UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”

ACADEMIA POKÉMON

Documentação da API

Carlos Roberto Nascimento Junior 181022605

Giovana Ballaminut Simioni 181024731

Leandro Adriano dos Santos 181020262

Lucas Pederzini Silva 181024578

BAURU 2020

RESUMO

O sistema criado neste projeto visa simular uma academia para pokémons. A principal linguagem utilizada no projeto foi Javascript para criar todo o sistema e o PostgreSQL como banco de dados. A academia é um sistema que permite a criação e manipulação de usuários e funcionários, que são separados hierarquicamente.

O processo de treinamento é definido por tempo e separado por departamento, até que o nível objetivo seja alcançado. A administração da academia é feita por um usuário que possui acesso a todas as funções do sistema.

SUMÁRIO

1 INTRODUÇÃO

Este relatório tem como objetivo documentar os passos da criação de uma interface de programação de aplicativos a fim de elucidar a implementação do código fonte. Além disso, cada segmento da interface da aplicação será comentada neste documento para que seja possível, de uma maneira mais clara, a sua visualização e funcionamento.

Desta forma, este documento será organizado em seções com características em comuns. Assim, serão 5 seções para esclarecimento e uso da interface de programação de aplicativos.

2 A ESCOLHA DO BANCO

A equipe optou pelo sistema gerenciador de banco de dados PostgreSQL, cujo projeto é uma das plataformas mais estáveis e avançadas de banco de dados de código aberto. Ademais, nós também escolhemos um serviço *on-line* que oferece a hospedagem gratuita de um banco em PostgreSQL. O serviço permite a criação de um banco com um limite de 20mb de dados e 5 conexões simultâneas.

A nossa escolha para um banco como serviço foi tomada a fim de que o grupo pudesse, em conjunto, compartilhar o mesmo banco no momento da programação da aplicação desenvolvida. Desta forma, a escolha baseou-se neste contexto da programação compartilhada e do oferecimento de uma interface web para consultas.

2.1 A criação das tabelas

O banco foi criado com o auxílio da interface web do ElephantSQL. Assim, o grupo estabeleceu um banco gratuito neste serviço. Após a inicialização do banco e fornecimento da chave de acesso para uma interface de programação de aplicativos, a equipe criou as tabelas colocadas abaixo.

```
CREATE TABLE Usuario (
    cpf varchar(24),
    nome varchar(64) not null,
    rg varchar(24) not null unique,
    data_nascimento date,
    rua varchar(64),
    num_casa integer,
    bairro varchar(64),
    cidade varchar(64),
    estado varchar(2) check(estado in ('sp', 'rj', 'mg')),
    cep integer,
    e_mail varchar(64) NOT NULL UNIQUE,
    password varchar(256) NOT NULL,
    num_cartao varchar(64),
    data_vencimento varchar(64),
```

```
  nome_cartao varchar(64),
  cod_cartao varchar(5),
  data_cadastro TIMESTAMP,
  PRIMARY KEY (cpf)
);

CREATE TABLE Plano (
   cpf varchar(24) REFERENCES Usuario (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
   codigo_plano integer,
   valor varchar(24),
   data_de_inicio TIMESTAMP,
   duracao integer,
   PRIMARY KEY  (cpf)
);

CREATE TABLE Telefone (
   cpf varchar(24) REFERENCES Usuario (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
   numero_de_telefone varchar(12),
   PRIMARY KEY (cpf, numero_de_telefone)
);

CREATE TABLE Pokemon (
   codigo_pokemon SERIAL,
   cpf varchar(24) REFERENCES Usuario (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
   nome varchar(64),
   raca varchar(64),
   classificacao varchar(64),
   nivel integer NOT NULL,
   nivel_objetivo integer NOT NULL,
   data_de_entrada TIMESTAMP,
   data_de_saida TIMESTAMP,
   data_cadastro TIMESTAMP,
   PRIMARY KEY (codigo_pokemon)
);

CREATE TABLE Treinador (
  cpf varchar(24) REFERENCES Usuario (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
  cpts varchar(24) NOT NULL UNIQUE,
  salario_base varchar(24),
  instituto varchar(64),
  data_cadastro TIMESTAMP,
  PRIMARY KEY (cpf)
```

```
);


CREATE TABLE Especialidade (
  cpf varchar(24) REFERENCES Treinador(cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
  especialidade varchar(64),
  PRIMARY KEY (cpf, especialidade)
);


CREATE TABLE Mestre (
  cpf varchar(24) REFERENCES Treinador (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
  data_cadastro TIMESTAMP,
  PRIMARY KEY (cpf)
);


CREATE TABLE Proficiencia (
  cpf varchar(24) REFERENCES Mestre (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
  proficiencia varchar(64),
  PRIMARY KEY (cpf, proficiencia)
);


CREATE TABLE Departamento (
  codigo_dept integer,
  nome_dept varchar(64) UNIQUE,
  classificacao varchar(64),
  gerente varchar(24) REFERENCES Mestre (cpf) ON DELETE SET NULL
ON UPDATE CASCADE,
  PRIMARY KEY (codigo_dept)
);


CREATE TABLE Aprimora (
  codigo_pokemon integer REFERENCES Pokemon (codigo_pokemon) ON
DELETE CASCADE ON UPDATE CASCADE,
  cpf varchar(24) REFERENCES Treinador (cpf) ON DELETE SET NULL ON
UPDATE CASCADE,
  hora_de_entrada TIMESTAMP,
  hora_de_saida TIMESTAMP,
  PRIMARY KEY (codigo_pokemon, cpf, hora_de_entrada, hora_de_saida)
);
```

```
CREATE TABLE Trabalha (
  codigo_dept integer REFERENCES Departamento (codigo_dept) ON
DELETE CASCADE ON UPDATE CASCADE,
  cpf varchar(24) REFERENCES Treinador (cpf) ON DELETE CASCADE ON
UPDATE CASCADE,
  PRIMARY KEY (cpf, codigo_dept)
);
```

