RENAN FERNANDES GUIMARÃES

RELATÓRIO DE ESTÁGIO MGCODE

LAVRAS – MG

2022

RENAN FERNANDES GUIMARÃES

RELATÓRIO DE ESTÁGIO MGCODE

Relatório de estágio apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para obtenção do título de Bacharel.

Orientador

LAVRAS - MG

2022

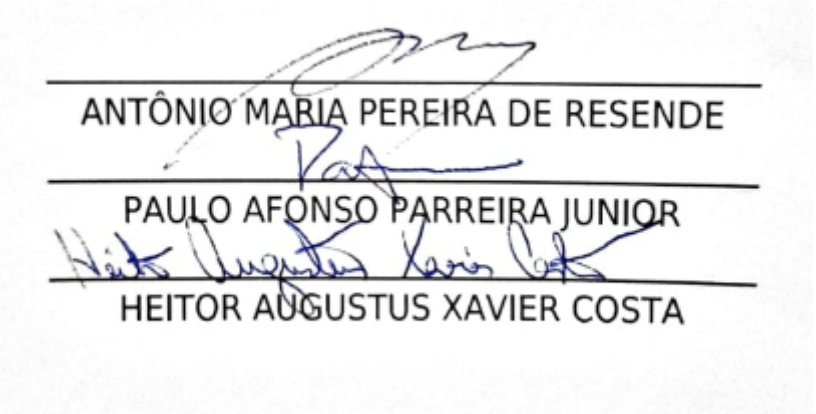
RENAN FERNANDES GUIMARÃES

RELATÓRIO DE ESTÁGIO MGCODE

Relatório de estágio apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para obtenção do título de Bacharel.

Aprovado em 14/09/2022

BANCA EXAMINADORA



AGRADECIMENTOS

Agradeço a todos os envolvidos durante meu processo de aprendizado, meus amigos e familiares que me apoiaram durante a jornada acadêmica e me auxiliaram em minhas dificuldades. Agradeço também a Universidade Federal de Lavras por proporcionar um ambiente incrível possibilitando construir amizades, conhecimento e diversão, como também todos os professores atenciosos e sempre dispostos a ajudar com minhas dificuldades. Agradeço também ao Professor Antônio Maria Pereira de Resende ao dispor do seu tempo para orientar esse trabalho de conclusão de curso.

RESUMO

Este trabalho apresenta o relatório do estágio realizado na empresa de consultoria e desenvolvimento de softwares MGCode. Foram levantadas as principais atividades realizadas e uma visão detalhada da API Rest MGPro. São demonstradas as diferentes áreas de conhecimento exploradas e como a experiência em uma empresa em que suas atividades são focadas em programação e manipulação de dados pode contribuir para o desenvolvimento das habilidades técnicas e trabalho em equipe de um profissional.

Palavras-chave: Relatório de estágio; Inteligencia de negocios; MGCode; API Restful; Áreas de Conhecimento;

ABSTRACT

This work presents the report of the internship carried out at MGCode, a consulting and software development company. The main activities were described and a detailed view of the Rest MGPro API was raised. The different areas of knowledge explored are listed and how the experience in a company in which its activities are focused on programming and data manipulation can contribute to the development of technical skills and teamwork of a professional.

Keywords: Internship Report; Business Intelligence; MGCode; API Restful; Knowledge Areas;

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BI - *Business Intelligence*

SQL - *Standard Query Language*

NPM - Node *Package Manager*

API - *Application Programming Interface*

HTTP - *Hypertext Transfer Protocol*

JSON - Javascript *Object Notation*

REST - *Representational State Transfer*

ORM - *Object-Relational Mapping*

CRUD - *Create Read Update Delete*

DTO - *Data Transfer Object*

DBA - *Database Administrator*

ISP - *Internet Access Provider*

URL - *Uniform Resource Locator*

JWT - JSON *Web Token*

SUMÁRIO

1 INTRODUÇÃO

O relatório técnico de estágio apresentado neste documento é elaborado no âmbito da disciplina PRG514 – Estágio Supervisionado/TCC , visando à conclusão do curso de graduação de Sistemas de Informação na Universidade Federal de Lavras – UFLA.

O estágio foi realizado na equipe de desenvolvimento da empresa MGCode com o início em 31/05/2022 e término em 26/08/2022. Todas as atividades foram realizadas presencialmente no escritório da empresa localizado em Perdões, Minas Gerais. A oportunidade de estágio foi aberta na empresa devido a necessidade de profissionais para atuar no âmbito de prestação de serviços em BI (*Business Inteligence*), com o objetivo de crescer como uma empresa de desenvolvimento de software usando as tecnologias do mercado atual.

A escolha da empresa representa para o estagiário uma oportunidade de conhecer o mercado de desenvolvimento de software e seus desafios, como também a atuação na área de BI e análise de negócios. Com a equipe de desenvolvedores da MGCode visando o crescimento da empresa e de seus colaboradores, o estagiário ao se inserir nesse ambiente, tem os recursos para evoluir sua carreira como um programador e analista de negócios.

1.1 Objetivos

O objetivo principal deste relatório é a apresentação das principais atividades de desenvolvimento e BI atribuídas durante o período de estágio, com uma visão detalhada da API (*Application Programming Interface*) *restful* denominada MGPro. Além disso, é apresentado o crescimento individual obtido durante o estágio, na área técnica e no desenvolvimento pessoal.

1.2 Organização do trabalho

O trabalho está organizado em quatro capítulos, o primeiro capítulo consiste na introdução abordando os objetivos e organização do trabalho. O segundo capítulo contém o referencial teórico, o qual apresenta a empresa MGCode, seu ramo de

negócio e o processo de seleção para a vaga de estágio. O terceiro capítulo contém a descrição das atividades, onde é apresentada uma síntese das atividades e cursos realizados e um detalhamento nas atividades de prestação de serviços BI e desenvolvimento da API Restful MGPRO. O quarto capítulo é dedicado às conclusões obtidas do estagiário ao fim do seu período de estágio, seus ganhos pessoais, ganhos da empresa e uma relação das matérias e atividades do estágio.

2 REFERENCIAL TEÓRICO

Neste capítulo, é apresentado a história da empresa, seu ambiente de trabalho e o processo de seleção do estágio.

2.1 A empresa MGCode

A empresa MGCode é situada no centro da cidade de Perdões no estado de Minas Gerais. A empresa iniciou como parte da provedora de Internet Minasnet, posteriormente desvinculou-se e tornou-se uma *startup* em consultoria e desenvolvimento de software.

O foco da empresa no primeiro momento foi a prestação de serviços em integração de sistemas e migração de dados, sendo essa a área de expertise do sócio proprietário. Durante 3 anos a MGCode prestou serviços para diversas empresas da área de *healthcare,* essas empresas tratam de sistemas para o setor de saúde, como Siemens, Agfa e Dedalus. A MGCode também desenvolve soluções tecnológicas para empresas regionais, pelo sistema de vendas *desktop* “DREAM” desenvolvido em Delphi para computadores com baixo poder de processamento, visando atender as necessidades de administração de vendas no comércio de produtos diversos e na administração de sócios em clubes esportivos. Apesar de ter se desvinculado da Minasnet, a MGCode ainda presta serviços para a provedora de internet por ter expertise na área. A Minasnet recentemente tornou-se parte da Sempre Internet, uma empresa de telecomunicações que atua em grande parte de Minas Gerais. A MGcode também investe em soluções *web*, no desenvolvimento de sistemas para seus clientes, abrindo portas para desenvolvedores da região com interesse em trabalhar com desenvolvimento e consultoria tecnológica.

