REPÚBLICA FEDERATIVA DO BRASIL
MINISTÉRIO DA ECONOMIA
INSTITUTO NACIONAL DA PROPRIEDADE INDUSTRIAL
DIRETORIA DE PATENTES, PROGRAMAS DE COMPUTADOR E TOPOGRAFIAS DE CIRCUITOS INTEGRADOS

Certificado de Registro de Programa de Computador

Processo Nº: **BR512022001662-1**

O Instituto Nacional da Propriedade Industrial expede o presente certificado de registro de programa de computador, válido por 50 anos a partir de 1° de janeiro subsequente à data de 29/03/2022, em conformidade com o §2°, art. 2° da Lei 9.609, de 19 de Fevereiro de 1998.

Título: SIGEE: SISTEMA DE GESTÃO DE ESTÁGIOS

Data de publicação: 29/03/2022

Data de criação: 25/03/2022

Titular(es): FUNDAÇÃO UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE - FUERN

Autor(es): SEBASTIÃO EMIDIO ALVES FILHO; GABRIEL NASCIMENTO JALES; JONAS SILVA RODRIGUES; JEFFERSON XIMENES ROCHA DE SOUSA; RAFAEL DA SILVA XIMENES

Linguagem: JAVA SCRIPT

Campo de aplicação: AD-01; IF-02; IF-10

Tipo de programa: AP-01; FA-01; GI-01

Algoritmo hash: SHA-512

Resumo digital hash:
616861684bfc34d8519e364fa5831e3b945cd3a069367605bf716d058e25b8f0ad98ecee3d606dd3dc94b77a6ef749a0560386f732a9c7d376f21e390bb4956a

Expedido em: 12/07/2022

Aprovado por:
Joelson Gomes Pequeno
Chefe Substituto da DIPTO - PORTARIA/INPI/DIRPA Nº 02, DE 10 DE FEVEREIRO DE 2021

UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE – UERN
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS – FANAT
DEPARTAMENTO DE INFORMÁTICA – DI

JONAS SILVA RODRIGUES

SIGEE PARTE III: UMA PERSPECTIVA DO BANCO DE DADOS

MOSSORÓ - RN
2023

JONAS SILVA RODRIGUES

SIGEE PARTE III: UMA PERSPECTIVA DO BANCO DE DADOS

Relatório apresentado ao curso de Ciência da Computação da Universidade do Estado do Rio Grande no Norte como requisito da disciplina de Trabalho de Diplomação, sob a orientação do Prof. Dr. Sebastião Emidio Alves Filho.

MOSSORÓ - RN
2023

SUMÁRIO

1 INTRODUÇÃO

O estágio é uma etapa educacional essencial e orientada, realizada no ambiente de trabalho, com o propósito de preparar os estudantes para ingressarem no mercado profissional. Seu foco é permitir que os alunos adquiram as habilidades específicas necessárias para sua futura carreira, estabelecendo uma ligação prática entre o conhecimento teórico e a realidade do campo de atuação.

Nesse processo, estão envolvidos três principais agentes: o estudante, a concedente e a instituição de ensino, cada um desempenhando papéis complementares e distintos. O estudante, como protagonista, busca aplicar na prática os conhecimentos adquiridos ao longo do curso, vivenciando situações reais.

A parte concedente, representada pela instituição ou empresa que abre suas portas ao estudante, é responsável por oferecer atividades práticas alinhadas ao desenvolvimento educativo do aluno. Isso proporciona uma oportunidade para o aluno se familiarizar com os desafios do cenário profissional.

Por último, a instituição de ensino, frequentemente chamada de proponente, é responsável por incorporar o estágio em seu currículo acadêmico. Ela supervisiona e avalia o desempenho do aluno durante o período de estágio, assegurando uma experiência de aprendizado abrangente e eficaz.

Com esta descrição, percebe-se que além da operação pedagógica própria do estágio, existe um processo administrativo que regula e rege a relação entre estudantes, instituições de ensino e concedentes. Para gerenciar os dados dos estagiários e permitir que o sistema acesse os dados de maneira segura, confiável e com alta disponibilidade, optou-se por desenvolver um sistema com o banco de dados utilizando o PostgreSQL. Então, o Sistema de Gestão de Estágios (SIGEE) tem como objetivo principal fornecer um sistema seguro e robusto para armazenar os dados de estágios de uma instituição, além de providenciar uma interface simples e de fácil navegação para que a gestão de estágios ocorra sem complicações.

2 OBJETIVOS

O objetivo geral deste trabalho é abordar sobre a construção do SIGEE na perspectiva do banco de dados, ou seja apresentar o banco de dados do sistema, como ele foi construído e quais tecnologias foram utilizadas para fazer a ligação do banco de dados com a aplicação. Ademais, podem ser citados como objetivos específicos: a utilização da metodologia ágil Scrum durante o desenvolvimento do projeto, a utilização do ORM (Object-relational mapping - Mapeamento objeto-relacional) Sequelize que abstrai os comandos de operações SQL, e faz com que seja possível usar uma linguagem de programação que já está sendo usada no back-end para se conectar e operar o banco de dados.

