UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

JOÃO HENRIQUE DE ARAÚJO SOUSA

CONSTRUÇÃO DE UM WEBAPP PARA EXIBIÇÃO E CONTROLE DE CONTEÚDO DA CLÍNICA SÃO GONÇALO

SOBRAL

2021

JOÃO HENRIQUE DE ARAÚJO SOUSA

CONSTRUÇÃO DE UM WEBAPP PARA EXIBIÇÃO E CONTROLE DE CONTEÚDO DA CLÍNICA SÃO GONÇALO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em ENGENHARIA DA COMPUTAÇÃO do CAMPUS SOBRAL da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em ENGENHARIA DA COMPUTAÇÃO.

Orientador: Prof. Dr. Iális Cavalcante de Paula Júnior

SOBRAL

2021

JOÃO HENRIQUE DE ARAÚJO SOUSA

CONSTRUÇÃO DE UM WEBAPP PARA EXIBIÇÃO E CONTROLE DE CONTEÚDO DA CLÍNICA SÃO GONÇALO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em ENGENHARIA DA COMPUTAÇÃO do CAMPUS SOBRAL da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em ENGENHARIA DA COMPUTAÇÃO.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Iális Cavalcante de Paula Júnior (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Xxxxxxx Xxxxxx Xxxxxxx
Universidade do Membro da Banca Dois (SIGLA)

Prof. Dr. Xxxxxxx Xxxxxx Xxxxxxx
Universidade do Membro da Banca Três (SIGLA)

Prof. Dr. Xxxxxxx Xxxxxx Xxxxxxx
Universidade do Membro da Banca Quatro (SIGLA)

AGRADECIMENTOS

À Deus, pela vida e saúde em tempos tão difíceis.

Aos familiares, pelo apoio incondicional para que nunca desistisse dos meus sonhos.

Ao Prof. Dr. Iális Cavalcante, pela excelente orientação e colaboração com o trabalho elaborado.

Ao meu amigo e colega de turma Hennan Lewis, por todo o conhecimento compartilhado, reflexões, críticas e sugestões ao longo da realização deste trabalho.

RESUMO

O presente documento apresenta a teoria, metodologia e aplicabilidade de um projeto web para gerenciar informações e fornecer uma identidade visual para a Clínica São Gonçalo, localizada na cidade de Sobral/CE. A escolha dessa clínica deu-se devido à presença de familiares próximos em sua administração. O projeto foi divido em três etapas, sendo a primeira a consolidação de uma moderna estrutura para o servidor backend por meio do ambiente *NodeJS*, que permite uma alta escalabilidade com o uso de API's REST, utilização de middlewares e bibliotecas de uso específico. Concomitantemente ao desenvolvimento do backend, teve-se a organização dos dados de pacientes e profissionais habilitados na clínica com o banco de dados relacional *PostgreSQL*, aliado ao uso do *querybuilder KnexJS* para a criação das tabelas requeridas no banco. E por fim, o uso da tecnologia *VueJS* para o frontend foi importante na implementação das páginas web necessárias, pois, além de fornecer um ambiente padronizado para centralizar os códigos, o *VueJS* facilita o entendimento de cada componente visual estabelecido e sua atribuição na aplicação. O projeto concluído ainda possibilita, por conta das tecnologias usadas, atualizações pontuais ou de grande porte, graças a simples manutenção do código e fácil inserção de novas funcionalidades.

Palavras-chave: *NodeJS. VueJS. PostgreSQL. KnexJS.*

ABSTRACT

This document presents the theory, methodology and applicability of a web project to manage information and provide a visual identity for São Gonçalo Clinic. This clinic has been chosen due to the presence of family members on administration. The project was divided into three stages, the first being the consolidation of a modern structure for the backend server through the NodeJS environment, which allows high scalability with the use of REST APIs, use of Middlewares and specific-use libraries. Concomitantly with the development of the backend, there was an organization of the data of patients and professionals qualified in the clinic with the PostgreSQL relational database, combined with the use of the KnexJS querybuilder to create the required tables in the database. And finally, the use of VueJS technology for the frontend is important when implementing the web pages it requires, as in addition to providing a standardized environment to centralize the codes, VueJS facilitates the understanding of each established visual component and its definition in the application. The completed project still allows, due to the used technologies, punctual or large-scale updates thanks to the simple maintenance of the code and easy insertion of new functions.

Keywords: NodeJS. VueJS. PostgreSQL. KnexJS.

LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura do NodeJS. 14
Figura 2 – Modelagem do banco de dados, com as tabelas de usuários, artigos, categorias, pacientes, profissionais e formações. 18
Figura 3 – Rotas e métodos HTTP utilizados na API users. 20
Figura 4 – Método POST. 21
Figura 5 – Método GET. 21
Figura 6 – Método PUT. 21
Figura 7 – Método DELETE. 22
Figura 8 – Dados do payload que são armazenados. 23
Figura 9 – Função validadeToken. 23
Figura 10 – Função remove. 24
Figura 11 – Funções toTree e getTree. 25
Figura 12 – Método *get* na API artigos. 25
Figura 13 – Método *getById* na API de artigos. 26
Figura 14 – Função *getByCategory* na API de artigos. 26
Figura 15 – Visualização do cabeçalho com usuário logado. 28
Figura 16 – Menu central exibido em lista vertical. 29
Figura 17 – Componente da tela de login. 29
Figura 18 – Componente *userDropdown*. 30
Figura 19 – Página introdutória da clínica. 31
Figura 20 – Estilização CSS da área de conteúdo. 31
Figura 21 – Componente *slider* inicial da página inicial. 32
Figura 22 – Parte inferior da tela inicial. 33
Figura 23 – Menu lateral para pesquisa de categorias e artigos. 33
Figura 24 – Área administrativa. 34
Figura 25 – Exibição de profissionais. 35
Figura 26 – Categoria com seus artigos exibida na tela. 36
Figura 27 – Artigo exibido na tela. 36
Figura 28 – Rodapé da aplicação. 37
Figura 29 – Exemplo de ficha de acompanhamento de paciente da Clinica Espaço Recriar. 38
Figura 30 – Página de download do NodeJS. 41

Figura 31 – NodeJS adicionado às variáveis de ambiente do Windows. 41
Figura 32 – Página de download do *PostgreSQL*. 42
Figura 33 – Criação do arquivo package.json com as informações sobre a aplicação. . . 42
Figura 34 – Exemplo de instalação do pacote *ExpressJS*. 44
Figura 35 – Configuração básica do servidor no arquivo index.js. 44
Figura 36 – Arquivo .env e suas informações. 45
Figura 37 – Criação da tabela users por meio de uma migration. 46
Figura 38 – Migration para a tabela de artigos. 46
Figura 39 – Migration para categoria de artigos. 46
Figura 40 – Arquivos gerados automaticamente pelo Vue CLI. 47
Figura 41 – Distribuição de pastas na aplicação *frontend*. 48

SUMÁRIO

1 INTRODUÇÃO 10

1.1 Motivação 10

1.2 Objetivos Gerais e Específicos 11

1.2.1 Objetivo geral 11

1.2.2 Objetivos específicos 11

2 TECNOLOGIAS WEB: FUNDAMENTAÇÃO TEÓRICA 12

2.1 NODEJS: Origem e características 13

2.2 FRAMEWORK VUEJS: Origem e características 14

2.3 POSTGRESQL: Origem e características 15

3 METODOLOGIA 17

3.1 CRIAÇÃO DAS API'S DO BACKEND 17

3.1.1 MODELAGEM DO BANCO DE DADOS 17

3.1.2 API DE VALIDAÇÃO 18

3.1.3 API DE USUÁRIOS 19

3.1.4 API DE AUTENTICAÇÃO 22

3.1.5 API DE CATEGORIAS 23

3.1.6 API DE ARTIGOS 25

3.1.7 API DE FORMAÇÃO 27

3.1.8 API DE PROFISSIONAIS 27

3.1.9 API DE PACIENTES 27

3.2 COMPONENTES DO WEPAPP 27

3.2.1 Cabeçalho 28

3.2.2 Tela de login/Acesso restrito 29

3.2.3 Menu dropdown de usuário 30

3.2.4 Introdução sobre a Clínica 30

3.2.5 Contéudo 31

3.2.6 Tela de Início 32

3.2.7 Menu Lateral 33

3.2.8 Área Administrativa 34

3.2.9 Exibição de Profissionais 35

3.2.10 *Exibição de Categorias e Artigos* 35
3.2.11 *Rodapé* 37
4 **TESTE DE ACEITAÇÃO E AVALIAÇÃO DO SISTEMA** 38
5 **CONCLUSÕES E TRABALHOS FUTUROS** 39
REFERÊNCIAS 40
APÊNDICES 41
APÊNDICE A – ESTRUTURAÇÃO DO PROJETO *BACKEND* 41
A.1 **DEPENDÊNCIAS UTILIZADAS NO BACKEND** 43
A.2 **CONFIGURANDO O POSTGRESQL E KNEXJS** 44
APÊNDICE B – ESTRUTURAÇÃO DO PROJETO *FRONTEND* 47
B.1 **DEPENDÊNCIAS UTILIZADAS NO FRONTEND** 49

1 INTRODUÇÃO

O desenvolvimento de aplicações web utilizando Javascript, CSS(Cascading Style Sheet) e HTML(HyperText Markup Language) recebe constantemente melhorias acerca da produtividade, escalabilidade e otimização de recursos, a fim de facilitar a realização de projetos. Com o advento dos frameworks e as novas versões das linguagens utilizadas, como o novo ECMAScript 6 que padroniza o JavaScript, toda a estrutura de uma aplicação pode ser organizada de maneira coesa seguindo um fluxo proposto pelo programador, seja para facilitar a manutenção ou implementação de novas funcionalidades.

O referido trabalho propõe a criação de uma aplicação web para a Clínica São Gonçalo, localizada na cidade de Sobral e administrada por membros da família. Haverá um sistema para cadastro de pacientes, profissionais/funcionários da clínica e uma área administrativa para escrita de artigos com assuntos correlatos aos tipos de atendimentos da clínica. Serão utilizados alguns dos recursos mais populares entre os programadores no que se refere a desenvolvimento web, com a utilização do framework *VueJS* aliado ao modelo de layout *Flexbox* para criar o FrontEnd, um SGBD(Sistema de Gerenciamento de Banco de Dados) chamado *PostgreSQL*, para a construção do banco de dados, associado a uma dependência chamada *KnexJS* para a criação das tabelas e a plataforma NodeJS, que, aliada ao framework *ExpressJS* para servidores web, permite a execução de códigos JavaScript no desenvolvimento do BackEnd da aplicação. As referidas tecnologias, a metodologia aplicada e toda a construção do software será apresentada adiante neste documento.

1.1 Motivação

O presente trabalho teve como impulso a necessidade de facilitar o controle de algumas operações na clínica, permitindo que, por meio das tecnologias que serão apresentadas, estabeleça-se um controle sobre dados de pacientes e profissionais, assim como propor um ambiente estruturado na rede, aliado a uma estética moderna que apresenta a instituição para quem não a conhece, com informações por meio de artigos sobre assuntos correlatos à clínica, além de dados sobre a sua estrutura, atendimento e profissionais à disposição.

1.2 Objetivos Gerais e Específicos

1.2.1 Objetivo geral

Construir uma aplicação web para uma clínica, com funções de exibição de conteúdo via artigos e visualização da estrutura física e humana da clínica, além de assegurar um ambiente restrito de acesso administrativo para gerenciar o controle de informações de profissionais, pacientes, artigos, usuários e quaisquer outras informações associadas à clínica.

1.2.2 Objetivos específicos

- Desenvolver o *BackEnd*, especificando todas as rotas e tratamento de dados;
- Elaborar uma área administrativa para login de usuário administrador;
- Implementar um sistema para cadastro de pacientes e profissionais a partir da área administrativa;
- Designar um espaço para a escrita de artigos, utilizando um editor de texto e um espaço próprio para torná-los públicos e visíveis na aplicação;
- Implementação de um banco de dados com o *PostgreSQL*, utilizando o *KnexJS* para definir as estrutura das tabelas;
- Construir um *FrontEnd* responsivo com as tecnologias *VueJS* e *Flexbox* para implementar todas as telas da aplicação.

