL. Um grupo de indivíduos se reuniu e redigiu o que deve ser incluído na linguagem (e o que deve ser deixado de fora dela). Esse texto serve como diretriz para todas as implementações de GraphQL.

Quando a especificação foi lançada, os criadores do GraphQL também compartilharam uma implementação de referência de um servidor GraphQL em JavaScript – `graphql.js` (<https://github.com/graphql/graphql-js>). Ele é conveniente como um modelo, mas o objetivo dessa implementação de referência não é impor a linguagem que você usará para implementar o seu serviço. É meramente um guia. Depois que tiver uma compreensão da linguagem de consulta e do sistema de tipos, você poderá implementar o seu servidor em qualquer linguagem que quiser.

Se uma especificação e uma implementação são diferentes, o que

está, realmente, na especificação? A especificação descreve a linguagem e a gramática que você deve usar quando escrever consultas. Ela também define um sistema de tipos, além das engines de execução e validação desse sistema de tipos. Exceto por isso, a especificação não é particularmente autoritária. O GraphQL não determina a linguagem a ser usada nem como os dados devem ser armazenados ou quais clientes serão aceitos. A linguagem de consulta oferece diretrizes, mas o design propriamente dito de seu projeto é de sua responsabilidade. (Se quiser explorar melhor o assunto como um todo, você poderá consultar a documentação em <http://facebook.github.io/graphql/>.)

Princípios de design do GraphQL

Apesar de não controlar o modo como você implementará a sua API, o GraphQL apresenta algumas diretrizes sobre como pensar em um serviço³:

Hierárquico

Uma consulta GraphQL é hierárquica. Campos são aninhados em outros campos e a consulta é formatada do mesmo modo como os dados serão devolvidos.

Centrado em produtos

O GraphQL é orientado de acordo com as necessidades de dados do cliente, além da linguagem e do runtime que dão suporte ao cliente.

Tipagem forte

Um servidor GraphQL tem o sistema de tipos do GraphQL como base. No esquema, cada dado tem um tipo específico em relação ao qual será validado.

Consultas especificadas pelo cliente

Um servidor GraphQL oferece os recursos que os clientes têm permissão para consumir.

Introspectivo

A linguagem GraphQL é capaz de fazer consultas ao sistema de tipos do servidor GraphQL.

Agora que temos uma compreensão inicial do que é a especificação GraphQL, vamos ver por que ela foi criada.

Origens do GraphQL

Em 2012, o Facebook decidiu que precisava reconstruir os aplicativos móveis nativos. Os aplicativos para iOS e Android da empresa eram somente wrappers finos em torno das views do site móvel. O Facebook tinha um servidor RESTful e tabelas de dados FQL (versão de SQL do Facebook). Havia problemas de desempenho e os aplicativos, com frequência, apresentavam falhas. Naquele momento, os engenheiros perceberam que precisavam melhorar o modo como os dados estavam sendo enviados às suas aplicações clientes⁴.

A equipe de Lee Byron, Nick Schrock e Dan Schafer decidiu repensar nos dados do lado cliente. Eles decidiram implementar o GraphQL, uma linguagem de consulta que descreveria os recursos e os requisitos dos modelos de dados para as aplicações cliente/servidor da empresa.

Em julho de 2015, a equipe lançou a especificação inicial do GraphQL, além de uma implementação de referência em JavaScript, chamada graphql.js. Em setembro de 2016, o GraphQL deixou sua etapa de “versão preliminar técnica”. Isso significava que o GraphQL estava oficialmente pronto para o ambiente de produção, apesar de já estar sendo usado aí há anos no Facebook. Atualmente, o GraphQL é utilizado em quase todas as buscas de dados no Facebook e no ambiente de produção da IBM, da Intuit, da Airbnb e de outras empresas.

História do transporte de dados

O GraphQL apresenta algumas ideias muito novas, mas todas elas devem ser compreendidas em um contexto histórico do transporte de dados. Quando pensamos em transporte de dados, estamos tentando entender como passar dados de um lado para outro entre os computadores. Pedimos alguns dados de um sistema remoto e esperamos uma resposta.

Remote Procedure Call (RPC)

Nos anos 1960, a RPC (Remote Procedure Call, ou Chamada de Procedimento Remoto) foi inventada. Uma RPC era iniciada pelo cliente, que enviava uma mensagem de requisição a um computador remoto para que fizesse algo. O computador remoto enviava uma resposta ao cliente. Esses computadores eram diferentes dos clientes e servidores que usamos hoje em dia, mas o fluxo de informações era basicamente o mesmo: requisição de alguns dados pelo cliente, obtenção de uma resposta do servidor.

Simple Object Access Protocol (SOAP)

No final dos anos 1990, o SOAP (Simple Object Access Protocol, ou Protocolo Simples de Acesso a Objetos) surgiu na Microsoft. O SOAP usava XML para codificar uma mensagem e HTTP como transporte. Também usava um sistema de tipos e introduziu o conceito de chamadas orientadas a recursos para dados. O SOAP oferecia resultados razoavelmente previsíveis, mas causava frustração porque suas implementações eram muito complicadas.

REST

O paradigma de API com o qual você provavelmente tem mais familiaridade atualmente é o REST. O REST foi definido em 2000 na dissertação de doutorado de Roy Fielding (<http://bit.ly/2j4SIKI>), na Universidade da Califórnia, em Irvine. Ele descreveu uma arquitetura orientada a recursos, em que os usuários acessariam recursos web realizando operações como GET, PUT, POST e

DELETE. A rede de recursos pode ser pensada como uma *máquina de estados virtual*, e as ações (GET, PUT, POST, DELETE) são mudanças de estado na máquina. Podemos considerar que isso seja lugar-comum hoje em dia, mas essa foi uma mudança significativa. (Ah, e Fielding conseguiu seu título de doutorado.)

Em uma arquitetura RESTful, rotas representam informações. Por exemplo, requisitar informação de cada uma das rotas a seguir resultará em uma resposta específica:

```
/api/food/hot-dog  
/api/sport/skiing  
/api/city/Lisbon
```

O REST nos permite criar um modelo de dados com uma variedade de endpoints – uma abordagem muito mais simples que as arquiteturas anteriores. Ele possibilitou uma nova maneira de lidar com os dados em uma web cada vez mais complexa, sem impor um formato de resposta específico para os dados. No início, o REST era usado com XML. O AJAX originalmente era um acrônimo, que significava Asynchronous JavaScript And XML (JavaScript Assíncrono e XML), pois os dados de resposta de uma requisição Ajax eram formatados como XML (atualmente é uma palavra independente, e é grafada como “Ajax”). Isso gerou um passo complicado para os desenvolvedores web: a necessidade de fazer parse de respostas XML antes que os dados pudessem ser usados em JavaScript.

Logo depois, o JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript) foi desenvolvido e padronizado por Douglas Crockford. JSON é uma linguagem independente e define um formato de dados elegante, do qual muitas linguagens distintas podem fazer parse e consumir. Crockford prosseguiu escrevendo um livro de importância fundamental, *JavaScript: The Good Parts* (O’Reilly, 2008)⁵, no qual afirma que JSON é uma das partes boas.

Não há como negar a influência do REST. Ele é usado para implementar inúmeras APIs. Desenvolvedores ao longo de toda a pilha têm se beneficiado com ele. Há até mesmo devotos tão

interessados em argumentar sobre o que é e o que não é RESTful, a ponto de terem sido apelidados de *RESTafarianos*. Então, se essa era a situação, por que Byron, Schrock e Schafer embarcaram em sua jornada para criar algo novo? Podemos encontrar a resposta observando algumas das deficiências do REST.

Desvantagens do REST

Quando o GraphQL foi inicialmente lançado, alguns o aclamaram como um substituto do REST. “O REST está morto!”, gritaram alguns que adotaram inicialmente o GraphQL, e então incentivaram a todos que jogassem fora suas inocentes APIs REST. Isso foi ótimo para obter cliques em blogs e iniciar conversas em conferências, mas pintar o GraphQL como assassino do REST é uma simplificação exagerada. Uma interpretação mais cuidadosa consiste em dizer que, à medida que a web evoluiu, o REST mostrou sinais de esgotamento em certas condições. O GraphQL foi criado para reduzir esses sinais.

Overfetching

Suponha que estejamos construindo um aplicativo que use dados da versão REST da SWAPI. Em primeiro lugar, precisamos carregar alguns dados sobre o personagem número 1, Luke Skywalker⁶. Uma requisição GET pode ser feita para essa informação acessando <https://swapi.co/api/people/1/>. A resposta são estes dados JSON:

```
{
  "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/"
}

```

É uma resposta enorme. Ela está muito além das necessidades de dados de nossa aplicação. Precisamos somente de informações para nome, peso e altura:

```

{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77"
}

```

É um caso claro de *overfetching* (busca de dados em excesso) – estamos obtendo muitos dados de que não precisaremos. O cliente precisa de três pontos de dados, mas estamos obtendo de volta um objeto com 16 chaves e enviando informações inúteis pela rede.

Em uma aplicação GraphQL, como seria a aparência dessa requisição? Ainda queremos o nome, a altura e o peso de Luke Skywalker na Figura 1.3.



```
query {  
  person(personID:1) {  
    name  
    height  
    mass  
  }  
}  
  
{  
  "data": {  
    "person": {  
      "name": "Luke Skywalker",  
      "height": 172,  
      "mass": 77  
    }  
  }  
}
```

Figura 1.3 – Consulta para Luke Skywalker.

À esquerda, executamos nossa consulta GraphQL. Pedimos somente os campos que queremos. À direita, recebemos uma resposta JSON, dessa vez contendo somente os dados que solicitamos, e não os 13 campos extras que deveriam ser transmitidos de uma torre de celular até um telefone sem que houvesse motivo para isso. Pedimos os dados em determinado formato e recebemos esses dados de volta nesse formato. Nada mais, nada menos. É uma maneira mais declarativa e, provavelmente, produzirá uma resposta mais rápida, considerando que dados irrelevantes não estão sendo buscados.

Underfetching

Nosso gerente de projetos acabou de passar em nossa mesa e disse que quer acrescentar outro recurso no aplicativo Star Wars. Além do nome, do peso e da altura, temos agora que exibir uma lista com todos os nomes de filmes em que Luke Skywalker estiver. Depois de requisitar os dados de <https://swapi.co/api/people/1/>, ainda temos que fazer requisições adicionais para obter mais dados. Isso significa que fizemos um *underfetching* (busca com dados insuficientes).

Para obter o título de cada filme, devemos buscar dados de cada uma das rotas no array de filmes:

```
"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/"  
]
```

Obter esses dados exige uma requisição para Luke Skywalker (<https://swapi.co/api/people/1/>) e então mais cinco, para cada um dos filmes. Para cada filme recebemos outro objeto grande. Dessa vez, usamos somente um valor.

```
{  
  "title": "The Empire Strikes Back",  
  "episode_id": 5,  
  "opening_crawl": "...",  
  "director": "Irvin Kershner",  
  "producer": "Gary Kurtz, Rick McCallum",  
  "release_date": "1980-05-17",  
  "characters": [  
    "https://swapi.co/api/people/1/",  
    "https://swapi.co/api/people/2/",  
    "https://swapi.co/api/people/3/",  
    "https://swapi.co/api/people/4/",  
    "https://swapi.co/api/people/5/",  
    "https://swapi.co/api/people/10/",  
    "https://swapi.co/api/people/13/",  
    "https://swapi.co/api/people/14/",  
    "https://swapi.co/api/people/18/",  
    "https://swapi.co/api/people/20/",  
    "https://swapi.co/api/people/21/",  
    "https://swapi.co/api/people/22/",  
    "https://swapi.co/api/people/23/",  
    "https://swapi.co/api/people/24/",  
    "https://swapi.co/api/people/25/",  
    "https://swapi.co/api/people/26/"  
  ],  
  "planets": [  
    //... Long list of routes  
  ],  
  "starships": [  
    //... Long list of routes  
  ],  
  "vehicles": [  
    //... Long list of routes  
  ],  
}
```

```
"species": [  
  //... Long list of routes  
],  
"created": "2014-12-12T11:26:24.656000Z",  
"edited": "2017-04-19T10:57:29.544256Z",  
"url": "https://swapi.co/api/films/2/"  
}
```

Se quiséssemos listar as personagens que fazem parte desse filme, teríamos que fazer várias outras requisições. Nesse caso, teríamos que acessar mais 16 rotas e fazer outras 16 viagens de ida e volta para o cliente. Cada requisição HTTP usa recursos do cliente e faz um overfetching dos dados. O resultado é uma experiência de usuário com mais lentidão, e os usuários com velocidades de rede mais baixas ou com dispositivos mais lentos talvez nem sequer consigam visualizar o conteúdo.

A solução com GraphQL para um underfetching consiste em definir uma requisição aninhada e então requisitar todos os dados em uma só busca, como mostra a Figura 1.4.

```

1 query {
2   person(personID:1) {
3     name
4     height
5     mass
6     filmConnection {
7       films {
8         title
9       }
10    }
11  }
12 }

```

```

{
  "data": {
    "person": {
      "name": "Luke Skywalker",
      "height": 172,
      "mass": 77,
      "filmConnection": {
        "films": [
          {
            "title": "A New Hope"
          },
          {
            "title": "The Empire Strikes Back"
          },
          {
            "title": "Return of the Jedi"
          },
          {
            "title": "Revenge of the Sith"
          },
          {
            "title": "The Force Awakens"
          }
        ]
      }
    }
  }
}

```

Figura 1.4 – Conexão com filmes.

Nesse exemplo, recebemos somente os dados de que precisamos em uma requisição. Como sempre, o formato da consulta corresponde ao formato dos dados devolvidos.

Administrando endpoints REST

Outra reclamação comum sobre as APIs REST é a falta de flexibilidade. À medida que as necessidades do cliente mudam, em geral é necessário criar novos endpoints, e esses endpoints podem começar a se multiplicar rapidamente. Para parafrasear a Oprah, “Adquira uma rota! Adquira uma rota! Todo! Mundo! Adquira! Uma! Rota!”.

Com a API REST SWAPI era necessário fazer requisições para diversas rotas. Aplicações maiores em geral usam endpoints personalizados para minimizar requisições HTTP. Você poderá ver

endpoints como `/api/character-with-movie-title` comecem a se multiplicar. A velocidade de desenvolvimento pode ficar mais lenta, pois configurar novos endpoints muitas vezes significa que as equipes de frontend e de backend terão que fazer mais planejamento e se comunicar mais entre si.

Com o GraphQL, a arquitetura típica envolve um único endpoint. Esse endpoint único pode atuar como um gateway e articular várias fontes de dados, mas ainda facilitará a sua organização.

Nesta discussão sobre as desvantagens de REST, é importante observar que muitas empresas usam GraphQL e REST em conjunto. Configurar um endpoint GraphQL que busque dados de endpoints REST é um modo perfeitamente válido de usar o GraphQL. Pode ser uma ótima maneira de adotar o GraphQL gradualmente em sua empresa.

GraphQL no mundo real

O GraphQL é usado por uma variedade de empresas em suas aplicações, sites e APIs. Uma das empresas de maior destaque a adotar logo o GraphQL foi o GitHub. Sua API REST passou por três iterações, e a versão 4 de sua API pública usa o GraphQL. Conforme menciona no site (<https://developer.github.com/v4/>), o GitHub percebeu que “a capacidade de definir exatamente os dados que você quer – e somente os dados que você quer – é uma grande vantagem em relação aos endpoints da v3 da API REST”⁷.

Outras empresas como *The New York Times*, IBM, Twitter e Yelp acreditaram no GraphQL também, e podemos ver os desenvolvedores dessas equipes muitas vezes argumentando a favor das vantagens do GraphQL em conferências.

Há pelo menos três conferências dedicadas especificamente ao GraphQL: Summit, em San Francisco, GraphQL Finland, em Helsinque, e GraphQL Europe, em Berlim. A comunidade continua crescendo por meio de meetups locais e diversas conferências de

software.

Clientes GraphQL

Como já dissemos inúmeras vezes, o GraphQL é apenas uma especificação. Ela não se preocupa se você a está usando com React ou com Vue ou com JavaScript, ou até mesmo com um navegador. O GraphQL tem opiniões sobre alguns tópicos específicos, mas, exceto por isso, as decisões de arquitetura caberão a você. Isso levou ao surgimento de ferramentas para impor algumas opções além do que está na especificação. Entram em cena os clientes GraphQL.

Os clientes GraphQL surgiram para agilizar o fluxo de trabalho das equipes de desenvolvedores e melhorar a eficiência e o desempenho das aplicações. Eles cuidam de tarefas como requisições de rede, caching de dados e injeção de dados na interface de usuário. Há muitos clientes GraphQL, mas os líderes na área são Relay (<https://facebook.github.io/relay/>) e Apollo (<https://www.apollographql.com/>).

O Relay é um cliente do Facebook que funciona com React e React Native. Tem como finalidade servir de conexão entre os componentes do React e os dados buscados no servidor GraphQL. O Relay é usado pelo Facebook, pelo GitHub, pelo Twitch e outros.

O Apollo Client foi desenvolvido na Meteor Development Group, e é um esforço voltado à comunidade para a criação de ferramentas mais completas em torno do GraphQL. O Apollo Client oferece suporte para todas as plataformas principais de desenvolvimento de frontend e é independente de framework. A empresa também desenvolve ferramentas que dão assistência na criação de serviços GraphQL, na melhoria do desempenho de serviços de backend e em ferramentas para monitorar o desempenho das APIs GraphQL. Empresas incluindo Airbnb, CNBC, *The New York Times* e Ticketmaster usam o Apollo Client em ambiente de produção.

O ecossistema é amplo e continua a se modificar, mas a boa notícia

é que a especificação GraphQL é um padrão bem estável. Nos próximos capítulos discutiremos como escrever um esquema e criar um servidor GraphQL. Nesse processo há recursos de aprendizado para suporte à sua jornada, disponíveis no repositório do GitHub que acompanha este livro: <https://github.com/moonhighway/learning-graphql/>. Nesse local você encontrará links úteis, exemplos e todos os arquivos de projeto por capítulo.

Antes de explorar as táticas para trabalhar com o GraphQL, vamos conversar um pouco sobre a teoria de grafos e a rica história das ideias que se encontram no GraphQL.

-
- 1 N.T.: Feature phones são telefones celulares mais simples que os smartphones, com funcionalidades limitadas. Podem oferecer serviços simples de SMS, acesso limitado à internet ou ter recursos de multimídia básicos.
 - 2 Veja as bibliotecas de servidores GraphQL em <https://graphql.org/code/>.
 - 3 Veja a especificação do GraphQL, junho de 2018 (<http://facebook.github.io/graphql/June2018/#sec-Overview>).
 - 4 Veja “Data Fetching for React Applications” (Busca de dados para aplicações React), de Dan Schafer e Jing Chen, <https://www.youtube.com/watch?v=9sc8Pyc51uU>.
 - 5 N.T.: Edição brasileira: *O melhor do JavaScript* (Altabooks, 2008).
 - 6 Observe que os dados da SWAPI não incluem os filmes mais recentes de Star Wars (Guerra nas estrelas).
 - 7 N.T.: Tradução livre com base no original em inglês: “the ability to define precisely the data you want – and only the data you want – is a powerful advantage over the REST API v3 endpoints”.

CAPÍTULO 2

Teoria dos grafos

O alarme toca. Você pega o seu telefone. Ao desligar o alarme, você vê duas notificações. Quinze pessoas curtiram um tuíte que você escreveu na noite passada. Bom. Três pessoas retuitaram. Muito bom. Sua notoriedade momentânea no Twitter foi apresentada a você por meio de um grafo (conforme vemos na Figura 2.1).

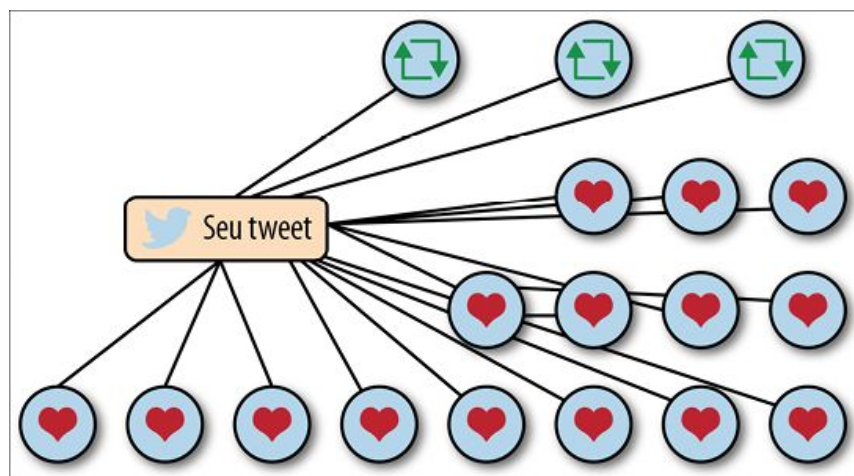


Figura 2.1 – Diagramas de curtidas e retuites.

Você sobe correndo as escadas para pegar o metrô do sistema “L”¹ em Irving Park. Você consegue entrar rapidamente, logo antes de as portas se fecharem. Perfeito. O trem se sacode por inteiro à medida que avança, conectando cada uma das estações.

As portas se abrem e se fecham em cada parada. Em primeiro lugar, Addison. Depois, Paulina, Southport e Belmont. Em Belmont, você cruza a plataforma para passar para a Linha Vermelha. Na Linha Vermelha, você faz mais duas paradas: Fullerton e North/Clybourn. O grafo, conforme vemos na Figura 2.2, mostra como você chegou ao trabalho.

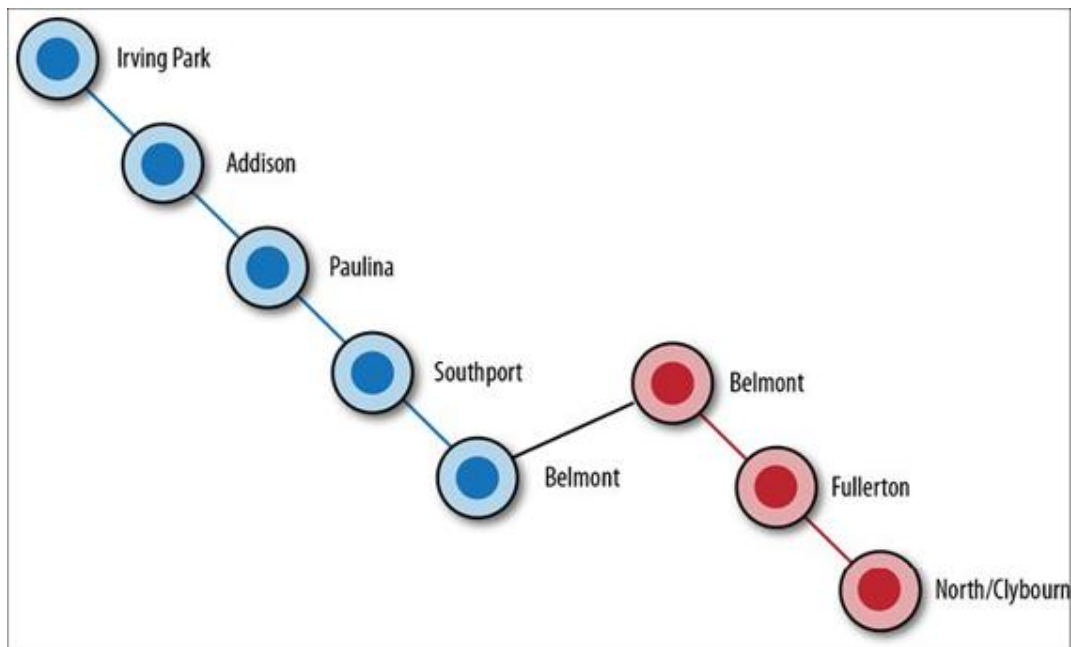


Figura 2.2 – Mapa do sistema “L” de Chicago.

Você está subindo a escada rolante até o nível da rua quando seu telefone toca. É a sua irmã. Ela diz que quer comprar passagens de trem para ir à festa de aniversário de 80 anos de seu avô, em julho. “É o pai da mamãe ou o pai do papai?”, você pergunta. “Do papai, mas acho que os pais da mamãe também estarão lá. E tia Linda, e tio Steve.” Você começa a imaginar quem estará lá. Outra festa planejada com outro grafo: uma árvore genealógica. A Figura 2.3 mostra esse grafo.

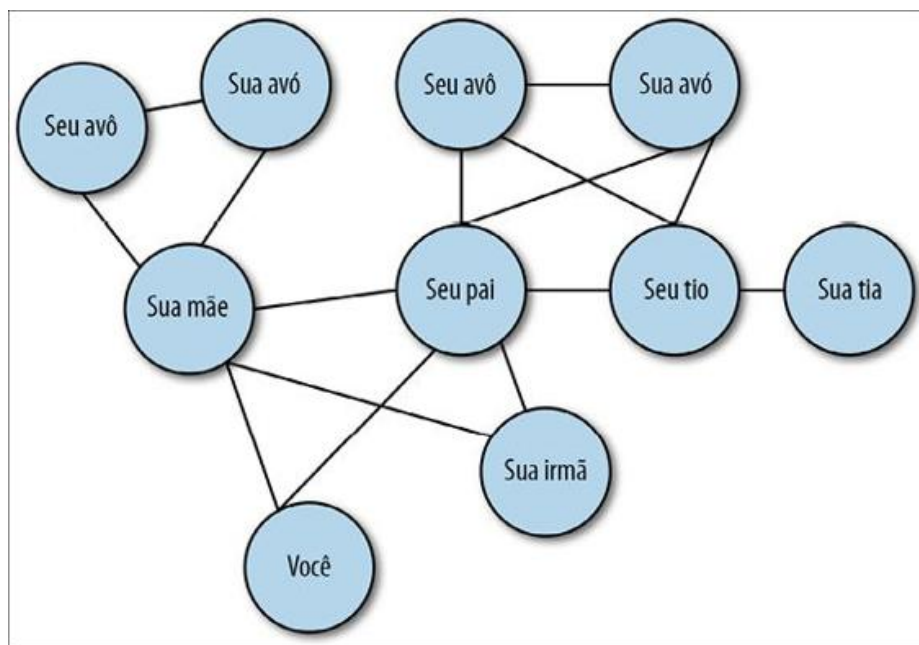


Figura 2.3 – Árvore genealógica.

Sem demora, você começa a perceber grafos por todos os lados. Você os vê em aplicações de redes sociais, mapas de rotas e nas árvores de telefones para alertas sobre feriado por causa de neve². E em constelações celestiais maravilhosas, como vemos na Figura 2.4.

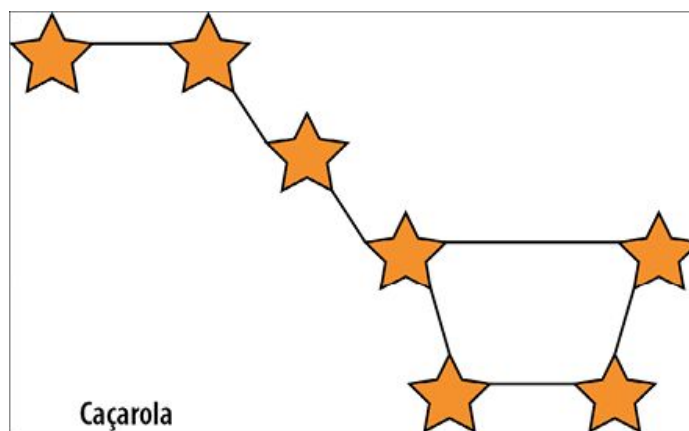


Figura 2.4 – Caçarola³.

Ou, ainda, no menor dos blocos de construção da natureza, como mostra a Figura 2.5.

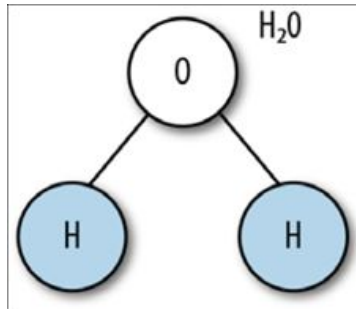


Figura 2.5 – Diagrama de H₂O.

Os grafos estão em toda parte, pois são uma ótima maneira de representar itens, pessoas, ideias ou dados interconectados. Porém, de onde veio o conceito de grafo? Para entender isso, podemos analisar mais de perto a *teoria dos grafos* e a sua origem na matemática.



Você não precisa saber nada sobre a teoria dos grafos para trabalhar de forma bem-sucedida com o GraphQL. Não haverá nenhuma prova. Achamos, porém, que é interessante explorar a história por trás desses conceitos a fim de termos um pouco mais de contexto.

Vocabulário da teoria dos grafos

A teoria dos grafos é o estudo dos grafos. Os grafos são usados formalmente para representar um conjunto de objetos interconectados. Podemos pensar em um grafo como um objeto contendo pontos de dados e suas conexões. Em ciência da computação, os grafos em geral descrevem redes de dados. Um grafo pode ter a aparência mostrada na Figura 2.6.

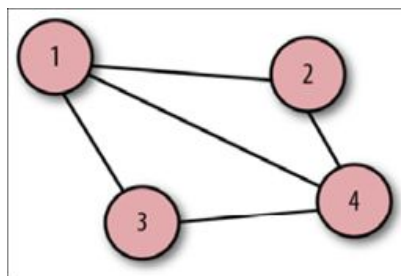


Figura 2.6 – Diagrama de grafo.

Esse diagrama de grafo é composto de quatro círculos que

representam os pontos de dados. Na terminologia de grafos, eles são chamados de *nós* ou *vértices*. As linhas ou conexões entre esses nós são chamadas de *arestas* (edges), e há cinco delas⁴.

Na forma de uma equação, um grafo é representado por $G = (V, E)$.

Começando pelas abreviaturas mais simples, G quer dizer grafo e V descreve o conjunto de vértices ou nós. Nesse grafo, V seria:

vertices = { 1, 2, 3, 4 }

E representa um conjunto de arestas (edges). As arestas são representadas por pares de nós.

edges = { {1, 2},
 {1, 3},
 {1, 4},
 {2, 4},
 {3, 4} }

Na lista de pares das arestas, o que aconteceria se reorganizássemos sua ordem? Por exemplo:

edges = { {4, 3},
 {4, 2},
 {4, 1},
 {3, 1},
 {2, 1} }

Nesse caso, o grafo permaneceria inalterado, como vemos na Figura 2.7.

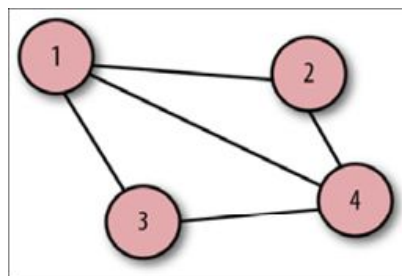


Figura 2.7 – Diagrama de grafo.

A equação continua representando o grafo, pois não há nenhuma direção ou hierarquia entre os nós. Na teoria dos grafos, denominamos isso de *grafo não direcionado* (<https://algs4.cs.princeton.edu/41graph/>). As definições das arestas,

ou as conexões entre os pontos de dados, são *pares não ordenados*.

Ao percorrer ou visitar diferentes nós desse grafo poderíamos começar e terminar em qualquer ponto, movendo-nos para qualquer direção. Os dados não seguem uma ordem numérica evidente, e um grafo não direcionado é, portanto, uma estrutura de dados não linear. Consideremos outro tipo de grafo, o *grafo direcionado*, que pode ser visto na Figura 2.8.

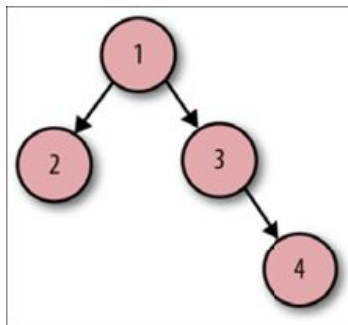


Figura 2.8 – Diagrama de um grafo direcionado.

O número de nós é o mesmo, porém as arestas têm uma aparência diferente. Em vez de linhas, são setas. Há uma direção ou fluxo entre os nós desse grafo. Para representar isso, usamos o seguinte:

```
vertices = {1, 2, 3, 4}
edges = ( {1, 2},
          {1, 3}
          {3, 4} )
```

Reunindo tudo, esta é a equação de nosso grafo:

```
graph = ( {1, 2, 3, 4},
          ({1, 2}, {1, 3}, {3, 4}) )
```

Observe que os pares estão entre parênteses em vez de chaves. Os parênteses significam que as definições dessas arestas são *pares ordenados*. Sempre que as arestas forem pares ordenados, teremos um grafo direcionado, ou um *digrafo*. O que aconteceria se reorganizássemos esses pares ordenados? Nosso diagrama teria a mesma aparência, como ocorreu no caso do grafo não direcionado?

```
graph = ( {1, 2, 3, 4},
          ( {4, 3}, {3, 1}, {1, 2} ) )
```

O diagrama resultante seria um tanto quanto diferente, com o nó 4 agora sendo a raiz, conforme mostra a Figura 2.9.

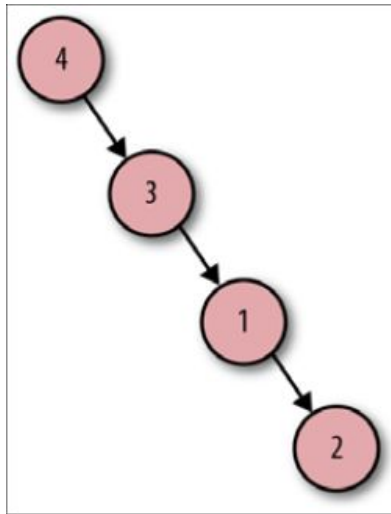


Figura 2.9 – Diagrama de um grafo direcionado.

Para percorrer o grafo, seria necessário começar pelo nó 4 e visitar cada nó do grafo seguindo as setas. Para ajudar a visualizar o percurso, talvez seja conveniente representar fisicamente a travessia de um nó para outro. Com efeito, foram com percursos físicos que esses conceitos da teoria dos grafos surgiram.

História da teoria dos grafos

Podemos traçar o estudo da teoria dos grafos de volta à cidade de Königsberg, na Prússia (<http://bit.ly/2AQhU47>), em 1735. Situada no Rio Pregel, a cidade era um centro de navegação, com duas grandes ilhas conectadas por sete pontes que ligavam quatro áreas principais de terra firme, conforme mostra a Figura 2.10.

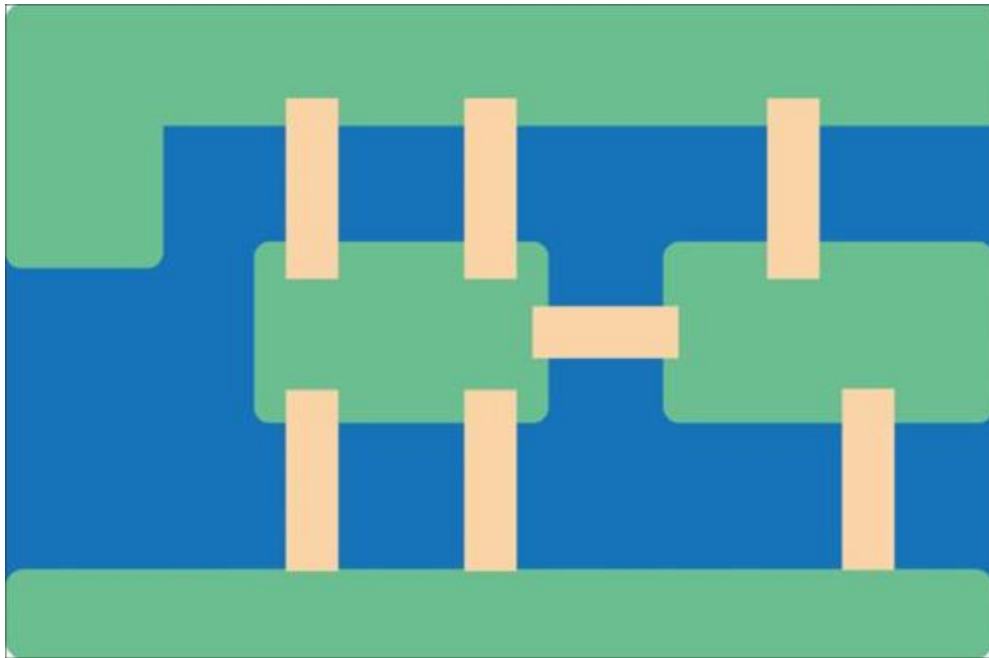


Figura 2.10 – As pontes de Königsberg.

Königsberg era uma cidade linda, e seus habitantes gostavam de passar seus domingos tomando ar fresco e caminhando pelas pontes. Com o tempo, os moradores da cidade se tornaram obcecados em tentar resolver um quebra-cabeça: Como poderiam cruzar cada uma das sete pontes uma vez sem jamais cruzarem de volta uma mesma ponte? Eles caminhavam pela cidade tentando visitar cada ilha e cruzando todas as pontes sem repeti-las, mas se viam sem solução. Esperando ter um pouco de ajuda para o problema, chamaram Leonhard Euler. Euler era um matemático suíço prolífico, que publicou mais de 500 livros e artigos durante sua vida.

Ocupado por ser um gênio, Euler não se preocupou com o que lhe parecia ser um problema trivial. Porém, depois de pensar um pouco mais sobre o assunto, Euler ficou tão interessado quanto os habitantes da cidade e tentou, com afinho, encontrar uma solução. Em vez tomar nota de todos os caminhos possíveis, Euler decidiu que seria mais simples observar as ligações (pontes) entre as porções de terra firme, como mostra a Figura 2.11.

Então ele fez uma simplificação, desenhando as pontes e as

porções de terra firme de um modo que passou a ser conhecido como diagrama de grafo. A aparência era semelhante à da Figura 2.12.

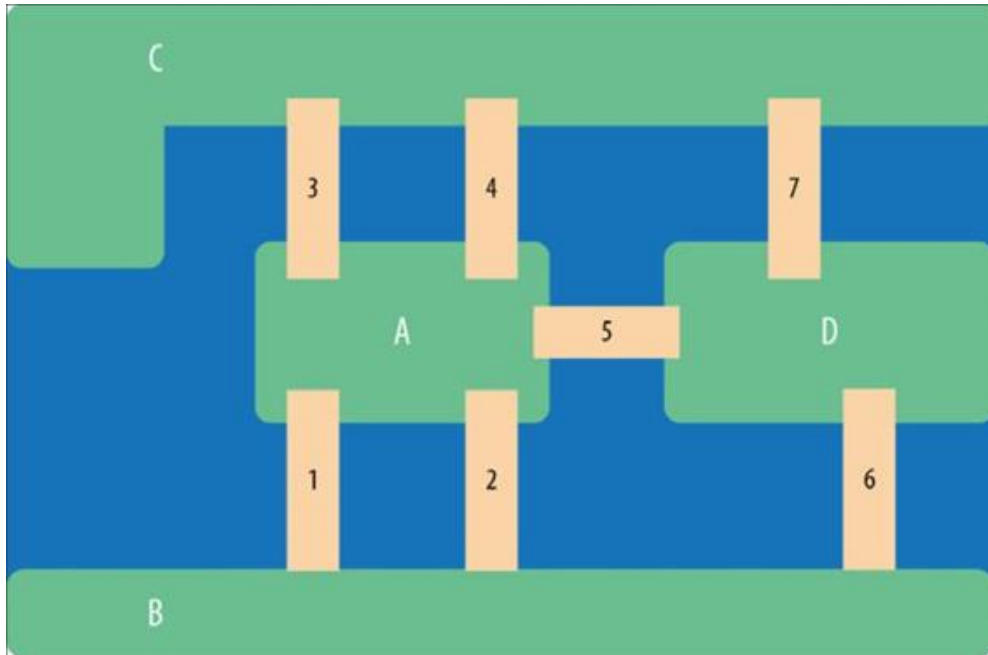


Figura 2.11 – As pontes de Königsberg enumeradas.

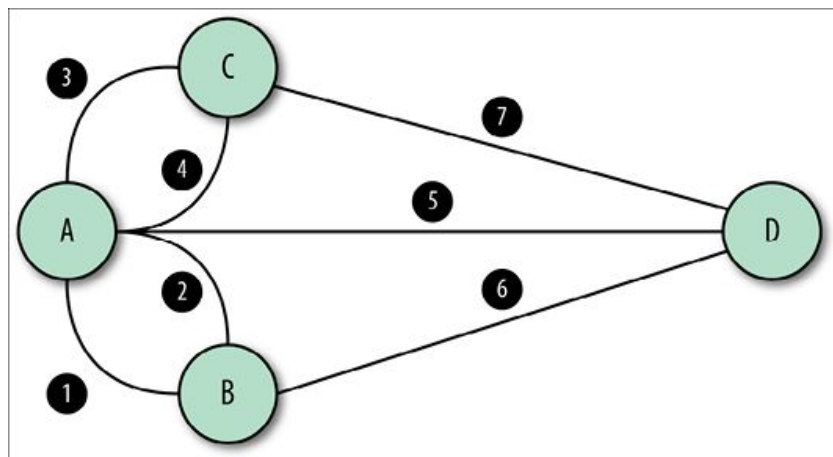


Figura 2.12 – As pontes de Königsberg na forma de um diagrama.

Na Figura 2.12, A e B são *adjacentes* porque estão conectados por uma aresta. Usando essas conexões das arestas, podemos calcular o *grau* de cada nó. O grau de um nó é igual ao número de arestas associadas a esse nó. Se observarmos os nós do Problema das Pontes, veremos que todos os graus são ímpares.

- A: cinco arestas para nós adjacentes (ímpar)
- B: três arestas para nós adjacentes (ímpar)
- C: três arestas para nós adjacentes (ímpar)
- D: três arestas para nós adjacentes (ímpar)

Pelo fato de cada um dos nós ter graus ímpares, Euler percebeu que cruzar todas as pontes sem retornar era impossível. Resumindo uma longa história: se você tomar uma ponte para chegar a uma ilha, deverá sair usando uma ponte diferente. O número de arestas ou pontes deve ser par se você não quiser passar por uma mesma ponte novamente.

Atualmente chamamos um grafo no qual cada aresta é visitada uma só vez de *caminho euleriano* (ou caminho de Euler). Para caracterizar, o grafo não direcionado terá dois vértices com grau ímpar ou todos os vértices terão um grau par. No exemplo, temos dois vértices com grau ímpar (1, 4), como mostra a Figura 2.13.

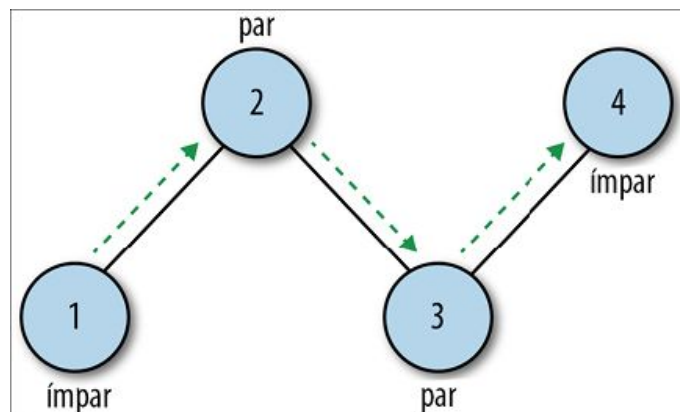


Figura 2.13 – Um caminho euleriano.

Outra ideia associada a Euler é a de um circuito ou *ciclo euleriano*. Nesse caso, o nó de início é igual ao nó de fim. Cada aresta é visitada somente uma vez, mas o nó de início e de fim se repetem (Figura 2.14).

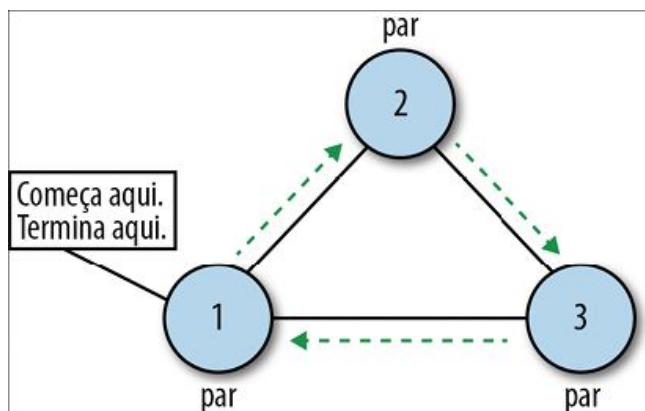


Figura 2.14 – Um ciclo euleriano.

O Problema das Pontes de Königsberg se tornou o primeiro teorema da teoria dos grafos. Além de ser considerado o criador da teoria dos grafos, Euler é conhecido por criar a constante e e a *unidade imaginária* i . Até mesmo a sintaxe de função matemática $f(x)$ – uma função f aplicada à variável x – pode ter sua origem traçada de volta a Leonhard Euler⁵.

