

GraphQL

A revolucionária linguagem de consulta e manipulação de dados para APIs



Sumário

- [ISBN](#)
- [Sobre o livro](#)
- [Para quem é este livro?](#)
- [Para me seguir:](#)
- [Sobre o autor](#)
- [Agradecimentos](#)
- [Prefácio](#)
- Parte 1 — Introdução ao GraphQL
 - [1 Conhecendo um novo mundo](#)
 - [2 Conceitos básicos do GraphQL](#)
- Parte 2 — GraphQL e Prisma Playground
 - [3 Conheça onde vamos brincar — Prisma](#)
 - [4 Buscando e alterando dados](#)
 - [5 Conhecendo melhor os types](#)
- Parte 3 — Criando uma aplicação
 - [6 Desenvolvimento front-end](#)
- Parte 4 — Programando um servidor
 - [7 Desenvolvimento back-end](#)

ISBN

Impresso e PDF: 978-85-7254-011-7

EPUB: 978-85-7254-012-4

MOBI: 978-85-7254-013-1

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

Sobre o livro

Neste livro, aprenderemos sobre o que realmente é o tão falado *GraphQL* e desenvolveremos uma aplicação no front-end e outra no back-end. Este livro está separado em quatro partes.

Na **Parte 1** seremos introduzidos ao mundo do GraphQL. Conheceremos essa ferramenta, quem a usa, onde podemos usá-la para facilitar nosso trabalho e aprenderemos seus conceitos básicos.

Chegando à **Parte 2**, nós aprenderemos os comandos do GraphQL em um ambiente online chamado *Prisma*, que nos poupa o trabalho de ter que criar uma aplicação para aprender a trabalhar com GraphQL.

Com o Prisma nos fornecendo um servidor gerado automaticamente, desenvolveremos uma aplicação front-end com JavaScript puro para consumir este serviço na **Parte 3**.

E, para finalizar, na **Parte 4** nós criaremos um servidor com serviço GraphQL que substituirá o Prisma utilizando o *Apollo*.

Para quem é este livro?

O GraphQL é independente de linguagem de programação ou banco de dados, então aprender sobre ele não exige um conhecimento avançado em programação.

Porém, este livro é destinado a desenvolvedores web que já possuam pelo menos um conhecimento básico com front ou back-end e queiram aprender um novo modo de construir aplicações modernas e flexíveis de forma rápida, simples e produtiva.

No final do livro, desenvolveremos uma aplicação completa (cliente e servidor), então conhecer um pouco de JavaScript ajudará no entendimento.

Para me seguir:

- **GitHub:** <https://github.com/hanashiro>
- **Facebook:** <https://www.facebook.com/TreinaWeb/>
- **YouTube:** <https://www.youtube.com/user/TreinaWeb>
- **Instagram:** <https://www.instagram.com/treinaweb/>
- **Meus cursos:** <https://treinaweb.com.br>
- **Blog:** <https://treinaweb.com.br/blog>

Sobre o autor



Figura -2.1: Akira Hanashiro

- Formado em Análise e Desenvolvimento de Sistemas pela Fatec (2011 - 2013)
- Pós-graduado em Projetos e Desenvolvimento de Aplicações Web pela UniFatea (2014 - 2015)
- MBA em Machine Learning pelo IGTI (2017 - 2019)

Sou desenvolvedor Web. Gosto de cozinhar, tocar violão e teclado, desenhar, malhar, games, séries, livros, estudar sobre hipnose, economia, finanças, desenvolvimento de jogos (Unity) e várias outras coisas, mas gosto muito mesmo de front-end.

Nasci em São Paulo, cresci no Japão e aos 15 anos voltei ao Brasil. Iniciei minha carreira em TI aos 18 anos, lecionando desenvolvimento de jogos digitais.

Já estudei várias linguagens de programação, mas a que me prendeu mesmo foi o JavaScript. Porém, utilizo a que for melhor dependendo da situação.

Trabalhei em projetos para empresas como Ambev, Siemens, Vivo, Itaú, Gauge, GoCart e Embraer.

Hoje em dia sou instrutor de cursos online na <https://treinaweb.com.br> e desenvolvedor nas horas vagas.

Agradecimentos

Agradeço a Deus por todas as oportunidades que tive na vida.

Dedico o livro aos meus pais, que sempre se esforçaram para me dar uma vida confortável, me cobrando apenas o estudo.

Muito obrigado à família e amigos que sempre me incentivam, me apoiam e confiam em minhas escolhas por mais insanas que possam parecer.

Obrigado a todos com quem já estudei e trabalhei, pois foi assim que adquiri minha experiência. Principalmente à TreinaWeb, pois está me ajudando a alcançar vários objetivos.

Obrigado à equipe da Casa do Código, principalmente Vivian Matsui, por me orientar e ajudar a realizar um dos meus objetivos deste ano: escrever meu primeiro livro.

E, claro, agradeço a você por ter escolhido este meu livro para obter mais conhecimento e evoluir em sua carreira.

Prefácio

A revolucionária linguagem de consulta e manipulação de dados, GraphQL, definitivamente veio para ficar. Ela tem conquistado grandes empresas nos mais diversos ramos, desde redes sociais, até mesmo no sistema financeiro. Tem sido colocada à prova em muitas aplicações, e vem demonstrando sua grande flexibilidade para resolver os mais variados tipos de problemas em APIs.

Neste livro, você vai descobrir o que vem a ser GraphQL, o porquê de ser tão poderosa, como implementá-la de forma rápida, com dois exemplos (em Prisma e em Node.JS + Apollo-Server), e como consumi-la de forma rápida, com dois exemplos (no GraphQL Playground e com JavaScript na Web).

Este livro vai desafiar você, em várias etapas, a ultrapassar os limites desta brochura, para que você realize outras implementações aqui não contempladas: uma grande oportunidade para aprender ainda mais.

Boa leitura!

Nic Marcondes

Parte 1 — Introdução ao GraphQL



CAPÍTULO 1 Conhecendo um novo mundo

Olá *Web Developer*! Então você tem interesse em conhecer o GraphQL? Ótimo, tenho certeza de que não vai se arrepender.

No começo, eu fiquei curioso por tanta gente estar falando sobre ele, mas para mim nunca tinha ficado clara a razão de ele existir e o que ele realmente fazia. Afinal, todos nós sobrevivemos até hoje sem ele, não é mesmo?

Mas quando o estudei a fundo, finalmente entendi o seu propósito e realmente percebi que em muitos cenários ele seria muito interessante.

Neste capítulo você vai conhecer o que realmente é o GraphQL. Boa leitura!

1.1 Conheça o GraphQL

Você sabe o que realmente é o GraphQL? Muitas pessoas pensam que sabem, mas costumam estar enganadas. Eis o que muitas vezes encontramos sendo dito nas comunidades:

- "Ah, é uma nova forma de fazer consultas para bancos não relacionais."
- "Não passa de um novo framework JavaScript para se criar APIs."
- "Ele está querendo substituir o SQL."
- "É só mais uma camada para colocar na aplicação e nos dar mais dor de cabeça."
- "É a biblioteca que vai substituir o REST."

Está tudo errado! Mas não as culpo, pois eu também pensava que era uma coisa totalmente diferente do que realmente é.

Mas para que você também o entenda, vamos imaginar um cenário com um determinado problema, e você conseguirá enxergar por

conta própria o que o GraphQL resolve.

O problema

Vamos imaginar que estamos em um projeto de um sistema que gerencia escolas. O back-end será só um, mas o lado do cliente terá várias aplicações bem distintas:

- **Sistema Administrador** - responsável pelo cadastro de escolas, professores, alunos, pais, grades curriculares etc.
- **Sistema do Professor** - organiza a agenda do professor, permite lançar notas de provas e trabalhos, fazer chamada, enviar mensagens para os pais de um aluno, dar advertências etc.
- **Sistema do Responsável** - tem uma lista dos filhos do responsável pela qual ele pode se comunicar com os professores, visualizar todas as notas, advertências, datas de provas, ver a grade de aulas etc.

Muitas APIs serão consumidas pelas três aplicações, tornando o back-end bem simples de se administrar e evitando código repetido. Cada um desses sistemas terá uma versão Web e uma versão *mobile*.

A versão mobile será mais resumida e poderá ser acessada offline. Também teremos que permitir que pessoas com internet móvel lenta tenham uma boa experiência, então não fará sentido enviar dados que não serão usados no mobile. Aí começam a surgir alguns problemas:

- O administrador, ao receber a lista de professores, terá os dados pessoais, de contato e endereço, que o responsável não deve ver ao receber a lista de professores;
- O professor tem acesso a todos os dados dos alunos no sistema Web, mas no mobile ele teria apenas as informações mais importantes que fossem úteis para o seu trabalho em sala de aula;

- O responsável tem todos os detalhes das atividades de seus filhos no sistema Web, mas o aplicativo mobile só exibirá informações mais básicas, como as notas tiradas;
- O professor acessa suas aulas e pelas aulas acessa os alunos delas. O responsável acessa seus filhos, pelos filhos acessa suas aulas e pelas aulas acessa o professor responsável por aquela disciplina. Ou seja: são os mesmos dados, mas acessados por caminhos diferentes.

O primeiro problema podemos resolver facilmente verificando qual o perfil de quem está fazendo a requisição dos dados. Assim, podemos enviar apenas o que aquele usuário tem permissão de ver. Mas e quanto aos outros problemas? Como enviar menos dados para a aplicação mobile?

E pior: pode ser que web e mobile usem bibliotecas que exijam um certo formato nos dados a serem exibidos. Então, além de menos dados, pode ser que o aplicativo precise dos dados em uma estrutura levemente diferente.

O que poderíamos fazer?

Uma solução rápida? Podemos pegar a API de listar professores, `/professores`, e criar outra, `/professores/mobile`. O mesmo para as outras APIs que no mobile teriam menos dados ou cujos dados precisariam estar levemente diferentes.

Quanto ao quarto problema, sem chance! Seriam duas APIs totalmente diferentes.

Parece algo bobo e simples, mas imagine esse sistema crescendo cada vez mais, e você tendo que, a cada alteração, mudar em dois lugares que são bem parecidos.

Isso foi só um exemplo, pois com o tempo podem aparecer mais lugares em que a API específica para mobile seja necessária. Não é preciso muita experiência para saber que algum dia isso pode trazer muito retrabalho.

O que poderia ser feito?

E se o front-end pudesse ter a capacidade de dizer ao back-end quais informações ele quer e até mesmo a estrutura desses dados? Não teríamos que ficar criando APIs repetidas e nem forçando uma aplicação a receber dados que ela não vai usar.

Essa é uma das principais funções do GraphQL! Com ele, o cliente pode dizer ao servidor quais dados ele quer e como é a estrutura que ele quer.

E também é comum que grandes sistemas utilizem mais de um banco de dados para aproveitar as vantagens de cada um. Podemos, por exemplo, ter dados relacionais no MySQL, guardar certas informações em forma de documentos no MongoDB e ter pequenos dados para consultas rápidas no Redis. Também seria bom não se preocupar em como ir a cada um desses dados, não é mesmo? É outro ponto em que o GraphQL pode ajudar!

1.2 Então, o que realmente é o GraphQL?

O GraphQL foi criado pelo Facebook, que começou a usá-lo em seus aplicativos em 2012. Foi anunciado ao público na React.js Conf 2015 e teve seu código aberto em 2016.

Podemos dizer que o GraphQL é uma linguagem de consulta e manipulação, assim como o SQL. Enquanto o SQL é um padrão para fazer consultas em bancos de dados, o GraphQL nos dá a possibilidade de indicar a uma API os dados que queremos.

Isso dá liberdade ao front-end de indicar o que ele realmente precisa a partir de um único endpoint disponibilizado pelo back-end. Sei que isso parece estranho e perigoso, mas acredite: é simples e seguro. Você entenderá melhor conforme formos avançando no livro.

Hoje em dia, o GraphQL é um projeto da GraphQL Foundation hospedado pela Linux Foundation e mantido por várias empresas como Airbnb, Apollo, Coursera, Elementl, Facebook, GitHub, Hasura, Prisma, Shopify, e Twitter.

1.3 Vantagens do GraphQL

Schema

No GraphQL nós declaramos a estrutura do dado, então o cliente sempre sabe a estrutura do dado que receberá. Isso também gera automaticamente uma documentação e validação de tipos.

Tudo em uma única requisição

Já precisou fazer uma requisição, pegar o ID do objeto e fazer uma segunda requisição? Chega disso! Com o GraphQL, você indica tudo o que precisa e recebe tudo certinho com uma única requisição.

Nem mais, nem menos

O cliente indica o que precisa receber.

Você é front-end? Não precisa mais ficar chamando alguém do back-end para dizer que tem dado faltando ou que estão vindo dados que você não vai usar.

Você é back-end? Não precisa mais ficar perguntando para alguém do front-end o que ele precisa e como é a estrutura que melhor atende à tela que ele vai criar.

Desenvolvimento rápido

Há bibliotecas que nos permitem trabalhar facilmente com GraphQL que já implementam de forma bem simples coisas como cache, melhoria na atualização da interface e atualizações em tempo real com Web Sockets.

Criação de APIs de forma rápida e simples

Você verá que vamos criar uma única API. E para que ela entregue algo ao cliente, utilizaremos bem pouco código.

1.4 Onde posso usar?

Você pode usar em qualquer lugar onde haja uma comunicação entre cliente e servidor via API, seja Web, desktop ou mobile.

Um cenário muito interessante para se usar o GraphQL é quando você vai expor a sua API para que terceiros a usem.

Provavelmente haverá situações em que essas pessoas já possuam um sistema funcionando com uma estrutura de dados própria, e cada campo com um nome diferente dos campos presentes nos seus dados. E há também chances de nem usarem 100% dos dados que a sua API fornece.

Com o GraphQL, essas pessoas poderão indicar exatamente os dados de que precisam na estrutura, com os campos nomeados de acordo com suas necessidades.

1.5 Nem tudo são flores

Como sempre, tudo tem um lado bom e um lado ruim. Se você trabalha há um tempo com programação já deve saber que tudo

depende de outra coisa. Raramente teremos uma resposta definitiva.

Então, não podemos basear nossas escolhas de ferramentas apenas porque algo está na moda e as pessoas dizem que é bom. A ferramenta precisa fazer sentido e trazer algum valor para nós durante o desenvolvimento de uma solução.

Vamos ver alguns problemas de se usar o GraphQL.

Complexidade da consulta

Imagine um sistema de uma livraria e buscamos os dados de um autor. Esta operação será bem simples e leve. Imagine o seguinte json como resultado:

```
{
  "nome": "Carlos Abreu",
  "pais": "Brasil",
  "site": "livrosdocarlos.exemplo.com"
}
```

Porém, se fizermos uma busca de vários autores, cada um com uma lista dos livros que escreveu e cada livro com os respectivos comentários feitos por clientes que compraram o livro, esta operação ficará mais pesada.

Imagine a seguinte resposta:

```
[
  {
    "nome": "Carlos Abreu",
    "pais": "Brasil",
    "site": "livrosdocarlos.exemplo.com",
    "livros": [
      {
        "titulo": "A Revelação",
        "ano": 2019,
        "comentarios": [
          {
            "nome": "Marta Silva",
```

```

        "avaliacao": 5,
        "comentario": "Muito interessante!"
    },
    ...
]
},
...
]

```

E ainda é possível fazer operações que retornem dados repetidos como, por exemplo, buscar um autor com sua lista de livros, e esta lista de livros vir junto com o nome do autor junto aos seus livros. Teríamos a seguinte resposta:

```

[
  {
    "nome": "Carlos Abreu",
    "pais": "Brasil",
    "site": "livrosdocarlos.exemplo.com",
    "livros": [
      "titulo": "A Revelação",
      "ano": 2019,
      "autor": {
        "nome": "Carlos Abreu",
        "pais": "Brasil",
        "site": "livrosdocarlos.exemplo.com",
        "livros": [
          "titulo": "A Revelação",
          "ano": 2019,
          "autor": {
            ...
          }
        ]
      }
    ]
  },
  ...
]

```

Estas operações mais complexas podem ser muito pesadas para o servidor, então pode ser necessário limitar a quantidade de níveis que podem ser acessados.

Avaliação de limites

Há casos em que disponibilizamos uma API a terceiros para que tenham acesso a um serviço. Um exemplo é a API do YouTube, que nos permite fazer busca de vídeos da plataforma e exibí-los em nossa aplicação.

Pode ser que você queira limitar a quantidade de requisições e apenas permitir mais operações mediante pagamento do seu serviço. Isso pode ser difícil no GraphQL, já que uma única requisição pode ser leve como também pode ser muito pesada, como vimos anteriormente.

Cache

Normalmente tratamos o cache das aplicações de acordo com a URL que acessamos. Cache é bom porque nos permite retornar dados que já foram acessados anteriormente com mais rapidez, e também diminui a quantidade de requisições de um cliente ao servidor. Mas lembra que eu disse que no GraphQL nós criamos apenas uma única API?

Observação: a lista original de desvantagens do GraphQL era outra, mas durante o desenvolvimento deste livro todos os itens foram solucionados e precisei pesquisar sobre outras desvantagens.

Mesmo assim, os itens aqui apresentados já possuem soluções sendo desenvolvidas, o que mostra como esta tecnologia está se desenvolvendo rapidamente e pode ser utilizada sem receio de aparecer problemas em sua implementação.

1.6 Quem usa?

Empresas como Coursera e Netflix estavam com problemas parecidos e começaram a trabalhar em projetos como o GraphQL.

Quando o GraphQL teve seu código aberto, Coursera cancelou seu projeto e adotou a solução do Facebook. A Netflix continuou com seu projeto, Falcor.

Hoje em dia muitas empresas usam o GraphQL, como Facebook, GitHub, Twitter, PayPal, The New York Times, KLM, GetNinjas, Dailymotion, Shopify e Pinterest. Até a própria Netflix hoje em dia utiliza o GraphQL.

Você pode ver uma lista mais completa de empresas que usam o GraphQL em: <https://graphql.org/users/>

1.7 Interessado? Interessada?

Neste capítulo nós conhecemos o que realmente é o GraphQL: uma linguagem de consultas de APIs e manipulação de dados. Juntos nós imaginamos um cenário em que o uso dele seria bem interessante.

Vimos suas vantagens e também aprendemos que em certos momentos ele não é a melhor solução. Portanto, o melhor é analisar bem as necessidades do projeto para definir as melhores ferramentas para podermos desenvolver uma solução.

CAPÍTULO 2

Conceitos básicos do GraphQL

Espero que a apresentação que fizemos no capítulo anterior tenha deixado um gostinho de "quero mais".

Agora nós veremos alguns conceitos básicos do GraphQL. Ainda não há código a ser executado para testar. Apenas teremos a apresentação da sintaxe para que você já se acostume antes de botarmos a mão na massa.

2.1 Conheça a SDL

Um esquema (*schema*) é uma representação de um objeto. Como usamos o GraphQL para fazer consultas de dados, ele precisa conhecer a estrutura desses dados para saber quais campos estão disponíveis. Por isso, precisamos de um modo de declarar esses esquemas.

Como uma das ideias bases do GraphQL é funcionar com qualquer linguagem de programação e framework, foi criada uma linguagem própria de declaração de esquemas, a *Schema Definition Language* (Linguagem de Definição de Esquema) ou, simplesmente, SDL. Assim podemos declarar nossos esquemas de forma única para qualquer linguagem.

Imagine que queremos definir o esquema de um aluno. Vamos ver como modelá-lo com a SDL:

```
type Aluno{
  id: ID!
  nomeCompleto: String!
  idade: Int
}
```

É diferente de muita coisa com que você deve estar acostumado, não é mesmo? Vamos por partes, então.

Os elementos mais básicos do esquema do GraphQL são os *types*. Com eles, nós definimos a estrutura dos objetos que poderão ser buscados (seus campos e respectivos tipos). Eles são declarados com a palavra-chave `type`.

Em seguida, declaramos o nome do nosso *type*, `Aluno`. Os campos ficam entre chaves e dentro delas nós indicamos um campo em cada linha. Primeiro vem o nome e depois o tipo do dado, separados por dois pontos.

Então temos, por exemplo, o campo com o nome **idade** que é do tipo `Int`. `Int` indica um campo que aceita apenas números inteiros. Mais adiante vamos conhecer os tipos que o GraphQL aceita.

Note que não há vírgulas ou ponto e vírgula separando os campos, nós simplesmente pulamos linha. O `!` após o tipo do dado indica que o campo é obrigatório. Então os campos **id** e **nomeCompleto** são obrigatórios, enquanto o campo **idade** pode ficar em branco.

Que tal agora criar um outro *type*? Teremos agora o *type* `Curso`, que terá uma lista de alunos.

```
type Curso{
  id: ID!
  disciplina: String!
  alunos: [Aluno!]!
}
```

Fizemos algo parecido com a estruturação de `Aluno`. Mas veja que agora temos um campo do tipo `Aluno`, que é um tipo que declaramos anteriormente. Como será uma lista de alunos, declaramos com colchetes.

O `!` de fora indica que a lista não pode ser vazia. Isso significa que, se não tivermos nenhum aluno para retornar, receberemos um *Array*

vazio. Sem esse `!` do lado de fora dos colchetes, na ausência de alunos, o campo `alunos` teria seu valor como `null` .

Já o `!` de dentro dos colchetes, junto ao nome `Aluno` , indica que os alunos da lista não poderão ter valor nulo. Isso significa que não teremos um *Array* com elementos vazios, apenas objetos do tipo `Aluno` . Sem esse `!` , poderíamos ter um *Array* assim: `[null, null]` .

Vamos agora indicar que cada aluno só pode estar matriculado em apenas um curso. Então, vamos arrumar o nosso *type* `Aluno` .

```
type Aluno{
  id: ID!
  nomeCompleto: String!
  idade: Int
  curso: Curso
}
```

Indicamos um curso do tipo `curso` . Note que não colocamos `!` porque o aluno pode estar sem nenhum curso. Também não colocamos colchetes, o que indica que cada aluno só pode ter um curso.