3 METODOLOGIA

A metodologia utilizada para o desenvolvimento do SIGEE foi a abordagem ágil Scrum, que possui três fases: planejamento geral, ciclo de sprints e o encerramento do projeto (SOMMERVILLE, 2011). Na primeira fase, foram estabelecidos os principais objetivos do projeto, as tecnologias que seriam utilizadas, o backlog do sistema listando todo o trabalho a ser feito e a duração fixa dos ciclos de sprints, definida em quinze dias. Na segunda fase, cada membro da equipe trabalhava em uma nova funcionalidade para ser incrementada no sistema, com reuniões diárias para analisar os progressos e dificuldades, além de reuniões a cada quinze dias para finalizar a sprint. Por fim, na última etapa encerrou-se o desenvolvimento do projeto e foram feitas avaliações sobre as lições aprendidas no projeto.

Para facilitar o gerenciamento e divisão das tarefas que precisavam ser feitas e manter a equipe alinhada, foi utilizada a ferramenta web de gerenciamento de projetos Trello. No Trello os projetos são organizados em quadros, nesses quadros são definidas listas com todas as tarefas do projeto e por sua vez, essas tarefas podem ser atribuídas a membros do quadro (TRELLO, 2023). A figura 1 a seguir mostra a organização dos quadros e tarefas no Trello

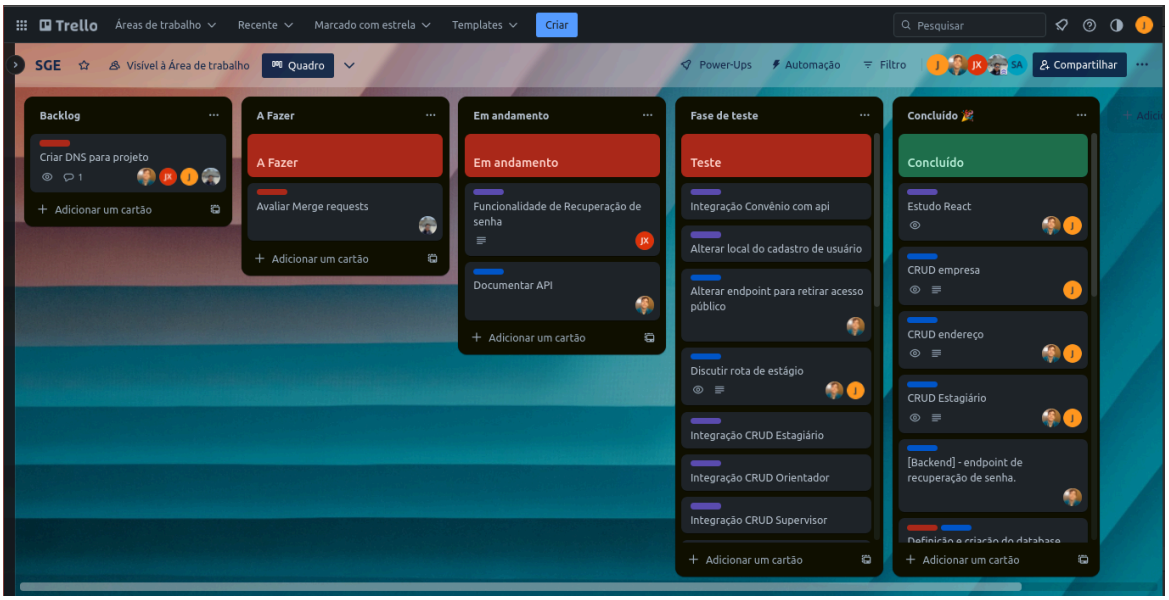

Figura 1: Ilustrando o funcionamento do trello. Autoria própria

4 DESCRIÇÃO DO PROJETO

O SIGEE trata-se de um sistema WEB criado para a gestão de dados relacionados a estágios, onde um usuário cadastrado e que foi autenticado pode realizar operações de registro, consulta, atualização e remoção de informações como dados de estágios, estagiários, cursos, orientadores acadêmicos, supervisores de estágio, empresas e convênios.

O sistema é composto por duas aplicações principais, o back-end e o front-end. De forma simplificada, é possível entender o back-end responsável pela regra de negócio do projeto e o front-end seria a camada responsável por interagir com o usuário, sendo a interface gráfica da aplicação.

A aplicação utiliza o modelo cliente-servidor, ou seja, o cliente (front-end) envia requisições ao servidor (back-end) das informações desejadas e o servidor envia uma resposta com as informações, possibilitando ao usuário realizar operações de CRUD (Create, Read, Update, Delete) com os dados do estágio, como demonstrado na figura 2 a seguir.