A empresa tem ao todo dez colaboradores, a sua maioria trabalha presencialmente no escritório e os outros colaboradores trabalham no modelo *home office,* esse modelo misto de trabalho foi devido a dificuldade de encontrar profissionais capacitados na região, resultando na necessidade de buscar colaboradores distantes de sua sede. Ao prestar serviços para as empresas parceiras, a MGCode trabalha em equipe com os setores de TI e desenvolvimento

dessas empresas, assim formando um time misto e sempre agregando conhecimento no seu acervo intelectual ao se relacionar com profissionais de outras empresas.

A MGcode adota um modelo simples de gerência por ter poucos colaboradores locais. A empresa possui um time de três desenvolvedores para o desenvolvimento de *Front-end*, três desenvolvedores para o *Back-end* e os demais são *Full-stack*. A meta da empresa é nivelar o conhecimento dos colaboradores tornando-os *Full-stacks*. Um gerente de projetos é responsável por distribuir as tarefas e gerenciar os recursos dos projetos e contratos da empresa.

2.2 Processo de Seleção do Estágio

Foram ofertadas duas vagas de estágio para desenvolvedores *full-stack*, essas vagas foram divulgadas pelos colaboradores já integrantes da empresa. No total houve quatro desenvolvedores que se candidataram para essa vaga. Para participar do processo de seleção os candidatos enviaram um currículo simples para explanar suas habilidades técnicas e suas aspirações de carreira. Os requisitos da vaga não incluíam habilidades técnicas, porém era explicitado a necessidade de pessoas pró-ativas e autodidatas.

O processo de seleção consistiu em um teste prático realizado presencialmente no escritório da empresa. O teste proposto foi desenvolver um sistema para consulta, cadastros, alterações e remoções de usuários. Esse sistema deve ter um *front-end* e *back-end* básico sem nenhuma regra específica além das linguagens na sua implementação, as linguagens e tecnologias usadas foram: banco de dados MYSQL e PHP para o *back-end* e para o *front-end* Javascript, HTML5 e CSS3. Uma semana antes do teste prático foram disponibilizados cursos para essas tecnologias, esses cursos tinham material suficiente para os participantes conseguirem realizar o teste.

O critério de aprovação foi a análise de desempenho dos participantes nas seguintes características: o tempo total gasto pelo participante para desenvolver o sistema, a clareza do código, o uso inteligente de recursos como bibliotecas e componentes open source e a usabilidade do sistema.

3 DESCRIÇÃO DAS ATIVIDADES

Esse capítulo é composto pelas atividades realizadas durante o estágio na MGCode. O capítulo 3.1 é uma sintetização das atividades para uma noção evolutiva, respeitando a ordem dos acontecimentos. Nos capítulos 3.2 e 3.3 são detalhadas as atividades de desenvolvimento BI e a API Restful MGPro respectivamente.

3.1 Síntese das atividades

A síntese das atividades aborda as atividades e treinamentos técnicos realizados durante o tempo de estágio na empresa.

3.1.1 Escolha de atuação e Atuação no BI

Ao se juntar à equipe de desenvolvedores da MGCode, duas áreas de atuação inicial são propostas, Front-end e *Back-end*. O estagiário tem a liberdade de escolher sua área inicial para a realização de cursos em tecnologias específicas disponibilizados pela empresa, porém posteriormente é necessário do colaborador aprender as duas frentes de desenvolvimento. A escolha inicial do estagiário foi se capacitar em *Back-end* usando como tecnologia base o NodeJS.

Em paralelo ao treinamento, o estagiário é inserido na prestação de serviços BI, realizada em conjunto com o DBA (*Database Administrator*) da empresa Sempre Internet, a qual foi prestado esses serviços. Durante a primeira semana de estágio o estagiário foi treinado para entender a estrutura do banco de dados e as regras de negócios básicas, com intuito de adquirir um conhecimento significativo e relacioná-lo ao entendimento das tabelas deste banco de dados. Esse treinamento foi feito com o auxílio do DBA em conjunto com um de seus colegas de equipe, o qual prestava esse tipo de serviço.

Portanto, a primeira divisão de horários foi 4 horas diárias no atendimento e desenvolvimento de soluções BI e 3 horas diárias de estudos na tecnologia escolhida. A prestação de serviços BI para a Sempre Internet foi contínua até o

término do estágio.

3.1.2 Cursos introdutórios realizados

Nas três primeiras semanas, foram realizados os cursos de programação do portal de ensino B7Web chamados “Introdução ao NodeJS” e “Introdução ao Typescript” (disponíveis em https://b7web.com.br/). Esses cursos foram adquiridos pela empresa para o treinamento de novos colaboradores para capacitá-los nas novas tecnologias de mercado.

Aproximadamente, foram realizadas 45 horas de treinamento, alternando entre a prática, teoria e implementação de pequenos projetos para a fixação do conteúdo, com a supervisão dos colaboradores para acompanhar a evolução do estagiário.

3.1.3 Projeto de desenvolvimento

Logo após o término dos cursos, surgiu a ideia na empresa de desenvolver uma API em NodeJS seguindo o padrão REST (*Representational State Transfer*) de arquitetura. Até esse momento, todos os sistemas tinham seu *Back-end* implementados em PHP 7, portanto houve a necessidade de substituir esses sistemas. Além da necessidade de substituir sistemas legados, a construção de um novo sistema em NodeJS também padroniza a linguagem para o *back-end* e *front-end* pois ambos usam o Typescript, diminuindo assim a curva de aprendizado do desenvolvedor ao mudar de *stack*. O objetivo desse sistema é se comunicar com o aplicativo *front-end* em construção, implementado com o *framework* ReactJs, o qual possui o Typescript como sua sintaxe principal.

Após uma pesquisa de mercado e conformidade com a linguagem usada no *Front-end*, a equipe decidiu usar o *framework* NestJS para o desenvolvimento dessa nova API Restful. O desenvolvimento da API se estendeu até o fim do estágio.

3.2 Desenvolvimento BI

O desenvolvimento BI é um serviço prestado para a empresa de telecomunicações Sempre Internet. O serviço baseia-se em sintetizar informações do banco de dados em formas visuais ou lógicas (tabelas). As demandas da empresa seguem um fluxo padronizado, na forma de abertura de chamados para a maioria dos casos.

3.2.1 Ambiente Organizacional

A Sempre Internet é uma empresa baseada na estrutura organizacional matricial, ou seja, existem diversos setores e cada um desses setores tem seu gerente. O setor responsável pelo BI é chamado de SIS/TD (Sistemas e Transformações Digitais). O SIS/TD é responsável pela administração do principal software de gestão ISP (*Internet Access Provider*) usado pela empresa denominado Hubsoft. Cada setor tem sua necessidade administrativa, o SIS/TD existe para apoiar outros setores em soluções tecnológicas para a administração da empresa e prover informações de negócios. Como esse setor existe para auxiliar outros setores, seu contato se limita em apenas colaboradores internos da organização e empresas de auditorias, portanto não entra em contato direto com clientes da empresa.