O Problema das Pontes de Königsberg definia que uma ponte não poderia ser cruzada mais de uma vez. Não havia nenhuma regra dizendo que o percurso deveria começar ou terminar em um nó específico. Isso significa que tentar resolver o problema era um exercício de travessia em um grafo não direcionado. E se quiséssemos tentar solucionar o problema das pontes, mas tivéssemos que começar em um nó em particular?

Se você morasse na ilha B, esse seria o local em que sempre teria que iniciar o seu percurso. Nesse caso, você estaria lidando com um grafo direcionado, mais comumente chamado de árvore.

Árvores e grafos

Vamos considerar outro tipo de grafo: uma árvore. Uma árvore é um grafo no qual os nós estão organizados hierarquicamente. Você sabe que estará olhando para uma árvore se houver um nó-raiz. Em outras palavras, a raiz é o ponto inicial da árvore, e então todos os demais nós estarão ligados à raiz como filhos.

Considere um organograma. Essa é uma árvore didática. O CEO está no topo e todos os demais funcionários estão abaixo do CEO. O CEO é a raiz da árvore e todos os outros nós são filhos do nó-raiz, como mostra a Figura 2.15.

As árvores têm vários usos. Você pode usar uma para representar a genealogia de uma família. As árvores podem mapear algoritmos de tomada de decisão. Elas ajudam a acessar informações em bancos de dados de modo rápido e eficiente. Um dia, você poderia até mesmo ter que inverter uma árvore binária em um quadro branco para as cinco pessoas que estiverem entre você e o seu novo emprego, e talvez jamais tenha que fazer isso novamente.

Podemos determinar se um grafo é uma árvore com base no fato de ela ter um nó-raiz ou um nó de início. A partir do nó-raiz, uma árvore está conectada aos nós-filhos por meio de arestas. Se um nó estiver conectado com um nó-filho, esse nó é chamado de pai. Quando um filho tiver filhos, esse nó será chamado de galho (branch). Se um nó não tiver filhos, ele é chamado de folha (leaf).

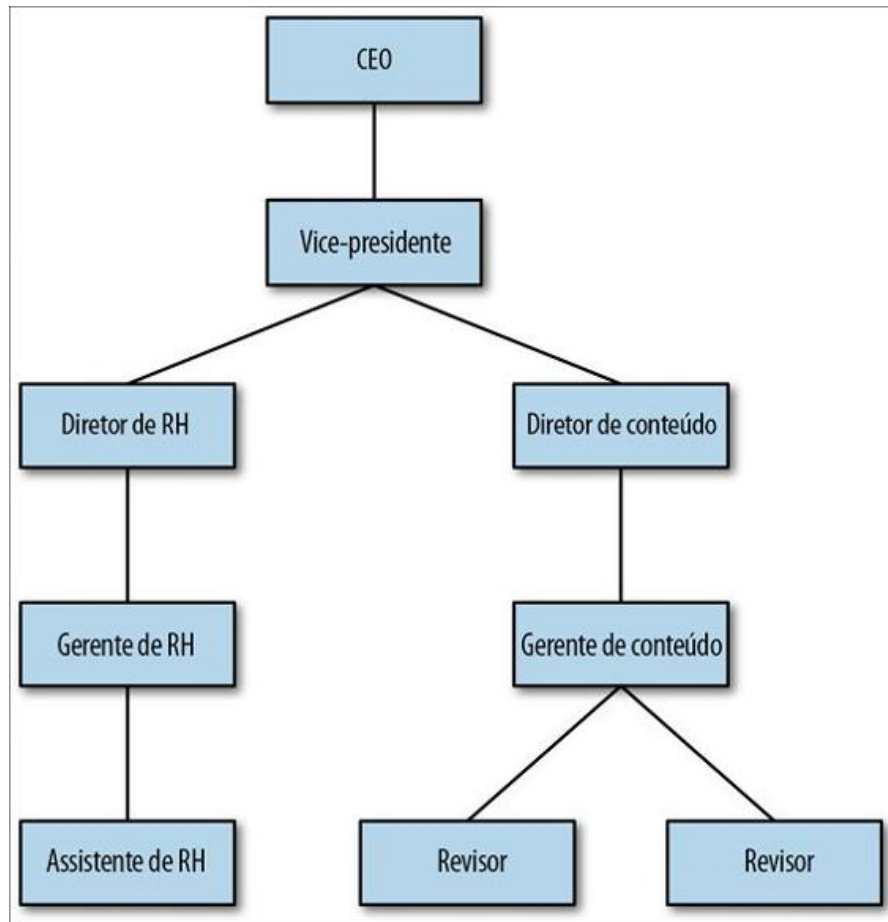


Figura 2.15 – Um organograma.

Os nós contêm pontos de dados. Por esse motivo, é importante compreender onde estão os dados na árvore para que possam ser rapidamente acessados. Para encontrar rapidamente os dados, devemos calcular a *profundidade* dos nós individuais. A profundidade de um nó simplesmente se refere à distância que o nó se encontra da raiz de uma árvore. Consideremos a árvore $A \rightarrow B \rightarrow C \rightarrow D$. Para descobrir qual é a profundidade do nó C, conte as ligações entre C e a raiz. Há exatamente duas ligações entre C e a raiz (A), portanto a profundidade do nó C é 2, e do nó D é 3.

A estrutura hierárquica de uma árvore implica que, com frequência, as árvores incluem outras árvores. Uma árvore aninhada em outra árvore é chamada de subárvore. Uma página HTML em geral tem várias subárvores. A raiz da árvore é a tag `<html>`. Há, então, duas subárvores com `<head>` na raiz da subárvore à esquerda e `<body>` na

raiz da subárvore à direita. A partir daí, <header>, <footer> e <div> são todos raízes de subárvores distintas. Se houver muitos itens aninhados, haverá muitas subárvores, como mostra a Figura 2.16.

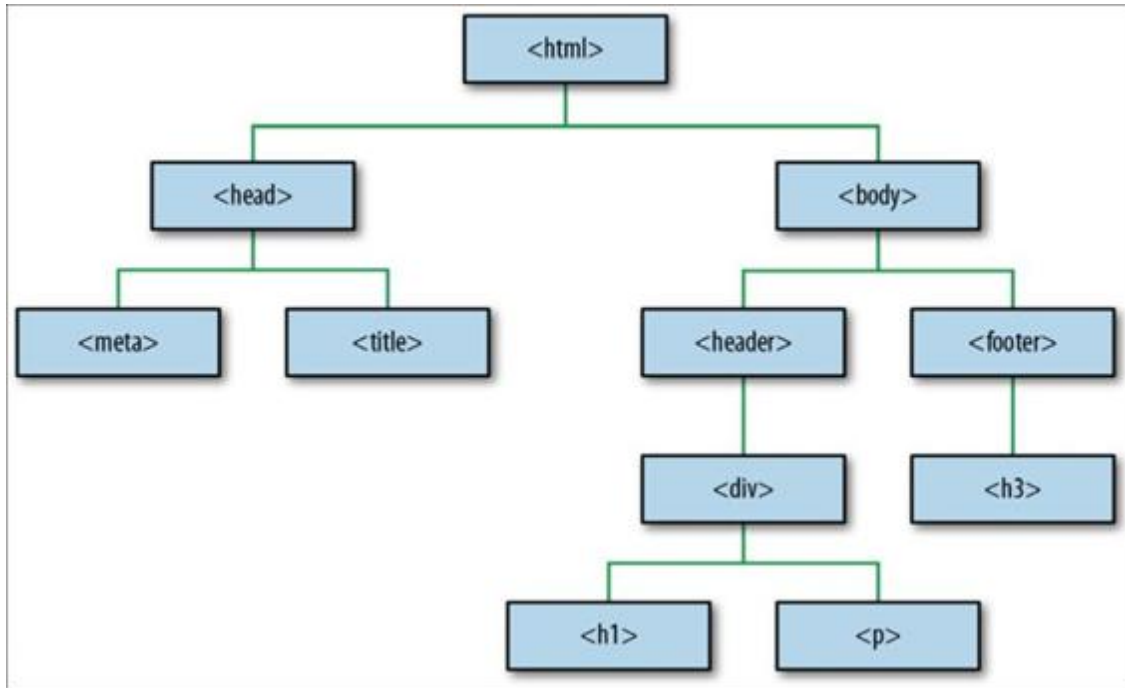


Figura 2.16 – Uma árvore HTML.

Assim como uma árvore é um tipo específico de grafo, uma *árvore binária* é um tipo específico de árvore. Uma árvore binária implica que cada nó não terá mais que dois nós-filhos. Ao falar sobre árvores binárias, com frequência estaremos nos referindo a *árvores binárias de busca*⁶. Uma árvore binária de busca é uma árvore binária que segue regras específicas de ordenação. As regras de ordenação e a estrutura da árvore nos ajudam a encontrar rapidamente os dados de que precisamos. A Figura 2.17 mostra um exemplo de uma árvore binária de busca.

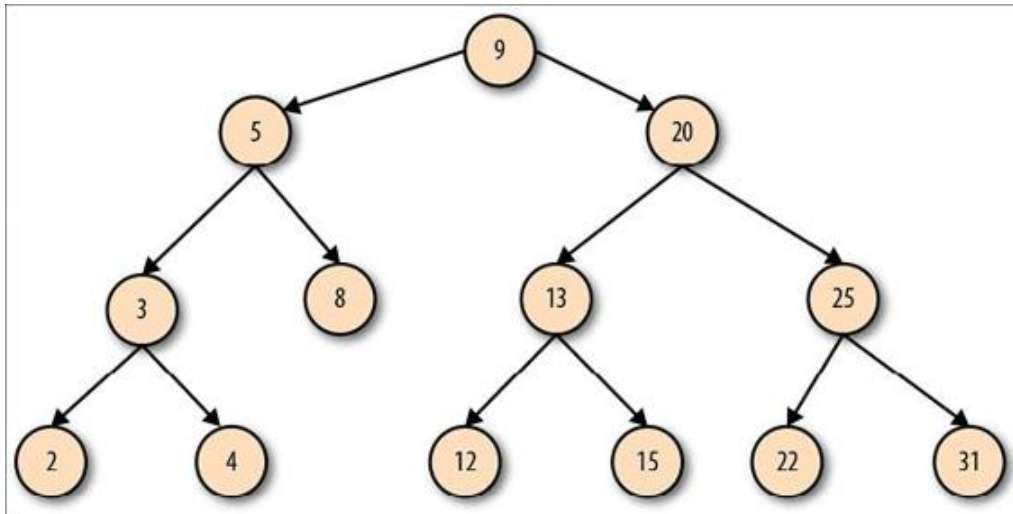


Figura 2.17 – Árvore binária de busca.

A árvore tem um nó-raiz e segue a regra segundo a qual cada nó não deve ter mais que dois nós-filhos. Suponha que quiséssemos encontrar o nó 15. Sem uma árvore binária de busca, teríamos que visitar cada um dos nós até encontrar o nó 15. Talvez tivéssemos sorte e desceríamos pelo galho correto. Mas, quem sabe, não seríamos tão afortunados e teríamos que retroceder de forma ineficaz pela árvore.

Com a árvore binária de busca podemos localizar o nó 15 habilmente se compreendermos as regras dos lados esquerdo e direito. Se iniciarmos o percurso na raiz (9), nós nos perguntaríamos: “Quinze é maior ou igual a 9?” Se for menor, nos movemos para a esquerda. Se for maior, iremos para a direita. Quinze é maior que 9, portanto vamos para a direita e, ao fazermos isso, excluimos metade dos nós da árvore de nossa busca. Nesse ponto, temos o nó 20. Quinze é maior ou menor que 20? É menor, portanto nos movemos para a esquerda, eliminando metade dos nós restantes. Agora, no nó 13, 15 é maior ou menor que 13? É maior, portanto vamos para a direita. Encontramos! Ao usar a esquerda e a direita para eliminar opções, podemos encontrar muito mais rapidamente o dado no qual estamos interessados.

Grafos no mundo real

Você poderia se deparar com esses conceitos da teoria dos grafos todos os dias, dependendo do trabalho que fizer com o GraphQL. Ou poderia simplesmente usar o GraphQL como uma forma eficiente de carregar dados nas interfaces do usuário. Independentemente da situação, todas essas ideias estarão na base dos projetos com GraphQL. Como já vimos, os grafos são particularmente apropriados para lidar com os requisitos de aplicações com muitos pontos de dados.

Pense no Facebook. De posse de nosso vocabulário da teoria dos grafos, sabemos que cada pessoa no Facebook é um nó. Quando uma pessoa está conectada a outra, há uma conexão bidirecional por meio de uma aresta. O Facebook é um grafo não direcionado. Sempre que eu me conectar com alguém no Facebook, essa pessoa estará conectada comigo. Minha conexão com minha melhor amiga Sarah é uma conexão bidirecional. Somos amigas uma da outra (como mostra a Figura 2.18).

Por ser um grafo não direcionado, cada nó do grafo do Facebook faz parte de uma rede de vários relacionamentos interconectados – uma rede social. Você está conectado com todos os seus amigos. No mesmo grafo, esses amigos estão conectados com todos os amigos deles. Um percurso pode começar e terminar em qualquer nó (Figura 2.19).

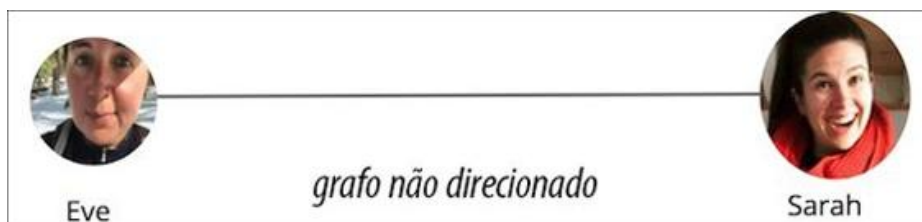


Figura 2.18 – Grafo não direcionado do Facebook.



Figura 2.19 – Rede não direcionada do Facebook.

Como alternativa, temos o Twitter. De modo diferente do Facebook, em que todos estão pareados por meio de uma conexão bidirecional, o Twitter é um grafo direcionado, pois cada conexão é unidirecional, como mostra a Figura 2.20. Se você segue Michelle Obama, ela pode não seguir você de volta, apesar de ela ser sempre muito bem-vinda para fazê-lo (@eveporcello (<https://twitter.com/eveporcello>), @moontahoe (<https://twitter.com/moontahoe>)).

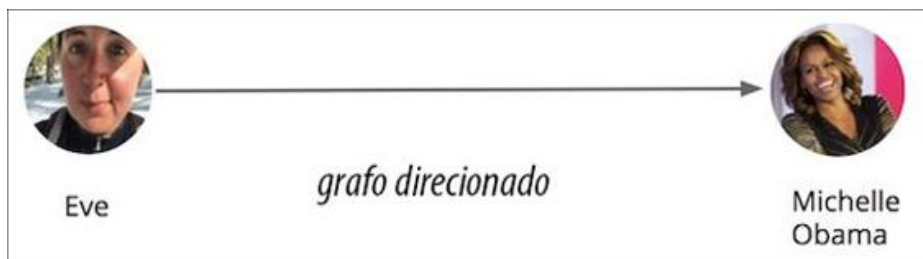


Figura 2.20 – Grafo do Twitter.

Se uma pessoa observar todas as suas amizades, ela se tornará a raiz de uma árvore. Ela está conectada com seus amigos. Seus amigos, então, estão conectados com os amigos deles em subárvores (Figura 2.21).

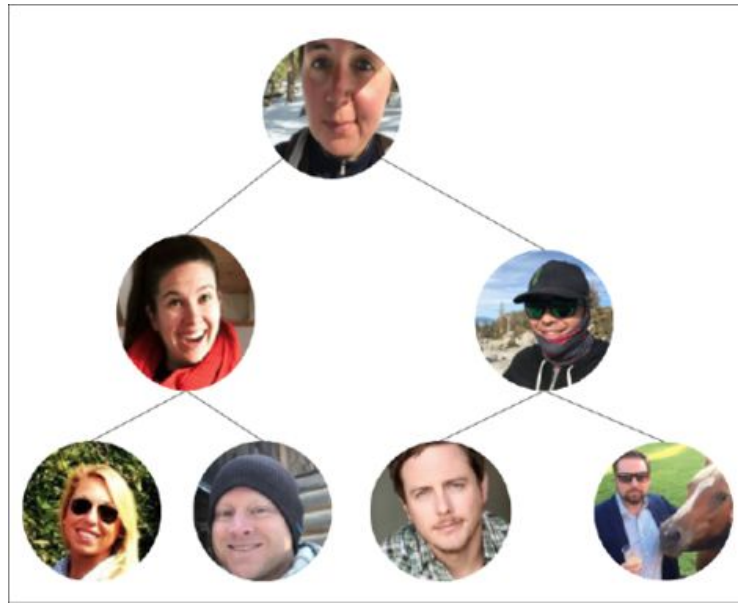


Figura 2.21 – Árvore de amigos.

O mesmo vale para qualquer outra pessoa no grafo do Facebook. Assim que você isolar uma pessoa e pedir seus dados, a requisição terá o aspecto de uma árvore. A pessoa estará na raiz e todos os dados que você quiser sobre essa pessoa estarão em um nó-filho. Nessa requisição, uma pessoa estará conectada a todos os seus amigos por meio de uma aresta:

- pessoa
 - nome
 - local
 - data de nascimento
 - amigos
 - nome do(a) amigo(a)
 - local do(a) amigo(a)
 - data de nascimento do(a) amigo(a)

Essa estrutura é muito parecida com uma consulta GraphQL:

```
{
  me {
    name
    location
    birthday
  }
}
```



```
    friends {  
      name  
      location  
      birthday  
    }  
  }  
}
```

Com o GraphQL, nossa meta é simplificar grafos complexos de dados realizando consultas para os dados de que precisamos. No próximo capítulo, exploraremos melhor o funcionamento de uma consulta GraphQL e como ela é validada em relação a um sistema de tipos.

-
- 1 N.T.: O sistema “L” é o nome como é conhecido o sistema de trens e metrô de Chicago nos Estados Unidos.
 - 2 N.T.: Uma árvore de telefones (*phone tree*) é um sistema tradicional para envio de mensagens ou alertas a um grupo de assinantes. Cada um que receber uma chamada deve reenviar a mensagem a outros números, que farão o mesmo. Em dias de muita neve nos Estados Unidos, alguns estabelecimentos, por exemplo, as escolas, podem ser fechados (*snow day*), e as árvores de telefone eram usadas para repassar rapidamente o aviso a todos os membros da comunidade.
 - 3 N.T.: Caçarola é um conjunto de sete estrelas brilhantes que faz parte da constelação da Ursa Maior.
 - 4 Para leituras complementares sobre nós e arestas, consulte a postagem de blog de Vaidehi Joshi, “A Gentle Introduction to Graph Theory” (Uma breve introdução à teoria dos grafos, <https://dev.to/vaidehijoshi/a-gentle-introduction-to-graph-theory>).
 - 5 Mais informações sobre Euler e seu trabalho podem ser encontradas em http://www.storyofmathematics.com/18th_euler.html.
 - 6 Veja a postagem de blog de Vaidehi Joshi, “Leaf It Up to Binary Trees” (Deixa por conta das árvores binárias, em <http://bit.ly/2vQyKd5>).

CAPÍTULO 3

A linguagem de consulta GraphQL

Quarenta e cinco anos antes de o código aberto do GraphQL ser lançado, um funcionário da IBM, Edgar M. Codd, publicou um artigo relativamente conciso com um título bem longo. “A Relational Model of Data for Large Shared Databanks” (Um modelo de dados relacional para bancos de dados grandes e compartilhados, <http://bit.ly/2Ms7jxn>) não possuía um título sintético, mas continha algumas ideias sólidas. O artigo descrevia um modelo para armazenar e manipular dados usando tabelas. Logo depois disso, a IBM começou a trabalhar em um banco de dados relacional que podia ser consultado usando a *SEQUEL* (Structured English Query Language, ou Linguagem de Consulta Estruturada em Inglês), que mais tarde se tornou conhecida somente como *SQL*.

A *SQL* (Structured Query Language, ou Linguagem de Consulta Estruturada) é uma linguagem específica de domínio, usada para acessar, administrar e manipular dados em um banco de dados. A *SQL* introduziu a ideia de acessar vários registros com um único comando. Também tornou possível acessar qualquer registro usando qualquer chave, e não somente um ID.

Os comandos que podiam ser executados com a *SQL* eram muito simples: *SELECT*, *INSERT*, *UPDATE* e *DELETE*. É tudo que podemos fazer com os dados. Com a *SQL*, podemos escrever uma única consulta que devolva dados conectados em várias tabelas de dados em um banco de dados.

Essa ideia – a de que os dados só podem ser lidos, criados,

atualizados ou apagados – chegou até o REST (Representational State Transfer, ou Transferência de Estado Representativo), que exige que usemos métodos HTTP distintos, de acordo com estas quatro operações básicas nos dados: GET, POST, PUT e DELETE. No entanto, o único modo de especificar o tipo do dado que queremos ler ou modificar com o REST é por meio das URLs de endpoints, e não por uma linguagem de consulta.

O GraphQL usa as ideias que foram originalmente desenvolvidas para consultar bancos de dados e as aplica à internet. Uma única consulta GraphQL é capaz de devolver dados conectados. Assim como no caso da SQL, podemos usar consultas GraphQL para alterar ou remover dados. Afinal de contas, o QL em SQL e em GraphQL representam o mesmo termo: Query Language (Linguagem de Consulta).

Apesar de ambas serem linguagens de consulta, o GraphQL e a SQL são completamente diferentes. Elas estão voltadas para ambientes totalmente distintos. As consultas SQL são enviadas a um banco de dados. As consultas GraphQL vão para uma API. Dados para SQL são armazenados em tabelas de dados. Dados para GraphQL podem ser armazenados em qualquer lugar: um banco de dados, vários bancos de dados, sistemas de arquivo, APIs REST, WebSockets e até mesmo outras APIs GraphQL. A SQL é uma linguagem de consulta para bancos de dados. O GraphQL é uma linguagem de consulta para a internet.

GraphQL e SQL também têm sintaxes totalmente distintas. Em vez de SELECT, o GraphQL usa Query para requisitar dados. Essa operação está no coração de tudo que fazemos com o GraphQL. Em vez de INSERT, UPDATE ou DELETE, o GraphQL encapsula todas essas alterações de dados em um só tipo de dado: Mutation. Como o GraphQL foi desenvolvido para a internet, ele inclui um tipo Subscription que pode ser usado para saber se houve modificações nos dados utilizando conexões socket. A SQL não tem nada que se assemelhe a uma subscription (inscrição ou assinatura). Ela é como

um avô que não se parece em nada com seu neto, mas sabemos que são parentes porque têm o mesmo sobrenome.

O GraphQL é padronizado de acordo com a sua especificação. Não importa qual linguagem você esteja usando: uma consulta GraphQL é uma consulta GraphQL. A sintaxe da consulta é uma string que tem a mesma aparência, independentemente de o projeto usar JavaScript, Java, Haskell ou outra linguagem.

As consultas são strings simples, enviadas no corpo de requisições POST para um endpoint GraphQL. A seguir, apresentamos uma consulta GraphQL – uma string escrita na linguagem de consulta GraphQL:

```
{
  allLifts {
    name
  }
}
```

Você poderia enviar essa consulta para um endpoint GraphQL usando *curl*:

```
curl 'http://snowtooth.herokuapp.com/'
-H 'Content-Type: application/json'
--data '{"query":"{ allLifts {name } }"}
```

Supondo que o esquema do GraphQL aceite uma consulta nesse formato, você receberá uma resposta JSON diretamente no terminal. Essa resposta JSON conterá os dados que você solicitou em um campo chamado `data`, ou o campo `errors`, se algo der errado. Fazemos uma requisição. Recebemos uma resposta.

Para modificar dados, podemos enviar *mutações* (mutations). As mutações se parecem bastante com as consultas, mas seu propósito é alterar algo sobre o estado de uma aplicação em geral. Os dados necessários para fazer uma alteração podem ser enviados diretamente com a mutação, como vemos a seguir:

```
mutation {
  setLiftStatus(id: "panorama" status: OPEN) {
    name
    status
  }
}
```

```
}  
}
```

A mutação anterior está escrita na linguagem de consulta GraphQL, e podemos supor que ela modificará o status de um teleférico (lift) cujo id é igual a `panorama` para `OPEN`. Novamente, podemos enviar essa operação a um servidor GraphQL usando `cURL`:

```
curl 'http://snowtooth.herokuapp.com/'  
-H 'Content-Type: application/json'  
--data '{"query":"mutation {setLiftStatus(id: \"panorama\" status: OPEN) {name status}}}'
```

Há maneiras mais elegantes de mapear variáveis para uma consulta ou uma mutação, mas apresentaremos esses detalhes mais adiante no livro. Neste capítulo, nosso foco será como compor consultas, mutações e subscriptions usando o GraphQL.

Ferramentas para APIs GraphQL

A comunidade do GraphQL criou várias ferramentas de código aberto que podem ser usadas para interagir com APIs GraphQL. Essas ferramentas permitem escrever consultas na linguagem GraphQL, enviá-las aos endpoints GraphQL e inspecionar a resposta JSON. Na próxima seção veremos duas das ferramentas mais populares para testar consultas a uma API GraphQL: GraphiQL e GraphQL Playground.

GraphiQL

O GraphiQL é o IDE (Integrated Development Environment, ou Ambiente de Desenvolvimento Integrado) para navegador, criado pelo Facebook para permitir consultar e explorar uma API GraphQL. O GraphiQL oferece destaque de sintaxe, preenchimento automático de código e avisos sobre erros, além de possibilitar a execução e a visualização dos resultados de consultas diretamente no navegador. Muitas APIs públicas oferecem uma interface GraphiQL com a qual você poderá consultar dados ativos.

A interface é relativamente simples. Há um painel no qual você pode

escrever sua consulta, um botão para executá-la e um painel para exibir a resposta, como vemos na Figura 3.1.

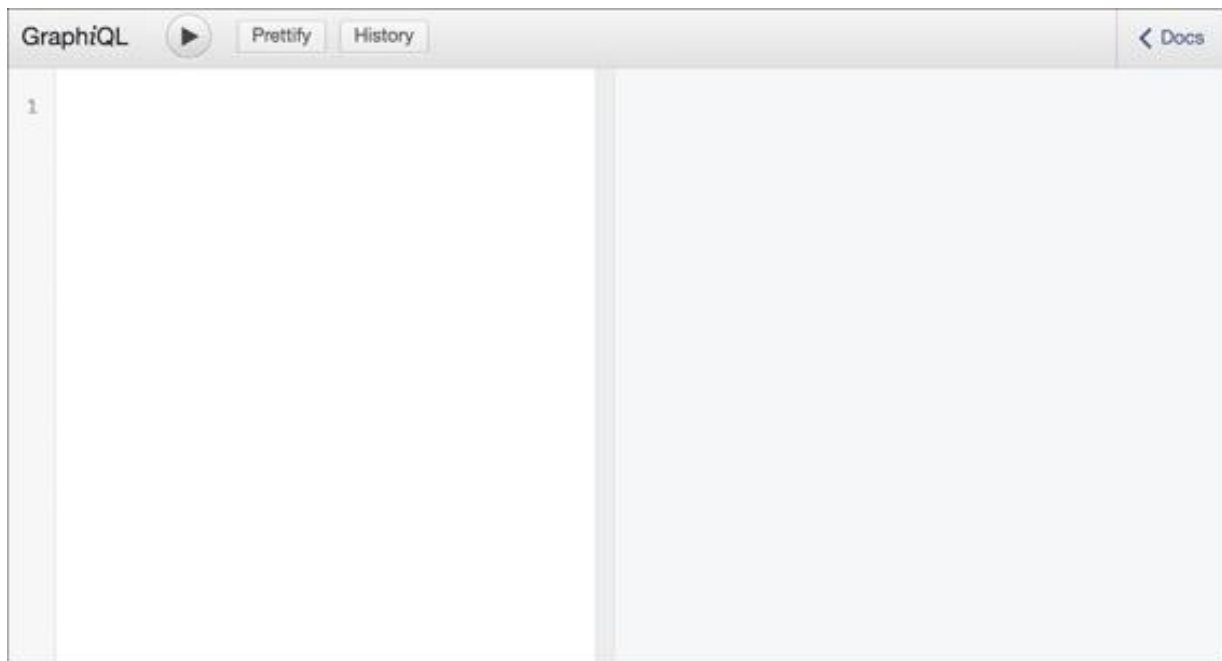


Figura 3.1 – A interface do GraphQL.

Nossas consultas começam como um texto escrito na Linguagem de Consulta GraphQL. Referimo-nos a esse texto como *documento de consulta* (query document). Esse texto deve ser inserido no painel à esquerda. Um documento GraphQL pode conter as definições de uma ou mais *operações*. Uma operação é uma Query, uma Mutation ou uma Subscription. A Figura 3.2 mostra como adicionaríamos uma operação Query em nosso documento.

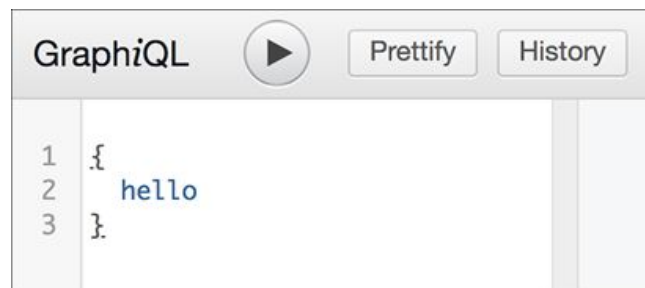


Figura 3.2 – Uma consulta no GraphQL.

Clicar no botão Play (Executar) fará a consulta ser executada. Então, no painel à direita, você receberá uma resposta formatada

como JSON (Figura 3.3).

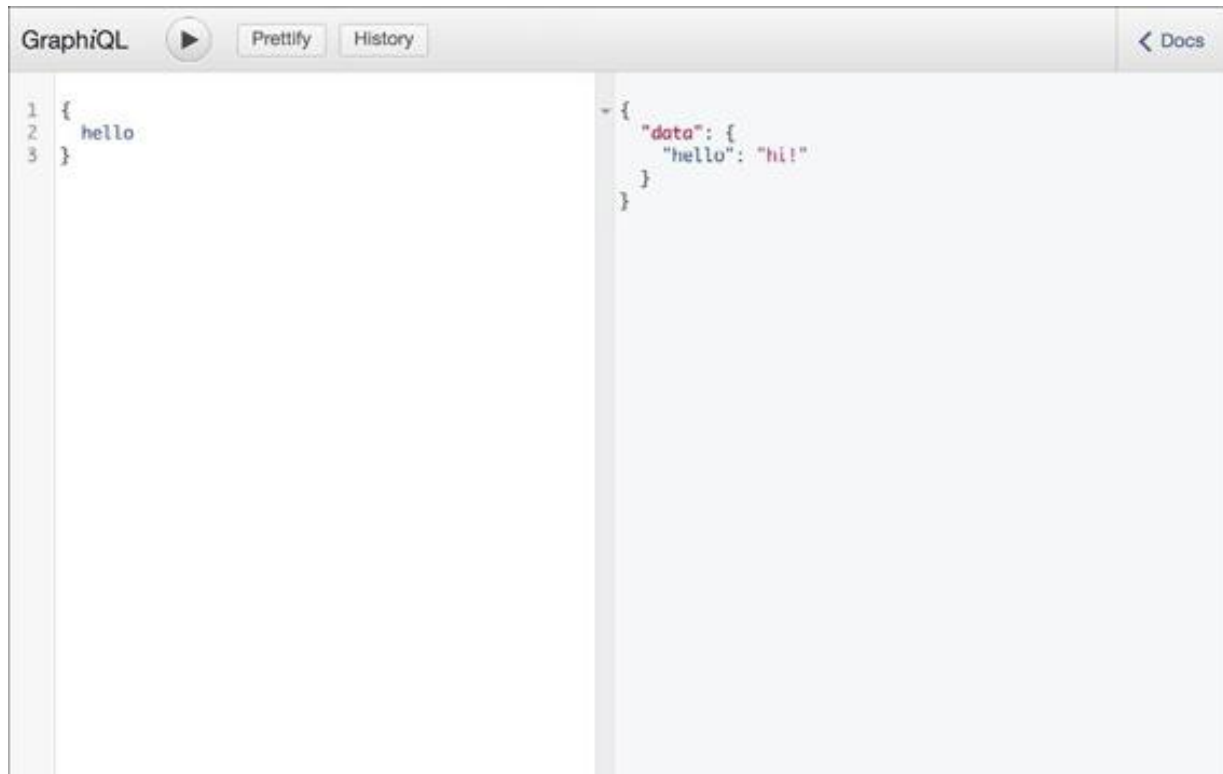


Figura 3.3 – GraphQL.

Você pode clicar no canto superior direito para abrir a janela Docs, que define tudo que precisamos saber para interagir com o serviço atual. Essa documentação é automaticamente adicionada no GraphQL porque é lida do esquema do serviço. O esquema define os dados disponibilizados pelo serviço, e o GraphQL monta automaticamente a documentação executando uma consulta de introspecção no esquema. É sempre possível consultar essa documentação no Documentation Explorer (Explorador de documentação), como vemos na Figura 3.4.

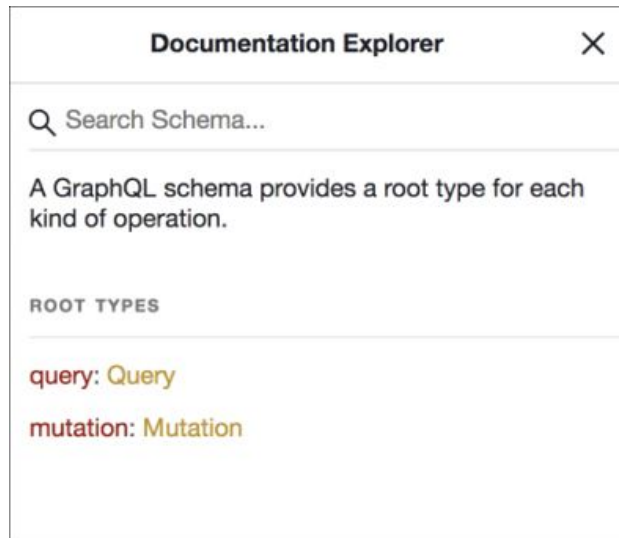


Figura 3.4 – Painel do Documentation Explorer (Explorador de documentação) do GraphQL.

Com muita frequência, você acessará o GraphQL por meio de um URL hospedado junto com o próprio serviço GraphQL. Se você implementar o seu próprio serviço GraphQL, poderá adicionar uma rota que renderize a interface GraphQL de modo que seus usuários possam explorar os dados que você tornar públicos. Também é possível fazer download de uma versão standalone do GraphQL.

GraphQL Playground

Outra ferramenta para explorar as APIs GraphQL é o GraphQL Playground. Criado pela equipe do Prisma, o *GraphQL Playground* espelha as funcionalidades do GraphQL e acrescenta algumas opções interessantes. O modo mais fácil de interagir com um GraphQL Playground é acessá-lo com o navegador em <https://www.graphqlbin.com>. Depois de fornecer um endpoint, você poderá interagir com os dados usando o Playground.

O GraphQL Playground é muito parecido com o GraphQL, mas inclui vários recursos extras que você poderá achar convenientes. O recurso mais importante é a capacidade de enviar cabeçalhos HTTP personalizados junto com sua requisição GraphQL, como vemos na Figura 3.5. (Discutiremos esse recurso com mais detalhes quando

descrevermos a autorização no Capítulo 5.)

O GraphQL Bin também é uma ferramenta incrível para colaboração, pois você pode compartilhar links de seus bins com outras pessoas, como vemos na Figura 3.6.

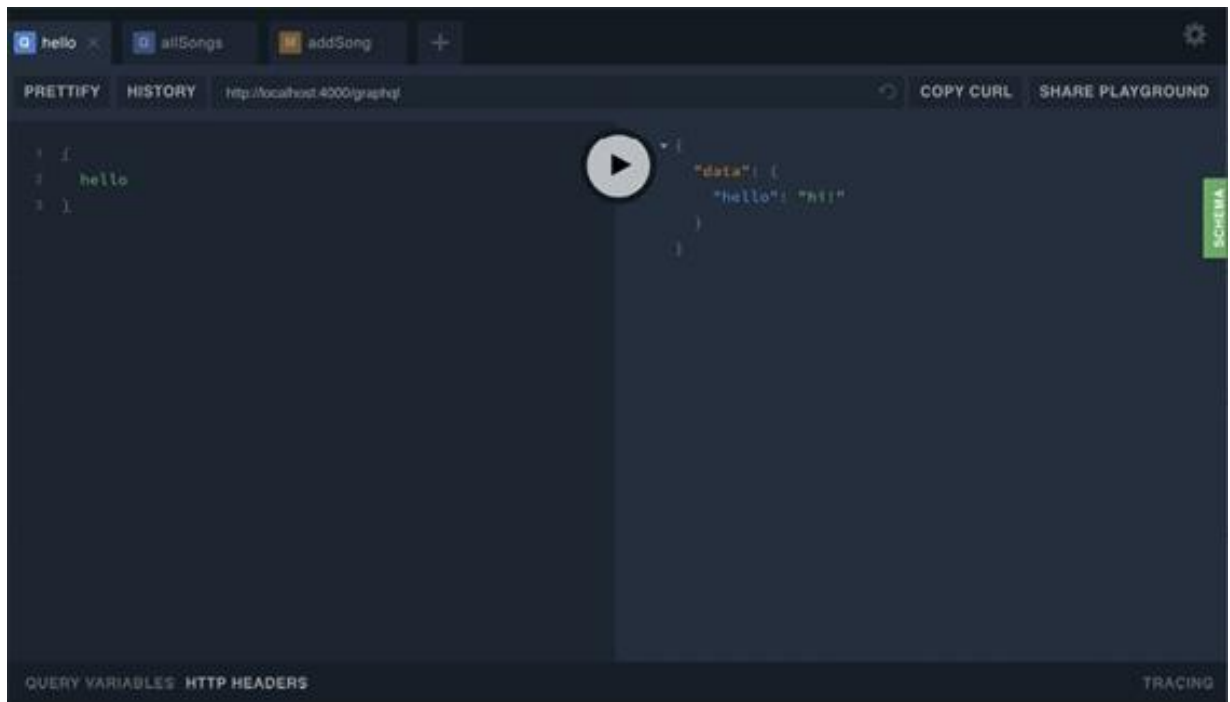


Figura 3.5 – GraphQL Playground.

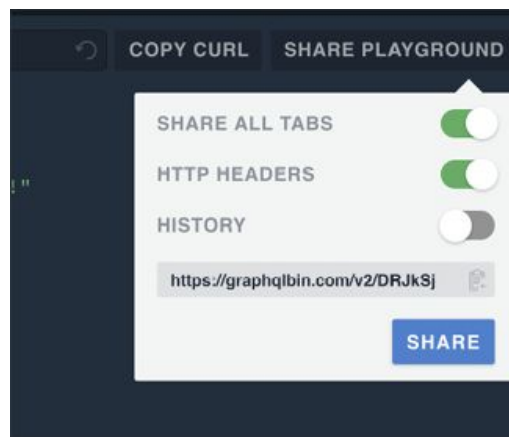


Figura 3.6 – Compartilhando bins.

O GraphQL Playground tem uma versão desktop que pode ser instalada localmente usando o Homebrew:

```
brew cask install graphql-playground
```

Ou você pode simplesmente fazer o download a partir do site em <http://bit.ly/graphql-pg-releases>.

Após a instalação, ou depois que tiver acessado o GraphQL Bin, você poderá começar a enviar consultas. Para começar a trabalhar rapidamente, copie e cole um endpoint de API no Playground. Pode ser uma API pública ou o seu projeto executando em uma porta localhost.

API GraphQL públicas

Uma das melhores maneiras de começar a trabalhar com o GraphQL é exercitar o envio de consultas usando uma API pública. Várias empresas e organizações disponibilizam uma interface GraphiQL que poderá ser usada para consultar dados públicos:

SWAPI (<http://graphql.org/swapi-graphql>) (a API Star Wars)

É um projeto do Facebook, e é um wrapper em torno da API REST SWAPI.

API do GitHub (<https://developer.github.com/v4/explorer/>)

A API GraphQL do GitHub é uma das maiores APIs GraphQL públicas e permite enviar consultas e mutações a fim de visualizar e modificar seus dados ativos no GitHub. É necessário fazer login com sua conta do GitHub para interagir com os dados.

Yelp (<https://www.yelp.com/developers/graphiql>)

O Yelp mantém uma API GraphQL que pode ser usada para consultas com o GraphiQL. É necessário criar uma conta de desenvolvedor no Yelp para interagir com os dados da API do Yelp.

Há muitos exemplos adicionais (<https://github.com/APIs-guru/graphql-apis>) de APIs GraphQL públicas disponíveis.

A consulta GraphQL

O Snowtooth Mountain é um resort de esqui hipotético. Para servir de exemplo neste capítulo, imaginaremos que é uma montanha de verdade e que trabalhamos lá. Veremos como a equipe web do Snowtooth Mountain usa o GraphQL para disponibilizar informações atualizadas em tempo real sobre o status dos teleféricos e das trilhas. A Snowtooth Ski Patrol (Patrulha de Esqui do Snowtooth) pode abrir e fechar os teleféricos e as trilhas diretamente a partir de seus telefones. Para acompanhar os exemplos deste capítulo, consulte a interface do GraphQL Playground para o Snowtooth (<http://snowtooth.moonhighway.com>).

Podemos usar a operação de *consulta* (query) para requisitar dados de uma API. Uma consulta descreve os dados que você quer buscar em um servidor GraphQL. Ao enviar uma consulta, pedimos as unidades de dados por *campo* (field). Esses campos são mapeados para os mesmos campos na resposta contendo dados JSON que você receberá de seu servidor. Por exemplo, se você enviar uma consulta para `allLifts` e pedir os campos `name` e `status`, deverá receber uma resposta JSON contendo um array para `allLifts` e uma string para o `name` e o `status` de cada teleférico, como vemos a seguir:

```
query {  
  allLifts {  
    name  
    status  
  }  
}
```



Tratando erros

Consultas bem-sucedidas devolvem um documento JSON contendo uma chave “data”. Consultas malsucedidas devolvem um documento JSON contendo uma chave “errors”. Os detalhes sobre o que houve de errado são passados como dados JSON por meio dessa chave. Uma resposta JSON pode conter tanto “data” quanto “errors”.

Podemos adicionar várias consultas em um documento, mas somente uma operação pode ser executada de cada vez. Por exemplo, poderíamos colocar duas operações de consulta em um documento:

```
query lifts {
```

```
allLifts {  
  name  
  status  
}  
  
}  
  
query trails {  
  allTrails {  
    name  
    difficulty  
  }  
}
```

Ao pressionar o botão de execução, o GraphQL Playground pedirá que você selecione uma dessas duas operações. Se quiser enviar uma requisição para todos esses dados, será necessário colocar tudo na mesma consulta:

```
query liftsAndTrails {  
  liftCount(status: OPEN)  
  allLifts {  
    name  
    status  
  }  
  allTrails {  
    name  
    difficulty  
  }  
}
```

É nesse cenário que as vantagens do GraphQL começam a se tornar evidentes. Podemos receber todo tipo de dado diferente em uma única consulta. Estamos pedindo `liftCount` por `status`, o que nos dá o número de teleféricos que têm esse status no momento. Também estamos pedindo os campos `name` e `status` de cada teleférico. Por fim, pedimos os campos `name` e `difficulty` de todas as trilhas na mesma consulta.

Uma Query é um tipo do GraphQL. Ela é chamada de *tipo raiz* (root type) porque é um tipo mapeado para uma operação, e as operações representam as raízes de nosso documento de consulta. Os campos disponíveis à query em uma API GraphQL são definidos

no esquema dessa API. A documentação nos informará quais campos estão disponíveis para seleção no tipo `Query`.

Essa documentação nos diz que podemos selecionar os campos `liftCount`, `allLifts` e `allTrails` quando fazemos consultas nessa API. Ela também define mais campos que estão disponíveis para seleção, mas a questão importante no caso de uma consulta é o fato de podermos escolher os campos de que precisamos e aqueles que serão omitidos.

Quando escrevemos consultas, selecionamos os campos necessários colocando-os entre chaves. Esses blocos são chamados de *conjuntos de seleção* (selection sets). Os campos que definimos em um conjunto de seleção estão diretamente relacionados aos tipos do GraphQL. Os campos `liftCount`, `allLifts` e `allTrails` estão definidos no tipo `Query`.