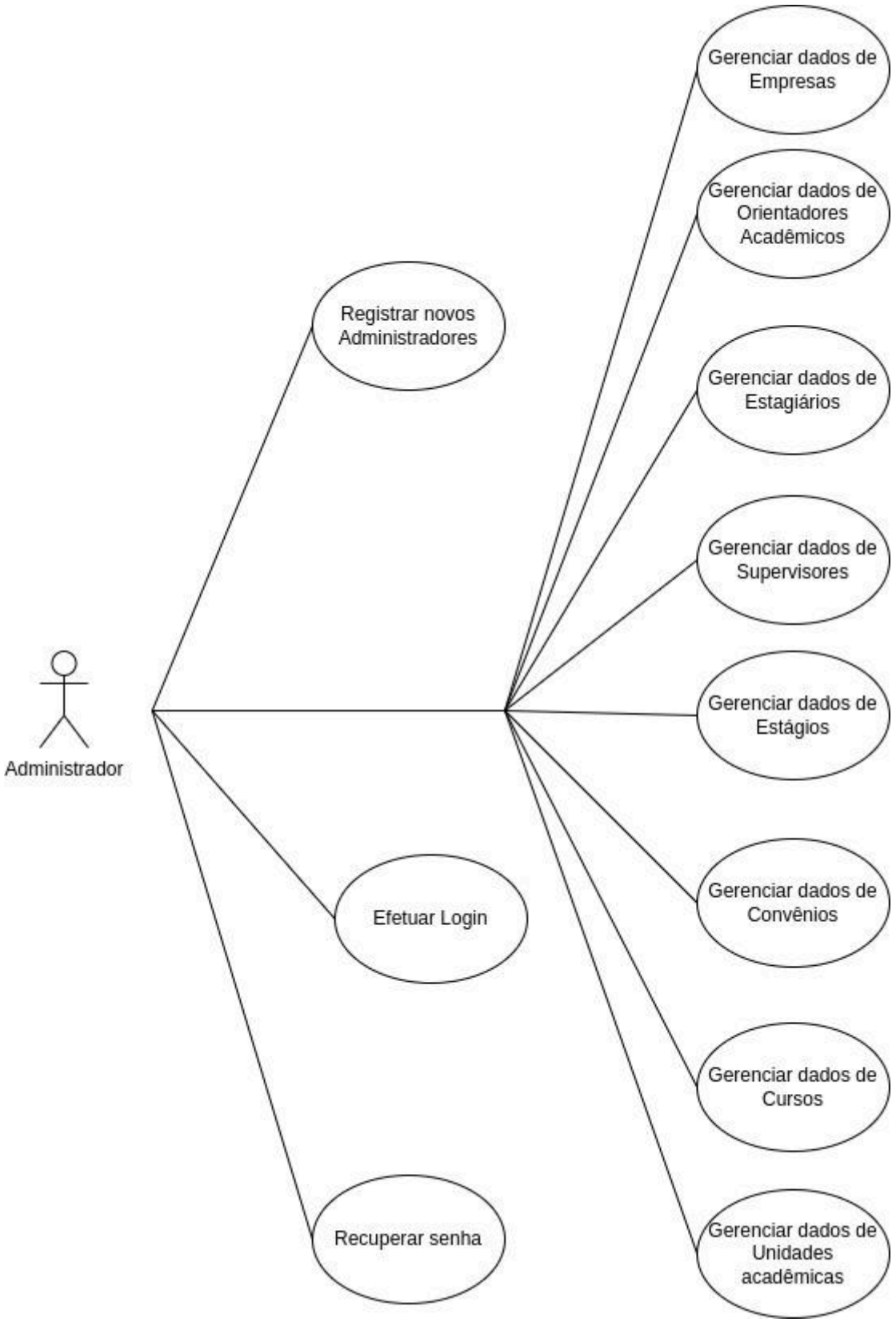

Figura 2: Diagrama de caso de uso demonstrando as operações que um administrador pode realizar no sistema. Autoria própria

Porém, para que essas operações sejam realizadas, é necessário que o sistema possua um SGBD (Sistema de gerenciamento de banco de dados) configurado no back-end, pois garante o armazenamento adequado, organização, recuperação eficiente, segurança e gerenciamento dos dados. Ele fornece as ferramentas e recursos necessários para lidar com os desafios de manipulação e persistência de dados em um ambiente de desenvolvimento de aplicativos.

4.1 TECNOLOGIAS UTILIZADAS

Para desenvolver o banco de dados do SIGEE foi utilizado o banco de dados PostgreSQL (POSTGRESQL, 2023), a linguagem de programação JavaScript (JAVASCRIPT, 2023), o ambiente de execução Node.JS (NODEJS, 2023), que permite a criação e execução de aplicações JavaScript sem depender de um navegador para executá-las, o ORM Sequelize (SEQUELIZE, 2023), que permite realizar operações no banco de dados sem a necessidade de utilizar a linguagem de consulta estruturadas SQL, o Docker (DOCKER, 2023) para a criação do ambiente de desenvolvimento e, por fim, foi utilizado o Postman (POSTMAN, 2023) para a validação da aplicação por meio de testes.

4.2 MODELAGEM DO BANCO DE DADOS

A primeira etapa para fazer a construção do banco de dados foi decidir quais seriam as entidades a serem definidas e como elas seriam relacionadas. Para isso, foi construído um diagrama de classes, que serve para demonstrar cada entidade, suas propriedades e as relações de cada uma, como mostrado na figura 3.