3.2.2 Fluxo de atendimento

O fluxo de atendimento começa a partir de chamados modelo *helpdesk* abertos com software GLPI. Nesse software algum representante do setor cadastra o chamado na categoria “BI e relatórios”, sintetizando sua necessidade por forma de texto ou arquivos e apontando quais dados serão necessários e quais regras devem ser aplicadas ao extrair esses dados. Entretanto, nem sempre tudo é explicado no primeiro contato, já que em muitas ocasiões, o requerente não sabe quais dados estão a disposição para sua análise. Portanto, caso haja necessidade, o responsável pelo desenvolvimento do chamado entra em contato com o solicitante, com o intuito

de entender melhor a necessidade e apresentar as possíveis soluções, esse contato pode ocorrer por troca de emails ou reuniões virtuais.

O chamado entra em uma espécie de fila ao ser criado, onde cada chamado tem seu nível de prioridade, esses níveis consistem em “Baixa”, “Normal” e “Alta”. As prioridades impactam no tempo hábil para a resolução do chamado, respectivamente o tempo limite para a resolução é 32 horas, 16 horas e 8 horas entendendo que esse horário leva em consideração apenas o tempo da jornada de trabalho diária de 8 horas. Geralmente o nível de prioridade é decidido pelo próprio requerente. Essa fila de prioridade é visível apenas para o setor a qual pertence a categoria do chamado.

Os chamados possuem o campo de status, que indica o seu andamento, sendo “Aguardando”, “Pendente”, “Em andamento” e “Finalizado”. Após o registro do chamado, o status por padrão é “Aguardando”, contudo, ao ser iniciado o atendimento, o status muda para “Em andamento”. Um chamado tem seu status “Finalizado” apenas após a validação lógica do requerente e a validação de dados por parte do DBA.

3.2.3 Tipos de chamados

Dentro dos chamados de BI e relatórios, existem três tipos de Categorias: “Criação de relatórios e *dashboards*”, “Ajustes de relatórios e *dashboards*” e “Erros de relatórios e *dashboards*”.

Os chamados na categoria “criação de relatórios e *dashboards”* consistem em requisições de novas análises não implementados. Nessa categoria, ao abrir a requisição, o requerente recebe um *template* opcional de perguntas objetivas. Esse *template* tem cinco campos para o preenchimento: “Qual o objetivo do relatório?”, “Quais dados são necessários?”, “Qual o modelo de visualização desejado (ex. Gráficos de pizza, Indicadores de meta, tabelas excel, etc)?”, “Quais filtros são necessários no relatório? (ex filtro de data, tipo de pessoa, tecnologia usada, etc)”, “Quais são os meios de contato e disponibilidade para sanar dúvidas? (email, whatsapp, discord, etc)”.

A categoria “Ajustes de relatórios e *dashboards*” consiste nas requisições

para implementar novas regras de negócios adicionando ou removendo campos e filtros lógicos em relatórios em produção. Esse tipo de requisição tem um *template* com duas perguntas: “Qual o nome do relatório/*dashboard* em questão?” e “Qual o ajuste necessário?”.

A categoria “Erros de relatórios e *dashboards*” tem a finalidade de reportar erros em relatórios e *dashboards* em produção. O *template* de perguntas segue a idéia do modelo anterior: “Qual o nome do relatório/*dashboard* em questão?”, “Qual ajuste necessário?”, além disso existe um campo para *upload* de anexos para caso seja necessário ter alguma prova do erro. Os chamados dessa categoria geralmente são abertos quando há instabilidade no servidor que ocasionalmente acarreta a perda de acesso aos relatórios.

3.2.4 Desenvolvimento e entrega do produto final

O desenvolvimento do produto é feito pelo software Metabase, uma ferramenta BI que possibilita a criação de relatórios e dashboards utilizando *queries* SQL(*Standard Query Language*) executadas no banco de dados, esse software é acessível por navegadores. Geralmente, a comunicação entre o solicitante e o desenvolvedor é contínua até o término e entrega do produto final. Utilizando um *link* produzido pelo Metabase, o requerente tem o acesso ao produto em desenvolvimento, facilitando seu acompanhamento.

Todos os dados usados para a construção dos relatórios e *dashboards* são consultados em um banco de dados Postgres. Por regra, como são muitos dados, é sempre feito um filtro de data ou categoria para diminuir o tempo de resposta da consulta inicial. Esses filtros dinâmicos permitem ao usuário do *dashboard* buscar o período ou as categorias desejadas, tornando o relatório um produto reusável em diversas ocasiões.

Após a validação de dados e estrutura do produto, o *link* gerado pelo Metabase é desativado, e o relatório é integrado na plataforma de gestão da empresa, essa integração é feita com um *iframe*, um componente HTML produzido pelo Metabase. A disponibilização do produto final no sistema de gestão, permite apenas os usuários com a permissão de visualização acessarem o relatório final.

3.3 Desenvolvimento MGPro

O projeto MGPro nasceu pela necessidade de inovação dentro da empresa que busca usar tecnologias novas no mercado, como ReactJS para *Front-end* e o NodeJS desenvolvido com o *framework* NestJS para o *Back-end*. A comunicação entre o *Front-end* e o *Back-end* é feita por meio de requisições HTTP (*Hypertext Transfer Protocol*), respeitando o padrão REST de arquitetura.

O projeto MGPro tem em sua primeira etapa um escopo limitado de um módulo base sem regras específicas de negócios contendo a implementação do CRUD (*Create Read Update Delete*), um sistema de autenticação de usuários e permissões para esses usuários. A intenção desse desenvolvimento é ter um sistema base com a possibilidade de ter módulos distintos adicionados futuramente, com a finalidade de atender clientes de diversas necessidades administrativas.

Após a definição do escopo de projeto em uma reunião, o estagiário formalizou as informações em uma tabela com os principais requisitos técnicos para a implementação do módulo base. Essa tabela contém o total de quatro requisitos técnicos, representados na Tabela 1.

Tabela 1: Requisitos Técnicos da API Restful.

Código	**Requisito**	**Descrição**
RT1	Modelagem de tabelas para entidades	Realizar a modelagem das tabelas do módulo Base para o sistema MGPRO, respeitando a especificação das entidades, a entidade Usuário deve conter os atributos: id, nome, email, senha, created_at, updated_at, updated_by, avatar, last_login. A entidade Empresa deve conter os atributos: id, nome_razao_social, nome_fantasia, documento, email, telefone, endereco;
RT2	Rotas para operações CRUD nas entidades	Desenvolver rotas para operações CRUD para as entidades Usuário e Empresa, as operações devem permitir a Adição, leitura, atualização e remoção de objetos;
RT3	Sistema de Permissões de usuário	Os usuários devem ser vinculados a um grupo de permissões, o qual será usado para definir o acesso das rotas, as permissões devem ser separadas para todas as operações CRUD;
RT4	Sistema de Autenticação	O Acesso "HTTP" das requisições deve ter a segurança de verificação de tokens JWT ("JSON" Web Token), a obtenção dessa token será feita por meio da rota de *Login*.