2.2 Fazendo as primeiras consultas (queries)

Falamos tanto sobre o GraphQL ter sido criado para se fazer consultas mas ainda nem sequer vimos como isso funciona.

Primeiro, temos que ter um esquema já declarado em nosso back-end, como o `Aluno` e `Curso` que vimos anteriormente. Então o front-end faria uma requisição a uma API, enviando o que ele quer que seja retornado. Chamamos isso de *query* (consulta).

Temos o esquema `Aluno` com os campos `id` , `nomeCompleto` , `idade` e `curso` . Imagine que a gente queira buscar todos os alunos, mas só os campos `nomeCompleto` e `idade` . Ficaria da seguinte maneira:


```
query {  
  allAlunos{  
    nomeCompleto  
    idade  
  }  
}
```

Indicamos com a palavra-chave `query` que queremos fazer uma consulta. Como consultas são as operações mais comuns do GraphQL, essa palavra pode ser omitida do código.

Em seguida, abrimos chaves para indicar que tipo de operação de consulta queremos fazer.

Como queremos todos os alunos, vamos pedir a operação `allAlunos`. Esses nomes de operações somos nós mesmos que definimos. Mais adiante veremos onde e como fazer isso.

Esse `allAlunos` é chamado de "campo raiz" (*root field*). Após ele, nós simplesmente indicamos o *payload*, que é o grupo de campos que queremos que sejam retornados.

Poderíamos enviar essa *query* para um back-end já com alguns dados em um banco de dados. Com essa consulta, teríamos uma resposta como o seguinte JSON:

```
{  
  "data": {  
    "allAlunos": [  
      {  
        "nomeCompleto": "Maria Souza",  
        "idade": 25  
      },  
      {  
        "nomeCompleto": "João Silva",  
        "idade": 20  
      }  
    ]  
  }  
}
```

E se quisermos que também venha o curso de cada aluno da nossa lista? Lembre-se de que `Aluno` possui um campo `curso` que é do tipo `Curso`, o qual possui os campos `id`, `disciplina` e `alunos`. Então basta indicarmos o campo `curso`, abrirmos chaves novamente e indicarmos o que queremos que seja retornado.

```
{
  allAlunos{
    idade
    nomeCompleto
    curso{
      id
      materia: disciplina
    }
  }
}
```

Com a adição de `curso` e seus respectivos campos, teríamos um retorno como o seguinte JSON:

```
{
  "data": {
    "allAlunos": [
      {
        "idade": 25,
        "nomeCompleto": "Maria Souza",
        "curso": {
          "id": "123",
          "materia": "GraphQL"
        }
      },
      {
        "idade": 20,
        "nomeCompleto": "João Silva",
        "curso": {
          "id": "321",
          "materia": "RxJS"
        }
      }
    ]
  }
}
```

```
}  
}
```

Veja que dessa vez nós já omitimos a palavra-chave `query` . Na consulta, nós alteramos a posição dos campos `nomeCompleto` e `idade` do aluno. Isso fez com que o JSON retornado também tivesse a posição dos campos alterada.

Repare também que, ao indicar o campo `disciplina` , nós colocamos a palavra `materia` e depois de dois pontos (`:`) colocamos o nome do campo, `disciplina` . E o resultado foi que o campo `disciplina` veio como `materia` . Ou seja, podemos até mesmo renomear o nome dos campos que serão retornados.

A possibilidade de renomear campos é muito útil caso você disponibilize a sua API a terceiros e eles já possuam uma estrutura com outros nomes. Isso evitaria o trabalho de reestruturar objetos por estarem com nomes de campos ou estruturas diferentes.

2.3 Passando parâmetros para as consultas

Poder passar parâmetros para nossas consultas é muito importante, já que nem sempre vamos querer que qualquer dado seja retornado. Precisamos de uma maneira de indicar o que queremos com mais detalhes.

Com isso poderemos indicar, por exemplo, que queremos que apenas três dados sejam retornados, ou que queremos todos os alunos que possuam o nome **João** ou que os alunos venham ordenados pelo nome em ordem alfabética.

```
{  
  allAlunos(first: 3){  
    nomeCompleto  
    idade
```

```
}  
}
```

Para indicar parâmetros, basta passá-los entre parênteses junto ao campo raiz. No exemplo mostrado, indicamos que queremos que sejam retornados apenas os três primeiros objetos da lista que o nosso back-end retornaria.

Os parâmetros disponíveis somos nós mesmos que indicamos, e podemos passar vários ao mesmo tempo. Mais adiante você verá como declarar isso com a SDL.

2.4 Criando, alterando e apagando dados (mutations)

O GraphQL não serve apenas para buscar dados. Ele também nos permite criar, alterar e apagar dados. Como essas três operações causam modificações, no GraphQL nós as chamamos de *mutations*.

As *mutations* nos permitem indicar ao back-end que queremos fazer alguma modificação nos dados presentes em nosso banco de dados ou criar um novo. Quando o back-end receber esses comandos, nós vamos escrever um código para fazer a conexão com o banco de dados para fazer a operação desejada. Veremos como programar o back-end mais para frente.

Mutations seguem a mesma estrutura que vimos anteriormente com as *queries*, mas é obrigatório indicar a palavra-chave `mutation`.

Vamos ver como fazer para criar um Aluno.

```
mutation{  
  createAluno(nomeCompleto: "Akira Hanashiro", idade: 25){  
    id  
  }  
}
```

Primeiro, indicamos a palavra-chave `mutation` . Em seguida, temos o campo raiz que indica o nome da operação, `createAluno` . As operações somos nós mesmos que criamos, assim como a `allAlunos` . Logo veremos como declarar essas operações.

Uma boa prática para o nome das operações é indicar a ação que se quer fazer junto com o nome do esquema com o qual se está trabalhando. Então aqui no nosso exemplo de *mutations* teríamos basicamente as operações `createAluno` para criar um aluno, `updateAluno` para atualizar um aluno existente e `deleteAluno` para apagar um aluno.

Depois passamos os parâmetros necessários para a operação dentro de parênteses. Como estamos criando um `Aluno` , precisamos passar o valor dos campos `nomeCompleto` e `idade` . Lembre-se de que definimos no nosso esquema de `Aluno` que o campo `idade` não é obrigatório, então não daria erro se nós não tivéssemos passado a `idade` como parâmetro.

O campo `id` não é passado na criação porque ele é gerado no banco de dados durante a criação.

E por fim nós indicamos o que queremos de retorno. Podemos pedir todos os campos do aluno criado, assim como em uma *query*, mas neste exemplo nós pedimos apenas para retornar o `id` que foi gerado. Afinal, se estamos criando um aluno, temos todos os dados aqui. Só não temos ainda o `id` , pois ele é gerado no banco de dados.

Com isso teríamos um retorno como o seguinte:

```
{
  "data": {
    "createAluno": {
      "id": "cjbqocnoyfehq0121mcx584gc"
    }
  }
}
```

```
}  
}
```

E para inserir um `Curso` bastaria executar um `createCurso`. Depois bastaria ligar os cursos e alunos. Mas isso será assunto de outro capítulo.

A alteração e remoção de dados seguem o mesmo padrão. Veremos com mais detalhes quando finalmente botarmos nossas mãos no código.

2.5 Sendo notificado sobre alterações de dados (subscriptions)

Até aqui vimos sobre fazer consultas e modificar dados. Com apenas isso já estaria bom, não é mesmo?

Mas hoje em dia é muito comum a necessidade de manter as nossas aplicações com os dados atualizados em tempo real. Um bom exemplo é o Facebook: não precisamos ficar atualizando a página para ver se recebemos comentários, curtidas ou mensagens. Elas são atualizadas em nossas telas automaticamente assim que algum contato nosso fizer alguma ação.

É aí que entra outra parte muito importante do GraphQL, as *subscriptions*. Por meio delas o front-end pode indicar ao back-end que quer ser notificado sobre algum evento, como uma operação de *mutation*.

Imagine que criamos um aplicativo que exibe uma lista de alunos e que fizemos uma *subscription* indicando que queremos saber sobre novos alunos. Com isso, cada aluno que for criado a partir de uma *mutation* será enviado para a nossa aplicação e nós receberemos os dados dele. Dessa maneira poderemos atualizar a nossa lista de alunos automaticamente.

Voltando para o nosso esquema de `Aluno` , teríamos o seguinte código para indicar uma inscrição para sermos notificados sobre a criação de um novo aluno:

```
subscription{
  newAluno{
    nomeCompleto
    idade
  }
}
```

Primeiro, indicamos a palavra-chave `subscription` . Em seguida, temos o campo raiz que indica o nome da operação, `newAluno` . Isso indica que queremos ser notificados sobre um novo aluno. Podemos especificar as operações cujas notificações desejamos receber (criação, alteração e remoção).

Também somos nós que definimos os nomes das operações aqui e veremos como declarar isso a seguir. Quando um aluno for criado com a operação `createAluno` ficaremos esperando por um novo aluno. Então demos o nome de `newAluno` .

No final, como já vimos anteriormente, passamos os nomes dos campos que queremos receber. Assim que um novo aluno for criado, receberemos por *WebSockets* os novos dados.

Então imagine que acabamos de inserir um novo aluno. A nossa aplicação receberá o novo dado inserido.

```
{
  "newAluno": {
    "nomeCompleto": "Carlos Monteiro",
    "idade": 22
  }
}
```

WEBSOCKETS

A comunicação entre cliente e servidor normalmente é feita quando o cliente faz uma requisição ao servidor, criando uma conexão. Quando o servidor responde, essa conexão é encerrada. Um servidor não pode enviar dados a um cliente sem este ter feito uma requisição.

WebSocket é uma tecnologia na qual uma conexão entre cliente e servidor é criada e mantida, permitindo que tanto cliente quanto servidor enviem dados livremente em ambas as direções. Para saber mais, acesse: <https://youtu.be/VQ4cAHbWi0Y>

2.6 Definindo esquemas

Vimos as operações que o GraphQL nos permite fazer: *queries*, *mutations* e *subscriptions*. Mas seria um grande problema de segurança se pudéssemos fazer qualquer operação, não é mesmo? E de onde será que saíram aqueles nomes de operações como "allAlunos", "createAlunos" e "newAluno"?

No começo deste capítulo nós usamos a SDL para declarar os *types* `Aluno` e `Curso`. Na verdade não é só isso que devemos declarar. Cada tipo de operação que será aceita pelo back-end também deve ser declarado no esquema. Isso significa que temos que declarar nossas *queries*, *mutations* e *subscriptions*, incluindo se aceitam ou não parâmetros.

Isso é muito importante porque nos permite indicar as operações que o front-end poderá fazer. É como um contrato entre cliente e servidor.

Você se lembra da nossa operação de buscar todos os alunos?


```
query {  
  allAlunos{  
    nomeCompleto  
    idade  
  }  
}
```

Temos aqui uma *query* em que a operação se chama `allAlunos` e que retorna dados do *type* `Aluno`. Para permitir essa consulta, precisamos fazer a seguinte declaração em nosso esquema:

```
type Query{  
  allAlunos: [Aluno!]!  
}
```

Vimos que a palavra-chave `type` serve para declararmos nossos tipos, como fizemos com `Aluno` e `Curso`. Mas quando estamos escrevendo nosso esquema para a nossa API teremos também três *types* especiais: `Query`, `Mutation` e `Subscription`. Dentro deles nós declaramos as operações que vamos aceitar do cliente e o que cada uma delas retorna.

Então, para aceitar uma consulta de `Aluno` indicamos o *type* `Query`. Dentro dele, nós declaramos o nome da operação de retornar todos os alunos. As boas práticas nos indicam usar a palavra *all* junto com o nome do *type* no plural. Por isso, criamos o nome da operação como `allAlunos`.

Após dois pontos, nós indicamos o que essa operação retorna. Indicamos que é um *Array* de `Aluno`. Lembre-se de que o `!` indica que o valor retornado é obrigatório, não pode ser nulo. Então o `Aluno!` impede que o *Array* possua valores nulos, e o último `!`, fora dos colchetes, indica que o valor retornado também não pode ser nulo. Isso significa que, no caso de não haver nenhum aluno, retornaremos um *Array* vazio em vez de *null*.

Depois também vimos que podemos passar parâmetros.

```
{
  allAlunos(first: 3){
    nomeCompleto
    idade
  }
}
```

Mas também precisamos explicitar que parâmetros são permitidos. Para declarar os parâmetros, basta passá-los entre parênteses. Apenas o nome e o tipo de dado do parâmetro são necessários na declaração do esquema. A ação que eles farão em nossas operações serão implementadas por nós em nosso código no back-end.

```
type Query{
  allAlunos(first: Int): [Aluno!]!
}
```

Depois aprendemos sobre *mutations* e *subscriptions*.

```
mutation{
  createAluno(nomeCompleto: "Akira Hanashiro", idade: 25){
    id
  }
}

subscription{
  newAluno{
    nomeCompleto
    idade
  }
}
```

Para permitir essas operações basta seguir o mesmo padrão e declará-las em nosso esquema.

```
type Mutation{
  createAluno(nomeCompleto: String!, idade: Int): Aluno!
}

type Subscription{
```

```
    newAluno: Aluno!  
}
```

Nosso esquema completo ficaria da seguinte maneira:

```
type Aluno{  
    id: ID!  
    nomeCompleto: String!  
    idade: Int  
    curso: Curso  
}
```

```
type Curso{  
    id: ID!  
    disciplina: String!  
    alunos: [Aluno!]!  
}
```

```
type Query{  
    allAlunos(first: Int): [Aluno!]!  
}
```

```
type Mutation{  
    createAluno(nomeCompleto: String!, idade: Int): Aluno!  
}
```

```
type Subscription{  
    newAluno: Aluno!  
}
```

2.7 Preparado? Preparada?

Aprendemos alguns conceitos básicos do GraphQL como as *queries*, *mutations* e *subscriptions*. Também fomos introduzidos à sintaxe usada para declararmos esquemas e fazer as operações.

Agora você está pronto para mergulhar neste novo mundo. A partir deste ponto nós finalmente colocaremos nossas mãos no código.

Tudo pronto?

Parte 2 — GraphQL e Prisma Playground



CAPÍTULO 3

Conheça onde vamos brincar — Prisma

Agora que já conhecemos a base do GraphQL (*schemas*, *queries*, *mutations* e *subscriptions*), finalmente começaremos a executar um pouco de código.

Para não termos o grande trabalho de criar um back-end completo logo no início de nossos estudos, vamos utilizar o **Prisma**, um serviço gratuito que monta um servidor completo com GraphQL a partir do esquema que vamos escrever.

Isso nos possibilitará ter um ambiente com GraphQL onde poderemos testar comandos sem nos preocuparmos com programação no momento.

No final do livro veremos como montar um servidor passo a passo com Node.js.

3.1 Configurando o Prisma

Antes de começarmos, você precisa ter o Node.js instalado em seu computador. Caso ainda não o tenha, você pode obtê-lo em <https://nodejs.org/>.

Com o Node.js já instalado, abra o seu terminal e execute o comando: `npm i -g prisma`. Isso vai instalar o **Prisma** em sua máquina, permitindo-nos acessá-lo diretamente do terminal.

Assim que a instalação for finalizada, vamos criar um diretório para guardar nossos códigos que escreveremos durante este livro. Você pode dar o nome que quiser para este diretório.

Dentro do diretório criado, vamos abrir o terminal e executar o comando `prisma init cursos-online`. Isso gera os arquivos básicos para podermos criar um novo projeto no Prisma.

Utilize as setas do teclado para selecionar a opção *Demo Server*. Esta opção vai disponibilizar um novo ambiente com banco de

dados para podermos usar. As demais opções são para o caso de já termos um servidor ou banco de dados.

```
$ prisma init cursos-online
? Set up a new Prisma server or deploy to an existing server?
  Use existing database      Connect to existing database
  Create new database        Set up a local database using Docker

  Or deploy to an existing Prisma server:
  > Demo server              Hosted demo environment incl. database (requires login)
  Use other server          Manually provide endpoint of a running Prisma server
(Move up and down to reveal more choices)
```

Figura 3.1: Iniciando projeto Prisma pelo terminal

Caso você ainda não esteja autenticado, o seu navegador vai se abrir para que possamos fazer login no Prisma. Caso ainda não tenha uma conta no Prisma, você pode fazer login com uma conta do GitHub. Assim que fizer login, uma pequena mensagem no seu navegador dirá que você já pode fechá-lo.

Voltando ao terminal você verá que agora ele fará algumas perguntas para a base de configuração do serviço que será criado. Serão perguntas sobre a preferência da localização do servidor a ser utilizado, nome do serviço a ser criado (será utilizado por padrão o nome que demos, `cursos-online`) e nome do estágio de desenvolvimento da aplicação (será usado `dev` por padrão). Basta ir pressionando a tecla `Enter` para continuar, pois as respostas padrões já servirão para os nossos estudos.

Se olharmos para nosso diretório, veremos que um diretório com o nome `cursos-online` foi criado. É ali que mora o nosso projeto.

3.2 Declarando esquemas para o Prisma

Abra o recém-criado diretório `cursos-online` . Você verá dois arquivos: `datamodel.graphql` e `prisma.yml` .

O `prisma.yml` possui as configurações básicas do nosso projeto, então nós não vamos mexer nele. O arquivo `datamodel.graphql` possui um simples esquema de usuário (*User*). Ele vem só de exemplo, então poderemos apagá-lo.

Veja que há um elemento que não havíamos visto antes: `@unique` . Trata-se de uma *constraint* do próprio Prisma. *Constraints* são comandos que nos possibilitam indicar restrições a um campo de um banco de dados.

```
type User {  
  id: ID! @unique  
  name: String!  
}
```

Como o Prisma cria um servidor com banco de dados automaticamente para nós, não teremos acesso a nenhum código no servidor ou banco de dados. Então ele possui essas *constraints* para nos dar mais controle sobre o comportamento dos dados no banco de dados.

Como o foco do livro é o GraphQL, não vamos nos aprofundar nas funcionalidades específicas do Prisma, como *constraints*. Veremos apenas o suficiente para podermos iniciar um servidor sem precisar programar no momento.

Caso queira saber mais informações sobre as funcionalidades específicas do Prisma, acesse <https://prisma.io/>.

No exemplo que podemos ver no conteúdo do arquivo `datamodel.graphql` , a *constraint* `@unique` indica um campo que deve ser único no banco de dados, ou seja, não pode haver nenhum valor repetido.

Quando criamos um campo do tipo `ID!` no Prisma, inserir a *constraint* `@unique` é obrigatório. Mas lembre-se: essa *constraint* é do Prisma, e não do GraphQL. Portanto, quando formos trabalhar fora do Prisma, vamos apagá-la.

Vamos apagar o conteúdo do arquivo `datamodel.graphql` e colocar o nosso esquema de `Aluno` e `Curso` que vimos anteriormente. Lembre-se de que agora também temos que acrescentar a *constraint* `@unique` nos campos `id`.

```
type Aluno{
  id: ID! @unique
  nomeCompleto: String!
  idade: Int
  curso: Curso
}

type Curso{
  id: ID! @unique
  disciplina: String!
  alunos: [Aluno!]!
}
```

Note que só declaramos `type`. Não há nenhuma `query`, `mutation` ou `subscription`. Isso se deve ao fato de o Prisma já gerar tudo automaticamente a partir dos *types* que declaramos neste arquivo.

Após salvar o arquivo, abra o terminal no diretório `cursos-online` e execute o comando: `prisma deploy`. Esse comando vai enviar nossos arquivos para o Prisma, que o utilizará para criar um servidor, configurar um banco de dados e disponibilizar uma API com suporte aos comandos do GraphQL que aprendemos no capítulo passado.

Após o projeto ser criado, uma mensagem vai exibir duas URLs que serão geradas para nosso projeto. A URL HTTP servirá para fazermos *queries* e *mutations*, e a URL WS servirá para trabalharmos com *subscriptions*.

Não utilizaremos essas URLs no momento, pois agora nós vamos acessar o ambiente online no próprio site do Prisma para executar nossos comandos. Nós as utilizaremos quando formos desenvolver nosso front-end. Não se preocupe em guardar as URLs, pois teremos acesso a elas no próprio ambiente em que vamos trabalhar.

Caso queira alterar o esquema, basta alterar o arquivo `datamodel.graphql` e executar o comando `prisma deploy` novamente.

3.3 Venha brincar no playground!

Neste momento, já temos um servidor com banco de dados e uma API do GraphQL pronta sem termos programado nada, graças ao Prisma.

Neste primeiro momento, vamos utilizar o serviço do Prisma, executando comandos do GraphQL, a partir de uma ferramenta que o próprio Prisma nos disponibiliza, o GraphQL Playground.

O GraphQL Playground é uma ferramenta que se conecta ao nosso servidor e nos permite executar comandos do GraphQL. Portanto, poderemos inserir, alterar, remover e buscar dados diretamente nele, sem a necessidade de criar um front-end no momento.

Para iniciar o GraphQL Playground, com o terminal aberto no diretório do projeto, execute o comando `prisma playground`. A ferramenta será servida no endereço `http://localhost:3000/playground`. Ao executar esse comando, o seu navegador já abrirá automaticamente neste endereço.



```
$ prisma playground
Serving playground at http://localhost:3000/playground
```

Figura 3.2: Terminal iniciando o Playground

Ao abrir o navegador, teremos uma tela como a seguinte imagem.