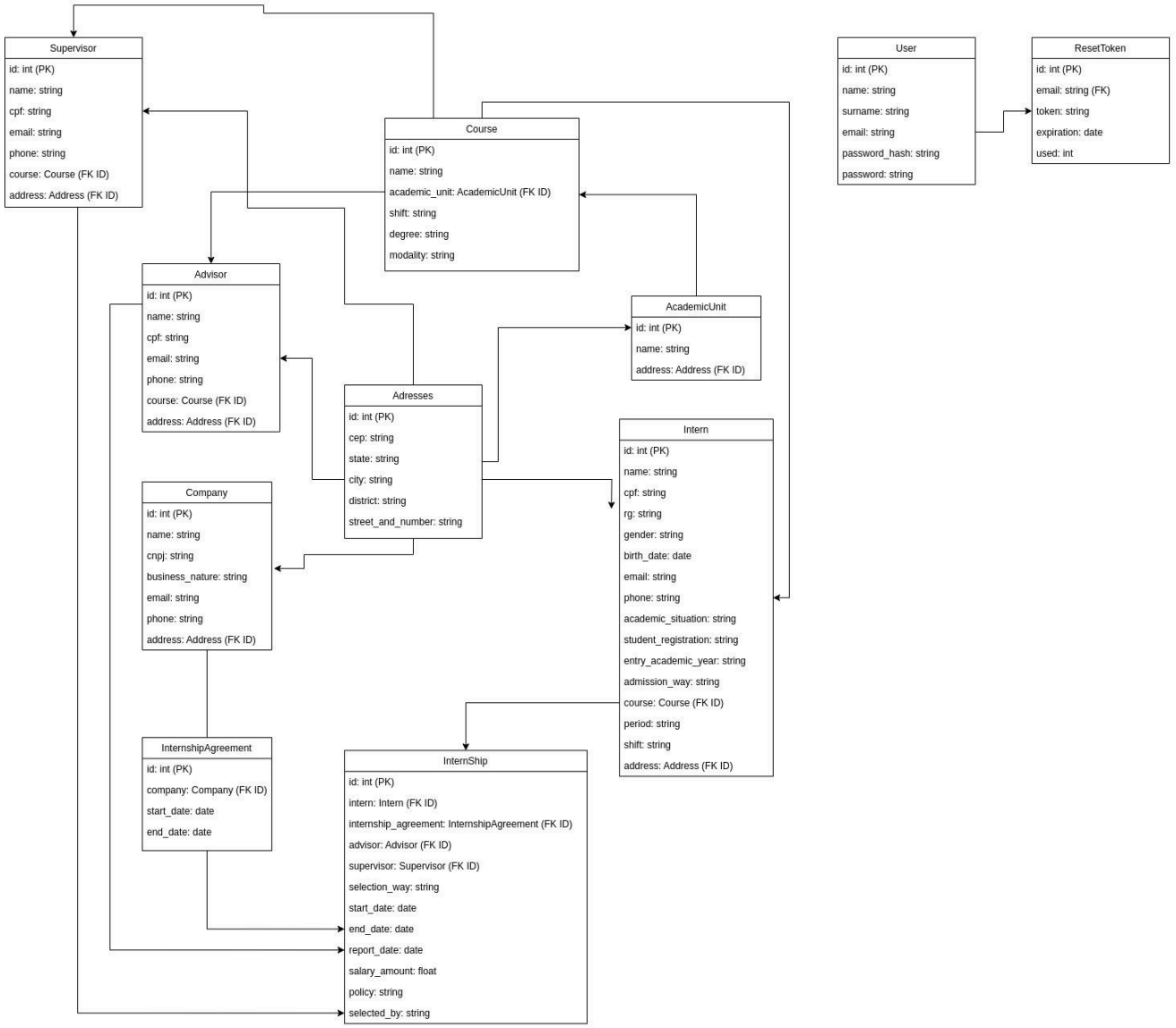

Figura 3: Diagrama de classes. Autoria própria

A figura acima mostra as seguintes classes e relacionamentos:

- **AcademicUnit:** responsável por representar uma Unidade Acadêmica. Ela se relaciona com as classes Address e Course;
- **Course:** representa um Curso pertencente a em uma Unidade Acadêmica e, além dessa relação, possui relações com as classes Intern, Advisor e Supervisor;
- **Advisor e Supervisor:** essas classes são responsáveis por representarem um Orientador e um Supervisor do estágio. Ambas possuem relações com Address, Course e Internship;
- **Internship:** representa o estágio e possui relações com as classes Intern, Agreement, Advisor e Supervisor;

- **Intern:** representa o estagiário e possui relações com Address, Course e Internship;
- **Agreement:** responsável por representar o contrato de estágio. Possui relações com Internship e Company;
- **Company:** representa a empresa que fornece o estágio. Tem relações com Address e Agreement;
- **Address:** representa o endereço das entidades presentes no sistema. Se relaciona com Intern, Company, Advisor, Supervisor e AcademicUnit
- **User:** Representa o usuário que irá administrar o sistema. Se relaciona com a classe ResetToken;
- **ResetToken:** representa o token responsável pela redefinição de senha do usuário. Se relaciona com a classe User;

A próxima etapa foi definir os *models* e *migrations* na aplicação e para isso foi utilizado o ORM Sequelize, que tem como função interagir com o banco de dados, simplificando o processo e permitindo que o desenvolvedor trabalhe com objetos e métodos em vez de escrever consultas SQL manualmente.