Fonte: autor

O requisito técnico RT1 representa a necessidade da modelagem das tabelas “usuario” e “empresa”, entidades iniciais do módulo base. O RT2 específica que são necessárias todas as rotas para as operações CRUD. O RT3 descreve a necessidade do desenvolvimento de um sistema de permissões de usuários para controlar o acesso das rotas descritas no RT2. O RT4 documenta a necessidade de um sistema de autenticação, esse sistema deverá usar a JWT (JSON *Web Token*) como chave de segurança, essa *token* é obtida pelo *login* do usuário.

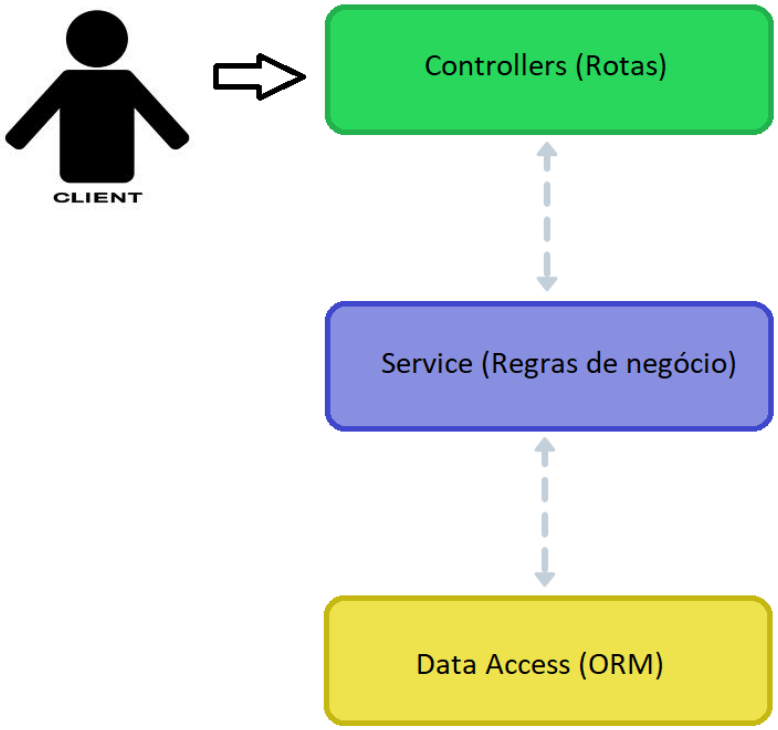
3.3.1 NestJS

Segundo a documentação oficial do NestJS (2022), o Nestjs é um framework para construção de servidores Nodejs, sua proposta é ser eficiente e escalável, usa-se o Javascript progressivo com suporte total ao Typescript.

A sua arquitetura de software é composta por três camadas, oque torna a aplicação encapsulada. Cada camada é responsável por uma parte lógica da aplicação, as camadas presentes na arquitetura são: *Controllers*, *Service* e *Data Access.*

Os *controllers* são responsáveis pelas rotas de acesso e da segurança de autenticação, se presente. Por essa camada a requisição do cliente HTTP é reconhecida, filtrada e direcionada para a próxima camada. Os *services* são responsáveis pela implementação da lógica de negócio, por exemplo, as operações CRUD e os métodos para definir como os dados serão inseridos, modificados e deletados. O *data access* é responsável em se comunicar com o banco de dados e possui as funções de inserção, atualização, remoção e seleção definidas pelos serviços. Geralmente essa camada é representada por um ORM (*Object-Relational Mapping*) de escolha do desenvolvedor. Os arquivos dessa camada são representados como *entity* o qual possui os atributos e métodos da classe e arquivos DTO (*Data Transfer Object*) usados para vários tipos de operações que incluem entrada e saída de dados, por exemplo um filtro validador. A Figura 1 representa o fluxo da aplicação.

Figura 1: Fluxo da arquitetura Nestjs



fonte: autor

O software cliente, representado como “CLIENT” na figura anterior, inicia o processo de requisição na API. A primeira camada que a requisição encontra é o *Controller,* que interpreta a requisição aplicando recursos de seguranças especificados em cada contexto de aplicação. Em seguida o *controller* faz uma chamada para a próxima camada, os *services,* realizam as operações de regras de negócios e consolidam essas informações através da próxima camada. A última camada o *Data Access* retorna então o resultado do procedimento especificado no *service,* e a informação retorna para o *controller* que responde o autor da requisição.

3.3.2 Configurações iniciais

As configurações iniciais envolvem a criação do banco escolhido e a instalação do servidor NodeJS implementado com NestJS.

O Postgres foi selecionado como o banco de dados principal para a

aplicação. A escolha foi tomada por causa da experiência do estagiário e pela fácil integração com a biblioteca de mapeamento objeto-relacional TypeORM, utilizada pelo NestJS. Com a criação desse banco de dados, um usuário raiz é definido para ser usado pela aplicação com intuito de ter o total controle sobre as operações SQL e manutenção de tabelas.

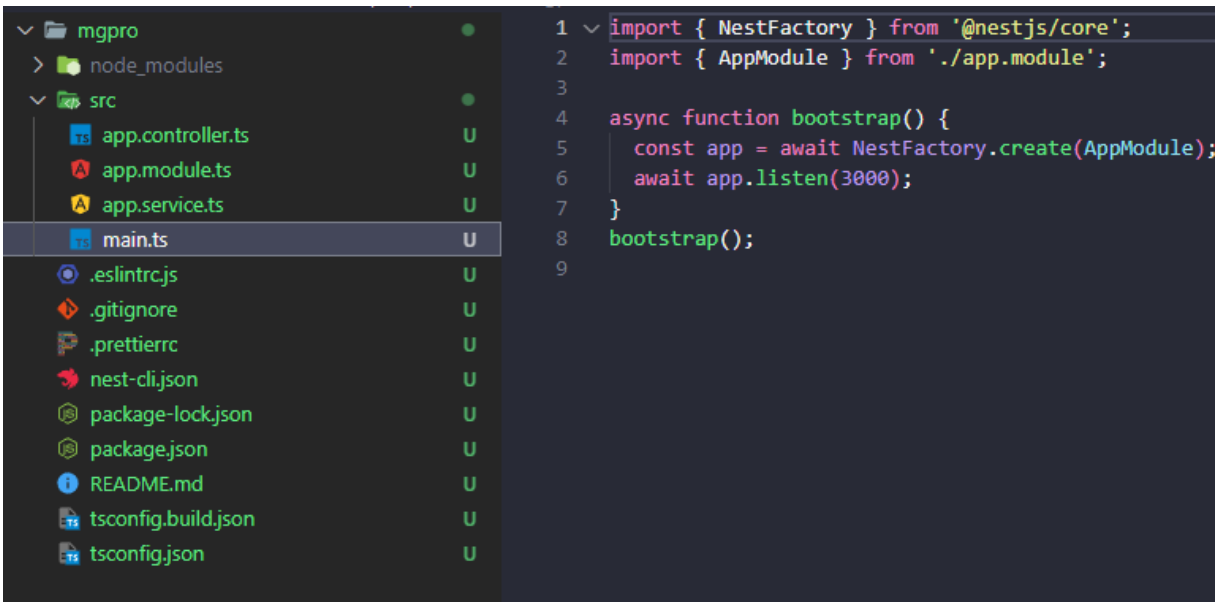
Após a criação do banco de dados, o próximo passo é a instalação inicial do NestJS que ocorre por meio de comandos executados no terminal.