Podemos aninhar os conjuntos de seleção, uns nos outros. Como o campo `allLifts` devolve uma lista de tipos `Lift`, devemos usar chaves para criar um novo conjunto de seleção nesse tipo. Há todo tipo de dado que podemos pedir sobre um teleférico, mas, neste exemplo, queremos selecionar somente o `name` e o `status`. De modo semelhante, a consulta a `allTrails` devolverá tipos `Trail`.

A resposta JSON contém todos os dados que requisitamos na consulta. Esses dados são formatados como JSON e são entregues no mesmo formato de nossa consulta. Cada campo JSON recebe o mesmo nome do campo em nosso conjunto de seleção. Podemos alterar os nomes dos campos no objeto de resposta da consulta especificando um *alias* (apelido), como vemos a seguir:

```
query liftsAndTrails {  
  open: liftCount(status: OPEN)  
  chairlifts: allLifts {  
    liftName: name  
    status  
  }  
  skiSlopes: allTrails {  
    name  
    difficulty  
  }  
}
```

```
}  
}
```

Eis a resposta:

```
{  
  "data": {  
    "open": 5,  
    "chairlifts": [  
      {  
        "liftName": "Astra Express",  
        "status": "open"  
      }  
    ],  
    "skiSlopes": [  
      {  
        "name": "Ditch of Doom",  
        "difficulty": "intermediate"  
      }  
    ]  
  }  
}
```

Temos agora os dados de volta no mesmo formato, porém renomeamos vários campos de nossa resposta. Uma maneira de filtrar o resultado de uma consulta GraphQL é passar *argumentos de consulta*. Argumentos são um par (ou pares) chave-valor associados a um campo da consulta. Se quisermos os nomes somente dos teleféricos fechados, podemos enviar um argumento que filtrará nossa resposta:

```
query closedLifts {  
  allLifts(status: "CLOSED" sortBy: "name") {  
    name  
    status  
  }  
}
```

Os argumentos também podem ser usados para selecionar dados. Por exemplo, suponha que queremos consultar o status de um teleférico em particular. Podemos selecioná-lo com base em seu identificador único:

```
query jazzCatStatus {
```

```
Lift(id: "jazz-cat") {  
  name  
  status  
  night  
  elevationGain  
}
```

Nesse caso, vemos que a resposta contém `name`, `status`, `night` e `elevationGain` para o teleférico “Jazz Cat”.

Arestas e conexões

Na linguagem de consulta GraphQL, os campos podem ser *tipos escalares* ou *tipos objeto*. Os tipos escalares são semelhantes às primitivas em outras linguagens. São as folhas de nossos conjuntos de seleção. O GraphQL possui cinco tipos escalares embutidos: inteiros (`Int`), números de ponto flutuante (`Float`), strings (`String`), Booleanos (`Boolean`) e identificadores únicos (`ID`). Tanto inteiros quanto números de ponto flutuante devolvem números JSON, enquanto `String` e `ID` devolvem strings JSON. `Boolean`s devolvem booleanos. Apesar de `ID` e `String` devolverem o mesmo tipo de dado JSON, o GraphQL ainda garante que os IDs devolvam strings únicas.

Os tipos objeto do GraphQL são grupos de um ou mais campos definidos por você em seu esquema. Eles definem o formato do objeto JSON que deve ser devolvido. O JSON pode aninhar objetos indefinidamente nos campos, e o GraphQL é capaz de fazer o mesmo. Podemos conectar objetos consultando um objeto e obtendo detalhes sobre os objetos relacionados.

Por exemplo, suponha que queremos receber uma lista de trilhas que podem ser acessadas a partir de um teleférico em particular:

```
query trailsAccessedByJazzCat {  
  Lift(id: "jazz-cat") {  
    capacity  
    trailAccess {  
      name  
      difficulty  
    }  
  }  
}
```

```
    }  
  }  
}
```

Na consulta anterior, pedimos alguns dados sobre o teleférico “Jazz Cat”. Nosso conjunto de seleção inclui uma requisição para o campo `capacity`. Esse campo tem um tipo escalar: devolve um inteiro que representa o número de pessoas que podem ocupar uma cadeira. O campo `trailAccess` é do tipo `Trail` (um tipo objeto). Nesse exemplo, `trailAccess` devolve uma lista filtrada de trilhas que são acessíveis a partir de Jazz Cat. Como `trailAccess` é um campo dentro do tipo `Lift`, a API pode usar detalhes sobre o objeto-pai, o `Lift Jazz Cat`, para filtrar a lista de trilhas devolvidas.

Nessa operação de exemplo, há uma consulta de uma *conexão de um para muitos* (one-to-many) entre dois tipos de dados, teleféricos e trilhas. Um teleférico está conectado a várias trilhas relacionadas. Se começarmos a percorrer o nosso grafo a partir do nó `Lift`, podemos chegar até um ou mais nós `Trail` conectados a esse teleférico por meio de uma aresta que chamamos de `trailAccess`. Para que nosso grafo seja considerado não direcionado, teríamos que retornar ao nó `Lift` a partir do nó `Trail`, e é possível fazer isso:

```
query liftToAccessTrail {  
  Trail(id:"dance-fight") {  
    groomed  
    accessedByLifts {  
      name  
      capacity  
    }  
  }  
}
```

Na consulta `liftToAccessTrail`, estamos selecionando uma `Trail` chamada “Dance Fight”. O campo `groomed` devolve um tipo escalar booleano que nos permite saber se Dance Fight está preparada. O campo `accessedByLifts` devolve os teleféricos que levam os esquiadores para a trilha Dance Fight.

Fragmentos

Um documento de consulta GraphQL pode conter definições para operações e *fragmentos* (fragments). Fragmentos são conjuntos de seleção que podem ser reutilizados em várias operações. Observe a consulta a seguir:

```
query {  
  Lift(id: "jazz-cat") {  
    name  
    status  
    capacity  
    night  
    elevationGain  
    trailAccess {  
      name  
      difficulty  
    }  
  }  
  Trail(id: "river-run") {  
    name  
    difficulty  
    accessedByLifts {  
      name  
      status  
      capacity  
      night  
      elevationGain  
    }  
  }  
}
```

Essa consulta solicita informações sobre o teleférico Jazz Cat e a trilha “River Run”. Lift inclui name, status, capacity, night e elevationGain em seu conjunto de seleção. As informações que queremos obter sobre a trilha River Run incluem um subconjunto do tipo Lift para os mesmos campos. Podemos criar um fragmento que nos ajudará a reduzir a redundância em nossa consulta:

```
fragment liftInfo on Lift {  
  name  
  status  
  capacity  
  night  
  elevationGain
```

```
}
```

Crie fragmentos usando o identificador `fragment`. Os fragmentos são conjuntos de seleção em tipos específicos, portanto você deve incluir o tipo associado a cada fragmento em sua definição. O fragmento nesse exemplo se chama `liftInfo` e é um conjunto de seleção no tipo `Lift`.

Se quisermos adicionar os campos do fragmento `liftInfo` em outro conjunto de seleção, podemos fazer isso usando três pontos com o nome do fragmento.

```
query {  
  Lift(id: "jazz-cat") {  
    ...liftInfo  
    trailAccess {  
      name  
      difficulty  
    }  
  }  
  Trail(id: "river-run") {  
    name  
    difficulty  
    accessedByLifts {  
      ...liftInfo  
    }  
  }  
}
```

A sintaxe é parecida com a do operador `spread` de JavaScript, usado com um propósito semelhante: atribuir as chaves e os valores de um objeto para outro. Esses três pontos instruem o GraphQL a atribuir os campos do fragmento no conjunto de seleção atual. Nesse exemplo, podemos selecionar `name`, `status`, `capacity`, `night` e `elevationGain` em dois lugares diferentes de nossa consulta usando um fragmento.

Não poderíamos acrescentar o fragmento `LiftInfo` no conjunto de seleção `Trail` porque ele define campos somente do tipo `Lift`. Podemos acrescentar outro fragmento para as trilhas:

```
query {  
  Lift(id: "jazz-cat") {  
    ...liftInfo
```

```

    trailAccess {
      ...trailInfo
    }
  }
  Trail(id: "river-run") {
    ...trailInfo
    groomed
    trees
    night
  }
}

```

```

fragment trailInfo on Trail {
  name
  difficulty
  accessedByLifts {
    ...liftInfo
  }
}

```

```

fragment liftInfo on Lift {
  name
  status
  capacity
  night
  elevationGain
}

```

Nesse exemplo, criamos um fragmento chamado `trailInfo` e o usamos em dois lugares na nossa consulta. Também estamos usando o fragmento `liftInfo` no fragmento `trailInfo` para selecionar detalhes sobre os teleféricos conectados. Você pode criar quantos fragmentos quiser e usá-los de modo intercambiável. No conjunto de seleção usado na consulta à Trilha River Run, estamos combinando o nosso fragmento com detalhes adicionais que queremos selecionar sobre essa trilha. Os fragmentos podem ser usados junto com outros campos em um conjunto de seleção. Também é possível combinar vários fragmentos no mesmo tipo em um único conjunto de seleção:

```

query {
  allTrails {
    ...trailStatus

```

```
    ...trailDetails
  }
}

fragment trailStatus on Trail {
  name
  status
}

fragment trailDetails on Trail {
  groomed
  trees
  night
}
```

Um aspecto interessante sobre os fragmentos é que você pode modificar os conjuntos de seleção usados em várias consultas diferentes simplesmente modificando um fragmento:

```
fragment liftInfo on Lift {
  name
  status
}
```

Essa mudança no conjunto de seleção no fragmento `liftInfo` faz com que todas as consultas que estejam usando esse fragmento selecionem menos dados.

Tipos união

Já vimos como devolver listas de objetos, mas em cada caso até agora devolvemos listas de um só tipo. Se quiser que uma lista devolva mais de um tipo, você poderá criar um *tipo união*, que faz uma associação entre dois tipos diferentes de objetos.

Suponha que estamos implementando um aplicativo de agenda para estudantes universitários, com o qual eles possam adicionar eventos `Workout` e `Study Group`. Você pode conferir a execução desse exemplo em <https://graphqlbin.com/v2/ANgjtr>.

Se consultar a documentação do GraphQL Playground, você verá que um `AgendaItem` é um tipo união, o que significa que ele pode devolver vários tipos. Mais especificamente, `AgendaItem` pode devolver

um Workout ou um StudyGroup – são itens que podem fazer parte da agenda de um estudante universitário.

Ao escrever uma consulta para a agenda de um estudante, podemos usar fragmentos para definir quais campos devem ser selecionados quando Agendaltem for um Workout, e quais devem ser selecionados quando Agendaltem for um StudyGroup:

```
query schedule {  
  agenda {  
    ...on Workout {  
      name  
      reps  
    }  
    ...on StudyGroup {  
      name  
      subject  
      students  
    }  
  }  
}
```

Eis a resposta:

```
{  
  "data": {  
    "agenda": [  
      {  
        "name": "Comp Sci",  
        "subject": "Computer Science",  
        "students": 12  
      },  
      {  
        "name": "Cardio",  
        "reps": 100  
      },  
      {  
        "name": "Poets",  
        "subject": "English 101",  
        "students": 3  
      },  
      {  
        "name": "Math Whiz",  
        "subject": "Mathematics",
```

```

      "students": 12
    },
    {
      "name": "Upper Body",
      "reps": 10
    },
    {
      "name": "Lower Body",
      "reps": 20
    }
  ]
}

```

Nesse caso, estamos usando *fragmentos inline*. Os fragmentos inline não têm nomes. Eles atribuem conjuntos de seleção a tipos específicos diretamente na consulta. São usados para definir quais campos devem ser selecionados quando uma união devolve tipos diferentes de objetos. Para cada `Workout`, essa consulta pede os campos `name` e `reps` do objeto `Workout` devolvido. Para cada grupo de estudo, pedimos `name`, `subject` e `students` do objeto `StudyGroup` devolvido. A agenda devolvida consistirá de um único array contendo diferentes tipos de objetos.

Também podemos usar fragmentos nomeados para consultar um tipo união:

```

query today {
  agenda {
    ...workout
    ...study
  }
}

```

```

fragment workout on Workout {
  name
  reps
}

```

```

fragment study on StudyGroup {
  name
  subject
}

```

```
    students
  }
```

Interfaces

As *interfaces* são outra opção para lidar com vários tipos de objetos que possam ser devolvidos por um único campo. Uma interface é um tipo abstrato que define uma lista de campos implementados em tipos objeto semelhantes. Quando outro tipo implementar a interface, ele incluirá todos os campos da interface e, em geral, alguns de seus próprios campos. Se quiser acompanhar esse exemplo, você poderá encontrá-lo em GraphQL Bin (<https://graphqlbin.com/v2/yoyPfz>).

Ao observar o campo `agenda` na documentação, você perceberá que ele devolve a interface `ScheduleItem`. Essa interface define os campos: `name`, `start` (horário de início) e `end` (horário de fim). Qualquer tipo objeto que implemente a interface `ScheduleItem` deve implementar esses campos.

A documentação também nos informa que os tipos `StudyGroup` e `Workout` implementam essa interface. Isso significa que podemos supor, com segurança, que ambos os tipos têm campos para `name`, `start` e `end`:

```
query schedule {
  agenda {
    name
    start
    end
  }
}
```

A consulta `schedule` não parece se importar com o fato de o campo `agenda` devolver vários tipos. Ela só precisa do nome e dos horários de início e de fim do item a fim de criar a agenda de quando e onde esse estudante estará.

Ao consultar uma interface, também podemos usar fragmentos para selecionar campos adicionais quando um tipo objeto específico for devolvido:

```
query schedule {  
  agenda {  
    name  
    start  
    end  
    ...on Workout {  
      reps  
    }  
  }  
}
```

A consulta `schedule` foi modificada para requisitar também o campo `reps` quando `ScheduleItem` for um `Workout`.

Mutações

Até agora falamos muito sobre leitura de dados. As consultas descrevem todas as *leituras* que ocorrem no GraphQL. Para escrever novos dados, usamos as *mutações* (mutations). As mutações são definidas como as consultas. Elas têm nomes e podem ter conjuntos de seleção que devolvem tipos objeto ou tipos escalares. A diferença é que as mutações fazem algum tipo de modificação nos dados que afetarão o estado dos dados de seu backend.

Por exemplo, uma mutação perigosa a ser implementada teria este aspecto:

```
mutation burnItDown {  
  deleteAllData  
}
```

`Mutation` é um tipo objeto-raiz. O esquema da API define os campos disponíveis nesse tipo. A API no exemplo anterior tem a capacidade de limpar todos os dados para o cliente, por meio da implementação de um campo chamado `deleteAllData` que devolve um tipo escalar: `true` se todos os dados forem apagados com sucesso, e `false` se algo deu errado, e é hora de começar a procurar um novo emprego. O fato de os dados serem realmente apagados é tratado pela implementação da API,

sobre a qual discutiremos mais no Capítulo 5.

Consideremos outra mutação. Contudo, em vez de destruir algo, vamos criar:

```
mutation createSong {  
  addSong(title:"No Scrubs", numberOne: true, performerName:"TLC") {  
    id  
    title  
    numberOne  
  }  
}
```

Podemos usar esse exemplo para criar novas músicas. `title`, o status `numberOne` e `performerName` são enviados para essa mutação como argumentos, e podemos supor que a mutação adicionará essa nova música em um banco de dados. Se um campo da mutação devolver um objeto, será necessário acrescentar um conjunto de seleção após a mutação. Nesse caso, depois de concluída, a mutação devolverá o tipo `Song` contendo detalhes sobre a música que acabou de ser criada. Podemos selecionar `id`, `title` e o status `numberOne` da nova música após a mutação:

```
{  
  "data": {  
    "addSong": {  
      "id": "5aca534f4bb1de07cb6d73ae",  
      "title": "No Scrubs",  
      "numberOne": true  
    }  
  }  
}
```

O código anterior é um exemplo de como poderia ser a aparência da resposta a essa mutação. Se algo der errado, a mutação devolverá o erro na resposta JSON em vez de devolver o nosso objeto `Song` recém-criado.

As mutações também podem ser usadas para alterar dados existentes. Suponha que quiséssemos alterar o status de um teleférico do Snowtooth. Poderíamos usar uma mutação para isso:

```
mutation closeLift {
```

```
setLiftStatus(id: "jazz-cat" status: CLOSED) {  
  name  
  status  
}  
}
```

Essa mutação pode ser usada para modificar o status do teleférico Jazz Cat de aberto para fechado. Após a mutação, podemos então selecionar os campos de `Lift` que foram recentemente alterados em nosso conjunto de seleção. Nesse caso, recebemos o nome (`name`) do teleférico que foi alterado e o novo `status`.

Usando variáveis de consulta

Até agora modificamos dados enviando novos valores de string como argumentos de uma mutação. Como alternativa, variáveis de *entrada* podem ser usadas. As variáveis substituem o valor estático na consulta para que possamos passar valores dinâmicos em seu lugar. Consideremos nossa mutação `addSong`. Em vez de lidar com strings, usaremos nomes de variáveis, que, no GraphQL, são sempre precedidos por um caractere `$`:

```
mutation createSong($title:String! $numberOne:Int $by:String!) {  
  addSong(title:$title, numberOne:$numberOne, performerName:$by) {  
    id  
    title  
    numberOne  
  }  
}
```

O valor estático é substituído por uma `$variable`. Então definimos que `$variable` pode ser aceita pela mutação. A partir daí, mapeamos cada um dos nomes de `$variable` com o nome do argumento. No GraphQL ou no Playground, há uma janela para Query Variables (Variáveis de consulta). É aí que enviamos os dados de entrada como um objeto JSON. Certifique-se de que usará o nome correto da variável como a chave JSON:

```
{  
  "title": "No Scrubs",  
  "numberOne": true,
```

```
"by": "TLC"  
}
```

As variáveis são muito úteis quando enviamos dados como argumentos. Isso não só deixará nossas mutações mais organizadas em um teste, como também o fato de permitir entradas dinâmicas será muito conveniente mais tarde, quando uma interface cliente for conectada.

Subscriptions

O terceiro tipo de operação disponível no GraphQL é a subscription (inscrição ou assinatura). Há ocasiões em que um cliente pode querer que atualizações sejam enviadas pelo servidor em tempo real. Uma subscription nos permite ouvir a API GraphQL para saber se houve mudanças nos dados em tempo real.

As subscriptions no GraphQL surgiram em virtude de um caso de uso real do Facebook. A equipe queria ter uma forma de mostrar informações em tempo real sobre o número de curtidas (Live Likes) que uma postagem estava obtendo, sem ter que atualizar a página. O Live Likes é um caso de uso de tempo real que funciona graças às subscriptions. Todo cliente é inscrito para receber o evento de curtida (like) e vê as curtidas sendo atualizadas em tempo real.

Assim como a mutação e a consulta, uma subscription é um tipo raiz. Modificações nos dados que os clientes podem ouvir são definidas em um esquema da API na forma de campos do tipo subscription. Escrever a consulta GraphQL para ouvir uma subscription também é feito de forma semelhante à definição de outras operações.

Por exemplo, com o Snowtooth (<http://snowtooth.moonhighway.com>), podemos ouvir uma mudança de estado de qualquer teleférico usando uma subscription:

```
subscription {  
  liftStatusChange {  
    name  
    capacity
```

```
    status
  }
}
```

Quando executarmos essa subscription, ouviremos mudanças de status de um teleférico por meio de uma WebSocket. Observe que clicar no botão de execução no GraphQL Playground não fará com que dados sejam devolvidos imediatamente. Quando a subscription é enviada ao servidor, ela ouvirá qualquer mudança nos dados.

Para ver dados enviados para a subscription, é necessário fazer uma alteração. Temos que abrir uma nova janela ou aba para enviar essa mudança usando uma mutação. Depois que uma operação de subscription estiver executando em uma aba com o GraphQL Playground, não poderemos executar outras operações usando a mesma janela ou aba. Se você estiver usando o GraphQL para escrever as subscriptions, basta abrir uma segunda janela de navegador para a interface do GraphQL. Se estiver usando o GraphQL Playground, uma nova aba poderá ser aberta para adicionar a mutação.

A partir da nova janela ou aba, vamos enviar uma mutação para modificar o status de um teleférico:

```
mutation closeLift {
  setLiftStatus(id: "astra-express" status: HOLD) {
    name
    status
  }
}
```

Quando essa mutação for executada, o status de “Astra Express” mudará, e name, capacity e status do teleférico Astra Express serão enviados para a nossa subscription. O Astra Express é o último teleférico que foi alterado, e o novo status é enviado para a subscription.

Vamos mudar o status de um segundo teleférico. Experimente definir o status do teleférico “Whirlybird” para fechado. Observe que essa nova informação foi passada para a nossa subscription. O GraphQL Playground nos permite ver os dois conjuntos de dados de

resposta, juntamente com o horário em que os dados foram enviados para a subscription.

De modo diferente das consultas e das mutações, as subscriptions permanecem abertas. Novos dados serão enviados para essa subscription sempre que houver uma mudança de status em um teleférico. Para parar de ouvir as mudanças de status, é necessário cancelar a inscrição de sua subscription. Para fazer isso com o GraphQL Playground, basta pressionar o botão stop (parar). Infelizmente, o único modo de cancelar a inscrição em uma subscription com o GraphQL é fechar a aba do navegador na qual a subscription estiver executando.

Introspecção

Um dos recursos mais eficazes do GraphQL é a *introspecção*. A introspecção é a capacidade de consultar detalhes sobre o esquema atual da API. A introspecção é o modo pelo qual aqueles documentos elegantes do GraphQL são adicionados à interface do GraphQL Playground.

Podemos enviar consultas para toda API GraphQL que devolva dados sobre o esquema de uma dada API. Por exemplo, se quisermos saber quais tipos do GraphQL estão disponíveis no Snowtooth, podemos visualizar essas informações executando uma consulta `__schema`, como vemos a seguir:

```
query {  
  __schema {  
    types {  
      name  
      description  
    }  
  }  
}
```

Quando essa consulta é executada, vemos todos os tipos disponíveis na API, incluindo os tipos raiz, os tipos personalizados e até mesmo os tipos escalares. Se quisermos ver os detalhes de um

tipo específico, podemos executar a consulta `__type` e enviar o nome do tipo que queremos consultar como argumento:

```
query liftDetails {
  __type(name:"Lift") {
    name
    fields {
      name
      description
      type {
        name
      }
    }
  }
}
```

Essa consulta de introspecção nos mostra todos os campos que estão disponíveis para consulta no tipo `Lift`. Ao conhecer uma nova API GraphQL, descobrir quais campos estão disponíveis nos tipos raiz é uma boa ideia:

```
query roots {
  __schema {
    queryType {
      ...typeFields
    }
    mutationType {
      ...typeFields
    }
    subscriptionType {
      ...typeFields
    }
  }
}

fragment typeFields on __Type {
  name
  fields {
    name
  }
}
```

Uma consulta de introspecção segue as regras da linguagem de consulta GraphQL. A redundância na consulta anterior foi reduzida

com o uso de fragmento. Estamos consultando o nome do tipo e os campos disponíveis em cada tipo raiz. A introspecção dá ao cliente a capacidade de descobrir como o esquema atual da API funciona.

Árvores sintáticas abstratas

O documento de consulta é uma string. Quando enviamos uma consulta para uma API GraphQL, um parse dessa string é feito, gerando uma *árvore sintática abstrata*, e ela será validada antes que a operação seja executada. Uma AST (Abstract Syntax Tree, ou Árvore Sintática Abstrata) é um objeto hierárquico que representa a nossa consulta. A AST é um objeto contendo campos aninhados que representam os detalhes de uma consulta GraphQL.

O primeiro passo nesse processo é fazer parse da string, gerando um conjunto de partes menores. Isso inclui fazer parsing de palavras reservadas, argumentos e até mesmo de colchetes e dois-pontos, criando um conjunto de tokens individuais. Esse processo se chama *análise léxica* (lexing ou lexical analysis). Em seguida, um parse da consulta após a análise léxica é feito gerando uma AST. Uma consulta é muito mais fácil de ser modificada dinamicamente e validada na forma de uma AST.

Por exemplo, suas consultas começam como um *documento* GraphQL. Um documento contém pelo menos uma *definição*, mas pode conter também uma lista de definições. As definições são de um entre dois tipos: `OperationDefinition` ou `FragmentDefinition`. A seguir, apresentamos um exemplo de um documento que contém três definições: duas operações e um fragmento:

```
query jazzCatStatus {  
  Lift(id: "jazz-cat") {  
    name  
    night  
    elevationGain  
    trailAccess {  
      name  
      difficulty  
    }  
  }  
}
```

```

    }
  }

  mutation closeLift($lift: ID!) {
    setLiftStatus(id: $lift, status: CLOSED ) {
      ...liftStatus
    }
  }

  fragment liftStatus on Lift {
    name
    status
  }

```

Uma `OperationDefinition` pode conter somente um dentre três tipos de operação: `mutation`, `query` ou `subscription`. Cada definição de operação contém o `OperationType` e o `SelectionSet`.

As chaves, após cada operação, contêm o `SelectionSet` da operação. Esses são os campos propriamente ditos que estamos consultando, junto com seus argumentos. Por exemplo, o campo `Lift` é o `SelectionSet` da consulta `jazzCatStatus`, e o campo `setLiftStatus` representa o conjunto de seleção da mutação `closeLift`.

Os conjuntos de seleção estão aninhados uns nos outros. A consulta `jazzCatStatus` tem três conjuntos de seleção aninhados. O primeiro `SelectionSet` contém o campo `Lift`. Aninhado aí, encontra-se um `SelectionSet` que contém os campos: `name`, `night`, `elevationGain` e `trailAccess`. Aninhado no campo `trailAccess`, temos outro `SelectionSet` que contém os campos `name` e `difficulty` para cada trilha.

O GraphQL é capaz de percorrer essa AST e validar seus detalhes com base na linguagem GraphQL e no esquema atual. Se a sintaxe da linguagem de consulta estiver correta e o esquema contiver os campos e os tipos que estamos requisitando, a operação será executada. Caso contrário, um erro específico será devolvido.

Além do mais, esse objeto AST é mais fácil de ser modificado se comparado com uma string. Se quisermos concatenar o número de teleféricos abertos na consulta `jazzCatStatus`, poderíamos fazer isso modificando diretamente a AST. Tudo que temos a fazer é

acrescentar mais um `SelectionSet` à operação. As ASTs são uma parte essencial do GraphQL. O parse de cada operação é feito e uma AST é gerada, de modo que possa ser validada e, posteriormente, executada.

Neste capítulo, conhecemos a linguagem de consulta GraphQL. Podemos agora usar essa linguagem para interagir com um serviço GraphQL. Contudo, nada disso seria possível se não houvesse uma definição específica das operações e campos que estão disponíveis em um serviço GraphQL em particular. Essa definição específica se chama *esquema do GraphQL*, e veremos como criar esquemas com mais detalhes no próximo capítulo.

CAPÍTULO 4

Design de um esquema

O GraphQL mudará o seu processo de design. Em vez de ver suas APIs como um conjunto de endpoints REST, você começará a vê-las como coleções de tipos. Antes de pôr a mão na massa em sua nova API, é necessário pensar, discutir e definir formalmente os tipos de dados que ela exporá. Esse conjunto de tipos é chamado de *esquema* (schema).

Schema First (Esquema Primeiro) é uma metodologia de design que fará com que toda a sua equipe esteja igualmente ciente dos tipos de dados que compõem a sua aplicação. A equipe de backend terá uma compreensão clara sobre os dados que deverá armazenar e entregar. A equipe de frontend terá as definições de que precisa para começar a implementar as interfaces de usuário. Todos terão um vocabulário claro que poderão usar para se comunicar sobre o sistema que estão desenvolvendo. Em suma, todos poderão trabalhar.

Para facilitar a definição dos tipos, o GraphQL inclui uma linguagem que pode ser usada para definir os esquemas; ela se chama *SDL* (Schema Definition Language, ou Linguagem de Definição de Esquema). Assim como a Linguagem de Consulta GraphQL, a SDL do GraphQL não muda, independentemente da linguagem ou do framework que você usar para construir suas aplicações. Os documentos de esquema do GraphQL são documentos em formato texto que definem os tipos disponíveis em uma aplicação, e são usados posteriormente tanto pelos clientes quanto pelos servidores para validar requisições GraphQL.

Neste capítulo, conheceremos a SDL do GraphQL e construiremos

um esquema para uma aplicação de compartilhamento de fotos.

Definindo tipos

A melhor maneira de conhecer os tipos e esquemas do GraphQL é implementando um esquema. A aplicação de compartilhamento de fotos permitirá que os usuários façam login com suas contas do GitHub para postar fotos e marcar usuários (atribuir tags) nessas fotos. Administrar usuários e postagens representa uma funcionalidade que é essencial para praticamente qualquer tipo de aplicação na internet.

A aplicação PhotoShare terá dois tipos principais: `User` e `Photo`. Vamos começar fazendo o design do esquema para a aplicação toda.

Tipos

A unidade básica de qualquer esquema do GraphQL é o tipo. No GraphQL, um *tipo* representa um objeto personalizado, e esses objetos descrevem os recursos principais de sua aplicação. Por exemplo, uma aplicação de mídia social é composta de `Users` e `Posts`. Um blog seria constituído de `Categories` e `Articles`. Os tipos representam os dados de sua aplicação.

Se fôssemos desenvolver o Twitter do zero, um `Post` conteria o texto que o usuário quer divulgar. (Nesse caso, um `Tweet` poderia ser um nome melhor para esse tipo.) Se estivéssemos desenvolvendo o Snapchat, um `Post` conteria uma imagem, e `Snap` seria um nome mais apropriado. Ao definir um esquema, você estabelecerá uma linguagem comum que sua equipe usará quando falar sobre os objetos de seus domínios.

Um tipo tem *campos* que representam os dados associados a cada objeto. Cada campo devolve um tipo específico de dado. Pode ser um inteiro ou uma string, mas poderia ser também um tipo objeto personalizado ou uma lista de tipos.

Um esquema é um conjunto de definições de tipos. Você pode escrever seus esquemas em um arquivo JavaScript como uma string ou em qualquer arquivo texto. Em geral, esses arquivos usam a extensão `.graphql`.

Vamos definir o primeiro tipo de objeto GraphQL, `Photo`, em nosso arquivo de esquema:

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

Entre as chaves, definimos os campos de `Photo`. O `url` de `Photo` é uma referência ao local em que está o arquivo da imagem. Essa descrição contém também alguns metadados sobre `Photo`: um `name` e uma `description`. Por fim, cada `Photo` terá um `ID`, isto é, um identificador único que pode ser usado como chave para acessar a foto.

Cada campo contém dados de um tipo específico. Definimos somente um tipo personalizado em nossos esquemas, isto é, `Photo`, mas o GraphQL inclui alguns tipos embutidos que podem ser usados em nossos campos. Esses tipos embutidos são chamados de *tipos escalares*. Os campos `description`, `name` e `url` usam o tipo escalar `String`. Os dados devolvidos quando consultamos esses campos serão strings JSON. O ponto de exclamação estabelece que o campo é *não nullable* (não pode ser null), isto é, significa que os campos `name` e `url` devem devolver algum dado em cada consulta. O campo `description` é *nullable* (pode ser null), ou seja, as descrições das fotos são opcionais. Quando consultado, esse campo pode devolver `null`.

O campo `ID` de `Photo` define um identificador único para cada foto. No GraphQL, o tipo escalar `ID` é usado quando um identificador único tiver que ser devolvido. O valor JSON para esse identificador será uma string, mas essa string será validada como um valor único.

Tipos escalares

Os tipos escalares embutidos no GraphQL (Int, Float, String, Boolean, ID) são muito úteis, mas pode haver situações em que você vai querer definir seus próprios tipos escalares personalizados. Um tipo escalar não é um tipo objeto. Ele não tem campos. No entanto, ao implementar um serviço GraphQL, você poderá definir como os tipos escalares personalizados serão validados. Por exemplo:

```
scalar DateTime
```

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

Nesse caso, criamos um tipo escalar personalizado: `DateTime`. Agora podemos descobrir quando cada foto foi criada consultando `created`. Qualquer campo marcado com `DateTime` devolverá uma string JSON, mas podemos usar o escalar personalizado para garantir que a string possa ser serializada, validada e formatada como uma data e hora oficiais.

É possível declarar escalares personalizados para qualquer tipo que tiver que ser validado.



O pacote `graphql-custom-types` do npm contém alguns tipos escalares personalizados de uso comum, que poderão ser rapidamente adicionados ao seu serviço GraphQL Node.js.

Enumerados

Os *tipos enumerados*, ou *enums*, são tipos escalares que possibilitam que um campo devolva um conjunto restrito de valores de string. Se quiser garantir que um campo devolva um valor de um conjunto limitado de valores, um tipo `enum` poderá ser usado.

Por exemplo, vamos criar um tipo `enum` chamado `PhotoCategory` que

define o tipo da foto sendo postada, de um conjunto de cinco opções possíveis: SELFIE, PORTRAIT, ACTION, LANDSCAPE OU GRAPHIC:

```
enum PhotoCategory {  
    SELFIE  
    PORTRAIT  
    ACTION  
    LANDSCAPE  
    GRAPHIC  
}
```

Os tipos enumerados podem ser usados na definição dos campos. Vamos adicionar um campo `category` em nosso tipo objeto `Photo`:

```
type Photo {  
    id: ID!  
    name: String!  
    url: String!  
    description: String  
    created: DateTime!  
    category: PhotoCategory!  
}
```

Agora que adicionamos `category`, garantimos que ele devolverá um dos cinco valores válidos quando o serviço for implementado.



Não importa se sua implementação tenha suporte completo para tipos enumerados. Os campos com enumerados no GraphQL podem ser implementados em qualquer linguagem.

Conexões e listas

Ao criar esquemas GraphQL, é possível definir campos que devolvam listas de qualquer tipo GraphQL. As listas são criadas com a inserção de colchetes ao redor de um tipo GraphQL. `[String]` define uma lista de strings e `[PhotoCategory]` define uma lista de categorias de fotos. Conforme discutimos no Capítulo 3, as listas também podem ser compostas de vários tipos se os tipos `union` ou `interface` forem incluídos. Discutiremos esses tipos de lista com mais detalhes ao final do capítulo.

Às vezes, o ponto de exclamação pode causar um pouco de

confusão na definição de listas. Quando o ponto de exclamação vier depois do colchete de fechamento, é sinal de que o próprio campo é não nullable. Se vier antes do colchete de fechamento, significa que os valores contidos na lista são não nullable. Sempre que você vir um ponto de exclamação, o valor será obrigatório, e null não poderá ser devolvido. A Tabela 4.1 define essas situações variadas.

Tabela 4.1 – Regras para nullable com listas

Declaração da lista	Definição
[Int]	Uma lista de valores inteiros nullable
[Int!]	Uma lista de valores inteiros não nullable
[Int]!	Uma lista não nullable de valores inteiros nullable
[Int!]!	Uma lista não nullable de valores inteiros não nullable

A maioria das definições de lista são de listas não nullable com valores não nullable. Isso ocorre porque, em geral, não queremos que os valores em nossa lista sejam null. Devemos filtrar eliminando qualquer valor null com antecedência. Se nossa lista não contiver nenhum valor, poderemos simplesmente devolver um array JSON vazio, por exemplo, []. Um array vazio tecnicamente não é null: é somente um array que não contém nenhum valor.

A capacidade de conectar dados e consultar vários tipos de dados relacionados é um recurso muito importante. Quando criamos listas com nossos tipos objeto personalizados, estamos usando esse recurso eficaz e conectando objetos uns aos outros.

Nesta seção discutiremos como usar uma lista para conectar tipos objeto.

Conexões de um para um

Quando criamos campos com base em tipos objeto personalizados, estamos conectando dois objetos. Na teoria dos grafos, uma conexão ou ligação entre dois objetos é chamada de *aresta* (edge).

O primeiro tipo de conexão é uma conexão de um para um (one-to-one), na qual conectamos um único tipo objeto a outro tipo objeto.

As fotos são postadas por usuários, portanto toda foto em nosso sistema deve ter uma aresta que conecte a foto ao usuário que a postou. A Figura 4.1 mostra uma conexão unidirecional entre dois tipos: Photo e User. A aresta que conecta os dois nós se chama postedBy.

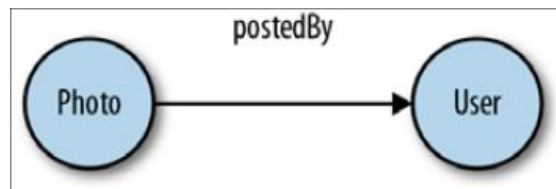


Figura 4.1 – Conexão de um para um.

Vamos ver como podemos definir isso no esquema:

```
type User {  
  githubLogin: ID!  
  name: String  
  avatar: String  
}  
  
type Photo {  
  id: ID!  
  name: String!  
  url: String!  
  description: String  
  created: DateTime!  
  category: PhotoCategory!  
  postedBy: User!  
}
```

Inicialmente adicionamos um novo tipo, `User`, em nosso esquema. Os usuários da aplicação PhotoShare farão login usando o GitHub. Quando o usuário fizer login, obteremos o seu `githubLogin` e o usaremos como identificador único para o seu registro de usuário. Opcionalmente, se eles adicionarem seu nome ou uma foto no GitHub, salvaremos essas informações nos campos `name` e `avatar`.

Em seguida, inserimos a conexão adicionando um campo `postedBy` ao

objeto foto. Toda foto deve ser postada por um usuário, portanto esse campo é definido com o tipo `User!`; o ponto de exclamação foi adicionado para deixar esse campo não nullable.

Conexões de um para muitos

Quando possível, é uma boa ideia manter os serviços GraphQL não direcionados. Isso possibilita que nossos clientes tenham o máximo de flexibilidade para criar consultas, pois poderão iniciar um percurso no grafo partindo de qualquer nó. Tudo que temos que fazer para seguir essa prática é prover um caminho de volta, dos tipos `User` para os tipos `Photo`. Isso significa que, quando consultarmos um `User`, devemos ser capazes de ver todas as fotos postadas por esse usuário em particular:

```
type User {  
  githubLogin: ID!  
  name: String  
  avatar: String  
  postedPhotos: [Photo!]!  
}
```

Ao adicionar o campo `postedPhotos` no tipo `User`, possibilitamos um caminho de volta para `Photo` a partir do usuário. O campo `postedPhotos` devolverá uma lista de tipos `Photo`; essas fotos foram postadas pelo usuário-pai. Como um usuário pode postar várias fotos, criamos uma conexão de um para muitos (one-to-many). As conexões de um para muitos, como vemos na Figura 4.2, são conexões comuns, criadas quando um objeto-pai contém um campo que lista outros objetos.

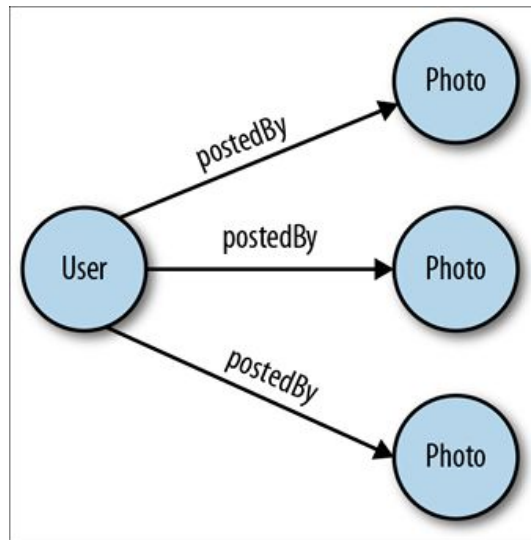


Figura 4.2 – Conexão de um para muitos.

Um lugar comum para adicionar conexões de um para muitos é em nossos tipos raiz. Para deixar nossas fotos ou usuários disponíveis a uma consulta, devemos definir os campos de nosso tipo raiz `Query`. Vamos observar como podemos adicionar nossos novos tipos personalizados ao tipo raiz `Query`.

```
type Query {  
  totalPhotos: Int!  
  allPhotos: [Photo!]!  
  totalUsers: Int!  
  allUsers: [User!]!  
}  
  
schema {  
  query: Query  
}
```

A criação do tipo `Query` define as consultas que estão disponíveis em nossa API. Nesse exemplo, adicionamos duas consultas para cada tipo: uma para fornecer o número total de registros disponíveis de cada tipo e outra para obter a lista completa desses registros. Além do mais, acrescentamos o tipo `Query` em `schema` como um arquivo. Isso deixa nossas consultas disponíveis em nossa API GraphQL.

Agora nossas fotos e usuários podem ser consultados usando a string de consulta a seguir:

```

query {
  totalPhotos
  allPhotos {
    name
    url
  }
}

```

Conexões de muitos para muitos

Às vezes, queremos conectar listas de nós a outras listas de nós. Nossa aplicação PhotoShare permitirá que os usuários identifiquem outros usuários em cada foto que postarem. Esse processo se chama *marcação* (tagging). Uma foto pode exibir vários usuários, e um usuário pode ser marcado em muitas fotos, como mostra a Figura 4.3.

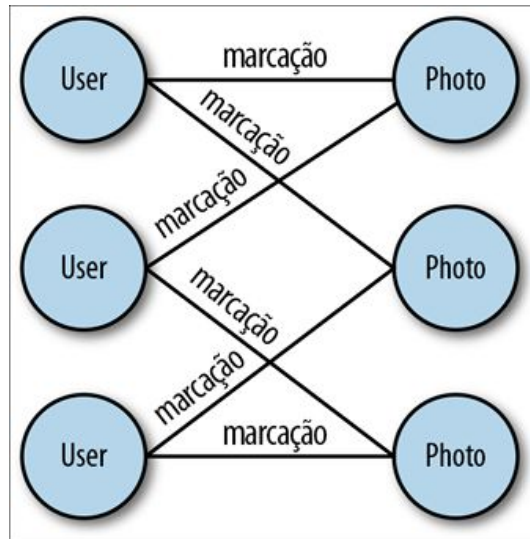


Figura 4.3 – Conexão de muitos para muitos.

Para criar esse tipo de conexão, devemos adicionar campos de lista nos tipos User e Photo.

```

type User {
  ...
  inPhotos: [Photo!]!
}

type Photo {
  ...

```

```
    taggedUsers: [User!]!  
  }
```

Como podemos ver, uma conexão de *muitos para muitos* (many-to-many) é composta de duas conexões de um para muitos. Nesse caso, uma `Photo` pode ter muitos usuários marcados, e um `User` pode estar marcado em várias fotos.

Through types

Às vezes, quando criamos relacionamentos de muitos para muitos, talvez você queira armazenar alguma informação sobre o próprio relacionamento. Como não há nenhuma verdadeira necessidade de um through type em nossa aplicação de compartilhamento de fotos, usaremos um exemplo diferente para defini-lo: uma amizade entre usuários.

Podemos conectar vários usuários a vários usuários definindo um campo em `User` que contenha uma lista de outros usuários:

```
type User {  
  friends: [User!]!  
}
```

Nesse caso, definimos uma lista de amigos para cada usuário. Considere um caso em que quiséssemos salvar algumas informações sobre a própria amizade, por exemplo, há quanto tempo os usuários se conhecem ou onde se conheceram.

Nessa situação, temos que definir a aresta como um tipo objeto personalizado. Chamamos a esse objeto de *through type* porque é um nó criado para conectar dois nós. Vamos definir um through type chamado `Friendship`, que pode ser usado para conectar dois amigos, mas que também fornece dados sobre como os amigos estão conectados:

```
type User {  
  friends: [Friendship!]!  
}  
type Friendship {  
  friend_a: User!  
  friend_b: User!  
  howLong: Int!
```

```
    whereWeMet: Location
  }
```

Em vez de definir o campo `friends` diretamente como uma lista de outros tipos `User`, criamos um `Friendship` para conectar os amigos. O tipo `Friendship` define os dois amigos conectados: `friend_a` e `friend_b`. Define também alguns campos com detalhes sobre como os amigos estão conectados: `howLong` e `whereWeMet`. O campo `howLong` é um `Int` que define a duração da amizade, enquanto o campo `whereWeMet` está associado a um tipo personalizado chamado `Location`.

Podemos melhorar o design do tipo `Friendship` permitindo que um grupo de amigos possa fazer parte da amizade. Por exemplo, talvez você tenha conhecido seus melhores amigos na mesma época, no primeiro ano do Ensino Fundamental. Podemos permitir que dois ou mais amigos façam parte da amizade adicionando um único campo chamado `friends`:

```
type Friendship {
  friends: [User!]!
  how_long: Int!
  where_we_met: Location
}
```

Incluímos somente um campo para todos os amigos (`friends`) em `Friendship`. A partir de agora, esse campo pode representar dois ou mais amigos.

Lista de tipos diferentes

No GraphQL, nossas listas nem sempre precisam devolver o mesmo tipo. No Capítulo 3, apresentamos os tipos `union` e as `interfaces`, e aprendemos a escrever consultas para esses tipos usando fragmentos. Vamos ver como podemos acrescentar esses tipos em nosso esquema.