No Sequelize, os *models* são classes que representam as tabelas do banco de dados na aplicação. Eles definem a estrutura dos dados e o relacionamento entre as tabelas. Cada *model* geralmente corresponde a uma tabela específica no banco de dados e possui propriedades que representam as colunas da tabela, sendo essas informações o nome da coluna, tipo de dado, restrições, entre outros.

A figura 4 a seguir demonstra o código de como é definido um *model* no Sequelize é criado e configurado.

Model: Course

```
const { Sequelize, Model } = require('sequelize');

class Course extends Model {
  static init(sequelize) {
    super.init({
      name: {
        type: Sequelize.STRING,
        defaultValue: '',
        validate: {
          len: {
            args: [3, 255],
            msg: 'Nome muito curto!',
          },
        },
      },
      shift: {
        type: Sequelize.STRING,
        defaultValue: '',
      },
      degree: {
        type: Sequelize.STRING,
        defaultValue: '',
      },
      modality: {
        type: Sequelize.STRING,
        defaultValue: '',
      },
    },
      {
        sequelize
      });
    return this;
  }
  static associate(models) {

    this.belongsTo(models.AcademicUnit, {
      foreignKey: 'academic_unit_id',
      as: 'academic_unit',
    });

    this.hasMany(models.Intern, {
      foreignKey: 'course_id',
      as: 'intern_course'
    });

    this.hasMany(models.Advisor, {
      foreignKey: 'course_id',
      as: 'advisor_course'
    });

    this.hasMany(models.Supervisor, {
      foreignKey: 'course_id',
      as: 'supervisor_course'
    });
  }
}

module.exports = Course;
```

Figura 4: definição do model *Course.* Autoria própria

A primeira parte do código define os atributos do *model* Course (curso) e suas configurações. O método init é um método estático que recebe um objeto sequelize como parâmetro. Esse objeto é uma instância do sequelize que representa a conexão com o banco de dados.

Dentro do método *init*, é chamado o método *super.init()* para inicializar o *model* Course usando a função *init* da classe pai Model. Depois é passado um objeto contendo as definições dos atributos do *model*. Dito isso, os atributos são os seguintes:

- **name:** um campo de texto que representa o nome do curso. O tipo é *Sequelize.STRING* e possui um valor padrão vazio (*defaultValue: ''*). Também possui uma validação *len* que verifica se o comprimento da string está entre 3 e 255 caracteres.
- **shift, degree e modality:** são campos de texto semelhantes ao atributo *name*, mas não possuem validação e têm valores padrão vazios.

A segunda parte do código define as associações entre o *model* Course e outros *models*, utilizando os métodos *belongsTo* e *hasMany*. Essas associações são usadas para relacionar objetos Course com objetos de outras classes. As associações são as seguintes:

- **belongsTo**: Course pertence a uma AcademicUnit (unidade acadêmica). Isso significa que um curso tem uma unidade acadêmica associada. Logo depois, é definido a *foreignKey* (chave estrangeira) usada para realizar essa associação, que nesse caso é academic_unit_id. O apelido (alias) dessa associação é definido como 'academic_unit'.
- **hasMany:** Course possui muitos Interns (estagiários), Advisors (orientadores) e Supervisors (supervisores). Isso significa que um curso pode ter vários estagiários, orientadores e supervisores associados. As chaves estrangeiras usadas para essas associações são course_id. Os apelidos dessas associações são definidos como 'intern_course', 'advisor_course' e 'supervisor_course', respectivamente.

Por fim, o código exporta a classe Course como um módulo, para que possa ser usado em outros arquivos do projeto.

Após definir os *models*, vem a parte de configurar as *migrations,* que são uma maneira de controlar e gerenciar as alterações no esquema do banco de dados ao longo do tempo, facilitando a evolução e sincronização entre o modelo de dados e a estrutura real do banco de dados.

Uma *migration* no Sequelize basicamente descreve as alterações a serem feitas no esquema do banco de dados. Essas alterações podem incluir criação de tabelas, adição de colunas, modificação de tipos de dados, criação de relacionamentos, entre outros.

