UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus dos Santos Mendes

Proposta de Nova Implementação do Sistema Online de Distribuição de Disciplinas

Uberlândia, Brasil

2022

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus dos Santos Mendes

Proposta de Nova Implementação do Sistema Online de Distribuição de Disciplinas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Bruno Augusto Nassif Travençolo

Universidade Federal de Uberlândia – UFU

Faculdade da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2022

Dedico esse trabalho a todos que me ajudaram a chegar até o presente momento.
Família, amigos, colegas e, principalmente, meus professores

Agradecimentos

Meus sinceros agradecimentos aos professores e mentores que me impulsionaram e me proporcionaram acesso ao conhecimento e também aos meus queridos amigos por não me deixarem desistir dos meus objetivos.

Resumo

Softwares, assim como qualquer outro produto, estão em constante evolução e manutenção. Quando há a necessidade de se reinventar, um estudo é feito para que os próximos passos desse processo sejam dados de forma planejada e incremental. O Sistema Online de Distribuição de Disciplinas, ou SODD, um software desenvolvido pela Faculdade de Computação (FACOM) da UFU cujo objetivo é automatizar o processo de distribuir as disciplinas ofertadas pela FACOM aos professores disponíveis, requer, neste momento, grande manutenção e revisão de código. Este Trabalho de Conclusão de Curso tem como ponto central reescrever o software de forma que sua manutenção se torne menos trabalhosa, seu código fique mais limpo e legível, seus processos se tornem mais independentes, e haja uma estrutura de testes.

Palavras-chave: Sistema Online de Distribuição de Disciplinas, software, manutenção.

Lista de ilustrações

Figura 1 – JSON Web Token 16
Figura 2 – Diagrama de interação dos módulos da aplicação 19
Figura 3 – Diretório raiz do projeto 22
Figura 4 – Diretório modules 23
Figura 5 – Diretório shared 24
Figura 6 – Estrutura do banco de dados 26
Figura 7 – Migrations criadas para construção das tabelas do banco de dados . . . 29
Figura 8 – Resumo dos testes 45
Figura 9 – Relatório dos testes 46
Figura 10 – Interface Swagger 47
Figura 11 – Raíz do projeto frontend 49
Figura 12 – Tela de visualização de cursos 50
Figura 13 – Tela de Criação de curso 50
Figura 14 – Tela de Edição de curso 51

Sumário

1 INTRODUÇÃO . . . 8
1.1 Objetivos . . . 9
1.1.1 Objetivo Geral . . . 9
1.1.2 Objetivos Específicos . . . 9
1.2 Organização do Trabalho . . . 10

2 TRABALHOS E TECNOLOGIAS RELACIONADAS . . . 11
2.1 Trabalhos Correlatos . . . 11
2.2 Metodologias Utilizadas . . . 12
2.2.1 Princípios SOLID . . . 12
2.2.1.1 *Single Responsability Principle* . . . 12
2.2.1.2 *Open-Closed Principle* . . . 13
2.2.1.3 *Liskov Substitution Principle* . . . 13
2.2.1.4 *Interface Segregation Principle* . . . 13
2.2.1.5 *Dependency Inversion Principle* . . . 13
2.2.2 *Domain Driven Design* . . . 13
2.3 Tecnologias Utilizadas . . . 14
2.3.1 Javascript . . . 14
2.3.1.1 Typescript . . . 14
2.3.2 Yarn . . . 14
2.3.3 TypeORM . . . 14
2.3.4 Jest . . . 15
2.3.5 Swagger . . . 15
2.3.6 ESLint e Prettier . . . 15
2.3.7 JSON Web Token . . . 16
2.3.8 Node.js . . . 17
2.3.8.1 Express . . . 17
2.3.9 React . . . 17
2.3.9.1 Next.js . . . 17
2.3.9.2 Chakra UI . . . 18

3 DESENVOLVIMENTO DO PROJETO . . . 19
3.1 Interação da aplicação . . . 19
3.2 Início da API backend . . . 20
3.2.1 Adicionando Typescript . . . 20
3.2.2 Configurando ESLint e Prettier . . . 21

3.2.3 Configurando TypeORM . . . 21
3.2.4 Estrutura de diretórios . . . 21
3.2.4.1 Diretório *modules* . . . 23
3.2.4.2 Diretório *shared* . . . 24
3.2.5 Criação dos módulos . . . 25
3.2.5.1 Criação das *migrations* . . . 25
3.2.5.2 Repositórios e Entidades . . . 29
3.2.5.3 Construção dos Serviços . . . 31
3.2.5.4 Construção das classes *Controllers* . . . 33
3.2.5.5 Construção das Rotas . . . 34
3.2.6 Criação de Sessões . . . 35
3.2.6.1 Autenticando o Usuário . . . 35
3.2.6.2 Renovando a Sessão . . . 37
3.2.7 *Middlewares* . . . 38
3.2.7.1 *Middleware* de Autenticação . . . 39
3.2.7.2 *Middleware* de Administrador . . . 40
3.2.8 Testes com Jest . . . 41
3.2.8.1 Testes unitários . . . 42
3.2.8.2 Testes de integração . . . 43
3.2.8.3 Cobertura de código . . . 45
3.2.9 Documentação com Swagger . . . 46
3.3 Início da API *frontend* . . . 47
3.3.1 Adicionando Typescript . . . 48
3.3.2 Configurando ESLint e Prettier . . . 48
3.3.3 Módulos do projeto . . . 48
3.3.4 Criação das páginas . . . 49

4 RESULTADOS . . . 52
4.1 Padronização de código . . . 52
4.2 Histórico do banco de dados . . . 52
4.3 Automatização de testes no *backend* . . . 52
4.4 Rotas Seguras . . . 52
4.5 Criação do CRUD das tabelas . . . 53
4.6 Configuração do ambiente *frontend* . . . 53
4.7 *Layout* de páginas construído . . . 53

5 CONCLUSÃO . . . 54

REFERÊNCIAS . . . 55

1 Introdução

A Faculdade da Computação (FACOM) da Universidade Federal de Uberlândia (UFU) possui a necessidade de distribuir, semestralmente, as disciplinas disponíveis dos cursos sob sua responsabilidade e as disciplinas ofertadas pela FACOM a cursos de outras unidades acadêmicas da UFU para professores da FACOM elegíveis a ministrarem essas disciplinas. Atualmente, o processo é conduzido pela Comissão de Distribuição de Disciplinas (CDD) com base em norma específica da FACOM e é utilizado o Sistema Online de Distribuição de Disciplinas (SODD) para atender essa necessidade. O sistema, cuja primeira versão foi implementada por (LOCATELLI, 2015), utilizando como linguagem de programação o Java e padrão arquitetural MVC, conseguiu suprir grande parte da necessidade da comissão, porém com alguns requisitos sendo realizados de forma manual, utilizando *scripts* de Linguagem de Consulta Estruturada, abrindo janelas para possíveis falhas lógicas e de execução de *scripts*.

Com o passar do tempo, o sistema foi sendo incrementado, ganhando funcionalidades como configurações de disciplinas, configurações de filas, configurações de curso, dentre várias outras. Tais requisitos foram implementados por desenvolvedores diferentes, com técnicas e habilidades diferentes, adicionando e corrigindo funcionalidades ao sistema, o que fez com que o mesmo se tornasse cada vez mais difícil de se dar manutenção, de se ler, e até mesmo implementar novos requisitos. Vale ressaltar que esse processo é muito comum no ciclo de vida da maioria dos softwares existentes. Por todos esses motivos, este é o momento adequado para propor uma nova implementação para o Sistema Online de Distribuição de Disciplinas.

Apesar de robusto e apresentar muitas funcionalidades que atendem bem à CDD, alguns pontos de melhoria foram identificados no SODD. O sistema, por exemplo, não conta com nenhuma estratégia de identificação, cadastro ou sessão do usuário que o está acessando, sendo que, qualquer um com a URL de acesso consegue utilizá-lo, trazendo perigo à dinâmica de distribuição de disciplinas. Além disso, o SODD também não possui arquitetura de testes automatizados, dificultando a garantia de qualidade e funcionalidade correta dos requisitos implementados.

Muito se falará sobre código limpo e padronizado ao longo deste trabalho. Como é observado no livro (MARTIN, 2019), não há uma definição concreta e universal para um código limpo, porém, há formas de escrevê-lo e pensá-lo: um código limpo deve ser simples e direto, de fácil leitura, devidamente testado, inteligível, dentre outras características que o tornam intuitivo não somente para quem o desenvolveu inicialmente, mas para os próximos desenvolvedores que irão conduzir o projeto. Neste sentido, a nova im-

plementação busca padronizar o desenvolvimento de forma que o código se torne de fácil leitura e manutenção, sempre buscando seguir o conceito de código limpo.

Considerando o histórico do projeto, bem como seus pontos fracos e com o intuito de evoluir, documentar e melhorar a qualidade de código do Sistema Online de Distribuição de Disciplinas, identificou-se a oportunidade de realizar uma nova implementação do sistema utilizando novas tecnologias, bem como a implementação de novas funcionalidades, dando ainda mais robustez ao SODD. Como parte desta nova fase, o módulo de contas é implementado para trazer mais segurança ao projeto, fazendo com que novos usuários se cadastrem no sistema, acessem-o utilizando seu endereço eletrônico e senha cadastrada, e iniciem uma sessão que lhe dará acesso as funcionalidades presentes. A nova implementação do SODD também contará com a possibilidade de documentação de API, tornando o processo de aprendizado sobre o software mais rápido para os outros desenvolvedores. Além disso, uma arquitetura de testes foi implementada para que o projeto conte com testes automatizados, aumentando a sua qualidade e facilitando sua manutenção.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo deste trabalho é propor uma nova implementação para o Sistema Online de Distribuição de Disciplinas (SODD) da FACOM-UFU.

1.1.2 Objetivos Específicos

Tendo em vista o objetivo geral, os pontos específicos que o trabalho visa contemplar ao término deste trabalho são:

- Identificar os requisitos que o sistema já contempla a fim de possibilitar uma nova implementação utilizando novas tecnologias.
- Iniciar o novo projeto, criando a base para que seja possível implementar a estrutura de testes, documentação e o código compartimentado.
- Implementar um módulo de autenticação que traga ao sistema maior segurança e confiabilidade, sendo acessado somente por pessoas autorizadas.
- Identificar pontos de vantagem nas tecnologias utilizadas a fim de extrair o melhor delas no decorrer do desenvolvimento.
- Utilizar uma metodologia de desenvolvimento para se obter um software com código padronizado, reduzindo tempo de projeto e eventuais manutenções.

1.2 Organização do Trabalho

O presente trabalho foi estruturado de forma a conduzir o leitor pela linha de raciocínio criado para desenvolver o projeto. No Capítulo 2 são apresentados os trabalhos correlatos e as principais ferramentas e metodologias utilizadas para a construção do software. No Capítulo 3 é apresentado o processo de construção geral da implementação, incluindo um exemplo prático do desenvolvimento. No Capítulo 4 são encontrados os resultados obtidos pelo presente trabalho e, por fim, no Capítulo 5, é possível encontrar a conclusão após o desenvolvimento.

2 Trabalhos e Tecnologias Relacionadas

Este capítulo contém as ferramentas necessárias para o desenvolvimento da proposta da nova implementação do Sistema Online de Distribuição de Disciplinas, como trabalhos correlatos, metodologias e tecnologias utilizadas.

2.1 Trabalhos Correlatos

Em (DAMASCENO, 2021), há um capítulo em que é detalhado todo o histórico de entregas do Sistema Online de Distribuição de Disciplinas. Nele, pode-se ter uma boa noção da história do projeto, dos desenvolvedores que participaram do projeto bem como os requisitos implementados. Dito isso, é importante entender que não necessariamente os atuantes do projeto tiveram contato para discutirem sua implementação, padronização e documentação e, com o passar do tempo, o software se tornou cada vez mais robusto, com alguns requisitos não mais necessários ou funcionando de forma incorreta, além de trabalhar com tecnologias um pouco desatualizadas que poderiam ser substituídas por outras que atendem melhor o projeto em seu momento atual. A monografia apresenta o histórico do projeto, dividido em entregas, de forma que cada entrega teve o seu foco:

- Primeira entrega: nesta entrega foi realizado o levantamento dos requisitos funcionais e não funcionais para a implementação do sistema. Além disso, deu-se início ao desenvolvimento, implementando funcionalidades como a verificação de horários das disciplinas cadastradas, exibição de fila por disciplina, exibição de fila por professor com e sem turmas, e exibição de fila por turma.
- Segunda entrega: nesta entrega foi implantado um controle de versões do projeto utilizando a ferramenta Git. Além disso, foram implementadas mais funcionalidades, como as configurações de Disciplinas, Prioridades, Turmas, dentre outras que podem ser encontradas na própria monografia. Foram implementados, também, relatórios de distribuição e filas gerais, além de melhoras visuais na interface de usuário.
- Terceira entrega: nesta entrega deu-se continuação à implementação do SODD, levantando requisitos, dando manutenção no código e implementando novas funcionalidades. Algumas delas foram: criação e edição de disciplinas, duplicação de turmas, criação e edição de professores, dentre outros encontrados na monografia.
- Quarta entrega: na quarta entrega houve manutenção de código, correção de *bugs* e implementação de novas funcionalidades como cadastramento de novos semestres,

visalização de filas de disciplinas, turmas e de um professor específico, relatórios de distribuição de turmas, dentre outros encontrados na monografia.

- Quinta entrega: nesta entrega foi desenvolvido o algoritmo responsável por realizar a distribuição das disciplinas. Diferente dos desenvolvimentos anteriores, este foi realizado em C# e anexado ao sistema.

