Politécnica UFRJ

INTERFACE EM 'PERFIS E LISTAS' PARA SISTEMA GERENCIADOR DE BANCO DE DADOS EM GRAFO

Filipe Santos Pacheco Prates

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro

Orientador: Daniel Ratton Figueiredo

Rio de Janeiro
Fevereiro de 2024

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica

Engenharia de Computação e Informação

Politécnica UFRJ

INTERFACE EM 'PERFIS E LISTAS' PARA SISTEMA GERENCIADOR DE BANCO DE DADOS EM GRAFO

Filipe Santos Pacheco Prates

PROJETO FINAL SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO.

Autor:

_____

Filipe Santos Pacheco Prates

Aprovada por:

_____

Prof. Daniel Ratton Figueiredo, D.Sc.

_____

Prof. Claudia Maria Lima Werner, D.Sc.

_____

Prof. Marta Mattoso, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

FEVEREIRO DE 2024

Declaração de Autoria e de Direitos

Eu, Filipe Santos Pacheco Prates, CPF *146.334.207-12*, autor da monografia *Interface em 'Perfis e Listas' para Sistema Gerenciador de Banco de Dados em Grafo*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.

2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.

3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.

4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.

5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.

6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.

7. Por ser verdade, firmo a presente declaração.

_____

Filipe Santos Pacheco Prates

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

Agradecimentos

Agradeço as minhas conexões.

Obrigado em especial ao meu irmão Bruno, minha irmã Aya, meu pai Antonio João, minha mãe Jacqueline, minha vó Dorita, e meus Professores, em especial ao Daniel, ao Fernando e o Bernard (e o Luis) por terem me disponibilizado um ambiente de criação, a Rô e a Erica (e o Andy e outros tantos) que tankaram o suporte, ao Cadu pelas conversas de Design, e os meus amigos Pedro, Bernardo, Nas, Ian, Josh, os devs brabos da JG, os colegas da escalada, e todos os muitos outros que me influenciaram.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro

INTERFACE EM 'PERFIS E LISTAS' PARA SISTEMA GERENCIADOR DE BANCO DE DADOS EM GRAFO

Filipe Santos Pacheco Prates

Fevereiro/2024

Orientador: Daniel Ratton Figueiredo

Departamento: Engenharia de Computação e Informação

Desenvolvimento de uma interface de gerenciamento de dados para bancos de dados Neo4j, através de uma API em Graphql, com foco na visualização, navegação e entendimento intuitivo dos dados do grafo, permitindo exploração e manipulação arbitrária dos nós e relações.

Palavras-chave. Interface, Banco de Dados, Grafos, GraphQL

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Computer Engineer

Filipe Santos Pacheco Prates

February/2024

Advisor: Daniel Ratton Figueiredo

Department: Computer and Information Engineering

Development of an interface for data management for Neo4j graph database, through an Graphql API, with focus on visualization, navigation, and intuitive understanding of the data, allowing for arbitrary exploration and manipulation of the graph.

Keywords. Interface, Database, Graphs, GraphQL

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Gerenciar adequadamente os dados armazenados em um banco de dados é fundamental para qualquer sistema de informação. Tal gerenciamento precisa ser eficiente e intuitivo, de maneira a evitar alterações equivocadas e garantir que os usuários do sistema interajam com dados atualizados e corretos.

O gerenciamento de grandes quantidades de dados armazenados em grafos podem ser particularmente complexos. As ferramentas de gerenciamento de dados disponíveis, em especial quando lidando com bancos de dados Neo4j Database, se limitam à interações diretas por DCL (Data Control Language / Linguagem de Controle de Dados) e resultados como visualizações gráficas e textuais (no Neo4j Browser, a Graph result frame). Quando se tratam de um número considerável de nós e relações armazenadas, a visualização gráfica ou em lista pode ser de difícil entendimento, especialmente quando utilizados por usuários sem conhecimento prévio ao schema de dados. Além disso, existe o risco de alterações equivocadas de grande alcance nos dados.

1.2 Contribuição

Este trabalho apresenta uma interface de gerenciamento de dados para bancos de dados Neo4j, através de uma API em Graphql, com foco na visualização, navegação e gerenciamento intuitivo dos dados do grafo armazenado. A interface recebe o

schema do banco de dados e gera um ambiente de "Perfis e Listas", que permite a criação e edição de nós, seus relacionamentos, e os dados de cada um destes. Hoje o sistema é usado amplamente por mais de 30 colaboradores da empresa para qual foi desenvolvido, principalmente dos times de Qualidade, Desenvolvimento, Suporte e Educacional.

O trabalho documenta o funcionamento e a arquitetura do sistema, tanto a interface do usuário, que é gerada através de componentes reutilizáveis e as ferramentas Nuxt.js, Vue.js e Apollo Client, como a camada intermediária, que conecta a interface ao banco de dados. Para tal, utiliza-se o o Node.js, Express.js, e a biblioteca oficial do Neo4j que conecta o banco de dados à uma API em GraphQL, @neo4j/graphql.

A interface do usuário segue uma dinâmica de navegação diferente, onde é mapeado o isomorfismo de um nó e seus vizinhos com uma página de perfil, suas abas e seus elementos, de maneira a conseguirmos gerar uma página de perfil para cada um dos nós no banco e assim navegar facilmente através de suas relações para os perfis de outros nós – alterando ou acrescentando dados arbitrariamente no processo.

1.3 Descrição do uso da interface para gerenciamento de dados em suporte à plataforma adaptativa de aprendizagem e avaliação gamificada

A interface descrita neste trabalho foi inicialmente desenvolvida na empresa de tecnologia em educação Jovens Gênios Provedor de Conteúdo LTDA. A Jovens Gênios disponibiliza uma plataforma adaptativa de aprendizagem e avaliação gamificada para alunos do ensino público e privado em diversos estados do Brasil. Seu principal domínio de dados consiste de nós das estruturas escolares (Redes de ensino, Escolas, Turmas, Alunos, Professores, Gestores), nós dos conteúdos gerados pela empresa (Questões, Resumos, Tópicos, Cursos, Disciplinas), nós das atividades que acontecem nas plataformas (Tarefas, Provas Diagnósticas, Batalhas, etc), e, principalmente, os nós referentes às respostas dos alunos, que conecta o aluno à

questão e à atividade que levou o aluno àquele conteúdo.