2 TECNOLOGIAS WEB: FUNDAMENTAÇÃO TEÓRICA

Antes da criação da primeira API (*Application Programming Interface*) do projeto, é necessário esclarecer alguns pontos sobre o que é uma API, as diretrizes da arquitetura REST (*Representational State Transfer*) e a importância desses elementos na criação do *backend*. De acordo com (SOUSA, 2020): "API trata-se de um conjunto de requisições que permite a comunicação de dados entre aplicações. Para isso, a API utiliza requisições HTTP responsáveis pelas operações básicas necessárias para a manipulação dos dados". As operações em questão são:

- POST: usado para salvar, no banco de dados, qualquer dado cadastrado;
- PUT: altera algum dado de usuário previamente salvo no banco;
- GET: utilizado para retornar toda e qualquer informação registrada no banco, sem alterá-la. A exibição de usuários em lista, por exemplo, se dá por meio de um GET;
- DELETE: apaga qualquer dado já registrado no banco. Pode ou não ser uma ação reversível, dependendo apenas de como a exclusão dos dados é feita.

Ainda segundo (SOUSA, 2020), o padrão REST adotado nas API's estabelece algumas restrições, são elas:

- Cliente-servidor: deve haver uma separação entre as aplicações do servidor e cliente;
- Sem estado: as requisições são independentes e executam uma única ação por vez;
- Cache: a API deve utilizar o cache para evitar chamadas recorrentes ao servidor, garantindo uma maior velocidade de acesso aos dados;
- Interface uniforme: consiste em alguns conceitos que determinam que os recursos (API's) utilizados em uma aplicação devem ser identificados de forma padronizada e imutável, com o uso de URL's representativas aliadas aos métodos HTTP já elucidados. Os recursos possuem as requisições validadas através de um controle de respostas autodescritivas adequadas para cada situação, como um erro no servidor(código 5xx) ou solicitação bem sucedida pelo cliente (código 2xx).

Para a construção da aplicação web, temos diversas tecnologias aplicadas. Mas antes de entender essas tecnologias, é necessário configurar o ambiente de desenvolvimento, tanto para construção do *backend* quando do *frontend*. Toda a configuração inicial do projeto pode ser encontrada nos apêndices A e B desse trabalho. A seguir, será abordada toda a base necessária que envolve essas tecnologias, desde sua origem até a consolidação de alguns preceitos essenciais em suas composições. A arquitetura envolvida e a abordagem de cada tecnologia dentro do

projeto serão demonstradas adiante, para firmar os conceitos estabelecidos nesse tópico.

2.1 NODEJS: Origem e características

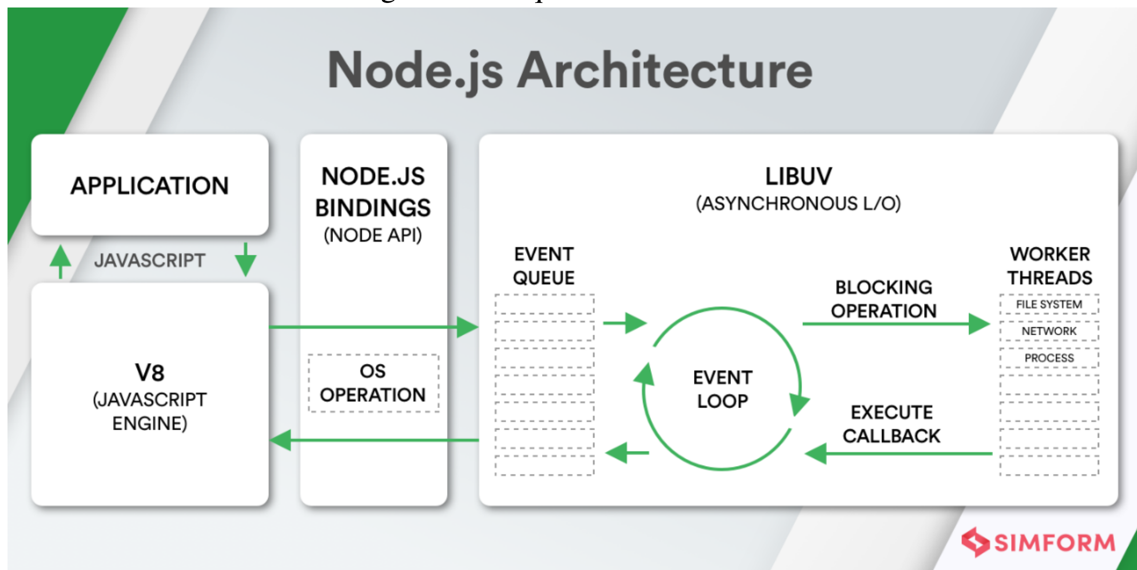
O *NodeJS* é uma plataforma de tempo de execução construída sobre a engine Javascript V8, desenvolvido pela Google em 2008, aliada ao Libuv, que é uma biblioteca escrita em C e multiplataforma, que fornece suporte para operações I/O (Input/Output) assíncronas (CANTELON, 2014). Já em termos práticos, o *NodeJS* é um ambiente de execução Javascript *server-side*, ou seja, direcionado para operações voltadas à servidores web na construção de *backend's*. Por conta de sua estrutura consolidada, há uma grande flexibilidade, alta escalabilidade e baixo custo para programação. Criado em 2009 por Ryan Dahl, ainda possui uma grande relevância, dada a sua forte presença em projetos ao redor do mundo.

De acordo com pesquisa feita em 2020 pelo site StackOverflow (STACKOVERFLOW, 2020) com 40314 usuários, o *NodeJS* é a tecnologia mais utilizada entre as principais linguagens presentes, com 51,4% dos votos, provando ser uma das melhores opções no que diz respeito a interpretar e executar códigos JavaScript em ambientes fora da web. Após a criação do navegador Google Chrome em meados de 2008, a performance do Javascript recebeu diversas melhorias, graças a competição existente entre as outras gigantes do mercado de navegadores (Mozilla, Microsoft, Opera, Apple e Google). As inovações existentes em cada um desses ambientes alteraram a forma como os tipos de aplicação podem ser construídos para exibição na internet (CANTELON, 2014).

Através da utilização do V8, o *NodeJS* consegue alta performance, permitindo compilação direta de código Javascript para uma sequência de instruções em linguagem de máquina. Ao adentrar ainda mais na arquitetura do *NodeJS* (Figura 1), temos que a biblioteca *Libuv* é responsável por executar as operações de I/O como o acesso ao banco de dados, envio de requisições na rede, entre outras. Para isso, há um processo de execução constante chamado laço de eventos (*Event Loop*), que recebe todas as requisições de uma fila de eventos. Cada evento dessa fila entra no *event loop*, onde são distribuídas em *worker threads* que executam as requisições propriamente ditas. Após a execução, uma *callback* é enviada retornando a resposta daquela requisição específica, liberando o *event loop* para receber novas tarefas (CANTELON, 2014). Para a designação de outras características fundamentais do servidor web, o *NodeJS* faz uso de diversas bibliotecas que podem ser instaladas a partir de um gerenciador de pacotes, chamado NPM(*Node Package Manager*), e um pacote de absoluta importância em diversos

projetos é o *ExpressJS*. Este pode ser considerado um framework que funciona sobre o ambiente *NodeJS* em tempo de execução, com utilidade para determinar o roteamento, definição de *middlewares* e manipulação de erros durante o acesso das requisições.

Figura 1 – Arquitetura do NodeJS.



Fonte: (KANERIYA, 2020)

2.2 FRAMEWORK VUEJS: Origem e características

No ano de 2013, um jovem programador chinês chamado Evan You, após trabalhar para o Google, decidiu utilizar os seus conhecimentos para criar seu próprio framework, o *VueJS*. Em fevereiro de 2014, o projeto foi oficialmente lançado, e hoje é mantido por um time de desenvolvedores que auxiliam nas constantes melhorias que a tecnologia vem recebendo ao longo dos anos.

Atualmente na versão 3, o framework frontend *VueJS* possui uma curva rápida de aprendizado, sendo necessária apenas conhecimentos em HTML, CSS e Javascript para já começar a utilizar a ferramenta, bem como o conjunto de bibliotecas e ferramentas disponíveis para uso. Em um arquivo *VueJS*, temos uma estrutura de código com as tags *template*, *script* e *style*. Dentro de template, há o código HTML dos elementos dispostos na tela, com o projeto em questão utilizando alguns recursos da biblioteca BootstrapVue para a construção das páginas (BOOTSTRAPVUE, 2021). Entre as tags *script*, temos algumas funções javascript que são necessárias para lidar com mutações em elementos específicos, além de retornar respostas em métodos que acessam o *backend* por meio das requisições do usuário. Já entre as tags *style*, está

toda a estilização CSS dos componentes da página. Entre as bibliotecas oficiais geridas pelo próprio time do Vue, temos:

- Vue Router: utilizado para configurar a navegação entre páginas;
- Vuex: auxilia no gerenciamento de estados e/ou estados compartilhados entre componentes e páginas;
- Vue Server Renderer: necessário para a renderização de aplicações vue a partir do servidor.

Cada componente do *webapp* ficou organizado em pastas nomeadas de acordo com a sua finalidade, e todos serão abordados em seções próximas.

2.3 POSTGRESQL: Origem e características

O *PostgreSQL* é um SGBD (Sistema de Gerenciamento de Banco de Dados) relacional de código aberto, conhecido pela sua robustez, estabilidade, confiabilidade e integridade no tratamento dos dados, além da dedicação persistente da comunidade em oferecer soluções inovadoras com alto desempenho. O software se originou em 1986 na Universidade de Berkeley, na Califórnia, como parte de uma evolução de outro banco de dados relacional, o Ingres. O então líder do time Ingres, o cientista da computação Michael Stonebraker, liderou o novo projeto chamado POSTGRES, e desde então já são mais de trinta anos de melhorias desenvolvidas na plataforma. Ele utiliza a linguagem SQL (*Structured Query Language*) combinada com diversos recursos que auxiliam no armazenamento e dimensionamento seguro de volume de dados mais complicados (WIKIPEDIA, 2021).

De acordo com (MARIADB, 2018), O *PostgreSQL* também está em conformidade com as principais propriedades exigidas em transações de bancos de dados, que são definidas no acrônimo ACID (Atomicidade, Consistência, Isolamento e Durabilidade). A garantia dada aos usuários por essas propriedades é de que:

- A **atomicidade** indica que a integridade de uma transação está sujeita ao êxito de todas as suas etapas envolvidas, prevenindo que transações falhas ocorram no banco de dados;
- A **consistência** está atrelada à validação dos dados, que é feita de acordo com regras bem estabelecidas. Dessa forma, se, após uma transação, a informação resultante não está de acordo com essas regras, há uma reversão (*Rollback*) na operação para o estado anterior do Banco de Dados;
- O **isolamento** garante a não interferência de transações simultâneas umas nas outras em um banco de dados, para que não ocorram resultados indesejados;

- A **durabilidade** afirma que todas as informações armazenadas em um banco de dados devem ser preservadas, mesmo após quedas na rede elétrica local ou falhas no sistema operacional da máquina operante.

3 METODOLOGIA

A seguir, serão elucidadas as fases de desenvolvimento, iniciando-se pelo projeto do servidor web com os arquivos de configuração e as API's necessárias, além do banco de dados, seguidamente pelo projeto do cliente/*frontend* e toda a comunicação existente entre ambas as aplicações.

