UNITAU - Universidade de Taubaté

Pedro Ivo de Faria Ramos

Protótipo de Projeto com base na Metodologia da Matriz GUT: Noelle.

Taubaté

2020

Pedro Ivo de Faria Ramos

Protótipo de Projeto com base na Metodologia da Matriz GUT: Noelle.

Trabalho apresentado para obtenção do Certificado de Graduação pelo curso de Engenharia da Computação do Departamento de Informática da Universidade de Taubaté, Área de Concentração: Gestão e organização em tomada de decisão. Orientador: José Alberto Fernandes Ferreira

Taubaté – SP

2020

Grupo Especial de Tratamento da Informação - GETI
Sistema Integrado de Bibliotecas – SIBi
Universidade de Taubaté - Unitau

R175p Ramos, Pedro Ivo de Faria
Protótipo de projeto com base na Metodologia da Matriz Gut : Noelle / Pedro Ivo de Faria Ramos. -- 2020.
45 f. : il.

Monografia (graduação) – Universidade de Taubaté, Departamento de Informática, 2020.
Orientação: Prof. Dr. José Alberto Fernando Ferreira, Departamento de Informática.

1. Noelle. 2. Matriz Gut. 3. Ferramentas de gestão. I. Universidade de Taubaté. Departamento de Informática. Graduação em Engenharia de Computação. II. Título.

CDD – 658.404

Ficha catalográfica elaborada pela Bibliotecária Luciene Lopes - CRB 8/5275

Pedro Ivo de Faria Ramos

Protótipo de Projeto com base na Metodologia da Matriz GUT: Noelle.

Trabalho apresentado para obtenção do Certificado de Graduação pelo curso de Engenharia da Computação do Departamento de Informática da Universidade de Taubaté, Área de Concentração:Gestão e organização em tomada de Decisão.

Data:_____

Resultado: _____

BANCA EXAMINADORA

Universidade de Taubaté

Prof. Dr. José Alberto Fernandes Ferreira _____

Assinatura _____

Prof. Mestre Dawilmar Guimarães de Araujo _____

Assinatura _____

Prof. Especialista Edgar Israel _____

Assinatura _____

Dedico à minha mãe e irmã que sempre me fizeram seguir em frente, e aos meus amigos que nunca me abandonaram.

AGRADECIMENTOS

Ao Prof. Dr. José Alberto Fernandes Ferreira, que me orientou durante o desenvolvimento do projeto de graduação e me apoiou sempre durante minha formação durante todos estes 5 anos. A todos os professores com quem tive o prazer de ter aulas, e aprender cada vez mais sobre a área e à mim mesmo. E a Universidade de Taubaté, por ser uma segunda casa durante esse período de minha vida, que carregarei comigo por toda vida.

Protótipo de Projeto com base na Metodologia da Matriz GUT: Noelle

Pedro Ivo de Faria Ramos

Resumo

Metodologias de priorização foram criadas com o intuito de auxiliar times de planejamento entre outros, em sua priorização de atividades para ajuda na tomada de decisões. A GUT foi selecionada por ser uma metodologia mais simples, e nunca antes usada por nenhum tipo de portal online, e as ferramentas disponíveis com uso dessa metodologia, e com isso o Noelle se destaca por trazer inovação, apesar de simples, sendo único. Com um design maduro, usando tecnologias de ponta e com isso tendo um protótipo grande e poderoso.

Palavras chave: Matriz GUT, Noelle, ferramenta de priorização.

Abstract

Prioritization methodologies were created in order to assist planning teams, among others, in their prioritization of activities to aid decision making. GUT was selected for being a simpler methodology, and never used before by any type of online portal, and the tools available using this methodology, and with that Noelle stands out for bringing innovation, despite being simple, being unique. With a mature design, using cutting-edge technologies and having a large and powerful prototype.

Keywords: GUT matrix, Noelle, prioritization tool.

Lista de Ilustrações

Sumário

1 - Introdução:

O protótipo que este trabalho visa como processo de criação, se trata de uma ferramenta web criada para implementação das metodologias presentes na ferramenta de priorização **matriz GUT**, o projeto visa implementar usando as mais diversas e atuais ferramentas disponíveis, um portal web, acessível, de interface amigável, e com acessibilidade, para que usuários de diversas áreas possam usar dessa ferramenta para organização e priorização de suas tarefas, tanto do dia-a-dia como profissionais.

A partir da vivência em times de planejamento foi analisado como uma ferramenta com uso de metodologia em questão auxiliaria em muito times no momento de criar suas *sprints* e organização de priorização das tarefas a serem realizadas durante o trabalho. Com isso, surgiu a idéia deste protótipo, onde de maneira fácil e gratuíta times inteiros, ou pessoas em seus lazeres podem usufruir da metologia GUT para terem uma ajuda em suas tomadas de decisão.