A estrutura em árvore da organização de tópicos, com questões nos tópicos folha, e a da estrutura escolar, assim como a facilidade de realizar a manutenção e desenvolver novas funcionalidades com as ferramentas disponíveis ao time de desenvolvimento, foram os fatores determinantes para a escolha de um banco em grafo, em especial o Neo4j.

O gerenciamento dos dados, porém, se tornou um obstáculo à medida que novos conteúdos foram sendo produzidos, e mais alunos e turmas sendo adicionados no grafo, entidades estas que se alteram com o tempo e precisam ser regularmente atualizados. Para resolver a questão da manipulação direta ao grafo por usuários sem um conhecimento prévio maior do schema, foi desenvolvido então a interface ao sistema gerenciador descrito neste trabalho.

1.4 Papeis dos Usuários da interface

Os perfis dos usuários que utilizam a interface na empresa de tecnologia mencionada são os seguintes:

- Educacional (Responsáveis pelo contato direto com as escolas e definição das turmas)
- Suporte (Contato online com alunos e professores para esclarecer dúvidas, mapear e resolver eventuais problemas)
- Controle De Qualidade (Auxílio para setup de testes e entendimento do comportamentos das plataformas em diferentes casos)
- Desenvolvedores (Visualizar estado do grafo no banco de dados e realizar eventuais manipulações)

Capítulo 2

Tecnologias Utilizadas

2.1 Arquitetura em três camadas

2.2 Neo4j

O Neo4j é um sistema de gerenciamento de banco de dados orientado a grafos desenvolvido pela Neo4j Inc., que opera sob uma estrutura de dados que consiste em nós, relacionamentos e propriedades. Isso faz do Neo4j uma escolha ideal para cenários onde as relações entre os dados são tão importantes quanto os dados em si.

Cada Nó e cada Aresta possui um ou mais Rótulos (*labels*), que instancia um index de lookup, funcionando similar à uma tabela num banco relacional. O que permite, por exemplo, eficientemente recuperar os dados de todos os nós ou todas as arestas de um certo rótulo.

INC. (2024a)

2.2.1 Cypher Query Language

O banco Neo4j não utiliza SQL como a linguagem de controle de dados, e sim uma linguagem própria, a Cypher Query Language.

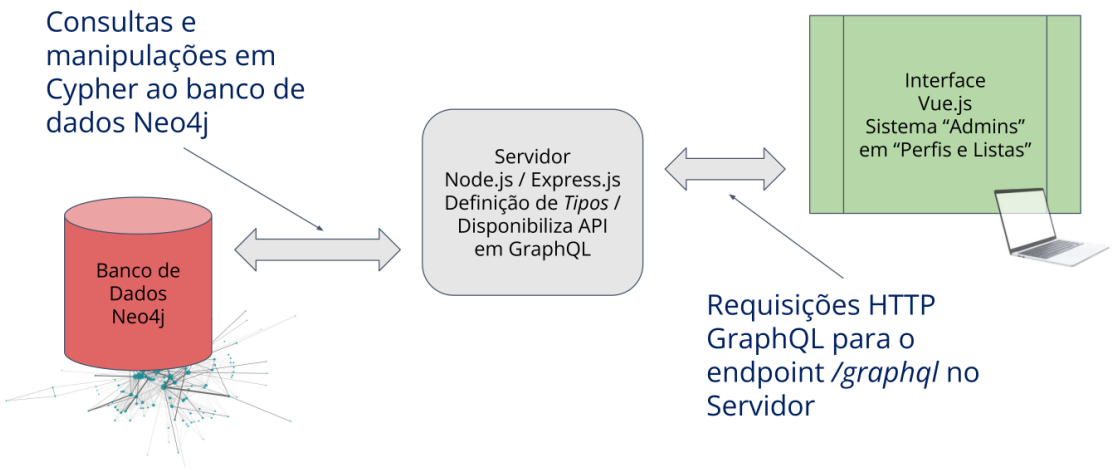

Figura 2.1: Diagrama da arquitetura em três camadas em que uma interface desenvolvida em Vue.js envia algum pedido através requisições HTTP em GraphQL. No servidor, o pedido gerado pela interação do usuário com a interface é traduzido para expressões em Cypher Query Language, que então são executadas no banco de dados.

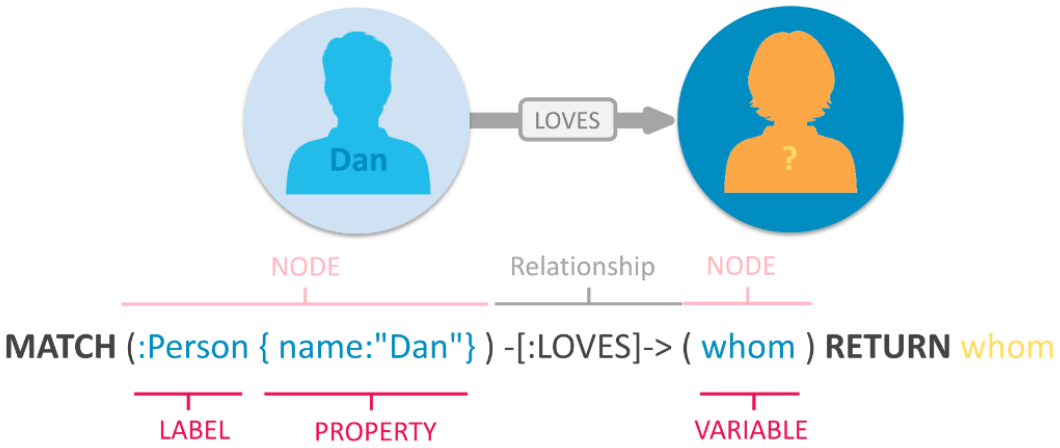

Figura 2.2: Exemplo de uma consulta em Cypher. Encontre todos os nós com *rótulo* "*Person*"(Pessoa) e propriedade "*name*"(nome) igual à string "Dan"; que também possuem uma relação com *rótulo* "LOVES"com algum outro nó sem condicional de rótulo, e então retornar os dados dos nós desses outros nós. Ou seja, *retorne as pessoas que os "Dan"s amam.* Imagem retirada de https://neo4j.com/developer/cypher/.