O sistema atual utiliza AngularJS para criação da interface do usuário. Apesar de atender as necessidades do projeto, o AngularJS é a primeira versão de um *framework* mantido pelo Google, sendo muito diferente de suas versões posteriores, o que traz muita dificuldade de migração entre estas versões. Para compor a proposta de nova implementação e pensando na manutenibilidade do projeto futuramente, esta tecnologia será substituída pela versão mais atual, até o presente momento, do React, a biblioteca mantida pelo Facebook e com foco em criação de interfaces de usuários para aplicações *web*.

O Sistema Online de Distribuição de Disciplinas também conta com Java para compor sua estrutura *backend*. O projeto usa o Java em sua versão 1.8.0-151, que também está desatualizado, fazendo com que nem todas as vantagens da linguagem sejam utilizadas no momento atual. Além disso, não utiliza nenhum *framework* para esta linguagem, o que pode dificultar implementações de novas funcionalidades no futuro.

2.2 Metodologias Utilizadas

2.2.1 Princípios SOLID

Para o desenvolvimento e aprimoramento do Sistema Online de Distribuição de Disciplinas foram adotadas algumas posturas, princípios e conceitos de programação com o intuito de tornar o software o mais inteligível, seguro, completo e de mais fácil manutenção possível. Com isso, os princípios SOLID foram grandes aliados ao decorrer do desenvolvimento.

No universo da tecnologia da informação, SOLID é um acrônimo que diz respeito a cinco princípios e posturas que devem ser adotadas para que o design e arquitetura do software seja fácil de manter, escalar e testar. Esse acrônimo foi introduzido por Michael Feathers, com princípios que podem ser identificados no livro (MARTIN, 2019). Cada letra do acrônimo representa um princípio, mais detalhado a seguir:

2.2.1.1 *Single Responsability Principle*

O Princípio de Responsabilidade Única determina que cada classe deve ter uma única função. Esse princípio foca na interoperabilidade dentro de um sistema, ditando que

uma classe deve dominar e possuir apenas uma responsabilidade, uma única tarefa para executar.

2.2.1.2 *Open-Closed Principle*

Neste princípio, classes ou objetos devem estar abertos para extensão, mas fechados para modificação, ou seja, novos comportamentos que surgem à medida que a complexidade do sistema aumenta devem ser adicionados de forma extensiva, seja por composição, herança ou alguma outra estratégia que não altere o comportamento original da classe.

2.2.1.3 *Liskov Substitution Principle*

Uma classe derivada deve ser substituível por sua classe base. Este princípio, descrito por Barbara Liskov, evidencia que, antes de optar por herança, deve-se pensar nas condições que antecedem e sucedem sua classe atual.

2.2.1.4 *Interface Segregation Principle*

Este princípio diz que uma classe não deve ser forçada a implementar interfaces e métodos que não irá utilizar. Ele evidencia que é melhor ter muitas interfaces específicas para cada propósito do que uma única interface genérica com muitos métodos, de propósito geral.

2.2.1.5 *Dependency Inversion Principle*

Este princípio que pode ser encontrado no livro (MARTIN, 2019) define que "Módulos de alto nível não devem depender de detalhes de baixo nível" e "as abstrações não devem depender de detalhes. Os detalhes devem depender das abstrações".

2.2.2 *Domain Driven Design*

Além dos princípios de SOLID, o uso do *Domain Driven Design*, ou DDD, reforça a qualidade da implementação e aprimoramento do Sistema de Distribuição Online de Disciplinas. DDD combina práticas de desenvolvimento e design, além de oferecer ferramentas de modelagem para entregar softwares de alta qualidade de modo a potencializar o uso do tempo. Segundo (VALENTE, 2020), DDD defende e afirma que os desenvolvedores devem ter pleno conhecimento do domínio do sistema que estão implementando. Este conhecimento vem de discussões e conversas com quem é especialista no domínio do problema. No caso do Sistema Online de Distribuição de Disciplinas, este especialista é quem propõe a sua implementação, o orientador deste trabalho, Prof. Bruno Travençolo. Os três pilares do *Domain Driven Design* são: linguagem ubíqua, contextos limitados e

mapas de contexto. É importante ressaltar que DDD não é uma tecnologia ou metodologia específica, é independente de linguagem ou framework, e traz como principal fator de sucesso a possibilidade de manter o software totalmente alinhado com o seu propósito.

2.3 Tecnologias Utilizadas

2.3.1 Javascript

Javascript, segundo (MOZILLA, 2022) é uma linguagem de programação de alto nível, utilizada principalmente para aplicações e desenvolvimento *web*. Quando utilizada juntamente com HTML e CSS, torna possível a criação de páginas poderosas, com interface dinâmica e amigável, além de possuir alta performance. Foi criada com o intuito de funcionar do lado dos navegadores (do usuário), mas devido ao seu poder e dinamismo, passou-se a utilizá-la também do lado dos servidores, com a ajuda do Node.js, por exemplo. Com isso, além de ser utilizada para criação de páginas *web*, Javascript pode ser utilizada também no *backend*, criando sistemas inteiros ou micro-serviços disponibilizados para outras aplicações.

2.3.1.1 Typescript

Typescript veio para complementar o Javascript e, por isso, é importante frisar que não é uma outra versão da linguagem, ou mesmo um concorrente. Tecnicamente falando, é um *superset* para Javascript, ou seja, é um conjunto de ferramentas adicionais que funcionam com a linguagem. Como pode ser encontrado em (MICROSOFT, 2022), com Typescript é possível, por exemplo, adicionar tipagem estática e orientação a objetos em códigos Javascript, trazendo mais poder, segurança e padronização ao desenvolvimento. Typescript precisa ser convertido em Javascript “puro” antes de ser executado de fato.

2.3.2 Yarn

Yarn é um gerenciador de pacotes que permite adicionar *frameworks* e ferramentas ao projeto de forma segura e rápida utilizando instruções em linhas de comando. Tal gerenciador de pacotes pode ser utilizado juntamente com o gerenciador NPM, muitas vezes instalado junto com o Node.js, como também pode o substituir. Yarn e NPM possuem eficácia e papéis muito semelhantes, tornando a opção de uso facultativa ao desenvolvedor.

2.3.3 TypeORM

Antes de descrever TypeORM especificamente, deve-se entender o que é ORM. Um ORM, sigla em inglês para *Object Relational Mapper*, aproxima o paradigma de desenvolvimento de aplicações orientadas a objetos ao paradigma do banco de dados relacional.

É uma técnica que possibilita a escrita de consultas e operações DML utilizando Typescript, por exemplo. Segundo (TYPEORM, 2022), TypeORM é um *framework* que pode ser utilizado com Node.js, utilizando Javascript ou Typescript. Essa ferramenta possui um recurso muito útil chamado *migration*, que pode ser descrito como uma espécie de controle de versão do banco de dados. Com as *migrations*, pode-se criar tabelas, editá-las, adicionar colunas, chaves primárias e outras operações das quais se tem rastreabilidade, possibilitando entender quando e como a estrutura do banco de dados e suas tabelas foram modificadas. O projeto utiliza PostgreSQL, um famoso sistema gerenciador de banco de dados relacional. TypeORM é utilizado para estabelecer a conexão com o PostgreSQL e realizar operações relacionadas ao banco de dados.

2.3.4 Jest

O site oficial do Jest, (SOURCE, 2022), aponta-o como um *framework* que trabalha com Javascript e Typescript. Além de muito popular, possibilita a escrita de testes unitários de forma simples e intuitiva. Pode ser utilizado com a tecnologia *supertest* para realizar testes de integração também. Jest foi desenvolvido inicialmente pelo Facebook para testar código React mas acabou crescendo e se popularizando tanto que hoje é utilizado para testes utilizando Typescript, Node, Angular e outras tecnologias.

2.3.5 Swagger

Swagger é um *framework* para documentar, consumir e visualizar os serviços de uma API REST. Com ele, é possível interagir com o *backend* de forma amigável, com possibilidade de informar parâmetros e assim visualizar a resposta de uma requisição, por exemplo. Como visto em (SMARTBEAR, 2022), Swagger é um conjunto de ferramentas, cujas de destaque são: Swagger Editor, Swagger UI e Swagger Codegen. Além disso, ele opera com *OpenAPI Specification*, capaz de descrever o formato das requisições, recursos, métodos HTTP aceitos pelo serviço e respostas aceitas.

2.3.6 ESLint e Prettier

ESLint e Prettier são poderosos aliados para padronizar a escrita do código e manter as boas práticas. Cada ferramenta tem a sua responsabilidade, sendo ESLint responsável por definir o conjunto de regras que o padrão de código vai seguir, além de identificar o código que não estiver seguindo este conjunto pré-estabelecido. Prettier é responsável por realizar a formatação do código baseando-se nas regras definidas pelo ESLint. É interessante utilizar estas duas ferramentas no projeto pois ele será modificado por outros desenvolvedores que nem sempre terão contato entre si e com formas diferentes de codificar. Sendo assim, com as ferramentas devidamente configuradas, o projeto definirá

um conjunto de regras de padronização que independerá do desenvolvedor que estará atuando.

2.3.7 JSON Web Token

Como pode ser encontrado em (AUTH0, 2022), um JSON Web Token, ou JWT, é um padrão da indústria definido pela RFC7519 e tem por função transmitir de forma segura objetos JSON entre duas partes. Ele pode ser enviado via POST ou em um cabeçalho HTTP e pode ser assinado digitalmente tanto por um algoritmo HMAC quanto por um par de chaves usando RSA. O JWT é composto por três partes, conforme exibido na Figura 1.

Figura 1 – JSON Web Token

- *Header*: é a configuração do *token*. Nele consta o tipo de algoritmo que está sendo utilizado para encriptar o *token* e o seu tipo que, no caso, é o JWT.
- *Payload*: com o *payload* podemos transportar dados por meio das requisições. Por exemplo, podemos passar o nome de usuário e um valor booleano para identificar se este é um administrador do sistema ou não, mas não é recomendável utilizá-lo para dados sensíveis, como por exemplo a senha de um usuário. Os *payloads* contém *claims* onde se encontram informações sobre uma entidade, e podem ser de três tipos: *public*, *reserved* ou *private*.
- *Signature*: é a assinatura única que cada *token* possui, além de ser o componente mais sensível da composição por se tratar da sua segurança. Para compor esta parte

do JWT, é utilizado o header, o payload e uma chave secreta de posse da aplicação que pode ser usada para alteração, criação ou validação do *token*.

O JSON Web Token foi útil à aplicação por utilizar uma estrutura leve para ser transportado e relativamente simples de se manipular, que é o JSON. Além de ser seguro, visto que é possível verificar e validar o conteúdo do *token* por meio de uma chave segura que somente a aplicação conhece.

2.3.8 Node.js

Node.js é um ambiente de execução utilizado do lado do servidor. Utilizando Javascript, é possível criar aplicações que não dependem de um navegador para serem executadas. Sua principal característica é o fato de trabalhar assincronamente em uma única *thread* de execução, ou seja, ao receber uma requisição, o Node.js não bloqueia o processo esperando uma resposta, sendo capaz de atender múltiplas requisições simultaneamente. Node.js é uma tecnologia altamente escalável e serve para construir APIs, aplicações em tempo real, *backend* de jogos, dentre outros cenários. Mais informações podem ser encontradas em sua documentação oficial, (FOUNDATION, 2022a).

2.3.8.1 Express

Express, segundo o guia (FOUNDATION, 2022b), é um *framework* mantido pelos mesmos responsáveis do Node.js que consegue atender várias necessidades do projeto, como gerenciamento de requisições HTTP e gerenciamento do sistema de rotas da aplicação, além de possibilitar o tratamento de exceções que podem acontecer dentro dele. Express é uma ferramenta poderosa criada em 2010 por TJ Holowaychuk e ajudará a organizar a aplicação no lado do servidor.

2.3.9 React

React é uma das mais populares bibliotecas Javascript para criação de interfaces de usuário. Desenvolvida pelo Facebook, com documentação que pode ser encontrada em (FACEBOOK, 2022), tem sido usada por grandes empresas como Netflix, Airbnb ou eBay. Esta biblioteca funciona com uma estratégia de dividir os elementos de uma página em componentes para que assim possam ser trabalhados de forma individual e reutilizados em diversas ocasiões. Desta forma, há uma maior padronização e eficiência na construção de aplicações, visto que um mesmo componente é utilizado várias vezes.

2.3.9.1 Next.js

Next.js é um *framework* para React, desenvolvido e mantido pela equipe da Vercel, com documentação encontrada em (VERCEL, 2022a). Esta ferramenta tem como princi-

pal objetivo tornar a aplicação React mais performática e possui várias funcionalidades, como suporte ao Typescript, roteamento automático, de forma que as URL's da aplicação são mapeadas com base em um diretório *pages*, reduzindo codificação, *Hot Code Reloading*, onde alterações feitas no código da aplicação durante desenvolvimento refletem na página de forma automática, otimização de imagens, dentre outros recursos.

2.3.9.2 Chakra UI

Esta ferramenta é uma biblioteca de componentes pré-prontos do React que facilita o desenvolvimento da interface de usuário da aplicação. Fornece uma série de componentes modularizados e simples que otimizam o tempo de desenvolvimento. Possui uma boa documentação encontrada em (VERCEL, 2022b), que exemplifica seus diversos recursos para criação de tabelas, botões, campos de *input* de dados e vários outros componentes para construção de interfaces.

3 Desenvolvimento do Projeto

Este capítulo contém a explicação de como a proposta da nova implementação do Sistema Online de Distribuição de Disciplinas foi desenvolvida.

3.1 Interação da aplicação

O diagrama apresentado na Figura 2 demonstra a forma como a interface do usuário interage com a *API backend*.