2.2 Controle de integridade

O único gatilho necessário para este banco foi criado durante a inserção de dados na tabela plano a fim de inicializar um novo usuário com um plano gratuito no momento de sua criação.

```
CREATE OR REPLACE FUNCTION add_plan()
RETURNS trigger AS
$BODY$
    BEGIN
        INSERT INTO Plano (cpf, codigo_plano, valor, data_de_inicio, duracao)
VALUES (new.cpf, 0, '0', new.data_cadastro, 999);
        RETURN NEW;
    END;
$BODY$
LANGUAGE 'plpgsql';


CREATE TRIGGER addPlanBasic
AFTER INSERT ON Usuario
FOR EACH ROW
EXECUTE PROCEDURE add_plan();
```

Além disso, cada chave estrangeira possui um atributo de cascata para quando eles forem deletados ou modificados. Desta forma, por exemplo, ao alterar ou apagar um usuário, qualquer informação relacionada a este usuário é apagada de todas as outras tabelas no qual esta informação é utilizada.

3 A INTERFACE DE PROGRAMAÇÃO DE APLICATIVOS WEB

O nosso projeto foi desenvolvido com o uso das ferramentas NodeJS, Npm e PostgreSQL - com o uso do ElephantSQL como serviço. O servidor para fornecer os dados para a nossa interface web foi construída com o auxílio do ExpressJS, uma ferramenta para gerenciar as rotas da aplicação. Nas subseções seguintes, este documenta detalhará a rota que foi criada para o usuário e seu funcionamento.

O arquivo index.js dentro da pasta backend/src é utilizado pelo ExpressJS para gerenciar as rotas. Cada rota é definida dentro da pasta /routes e importada para o index.js. Caso o usuário tente acessar uma rota não especificada, o servidor envia um erro de que a rota não foi encontrada.

3.1 A conexão com o banco

A conexão com o banco é feita com o uso de uma chave, fornecida pela aplicação do ElephantSQL, para acessar a interface de aplicação do banco de dados. Dentro da pasta backend/src/db, temos um arquivo com o nome elephant-sql.js, cuja função é realizar a conexão com o banco.

Por meio da biblioteca PG, uma chave é fornecida como string para inicializar a conexão. Podemos observar pelo trecho do código abaixo onde a chave é aplicada.

```
const connectionString = process.env.PG_KEY
```

O process.env.PG_KEY é uma variável de ambiente, que está determinada dentro da pasta backend/config no arquivo dev.env.