Ultimamente é através de Cyphers que é feita a comunicação com os dados armazenados no Neo4j.INC. (2024b)

2.3 GraphQL

Optamos por utilizar uma API em GraphQL na comunicação entre a interface e o servidor. GraphQL é uma linguagem de consulta e manipulação de dados, que permite um único endpoint no servidor escutar e resolver diferentes pedidos de diversas interfaces clientes. O pedido é especificado através de uma linguagem chamada GraphQL (em arquivos .gql), que é enviada no corpo de uma requisições HTTP, sempre com verbo POST.

O servidor disponibiliza uma série de consultas e requisições de alterações (*queries* e *mutations*) para o cliente. Em todas as operações também permite configuração no pedido de com argumentos opcionais e escolhas de quais propriedades do tipo da sua resposta o cliente quer exatamente.

O cliente então pode requisitar qualquer uma das funções disponíveis, e, da resposta, escolher exatamente o que precisa como retorno naquele momento. Tal escolha se dá através da estrutura em árvore da linguagem - que lembra a estrutura de um JSON.

FOUNDATION (2024a)

2.3.1 @neo4j/graphql

Para gerar todas as queries e mutations de CREATE READ UPDATE DELETE para os *tipos* de nós e arestas disponíveis para o banco de dados, utilizamos a biblioteca oficial da neo4j que disponibiliza também linguagens para definir e gerar o schema do banco de dados. INC. (2024c)

Abaixo, vemos exemplos de possíveis pedidos em GraphQL enviados pelo cliente. Primeiro uma requisição para recuperar os alunos de uma certa escola, e as notas em suas provas.

```
1 # .gql
2 query GetSchoolStudentsWithAssignmentGrades(schoolId: ID!){
3     schools( where: {id: $schoolId}){
4         students{
```

```
            id
            name
            assignments {
                id
                grade
            }
        }
    }
}
```

Se, além da nota (*grade*) do aluno, o cliente precisar de uma descrição das provas (*assignments*) dele, podemos apenas acrescentar tal propriedade na requisição. O mesmo pode ser feito com qualquer propriedade (incluindo arestas) definida no *tipo*.

```
            [...]
            assignments {
                id
                grade
                description
            }
            [...]
```

Andando pelas arestas (abrindo chaves) é possível retornar qualquer subconjunto de dados a partir do resultado da *query* realizada (dado que é autorizado para tal), podemos requisitar exatamente o que o frontend precisa, minimizando os recursos de banda.

Segundo, um exemplo de requisição de operação de alteração (*mutation*) na linguagem GraphQL.

```
# .gql
mutation ChangeStudentEmail(id: ID! newEmail: string){
    updateStudent(
        where: {id: $id},
        update: {email: newEmail}
    ){
        student {
            id
        }
    }
}
```

Tais requisições seriam resolvidas quando eviadas à um servidor no qual utilizamos a biblioteca @neo4j/graphql com as definições de *tipos* simplificadas como a seguir:

```
# types.graphql
type School{
    id: ID
    students: [Student] @relationship
    [...]
}

type Student {
    id: ID
    assignments [Assignment] @relationship
    [...]
}

type Assignment {
    id: ID
    grade: Float
    description: String
    [...]
}
```

2.4 Apollo

Biblioteca de gerenciamento de estados para JavaScript, através de GraphQL. É composta pois duas partes que se comunicam entre si:

- **Apollo Client** - Utilizado na aplicação cliente. Permite funcionalidade de controle de requisição de dados, variáveis de carregamento, controle de cache, e variáveis de estados locais. Auxilia o processo de realizar requisições em GraphQL.

- **Apollo Server** - Servidor GraphQL compatível com qualquer cliente que envia uma requisição GraphQL à ele, incluindo Apollo Clients. Utilizado na geração do schema que as requisições (e consequentemente os dados) precisam

seguir. É conectado com poder de gerenciamento à qualquer fonte de dados (no caso do projeto do trabalho, um banco Neo4j).

INC. (2024d)

2.5 Node.js

Software de código aberto baseado no interpretador V8 que permite a execução de códigos JavaScript em múltiplas plataformas. FOUNDATION (2024b)

2.6 Express.js

Framework web mais popular para node.js. Minimalista, disponibilizando apenas funcionalidades básicas para desenvolvimento web como handling para diferentes verbos HTTP, e, crucialmente para o projeto, middlewares para camadas de autenticação e autorização. FOUNDATION (2024c)

2.7 Vue.js

Vue.js é um framework para construir interfaces web. Disponibiliza um processo intuitivo de desenvolvimento de componentes e telas reativas. Os arquivos .vue contêm três partes:

- o **Template** em HTML que serve para organizar os componentes na tela, com interação direta com o JavaScript através das diretivas lógicas como v-for (permitindo renderização iterativa de componentes), v-if (permitindo renderização condicional), v-model (que permite criar ligações bidirecionais entre variáveis em diferentes componentes) dentre outras.
- o **Script**, que abriga as funcionalidades lógicas do componente, incluindo requisições ao banco de dados e controle de estados da interface
- o **Style**, onde estão as regras de CSS que ditam os estilos dos componentes

VUEJS (2024)

Capítulo 3

Descrição do sistema desenvolvido

O objetivo norteador do desenvolvimento da interface foi de proporcionar uma maneira amigável para mais facilmente gerenciar os dados de um banco de dados em grafo complexo com o Neo4j. O sistema deveria permitir ao usuário entender o estado do grafo, e criar, editar, conectar e desconectar arbitrariamente qualquer nó armazenado no banco de dados.