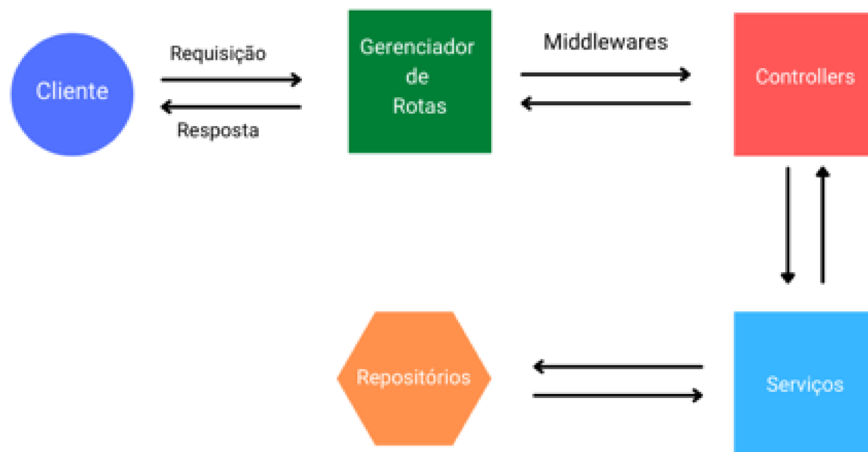

Figura 2 – Diagrama de interação dos módulos da aplicação

A interface do usuário envia uma requisição para a API. Esta, por sua vez, captura a requisição por meio de um gerenciador de rotas que processa qual método HTTP está sendo enviado, como por exemplo um POST ou GET. A partir disso, é invocado o processamento dos *middlewares* e das *controllers* (ambos explicados posteriormente). As *controllers* invocam os serviços que farão o processamento da regra de negócio da aplica-

ção e, caso precisem acessar o banco de dados, o farão por intermédio dos repositórios, que detém a responsabilidade deste acesso. Os repositórios farão as operações necessárias no banco de dados, devolverão uma resposta para os serviços, que por sua vez devolverão uma resposta para as *controllers*, passando pelo gerenciador de rotas que então informará o resultado do processamento da requisição ao cliente.

3.2 Início da API backend

Para iniciar o projeto utilizando o Node.js, Javascript e Typescript, dando vida ao novo Sistema Online de Distribuição de Disciplinas, um primeiro comando foi executado no terminal da máquina de desenvolvimento. Trata-se do comando

```
1 yarn init -y
```

Utilizando-o, o arquivo *package.json*, responsável por detalhar especificações, configurações e dependências do projeto, é criado na raíz do diretório principal. A partir disso, foi criado o diretório *src* para armazenar todo o código fonte relacionado à nova implementação. Antes de partir para o desenvolvimento em si, algumas ferramentas e dependências foram adicionadas ao projeto para que a codificação pudesse seguir de forma padronizada e com os recursos necessários. Inicialmente, o projeto não conta com nenhuma dependência configurada, e, com isso, o comando

```
1 yarn add express
```

foi utilizado para adicionar o *framework* Express, de forma a deixá-lo preparado para ser utilizado posteriormente.

3.2.1 Adicionando Typescript

Como dito antes, o projeto não conta com o Typescript já instalado e configurado, sendo necessário a adição dos seus recursos utilizando o yarn. Para tal, o comando

```
1 yarn add typescript -D
```

foi utilizado. Feito isso, o comando

```
1 yarn tsc --init
```

foi executado para criar o arquivo *tsconfig.json*, que guarda as regras e configurações sobre como o Typescript vai se comportar. No primeiro momento, este arquivo não foi modificado, utilizando os recursos padrões.

3.2.2 Configurando ESLint e Prettier

As dependências referentes ao ESLint e ao Prettier são adicionadas ao projeto utilizando comandos similares aos anteriores. Para o projeto, depois de adicionado, o ESLint seguirá a padronização do Airbnb e, com ela, será definida a utilização de ponto e vírgula ao final de cada linha, a utilização de aspas simples e algumas outras configurações. As dependências utilizadas por ambas as ferramentas também foram integralizadas à aplicação para garantir o funcionamento correto e esperado das mesmas. Ao utilizar estes recursos, os arquivos *prettier.config.js*, *.eslintignore* e *.eslintrc.json* são criados no projeto para manter todo o conjunto de regras e configurações utilizados por elas.

3.2.3 Configurando TypeORM

Para configurar o *framework* no projeto, foram adicionadas as dependências do TypeORM e o driver do PostgreSQL, o banco de dados relacional que a aplicação utiliza. Também foi criado o arquivo *ormconfig.json*, que armazena as informações que o ORM precisa para se conectar ao banco de dados, como o tipo de banco utilizado, seu nome, credenciais utilizadas para acessá-lo, e outras informações. O *framework* trouxe muitas vantagens ao projeto, de forma que o ciclo de vida do banco de dados anda em paridade com o ciclo de vida da aplicação. Com o TypeORM:

- Há registros em ordem cronológica das tabelas e alterações feitas nas mesmas.
- O banco de dados é criado de forma uniforme para todos os desenvolvedores que assumirem o projeto, evitando possíveis discrepâncias em sua construção.
- Há comandos CLI simples que controlam o banco de dados.

3.2.4 Estrutura de diretórios

A estrutura de diretórios foi evoluindo juntamente com a aplicação, o que é bastante comum na esfera de desenvolvimento de software. É importante frisar que, hoje em dia, não há uma estrutura de diretórios padrão para todas as aplicações, mas sim o consenso de que a estrutura deve atender a aplicação de forma que seja intuitivo encontrar os arquivos, além de seguir um raciocínio lógico e padronizado para que faça sentido a disposição dos mesmos. O diretório *src* mantém todo o código produzido como se demonstra na Figura 3.

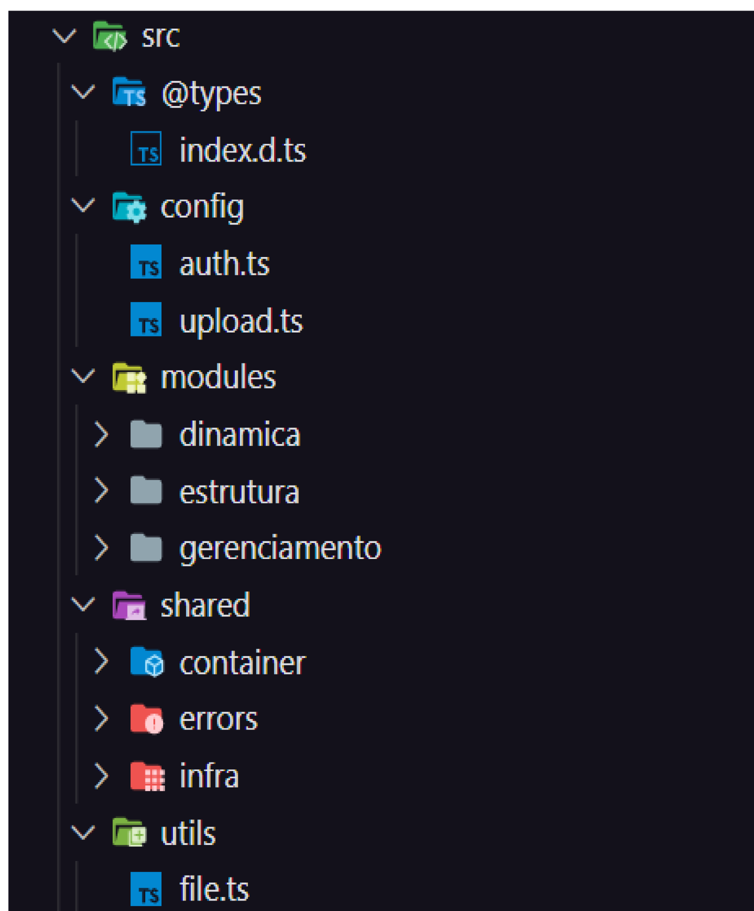

Figura 3 – Diretório raiz do projeto

- *@types* armazena tipagens criadas para atender o projeto e tipagens que sobrescrevem aquelas definidas por alguma biblioteca utilizada.
- *config* armazena arquivos que dizem respeito a configurações gerais dos serviços que o projeto oferece. Configurações e regras de autenticação e de *upload* de arquivos ficam armazenados aqui, por exemplo.
- *modules* armazena os serviços em formatos de módulos que a aplicação disponibiliza. Tudo o que diz respeito às regras de negócio e as funcionalidades do projeto são armazenadas neste diretório. Um exemplo disso é o cadastro de um professor: o diretório *HandleProfessorService*, contida em *modules*, contém toda a regra de negócio relacionada à tabela *professor*.
- *shared* armazena recursos compartilhados entre os módulos. Tratamentos de erro customizados, rotas da aplicação, *middlewares*, serviços de data, dentre outros arquivos compõem a gama de arquivos deste diretório.
- *utils* mantém arquivos com funcionalidades genéricas. A lógica de deletar arquivos que foram enviados para a aplicação, por exemplo, encontram-se neste diretório.

3.2.4.1 Diretório *modules*

Os diretórios dentro de *modules* foram subdivididos em três categorias: dinâmica, estrutura e gerenciamento, além de terem sido construídas de forma padronizada. Observe a figura que mostra a organização desse diretório.

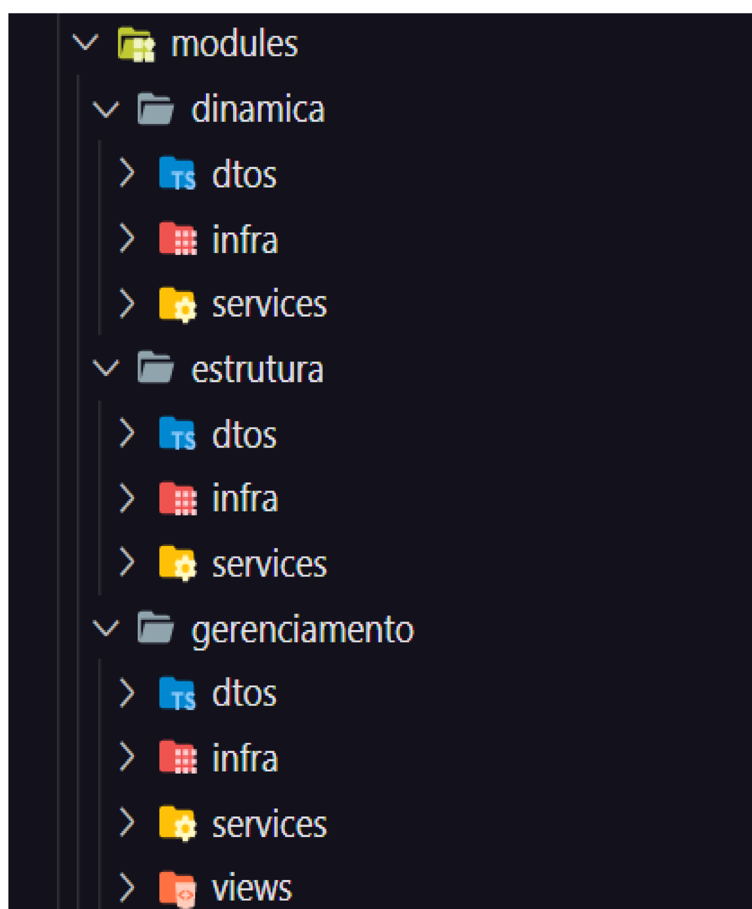

Figura 4 – Diretório modules

- *estrutura* é responsável por armazenar os módulos que trabalham em cima das tabelas estruturantes do banco de dados. Serviços de criação de professor, de disciplinas, construções de semanas, turmas, dentre outras funcionalidades são encontrados neste diretório.
- *dinâmica* é responsável por armazenar a lógica e regra de negócio que trabalham com as tabelas de estrutura da aplicação. Serviços de criação de filas, atribuições de turma, restrições, possibilidades, dentre outros assuntos são encontrados dentro deste diretório.
- Em *gerenciamento* tem-se as regras de negócios relativas à própria aplicação e sua segurança como os serviços que lidam com usuários, atualização de *token* de sessão ou envio de e-mails.

3.2.4.2 Diretório *shared*

O diretório *shared* armazena a lógica que é compartilhada entre os módulos. Provedores de serviços, tratamentos de erros customizados e arquivos de rotas são encontrados neste diretório, como se demonstra na Figura 5.

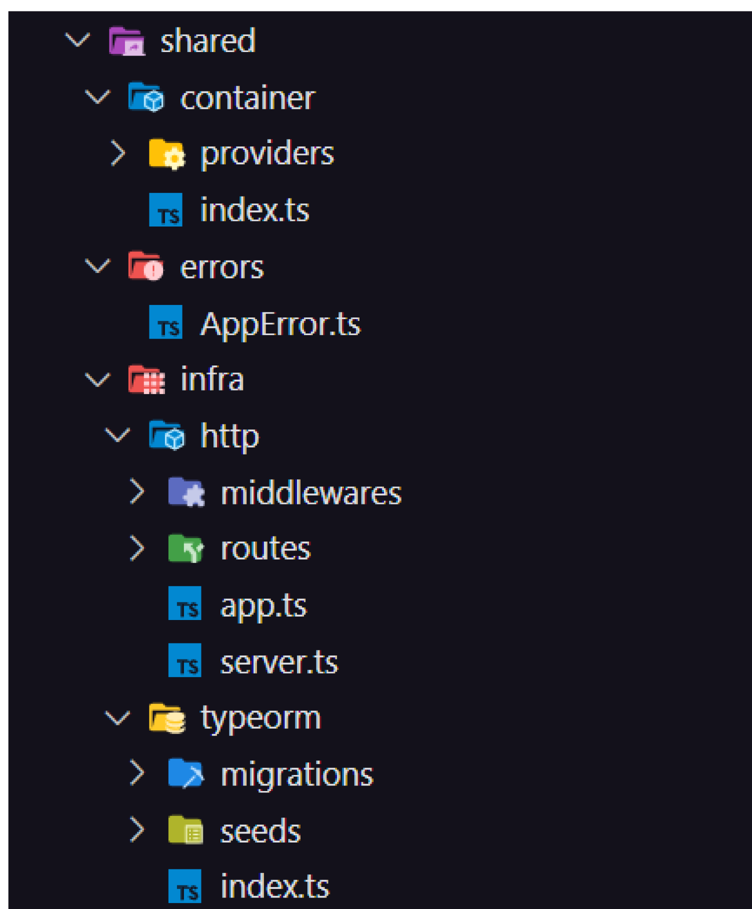

Figura 5 – Diretório shared

- *container* armazena a lógica utilizada para aplicar a injeção de dependência que a aplicação utiliza, evitando nível elevado de acoplamento dentre os módulos. Além disso, provedores de serviços customizados e disponibilizados para os outros módulos também se encontram neste diretório.