O primeiro comando executado foi “npm i -g @NestJS/cli”, esse comando utiliza o NPM (*Node Package Manager)* para instalar o cliente global do NestJS. Após a instalação, o sistema operacional consegue reconhecer os comandos com o prefixo *nest*, utilizados para executar funções do *framework*.

O segundo comando executado foi “nest new mgpro”, gerando uma estrutura de arquivos e pastas iniciais: “node_modules”, pasta padrão para armazenamento dos arquivos fontes de bibliotecas do NodeJS, “src”, pasta principal do projeto, e os arquivos JSON utilizados para configurações.

O NestJS cria três arquivos como exemplo inicial, os quais representam uma unidade lógica (“app.controller.ts”, “app.module.ts” e “app.service.ts”). Cria-se também o arquivo responsável pela parte de inicialização do servidor (“main.ts”), nesse arquivo é configurada a porta do servidor, que como padrão responde na porta 3000. A Figura 2 demonstra o resultado dessa configuração inicial.

Figura 2: Estrutura Inicial e conteúdo do arquivo “main.ts”.



Fonte: autor

Após a criação da estrutura inicial, uma plataforma "HTTP" é configurada. O NestJS suporta nativamente duas plataformas, sendo elas ExpressJS e Fastify, dentre elas a plataforma utilizada foi o ExpressJS por causa da maior quantidade de recursos e documentações disponíveis na Internet. Para instalar a plataforma ExpressJS executa-se o comando “npm i -s @NestJS/platform-express”. Após a execução desse comando as dependências da plataforma ExpressJS são integradas como um módulo para o NestJS.

O último passo antes de iniciar o desenvolvimento é instalar o TypeORM, essa biblioteca ORM disponibiliza funcionalidades como a interação entre a aplicação e banco de dados e facilitadores para filtragem de dados. Para isso executa-se o comando: “npm i --save @NestJS/TypeORM TypeORM postgres”. Após a execução desse comando, as funcionalidades do TypeORM para o banco de dados Postgres são disponibilizadas e configuradas como o padrão.

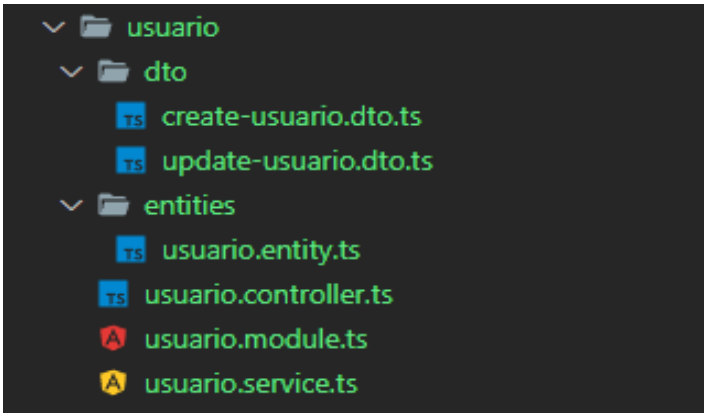
3.3.3 Criação das Entidades

A primeira etapa do desenvolvimento se baseia em modelar as entidades utilizando as funcionalidades do TypeORM. Para isso o NestJS possui um atalho de

comando para facilitar a criação de uma estrutura base da entidade nos modelos que o TypeORM espera. A primeira entidade criada foi a usuário, usada para autenticação. O comando usado foi o “nest g resource Usuario” para gerar a entidade e seus arquivos base respeitando o padrão de arquitetura do *framework*.

Logo após a execução do comando, um *prompt* aparece com as opções arquiteturais usadas nativamente pelo NestJS. Essas opções são apresentadas como: “REST API”, “GraphQL (code first)”, “GraphQL (schema first)”, “Microservice (non-HTTP)” e “WebSockets”. A escolhida para esse desenvolvimento foi a “Rest API”, pois, o projeto baseia-se na construção de uma API Restful. A Figura 3 representa a estrutura criada.

Figura 3: Estrutura criada pelo comando “g resource”.



Fonte: autor

Além dos arquivos lógicos base (*controller, module e service*), são gerados dois tipos de arquivos “.dto”, usados para a criação e edição de usuários e um arquivo “.entity”, usado para representar a entidade usuário.

Os atributos para a entidade usuário são criados seguindo as especificações do RT1, sendo eles “id” e “updated_by” como tipo inteiro, “nome”, “email”, “senha” e “avatar” como *string* , “created_at”, “updated_at” e “last_login” como tipo de data *datetime*.

Os atributos “updated_by”, “created_at” e “updated_at” são ocultos do usuário, ou seja, são atributos somente visíveis no *Back-end* para fim de depuração e *logs*.

A senha do usuário, por se tratar de um dado sensível, é armazenada criptografada, para isso foi utilizada a funcionalidade de encriptação da biblioteca Bcrypt. Usando essa funcionalidade, é possível interceptar as senhas antes de serem salvas em um banco de dados, executando o encriptamento da *string* referente a senha.

A entidade “empresa” tem os atributos “id” como inteiro, “nome_razao_social”, “nome_fantasia”, “documento”, “email” e “telefone” como *string*. Para o endereço foi criada outra tabela chamada “endereco”, essa tabela contém os campos “id”, “numero_rua” como inteiros, e os campos “rua”, “bairro”, “cidade”, “estado”, “complemento” e “tipo” como *string.* Para ter conexão com a empresa, uma chave estrangeira foi criada usando o *id* da tabela “endereco” contido na tabela empresa, representando uma relação de um para um. A tabela endereço também poderá ser utilizada por outras entidades do sistema. Essa modelagem não é final, podendo ser mudada de acordo com as regras de negócios específicas de projetos futuros.

3.3.4 Criação das Rotas

A criação de rotas limita-se ao arquivo *controller*. Nesse arquivo toda a lógica envolvendo a requisição do usuário é implementada.

O acesso a rota por requisições HTTP externas é alcançado com a URL (*Uniform Resource Locator*) gerada pela API. A URL base durante o desenvolvimento encontra-se em um servidor local acessível em “localhost”. A porta para esse servidor é 3000, parametrizada no arquivo “main.ts”. A URL das rotas são definidas pelo nome da entidade em questão, portanto o acesso a rotas da entidade usuário é “localhost:3000/usuario”.

As rotas para requisições de informações usam o método *Get*. Para consultar todos os usuários é necessária a requisição para a URL “localhost:3000/usuario”. Para requisitar um usuário, usa-se “localhost:3000/usuario/:id”, onde “:id” representa o id do usuário procurado.

As rotas de inserção de usuário usam a mesma URL, porém o método utilizado é o *Post*. Esse método é composto por um objeto JSON o qual contém as informações do usuário. A validação dessas informações passadas no JSON são

feitas usando um objeto DTO importado do arquivo “.dto” referente a criação de entidades. Esse arquivo faz a comparação do objeto JSON recebido, caso essas informações recebidas estejam em conformidade com o objeto DTO, a requisição é aceita e o objeto é passado para o serviço realizar as operações. Entretanto, caso não estejam em conformidade, a API retorna o status de código 422 para sinalizar ao cliente a existência de erros referente ao objeto recebido. O conteúdo da mensagem de erro contém as propriedades esperadas pelos atributos da entidade.