3.1 Aplicação Cliente

O objetivo da aplicação cliente é ser a interface que um usuário sem conhecimento prévio dos dados armazenados consegue interagir, e, para cada ação realizada na aplicação cliente, gera-se uma requisição em GraphQL que é enviada em uma requisição HTTP POST ao endpoint endpoint graphql do servidor.

Utilizamos o framework web Vue.js para o desenvolvimento das páginas definidas abaixo.

3.1.1 Perfis e Listas

Junto com o time de Design, identificamos uma maneira de entender e visualizar o grafo que está armazenado no banco, dando para cada nó uma página de perfil, que mostra e permite edição de suas propriedades e relações, além de uma página de lista/pesquisa para cada rótulo (relevante) no banco de dados. A partir deste isomorfismo, foi possível criar uma interface que permite aos usuários navegarem e editarem o grafo de maneira intuitiva.

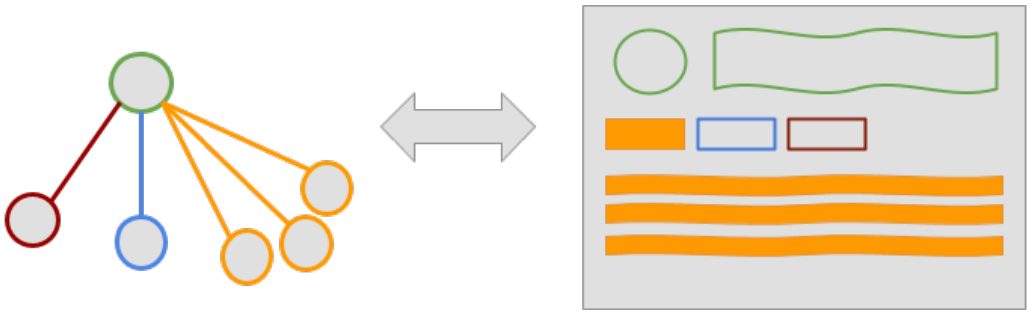

Figura 3.1: Abstração de um **nó dono** (verde) está relacionado à **um nó de rótulo X** (vermelho), **um de rótulo Y** (azul), e possui relação com **três nós de rótulos Z** (amarelos) no grafo armazenado no banco de dados.

Temos então dois tipos de telas:

- Perfis - Com as propriedades do nó dono na parte superior da tela, e diferentes abas (uma para cada tipo de relação que o nó possui) na parte inferior. Dentro de cada aba, uma lista de elementos com os nós vizinhos através daquele tipo de relação. Tal lista de elementos possui hiperlinks para páginas de perfis deste vizinhos. Desta maneira conseguimos andar pelo grafo através dos nós e relações, e em cada passo do caminho passamos pelo perfil do dó atual, com ações como edição, criação ou conexão entre nós.

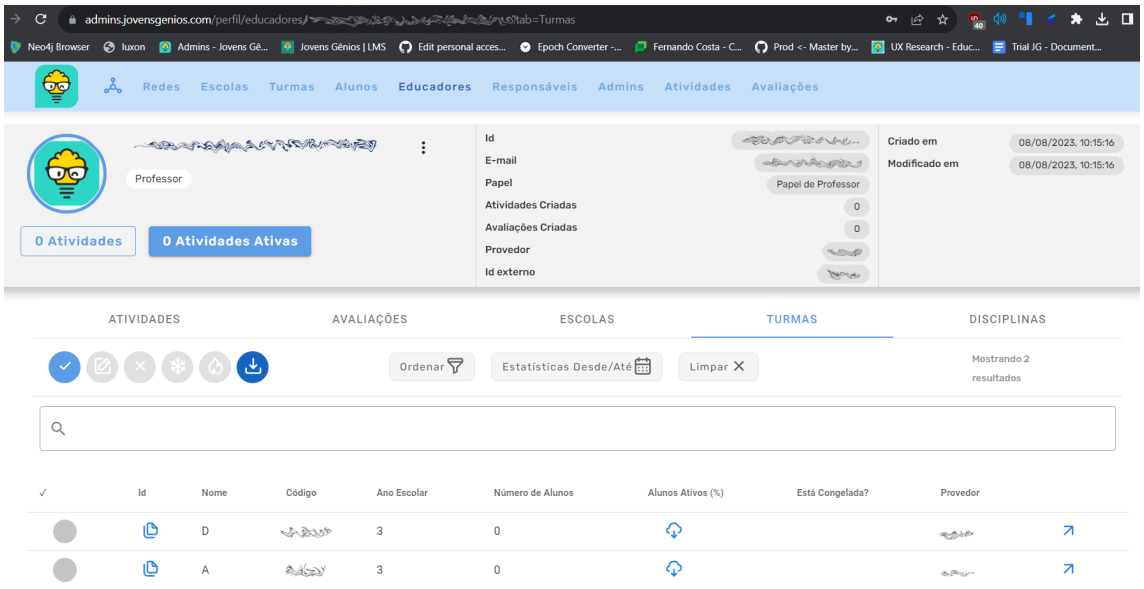

Figura 3.2: Página de Perfil de um educador. Note as ações em azul e cinza no cabeçalho da lista de vizinhos. Inclui edição, remoção e criação da aresta, entre o nó do perfil e os elementos selecionados, assim como outras ações específicas do *tipo* da aba (Congelar/Descongelar turma).

- Listas - Precisamos começar a navegar pelo grafo a partir de algum nó inicial. Para isso existem as páginas de listas. Nelas são listados todos os elementos de um certo *tipo* (de forma paginada), com hiperlinks para suas páginas de perfil. Também possui funcionalidades de busca e filtragem genéricas, além de acesso à ações diretas com os elementos listados.

A vantagem dessa modelagem é que conseguimos perceber que toda página de perfil (e de lista) segue a mesma regra, logo o código referente à sua implementação pode ser reutilizado. Ná prática, a aplicação consiste em apenas uma página de perfil e uma página de lista (além da página de login/autenticação), além de mapas que definem as especificidades de cada caso, facilitando sua manutenção e minimizando o tamanho do código, consequentemente minimizando também bugs em produção.