3.2 Os modelos da API

Os modelos são um objeto do Javascript com duas propriedades principais, a chave *text* associada ao valor da consulta que será realizada e o valor *values* com todos os valores que serão enviados juntos com aquela consulta.

```
const queryInsert = {
    text: 'INSERT INTO Aprimora (codigo_pokemon, cpf, hora_de_entrada, hora_de_saida) VALUES ($1, $2, $3, $4)'
}
```

No exemplo acima, temos uma consulta de inserção na tabela Aprimora. O *text* é a consulta *INSERT INTO*, os valores estão definidos como $1, $2, $3 e $4. Assim, na rota da tabela Aprimora, antes de enviar a consulta ao banco, o servidor adiciona uma nova propriedade a este objeto. Esta propriedade adicionada é o *values*, que conterá os valores para inserção.

3.3 Rotas

As rotas são definidas de acordo com as tabelas criadas, com exceção da rota admin. Todas as rotas terão algumas constantes estabelecidas, como:

```
const express = require('express')
const pool = require('../db/elephant-sql')
const router = new express.Router()
const auth = require('../middlewares/auth')
```

O express é o gerenciador global das rotas, estas rotas estão dentro da pasta backend/src/routes. Cada arquivo .js é uma rota diferente, por exemplo, a rota /admin quando acessada leva a execução de um script, que está dentro do arquivo admin.js.

O pool é uma constante para realizar a consulta SQL, o seu código está definido dentro da pasta backend/src/db, no arquivo elephant-sql.js. Esta constante é uma extensão do valor recebido do envio da chave fornecida pelo ElephantSQL para todos os arquivos de rota. Portanto, a sua função é permitir o envio da consulta ao banco.

O router é utilizado pelo express para gerenciar múltiplos arquivos .js de rotas diferentes. Cada rota é importada no arquivo index.js na pasta backend/src. Desta

forma, todas as rotas criadas são conhecidas pelo ExpressJS e podem ser acessadas.

A constante auth serve para autenticar determinado usuário com o banco, ou seja, só será permitida a realização de alguma consulta se o usuário estiver cadastrado no banco de dados.

3.3.1 Exemplo da rota Usuário

Cada rota tem uma constante com a importação dos objetos presentes nos modelos, esta importação serve para utilizar tais objetos dentro do arquivo da rota.

```
const {
        queryInsertUser,
        queryFindByCpf,
        queryDeleteByCpf,
        queryFindByEmail
      }=
require('../models/user')
```

No exemplo acima, cada propriedade do objeto é uma referência do que está escrito no arquivo do modelo daquela rota.

Os métodos de uma API Restful

Cada rota possui pelo menos quatro métodos básicos, sendo os caminhos de *GET, POST,* UPDATE e *DELETE.*

O método GET do Usuário

O método *GET* - específico do usuário - no diretório backend/src/routes/user.js no percurso /user/me realiza a consulta na tabela Usuário por um determinado CPF fornecido na requisição da aplicação.

Todas as rotas são declaradas privadas com a passagem por parâmetro da função auth, visto que esta função busca no banco o CPF de determinado usuário e retorna os seus dados, confirmando a sua autenticação.

```
const queryFindByCpf = {
  text: 'SELECT * FROM Usuario WHERE cpf = $1'
}
```

Desta maneira, o método GET da rota Usuário envia uma resposta com os dados encontrados, caso contrário, o servidor envia um erro com mensagem de que não foi encontrado nenhum usuário com aquele CPF.

O método POST do Usuário

Na rota /user, o método POST realiza a inserção de determinado usuário no sistema, esta rota, como a rota do login, são públicas, ou seja, qualquer pessoa pode acessá-las. Porém, antes, de fato, enviar a consulta ao banco, o servidor realiza duas verificações para validar os dados sobre o CPF e o e-mail do usuário. Caso estes dados informados não estejam no padrão desejado, o servidor envia um erro como resposta.

Uma etapa de segurança é adicionada após estas duas verificações. O servidor criptografa a senha enviada na requisição com o uso da biblioteca Bcrypt. Depois disto, a propriedade *values* do objeto do modelo daquela rota é definido.

```
queryInsertUser.values = await toArr(req.body)
```

O *values* será um vetor com os valores já formatados, de forma a seguir a ordem imposta no modelo. Por exemplo, teremos um vetor com os valores de: [‘40960324097’, ‘nilceu’, ‘01471421’, ‘2020-06-22’, ‘rua dos travesseiros’, ‘124’, ‘vila das gaivotas’, ‘bauru’, ‘sp’, ‘14210510’, ‘nilceu@email.com’, ‘senha123’, ‘2020-06-22’

]. O último valor é a data de cadastro, cuja data é do sistema no momento da criação do usuário.

```
const newUser = await pool.query(queryInsertUser)
```

Após isto, a consulta é enviada ao banco por meio de uma requisição assíncrona, isto é, o servidor espera o banco responder com um sucesso ou erro. O parâmetro enviado dentro da função pool.query() é o objeto determinado pelo modelo daquela rota.