1.1 - Objetivo:

O objetivo deste projeto é trazer a tona a facilidade e produtividade com que a metodologia de Matriz GUT pode trazer para encontrar a melhor maneira de se iniciar a resolução de problemas com base no que a metodologia diz. Para isso foi realizado um estudo da melhor aplicação da matriz GUT, e contatou-se que a mesma é muito usada em planilhas, e disponibilizada de maneira privada (paga) para usuários pela internet.

A idéia do projeto, é criar um portal web, onde a pessoa possa usufruir desta matriz e de todo o poder e facilidade de clarear caminhos sombrios para resolução de problemas, ajudando usuários com problemas organizacionais, de maneira fácil, prática, com user experience bom, e além de tudo gratuita. Este projeto consiste então, em um portal web com intuito de ajudar aquelas pessoas que querem algum tipo de direcionamento para resolução de problemas, tornando o aplicativo desenvolvido como uma segunda mão não só de usuários comuns, como também de pessoas de área de planejamento.

O portal servirá como guia para aqueles que desejam melhorar seu desempenho em trabalhos, sejam eles corporativos, planejamento de atividades rotineiras, entre qualquer tipo de atividade em que o usuário queira se "planejar", criando uma poderosa ferramenta que proporcione auxílio, organização, solução de problemas e tomada de decisão. O protótipo conta com tecnologias atuais para criar o melhor do *user experience*, além de ser feito para ser rápido, simples, mas muito poderoso.

Sendo assim, o protótipo traz um portal web acessível de maneira gratuita para os usuários de internet que queiram usar do poder e auxílio da análise da matriz GUT.

1.2 - A metodologia da matriz GUT:

A matriz GUT, é uma ferramenta de priorização e ajuda de tomada de decisão criada por Charles Kepner e Benjamin Tregoe nos anos 80, com intuito de priorizar soluções de problemas complexos nas indústrias americanas e japonesas. A ferramenta e sua metodologia criaram fama e passaram a ser usadas em mais setores, como financeiro, e podendo ser adaptada também para áreas do dia-dia, como um método de organização e priorização de atividades a serem exercidas pelo usuário.

Esta metodologia se divide em três pontos principais, medindo seus valores a partir de notas dadas pela pessoa que está usufruindo da ferramenta, atribuindo valores de 1 à 5 para os três pontos e por meio da multiplicação dos valores dados, é feito então a organização da priorização de quais situações devem ser consideradas pelo usuário como mais criteriosas e a serem resolvidas antes das demais. Os 3 pontos principais adotados pela metodologia da **matriz GUT** são a Gravidade, Urgência e Tendência de uma determinada situação problema. Cada ponto a ser analisado tem uma situação de existência e a ser analisado, antes de ser avaliado, sendo eles:

A Gravidade: é o critério que mede o impacto da situação problema. Levando em consideração o possível impacto que a situação pode ter caso não seja resolvida

logo, ao analisar a gravidade deve-se pensar quais efeitos a não realização da determinada tarefa poderá causar ao longo do tempo. Como por exemplo, uma contenção de vazamento em uma loja, ou em um restaurante, é uma situação com uma gravidade muito maior do que o lançamento de um produto, afinal, não executar a tarefa de conter tal vazamento pode gerar danos conforme o tempo passa que são irremediáveis, podendo assim afetar a tarefa de um lançamento de um novo problema. Esta situação de vazamento, com certeza, é uma situação que levaria nota 5 em **Gravidade** de acordo com o que a Matriz GUT diz.

Os níveis de Gravidade são: 1 - Sem Gravidade; 2 - Pouco Grave; 3 - Grave; 4 - Muito Grave; 5 - Extremamente Grave.

A partir destes níveis citados acima o usuário define o valor a ser dado a sua situação, conforme sua gravidade for avaliada.

A Urgência: é o critério de avaliação que mede o tempo, ele leva em consideração o prazo disponível para realizar o projeto. Quanto menor for o prazo, maior é a urgência, pois o mesmo precisa ser realizado com mais agilidade (e vice-versa). Então, quando se analisa a urgência, se analisa quanto tempo essa situação pode esperar para ser realizado. Como no exemplo citado acima, o projeto de conter o vazamento de um restaurante com certeza terá um menor tempo do que a de lançamento de um novo produto.

Os níveis de Urgência são medidos da seguinte maneira, e seus valores atribuídos: 1 - sem urgência; 2 - Pouco urgente; 3 - Urgente; 4 - Muito Urgente; 5 - Extremamente Urgente.