- *errors* armazena a lógica de tratamentos de erros customizados para trazer mais liberdade na hora de tratar exceções e disponibilizar mensagens de erro mais amigáveis aos consumidores da API.

- *infra* trata da infraestrutura que a aplicação precisa ter para que os módulos possam ser criados e utilizados de forma correta. Aqui encontra-se a lógica das rotas, dos *middlewares*, das *migrations* utilizadas para criar as tabelas do banco de dados, além do ponto de início da aplicação, ou seja, o primeiro arquivo a ser acessado quando o projeto é executado.

3.2.5 Criação dos módulos

3.2.5.1 Criação das *migrations*

Para iniciar a construção do novo SODD, deve-se ter em mente que, anteriormente, já existia um banco de dados com uma estrutura já definida e que houve a necessidade de ser reavaliada. Depois de avaliada a necessidade da aplicação, a Tabela 1 apresenta as tabelas do banco de dados que foram selecionadas para participar da proposta de nova implementação:

atribuicao_manual	auditoria_fila	auditoria_fila_turma_new
auditoria_prioridade	carga_docente	cenario
auditoria_prioridade	carga_docente	cenario
cenario_fila_turma	curso	disciplina
distribuicao_carga	distribuicoes_possibilidade	etapa
fila	fila_turma_new	horario
ministra	oferta	possibilidades
prioridades	professor	restricoes
semana	semestres	status_distribuicao
status_possibilidades	turma	

Além disso, a Figura 6 mostra o diagrama da estrutura do banco de dados resultante.

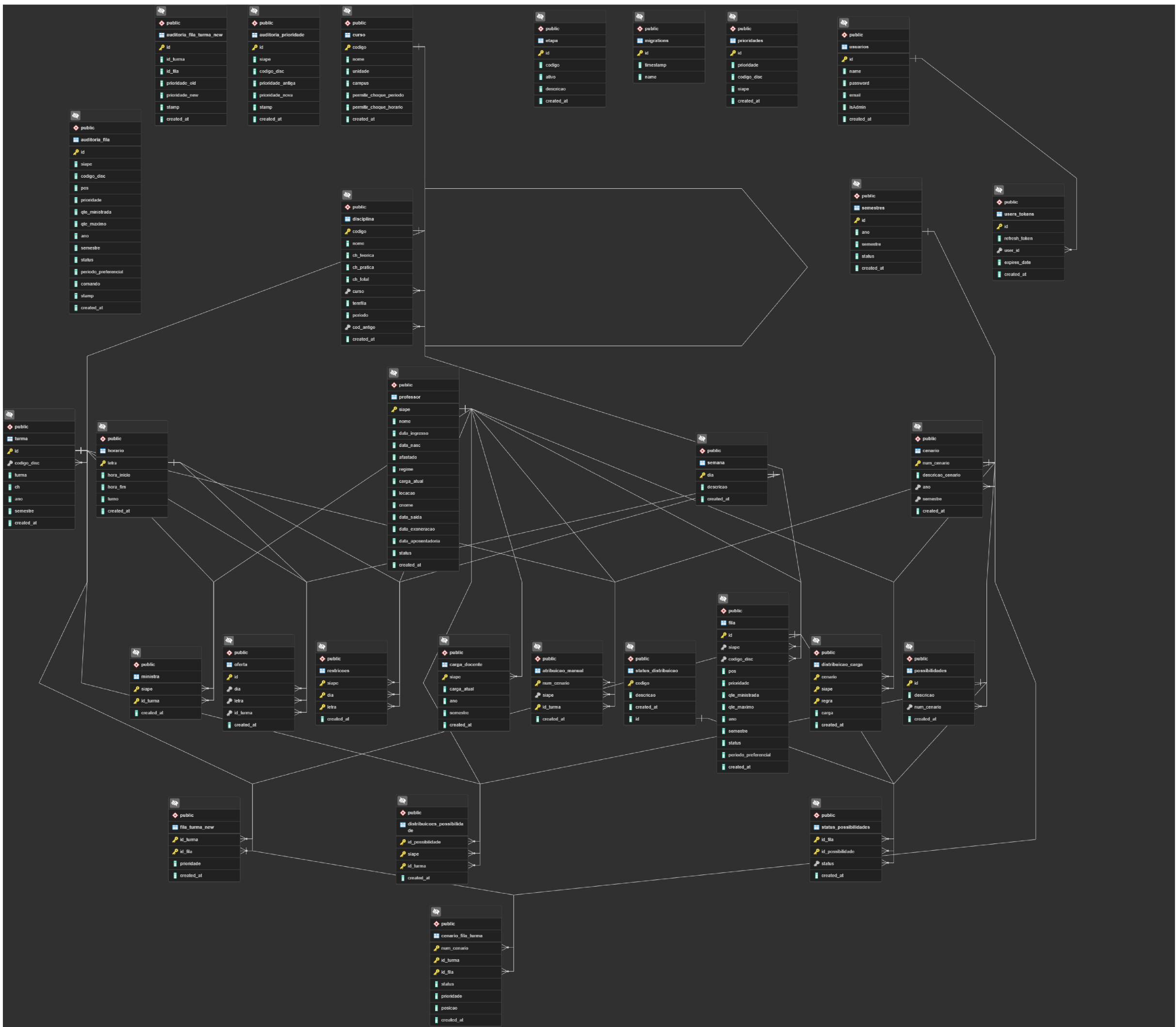

Figura 6 – Estrutura do banco de dados

As tabelas não tiveram grande mudança em sua estrutura, apenas foram criadas utilizando o TypeORM para que, com a sua criação utilizando as *migrations*, houvesse registros de quando e do que foi criado. O processo de criação das tabelas foi feito de forma padrão para todas elas, com cada *migration* descrevendo suas propriedades. Observe, por exemplo, a criação da tabela *semana*: o comando

```
1    yarn typeorm migration:create -n CreateSemana
```

cria o arquivo da *migration* que será utilizado para a implementação da mesma. Observe a criação de *semana*:

```
1 import { MigrationInterface, QueryRunner, Table } from "typeorm";
2
3 export class CreateSemana1630029377547 implements MigrationInterface {
4   public async up(queryRunner: QueryRunner): Promise<void> {
```

```
5      await queryRunner.createTable(
6        new Table({
7          name: "semana",
8          columns: [
9            {
10             name: "dia",
11             type: "char",
12             length: "1",
13             isNullable: false,
14           },
15           {
16             name: "descricao",
17             type: "char",
18             length: "13",
19             isNullable: false,
20           },
21           {
22             name: "created_at",
23             type: "timestamp",
24             default: "now()",
25           },
26         ],
27       })
28     );
29
30     await queryRunner.createPrimaryKey("semana", ["dia"]);
31   }
32
33   public async down(queryRunner: QueryRunner): Promise<void> {
34     await queryRunner.dropPrimaryKey("semana");
35     await queryRunner.dropTable("semana");
36   }
37 }
```

Uma *migration* é definida por dois métodos principais: *up* e *down*. Em *up* define-se o que acontece quando a *migration* roda com sucesso e em *down* define-se o que ocorre quando é necessário reverter o processo descrito em *up* da *migration* atual. Utilizando o *QueryRunner*, que estabelece a conexão com o banco de dados, tem-se o método *createTable* onde pode-se criar uma nova tabela. Com *new Table*, descreve-se o nome da tabela, bem como as colunas que ela irá possuir. No caso de *semana*, tem-se a coluna *dia*, um caractere não nulo de tamanho 1, *descricao*, uma cadeia de caracteres não nula de tamanho 13, e uma terceira coluna chamada *created_at* que guardará a data e a hora de criação do registro. Além disso, é definido que a chave primária será a coluna *dia* por meio do método *createPrimaryKey* de *QueryRunner*. Caso haja algum erro, a criação da tabela é revertida para que o banco de dados não seja afetado.

Para que a *migration* seja executada e a tabela seja criada no banco de dados, utiliza-se o comando

```
1    yarn typeorm migration:run
```

Este comando fará a conexão com o banco de dados e irá construir a tabela da forma que foi descrita na *migration*. Ao final, quando a estrutura da tabela no banco de dados é observada, é identificada a tabela construída conforme vê-se a seguir:

```
1  -- Table: public.semana
2
3  -- DROP TABLE public.semana;
4
5  CREATE TABLE public.semana
6  (
7      dia character(1) COLLATE pg_catalog."default" NOT NULL,
8      descricao character(13) COLLATE pg_catalog."default" NOT NULL,
9      created_at timestamp without time zone NOT NULL DEFAULT now(),
10     CONSTRAINT "PK_0c9064890201d2fa8d16040445c" PRIMARY KEY (dia)
11 )
12
13 TABLESPACE pg_default;
14
15 ALTER TABLE public.semana
16     OWNER to postgres;
```

Realizando este processo para todas as tabelas necessárias, é criada toda a estrutura do banco de dados por meio de código Typescript, organizados e salvos em ordem cronológica por meio do prefixo em formato *timestamp* que é adicionado ao nome da *migration* na hora de sua criação. Além da criação, alterações nas tabelas são feitas através das migrations como se pode observar na Figura 7.

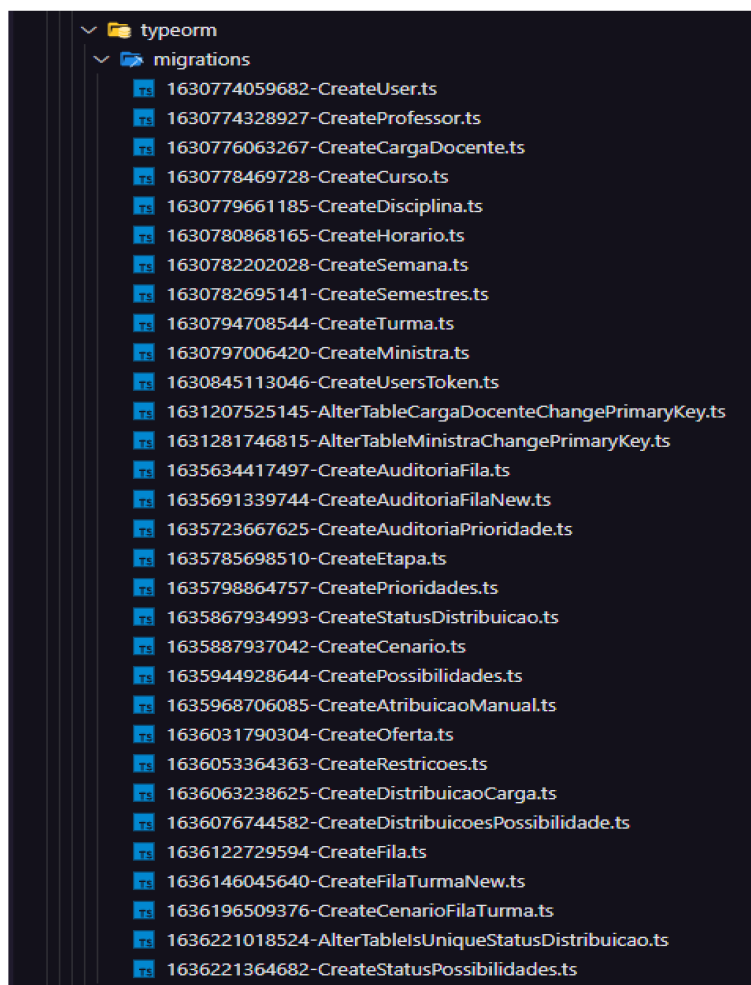

Figura 7 – Migrations criadas para construção das tabelas do banco de dados

3.2.5.2 Repositórios e Entidades

Continuando o exemplo de *semana*, com a tabela criada no banco de dados, a aplicação deve ser capaz de realizar operações sobre ela, respeitando um conjunto de regras de negócio. Aqui, com os conceitos SOLID em mente, começa-se a trabalhar a separação de responsabilidades. Para realizar as operações no banco de dados, tal como criação, edição, deleção ou atualização, os arquivos de repositórios (e suas interfaces) foram criados. Com esta separação, a responsabilidade de comunicação com o banco de dados fica desacoplado dos serviços e controladoras, tornando o código mais coeso e facilitando a manutenção. Além disso, para que o código possa operar com as tabelas, bem como as suas colunas, é necessário o uso de classes *Entity*. Essas classes são uma coleção de campos e são utilizadas para mapear as colunas das tabelas do banco de dados. Observe a *Entity* da tabela *semana*:

```
1   import { Column, CreateDateColumn, Entity, PrimaryColumn } from "typeorm";
2
3   @Entity("semana")
```

```
4   class Semana {
5     @PrimaryColumn()
6     dia: string;
7
8     @Column()
9     descricao: string;
10
11    @CreateDateColumn()
12    created_at: Date;
13  }
14
15  export { Semana };
```

Para identificar uma classe como *Entidade*, utiliza-se a notação *@Entity*, bem como a notação *@Column* é utilizada para identificar os campos. O campo *dia* é mapeado com *@PrimaryColumn* para identificar que ele será a chave primária e o campo created_at é mapeado com *@CreateDateColumn* para que a data de criação seja adicionada à coluna referente na tabela automaticamente, sem a necessidade de mais código.