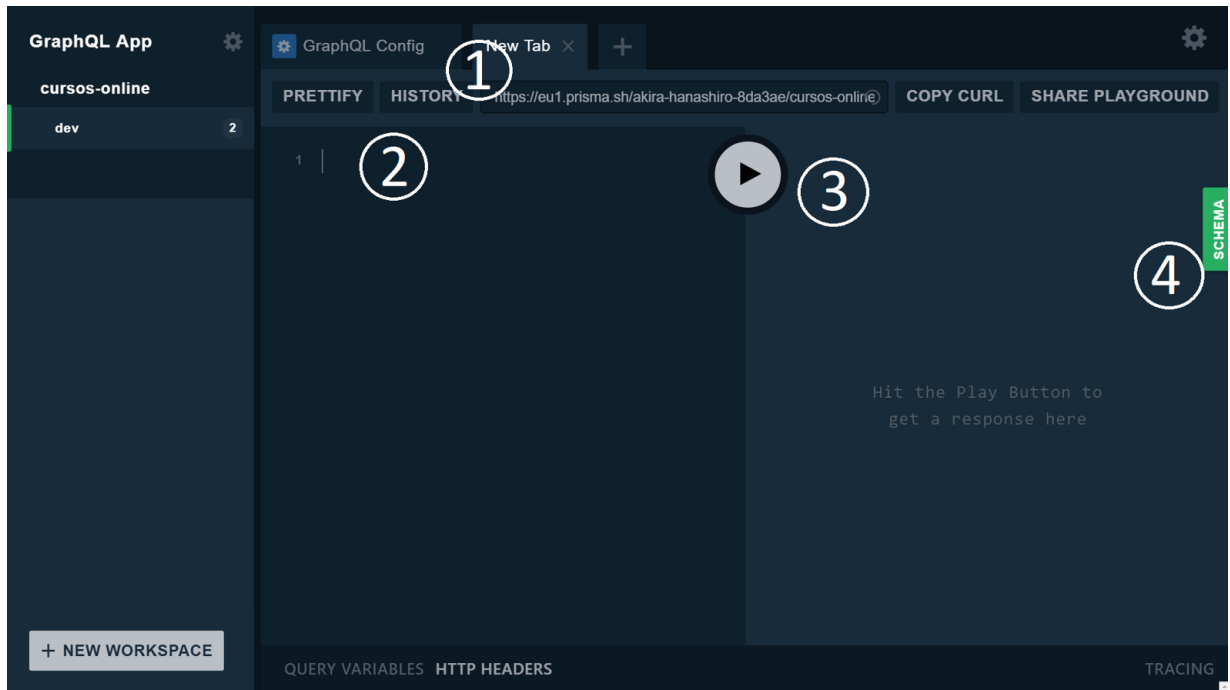


Figura 3.3: Prisma Playground

Próximo ao número 1 temos um endereço. Este é o endereço público do GraphQL Playground para acessar o seu projeto. Isso significa que você pode abrir a ferramenta por este endereço em vez de executar o comando `prisma playground`, e também compartilhar com quem você quiser.

O número 2 indica onde vamos escrever nossos comandos. O botão próximo ao número 3 é onde vamos clicar para executar o nosso comando (ou você pode executar pelo atalho `Ctrl + Enter`). A área onde está o número 3 é onde vai aparecer a resposta do servidor.

O botão ao lado do número 4, se clicado, exibirá todos os comandos que temos disponíveis. Quando criarmos nosso servidor, veremos que somos nós que definimos os comandos disponíveis.

O Prisma já cria todos os possíveis comandos automaticamente a partir do `type` que nós declaramos no arquivo `datamodel.graphql`.

3.4 Ansioso? Ansiosa?

Neste capítulo, nós vimos como preparar um ambiente com servidor, banco de dados e API do GraphQL automaticamente com o Prisma. Também conhecemos o GraphQL Playground, no qual finalmente vamos executar nossos comandos.

No capítulo seguinte nós utilizaremos o GraphQL Playground para inserir, remover, alterar e buscar dados do nosso banco de dados por meio de *queries*, *mutations* e *subscriptions*.

CAPÍTULO 4

Buscando e alterando dados

Finalmente chegou o momento de colocar em prática o que aprendemos até agora. Vamos utilizar o Playground do Prisma para executar alguns comandos. Ele também será um ótimo lugar para aprofundarmos nossos conhecimentos sobre o GraphQL.

Lembre-se de que para iniciar o Playground basta abrir o terminal no diretório do projeto e executar o comando `prisma playground`.

4.1 Inserindo dados

Para poder buscar, alterar e remover nossos dados, precisamos deles em nosso banco de dados. Então vamos iniciar nossas atividades com a criação de dados.

Como vimos anteriormente, utilizamos *mutations* para inserir dados. Então precisamos escrever que queremos uma *mutation* e, dentro dela, nós indicamos o nome da operação que queremos executar. Quando formos criar nosso servidor, veremos que somos nós que definimos o nome das operações. Como o Prisma gera tudo automaticamente, já teremos várias operações prontas.

Para criar um novo aluno, o Prisma gerou a operação `createAluno`. Assim que você começar a escrever, o Prisma vai sugerir algumas opções, tentando adivinhar o que você quer, como pode ser visto na imagem a seguir.

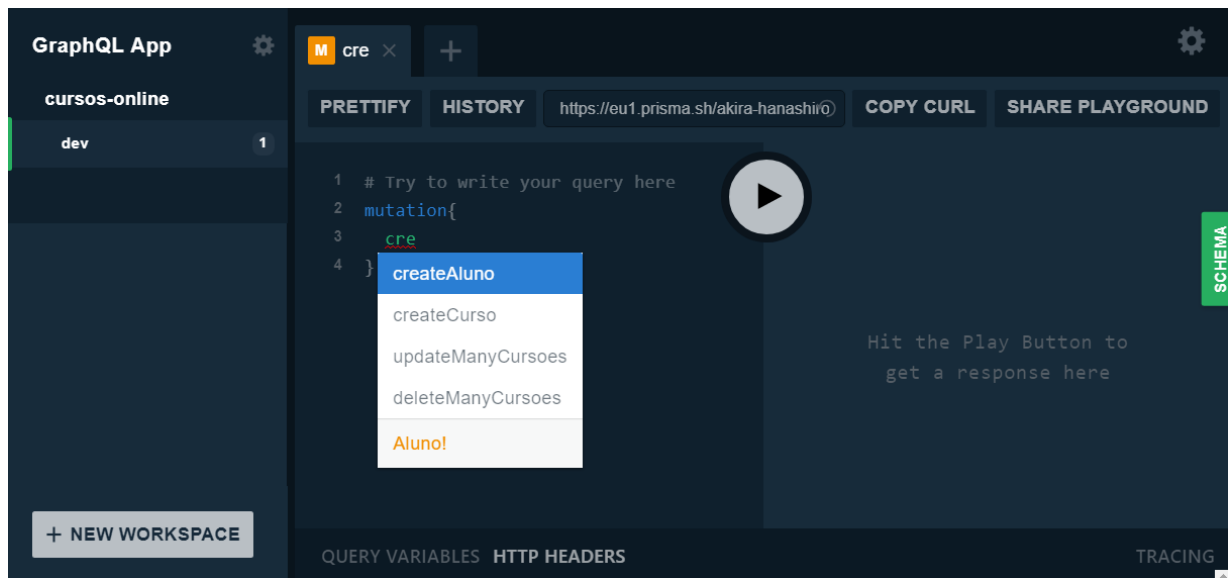


Figura 4.1: Playground — sugestão automática de comandos

Sempre que quiser uma sugestão do Prisma sobre o que pode ser escrito, basta pressionar as teclas `Ctrl + Barra de Espaço`. Isso facilita muito nosso trabalho, pois nos poupa de abrir a documentação a todo momento para procurar o nome exato de algum comando disponível.

Vamos lembrar o modo de se inserir um novo dado com GraphQL:

```
mutation{
  createAluno(
    nomeCompleto: "Maria Silva",
    idade: 25
  ){
    id
  }
}
```

Indicamos o nome da operação (`createAluno`), e passamos os valores dos campos dentro de parênteses (`nomeCompleto` e `idade`). Dentro de chaves nós indicamos o nome dos campos que queremos que sejam retornados. Aqui indicamos apenas o campo `id`, que é gerado automaticamente no banco de dados.

Acontece que no Prisma há uma pequena alteração: em vez de passar os campos diretamente dentro de parênteses, devemos passar dentro de um campo chamado `data`. É apenas algo que o Prisma implementa para manter a organização, mas quando formos criar nosso próprio servidor teremos a liberdade de criar nossa API como quisermos.

Então, para inserir um novo dado na API disponibilizada pelo Prisma, basta inserir os campos dentro de um campo `data`, e teremos o comando no seguinte formato:

```
mutation{
  createAluno(
    data: {
      nomeCompleto: "Maria Silva",
      idade: 25
    }
  ){
    id
  }
}
```

Aperte o botão com o ícone de Play ou pressione as teclas `Ctrl + Enter` para executar o comando. A resposta da API aparecerá no painel da direita com o campo que pedimos que fosse retornado, `id`.



Figura 4.2: Playground — criação de aluno

4.2 Busca simples

Agora que já temos um aluno em nosso banco de dados, vamos fazer uma consulta para que esse dado possa ser retornado sempre que quisermos.

Você se lembra de que anteriormente vimos a operação `allAlunos` ? Esse é um nome que podemos escolher para a nossa operação quando formos criar nosso próprio back-end, mas o Prisma simplesmente tenta passar o nome do esquema para o plural. E como ele só funciona em inglês, ele vai tentar seguir as regras da língua inglesa para passar "aluno" para o plural, resultando na palavra "alunos". Infelizmente, ele não possibilita alterar esses nomes no momento. Como dito anteriormente, em caso de dúvidas sobre quais comandos estão disponíveis, basta pressionar as teclas `Ctrl + Espaço`.

Para retornar todos os alunos, basta indicarmos que queremos uma query e passar o nome da operação, `alunos`. Em seguida, passamos o nome dos campos que queremos que sejam retornados.

```
query{
  alunos{
    id
    nomeCompleto
    idade
  }
}
```

Lembre-se também de que, quando estamos escrevendo *queries*, a palavra `query` pode ser omitida, deixando nosso comando da seguinte forma:

```
query{
  alunos{
    id
    nomeCompleto
  }
}
```



```
        idade
      }
    }
  }
```

Teremos a seguinte resposta:

```
{
  "data": {
    "alunos": [
      {
        "id": "cjk7k0ero1plm0b98mwgzghxv",
        "nomeCompleto": "Maria Silva",
        "idade": 25
      }
    ]
  }
}
```

4.3 Atualizando dados

Vamos alterar o nosso dado de aluno, com a operação `updateAluno`. Para fazer essa atualização de dado, precisamos passar os novos valores dos campos que queremos alterar e um identificador para o GraphQL saber qual objeto deve ser alterado. No nosso caso, nosso identificador é o campo `id`.

Essas informações serão passadas como parâmetros. Os novos valores dos campos devem ser passados no parâmetro `data` e o identificador deve ser passado no parâmetro `where`.

Após os parâmetros, podemos indicar os campos que queremos que sejam retornados do objeto que acabou de ser atualizado. No exemplo a seguir, pegamos nosso aluno e alteramos sua idade para o valor 26.

```
mutation{
  updateAluno(
```

```

        data: {
            idade: 26
        }
        where: {
            id: "cjk7k0ero1plm0b98mwgzghxv"
        }
    ){
        id
        nomeCompleto
        idade
    }
}

```

Teremos a seguinte resposta:

```

{
  "data": {
    "updateAluno": [
      {
        "id": "cjk7k0ero1plm0b98mwgzghxv",
        "nomeCompleto": "Maria Silva",
        "idade": 26
      }
    ]
  }
}

```

4.4 Removendo dados

O que falta agora é aprendermos a apagar o dado existente. Isso é bem mais simples, pois basta executar a operação `deleteAluno` e passar o identificador do objeto que queremos apagar para o parâmetro `where` .

Por fim, também podemos indicar o nome dos campos que queremos que sejam retornados do objeto que estamos apagando. Isso pode ser útil caso você queira, por exemplo, listar ao usuário os

dados que foram apagados. Após o objeto ter sido apagado não há mais como recuperá-lo, então caso queira exibi-lo, esta é a última chance.

```
mutation{
  deleteAluno(
    where: {
      id: "cjk7k0ero1plm0b98mwgzghxv"
    }
  ){
    id
    nomeCompleto
    idade
  }
}
```

Teremos a seguinte resposta:

```
{
  "data": {
    "deleteAluno": [
      {
        "id": "cjk7k0ero1plm0b98mwgzghxv",
        "nomeCompleto": "Maria Silva",
        "idade": 26
      }
    ]
  }
}
```

O GraphQL exige que toda operação retorne pelo menos algum dado.

Mesmo que você não precise que nada seja retornado ao apagar um objeto, precisamos indicar no mínimo um campo para ser retornado.

Agora nosso banco está vazio, pois apagamos o único dado que tínhamos. Execute a operação `alunos` novamente e teremos um

array vazio como retorno.

```
{
  "data": {
    "alunos": []
  }
}
```

4.5 Relacionando dados

Vimos as operações com apenas um aluno, mas lembre-se de que `Aluno` também possui o campo `curso`. Como fazer para criar um aluno com um curso e manter esse relacionamento? Temos três possibilidades: criar um aluno com um curso, conectar um aluno e um curso existentes e remover essa conexão.

Criando dados relacionados

Vamos começar criando um novo aluno e um novo curso. Primeiro começamos escrevendo da mesma maneira escrevemos quando criamos um aluno.

O que vamos adicionar agora é o campo `curso`. Como há a possibilidade de criar um novo curso ou conectar o novo aluno a um curso já existente, precisamos indicar ao Prisma que queremos criar um novo curso. Então, passamos o campo `create` e em seguida passamos os campos do novo curso.

No final, indicamos os campos que queremos que sejam retornados.

```
mutation{
  createAluno(
    data: {
      nomeCompleto: "Maria Silva"
      idade: 26
      curso: {
        create: {
```

```

        disciplina: "GraphQL"
      }
    }
  }
){
  id
  nomeCompleto
  idade
  curso{
    id
    disciplina
  }
}
}

```

Teremos a seguinte resposta:

```

{
  "data": {
    "createAluno": {
      "id": "cjkhiveqcdnuw0b58nfc0zobt",
      "nomeCompleto": "Maria Silva",
      "idade": 26,
      "curso": {
        "id": "cjkhiveqgdnuw0b58jrg55mho",
        "disciplina": "GraphQL"
      }
    }
  }
}

```

Agora temos um novo aluno e um novo curso criados, e eles estão conectados. Podemos buscar alunos e também indicar que queremos que venham as informações do curso em que o aluno está.

```

{
  alunos{
    id
    nomeCompleto
    curso{

```

```

        id
        disciplina
      }
    }
  }
}

```

E também podemos fazer a busca na outra direção, ou seja, procurar por cursos e pedir para virem os alunos que estão ligados a ele.

```

{
  cursoes{
    id
    disciplina
    alunos{
      nomeCompleto
      idade
    }
  }
}

```

Note que chamamos o campo de `alunos`, e não `alunoes`. Isso porque nós indicamos no esquema que `Curso` possui um campo chamado `alunos`. O que o Prisma gera é apenas o nome das operações.

Conectando dados já existentes

Vimos como ligar dados no momento de sua criação. Mas e se quisermos conectar um dado já existente? Há duas situações: criar um dado novo e ligar a um já existente ou conectar dois dados já existentes.

Quando criamos um aluno e indicamos que queríamos criar um novo curso também, indicamos o campo `create`. Para criar um novo aluno e conectá-lo a um curso já existente, basta indicar o campo `connect` e passar o `id` do curso ao qual queremos conectar o nosso novo aluno.

```

mutation{
  createAluno(

```

```

data: {
  nomeCompleto: "João Silva"
  idade: 20
  curso: {
    connect: {
      id: "cjkhiveqgdnu0b58jrg55mho"
    }
  }
}
){
  id
  nomeCompleto
  idade
  curso{
    id
    disciplina
  }
}
}

```

Agora vamos criar um novo aluno, mas ainda não vamos conectá-lo a nenhum curso.

```

mutation{
  createAluno(
    data: {
      nomeCompleto: "Carlos Monteiro"
      idade: 22
    }
  ){
    id
    nomeCompleto
    idade
  }
}

```

Queremos fazer uma alteração nos dados já existentes para poder conectá-los. Para isso, basta usar a operação `updateAluno` que já conhecemos e, no campo `curso`, indicar o `connect` que vimos anteriormente.

```

mutation{
  updateAluno(
    where: {
      id: "cjkjhjrhf6dr220b58qke4mepk"
    }
    data: {
      curso: {
        connect: {
          id: "cjkhiveqgdnu0b58jrg55mho"
        }
      }
    }
  ){
    id
    nomeCompleto
    curso{
      id
      disciplina
    }
  }
}

```

Com isso, podemos conectar dois dados que já existiam mas que não estavam conectados.

Desconectando dados

Para desconectar uma ligação é bem simples. Fazemos o *update* e, em vez de indicar o campo `connect`, indicamos o `disconnect` passando o valor `true`.

```

mutation{
  updateAluno(
    where: {
      id: "cjkjhjrhf6dr220b58qke4mepk"
    }
    data: {
      curso: {
        disconnect: true
      }
    }
  )
}

```



```

    }
  ){
    id
    nomeCompleto
    curso{
      disciplina
    }
  }
}

```

Agora desconectamos do curso o aluno que acabamos de criar.

4.6 Buscando dados

Agora que temos dados em nosso banco de dados, vamos aprender como fazer buscas.

A primeira forma, e mais simples, é aquela em que são retornados todos os dados. Como já vimos, no Prisma basta indicar o nome do nosso esquema no plural como nome da operação. Em seguida, basta indicar os campos que queremos que sejam retornados.

```

{
  alunos{
    nomeCompleto
    idade
    curso{
      disciplina
    }
  }
}

```

Se quisermos que apenas um aluno venha, indicamos o nome do próprio esquema. Para o Prisma saber qual dado deve ser retornado, passamos um identificador único como parâmetro. No nosso caso, indicamos o campo `id` como identificador quando montamos nosso esquema.

Parâmetros assim, com o objetivo de indicar qual dado queremos que seja retornado, são chamados de filtros. No Prisma, os filtros devem ser passados dentro de um campo chamado `where`. No GraphQL puro, como seremos nós que criaremos as regras, poderemos escolher seguir esse mesmo modelo ou simplesmente receber um objeto qualquer no formato que quisermos.

No exemplo a seguir, estamos procurando por um único aluno com o `id` "cjkhiveqcdnuw0b58nfc0zobt".

```
{
  aluno(
    where: {
      id: "cjkhiveqcdnuw0b58nfc0zobt"
    }
  ){
    nomeCompleto
    idade
  }
}
```

4.7 Variáveis

Até o momento nós passamos valores fixos nos parâmetros de nossas operações como, por exemplo, o `id` do aluno que queríamos que fosse retornado.

O problema é que, quando quisermos alterar nossas operações, sejam *queries* ou *mutations*, teremos que alterar esses valores no código que escrevemos. Isso não é bom, principalmente no momento em que formos criar nosso projeto.

O melhor jeito é deixar o nosso código de uma maneira estática, tendo a certeza de que não alteramos algo por acidente. Quanto aos valores dos parâmetros, podemos passá-los de variáveis, permitindo que eles, sim, sejam dinâmicos.

No Playground as variáveis são inseridas no espaço indicado pelo número 1 na imagem a seguir. Caso em sua tela esse campo `QUERY VARIABLES` esteja encostado na parte inferior da tela, basta clicar e arrastar para cima para poder escrever nele.

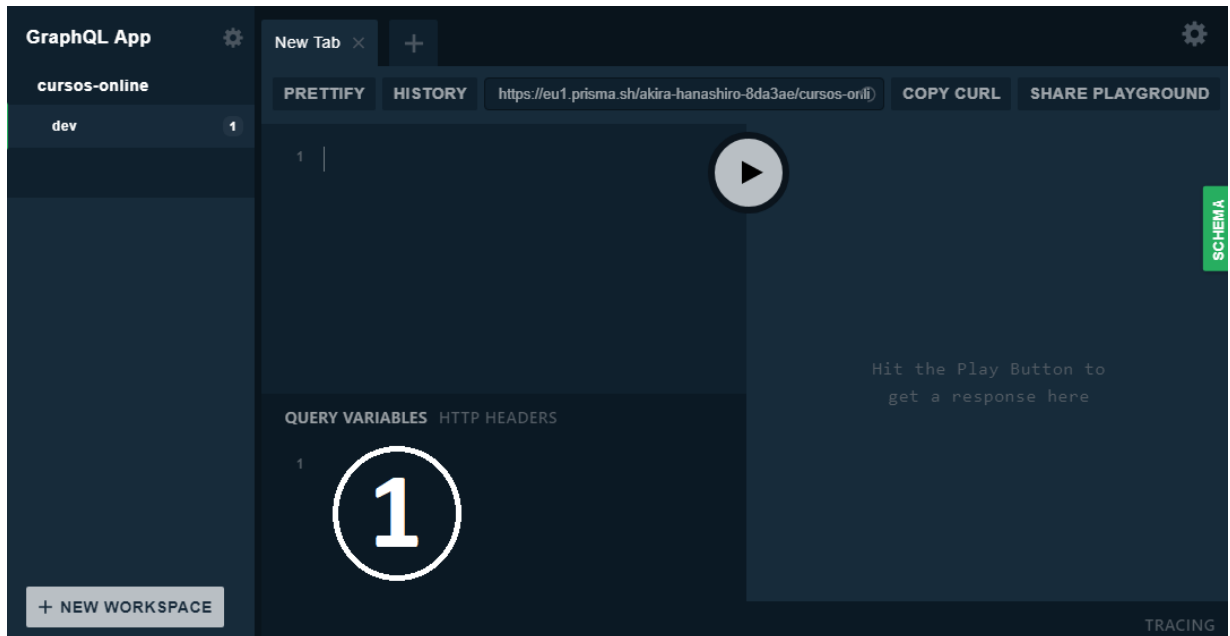


Figura 4.3: Playground — local para escrever variáveis

Anteriormente nós fizemos uma *query* que buscou um aluno pelo seu `id`.

```
{
  aluno(
    where: {
      id: "cjkhiveqcdnuw0b58nfc0zobt"
    }
  ){
    nomeCompleto
    idade
  }
}
```

Para começar a trabalhar com variáveis basta alterarmos três coisas:

1 — Remover valores fixos

O valor fixo do nosso parâmetro será trocado pelo nome de uma variável que nós vamos criar. Variáveis devem ter seu nome iniciado por \$.