A figura 5 a seguir demonstra o código de como é definida uma *migration* no Sequelize:

Migration: course

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('courses', {
      id: {
        type: Sequelize.INTEGER,
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
      },
      name: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      shift: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      degree: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      modality: {
        type: Sequelize.STRING,
        allowNull: false,
      },
      academic_unit_id: {
        type: Sequelize.INTEGER,
        references: { model: 'academic_units', key: 'id' },
        onDelete: 'CASCADE',
        onUpdate: 'CASCADE',
      },
      created_at: {
        type: Sequelize.DATE,
        allowNull:false,
      },
      updated_at: {
        type:Sequelize.DATE,
        allowNull: false,
      },
    });
  },

  down: async (queryInterface) => {
    await queryInterface.dropTable('courses')
  }
};
```

Figura 5: Definição da migration *Course.* Autoria própria

O arquivo de *migrations* define duas funções: *up* e *down*. A função up é responsável por criar a tabela "courses" no banco de dados, enquanto a função down é responsável por remover essa tabela.

Ao analisar o código, podemos observar alguns trechos importantes:

- **module.exports = { ... }** é responsável por exportar um objeto que contém as funções *up* e *down*, tornando-as acessíveis em outros arquivos.
- **up: async (queryInterface, Sequelize) => { ... }** define a função *up* como uma função assíncrona que recebe dois parâmetros: *queryInterface* e *Sequelize*.
    - O *queryInterface* é uma interface fornecida pelo Sequelize para executar consultas e realizar alterações no banco de dados.
    - O *Sequelize* é o objeto do Sequelize que contém métodos e tipos de dados.
- **await queryInterface.createTable('courses', { ... })** usa o *queryInterface* para criar uma tabela chamada 'courses' no banco de dados. Em seguida, são definidos os campos da tabela, *como id, name, shift, degree, modality, academic_unit_id, created_at e updated_at,* juntamente com os tipos de dados correspondentes.
- **down: async (queryInterface) => { ... }** define a função *down* como uma função assíncrona que recebe o parâmetro *queryInterface*. Essa função é responsável por reverter as alterações feitas pela função *up*. Neste caso, a função *down* remove a tabela 'courses' do banco de dados usando o *queryInterface.dropTable('courses')*

Após montar a estrutura do banco de dados, é importante certificar de que ao realizar alguma operação de CRUD, nenhum dado fique comprometido caso ocorra algum erro durante alguma operação. Para garantir a consistência dos dados, foram configuradas as *transactions*.

Transactions ou transações são mecanismos que permitem agrupar um conjunto de operações de banco de dados em uma unidade lógica e indivisível. Elas garantem a consistência e a integridade dos dados durante a execução de várias operações, permitindo um melhor o fluxo de execução e desfaça todas as alterações

caso ocorra um erro. Na figura 6 que se segue é possível ver como é configurada uma transaction em uma operação de CRUD, *create* (criar):

Controller: company

```
exports.create = async (req, res, next) => {
  let transaction;

  try {
    transaction = await sequelize.transaction();

    const address = await Address.create(req.body.address, { transaction });
    await Company.create({
      name: req.body.name,
      cnpj: req.body.cnpj,
      business_nature: req.body.business_nature,
      email: req.body.email,
      phone: req.body.phone,
      address_id: address.id
    }, { transaction });

    await transaction.commit();

    return res.status(201).json({ status: 'ok' });
  } catch (err) {
    if (transaction) {
      await transaction.rollback();
    }