Com a *Entidade* criada, pode-se partir para a análise do repositório. O repositório é quem detém a responsabilidade de comunicar-se com o banco de dados, e assim, quando classes de serviços precisam realizar alguma operação no banco, elas passam primeiro por eles.

Segue alguns pontos importantes do código de criação do repositório de *semana*, além do método de criação de um registro:

```
1  import { getRepository, Repository } from "typeorm";
2
3  class SemanasRepository implements ISemanasRepository {
4    private repository: Repository<Semana>;
5
6    constructor() {
7      this.repository = getRepository(Semana);
8    }
9
10   async createSemana({ dia, descricao }: ICreateSemanaDTO): Promise<Semana> {
11     const semana = await this.repository.create({ dia, descricao });
12
13     await this.repository.save(semana);
14
15     return semana;
16   }
17 }
```

getRepository e *Repository* são métodos do TypeORM utilizados para manipular

a tabela no banco de dados. A classe *SemanasRepository* implementa uma interface chamada *ISemanasRepository* que contém as operações necessárias até o momento no projeto (no exemplo, têm-se apenas o método *createSemana*). Para que o registro seja salvo de fato no sistema, primeiro é criado uma instância da *Entity Semana* e então esta é salva de fato no banco. Por fim, esta instância é retornada para que o método que esteja utilizando *createSemana* possa ter ciência de que tudo ocorreu sem problemas.

A estrutura de repositórios segue o mesmo padrão para todas as tabelas criadas, análogo ao processo que é feito para a criação das *migrations* e das *Entidades*. Ou seja, para cada tabela que se deseja criar, há uma *migration*, uma *Entidade* e um repositório para que seja possível criá-la, realizar consultas, além de atualizar ou deletar registros.

3.2.5.3 Construção dos Serviços

Serviços são responsáveis por aplicar as regras de negócio dos requisitos e também por se comunicar com os repositórios. Por exemplo, quando o usuário tenta criar um registro de professor com um determinado SIAPE no banco de dados, a regra de negócio que verifica se já existe um professor com o mesmo identificador e, em caso positivo, barra a criação, situa-se na camada de serviços.

Afim de garantir baixo nível de acoplamento entre os módulos, seguindo os princípios SOLID, os serviços foram construídos seguindo o padrão de desenvolvimento chamado injeção de dependência. No presente momento, com o projeto relativamente pequeno, pode não ser claro o uso deste padrão de desenvolvimento, mas à medida que o SODD crescer e se tornar mais robusto, gerenciar as dependências entre as classes pode se tornar complicado. O módulo de professor pode ter duas, três dependências, por exemplo, e sem um padrão, tanto a garantia do funcionamento correto quanto a escrita de testes para ele podem se tornar muito difíceis. A ferramenta utilizada no projeto para gerenciar as injeções de dependências é chamada TSyringe e para adicioná-la foi utilizado o comando:

```
1  yarn add tsyringe
```

Com a dependência adicionada, foi criado um arquivo que registra as instâncias dos repositórios para que possam ser utilizadas posteriormente. Todos os registros foram feitos como segue o exemplo de *semana*:

```
1  import { container } from "tsyringe";
2
3  import { ISemanasRepository } from "../../modules/estrutura/
4  infra/typeorm/repositories/interfaces/ISemanasRepository";
5  import { SemanasRepository } from
       "../../modules/estrutura/infra/typeorm/repositories/SemanasRepository";
6
7  container.registerSingleton<ISemanasRepository>(
```

```
8    "SemanasRepository",
9    SemanasRepository
10 );
```

Com o registro da instância do repositório feito, pode-se injetá-lo na classe de serviço de *semana*. Seguem os pontos de destaque da implementação:

```
1  import { inject, injectable } from "tsyringe";
2  import { AppError } from "../../../../shared/errors/AppError";
3  import { Semana } from "../../infra/typeorm/entities/Semana";
4  import { ISemanasRepository } from
       "../../infra/typeorm/repositories/interfaces/ISemanasRepository";
5
6
7  interface IRequest {
8    dia: string;
9    descricao: string;
10 }
11
12 @injectable()
13 class HandleSemanaService {
14   constructor(
15     @inject("SemanasRepository")
16     private semanasRepository: ISemanasRepository
17    ) {}
18
19   async create({ dia, descricao }: IRequest): Promise<Semana> {
20     const semanaExistent = await this.semanasRepository.queryByDia(dia);
21
22     if (semanaExistent) {
23       throw new AppError("Dia ja cadastrado!");
24     }
25
26     const semana = await this.semanasRepository.createSemana({
27       dia,
28       descricao,
29     });
30
31     return semana;
32   }
33 }
```

Utilizando os decoradores *inject* e *injectable*, que são declarações identificadas por “@” e utilizadas para adicionar funcionalidades extras a uma classe, método ou parâmetro, a classe chamada *HandleSemanaService* pode ter acesso ao repositório registrado no arquivo *container* e, com isso, pode acessar os métodos disponibilizados. Para isso, utiliza-

se *semanasRepository* para guardar a instância do repositório de *semana*. O exemplo demonstra o método *create* que verifica se um dia específico da semana já foi cadastrado e, em caso positivo, lança uma exceção com a mensagem de erro apropriada. Em caso de sucesso, o método do serviço acessa o método *createSemana* do repositório, repassa as informações recebidas como parâmetros, e armazena o retorno na constante *semana*, que é retornado ao final da execução. Se houvessem mais regras de negócio, condições ou tratativas, estas ficariam na classe de serviço, como mencionado anteriormente. O serviço também foi responsável pela atualização, leitura e deleção dos registros da tabela, e este modelo de construção foi utilizado para todas as tabelas do projeto.

3.2.5.4 Construção das classes *Controllers*

As classes *controllers* da aplicação têm por responsabilidade receber a requisição, capturar os parâmetros que vêm com ela, repassá-los para o serviço correspondente, aguardar o retorno deste serviço e retornar o resultado, juntamente com o código HTTP adequado. Para manipular tanto a requisição quanto a sua resposta, utiliza-se o Express, que havia sido adicionado no início do projeto. Seguem pontos de destaque de *HandleSemanaController*:

```
1   import { Request, Response } from "express";
2   import { container } from "tsyringe";
3
4   import { HandleSemanaService } from "./HandleSemanaService";
5
6   class HandleSemanaController {
7     async create(request: Request, response: Response): Promise<Response> {
8       const { dia, descricao } = request.body;
9
10      const handleSemanaService = container.resolve(HandleSemanaService);
11
12      const semana = await handleSemanaService.create({ dia, descricao });
13
14      return response.status(201).json(semana);
15    }
16  }
```

Para cada operação que se deseja realizar, cria-se um método referente na *controller*. No exemplo, pode-se ver o método *create*. Este método recebe os parâmetros desestruturados vindos do corpo da requisição, os repassa para *HandleSemanaService* e aguarda a criação de um registro na tabela *semana*. Por fim, caso nenhuma exceção tenha sido lançada pela classe de serviço, o método retorna o registro em formato *json*, juntamente com o código HTTP 201 informando que a criação foi realizada com sucesso utilizando os parâmetros repassados.

3.2.5.5 Construção das Rotas

Para construir as rotas, utilizou-se o Router, um objeto do Express. Com este objeto é possível nomear rotas e repassar as funções que devem ser executadas quando estas rotas são acessadas. A seguir, algumas rotas de *semana*:

```
import { Router } from "express";

const semanasRoutes = Router();

const handleSemanaController = new HandleSemanaController();

semanasRoutes.post(
  "/",
  ensureAuthenticated,
  ensureAdmin,
  handleSemanaController.create
);

semanasRoutes.get(
  "/",
  ensureAuthenticated,
  ensureAdmin,
  handleSemanaController.read
);

export { semanasRoutes };
```

Como é possível observar, cada rota possui um método HTTP referente. No exemplo demonstra-se POST e GET, mas *semanasRoutes* também é utilizado para tratar PATCH e DELETE. É repassado para os métodos *post* e *get* o nome da rota, alguns *middlewares* que serão explicados posteriormente e os métodos correspondentes da classe *HandleSemanaController*. Por fim, esta classe é exportada para que a classe principal de roteamento, que armazena todas as rotas, possa utilizá-la. A classe principal, que contém todas as rotas, é configurada desta forma:

```
import { Router } from "express";

import { semanasRoutes } from "./semanas.routes";

const mainRouter = Router();

mainRouter.use("/semanas", semanasRoutes);
```

Assim, quando os métodos da API referentes a *semana* são consumidos, eles são acessados por meio da rota “semanas”.

Com isso, foi observado por meio do exemplo de *semana* a metodologia utilizada para a criação de todos os módulos. Para todas as tabelas, foram criadas as devidas *migrations*, repositórios, serviços, classes controladoras e rotas, mantendo um padrão de codificação e respeitando ao máximo as boas práticas de programação.

3.2.6 Criação de Sessões

Criar uma sessão no SODD significa informar e-mail e senha de um usuário que esteja devidamente cadastrado e obter um *token* de acesso, além de informações do próprio usuário e um outro *token* que será utilizado para renovar a sessão, para que assim este usuário possa consumir a API por meio das rotas disponibilizadas. Um *token* dentro da aplicação tem duração finita, ou seja, após um determinado período de tempo, a sessão do usuário expira e, dependendo da situação, ele deve renová-la ou criar uma nova. Esta abordagem é utilizada para aumentar a segurança da aplicação, não permitindo que o usuário fique *logado* mais tempo que o necessário.

Para melhorar a experiência com a aplicação, foi utilizada a abordagem de atualização de *token*. Nela, toda vez que o *token* expira, para que o usuário não precise inserir suas credenciais novamente, o próprio sistema é capaz de renovar a sessão utilizando um segundo *token*, de duração maior, gerado no momento da autenticação e salvo no banco de dados. Assim, a aplicação, ao identificar que a sessão do usuário expirou, pode renová-la, estando de posse desse *token* de renovação. Desta forma, o usuário pode permanecer *logado* na aplicação, porém o *token* que é gerado e utilizado para autenticar a sessão, muda conforme o tempo.

3.2.6.1 Autenticando o Usuário

Nesta seção serão destacados alguns pontos da implementação do método de autenticação, que está contido junto com outros métodos relacionados ao usuário no arquivo *HandleUserService*:

```
1  import { compare, hash } from "bcrypt";
2  import { sign } from "jsonwebtoken";
3  import auth from "../../../../config/auth";
4
5  @injectable()
6  class HandleUserService {
7    constructor(
8      @inject("UsersRepository")
9      private usersRepository: IUsersRepository,
10     @inject("UsersTokensRepository")
11     private usersTokensRepository: IUsersTokensRepository,
12     @inject("DayjsDateProvider")
13     private dateProvider: IDateProvider
```

```
14    ) {}
15
16    async authenticate({ email, password }: IRequest): Promise<IResponse> {
17      const user = await this.usersRepository.queryByEmail(email);
18      const { expires_in_token, expires_in_refresh_token, secret_token,
19        secret_refresh_token, expires_refresh_token_days } = auth;
20
21      if (!user) { throw new AppError("Email ou senha incorreta", 401); }
22
23      const passwordMatch = await compare(password, user.password);
24
25      if (!passwordMatch) { throw new AppError("Email ou senha incorreta", 401); }
26
27      const token = sign({}, secret_token, {
28        subject: user.id,
29        expiresIn: expires_in_token,
30      });
31
32      const refresh_token = sign({ email }, secret_refresh_token, {
33        subject: user.id,
34        expiresIn: expires_in_refresh_token,
35      });
36
37      const refresh_token_expires_date = this.dateProvider.addDays(
38        expires_refresh_token_days
39      );
40
41      await this.usersTokensRepository.create({
42        user_id: user.id,
43        refresh_token,
44        expires_date: refresh_token_expires_date,
45      });
46
47      const tokenReturn: IResponse = {
48        token,
49        user: {
50          name: user.name,
51          email: user.email,
52        },
53        refresh_token,
54      };
55
56      return tokenReturn;
57    }
58  }
```

Primeiro, as dependências: foram utilizados os métodos de comparação e criação de

hash da biblioteca do Node.js chamada *bcrypt*. Também utilizou-se o método de assinatura da biblioteca *jsonwebtoken* para assinar o *token* utilizando a chave secreta de posse da aplicação. A chave do *token* principal, do *token* de atualização e seus respectivos tempos de duração estão armazenados em um arquivo de configuração a parte chamado *auth*. Há também outras importações necessárias para o método de autenticação, sendo omitidos no exemplo para efeito didático.

O método, chamado *authenticate*, recebe como parâmetros o e-mail e a senha do usuário e verifica, primeiro, se o e-mail consta no sistema. Em caso positivo, verifica-se se a senha informada é a senha correspondente à aquele e-mail. Para tal verificação, utiliza-se o *compare* do *bcrypt* pois quando se é criado um usuário, sua senha é criptografada para sua segurança. Caso o usuário tenha informado um e-mail e senha válida, cria-se um *token* que é assinado pela aplicação, além de criar o *token* de atualização de sessão. Nota-se que o *token* em si não é salvo no banco de dados, diferente do *token* de atualização de sessão que, por possuir tempo de expiração maior, deve constar no banco e assim, quando o *token* do usuário expirar e ele solicitar uma renovação de sessão, a aplicação possa ter conhecimento da sua sessão anterior e gerar um novo *token* de acesso a ele. Após assinar os *tokens* com suas respectivas chaves, o método compõe o retorno com os dados do usuário e estes mesmos *tokens*, e então devolve este retorno terminando sua execução.