Vamos chamar a nossa variável de `$alunoID` .

2 — Declarar variáveis

Devemos declarar o nome da nossa variável no início do comando para o GraphQL saber que aquela é uma variável aceita pela nossa consulta.

Para isso indicamos o nome da variável dentro de parênteses logo após a palavra-chave `query` ou `mutation` . Após o nome da variável, nós indicamos o tipo dela. No nosso caso, o campo `id` é do tipo `ID` .

3 — Passar o valor da variável

Por último, precisamos indicar o valor dessa variável que o nosso comando está recebendo. No Playground nós passamos no campo indicado anteriormente, `QUERY VARIABLES` .

Os valores são passados em formato JSON, e o nome dos campos do JSON devem ser iguais aos nomes das variáveis declaradas, apenas ignorando o \$ no início do nome do campo.

Com essas alterações, nossa nova consulta ficará da seguinte maneira:

```
query ($alunoID: ID){  
  aluno(  
    where: {  
      id: $alunoID  
    }  
  ){  
    nomeCompleto  
    idade  
  }  
}
```

```
}  
}
```

E passaremos o seguinte JSON:

```
{  
  "alunoID": "cjkhiveqcdnuw0b58nfc0zobt"  
}
```

Perceba que agora podemos mudar os valores dos nossos parâmetros apenas alterando o nosso JSON, e o código da nossa consulta ficará intocado. Isso permite separar comandos de dados, o que é ótimo em programas, já que os comandos são estáticos e os dados são dinâmicos.

Foi dito anteriormente que em operações de busca a palavra-chave `query` pode ser omitida. Porém, quando estamos trabalhando com variáveis, é necessário indicar `query` no começo do comando.

4.8 Filtros

Filtros são como nós indicamos a uma consulta o que queremos que seja retornado. Assim, tudo o que não satisfaz o filtro não virá para nós. Um exemplo bem simples é caso queiramos que sejam retornados apenas os alunos com idade maior que 21.

Filtros são passados por parâmetros. Quando criarmos nosso próprio servidor, poderemos passar qualquer tipo de parâmetro que desejarmos e decidir o que esse parâmetro fará em nosso código.

No caso do Prisma, vários filtros já são gerados. Basta usar o campo `where` quando formos fazer uma busca de alunos e indicar o campo pelo qual queremos fazer o filtro.

Se indicarmos que queremos o campo `idade` igual a 21, nada será retornado, pois não cadastramos nenhum aluno com essa idade.

```

{
  alunos(
    where: {
      idade: 21
    }
  ){
    nomeCompleto
    idade
  }
}

```

Esse código fará uma busca por todos os alunos com o campo `idade` que possua exatamente o valor que passamos, 21. Mas como dissemos antes, e se quisermos buscar pelos alunos com mais de 21?

Para casos como esse, o Prisma gera vários filtros. Para procurarmos por `idade` maior que 21, basta passar o campo `idade_gt`, como no código a seguir:

```

{
  alunos(
    where: {
      idade_gt: 21
    }
  ){
    nomeCompleto
    idade
  }
}

```

Esse sufixo `_gt` vem do inglês *greater than* (maior do que). Um outro filtro é o `idade_in`. Para ele nós passamos uma lista de valores, como `[21, 22]`. Então o Prisma retornará todos os alunos que possuam idade igual a 21 ou 22.

Ao escrever o campo `idade` e pressionar as teclas `Ctrl + Espaço`, você verá todos os filtros que o Prisma gerou para nós.

1. `idade_in` — retorna alunos com idades contidas na lista que passarmos;
2. `idade_lt` — retorna alunos com idades menores da que passarmos;
3. `idade_gt` — retorna alunos com idades maiores da que passarmos;
4. `idade_not` — retorna alunos com idades diferentes da que passarmos;
5. `idade_lte` — retorna alunos com idades menores ou iguais à que passarmos;
6. `idade_gte` — retorna alunos com idades maiores ou iguais à que passarmos;
7. `idade_not_in` — retorna alunos com idades diferentes das contidas na lista que passarmos.

Esses filtros foram gerados pelo fato de o campo `idade` ser do tipo `Int`. No caso de um campo do tipo `String`, como é o caso do `nomeCompleto`, precisaremos de mais alguns filtros além desses.

Além dos que vimos anteriormente, o campo `nomeCompleto` também terá os seguintes filtros:

1. `nomeCompleto_ends_with` — retorna alunos cujo nome termina com a *string* que passarmos;
2. `nomeCompleto_starts_with` — retorna alunos cujo nome começa com a *string* que passarmos;
3. `nomeCompleto_not_contains` — retorna alunos cujo nome não possua a *string* que passarmos;
4. `nomeCompleto_not_ends_with` — retorna alunos cujo nome não termina com a *string* que passarmos;
5. `nomeCompleto_not_starts_with` — retorna alunos cujo nome não começa com a *string* que passarmos.

Podemos juntar mais de um filtro e criar condições com os campos `OR` e `AND`. Esses campos recebem uma lista de objetos com filtros que vimos anteriormente.

Podemos buscar, por exemplo, todos os alunos cujo `nomeCompleto` contenha a letra `o` ou alunos cuja `idade` seja maior do que 21.

```
{
  alunos(
    where: {
      OR: [
        {nomeCompleto_contains: "o"},
        {idade_gt: 21}
      ]
    }
  ){
    nomeCompleto
    idade
  }
}
```

Ou também podemos buscar, por exemplo, todos os alunos que o `nomeCompleto` contenha a letra `o` e cuja `idade` seja maior do que 21.

```
{
  alunos(
    where: {
      AND: [
        {nomeCompleto_contains: "o"},
        {idade_gt: 21}
      ]
    }
  ){
    nomeCompleto
    idade
  }
}
```

4.9 Paginação

Quando estamos fazendo buscas em um banco de dados com muitos dados, provavelmente não vamos querer que todos os dados

venham de uma só vez, pois causaria um grande consumo de memória na aplicação tanto do servidor quanto do cliente, sem contar o tempo para transferir os dados.

Por isso, existe a paginação. Paginação nos permite requisitar apenas uma determinada quantidade de dados, como os 10 primeiros resultados ou os 10 últimos resultados.

O Prisma nos fornece o parâmetro `first` . Se quisermos que apenas os dois primeiros resultados sejam retornados, escreveremos o seguinte código:

```
{
  alunos(
    first: 2
  ){
    nomeCompleto
    idade
  }
}
```

Se quisermos que sejam retornados dois elementos, mas que o primeiro seja pulado, retornando na verdade o segundo e terceiro elemento apenas, podemos usar o parâmetro `skip` , para o qual passamos quantos elementos desejamos pular.

```
{
  alunos(
    first: 2,
    skip: 1
  ){
    nomeCompleto
    idade
  }
}
```

Podemos indicar também para o Prisma que queremos os `x` primeiros elementos logo após um determinado elemento. Para isso, nós passamos o `id` do elemento que queremos pular para o parâmetro `after` .

```

{
  alunos(
    first: 2,
    after: "cjkhiveqcdnuw0b58nfc0zobt"
  ){
    nomeCompleto
    idade
  }
}

```

O Prisma também nos fornece o parâmetro `last` . Se quisermos que apenas os dois últimos resultados sejam retornados, escreveremos o seguinte código:

```

{
  alunos(
    last: 2
  ){
    nomeCompleto
    idade
  }
}

```

Podemos indicar também para o Prisma que queremos os `x` últimos elementos logo antes de um determinado elemento. Para isso, nós passamos o `id` do elemento que queremos pular para o parâmetro `before` .

```

{
  alunos(
    last: 2,
    before: "cjkjhjrhf6dr220b58qke4mepk"
  ){
    nomeCompleto
    idade
  }
}

```

`after` só pode ser usado em conjunto com `first` , e `before` só pode ser usado em conjunto com `last` .

4.10 Ordenação

Já que a ideia do GraphQL é pedirmos exatamente o que queremos, claro que não podia faltar uma maneira de pedirmos para nossos resultados virem ordenados de acordo com nossas necessidades.

Para ordená-los, basta utilizar o parâmetro `orderBy` . Ele recebe o nome do campo a ser utilizado para a ordenação. O nome do campo virá seguido por um sufixo `_ASC` , caso queiramos na ordem crescente, ou `_DESC` , caso queiramos na ordem decrescente.

No exemplo a seguir, nós pedimos que os alunos venham ordenados pelo nome em ordem crescente.

```
{
  alunos(
    orderBy: nomeCompleto_ASC
  ){
    nomeCompleto
    idade
  }
}
```

Pedimos que os campos retornados fossem `nomeCompleto` e `idade` . Porém, não somos obrigados a requisitar o campo pelo qual estamos ordenando. Isso significa que, no exemplo anterior, mesmo pedindo para ordenar pelo campo `nomeCompleto` , poderíamos ter pedido apenas o campo `idade` .

Nas vezes em que não passamos nenhum campo como índice de ordenação, o campo `id` é utilizado e os resultados retornam na ordem crescente.

No momento da escrita deste livro, a API do Prisma não permite ainda indicar mais de um campo para ser utilizado como índice de ordenação.

4.11 Renomeando campos

Há momentos em que será necessário renomear o nome de campos. Vamos ver um simples exemplo de uma busca com um filtro para entender o problema:

```
{
  alunos(
    where: {
      idade_gt: 21
    }
  ){
    nomeCompleto
    idade
  }
}
```

Teremos o seguinte resultado:

```
{
  "data": {
    "alunos": [
      {
        "nomeCompleto": "Maria Silva",
        "idade": 26
      },
      {
        "nomeCompleto": "Carlos Monteiro",
        "idade": 22
      }
    ]
  }
}
```

Veja que dentro do campo `data` é retornado primeiro o nome da operação que fizemos, `alunos`. E se quisermos fazer duas operações ao mesmo tempo com filtros diferentes? Dessa vez uma que retorne alunos com `idade` maior que 21 e alunos com `idade` menor do que 21. Podemos fazer isso, pois, afinal, uma das ideias do GraphQL é permitir que solicitemos tudo o que precisamos em uma única requisição. Teríamos o seguinte código:

```
{
  alunos(
    where: {
      idade_gt: 21
    }
  ){
    nomeCompleto
    idade
  }

  alunos(
    where: {
      idade_lt: 21
    }
  ){
    nomeCompleto
    idade
  }
}
```

Isso não vai funcionar, pois estamos chamando a operação `alunos` duas vezes, e o JSON retornado não pode conter dois campos com os mesmos nomes. É neste momento que se faz necessária a renomeação dos campos retornados.

Assim como vimos anteriormente na parte de renomear campos, basta passarmos o nome que queremos, passar dois pontos e, por último, o nome do campo. Podemos fazer exatamente o mesmo com operações.

```
{
  resultado1: alunos(
```

```

        where: {
            idade_gt: 21
        }
    ){
        nomeCompleto
        idade
    }

    resultado2: alunos(
        where: {
            idade_lt: 21
        }
    ){
        nomeCompleto
        idade
    }
}

```

Agora teremos dois grupos em nosso resultado:

```

{
  "data": {
    "resultado1": [
      {
        "nomeCompleto": "Maria Silva",
        "idade": 26
      },
      {
        "nomeCompleto": "Carlos Monteiro",
        "idade": 22
      }
    ],
    "resultado2": [
      {
        "nomeCompleto": "João Silva",
        "idade": 20
      }
    ]
  }
}

```

4.12 Fragmentos

Acabamos de ver como fazer consultas diferentes para o mesmo esquema. Porém, note que tivemos que repetir os campos

`nomeCompleto` e `idade`.

```
{
  resultado1: alunos(
    where: {
      idade_gt: 21
    }
  ){
    nomeCompleto
    idade
  }

  resultado2: alunos(
    where: {
      idade_lt: 21
    }
  ){
    nomeCompleto
    idade
  }
}
```

Neste exemplo não parece ser um grande problema. Porém, em um sistema de verdade, provavelmente teremos vários campos. Será um grande problema caso precisemos manter as consultas com os mesmos campos. Caso você altere em um lugar, corre o risco de esquecer de alterar em outro, sem contar o trabalho de ficar escrevendo campos repetidos.

Por isso, existem os *fragments* (fragmentos). Eles nos permitem declarar um grupo de campos de um determinado esquema. Assim nós só precisamos indicar em nossas consultas quais fragmentos queremos. Caso tenhamos que alterar os campos a serem chamados, alteramos em um lugar só.

Para declarar um fragmento basta utilizar a palavra-chave `fragment` , indicar um nome e dizer de qual esquema estamos criando esse fragmento. Após isso basta indicarmos os campos que queremos entre chaves, como estamos acostumados.

```
fragment meusCampos on Aluno{
  nomeCompleto
  idade
}
```

Para utilizar um fragmento, basta passar o nome do fragmento no lugar do nome dos campos lá em nossa consulta. Para o GraphQL saber que se trata de um fragmento, e não do nome de um campo, precisamos passar ... antes do nome do fragmento.

Nossa consulta anterior ficará da seguinte maneira:

```
{
  resultado1: alunos(
    where: {
      idade_gt: 21
    }
  ){
    ...meusCampos
  }

  resultado2: alunos(
    where: {
      idade_lt: 21
    }
  ){
    ...meusCampos
  }
}
```

Como declaramos que `meusCampos` é um fragmento de `Aluno` , só podemos utilizá-lo dentro de uma consulta de `Aluno` .

4.13 Recebendo notificações

As *subscriptions* do GraphQL nos permitem que sejamos notificados assim que alguma alteração ocorrer com os nossos dados.

Vimos que há três possibilidades: quando um novo objeto é criado, quando um objeto existente é alterado e quando um objeto existente é apagado.

Para começar a receber notificações sobre o que está acontecendo com nossos dados precisamos executar uma operação do tipo `subscription`. Vamos criar uma que daremos o nome de `updatedAlunos`.

Dentro dela, nós precisamos indicar o tipo do qual queremos receber notificações. Vamos colocar `aluno`. Em seguida, precisamos passar o `where` para indicar o tipo de notificação queremos receber.

Como notificações ocorrem quando um dado é alterado, receberemos assim que uma operação `mutation` ocorrer. Podemos indicar mais de um tipo de `mutation`. Como vimos anteriormente, o Prisma cria um campo com sufixo `_in` para indicarmos uma lista de valores que queremos. Então passaremos o campo `mutation_in` e indicaremos em um *Array* os tipos que queremos receber.

Os tipos disponíveis são `CREATED`, `UPDATED` e `DELETED`. Vamos colocar apenas o `UPDATED` para recebermos notificações apenas quando um dado for atualizado.

Então, nós indicamos os campos que queremos receber. Podemos receber um campo chamado `mutation`, que indica a operação que ocorreu. No nosso caso, receberemos apenas o `UPDATED`. Esse campo é importante, pois pode ser que queremos receber notificações também na criação e remoção, e este campo vai nos indicar qual operação ocorreu.

Em seguida, nós devemos passar o campo `node`, e dentro dele indicamos os campos do nosso objeto `Aluno`.

```

subscription updatedAlunos {
  aluno(where: {
    mutation_in: [UPDATED]
  }) {
    mutation
    node {
      id
      nomeCompleto
      idade
    }
  }
}

```

Execute este comando no Playground. Ele apenas começará a esperar, como mostrado no número 1 na figura seguinte.

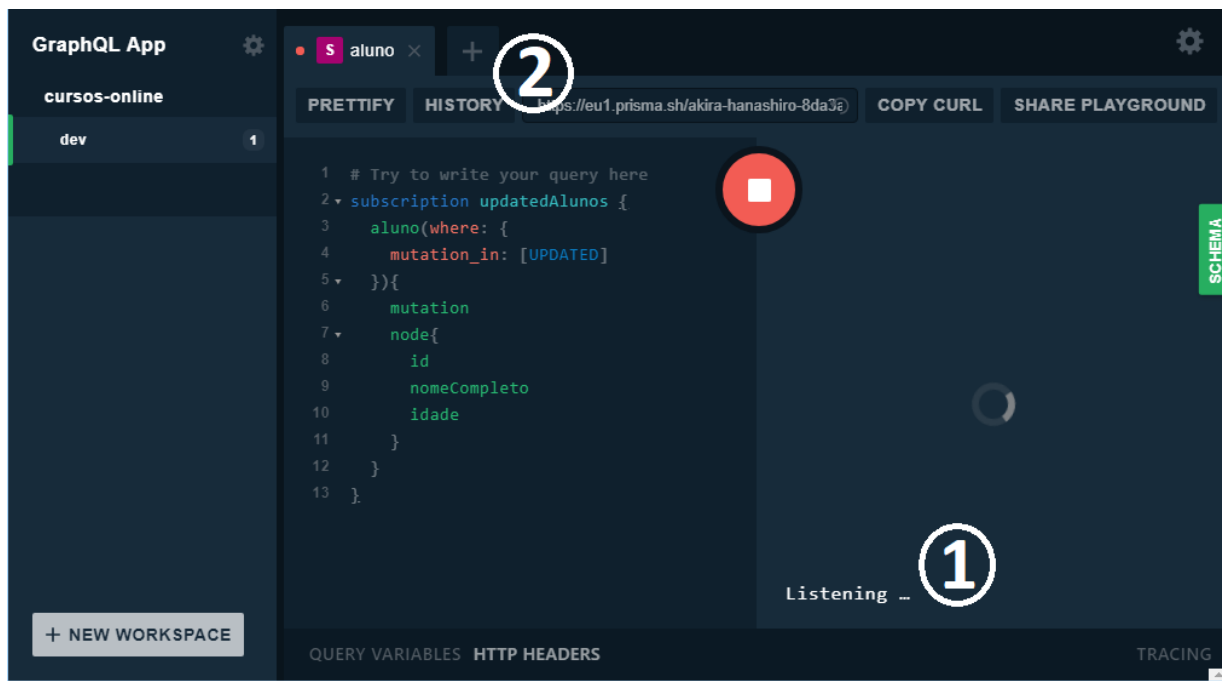


Figura 4.4: Playground — criação de subscription

Agora precisamos deixar esta tela dessa maneira, pois ela está esperando por atualizações em nosso banco de dados. Para podermos executar outros comandos, abra uma nova aba no Playground como indicado no número 2 da imagem.

Vamos simplesmente atualizar algum aluno qualquer, como já fizemos anteriormente. Execute o comando a seguir na nova aba que se abriu no Playground:

```
mutation{
  updateAluno(
    data: {
      idade: 25
    }
    where: {
      id: "cjkjhjrhf6dr220b58qke4mepk"
    }
  ){
    id
    nomeCompleto
    idade
  }
}
```

Ao executar este comando, volte à aba onde executamos a `subscription`. Teremos agora ao lado direito a seguinte resposta:

```
{
  "data": {
    "aluno": {
      "mutation": "UPDATED",
      "node": {
        "id": "cjkjhjrhf6dr220b58qke4mepk",
        "nomeCompleto": "Carlos Monteiro",
        "idade": 25
      }
    }
  }
}
```

Recebemos pelo campo `mutation` o tipo de operação que ocorreu, `UPDATED`, e dentro de `node` recebemos os campos do aluno que teve seus dados modificados. Se você alterar novamente qualquer aluno, receberemos os novos dados.

4.14 Estamos indo muito bem!

Neste capítulo nós pegamos nossos conhecimentos básicos que adquirimos sobre a sintaxe do GraphQL e botamos em prática no Playground do Prisma. Finalmente, pudemos ver nossos comandos em ação ao buscar, criar, atualizar e apagar dados. Além disso, também aprendemos a receber notificações assim que uma operação modifica nossos dados.

A seguir, vamos nos aprofundar mais em alguns elementos que já conhecemos e ver sobre mais alguns elementos novos que fazem parte da estrutura do GraphQL.

CAPÍTULO 5

Conhecendo melhor os *types*

Estamos muito próximos de criar nosso próprio front-end e back-end. Mas antes disso vamos conhecer um pouco mais sobre os tipos de dados presentes no GraphQL. Isso nos permitirá ter um controle melhor da nossa aplicação ao criar nossos *schemas*.

5.1 Types

Como vimos anteriormente, os elementos mais básicos do esquema do GraphQL são os *Types*, também chamados de "tipos". Nós usamos *types* para definir a estrutura dos objetos e seus respectivos campos e tipos.

Vamos revisar rapidamente como os *types* funcionam. Aqui temos um exemplo de um *type* que chamamos de `Aluno`.

```
type Aluno{
  id: ID!
  nomeCompleto: String!
  idade: Int
}
```

Types são declarados com a palavra-chave `type`. Em seguida, abrimos chaves e declaramos em cada linha o nome dos campos que esse objeto terá. Na frente de cada um dos campos nós devemos colocar dois pontos e indicar seu tipo. A seguir veremos melhor sobre tipos de campos.

Acima nós declaramos os tipos `ID`, `String` e `Int`, mas os campos também podem ser do tipo que nós mesmos criamos com a declaração `type`. Então podemos ter um campo que seja do tipo `Aluno`, como no exemplo a seguir:

```
type Aluno{
  id: ID!
  nomeCompleto: String!
  idade: Int
  melhorAmigo: Aluno
}
```

5.2 Scalar Types

Sabemos como declarar *types* para definir a estrutura de objetos e que podemos utilizá-los para tipar campos, como o *type* `Aluno`. Mas para declarar certos campos, como um simples número ou palavra, precisaremos de dados concretos em vez de outros objetos. Um exemplo de dado concreto é o campo `nomeCompleto` de `Aluno`, que declaramos como `String`. Esses tipos que já vêm com o GraphQL são chamados de *Scalar Types*. São cinco no total:

Int

Indica um número inteiro, ou seja, sem casas decimais. Esse campo aceita números positivos e negativos. Exemplo: a idade de uma pessoa.