A rota de atualização utiliza o método *Patch*, semelhante ao *Post, por* ter seu corpo composto por um objeto JSON. Esse método altera os dados de uma entidade existente em apenas campos alterados, sendo necessário enviar somente os atributos alterados. A identificação do objeto é feita pelo id, na URL “localhost:3000/usuario/:id”. No caso do *Patch* o DTO utilizado é referente a atualização, esse DTO verifica se os campos recebidos no JSON estão em conformidade especificadas no objeto, mas não obriga a ter todos os campos.

A rota responsável por deletar as entidades usa o método *Delete*. Essa rota não espera um corpo, sendo similar ao *Get,* contém a mesma URL para acesso. Na Figura 4 é exemplificado o *controller* da entidade usuário.

Figura 4: Classe *controller* da entidade usuário.

```
1  import { Controller, Get, Post, Body, Patch, Param, Delete, Request } from '@nestjs/common';
2  import { UsuarioService } from './usuario.service';
3  import { CreateUsuarioDto } from './dto/create-usuario.dto';
4  import { UpdateUsuarioDto } from './dto/update-usuario.dto';
5  import { AbilityGuard } from 'src/ability/ability.guard';
6  import { JwtGuard } from 'src/auth/login/jwt.guard';
7
8  @UseGuards(JwtGuard, AbilityGuard)
9  @Controller('usuario')
10 export class UsuarioController {
11   constructor(private readonly usuarioService: UsuarioService) { }
12
13   @CheckAbilities({ action: Action.Create, subject: 'Usuario' })
14   @Post()
15   create(
16     @Body() createUsuarioDto: CreateUsuarioDto,
17     @Request() req) {
18     return this.usuarioService.create(createUsuarioDto, req.user.id);
19   }
20
21   @CheckAbilities({ action: Action.Read, subject: 'Usuario' })
22   @Get()
23   findAll() {
24     return this.usuarioService.findAll();
25   }
26
27   @CheckAbilities({ action: Action.Read, subject: 'Usuario' })
28   @Get(':id')
29   findOne(@Param('id') id: string) {
30     return this.usuarioService.findOne(+id);
31   }
32
33   @CheckAbilities({ action: Action.Update, subject: 'Usuario' })
34   @Patch(':id')
35   update(
36     @Param('id') id: string,
37     @Body() updateUsuarioDto: UpdateUsuarioDto,
38     @Request() req) {
39     return this.usuarioService.update(+id, updateUsuarioDto, req.user.id);
40   }
41
42   @CheckAbilities({ action: Action.Delete, subject: 'Usuario' })
43   @Delete(':id')
44   remove(@Param('id') id: string) {
45     return this.usuarioService.remove(+id);
46   }
47 }
```

fonte: autor

A Figura anterior representa a classe *controller* para a entidade usuário. Nessa classe estão implementadas as medidas de controle de acesso explicadas nos capítulos 3.3.6 e 3.3.7. Percebe-se que a próxima camada está importada no construtor da classe, na linha *11*. Os métodos do *controller* chamam as funções do *service* e retornam para o autor da requisição as respostas desses métodos.

3.3.5 Criação dos Serviços e Conexão com TypeORM

Os serviços representam a camada da lógica de negócios. As funções desta camada são chamadas pelo *controller,* como mostrado na Figura 4, o qual é passado o objeto DTO no caso das rotas *Post* e *Patch*.

Para buscar dados existentes ou inserir dados, é necessário usar as funcionalidades ORM disponibilizadas pela biblioteca TypeORM. Com ela é possível construir *queries* dinâmicas por configurações ou simplesmente executar *queries* nativas. Para indicar a conexão com o banco de dados, é necessária uma configuração no módulo principal onde são parametrizados os atributos do objeto de configuração a fim de criar uma conexão. Esses atributos são: “host”, “port”, “username”, “password” e “database”. Outros parâmetros extras de configuração são necessários, como a lista de entidades passadas como um *array* e o tipo de banco de dados que, no caso do projeto, é usado o Postgres.

Outro parâmetro importante porém opcional é o *boolean “*synchronize”, que quando sinalizado como *true* é habilitado. Esse parâmetro permite a configuração automática das entidades no banco de dados, ou seja, ao criar um arquivo “.entity”, essa entidade e seus atributos são refletidos diretamente no banco de dados quando o servidor é iniciado. Essa funcionalidade permite bastante agilidade ao desenvolver, mas é altamente perigosa em ambientes de produção, de forma que, ao deletar uma propriedade no arquivo “.entity”, todos os dados dessa propriedade são excluídos do banco de dados sem a possibilidade de recuperação.

Ao configurar a conexão com o banco de dados, as funcionalidades de repositório do TypeORM são habilitadas, as quais possibilitam maneiras eficientes de buscar ou inserir dados. Algumas delas exemplificadas na Figura 5:

Figura 5: Exemplo de alguns métodos do TypeORM.

```
1  import { Injectable, HttpException } from '@nestjs/common';
2  import { InjectRepository } from '@nestjs/typeorm';
3  import { CreateUsuarioDto } from './dto/create-usuario.dto';
4  import { UpdateUsuarioDto } from './dto/update-usuario.dto';
5  import { Usuario } from './entities/usuario.entity';
6  @Injectable()
7  export class UsuarioService {
8
9    constructor(
10     @InjectRepository(Usuario)
11     private userRepo: Repository<Usuario>,
12   ) { }
13
14   async create(createUsuarioDto: CreateUsuarioDto, userId: number) {
15     try {
16       const user = this.userRepo.create(createUsuarioDto);
17       user.updatedBy = userId;
18       return await this.userRepo.save(user);
19     } catch (err) {
20       throw new HttpException(err.message, err.status);
21     }
22   }
23
24   async findAll() {
25     try {
26       return await this.userRepo.find({
27         select: ["id", "avatar", "email", "name", "lastLogin"],
28         where: { lastLogin: 'is not null' }
29       })
30     } catch (err) {
31       throw new HttpException(err.message, err.status);
32     }
33   }
34
35   async findOne(id: number) {
36     try {
37       return await this.userRepo.findOne(id);
38     } catch (err) {
39       throw new HttpException(err.message, err.status);
40     }
41   }
```

fonte: autor

O método *find,* exemplificado na figura anterior na linha 26, funciona como um *select*. Nessa função é possível passar um objeto de configuração por parâmetro contendo as condições de uma *query* SQL. Neste exemplo são buscados no banco todos os usuários em que a propriedade “lastLogin” não é nula. Na linha 37, é mostrado o *findOne*, que seleciona um usuário com base no seu id.

Outras duas funcionalidades importantes são o *create* e o *save*, como mostrado nas linhas *16* e *18* respectivamente. Nesse exemplo, é possível ver o

objeto DTO interagindo com o ORM ao receber o objeto pelo *controller*. A função *create* é usada para criar um objeto em conformidade com a entidade, o qual a representa na tabela do banco de dados. Após criar o objeto é possível editar seus atributos antes de salvar e, em sequência, com o uso do método *save,* irá executar a inserção desse objeto no banco de dados. Caso algum erro ocorra durante esse processo, o bloco *try catch* consegue identificar o erro e retorna-lo como uma exceção *HttpException* para o cliente, contendo o status e mensagem referente ao erro.