3.2.6.2 Renovando a Sessão

Para renovar a sessão do usuário, permitindo o acesso à aplicação sem que ele precise informar suas credenciais, foi implementado o serviço *HandleRefreshTokenService*, mostrado a seguir:

```
1  async refresh(token: string): Promise<ITokenResponse> {
2  const { email, sub } = verify(token, auth.secret_refresh_token) as IPayload;
3  const user_id = sub;
4
5  const userToken = await this.usersTokensRepository.queryByUserIdAndRefToken(
6    user_id,
7    token
8  );
9
10 if (!userToken) {
11   throw new AppError("Refresh Token does not exists!");
12 }
13
14 await this.usersTokensRepository.deleteById(userToken.id);
15
16 const refresh_token = sign({ email }, auth.secret_refresh_token, {
17   subject: sub,
18   expiresIn: auth.expires_in_refresh_token,
```

```
19      });
20
21      const expires_date = this.dateProvider.addDays(
22        auth.expires_refresh_token_days
23      );
24
25      await this.usersTokensRepository.create({
26        expires_date,
27        refresh_token,
28        user_id,
29      });
30
31      const newToken = sign({}, auth.secret_token, {
32        subject: user_id,
33        expiresIn: auth.expires_in_token,
34      });
35
36      return {
37        refresh_token,
38        token: newToken,
39      };
40    }
```

O método recebe como parâmetro o *token* de renovação, verifica sua autenticidade dada a palavra secreta de posse da aplicação, e armazena o *id* do usuário e seu *e-mail* em duas contantes. Estes valores são utilizados para verificar no banco de dados se já existe um *token* de renovação salvo e, caso não exista, informar ao usuário. Em caso positivo, é deletado o *token* existente e criado um novo, com uma nova data de expiração, que será a data da requisição somado ao tempo de período de expiração configurado para um *token* de renovação. Por fim, é gerado um novo *token* de acesso e assinado, para ser retornado ao usuário, juntamente com seu novo *token* de renovação.

3.2.7 *Middlewares*

A versão anterior do SODD não contava com um gerenciamento de usuários, onde não era possível autenticar um usuário utilizando uma senha, por exemplo. Na proposta de nova implementação, foi criado o módulo de usuários utilizando o processo demonstrado anteriormente para que assim, por meio de de um endereço de e-mail e uma senha fornecida por ele, o sistema possa ser capaz de identificar quem o está acessando e possa criar uma sessão para que sua API possa ser consumida por aquele indivíduo.

Em um contexto geral, *middlewares* são métodos que conseguem tratar tanto as requisições quanto as respostas de uma rota antes ou depois do seu processamento. No Express, eles têm acesso ao objeto de solicitação (*Request*) e no objeto de resposta (*Res-*

ponse). Para o projeto, foram criados os *middlewares* que garantem que o usuário que está *logado* é um usuário administrador, e que garantem que o usuário está autenticado e é, de fato, cadastrado no sistema. Estes métodos podem ser aplicados em cada rota, não necessariamente sendo obrigatórios para todas elas (por exemplo, nem toda rota precisa ser disponibilizada apenas para usuários administradores).

3.2.7.1 *Middleware* de Autenticação

No mundo de desenvolvimento web, existem várias abordagens para o problema da autenticação. Seja autenticando por credenciais como usuário e senha, autenticação por dois fatores, sessões ou *tokens*, o fato de a aplicação validar se um usuário é, de fato, cadastrado no sistema e tem suas devidas permissões, torna-a menos exposta a possíveis ataques e usuários indevidos com intenções maliciosas, aumentando assim, a sua segurança. Para o projeto, foi utilizado o JSON Web Token (JWT), uma maneira de realizar a autenticação entre pares por meio de um *token* assinado digitalmente utilizando uma chave secreta. A ideia do *middleware* de autenticação implementado no SODD é a seguinte: o sistema conta com uma rota de sessão, em que o usuário informa seu e-mail cadastrado e senha e, com isso, avalia se essas credenciais estão corretas para que assim, um *token* de acesso seja gerado para este usuário e ele possa utilizar os seus recursos. Ou seja, em posse deste *token*, todas as requisições que este usuário venha a fazer, como a criação ou a atualização de um registro, ele precisará informá-lo para que a aplicação consiga garantir que ele está devidamente autenticado e tem a permissão de fazer tais operações.

Os *middlewares* no Express geralmente utilizam três parâmetros: Request, Response e NextFunction. Este último parâmetro é responsável por chamar a função seguinte após o término da execução. A seguir, alguns pontos importantes da implementação do método de validação:

```
1   import { Request, Response, NextFunction } from "express";
2   import { verify } from "jsonwebtoken";
3
4   import auth from "../../../../config/auth";
5   import { AppError } from "../../../errors/AppError";
6
7   interface IPayload {
8     sub: string;
9   }
10
11  export async function ensureAuthenticated(
12    request: Request,
13    response: Response,
14    next: NextFunction
15  ): Promise<void> {
```

```
16      const authHeader = request.headers.authorization;
17
18      if (!authHeader) {
19        throw new AppError("Token nao encontrado!", 401);
20      }
21
22      const [, token] = authHeader.split(" ");
23
24      try {
25        const { sub: user_id } = verify(token, auth.secret_token) as IPayload;
26
27        request.user = {
28          id: user_id,
29        };
30
31        next();
32      } catch {
33        throw new AppError("Token invalido!", 401);
34      }
35    }
```

O *middleware* começa importando as dependências necessárias para implementar a lógica, como as tipagens do Express para identificar os parâmetros, o método que verifica a autenticidade do *token*, o arquivo que contém as informações para tal verificação e o método de tratativa de erros personalizado. O método *ensureAuthenticated* começa recebendo o *token* do cabeçalho da requisição e primeiro valida se há algum conteúdo neste *token* ou não. Em caso positivo, este tipo de *token* é identificado pelo prefixo *Bearer* seguido de seu conteúdo em si e, como o método deseja verificar só o conteúdo principal, ele utiliza a função *split* para armazenar o conteúdo na constante *token* e ignorar tal prefixo. Então, é utilizado o método de verificação da biblioteca *jsonwebtoken* para checar se aquele conteúdo foi assinado utilizando a chave secreta de posse da aplicação para que assim, possa invocar a próxima função, e continuar o fluxo da requisição. Nota-se que é incorporado em *request* o identificador do usuário: isto é feito para que a aplicação conheça o usuário que está acessando aquela rota. Em caso negativo, ou seja, em que o *token* é inválido, é retornado o código HTTP 401, proibindo o usuário de continuar o processo, além de uma mensagem amigável.

3.2.7.2 *Middleware* de Administrador

Além de validar a autenticação, é validado também se o usuário que iniciou a sessão é um usuário administrador para que a aplicação possa ter ainda mais controle sobre as suas rotas. Isto é realizado pelo *middleware* *ensureAdmin*:

```
1    import { Request, Response, NextFunction } from "express";
```

```
2
3  import { UsersRepository } from
     "../../../../modules/gerenciamento/infra/typeorm/repositories/UsersRepository";
4  import { AppError } from "../../../errors/AppError";
5
6  export async function ensureAdmin(
7    request: Request,
8    response: Response,
9    next: NextFunction
10 ) {
11   const { id } = request.user;
12
13   const usersRepository = new UsersRepository();
14
15   const user = await usersRepository.queryById(id);
16
17   if (!user.isAdmin) {
18     throw new AppError("Usuario nao e um administrador!", 403);
19   }
20
21   return next();
22 }
```

Este método busca o usuário que está acessando a rota no banco de dados utilizando o *UsersRepository*, repassando o próprio *id* do usuário que é incorporado na requisição através do *middleware* de autenticação. Caso o usuário não seja um administrador, uma exceção é lançada com código HTTP 403, proibindo a requisição de continuar. Em caso de sucesso, a próxima função é chamada e o fluxo ocorre normalmente.

3.2.8 Testes com Jest

Testar a aplicação é essencial para garantir sua qualidade e bom funcionamento. O SODD é um projeto que tende a crescer e ficar mais robusto, com mais requisitos e funcionalidades sendo adicionadas gradualmente e, com este crescimento, cresce também a dificuldade de manter tudo funcionando corretamente, integrando entre si e com sistemas externos. Além disso, pensando na expansão da aplicação, também ficará mais difícil realizar testes manuais das funcionalidades, trazendo a tona a necessidade de automatizar esses testes para abranger mais casos de uso. A automatização dos testes consiste em escrever rotinas que testam as funcionalidades que o sistema implementa, trazendo o *input* e *output* para um ambiente controlado e assim, identificar como a regra de negócio está se comportando. Esta automatização foi implementada com o uso do *Jest*, e foi construída para realizar dois tipos de testes: unitários e de integração. Para cada serviço que interage com uma tabela no banco de dados há o seu devido *script* de teste construído de forma

padronizada. São destacados alguns pontos importantes utilizando o exemplo do serviço da tabela *semana*.

3.2.8.1 Testes unitários

Testes unitários verificam o comportamento de porções menores da aplicação. Assim, ao invés de testar a aplicação como um todo, são testados os métodos que contém regras de negócio importantes para o software. Para os testes do SODD, assim como de outras aplicações, não é recomendável interagir com o banco de dados real, pois isto implicaria em criação, deleção ou edição de registros reais. Além disso, como o intuito desse teste é verificar o funcionamento correto do serviço e não do repositório, pode-se criar um repositório para testes que não interage com o banco em si. Com isso, segue os pontos de destaque de *HandleSemanaService.spec*, o arquivo de testes unitários de *semana*:

```
1  describe("Handle CRUD operations related to semana", () => {
2    let semanasRepository: SemanasRepositoryTestMock;
3    let handleSemanaService: HandleSemanaService;
4
5    beforeEach(() => {
6      semanasRepository = new SemanasRepositoryTestMock();
7      handleSemanaService = new HandleSemanaService(semanasRepository);
8    });
9
10   it("Should be able to create a semana record", async () => {
11     const semana = await handleSemanaService.create({
12       dia: "0",
13       descricao: "Domingo",
14     });
15
16     expect(semana.dia).toBe("0");
17     expect(semana.descricao).toBe("Domingo");
18   });
19
20   it("Should not be able to create a semana with same day", async () => {
21     await expect(async () => {
22       await handleSemanaService.create({
23         dia: "1",
24         descricao: "Domingo",
25       });
26
27       await handleSemanaService.create({
28         dia: "1",
29         descricao: "Segunda",
30       });
31
32       await handleSemanaService.create({
```

```
33        dia: "1",
34        descricao: "Segunda",
35      });
36    }).rejects.toBeInstanceOf(AppError);
37  });
38 });
```

Os testes com o Jest começam com o método *describe*, que dão o nome ao teste e iniciam uma função para encapsular os métodos de teste em si. O método *beforeEach* executa antes de cada método de teste *it* e instancia o serviço de *semana* utilizando o repositório de testes. Os métodos *it* também possuem um identificador e uma função. A criação de um registro de *semana* pode ser realizada normalmente ou pode ser barrada caso haja um registro com o mesmo dia. Assim, os métodos "*Should be able to create a semana record*" e "*Should not be able to create a semana with same day*" verificam se o serviço está se comportando normalmente invocando o método de criação e verificando o resultado com *expect*. No caso de erro, foi utilizado *rejects.toBeInstanceOf* para identificar se o método disparou um erro do tipo *AppError*, a classe customizada da aplicação para tratativa de erros.

3.2.8.2 Testes de integração

Os testes de integração verificam se os módulos interagem corretamente entre si. A aplicação testa as *controllers* para verificar se o comportamento das rotas seguirá de acordo com o esperado. Para construir este tipo de teste, foi utilizado o *supertest*, uma ferramenta que é capaz de criar requisições do Express. Observe o arquivo de testes de integração da tabela *semana*, *HandleSemanaController.spec*:

```
1  let connection: Connection;
2
3  describe("Handle CRUD routes related to semana", () => {
4    beforeAll(async () => {
5      connection = await createConnection();
6      await connection.runMigrations();
7
8      const id = uuidV4();
9      const password = await hash("admin", 8);
10
11     await connection.query(
12       `INSERT INTO usuarios(id, name, email, password, "isAdmin", created_at)
13       values('${id}', 'admin', 'sodd_tcc@outlook.com', '${password}', 'true',
             'now()')`
14     );
15   });
16
```

```
17      afterAll(async () => {
18        await connection.dropDatabase();
19        await connection.close();
20      });
21
22      it("Should be able to create a new semana record", async () => {
23        const responseToken = await request(app).post("/sessions").send({
24          email: "sodd_tcc@outlook.com",
25          password: "admin",
26        });
27
28        const { token } = responseToken.body;
29
30        const response = await request(app)
31          .post("/semanas")
32          .send({
33            dia: "0",
34            descricao: "Domingo",
35          })
36          .set({
37            Authorization: `Bearer ${token}`,
38          });
39
40        expect(response.status).toBe(201);
41      });
42    });
```

Diferente dos testes unitários, os testes de integração querem garantir a funcionalidade total de um fluxo. O exemplo procura testar desde a requisição até a criação do registro no banco de dados. Como dito antes, não é recomendável interagir com o banco de dados real, e por isso, um banco de dados de teste, com a estrutura semelhante ao banco da aplicação, é criado seguindo o modelo descrito nas *migrations* e logo ao fim dos testes, destruído. Além disso, é necessário garantir que a sessão estará autenticada, e por isso, um usuário administrador é criado neste banco de testes.