Float

Indica números flutuantes, ou seja, com casas decimais. Aceita números positivos e negativos. Exemplo: o preço de um produto.

String

Indica uma sequência de caracteres, sejam letras, números ou símbolos. Exemplo: nome ou e-mail de uma pessoa.

Boolean

Indica um dos possíveis valores: `true` (verdadeiro) ou `false` (falso).
Exemplo: se um aluno está ou não aprovado em uma matéria.

ID

É um tipo de dado do GraphQL que indica um identificador único para cada objeto existente em nosso banco de dados. Exemplo: chaves primárias dos bancos de dados.

5.3 Enumeration Types

Enumeration Types, também conhecidos como *Enums*, são um tipo especial de *Scalar Types*. A diferença é que os *Enums* são restritos a um grupo de valores que nós definimos.

Um bom exemplo é quando vimos sobre *subscriptions*. Lembre-se da resposta que recebemos quando fizemos uma alteração:

```
{
  "data": {
    "aluno": {
      "mutation": "UPDATED",
      "node": {
        "id": "cjkjhjrhf6dr220b58qke4mepk",
        "nomeCompleto": "Carlos Monteiro",
        "idade": 25
      }
    }
  }
}
```

Veja o campo `mutation`. Ele recebeu o valor `UPDATED`. Vimos que podemos receber três tipos de valores nesse campo: `CREATED`, `UPDATED` e `DELETED`. Então os *Enums* são uma forma de restringirmos os valores que um campo pode ter.

O Prisma já cria esse *Enum* para nós, mas, ao criarmos nosso próprio servidor GraphQL, nós teremos que declarar este *Enum* também.

Seguindo este exemplo, poderíamos criar um *Enum* com o nome `MutationType` e indicar os valores possíveis. Teríamos o seguinte código para sua declaração:

```
enum MutationType{  
  CREATED  
  UPDATED  
  DELETED  
}
```

É uma boa prática manter os valores de *Enums* com letras maiúsculas, mas isso não é obrigatório.

Agora, se declararmos algum campo com o tipo `MutationType`, este campo só poderá ter um dos três valores que declaramos.

5.4 Input Types

Quando estávamos trabalhando com *mutations* nós passamos valores simples como strings e números pelos parâmetros, como podemos ver no exemplo a seguir:

```
mutation{  
  createAluno(  
    nomeCompleto: "Maria Silva",  
    idade: 25  
  ){  
    id  
  }  
}
```

Mas nós também podemos passar objetos mais complexos como parâmetros. `Aluno` possui os campos `id`, `nomeCompleto`, `idade` e `curso`.

Se quisermos indicar que todo `Aluno` a ser criado deverá receber um objeto com os campos `nomeCompleto` e `idade`, podemos criar um `type` com esses campos e dar um nome como `AlunoParametros`.

O problema é que todo `type` declarado passa a fazer parte do *schema*. Isso sujaria nosso banco, pois teríamos então `Aluno` e `AlunoParametros`. Quando queremos fazer uma busca de alunos, vamos em `Aluno`, nunca fazemos uma busca de `AlunoParametros`, já que este tipo teria sido declarado apenas para termos uma estrutura de dados.

Quando queremos declarar uma estrutura que seja apenas para uso de entrada de dados, usamos os *inputs*.

Inputs se parecem com *types*, mas nós os declaramos com a palavra-chave `input`. É também uma boa prática colocar **Input** como sufixo no nome de *inputs*.

```
input AlunoInput{
  nomeCompleto: String!
  idade: Int
}
```

Note que, como estamos apenas declarando a estrutura de um objeto que poderá ser passado como parâmetro, e não um objeto representando algo retornado do banco de dados, não precisamos do campo `id`.

Por essa mesma razão você não pode fazer *queries*, *mutations* e *subscriptions* de *inputs*, pois eles apenas representam uma estrutura de dados que nós vamos enviar ao servidor, e não algo que está sendo salvo no banco.

Teríamos algo no seguinte formato:

```
mutation CreateAluno($dados: AlunoInput!) {
  createAluno(dados: $dados) {
    id
    nomeCompleto
  }
}
```

```
}  
}
```

Outro cenário em que os *inputs* se tornam úteis é quando precisamos passar dados por *mutations* para o nosso servidor mas não queremos que esses dados estejam disponíveis em outras operações.

Imagine que desejamos fazer login em um sistema. Poderíamos criar o seguinte *input*:

```
input LoginInput{  
  email: String!  
  senha: String!  
}
```

Por razões de segurança, não vamos querer que alguém crie uma *query* que retorne os e-mails e senhas dos nossos usuários. Criar um *input* vai possibilitar que trabalhemos com essa estrutura de dado mas sem permitir que *queries* sejam feitas para que estes dados sejam retornados.

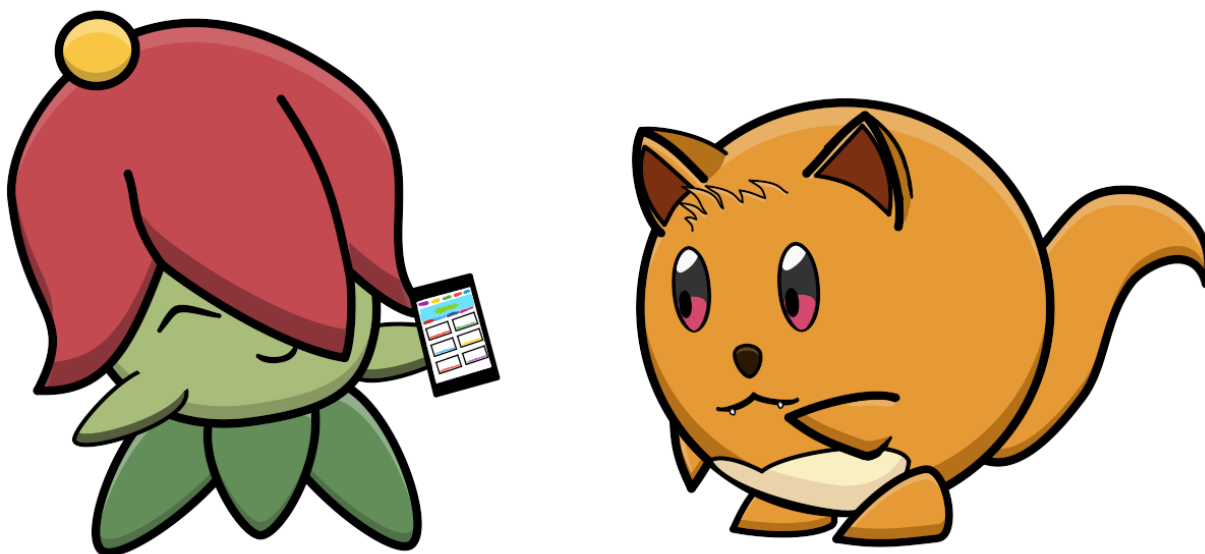
Veremos melhor sobre o uso de *inputs* no momento em que criarmos nosso próprio serviço com GraphQL no back-end.

5.5 Acabou a brincadeira

Anteriormente aprendemos a brincar no Playground do Prisma e agora aprofundamos um pouco mais nossos conhecimentos sobre os tipos de dados do GraphQL. Pudemos entender melhor os *types* e conhecemos os *inputs*. Vimos também como criar *Enums*, muito útil quando queremos limitar os possíveis valores de um campo.

Agora que já brincamos bastante vamos finalmente começar a produzir código!

Parte 3 — Criando uma aplicação



CAPÍTULO 6 Desenvolvimento front-end

Agora que conhecemos bem o GraphQL e temos o Prisma como servidor, vamos criar um simples programa que vai consumir este serviço.

Para focarmos no funcionamento do GraphQL, utilizaremos JavaScript puro, sem adição de nenhuma biblioteca ou framework.

6.1 Conhecendo o nosso projeto

Vamos montar uma aplicação bem simples para consumir nosso serviço GraphQL fornecido pelo Prisma. Ela exibirá uma lista com os

alunos cadastrados. Poderemos inserir novos alunos e apagar os já presentes na lista.

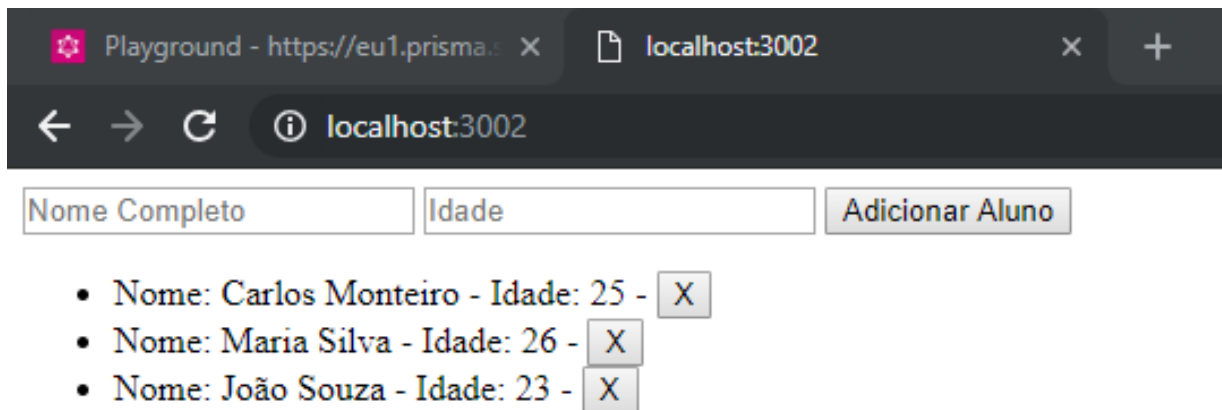


Figura 6.1: Front-end — aplicação

Para focarmos nossos estudos no GraphQL, o layout da aplicação será simples. Não vamos utilizar CSS, mas você pode ficar livre e estilizar como quiser.

O código completo deste projeto pode ser baixado pelo GitHub, em <https://github.com/hanashiro/cdc-livro-graphql/>.

Ele também será apresentado por completo no final deste capítulo.

6.2 O nosso HTML

Vamos iniciar nosso projeto. Para deixar tudo mais bem organizado, crie uma pasta com o nome `front`. Dentro desta pasta, vamos criar um arquivo chamado `index.html`, onde teremos a estrutura da nossa aplicação, e um arquivo chamado `scripts.js`, onde programaremos a conexão com o GraphQL.

Vamos iniciar nosso `index.html` com a estrutura básica de um documento HTML. Também já vamos incluir nosso arquivo `scripts.js` no nosso documento usando a tag `<script>`.

```
<!DOCTYPE html>
<html>
<body>

<script src="./scripts.js"></script>
</body>
</html>
```

A nossa estrutura será escrita antes da tag `<script>` .

Agora vamos criar nosso formulário. Como vimos anteriormente, na apresentação do nosso projeto, teremos um campo para colocar o nome do aluno e outro para a idade.

Primeiro criamos o formulário com a tag `<form>` e damos um nome ao formulário usando o atributo `name` .

```
<form name="novoAluno">

</form>
```

Dentro da tag `<form>` , vamos colocar os campos para podermos escrever o nome e idade. Para isso, utilizaremos a tag `<input>` . Também precisamos passar um nome para cada campo usando o atributo `name` .

O campo de nome será do tipo texto, então passaremos `text` para o atributo `type` da tag. O campo de idade será do tipo número, então passaremos `number` para o atributo `type` da tag.

Para o usuário saber o que escrever em cada campo, podemos passar um texto que será exibido quando o campo estiver vazio. Fazemos isso por meio do atributo `placeholder` .

```
<form name="novoAluno">
  <input type="text" name="nomeCompleto" placeholder="Nome Completo" >
  <input type="number" name="idade" placeholder="Idade">
</form>
```

Para finalizar nosso formulário, vamos colocar um botão escrito "Adicionar Aluno" após os campos que acabamos de criar, com a tag `button`. Para indicar que ele é responsável por enviar os dados do formulário, passamos no botão o atributo `type` com o valor `submit`.

```
<form name="novoAluno">
  <input type="text" name="nomeCompleto" placeholder="Nome Completo" >
  <input type="number" name="idade" placeholder="Idade">
  <button type="submit" >Adicionar Aluno</button>
</form>
```

Teremos o resultado da imagem seguinte:

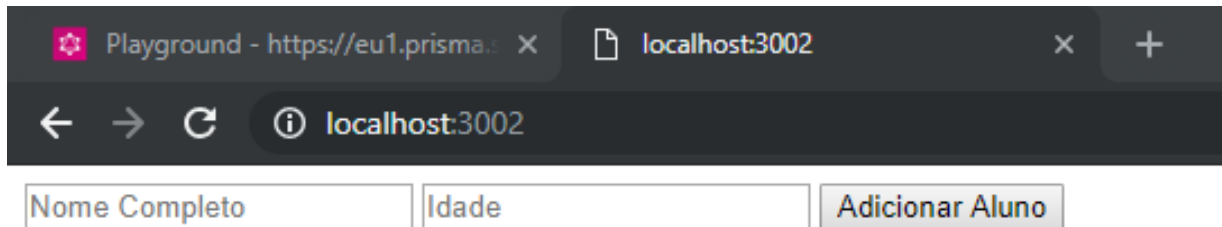


Figura 6.2: Front-end — formulário

6.3 Abrindo nossa aplicação

Para certas operações que faremos, como requisições ao Prisma, precisaremos que nossa aplicação seja fornecida por um servidor. Podemos subir facilmente um servidor local em nossa máquina utilizando o Node.js.

Vamos utilizar o `tw-dev-server` para isso. Esta ferramenta rápida e leve nos permite subir um servidor de arquivos estáticos, como HTML e JavaScript.

O `tw-dev-server` também recebe requisições para simular um servidor com banco de dados, sendo muito útil quando estamos prototipando um sistema ou estudando.

Para saber mais sobre esta ferramenta, acesse:

<https://github.com/treinaweb/tw-dev-server/>.

O `tw-dev-server` é instalado utilizando o `npm`. Abra o seu terminal e execute o comando `npm i -g @treinaweb/tw-dev-server`.

Para utilizá-lo, basta abrir o terminal dentro da pasta com os arquivos do seu projeto e executar `tw-dev-server`.

```

$ tw-dev-server

      ,*****.
    #,./(((((((((((((((((((((((*(
    %%#,.          ./((*(
    %%%/           /((*(
    %%%/           /((*(
    %%%%%%%%%%%%(./  /((*(
    %%%%(.....*%%%(  /((*(
    %%%#/////.*%%%(  /((*(
    %%%(, , , , *%%%(  /((*(
    #%%%,.      ..#%%%(  /((*(
    .(%%%%%%%%%%%%#./  /((*(
                          /((*(
                          /((*(
    ((((((((((((((((((((((((((*(
    ((((((((((((((((((((((((((*(

  _ _ _ _ _  _ _ _ _ _  _ _ _ _ _  _ _ _ _ _  _ _ _ _ _
 | | | | |  | | | | |  | | | | |  | | | | |  | | | | |
 | | | | |  | | | | |  | | | | |  | | | | |  | | | | |
 | | | | |  | | | | |  | | | | |  | | | | |  | | | | |
 | | | | |  | | | | |  | | | | |  | | | | |  | | | | |

      https://treinaweb.com.br
Serving E:\Users\Felipe\Desktop\livro\livro-codigos\front/
Server running on:
http://localhost:3002
http://192.168.1.106:3002
Hit CTRL-C to stop the server

```

Figura 6.3: tw-dev-server

Agora você poderá acessar sua aplicação pelo navegador no endereço <http://localhost:3002/>.

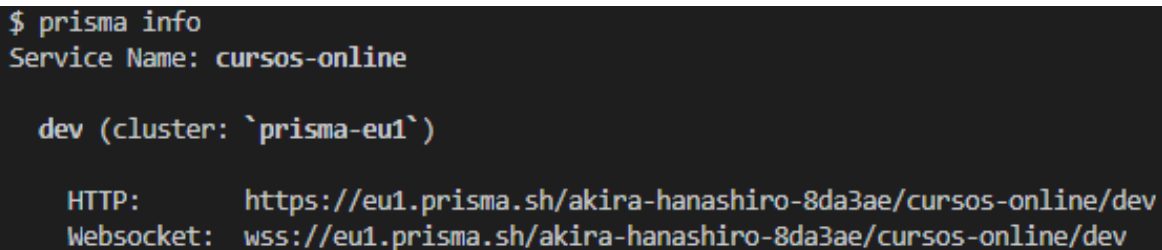
6.4 Conexão com o serviço do GraphQL

Para iniciar uma conexão com o nosso servidor GraphQL fornecido pelo Prisma, vamos utilizar JavaScript.

Abra o arquivo `scripts.js` que criamos anteriormente. Vamos deixar tudo o que for parte da conexão com o nosso servidor GraphQL dentro de um objeto que vamos chamar de `GraphQL` :

```
const GraphQL = {  
  
}
```

Primeiramente, para consumir o serviço fornecido pelo Prisma, precisamos acessar um endereço, que é mostrado quando iniciamos um projeto. Para que ele seja exibido novamente, vá ao diretório do nosso projeto Prisma, abra o terminal e execute o comando `prisma info` . O endereço gerado para o seu projeto será exibido no terminal.

A terminal window with a dark background showing the output of the 'prisma info' command. The text is as follows:

```
$ prisma info  
Service Name: cursos-online  
  
dev (cluster: `prisma-eu1`)  
  
HTTP:      https://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev  
Websocket: wss://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev
```

Figura 6.4: Prisma — Informações

Coloque o endereço HTTP que aparecer na sua tela dentro do nosso objeto `GraphQL` . Vamos inseri-lo em um campo que vamos chamar de `endpoint` .

```
const GraphQL = {  
  endpoint: 'https://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev'  
}
```

Por último, vamos usar a função `fetch` do JavaScript, que serve para criar requisições, para podermos criar uma função que fará as *queries* para o Prisma e retornará sua resposta. Chamaremos esta função de `exec()`, que receberá dois parâmetros: `query` e `variaveis`. Criaremos esta função dentro do objeto `GraphQL`.

A função `fetch()` recebe como primeiro parâmetro o endereço a ser acessado e, como segundo, um objeto de configuração.

O objeto de configuração de requisições para um servidor com GraphQL deve ter as seguintes propriedades:

- **method**: tipo da requisição (`GET`, `POST`, `PUT`, `DELETE` etc.). Requisições para o GraphQL devem ser do tipo `POST`.
- **headers**: cabeçalho da requisição, onde indicamos algumas informações complementares sobre a requisição. Aqui precisamos indicar que o conteúdo transferido é do tipo JSON. Fazemos isso indicando o campo `Content-Type` com o valor `application/json`.
- **body**: objeto enviado ao servidor. Requisições para o GraphQL devem ter um campo `query` e um campo `variables`.

O objeto passado para a requisição deve ser convertido em String. Para isso usamos a função `JSON.stringify()`.

O `fetch()` receberá uma resposta do servidor que precisaremos converter para JSON. Pegamos essa resposta com `then()` e passamos uma função que recebe a resposta. O objeto recebido já possui um método chamado `json()`, que fará a conversão para nós.

```
const GraphQL = {
  endpoint: 'https://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev',

  exec: function(query, variaveis){
    return fetch(GraphQL.endpoint, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      }
    })
    .then(response => response.json())
  }
}
```

```

    },
    body: JSON.stringify({ query: query, variables: variaveis }),
  })
  .then(resposta => resposta.json())
}
}

```

6.5 Buscando dados

Agora já criamos um objeto com uma função que nos permite enviar requisições ao nosso servidor GraphQL fornecido pelo Prisma. Anteriormente, nós já havíamos cadastrado alguns alunos. Vamos permitir que nossa aplicação os busque para depois podermos listá-los.

Tudo o que estiver relacionado aos dados dos alunos (buscar, criar, apagar e alterar) será posto dentro de um objeto chamado `Aluno`. Assim teremos nossas funções organizadas e centralizadas em um único objeto.

Primeiro, vamos criar ao final do arquivo `scripts.js` um objeto chamado `Aluno`. Dentro dele, nós criaremos uma propriedade chamada `lista`, que iniciará com um *Array* vazio como valor. É nesta lista onde armazenaremos os alunos que o nosso servidor GraphQL retornará.

```

const Aluno = {
  lista: []
}

```

Ainda dentro deste objeto nós também vamos criar uma função com o nome `buscar`. Ela vai executar a nossa função `GraphQL.exec()`, passando uma *query* de busca de todos os alunos presentes lá no Prisma.

```

const Aluno = {
  lista: [],
  buscar: function(){
    const query = `
      query{
        alunos{
          id
          nomeCompleto
          idade
        }
      }
    `;
    return GraphQL.exec(query);
  }
}

```

6.6 Exibindo lista de alunos

Temos uma função que busca todos os alunos do nosso servidor GraphQL. Agora nós precisamos de um jeito de executar esta função, pegar os dados retornados e exibi-los ao usuário. Para isso, criaremos uma lista.

Vamos voltar ao arquivo `index.html` e adicionar um elemento de lista logo abaixo do formulário. Vamos colocar um `id` na lista para podermos acessá-la pelo JavaScript.