3.3.6 Implementando a Autenticação JWT

A autenticação é uma parte essencial da maioria das aplicações, principalmente em sistemas *Web* com comunicação HTTP. Existem diversas abordagens para a autenticação, nesse projeto o método selecionado foi a autenticação via token JWT. Basicamente essa autenticação processa uma chave de acesso ao receber uma requisição para validar a autenticidade desta chave.

O recurso utilizado na autenticação é o Passport, uma biblioteca bastante usada em aplicações NodeJS e facilmente integrável ao *framework* NestJS. Essa biblioteca torna a aplicação capaz de autenticar um usuário utilizando processos de verificação de credencial e permitindo o estado de autenticação ser utilizado pela aplicação nas diversas rotas (*controllers*). O estado de autenticação contém informações sobre o usuário da requisição, essas informações estão no *payload* da *token*.

O primeiro passo definido na implementação da autenticação é permitir o usuário requisitar a JWT. Para isso é necessário um identificador de usuário e senha, campos da entidade usuário. Para a implementação da autenticação, um módulo novo é criado utilizando o comando “nest g module auth” e “nest g service auth”. O serviço “auth” é responsável pela lógica de autenticação.

Para a requisição da JWT é criada uma rota de *login*. Essa rota usa o método *Post* contendo em seu corpo o email do usuário e senha. Essas informações estão contidas na entidade usuário, e para buscá-las foi usada uma função no serviço do usuário que retorna a senha referente ao email passado na requisição de *login*.

Após coletar a *string,* o método implementado no serviço da autenticação faz a verificação da senha, que consiste em comparar a string criptografada com a *string* enviada pelo usuário no momento da requisição. Para isso é utilizada uma funcionalidade do Bcrypt. Caso a comparação tenha êxito, a criação da JWT para o usuário é liberada.

A criação da JWT é feita pelo serviço de autenticação, que consiste em uma *string* compactada, codificada em *base64* e separada em três partes delimitadas por pontos. Portanto uma JWT tipicamente terá a estrutura “xxxxx.yyyyy.zzzzz”.

A primeira parte dessa estrutura representa o *header,* que contém a informação do algoritmo usado para autenticação e o tipo de *token*. A segunda parte da *token* contém o *payload*, parte responsável por armazenar os dados de quem faz a requisição, que nesse caso é o usuário. O *payload* carrega as informações: “id” do usuário, nome do usuário, “id” do cargo do usuário (explicado no capítulo 3.3.7) e a data de emissão, em conjunto com a data de expiração da validade de uso dessa *token*. A última parte da JWT contém o *secret*, utilizado para validar a autenticidade da *token*.

O *secret* é uma *string* criptografada gerada a partir de uma string definida pelo sistema, essa *string* definida é a chave de decriptação do *secret*. A checagem é feita pela funcionalidade *Strategy*, da classe “PassportStrategy”, parte da biblioteca Passport. A funcionalidade *Strategy* torna possível passar as configurações desejadas ao validar a autenticidade da JWT enviada pela requisição.

Após a implementação da criação e validação da token JWT, é criado um *Guard* para ser implantado em todos os controllers. O *Guard* é um arquivo da estrutura NestJS o qual permite a execução de uma função validadora antes de acessar algum recurso. Essa função é a *validate* da *Token* JWT, ou seja, esse *Guard* define se uma requisição deve ser aceita ou não com base na *Token* recebida pelo usuário. O exemplo de uso desse *Guard* é mostrado na Figura 4 na linha 8.

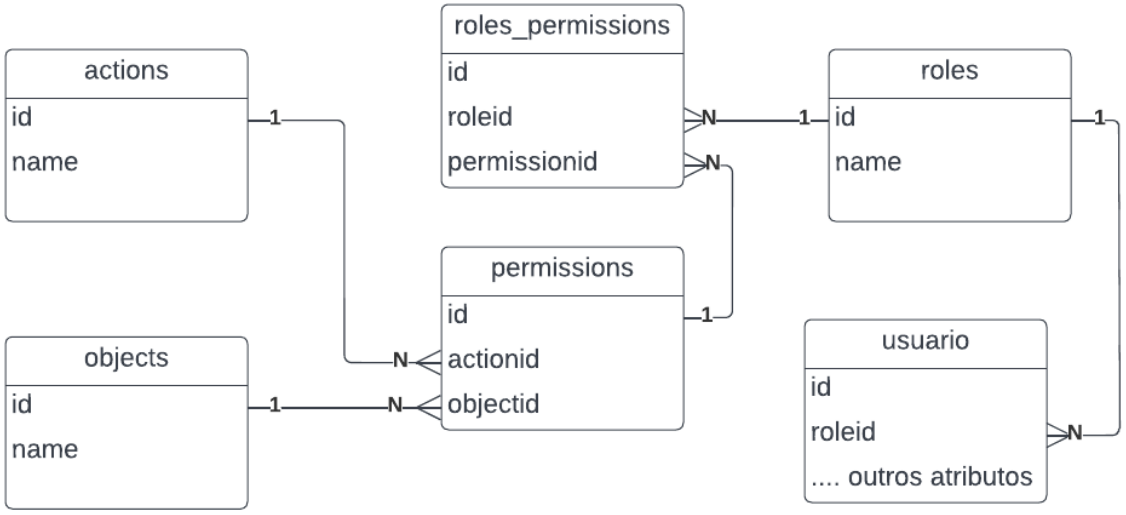
3.3.7 Sistema de permissões

O sistema de permissões foi implementado com o auxílio da biblioteca CASL, um módulo NPM (Node *Package Manager*) que implementa funções para facilitar a

checagem de permissões de usuários.

Esse sistema consiste em grupo de permissões e cargos. A permissão tem dois elementos principais: *action,* para definir as ações (operações CRUD), e *object,* para definir as entidades em que as ações terão efeito. As permissões são ligadas aos cargos considerando uma relação de muito para muitos, ou seja, um cargo pode ter N permissões e uma permissão pode ser atribuída para N cargos. O *design* das tabelas no banco de dados é representado na Figura 6.

Figura 6: *Design* do sistema de permissões.



fonte: autor

Na figura anterior, os cargos e permissões são representados por *roles* e *permissions* respectivamente. Ao atribuir um cargo para o usuário, a entidade recebe uma chave estrangeira da tabela *roles*. Os cargos são formados pelas permissões, essa junção se concretiza na tabela *roles_permissions.*

As ações padrões definidas foram *create*, usada para proteger as rotas que utilizam o método *Post*, *read* usada nas rotas *Get*, *update* usada nas rotas de *Put* e *Patch*, *delete* usadas para as rotas responsáveis pela deleção dados e *manage* usada para representar uma junção de todas as ações anteriores. A implantação da validação de permissões é representada na Figura 7.

Figura 7: Função de validação de permissões

```
1  export class AbilityFactory {
2      constructor(
3          private utils: UtilsService
4      ) {}
5
6      async defineAbility(user : any){
7          const allPerms = await this.utils.getAllPermissionsByRole(user._role);
8          const {can, cannot, build} = new AbilityBuilder(Ability as AbilityClass<AppAbility>);
9
10         if(allPerms.length > 0){
11             await allPerms.map((el:ExpectedRules)=>{
12                 can(el.action,el.subject);
13             })
14         } else {
15             cannot(Action.Manage,'all');
16         }
17
18         return build({
19             detectSubjectType: (item) =>
20                 item.constructor as ExtractSubjectType<Subjects>
21         })
22     }
23 }
```

Fonte: autor

Ao usar o cargo do usuário, na linha 7 do código da figura anterior, são extraídas do banco de dados as permissões e suas respectivas ações usando a *query* representada na Figura 8.