O frontend então é composto pelas duas principais páginas do sistema, uma camada superior que controla os formulários e ações globais, e diversos componentes Vue específicos que são utilizados em ambas as páginas. Além disso possuímos diversos arquivos JSON,que informam o que cada perfil de cada *tipo* possui de específico,

como por exemplo quais as ações específicas que são liberadas, dependendo do *tipo* do perfil e do *tipo* da aba. Tais arquivos ficaram armazenados na pasta utils.

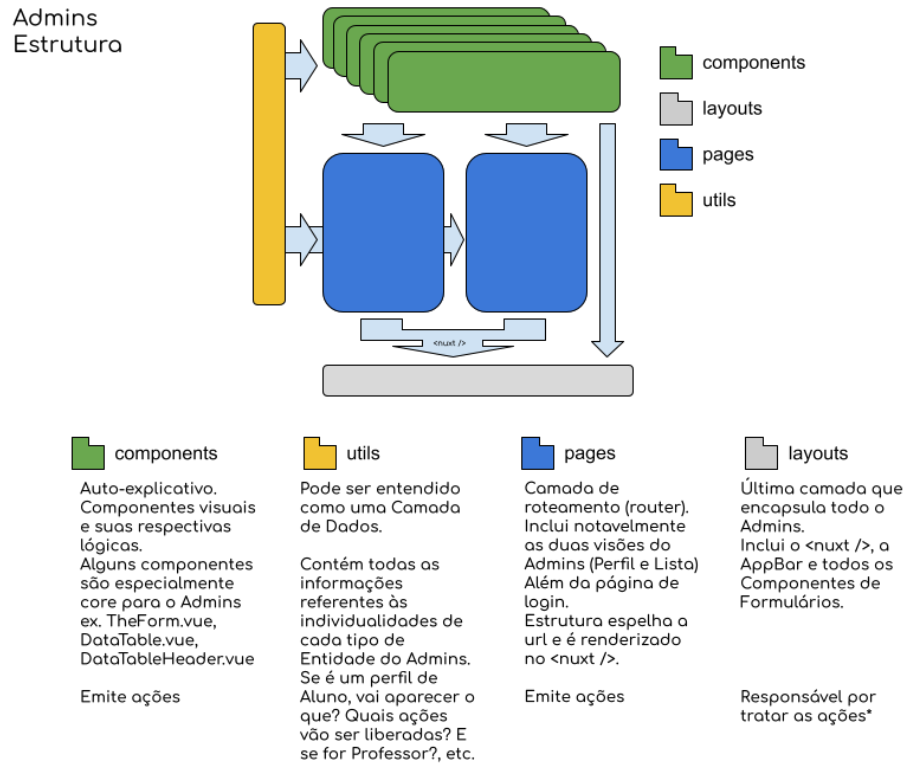

Figura 3.3: Documentação da organização dos diretórios, e comunicações entre partes, do código da Interface em "Perfil e Listas"desenvolvida para Jovens Gênios em Vue.js.

3.1.2 Uso da InstropectionQuery

A IntrospectionQuery é uma requisição disponibilizada pela neo4j/graphql que pede ao servidor o schema de dados armazenado nele, além de quais são as consultas (queries) e mutações (mutations) permitidas pela API. Assim conseguimos não só conhecer todas as requisições possíveis, seus argumentos e tipos, como também todos os *tipos*, como definidos nas definições de *tipos*. Utilizamos tais informações, por exemplo, quando abrimos um formulário de edição, utilizando a resposta para montar dinamicamente o número e tipo de componentes de input de dados, dependendo das propriedades do *tipo* informadas pela requisição. Se acrescentarem mais

uma propriedade na definição de tipos no backend, automaticamente o frontend vai mostrar um campo para esta, sem nenhum deploy do frontend necessário.

Tal abordagem de construção dinâmica de formulários e perfis é interessante e foi utilizada durante alguns meses, porém percebemos que muitas vezes não queremos que o usuário desse gerenciador tenha acesso direto à edição de alguns dos atributos de alguns nós. Alguns atributos deveriam ser populados automaticamente por outros fluxos, e não editados manualmente. Mostrar qualquer propriedade automaticamente no sistema "Admins" acabava confundindo os usuários.

Optamos então para uma abordagem híbrida, na qual recebemos no frontend quais são os atributos do *tipo* referente na montagem dos formulários de criação e edição - porém os filtramos, mostrando apenas os que são editáveis, através de mapas em arquivos guardados no "utils". Esta abordagem híbrida também é utilizada nas colunas das tabelas das listas, e na definição das propriedades que aparecem no cabeçalho do perfil.

Abaixo mostramo parte de um dos arquivos JSON do diretório "utils"que é usado para definir as propriedades a serem mostradas em cada cabeçalho de cada perfil, e como devem ser exibidas.

```
{
  "Student": {
    "header": [
      "Id",
      "email",
      "schoolYear",
      ...
      "provider",
      "externalId"
    ],
    "custom": [
      {
        "prop": "questionsAnsweredCount",
        "extra": "Questoes Feitas",
        "color": "#5C9CE5"
      },
      {
        "prop": "correctQuestionsAnsweredCount",
```