Como os código começarão a ficar mais compridos daqui para a frente, vou omitir parte do código com reticências (...). Lembre-se de que o código completo será mostrado no final do capítulo e também está disponível pelo endereço <https://github.com/hanashiro/cdc-livro-graphql/>.

```

...
<button type="submit" >Adicionar Aluno</button>

```

```
</form>
<ul id="listaAlunos" ></ul>
```

Para podermos acessar este elemento de lista do HTML no nosso JavaScript, vamos guardá-lo em uma variável chamada `listaAlunos` no começo do arquivo `scripts.js`.

```
const listaAlunos = document.querySelector('#listaAlunos');
```

```
const GraphQL = {
  ...
```

Agora, para exibir nossos dados nessa lista do HTML, vamos criar um objeto que terá funções relacionadas à manipulação da nossa tela. Vamos chamar este objeto de `Template`. Dentro dele, vamos criar uma função chamada `listarAluno`. Esta função vai pegar a lista de alunos presente no objeto `Aluno` e utilizá-la para gerar os itens que serão exibidos.

Começamos a função declarando uma variável chamada `html`, que armazenará o HTML dos itens da lista que será exibida. Em seguida, pegamos o *Array* de alunos presente no objeto `Aluno` e executamos o `forEach()`. Trata-se de um método que nos permite executar uma função para cada item do *Array*. Dentro da função, recebemos cada um dos alunos e vamos criar uma *string* com o HTML referente ao item da lista, com o nome e idade do aluno. Cada uma dessas *strings* será armazenada na nossa variável `html`.

Após executar essa função para todos os alunos do *Array*, vamos pegar nosso elemento de lista que guardamos na variável `listaAlunos` e injetar através da propriedade `innerHTML` o HTML que criamos.

```
const Template = {
  listarAluno: function(){
    let html = ''
    Aluno.lista.forEach((aluno) => {
      html += `<li>Nome: ${aluno.nomeCompleto} - Idade:
${aluno.idade}`
    })
```

```

        listaAlunos.innerHTML = html;
    }
}

```

Precisamos de uma função que chame a função de busca de alunos, armazene os dados na lista e chame a função `listarAluno()` para exibirmos os dados na tela. Dentro de `Template` vamos criar uma função que chamaremos de `iniciar()`.

A função `iniciar()` chamará a função `Aluno.buscar()`. Ao executá-la, receberemos um objeto com o campo `data`, e dentro dele teremos o campo `alunos`, como vimos anteriormente nos retornos das buscas no Prisma Playground. Vamos guardar este retorno em `Aluno.lista`.

Após armazenar os dados retornados do servidor na lista, basta chamarmos `Template.listarAluno()`.

```

const Template = {
  iniciar: function(){
    Aluno.buscar()
      .then(({data: {alunos}}) => {
        Aluno.lista = alunos;
        Template.listarAluno();
      })
  },
  listarAluno: function(){
    ...
  }
}

```

Sempre que quisermos que os dados do servidor sejam exibidos, basta chamar `Template.iniciar()`. Para que nossa aplicação já inicie com os dados sendo apresentados, adicione a chamada desta função na última linha do arquivo `scripts.js`.

```

...
    listaAlunos.innerHTML = html;
  }
}

```

```

Template.iniciar();

```

6.7 Enviando dados

Agora que já sabemos listar dados, vamos ver como criá-los. Como estamos falando de manipulação de dados, vamos voltar ao nosso objeto `Aluno`. Dentro dele vamos criar uma função chamada `criar()`, que receberá um objeto chamado `novoAluno`, contendo o nome completo e a idade.

Chamaremos a função `GraphQL.exec()`, passando o código de `mutation` para a criação de aluno que aprendemos anteriormente, e passando `novoAluno` como variável com os dados da nossa requisição.

```
const Aluno = {
  ...
  criar: function(novoAluno){
    const query = `
      mutation ($nomeCompleto: String!, $idade: Int!){
        createAluno(data: {
          nomeCompleto: $nomeCompleto
          idade: $idade
        }){
          id
          nomeCompleto
          idade
        }
      }
    `;
    return GraphQL.exec(query, novoAluno);
  }
}
```

6.8 Criando novos alunos

Acabamos de preparar uma função que envia dados ao servidor para criar um novo aluno. Vamos agora fazer com que possamos pegar dados de um novo aluno a partir do nosso formulário e inseri-

lo na nossa lista. Como é algo que vai manipular nosso HTML, vamos voltar ao nosso objeto `Template` .

Vamos criar uma função chamada `inserirAlunoLista()` , que receberá um objeto `novoAluno` . Dentro dela, nós simplesmente vamos inserir o novo aluno no final da lista `Aluno.lista` e chamar `Template.listarAluno()` , que cria a lista e a exibe, para atualizar a nossa tela.

```
const Template = {  
  ...  
  inserirAlunoLista: function(novoAluno){  
    Aluno.lista.push(novoAluno);  
    Template.listarAluno();  
  }  
}
```

Precisamos pegar o formulário com os dados do novo aluno, enviá-los ao servidor e atualizar nossa lista. Ainda dentro de `Template` , vamos criar uma função `criarAluno()` , que será chamada pelo formulário.

O comportamento padrão de um formulário HTML submetido é nos enviar para outra página, mas precisamos evitar isto, pois nossa aplicação possui apenas uma única tela. Então temos que executar `event.preventDefault()` .

Em seguida, vamos acessar nosso formulário e armazená-lo em uma variável `formulario` para podermos ter acesso aos dados inseridos pelo usuário. Isso é feito pelo objeto `document.forms` , passando o nome que demos a ele, `novoAluno` .

Vamos criar um objeto que terá dois campos: `nomeCompleto` e `idade` . Para pegar estes valores dos campos do formulário, basta acessá-lo pela nossa variável `formulario` , acessar o nome do campo que definimos no HTML pelo atributo `name` e a propriedade `value` .

Lembre-se de que definimos no *schema* que o campo `idade` é do tipo *number*. Acontece que os campos do nosso formulário retornam

strings. Portanto, precisamos converter a idade para número com a função `parseInt()` .

Para impedir que um aluno com dados em branco seja enviado ao servidor, vamos primeiro fazer uma verificação com `if` para ter certeza de que ambos os campos estão preenchidos. Se tudo estiver certo, vamos apagar os valores dos campos do formulário para que depois possamos já criar outro aluno.

Basta então chamar `Aluno.criar()` e passar o objeto `novoAluno` com os dados que pegamos do nosso formulário antes de apagá-lo.

Receberemos uma resposta com um campo `data` , que terá o objeto `createAluno` com os dados do novo aluno, inclusive o ID gerado pelo banco. Vale lembrar que os campos retornados aqui foram definidos por nós mesmos quando criamos a função `Aluno.criar()` .

Com o novo aluno em mãos, basta passá-lo para `Template.inserirAlunoLista()` para que ele seja adicionado ao *Array* e exibido na tela.

```
const Template = {
  criarAluno: function(){
    event.preventDefault();
    const formulario = document.forms.novoAluno,
      novoAluno = {
        nomeCompleto: formulario.nomeCompleto.value,
        idade: parseInt(formulario.idade.value)
      };
    if(novoAluno.nomeCompleto && novoAluno.idade){
      formulario.nomeCompleto.value = '';
      formulario.idade.value = '';
      Aluno.criar(novoAluno)
        .then(({data: {createAluno}}) => {
          Template.inserirAlunoLista(createAluno);
        })
    }
  },
  inserirAlunoLista: function(novoAluno){
```

```
    ...  
  }
```

Para que esta função seja executada, vamos voltar ao nosso `index.html` e chamar esta função no evento de envio do formulário (`onsubmit`).

```
<form name="novoAluno" onsubmit="Template.criarAluno()">  
  <input type="text" name="nomeCompleto" placeholder="Nome Completo" >  
  <input type="number" name="idade" placeholder="Idade">  
  <button type="submit" >Adicionar Aluno</button>  
</form>
```

6.9 Apagando dados

Já podemos criar alunos, então precisamos poder apagar também. Por ser manipulação de dados, vamos voltar ao nosso objeto `Aluno`. Dentro dele vamos criar uma função chamada `apagar()`, que vai receber o ID do aluno que queremos remover.

Chamaremos a função `GraphQL.exec()`, passando o código de `mutation` para a remoção de aluno que aprendemos anteriormente e a variável `id`.

```
const Aluno = {  
  ...  
  apagar: function(id){  
    const query = `  
    mutation ($id: ID!){  
      deleteAluno(  
        where: {  
          id: $id  
        }  
      ){  
        id  
      }  
    }  
  }
```

```

    `;
    return GraphQL.exec(query, {id});
  }
}

```

Removendo alunos da lista

Preparamos a função que indica ao Prisma que queremos apagar um aluno. Agora vamos integrar essa função com a nossa tela, possibilitando que o usuário indique qual aluno ele que apagar. Como é algo que vai manipular nosso HTML, vamos voltar ao nosso objeto `Template`.

Vamos criar uma função chamada `removeAlunoLista()`, que receberá uma variável `id`. Dentro desta função nós simplesmente vamos procurar a posição do aluno na lista. Caso o aluno exista (tenha índice maior do que 0), vamos removê-lo de `Aluno.lista` e chamar `Template.listarAluno()`, que cria a lista e a exibe, para atualizar a nossa tela.

```

const Template = {
  ...
  removeAlunoLista: function(id){
    const alunoIndice = Aluno.lista.findIndex(aluno => aluno.id ===
id);
    if(alunoIndice >= 0){
      Aluno.lista.splice(alunoIndice, 1);
      Template.listarAluno();
    }
  }
}

```

Então já temos o `Aluno.apagar()`, que remove o aluno do banco, e `Template.removeAlunoLista()`, que remove o aluno da nossa tela.

Para não precisarmos ficar chamando uma função de cada vez sempre que quisermos apagar um aluno, vamos criar uma função que chame essas duas funções. Vamos chamá-la de `apagarAluno()` e

a criaremos dentro de `Template` . Ela precisará receber o `id` do aluno a ser apagado.

```
const Template = {
  ...
  apagarAluno: function(id){
    Aluno.apagar(id)
      .then(() => {
        Template.removerAlunoLista(id);
      })
  },
  removerAlunoLista: function(novoAluno){
    ...
  }
}
```

Para que possamos indicar qual aluno será apagado, vamos adicionar um botão ao lado de cada aluno na lista.

Vamos voltar à nossa função `Template.listarAluno()` e alterar o HTML criado e armazenado na variável `html` . Adicionaremos um botão que, ao ser clicado, chamará a função `Template.apagarAluno()` com o `id` do aluno a ser apagado.

```
const Template = {
  ...
  listarAluno: function(){
    let html = ''
    Aluno.lista.forEach((aluno) => {
      html += `<li>Nome: ${aluno.nomeCompleto} - Idade:
${aluno.idade} - <button onclick="Template.apagarAluno('${aluno.id}')"
>X</button></li>`
    })
    listaAlunos.innerHTML = html;
  }
  ...
}
```

6.10 Recebendo dados com Subscriptions

Como vimos anteriormente, podemos também receber dados do servidor quando houver alguma alteração neles. Para isso, é possível nos inscrevermos para alguma alteração, por meio de *subscriptions*.

Subscriptions do GraphQL funcionam pelo protocolo `ws` (WebSocket). Vamos pegar o endereço que precisamos para iniciar a conexão com WebSocket.

Para que ele seja exibido novamente, basta seguir os passos que fizemos para rever o endereço HTTP. Vá ao diretório do nosso projeto Prisma, abra o terminal e execute o comando `prisma info`. O endereço gerado para o seu projeto será exibido no terminal.

```
$ prisma info
Service Name: cursos-online

dev (cluster: `prisma-eu1`)

HTTP:      https://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev
Websocket: wss://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev
```

Figura 6.5: Prisma — Informações

Pegue o endereço WebSocket que aparecer na sua tela. Vamos passá-lo para o método construtor de uma nova conexão WebSocket, que é criada com `new WebSocket()`. Primeiro passamos o endereço e depois passamos o nome do subprotocolo que o Prisma exige, que deve ser `graphql-subscriptions`.

Armazenaremos essa conexão dentro de um campo chamado `wsConnection` dentro do nosso objeto `GraphQL`.

```
const GraphQL = {
  endpoint: 'https://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev',
  wsConnection: new WebSocket('wss://eu1.prisma.sh/akira-hanashiro-8da3ae/cursos-online/dev', 'graphql-subscriptions'),
```

```
...  
}
```

Vamos configurar o que deve ser feito quando esta conexão iniciar. Ainda no objeto `GraphQL`, vamos criar uma função chamada `iniciarWs()`. Como teremos que executar outras tarefas após a conexão ser feita, e criar conexões é algo assíncrono, vamos começar a `iniciarWs()` retornando uma `Promise` para podermos saber quando a conexão estiver estabelecida.

```
const GraphQL = {  
  ...  
  iniciarWS: function(){  
    return new Promise(resolve => {  
  
    })  
  }  
}
```

Dentro da `Promise` que criamos vamos indicar o que fazer quando a conexão `WebSocket` for aberta. Para isso, passamos uma função para o evento `onopen` do objeto que criamos, `wsConnection`. Nós enviaremos um objeto chamado `mensagem` com uma propriedade `type` de valor `init`, pelo método `send()` do objeto `wsConnection`. Ele precisa ser convertido para *String* antes de ser enviado.

Com isso nós indicamos ao `Prisma` que queremos iniciar uma conexão. Ao final disso executamos o `resolve()` da `Promise`, indicando que nossa ação acabou.

```
...  
iniciarWS: function(){  
  return new Promise(resolve => {  
    GraphQL.wsConnection.onopen = function(){  
      const mensagem = {  
        type: 'init'  
      };  
      GraphQL.wsConnection.send(JSON.stringify(mensagem));  
  
      resolve();  
    }  
  });  
}
```

```
    }  
  })  
}
```

E como tratamos as notificações que o servidor nos enviar?
Fazemos isso passando uma função para o evento `onmessage` do objeto `wsConnection` .

Na linha antes de executar `resolve()` , vamos passar uma função ao evento `onmessage` do objeto `wsConnection` . Essa função recebe um parâmetro `event` , o qual possui informações sobre o evento de mensagem que estamos recebendo do Prisma. O objeto `event` possui um campo `data` , o qual vamos converter para JSON e guardar dentro de uma variável `resposta` . Esse será o retorno do serviço do GraphQL.

Esse retorno possui alguns tipos que nos permitem saber sobre o que está acontecendo na conexão, mas o que nos interessa será o tipo `subscription_data` , pois ele contém os dados relacionados à nossa inscrição. Fazemos uma verificação com `if` para saber se o dado é do tipo `subscription_data` .

Além do `subscription_data` , podemos ter outros tipos de respostas vindas do servidor do Prisma, como:

- `init_success` : a conexão com o servidor foi realizada com sucesso
- `init_fail` : a conexão com o servidor falhou
- `subscription_success` : a sua inscrição foi feita com sucesso
- `subscription_fail` : a sua inscrição falhou

Você pode utilizar cada uma dessas respostas do servidor para realizar alguma ação na sua aplicação, por exemplo, pedir para o usuário tentar novamente mais tarde caso o servidor esteja indisponível no momento.

O objeto `resposta` terá um campo chamado `payload` e dentro dele terá um objeto `data`. Como estamos trabalhando com a inscrição de `Aluno`, virá um campo `aluno` dentro dele. Lembre-se de que, quando nos inscrevemos, recebemos um campo chamado `mutation` que indica o tipo da operação que foi realizada no servidor (`CREATED`, `DELETED`, `UPDATED`). Vamos verificar qual o tipo de operação realizada.

Se o tipo for `CREATED`, receberemos um novo aluno dentro do campo `node`. Então, para atualizar nossa lista, basta utilizarmos nossa função `Template.inserirAlunoLista()` passando o aluno recebido. Se o tipo for `DELETED`, os dados estarão dentro do atributo `previousValues`. Vamos pegar o `id` do aluno.

O Prisma retorna o `id` com um prefixo. Por exemplo, se tivermos o `id` "123", o Prisma o retornará como "StringIdGCValue123".

Vamos utilizar expressão regular para remover esse prefixo da string para obtermos o ID do aluno e armazená-lo numa variável `id`. Para atualizar nossa lista, basta passar este `id` para a nossa função `Template.removerAlunoLista()`.

```
GraphQL.wsConnection.onmessage = function (event) {
  const resposta = JSON.parse(event.data);
  if(resposta.type === 'subscription_data'){
    const aluno = resposta.payload.data.aluno;
    if(aluno.mutation === 'CREATED'){
      Template.inserirAlunoLista(aluno.node);
    }else if(aluno.mutation === 'DELETED'){
      const id = aluno.previousValues.id.replace(/StringIdGCValue\
((.*)\\)/, '$1');
      Template.removerAlunoLista(id);
    }
  }
}
```

Estamos apenas vendo como criar e apagar dados. Ao final deste capítulo você estará apto para desenvolver sozinho a funcionalidade "UPDATED", que deixarei como um desafio.

Essas são as funções a serem executadas ao recebermos dados do servidor. Mas lembre-se de que precisamos enviar ao Prisma uma inscrição, indicando os dados que queremos. Como estamos falando de dados, vamos ao nosso objeto `Aluno` e vamos criar uma função chamada `subscription`.

Dentro desta função nós criaremos nossa `subscription` como aprendemos anteriormente e pediremos para escutar eventos `CREATED` e `DELETED`.

Essa operação será enviada ao Prisma pelo método `send()` do objeto `wsConnection` e o objeto a ser enviado deve ser convertido em *string*. Junto com a operação, precisamos passar um campo `id` para identificar a ação e indicar o tipo da mensagem que estamos enviando, que será do tipo `subscription_start`.

```
const Aluno = {
  ...
  subscription: function(){
    const query = `
      subscription updatedAlunos {
        aluno(where: {
          mutation_in: [CREATED, DELETED]
        }){
          mutation
          node{
            id
            nomeCompleto
            idade
          }
          previousValues{
            id
          }
        }
      }
    `;
    GraphQL.wsConnection.send(JSON.stringify({
      id: '1',
      type: 'subscription_start',
      query
```

```

    })))
  }
}

```

Agora que já definimos como lidar com os dados enviados do servidor e quais dados queremos receber, precisamos iniciar estas configurações.

Vamos à função `iniciar` do objeto `Template`. No final dela, vamos chamar a função que criamos, `GraphQL.iniciarWS()`. Como ela retorna uma `Promise` assim que a operação de conexão for finalizada, definimos que após tudo pronto executaremos a função `Aluno.subscription()` para indicar ao Prisma os dados que queremos receber.

```

const Template = {
  iniciar: function(){
    Aluno.buscar()
      .then(({data: {alunos}}) => {
        Aluno.lista = alunos;
        Template.listarAluno();
      })

    GraphQL.iniciarWS().then(Aluno.subscription);
  },
  ...
}

```

Precisamos arrumar um único detalhe agora: quando alguém criar ou apagar um dado, receberemos esta informação e atualizaremos nossa lista. Porém, nós também seremos notificados sobre nossas próprias ações de criar e apagar.

Como nossas operações de criar e apagar já estão atualizando a lista, precisamos apagar isso para que a ação de tentar atualizar a lista não ocorra duas vezes.

Em `Template.criarAluno()`, vamos remover o `Template.inserirAlunoLista()`.

```

criarAluno: function(){
  event.preventDefault();
  const formulario = document.forms.novoAluno,
    novoAluno = {
      nomeCompleto: formulario.nomeCompleto.value,
      idade: parseInt(formulario.idade.value)
    };
  if(novoAluno.nomeCompleto && novoAluno.idade){
    formulario.nomeCompleto.value = '';
    formulario.idade.value = '';
    Aluno.criar(novoAluno);
    // removido
  }
},

```

Em `Template.apagarAluno()` , vamos remover o `Template.removerAlunoLista()`

```

apagarAluno: function(id){
  Aluno.apagar(id);
  // apagado
},

```

Agora se você for ao Prisma Playground e criar ou remover algum aluno, nossa página será atualizada. Você pode testar também abrindo dois navegadores com nossa aplicação. Ao criar ou remover algum aluno em uma página, a lista da outra janela também será atualizada.