Usaremos uma agenda, nesse caso, como exemplo. Você pode ter uma agenda composta de diferentes eventos, cada um exigindo campos de dados distintos. Por exemplo, os detalhes sobre uma reunião de grupo de estudos ou de uma atividade de ginástica

podem ser totalmente diferentes, mas você deverá ser capaz de adicionar ambos em uma agenda. Podemos pensar em uma agenda diária como uma lista de diferentes tipos de atividades.

Há duas maneiras de lidar com a definição de um esquema para uma agenda no GraphQL: uniões e interfaces.

Tipos união

No GraphQL, um *tipo união* é um tipo que podemos usar para devolver um entre vários tipos diferentes. Lembre-se de como, no Capítulo 3, escrevemos uma consulta chamada `schedule` que consultava uma agenda e devolvia dados distintos conforme o item da agenda fosse fazer ginástica (`workout`) ou uma reunião de grupo de estudos (`study group`). Vamos ver o código novamente:

```
query schedule {  
  agenda {  
    ...on Workout {  
      name  
      reps  
    }  
    ...on StudyGroup {  
      name  
      subject  
      students  
    }  
  }  
}
```

Na agenda diária do estudante, poderíamos lidar com isso criando um tipo `union` chamado `AgendaItem`:

```
union AgendaItem = StudyGroup | Workout  
  
type StudyGroup {  
  name: String!  
  subject: String  
  students: [User!]!  
}  
  
type Workout {  
  name: String!
```

```
    reps: Int!  
  }  
  
  type Query {  
    agenda: [AgendaItem!]!  
  }
```

AgendaItem combina grupos de estudo e ginástica em um só tipo. Quando adicionamos o campo `agenda` em nossa `Query`, nós a definimos como uma lista de atividades de ginástica ou de grupos de estudo.

É possível reunir quantos tipos quisermos em uma única união. Basta separar cada tipo com um caractere pipe:

```
union = StudyGroup | Workout | Class | Meal | Meeting | FreeTime
```

Interfaces

Outra maneira de tratar campos que possam conter vários tipos é usar uma interface. As *interfaces* são tipos abstratos que podem ser implementados por tipos objeto. Uma interface define todos os campos que devem ser incluídos em qualquer objeto que a implemente. São uma ótima maneira de organizar código em seu esquema. Elas garantem que determinados tipos sempre incluam campos específicos que possam ser consultados, independentemente do tipo devolvido.

No Capítulo 3 escrevemos uma consulta para uma `agenda` usando uma interface para devolver campos de diferentes itens de agenda. Vamos rever esse código:

```
query schedule {  
  agenda {  
    name  
    start  
    end  
    ...on Workout {  
      reps  
    }  
  }  
}
```

Essa é a aparência de uma consulta a uma `agenda` que implementava

interfaces. Para que um tipo tenha interface com nosso item de agenda, ele deve conter campos específicos que todos os itens de agenda implementarão. Esses campos incluem `name`, `start` (horário e início) e `end` (horário de fim). Não importa o tipo do item de agenda que você tiver, todos eles devem ter esses detalhes para que possam ser listados em uma agenda.

Eis o modo como implementaríamos essa solução em nosso esquema GraphQL:

```
scalar DateTime
```

```
interface Agendaltem {  
  name: String!  
  start: DateTime!  
  end: DateTime!  
}
```

```
type StudyGroup implements Agendaltem {  
  name: String!  
  start: DateTime!  
  end: DateTime!  
  participants: [User!]!  
  topic: String!  
}
```

```
type Workout implements Agendaltem {  
  name: String!  
  start: DateTime!  
  end: DateTime!  
  reps: Int!  
}
```

```
type Query {  
  agenda: [Agendaltem!]!  
}
```

Nesse exemplo, criamos uma interface chamada `Agendaltem`. Essa interface é um tipo abstrato que outros tipos podem implementar. Quando outro tipo implementa uma interface, ele deverá conter os campos definidos pela interface. Tanto `StudyGroup` quanto `Workout` implementam a interface `Agendaltem`, portanto ambos devem ter os

campos `name`, `start` e `end`. A consulta `agenda` devolve uma lista de tipos `AgendaItem`. Qualquer tipo que implemente a interface `AgendaItem` pode ser devolvido na lista de agenda.

Note também que esses tipos podem implementar outros campos. Um `StudyGroup` tem um `topic` e uma lista de `participants`, enquanto um `Workout` tem `reps`. Esses campos adicionais podem ser selecionados em uma consulta usando fragmentos.

Tanto os tipos união quanto as interfaces são ferramentas que podem ser usadas para criar campos contendo diferentes tipos de objeto. Cabe a você decidir quando usar um ou outro. Em geral, se os objetos contiverem campos totalmente distintos, é uma boa ideia usar tipos união. Eles são mais eficientes. Se um tipo objeto tiver que conter campos específicos para fazer interface com outro tipo de objeto, você deverá usar uma interface no lugar de um tipo união.

Argumentos

Argumentos podem ser adicionados em qualquer campo no GraphQL. Eles nos permitem enviar dados que podem afetar o resultado de nossas operações com o GraphQL. No Capítulo 3 vimos como usar argumentos de usuário em nossas consultas e mutações. Vamos agora aprender a definir argumentos em nosso esquema.

O tipo `Query` contém campos que listarão `allUsers` ou `allPhotos`, mas o que aconteceria se quiséssemos selecionar somente um `User` ou uma `Photo`? Seria necessário fornecer algumas informações sobre o usuário ou a foto que gostaríamos de selecionar. Essas informações podem ser enviadas com a consulta na forma de argumento:

```
type Query {  
  ...  
  User(githubLogin: ID!): User!  
  Photo(id: ID!): Photo!  
}
```

Assim como um campo, um argumento deve ter um tipo. Esse tipo

pode ser definido com qualquer um dos tipos escalares ou tipos objeto disponíveis em nosso esquema. Para selecionar um usuário específico, devemos enviar o `githubLogin` único desse usuário como argumento. A consulta a seguir seleciona somente o nome e o avatar de MoonTahoe:

```
query {  
  User(githubLogin: "MoonTahoe") {  
    name  
    avatar  
  }  
}
```

Para selecionar detalhes sobre uma foto individual, devemos fornecer o ID dessa foto:

```
query {  
  Photo(id: "14TH5B6NS4KIG3H4S") {  
    name  
    description  
    url  
  }  
}
```

Nos dois casos argumentos foram necessários para consultar detalhes sobre um registro específico. Pelo fato de serem obrigatórios, esses argumentos são definidos como campos não nullable. Se não especificarmos o `id` ou o `githubLogin` nessas consultas, o parser do GraphQL devolverá um erro.

Filtrando dados

Os argumentos não precisam ser não nullable. Podemos acrescentar argumentos opcionais usando campos nullable. Isso significa que podemos fornecer argumentos como parâmetros opcionais quando executamos operações de consulta. Por exemplo, poderíamos filtrar a lista de fotos devolvida pela consulta `allPhotos` de acordo com a categoria da foto:

```
type Query {  
  ...  
  allPhotos(category: PhotoCategory): [Photo!]!
```

```
}
```

Adicionamos um campo `category` opcional à consulta `allPhotos`. A categoria deve coincidir com os valores do tipo enumerado `PhotoCategory`. Se um valor não for enviado com a consulta, podemos supor que esse campo devolverá todas as fotos. No entanto, se uma categoria for especificada, teremos uma lista filtrada de fotos com a mesma categoria:

```
query {  
  allPhotos(category: "SELFIE") {  
    name  
    description  
    url  
  }  
}
```

Essa consulta devolve os campos `name`, `description` e `url` de todas as fotos cuja categoria é `SELFIE`.

Paginação de dados

Se nossa aplicação `PhotoShare` for bem-sucedida – e será –, ela terá muitos `Users` e `Photos`. Devolver todos os `Users` e todas as `Photos` em nossa aplicação talvez não seja possível. É possível usar argumentos do GraphQL para controlar o volume de dados devolvido por nossas consultas. Esse processo se chama *paginação de dados* (data paging), pois um número específico de registros é devolvido, representando uma página de dados.

Para implementar a paginação de dados, adicionaremos dois argumentos opcionais: `first` para obter o número de registros que devem ser devolvidos de uma só vez em uma única página de dados, e `start` para definir a posição inicial ou índice do primeiro registro a ser devolvido. Esses argumentos podem ser adicionados em nossas duas consultas de listas:

```
type Query {  
  ...  
  allUsers(first: Int=50 start: Int=0): [User!]!  
  allPhotos(first: Int=25 start: Int=0): [Photo!]!  
}
```

No exemplo anterior, adicionamos argumentos opcionais para `first` e `start`. Se o cliente não especificar esses argumentos na consulta, os valores default definidos serão usados. Por padrão, toda consulta a `allUsers` devolve somente os 50 primeiros usuários, enquanto a consulta a `allPhotos` devolve as 25 primeiras fotos.

O cliente pode consultar um intervalo diferente de usuários ou de fotos se fornecer valores para esses argumentos. Por exemplo, se quisermos selecionar os usuários de 90 a 100, poderemos fazer isso usando a consulta a seguir:

```
query {  
  allUsers(first: 10 start: 90) {  
    name  
    avatar  
  }  
}
```

Essa consulta seleciona somente 10 usuários, começando pelo nonagésimo. Ela deve devolver o `name` e o `avatar` desse intervalo específico de usuários. Podemos calcular o número total de páginas disponíveis ao cliente dividindo o número total de itens pelo tamanho de uma página de dados:

```
pages = pageSize/total
```

Ordenação

Ao consultar uma lista de dados, é possível definir como a lista devolvida estará ordenada. Podemos usar argumentos para isso também.

Considere um cenário em que queremos incluir a capacidade de ordenar qualquer lista de registros de `Photo`. Uma maneira de encarar esse desafio é criar `enums` que especifiquem quais campos podem ser usados para ordenar objetos `Photo` e instruções sobre como ordenar esses campos:

```
enum SortDirection {  
  ASCENDING  
  DESCENDING  
}
```

```

enum SortablePhotoField {
  name
  description
  category
  created
}

Query {
  allPhotos(
    sort: SortDirection = DESCENDING
    sortBy: SortablePhotoField = created
  ): [Photo!]!
}

```

Nesse caso, adicionamos os argumentos `sort` e `sortBy` à consulta `allPhotos`. Criamos um tipo enumerado chamado `SortDirection` que pode ser usado para limitar os valores do argumento `sort` para `ASCENDING` ou `DESCENDING`. Também criamos outro tipo enumerado para `SortablePhotoField`. Não queremos ordenar as fotos com base em qualquer campo, portanto restringimos os valores de `sortBy` para que incluam somente quatro dos campos das fotos: `name`, `description`, `category` ou `created` (a data e a hora em que a foto foi adicionada). Tanto `sort` quanto `sortBy` são argumentos opcionais, assim seus defaults serão `DESCENDING` e `created` caso os argumentos não sejam especificados.

Os clientes agora podem controlar como suas fotos serão ordenadas quando executarem uma consulta `allPhotos`:

```

query {
  allPhotos(sortBy: name)
}

```

Essa consulta devolverá todas as fotos em ordem decrescente dos nomes.

Até agora, adicionamos argumentos somente em campos do tipo `Query`, mas é importante observar que podemos fazer isso em qualquer campo. É possível adicionar argumentos para filtragem, ordenação e paginação das fotos postadas por um único usuário:

```

type User {

```

```
postedPhotos(  
  first: Int = 25  
  start: Int = 0  
  sort: SortDirection = DESCENDING  
  sortBy: SortablePhotoField = created  
  category: PhotoCategory  
): [Photo!]
```

A adição de filtros de paginação pode ajudar a reduzir o volume de dados que uma consulta pode devolver. Discutiremos a ideia de limitar dados com mais detalhes no Capítulo 7.

Mutações

As mutações devem ser definidas no esquema. Assim como as consultas, as mutações também são definidas com seus próprios tipos objeto personalizados, e são adicionadas no esquema. Tecnicamente, não há diferença entre o modo de definir uma mutação ou uma consulta em seu esquema. A diferença está na intenção. Devemos criar mutações somente quando uma ação ou evento mudar algo que diz respeito ao estado de nossa aplicação.

As mutações devem representar os *verbos* em sua aplicação. Devem incluir aquilo que os usuários poderão *fazer* com o seu serviço. Ao fazer o design de seu serviço GraphQL, defina uma lista com todas as ações que um usuário poderá executar com a sua aplicação. Essas, provavelmente, serão as suas mutações.

Na aplicação PhotoShare, os usuários podem fazer login com o GitHub, postar fotos e marcá-las. Todas essas ações alteram algum aspecto do estado da aplicação. Depois que tiverem feito login com o GitHub, os usuários atuais que estarão acessando o cliente mudarão. Quando um usuário postar uma foto, haverá uma foto adicional no sistema. O mesmo vale para marcar fotos. Novos registros com dados de marcações em fotos serão gerados sempre que houver uma marcação em uma foto.

Podemos adicionar essas mutações no nosso esquema em nosso tipo mutação-raiz e disponibilizá-las aos clientes. Vamos começar

com nossa primeira mutação, `postPhoto`:

```
type Mutation {  
  postPhoto(  
    name: String!  
    description: String  
    category: PhotoCategory=PORTRAIT  
  ): Photo!  
}  
  
schema {  
  query: Query  
  mutation: Mutation  
}
```

Acrescentar um campo chamado `postPhoto` no tipo `Mutation` possibilita aos usuários postar fotos. Bem, no mínimo permite que os usuários postem metadados sobre as fotos. Cuidaremos do upload propriamente dito das fotos no Capítulo 7.

Quando um usuário posta uma foto, no mínimo, o nome (`name`) da foto será necessário. Os campos `description` e `category` são opcionais. Se um argumento `category` não for especificado, a foto postada terá `PORTRAIT` como default. Por exemplo, um usuário pode postar uma foto enviando a seguinte mutação:

```
mutation {  
  postPhoto(name: "Sending the Palisades") {  
    id  
    url  
    created  
    postedBy {  
      name  
    }  
  }  
}
```

Depois que o usuário postar uma foto, ele poderá selecionar informações sobre a foto que acabou de postar. Isso é conveniente porque alguns dos detalhes do registro da nova foto serão gerados no servidor. O `ID` da nova foto será criado pelo banco de dados. O `url` da foto será gerado automaticamente. A foto também receberá um

timestamp com a data e a hora em que ela foi criada (`created`). Essa consulta seleciona todos esses campos novos depois que a foto foi postada.

Além do mais, o conjunto de seleção inclui informações sobre o usuário que postou a foto. Um usuário deve estar logado para postar uma foto. Se nenhum usuário estiver logado no momento, essa mutação deve devolver um erro. Supondo que um usuário esteja logado, podemos obter detalhes sobre quem postou a foto por meio do campo `postedBy`. No Capítulo 5, discutiremos como autenticar um usuário autorizado usando um token de acesso.

Variáveis de mutação

Quando usamos mutações, com frequência é uma boa ideia declarar variáveis de mutação, como fizemos antes no Capítulo 3. Isso deixa a sua mutação reutilizável quando muitos usuários são criados. Também deixa você preparado para usar essa mutação em um cliente de verdade. Por questões de concisão, omitiremos esse passo no restante do capítulo, mas eis a aparência do código:

```
mutation postPhoto(
  $name: String!
  $description: String
  $category: PhotoCategory
){
  postPhoto(
    name: $name
    description: $description
    category: $category
  ){
    id
    name
    email
  }
}
```

Tipos input

Como você deve ter percebido, os argumentos de algumas de nossas consultas e mutações estão se tornando um tanto quanto extensos. Há uma maneira melhor de organizar esses argumentos usando *tipos input* (tipos de entrada). Um tipo input é semelhante ao

tipo objeto do GraphQL, exceto pelo fato de ser usado somente para argumentos de entrada.

Vamos melhorar a mutação `postPhoto` usando um tipo `input` em nossos argumentos:

```
input PostPhotoInput {  
  name: String!  
  description: String  
  category: PhotoCategory=PORTRAIT  
}  
  
type Mutation {  
  postPhoto(input: PostPhotoInput!): Photo!  
}
```

O tipo `PostPhotoInput` é como um tipo objeto, mas foi criado somente para argumentos de entrada. Ele exige `name`, mas os campos `description` e `category` continuam sendo opcionais. A partir de agora, ao enviar a mutação `postPhoto`, os detalhes sobre a nova foto devem ser incluídos em um objeto:

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

Quando criamos essa mutação, definimos o tipo da variável de consulta `$input` para que coincida com o nosso tipo `input PostPhotoInput!`. É não nullable porque, no mínimo, precisamos acessar o campo `input.name` para adicionar uma nova foto. Ao enviar a mutação, devemos fornecer os dados da nova foto em nossas variáveis de consulta, aninhados no campo `input`:

```
{  
  "input": {  
    "name": "Hanging at the Arc",  
    "description": "Sunny on the deck of the Arc",  
    "category": "LANDSCAPE"  
  }  
}
```

```
}
```

Nossa entrada é agrupada em um objeto JSON e é enviada com a mutação nas variáveis de consulta, com a chave “input”. Como as variáveis de consulta são formatadas como JSON, a categoria deve ser uma string que coincida com uma das categorias do tipo `PhotoCategory`.

Os tipos input são essenciais para organizar e escrever um esquema GraphQL claro. Podemos usar tipos input como argumentos em qualquer campo. Eles podem ser usados para melhorar tanto a paginação quanto a filtragem dos dados nas aplicações.

Vamos ver como podemos organizar e reutilizar todos os nossos campos de ordenação e filtragem usando tipos input:

```
input PhotoFilter {
  category: PhotoCategory
  createdBetween: DateRange
  taggedUsers: [ID!]
  searchText: String
}

input DateRange {
  start: DateTime!
  end: DateTime!
}

input DataPage {
  first: Int = 25
  start: Int = 0
}

input DataSort {
  sort: SortDirection = DESCENDING
  sortBy: SortablePhotoField = created
}

type User {
  ...
  postedPhotos(filter:PhotoFilter paging:DataPage sorting:DataSort): [Photo!]!
  inPhotos(filter:PhotoFilter paging:DataPage sorting:DataSort): [Photo!]!
```

```

}

type Photo {
  ...
  taggedUsers(sorting:DataSort): [User!]!
}

type Query {
  ...
  allUsers(paging:DataPage sorting:DataSort): [User!]!
  allPhotos(filter:PhotoFilter paging:DataPage sorting:DataSort): [Photo!]!
}

```

Organizamos diversos campos em tipos input e reutilizamos esses campos como argumentos em nosso esquema.

O tipo input `PhotoFilter` contém campos de entrada opcionais que permitem que o cliente filtre uma lista de fotos. Esse tipo inclui outro tipo input aninhado, `DateRange`, no campo `createdBetween`. `DateRange` deve incluir datas de início e de fim. Ao usar `PhotoFilter`, podemos também filtrar fotos por `category`, `search` (string de busca) ou `taggedUsers`. Adicionamos todas essas opções de filtro em todos os campos que devolvam uma lista de fotos. Isso dá ao cliente bastante controle sobre quais fotos serão devolvidas em cada lista.

Tipos input também foram criados para paginação e ordenação. O tipo input `DataPage` contém os campos necessários para requisitar uma página de dados, enquanto o tipo `DataSort` contém os campos para ordenação. Esses tipos input foram adicionados em todos os campos de nosso esquema que devolvam uma lista de dados.

Podemos escrever uma consulta que aceite alguns dados de entrada bem complexos usando os tipos input disponíveis:

```

query getPhotos($filter:PhotoFilter $page:DataPage $sort:DataSort) {
  allPhotos(filter:$filter paging:$page sorting:$sort) {
    id
    name
    url
  }
}

```

Essa consulta aceita opcionalmente argumentos para três tipos

input: \$filter, \$page e \$sort. Usando variáveis de consulta, podemos enviar alguns detalhes específicos sobre as fotos que gostaríamos de receber:

```
{
  "filter": {
    "category": "ACTION",
    "taggedUsers": ["MoonTahoe", "EvePorcello"],
    "createdBetween": {
      "start": "2018-11-6",
      "end": "2018-5-31"
    }
  },
  "page": {
    "first": 100
  }
}
```

Essa consulta encontrará todas as fotos ACTION em que os usuários MoonTahoe e EvePorcello do GitHub foram marcados entre 6 de novembro e 31 de maio, que, por acaso, é a temporada de esqui. Também pedimos as 100 primeiras fotos com essa consulta.

Os tipos input nos ajudam a organizar nosso esquema e a reutilizar argumentos. Também melhoram a documentação do esquema gerado automaticamente pelo GraphiQL ou pelo GraphQL Playground. Isso deixará nossa API mais acessível e mais fácil de ser compreendida e digerida. Por fim, podemos usar os tipos input para que um cliente tenha mais eficácia na execução de consultas organizadas.

Tipo de retorno

Todos os campos de nosso esquema têm devolvido nossos tipos principais, User e Photo. Às vezes, porém, teremos que devolver informações meta sobre consultas e mutações, além dos dados de payload propriamente ditos. Por exemplo, quando um usuário estiver logado e tiver sido autenticado, devemos devolver um token, além do payload com User.

Para fazer login com o GitHub OAuth, devemos obter um código OAuth do GitHub. Discutiremos como configurar nossa própria conta GitHub OAuth e obter o código do GitHub na seção “Autorização do GitHub”. Por enquanto, suponhamos que você tem um código GitHub válido que pode ser enviado na mutação `githubAuth` para login de um usuário:

```
type AuthPayload {  
  user: User!  
  token: String!  
}  
  
type Mutation {  
  ...  
  githubAuth(code: String!): AuthPayload!  
}
```

Os usuários são autenticados enviando um código GitHub válido para a mutação `githubAuth`. Se essa requisição for bem-sucedida, devolveremos um tipo objeto personalizado contendo tanto as informações do usuário cujo login foi feito com sucesso como também um token que poderá ser usado para autorização de outras consultas e mutações, incluindo a mutação `postPhoto`.

Você pode usar tipos de retorno personalizados em qualquer campo para o qual precisar de informações extras além dos dados do payload. Podemos querer saber quanto tempo demora para uma consulta entregar uma resposta ou quantos resultados foram encontrados em uma resposta particular, além dos dados de payload da consulta. Tudo isso pode ser tratado com o uso de um tipo de retorno personalizado.

A essa altura, já apresentamos todos os tipos disponíveis a você para criar esquemas GraphQL. Dedicamos até mesmo um pouco de tempo para discutir técnicas que podem ajudar você a melhorar o design de seu esquema. Contudo, há um último tipo objeto-raiz que devemos apresentar: é o tipo `Subscription`.

Subscriptions

Os tipos `Subscription` não são diferentes de outros tipos objeto na linguagem de definição de esquema do GraphQL. Nesse caso, definimos as subscriptions (inscrições) disponíveis como campos de um tipo objeto personalizado. Caberá a nós garantir que as subscriptions implementem o padrão de projeto (design pattern) PubSub, junto com algum tipo de transporte de tempo real quando construirmos o serviço GraphQL mais tarde no Capítulo 7.

Por exemplo, podemos adicionar subscriptions que permitam que nossos clientes ouçam a criação de novos tipos `Photo` ou `User`:

```
type Subscription {
  newPhoto: Photo!
  newUser: User!
}

schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}
```

Nesse exemplo, criamos um objeto `Subscription` personalizado contendo dois campos: `newPhoto` e `newUser`. Quando uma nova foto é postada, essa foto será enviada a todos os clientes que tenham se inscrito na subscription `newPhoto`. Quando um novo usuário for criado, seus detalhes serão enviados a todos os clientes que estiverem ouvindo novos usuários.

Assim como as consultas ou as mutações, as subscriptions podem tirar proveito dos argumentos. Suponha que quiséssemos adicionar filtros na subscription `newPhoto` para ouvir somente novas fotos `ACTION`:

```
type Subscription {
  newPhoto(category: PhotoCategory): Photo!
  newUser: User!
}
```

Quando os usuários se inscreverem na subscription `newPhoto`, eles agora terão a opção de filtrar as fotos que serão enviadas a essa subscription. Por exemplo, para filtrar somente novas fotos `ACTION`,

os clientes poderão enviar a seguinte operação para a nossa API GraphQL:

```
subscription {
  newPhoto(category: "ACTION") {
    id
    name
    url
    postedBy {
      name
    }
  }
}
```

Essa subscription deve devolver detalhes somente de fotos ACTION.

Uma subscription é uma ótima solução quando for importante tratar dados em tempo real. No Capítulo 7, discutiremos melhor a implementação das subscriptions para todos os seus requisitos de tratamento de dados em tempo real.

Documentação dos esquemas

No Capítulo 3 explicamos como o GraphQL tem um sistema de introspecção capaz de dar informações sobre as consultas aceitas pelo servidor. Ao escrever um esquema GraphQL, podemos adicionar descrições opcionais para cada campo, as quais fornecerão informações extras sobre os tipos e campos do esquema. Prover descrições pode fazer com que seja mais fácil para a sua equipe, para você mesmo e para outros usuários compreenderem o seu sistema de tipos.

Por exemplo, vamos adicionar comentários no tipo User em nosso esquema:

```
"""
A user who has been authorized by GitHub at least once
"""
type User {
  """
  The user's unique GitHub login
  """
  login: String!
```

```

    """
    githubLogin: ID!

    """
    The user's first and last name
    """
    name: String

    """
    A url for the user's GitHub profile photo
    """
    avatar: String

    """
    All of the photos posted by this user
    """
    postedPhotos: [Photo!]!

    """
    All of the photos in which this user appears
    """
    inPhotos: [Photo!]!
}

```

Ao adicionar três aspas antes e depois de seu comentário em cada tipo ou campo, você fornecerá aos usuários um dicionário de sua API. Além dos tipos e dos campos, os argumentos também podem ser documentados. Observe a mutação `postPhoto`:

Replace with:

```

type Mutation {
    """
    Authorizes a GitHub User
    """
    githubAuth(
        "The unique code from GitHub that is sent to authorize the user"
        code: String!
    ): AuthPayload!
}

```

Os comentários dos argumentos compartilham o nome do argumento e o fato de o campo ser opcional. Se você estiver usando

tipos input, poderá documentá-los como faz com qualquer outro tipo:

```
""
The inputs sent with the postPhoto Mutation
""

input PostPhotoInput {
  "The name of the new photo"
  name: String!
  "(optional) A brief description of the photo"
  description: String
  "(optional) The category that defines the photo"
  category: PhotoCategory=PORTRAIT
}

postPhoto(
  "input: The name, description, and category for a new photo"
  input: PostPhotoInput!
): Photo!
```

Todas essas notas sobre documentação são então listadas na documentação do esquema no GraphQL Playground ou no GraphiQL, como vemos na Figura 4.4. É claro que você também pode executar consultas de introspecção para ver as descrições desses tipos.

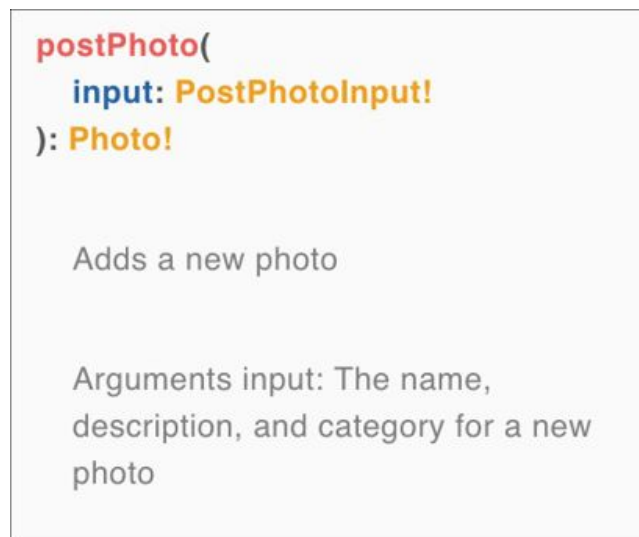


Figure 4.4 – Documentação de postPhoto.

No coração de todos os projetos GraphQL, encontra-se um esquema sólido e bem definido. Ele serve como um guia e um

contrato entre as equipes de frontend e de backend, com o intuito de garantir que o produto implementado sempre servirá o esquema.

Neste capítulo criamos um esquema para nossa aplicação de compartilhamento de fotos. Nos próximos três capítulos mostraremos como implementar uma aplicação GraphQL full-stack que obedeça ao contrato do esquema que acabamos de criar.

CAPÍTULO 5

Criando uma API GraphQL

Você explorou a história. Escreveu algumas consultas. Criou um esquema. Agora está pronto para criar um serviço GraphQL totalmente funcional. Isso pode ser feito com uma variedade de tecnologias, mas usaremos JavaScript. As técnicas compartilhadas neste capítulo são razoavelmente universais, portanto, mesmo que os detalhes de implementação sejam diferentes, a arquitetura geral será semelhante, não importa a linguagem ou o framework que você escolher.

Se estiver interessado em bibliotecas de servidor para outras linguagens, consulte as várias bibliotecas existentes em GraphQL.org (<http://graphql.org/code/>).

Quando a especificação do GraphQL foi lançada em 2015, seu foco era apresentar uma explicação clara da linguagem de consulta e do sistema de tipos. Ela deixou os detalhes sobre a implementação do servidor propositalmente mais vagos para permitir que desenvolvedores com experiências anteriores em linguagens variadas usassem o que lhes deixasse mais à vontade. A equipe do Facebook, porém, disponibilizou uma implementação de referência em JavaScript chamada GraphQL.js. Junto com essa implementação eles lançaram o *express-graphql*, uma maneira simples de criar um servidor GraphQL com o Express e, notadamente, é a primeira biblioteca a ajudar os desenvolvedores a realizar essa tarefa.

Depois de termos explorado as implementações JavaScript de servidores GraphQL, optamos pelo uso do Apollo Server (<https://www.apollographql.com/docs/apollo-server/v2/>), uma solução

de código aberto da equipe do Apollo. O Apollo Server é razoavelmente simples de configurar e oferece um conjunto de funcionalidades prontas para um ambiente de produção, incluindo suporte a subscriptions, upload de arquivos, uma API como fonte de dados para engatar rapidamente serviços existentes e integração imediata com a Apollo Engine. Inclui também o GraphQL Playground para escrever consultas diretamente no navegador.

Configuração do projeto

Vamos começar criando o projeto `photo-share-api` como uma pasta vazia em nosso computador. Lembre-se de que você sempre pode acessar o repositório deste livro (<https://83github.com/MoonHighway/learning-graphql/tree/master/chapter-05/photo-share-api/>) para ver o projeto completo ou vê-lo executando no Glitch. Nessa pasta, geraremos um novo projeto npm usando o comando `npm init -y` no Terminal ou no Prompt de Comandos. Esse utilitário gerará um arquivo `package.json` e definirá todas as opções como default, pois estamos usando a flag `-y`.

Em seguida, instalaremos as dependências do projeto: `apollo-server` e `graphql`. Instalaremos também o `nodemon`:

```
npm install apollo-server graphql nodemon
```

`apollo-server` e `graphql` são necessários para configurar uma instância do Apollo Server. O `nodemon` observará se houve mudanças nos arquivos e reiniciará o servidor quando elas ocorrerem. Desse modo, não teremos que parar e reiniciar o servidor sempre que uma alteração for feita. Adicionaremos o comando para o `nodemon` em `package.json`, na chave `scripts`:

```
"scripts": {  
  "start": "nodemon -e js,json,graphql"  
}
```

A partir de agora, sempre que executarmos `npm start`, nosso arquivo `index.js` executará e o `nodemon` ficará observando se houve mudanças

em qualquer arquivo cuja extensão seja `js`, `json` ou `graphql`. Além disso, devemos criar um arquivo `index.js` na raiz do projeto. Lembre-se de que o arquivo `main` em `package.json` aponta para `index.js`:

```
"main": "index.js"
```

Resolvers

Em nossa discussão sobre o GraphQL até agora mantivemos o foco em muitas consultas. Um esquema define as operações de consulta que os clientes poderão fazer e como os diferentes tipos estão relacionados. Ele descreve os requisitos de dados, mas não faz o trabalho de obtenção desses dados. Esse trabalho é feito pelos resolvers.

Um *resolver* é uma função que devolve dados para um campo em particular. As funções resolver devolvem dados no tipo e formato especificados pelo esquema. Os resolvers podem ser assíncronos e podem buscar ou atualizar dados de uma API REST, um banco de dados ou qualquer outro serviço.

Vamos observar a possível aparência de um resolver para nossa consulta-raiz. Em nosso arquivo `index.js` na raiz do projeto, adicionaremos o campo `totalPhotos` em `Query`:

```
const typeDefs = `
  type Query {
    totalPhotos: Int!
  }
`

const resolvers = {
  Query: {
    totalPhotos: () => 42
  }
}
```

A variável `typeDefs` serve para definir o nosso esquema. É somente uma string. Sempre que criarmos uma consulta como `totalPhotos`, ela deverá ter o suporte de uma função resolver de mesmo nome. A definição de tipo descreve o tipo que o campo deve devolver. A

função resolver devolve os dados desse tipo de algum lugar – nesse caso, é somente um valor estático igual a 42.

Também é importante observar que o resolver deve estar definido em um objeto com o mesmo `typename` do objeto no esquema. O campo `totalPhotos` faz parte do objeto de consulta. O resolver para esse campo também deve fazer parte do objeto `Query`.

Criamos as definições iniciais de tipo para a nossa consulta-raiz. Também criamos nosso primeiro resolver que dá suporte ao campo de consulta `totalPhotos`. Para criar o esquema e permitir a execução de consultas, usaremos o `Apollo Server`:

```
// 1. Require do 'apollo-server'
const { ApolloServer } = require('apollo-server')

const typeDefs = `
  type Query {
    totalPhotos: Int!
  }
`

const resolvers = {
  Query: {
    totalPhotos: () => 42
  }
}

// 2. Cria uma nova instância do servidor
// 3. Envia-lhe um objeto com typeDefs (o esquema) e resolvers
const server = new ApolloServer({
  typeDefs,
  resolvers
})

// 4. Chama listen no servidor para iniciar o servidor web
server
  .listen()
  .then(({url}) => console.log(`GraphQL Service running on ${url}`))
```

Após exigir o `ApolloServer`, criamos uma nova instância do servidor, enviando-lhe um objeto com dois valores: `typeDefs` e `resolvers`. Essa é

uma configuração mínima e rápida de um servidor, mas que nos permitirá ter uma API GraphQL robusta. Mais adiante no capítulo discutiremos como estender as funcionalidades do servidor usando o Express.

Agora, estamos prontos para executar uma consulta a `totalPhotos`. Depois de executar `npm start`, veremos o GraphQL Playground executando em `http://localhost:4000`. Vamos testar a consulta a seguir:

```
{
  totalPhotos
}
```

O dado devolvido para `totalPhotos` é 42, conforme esperado:

```
{
  "data": {
    "totalPhotos": 42
  }
}
```

Os resolvers são essenciais na implementação do GraphQL. Todo campo deve ter uma função resolver correspondente. O resolver deve obedecer às regras do esquema. Deve ter o mesmo nome que o campo definido no esquema e deve devolver o tipo de dado definido lá.

Resolvers raiz

Conforme discutimos no Capítulo 4, as APIs GraphQL têm tipos raiz para Query, Mutation e Subscription. Esses tipos se encontram no nível superior e representam todos os possíveis pontos de entrada na API. Até agora, adicionamos o campo `totalPhotos` no tipo Query, o que significa que nossa API pode consultar esse campo.

Vamos expandir isso criando um tipo raiz para Mutation. O campo da mutação se chama `postPhoto` e aceitará `name` e `description` como argumentos do tipo String. Quando a mutação for enviada, ela deverá devolver um Boolean:

```
const typeDefs = `
```

```

type Query {
  totalPhotos: Int!
}

type Mutation {
  postPhoto(name: String! description: String): Boolean!
}

```

Depois de criada a mutação `postPhoto`, devemos acrescentar um resolver correspondente no objeto `resolvers`:

```

// 1. Um tipo de dado para armazenar nossas fotos na memória
var photos = []

const resolvers = {
  Query: {

    // 2. Devolve o tamanho do array de fotos
    totalPhotos: () => photos.length

  },

  // 3. A mutação e o resolver postPhoto
  Mutation: {
    postPhoto(parent, args) {
      photos.push(args)
      return true
    }
  }
}

```

Inicialmente temos que criar uma variável chamada `photos` para armazenar os detalhes das fotos em um array. Ainda neste capítulo armazenaremos as fotos em um banco de dados.

Em seguida, melhoramos o resolver `totalPhotos` de modo que devolva o tamanho do array `photos`. Sempre que for consultado, esse campo devolverá o número de fotos armazenadas no array no momento.

Em seguida, adicionamos o resolver `postPhoto`. Dessa vez estamos usando argumentos com nossa função `postPhoto`. O primeiro argumento é uma referência ao objeto-pai. Às vezes, você verá isso

representado como `_`, `root` ou `obj` na documentação. Nesse caso, o pai do resolver `postPhoto` é uma `Mutation`. O pai não contém nenhum dado que precisamos usar no momento, mas será sempre o primeiro argumento enviado a um resolver. Assim, é necessário adicionar um argumento placeholder `parent` para que possamos acessar o segundo argumento enviado ao resolver: os argumentos da mutação.

O segundo argumento enviado ao resolver `postPhoto` contém os argumentos de GraphQL que foram enviados para essa operação: o `name` e, opcionalmente, a `description`. A variável `args` é um objeto que contém esses dois campos: `{name,description}`. No momento, os argumentos representam um objeto foto, portanto nós os enviaremos diretamente para o array `photos`.

É hora de testar a mutação `postPhoto` no GraphQL Playground, enviando uma string para o argumento `name`:

```
mutation newPhoto {  
  postPhoto(name: "sample photo")  
}
```

Essa mutação adiciona os detalhes da foto no array e devolve `true`. Vamos modificar essa mutação para que use variáveis de consulta:

```
mutation newPhoto($name: String!, $description: String) {  
  postPhoto(name: $name, description: $description)  
}
```

Depois que as variáveis são adicionadas à mutação, dados deverão ser passados para fornecer as variáveis de string. No canto inferior à esquerda do Playground, adicione valores para `name` e `description` na janela Query Variables (Variáveis de consulta):

```
{  
  "name": "sample photo A",  
  "description": "A sample photo for our dataset"  
}
```

Resolvers de tipo

Quando uma consulta, mutação ou subscription GraphQL é

executada, ela devolverá um resultado cujo formato é o mesmo da consulta. Vimos como os resolvers podem devolver valores de tipo escalar como inteiros, strings e booleanos, mas os resolvers também podem devolver objetos.

Em nossa aplicação de fotos, criaremos um tipo `Photo` e um campo de consulta `allPhotos` que devolverá uma lista de objetos `Photo`:

```
const typeDefs = `

# 1. Acrescenta a definição do tipo Photo
type Photo {
  id: ID!
  url: String!
  name: String!
  description: String
}

# 2. Devolve Photo em allPhotos
type Query {
  totalPhotos: Int!
  allPhotos: [Photo!]!
}

# 3. Devolve a foto recém-postada da mutação
type Mutation {
  postPhoto(name: String! description: String): Photo!
}
```

Como adicionamos o objeto `Photo` e a consulta `allPhotos` em nossas definições de tipos, é necessário que esses ajustes se reflitam nos resolvers. A mutação `postPhoto` deve devolver dados no formato do tipo `Photo`. A consulta `allPhotos` deve devolver uma lista de objetos com o mesmo formato do tipo `Photo`:

```
// 1. Uma variável que será incrementada para ids únicos
var _id = 0
var photos = []

const resolvers = {
  Query: {
    totalPhotos: () => photos.length,
```

```

    allPhotos: () => photos
  },
  Mutation: {
    postPhoto(parent, args) {

      // 2. Cria uma nova foto e gera um id
      var newPhoto = {
        id: _id++,
        ...args
      }
      photos.push(newPhoto)

      // 3. Devolve a nova foto
      return newPhoto
    }
  }
}

```

Como o tipo `Photo` exige um ID, criamos uma variável para armazená-lo. No resolver `postPhoto`, geramos IDs incrementando esse valor. A variável `args` fornece os campos `name` e `description` para a foto, mas precisamos também de um ID. Em geral, cabe ao servidor criar variáveis como identificadores e timestamps. Assim, quando criamos um objeto foto no resolver `postPhoto`, adicionamos o campo ID e atribuímos os campos `name` e `description` de `args` ao nosso novo objeto foto.

Em vez de devolver um booleano, a mutação devolve um objeto cujo formato corresponde ao formato do tipo `Photo`. Esse objeto é construído com o ID gerado e os campos de nome e descrição que foram passados com `data`. Além do mais, a mutação `postPhoto` adiciona objetos foto no array `photos`. Esses objetos têm o mesmo formato do tipo `Photo` definido em nosso esquema, portanto podemos devolver o array completo `photos` na consulta `allPhotos`.



Gerar IDs únicos com uma variável incrementada é, claramente, uma maneira não escalável de criar IDs, mas servirá aos nossos propósitos de demonstração neste caso. Em uma aplicação real, seu ID provavelmente será gerado pelo banco de dados.

Para conferir se `postPhoto` está funcionando corretamente, podemos ajustar a mutação. Como `Photo` é um tipo, devemos acrescentar um conjunto de seleção à nossa mutação:

```
mutation newPhoto($name: String!, $description: String) {  
  postPhoto(name: $name, description: $description) {  
    id  
    name  
    description  
  }  
}
```

Depois de adicionar algumas fotos usando mutações, a consulta `allPhotos` a seguir deve devolver um array com todos os objetos `Photo` adicionados:

```
query listPhotos {  
  allPhotos {  
    id  
    name  
    description  
  }  
}
```

Acrescentamos também um campo `url` não nullable em nosso esquema de fotos. O que acontecerá se adicionarmos uma `url` em nosso conjunto de seleção?

```
query listPhotos {  
  allPhotos {  
    id  
    name  
    description  
    url  
  }  
}
```

Quando `url` é adicionada no conjunto de seleção de nossa consulta, um erro será exibido: `Cannot return null for non-nullable field Photo.url` (Não é possível devolver null para o campo `Photo.url` não nullable). Não adicionamos um campo `url` no conjunto de dados. Não precisamos armazenar as URLs porque elas podem ser geradas automaticamente. Cada campo de nosso esquema pode ser

mapeado para um resolver. Tudo que temos que fazer é adicionar um objeto `Photo` em nossa lista de resolvers e definir os campos que queremos mapear para funções. Nesse caso, queremos usar uma função para nos ajudar a resolver as URLs:

```
const resolvers = {  
  Query: { ... },  
  Mutation: { ... },  
  Photo: {  
    url: parent => `http://yoursite.com/img/${parent.id}.jpg`  
  }  
}
```

Como usaremos um resolver para as URLs das fotos, adicionamos um objeto `Photo` em nossos resolvers. Esse resolver de `Photo` adicionado na raiz se chama *resolver trivial*. Resolvers triviais são adicionados no nível superior do objeto `resolvers`, mas não são obrigatórios. Temos a opção de criar resolvers personalizados para o objeto `Photo` usando um resolver trivial. Se você não especificar um resolver trivial, o GraphQL recorrerá a um resolver default que devolve uma propriedade de mesmo nome que o campo.

Quando selecionamos a `url` de uma foto em nossa consulta, a função resolver correspondente é chamada. O primeiro argumento enviado aos resolvers é sempre o objeto `parent`. Nesse caso, `parent` representa o objeto `Photo` atual sendo resolvido. Estamos supondo, nesse caso, que nosso serviço trata somente imagens JPEG. Essas imagens são nomeadas de acordo com o ID da foto e podem ser encontradas na rota `http://yoursite.com/img/`. Como o `parent` é a foto, podemos obter seu ID por meio desse argumento e usá-lo para gerar automaticamente um URL para a foto atual.

Quando definimos um esquema GraphQL, descrevemos os requisitos de dados de nossa aplicação. Com os resolvers, podemos atender a esses requisitos de forma eficaz e flexível. As funções nos proporcionam essa capacidade e flexibilidade. Elas podem ser assíncronas, devolver tipos escalares e objetos, além de devolver dados de várias fontes. Os resolvers são apenas funções, e qualquer campo de nosso esquema GraphQL pode ser mapeado

para um resolver.

Usando entradas e enumerados

É hora de introduzir um tipo enumerado, `PhotoCategory`, e um tipo input (tipo de entrada), `PostPhotoInput`, em nosso `typeDefs`:

```
enum PhotoCategory {  
  SELFIE  
  PORTRAIT  
  ACTION  
  LANDSCAPE  
  GRAPHIC  
}  
  
type Photo {  
  ...  
  category: PhotoCategory!  
}  
  
input PostPhotoInput {  
  name: String!  
  category: PhotoCategory=PORTRAIT  
  description: String  
}  
  
type Mutation {  
  postPhoto(input: PostPhotoInput!): Photo!  
}
```

No Capítulo 4 criamos esses tipos quando fizemos o design do esquema para a aplicação PhotoShare. Também adicionamos o tipo enumerado `PhotoCategory` e um campo `category` em nossas fotos. Ao resolver as fotos, devemos garantir que a categoria delas – uma string que corresponde aos valores definidos no tipo enumerado – esteja disponível. Também temos que obter uma categoria quando os usuários postam novas fotos.