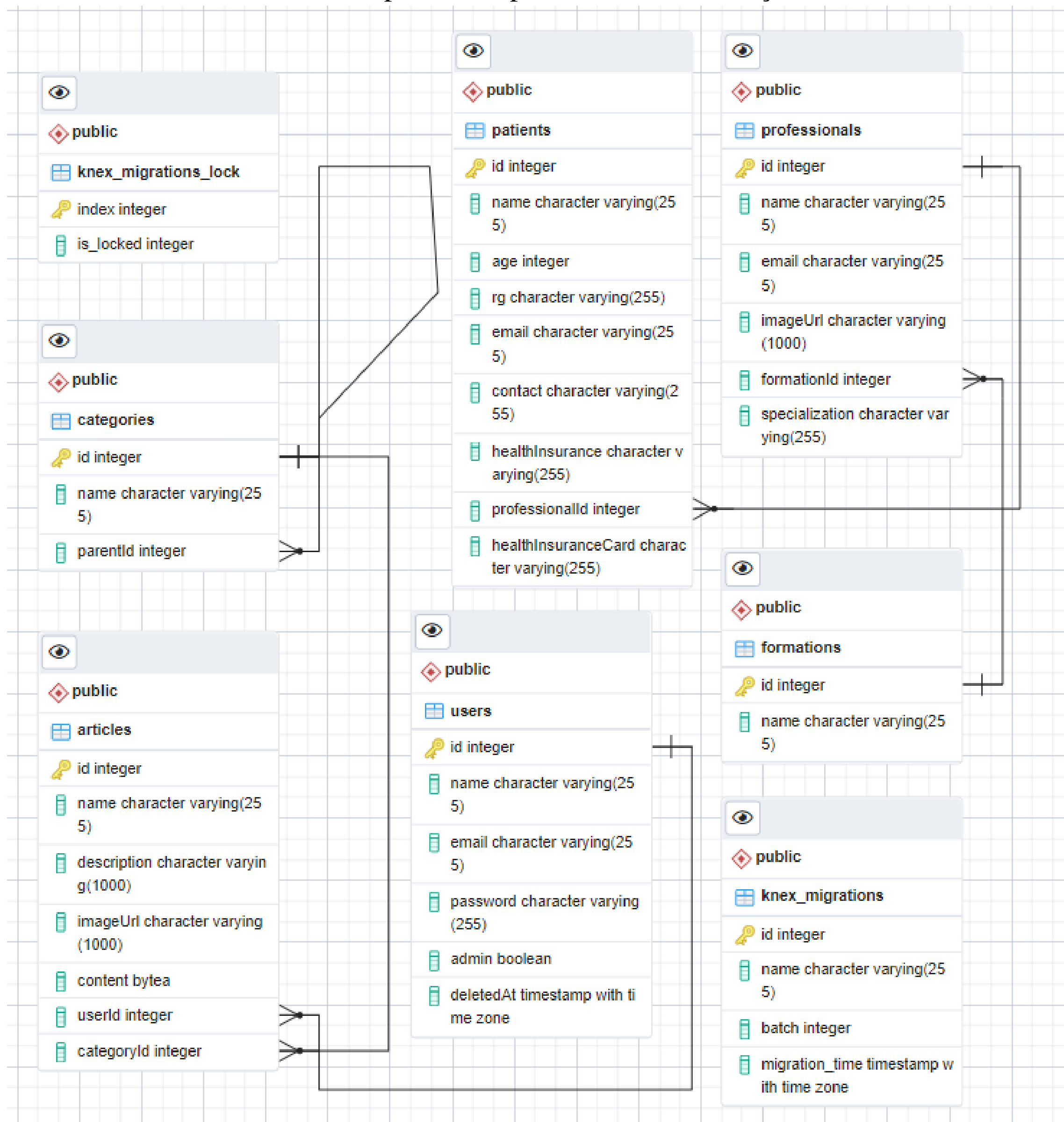
3.1 CRIAÇÃO DAS API'S DO BACKEND

Primeiramente, será necessário criar alguns arquivos de configuração essenciais para o funcionamento do *backend*. O primeiro arquivo é o admin.js, que libera o acesso para alguns serviços da aplicação que requerem privilégios administrativos. O segundo arquivo é o arquivo middlewares.js, que apresenta os *middlewares Cors* e *ExpressJS*, que serão utilizados neste arquivo, respectivamente, para estabelecer conexão entre *backend* e *frontend* e interpretar o corpo das requisições e transformá-los no formato JSON. Em seguida, temos o arquivo *"passport.js"*, que será abordado no tópico sobre a API de autenticação e por último o arquivo *"routes.js"*, com todas as rotas do servidor definidas nesse arquivo.

3.1.1 MODELAGEM DO BANCO DE DADOS

Com as devidas configurações finalizadas, o banco de dados e suas devidas tabelas serão criadas para atender as API's do sistema. Logo após a configuração do *PostgreSQL* e criação das *migrations* definidas no apêndice A.2, a modelagem final do banco de dados pode ser vista a seguir na Figura 2. Como destaque desse modelo, temos que a maioria das relações obecedem a proporção 1:N, como visto entre categorias e artigos, assim como entre profissionais e pacientes. O diagrama entidade-relacionamento foi gerado pelo *pgAdmin*, que é uma plataforma de administração do banco *PostgreSQL*.

Figura 2 – Modelagem do banco de dados, com as tabelas de usuários, artigos, categorias, pacientes, profissionais e formações.



Fonte: Elaborada pelo autor.

3.1.2 *API DE VALIDAÇÃO*

Inicialmente, toda a entrada de dados nos campos das requisições precisa passar por um crivo de validação. Isso é importante para interpretar quais informações estão sendo transmitidas e se elas cumprem determinadas regras que as validam. Para isso, é criado um arquivo chamado *"validation.js"*, que trata das condições de inserção de dados. São 3 funções que validam os campos inseridos:

- *ExistsOrError*: verifica se algum valor foi enviado no corpo da requisição. Caso não haja

entrada de dados, é lançada uma mensagem de erro ao usuário de que os campos não foram preenchidos;

- *NotExistsOrError*: lança uma mensagem de erro caso a verificação identifique a existência de um dado específico;
- *EqualsOrError*: compara duas informações, lançando um erro caso sejam diferentes.

Para todas as API's na aplicação *backend*, o corpo da requisição é verificado por meio dessas funções, garantindo a integridade dos dados. Ademais, os dados enviados pelo cliente via corpo da requisição são tratados pelo middleware *ExpressJS*, que transformará os dados em um JSON. Anteriormente, essa função poderia ser realizada com o middleware *bodyParser*, mas ele acabou sendo reintegrado ao express. Com esses conceitos consolidados, as outras API's serão definidas, desde suas rotas até as funcionalidades que irão exigir de todos os métodos já mencionados. A seguir estão as características de todas as API's do *backend*.

3.1.3 API DE USUÁRIOS

No arquivo *"user.js"*, estão inseridos os métodos que salvam, atualizam, retornam e removem usuários do banco de dados. Para facilitar o acesso ao banco, é utilizada a variável de caminho *app.db*, criada no arquivo *"index.js"* para acesso ao banco com as funções do *KnexJS*. Na função assíncrona save, temos a inserção e atualização de usuários, havendo o uso de confirmação de dados por espera(*await*) para analisar se o usuário já existe no banco no momento de uma inserção, para assim evitar duplicatas. Este conceito é baseado em *promise*, para que o código seja pausado na linha onde se encontra a palavra chave *await*, de forma que a *promise* seja resolvida.

Os usuários cadastrados na aplicação possuem acesso à área administrativa, por meio de um login e senha definidos por um administrador já existente. A diferença entre um usuário administrador e não administrador está no acesso à algumas funções na página administrativa, como por exemplo a área de cadastro de pacientes, profissionais e usuários, que são restritas apenas para usuários administradores. Abaixo (Figura 3), estão as rotas e os métodos para a API de usuários. Todas as rotas criadas estão no arquivo *"routes.js"*, definido na pasta config do projeto *backend*.

Figura 3 – Rotas e métodos HTTP utilizados na API users.

```
app.route('/users')
.all(app.config.passport.authenticate())
.post(admin(app.api.user.save))
.get(app.api.user.get)

app.route('/users/:id')
.all(app.config.passport.authenticate())
.put(admin(app.api.user.save))
.get(admin(app.api.user.getById))
.delete(admin(app.api.user.remove))
```

Fonte: Elaborada pelo autor.

Em algumas rotas, é utilizado um middleware de grande importância chamado Passport, que autoriza algumas API's para que sejam executadas mediante um token de autenticação. Dessa forma, limita-se o escopo de operações e páginas acessadas apenas para usuários logados. No arquivo passport.js, a estratégia definida para autenticação foi pelo método JWT(JSON Web Token), que possui uma estrutura de token compacta e dividida em três partes separadas por pontos (JWT, 2021). A primeira retorna o tipo de token e criptografia do algoritmo, a segunda o formulário *payload* a quem o token se refere (neste caso, o usuário logado) e a terceira parte possui a assinatura de verificação, que será o *authSecret* definido no arquivo *.env*. Após concluído o método de autenticação JWT, deve-se utilizar a função *authenticate* nas rotas que requerem um usuário administrador logado (Figura 3), para garantir a segurança no acesso de determinadas páginas e funcionalidades.

Com as rotas de usuários definidas e a conexão com o banco estabelecida, serão testados os métodos desta API. Para isto, fez-se uso de um software dedicado para testar requisições chamado *Insomnia*. Com ele, pôde-se comprovar o funcionamento da API de acordo com as imagens a seguir. Na Figura 4, temos a inserção pelo método POST de um usuário que irá acessar a aplicação. Em seguida (Figura 5), é utilizado o método GET para retornar todos os usuários já cadastrados. Na Figura 6, temos uma modificação no usuário Marcelo Henrique através do método PUT. E por fim, na Figura 7 acontece a remoção do usuário Marcelo. É válido mencionar que após a conclusão de toda API do projeto *backend*, os testes serão realizados via *Insomnia*, para verificar a existência de qualquer erro na resposta desejada.

Figura 4 – Método POST.

```
POST ▾  http://localhost:3000/users/   Send   204 No Content   211 ms
JSON ▾  Auth ▾  Query  Header 2  Docs      Preview ▾  Header 5
1 {
2   "name": "Marcelo Henrique",
3   "password": 123456,
4   "confirmPassword": 123456,
5   "admin": false,
6   "email": "marceloH@hotmail.com"
7 }
                                           No body returned for response
```

Fonte: Elaborada pelo autor.

Figura 5 – Método GET.

```
GET ▾  http://localhost:3000/users/   Send   200 OK   24.2 ms   307 B
JSON ▾  Auth ▾  Query  Header 2  Docs      Preview ▾  Header 8  Cookie  Timel
1 {                                          1 [
2   "name": "Marcelo Henrique",              2   {
3   "password": 123456,                      3     "id": 5,
4   "confirmPassword": 123456,               4     "name": "João Henrique",
5   "admin": true,                           5     "email": "joaohenrique_as@hotmail.com",
6   "email": "marceloH@hotmail.com"          6     "admin": true
7 }                                          7   },
                                             8   {
                                             9     "id": 6,
                                             10    "name": "novo usuario",
                                             11    "email": "email@hotmail.com",
                                             12    "admin": false
                                             13  },
                                             14  {
                                             15    "id": 7,
                                             16    "name": "Marcelo Henrique",
                                             17    "email": "marceloH@hotmail.com",
                                             18    "admin": false
                                             19  },
```

Fonte: Elaborada pelo autor.

Figura 6 – Método PUT.

```
PUT ▾  http://localhost:3000/users/   Send   200 OK   2.73 ms   248 B
JSON ▾  Auth ▾  Query  Header 2  Docs      Preview ▾  Header 8  Cookie  Timel
1 {                                          1 [
2   "name": "Marcelo Henrique Tavares",      2   {
3   "password": 123456,                      3     "id": 5,
4   "confirmPassword": 123456,               4     "name": "João Henrique",
5   "admin": false,                          5     "email": "joaohenrique_as@hotmail.com",
6   "email": "marcelinho@hotmail.com"        6     "admin": true
7 }                                          7   },
                                             8   {
                                             9     "id": 6,
                                             10    "name": "novo usuario",
                                             11    "email": "email@hotmail.com",
                                             12    "admin": false
                                             13  },
                                             14  {
                                             15    "id": 7,
                                             16    "name": "Marcelo Henrique Tavares",
                                             17    "email": "marcelinho@hotmail.com",
                                             18    "admin": false
                                             19  }
                                             20 ]
```

Fonte: Elaborada pelo autor.