6.11 Código do front-end completo

Nosso HTML final ficará da seguinte maneira:

```

<!DOCTYPE html>
<html>
<body>
  <form name="novoAluno" onsubmit="Template.criarAluno()" >
    <input type="text" name="nomeCompleto" placeholder="Nome Completo"

```

```

>
    <input type="number" name="idade" placeholder="Idade">
    <button type="submit" >Adicionar Aluno</button>
  </form>

  <ul id="listaAlunos" ></ul>

  <script src="./scripts.js"></script>
</body>
</html>

```

E nosso arquivo `scripts.js` ficará assim:

```

const listaAlunos = document.querySelector('#listaAlunos');

const GraphQL = {
  endpoint: 'http://localhost:3000',
  wsConnection: new WebSocket('ws:http://localhost:3000/graphql',
'graphql-subscriptions'),
  exec: function(query, variaveis){
    return fetch(GraphQL.endpoint, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ query: query, variables: variaveis }),
    })
    .then(resposta => resposta.json())
  },
  iniciarWS: function(){
    return new Promise(resolve => {
      GraphQL.wsConnection.onopen = function(){
        const mensagem = {
          type: 'init'
        };
        GraphQL.wsConnection.send(JSON.stringify(mensagem));

        GraphQL.wsConnection.onmessage = function (event) {
          const resposta = JSON.parse(event.data);
          if(resposta.type === 'subscription_data'){

```

```

        const aluno = resposta.payload.data.aluno;
        if(aluno.mutation === 'CREATED'){
            Template.inserirAlunoLista(aluno.node);
        }else if(aluno.mutation === 'DELETED'){
            const id =
aluno.previousValues.id.replace(/StringIdGCValue\((.*)\)/, '$1');
            Template.removerAlunoLista(id);
        }
    }
}

    resolve();
}
})
}
}

```

```

const Aluno = {
    lista: [],
    buscar: function(){
        const query = `
        query{
            alunos{
                id
                nomeCompleto
                idade
            }
        }
        `;
        return GraphQL.exec(query);
    },
    criar: function(novoAluno){
        const query = `
        mutation ($nomeCompleto: String!, $idade: Int!){
            createAluno(data: {
                nomeCompleto: $nomeCompleto
                idade: $idade
            }){
                id
                nomeCompleto
                idade
            }
        }
        `;
        return GraphQL.exec(query);
    }
}

```

```

    }
  }
  `;
  return GraphQL.exec(query, novoAluno);
},
apagar: function(id){
  const query = `
mutation ($id: ID!){
  deleteAluno(
    where: {
      id: $id
    }
  ){
    id
  }
}
`;
  return GraphQL.exec(query, {id});
},
subscription: function(){
  const query = `
subscription updatedAlunos {
  aluno(where: {
    mutation_in: [CREATED, DELETED]
  )){
    mutation
    node{
      id
      nomeCompleto
      idade
    }
    previousValues{
      id
    }
  }
}
`;
  GraphQL.wsConnection.send(JSON.stringify({
    id: '1',
    type: 'subscription_start',
    query

```

```

    )))
  }
}

const Template = {
  iniciar: function(){
    Aluno.buscar()
      .then(({data: {alunos}}) => {
        Aluno.lista = alunos;
        Template.listarAluno();
      })

    GraphQL.iniciarWS().then(Aluno.subscription);
  },
  listarAluno: function(){
    let html = ''
    Aluno.lista.forEach((aluno) => {
      html += `<li>Nome: ${aluno.nomeCompleto} - Idade:
${aluno.idade} - <button onclick="Template.apagarAluno('${aluno.id}')"
>X</button></li>`
    })
    listaAlunos.innerHTML = html;
  },
  criarAluno: function(){
    event.preventDefault();
    const formulario = document.forms.novoAluno,
      novoAluno = {
        nomeCompleto: formulario.nomeCompleto.value,
        idade: parseInt(formulario.idade.value)
      };
    if(novoAluno.nomeCompleto && novoAluno.idade){
      formulario.nomeCompleto.value = '';
      formulario.idade.value = '';
      Aluno.criar(novoAluno);
    }
  },
  inserirAlunoLista: function(novoAluno){
    Aluno.lista.push(novoAluno);
    Template.listarAluno();
  },
  apagarAluno: function(id){

```

```

        Aluno.apagar(id);
    },
    removerAlunoLista: function(id){
        const alunoIndice = Aluno.lista.findIndex(aluno => aluno.id ===
id);
        if(alunoIndice >= 0){
            Aluno.lista.splice(alunoIndice, 1);
            Template.listarAluno();
        }
    }
}

Template.iniciar();

```

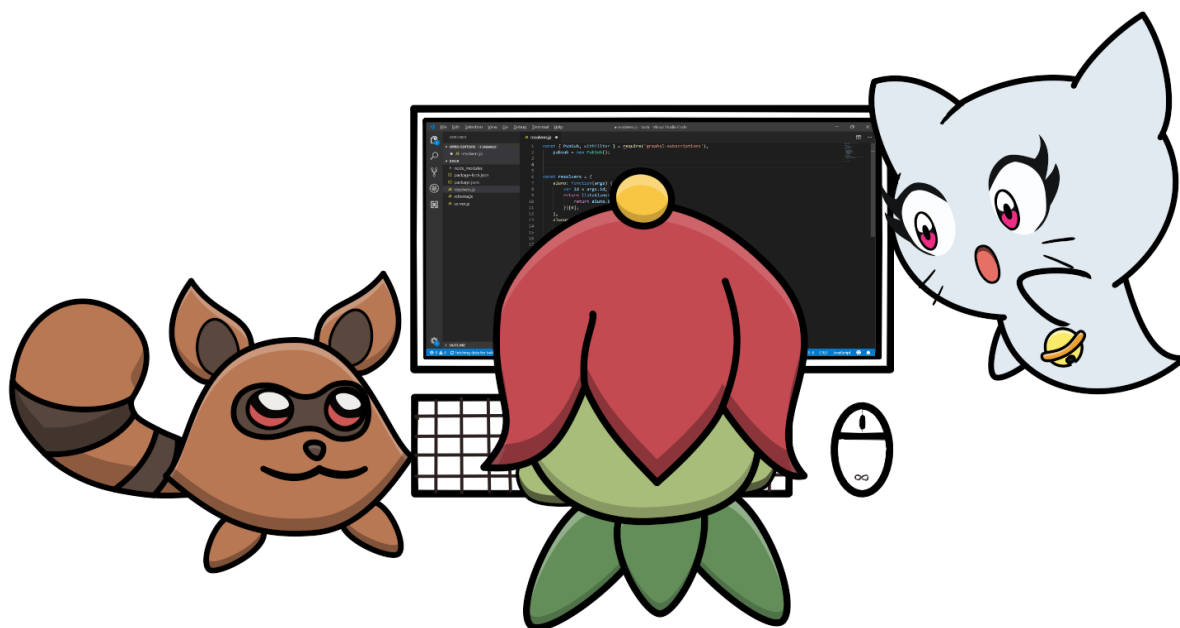
6.12 Tela feita!

Neste capítulo vimos como criar um front-end simples e funcional conectado ao nosso servidor GraphQL do Prisma. Podemos buscar, criar e remover alunos. Também vimos como ter a aplicação atualizada automaticamente com as *subscriptions*.

Não deixe de tentar criar a funcionalidade de atualizar dados já existentes que deixei como desafio. Caso prefira adicionar este novo recurso após terminar de ler o livro, não tem problema. Isso não influenciará na continuação da leitura.

No próximo capítulo veremos como criar nosso próprio servidor GraphQL, igual ao fornecido pelo Prisma!

Parte 4 — Programando um servidor



CAPÍTULO 7 Desenvolvimento back-end

Agora temos um front-end que consome um serviço GraphQL que foi gerado automaticamente pelo Prisma. Muitas vezes vamos querer ter nosso próprio servidor fornecendo um serviço mais customizado de acordo com as necessidades de nossos projetos.

Neste último capítulo veremos como criar um serviço GraphQL no back-end utilizando o próprio Node.js que já temos instalado em nossa máquina. Faremos um serviço que funcionará igual ao Prisma, então nosso front-end não precisará ser reestruturado.

Utilizaremos conceitos já apresentados nos capítulos anteriores, os quais ficarão mais claros agora que os veremos na prática.

7.1 Iniciando nosso projeto Node.js

O código completo deste projeto pode ser baixado pelo GitHub através do endereço <https://github.com/hanashiro/cdc-livro-graphql/>. Ele também será apresentado por completo no final deste capítulo.

Para iniciar nosso projeto back-end, vamos criar uma pasta chamada `back`.

Projetos Node.js utilizam um arquivo chamado `package.json`, que contém informações como nome do projeto, versão do código, descrição etc. Uma das partes mais importantes desse arquivo é ter a lista de bibliotecas e frameworks que utilizaremos. Assim, ao compartilharmos nosso projeto, apenas enviamos o código que escrevemos. Bibliotecas e frameworks são instalados automaticamente na versão certa de acordo com o que chamamos de lista de dependências.

O NPM cria este arquivo automaticamente para nós. Dentro da pasta `back`, abra o terminal e execute o comando `npm init -y` para criá-lo.

Com o `package.json` criado poderemos salvar a lista de bibliotecas e frameworks que utilizaremos em nosso servidor. Ainda no terminal, execute o comando `npm install --save graphql apollo-server`.

O `npm install` indica ao `npm` que queremos instalar algo na pasta em que estamos. Você também pode executar `npm i` como atalho para `npm install`. O `--save` indica para salvar tudo o que estamos instalando no `package.json`. Você também pode usar o seu atalho, `-s`. Os nomes seguintes são as bibliotecas que queremos instalar. O `apollo-server` nos ajudará a criar um serviço GraphQL em poucas linhas e o `graphql` é para podermos trabalhar com GraphQL.

Apollo é uma ferramenta que facilita o trabalho com GraphQL tanto no cliente quanto no servidor. No lado do cliente há versões dele para serem integradas com Angular, React, Vue etc.

Saiba mais em: <https://apollographql.com/>.

7.2 Schema e Resolvers

Iniciando o schema da aplicação

Vimos no começo do livro que primeiro precisamos definir um *schema* para a aplicação. Para deixar tudo organizado, vamos criar um arquivo chamado `schema.js`. Dentro dele, vamos escrever nosso *schema* como se fosse uma *string* comum e guardá-la em uma variável chamada `schema`. No fim do arquivo nós exportaremos essa *string* para poder importá-la e utilizá-la em outro arquivo.

Essa *string* será processada pelo `gql`, que importaremos do `apollo-server`. O `gql` é uma função que recebe uma *string* e a converte em um objeto do GraphQL. Durante o processo, o *schema* também é validado para assegurar que não há erros.

Iniciaremos nosso *schema* com a estrutura do *type* `Aluno`, igual ao que montamos anteriormente ao iniciar com o Prisma.

```
const { gql } = require('apollo-server');
const schema = gql`
type Aluno{
  id: ID!
  nomeCompleto: String!
  idade: Int
}
```

```
module.exports = schema;
```

Resolvendo requisições

Quando um cliente faz uma requisição a um servidor com GraphQL, precisamos indicar uma função que será executada para tratar seu pedido e dar uma resposta. Para isso, vamos criar um arquivo com o nome `resolvers.js`.

Dentro deste arquivo teremos um objeto chamado `resolvers`, onde ficarão as funções que serão executadas para resolver as solicitações do cliente. No final do arquivo nós vamos exportar o objeto para podermos acessá-lo em outros arquivos.

```
const resolvers = {  
  
};  
module.exports = resolvers;
```

7.3 Criando um servidor com Apollo Server

Agora que já temos uma estrutura base para o GraphQL (*schema* e lugar para as funções que serão executadas), precisamos criar um servidor que receberá as requisições do cliente e se integrará com o GraphQL.

Vamos criar um arquivo chamado `server.js` e importar o `ApolloServer` do pacote `apollo-server`, que permitirá a criação de um servidor com GraphQL em poucas linhas.

Também importaremos os arquivos `schema.js` e `resolvers.js`, para que o servidor que vamos criar conheça o nosso *schema* e tenha acesso às funções que serão executadas.

```
const { ApolloServer } = require('apollo-server'),
      schema = require('./schema'),
      resolvers = require('./resolvers');
```

Agora que já importamos tudo o que precisamos, vamos começar a criar nosso servidor. Você verá que com Node.js e Apollo será bem mais simples do que imagina!

Criamos um servidor instanciando `ApolloServer` e o guardamos em uma variável `server`. Passamos um objeto de configuração com os `resolvers`, e o `schema` será passado no campo `typeDefs`.

Para iniciar o servidor, basta executar o método `listen()`, para o qual passamos o número da porta em que queremos que o servidor passe a escutar as requisições do cliente. Vamos passar a porta 3000.

```
...
const server = new ApolloServer({ typeDefs: schema, resolvers });

server.listen(3000).then(() => {
  console.log(`Servidor funcionando!`);
});
```

Agora com o nosso servidor pronto não precisaremos mais alterar o arquivo `server.js`. Todas as funcionalidades que formos criar daqui em diante serão feitas apenas nos arquivos `schema.js` e `resolvers.js`.

Para iniciar o servidor, execute o comando `node server.js`.

7.4 Erguendo o servidor automaticamente após cada alteração com nodemon

Após cada alteração que fizermos em nosso código, teremos que parar o servidor e iniciá-lo novamente. Isso pode ser bem cansativo e tomar tempo de desenvolvimento.

Para melhorar nosso fluxo de trabalho, vamos utilizar uma ferramenta feita em Node.js chamada `nodemon`. Ela nos permite executar um arquivo e ele é reexecutado automaticamente assim que qualquer arquivo presente no mesmo diretório for alterado.

Para instalar o `nodemon` globalmente em sua máquina, execute `npm install -g nodemon`. Após a instalação, ele estará acessível pela linha de comando a partir de qualquer lugar da sua máquina.

Agora, quando quiser subir o servidor, execute `nodemon server.js` em vez de `node server.js`. Conforme formos alterando nossos arquivos de nosso projeto, o `nodemon` se encarregará de subir o servidor de novo automaticamente.

O `nodemon` também pode ser utilizado para reexecutar qualquer comando quando um arquivo for alterado, não se limitando apenas a JavaScript.

Você pode, por exemplo, reexecutar um arquivo Ruby ou Python caso esteja estudando essas linguagens. Basta indicar `--exec`, passar o comando a ser reexecutado a cada alteração de arquivos entre aspas e por último indicar o arquivo a ser executado pelo comando.

- `nodemon --exec "ruby" ./app.rb`
- `nodemon --exec "python" ./app.py`

7.5 Permitindo busca de alunos

Quando iniciamos com o Prisma vimos que já podemos fazer buscas, criar, apagar e alterar dados assim que declaramos o *schema*. Porém, quando criamos um serviço GraphQL, apenas podemos fazer operações que declaramos no *schema*.

Então, para que nosso serviço GraphQL permita a busca de alunos, precisamos voltar ao `schema.js` e declarar o tipo de busca. Para isso, declaramos o *type* `Query` e, dentro dele, o tipo de *query* e o que ela retorna.

Como o Prisma chamou nossa busca de vários alunos de `alunos`, vamos manter este nome para não precisarmos alterar o nosso front-end que já está funcionando com esse nome. Vamos dizer que o retorno é uma lista que não pode ser vazia, e os dados dessa lista serão do tipo `Aluno` que já declaramos no início deste capítulo.

```
const schema = `
type Query {
  alunos: [Aluno]!
}
...

```

Agora que a busca de alunos está declarada no *schema*, precisamos criar uma função que será executada para tratar essa requisição e dar uma resposta ao cliente. Vamos ao arquivo `resolvers.js`.

Para simplificar nossos estudos, vamos utilizar um *Array* no lugar de um banco de dados para armazenar nossos dados. Vamos chamá-lo de `listaAlunos`, e vamos iniciar com alguns alunos, simulando um banco de dados já com alguns dados gravados. Os dados de alunos precisam ter pelo menos os campos que foram declarados no *schema*.

```
const listaAlunos = [
  {
    "id": "123",
    "nomeCompleto": "Carlos Monteiro",
    "idade": 25
  },
  {
    "id": "456",
    "nomeCompleto": "Maria Silva",
    "idade": 26
  }
]
```

```

    },
    {
      "id": "789",
      "nomeCompleto": "João Souza",
      "idade": 23
    }
  ];

  const resolvers = {
    ...

```

Dentro do objeto `resolvers` que criamos anteriormente vamos criar uma função que retornará nossa lista de alunos. Esta função deve ter o mesmo nome da operação que declaramos no *schema* (`alunos`) e precisa estar dentro de um objeto com o mesmo nome do tipo da operação (`Query`).

```

  const resolvers = {
    Query: {
      alunos: function() {
        return listaAlunos;
      }
    }
  };
  ...

```

Note que apenas retornamos o nosso *Array* que possui nossos alunos. A única diferença com uma aplicação real é que nós faríamos acesso a um banco de dados e então retornaríamos a lista de alunos vinda do banco.

Então agora nós já percebemos que não precisamos nos preocupar pensando que o GraphQL é inseguro por permitir que o cliente indique os dados que ele quer receber, pois a todo o momento estamos no controle de cada detalhe. Se o campo de um objeto não estiver no *schema* o cliente não consegue requisitar, e nós fazemos acesso ao banco de dados manualmente, o que nos dá flexibilidade e total controle da nossa aplicação.

7.6 Testando as requisições com Playground

Agora já declaramos a busca de nossos alunos. Mas como testaremos se tudo está funcionando?

O Apollo nos fornece uma ferramenta igual ao Playground que conhecemos do Prisma. Na verdade, o próprio Playground do Prisma onde brincamos no começo do livro é baseado nela, então você já vai se sentir em casa. Ela utiliza o nosso *schema* para saber quais as estruturas dos dados e quais comandos estão disponíveis.

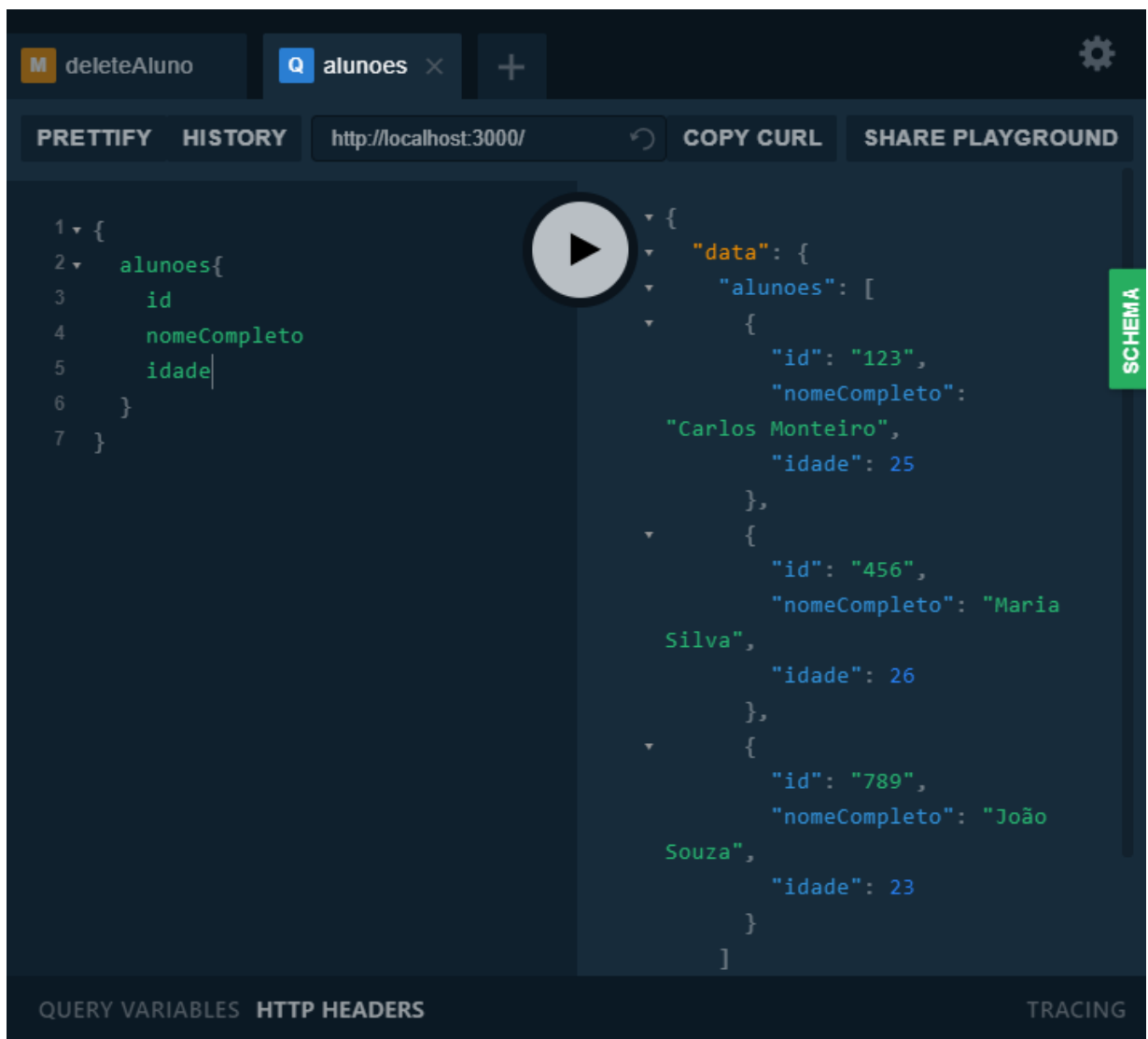


Figura 7.1: GraphQL Playground

Como indicamos que nosso servidor subiria na porta 3000, basta acessar `http://localhost:3000/graphql` em seu navegador para acessar o Playground. Faça um teste de busca de alunos como aprendemos anteriormente:

```
{
  alunos{
    id
    nomeCompleto
    idade
  }
}
```

Teremos o seguinte resultado:

```
{
  "data": {
    "alunos": [
      {
        "id": "123",
        "nomeCompleto": "Carlos Monteiro",
        "idade": 25
      },
      {
        "id": "456",
        "nomeCompleto": "Maria Silva",
        "idade": 26
      },
      {
        "id": "789",
        "nomeCompleto": "João Souza",
        "idade": 23
      }
    ]
  }
}
```

Vamos experimentar a funcionalidade de renomear campos que vimos no começo de nossos estudos de GraphQL. Renomeie o

campo `nomeCompleto` para `nome` . Vamos também remover o campo `idade` , para testar se ele realmente não virá.

```
{
  alunos{
    id
    nome: nomeCompleto
  }
}
```

Teremos o seguinte resultado:

```
{
  "data": {
    "alunos": [
      {
        "id": "123",
        "nome": "Carlos Monteiro"
      },
      {
        "id": "456",
        "nome": "Maria Silva"
      },
      {
        "id": "789",
        "nome": "João Souza"
      }
    ]
  }
}
```

Note que já temos um serviço do GraphQL funcionando e permitindo que, além de buscas, também indiquemos quais campos queremos que sejam retornados, além de podermos renomeá-los. Nosso único trabalho foi declarar uma busca no *schema* e criar uma função que retorna todos os alunos; quanto ao resto, o próprio GraphQL se encarregou!

Por enquanto nosso serviço apenas permite a busca de todos os alunos. Vamos continuar evoluindo-o.

7.7 Fazendo busca de um único aluno

Permitimos a busca de todos os alunos. Para permitir a busca de apenas um aluno precisamos declarar outra *query* no *schema*.

Vamos ao arquivo `schema.js` e no *schema* vamos declarar dentro do *type* `Query` uma busca de um único aluno. Para seguir o que tínhamos no Prisma, vamos chamar de `aluno`. Ele recebe um campo `id` do tipo `ID` como parâmetro e retorna um único objeto do tipo `Aluno`.

```
const schema = `
type Query {
  aluno(id: ID!): Aluno
  alunos: [Aluno]!
}
...