Acrescentamos um tipo `PostPhotoInput` para organizar o argumento da mutação `postPhoto` em um único objeto. Esse tipo input tem um campo para categoria. Se um usuário não fornecer um campo de categoria

como argumento, o default, `PORTRAIT`, será usado.

Para o resolver `postPhoto` é necessário fazer alguns ajustes também. Os detalhes da foto, `name`, `description` e `category`, agora estão aninhados no campo `input`. Temos que garantir que acessaremos esses valores em `args.input`, e não em `args`:

```
postPhoto(parent, args) {  
  var newPhoto = {  
    id: _id++,  
    ...args.input  
  }  
  photos.push(newPhoto)  
  return newPhoto  
}
```

Agora executamos a mutação com o novo tipo `input`:

```
mutation newPhoto($input: PostPhotoInput!) {  
  postPhoto(input:$input) {  
    id  
    name  
    url  
    description  
    category  
  }  
}
```

Também temos que enviar o JSON correspondente no painel `Query Variables` (Variáveis de consulta):

```
{  
  "input": {  
    "name": "sample photo A",  
    "description": "A sample photo for our dataset"  
  }  
}
```

Se a categoria não for especificada, o default usado será `PORTRAIT`. De modo alternativo, se um valor for fornecido para `category`, ele será validado em relação ao nosso tipo enumerado mesmo antes de a operação ser enviada para o servidor. Se a categoria for válida, ela será passada para o resolver como argumento.

Podemos fazer com que a passagem de argumentos para as

mutações seja mais reutilizável e menos suscetível a erros com os tipos `input`. Podemos ser mais específicos quanto aos tipos das entradas que podem ser fornecidas em campos específicos quando combinamos tipos `input` com enumerados. Tipos `input` e enumerados são extremamente importantes e melhores ainda quando usados em conjunto.

Arestas e conexões

Conforme já discutimos, a eficácia do GraphQL se deve às arestas: as conexões entre os pontos de dados. Quando criamos um servidor GraphQL, os tipos, em geral, são mapeados a modelos. Pense nesses tipos como sendo salvos em tabelas de dados semelhantes. A partir daí, ligamos os tipos usando conexões. Exploraremos os tipos de conexões que podem ser usados para definir os relacionamentos de interconexão entre os tipos.

Conexões de um para muitos

Os usuários precisam acessar a lista de fotos que postaram antes. Acessaremos esses dados em um campo chamado `postedPhotos`, que será resolvido como uma lista filtrada de fotos que o usuário postou. Como um `User` pode postar muitas `Photos`, denominamos isso de *relacionamento de um para muitos*. Vamos adicionar `User` em nosso `typeDefs`:

```
type User {  
  githubLogin: ID!  
  name: String  
  avatar: String  
  postedPhotos: [Photo!]!  
}
```

A essa altura, criamos um grafo direcionado. Podemos fazer o percurso do tipo `User` para o tipo `Photo`. Para ter um grafo não direcionado, devemos prover uma forma de voltar ao tipo `User` a partir do tipo `Photo`. Vamos adicionar um campo `postedBy` no tipo `Photo`:

```
type Photo {  
  id: ID!
```



```
url: String!  
name: String!  
description: String  
category: PhotoCategory!  
postedBy: User!  
}
```

Ao adicionar o campo `postedBy`, criamos uma ligação de volta ao `User` que postou a `Photo`, criando um grafo não direcionado. Essa é uma *conexão de um para um*, pois uma foto só pode ser postada por um `User`.

Usuários de exemplo

Para testar o nosso servidor, vamos adicionar alguns dados de exemplo em nosso arquivo `index.js`. Lembre-se de remover a variável `photos`, que está definida atualmente como um array vazio:

```
var users = [  
  { "githubLogin": "mHattrup", "name": "Mike Hattrup" },  
  { "githubLogin": "gPlake", "name": "Glen Plake" },  
  { "githubLogin": "sSchmidt", "name": "Scot Schmidt" }  
]  
  
var photos = [  
  {  
    "id": "1",  
    "name": "Dropping the Heart Chute",  
    "description": "The heart chute is one of my favorite chutes",  
    "category": "ACTION",  
    "githubUser": "gPlake"  
  },  
  {  
    "id": "2",  
    "name": "Enjoying the sunshine",  
    "category": "SELFIE",  
    "githubUser": "sSchmidt"  
  },  
  {  
    id: "3",  
    "name": "Gunbarrel 25",  
    "description": "25 laps on gunbarrel today",  
    "category": "LANDSCAPE",  
    "githubUser": "sSchmidt"  
  }  
]
```

]

Como as conexões são criadas usando os campos de um tipo objeto, elas podem ser mapeadas para funções resolver. Nessas funções, podemos usar os detalhes sobre o pai para nos ajudar a localizar e a devolver os dados conectados.

Vamos adicionar os resolvers `postedPhotos` e `postedBy` em nosso serviço:

```
const resolvers = {  
  ...  
  Photo: {  
    url: parent => `http://yoursite.com/img/${parent.id}.jpg`,  
    postedBy: parent => {  
      return users.find(u => u.githubLogin === parent.githubUser)  
    }  
  },  
  User: {  
    postedPhotos: parent => {  
      return photos.filter(p => p.githubUser === parent.githubLogin)  
    }  
  }  
}
```

No resolver de `Photo` devemos acrescentar um campo para `postedBy`. Nesse resolver cabe a nós descobrir como encontrar os dados conectados. Usando o método de array `.find()` podemos obter o usuário cujo `githubLogin` coincida com o valor `githubUser` salvo em cada foto. O método `.find()` devolve um único objeto de usuário.

No resolver de `User` obtemos uma lista de fotos postadas por esse usuário com o método `.filter()` de array. Esse método devolve um array somente com as fotos que contenham um valor `githubUser` correspondente ao valor `githubLogin` do usuário-pai. O método de filtro devolve um array de fotos.

Vamos agora tentar enviar a consulta `allPhotos`:

```
query photos {  
  allPhotos {  
    name  
    url  
    postedBy {  
      name  
    }  
  }  
}
```

```
}  
}
```

Quando consultamos cada foto podemos ver o usuário que a postou. O objeto usuário está sendo localizado e devolvido pelo resolver. Nesse exemplo selecionamos somente o nome do usuário que postou a foto. Considerando nossos dados de exemplo, os dados JSON a seguir devem ser devolvidos como resultado:

```
{  
  "data": {  
    "allPhotos": [  
      {  
        "name": "Dropping the Heart Chute",  
        "url": "http://yoursite.com/img/1.jpg",  
        "postedBy": {  
          "name": "Glen Plake"  
        }  
      },  
      {  
        "name": "Enjoying the sunshine",  
        "url": "http://yoursite.com/img/2.jpg",  
        "postedBy": {  
          "name": "Scot Schmidt"  
        }  
      },  
      {  
        "name": "Gunbarrel 25",  
        "url": "http://yoursite.com/img/3.jpg",  
        "postedBy": {  
          "name": "Scot Schmidt"  
        }  
      }  
    ]  
  }  
}
```

Somos responsáveis por conectar os dados aos resolvers, mas, assim que pudermos devolver esses dados conectados, nossos clientes poderão começar a escrever consultas eficazes. Na próxima seção mostraremos algumas técnicas para criar conexões de muitos para muitos.

Muitos para muitos

A próxima funcionalidade que queremos adicionar em nosso serviço é a capacidade de marcar usuários nas fotos (atribuir tags). Isso significa que um `User` pode ser marcado em várias fotos distintas, e `Photo` pode ter muitos usuários diferentes marcados. O relacionamento que as marcações (tags) nas fotos criarão entre usuários e fotos pode ser referenciado como de *muitos para muitos* – muitos usuários para muitas fotos.

Para facilitar o relacionamento de muitos para muitos, adicionamos o campo `taggedUsers` em `Photo` e um campo `inPhotos` em `User`. Vamos modificar o `typeDefs`:

```
type User {  
  ...  
  inPhotos: [Photo!]!  
}  
  
type Photo {  
  ...  
  taggedUsers: [User!]!  
}
```

O campo `taggedUsers` devolve uma lista de usuários, e o campo `inPhotos` devolve uma lista de fotos em que um usuário aparece. Para facilitar essa conexão de muitos para muitos, adicionaremos um array de marcações. Para testar a funcionalidade de marcação, devemos preencher alguns dados com exemplos de marcações:

```
var tags = [  
  { "photoID": "1", "userID": "gPlake" },  
  { "photoID": "2", "userID": "sSchmidt" },  
  { "photoID": "2", "userID": "mHattrup" },  
  { "photoID": "2", "userID": "gPlake" }  
]
```

Quando tivermos uma foto, pesquisaremos nossos conjuntos de dados a fim de encontrar os usuários que foram marcados na foto. Quando tivermos um usuário, caberá a nós encontrar a lista de fotos em que esse usuário aparece. Como nossos dados estão atualmente armazenados em arrays JavaScript, usaremos métodos

de arrays nos resolvers para encontrar os dados:

```
Photo: {  
  ...  
  taggedUsers: parent => tags  
  
  // Devolve um array de marcações contendo somente a foto atual  
  .filter(tag => tag.photoID === parent.id)  
  
  // Converte o array de marcações em um array de userIDs  
  .map(tag => tag.userID)  
  
  // Converte o array de userIDs em um array de objetos de usuário  
  .map(userID => users.find(u => u.githubLogin === userID))  
},  
User: {  
  ...  
  inPhotos: parent => tags  
  
  // Devolve um array de marcações contendo somente o usuário atual  
  .filter(tag => tag.userID === parent.id)  
  
  // Converte o array de marcações em um array de photoIDs  
  .map(tag => tag.photoID)  
  
  // Converte o array de photoIDs em um array de objetos de foto  
  .map(photoID => photos.find(p => p.id === photoID))  
}
```

O resolver do campo `taggedUsers` filtra qualquer foto que não seja a foto atual e mapeia essa lista filtrada para um array de objetos `User`. O resolver do campo `inPhotos` filtra as marcações por usuário e mapeia as marcações do usuário para um array de objetos `Photo`.

Podemos ver agora quais usuários estão marcados em cada foto enviando uma consulta GraphQL:

```
query listPhotos {  
  allPhotos {  
    url  
    taggedUsers {  
      name  
    }  
  }  
}
```

```
}  
}
```

Talvez você tenha percebido que temos um array para `tags`, mas não temos um tipo GraphQL chamado `Tag`. O GraphQL não exige que nossos modelos de dados coincidam exatamente com os tipos em nosso esquema. Nossos clientes podem encontrar os usuários marcados em cada foto, e as fotos em que um usuário esteja marcado consultando o tipo `User` ou o tipo `Photo`. Eles não precisam consultar um tipo `Tag`: isso só complicaria a situação. Já fizemos o trabalho pesado de encontrar os usuários marcados ou as fotos em nosso resolver especificamente para que nossos clientes consultem esses dados mais facilmente.

Escalares personalizados

Conforme discutimos no Capítulo 4, o GraphQL tem um conjunto de tipos escalares default que podem ser usados em qualquer campo. Escalares como `Int`, `Float`, `String`, `Boolean` e `ID` são apropriados na maioria das situações, mas pode haver casos em que teremos de criar um tipo escalar personalizado para atender aos nossos requisitos de dados.

Quando implementamos um escalar personalizado, devemos criar regras sobre como o tipo deve ser serializado e validado. Por exemplo, se criarmos um tipo `DateTime`, será necessário definir o que deve ser considerado um `DateTime` válido.

Vamos adicionar esse escalar `DateTime` personalizado em nosso `typeDefs` e usá-lo no tipo `Photo`, no campo `created`. Esse campo é usado para armazenar a data e a hora em que uma foto específica foi postada:

```
const typeDefs = `  
  scalar DateTime  
  type Photo {  
    ...  
    created: DateTime!  
  }  
  ...
```

Todo campo em nosso esquema deve ser mapeado para um resolver. O campo `created` deve ser mapeado com um resolver para o tipo `DateTime`. Criamos um tipo escalar personalizado para `DateTime` porque queremos fazer parse e validar qualquer campo que use esse escalar como tipos `Date` de JavaScript.

Considere as várias maneiras com que podemos representar uma data e hora como string. Todas as strings a seguir representam datas válidas:

- “4/18/2018”
- “4/18/2018 1:30:00 PM”
- “Sun Apr 15 2018 12:10:17 GMT-0700 (PDT)”
- “2018-04-15T19:09:57.308Z”

Qualquer uma dessas strings pode ser usada para criar objetos `datetime` em JavaScript:

```
var d = new Date("4/18/2018")
console.log( d.toISOString() )
// "2018-04-18T07:00:00.000Z"
```

Nesse caso criamos um novo objeto `data` usando um formato e então convertemos essa string `datetime` para uma string de data em formato ISO.

Tudo que o objeto `Date` de JavaScript não entender será inválido. Você pode experimentar fazer o parse dos dados a seguir:

```
var d = new Date("Tuesday March")
console.log( d.toString() )
// "Invalid Date"
```

Quando consultarmos o campo `created` da foto, queremos garantir que o valor devolvido por esse campo contenha uma string no formato ISO de data e hora. Sempre que um campo devolver um valor de data, serializaremos esse valor como uma string em formato ISO usando `serialize`:

```
const serialize = value => new Date(value).toISOString()
```

A função de serialização obtém os valores do campo de nosso

objeto, e desde que esse campo contenha uma data formatada como um objeto JavaScript ou qualquer string `datetime` válida, ele sempre será devolvido pelo GraphQL no formato ISO de `datetime`.

Se seu esquema implementar um escalar personalizado, este poderá ser usado como argumento em uma consulta. Vamos supor que tenhamos criado um filtro para a consulta `allPhotos`. Essa consulta devolveria uma lista de fotos tiradas após uma data específica:

```
type Query {  
  ...  
  allPhotos(after: DateTime): [Photo!]!  
}
```

Se tivéssemos esse campo, os clientes poderiam nos enviar uma consulta contendo um valor de `DateTime`:

```
query recentPhotos(after:DateTime) {  
  allPhotos(after: $after) {  
    name  
    url  
  }  
}
```

Então eles enviariam o argumento `$after` usando variáveis de consulta:

```
{  
  "after": "4/18/2018"  
}
```

Queremos garantir que o parse do argumento `after` seja feito gerando um objeto JavaScript `Date` antes que seja enviado ao resolver:

```
const parseValue = value => new Date(value)
```

A função `parseValue` pode ser usada para fazer parse dos valores das strings de entrada enviadas com as consultas. O que quer que `parseValue` devolva será passado como argumento do resolver:

```
const resolvers = {  
  Query: {  
    allPhotos: (parent, args) => {  
      args.after // Objeto Date de JavaScript  
      ...  
    }  
  }  
}
```



```
}  
}
```

Os escalares personalizados devem ser capazes de serializar e fazer parse de valores de data. Há mais um lugar em que teremos que tratar strings de datas. É quando os clientes adicionam a string de data diretamente na própria consulta:

```
query {  
  allPhotos(after: "4/18/2018") {  
    name  
    url  
  }  
}
```

O argumento `after` não está sendo passado como uma variável da consulta. Em vez disso, ela foi adicionada diretamente no documento de consulta. Antes de podermos fazer parse desse valor, devemos obtê-lo da consulta depois que o seu parse tiver sido feito e uma AST (Abstract Syntax Tree, ou Árvore Sintática Abstrata) tiver sido gerada. Usamos a função `parseLiteral` para obter esses valores do documento de consulta antes que o parse seja feito:

```
const parseLiteral = ast => ast.value
```

A função `parseLiteral` é usada para obter o valor da data adicionada diretamente no documento de consulta. Nesse caso, tudo que temos que fazer é devolver esse valor, mas, se for necessário, poderíamos executar passos extras de parsing nessa função.

Precisaremos de todas essas três funções que criamos para lidar com valores `DateTime` quando criarmos nosso escalar personalizado. Vamos adicionar o resolver para o nosso escalar personalizado `DateTime` em nosso código:

```
const { GraphQLScalarType } = require('graphql')  
...  
const resolvers = {  
  Query: { ... },  
  Mutation: { ... },  
  Photo: { ... },  
  User: { ... },  
  DateTime: new GraphQLScalarType({
```

```

    name: 'DateTime',
    description: 'A valid date time value.',
    parseValue: value => new Date(value),
    serialize: value => new Date(value).toISOString(),
    parseLiteral: ast => ast.value
  })
}

```

Usamos o objeto `GraphQLScalarType` a fim de criar resolvers para escalares personalizados. O resolver `DateTime` é colocado em nossa lista de resolvers. Ao criar um novo tipo escalar, temos de adicionar as três funções: `serialize`, `parseValue` e `parseLiteral`, que tratarão qualquer campo ou argumento que implemente o escalar `DateTime`.

Datas de exemplo

Em nossos dados, vamos garantir também que adicionaremos uma chave `created` e um valor de data para as duas fotos existentes. Qualquer string de data válida ou objeto data servirá porque o campo `created` será serializado antes de ser devolvido:

```

var photos = [
  {
    ...
    "created": "3-28-1977"
  },
  {
    ...
    "created": "1-2-1985"
  },
  {
    ...
    "created": "2018-04-15T19:09:57.308Z"
  }
]

```

Agora, quando adicionarmos campos `DateTime` em nossos conjuntos de seleção, poderemos ver essas datas e tipos formatados como strings de data ISO:

```

query listPhotos {
  allPhotos {
    name
    created
  }
}

```

A única tarefa restante é garantir que adicionaremos um timestamp em cada foto quando ela for postada. Fazemos isso adicionando um campo `created` em cada foto e atribuindo-lhe um timestamp com o `DateTime` atual usando o objeto `Date` de JavaScript:

```
postPhoto(parent, args) {  
  var newPhoto = {  
    id: _id++,  
    ...args.input,  
    created: new Date()  
  }  
  photos.push(newPhoto)  
  return newPhoto  
}
```

Assim, quando novas fotos forem postadas, elas terão um timestamp com a data e a hora em que foram criadas.

apollo-server-express

Pode ser que haja um cenário em que você queira adicionar o Apollo Server em uma aplicação existente ou queira tirar proveito do middleware Express. Nesse caso, você poderia considerar o uso do `apollo-server-express`. Com o Apollo Server Express, será possível usar todas as funcionalidades mais recentes do Apollo Server, além de usar também uma configuração mais personalizada. Em nosso caso, vamos refatorar o servidor para usar o Apollo Server Express a fim de configurar uma rota home personalizada, uma rota para o Playground e, mais tarde, permitir o upload das imagens postadas e salvá-las no servidor.

Começaremos removendo o `apollo-server`:

```
npm remove apollo-server
```

Em seguida, instalaremos o Apollo Server Express e o Express:

```
npm install apollo-server-express express
```



Express

O Express é, de longe, um dos projetos mais populares do ecossistema do Node.js. Ele permite configurar uma aplicação web Node.js de modo rápido e eficiente.

A partir de agora podemos refatorar o nosso arquivo `index.js`. Começaremos alterando a instrução `require` para incluir o `apollo-server-express`. Então incluiremos o `express`:

```
// 1. Require do `apollo-server-express` e do `express`
const { ApolloServer } = require('apollo-server-express')
const express = require('express')

...

// 2. Chama `express()` para criar uma aplicação Express
var app = express()

const server = new ApolloServer({ typeDefs, resolvers })

// 3. Chama `applyMiddleware()` para permitir o middleware montado no mesmo path
server.applyMiddleware({ app })

// 4. Cria uma rota home
app.get('/', (req, res) => res.end('Welcome to the PhotoShare API'))

// 5. Ouve uma porta específica
app.listen({ port: 4000 }, () =>
  console.log(`GraphQL Server running @ http://localhost:4000${server.graphqlPath}`)
)
```

Ao incluir o Express, podemos tirar proveito de todas as funções do middleware disponibilizadas a nós pelo framework. Para incorporá-lo no servidor, basta chamar a função `express`, chamar `applyMiddleware` e então podemos criar uma rota personalizada. Agora, se acessarmos `http://localhost:4000`, veremos uma página em que se lê “Welcome to the PhotoShare API” (Bem-vindo à API de PhotoShare). É um placeholder por enquanto.

Em seguida, vamos configurar uma rota personalizada para o GraphQL Playground executar em `http://localhost:4000/playground`. Podemos fazer isso instalando um pacote auxiliar do npm. Em primeiro lugar, é necessário instalar o pacote `graphql-playground-middleware-express`:

```
npm install graphql-playground-middleware-express
```

Então devemos exigir esse pacote no início do arquivo de índice:

```
const expressPlayground = require('graphql-playground-middleware-express').default
```

...

```
app.get('/playground', expressPlayground({ endpoint: '/graphql' })))
```

Usaremos então o Express para criar uma rota para o Playground, de modo que, sempre que quisermos usá-lo, acessaremos `http://localhost:4000/playground`.

Nosso servidor agora está configurado com o Apollo Server Express, e temos três rotas distintas executando:

- / para a página inicial;
- /graphql para o endpoint do GraphQL;
- /playground para o GraphQL Playground.

Nesse ponto, reduziremos também o tamanho de nosso arquivo de índice movendo `typeDefs` e `resolvers` para os seus próprios arquivos.

Inicialmente criaremos um arquivo chamado `typeDefs.graphql` e o colocaremos na raiz do projeto. Será apenas o esquema, contendo somente texto. Você também pode mover os `resolvers` para a sua própria pasta chamada `resolvers`. Essas funções podem ser colocadas em um arquivo `index.js`, ou você pode modularizar os arquivos de `resolvers` como fizemos no repositório (<https://github.com/MoonHighway/learning-graphql/tree/master/chapter-05/photo-share-api/resolvers>).

Feito isso, podemos importar os `typeDefs` e os `resolvers`, como vemos a seguir. Usaremos o módulo `fs` do Node.js para ler o arquivo `typeDefs.graphql`:

```
const { ApolloServer } = require('apollo-server-express')
const express = require('express')
const expressPlayground = require('graphql-playground-middleware-express').default
const { readFileSync } = require('fs')
```

```
const typeDefs = readFileSync('./typeDefs.graphql', 'UTF-8')
const resolvers = require('./resolvers')
var app = express()
```

```
const server = new ApolloServer({ typeDefs, resolvers })

server.applyMiddleware({ app })

app.get('/', (req, res) => res.end('Welcome to the PhotoShare API'))
app.get('/playground', expressPlayground({ endpoint: '/graphql' }))

app.listen({ port: 4000 }, () =>
  console.log(`GraphQL Server running at http://localhost:4000${server.graphqlPath}`)
)
```

Agora que refatoramos o servidor, estamos prontos para dar o próximo passo: integrar um banco de dados.

Contexto

Nesta seção descreveremos o *contexto*, que é o local em que você pode armazenar valores globais acessíveis a qualquer resolver. O contexto é um bom lugar para armazenar informações de autenticação, detalhes do banco de dados, cache de dados locais e tudo mais que uma operação GraphQL tenha que resolver.

Podemos chamar diretamente APIs REST e bancos de dados em nossos resolvers, mas, em geral, abstraímos essa lógica em um objeto que inserimos no contexto para impor uma separação de responsabilidades e permitir refatorações mais simples no futuro. O contexto também pode ser usado para acessar dados REST de um Apollo Data Source. Para mais informações sobre o assunto, consulte Apollo Data Sources na documentação (<http://bit.ly/2vac9ZC>).

Em nosso caso, porém, incorporaremos o contexto agora para cuidar de algumas das limitações atuais de nossa aplicação. Em primeiro lugar, estamos armazenando dados na memória, e essa não é uma solução muito escalável. Também estamos lidando com IDs de modo desleixado, incrementando esses valores a cada mutação. Em vez disso, contaremos com um banco de dados para cuidar da armazenagem de dados e da geração de IDs. Nossos

resolvers poderão acessar esse banco de dados a partir do contexto.

Instalando o Mongo

O GraphQL não se importa com o banco de dados que você usar. Você pode usar Postgres, Mongo, SQL Server, Firebase, MySQL, Redis, Elastic – o que quiser. Em virtude de sua popularidade na comunidade do Node.js, usaremos o Mongo como solução para armazenagem de dados em nossa aplicação.

Para começar a trabalhar com o MongoDB em um Mac, usaremos o Homebrew. Para instalá-lo, acesse <https://brew.sh/>. Feito isso, executaremos o processo de instalação do Mongo com o Homebrew executando os comandos a seguir:

```
brew install mongo
brew services list
brew services start
```

Depois de ter iniciado o MongoDB com sucesso, poderemos começar a ler e a escrever dados na instância local do Mongo.



Nota para usuários do Windows

Se quiser executar uma versão local do MongoDB no Windows, consulte <http://bit.ly/inst-mdb-windows>.

Você também pode usar um serviço Mongo online como o mLab, conforme mostrado na Figura 5.1. É possível criar um banco de dados sandbox gratuitamente.

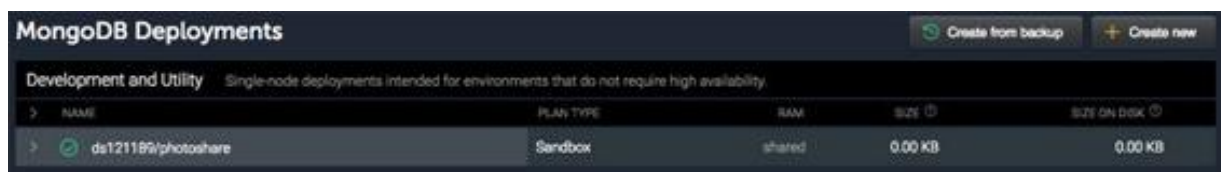


Figura 5.1 – mLab.

Adicionando um banco de dados no contexto

É hora de se conectar com o nosso banco de dados e adicionar a conexão no contexto. Usaremos um pacote chamado `mongodb` para comunicação com o nosso banco de dados. Ele pode ser instalado com o comando a seguir: `npm install mongodb`.

Depois de instalar esse pacote, modificaremos o arquivo de configuração do Apollo Server, `index.js`. Devemos esperar até que o `mongodb` se conecte com sucesso ao nosso banco de dados para iniciar o serviço. Também precisaremos extrair a informação de host do banco de dados de uma variável de ambiente chamada `DB_HOST`. Deixaremos essa variável de ambiente acessível ao nosso projeto, em um arquivo chamado `.env` na raiz do projeto.

Se você estiver usando o Mongo localmente, sua URL terá uma aparência semelhante a:

```
DB_HOST=mongodb://localhost:27017/<Nome-do-banco-de-dados>
```

Se você estiver usando o mLab, sua URL terá o aspecto a seguir. Lembre-se de criar um usuário e uma senha para o banco de dados e substituir `<usuariobd>` e `<senhabd>` com esses valores.

```
DB_HOST=mongodb://<usuariobd>:<senhabd>@5555.mlab.com:5555/<Nome-do-banco-de-dados>
```

Vamos nos conectar com o banco de dados e construir um objeto de contexto antes de iniciar o serviço. Usaremos também o pacote `dotenv` para carregar a URL em `DB_HOST`:

```
const { MongoClient } = require('mongodb')
require('dotenv').config()

...

// 1. Cria uma função assíncrona
async function start() {
  const app = express()
  const MONGO_DB = process.env.DB_HOST

  const client = await MongoClient.connect(
    MONGO_DB,
    { useNewUrlParser: true }
  )
```



```

const db = client.db()

const context = { db }

const server = new ApolloServer({ typeDefs, resolvers, context })

server.applyMiddleware({ app })

app.get('/', (req, res) => res.end("Welcome to the PhotoShare API"))

app.get('/playground', expressPlayground({ endpoint: '/graphql' }))

app.listen({ port: 4000 }, () =>
  console.log(
    `GraphQL Server running at http://localhost:4000${server.graphqlPath}`
  )
)
}

// 5. Chama start quando estiver pronto para iniciar
start()

```

Com `start`, fazemos a conexão com o banco de dados. Conectar-se com um banco de dados é um processo assíncrono. Demorará um pouco para que haja uma conexão bem-sucedida. Essa função assíncrona nos permite esperar que uma promise (promessa) se resolva, usando a palavra reservada `await`. Nossa primeira tarefa nessa função é esperar uma conexão bem-sucedida com o banco de dados local ou remoto. Depois que tivermos uma conexão com o banco de dados, essa conexão poderá ser adicionada no objeto de contexto e o nosso servidor será iniciado.

Podemos agora modificar os resolvers de nossa consulta para que devolvam informações de nossas coleções do Mongo em vez de devolver dados de arrays locais. Também adicionaremos consultas para `totalUsers` e `allUsers` e as acrescentaremos no esquema:

- Esquema

```

type Query {
  ...
  totalUsers: Int!
}

```

```
    allUsers: [User!]!
  }
```

- Resolvers

```
Query: {

  totalPhotos: (parent, args, { db }) =>
    db.collection('photos')
      .estimatedDocumentCount(),

  allPhotos: (parent, args, { db }) =>
    db.collection('photos')
      .find()
      .toArray(),

  totalUsers: (parent, args, { db }) =>
    db.collection('users')
      .estimatedDocumentCount(),

  allUsers: (parent, args, { db }) =>
    db.collection('users')
      .find()
      .toArray()
}
```

`db.collection('photos')` é a forma de acessar uma coleção do Mongo. Podemos contar os documentos da coleção usando `.estimatedDocumentCount()`. É possível listar todos os documentos de uma coleção e convertê-los em um array usando `.find().toArray()`. Nesse momento, a coleção `photos` está vazia, mas esse código funcionará. Os resolvers `totalPhotos` e `totalUsers` não devem devolver nada. Os resolvers `allPhotos` e `allUsers` devem devolver arrays vazios.

Para adicionar fotos no banco de dados, um usuário deve estar logado. Na próxima seção lidaremos com a autorização de um usuário com o GitHub e postaremos nossa primeira foto no banco de dados.

Autorização do GitHub

Autorizar e autenticar usuários é uma parte importante de qualquer

aplicação. Há uma série de estratégias que podem ser usadas para que isso ocorra. A autorização via um provedor social é uma estratégia popular porque deixa muitos dos detalhes de gerenciamento de contas a cargo desse provedor. Também pode ajudar os usuários a se sentirem mais seguros ao fazerem login porque o provedor social talvez seja um serviço com o qual já se sintam à vontade. Em nossa aplicação implementaremos uma autorização do GitHub porque é muito provável que você já tenha uma conta nesse local (e se não tiver, é fácil e rápido criar uma!)¹.

Configurando o OAuth do GitHub

Antes de começar, devemos configurar a autorização do GitHub para que essa aplicação funcione. Para isso, execute os passos a seguir:

1. Acesse <https://www.github.com> e faça login.
2. Vá para Account Settings (Configurações de conta).
3. Acesse Developer Settings (Configurações de desenvolvedor).
4. Clique em New OAuth App (Nova aplicação OAuth).
5. Adicione as configurações a seguir (conforme vemos na Figura 5.2):

Application name (Nome da aplicação)

Localhost 3000

Homepage URL (URL da página inicial)

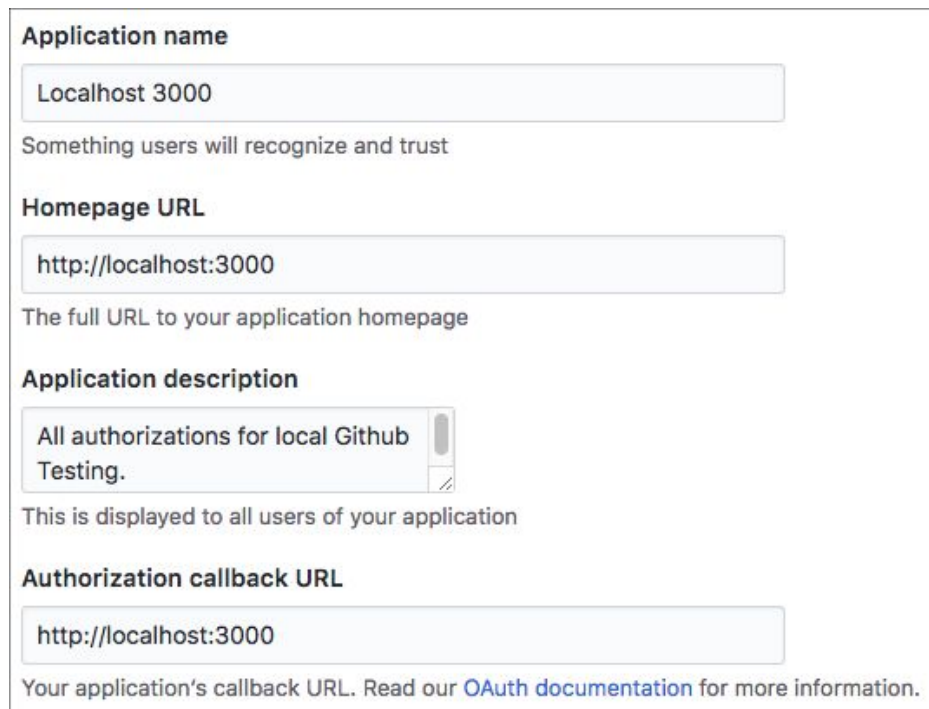
<http://localhost:3000>

Application description (Descrição da aplicação)

All authorizations for local GitHub Testing (Todas as autorizações para testes locais do GitHub)

Authorization callback URL (URL da callback de autorização)

<http://localhost:3000>



The image shows a web form for registering a new OAuth application on GitHub. The form is titled 'Application name' and contains several input fields and descriptive text. The 'Application name' field is filled with 'Localhost 3000'. Below it, a note says 'Something users will recognize and trust'. The 'Homepage URL' field is filled with 'http://localhost:3000', with a note 'The full URL to your application homepage'. The 'Application description' field is filled with 'All authorizations for local Github Testing.', with a note 'This is displayed to all users of your application'. The 'Authorization callback URL' field is filled with 'http://localhost:3000', with a note 'Your application's callback URL. Read our [OAuth documentation](#) for more information.'

Application name

Localhost 3000

Something users will recognize and trust

Homepage URL

http://localhost:3000

The full URL to your application homepage

Application description

All authorizations for local Github Testing.

This is displayed to all users of your application

Authorization callback URL

http://localhost:3000

Your application's callback URL. Read our [OAuth documentation](#) for more information.

Figura 5.2 – Nova aplicação OAuth.

6. Clique em Save (Salvar).
7. Acesse OAuth Account Page (Página de conta do OAuth) e obtenha o seu client_id e o client_secret, como vemos na Figura 5.3.

The screenshot shows the GitHub OAuth application configuration interface. At the top, it says 'Localhost 3000' and 'MoonTahoe owns this application.' with a 'Transfer ownership' button. Below this, there's a section to 'List this application in the Marketplace' with a corresponding button. The main section is titled '2 users' and displays the 'Client ID' and 'Client Secret' as masked text. There are buttons for 'Revoke all user tokens' and 'Reset client secret'. The 'Application logo' section has a 'Drag & drop' area and an 'Upload new logo' button. The 'Application name' field contains 'Localhost 3000'. The 'Homepage URL' field contains 'http://localhost:3000'. The 'Application description' field contains 'All authorizations for local Github Testing.'. The 'Authorization callback URL' field contains 'http://localhost:3000'. At the bottom, there are 'Update application' and 'Delete application' buttons.

Localhost 3000

MoonTahoe owns this application. [Transfer ownership](#)

You can list your application in the [GitHub Marketplace](#) so that other users can discover it. [List this application in the Marketplace](#)


2 users

Client ID
[REDACTED]

Client Secret
[REDACTED]

[Revoke all user tokens](#) [Reset client secret](#)

Application logo

 [Upload new logo](#)

You can also drag and drop a picture from your computer.

Drag & drop

Application name

Localhost 3000

Something users will recognize and trust

Homepage URL

http://localhost:3000

The full URL to your application homepage

Application description

All authorizations for local Github Testing.

This is displayed to all users of your application

Authorization callback URL

http://localhost:3000

Your application's callback URL. Read our [OAuth documentation](#) for more information.

[Update application](#) [Delete application](#)

Figura 5.3 – Configurações da aplicação OAuth.

Com essa configuração feita, podemos agora obter um token `auth` e informações sobre o usuário do GitHub. Especificamente, precisaremos de `client_id` e `client_secret`.

Processo de autorização

O processo de autorização de uma aplicação GitHub ocorre no

cliente e no servidor. Nesta seção discutiremos como lidar com o servidor, e no Capítulo 6 descreveremos a implementação do cliente. Como mostra a Figura 5.4, o processo de autorização completo ocorre de acordo com os passos a seguir. Os passos em negrito mostram o que acontecerá no servidor, conforme descrito neste capítulo:

1. Cliente: Pede um código ao GitHub usando um url com um `client_id`.
2. Usuário: Permite acesso a informações de conta no GitHub para a aplicação cliente.
3. GitHub: envia o código para url de redirecionamento do OAuth:
`http://localhost:3000?code=XYZ`
4. Cliente: Envia uma mutação GraphQL `githubAuth(code)` com o código.
5. API: Requisita um `access_token` do GitHub com as credenciais:
`Client_id`, `client_secret` e `Client_code`.
6. GitHub: Responde com o `access_token` que pode ser usado em futuras requisições de informações.
7. API: Requisita informações de usuário com o `access_token`.
8. GitHub: Responde com informações de usuário: `name`, `githubLogin` e `avatar`.
9. API: Resolve a mutação `authUser(code)` com `AuthPayload`, que contém um token e o usuário.
10. Cliente: Salva o token para que seja enviado em requisições GraphQL no futuro.

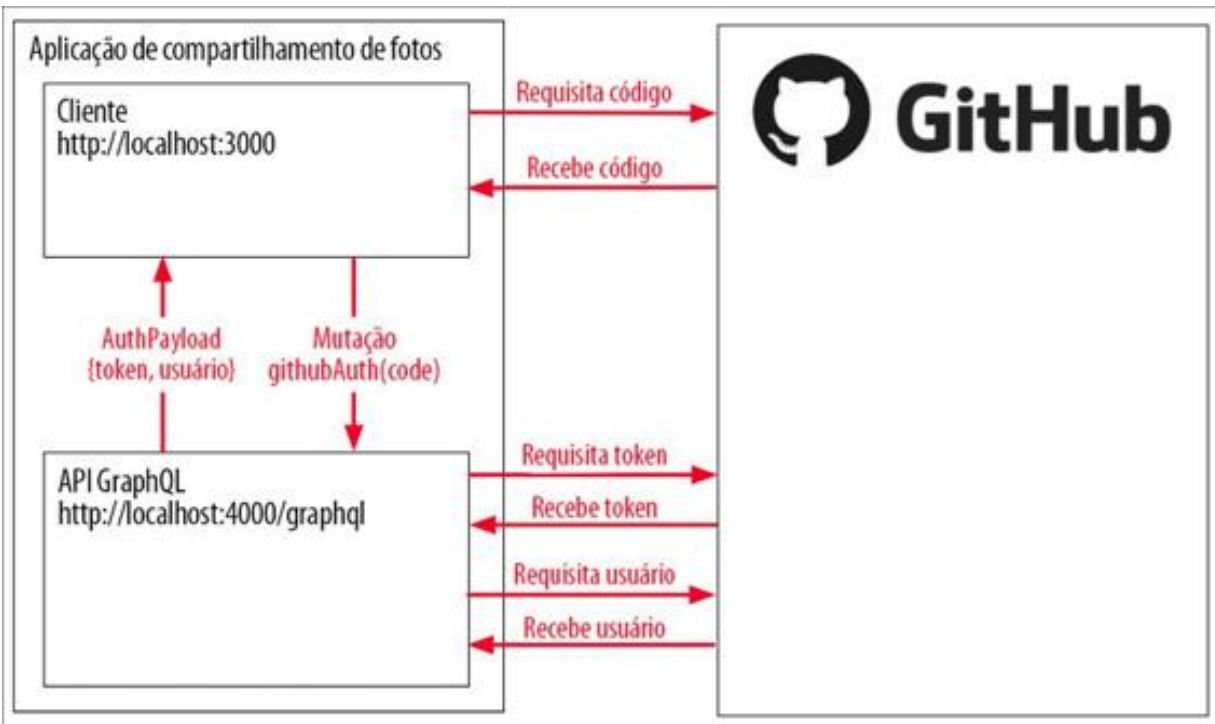


Figura 5.4 – Processo de autorização.

Para implementar a mutação `githubAuth`, suporemos que temos um código. Depois de usar o código para obter um token, salvaremos as informações do novo usuário e o token em nosso banco de dados local. Também devolveremos essas informações ao cliente. O cliente salvará o token localmente e o enviará de volta para nós a cada requisição. Usaremos o token para autorizar o usuário e acessar seu registro de dados.

Mutaçao githubAuth

Cuidaremos da autorização dos usuários utilizando uma mutação GraphQL. No Capítulo 4, criamos um tipo payload personalizado chamado `AuthPayload` em nosso esquema. Vamos acrescentar o `AuthPayload` e a mutação `githubAuth` em nosso `typeDefs`:

```
type AuthPayload {  
  token: String!  
  user: User!  
}
```

```
type Mutation {
```

```
...
  githubAuth(code: String!): AuthPayload!
}
```

O tipo `AuthPayload` é usado somente como uma resposta às mutações de autorização. Ele contém o usuário que foi autorizado pela mutação, além de um token que os usuários poderão utilizar para se identificarem em futuras requisições.

Antes de implementar o resolver `githubAuth`, devemos criar duas funções para lidar com as requisições para a API GitHub :

```
const requestGithubToken = credentials =>
  fetch(
    'https://github.com/login/oauth/access_token',
    {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Accept: 'application/json'
      },
      body: JSON.stringify(credentials)
    }
  )
  .then(res => res.json())
  .catch(error => {
    throw new Error(JSON.stringify(error))
  })
```

A função `requestGithubToken` devolve uma promise (promessa) de busca. As `credentials` são enviadas a uma URL da API do GitHub no corpo de uma requisição POST. Elas são constituídas de três itens: `client_id`, `client_secret` e `code`. Feito isso, um parse da resposta do GitHub é feito gerando dados JSON. Podemos agora usar essa função para requisitar um token de acesso no GitHub usando as `credentials`. Essa e outras funções auxiliares que criaremos futuramente podem ser encontradas em um arquivo `lib.js` no repositório em <https://github.com/MoonHighway/learning-graphql/blob/master/chapter-05/photo-share-api/lib.js/>.

Assim que tivermos um token do GitHub, devemos acessar as informações da conta do usuário atual. Especificamente, queremos

o login, o nome e a foto de perfil no GitHub. Para obter essas informações, devemos enviar outra requisição para a API do GitHub, junto com o token de acesso obtido na requisição anterior:

```
const requestGithubUserAccount = token =>
  fetch(`https://api.github.com/user?access_token=${token}`)
    .then(toJSON)
    .catch(throwError)
```

Essa função também devolve uma promise de busca. Nessa rota da API do GitHub, podemos acessar informações sobre o usuário atual, desde que tenhamos um token de acesso.

Vamos agora combinar as duas requisições em uma única função assíncrona que pode ser usada para autorizar um usuário com o GitHub:

```
async authorizeWithGithub(credentials) {
  const { access_token } = await requestGithubToken(credentials)
  const githubUser = await requestGithubUserAccount(access_token)
  return { ...githubUser, access_token }
}
```

Usar `async/await` nesse caso permite tratar várias requisições assíncronas. Em primeiro lugar, requisitamos o token de acesso e esperamos a resposta. Então, usando `access_token`, solicitamos as informações de conta do usuário do GitHub e esperamos uma resposta. Depois que tivermos os dados, reunimos tudo em um só objeto.

Criamos as funções auxiliares que oferecerão suporte à funcionalidade do resolver. Vamos agora escrever o resolver propriamente dito a fim de obter um token e uma conta de usuário do GitHub:

```
async githubAuth(parent, { code }, { db }) {
  // 1. Obtém dados do GitHub
  let {
    message,
    access_token,
    avatar_url,
    login,
    name
```

```

    } = await authorizeWithGithub({
      client_id: <YOUR_CLIENT_ID_HERE>,
      client_secret: <YOUR_CLIENT_SECRET_HERE>,
      code
    })
    // 2. Se houver uma mensagem, algo deu errado
    if (message) {
      throw new Error(message)
    }
    // 3. Empacota os resultados em um único objeto
    let latestUserInfo = {
      name,
      githubLogin: login,
      githubToken: access_token,
      avatar: avatar_url
    }
    // 4. Adiciona ou atualiza o registro com as novas informações
    const { ops:[user] } = await db
      .collection('users')
      .replaceOne({ githubLogin: login }, latestUserInfo, { upsert: true })
    // 5. Devolve dados do usuário e seu token
    return { user, token: access_token }

  }

```

Os resolvers podem ser assíncronos. Podemos esperar uma resposta da rede antes de devolver o resultado de uma operação a um cliente. O resolver `githubAuth` é assíncrono porque devemos esperar duas respostas do GitHub antes de ter os dados que devemos devolver.