Figura 7 – Método DELETE.

```
DELETE  http://localhost:3000/users/7   Send    200 OK  13.4 ms  158 B
JSON  Auth  Query  Header 2  Docs               Preview  Header 8  Cookie  Timeli

1 {                                              1 [
2   "name": "Marcelo Henrique Tavares",          2   {
3   "password": 123456,                          3     "id": 5,
4   "confirmPassword": 123456,                   4     "name": "João Henrique",
5   "admin": false,                              5     "email": "joaohenrique_as@hotmail.com",
6   "email": "marcelinho@hotmail.com"            6     "admin": true
7 }                                              7   },
                                                 8   {
                                                 9     "id": 6,
                                                10     "name": "novo usuario",
                                                11     "email": "email@hotmail.com",
                                                12     "admin": false
                                                13   }
                                                14 ]
```

Fonte: Elaborada pelo autor.

3.1.4 *API DE AUTENTICAÇÃO*

No arquivo *"auth.js"*, é tratada a função *signin*, que verifica o corpo da requisição de e-mail e senha, enviado pelo cliente, comparando com a tabela de usuários se aquele e-mail ou senha existem no banco de dados, para assim validar a autenticação. As importações realizadas são o *authSecret* definido no arquivo .env, o módulo jwt-simple para codificação do formulário *payload* com os dados do usuário logado e o *bcrypt*, para comparar a senha enviada pelo cliente com a senha criptografada do usuário existente no banco. Mais uma vez, é feito uso de uma função assíncrona, pois haverá a resolução de uma *promise*, que é essa checagem na tabela de usuários para comparar com o corpo da requisição enviada.

O token de um usuário logado tem validade de 3 dias, que é estipulado no *payload* (Figura 8) gerado no momento em que a pessoa entra na aplicação. A validação ocorre na função *validateToken* (Figura 9), que analisa a estrutura JWT gerada pelo login e o *authSecret* da aplicação. É válido ressaltar que a data atual computada no sistema operacional é retornada em milissegundos, logo a constante now, que recebe essa data atual, é dividida por mil para retornar o valor em segundos, facilitando as manipulações de valores para data de expiração e checagem de token válido.

Figura 8 – Dados do payload que são armazenados.

```
const payload = {
    id: user.id,
    name: user.name,
    email: user.email,
    admin: user.admin,
    iat: now,
    exp: now + (60* 60 *24 *3)
}

res.json({
    ...payload,
    token: jwt.encode(payload, authSecret)
})
```

Fonte: Elaborada pelo autor.

Figura 9 – Função validadeToken.

```
const validateToken = async (req, res) => {
    const userData = req.body || null
    try{
        if(userData){
            const token = jwt.decode(userData.token, authSecret)
            if(new Date(token.exp * 1000) > new Date()){
                return res.send(true)
            }
        }
    }catch(e){
    //problema com o token
    }

res.send(false)

}
```

Fonte: Elaborada pelo autor.

3.1.5 API DE CATEGORIAS

Para a API de categorias, temos algumas adições além dos métodos tradicionais já utilizados. A função *save* permite inserir ou atualizar uma categoria existente por meio das funções do *KnexJS*, além de inserir uma subcategoria ao informar qual é a categoria pai. Para deletar uma categoria, é verificado inicialmente se o código *"id"* foi ou não informado. Em seguida, a função *remove* (Figura 10) terá uma série de validações para determinar se é possível ou não deletar uma categoria. Para que não haja problemas, é necessário que a categoria exista

no banco de dados, que ela não tenha subcategorias ou que não existam artigos associados. Atendendo a essas condições, a exclusão da categoria se torna possível.

Figura 10 – Função remove.

```
const remove = async (req, res) => {
    try {
        existsOrError(req.params.id, 'Código da Categoria não informado.')

        const subcategory = await app.db('categories')
            .where({ parentId: req.params.id })
        notExistsOrError(subcategory, 'Categoria possui subcategorias.')

        const articles = await app.db('articles')
            .where({ categoryId: req.params.id })
        notExistsOrError(articles, 'Categoria possui artigos.')

        const rowsDeleted = await app.db('categories')
            .where({ id: req.params.id }).del()
            existsOrError(rowsDeleted, 'Categoria não foi encontrada.')

        res.status(204).send()
    } catch(msg) {
        res.status(400).send(msg)
    }
}
```

Fonte: Elaborada pelo autor.

Uma função interessante presente no arquivo category.js é a *withPath*, que tem por objetivo retornar uma lista de categorias ordenada, apresentando desde a categoria pai até as suas subcategorias existentes. Para isso, é realizada uma verificação por *id* partindo do nó mais profundo, para assim detectar quais categorias estão acima na ordem hierárquica de inserção. Após toda a verificação, os caminhos são retornados em ordem alfabética por meio de uma função *sort*. As próximas funções são a *get*, que retorna as categorias com os seus devidos caminhos e a *getById*, que retorna uma categoria específica por meio de um *id* informado.

Um novo serviço adicionado nessa API tem como propósito formar uma lista de categorias em formato de árvore, que será utilizada no menu da aplicação no projeto do *frontend*. A exibição é feita com o nó mais alto sendo a categoria pai e os nós filhos sendo as subcategorias. Para isto, foram utilizados os métodos *toTree*, que transforma um array de categorias em estrutura de árvore, e *getTree* (Figura 11), que exibe a estrutura da árvore em uma rota específica para este propósito, que é */categories/tree*.

Figura 11 – Funções toTree e getTree.

```
const toTree = (categories, tree) => {
    if(!tree) tree = categories.filter(c => !c.parentId)
    tree = tree.map(parentNode => {
        const isChild = node => node.parentId == parentNode.id
        parentNode.children = toTree(categories, categories.filter(isChild))
        return parentNode
    })
    return tree
}

const getTree = (req, res) => {
    app.db('categories')
        .then(categories => res.json(toTree(categories)))
        .catch(err => res.status(500).send(err))
}
```

Fonte: Elaborada pelo autor.

3.1.6 API DE ARTIGOS

No arquivo *"article.js"*, temos os métodos associados a API de artigos. Inicialmente, o método *save* irá, assim como em qualquer cadastro, testar se o corpo das requisições foi preenchido com os dados corretamente através das funções de validação. Após essa checagem, é feita a inserção ou atualização de um artigo no banco, através do *knex* com o uso da variável *"db"* em *app*. Em seguida, é tratada a remoção de um artigo com a função *remove*, que utiliza um método assíncrono para analisar o *id* do artigo passado como parâmetro na requisição para, assim, deletá-lo. Em seguida, temos a função *get* (Figura 12) que retorna os artigos que foram salvos na página administrativa, que também contará com um paginador que limita a quantidade de artigos exibidos por página. Neste caso, serão cinco artigos por página, delimitados através da função *offset*. Os dados exibidos na página administrativa de artigos serão apenas o id, nome e descrição, haja vista que a exibição do conteúdo de um artigo acontece por meio da função *getById* (Figura 13), que possui retorno único em uma página específica.

Figura 12 – Método *get* na API artigos.

```
const limit = 5 //artigos por página
const get = async(req, res) =>{
    const page = req.query.page || 1
    const result = await app.db('articles').count('id').first()
    const count = parseInt(result.count)

    app.db('articles')
    .select('id', 'name', 'description')
    .limit(limit).offset(page * limit - limit)
    .then(articles => res.json({data: articles, count, limit}))
    .catch(err => res.status(500).send(err))

}
```

Fonte: Elaborada pelo autor.

Figura 13 – Método *getById* na API de artigos.

```
const getById = (req, res) => {
    app.db('articles')
    .where({id: req.params.id})
    .first()
    .then(article => {
        article.content = article.content.toString()
        return res.json(article)
    })
    .catch(err => res.status(500).send(err))
}
```

Fonte: Elaborada pelo autor.

Já na função *getByCategory* (Figura 14), temos o retorno de uma categoria pai com todas as subcategorias filhas, que serão os artigos em si. Essa função apresenta algumas particularidades, como o uso de consultas adicionais no banco, como a que é feita no arquivo *"queries.js"* para retornar recursivamente todas as categorias e suas respectivas subcategorias encadeadas, além de utilizar mnemônicos para renomear as tabelas de artigos e usuários, reduzindo a carga de código escrito. A tabela de usuários é utilizada nesse contexto, afinal o autor do artigo precisa estar na tabela de usuários.

Figura 14 – Função *getByCategory* na API de artigos.

```
const getByCategory = async (req, res) => {
    const categoryId = req.params.id
    const page = req.query.page || 1
    const categories = await
        app.db.raw(queries.categoryWithChildren, categoryId)
    const ids = categories.rows.map(c => c.id)

    app.db({a: 'articles', u: 'users'})
    .select(
        'a.id', 'a.name',
        'a.description',
        'a.imageUrl',
        {author: 'u.name'})
    .limit(limit).offset(page*limit - limit)
    .whereRaw('?? = ??', ['u.id', 'a.userId'])
    .whereIn('categoryId', ids)
    .orderBy('a.id', 'desc')
    .then(articles => res.json(articles))
    .catch(err => res.status(500).send(err))
}
```

Fonte: Elaborada pelo autor.

3.1.7 API DE FORMAÇÃO

Essa API é bem simples e o arquivo *"formation.js"* possui os métodos *save*, *remove*, *get* e *getById*. O intuito é cadastrar as diferentes formações e gerar uma lista que será utilizada no cadastro de profissionais na área administrativa. Com o advento de novos profissionais de diferentes áreas, novas formações serão adicionadas por meio dessa API.

3.1.8 API DE PROFISSIONAIS

No arquivo *"professional.js"* são definidos os métodos para cadastro, atualiação, retorno e remoção de profissionais da Clínica São Gonçalo. Alguns métodos são semelhantes aos já apresentados, como o *save*, *get* e *getById*. A novidade existente é o método *getByFormation*, que retorna os profissionais por formação, que será utilizado na página de exibição de profissionais para separar cada indivíduo por área de atuação, sendo cada área representada por um *"id"* específico.

3.1.9 API DE PACIENTES

Para finalizar, a última API criada é a de pacientes, que no momento o arquivo *patient.js* permite todas as operações básicas como registrar, atualizar, retornar e remover um paciente que utiliza os serviços da Clínica São Gonçalo. Alguns dados importantes como RG e convênio de saúde estão presentes e são importantes para identificar cada cliente.

3.2 COMPONENTES DO WEPAPP

Todas as páginas e componentes *VueJS* da aplicação foram construídos sob a orientação de um modelo de layout chamado *Flexible Box Module*, ou Flexbox (COYIER, 2021). Introduzido no CSS3, esse recurso permite uma melhor organização dos elementos em estruturas chamadas *containers*, garantindo uma maior flexibilidade e responsividade dos elementos HTML de uma página de maneira dinâmica. A seguir, serão introduzidos todos os componentes do webapp e suas funcionalidades.

3.2.1 Cabeçalho

No topo da aplicação, o cabeçalho é um componente fixo, definido no arquivo *"header.vue"* e dividido em duas partes. A primeira trata-se do *top-header*, que possui três elementos principais em sua composição, que são o logo da clínica com um *hyperlink* para a página inicial, uma área para acesso às redes sociais (*Whatsapp*, *Facebook* e *Instagram*) e por último o menu de usuário cadastrado, que é ocultado por padrão caso ninguém faça login. A segunda parte é o *bottom-header*, que também conterá três elementos em sua estrutura. O primeiro é um botão para acesso ao menu lateral, que conterá todo o acervo de categorias e seus respectivos artigos escritos no site. Este botão fará uso de alguns recursos do *Vuex*, biblioteca para gerenciamento de estados de componentes, pois a ativação deste botão altera a visualização do menu lateral.