A Tendência: o último critério da matriz GUT é o critério de tendência que leva em consideração a predisposição de um problema piorar conforme o tempo avança. Esse critério existe porque um problema pode nascer pequeno, e com o passar do tempo se tornar algo maior e maior, como uma bola de neve. Quando se cria esta situação você precisa se perguntar: se não sou capaz de resolver este problema hoje, com qual intensidade ele vai piorar? Como nos exemplos já citados, caso o vazamento não fosse contido de maneira mais rápida possível a tendência

dele crescer é bem grande, diferente da situação de lançamento de um novo produto.

Os níveis de tendência são analisados por meio dos seguintes pontos: 1 - Sem tendência de piorar; 2 - Piorar em longo prazo; 3 - Piorar em médio prazo; 4 - Piorar em curto prazo; 5 - Agravar rápido.

Para criação de uma projeto de organização e priorização com ajuda da metodologia da Matriz GUT é recomendado que a pessoa primeiramente liste os projetos, ou situações, que precisa gerência, primeiramente então se faz a relação das suas situações a serem priorizadas, e que se precisa entregar. Em seguida, defina notas para os critérios da GUT, avaliando de 1 à 5 para seus valores.

Ao avaliar os dados, multiplica-se os valores entre Gravidade x Urgência x Tendência, e assim aquelas situações que tiverem maiores valores são as que precisam de mais atenção no devido momento, pois tendem a crescer e se tornar mais problemáticas de acordo com o que diz as prioridades de análise da matriz GUT. E desta maneira com a matriz montada e seus devidos valores atribuídos e multiplicados, a pessoa que está fazendo uso da mesma poderá definir um valor esperado, e com as notas mais altas ajudando o usuário na sua priorização.

Um bom exemplo de onde pode-se ser usado a matriz GUT é no caso de uma empresa de importação de vinhos e representação de alimentos, onde o responsável quer usar a Matriz GUT para priorização de suas situações e assim, decidir o que fazer primeiro, e consequentemente a última tarefa do mesmo. Se na segunda e terça feira, o usuário quer contatar todos os clientes de representações e pegar os pedidos, na quarta pegar pedidos de vinhos e solicitar nova remessa direto do Chile, na quinta-feira o representante quer liberar carretas com alimentos para atacados, sendo os atacados seus clientes. E sexta-feira gerar notas fiscais dos vinhos e enviar o pessoal para entrega e agendar apresentações.

De certa forma na situação apresentada acima o usuário já definiu suas prioridades, mas o mesmo possui certa dificuldade ou dúvida sobre quais prioridades poderiam ser adiantadas devido sua Gravidade, Tendência de

crescimento ou Urgência de ser feito, sendo assim, as datas que o mesmo definiu não equivalem de muita ajuda, se for levar em análise os pontos. Logo, com a ajuda de uma MAtriz GUT, essas datas que o usuário chegou a definir viriam a mudar, conforme suas prioridades mudam, devido a situações mais urgentes que outras. É nisso que a metodologia GUT veio para ajudar, na situação problema onde se precisa analisar melhor os critérios situacionais dos problemas da situação presente.

1.3 - O que foi usado durante o projeto:

Para desenvolvimento do projeto foi-se analisado as melhores tecnologias disponíveis no mercado, e que se tratem de tecnologias open-source, ou seja, de código aberto, para poder ser um portal onde pessoas possam contribuir com o mesmo de maneira íntegra, para melhoria dele e crescimento. Melhorando a experiência das pessoas que o usam para motivos privados, como motivos profissionais.

Então, as tecnologias selecionadas, antes fazendo observações rápidas sobre termos que serão utilizados a seguir, e o leitor que pode não conhecer os termos fique inteirado do assunto. Para isso uma breve explicação sobre *front-end* e *back-end*. O *back-end,* no desenvolvimento web em si, se trata da parte do servidor de uma aplicação, a parte, assim como o próprio nome diz, de trás do aplicativo, (o back do mesmo), é nele onde fica a lógica do seu programa, sobre como ele irá funcionar, o que ele irá fazer, a ligação com banco-de-dados, a métodos pesados e mais complexos, esta parte da aplicação fica armazenada em computadores de empresas como *amazon* ou *google*, assim deixando o processo do que será rodado pelo *browser* do usuário, mais dinâmico e acessível, deixando a aplicação mais ágil.

Já o *front-end* se trata da parte visual da aplicação, e de funcionalidades que envolvem o user experience, nos tempos atuais, possuímos sim tecnologias que fazem validações e devemos fazê-las no front-end, mas é visto como uma má prática de muitos desenvolvedores a aplicação de métodos muito pesados em seu *front-end* e sempre devemos lembrar que possivelmente a máquina do usuário, ou até mesmo seu provedor de internet não sejam os melhores, podendo gerar dor de cabeça para aquele que irá usufruir da sua aplicação.