O método "*Handle CRUD routes related to semana*" utiliza *beforeAll* para definir que, antes de todos os métodos, seja criado uma conexão com o banco de dados de teste, suas tabelas, e um usuário administrador. Define também, utilizando *afterAll*, que este banco de dados seja apagado e a conexão fechada. O método "*Should be able to create a new semana record*" cria uma sessão utilizando o usuário *sodd_tcc@outlook.com* e, com o *token* de acesso, faz uma requisição do tipo *post* para a rota de *semana*, para que assim, um registro seja criado. No fim, é verificado se a requisição teve como retorno o código HTTP 201, indicando que o processo ocorreu com sucesso.

3.2.8.3 Cobertura de código

Jest é capaz de gerar um relatório que exibe o quanto o código da aplicação foi coberta, para que assim seja possível analisar quais casos de uso foram testados, e quais cenários ainda faltam ser cobertos. Os testes na aplicação são executados com o comando

```
1 yarn test:windows
```

para ambiente Windows, ou

```
1 yarn test:linux
```

para ambientes Linux. O resultado de sua execução pode ser visualizado conforme a Figura 8.

```
=============================== Coverage summary ===============================
Statements   : 72.17% ( 4395/6090 )
Branches     : 95.56% ( 516/540 )
Functions    : 76.81% ( 265/345 )
Lines        : 72.17% ( 4395/6090 )
================================================================================

Test Suites: 56 passed, 56 total
Tests:       263 passed, 263 total
Snapshots:   0 total
Time:        161.286 s
Ran all test suites.
Done in 166.93s.
```

Figura 8 – Resumo dos testes

Além disso, o relatório é exibido em um arquivo *HTML* conforme Figura 9.

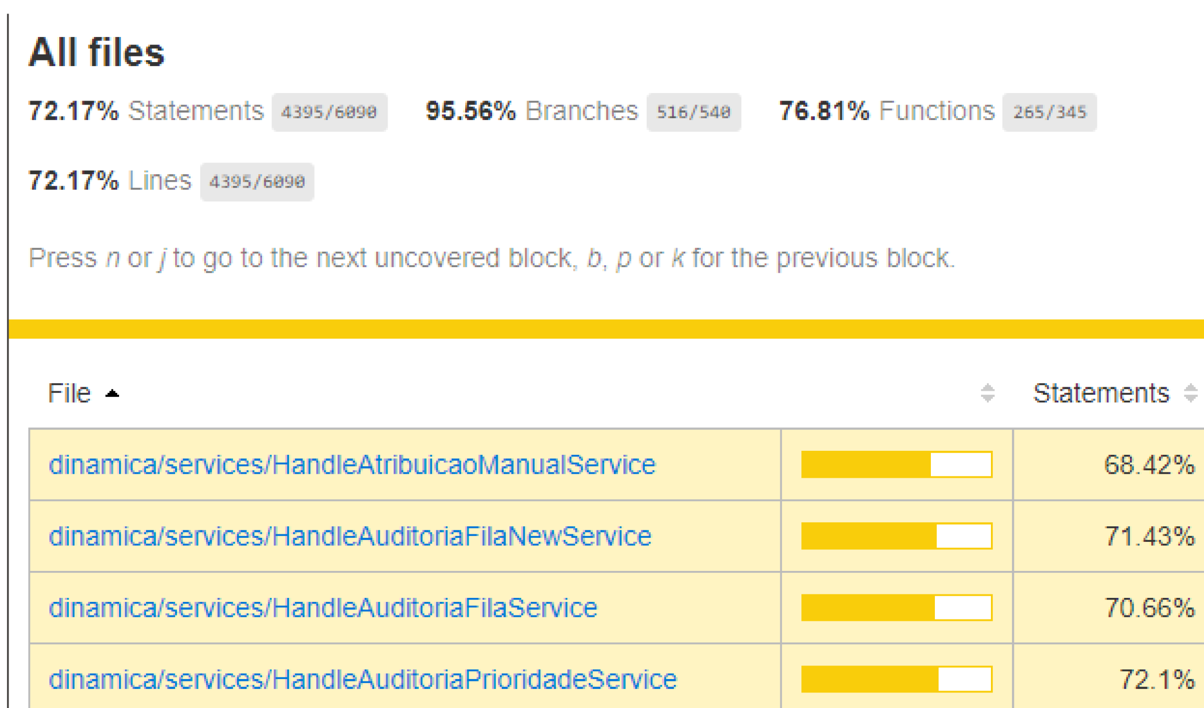

Figura 9 – Relatório dos testes

Atualmente, o código possui mais de 70% de cobertura pelos testes automatizados, porcentagem considerada satisfatória e que cobre grande parte dos cenários propostos pelas regras de negócio.

3.2.9 Documentação com Swagger

Quando uma API é criada para ser consumida, seja por uma aplicação que utiliza React ou outra biblioteca ou linguagem de programação, é essencial que ela seja bem documentada para que os desenvolvedores conheçam a forma de interagir com suas funcionalidades e as regras que devem seguir. Ao possuir esta documentação, desenvolvedores têm o conhecimento dos parâmetros que devem enviar ao realizar uma requisição, bem como o que devem esperar de resposta e, ao ter este conhecimento disponível, o tempo de desenvolvimento de novos recursos é reduzido. Para o projeto foi utilizado o Swagger, uma poderosa ferramenta de documentação de APIs REST.

Para adicionar o Swagger ao projeto, além de adicionar suas dependências, foi definido que a aplicação irá utilizá-lo desta forma:

```
1  import cors from "cors";
2  import express, { Request, Response, NextFunction } from "express";
3  import swaggerFile from "../../../swagger.json";
4  import swaggerUi from "swagger-ui-express";
5
```

```
6    const app = express();
7
8    app.use("/documentation", swaggerUi.serve, swaggerUi.setup(swaggerFile));
```

Desta forma, é informado que a rota para a documentação é *documentation* e sua configuração estará presente no arquivo *swaggerFile*. Este recurso ainda foi pouco explorado dentro da aplicação, porém, é possível ver o seu potencial com poucas definições através da interface gráfica gerada, como exibido na Figura 10.

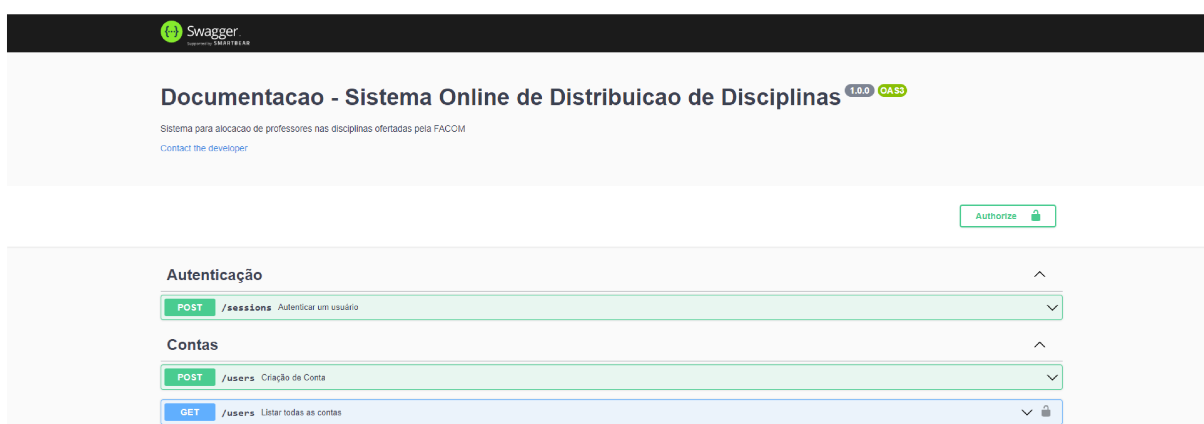

Figura 10 – Interface Swagger

Swagger será utilizado para documentar todas as rotas futuramente. Seja a criação de dados no banco de dados, ou a forma de se autorizar na aplicação, haverá um modelo de como fazer isso, e isto será possível pois a configuração do Swagger foi realizada.

3.3 Início da API *frontend*

Uma boa aplicação, além de ter um *backend* poderoso, deve possuir uma interface que seja útil ao usuário, tendo boa performance e usabilidade amigável. A ideia de utilizar React no SODD parte do princípio de que ele oferece muitas ferramentas para construir interfaces que são tão elegantes quanto funcionais. Sendo assim, para dar início ao *frontend* utilizando React, Javascript e Typescript, foi utilizado o comando:

```
1    yarn create next-app frontend
```

Assim, é criada uma aplicação React já com as dependências do Next.js instaladas e prontas para serem utilizadas. Com isso, é possível fazer a configuração do Typescript e das ferramentas de formatação e padronização de código.

3.3.1 Adicionando Typescript

Tal como foi feito na API *backend*, foi adicionado o Typescript no *frontend* para garantir maior qualidade de código. Sua instalação foi feita com:

```
1 yarn add typescript @types/react @types/node -D
```

Após isso, o comando

```
1 yarn tsc --init
```

adicionou ao projeto o arquivo *tsconfig.json* que contém as regras que o Typescript segue para o projeto, que também pode ser customizado a medida que o projeto se desenvolve.

3.3.2 Configurando ESLint e Prettier

No React também é utilizado ESLint e Prettier para padronizar o código seguindo uma convenção do mercado. Tal como foi feito no *backend*, foram adicionados ao projeto as dependências necessárias para o funcionamento das ferramentas. Além disso, foram criados os arquivos *prettier.config.js*, *.eslintignore* e *.eslintrc.json* para que as tecnologias tenham seu conjunto de regras definidos e assim, seguidos. Por fim, foi definido a convenção do Airbnb para padronização do código.

3.3.3 Módulos do projeto

Apesar de ainda pouco explorado no projeto, React traz a ideia de modularização dos componentes, afim de torná-los reutilizáveis. Alguns módulos do projeto foram criados, conforme segue a estrutura de diretórios exibida na Figura 11.

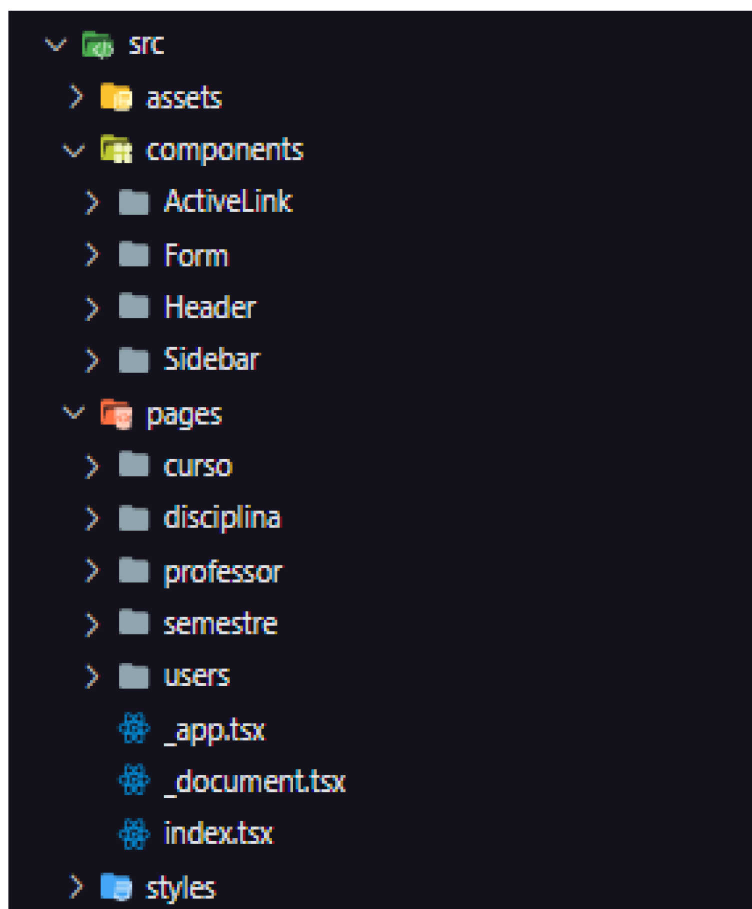

Figura 11 – Raíz do projeto frontend

- *assets* armazena recursos que serão utilizados pela aplicação para auxiliar na construção das interfaces, tal como imagens e logotipos.
- *components* armazena os componentes reutilizáveis utilizados na interfaces. Podem ser formulários, barras de navegação, botões, tudo o que pode ser utilizado em mais de uma página.
- *pages* armazena as páginas da aplicação com as quais o usuário interage.
- *styles* contém arquivos *.css* utilizados para personalizar as páginas e os componentes.

3.3.4 Criação das páginas

As páginas foram criadas utilizando Chakra UI, uma tecnologia cujas dependências foram adicionadas de forma análoga às outras do projeto. Para aumentar a padronização, as páginas foram criadas de forma semelhante, utilizando os componentes criados para atendê-las. Segue a tela de *curso* na Figura 12.