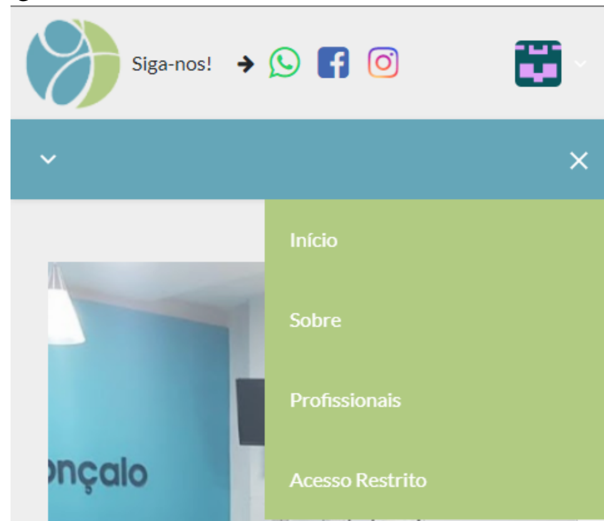
Em seguida, temos o menu central, inspirado em um modelo disponibilizado no site CodePen (MUTEDBLUES, 2021). Cada botão do menu possui um *hyperlink* que redireciona para a página inicial, área de informações sobre a clínica, página de profissionais e acesso à tela de login da área administrativa. E por fim, temos o menu *dropdown* para usuários logados na aplicação, que é um componente à parte localizado no arquivo userDropdown.vue, na pasta *templates*. Por padrão, esse componente é oculto para usuários comuns, estando visível apenas para usuários cadastrados. A visualização completa do cabeçalho será apresentada a seguir (Figura 15).

Figura 15 – Visualização do cabeçalho com usuário logado.

Fonte: Elaborada pelo autor.

O cabeçalho é responsivo, com o menu central sendo ocultado até a largura máxima de 911 pixels. Quando ocultado, um botão aparece do lado direito do cabeçalho azul, permitindo acessar o menu central (Figura 16) por meio de uma lista vertical. Já o botão do menu *dropdown* oculta o nome do usuário até a largura de 575 pixels.

Figura 16 – Menu central exibido em lista vertical.

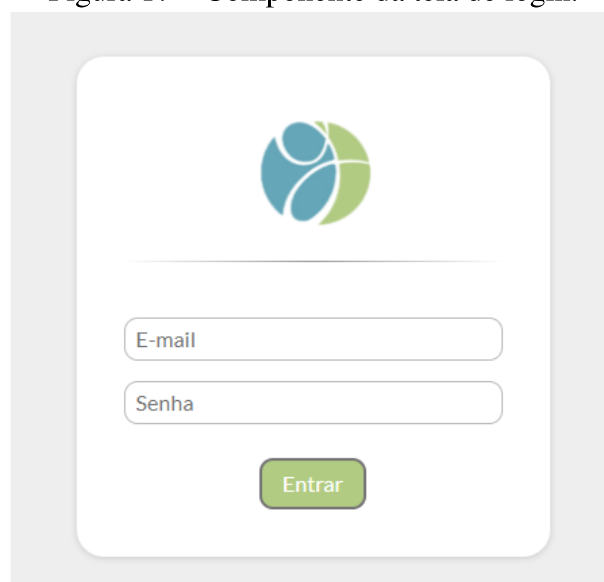


Fonte: Elaborada pelo autor.

3.2.2 *Tela de login/Acesso restrito*

A tela de login possui apenas um componente simples (Figura 17), com uma estrutura dividida em dois campos para inserção de dados, um para o e-mail e o outro para a senha do usuário. Além disso, há também a logo da clínica e um botão para confirmação/envio dos dados. No arquivo onde é gerado o template existem apenas dois métodos, que são o *sign in* e *sign up*. Ambos os métodos fazem uso da biblioteca *axios*, para realizar as requisições do tipo POST em usuários.

Figura 17 – Componente da tela de login.

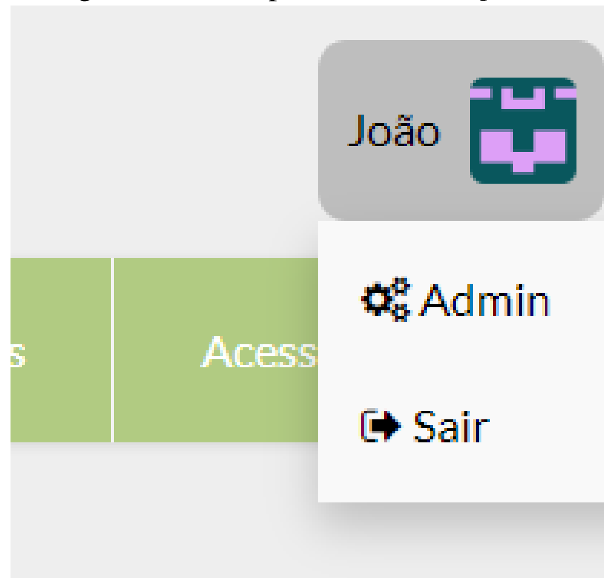


Fonte: Elaborada pelo autor.

3.2.3 *Menu dropdown de usuário*

Esse componente (Figura 18) foi criado à parte da estrutura do cabeçalho. Haja vista que não seria um componente visível ao usuário comum, foi dada uma atenção diferenciada, sendo então necessário cria-lo como uma extensão extra para cabeçalho.

Figura 18 – Componente *userDropdown*.



Fonte: Elaborada pelo autor.

A estrutura consiste em um botão no estilo *hover* com o nome e um avatar do usuário logado, exibindo assim o menu *dropdown* da aplicação, que possui dois botões. O primeiro é o de acesso a página administrativa, onde são permitidas diversas ações associadas a inserção, remoção e atualização de dados do banco, que serão abordadas na seção apropriada. E por último, temos um botão para sair da aplicação, removendo assim o token de autenticação de usuário e ocultando a visibilidade do componente *userDropdown*.

3.2.4 *Introdução sobre a Clínica*

Essa página possui algumas informações sobre a clínica São Gonçalo como a instituição, os seus valores e missão. É aqui onde está validado o compromisso e responsabilidade da clínica para com o cidadão que irá fazer uso dos serviços prestados. A visualização da página pode ser vista a seguir (Figura 19).

Figura 19 – Página introdutória da clínica.

Fonte: Elaborada pelo autor.

3.2.5 Conteúdo

Em todas as páginas, o componente *content* estará presente para exibir informações relevantes na área central da aplicação. O arquivo *"content.vue"*, que é originalmente vazio, terá apenas a estilização do espaço com as seguintes características destacadas na Figura 20.

Figura 20 – Estilização CSS da área de conteúdo.

```
<style>
    .content{
        grid-area: content;
        grid-column: span 3 / -1;
        background-color: rgb(238, 238, 238);
        padding: 50px 250px 100px 250px;
        box-sizing: border-box;
    }

    @media (max-width: 1366px) {
        .content{
            padding: 50px 30px 100px 30px;
        }
    }
</style>
```

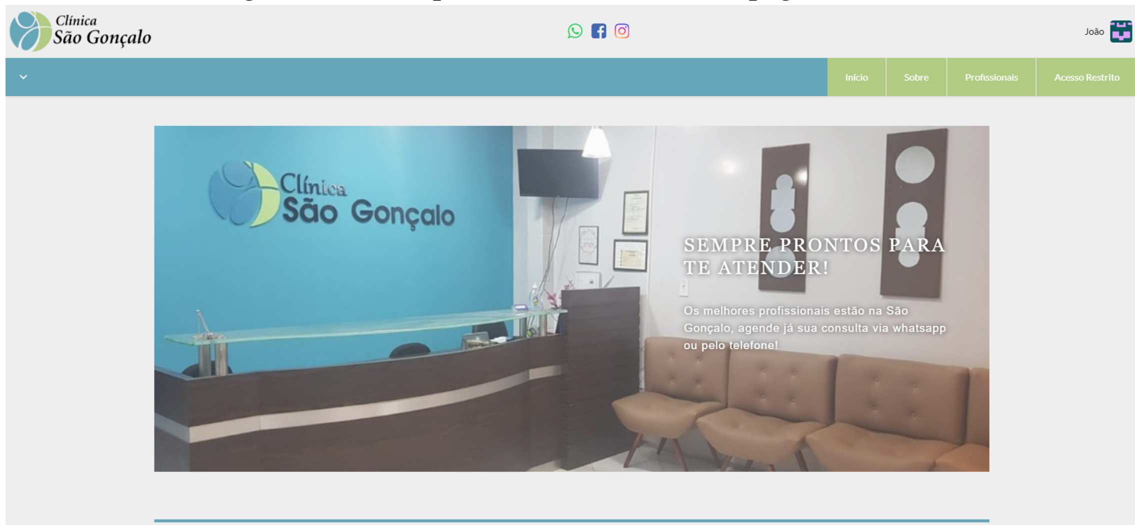
Fonte: Elaborada pelo autor.

A informação de maior relevância, além da cor em escala de cinza, seria o *padding* da área do conteúdo, com os quatro valores referentes às margens superior, direita, inferior e esquerda. Há ainda uma regra para aplicar um *padding* diferente quando a tela tiver até 1366 pixels de largura.

3.2.6 Tela de Início

Para ocupar o espaço na tela inicial criada, temos um arquivo chamado *"homePage.vue"*, que terá duas áreas ocupadas por componentes de imagens do tipo *carousel*, cada um com sua individualidade. O primeiro componente é o *slider* (Figura 21), em que imagens apresentam a clínica e sua estrutura. Para cria-lo, foi necessário apenas alguns conhecimentos de CSS para definir a transição, duração e efeitos de opacidade das imagens, posição e tamanho dos textos, além da responsividade dos textos e do próprio componente em si, que terá a largura máxima permitida dentro de *"content.vue"*.

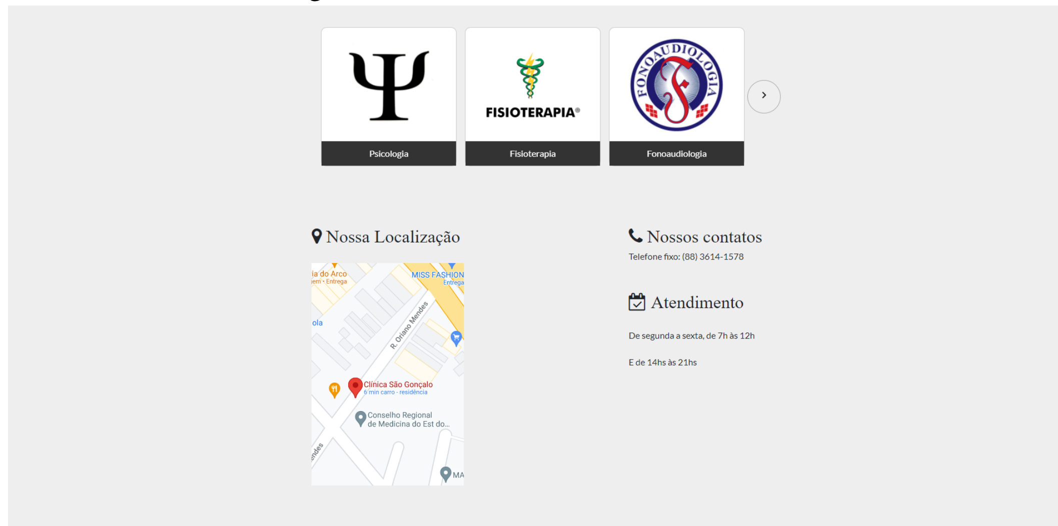
Figura 21 – Componente *slider* inicial da página inicial.



Fonte: Elaborada pelo autor.

Logo abaixo do componente slider, há um novo componente (Figura 22) do tipo *carousel* que apresenta os variados atendimentos oferecidos. E para finalizar, algumas informações acerca da localização, horários e telefones para contato estão disponíveis nesse mesmo espaço.

Figura 22 – Parte inferior da tela inicial.

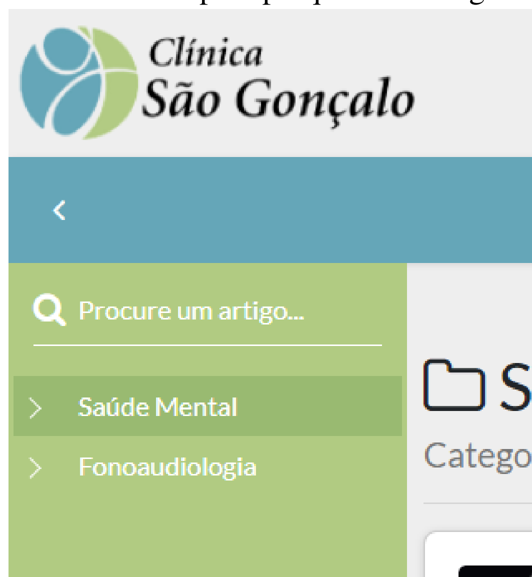


Fonte: Elaborada pelo autor.