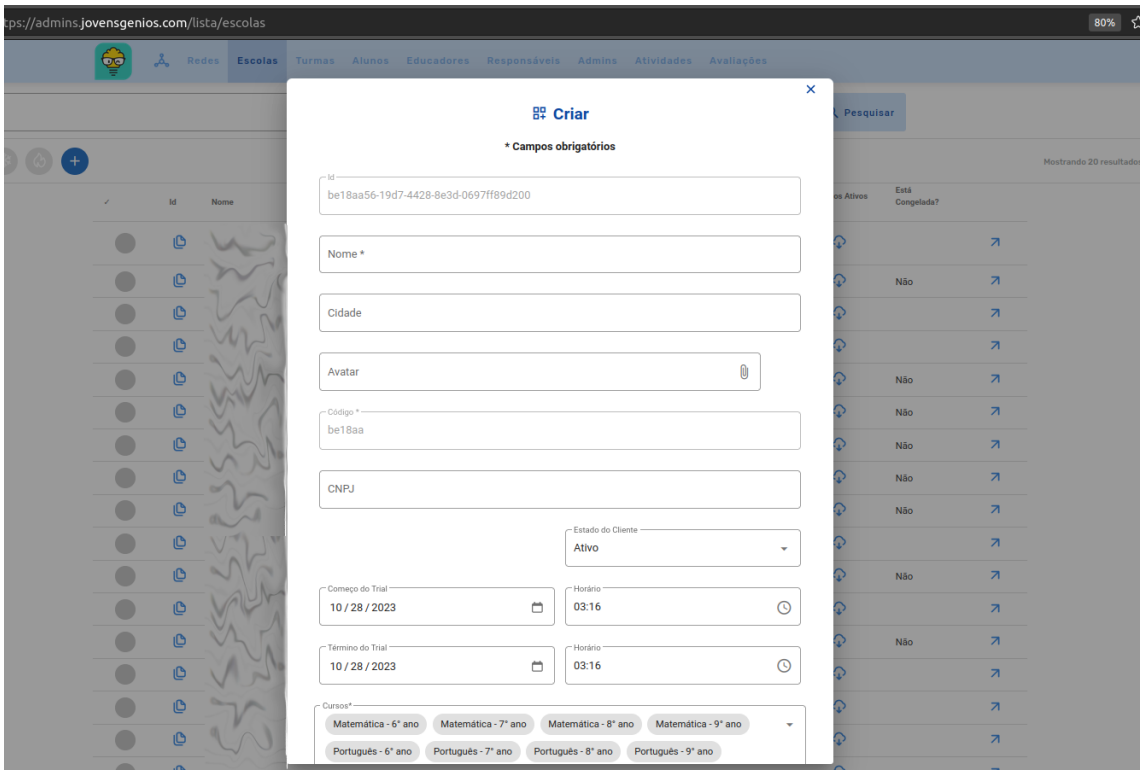

Figura 3.4: Formulário de Criação de nós do *tipo* Escola, na página de lista de Escolas.

```
19          "extra": "Corretas",
20          "color": "#5C9CE5"
21        }
22      ]
23      ...
24    },
```

o componente de busca e filtragem da página de listas também se beneficia do InstrospectionQuery, pois consegue listar de forma automática todas as possibilidades de uma certa propriedade ENUM em um seletor. Sabendo qual o tipo de cada propriedade, conseguimos realizar um render de um componente de input adequado para o usuário interagir.

3.1.3 Formulário reutilizável

Dentro do componente do formulário, um objeto chamado *properties* é populado com as informações resultantes do IntrospectionQuery, passando por uma filtragem

configurada no *utils*, que determina quais as propriedades que são editáveis. Para as restantes, dependendo de seu tipo, renderizamos um componente de interação específico no formulário.

```
1  <v-form>
2    <v-row no-gutters justify="end">
3      <v-col cols="12"
4        v-for="(property, i) in properties.strings"
5        :key="i"
6      >
7        <v-row no-gutters>
8          <v-col>
9            <v-text-field
10             v-model="property.value"
11             :label="`${ property.non_null ? translate(i) + ' *' :
                   translate(i)}`"
12             :disabled="property.disableEdit"
13             outlined
14           />
15         </v-col>
16       </v-row>
17     </v-col>
18     <v-col cols="12"
19       v-for="(property, w) in properties.dateTimes"
20       :key="w"
21     >
22       <date-time-picker
23         :label="`${ property.non_null ? translate(i) + ' *' :
              translate(i)}`"
24         :datetime="property.value"
25         @onChange="changedDateTime($event, w)"
26       />
27     </v-col>
28 ...
```

3.1.4 Ações

As ações de manipulação de dados disponibilizadas ao usuário através da interface foram classificadas em três categorias:

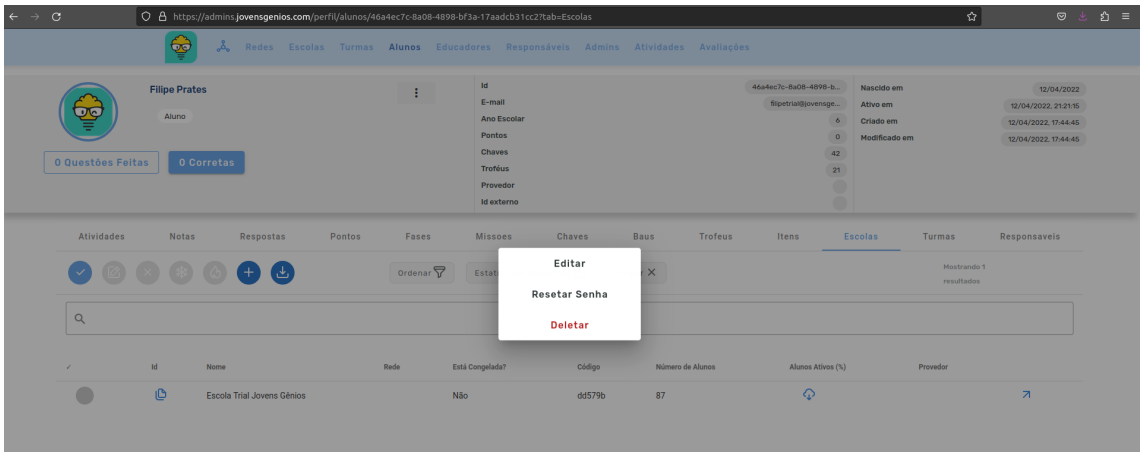

Figura 3.5: Interface das Ações de Perfil de um nó com *tipo* Aluno

- Ações de Perfil - Em toda página de perfil de um nó, é possível abrir executar ações diretas ao nó dono. Inclue a deleção, edição de suas propriedades, e opcionalmente outras ações personalizadas;

- Ações de Aba - Em cada aba dentro de um perfil, no cabeçalho de suas listas de elementos e com uma cor mais escura, existem botões quer permitem ações referentes às relações do nó dono com outros nós do *tipo* referente à aba selecionada. Inclue a criação de um novo nó do *tipo* da aba, já conectado ao dono, conectar o dono à um nó existente do *tipo* da Aba e opcionalmente outras ações personalizadas. Independem da seleção de elementos da lista;

- Ações de Elementos - Em cada aba dentro de um perfil, no cabeçalho de suas listas de elementos e com uma cor mais clara, é possível interagir com botões de ações referentes a elementos da lista selecionados. Inclue a edição do nó selecionado e a desconexão dele(s) ao dono, além de opcionalmente outras ações personalizadas. Dependem da seleção de elementos da lista.