Figura 8: *Query* para extração das permissões.

```
1  SELECT
2      roles."name" as "role",
3      a."name" as "action",
4      o."name" as "subject"
5  FROM
6      roles
7      join roles_permissions rp on rp.roleid = roles.id
8      join permissions p on p.id = rp.permissionid
9      join actions a on a.id = p.actionid
10     join objects o on o.id = p.objectid
11 WHERE
12     roles.id = ${id}
13
```

Fonte: autor

Na figura anterior, é demonstrado o uso das tabelas apresentadas

anteriormente para montar as permissões esperadas na função representada na Figura 7.

Após essa extração com o auxílio das funcionalidades do *AbilityBuilder* da biblioteca CASL, as permissões são mapeadas utilizando a função *can* para validar uma ação para um objeto. Caso esse cargo por algum motivo não tenha permissões, é definido o *cannot* para todas as permissões, demonstradas respectivamente nas linhas 12 e 15 da Figura 7. O *object all* funciona como o *manage* para as ações, ou seja, representa uma junção de todos os *objects*.

Para aplicar essas proteções nas rotas é usado um *decorator* denominado *CheckAbilities,* com o intuito de configurar qual permissão é necessária para cada rota com a definição da ação e sujeito. Esse *decorator* é demonstrado nas linhas 13,21,27,33 e 42 da Figura 4.

Com essa proteção, as rotas do *controller* tornam-se protegidas contra usuários sem permissões específicas, efetivamente permitindo melhor controle de acesso aos dados da aplicação.

3.3.8 Próximos passos

Testes básicos foram feitos utilizando o Postman, um software que permite testes em servidores por meio de requisições HTTP. Após os testes, a fase do módulo base foi cumprida e a aplicação foi considerada apta a ser integrada com um *Front-end* e também a receber módulos específicos para algum tipo de uso.

A API Restful desenvolvida é amigável a qualquer tipo de plataforma que se comunica pelo protocolo HTTP, pelo uso da arquitetura REST. Com essa aplicação implementada, permite a equipe de desenvolvedores a criar novas aplicações derivadas e também cria um padrão de desenvolvimento usando o NestJS como *framework*.

4 CONSIDERAÇÕES FINAIS

Ao ter contato com o mercado de trabalho na área de desenvolvimento de software e TI, o estagiário adquire conhecimentos além do passado em sala de aula durante o curso de formação e torna possível colocar em prática grande parte do conteúdo do curso de Sistemas de Informação.

4.1 Ganhos Pessoais

As atividades envolvidas no desenvolvimento de soluções BI proporcionaram ao estagiário grande entendimento dos negócios envolvendo a empresa de telecomunicações e também possibilitou aprender e desenvolver estratégias auxiliares na administração e análise do negócio. Tudo isso utilizando uma base de dados alimentada por um software de gestão. Houve também ganho na comunicação interpessoal ao lidar diretamente com integrantes da empresa, pois, durante o desenvolvimento de soluções, o estagiário é exposto a vários tipos de pessoas com diversas experiências de vida e conhecimento agregado.

Os cursos realizados e o desenvolvimento da aplicação MGPro tiveram grande impacto no conhecimento técnico do estagiário que o possibilitou aprender novas tecnologias, desenvolvendo um acervo intelectual sobre o ciclo de desenvolvimento WEB, e consequentemente, aumentando sua capacidade de auto-didatismo.

4.2 Ganhos da empresa

Como a empresa baseia boa parte de seu negócio em prestações de serviços, conseguir satisfazer a necessidade do cliente na elaboração e desenvolvimento de soluções BI agrega valor à empresa e aos seus contratantes, oque aumenta a confiança e permite o *networking* ser feito, resultando em mais visibilidade da MGCode no mercado de inteligência de negócios.

O desenvolvimento da API Restful abriu a possibilidade de construções e arquitetura de sistemas de comunicação HTTP mais eficientes. Anteriormente a

MGCode adotava o uso da linguagem PHP 7 para esse fim, portanto ter um produto *template* em outra tecnologia com mais recursos no mercado possibilita a empresa aumentar seu acervo tecnológico e posteriormente adquirir novos projetos envolvendo essa nova tecnologia.

4.3 Desafios e Conclusão

Ao se inserir na empresa, o estagiário se depara com o grande desafio da prestação de serviços BI o qual envolve não só o conhecimento em SQL e banco de dados relacionais como também o conhecimento de negócio e funcionamento da empresa e seus sistemas. Por ter um banco de dados alimentado por um sistema de gestão, responsável pela maioria das operações da Sempre Internet, a grande quantidade de operações nesse sistema resulta em um volume de dados e quantidade de tabelas relativamente grandes. Para ser apto a prestar serviços BI utilizando esse banco de dados, foi necessário o conhecimento do funcionamento e fluxo de dados que a empresa produzia.

Outro grande desafio foi desenvolver uma API com tecnologias novas às quais o estagiário não estava habituado. O NestJS foi um *framework* bem difícil de entender nos primeiros momentos. Como o estagiário não estava habituado em trabalhar com NodeJS, a curva de aprendizado no início foi árdua porém satisfatória a cada superação técnica.

Ao se inserir no mercado de trabalho, é possível concluir que os estudos no curso de Sistemas de Informação garantem uma base significativa de conhecimento. Um exemplo disso foi o contato com banco de dados relacionais, que as matérias “Introdução a Sistemas de Banco de Dados” e “Sistemas Gerenciadores de Banco de Dados” abordam. As matérias envolvendo o estudo da administração de empresas auxiliaram no entendimento e no desenvolvimento de soluções BI, pois essas matérias abordam processos organizacionais e estratégicos nas empresas. As matérias de engenharia e arquitetura de software ofereceram uma boa base nos processos do ciclo de desenvolvimento, como o levantamento de requisitos e os padrões arquiteturais de sistemas. Por ser uma empresa nova considerada *startup,* o estagiário tem a possibilidade de desenvolver e contribuir com o amadurecimento

de processos da empresa ao ter uma noção do estágio de maturidade que a empresa se encontra. Essa noção são contribuições da matéria “Qualidade de Software”, que aborda o uso de normas aplicadas aos processos que envolvem a criação de produtos.

REFERÊNCIAS

AUTH0.COM. JWT.IO - **"JSON" Web Tokens Introduction**. Disponível em: <https://jwt.io/introduction>. Acesso em: 29 de Junho de 2022.

Documentation | **NestJS - A progressive Node.js framework**. Disponível em: <https://docs.NestJS.com/>. Acesso em: 29 de Junho de 2022.

LIMA. **REST: Conceito e fundamentos**. Disponível em: <https://www.alura.com.br/artigos/rest-conceito-e-fundamentos>. Acesso em: 29 de Junho de 2022.

SBORO. **Nest.js — Architectural Pattern, Controllers, Providers, and Modules**. Disponível em: <https://medium.com/geekculture/nest-js-architectural-pattern-controllers-providers-and-modules-406d9b192a3a>. Acesso em: 29 de Junho de 2022.