3.2.7 *Menu Lateral*

Esse componente (Figura 23) atua em conjunto com o botão localizado no canto superior esquerdo do cabeçalho, que permite acesso ao menu lateral. A visualização desse menu é dada através da observação de estado do botão com o auxílio do *Vuex*, alternando entre visível e não visível. Os componentes *footer* e *content* utilizam técnicas do módulo CSS *grid*, utilizando valores de posicionamento para permitir a sobreposição do menu sem alterar os demais componentes na tela (GEDDES, 2021).

Figura 23 – Menu lateral para pesquisa de categorias e artigos.



Fonte: Elaborada pelo autor.

A estrutura do menu é construída através do uso da biblioteca *liquor-tree*, onde as categorias e seus respectivos artigos são coletados do *backend* e retornados pela biblioteca em formato de árvore. Com o uso da tag *transition* fornecida pelo *VueJS*, foi possível adicionar uma transição no estilo *slide-fade* para a exibição do componente. No arquivo *"menu.vue"*, onde está definida toda a estrutura do menu, existem dois métodos aplicados. O primeiro é o *getTreeData*, onde é retornada a árvore de categorias e artigos, e o segundo método é o *onNodeSelect*, que realiza um *push*(redirecionamento) para o artigo de cada nó da árvore.

3.2.8 Área Administrativa

A página administrativa da aplicação possui diversas abas, cada qual com suas devidas atribuições (Figura 24). O princípio por trás dessa página está associado a toda a manipulação envolvendo as API's e o banco de dados, onde os métodos que requerem privilégios administrativos podem ser executados. Todas as abas possuem a mesma estrutura visual, diferenciando apenas no tipo de dados enviados via formulário.

Figura 24 – Área administrativa.

Fonte: Elaborada pelo autor.

A primeira aba trata do cadastro de artigos, com a adição de todos os campos que foram previamente definidos no *backend* como nome, descrição, categoria e conteúdo. A segunda aba é a de categorias, onde também estão os campos para cadastro das categorias de artigos futuros. A terceira aba é voltada para as formações existentes entre os profissionais da clínica. A quarta aba trata dos usuários do banco de dados, onde podem ser criados novos usuários com privilégios administrativos para exercer as tarefas presentes em toda a área administrativa. A

quinta aba permite cadastrar os profissionais que trabalham na clínica e a sexta e última aba apresenta a tela para cadastro de pacientes/clientes.

3.2.9 Exibição de Profissionais

Nesta página (Figura 25), serão exibidos os profissionais que realizam atendimento na clínica São Gonçalo. A visualização do componente acontece por meio de *tabs*, assim como na área administrativa. Cada uma fornece os profissionais daquela área de atuação específica, mostrando foto, nome, e-mail e especialização.

Figura 25 – Exibição de profissionais.

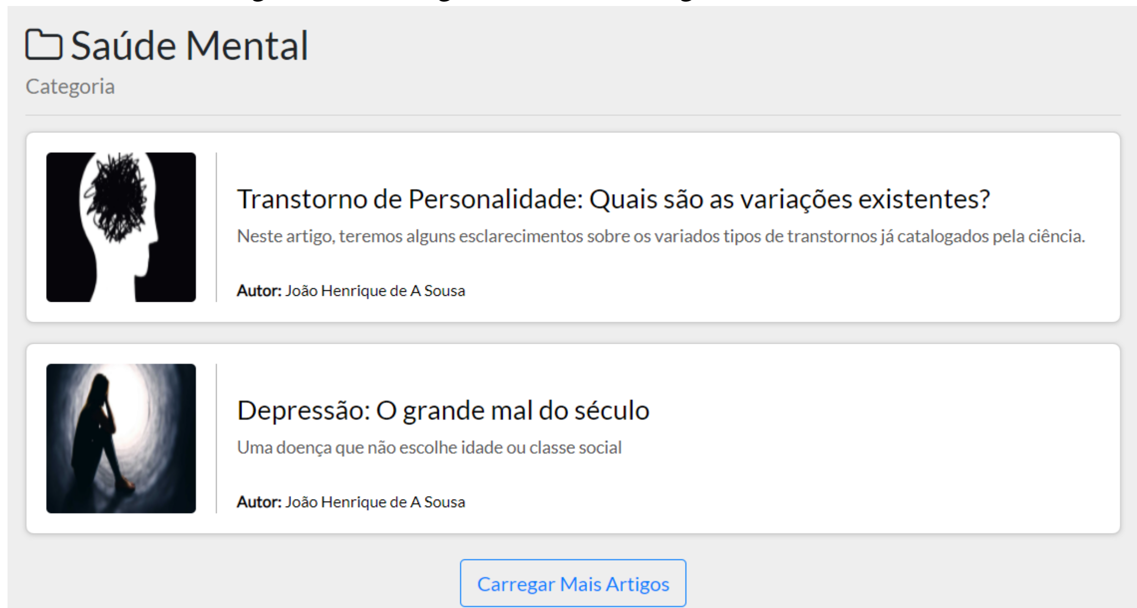
Fonte: Elaborada pelo autor.

3.2.10 Exibição de Categorias e Artigos

O componente em questão trata-se de três arquivos presentes no projeto *frontend*. São eles o *articleById.vue*, *articleItem.vue* e o *articlesByCategory.vue*. Todos esses arquivos comunicam entre si, de maneira que a disposição de seus elementos acontece da forma que será demonstrada adiante. Inicialmente, é necessário criar uma tela para mostrar determinada categoria selecionada pelo usuário no menu lateral. Ao clicar em uma categoria, é exibido o componente *articlesByCategory*, que possui uma tela padronizada com o componente *pageTitle*, exibindo o título da categoria selecionada e uma lista de todos os artigos e suas informações,

que são definidas no arquivo *articleItem*. Ademais, há um botão para exibir mais artigos como mostra a Figura 26.

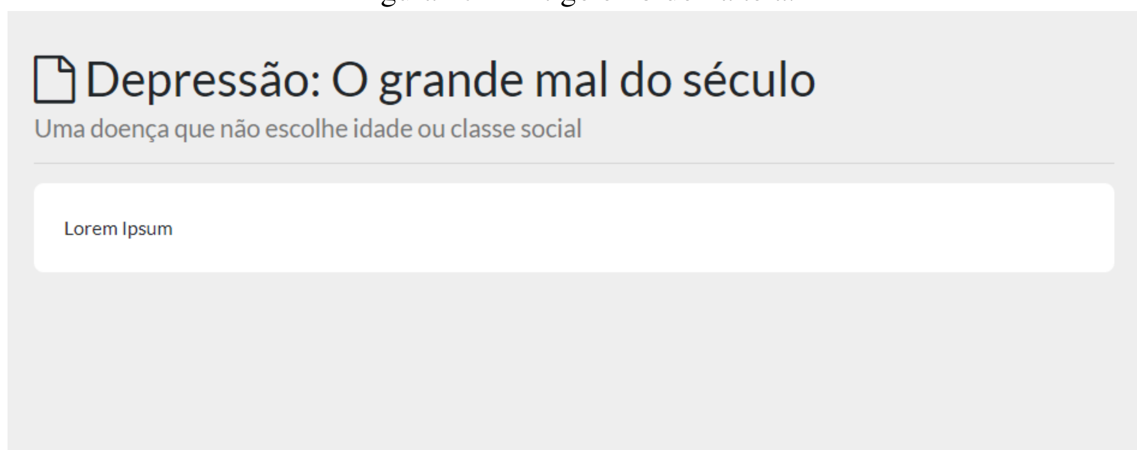
Figura 26 – Categoria com seus artigos exibida na tela.



Fonte: Elaborada pelo autor.

Após clicar em um dos artigos mostrados, é então exibido o seu conteúdo (Figura 27), que possui título, descrição e conteúdo especificados no arquivo *articleById*. A construção de um artigo está presente na aba administrativa, onde é permitido submeter um artigo e editá-lo posteriormente, além da possível remoção.

Figura 27 – Artigo exibido na tela.

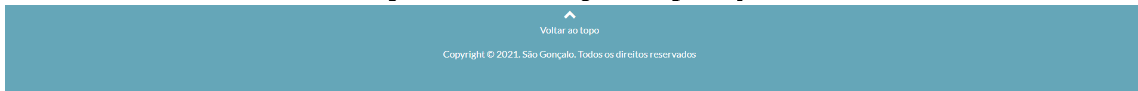


Fonte: Elaborada pelo autor.

3.2.11 Rodapé

A área inferior da página (Figura 28) possui um espaço que utiliza o mesmo tom de verde presente na logo da clínica, para assim preservar a identidade visual. O rodapé apresenta apenas uma menção aos direitos reservados da página, sejam associados a marca da clínica ou ao uso de imagens de terceiros. Há também o uso de um botão, que ao ser acionado retorna para o topo da aplicação.

Figura 28 – Rodapé da aplicação.



Fonte: Elaborada pelo autor.

4 TESTE DE ACEITAÇÃO E AVALIAÇÃO DO SISTEMA

Após a conclusão do sistema, foram analisadas as principais funções da aplicação para uso na clínica. Com alguns testes iniciais controlados, alguns pontos merecem destaque, bem como outros recursos que ainda precisam de uma futura implementação no código fonte. A área de conteúdos para artigos mostrou-se bastante útil para divulgação de informativos e conteúdos relevantes associados ao atendimentos dos profissionais. A visualização destes profissionais e suas áreas de especialidade também se mostrou um aspecto positivo para divulgação ao público. Já em relação ao cadastro de pacientes, foi sugerido anexar as fichas de acompanhamento (Figura 29) para cada cadastro, trocando assim o formulário físico pelo formato digital no próprio site, assim reduzindo o uso de papel. Outra questão que surge para futuras atualizações é a inserção de uma nova área administrativa voltada para gestão financeira e cálculo de produção dos profissionais. Assim, haverá uma facilitação no cálculo de valores pagos pela clínica com os convênios de saúde associados (Unimed, São Camilo, Hapvida, etc). Além dessas questões, algumas melhorias visuais também são bem vindas, facilitando assim a visualização das páginas existentes.

Figura 29 – Exemplo de ficha de acompanhamento de paciente da Clinica Espaço Recriar.

Fonte: Elaborado pelo autor.

5 CONCLUSÕES E TRABALHOS FUTUROS

O presente trabalho mostrou que o desenvolvimento web apresenta uma infinidade de recursos para o programador interessado em aprender novas ferramentas e tecnologias. As aplicações com *VueJS*, *PostgreSQL* e *NodeJS* são exemplos comuns nesse mercado, assim como outras tecnologias que proporcionam um vasto leque de possibilidades. Todo esse material é de fácil acesso na própria internet, sejam nos sites oficiais com suas documentações ou em livros/artigos que abordam o uso de *frameworks* e os procedimentos envolvidos.

Após o desenvolvimento inicial e testes da aplicação para a clínica São Gonçalo, ainda há um caminho a ser percorrido para atingir o resultado ideal. Até a realização deste trabalho, o sistema está bem estruturado, com um banco de dados construído no *PostgreSQL*, auxiliado pelo *KnexJS* para gerir usuários, artigos, profissionais e pacientes. Quanto à aparência, utilizando o *VueJS*, houve um cuidado para preservar as cores e identidade visual da clínica. Por fim, um *backend* sólido plenamente funcional que opera em *NodeJS* para tratar as ações do usuário na aplicação. Para futuras atualizações deste projeto está a implementação de novos recursos visuais, inserção de uma ficha digital para acompanhamento dos pacientes e uma área administrativa para gestão financeira. Por fim, a aplicação completa será registrada em um domínio na internet, com todas as operações entre *backend* e *frontend* executadas em alguma plataforma de nuvem como serviço como *Amazon Web Services, Heroku* ou *Firebase*.

REFERÊNCIAS

BOOTSTRAPVUE. **BootstrapVue**. 2021. Acessado em: 25/08/2021. Disponível em: <https://bootstrap-vue.org/docs/components>.

CANTELON, M. **Node.Js in action**. [S.l.]: Manning Publications, 2014.

COYIER, C. **A Complete Guide to Flexbox**. 2021. Acessado em: 09/08/2021. Disponível em: <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>.