    next(err);
  }
};
```

Figura 6: Implementação de uma transaction no controller de *company.* Autoria própria

O código acima lida com a operação *create* de uma nova empresa. Os trechos mais importantes são:

- ***let transaction;*:** Ele começa com uma variável *transaction*, que será utilizada para controlar a transação do banco de dados.
- ***transaction = await sequelize.transaction();*** inicia uma nova transação usando um objeto *sequelize* e o método *transaction()* retorna uma nova instância de transação que pode ser usada para executar várias operações do banco de dados.
- ***const address = await Address.create(req.body.address, { transaction });:*** Nessa linha, um endereço é criado usando o *model* Address. O método *create()* é chamado no *model* Address e recebe dois argumentos: *req.body.address* (os dados do endereço fornecidos na requisição) e *{*

transaction } (o objeto de transação). A função *create()* retorna uma nova instância de endereço, que é atribuída à variável *address*.

- **await Company.create({...}, { transaction });:** Nesta parte, uma nova empresa é criada usando o *model* Company. O método *create()* é chamado no *model* Company e recebe dois argumentos: um objeto contendo os dados da empresa (name, cnpj, business_nature, email, phone e address_id) e { transaction } (o objeto de transação). A função *create()* cria uma nova instância de empresa no banco de dados associada ao endereço criado anteriormente.
- **await transaction.commit();:** Aqui é confirmada a transação usando o método *commit()* do objeto de transação. É responsável por aplicar todas as alterações feitas no banco de dados durante a transação.
- **await transaction.rollback();:** Se ocorrer algum tipo de erro durante o momento da transação, o método *rollback()* é chamado para desfazer quaisquer alterações feitas no banco de dados durante a transação.

5 TESTES E VALIDAÇÃO

Para fazer a validação no banco de dados, foi utilizado o Postman, que é uma ferramenta que dá suporte à documentação das requisições feitas pela API. Ele possui ambiente para a documentação, execução de testes de APIs e requisições em geral. Ele foi responsável por enviar solicitações HTTP para a API e verificar as respostas retornadas. A API então seria responsável por executar as operações de CRUD no banco de dados, como demonstrado na figura 7 abaixo.

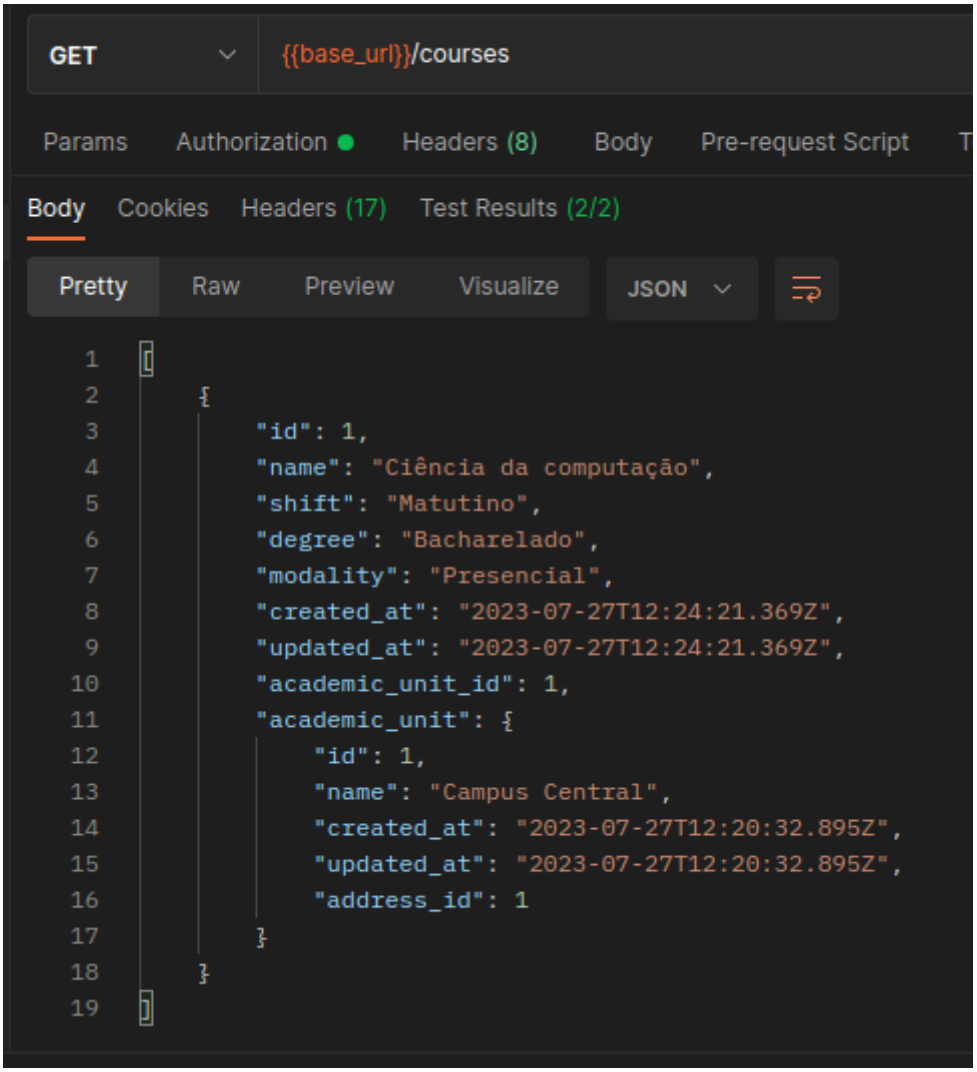

Figura 7: Teste de listagem usando o postman. Autoria própria

Na figura acima, a API realizou a operação de listar os cursos que estão presentes no banco de dados, permitindo que fosse verificado se os dados foram inseridos corretamente e se foram retornados de maneira correta.

Além disso, foi possível realizar outros tipos de testes, como testar a restrição de tamanho dos campos. É possível ver esse teste na figura 8 abaixo:

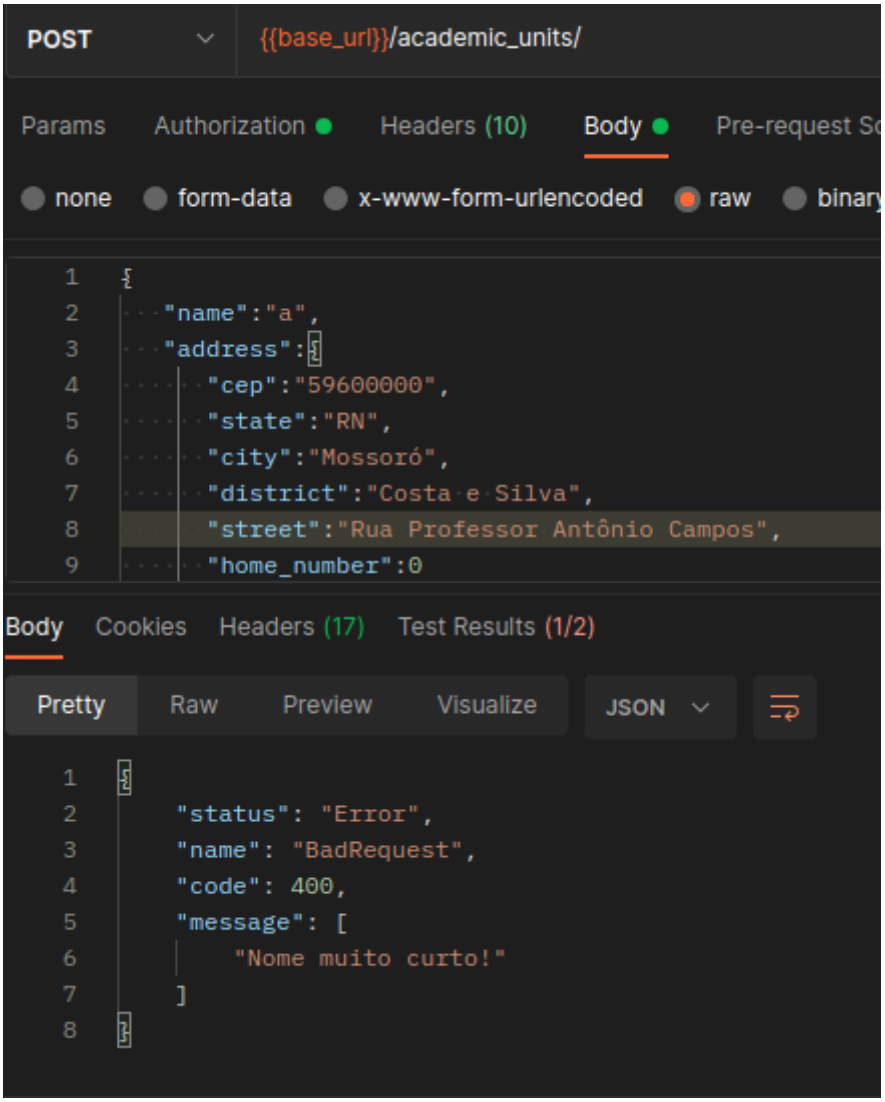

Figura 8: Teste de inserção incorreta de dados. Autoria própria

Acima é possível ver a restrição de tamanho funcionando, pois é definido que o nome da unidade acadêmica deve ser de entre 3 e 255 caracteres. Além disso, é também possível ver o *rollback* da transação ocorrendo, conforme demonstrado na figura 9 a seguir:

```
Executing (eb764caa-77ce-4ced-bc4c-edd5c093af86): START TRANSACTION;
Executing (eb764caa-77ce-4ced-bc4c-edd5c093af86): INSERT INTO "addresses
" ("id","cep","state","city","district","street","home_number","created_
at","updated_at") VALUES (DEFAULT,$1,$2,$3,$4,$5,$6,$7,$8) RETURNING "id
","cep","state","city","district","street","home_number","created_at","u
pdated_at";
Executing (eb764caa-77ce-4ced-bc4c-edd5c093af86): ROLLBACK;
```

FIgura 9: Análise do processo de transação. Autoria própria

Nessa imagem é possível identificar que, ao solicitar que a API realize a operação de inserção no banco de dados, uma transação é iniciada, porém ao tentar inserir um dado inválido, é realizado o rollback da transação, que desfaz quaisquer alterações feitas no banco de dados durante a transação, garantindo a consistência dos dados.

6 CONCLUSÃO

De acordo com o que foi mostrado, o objetivo deste trabalho foi o desenvolvimento de um banco de dados capaz de armazenar um grande volume de dados de diversos estágios de forma segura e eficiente, além de providenciar confiabilidade ao administrador do sistema. O banco de dados foi construído utilizando o PostgreSQL, junto com a linguagem de programação JavaScript em conjunto com o ORM Sequelize, no qual providenciou diversas facilidades nessa construção, pois dispensava o uso da linguagem SQL para a construção das tabelas e relações, permitindo que fosse trabalhado utilizando objetos e métodos.

O sistema foi apresentado ao setor de estágios da Pró-Reitoria de Assuntos Estudantis para que fosse feita uma análise sobre a funcionalidade do software como um todo, sendo ele composto pelo banco de dados, API e o front-end. O sistema foi validado utilizando o front-end e foi aprovado, atingindo os objetivos esperados.

Um trabalho futuro seria realizar uma avaliação do desempenho do sistema em situações de sobrecarga de dados, visando analisar possíveis otimizações e novas funcionalidades.

REFERÊNCIAS

DOCKER, Inc. Why docker. 2023. Disponível em: https://www.docker.com/why-docker/. Acesso em: 27 de junho de 2023.

JAVASCRIPT. About. 2023. Disponível em: https://www.javascript.com/about. Acesso em: 27 de junho de 2023.

NODEJS. About Node.js. Disponível em: https://nodejs.org/en/about. Acesso em: 27 de junho de 2023.

POSTGRESQL. About. 2023. Disponível em: https://www.postgresql.org/about/. Acesso em: 27 de junho de 2023.

POSTMAN. What is Postman. 2023. Disponível em: https://www.postman.com/product/what-is-postman/. Acesso em: 27 de junho de 2023.

SEQUELIZE. 2023. Disponível em: https://sequelize.org/. Acesso em: 27 de junho de 2023.

SOMMERVILLE, Ian. Engenharia de Software. 9.ed. São Paulo, Brasil: Pearson, 2011

TRELLO. Sobre o Trello. Disponível em: https://trello.com/about. Acesso em 27 de junho de 2023.