Neste exemplo, o queryInsertUser possui o valor do *text* abaixo com os valores recebidos da requisição:

```
const queryInsertUser = {
    text: 'INSERT INTO Usuario (cpf, nome, rg, data_nascimento, rua, num_casa, bairro, cidade, estado, cep, e_mail, password, data_cadastro) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13)',
    values: [‘40960324097’, ‘nilceu’, ‘01471421’, ‘2020-06-22’, ‘rua dos ztravesseiros’, ‘124’, ‘vila das gaivotas’, ‘bauru’, ‘sp’, ‘14210510’, ‘nilceu@email.com’, ‘senha123’, ‘2020-06-22’ ]

}
```

Caso dê algum erro, o servidor informará sobre tal evento. Entretanto, caso haja um sucesso, o servidor enviará os dados deste novo usuário de volta como resposta.

O método DELETE do Usuário

Na rota /user/me, o método DELETE é o responsável por apagar um usuário, utilizando sua PRIMARY KEY, neste caso o CPF. Uma requisição é enviada ao banco de dados com o CPF do usuário logado para verificar se o mesmo possui

autorização para realizar esta ação. Caso não possua, uma mensagem de erro de autenticação é retornada do servidor.

Após a primeira checagem, é verificado se o CPF em questão existe no banco de dados, caso ele não seja encontrado, um erro é retornado pelo servidor. No exemplo a seguir, o queryDeleteByCpf possui o número do CPF que deve ser apagado do banco de dados.

```
const queryDeleteByCpf = {
  text: 'DELETE FROM Usuario WHERE cpf = $1'
  values: [‘00100200304’]
}
```

O método PATCH do Usuário

O método PATCH utiliza uma função auxiliar - searchByKeyAndUpdate - localizada na pasta backend/src/utils no arquivo update.js para montar uma string a ser enviada ao servidor. Por exemplo, o usuário deseja alterar o seu nome.

Então, o novo valor para realizar a mudança é enviado na requisição, o servidor chama a função searchByKeyAndUpdate e concatena os valores recebidos em uma string com a consulta de *UPDATE*. Mais informações abaixo na descrição da função.

Rotas e métodos generalizados

Todas as outras rotas seguem os mesmos padrões acima descritos. Então, qualquer rota de qualquer tabela terá sempre, pelo menos, um caminho para executar a inserção, atualização, remoção e visualização, seguindo suas próprias particularidades.

Os casos de SELECT especiais

Algumas rotas de determinadas tabelas possuem um método GET com ordenação e paginação. Desta forma, as consultas destas rotas foram feitas de acordo com a página atual de visualização e a coluna que será usada para tal ordem.

```
SELECT * FROM Pokemon
WHERE cpf = '${req.user.cpf}'
ORDER BY ${parts[0]} ${parts[1]} LIMIT ${req.query.limit}0
```

O select acima está em alguns métodos de algumas rotas. No caso da tabela Pokémon, ele retorna todos os valores da tabela que possuem determinado CPF. Além disso, este resultado é retornado ordenado de acordo com o valor da coluna e da página atual de visualização recebido pelo servidor. Também é oferecida a possibilidade de ordem crescente e decrescente.

As variáveis parts 0 e 1 são os valores recebidos pelo servidor durante a requisição, assim como o valor da req.query.limit. Por exemplo, caso a requisição envie como valor raça, asc e 1, o servidor realizará as consultas com estes valores como base para a realização da consulta.

Da mesma forma do select acima descrito, existem alguns casos especiais como nos exemplos abaixo:

```
SELECT * FROM Pokemon
WHERE cpf = $1
ORDER BY data_cadastro DESC LIMIT 3
```

Este primeiro retorna os três primeiros pokémons ordenados pela data de cadastro para ser mostrado na tela inicial da interface gráfica.

```
SELECT classificacao, count(classificacao) as quantidade
FROM Pokemon
WHERE cpf = $1
GROUP BY classificacao
```

Este *SELECT* retorna os pokémons de determinado CPF agrupados de acordo a classificação a fim de que estes sejam mostrados no gráfico da tela inicial.

```
SELECT
        extract(year from data_cadastro) as year,
        extract(month from data_cadastro) as month,
        extract(day from data_cadastro) as day,
        count(codigo_pokemon) as count
FROM Pokemon
WHERE cpf = $1 AND data_cadastro >= $2 GROUP BY 1, 2, 3
```

O terceiro *SELECT* retorna os pokémons de determinado CPF e acima de uma data enviada pelo servidor agrupados de acordo com o ano, mês e dia a fim de serem colocados no gráfico de dias da tela inicial.