GEDDES, D. **Overlapping Grid Items**. 2021. Acessado em: 26/08/2021. Disponível em: <https://mastery.games/post/overlapping-grid-items/>.

JWT. **Introduction to JSON Web Tokens**. 2021. Acessado em: 23/07/2021. Disponível em: <https://jwt.io/introduction>.

KANERIYA, T. **What is Node.js? Where, When How To Use It (With Examples)**. 2020. Acessado em: 10/07/2021. Disponível em: <https://www.simform.com/blog/what-is-node-js/>.

MARIADB. **What is ACID Compliance? What It Means and Why You Should Care**. 2018. Acessado em: 24/06/2021. Disponível em: <https://mariadb.com/resources/blog/acid-compliance-what-it-means-and-why-you-should-care/>.

MUTEDBLUES. **Codepen**. 2021. Acessado em: 29/08/2021. Disponível em: <https://codepen.io/mutedblues/pen/MmPNPG>.

PERK. **Database Migrations with Knex**. 2021. Acessado em: 20/07/2021. Disponível em: <http://perkframework.com/v1/guides/database-migrations-knex.html>.

SOUSA, I. de. **Entenda o que é Rest API e a importância dele para o site da sua empresa.** 2020. Acessado em: 28/06/2021. Disponível em: <https://rockcontent.com/br/blog/rest-api/>.

STACKOVERFLOW. **Developer Survey**. 2020. Acessado em: 14/08/2021. Disponível em: <https://insights.stackoverflow.com/survey/2020#technology-other-frameworks-libraries-and-tools-all-respondents3>.

VUECLI. **Overview**. 2019. Acessado em: 05/08/2021. Disponível em: <https://cli.vuejs.org/guide/>.

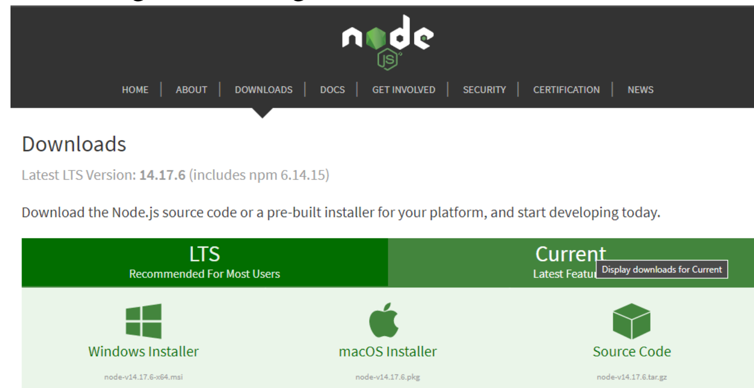
VUEJS. **VueJS Guide**. 2021. Acessado em: 24/08/21. Disponível em: <https://vuejs.org/v2/guide/>.

WIKIPEDIA. **PostgreSQL**. 2021. Acessado em: 20/07/2021. Disponível em: <https://en.wikipedia.org/wiki/PostgreSQL>.

APÊNDICE A – ESTRUTURAÇÃO DO PROJETO *BACKEND*

Para a aplicação web desenvolvida, a primeira etapa realizada foi a configuração e teste do ambiente *NodeJS* para execução geral do *backend*. Para isso, é necessária a correta instalação do *NodeJS* na máquina utilizada. O procedimento foi feito conforme a maioria dos programas instalados no sistema Windows, utilizando um arquivo de instalação no formato .msi (Windows installer), obtido do site oficial do *NodeJS* (Figura 30).

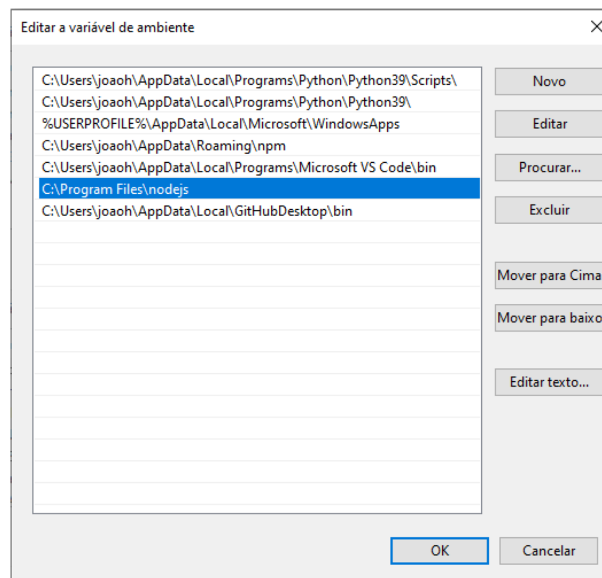
Figura 30 – Página de download do NodeJS.



Fonte: NodeJs.org.

Após instalado, é necessário adicionar o caminho de origem do *NodeJS* para as variáveis de ambiente do sistema operacional, para que os comandos do *NodeJS* via prompt de comando possam ser reconhecidos com sucesso (Figura 31).

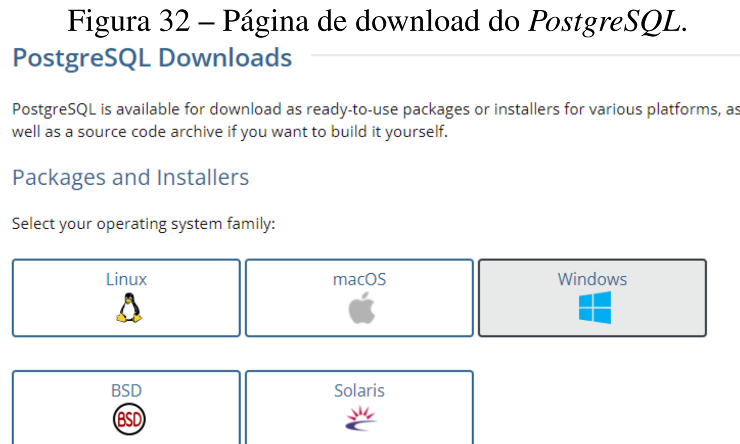
Figura 31 – NodeJS adicionado às variáveis de ambiente do Windows.



Fonte: Elaborado pelo autor.

Terminada a instalação e configuração básica do ambiente, é necessário instalar o banco de dados relacional utilizado, que nesta aplicação será o *PostgreSQL*. No Windows, a instalação acontece da mesma forma que o *NodeJS*, sendo necessário baixar o arquivo de instalação do site oficial (Figura 32) e, durante a instalação, criar um usuário e senha para acesso ao banco e suas funcionalidades. O diretório de instalação também deve estar adicionado na variável de sistema *Path*.

Figura 32 – Página de download do *PostgreSQL*.



Fonte: postgresql.org.

Para formar a base do *backend*, primeiramente é criada uma pasta vazia com um diretório para o *backend* (outra pasta também será criada para o *frontend*, de uso posterior). Após a criação das pastas, utiliza-se o terminal embutido no VsCode para acessar a pasta do *backend* e, por meio do comando *"npm init"*, iniciar o *NodeJS*. Com isto, alguns dados gerais sobre a aplicação serão criados e validados (Figura 33), como o arquivo *"index.js"*.

Figura 33 – Criação do arquivo package.json com as informações sobre a aplicação.

```
About to write to C:\Users\joaoh\Documents\testeTCC\backend\packa
ge.json:

{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
```

Fonte: Elaborado pelo autor.

Este será o ponto de entrada (*entry point*) para a execução do *backend*, além do arquivo *"package.json"*, que contará com todas as dependências e bibliotecas necessárias para o funcionamento do servidor. Cada dependência e sua finalidade será apresentada adiante.

A.1 DEPENDÊNCIAS UTILIZADAS NO BACKEND

A seguir, serão apresentados todos os pacotes, middlewares e frameworks presentes no arquivo *"package.json"* e suas finalidades no projeto do *backend*.

- **Bcrypt-nodejs**: Utilizado para criptografia de dados sensíveis, como senhas;
- **Consign**: organiza e carrega as API's e outros arquivos como rotas, scripts e configurações por meio de um parâmetro centralizado;
- **Cors**: *middleware* que permite que o *frontend* tenha acesso ao *backend*;
- **Express**: gerencia múltiplas requisições http em uma url específica, além de estruturar os web-services criados por meio de *Middlewares*. Também é utilizado para retornar o campo com os dados enviados pelo cliente via corpo da requisição, em formato JSON(JavaScript Object Notation);
- **Jwt-simple**: módulo que codifica e decodifica por meio de uma palavra-chave de segurança, garantindo a preservação dos dados;
- **Knex**: *querybuilder* necessário para criar as tabelas no banco de dados relacional *PostgreSQL* através das *migrations*;
- **Moment**: biblioteca associada a análise, validação, manipulação e formatação de datas;
- **Passport**: *middleware* de autenticação de requisições, para garantir acesso restrito em determinadas funcionalidades da aplicação;
- **Passport-jwt**: módulo de autenticação que utiliza como estratégia o uso de *web tokens* JSON para garantir o acesso em determinadas requisições;
- **Pg**: módulo para conexão com o banco de dados *PostgreSQL*;
- **Pm2**: gerenciador de processos automatizado e avançado para aplicações *NodeJS* em ambientes de produção;

Todas as dependências listadas para iniciar o servidor devem ser instaladas utilizando o gerenciador de pacotes do *NodeJS* chamado NPM (*Node Package Manager*). Como o arquivo *package.json* já possui a listagem com todos os pacotes necessários, basta acessar o diretório do *backend* no terminal e utilizar o comando *npm install*, que tudo será instalado de uma só vez. Para instalações individuais, o mesmo comando é utilizado seguido pelo nome do pacote, como

mostra a Figura 34.

Figura 34 – Exemplo de instalação do pacote *ExpressJS*.

```
PS C:\Users\joaoh\Documents\testeTCC\backend> npm install express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN backend@1.0.0 No description
npm WARN backend@1.0.0 No repository field.

+ express@4.17.1
added 50 packages from 37 contributors and audited 50 packages in 1.905s
found 0 vulnerabilities

PS C:\Users\joaoh\Documents\testeTCC\backend> 
```

Após a instalação de todos, serão importados no arquivo *index.js* alguns pacotes necessários, como o *ExpressJS* e o *Consign* (assim como o arquivo *db.js*, que será criado posteriormente e que executa o banco de dados). Em seguida, utiliza-se o *ExpressJS* para a configuração da porta do servidor que irá atender as requisições de usuário, que neste caso será de valor 3000. Enquanto o servidor estiver ativo, o uso dessa porta é restrito para a aplicação *backend*. Após isto, pode-se testar o servidor com o comando *npm start* no terminal do diretório da aplicação *backend*, que executará o arquivo *index.js* e todos os arquivos e dependências estabelecidas.

Figura 35 – Configuração básica do servidor no arquivo index.js.

```
backend > JS index.js > ...
1  const app = require('express')()
2
3  app.listen(3000, () =>{
4      console.log('Backend em execução...')
5  })
```

A.2 CONFIGURANDO O POSTGRESQL E KNEXJS

Antes de dar início ao desenvolvimento das API's do *backend*, é necessário criar um banco de dados vazio para a aplicação. Com o usuário e senha para o *PostgreSQL* definidos, deve-se seguir as etapas a seguir:

- Autenticar o usuário e senha do *PostgreSQL* através do cmd, por meio do comando 'psql -U nomedousuario', sem as aspas;
- Utilizar o comando 'CREATE DATABASE nomedobanco', para que o banco vazio seja criado.

Agora que o banco já existe, é necessário fazer a conexão da aplicação com o banco

e criar as tabelas que serão utilizadas, que são as de usuários, artigos, categorias de artigos, profissionais e pacientes. Para isto, foi feito uso de um *querybuilder* bastante útil chamado *KnexJS*, que realiza essa tarefa através do uso de *migrations*, possibilitando atualizações e mudanças nas tabelas de maneira dinâmica e simplificada. A inicialização do *KnexJS* acontece através do comando *knex init* no terminal da pasta da aplicação *backend*, acarretando na criação do arquivo *knexfile.js*, que trará informações como o SGBD utilizado, limites para *pooling* de conexão, o nome do banco e a tabela com a lista de migrations existentes.