Após ter obtido os dados de usuário do GitHub, verificamos nosso banco de dados local para ver se esse usuário fez login em nossa aplicação antes, o que significa que ele já tem uma conta. Se o usuário tiver uma conta, atualizaremos os detalhes dela com as informações que recebemos do GitHub. O usuário pode ter alterado seu nome ou a foto de perfil desde a última vez que fez login. Se não tiver uma conta ainda, adicionaremos o novo usuário em nossa coleção de usuários. Nos dois casos, esse resolver devolverá o `user` logado e o `token`.

É hora de testar esse processo de autorização e para isso, precisamos de um código. Para obter o código, você deve adicionar o seu ID de cliente no URL a seguir:

`https://github.com/login/oauth/authorize?client_id=SEU-ID-AQUI&scope=user`

Copie e cole a URL com o seu `client_id` do GitHub na barra de endereço em uma nova janela do navegador. Você será direcionado para o GitHub, no qual concordará em autorizar essa aplicação. Quando autorizar uma aplicação, o GitHub redirecionará você de volta para `http://localhost:3000` com um código:

`http://localhost:3000?code=XYZ`

Nesse caso, o código é XYZ. Copie o código da URL do navegador e então envie-o com a mutação `githubAuth`:

```
mutation {  
  githubAuth(code:"XYZ") {  
    token  
    user {  
      githubLogin  
      name  
      avatar  
    }  
  }  
}
```

Essa mutação autorizará o usuário atual e devolverá um token junto com informações sobre esse usuário. Salve o token. Teremos que enviá-lo no cabeçalho de futuras requisições.



Credenciais incorretas

Se vir o erro “Bad Credentials” (Credenciais incorretas), é sinal de que o ID do cliente, a senha ou o código enviados à API do GitHub estavam incorretos. Verifique o ID e a senha do cliente para confirmar; muitas vezes, é o código que causa esse erro.

Os códigos do GitHub servem somente por um período limitado e podem ser usados apenas uma vez. Se houver um bug no resolver depois que as credenciais foram requisitadas, o código usado na requisição não será mais válido. Em geral, você pode resolver esse erro pedindo outro código ao GitHub.

Autenticando usuários

Para se identificar em requisições futuras, é necessário enviar seu

token a cada requisição no cabeçalho `Authorization`. Esse token será usado para identificar o usuário consultando seu registro no banco de dados.

O GraphQL Playground tem um local em que você pode adicionar cabeçalhos para cada requisição. No canto inferior, há uma aba ao lado de “Query Variables” (Variáveis de consulta) que se chama “HTTP Headers” (Cabeçalhos HTTP). Podemos adicionar cabeçalhos HTTP em uma requisição usando essa aba. Basta enviar os cabeçalhos como JSON:

```
{
  "Authorization": "<SEU_TOKEN>"
}
```

Substitua `<SEU_TOKEN>` pelo token devolvido pela mutação `githubAuth`. A partir de agora, você enviará a chave para sua identificação em cada requisição GraphQL. É necessário usar essa chave para que sua conta seja encontrada e adicionada no contexto.

A consulta me

A seguir, queremos criar uma consulta que se refira às informações do nosso próprio usuário: a consulta `me`. Essa consulta devolve o usuário logado no momento com base no token enviado nos cabeçalhos HTTP da requisição. Se não houver nenhum usuário logado no momento, a consulta devolverá `null`.

O processo tem início quando um cliente envia a consulta GraphQL `me` com `Authorization: token` para informações de usuário seguras. A API então captura um cabeçalho `Authorization` e usa o token para procurar o registro do usuário atual no banco de dados. Também adiciona a conta do usuário atual no contexto. Depois que estiver no contexto, qualquer resolver terá acesso a esse usuário.

Cabe a nós identificar o usuário atual e inseri-lo no contexto. Vamos modificar a configuração de nosso servidor. Teremos que mudar o modo de construir o objeto de contexto. Em vez de um objeto, usaremos uma função para tratar o contexto:

```
async function start() {
```

```

const app = express()
const MONGO_DB = process.env.DB_HOST

const client = await MongoClient.connect(
  MONGO_DB,
  { useNewUrlParser: true }
)

const db = client.db()

const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: async ({ req }) => {
    const githubToken = req.headers.authorization
    const currentUser = await db.collection('users').findOne({ githubToken })
    return { db, currentUser }
  }
})
...
}

```

O contexto pode ser um objeto ou uma função. Para que nossa aplicação funcione, é necessário que seja uma função, de modo que possamos definir o contexto sempre que houver uma requisição. Quando o contexto é uma função, esta será chamada a cada requisição GraphQL. O objeto devolvido por essa função é o contexto que será enviado ao resolver.

Na função de contexto, podemos capturar e fazer parse do cabeçalho de autorização da requisição para obter o token. Depois que tivermos um token, este poderá ser usado para procurar um usuário em nosso banco de dados. Se tivermos um usuário, ele será adicionado no contexto. Caso contrário, o valor do usuário no contexto será null.

Com esse código implementado, é hora de adicionar a consulta `me`. Em primeiro lugar, temos que modificar nosso `typeDefs`:

```

type Query {
  me: User
  ...

```

```
}
```

A consulta `me` devolve um usuário nullable. Ele será null se um usuário atual autorizado não for encontrado. Vamos adicionar o resolver para a consulta `me`:

```
const resolvers = {  
  Query: {  
    me: (parent, args, { currentUser }) => currentUser,  
    ...  
  }  
}
```

Já fizemos o trabalho pesado de procurar o usuário com base em seu token. Nesse ponto, basta devolver o objeto `currentUser` do contexto. Novamente, essa informação será null se não houver nenhum usuário.

Se o token correto for adicionado no cabeçalho de autorização HTTP, você poderá enviar uma requisição para obter detalhes sobre si mesmo usando a consulta `me`:

```
query currentUser {  
  me {  
    githubLogin  
    name  
    avatar  
  }  
}
```

Ao executar essa consulta, você será identificado. Um bom teste para confirmar se tudo está correto é tentar executar essa consulta sem o cabeçalho de autorização ou com um token incorreto. Se você fornecer um token incorreto ou não fornecer um cabeçalho, verá que a consulta `me` devolve null.

A mutação `postPhoto`

Para postar uma foto em nossa aplicação, um usuário deve estar logado. A mutação `postPhoto` é capaz de determinar quem está logado verificando o contexto. Vamos modificar a mutação `postPhoto`:

```
async postPhoto(parent, args, { db, currentUser }) {
```

```

// 1. Se não houver um usuário no contexto, lança um erro
if (!currentUser) {
  throw new Error('only an authorized user can post a photo')
}

// 2. Salva o id do usuário atual com a foto
const newPhoto = {
  ...args.input,
  userID: currentUser.githubLogin,
  created: new Date()
}

// 3. Insere a nova foto, captura o id criado pelo banco de dados
const { insertedIds } = await db.collection('photos').insert(newPhoto)
newPhoto.id = insertedIds[0]

return newPhoto
}

```

A mutação `postPhoto` passou por várias mudanças para salvar uma nova foto no banco de dados. Em primeiro lugar, `currentUser` é obtido do contexto. Se esse valor for `null`, lançaremos um erro e impediremos que a mutação `postPhoto` prossiga na execução. Para postar uma foto, o usuário deve enviar o token correto no cabeçalho `Authorization`.

Em seguida, adicionamos o ID do usuário atual no objeto `newPhoto`. Podemos agora salvar o registro da nova foto na coleção de fotos do banco de dados. O Mongo cria um identificador único para cada documento que salvar. Quando a nova foto é adicionada, esse identificador pode ser obtido usando o array `insertedIds`. Antes de devolver a foto, devemos garantir que ela tenha um identificador único.

Também é necessário alterar os resolvers de `Photo`:

```

const resolvers = {
  ...
  Photo: {
    id: parent => parent.id || parent._id,
    url: parent => `/img/photos/${parent._id}.jpg`,
  },
}

```

```
    postedBy: (parent, args, { db }) =>
      db.collection('users').findOne({ githubLogin: parent.userID })
  }
```

Inicialmente, se o cliente pedir o ID de uma foto, devemos garantir que ele receba o valor correto. Se a foto-pai ainda não tiver um ID, podemos supor que um registro de banco de dados foi criado para a foto-pai e ele terá um ID salvo no campo `_id`. Devemos garantir que o campo ID da foto será resolvido com o ID do banco de dados.

Em seguida, vamos supor que estamos servindo essas fotos a partir do mesmo servidor web. Devolveremos a rota local para a foto. Essa rota local é criada usando o ID da foto.

Por fim, temos que modificar o resolver `postedBy` para procurar o usuário que postou a foto no banco de dados. Podemos usar o `userID` salvo com a foto `parent` para procurar o registro desse usuário no banco de dados. O `userID` da foto deve coincidir com o usuário `githubLogin`, portanto o método `.findOne()` deve devolver um registro de usuário – o usuário que postou a foto.

Com nosso cabeçalho de autorização definido, devemos ser capazes de postar novas fotos no serviço GraphQL:

```
mutation post($input: PostPhotoInput!) {
  postPhoto(input: $input) {
    id
    url
    postedBy {
      name
      avatar
    }
  }
}
```

Depois de postar a foto, será possível pedir o `id` e o `url` dela, além do `name` e do `avatar` do usuário que a postou.

Acréscimo da mutação para usuários simulados

Para testar nossa aplicação com usuários que não sejam nós mesmos, adicionaremos uma mutação que nos permitirá preencher o banco de dados com usuários simulados da API `random.me`.

Podemos cuidar disso com uma mutação chamada `addFakeUsers`. Inicialmente, adicionaremos o código a seguir no esquema:

```
type Mutation {  
  addFakeUsers(count: Int = 1): [User!]!  
  ...  
}
```

Observe que o argumento de contador aceita o número de usuários simulados a serem adicionados e devolve uma lista de usuários. Essa lista contém as contas dos usuários simulados adicionados por essa mutação. Por padrão, adicionamos um usuário de cada vez, mas podemos adicionar mais de um enviando essa mutação com um contador diferente:

```
addFakeUsers: async (root, {count}, {db}) => {  
  
  var randomUserApi = `https://randomuser.me/api/?results=${count}`  
  
  var { results } = await fetch(randomUserApi)  
    .then(res => res.json())  
  
  var users = results.map(r => ({  
    githubLogin: r.login.username,  
    name: `${r.name.first} ${r.name.last}`,  
    avatar: r.picture.thumbnail,  
    githubToken: r.login.sha1  
  })))  
  
  await db.collection('users').insert(users)  
  
  return users  
}
```

Para testar a adição de novos usuários, inicialmente devemos obter alguns dados simulados de `randomuser.me`. `addFakeUsers` é uma função assíncrona que pode ser usada para buscar esses dados. Em seguida, serializamos os dados de `randomuser.me`, criando objetos usuário que correspondam ao nosso esquema. Então adicionamos esses usuários novos no banco de dados e devolvemos a lista deles.

Agora podemos preencher o banco de dados usando uma mutação:

```
mutation {  
  addFakeUsers(count: 3) {  
    name  
  }  
}
```

Essa mutação adiciona três usuários simulados no banco de dados. Agora que temos usuários simulados, queremos também fazer login com uma conta de usuário simulada usando uma mutação. Vamos adicionar um `fakeUserAuth` em nosso tipo `Mutation`:

```
type Mutation {  
  fakeUserAuth(githubLogin: ID!): AuthPayload!  
  ...  
}
```

Em seguida, é necessário adicionar um resolver que devolva um token a ser usado para autorizar nossos usuários simulados:

```
async fakeUserAuth (parent, { githubLogin }, { db }) {  
  
  var user = await db.collection('users').findOne({ githubLogin })  
  
  if (!user) {  
    throw new Error(`Cannot find user with githubLogin "${githubLogin}"`)  
  }  
  
  return {  
    token: user.githubToken,  
    user  
  }  
  
}
```

O resolver `fakeUserAuth` obtém o `githubLogin` dos argumentos da mutação e o usa para encontrar esse usuário no banco de dados. Depois de encontrar esse usuário, o token e a conta desse usuário serão devolvidos no formato de nosso tipo `AuthPayload`.

Agora podemos autenticar os usuários simulados enviando uma mutação:

```
mutation {
```

```
fakeUserAuth(githubLogin:"jDoe") {  
  token  
}  
}
```

Adicione o token devolvido no cabeçalho de autorização HTTP para postar novas fotos com esse usuário simulado.

Conclusão

Bem, conseguimos. Implementamos um servidor GraphQL. Começamos compreendendo totalmente os resolvers. Lidamos com consultas e mutações. Adicionamos uma autorização do GitHub. Identificamos o usuário atual por meio de um token de acesso adicionado ao cabeçalho de todas as requisições. Por fim, modificamos a mutação que lê o usuário do contexto do resolver e permitimos que os usuários postem fotos.

Se quiser executar uma versão completa do serviço que implementamos neste capítulo, você poderá encontrá-la no repositório deste livro (<https://github.com/MoonHighway/learning-graphql/tree/master/chapter-05/photo-share-api/>). Essa aplicação deverá saber qual banco de dados deve usar e as credenciais de OAuth do GitHub. Esses valores podem ser acrescentados criando um novo arquivo chamado `.env` e colocando-o na raiz do projeto:

```
DB_HOST=<SEU_HOST_MONGODB>  
CLIENT_ID=<SEU_CLIENT_ID_GITHUB>  
CLIENT_SECRET=<SEU_CLIENT_SECRET_GITHUB>
```

Com o arquivo `.env` definido, você estará pronto para instalar as dependências: `yarn` ou `npm install`, e executar o serviço: `yarn start` ou `npm start`. Depois que o serviço estiver executando na porta 4000, você poderá enviar consultas a ele usando o Playground em `http://localhost:4000/playground`. Um código GitHub pode ser requisitado clicando no link que se encontra em `http://localhost:4000`. Se quiser acessar o endpoint do GraphQL a partir de outro cliente, você poderá encontrá-lo em `http://localhost:4000/graphql`.

No Capítulo 7 mostraremos como modificar essa API para que trate

subscriptions e uploads de arquivos. Antes de fazer isso, porém, temos que mostrar como os clientes consumirão essa API; portanto, no Capítulo 6, veremos como implementar um frontend que funcione com esse serviço.

¹ Você pode criar uma conta em <https://www.github.com>.

CAPÍTULO 6

Clientes GraphQL

Com seu servidor GraphQL implementado, é hora de configurar o GraphQL do lado cliente. De forma bem genérica, um cliente é somente uma aplicação que se comunica com um servidor. Por causa da flexibilidade do GraphQL, não há nenhuma prescrição sobre como implementar um cliente. Você pode estar implementando aplicações para navegadores web. Pode estar criando aplicações nativas para telefones. Pode estar implementando um serviço GraphQL para a tela de seu refrigerador. Para o cliente, também não importa a linguagem em que o serviço está escrito.

Tudo que você realmente precisa para enviar consultas e mutações é a capacidade de enviar uma requisição HTTP. Quando o serviço responder com alguns dados, você poderá usá-los em seu cliente, não importa qual seja esse cliente.

Usando uma API GraphQL

O modo mais fácil de começar é simplesmente fazendo uma requisição HTTP para o seu endpoint GraphQL. Para testar o servidor que implementamos no Capítulo 5, certifique-se de que seu serviço esteja executando localmente em *<http://localhost:4000/graphql>*. Você também pode ver todos esses exemplos executando no CodeSandbox, nos links que se encontram no repositório do Capítulo 6 (*<https://github.com/MoonHighway/learning-graphql/tree/master/chapter-06>*).

Requisições fetch

Como vimos no Capítulo 3, é possível enviar requisições para um serviço GraphQL usando o cURL. Tudo que precisamos são de alguns valores diferentes:

- Uma consulta: `{totalPhotos, totalUsers}`
- Um endpoint para GraphQL: `http://localhost:4000/graphql`
- Um tipo de conteúdo: `Content-Type: application/json`

Com isso, enviamos a requisição cURL diretamente do terminal/prompt de comandos usando o método POST:

```
curl -X POST \
  -H "Content-Type: application/json" \
  --data '{"query": "{totalUsers, totalPhotos}" }' \
  http://localhost:4000/graphql
```

Se enviarmos essa requisição, deveríamos ver os resultados corretos, `{"data":{"totalUsers":7,"totalPhotos":4}}`, na forma de dados JSON devolvidos para o terminal. Seus números para `totalUsers` e `totalPhotos` refletirão os dados atuais. Se seu cliente for um shell script, você poderá começar implementando esse script com o cURL.

Como estamos usando o cURL, podemos utilizar qualquer código que envie uma requisição HTTP. Poderíamos implementar um cliente minúsculo usando `fetch`, que funcionará no navegador:

```
var query = `{totalPhotos, totalUsers}`
var url = 'http://localhost:4000/graphql'

var opts = {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
}

fetch(url, opts)
  .then(res => res.json())
  .then(console.log)
  .catch(console.error)
```

Depois de buscar os dados, veremos o resultado esperado exibido

no console:

```
{
  "data": {
    "totalPhotos": 4,
    "totalUsers": 7
  }
}
```

Podemos usar os dados resultantes no cliente para construir aplicações. Consideremos um exemplo básico para ver como podemos listar `totalUsers` e `totalPhotos` diretamente no DOM:

```
fetch(url, opts)
  .then(res => res.json())
  .then(({data}) => `
    <p>photos: ${data.totalPhotos}</p>
    <p>users: ${data.totalUsers}</p>
  `)
  .then(text => document.body.innerHTML = text)
  .catch(console.error)
```

Em vez de apresentar os resultados no console, usamos os dados para construir um texto HTML. Podemos então tomar esse texto e escrevê-lo diretamente no corpo do documento. Tome cuidado: é possível sobrescrever qualquer dado que estiver no corpo após a requisição ser concluída.

Se você já sabe como enviar requisições HTTP usando o seu cliente favorito, já terá as ferramentas necessárias para implementar uma aplicação cliente que se comunique com qualquer API GraphQL.

graphql-request

Embora `cURL` e `fetch` funcionem bem, há outros frameworks que podem ser usados para enviar operações GraphQL para uma API. Um dos exemplos mais dignos de destaque é o `graphql-request`. O `graphql-request` encapsula requisições de busca em uma *promise* (promessa), que pode ser usada para fazer requisições ao servidor GraphQL. Ele também cuida dos detalhes de fazer a requisição e o parse dos dados para você.

Para começar a usar o `graphql-request`, é necessário instalá-lo antes:

```
npm install graphql-request
```

Em seguida, importe e use o módulo como `request`. Não se esqueça de manter o serviço de fotos executando na porta 4000:

```
import { request } from 'graphql-request'

var query = `
  query listUsers {
    allUsers {
      name
      avatar
    }
  }
`

request('http://localhost:4000/graphql', query)
  .then(console.log)
  .catch(console.error)
```

A função de requisição aceita `url` e `query`, faz a requisição ao servidor e devolve os dados em uma só linha de código. Os dados devolvidos, conforme esperado, são uma resposta JSON com todos os usuários:

```
{
  "allUsers": [
    { "name": "sharon adams", "avatar": "http://..." },
    { "name": "sarah ronau", "avatar": "http://..." },
    { "name": "paul young", "avatar": "http://..." },
  ]
}
```

Podemos começar a usar esses dados em nosso cliente imediatamente.

Também é possível enviar mutações com o `graphql-request`:

```
import { request } from 'graphql-request'

var url = 'http://localhost:4000/graphql'

var mutation = `
  mutation populate($count: Int!) {
```



```

    addFakeUsers(count:$count) {
      id
      name
    }
  }
}

```

```
var variables = { count: 3 }
```

```

request(url, mutation, variables)
  .then(console.log)
  .catch(console.error)

```

A função `request` aceita o URL da API, a mutação e um terceiro argumento para as variáveis. Esse é somente um objeto JavaScript que passa um campo e o valor para as variáveis de consulta. Depois de chamar `request`, executamos a mutação `addFakeUsers`.

Embora o `graphql-request` não ofereça nenhuma integração formal com bibliotecas e frameworks de UI, é possível incorporar uma biblioteca de modo razoavelmente simples. Vamos carregar alguns dados em um componente React usando o `graphql-request`, como vemos no Exemplo 6.1.

Exemplo 6.1 – Requisição GraphQL e React

```

import React from 'react'
import ReactDOM from 'react-dom'
import { request } from 'graphql-request'

```

```
var url = 'http://localhost:4000/graphql'
```

```

var query = `
  query listUsers {
    allUsers {
      avatar
      name
    }
  }
`

```

```

var mutation = `
  mutation populate($count: Int!) {

```

```

      addFakeUsers(count:$count) {
        githubLogin
      }
    }
  }
}

const App = ({ users=[] }) =>
  <div>
    {users.map(user =>
      <div key={user.githubLogin}>
        <img src={user.avatar} alt="" />
        {user.name}
      </div>
    )}
    <button onClick={addUser}>Add User</button>
  </div>

const render = ({ allUsers=[] }) =>
  ReactDOM.render(
    <App users={allUsers} />,
    document.getElementById('root')
  )

const addUser = () =>
  request(url, mutation, {count:1})
    .then(requestAndRender)
    .catch(console.error)

const requestAndRender = () =>
  request(url, query)
    .then(render)
    .catch(console.error)

requestAndRender()

```

Nosso arquivo começa com uma importação tanto do React quanto do ReactDOM. Criamos então um componente App. App mapeia os users que são passados como propriedades e cria elementos div contendo avatar e username. A função render renderiza App para o elemento #root e passa allUsers como uma propriedade.

Em seguida, requestAndRender chama request do graphql-request. A consulta

é executada, os dados são recebidos e então `render` é chamado, o qual fornece os dados para o componente `App`.

Essa pequena aplicação também trata mutações. No componente `App`, o botão tem um evento `onClick` que chama a função `addUser`. Quando chamada, ela envia a mutação e então chama `requestAndRender` para fazer uma nova requisição para os usuários do serviço, e renderiza novamente `<App />` com a nova lista de usuários.

Até agora vimos algumas maneiras diferentes de começar a implementar aplicações clientes usando o GraphQL. Podemos escrever shell scripts com `cURL`. Podemos construir páginas web com `fetch`. Podemos implementar aplicações um pouco mais rapidamente com o `graphql-request`. Poderíamos parar por aqui se quiséssemos, mas há clientes GraphQL mais eficazes ainda disponíveis. Vamos conhecê-los.

Apollo Client

Uma grande vantagem de usar REST (Representational State Transfer, ou Transferência de Estado Representativo) é a facilidade com que podemos lidar com caching. Com o REST, podemos salvar os dados de resposta de uma requisição em um cache na URL que foi usada para acessar essa requisição. Caching feito, sem problemas. O caching com o GraphQL é um pouco mais complicado. Não temos uma variedade de rotas em uma API GraphQL – tudo é enviado e recebido por meio de um único endpoint, portanto não podemos simplesmente salvar os dados devolvidos por uma rota na URL usada para requisitá-los.

Para construir uma aplicação robusta, com bom desempenho, precisamos de uma maneira de fazer cache de consultas e de seus objetos resultantes. Ter uma solução de caching local é essencial, pois nos esforçamos constantemente para criar aplicações rápidas e eficientes. Poderíamos criar algo desse tipo por conta própria, ou poderíamos contar com um dos clientes consolidados já existentes.

As soluções mais proeminentes de clientes GraphQL disponíveis hoje em dia são o Relay e o Apollo Client. O Relay foi lançado com código aberto pelo Facebook em 2015, na mesma época que o GraphQL. Ele reúne tudo que o Facebook aprendeu sobre o uso do GraphQL em ambiente de produção. O Relay é compatível somente com o React e o React Native, o que significa que havia uma oportunidade para criar um cliente GraphQL com suporte aos desenvolvedores que não usassem o React.

Entra em cena o Apollo Client. Disponibilizado pelo Meteor Development Group, o Apollo Client é um projeto voltado para a comunidade, cujo objetivo é implementar uma solução de cliente GraphQL flexível, capaz de lidar com tarefas como caching, atualizações de UI otimistas¹ e outras. A equipe criou pacotes que oferecem vínculos (bindings) para React, Angular, Ember, Vue, iOS e Android.

Já usamos várias ferramentas da equipe do Apollo no servidor, mas o Apollo Client tem como foco específico enviar e receber requisições do cliente para o servidor. Ele trata requisições de rede com o Apollo Link e cuida de tudo que está associado a caching com o Apollo Cache. O Apollo Client então encapsula o link e o cache, e administra todas as interações com o serviço GraphQL de modo eficaz.

No restante do capítulo, veremos o Apollo Client com mais detalhes. Usaremos o React para construir nossos componentes de UI, mas é possível aplicar muitas das técnicas descritas neste capítulo em projetos que utilizem bibliotecas e frameworks diferentes.

Apollo Client com o React

Como trabalhar com o React é o que nos levou ao GraphQL antes de tudo, escolhemos o React como a biblioteca de interface do usuário. Não demos muitas explicações sobre o React em si. É uma biblioteca criada pelo Facebook que usa uma arquitetura baseada em componentes para compor UIs. Se você é usuário de outra

biblioteca e não quiser olhar para o React novamente depois disso, tudo bem. As ideias apresentadas na próxima seção são aplicáveis a outros frameworks de UI.

Configuração do projeto

Neste capítulo mostramos como implementar uma aplicação React que interaja com um serviço GraphQL usando o Apollo Client. Para começar, precisamos criar a estrutura do frontend deste projeto usando `create-react-app`. O `create-react-app` permite gerar um projeto React completo sem definir nenhuma configuração de construção (build configuration). Caso ainda não tenha usado o `create-react-app`, talvez seja necessário instalá-lo:

```
npm install -g create-react-app
```

Uma vez instalado, você poderá criar um projeto React em qualquer local de seu computador executando:

```
create-react-app photo-share-client
```

Esse comando instala uma nova aplicação React básica em uma pasta chamada *photo-share-client*. Ele adiciona e instala automaticamente tudo que será necessário para começar a construir uma aplicação React. Para iniciar a aplicação, acesse a pasta `photo-share-client` e execute `npm start`. Você verá o seu navegador abrir com o endereço `http://localhost:3000`, que é o local em que sua aplicação React cliente está executando. Lembre-se de que você pode encontrar todos os arquivos deste capítulo no repositório em <http://github.com/moonhighway/learning-graphql>.

Configuração do Apollo Client

Será necessário instalar alguns pacotes para construir um cliente GraphQL com as ferramentas Apollo. Em primeiro lugar, você precisará do `graphql`, que inclui o parser da linguagem GraphQL. Em seguida, instale um pacote chamado `apollo-boost`. O Apollo Boost inclui os pacotes Apollo necessários para criar um Apollo Client e enviar operações para esse cliente. Por fim, precisaremos do `react-apollo`. O

React Apollo é uma biblioteca npm que contém componentes do React; estes serão usados para implementar uma interface de usuário com o Apollo.

Vamos instalar esses três pacotes ao mesmo tempo:

```
npm install graphql apollo-boost react-apollo
```

Agora estamos prontos para criar o nosso cliente. O construtor `ApolloClient` que se encontra em `apollo-boost` pode ser usado para criar o nosso primeiro cliente. Abra o arquivo `src/index.js` e substitua o código que está nesse arquivo pelo seguinte:

```
import ApolloClient from 'apollo-boost'
```

```
const client = new ApolloClient({ uri: 'http://localhost:4000/graphql' })
```

Criamos uma nova instância `client` usando o construtor `ApolloClient`. `client` está pronto para lidar com todas as comunicações de rede, com o serviço GraphQL hospedado em `http://localhost:4000/graphql`. Por exemplo, podemos usar o cliente para enviar uma consulta ao serviço PhotoShare:

```
import ApolloClient, { gql } from 'apollo-boost'
```

```
const client = new ApolloClient({ uri: 'http://localhost:4000/graphql' })
```

```
const query = gql`  
  {  
    totalUsers  
    totalPhotos  
  }  
`
```

```
client.query({query})  
  .then(({ data }) => console.log('data', data))  
  .catch(console.error)
```

Esse código usa `client` para enviar uma consulta a fim de saber o número total de fotos e o número total de usuários. Para que isso aconteça, importamos a função `gql` do `apollo-boost`. Essa função faz parte do pacote `graphql-tag`, automaticamente incluído no `apollo-boost`. A função `gql` é usada para fazer parse de uma consulta e gerar uma

AST (Abstract Syntax Tree, ou Árvore Sintática Abstrata).

Podemos enviar a AST para o cliente chamando `client.query({query})`. Esse método devolve uma promise (promessa). Ele envia a consulta na forma de uma requisição HTTP para o nosso serviço GraphQL e resolve os dados devolvidos por esse serviço. No exemplo anterior, a resposta é apresentada no console:

```
{ totalUsers: 4, totalPhotos: 7, Symbol(id): "ROOT_QUERY" }
```



O serviço GraphQL deve estar executando

Certifique-se de que o serviço GraphQL continua executando em `http://localhost:4000` para que você possa testar a conexão do cliente com o servidor.

Além de tratar todas as requisições de rede para o nosso serviço GraphQL, o cliente também faz cache das respostas localmente na memória. A qualquer momento podemos consultar o cache chamando `client.extract()`:

```
console.log('cache', client.extract())
client.query({query})
  .then(() => console.log('cache', client.extract()))
  .catch(console.error)
```

Nesse exemplo vimos o cache antes de a consulta ser enviada, e vimos novamente após a consulta ter sido resolvida. Note que agora temos os resultados salvos em um objeto local, administrado pelo cliente:

```
{
  ROOT_QUERY: {
    totalPhotos: 4,
    totalUsers: 7
  }
}
```

Na próxima vez que enviarmos uma consulta para esses dados ao cliente, ele os lerá do cache, em oposição a enviar outra requisição de rede para o nosso serviço. O Apollo Client nos oferece opções para especificar quando e com qual frequência devemos enviar requisições HTTP pela rede. Descreveremos essas opções mais adiante neste capítulo. Por enquanto, é importante compreender

que o Apollo Client é usado para tratar todas as requisições de rede para o nosso serviço GraphQL. Além do mais, por padrão, ele faz cache dos resultados localmente de modo automático e passa a responsabilidade para o cache local a fim de melhorar o desempenho de nossas aplicações.

Para começar a trabalhar com o `react-apollo`, tudo que precisamos fazer é criar um cliente e adicioná-lo em nossa interface de usuário com um componente chamado `ApolloProvider`. Substitua o código que se encontra no arquivo `index.js` pelo código a seguir:

```
import React from 'react'
import { render } from 'react-dom'
import App from './App'
import { ApolloProvider } from 'react-apollo'
import ApolloClient from 'apollo-boost'

const client = new ApolloClient({ uri: 'http://localhost:4000/graphql' })

render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  document.getElementById('root')
)
```

Esse é todo o código que você precisará para começar a usar o Apollo com o React. Nesse caso, criamos um cliente e então nós o colocamos no escopo global do React com a ajuda de um componente chamado `ApolloProvider`. Qualquer componente filho encapsulado pelo `ApolloProvider` terá acesso ao cliente. Isso significa que o componente `<App />` e qualquer um de seus filhos estão prontos para receber dados de nosso serviço GraphQL usando o Apollo Client.

O componente Query

Ao usar o Apollo Client, precisamos de uma maneira de lidar com consultas para buscar dados a serem carregados em nossa UI com o React. O componente `Query` cuidará de buscar os dados, tratará o

estado da carga e atualizará nossa UI. Esse componente pode ser usado em qualquer lugar no ApolloProvider. O componente envia uma query usando o cliente. Depois de resolvê-la, o cliente devolverá o resultado que usaremos na construção da interface de usuário.

Abra o arquivo `src/App.js` e substitua o código que está nesse arquivo atualmente pelo seguinte:

```
import React from 'react'
import Users from './Users'
import { gql } from 'apollo-boost'

export const ROOT_QUERY = gql`
  query allUsers {
    totalUsers
    allUsers {
      githubLogin
      name
      avatar
    }
  }
`

const App = () => <Users />

export default App
```

No componente `App`, criamos uma consulta chamada `ROOT_QUERY`. Lembre-se de que uma das vantagens de usar o GraphQL é poder requisitar tudo que você precisará para construir sua UI e receber todos esses dados em uma única resposta. Isso significa que pediremos tanto o contador `totalUsers` quanto o array `allUsers` em uma consulta que criamos na raiz de nossa aplicação. Usando a função `gql`, convertemos nossa string de consulta em um objeto AST chamado `ROOT_QUERY` e exportamos esse objeto para que outros componentes possam usá-lo.

Nesse ponto você deverá ver um erro. Isso ocorre porque dissemos a `App` para renderizar um componente que ainda não criamos. Crie novo arquivo chamado `src/Users.js` e insira o código a seguir nesse arquivo:

```
import React from 'react'
import { Query } from 'react-apollo'
import { ROOT_QUERY } from './App'

const Users = () =>
  <Query query={ROOT_QUERY}>
    {result =>
      <p>Users are loading: {result.loading ? "yes" : "no"}</p>
    }
  </Query>

export default Users
```

Agora você perceberá que o erro desaparecerá e a mensagem “Users are loading: no” (Usuários carregando: não) deverá ser exibida na janela do navegador. Internamente, o componente Query está enviando a ROOT_QUERY para o nosso serviço GraphQL e fazendo cache do resultado localmente. Obtemos o resultado usando uma técnica do React chamada props. A renderização de props nos permite passar propriedades como argumentos de função para os componentes filhos. Observe que estamos obtendo `result` de uma função e devolvendo um elemento de parágrafo.

O resultado contém mais informações além dos dados da resposta. Ele nos informará se uma operação está sendo carregada ou não por meio da propriedade `result.loading`. No exemplo anterior, podemos dizer ao usuário se a consulta atual está sendo carregada.



Deixando a requisição HTTP mais lenta

Talvez sua rede seja rápida demais e você não veja nada além de um flicker rápido da propriedade de carregamento no navegador. Você pode usar a aba Network (Rede) nas ferramentas do desenvolvedor do Chrome para deixar a requisição HTTP mais lenta. Nas ferramentas do desenvolvedor, você encontrará um menu suspenso com a opção “Online” selecionada. Se selecionar “Slow 3G” (3G lento) no menu, você simulará uma resposta mais lenta. Isso permitirá que você veja a carga ocorrendo no navegador.

Depois que os dados forem carregados, eles serão passados com o resultado.

Em vez de exibir “yes” (sim) ou “no” (não) quando o cliente estiver carregando dados, podemos exibir componentes de UI em seu

lugar. Vamos fazer alguns ajustes no arquivo `Users.js`:

```
const Users = () =>
  <Query query={ROOT_QUERY}>
    {{ { data, loading } } => loading ?
      <p>loading users...</p> :
      <UserList count={data.totalUsers} users={data.allUsers} />
    }
  </Query>
```

```
const UserList = ({ count, users }) =>
  <div>
    <p>{count} Users</p>
    <ul>
      {users.map(user =>
        <UserListItem key={user.githubLogin}
          name={user.name}
          avatar={user.avatar} />
      )}
    </ul>
  </div>
```

```
const UserListItem = ({ name, avatar }) =>
  <li>
    <img src={avatar} width={48} height={48} alt="" />
    {name}
  </li>
```

Se o cliente estiver carregando (`loading`) a consulta atual, exibiremos uma mensagem “loading users...” (carregando usuários...). Se os dados forem carregados, passaremos o contador do total de usuários, junto com um array contendo os campos `name`, `githubLogin` e `avatar` de cada usuário para o componente `UserList`: exatamente os dados que pedimos em nossa consulta. `UserList` utiliza os dados do resultado para construção da UI. Ele exibe o contador, além de uma lista que mostra a imagem do avatar do usuário e seu nome.

O objeto `results` também tem várias funções utilitárias para paginação, novas buscas (`refetching`) e `polling`. Vamos usar a função `refetch` para buscar novamente a lista de usuários quando clicamos em um botão:

```
const Users = () =>
  <Query query={ROOT_QUERY}>
    {({ data, loading, refetch }) => loading ?
      <p>loading users...</p> :
      <UserList count={data.totalUsers}
        users={data.allUsers}
        refetchUsers={refetch} />
    }
  </Query>
```

Nesse exemplo, temos uma função que pode ser usada para `refetch` da `ROOT_QUERY` ou para requisitar os dados do servidor novamente. A propriedade `refetch` é apenas uma função. Podemos passá-la para `UserList`, e ela poderá ser associada a um clique de botão:

```
const UserList = ({ count, users, refetch }) =>
  <div>
    <p>{count} Users</p>
    <button onClick={() => refetch()}>Refetch</button>
    <ul>
      {users.map(user =>
        <UserListItem key={user.githubLogin}
          name={user.name}
          avatar={user.avatar} />
      )}
    </ul>
  </div>
```

Em `UserList`, usamos a função `refetch` para requisitar os mesmos dados da raiz de nosso serviço GraphQL. Sempre que clicar no botão “Refetch Users” (Buscar usuários novamente), outra consulta será enviada ao endpoint do GraphQL para buscar quaisquer mudanças de dados novamente. Essa é uma maneira de manter sua interface de usuário em sincronia com os dados do servidor.



Para testar isso, podemos alterar os dados do usuário após a busca inicial. Você pode apagar a coleção de usuários, apagar os documentos de usuários diretamente no MongoDB ou adicionar usuários simulados enviando uma consulta com o GraphQL Playground do servidor. Ao alterar os dados no banco de dados, o botão “Refetch Users” terá que ser clicado a fim de renderizar novamente os dados mais recentes no navegador.

O polling é outra opção disponibilizada pelo componente `Query`.

Quando adicionamos a propriedade `pollInterval` no componente `Query`, os dados são automaticamente buscados repetidamente com base em um intervalo especificado:

```
<Query query={ROOT_QUERY} pollInterval={1000}>
```

Definir um `pollInterval` faz com que os dados sejam buscados novamente em um instante especificado, de modo automático. Nesse caso, buscaremos os dados do servidor novamente a cada segundo. Tome cuidado ao usar o polling, pois esse código, na verdade, envia uma nova requisição de rede a cada segundo.

Além de `loading`, `data` e `refetch`, o objeto de resposta tem algumas opções adicionais:

`stopPolling`

Uma função que para o polling.

`startPolling`

Uma função que inicia o polling.

`fetchMore`

Uma função que pode ser usada para buscar a próxima página de dados.

Antes de prosseguir, remova a propriedade `pollInterval` do componente `Query`. Não queremos que haja um polling enquanto continuamos com a iteração nesse exemplo.

Componente Mutation

Se quisermos enviar mutações para o serviço GraphQL, o componente `Mutation` pode ser usado. No próximo exemplo, utilizaremos esse componente para tratar a mutação `addFakeUsers`. Quando enviamos essa mutação, escrevemos a nova lista de usuários diretamente no cache.

Para começar, vamos importar o componente `Mutation` e adicionar uma mutação no arquivo `Users.js`:

```
import { Query, Mutation } from 'react-apollo'
import { gql } from 'apollo-boost'
```

...

```
const ADD_FAKE_USERS_MUTATION = gql`
  mutation addFakeUsers($count: Int!) {
    addFakeUsers(count: $count) {
      githubLogin
      name
      avatar
    }
  }
`
```

Depois que tivermos a mutação, ela poderá ser usada em conjunto com o componente `Mutation`. Esse componente passará uma função para seus filhos por meio das propriedades de renderização. Essa função pode ser usada para enviar a mutação quando estivermos prontos:

```
const UserList = ({ count, users, refetchUsers }) =>
  <div>
    <p>{count} Users</p>
    <button onClick={() => refetchUsers()}>Refetch Users</button>
    <Mutation mutation={ADD_FAKE_USERS_MUTATION} variables={{ count: 1 }}>
      {addFakeUsers =>
        <button onClick={addFakeUsers}>Add Fake Users</button>
      }
    </Mutation>
    <ul>
      {users.map(user =>
        <UserListItem key={user.githubLogin}
          name={user.name}
          avatar={user.avatar} />
      )}
    </ul>
  </div>
```

Do mesmo modo como enviamos `query` como uma propriedade do componente `query`, enviaremos uma propriedade `mutation` para o componente `Mutation`. Observe também que estamos usando a propriedade `variables`. Ela enviará as variáveis de consulta necessárias à mutação. Nesse caso, o contador é definido com 1, o que fará a mutação adicionar um usuário simulado de cada vez. O

componente `Mutation` usa uma função `addFakeUsers`, que enviará a mutação depois que for chamada. Quando o usuário clicar no botão “Add Fake Users” (Adicionar usuários simulados), a mutação será enviada para a nossa API.

No momento, esses usuários estão sendo adicionados no banco de dados, mas a única maneira de ver as alterações é clicando no botão “Refetch Users” (Buscar usuários novamente). Podemos dizer para o componente `Mutation` que faça buscas novamente com consultas específicas, depois que a mutação tiver sido concluída, em vez de esperar que nossos usuários cliquem em um botão:

```
<Mutation mutation={ADD_FAKE_USERS_MUTATION}
  variables={{ count: 1 }}
  refetchQueries={[{ query: ROOT_QUERY }]}>
  {addFakeUsers =>
    <button onClick={addFakeUsers}>Add Fake Users</button>
  }
</Mutation>
```

`refetchQueries` é uma propriedade que permite especificar quais consultas devem ser refeitas após o envio de uma mutação. Basta definir uma lista de objetos que contêm consultas. Cada uma das operações de consulta que estiver nessa lista fará uma nova busca de dados após a mutação ter sido concluída.

Autorização

No Capítulo 5 implementamos uma mutação para autorizar um usuário com o GitHub. Na próxima seção, mostraremos como configurar a autorização de usuário do lado cliente.

O processo de autorizar um usuário envolve vários passos. Os passos em **negrito** indicam as funcionalidades que serão acrescentadas no cliente:

Cliente

Redireciona o usuário para o GitHub com o `client_id`.

Usuário

Permite acesso a informações de conta no GitHub para a aplicação cliente.

GitHub

Redireciona de volta para o site com o código: `http://localhost:3000?code=XYZ`

Cliente

Envia a mutação GraphQL `authUser(code)` com o código.

API

Requisita um `access_token` do GitHub com `client_id`, `client_secret` e `client_code`.

GitHub

Responde com o `access_token` que poderá ser usado em futuras requisições de informações.

API

Requisita informações de usuário com o `access_token`.

GitHub

Responde com informações sobre o usuário: `name`, `github_login`, `avatar_url`.

API

Resolve a mutação `authUser(code)` com `AuthPayload`, que contém um token e o usuário.

Cliente

Salva o token para que seja enviado em requisições GraphQL no futuro.

Autorizando o usuário

Agora é hora de autorizar o usuário. Para facilitar este exemplo, usaremos o React Router, que instalamos com: `npm install react-router-dom`.

Vamos modificar nosso componente `<App />` principal.

Incorporaremos o `BrowserRouter` e adicionaremos um novo componente, `AuthorizedUser`, que poderá ser usado para autorizar usuários com o GitHub:

```
import React from 'react'
import Users from './Users'
import { BrowserRouter } from 'react-router-dom'
import { gql } from 'apollo-boost'
import AuthorizedUser from './AuthorizedUser'
```

```
export const ROOT_QUERY = gql`
  query allUsers {
    totalUsers
    allUsers { ...userInfo }
    me { ...userInfo }
  }
`
```

```
fragment userInfo on User {
  githubLogin
  name
  avatar
}
```

```
const App = () =>
  <BrowserRouter>
    <div>
      <AuthorizedUser />
      <Users />
    </div>
  </BrowserRouter>
```

```
export default App
```

O `BrowserRouter` encapsula todos os demais componentes que queremos renderizar. Adicionamos também um novo componente `AuthorizedUser`, que será implementado em um novo arquivo. Devemos ver um erro até que esse componente seja adicionado.

Também modificamos a `ROOT_QUERY` para que esteja preparada para a autorização. Estamos pedindo o campo `me` agora, que devolve informações sobre o usuário atual, quando houver alguém logado.

Se não houver um usuário logado, esse campo simplesmente devolverá `null`. Observe que acrescentamos um fragmento chamado `userInfo` no documento de consulta. Isso nos permite obter as mesmas informações sobre um `User` em dois lugares: no campo `me` e no campo `allUsers`.

O componente `AuthorizedUser` deve redirecionar o usuário para o GitHub a fim de requisitar um código. Esse código deve ser passado de volta, do GitHub para a nossa aplicação, em `http://localhost:3000`.