Toda ação, independente da categoria, é propagada à camada Controle. Consigo carrega um objeto, diferentes dependendo da categoria da ação, que contém as informações que define a ação do usuário.

Na camada Controle, então, conseguimos escolher qual dos arquivos .gql armazenados que contêm a requisição a ser enviada ao servidor, ou montá-lo dinamicamente.

Mostramos abaixo um exemplo de função da camada de controle que lida com a ação de deletar múltiplos Alunos selecionados em uma lista de elementos.

```
1  // ex. @/graphql/Student/delete.gql
2  mutation DeleteStudent($Id: ID!) {
3    deleteStudents(where: { id: $Id }) {
4      nodesDeleted
5    }
6  }
7
8  // ex. control.js
9  async handleDelete(payload) {
10   for (let el of payload.elements) {
11     if (confirm(`Certeza que quer deletar ${el.name}`)) {
12         try {
13           await this.$apollo.mutate({
14             mutation: require(`@/graphql/${payload.type}/delete.gql
                   `),
15             variables: el,
16           });
17         } catch (e) {
18           console.error(e);
19           alert(e);
20           return;
21         }
22       }
23     }
24   }
```

3.2 Servidor com endpoint GraphQL

A interface, entretanto, não se comunica diretamente com o banco de dados. O servidor serve como uma camada intermediária que autentica o pedido e gera a Cypher resultante e envia um pedido de execução da mesma no banco de dados. Nesta seção descreveremos como funciona o servidor responsável por este processo.

3.2.1 Definições dos Tipos

Uma das principais funções do servidor é a definição de cada *tipo* existente no banco de dados. E, nessa etapa, equivalente à definição das tabelas e campos num banco de dados relacional, modelamos os dados do domínio e definimos cada propriedade possível de cada entidade (*tipo*) que poderá ser manipulada através sistema gerenciador.

As definições de *tipo* são extensas, divididas em arquivos que respeitam um escopo definido, tendo como função primordial a definição de propriedades e *tipos* de propriedades, além de suas possíveis relações. Também é nos arquivos de definição de tipos que Queries e Mutations customizadas, que não as CRUD geradas automaticamente pela biblioteca neo4j-graphql, podendo contér qualquer lógica necessária, incluindo comunicação com outro sistema.

A biblioteca neo4j-graphql utiliza-os para gerar as operações de CRUD (Create, Read, Update, Delete), e suas execuções gerenciam nós no banco de dados que recebem um rótulo de mesmo nome que o *tipo*.

Segue abaixo um exemplo simplificado de um arquivo de definição de tipos.

```
1  """
2  Battles types
3  """
4  type Battle implements Activity @node(labels: ["Battle", "Activity"
       ]) {
5    id: ID @id
6    status: ActivityStatus
7    createdAt: DateTime! @timestamp(operations: [CREATE])
8    updatedAt: DateTime! @timestamp(operations: [CREATE, UPDATE])
9    topic: Topic @relationship(type: "SELECTED_IN_BATTLE", direction:
         OUT)
10   task: Task @relationship(type: "BATTLE_IN_TASK", direction: OUT)
11     ...
12 }
13
14 enum ActivityStatus{
15     NOT_STARTED
16     IN_PROGRESS
17     DONE
```

```
}

type Mutation {
  mergeStudentsBattle(
    studentsIds: [ID!]
    battleId: ID!
    type: ContextType
  ): [ID]
  setBattleResult(battleId: ID!): String
  deleteBattle(battleId: ID!): ID
}
```

3.2.2 Resolvers

Podemos definir propriedades nas definições de *tipos* que não necessariamente estão armazenadas no banco de dados Neo4j. Um exemplo seria uma propriedade que retorna o ranking de uma Atividade, tal ranking não é armazenado no Neo4j, e sim em um banco de dados (Redis) em outro servidor, precisamos então de uma maneira de redirecionar certas requisições e determinar um fluxo lógico alternativo responsável por retornar o conteúdo daquela propriedade.

Os *resolvers* são funções que são utilizadas quando queremos alterar, ou até impedir o fluxo padrão de conexão direta com o Neo4j através da neo4j-graphql e quando queremos criar um fluxo novo independente paralelo. Podemos buscar em outros bancos de dados, ou realizar cyphers específicas, iniciar processos de atualizaçao de dados ou qualquer outra função necessária.

3.2.3 Conexão com o Banco de Dados

Utilizamos a biblioteca "neo4j-driver"que, após autenticação com usuário e senha, gera o objeto *driver*, abrindo uma conexão direta do servidor à instância de um banco Neo4j.

```
import neo4j from "neo4j-driver";

const driver = neo4j.driver(
  process.env.NEO4J_URI || "bolt://localhost:7687",
  neo4j.auth.basic(
```

```
6       process.env.NEO4J_USER || "neo4j",
7       process.env.NEO4J_PASSWORD || "neo4j"
8     ),
9     {
10      maxConnectionLifetime: 8 * 60 * 1000,
11      maxConnectionPoolSize: 50,
12      connectionAcquisitionTimeout: 2 * 60 * 1000,
13      disableLosslessIntegers: true,
14    }
15  );
```

3.2.4 Autenticação e Autorização

Após definir nossos *resolvers* e definições de *tipos*, nos preocupamos com a segurança, inpedindo que qualquer usuário conectado pudesse utilizar qualquer das mutations disponíveis pelo endpoint do servidor.

Utilizamos então o Neo4jGraphQLAuthJWTPlugin, que permite a autenticação das requisições através de uma token no campo de "Authorization"no header de cada requisição GraphQL posterior à de login. Tal token carrega consigo, de maneira criptografada, um indentificador do usuário que a gerou, o *tipo* deste usuário (Aluno, Professor, Admin, etc), e uma assinatura que permite verificação da autenticidade do mesmo.