Esses dados são importados para o arquivo *"db.js"*, que executará as *migrations* criadas e terá o *KnexJS* instanciado com a configuração prevista no arquivo *knexfile.js*. Para não comprometer a segurança das informações, os dados sensíveis como usuário e senha do banco são colocados em um outro arquivo chamado *".env"* (Figura 36), que possui este formato para que não seja enviado ao repositório no Github ao final de um *push*, além de ser protegido por um código de autenticação (*authSecret*).

Figura 36 – Arquivo .env e suas informações.

```
backend > .env
1   module.exports = {
2       authSecret: 'octopathtraveler',
3       db: {
4           host: '127.0.0.1',
5           port: 5432,
6           database: 'clinicasg',
7           user: 'postgres',
8           password: '1234'
9       }
10  }
```

Após todas essas medidas adotadas, as tabelas são criadas por meio da linha de comando ‘knex:migrate make nomeDaTabela’, gerando um arquivo *javascript*. De acordo com as figuras abaixo (Figura 37, Figura 38 e Figura 39), a estrutura do código de uma *migration* consiste em duas funções: *up* e *down*. A função *up* cria o esquema da tabela, suas colunas e os tipos de atributos e a função *down* retorna o oposto, sendo geralmente uma função para desfazer a tarefa determinada na função *up*, caso seja necessário (PERK, 2021).

Figura 37 – Criação da tabela users por meio de uma migration.

```
backend > migrations > JS 20210319170516_create_table_users.js > ...
1
2   exports.up = function (knex, Promise) {
3       return knex.schema.createTable('users', table => {
4           table.increments('id').primary()
5           table.string('name').notNull()
6           table.string('email').notNull().unique()
7           table.string('password').notNull()
8           table.boolean('admin').notNull().defaultTo(false)
9       })
10  };
11
12  exports.down = function (knex, Promise) {
13      return knex.schema.dropTable('users')
14  };
15
```

Figura 38 – Migration para a tabela de artigos.

```
backend > migrations > JS 20210319170645_create_table_articles.js > up > up
1
2   exports.up = function(knex, Promise) {
3       return knex.schema.createTable('articles', table =>{
4           table.increments('id').primary()
5           table.string('name').notNull()
6           table.string('description', 1000).notNull()
7           table.string('imageUrl', 1000)
8           table.binary('content').notNull()
9           table.integer('userId').references('id')
10              .inTable('users').notNull()
11          table.integer('categoryId').references('id')
12              .inTable('categories').notNull()
13      })
14
15  };
16
17  exports.down = function(knex, Promise) {
18      return knex.schema.dropTable('articles')
19
20  };
```

Figura 39 – Migration para categoria de artigos.

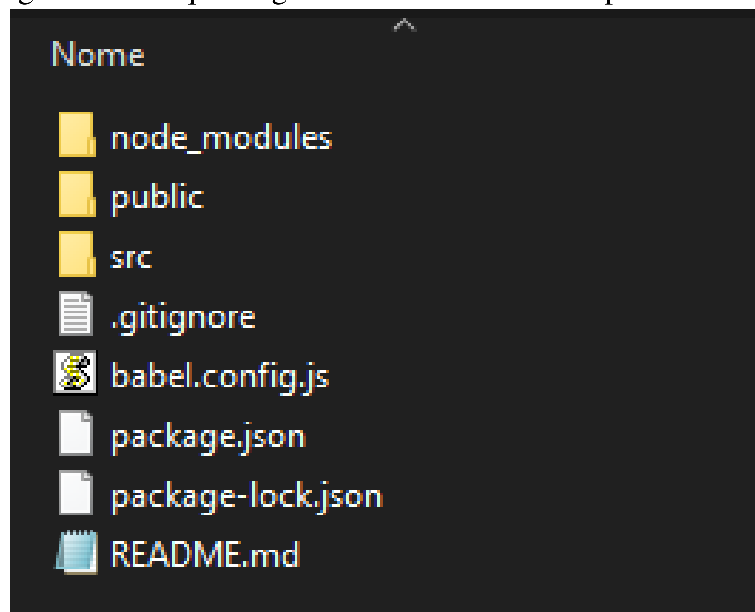
```
backend > migrations > JS 20210319170632_create_table_categories.js > ...
1
2   exports.up = function(knex, Promise) {
3       return knex.schema.createTable('categories', table =>{
4           table.increments('id').primary()
5           table.string('name').notNull()
6           table.integer('parentId').references('id')
7           .inTable('categories')
8       })
9
10  };
11
12  exports.down = function(knex, Promise) {
13      return knex.schema.dropTable('categories')
14
15  };
```

APÊNDICE B – ESTRUTURAÇÃO DO PROJETO *FRONTEND*

Para a criação do projeto *frontend*, é necessário obter todas as ferramentas básicas para montar a estrutura inicial. Como dito anteriormente, o framework utilizado foi o *VueJS*, e assim como outros frameworks como *Angular* e *React*, existem algumas etapas para montar uma aplicação *VueJS* sem maiores complicações. Toda a configuração e uso básico do *VueJS* pode ser acessado via documentação oficial em (VUEJS, 2021), que possui desde a instalação do framework até os arquivos de configuração que serão abordados adiante. O primeiro passo é acessar a pasta *frontend* pelo terminal do VScode e instalar o *VueJS* CLI(Command Line Interface), que nada mais é que uma interface de comando que facilita o desenvolvimento de aplicações *VueJS*. O comando para isso é o 'npm install -g @vue/cli'. De acordo com a documentação oficial do framework (VUECLI, 2019), 'O Vue CLI busca ser a ferramenta base padrão para o ecossistema *VueJS*, garantindo que as várias ferramentas de compilação funcionem perfeitamente em conjunto com padrões razoáveis, para que o foco principal seja em escrever a aplicação, sem perder tanto tempo revirando as configurações. Além disso, ainda oferece a flexibilidade de ajustar a configuração de várias ferramentas sem a necessidade de removê-las'.

Após a instalação do *Vue* CLI, podemos criar o arquétipo do projeto com o comando *"vue create nomedoprojeto"*, no terminal da pasta do projeto frontend, gerando todas as pastas e arquivos de configurações necessários como mostra a Figura 40.

Figura 40 – Arquivos gerados automaticamente pelo Vue CLI.

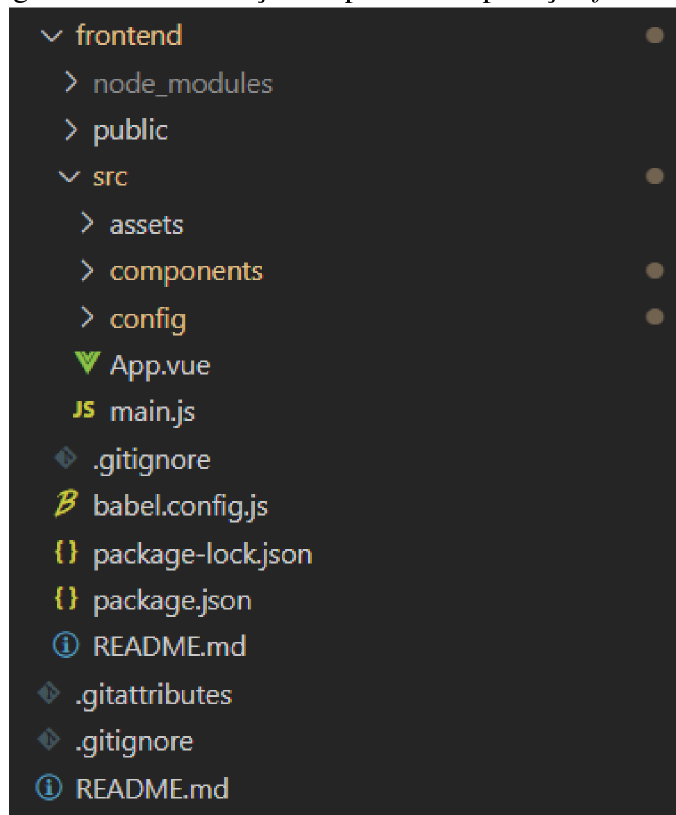


Fonte: Elaborada pelo autor.

A pasta *public* (Figura 41) contém os arquivos *"index.html"*, que possui algumas informações como o formato de codificação de caracteres e o arquivo *"logo.ico"*, onde será usado a logo da Clínica São Gonçalo na aba de navegação do browser. Na pasta *src*, temos os principais arquivos da aplicação, que serão organizados em pastas. A primeira delas é a pasta *Assets*, que conterá todas as imagens utilizadas no *frontend*. Em seguida, temos a pasta *components*, onde estarão todos os componentes *VueJS* das páginas da aplicação divididos em subpastas: *admin*, *article*, *auth*, *home* e *templates*.

A próxima pasta é a *config*, onde temos arquivos de configurações gerais, como as mensagens *toast* padronizadas e configuração do *axios*. Para finalizar, ainda na pasta *src* temos outros dois arquivos gerados pelo *client* do *VueJS*, são eles o componente *"App.vue"*, onde os principais componentes serão renderizados, mediante ou não à operações condicionais e o arquivo *"main.js"*, que conterá *imports* de arquivos de configurações essenciais para a execução do *frontend*, bem como a própria chamada/renderização do componente *"App.vue"*. Também no arquivo *"App.vue"* é utilizado um módulo de layout em grade(CSS Grid), que atribui seções da página para cada principal componente exibido.

Figura 41 – Distribuição de pastas na aplicação *frontend*.



Fonte: Elaborada pelo autor.

Em seguida, é necessário instalar as dependências, assim como no projeto *backend*, utilizando o gerenciador de pacotes do *NodeJS*. Assim como no projeto *backend*, todas as dependências baixadas ficarão em uma pasta chamada *node_modules*, que serão listadas a seguir.

B.1 DEPENDÊNCIAS UTILIZADAS NO FRONTEND

A seguir, serão apresentados todos os pacotes, *middlewares* e *frameworks* presentes no arquivo *"package.json"* e suas finalidades no projeto do *frontend*. São eles:

- **Axios**: Cliente HTTP baseado em *promises*, que intercepta e transforma requisições (*GET, PUT, POST*) e respostas (códigos de status) pelo navegador ou *NodeJS*;
- **Babel-eslint**: É uma biblioteca que permite usar o *ESlint* (analisador de códigos *javascript*) juntamente com código compatível com *Babel* (ECMAScript 6, tipos de fluxo, etc.). O *ESLint* por si só oferece suporte ao ES6, JSX (extensão de sintaxe da linguagem *javascript*), operadores *Spread* (espalhamento) e *Rest*. Qualquer coisa além disso requer *babel-eslint*;
- **Bootstrap-vue**: Biblioteca baseada no popular framework CSS bootstrap para aplicações *frontend*, fornecendo diversos componentes *VueJS* que renderizam páginas HTML com *bootstrap*;
- **Eslint**: É uma biblioteca para análise de código estática, com o intuito de identificar erros e padrões problemáticos em códigos *javascript*;
- **Eslint-plugin-vue**: Biblioteca com o propósito de encontrar erros de sintaxe, uso errôneo de diretivas e violações no uso das estilizações em arquivos .vue;
- **Font-awesome**: Biblioteca de ícones com propriedades vetoriais escaláveis e de fácil aplicação em páginas web;
- **Liquor-tree**: Pacote que auxilia na construção de um componente vue em formato de árvore, permitindo assim uma visão agradável das informações organizadas de forma hierárquica e lógica;
- **Vue**: Framework principal, utilizado para construção de interfaces de usuário em páginas web;
- **Vue-gravatar**: Pacote que cria avatares para perfis de usuários cadastrados na aplicação;
- **Vue-mq**: Utilizado na responsividade da aplicação, ao delimitar breakpoints para determinados tamanhos de telas;
- **Vue-router**: Dependência voltada para a definição de rotas para as páginas da aplicação *frontend*;

- **Vue-toasted**: Pacote que fornece uma solução simples e eficaz para mensagens *pop-up* como um *feedback* simples após alguma operação de requisição/resposta;
- **Vue2-editor**: Fornece um editor de textos completo para a página de escrita de artigos na aplicação;
- **Vuex**: Biblioteca para gerenciamento de estados em aplicações *VueJS*. Ele armazena dados sobre componentes da página que podem ser alterados após determinados eventos.