Em um novo arquivo chamado `AuthorizedUser.js`, vamos implementar este processo:

```
import React, { Component } from 'react'
import { withRouter } from 'react-router-dom'

class AuthorizedUser extends Component {

  state = { signingIn: false }

  componentDidMount() {
    if (window.location.search.match(/code=/)) {
      this.setState({ signingIn: true })
      const code = window.location.search.replace("?code=", "")
      alert(code)
      this.props.history.replace('/')
    }
  }

  requestCode() {
    var clientID = <YOUR_GITHUB_CLIENT_ID>
    window.location =
      `https://github.com/login/oauth/authorize?client_id=${clientID}&scope=user`
  }

  render() {
    return (
      <button onClick={this.requestCode} disabled={this.state.signingIn}>
        Sign In with GitHub
      </button>
    )
  }
}
```

```
}
```

```
export default withRouter(AuthorizedUser)
```

O componente `AuthorizedUser` renderiza um botão “Sign In with GitHub” (Login com o GitHub). Depois de clicado, esse botão redirecionará o usuário para o processo OAuth do GitHub. Feita a autorização, o GitHub passará um código de volta ao navegador: `http://localhost:3000?code=XYZGNARLYSENDABC`. Se o código for encontrado na string de consulta, o componente fará seu parse na barra de endereço da janela e o exibirá ao usuário em uma caixa de alerta antes de removê-lo com a propriedade `history`, enviada para esse componente com o React Router.

Em vez de enviar um alerta com o código do GitHub ao usuário, devemos enviá-lo para a mutação `githubAuth`:

```
import { Mutation } from 'react-apollo'
import { gql } from 'apollo-boost'
import { ROOT_QUERY } from './App'

const GITHUB_AUTH_MUTATION = gql`
  mutation githubAuth($code:String!) {
    githubAuth(code:$code) { token }
  }
`
```

A mutação anterior será usada para autorizar o usuário atual. Tudo que precisaremos é do código. Vamos adicionar esta mutação no método `render` desse componente:

```
render() {
  return (
    <Mutation mutation={GITHUB_AUTH_MUTATION}
      update={this.authorizationComplete}
      refetchQueries={[{ query: ROOT_QUERY }]}>

      {mutation => {
        this.githubAuthMutation = mutation
        return (
          <button
            onClick={this.requestCode}
            disabled={this.state.signingIn}>
```

```

        Sign In with GitHub
      </button>
    )
  }}
</Mutation>
)
}

```

O componente `Mutation` está vinculado a `GITHUB_AUTH_MUTATION`. Depois de concluído, ele chamará o método `authorizationComplete` do componente e fará uma nova busca de `ROOT_QUERY`. A função de mutação foi adicionada no escopo do componente `AuthorizedUser` com a definição: `this.githubAuthMutation = mutation`. Podemos agora chamar essa função `this.githubAuthMutation()` quando estivermos prontos (quando tivermos um código).

Em vez de informar o código com um alerta, nós o enviaremos junto com a mutação para autorizar o usuário atual. Depois que ele for autorizado, salvaremos o token resultante em `localStorage` e usaremos a propriedade `history` do roteador para remover o código do endereço da janela:

```

class AuthorizedUser extends Component {

  state = { signingIn: false }

  authorizationComplete = (cache, { data }) => {
    localStorage.setItem('token', data.githubAuth.token)
    this.props.history.replace('/')
    this.setState({ signingIn: false })
  }

  componentDidMount() {
    if (window.location.search.match(/code=/)) {
      this.setState({ signingIn: true })
      const code = window.location.search.replace("?code=", "")
      this.githubAuthMutation({ variables: { code } })
    }
  }

  ...
}

```

Para iniciar o processo de autorização, chame `this.githubAuthMutation()` e adicione `code` nas `variables` da operação. Feito isso, o método `authorizationComplete` será chamado. Os dados em `data` passados para esse método são aqueles que selecionamos na mutação. Esses dados contêm um `token`. Salvaremos o `token` localmente e usaremos a `history` do `React Router` para remover a string de consulta do código da barra de endereço da janela.

Nesse ponto, teremos o usuário atual logado com o GitHub. O próximo passo será garantir que enviaremos esse `token` em todas as requisições nos cabeçalhos `HTTP`.

Identificando o usuário

Nossa próxima tarefa é adicionar um `token` no cabeçalho de autorização de cada requisição. Lembre-se de que o serviço `photo-share-api` que criamos no último capítulo identificará usuários que passarem um `token` de autorização no cabeçalho. Tudo que temos que fazer é garantir que qualquer `token` salvo em `localStorage` seja enviado em toda requisição para o nosso serviço `GraphQL`.

Vamos modificar o arquivo `src/index.js`. Precisamos encontrar a linha em que criamos o `Apollo Client` e substituí-la pelo código a seguir:

```
const client = new ApolloClient({
  uri: 'http://localhost:4000/graphql',
  request: operation => {
    operation.setContext(context => ({
      headers: {
        ...context.headers,
        authorization: localStorage.getItem('token')
      }
    }))
  }
})
```

Adicionamos agora um método de requisição na configuração de nosso `Apollo Client`. Esse método passa os detalhes de toda `operation` imediatamente antes de ela ser enviada ao serviço `GraphQL`. Nesse caso, estamos definindo o contexto de toda `operation` para que inclu

um cabeçalho `authorization` contendo o token salvo na área de armazenagem local. Não se preocupe, se não tivermos um token salvo, o valor desse cabeçalho simplesmente será `null` e nosso serviço suporá que há um usuário não autorizado.

Agora que adicionamos o token de autorização em todos os cabeçalhos, nosso campo `me` deverá devolver os dados sobre o usuário atual. Vamos exibir esses dados em nossa UI. Localize o método `render` no componente `AuthorizedUser` e substitua-o pelo código a seguir:

```
render() {
  return (
    <Mutation
      mutation={GITHUB_AUTH_MUTATION}
      update={this.authorizationComplete}
      refetchQueries={[{ query: ROOT_QUERY }]}>
      {mutation => {
        this.githubAuthMutation = mutation
        return (
          <Me signingIn={this.state.signingIn}
            requestCode={this.requestCode}
            logout={() => localStorage.removeItem('token')} />
        )
      }}
    </Mutation>
  )
}
```

Em vez de renderizar um botão, esse componente `Mutation` agora renderiza um componente chamado `Me`. O componente exibirá informações sobre o usuário atual logado ou o botão de autorização. Ele terá que saber se o usuário está no processo de login no momento. Também deverá acessar o método `requestCode` do componente `AuthorizedUser`. Por fim, temos que disponibilizar uma função capaz de fazer logout do usuário atual. Por enquanto, simplesmente removeremos o token de `localStorage` quando o usuário fizer logout. Todos esses valores foram passados para o componente `Me` na forma de propriedades.

É hora de criar o componente `Me`. Adicione o código a seguir antes

da declaração do componente `AuthorizedUser`:

```
const Me = ({ logout, requestCode, signingIn }) =>
  <Query query={ROOT_QUERY}>
    {{{ loading, data }} => data.me ?
      <CurrentUser {...data.me} logout={logout} /> :
      loading ?
        <p>loading... </p> :
        <button
          onClick={requestCode}
          disabled={signingIn}>
            Sign In with GitHub
          </button>
        </Query>

const CurrentUser = ({ name, avatar, logout }) =>
  <div>
    <img src={avatar} width={48} height={48} alt="" />
    <h1>{name}</h1>
    <button onClick={logout}>logout</button>
  </div>
```

O componente `Me` renderiza um componente `Query` a fim de obter dados sobre o usuário atual a partir da `ROOT_QUERY`. Se houver um token, o campo `me` da `ROOT_QUERY` não será `null`. No componente de consulta, verificamos se `data.me` é `null`. Se houver um dado nesse campo, exibiremos o componente `CurrentUser` e passaremos os dados sobre o usuário atual para esse componente na forma de propriedades. O código `{...data.me}` usa o operador de espalhamento (spread operator) para passar todos os campos ao componente `CurrentUser` como propriedades individuais. Além disso, a função `logout` é passada para o componente `CurrentUser`. Quando o usuário clicar no botão de `logout`, essa função será chamada e seu token será removido.

Trabalhando com cache

Como desenvolvedores, estamos no negócio de minimização de requisições de rede. Não queremos que nossos usuários tenham

que fazer requisições irrelevantes. Para minimizar a quantidade de requisições de rede que nossas aplicações enviam, podemos explorar melhor a personalização do Apollo Cache.

Políticas de busca

Por padrão, o Apollo Client armazena dados em uma variável JavaScript local. Sempre que criamos um cliente, um cache é criado para nós. Sempre que enviarmos uma operação, a resposta será armazenada localmente em cache. A `fetchPolicy` informa ao Apollo Client em que local ele deverá procurar os dados para resolver uma operação: no cache local ou com uma requisição de rede. A `fetchPolicy` default é `cache-first`. Isso significa que o cliente procurará os dados para a resolução das operações localmente no cache. Se o cliente puder resolver a operação sem enviar uma requisição de rede, ele o fará. No entanto, se os dados para resolver a consulta não estiverem no cache, o cliente enviará uma requisição de rede para o serviço GraphQL.

Outro tipo de `fetchPolicy` é `cache-only`. Essa política diz ao cliente para procurar somente no cache, e jamais enviar uma requisição de rede. Se os dados para atender a consulta não estiverem no cache, um erro será lançado.

Consulte `src/Users.js` e encontra a `Query` no componente `Users`. Podemos alterar a política de busca para consultas individuais simplesmente adicionando a propriedade `fetchPolicy`:

```
<Query query={{ query: ROOT_QUERY }} fetchPolicy="cache-only">
```

No momento, se definirmos a política dessa `Query` para `cache-only` e atualizarmos o navegador, veremos um erro porque o Apollo Client procurará os dados somente no cache para resolver nossa consulta, e esses dados não estarão presentes quando a aplicação iniciar. Para eliminar o erro, altere a política de busca para `cache-and-network`:

```
<Query query={{ query: ROOT_QUERY }} fetchPolicy="cache-and-network">
```

A aplicação voltará a funcionar. A política `cache-and-network` sempre resolve a consulta imediatamente com o cache e, adicionalmente,

envia uma requisição de rede para obter os dados mais recentes. Se o cache local não existir, como é o caso quando a aplicação inicia, essa política simplesmente fará com que os dados sejam obtidos da rede. Outras políticas incluem:

`network-only`

Sempre envia uma requisição de rede para a resolução de uma consulta.

`no-cache`

Sempre enviar uma requisição de rede para resolver os dados e não coloca a resposta resultante no cache.

Persistência do cache

É possível salvar o cache localmente no cliente. Isso traz à tona a eficácia da política `cache-first`, pois o cache já existirá quando o usuário retornar para a aplicação. Nesse caso, a política `cache-first` resolverá imediatamente os dados usando o cache local, e uma requisição para a rede não será enviada.

Para salvar dados em cache localmente, devemos instalar um pacote npm:

```
npm install apollo-cache-persist
```

O pacote `apollo-cache-persist` contém uma função que melhora o uso do cache salvando-o em uma área de armazenagem local sempre que for alterado. Para implementar a persistência de cache, será necessário criar nosso próprio objeto `cache` e adicioná-lo no `client` quando nossa aplicação for configurada.

Adicione o código a seguir no arquivo `src/index.js`:

```
import ApolloClient, { InMemoryCache } from 'apollo-boost'
import { persistCache } from 'apollo-cache-persist'

const cache = new InMemoryCache()
persistCache({
  cache,
  storage: localStorage
})
```

```
const client = new ApolloClient({
  cache,

  ...

})
```

Inicialmente criamos nossa própria instância de cache usando o construtor `InMemoryCache` disponibilizado pelo `apollo-boost`. Em seguida, importamos o método `persistCache` do `apollo-cache-persist`. Usando `InMemoryCache` criamos uma nova instância `cache` e a enviamos para o método `persistCache`, junto com um local de armazenagem (`storage`). Optamos por salvar o cache no `localStorage` da janela do navegador. Isso significa que depois que nossa aplicação iniciar, veremos o valor de nosso cache salvo em nossa área de armazenagem. Você pode verificá-lo por meio da sintaxe a seguir:

```
console.log(localStorage['apollo-cache-persist'])
```

O próximo passo é verificar `localStorage` na inicialização para ver se já temos um cache salvo. Em caso afirmativo, inicializaremos nosso cache local com esses dados antes de criar o cliente:

```
const cache = new InMemoryCache()
persistCache({
  cache,
  storage: localStorage
})

if (localStorage['apollo-cache-persist']) {
  let cacheData = JSON.parse(localStorage['apollo-cache-persist'])
  cache.restore(cacheData)
}
```

Agora nossa aplicação carregará qualquer dado salvo em cache antes de iniciar. Se tivermos dados salvos com a chave `apollo-cache-persist`, usaremos o método `cache.restore(cacheData)` para adicioná-los na instância `cache`.

Minimizamos com sucesso o número de requisições de rede para o nosso serviço simplesmente usando o cache do Apollo Client de

modo eficiente. Na próxima seção, aprenderemos a escrever os dados diretamente no cache local.

Atualizando o cache

O componente `Query` é capaz de ler dados diretamente do cache. É isso que torna possível uma política de busca como `cache-only`. Também podemos interagir diretamente com o Apollo Cache. É possível ler os dados atuais do cache ou escrever dados diretamente no cache. Sempre que alterarmos os dados armazenados no cache, o `react-apollo` detectará essa mudança e renderizará novamente todos os componentes afetados. Tudo que temos que fazer é modificar o cache, e a UI se atualizará automaticamente para que esteja de acordo com a mudança.

Os dados são lidos do Apollo Cache com o GraphQL. Você lê as consultas. Os dados são escritos no Apollo Cache usando o GraphQL, e você escreve dados nas consultas. Considere a `ROOT_QUERY`, que está em `src/App.js`:

```
export const ROOT_QUERY = gql`
  query allUsers {
    totalUsers
    allUsers { ...userInfo }
    me { ...userInfo }
  }

  fragment userInfo on User {
    githubLogin
    name
    avatar
  }
`
```

Essa consulta tem três campos em seu conjunto de seleção: `totalUsers`, `allUsers` e `me`. Podemos ler qualquer dado que esteja armazenado no momento em nosso cache usando o método `cache.readQuery`:

```
let { totalUsers, allUsers, me } = cache.readQuery({ query: ROOT_QUERY })
```

Nessa linha de código, obtivemos valores para `totalUsers`, `allUsers` e `me`, armazenados no cache.

Também podemos escrever dados diretamente nos campos `totalUsers`, `allUsers` e `me` da `ROOT_QUERY` usando o método `cache.writeQuery`:

```
cache.writeQuery({
  query: ROOT_QUERY,
  data: {
    me: null,
    allUsers: [],
    totalUsers: 0
  }
})
```

Nesse exemplo estamos limpando todos os dados de nosso cache e restaurando valores default para todos os campos da `ROOT_QUERY`. Como estamos usando o `react-apollo`, essa alteração dispararia uma atualização de UI e limparia a lista completa de usuários do DOM atual.

Um bom local para escrever dados diretamente no cache é na função `logout`, no componente `AuthorizedUser`. No momento, essa função remove o token do usuário, mas a UI não será atualizada até que o botão “Refetch” (Buscar novamente) seja clicado ou o navegador seja atualizado. Para melhorar essa funcionalidade, limparemos o usuário atual do cache diretamente, quando o usuário fizer `logout`.

Em primeiro lugar, devemos garantir que esse componente tenha acesso ao `client` em suas propriedades. Um dos modos mais rápidos de passar essa propriedade é usar o componente de ordem superior `withApollo`. Isso fará o `client` ser adicionado nas propriedades do componente `AuthorizedUser`. Como esse componente já usa o componente de ordem superior `withRouter`, utilizaremos a função `compose` para garantir que o componente `AuthorizedUser` seja encapsulado com os dois componentes de ordem superior:

```
import { Query, Mutation, withApollo, compose } from 'react-apollo'
```

```
class AuthorizedUser extends Component {
  ...
```

```
}
```

```
export default compose(withApollo, withRouter)(AuthorizedUser)
```

Usando o `compose`, reunimos as funções `withApollo` e `withRouter` em uma só função. `withRouter` adiciona o `history` do `Router` às propriedades, enquanto `withApollo` adiciona o `Apollo Client` às propriedades.

Isso significa que podemos acessar o `Apollo Client` em nosso método `logout` e usá-lo para remover os detalhes sobre o usuário atual do cache:

```
logout = () => {  
  localStorage.removeItem('token')  
  let data = this.props.client.readQuery({ query: ROOT_QUERY })  
  data.me = null  
  this.props.client.writeQuery({ query: ROOT_QUERY, data })  
}
```

O código anterior não só remove o token do usuário atual de `localStorage` como também limpa o campo `me` desse usuário, salvo no cache. Agora, quando os usuários fizerem `logout`, eles verão o botão “Sign In with GitHub” (Login com o GitHub) imediatamente, sem que seja necessário atualizar o navegador. Esse botão será renderizado somente quando a `ROOT_QUERY` não tiver nenhum valor para `me`.

Outro local no qual podemos melhorar nossa aplicação por trabalhar diretamente com o cache é no arquivo `src/Users.js`. Atualmente, quando clicamos no botão “Add Fake User” (Adicionar usuário simulado), uma mutação é enviada para o serviço GraphQL. O componente `Mutation` que renderiza esse botão contém a seguinte propriedade:

```
refetchQueries=[[{ query: ROOT_QUERY }]]
```

Essa propriedade diz ao cliente para enviar uma consulta adicional ao nosso serviço depois que a mutação tiver sido concluída. Entretanto, já estamos recebendo uma lista de novos usuários simulados na resposta da própria mutação:

```
mutation addFakeUsers($count:Int!) {  
  addFakeUsers(count:$count) {  
    githubLogin
```

```

      name
      avatar
    }
  }
}

```

Como já temos uma lista dos novos usuários simulados, não há necessidade de voltar ao servidor para buscar as mesmas informações. O que temos que fazer é obter essa nova lista de usuários na resposta da mutação e adicioná-la diretamente no cache. Uma vez que o cache mudar, a UI o acompanhará.

Encontre o componente `Mutation` no arquivo `Users.js`, o qual trata a mutação `addFakeUsers`, e substitua `refetchQueries` por uma propriedade `update`:

```

<Mutation mutation={ADD_FAKE_USERS_MUTATION}
  variables={{ count: 1 }}
  update={updateUserCache}>
  {addFakeUsers =>
    <button onClick={addFakeUsers}>Add Fake User</button>
  }
</Mutation>

```

Agora, quando a mutação for concluída, os dados de resposta serão enviados para uma função chamada `updateUserCache`:

```

const updateUserCache = (cache, { data:{ addFakeUsers } }) => {
  let data = cache.readQuery({ query: ROOT_QUERY })
  data.totalUsers += addFakeUsers.length
  data.allUsers = [
    ...data.allUsers,
    ...addFakeUsers
  ]
  cache.writeQuery({ query: ROOT_QUERY, data })
}

```

Quando o componente `Mutation` chamar a função `updateUserCache`, ela enviará o cache e os dados devolvidos na resposta da mutação.

Queremos adicionar os usuários simulados no cache atual, portanto, leremos os dados que já estão no cache usando `cache.readQuery({ query: ROOT_QUERY })` e lhe faremos acréscimos. Em primeiro lugar, incrementaremos o total de usuários, `data.totalUsers += addFakeUsers.length`. Em seguida, concatenaremos os usuários simulados que

recebemos da mutação na lista de usuários atuais. Agora que os dados atuais foram alterados, eles poderão ser escritos de volta no cache com `cache.writeQuery({ query: ROOT_QUERY, data })`. Substituir os dados no cache fará com que a UI se atualize e exiba o novo usuário simulado.

A essa altura, completamos a primeira versão da porção de nossa aplicação referente ao usuário. Podemos listar todos os usuários, adicionar usuários simulados e fazer login com o GitHub. Implementamos uma aplicação GraphQL full stack usando o Apollo Server e o Apollo Client. Os componentes `Query` e `Mutation` são ferramentas que podem ser usadas rapidamente para começar a desenvolver clientes com o Apollo Client e o React.

No Capítulo 7 veremos como incorporar as subscriptions e o upload de arquivos na aplicação PhotoShare. Discutiremos também o surgimento de ferramentas no ecossistema do GraphQL, as quais poderão ser incorporadas em seus projetos.

¹ N.T.: Uma interface de usuário otimista (optimistic user interface) é aquela em que a atualização da UI é imediata como resposta a uma ação do usuário, mesmo que haja uma requisição pendente.

CAPÍTULO 7

GraphQL no mundo real

Até agora, fizemos o design de um esquema, construímos uma API GraphQL e implementamos um cliente usando o Apollo Client. Realizamos uma iteração full-stack completa com o GraphQL e desenvolvemos uma compreensão sobre como as APIs GraphQL são consumidas pelos clientes. É hora de preparar nossas APIs GraphQL e os clientes para um ambiente de produção.

Para levar suas novas habilidades a esse ambiente, você deverá atender aos requisitos de suas aplicações atuais. Nossas aplicações atuais provavelmente permitem transferências de arquivos entre o cliente e o servidor. Essas aplicações talvez usem WebSockets para enviar atualizações de dados em tempo real aos nossos clientes. Nossas APIs atuais são seguras e oferecem proteção contra clientes maliciosos. Para trabalhar com o GraphQL em ambiente de produção, temos que ser capazes de atender a esses requisitos.

Além disso, devemos pensar em nossas equipes de desenvolvimento. Pode ser que você esteja trabalhando com uma equipe full-stack, mas, com muita frequência, as equipes estarão separadas em desenvolvedores de frontend e de backend. De que modo os desenvolvedores de diferentes especialidades poderão trabalhar com nossa pilha GraphQL de forma eficiente?

E o que dizer do simples escopo de sua base de código atual? No momento, é provável que você tenha muitos serviços e APIs diferentes executando no ambiente de produção e que não tenha tempo nem recursos para reconstruir tudo do zero com o GraphQL.

Neste capítulo abordaremos todos esses requisitos e preocupações.

Começaremos implementando mais duas iterações na API do PhotoShare. Em primeiro lugar, incorporaremos as subscriptions (inscrições ou assinaturas) e o transporte de dados em tempo real. Em segundo, permitiremos que os usuários postem fotos implementando uma solução para transporte de arquivos com o GraphQL. Uma vez concluídas essas iterações na aplicação PhotoShare, veremos formas de proteger nossa API GraphQL contra consultas de clientes maliciosos. Concluiremos este capítulo analisando maneiras de as equipes poderem trabalhar juntas a fim de fazer uma migração eficaz para o GraphQL.

Subscriptions

Atualizações em tempo real são um recurso essencial para aplicações web e aplicativos móveis modernos. A tecnologia atual que permite transporte de dados em tempo real entre sites e aplicativos móveis chama-se Websockets. O protocolo WebSocket pode ser usado para abrir canais de comunicação duplex bidirecionais sobre um socket TCP. Isso significa que as páginas web e as aplicações podem enviar e receber dados por meio de uma única conexão. Essa tecnologia permite que atualizações sejam enviadas do servidor diretamente para a página web em tempo real.

Até agora implementamos consultas e mutações GraphQL usando o protocolo HTTP. O HTTP provê uma maneira de enviar e receber dados entre o cliente e o servidor, mas não nos ajuda a fazer uma conexão com um servidor e ouvir mudanças de estados. Antes de o WebSockets ter sido introduzido, a única maneira de ouvir mudanças de estado no servidor era enviar requisições HTTP periodicamente para ele a fim de determinar se algo havia mudado. Vimos como implementar facilmente um polling com a tag de consulta no Capítulo 6.

Porém, se quisermos realmente tirar total proveito da nova web, o GraphQL deve aceitar transporte de dados em tempo real com

WebSockets, além de tratar requisições HTTP. A solução está nas *subscriptions* (inscrições ou assinaturas). Vamos ver como podemos implementar as subscriptions no GraphQL.

Trabalhando com subscriptions

No GraphQL, usamos as subscriptions para ouvir a API e saber se houve mudanças de dados específicas. O Apollo Server já oferece suporte para as subscriptions. Ele inclui dois pacotes npm que são comumente usados para configurar o WebSockets em aplicações GraphQL: `graphql-subscriptions` e `subscriptions-transport-ws`. O pacote `graphql-subscriptions` é um pacote npm que oferece uma implementação do padrão de projeto publisher/subscriber, o PubSub. O PubSub é essencial para publicar mudanças de dados que podem ser consumidas por clientes que se inscreverem para recebê-las. O `subscriptions-transport-ws` é um servidor e cliente WebSocket que permite o transporte de subscriptions pelo WebSockets. O Apollo Server inclui automaticamente esses dois pacotes, oferecendo suporte pronto para as subscriptions.

Por padrão, o Apollo Server configura um WebSocket em `ws://localhost:4000`. Se você usar a configuração simples do servidor, que apresentamos no início do Capítulo 5, estará usando uma configuração que aceita prontamente o WebSockets.

Como estamos trabalhando com o `apollo-server-express`, será necessário executar alguns passos para que as subscriptions funcionem. Localize o arquivo `index.js` na `photo-share-api` e importe a função `createServer` do módulo `http`:

```
const { createServer } = require('http')
```

O Apollo Server configurará automaticamente o suporte às subscriptions, mas, para isso, ele precisará de um servidor HTTP. Usaremos `createServer` para criar um. Localize o código no final da função `start`, no ponto em que o serviço GraphQL é iniciado em uma porta específica com `app.listen(...)`. Substitua esse código por:

```
const httpServer = createServer(app)
```

```
server.installSubscriptionHandlers(httpServer)

httpServer.listen({ port: 4000 }, () =>
  console.log(`GraphQL Server running at localhost:4000${server.graphqlPath}`)
)
```

Inicialmente criamos um novo `httpServer` usando a instância da aplicação Express. O `httpServer` está pronto para tratar todas as requisições HTTP que lhe forem enviadas com base em nossa configuração atual do Express. Temos também uma instância de servidor na qual podemos acrescentar suporte ao Websocket. Na próxima linha de código, `server.installSubscriptionHandlers(httpServer)` é o que faz o WebSockets funcionar. É aí que o Apollo Server adiciona os handlers necessários para aceitar subscriptions com WebSockets. Além de ter um servidor HTTP, nosso backend agora está pronto para receber requisições em `ws://localhost:4000/graphql`.

Agora que temos um servidor preparado para aceitar subscriptions, é hora de implementá-las.

Postando fotos

Queremos ser informados se algum de nossos usuários postar uma foto. Esse é um bom caso de uso para uma subscription. Como tudo mais no GraphQL, para implementar as subscriptions devemos começar pelo esquema. Vamos adicionar um tipo subscription no esquema, logo após a definição do tipo `Mutation`:

```
type Subscription {
  newPhoto: Photo!
}
```

A subscription `newPhoto` será usada para enviar dados ao cliente quando houver fotos adicionadas. Enviaremos uma subscription com a seguinte operação da linguagem de consulta GraphQL:

```
subscription {
  newPhoto {
    url
    category
    postedBy {
      githubLogin
    }
  }
}
```

```

      avatar
    }
  }
}

```

Essa subscription enviará dados sobre novas fotos ao cliente. Assim como no caso de uma Query ou de uma Mutation, o GraphQL nos permite pedir dados sobre campos específicos usando conjuntos de seleção. Com essa subscription, sempre que houver uma nova foto, receberemos seu url e a category, além do githubLogin e do avatar do usuário que a postou.

Quando uma subscription é enviada ao nosso serviço, a conexão permanecerá aberta. Ela estará ouvindo mudanças de dados. Toda foto adicionada fará com que dados sejam enviados ao assinante (o subscriber). Se você configurar uma subscription com o GraphQL Playground, perceberá que o botão Play (Executar) mudará para um botão vermelho Stop (Parar).

O botão Stop informa que a subscription está ativa no momento, ouvindo dados. Ao pressionar esse botão, a subscription será cancelada. Ela deixará de ouvir mudanças de dados.

Finalmente é hora de ver a mutação `postPhoto`: a mutação que adiciona novas fotos no banco de dados. Queremos publicar detalhes das novas fotos em nossa subscription a partir desta mutação:

```

async postPhoto(root, args, { db, currentUser, pubsub }) {

  if (!currentUser) {
    throw new Error('only an authorized user can post a photo')
  }

  const newPhoto = {
    ...args.input,
    userID: currentUser.githubLogin,
    created: new Date()
  }

  const { insertedIds } = await db.collection('photos').insert(newPhoto)
  newPhoto.id = insertedIds[0]
}

```

```
pubsub.publish('photo-added', { newPhoto })

return newPhoto
}
```

Esse resolver espera que uma instância de `pubsub` tenha sido adicionada no contexto. Faremos isso no próximo passo. O `pubsub` é um mecanismo capaz de publicar eventos e enviar dados para o resolver de nossa subscription. É como o `EventEmitter` do `Node.js`. Você pode usá-lo para publicar eventos e enviar dados a qualquer handler que tenha se inscrito para receber um evento. Nesse caso, publicamos um evento `photo-added` logo depois de inserir uma nova foto no banco de dados. Os detalhes sobre a nova foto são passados no segundo argumento do método `pubsub.publish`. Esses detalhes serão passados para todo handler que tenha se inscrito para receber eventos `photo-added`.

A seguir, vamos adicionar o resolver `Subscription` que será usado para se inscrever aos eventos `photo-added`:

```
const resolvers = {

  ...

  Subscription: {
    newPhoto: {
      subscribe: (parent, args, { pubsub }) =>
        pubsub.asyncIterator('photo-added')
    }
  }
}
```

O resolver `Subscription` é um resolver raiz. Deve ser adicionado diretamente no objeto `resolvers`, ao lado dos resolvers `Query` e `Mutation`. No resolver `Subscription`, devemos definir resolvers para cada campo. Como definimos o campo `newPhoto` em nosso esquema, devemos garantir que haja um resolver `newPhoto` em nossos resolvers.

De modo diferente dos resolvers `Query` ou `Mutation`, os resolvers `Subscription` contêm um método de inscrição. O método de inscrição

recebe `parent`, `args` e `context`, assim como qualquer outra função `resolver`. Nesse método, nós nos inscrevemos para receber eventos específicos. Nesse caso, estamos usando `pubsub.asyncIterator` para nos inscrever aos eventos `photo-added`. Sempre que um evento `photo-added` for gerado por `pubsub`, ele será passado para essa `subscription` de nova foto.



Resolvers de subscription no repositório

Os arquivos de exemplo no repositório do GitHub separam os resolvers em vários arquivos. O código anterior pode ser encontrado em `resolvers/Subscriptions.js`.

Tanto o resolver `postPhoto` quanto o resolver da `subscription newPhoto` esperam que haja uma instância de `pubsub` no contexto. Vamos modificar o contexto para que inclua o `pubsub`. Localize o arquivo `index.js` e faça as seguintes alterações:

```
const { ApolloServer, PubSub } = require('apollo-server-express')
```

```
...
```

```
async function start() {
```

```
...
```

```
const pubsub = new PubSub()
```

```
const server = new ApolloServer({
```

```
  typeDefs,
```

```
  resolvers,
```

```
  context: async ({ req, connection }) => {
```

```
    const githubToken = req ?
```

```
      req.headers.authorization :
```

```
      connection.context.Authorization
```

```
    const currentUser = await db
```

```
      .collection('users')
```

```
      .findOne({ githubToken })
```

```
    return { db, currentUser, pubsub }
```

```
  }
```

```
})
```

```
...  
}
```

Em primeiro lugar, temos que importar o construtor `PubSub` do pacote `apollo-server-express`. Usamos esse construtor para criar uma instância `pubsub` e adicioná-la no contexto.

Talvez você tenha percebido também que mudamos a função de contexto. Consultas e mutações continuarão usando HTTP. Quando enviamos uma dessas operações ao serviço GraphQL, o argumento de requisição, `req`, é enviado ao handler de contexto. No entanto, se a operação for uma `Subscription`, não haverá requisição HTTP, portanto o argumento `req` será `null`. Informações das subscriptions são passadas quando o cliente se conecta ao `WebSocket`. Nesse caso, o argumento `connection` do `WebSocket` será enviado para a função de contexto. Se tivermos uma subscription, teremos que passar detalhes da autorização por meio do `context` da conexão, e não nos cabeçalhos de requisições HTTP.

Agora estamos prontos para testar nossa nova subscription. Abra o Playground e inicie uma subscription:

```
subscription {  
  newPhoto {  
    name  
    url  
    postedBy {  
      name  
    }  
  }  
}
```

Depois que a subscription estiver executando, abra uma nova aba do Playground e execute a mutação `postPhoto`. Sempre que executar essa mutação, você verá que os dados de sua nova foto serão enviados para a subscription.

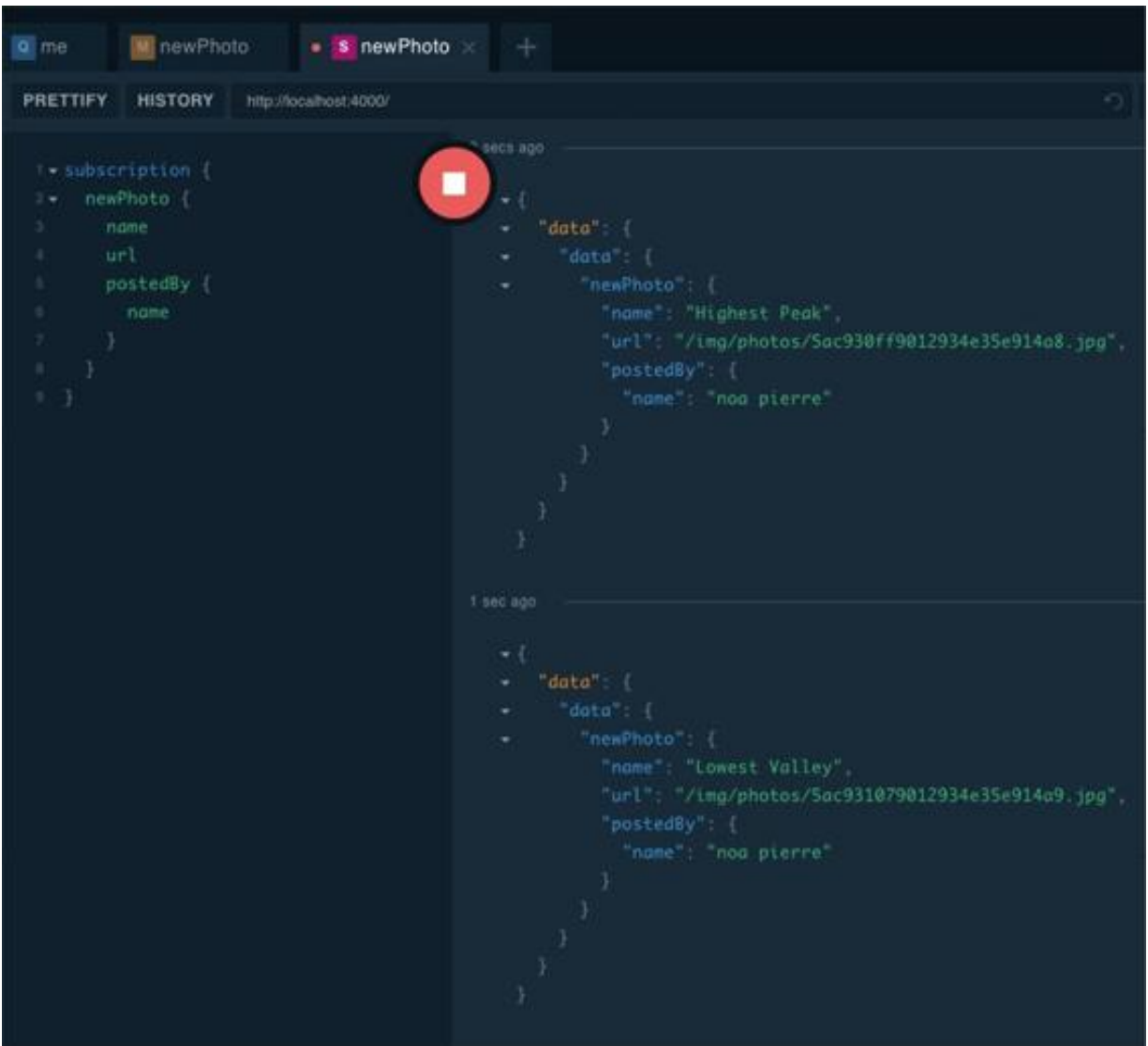


Figura 7.1 – A subscription newPhoto no Playground.

Desafio: a subscription newUser

Você é capaz de implementar uma subscription newUser? Sempre que novos usuários forem adicionados no banco de dados com o githubLogin ou a mutação addFakeUsers, você é capaz de publicar um novo evento *new-user* para uma subscription?

Dica: ao trabalhar com addFakeUsers, é possível que você tenha que publicar o evento algumas vezes, uma para cada usuário adicionado.

Se não souber o que fazer, você poderá encontrar a resposta no repositório (<https://github.com/MoonHighway/learning-graphql/tree/master/chapter-07>).

Consumindo subscriptions

Supondo que você tenha cumprido o desafio da caixa de texto anterior, o servidor Photoshare aceitará subscriptions para Photos e

Users. Na próxima seção faremos a inscrição para a subscription `newUser` e exibiremos imediatamente qualquer novo usuário na página. Antes de começar, devemos configurar o Apollo Client para que trate as subscriptions.

Adicionando o WebSocketLink

As subscriptions são usadas com WebSockets. Para ativá-lo no servidor, é necessário instalar alguns pacotes adicionais:

```
npm install apollo-link-ws apollo-utilities subscription-transport-ws
```

Em seguida devemos adicionar um link WebSocket na configuração do Apollo Client. Localize o arquivo `src/index.js` no projeto `photo-share-client` e acrescente as importações a seguir:

```
import {  
  InMemoryCache,  
  HttpLink,  
  ApolloLink,  
  ApolloClient,  
  split  
} from 'apollo-boost'  
import { WebSocketLink } from 'apollo-link-ws'  
import { getMainDefinition } from 'apollo-utilities'
```

Observe que importamos `split` de `apollo-boost`. Usaremos isso para separar as operações GraphQL entre requisições HTTP e WebSockets. Se a operação for uma mutação ou uma consulta, o Apollo Client enviará uma requisição HTTP. Se for uma subscription, o cliente se conectará com o WebSocket.

Internamente ao Apollo Client, as requisições de rede são administradas com o `ApolloLink`. Na aplicação atual ele tem sido responsável por enviar requisições HTTP ao serviço GraphQL. Sempre que uma operação é enviada com o Apollo Client, ela é enviada para um Apollo Link para que a requisição de rede seja tratada. Também podemos usar um Apollo Link para lidar com a rede no WebSockets.

Teremos que configurar dois tipos de links para suporte ao WebSockets: um `HttpLink` e um `WebSocketLink`:

```
const httpLink = new HttpLink({ uri: 'http://localhost:4000/graphql' })
const wsLink = new WebSocketLink({
  uri: `ws://localhost:4000/graphql`,
  options: { reconnect: true }
})
```

O `httpLink` pode ser usado para enviar requisições HTTP pela rede para `http://localhost:4000/graphql`, enquanto `wsLink` pode ser usado para se conectar a `ws://localhost:4000/graphql` e consumir dados do WebSockets.

Os links podem ser compostos. Isso significa que eles podem ser encadeados de modo a construir pipelines personalizados para nossas operações GraphQL. Além de ser capaz de enviar uma operação para um único `ApolloLink`, podemos enviar uma operação por uma cadeia de links reutilizáveis, em que cada link da cadeia pode manipular a operação antes que ela alcance o último link da cadeia, que trata a requisição e devolve um resultado.

Vamos criar uma cadeia de links com `httpLink` adicionando um `ApolloLink` personalizado, responsável pela adição do cabeçalho de autorização na operação:

```
const authLink = new ApolloLink((operation, forward) => {
  operation.setContext(context => ({
    headers: {
      ...context.headers,
      authorization: localStorage.getItem('token')
    }
  }))
  return forward(operation)
})
```

```
const httpAuthLink = authLink.concat(httpLink)
```

O `httpLink` é concatenado ao `authLink` para tratar a autorização de usuário em requisições HTTP. Tenha em mente que essa função `.concat` não é a mesma que você verá em JavaScript para concatenar arrays. É uma função especial, que concatena Apollo Links. Depois da concatenação demos um nome mais apropriado ao link, `httpAuthLink`, para descrever mais claramente o seu comportamento. Quando uma operação é enviada para esse link, inicialmente ela

será passada para `authLink`, que adicionará o cabeçalho de autorização na operação antes de ser encaminhada para `httpLink`, que cuidará da requisição de rede. Se você tiver familiaridade com `middleware` no `Express` ou no `Redux`, verá que o padrão é semelhante.

Agora temos que dizer ao cliente qual link deve ser usado. É nesse cenário que `split` se mostra conveniente. A função `split` devolve um de dois `Apollo Links` com base em um predicado. O primeiro argumento dessa função é o predicado. Um predicado é uma função que devolve `true` ou `false`. O segundo argumento dessa função representa o link a ser devolvido quando o predicado devolver `true`, enquanto o terceiro argumento representa o link a ser devolvido se for `false`.

Vamos implementar um link `split` que verificará se nossa operação por acaso é uma `subscription`. Em caso afirmativo, usaremos `wsLink` para tratar a rede; caso contrário, `httpLink` será utilizado:

```
const link = split(
  ({ query }) => {
    const { kind, operation } = getMainDefinition(query)
    return kind === 'OperationDefinition' && operation === 'subscription'
  },
  wsLink,
  httpAuthLink
)
```

O primeiro argumento é a função de predicado. Ela verificará a AST da operação de `query` usando a função `getMainDefinition`. Se essa operação for uma `subscription`, nosso predicado devolverá `true`. Quando o predicado devolver `true`, `link` devolverá `wsLink`. Quando o predicado devolver `false`, `link` devolverá `httpAuthLink`.

Por fim, temos que alterar a configuração de nosso `Apollo Client` para que use nossos links personalizados, passando-lhe o `link` e o `cache`:

```
const client = new ApolloClient({ cache, link })
```

Agora nosso cliente está pronto para lidar com as `subscriptions`. Na próxima seção enviaremos nossa primeira operação de `subscription`

usando o Apollo Client.

Ouvindo se há novos usuários

No cliente, podemos ouvir se há novos usuários criando uma constante chamada `LISTEN_FOR_USERS`. Ela contém uma string com nossa subscription, a qual devolverá o `githubLogin`, o `name` e o `avatar` de um novo usuário:

```
const LISTEN_FOR_USERS = gql`
  subscription {
    newUser {
      githubLogin
      name
      avatar
    }
  }
`
```

Então podemos usar o componente `<Subscription />` para ouvir se há novos usuários:

```
<Subscription subscription={LISTEN_FOR_USERS}>
  {({ data, loading }) => loading ?
    <p>loading a new user...</p> :
    <div>
      <img src={data.newUser.avatar} alt="" />
      <h2>{data.newUser.name}</h2>
    </div>
  }
</Subscription>
```

Como podemos ver nesse caso, o componente `<Subscription />` funciona como os componentes `<Mutation />` ou `<Query />`. Nós lhe enviamos a subscription, e quando um novo usuário é recebido seus dados são passados para uma função. O problema em usar esse componente em nossa aplicação está no fato de a subscription `newUser` passar um usuário de cada vez. Assim, o componente anterior mostrará somente o último novo usuário criado.

O que devemos fazer é ouvir se há novos usuários quando o cliente PhotoShare iniciar, e quando tivermos um novo usuário, ele deverá ser adicionado ao nosso cache local atual. Quando o cache for atualizado, a UI o acompanhará, portanto não haverá necessidade

de alterar nada na UI para tratar novos usuários.

Vamos modificar o componente `App`. Em primeiro lugar, vamos convertê-lo em um componente `class` para que possamos tirar proveito do ciclo de vida dos componentes do React. Quando o componente é montado, começamos a ouvir se há novos usuários por meio de nossa subscription. Quando o componente `App` é desmontado, paramos de ouvir, chamando o método para cancelamento de inscrição da subscription:

```
import { withApollo } from 'react-apollo'

...

class App extends Component {

  componentDidMount() {
    let { client } = this.props
    this.listenForUsers = client
    .subscribe({ query: LISTEN_FOR_USERS })
    .subscribe(({ data: { newUser } }) => {
      const data = client.readQuery({ query: ROOT_QUERY })
      data.totalUsers += 1
      data.allUsers = [
        ...data.allUsers,
        newUser
      ]
      client.writeQuery({ query: ROOT_QUERY, data })
    })
  }

  componentWillUnmount() {
    this.listenForUsers.unsubscribe()
  }

  render() {
    ...
  }
}

export default withApollo(App)
```

Quando exportamos o componente `<App />`, usamos a função `withApollo`

para passar o cliente para App usando props. Quando o componente é montado, usaremos o cliente para começar a ouvir se há novos usuários. Quando o componente é desmontado, cancelamos a subscription usando o método `unsubscribe`.