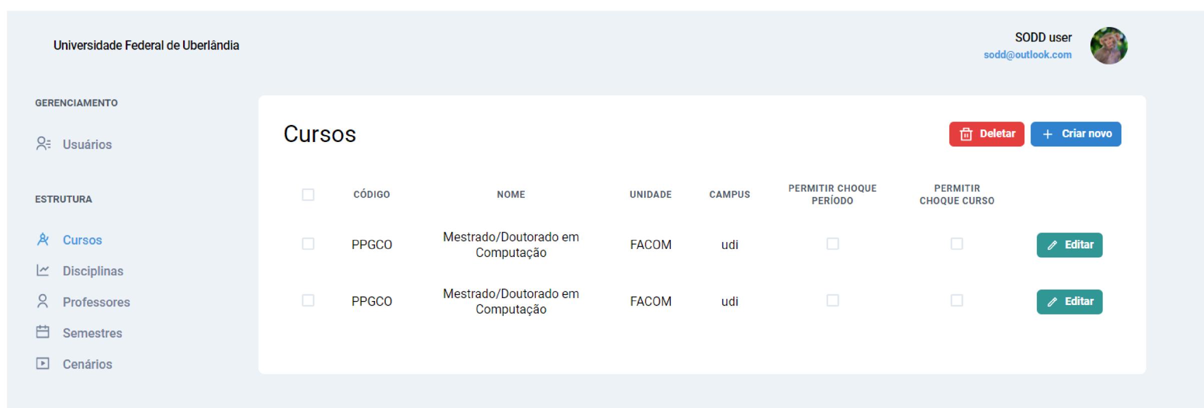

Figura 12 – Tela de visualização de cursos

A aplicação conta com um menu lateral para acessar as páginas construídas. Conta também com um cabeçalho que exibe informações do usuário e a identificação da universidade. No meio, há o painel onde é possível visualizar os registros de *curso*, bem como as ações de editar, criar um novo registro ou deletar um registro existente. Segue o layout para criação e edição nas Figuras 13 e 14, respectivamente.

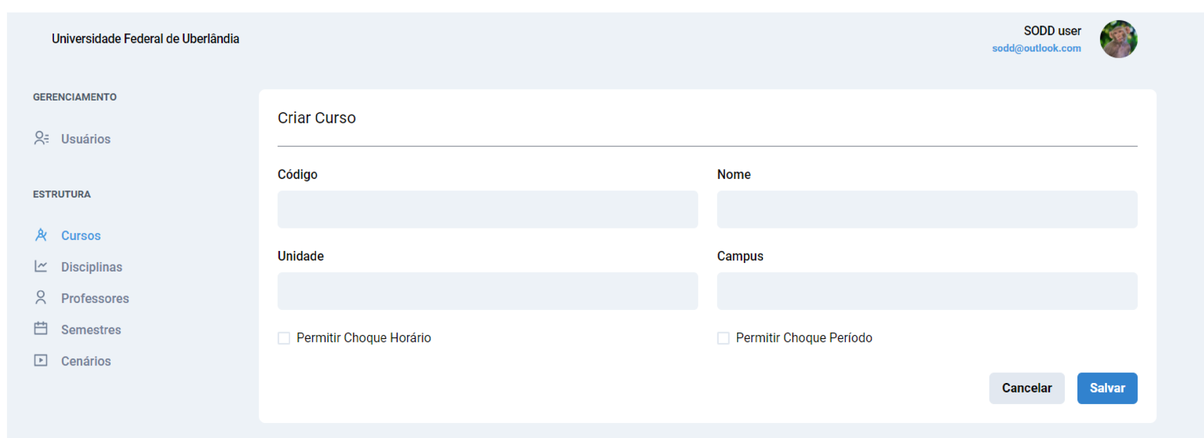

Figura 13 – Tela de Criação de curso

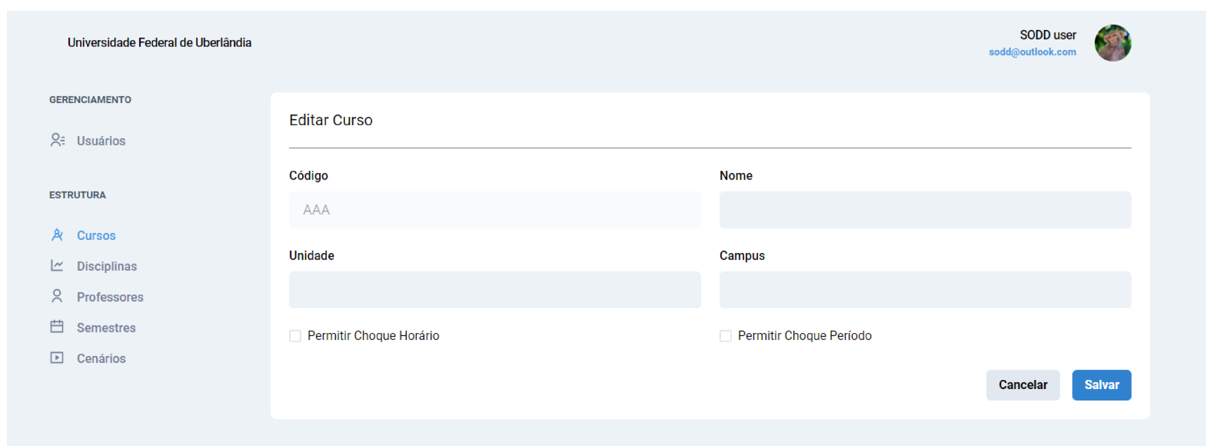

Figura 14 – Tela de Edição de curso

Esta padronização de layout serviu para a criação das outras páginas, como o módulo de Disciplinas ou de Professores. Até o presente momento, a aplicação *frontend* ainda não se conecta com o *backend*, sendo que os dados visualizados nas figuras são demonstrados de forma estática. O roteamento entre as páginas, como sair da página de curso e ir para a disciplina, é feita facilmente utilizando o Next.js pois ele identifica as páginas dentro do diretório *pages* e assim consegue rotear o usuário utilizando o nome dos subdiretórios contidas neste diretório. A construção da interface utiliza vários elementos do Chakra UI, como Heading, Button, Icon, Table e vários outros. Assim, não foi necessário criar algo do completo zero, e sim utilizado *templates* pré-prontos que se adéquam à necessidade.

4 Resultados

Com o intuito de iniciar o desenvolvimento de um novo projeto, utilizando novas tecnologias a fim de alcançar uma qualidade melhor de código, o desenvolvimento da nova implementação do SODD renderam resultados não só para o presente trabalho, mas para o futuro do projeto também. Conforme visto ao longo desta monografia, foram aplicadas diversas ferramentas e técnicas que deram início a um projeto que poderá ser escalável, com testes automatizados e maior segurança. Este capítulo relata os resultados obtidos deste desenvolvimento.

4.1 Padronização de código

Com a configuração e uso de ferramentas como ESLint e Prettier, realizou-se um desenvolvimento padronizado, que segue um conjunto de regras utilizadas no mercado e configuradas na aplicação. Com isto, desenvolvedores futuros conseguirão entender melhor como a aplicação se comporta e conseguirão desenvolver novos requisitos ou dar manutenção no projeto com mais facilidade.

4.2 Histórico do banco de dados

Com a configuração do TypeORM manteve-se o histórico da criação da estrutura das tabelas do banco de dados. Desta forma, desenvolvedores futuros terão acesso ao modelo das tabelas criadas na própria API, bem como o modo que foram implementadas e, além disso, caso precisem alterar o banco, haverá a possibilidade de rastreio de quem o fez, e quando o fez.

4.3 Automatização de testes no *backend*

Com os testes automatizados é possível garantir a qualidade do código, bem como o seu bom funcionamento. Grande parte das rotas construídas até o momento possuem cobertura satisfatória de testes, com *scripts* que pouparão trabalho manual de testar o comportamento da aplicação.

4.4 Rotas Seguras

Com a implantação dos recursos de segurança no novo SODD, apenas usuários autorizados poderão interagir com o sistema. Desta forma, garantimos que a criação de

uma fila, a criação de um novo registro de professor ou qualquer outra funcionalidade seja acessa apenas por usuários cadastrados e que fizeram *login* na aplicação, conseguindo acesso por uma porção de tempo finita.

4.5 Criação do CRUD das tabelas

Para as tabelas utilizadas pela aplicação, foi criada toda a estrutura necessária que contemplou desde a criação da sua estrutura utilizando o TypeORM, passando pelos serviços, criação de rotas, até os testes em si. Com isso, a base necessária para que requisitos mais complexos possam ser construídos já está devidamente pronta. Foram criadas rotas para acesso à todas as tabelas, possibilitando criação, edição, deleção e visualização de registros contidos no banco de dados.

4.6 Configuração do ambiente *frontend*

A nova implementação do SODD teve a configuração de sua interface iniciada. Para tal, foram adicionadas ferramentas de padronização de código (ESLint, Prettier, Typescript), além de ferramentas para construção de páginas e roteamento das mesmas. Algumas páginas foram criadas, porém, ainda servindo dados estáticos. Com isso, um ambiente propício para desenvolvimentos futuros foi configurado e está pronto para desenvolvedores futuros conseguirem implementar novos requisitos de forma padronizada e com tempo de desenvolvimento reduzido.

4.7 *Layout* de páginas construído

Apesar de totalmente mutável, um *layout* para o modo como os componentes vão se dispor para o usuário foi definido. Desta forma, utilizando componentes como barras laterais ou menus de navegação que já foram desenvolvidos, páginas futuras poderão ser criadas de forma padronizada e que sejam intuitivas ao usuário. No presente trabalho foram criados os *layouts* das telas:

- Visualização, edição e criação de registros da tabela *users*.
- Visualização, edição e criação de registros da tabela *curso*.
- Visualização, edição e criação de registros da tabela *disciplina*.
- Visualização, edição e criação de registros da tabela *professor*.
- Visualização, edição e criação de registros da tabela *semestre*.

5 Conclusão

Propor uma nova implementação para uma aplicação utilizando novas tecnologias abre portas para a aplicação de boas práticas de programação, além de utilizar ferramentas novas que podem agregar tanto no desenvolvimento quanto na performance do projeto. SODD é um projeto iniciado por volta de 2015, e por mais que desenvolvedores passados que mantiveram a aplicação tenham se esforçado para manterem as melhores práticas possíveis, é inevitável o surgimento de *bugs*, código não utilizado, tabelas não utilizadas, e técnicas de codificação que impactam na padronização do código. Reforçando que problemas deste tipo não são exclusivos deste projeto, mas de diversos softwares existentes atualmente. Com o desenvolvimento desta nova implementação, um ambiente novo com ferramentas e recursos para implementação de novos requisitos está pronto para ser utilizado, permitindo a evolução do SODD.

Devido a robustez da aplicação, sua quantidade de requisitos e o número de programadores que passaram por ela, houve dificuldade em implantar todas as funcionalidades já existentes. Porém, com o ambiente pronto e com a descrição destes requisitos detalhados em trabalhos anteriores, desenvolvedores futuros poderão implementar os requisitos restantes e, em breve, a nova versão do SODD poderá tomar o lugar do seu predecessor. O desenvolvimento deste trabalho irá abrir portas para que novas funcionalidades sejam implementadas de forma segura e padronizada, utilizando as melhores práticas e o que há de mais recente no mercado, fazendo com que a aplicação se torne cada vez mais poderosa e segura.

Apesar de muito desenvolvimento ter sido realizado, como a configuração dos ambientes *backend* e *frontend*, a implementação de novas funcionalidades para aumento de segurança, a estrutura de testes automatizados e operações nas tabelas, ainda há muito o que ser feito. Como passos para o futuro, SODD deve continuar os passos dados pelo presente trabalho: a implementação dos requisitos restantes, o enriquecimento do arquivo de documentação, e a conexão da interface gráfica com a API, além de desenvolver o restante das páginas de criação, visualização e edição para cada tabela não mencionada no Capítulo 4. Feito isso, desenvolver a estratégia de implantação do projeto em um servidor seguro, para que a comissão responsável por realizar a distribuição das disciplinas possa utilizar uma aplicação nova e totalmente pensada para atendê-la.

Por fim, todo o código da aplicação se encontra em uma organização privada hospedada no GitHub. Para solicitar acesso, deve-se contactar o orientador do projeto, Bruno Augusto Nassif Travençolo através de seu endereço eletrônico.

Referências

AUTH0. *Guia JWT*. 2022. <https://jwt.io/introduction>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 16.

DAMASCENO, M. I. R. *Manutenção do Sistema Online para Distribuição de Disciplina*. 2021. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação – Universidade Federal de Uberlândia, Uberlândia). Citado na página 11.

FACEBOOK. *Documentação React*. 2022. <https://pt-br.reactjs.org/docs/getting-started.html>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 17.

FOUNDATION, O. *Documentação Node.js*. 2022. <https://nodejs.org/dist/latest-v16.x/docs/api/>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 17.

FOUNDATION, O. *Guia Express*. 2022. <https://expressjs.com/pt-br/guide/routing.html>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 17.

LOCATELLI, J. A. *Sistema Online para Distribuição de Disciplinas*. 2015. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação – Universidade Federal de Uberlândia, Uberlândia). Citado na página 8.

MARTIN, R. *Código Limpo: Habilidades Práticas do Agile Software*. Alta Books, 2019. ISBN 9788550811482. Disponível em: <https://books.google.com.br/books?id=GXWkDwAAQBAJ>. Citado 3 vezes nas páginas 8, 12 e 13.

MICROSOFT. *Documentação Typescript*. 2022. <https://www.typescriptlang.org/docs/>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 14.

MOZILLA. *Guia Javascript*. 2022. <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Introduction>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 14.

SMARTBEAR. *Documentação Swagger*. 2022. <https://swagger.io/>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 15.

SOURCE, F. O. *Documentação Jest*. 2022. <https://jestjs.io/pt-BR/>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 15.

TYPEORM. *Documentação TypeORM*. 2022. <https://typeorm.io/>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 15.

VALENTE, M. T. Engenharia de software moderna (livro digital). 2020. Citado na página 13.

VERCEL. *Documentação Next.js*. 2022. <https://nextjs.org/docs>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 17.

VERCEL. *Guia Chakra*. 2022. <https://chakra-ui.com/docs/getting-started>. [Online; Acessado em 05 de Janeiro de 2022]. Citado na página 18.