3.3.2 Exemplo da rota Admin

A rota Admin só possui um caminho disponível, o /admin, cuja função é executar as operações de consulta de forma direta, isto é, o usuário poderá digitar os comandos do SQL diretamente na interface e ter um retorno dos valores enviados. Cada consulta é separada por um ponto e vírgula, e desta maneira, é permitido o envio de múltiplos comandos ao mesmo tempo ao banco de dados.

3.4 A função *searchByKeyAndUpdate*

Esta função tem como objetivo auxiliar as rotas *PATCH* da aplicação, de forma a simplificar em único arquivo a concatenação dos valores a serem atualizados.

Desta forma, a função busca uma coluna, de acordo com os valores enviados para busca e altera o único valor que foi modificado, que será passado por parâmetro. Além disso, esta função diferencia os valores inteiros e não inteiros de acordo com o vetor de tipo numérico enviado como parâmetro.

Os seus parâmetros são definidos segundo o código abaixo:

```
searchByKeyAndUpdate(
    OBJECT          corpo da requisição          (req.body),
    STRING          nome da tabela               ('Pokemon'),
    ARRAY           nomes das colunas para busca (['cpf']),
    ARRAY           valores das colunas para busca (['122455'])
    OBJECT          modelo do objeto de UPDATE   (queryFindByCpf),
    ARRAY           alterações permitidas        (['nome', 'rua', 'cep']),
    ARRAY           colunas com tipo númerico    (['num_cara', 'cep'])
)
```

*O último array é opcional.

Por exemplo, a função concatena uma string como a seguinte:

```
UPDATE Table SET column1 = value1, column2 = ‘value2’, … WHERE key = '123'
```

onde:

Table ➜ a tabela a ser modificado

column1 = value1 ➜ um conjunto chave e valor a ser modificado, que pode ser variável

key ➜ valor a ser buscado, que também pode ser variável

4 A INTERFACE GRÁFICA

A interface gráfica do projeto foi desenvolvida com o uso do *framework* ReactJS. Assim, para consumir as informações da nossa API foi construída um sistema Web a fim de que os dados do banco sejam visualizados de modo simples e intuitivo a usuários leigos, sem precisar digitar os comandos específicos de SQL.

Entretanto, caso o usuário seja avançado e tenha permissão de mestre, ele ainda poderá realizar consultas diretamente no banco com a interface gráfica dentro do sistema Web, que permite a realização de múltiplas consultas personalizadas.

O sistema Web é intuitivo, sem a necessidade de explicação detalhado do seu funcionamento. Basta acessar o link abaixo para visualizar este projeto hospedado em um servidor. Tanto o servidor, quanto a interface estão online.

Link da API: http://carona-backend-academia.herokuapp.com/

Link da interface Web: https://frontend-pokemon.now.sh/

5 COMO RODAR A APLICAÇÃO

Antes de tudo, é necessário a instalação do NodeJS na versão 13.x. Após a sua instalação, siga os passos enunciados abaixo para executar uma instância de um servidor do *backend* e outra da interface gráfica web.

5.1 A interface de programação de aplicativos (API)

1. Caso a pasta já venha com a pasta config, pule esta etapa. Crie uma pasta config dentro da pasta backend. Dentro da pasta config, faça um arquivo com o nome dev-env e coloque dentro dele o seguinte código:

```
PORT=3000
PG_KEY=ENDEREÇO-DO-BANCO-DE-DADOS-AQUI
```

2. Feito a configuração inicial, instale os pacotes com o comando abaixo dentro da pasta backend:

```
npm install
```

3. Execute o servidor com o script no modo de desenvolvimento para iniciar um servidor local:

```
npm run dev
```

5.2 O sistema Web

1. Dentro da pasta frontend, instale os pacotes utilizando o comando abaixo:

```
npm install
```

2. Rode o aplicativo em modo desenvolvimento.

```
npm start
```

3. Abra o endereço http://localhost:3000 para visualizar o site Web.