A subscription é criada com `client.subscribe().subscribe()`. A primeira função `subscribe` é um método do Apollo Client, usado para enviar a operação de subscription ao nosso serviço. Ela devolve um objeto observador (observer). A segunda função `subscribe` é um método desse objeto. É usado para inscrição de handlers junto ao observador. Os handlers são chamados sempre que a subscription enviar dados ao cliente. No código anterior adicionamos um handler que captura as informações sobre cada novo usuário e as adiciona diretamente no Apollo Cache usando `writeQuery`.

Agora, quando novos usuários forem adicionados, eles serão enviados prontamente para o nosso cache local, o que fará a UI ser atualizada de imediato. Como a subscription adiciona cada novo usuário na lista em tempo real, não haverá mais necessidade de atualizar o cache local em `src/Users.js`. Nesse arquivo, você deve remover a função `updateUserCache`, assim como a propriedade `update` da mutação. Uma versão completa do componente da aplicação pode ser vista no site do livro (<https://github.com/MoonHighway/learning-graphql/tree/master/chapter-07/photo-share-client>).

Upload de arquivos

Um último passo deve ser executado para criar nossa aplicação PhotoShare: fazer o upload propriamente dito de uma foto. Para implementar isso com o GraphQL, devemos modificar tanto a API quanto o cliente para que possam tratar `multipart/form-data`, o tipo de codificação necessário para passar um arquivo no corpo de um POST pela internet. Executaremos um passo adicional que nos permitirá passar um arquivo como um argumento GraphQL, de modo que o próprio arquivo possa ser tratado diretamente no resolver.

Para nos ajudar nessa implementação, usaremos dois pacotes npm: `apollo-upload-client` e `apollo-upload-server`. Os dois pacotes foram projetados para passar arquivos de um navegador web por meio de HTTP. O `apollo-upload-client` será responsável por capturar o arquivo no navegador e passá-lo para o servidor junto com a operação. O `apollo-upload-server` foi projetado para lidar com os arquivos passados para o servidor pelo `apollo-upload-client`. O `apollo-upload-server` captura o arquivo e o mapeia para o argumento de consulta apropriado antes de enviá-lo para o resolver.

Tratando uploads no servidor

O Apollo Server inclui automaticamente o `apollo-upload-server`. Não há necessidade de instalar esse pacote npm no projeto de sua API, pois ele já estará presente e funcionando. A API GraphQL deve estar pronta para aceitar um arquivo cujo upload tenha sido feito. Um tipo escalar personalizado `Upload` está disponível no Apollo Server. Ele pode ser usado para capturar o `stream`, o `mimetype` e o `encoding` do arquivo cujo upload foi feito.

Começaremos pelo esquema, adicionando um escalar personalizado em nossas definições de tipos. No arquivo do esquema, acrescente o escalar `Upload`:

```
scalar Upload
```

```
input PostPhotoInput {  
  name: String!  
  category: Photo_Category = PORTRAIT  
  description: String,  
  file: Upload!  
}
```

O tipo `Upload` nos permitirá passar o conteúdo de um arquivo com nosso `PostPhotoInput`. Isso significa que receberemos o próprio arquivo no resolver. O tipo `Upload` contém informações sobre o arquivo, incluindo um `stream` de upload que pode ser usado para salvar o arquivo. Vamos usar esse `stream` na mutação `postPhoto`. Acrescente o código a seguir no final da mutação `postPhoto`, que se encontra em

resolvers/Mutation.js:

```
const { uploadStream } = require('../lib')
const path = require('path')

...

async postPhoto(root, args, { db, user, pubsub }) => {

  ...

  var toPath = path.join(
    __dirname, '..', 'assets', 'photos', `${photo.id}.jpg`
  )

  await { stream } = args.input.file
  await uploadFile(input.file, toPath)

  pubsub.publish('photo-added', { newPhoto: photo })

  return photo
}
```

Nesse exemplo, a função `uploadStream` devolverá uma *promise* (promessa), que será resolvida quando o upload terminar. O argumento `file` contém o stream de upload, cujo pipe pode ser feito para um `writeStream`; os dados serão salvos localmente no diretório `assets/photos`. Cada foto recém-postada receberá um nome com base em seu identificador único. Estamos trabalhando somente com imagens JPEG neste exemplo, por questões de concisão.

Se quisermos servir esses arquivos de fotos a partir da mesma API, teremos que adicionar algum *middleware* em nossa aplicação Express, que nos permitirá servir imagens JPEG estáticas. No arquivo `index.js`, no qual configuramos o nosso Apollo Server, podemos adicionar o *middleware* `express.static`, que permitirá servir arquivos estáticos locais em uma rota:

```
const path = require('path')

...
```



```
app.use(
  '/img/photos',
  express.static(path.join(__dirname, 'assets', 'photos'))
)
```

Essa porção de código cuidará de servir arquivos estáticos de `assets/photos` para `/img/photos` em requisições HTTP.

Com isso, nosso servidor está pronto e pode agora tratar uploads de fotos. É hora de passar para o lado do cliente; criaremos aí um formulário capaz de administrar uploads de fotos.



Use um serviço de arquivos

Em uma verdadeira aplicação Node.js, em geral, salvaríamos os uploads dos usuários em um serviço de armazenagem de arquivos na nuvem. O exemplo anterior utiliza uma função `uploadFile` para fazer upload do arquivo em um diretório local, o que limita a escalabilidade dessa aplicação de exemplo. Serviços como AWS, Google Cloud ou Cloudinary são capazes de lidar com grandes volumes de uploads de arquivos em aplicações distribuídas.

Postando uma nova foto com o Apollo Client

Vamos agora tratar as fotos no cliente. Inicialmente, para exibir as fotos, teremos que adicionar o campo `allPhotos` em nossa `ROOT_QUERY`.

Modifique a consulta a seguir no arquivo `src/App.js`:

```
export const ROOT_QUERY = gql`
  query allUsers {
    totalUsers
    totalPhotos
    allUsers { ...userInfo }
    me { ...userInfo }
    allPhotos {
      id
      name
      url
    }
  }
`

fragment userInfo on User {
  githubLogin
  name
  avatar
}
```

```
,  
}
```

Agora, quando o site carregar, receberemos o id, o name e o url de cada foto armazenada no banco de dados. Podemos usar essas informações para exibir as fotos. Vamos criar um componente `Photos` que será usado para mostrar cada foto:

```
import React from 'react'  
import { Query } from 'react-apollo'  
import { ROOT_QUERY } from './App'  
  
const Photos = () =>  
  <Query query={ALL_PHOTOS_QUERY}>  
    {(loading, data) => loading ?  
      <p>loading...</p> :  
      data.allPhotos.map(photo =>  
        <img  
          key={photo.id}  
          src={photo.url}  
          alt={photo.name}  
          width={350} />  
      )  
    }  
  </Query>
```

```
export default Photos
```

Lembre-se de que o componente `Query` aceita `ROOT_QUERY` como uma propriedade. Então usamos o padrão de renderização de propriedade para exibir todas as fotos quando a carga terminar. Para cada foto no array `data.allPhotos` adicionaremos um novo elemento `img` com metadados extraídos de cada objeto foto, incluindo `photo.url` e `photo.name`.

Quando adicionarmos esse código no componente `App`, nossas fotos serão exibidas. Antes, porém, vamos criar outro componente. Criaremos um componente `PostPhoto` que conterá este formulário:

```
import React, { Component } from 'react'  
  
export default class PostPhoto extends Component {
```

```

state = {
  name: "",
  description: "",
  category: 'PORTRAIT',
  file: ""
}

postPhoto = (mutation) => {
  console.log('todo: post photo')
  console.log(this.state)
}

render() {
  return (
    <form onSubmit={e => e.preventDefault()}
      style={{
        display: 'flex',
        flexDirection: 'column',
        justifyContent: 'flex-start',
        alignItems: 'flex-start'
      }}>

      <h1>Post a Photo</h1>

      <input type="text"
        style={{ margin: '10px' }}
        placeholder="photo name..."
        value={this.state.name}
        onChange={({target}) =>
          this.setState({ name: target.value })} />

      <textarea type="text"
        style={{ margin: '10px' }}
        placeholder="photo description..."
        value={this.state.description}
        onChange={({target}) =>
          this.setState({ description: target.value })} />

      <select value={this.state.category}
        style={{ margin: '10px' }}
        onChange={({target}) =>
          this.setState({ category: target.value })}>

```

```

      <option value="PORTRAIT">PORTRAIT</option>
      <option value="LANDSCAPE">LANDSCAPE</option>
      <option value="ACTION">ACTION</option>
      <option value="GRAPHIC">GRAPHIC</option>
    </select>

    <input type="file"
      style={{ margin: '10px' }}
      accept="image/jpeg"
      onChange={({target}) =>
        this.setState({
          file: target.files && target.files.length ?
            target.files[0] :
            ""
        })
      } />

    <div style={{ margin: '10px' }}>
      <button onClick={() => this.postPhoto()}>
        Post Photo
      </button>
      <button onClick={() => this.props.history.goBack()}>
        Cancel
      </button>
    </div>

  </form>
)
}
}

```

O componente `PostPhoto` é somente um formulário. Esse formulário usa elementos de entrada para `name`, `description`, `category` e o próprio `file`. No React, chamamos isso de controlado porque cada elemento de entrada está ligado a uma variável de estado. Sempre que o valor de uma entrada mudar, o estado do componente `PostPhoto` mudará também.

Submetemos as fotos pressionando o botão “Post Photo” (Postar foto). A entrada para arquivos aceita um JPEG e define o estado para `file`. Esse campo de estado representa o arquivo propriamente dito, e não apenas um texto. Não adicionamos nenhuma forma de

validação nesse componente por questões de concisão.

É hora de adicionar nossos novos componentes no componente App. Quando fizermos isso garantiremos que a rota home exibirá nossos Users e Photos. Acrescentaremos também uma rota /newPhoto que poderá ser usada para exibir o formulário.

```
import React, { Fragment } from 'react'
import { Switch, Route, BrowserRouter } from 'react-router-dom'
import Users from './Users'
import Photos from './Photos'
import PostPhoto from './PostPhoto'
import AuthorizedUser from './AuthorizedUser'

const App = () =>
  <BrowserRouter>
    <Switch>
      <Route
        exact
        path="/"
        component={() =>
          <Fragment>
            <AuthorizedUser />
            <Users />
            <Photos />
          </Fragment>
        } />
      <Route path="/newPhoto" component={PostPhoto} />
      <Route component={({ location }) =>
        <h1>"{location.pathname}" not found</h1>
      } />
    </Switch>
  </BrowserRouter>

export default App
```

O componente <Switch> permite renderizar uma rota de cada vez. Se a url contiver a rota home, "/", exibiremos um componente contendo os componentes AuthorizedUser, Users e Photos. Fragment é usado no React quando queremos exibir componentes irmãos, sem que seja necessário encapsulá-los em um elemento div extra. Se a rota for "/newPhoto", exibiremos o formulário da nova foto. Quando a rota

não for reconhecida, mostraremos um elemento `h1` para informar aos usuários que a rota que eles especificaram não pôde ser encontrada.

Somente usuários autorizados podem postar fotos, portanto concatenaremos um `NavLink` “Post Photo” no componente `AuthorizedUser`. Clicar nesse botão fará `PostPhoto` ser renderizado.

```
import { withRouter, NavLink } from 'react-router-dom'

...

class AuthorizedUser extends Component {

  ...

  render() {
    return (
      <Query query={ME_QUERY}>
        {{{ loading, data }} => data.me ?
        <div>
          <img
            src={data.me.avatar_url}
            width={48}
            height={48}
            alt="" />
          <h1>{data.me.name}</h1>
          <button onClick={this.logout}>logout</button>
          <NavLink to="/newPhoto">Post Photo</NavLink>
        </div> :
      )
    )
  }

  ...
}
```

Nesse código, importamos o componente `<NavLink>`. Quando o link `Post Photo` for clicado, o usuário será enviado para a rota `/newPhoto`.

A essa altura, a navegação na aplicação deve estar funcionando. Um usuário pode navegar entre as telas, e quando postar uma foto, verá os dados de entrada necessários apresentados no console. É hora de tomarmos esses dados da postagem, incluindo o arquivo, e enviá-los com uma mutação.

Inicialmente vamos instalar o `apollo-upload-client`:

```
npm install apollo-upload-client
```

Substituiremos o link HTTP atual por um link HTTP fornecido pelo `apollo-upload-client`. Esse link aceitará requisições `multipart/form-data` contendo arquivos para upload. Para criar esse link, usaremos a função `createUploadLink`:

```
import { createUploadLink } from 'apollo-upload-client'

...

const httpLink = createUploadLink({
  uri: 'http://localhost:4000/graphql'
})
```

Substituímos o link HTTP antigo por um link novo usando a função `createUploadLink`. É bem parecido com o link HTTP. Ele tem a rota da API incluída como `uri`.

É hora de adicionar a mutação `postPhoto` no formulário `postPhoto`:

```
import React, { Component } from 'react'
import { Mutation } from 'react-apollo'
import { gql } from 'apollo-boost'
import { ROOT_QUERY } from './App'

const POST_PHOTO_MUTATION = gql`
  mutation postPhoto($input: PostPhotoInput!) {
    postPhoto(input:$input) {
      id
      name
      url
    }
  }
`
```

`POST_PHOTO_MUTATION` é a AST resultante do parse de nossa mutação, e pronta para ser enviada ao servidor. Importamos `ALL_PHOTOS_QUERY` porque teremos que usá-la quando for o momento de atualizar o cache local com a nova foto que será devolvida pela mutação.

Para adicionar a mutação, encapsularemos o elemento do botão `Post Photo` no componente `Mutation`:

```
<div style={{ margin: '10px' }}>
```

```

<Mutation mutation={POST_PHOTO_MUTATION}
  update={updatePhotos}>
  {mutation =>
    <button onClick={() => this.postPhoto(mutation)}>
      Post Photo
    </button>
  }
</Mutation>
<button onClick={() => this.props.history.goBack()}>
  Cancel
</button>
</div>

```

O componente `Mutation` passa a mutação como uma função. Quando o botão for clicado, passaremos a função de mutação para `postPhoto` para que ela possa ser usada a fim de alterar os dados da foto. Depois de concluída a mutação, a função `updatePhotos` será chamada para atualização do cache local.

Em seguida, enviamos a mutação:

```

postPhoto = async (mutation) => {
  await mutation({
    variables: {
      input: this.state
    }
  }).catch(console.error)
  this.props.history.replace('/')
}

```

Essa função de mutação devolve uma `promise`. Uma vez concluída, usaremos o `React Router` para levar o usuário de volta à página inicial, substituindo a rota atual usando a propriedade de histórico. Quando a mutação estiver concluída, precisaremos capturar os dados devolvidos por ela para atualizar o cache local:

```

const updatePhotos = (cache, { data: { postPhoto } }) => {
  var data = cache.readQuery({ query: ALL_PHOTOS_QUERY })
  data.allPhotos = [
    postPhoto,
    ...allPhotos
  ]
  cache.writeQuery({ query: ALL_PHOTOS_QUERY, data })
}

```


O método `updatePhotos` cuida da atualização de cache. Leremos as fotos do cache usando `ROOT_QUERY`. Então adicionaremos a nova foto no cache com `writeQuery`. Essa pequena dose de manutenção garantirá que nossos dados locais estejam sincronizados.

A essa altura, estamos prontos para postar novas fotos. Vá em frente e experimente fazer isso.

Vimos com mais detalhes como as consultas, as mutações e as subscriptions são tratadas do lado cliente. Ao usar o React Apollo, podemos tirar proveito dos componentes `<Query>`, `<Mutation>` e `<Subscription>` de modo a ajudar a conectar os dados de seu serviço GraphQL à interface de usuário.

Agora que a aplicação está funcionando, adicionaremos mais uma camada para cuidar da segurança.

Segurança

O serviço GraphQL oferece muita liberdade e flexibilidade a seus clientes. Eles têm flexibilidade para consultar dados de várias origens em uma só requisição. Também podem pedir grandes volumes de dados relacionados ou conectados em uma requisição. Se não forem supervisionados, seus clientes poderão requisitar muitos dados de seu serviço em uma única requisição. O esforço exigido por consultas volumosas não só afetará o desempenho do servidor como também poderá deixar o seu serviço totalmente inativo. Alguns clientes poderão fazer isso de forma ingênua ou inocente, enquanto outros talvez tenham propósitos mais maliciosos. De qualquer modo, é necessário implantar algumas medidas de segurança e monitorar o desempenho de seu servidor a fim de se proteger contra consultas extensas ou maliciosas.

Na próxima seção descreveremos algumas das opções disponíveis para melhorar a segurança de seu serviço GraphQL.

Timeout de requisições

Um *timeout de requisição* é a primeira defesa contra consultas extensas ou maliciosas. Esse timeout permite que haja somente uma determinada quantidade de tempo para processar cada requisição. Isso significa que as requisições ao seu serviço devem ser completadas em um intervalo de tempo específico. Timeouts de requisição são usados não só em serviços GraphQL, mas em quaisquer tipos de serviços e processos na internet. Talvez você já tenha implementado esses timeouts em sua API REST (Representational State Transfer, ou Transferência de Estado Representativo) para se proteger contra requisições extensas, com muitos dados de POST.

Podemos adicionar um timeout de requisição genérico no servidor Express configurando a chave `timeout`. No código a seguir acrescentamos um timeout de cinco segundos para nos protegermos contra consultas problemáticas:

```
const httpServer = createServer(app)
server.installSubscriptionHandlers(httpServer)
```

```
httpServer.timeout = 5000
```

Além disso, é possível definir timeouts para consultas em geral ou resolvers individuais. O truque para implementar timeouts para consultas ou resolvers é salvar o instante inicial de cada consulta ou resolver, e validá-lo em relação ao seu timeout preferido. Você pode registrar o instante de início de cada requisição no contexto:

```
const context = async ({ request }) => {
  ...
  return {
    ...
    timestamp: performance.now()
  }
}
```

Agora cada um dos resolvers saberá quando a consulta iniciou, e poderá lançar um erro caso haja muita demora.

Limitações nos dados

Outra medida de proteção simples que você pode implementar contra consultas extensas ou maliciosas é limitar a quantidade de dados que pode ser devolvida em cada consulta. Podemos devolver um número específico de registros, ou uma página de dados, permitindo que as consultas especifiquem quantos registros devem devolver.

Por exemplo, lembre-se de que, no Capítulo 4, fizemos o design de um esquema capaz de tratar paginação de dados. E se um cliente pedisse uma página extremamente longa de dados? Eis um exemplo de um cliente fazendo exatamente isso:

```
query allPhotos {  
  allPhotos(first=99999) {  
    name  
    url  
    postedBy {  
      name  
      avatar  
    }  
  }  
}
```

Você pode se proteger contra esses tipos de requisições extensas simplesmente definindo um limite para uma página de dados. Por exemplo, poderíamos definir um limite de 100 fotos por consulta no servidor GraphQL. Esse limite pode ser imposto pelo resolver da consulta verificando um argumento:

```
allPhotos: (root, data, context) {  
  if (data.first > 100) {  
    throw new Error('Only 100 photos can be requested at a time')  
  }  
}
```

Se for possível solicitar um número elevado de registros, será sempre uma boa ideia implementar a paginação de dados. Ela pode ser implementada simplesmente definindo o número de registros que devem ser devolvidos em uma consulta.

Limitando a profundidade da consulta

Uma das vantagens que o GraphQL oferece ao cliente é a capacidade de consultar dados conectados. Por exemplo, em nossa API de fotos podemos escrever uma consulta que forneça informações sobre uma foto, quem a postou e todas as demais fotos postadas por esse fotógrafo, tudo em uma só requisição:

```
query getPhoto($id:ID!) {  
  Photo(id:$id) {  
    name  
    url  
    postedBy {  
      name  
      avatar  
      postedPhotos {  
        name  
        url  
      }  
    }  
  }  
}
```

Essa é realmente uma funcionalidade interessante, capaz de melhorar o desempenho de rede em nossas aplicações. Podemos dizer que a consulta anterior tem profundidade igual a 3 porque ela consulta a foto em si, juntamente com dois campos conectados: `postedBy` e `postedPhotos`. A consulta raiz tem profundidade igual a 0, o campo `Photo` tem profundidade igual a 1, o campo `postedBy` tem profundidade 2 e o campo `postedPhotos` tem profundidade 3.

Os clientes podem tirar proveito dessa funcionalidade. Considere a consulta a seguir:

```
query getPhoto($id:ID!) {  
  Photo(id:$id) {  
    name  
    url  
    postedBy {  
      name  
      avatar  
      postedPhotos {
```

```

    name
    url
    taggedUsers {
      name
      avatar
      postedPhotos {
        name
        url
      }
    }
  }
}

```

Adicionamos dois outros níveis de profundidade nessa consulta: `taggedUsers` (usuários marcados) em todas as fotos postadas pelo fotógrafo da foto original, e `postedPhotos` (fotos postadas) por todos os `taggedUsers` para todas as fotos postadas pelo fotógrafo da foto original. Isso significa que, se eu postei a foto original, essa consulta também resolveria todas as fotos que postei, todos os usuários marcados nessas fotos e todas as fotos postadas por todos esses usuários marcados. São muitos dados para pedir. Também representa muito trabalho a ser feito pelos resolvers. A profundidade das consultas aumenta de forma exponencial, e pode facilmente sair do controle.

Podemos implementar um limite para a profundidade das consultas em nossos serviços GraphQL a fim de evitar que consultas profundas deixem seu serviço inativo. Se tivéssemos definido um limite de profundidade de consulta igual a 3, a primeira consulta estaria dentro do limite, enquanto a segunda não estaria, pois ela tem uma profundidade de consulta igual a 5.

As limitações na profundidade das consultas em geral são implementadas fazendo parse da AST da consulta e determinando o nível de profundidade dos conjuntos de seleção nesses objetos. Há pacotes npm como o `graphql-depth-limit` que podem oferecer assistência para essa tarefa:

```
npm install graphql-depth-limit
```

Depois de instalá-lo, você poderá acrescentar uma regra de validação na configuração de seu servidor GraphQL usando a função `depthLimit`:

```
const depthLimit = require('graphql-depth-limit')

...

const server = new ApolloServer({
  typeDefs,
  resolvers,
  validationRules: [depthLimit(5)],
  context: async({ req, connection }) => {
    ...
  }
})
```

Nesse caso, definimos o limite de profundidade da consulta para 5, o que significa que oferecemos aos nossos clientes a capacidade de escrever consultas que podem ter conjuntos de seleção com profundidade 5. Se eles se aprofundarem mais, o servidor GraphQL impedirá que a consulta execute e devolva um erro.

Limitando a complexidade das consultas

Outra medida que pode ajudar a identificar consultas problemáticas é a *complexidade da consulta*. Há algumas consultas de clientes que poderão não ter muita profundidade, mas, apesar disso, serão custosas em virtude da quantidade de campos consultados. Considere a consulta a seguir:

```
query everything($id:ID!) {
  totalUsers
  Photo(id:$id) {
    name
    url
  }
  allUsers {
    id
    name
    avatar
  }
}
```

```

    postedPhotos {
      name
      url
    }
    inPhotos {
      name
      url
      taggedUsers {
        id
      }
    }
  }
}

```

A consulta `everything` não excede nosso limite de profundidade de consulta, mas continua sendo bastante custosa por causa do número de campos consultados. Lembre-se de que cada campo está mapeado para uma função resolver que deve ser chamada.

Para calcular a complexidade de uma consulta, atribuímos um valor de complexidade para cada campo e então totalizamos a complexidade geral de cada consulta. Podemos definir um limite geral que estabeleça a complexidade máxima que uma dada consulta pode ter. Ao implementar a complexidade de consultas, é possível identificar os resolvers custosos e atribuir um valor de complexidade mais elevado a esses campos.

Há vários pacotes npm disponíveis para auxiliar na implementação de limites para a complexidade das consultas. Vamos observar como poderíamos implementar a complexidade de consultas em nosso serviço usando `graphql-validation-complexity`:

```
npm install graphql-validation-complexity
```

Esse pacote GraphQL tem um conjunto de regras default pronto para determinar a complexidade das consultas. Um valor igual a 1 é atribuído a cada campo escalar. Se esse campo estiver em uma lista, o valor será multiplicado por um fator de 10.

Por exemplo, vamos analisar como o `graphql-validation-complexity` calcularia a pontuação da consulta `everything`:

```
query everything($id:ID!) {
```

```

totalUsers # complexidade 1
Photo(id:$id) {
  name # complexidade 1
  url # complexidade 1
}
allUsers {
  id # complexidade 10
  name # complexidade 10
  avatar # complexidade 10
  postedPhotos {
    name # complexidade 100
    url # complexidade 100
  }
  inPhotos {
    name # complexidade 100
    url # complexidade 100
    taggedUsers {
      id # complexidade 1000
    }
  }
}
} # complexidade total 1433

```

Por padrão, o `graphql-validation-complexity` atribui um valor a cada campo. Esse valor é multiplicado por um fator de 10 para qualquer lista. Nesse exemplo, `totalUsers` representa um único campo inteiro e recebe uma complexidade igual a 1. Campos de consulta em uma única foto recebem o mesmo valor. Observe que os campos consultados na lista `allUsers` recebem um valor igual a 10. Isso ocorre porque estão em uma lista. Todo campo de lista é multiplicado por 10. Portanto, uma lista dentro de uma lista recebe um valor igual a 100. Como `taggedUsers` é uma lista dentro da lista `inPhotos`, que está dentro da lista `allUsers`, o valor do campo de `taggedUser` é $10 \times 10 \times 10$, isto é, 1.000.

Podemos impedir que essa consulta em particular execute se definirmos um limite geral de 1.000 para a complexidade das consultas:

```
const { createComplexityLimitRule } = require('graphql-validation-complexity')
```

...


```
const options = {  
  
  ...  
  
  validationRules: [  
    depthLimit(5),  
    createComplexityLimitRule(1000, {  
      onCost: cost => console.log('query cost: ', cost)  
    })  
  ]  
}
```

Nesse exemplo, definimos o limite máximo da complexidade para 1.000 com o uso de `createComplexityLimitRule`, que está no pacote `graphql-validation-complexity`. Também implementamos a função `onCost`, que será chamada com o custo total de cada consulta assim que esse valor for calculado. A consulta anterior não teria permissão para executar nessas circunstâncias, pois excede a complexidade máxima de 1.000.

A maioria dos pacotes de complexidade de consultas permite que você defina suas próprias regras. Poderíamos modificar os valores das complexidades atribuídas a escalares, objetos e listas com o pacote `graphql-validation-complexity`. Também é possível definir valores personalizados de complexidade para qualquer campo que consideremos muito complicado ou custoso.

Apollo Engine

Não é recomendável simplesmente implementar recursos de segurança e esperar pelo melhor. Qualquer boa estratégia de segurança e de desempenho exige métricas. É necessário ter uma forma de monitorar seu serviço GraphQL de modo que seja possível identificar suas consultas populares e ver os pontos de gargalo no que diz respeito ao desempenho.

Podemos usar o *Apollo Engine* para monitorar o serviço GraphQL, mas ele é mais que somente uma ferramenta de monitoração. O

Apollo Engine é um serviço de nuvem robusto, que provê insights sobre sua camada GraphQL para que você possa executar o serviço em ambiente de produção com confiança. Essa ferramenta monitora as operações GraphQL enviadas aos seus serviços e disponibiliza um relatório detalhado, ao vivo e online, em <https://engine.apollographql.com>; esse relatório pode ser usado para identificar suas consultas mais populares, monitorar o tempo de execução e os erros, além de ajudar na identificação de gargalos. O Apollo Engine também oferece ferramentas para gerenciamento de esquemas, incluindo a validação.

O Apollo Engine já está incluído na implementação de seu Apollo Server 2.0. Com apenas uma linha de código, podemos executar o Engine em qualquer lugar em que o Apollo Server estiver executando, inclusive em ambientes sem servidores (serveless) e na fronteira. Tudo que você precisa fazer é ativá-lo configurando a chave `engine` com `true`:

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  engine: true
})
```

O próximo passo é garantir que haja uma variável de ambiente chamada `ENGINE_API_KEY` configurada com sua chave de API para o Apollo Engine. Acesse <https://engine.apollographql.com> para criar uma conta e gerar uma chave.

Para publicar sua aplicação no Apollo Engine, será necessário instalar as ferramentas Apollo CLI:

```
npm install -g apollo
```

Depois de instalado, você poderá usar o CLI para publicar sua aplicação:

```
apollo schema:publish
--key=<SUA CHAVE DE API DO ENGINE>
--endpoint=http://localhost:4000/graphql
```

Lembre-se de acrescentar a sua `ENGINE_API_KEY` nas variáveis de

ambiente também.

A partir de agora, quando executarmos a API GraphQL do PhotoShare, todas as operações enviadas ao serviço GraphQL serão monitoradas. Poderemos ver um relatório de atividades no site do Engine. Esse relatório de atividades pode ser usado para ajudar a localizar gargalos e aliviá-los. Além do mais, com o Apollo Engine poderemos melhorar o desempenho e o tempo de resposta de nossas consultas e monitorar o desempenho de nosso serviço.

Próximos passos

Ao longo deste livro, conhecemos a teoria dos grafos, escrevemos consultas, fizemos o design de esquemas, criamos servidores GraphQL e exploramos soluções para clientes GraphQL. As bases estão definidas, portanto, podemos usar o que for necessário para melhorar nossas aplicações com o GraphQL. Nesta seção compartilharemos alguns conceitos e recursos que oferecerão mais suporte para suas futuras aplicações GraphQL.

Migração gradual

Nossa aplicação PhotoShare é um ótimo exemplo de um projeto Greenfield¹. Quando estiver trabalhando com seus próprios projetos, talvez você não possa se dar ao luxo de começar do zero. A flexibilidade do GraphQL permite que você comece a incorporá-lo gradualmente. Não há motivos para se desfazer de tudo e recomeçar do zero para tirar proveito dos recursos do GraphQL. Você pode iniciar lentamente, aplicando as ideias a seguir:

Busque dados do REST em seus resolvers

Em vez de reconstruir todos os endpoints REST, use o GraphQL como um gateway e crie uma requisição de busca desses dados no servidor, dentro de um resolver. Seu serviço também poderá fazer cache dos dados enviados pelo REST a fim de melhorar o tempo de resposta das consultas.

Ou use uma requisição GraphQL

Soluções robustas de clientes são ótimas, mas implementá-las no início pode implicar em muitas configurações. Para iniciar de modo simples, use o `graphql-request` e faça uma requisição no mesmo lugar em que você usa `fetch` para uma API REST. Essa abordagem fará com que você comece a trabalhar criando um vínculo com o GraphQL e, provavelmente, o levará a uma solução de cliente mais completa quando estiver pronto para otimizar e ter melhor desempenho. Não há motivos para não poder buscar dados de quatro endpoints REST e de um serviço GraphQL na mesma aplicação. Nem tudo precisa ser migrado para o GraphQL ao mesmo tempo.

Incorpore o GraphQL em um ou dois componentes

Em vez de reconstruir todo seu site, escolha um único componente ou página e direcione os dados para esse recurso em particular usando o GraphQL. Deixe o restante em seu site como está, enquanto monitora a experiência de migrar um único componente.

Não crie outros endpoints REST

Em vez de expandir o REST, crie um endpoint GraphQL para o seu novo serviço ou funcionalidade. Você pode hospedar um endpoint GraphQL no mesmo servidor em que estão seus endpoints REST. O Express não se importa se está roteando uma requisição para uma função REST ou um resolver GraphQL. Sempre que uma tarefa exigir um novo endpoint REST, acrescente essa funcionalidade em seu serviço GraphQL.

Não faça manutenções em seus endpoints REST atuais

Na próxima vez que houver uma tarefa para modificar um endpoint REST ou criar um endpoint personalizado para alguns dados, não faça isso! Invista tempo em separar esse único endpoint e atualize-o para que use o GraphQL. Você poderá lentamente fazer a migração de toda a sua API REST dessa maneira.

Fazer a mudança lentamente para o GraphQL permitirá que você tire proveito dos recursos de imediato, sem as dificuldades associadas a começar do zero. Comece com o que você tiver, e será possível fazer a transição para o GraphQL de modo suave e gradual.

A estratégia de desenvolvimento Schema-First

Você está em uma reunião discutindo um novo projeto web. Membros de diferentes equipes de frontend e de backend estão representados. Após a reunião, alguém pode definir algumas especificações, mas esses documentos, muitas vezes, são longos e subutilizados. As equipes de frontend e de backend começam a escrever código, mas sem diretrizes claras, o projeto é entregue com atrasos no cronograma e não atende às expectativas iniciais de cada um.

Problemas com projetos web em geral surgem como consequência da falta de comunicação ou de uma comunicação indevida sobre o que deve ser construído. Os esquemas oferecem clareza e permitem que haja comunicação; é por isso que muitos projetos usam a *estratégia de desenvolvimento Schema-First* (Esquema Primeiro). Em vez de se perder nos detalhes de implementação específicos do domínio, equipes diferentes devem trabalhar em conjunto para deixar o esquema robusto antes de implementar qualquer funcionalidade.

Os esquemas são um acordo entre as equipes de frontend e de backend, e definem todos os relacionamentos entre os dados de uma aplicação. Quando as equipes se comprometem com um esquema, elas podem trabalhar de forma independente para atender a esse esquema. Trabalhar para servir ao esquema proporciona resultados melhores, pois há clareza nas definições dos tipos. As equipes de frontend sabem exatamente quais consultas devem fazer para carregar dados nas interfaces de usuário. As

equipes de backend sabem exatamente quais dados são necessários e como oferecer suporte a eles. Um desenvolvimento Schema-First permite que haja um planejamento claro, e as equipes podem desenvolver o projeto com mais consenso e menos estresse.

A simulação (mocking) é uma parte importante da estratégia de desenvolvimento Schema-First. Depois que a equipe de frontend tiver o esquema, ela poderá usá-lo para começar a desenvolver imediatamente os componentes. O código a seguir é tudo que é necessário para criar um serviço GraphQL simulado executando em `http://localhost:4000`.

```
const { ApolloServer } = require('apollo-server')
const { readFileSync } = require('fs')

var typeDefs = readFileSync('./typeDefs.graphql', 'UTF-8')

const server = new ApolloServer({ typeDefs, mocks: true })

server.listen()
```

Supondo que você tenha disponibilizado o arquivo `typeDefs.graphql` criado durante o processo Schema-First, será possível começar o desenvolvimento dos componentes de UI que enviam operações de consulta, mutação e subscription para o serviço GraphQL simulado, enquanto a equipe de backend implementa o serviço real.

As simulações funcionam de imediato, fornecendo valores default para cada tipo escalar. Em qualquer ponto em que um campo deva ser resolvido com uma string, você verá “Hello, World” como dado.

É possível personalizar os dados devolvidos por um servidor simulado. Isso possibilita devolver dados que se pareçam mais com os dados reais. É um recurso importante, que ajudará na tarefa de estilizar os componentes de sua interface de usuário:

```
const { ApolloServer, MockList } = require('apollo-server')
const { readFileSync } = require('fs')

const typeDefs = readFileSync('./typeDefs.graphql', 'UTF-8')
const resolvers = {}
```

```

const mocks = {
  Query: () => ({
    totalPhotos: () => 42,
    allPhotos: () => new MockList([5, 10]),
    Photo: () => ({
      name: 'sample photo',
      description: null
    })
  })
}

const server = new ApolloServer({
  typeDefs,
  resolvers,
  mocks
})

server.listen({ port: 4000 }, () =>
  console.log('Mock Photo Share GraphQL Service')
)

```

O código anterior adiciona uma simulação para os campos `totalPhotos` e `allPhotos`, junto com o tipo `Photo`. Sempre que consultarmos `totalPhotos`, o número 42 será devolvido. Se consultarmos o campo `allPhotos`, receberemos entre 5 e 10 fotos. O construtor `MockList` está incluído no `apollo-server` e é usado para gerar tipos lista com tamanhos específicos. Sempre que um tipo `Photo` for resolvido pelo serviço, o nome da foto (`name`) será “sample photo” (foto de exemplo) e a descrição será `null`. Podemos criar simuladores bem robustos com pacotes como `faker` ou `casual`. Esses pacotes npm oferecem todo tipo de dados simulados, que podem ser usados na implementação de simulações realistas.

Para saber mais sobre como simular um Apollo Server, consulte a documentação do Apollo (<https://www.apollographql.com/docs/apollo-server/v2/features/mocking.html>).

Eventos associados ao GraphQL

Há uma série de conferências e meetups que têm como foco assuntos relacionados ao GraphQL.

GraphQL Summit (<https://summit.graphql.com/>)

É uma conferência organizada pelo Apollo GraphQL.

GraphQL Day (<https://www.graphqlday.org/>)

É uma conferência na Holanda para desenvolvedores que põem a mão na massa.

GraphQL Europe (<https://www.graphql-europe.org/>)

É uma conferência GraphQL sem fins lucrativos na Europa.

GraphQL Finland (<https://graphql-finland.fi/>)

É uma conferência GraphQL em Helsinque, na Finlândia, organizada pela comunidade.

Você também encontrará conteúdos sobre GraphQL em quase todas as conferências de desenvolvimento, em particular naquelas que tenham JavaScript como foco.

Se estiver à procura de eventos próximos a você, há também meetups sobre GraphQL em cidades por todo o mundo (<http://bit.ly/2lnBMB0>). Se não houver nenhum por perto, você poderia ser responsável por criar um grupo local!

Comunidade

O GraphQL é popular porque é uma tecnologia incrível. Também é popular em virtude do suporte fervoroso de sua comunidade. A comunidade é muito receptiva, e há uma série de maneiras de se envolver e estar sintonizado com as alterações mais recentes.

O conhecimento obtido sobre o GraphQL servirá como uma boa base para quando você explorar outras bibliotecas e ferramentas. Se estiver pensando nos próximos passos para aperfeiçoar suas habilidades, eis alguns tópicos que você deve verificar:

Schema stitching

Schema stitching (literalmente, costura de esquemas) é um processo que permite criar um só esquema GraphQL a partir de várias APIs GraphQL. A Apollo disponibiliza algumas ferramentas ótimas para composição de esquemas remotos. Saiba mais sobre como desenvolver um projeto como esse consultando a documentação da Apollo (<http://bit.ly/2KcibP6>).

Prisma

Ao longo do livro, usamos o GraphQL Playground e o GraphQL Request: duas ferramentas da equipe do Prisma. O Prisma é uma ferramenta que transforma seu banco de dados existente em uma API GraphQL, não importa o banco de dados que você estiver usando. Enquanto uma API GraphQL está entre o cliente e o banco de dados, o Prisma se posiciona entre a API GraphQL e o banco de dados. É uma ferramenta de código aberto, portanto você pode implantar o seu serviço Prisma em ambiente de produção usando qualquer provedor de nuvem.

A equipe também lançou uma ferramenta relacionada chamada Prisma Cloud: uma plataforma de hospedagem para serviços Prisma. Em vez de ter que configurar a sua própria hospedagem, você pode usar o Prisma Cloud para gerenciar todas as suas preocupações com DevOps.

AWS AppSync

Outro novo personagem no ecossistema é o Amazon Web Services. Ele lançou um novo produto baseado no GraphQL e em ferramentas Apollo para simplificar o processo de configuração de um serviço GraphQL. Com o AppSync, você cria um esquema e então o conecta com sua fonte de dados. O AppSync atualiza os dados em tempo real e trata até mesmo mudanças de dados offline.

Canais Slack da comunidade

Outra ótima maneira de se envolver é associar-se a um dos muitos canais Slack da comunidade GraphQL. Você não só poderá estar em sintonia com as notícias mais recentes sobre o GraphQL como também poderá fazer perguntas, às vezes respondidas pelos criadores dessas tecnologias.

Você também pode compartilhar seu conhecimento com outras pessoas nessas comunidades em expansão, de qualquer lugar em que estiver:

- GraphQL Slack (<https://graphql-slack.herokuapp.com/>)
- Apollo Slack (<https://www.apollographql.com/#slack>)

À medida que prosseguir em sua jornada com o GraphQL, você poderá se envolver mais com a comunidade também como colaborador. Atualmente há projetos muito relevantes como React Apollo, Prisma e o próprio GraphQL com problemas em aberto, marcados com `help wanted` (precisa-se de ajuda). Sua colaboração em um desses problemas poderia ajudar muitos outros! Há também várias oportunidades para contribuir com novas ferramentas para o ecossistema.

Embora mudanças sejam inevitáveis, o terreno sob nossos pés, enquanto desenvolvedores de APIs GraphQL, é muito sólido. No coração de tudo que fizermos, estaremos criando um esquema e escrevendo resolvers a fim de atender aos requisitos de dados do esquema. Não importa quantas ferramentas surgirem para agitar o ecossistema, poderemos contar com a estabilidade da própria linguagem de consulta. Na linha de tempo das APIs, o GraphQL é bem recente, mas o futuro é muito promissor. Então vamos todos construir algo maravilhoso.

¹ N.T.: Um projeto Greenfield é um projeto que não apresenta restrições impostas por trabalhos anteriores (baseado em https://en.wikipedia.org/wiki/Greenfield_project).

Sobre os autores

Alex Banks e **Eve Porcello** são engenheiros de software e

instrutores, e vivem na cidade de Tahoe, na Califórnia. Com sua empresa, a Moon Highway, eles desenvolvem e oferecem um currículo de treinamento personalizado para clientes corporativos e online para o LinkedIn Learning. Também são autores do livro *Learning React*, da O'Reilly Media.

Colofão

O animal na capa de *Introdução ao GraphQL* é a água-perdigueira ou águia-de-bonelli (*Aquila fasciata*). Essa grande ave de rapina é encontrada em todo o sudeste da Ásia, no Oriente Médio e no Mediterrâneo, preferindo climas mais secos e áreas em que possa fazer ninhos em escarpas ou árvores altas. Tem uma envergadura média de aproximadamente 1,5 metro e distingue-se pela cabeça marrom escura e asas com a parte inferior branca, decorada com listras e manchas escuras.

Em geral, esse caçador discreto é silencioso fora do ninho e se alimenta principalmente de outras aves, incluindo até mesmo outras aves de rapina, mas sabe-se que ele se alimenta também de pequenos mamíferos e répteis. Apesar de sua propensão a se alimentar de outras aves de rapina, casais adultos são conhecidos pela afeição a filhotes, independentemente da linhagem, e já foram observados chocando ovos e filhotes recém-nascidos em ninhos abandonados, tanto da espécie *Aquila fasciata* quanto de outras espécies de rapina com as quais não se observa uma agressividade letal entre as espécies irmãs.

Muitos dos animais das capas dos livros da O'Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse animals.oreilly.com.

A imagem da capa foi extraída do livro *Brehms Tierleben*.

O'REILLY®

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações
nativas de nuvem



novatec

Bilgin Ibryam
Roland Huß

Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres.
- Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

[Compre agora e leia](#)

CANDLESTICK

Um método para ampliar lucros na Bolsa de Valores



novatec

Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos.

Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



AVALIANDO
EMPRESAS

INVESTINDO EM AÇÕES

A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS

novatec

CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)

Marcos Abe

MANUAL DE ANÁLISE TÉCNICA

ESSÊNCIA E ESTRATÉGIAS AVANÇADAS

TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE

novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá:

- os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado;
- identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos;
- estruturar e proteger operações por meio do gerenciamento de capital.

Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)

Fundos de Investimento Imobiliário

ASPECTOS GERAIS E
PRINCÍPIOS DE ANÁLISE

novatec

Roni Antônio Mendes

Fundos de Investimento Imobiliário

Mendes, Roni Antônio

9788575226766

256 páginas

[Compre agora e leia](#)

Você sabia que o investimento em imóveis é um dos preferidos dos brasileiros? Você também gostaria de investir em imóveis, mas tem pouco dinheiro? Saiba que é possível, mesmo com poucos recursos, investir no mercado de imóveis por meio dos Fundos de Investimento Imobiliário (FIIs). Investir em FIIs representa uma excelente alternativa para aumentar o patrimônio no longo prazo. Além disso, eles são ótimos ativos geradores de renda que pode ser usada para complementar a aposentadoria. Infelizmente, no Brasil, os FIIs são pouco conhecidos. Pouco mais de 100 mil pessoas investem nesses ativos. Lendo este livro, você aprenderá os aspectos gerais dos FIIs: o que são; as vantagens que oferecem; os riscos que possuem; os diversos tipos de FIIs que existem no mercado e como proceder para investir bem e com segurança. Você também aprenderá os princípios básicos para avaliá-los, inclusive empregando um método poderoso, utilizado por investidores do mundo inteiro: o método do Fluxo de Caixa Descontado (FCD). Alguns exemplos reais de FIIs foram estudados neste livro e os resultados são apresentados de maneira clara e didática, para que você aprenda a conduzir os próprios estudos e tirar as próprias

conclusões. Também são apresentados conceitos gerais de como montar e gerenciar uma carteira de investimentos. Aprenda a investir em FIs. Leia este livro.

[Compre agora e leia](#)