Após a autenticação do token como válido no login ainda é necessário nas requisições posteriores, uma uma camada de autorização para permitir que o usuário em questão pode executar aquela requisição GraphQL específica. Não podemos permitir um Aluno executar a requisição de Deletar Escola. Para isso, a @neo4j/graphql disponibiliza de ferramentas de Role Based Access Control, onde definimos quais operações (CREATE, READ, UPDATE, DELETE) são permitidas para quais tipos de usuários em quais *tipos* de dados, podendo utilizar a informação do indentificador do usuário contido no token para uma maior granularidade.

Tais regras são críticas para a segurança do sistema, já que a toda a informação sobre o schema é aberta à clientes através da IntrospectionQuery e o GraphQL permite alterações genéricas nos dados.

```
1       export function isRequestAllowed(req) {
```

```
2       const token = req.headers?.authorization?.trim().split(" ")
            [1];
3       if (parseJwt(token)?.userType === "ExternalAdmin") {
4         const operationName = req.body?.query
5           .match(/(?<={)\s*[a-zA-Z0-9]*/)?.[0]
6           .trim();
7         return EXTERNAL_ADMIN_ALLOWED_REQUESTS.includes(
              operationName);
8   }
9   return true;
10 }
```

3.2.5 Geração do Schema

O *schema* para o banco de dados Neo4j é gerado então após um objeto Neo4jGraphQL(), da @neo4j/graphql, ser instanciado. Ele recebe como argumentos o *driver* conectado ao banco de dados, as definições de *tipos*, os *resolvers* e eventuais plugins utilizados.

```
1  import { Neo4jGraphQL } from "@neo4j/graphql";
2  import { Neo4jGraphQLAuthJWTPlugin } from "@neo4j/graphql-plugin-
       auth";
3  import { resolvers } from "./resolvers";
4  import { typeDefs } from "./typeDefinitions";
5
6  async function initializeNeo4jGraphQL() {
7    const neo4jGraphQL = new Neo4jGraphQL({
8      driver,
9      typeDefs: gql`${typeDefs}`,
10     resolvers,
11     plugins: {
12       auth: new Neo4jGraphQLAuthJWTPlugin({
13         secret: process.env.JWT_SECRET,
14         algorithms: ["HS256"],
15         credentialsRequired: false,
16       }),
17       config: {
18         enableDebug: process.env.NODE_APP_ENV === "staging",
19       },
```

```
20      },
21    });
22    return await neo4jGraphQL.getSchema();
23  }
```

Em seguida o Apollo Server é inicializado com a informação este *schema* e utilizamos a biblioteca express para publica-lo nas portas definidas.

```
1  import { expressMiddleware } from "@apollo/server/express4";
2      const app = express();
3    const apolloServer = new ApolloServer({
4      schema: schema,
5      introspection: true,
6      plugins: [
7              ...
8      ],
9    });
10
11   app.use(
12     path,
13     expressMiddleware(apolloServer, {
14       context: ({ req }) => ({
15         driver,
16         neo4jDatabase: process.env.NEO4J_DATABASE,
17         req
18       }),
19     })
20   );
```

Capítulo 4

Conclusões

4.1 Resultados

O sistema desenvolvido e descrito neste trabalho se demonstrou uma interface de gerenciamento de dados útil e eficiente, sendo utilizado diariamente desde 2020 por múltiplos usuários de diversas áreas da empresa para gerenciar dados armazenados em uma banco de dados Neo4j.

A abstração do grafo em "Perfis e Listas"se mostrou uma maneira intuitiva de navegar e interagir com os dados. A possibilidade de realizar uma mesma ação por diferentes caminhos permite rapidez e facilidade na realização de ações de manipulação.

Na área de Suporte, como exemplo, apenas um treinamento de duas horas de duração no processo de onboarding é suficiente para novos trabalhadores entenderem e utilizarem o sistema. Mesmo sem um conhecimento prévio da modelagem dos dados existentes, usuários começam a utilizar o sistema como ferramenta de trabalho, com supervisão, no mesmo dia que tem o contato inicial com a interface.

4.2 Trabalhos Futuros

Dentre as possíveis implementações adicionais, podemos citar:

- Gerar automaticamente os arquivos que definem as requisições .gql com as informações retornadas pela InstrospectionQuery. A configuração constante das definições de *tipos* foi um ponto de dificuldade para o desenvolvimento e

manutenção do sistema.

- Desenvolver em "Perfil e Listas"para gerenciamento de dados em grafo cega ao servidor. Uma interface capaz de conectar à qualquer endpoint GraphQL com IntrospectionQuery aberta, e permitir, após autenticação, pedidos de visualização e alteração arbitrárias dos dados.

Referências Bibliográficas

INC., N. “Neo4j Graph Database: High-speed graph database with unbounded scale, security, and data integrity for mission-critical intelligent applications.” 2024a. Disponível em: <https://neo4j.com/product/neo4j-graph-database/>.

INC., N. “Query a Neo4j database using Cypher”. 2024b. Disponível em: <https://neo4j.com/docs/getting-started/cypher-intro/>.

FOUNDATION, T. G. “A query language for your API”. 2024a. Disponível em: <https://graphql.org/>.

INC., N. “A GraphQL to Cypher query execution layer for Neo4j and JavaScript GraphQL implementations.” 2024c. Disponível em: <https://github.com/neo4j/graphql>.

INC., A. G. “Streamlining APIs, Databases, & Microservices: Apollo GraphQL”. 2024d. Disponível em: <https://www.apollographql.com/>.

FOUNDATION, O. “Node.js® is an open-source, cross-platform JavaScript runtime environment”. 2024b. Disponível em: <https://nodejs.org/en>.

FOUNDATION, O. “Fast, unopinionated, minimalist web framework for Node.js”. 2024c. Disponível em: <https://expressjs.com>.

VUEJS. “Vue.js - The Progressive JavaScript Framework”. 2024. Disponível em: <https://vuejs.org/>.