```

Agora que temos o *schema* preparado, vamos ao arquivo `resolvers.js` criar nossa função que tratará da busca deste aluno pelo seu ID. Como aprendemos, o nome da função tem que ter o mesmo nome da operação. Então nossa função se chamará `aluno()`, e ficará dentro de `Query`.

Toda função do *resolvers* recebe um objeto `root` e um parâmetro que contém os dados enviados pelo cliente. Então, aqui, como declaramos no *schema* que precisamos receber um campo chamado `id`, receberemos o campo `id` dentro deste parâmetro da função.

Vamos acessar nosso *Array* de alunos e executar o método `find()`, comparando o `id` recebido com o `id` de cada aluno.

```
const resolvers = {
  aluno: function(root, args) {
    var id = args.id;
    return listaAlunos.filter(aluno => {
      return aluno.id == id;
    })[0];
  },

```

```
alunos: function() {  
  ...  
}
```

No Playground teste a seguinte busca:

```
{  
  aluno(id: "456"){  
    id  
    nome: nomeCompleto  
  }  
}
```

Teremos o seguinte resultado:

```
{  
  "data": {  
    "aluno": {  
      "id": "456",  
      "nome": "Maria Silva"  
    }  
  }  
}
```

Da segunda vez ficou mais fácil, não é mesmo? Este é o fluxo que vamos seguir para montar as demais operações, mas as seguintes terão alguns detalhes a mais.

7.8 Apagando alunos

Vamos permitir a remoção de alunos do nosso "banco de dados". Você já sabe que a primeira coisa a se fazer é ir ao arquivo `schema.js` declarar esta operação. Mas agora teremos um pouco mais de trabalho.

Uma das diferenças agora é que apagar um dado não é uma *query*, é um *mutation*. Então vamos declarar o *type* `Mutation`. Para seguir o Prisma, vamos criar a remoção com o nome `deleteAluno`.

Seguindo a mesma ideia da busca de um único aluno, poderíamos criar nosso *mutation* da seguinte maneira:

```
const schema = `
type Query {
  aluno(id: ID!): Aluno
  alunos: [Aluno]!
}

type Mutation {
  deleteAluno(id: ID!): Aluno
}
```

Perfeito! Isso já daria certo. Mas como estamos seguindo uma estrutura igual à do Prisma, precisamos nos lembrar de que lá a operação de apagar não recebe um simples campo ID. Ela recebe um objeto com o campo `where`, como esse comando que fizemos anteriormente:

```
mutation{
  deleteAluno(
    where: {
      id: "cjk7k0ero1plm0b98mwgzghxv"
    }
  ){
    id
    nomeCompleto
    idade
  }
}
```

Como declarar este objeto que queremos receber? É agora que precisaremos dos *Inputs* que vimos anteriormente.

Primeiro vamos declarar um *Input*. Ele possui a mesma estrutura de um *Type*, mas é declarado com a palavra-chave `input`. Como queremos utilizá-lo para receber o `id` do aluno que será apagado, vamos declarar este campo dentro dele. Nomearemos este *input* de `AlunoWhere`.

```
...
input AlunoWhere{
  id: ID!
}
```

Agora vamos modificar nosso *mutation* `deleteAluno` para receber um `AlunoWhere` como parâmetro.

```
...
type Mutation {
  deleteAluno(where: AlunoWhere!): Aluno
}
```

Vamos ao arquivo `resolvers.js` para criar a função de remoção do aluno. Chamaremos a função de `deleteAluno`, que deverá ficar dentro de um objeto chamado `Mutation`. Ela vai receber os parâmetros enviados pelo cliente, no caso, um objeto `where` com um campo `id`.

Vamos procurar a posição do aluno no *Array* pelo método `findIndex()`, comparando o ID enviado pelo cliente com os campos `id` dos alunos. Se o método `findIndex()` não encontrar o aluno indicado pelo `id`, será retornado o valor `-1`. Se encontrado, teremos um índice maior ou igual a zero, que indica a posição deste aluno no *Array*, então utilizaremos o método `splice()` para removê-lo do *Array*.

O `splice()` recebe dois parâmetros: o índice do primeiro item que queremos remover e quantos itens desejamos que sejam removidos. Este método retorna um novo *Array* com os itens eliminados. Como tiramos apenas um aluno, vamos pegar o primeiro (e único) objeto do novo *Array* retornado, ou seja, posição `0`.

Se nenhum aluno for encontrado, retornaremos `null`:

```
...
Mutation: {
  deleteAluno: function(root, args){
    const indice = listaAlunos.findIndex(aluno => aluno.id ==
args.where.id);
    if(indice >= 0){
```

```
        return listaAlunos.splice(indice, 1)[0];
    }
    return null;
}
}
```

Agora vamos testar o seguinte comando no Playground:

```
mutation{
  deleteAluno(where: {
    id: "123"
  }){
    id
    nomeCompleto
  }
}
```

Se agora você testar listar todos os alunos, verá que um foi apagado.

Como estamos utilizando um *Array* no lugar do banco de dados, sempre que o servidor for reiniciado teremos o valor do nosso *Array* reiniciado também.

7.9 Criando novos alunos

Agora que aprendemos a apagar alunos e criar *Inputs*, vamos ver como criar alunos. Será um processo bem parecido.

Primeiro, vamos ao arquivo `schema.js` . Como recebemos um objeto chamado `data` na criação de um aluno, precisamos criar um *input* para ele também. Os alunos recebem um campo `nomeCompleto` , que é obrigatório, e um campo `idade` . Nosso *input* para a criação de aluno se chamará `AlunoInput` .


```
input AlunoInput{
  nomeCompleto: String!
  idade: Int
}
```

E agora podemos criar nosso *mutation* de criação de aluno, que chamaremos de `createAluno` .

```
type Mutation {
  createAluno(data: AlunoInput!): Aluno
  deleteAluno(where: AlunoWhere!): Aluno
}
```

Agora vamos ao `resolvers.js` para criar nossa função `createAluno()` dentro de `Mutation` . Ela receberá um parâmetro com o objeto `data` que terá o nome e idade do aluno. Em um banco de dados comum teríamos um `id` gerado automaticamente. Vamos simular isso pegando o tempo atual e inserindo no campo `id` .

Por fim, vamos inserir o novo aluno no nosso *Array* e retorná-lo.

```
createAluno: function(root, args){
  const novoAluno = args.data;
  novoAluno.id = Date.now();

  listaAlunos.push(novoAluno);
  return novoAluno;
}
```

Para testar, vamos ao Playground para executar o seguinte comando:

```
mutation{
  createAluno(data: {
    nomeCompleto: "Akira Kamiya",
    idade: 20
  }){
    id
    nomeCompleto
  }
}
```

Teremos o seguinte retorno:

```
{
  "data": {
    "createAluno": {
      "id": "742024200000",
      "nomeCompleto": "Akira Kamiya"
    }
  }
}
```

7.10 Permitindo subscriptions

Para deixar como o Prisma, ainda falta permitir inscrições por meio de *websockets*. Vamos ao nosso `schema.js` .

Lembra que indicamos o tipo de filtro para as operações para as quais queremos nos inscrever, e elas apenas podiam ter três possíveis valores (`CREATED` , `UPDATED` e `DELETED`)? Para limitarmos os possíveis valores que serão passados aqui, criaremos um `enum`. Vamos chamá-lo de `ModelMutationType` .

```
enum ModelMutationType{
  CREATED
  UPDATED
  DELETED
}
```

Para as inscrições, passamos os tipos de operações como um filtro. Como é uma estrutura que recebemos do cliente, precisamos declarar isso como um *input*. Ele terá o nosso conhecido campo `mutation_in` , que será um *Array* que poderá receber os valores que declaramos em nosso *enum*. Vamos chamar de `AlunoSubscriptionFilter` .

```
input AlunoSubscriptionFilter {
  mutation_in: [ ModelMutationType! ]
}
```

```
}
```

O retorno que recebemos de uma inscrição possui três campos: `mutation` (indica o tipo da operação realizada), `node` (objeto que foi criado ou atualizado) e `previousValue` (objeto que foi apagado ou valor anterior do objeto que foi atualizado). Para declarar essa estrutura que servirá de resposta para as inscrições, vamos criar um *type* que chamaremos de `AlunoSubscriptionPayload`.

```
type AlunoSubscriptionPayload{  
  mutation: ModelMutationType  
  node: Aluno  
  previousValues: Aluno  
}
```

Agora teremos que declarar a inscrição de aluno. Isso será dentro do *type* `Subscription`. Chamaremos de `aluno`, que receberá um parâmetro do tipo `AlunoSubscriptionFilter` e retornará um objeto do tipo `AlunoSubscriptionPayload`.

```
type Subscription {  
  aluno(where: AlunoSubscriptionFilter!): AlunoSubscriptionPayload  
}
```

Agora precisamos ir ao `resolvers.js`. Porém, aqui teremos alguns detalhes a mais para resolver.

Primeiro, precisamos do canal de comunicação que enviará dados ao cliente. A comunicação é feita com o `PubSub`, fornecida pelo `apollo-server`. Esta classe nos permitirá enviar dados por WebSockets aos clientes que fizerem uma *subscription*.

No começo do `resolvers.js`, vamos importar o `PubSub`, instanciá-lo e guardar na variável `pubsub`.

```
const { PubSub } = require('apollo-server');  
const pubsub = new PubSub();
```

Dentro do objeto `resolvers` precisamos criar um objeto `Subscription`. Dentro teremos um objeto `aluno` com uma função `subscribe`.

Os dados recebidos do cliente terão o objeto `where` com o campo `mutation_in`, que é um *Array* com um ou mais dos três valores que declaramos como *enum* (`CREATED`, `UPDATED` e `DELETED`).

Vamos utilizar o método `.map()` para inserir um prefixo `aluno_` nesses nomes. Assim, teremos canais específicos para cada operação de aluno. Então se um cliente quiser ouvir sobre criação e remoção de dados, nos passará `[CREATED, DELETED]`. Nós transformaremos isso em `[aluno_CREATED, aluno_DELETED]`. Assim, se futuramente você quiser incluir outros dados além de aluno, não haverá conflito nos canais de comunicação.

Por fim, precisamos passar esta lista para o método `asyncIterator()` de `pubsub`. Isso permitirá o servidor saber onde o cliente quis se inscrever.

```
...
Subscription: {
  aluno: {
    subscribe: (root, args) => {
      const eventNames = args.where.mutation_in.map(eventName =>
`aluno_${eventName}`);
      return pubsub.asyncIterator(eventNames);
    }
  },
}
```

Agora que criamos a possibilidade de inscrição, nosso servidor já saberá quem quer receber dados, e quais tipos de dados querem receber. Mas ainda precisamos dizer ao servidor que ele deve notificar os clientes quando os dados forem criados ou removidos. Isso é bem simples! Basta executar o método `publish()` do `pubsub`, passando o canal em que queremos enviar os dados e os dados que serão enviados ao cliente.

Em `resolvers.Mutation.createAluno()` adicionaremos este código, passando o canal `aluno_CREATED` e o objeto `aluno`, que possui o campo `mutation` com o valor `CREATED` e o novo aluno no campo `node`.

```

Mutation: {
  createAluno: function(root, args){
    const novoAluno = args.data;
    novoAluno.id = Date.now();

    listaAlunos.push(novoAluno);

    pubsub.publish('aluno_CREATED', {
      aluno: {
        mutation: 'CREATED',
        node: novoAluno,
        previousValues: null

      }
    });
    return novoAluno;
  },
  ...
}

```

Vamos fazer o mesmo em `resolvers.Mutation.deleteAluno()` . Nós estávamos retornando o aluno removido do *Array*. Vamos agora armazenar este aluno apagado da lista em uma variável `alunoDeletado` para poder reutilizá-lo.

Ao executar o `publish()` passaremos `alunoDeletado` no campo `previousValues` , e o canal utilizado será o `aluno_DELETED` .

```

Mutation: {
  ...
  deleteAluno: function(root, args){
    const indice = listaAlunos.findIndex(aluno => aluno.id ==
args.where.id);
    if(indice >= 0){
      const alunoDeletado = listaAlunos.splice(indice, 1)[0];
      pubsub.publish('aluno_DELETED', {
        aluno: {
          mutation: 'DELETED',
          node: null,
          previousValues: alunoDeletado
        }
      })
    }
  }
}

```

```
    });  
    return alunoDeletado;  
  }  
  return null;  
}  
}
```

Agora você pode ir ao Playground e testar a inscrição como aprendemos anteriormente a fazer com o Prisma. Execute o seguinte comando no Playground:

```
subscription updatedAlunos {  
  aluno(where: {  
    mutation_in: [CREATED, DELETED]  
  }) {  
    mutation  
    node {  
      id  
      nomeCompleto  
      idade  
    }  
  }  
}
```

O Playground começará a esperar por notificações, como mostrado no número 1 na figura seguinte.

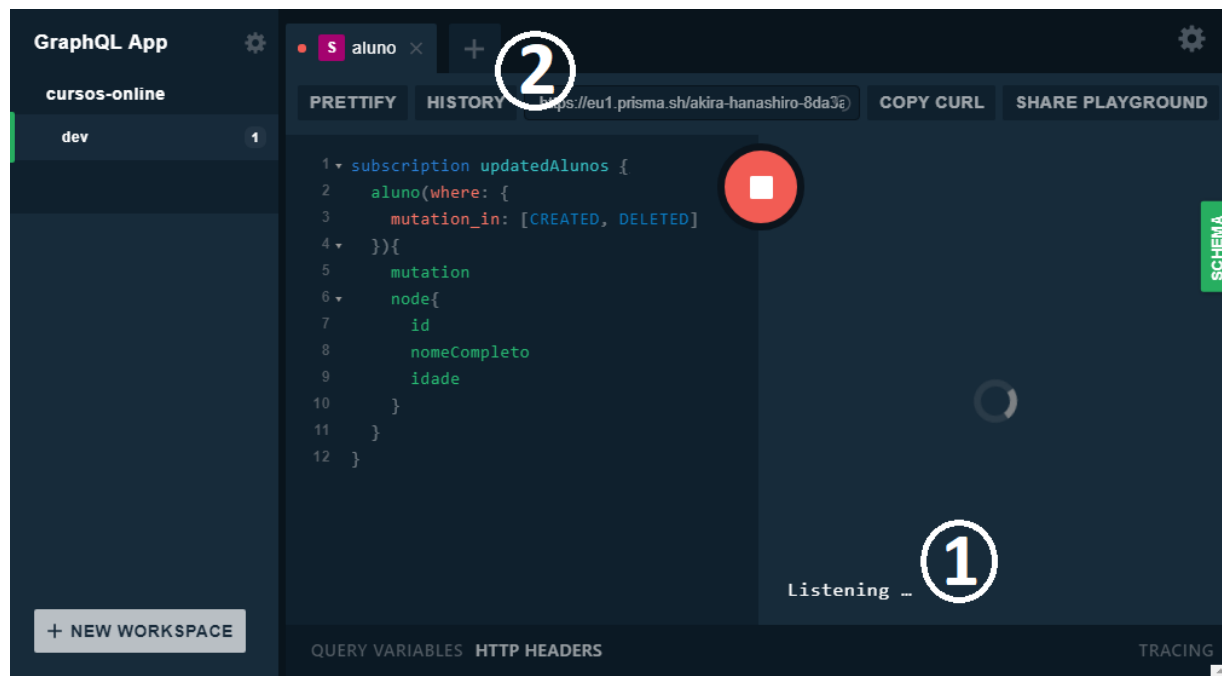


Figura 7.2: Playground — criação de subscription

Agora precisamos deixar esta tela dessa maneira, pois ela está esperando por atualizações em nossa lista de alunos. Para podermos executar outros comandos, abra uma nova aba no Playground como indicado no número 2 da imagem.

Crie um novo aluno na nova aba que se abriu no Playground:

```
mutation{
  createAluno(data: {
    nomeCompleto: "Amábile Brizon",
    idade: 19
  }){
    id
    nomeCompleto
    idade
  }
}
```

Ao executar este comando, volte à aba onde executamos a subscription . Teremos agora ao lado direito da tela a seguinte resposta:

```

{
  "data": {
    "aluno": {
      "mutation": "CREATED",
      "node": {
        "id": "782802000000",
        "nomeCompleto": "Amábile Brizon",
        "idade": 19
      }
    }
  }
}

```

7.11 Integrando front-end e back-end

Temos um serviço completo de GraphQL criado completamente por nós! Só precisamos fazer nosso front-end consumi-lo.

Como nós criamos nosso serviço GraphQL seguindo a mesma estrutura utilizada pelo Prisma, a única alteração que precisaremos fazer no front-end é mudar os endereços que ele acessa para fazer as requisições.

No projeto do front-end, vamos ao arquivo `scripts.js`. No objeto GraphQL vamos colocar o endereço do nosso serviço, `http://localhost:3000`. Na criação da conexão WebSocket, colocamos `ws://localhost:3000/graphql`.

```

const GraphQL = {
  endpoint: 'http://localhost:3000',
  wsConnection: new WebSocket('ws://localhost:3000/graphql', 'graphql-
subscriptions'),
  ...

```

Basta subir o servidor do front-end com o `tw-dev-server` e agora teremos um front-end consumindo um serviço GraphQL criado

totalmente por nós. Agora estamos totalmente independentes do Prisma.

A ação de alterar um dado já existente eu vou deixar como um desafio para você. Utilize os conhecimentos adquiridos até o momento para adicionar esta funcionalidade à nossa aplicação.

7.12 Código do back-end completo

Nosso arquivo `schema.js` ficou assim:

```
const { gql } = require('apollo-server');
const schema = gql`
type Query {
  aluno(id: ID!): Aluno
  alunos: [Aluno]!
}

type Mutation {
  createAluno(data: AlunoInput!): Aluno
  deleteAluno(where: AlunoWhere!): Aluno
}

type Subscription {
  aluno(where: AlunoSubscriptionFilter!): AlunoSubscriptionPayload
}

type Aluno{
  id: ID!
  nomeCompleto: String!
  idade: Int
}

input AlunoInput{
  nomeCompleto: String!
  idade: Int
}
```

```

input AlunoWhere{
  id: ID!
}

input AlunoSubscriptionFilter {
  mutation_in: [ ModelMutationType! ]
}

type AlunoSubscriptionPayload{
  mutation: ModelMutationType
  node: Aluno
  previousValues: Aluno
}

enum ModelMutationType{
  CREATED
  UPDATED
  DELETED
}
`;
module.exports = schema;

```

Nosso arquivo `resolvers.js` ficou da seguinte maneira:

```

const { PubSub } = require('apollo-server');
const pubsub = new PubSub();

const listaAlunos = [
  {
    "id": "123",
    "nomeCompleto": "Carlos Monteiro",
    "idade": 25
  },
  {
    "id": "456",
    "nomeCompleto": "Maria Silva",
    "idade": 26
  },
  {
    "id": "789",
    "nomeCompleto": "João Souza",

```

```

        "idade": 23
    }
];

const resolvers = {
  Query: {
    aluno: function(root, args) {
      var id = args.id;
      return listaAlunos.filter(aluno => {
        return aluno.id == id;
      })[0];
    },
    alunos: function() {
      return listaAlunos;
    }
  },
  Mutation: {
    createAluno: function(root, args){
      const novoAluno = args.data;
      novoAluno.id = Date.now();

      listaAlunos.push(novoAluno);

      pubsub.publish('aluno_CREATED', {
        aluno: {
          mutation: 'CREATED',
          node: novoAluno,
          previousValues: null
        }
      });
      return novoAluno;
    },
    deleteAluno: function(root, args){
      const indice = listaAlunos.findIndex(aluno => aluno.id ==
args.where.id);
      if(indice >= 0){
        const alunoDeletado = listaAlunos.splice(indice, 1)[0];
        pubsub.publish('aluno_DELETED', {
          aluno: {
            mutation: 'DELETED',

```

```

        node: null,
        previousValues: alunoDeletado
      }
    });
    return alunoDeletado;
  }
  return null;
},
Subscription: {
  aluno: {
    subscribe: (root, args) => {
      const eventNames = args.where.mutation_in.map(eventName =>
`aluno_${eventName}`);
      return pubsub.asyncIterator(eventNames);
    }
  },
},
},
};

```

```
module.exports = resolvers;
```

E por fim, nosso arquivo `server.js` :

```

const { ApolloServer } = require('apollo-server'),
  schema = require('./schema'),
  resolvers = require('./resolvers');

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

server.listen(3000).then(() => {
  console.log(`Servidor funcionando!`);
});

```

7.13 Missão cumprida!

Missão cumprida, finalizamos nossos estudos sobre GraphQL!

Vimos que montar um serviço GraphQL pode ser bem simples. Lembre-se que o GraphQL pode ser utilizado com qualquer linguagem de programação, banco de dados ou framework.

Se você chegou até aqui com tudo funcionando, então você finalizou este livro com sucesso e agora já sabe muito sobre como trabalhar com GraphQL, tanto no front quanto no back. Parabéns!

Eu realmente espero que este livro tenha sido de grande valor a você e para a sua carreira.

O código completo deste projeto pode ser encontrado no meu GitHub pelo endereço <https://github.com/hanashiro/cdc-livro-graphql/>.

Outros links para continuar aprendendo sobre GraphQL:

- **Meu curso de GraphQL e outros na TreinaWeb:** <https://treinaweb.com.br/cursos-online>
- **Nosso blog:** <https://treinaweb.com.br/blog>
- **Site oficial do GraphQL:** <https://graphql.org>
- **Tutoriais de GraphQL:** <https://www.howtographql.com>
- **Apollo:** <https://www.apollographql.com/>
- mais links em *sobre o autor*

Muito obrigado por escolher este livro para o seu aprendizado. Até nosso próximo encontro!