Para isso foi analisado as melhores tecnologias disponíveis para desenvolver um portal atualizado, com tecnologias de ponta, novas, e com futuro promissor, além de um bom suporte, para que o desenvolvimento seja feito na melhor qualidade possível. Foi-se então que chegou nas seguintes conclusões:

Back-end: será usado o NestJS, essa tecnologia é dita em seu próprio portal como uma framework em NodeJS para desenvolvimento *back-end* e padronização de código, tornando a estrutura de todo seu *back* mais legível, organizada e padronizada, por meio de padrões muito semelhantes ao MVC (Model, View, Controller), o NestJS se destaca pela liberdade que dá ao desenvolvedor e sua facilidade em criar rotas para acesso dos dados gerados no *back-end* por aplicações externas, sejam ela o *front-end* do portal, ou até mesmo num futuro uma aplicação *mobile*. Também será usado o TypeORM (*ORM - Object Relational Mapping*), que se trata de um conceito onde ele se cria relações entre os objetos criados no seu código e passa-os para o banco-de-dados de maneira relacional, evitando com que o desenvolvedor tenha de agir diretamente no banco. No caso, será usado o **Type**ORM, pois a linguagem usada no desenvolvimento da aplicação é o **Typescript**, tanto para no *front-end* como para o *back-end.*

Typescript: ou TS, assim conhecido pois seus arquivos possuem o final .ts, se trata de um *superset* do já conhecido JavaScript, trazendo como novidade a possibilidade de criação de interfaces, tipagem de objetos, e uma orientação a objeto muito mais dinâmica, e legível para aquele que olha para o código.

Front-end: aqui foi escolhido a biblioteca JavaScript **ReactJs**, hoje conhecida somente como **React**, que se trata de uma biblioteca para a linguagem com enfoque em browsers, o React veio para facilitar na criação de páginas web SPA (single-page application) aquelas páginas que não precisam de reload a todo momento, assim não criando sites pesados. O React é uma biblioteca desenvolvida e disponibilizada pelo Facebook, que ao criá-la e notar sua potência, tornou-o open-source para que os desenvolvedores da comunidade pudessem cooperar para o seu crescimento, o que vem ocorrendo desde então. O React trouxe a inovação para a criação de

páginas dinâmicas, sendo uma grande tecnologia do mercado atual de tecnologia da informação.

Para todos os tópicos acima, foi-se usado o **NodeJS**, que se trata de um gerenciador de pacotes para linguagem JavaScript, com suporte para TS, a partir do NodeJS, foi possível o uso de JavaScript para criação de tecnologias back-end, trazendo uma linguagem antes interpretada somente para browsers, para trás das cortinas, criando assim, grandes sites e um universo online cada vez mais potentes. O NodeJS se mantém na frente, e foi o pioneiro e grande criador de muitas qualidades de suporte e crescimento para a carreira de muitos desenvolvedores, hoje o mesmo possui alguns concorrentes, mas nenhum ainda chega a ameaçá-lo. Já que o gerenciador de pacotes é utilizado não somente por pequenas, como grandes empresas, como o próprio Google, entre outras.

E para o armazenamento dos dados foi utilizado o famoso SQL, com o uso do banco-de-dados PostgreSQL, que se trata de um sistema gerenciador de banco de dados objeto relacional, desenvolvido como projeto de código aberto, ou seja, totalmente gratuíto.

E para instalação de todos os pacotes e uso do NodeJS como um administrador de pacotes, foi usado o **Yarn**, o NodeJS já vem com um comando de terminal para instalação de pacotes externos em seu projeto para auxílio (como o próprio nestJS, entre outros), o então conhecido npm, mas para desenvolvimento do protótipo do projeto atual, foi utilizado o yarn, por ser um gerenciador mais rápido, e o mesmo guarda um histórico evitando que seu projeto crie um bundle muito grande de dados e bibliotecas, mantendo o mesmo em um tamanho mais conciso e coeso.

2 - Fundamentação Teórica:

A metodologia GUT foi a escolhida para desenvolvimento deste trabalho de graduação por não possuir nenhum outro portal web, ou aplicativo mobile disponível que use a mesma como fonte mãe de desenvolvimento lógico na base de critérios

de priorização. Por este motivo o GUT, apesar de ser o mais simples método de Priorização, se torna um método ótimo, para um trabalho de graduação.

Existem diversas outras metodologias de análise de situações problema, para priorização de atividades, tais como Matriz de Eisenhower, Matriz Saaty, Matriz B.A.S.I.C.O e o Princípio de Pareto. Todas muito boas e usuais, mas com um nível de dificuldade acima do GUT, e sem a simplicidade do GUT. As matrizes, citadas, como a Saaty, analisam mais dados, além dos três picos do GUT como também pesos e outras análises, tornando a aplicação muito mais complexo e assim dificultando sua aplicação, e dado ao tempo e idéia do protótipo acabam não sendo metodologias cabíveis.

3 - Desenvolvimento do Projeto:

Para início do desenvolvimento do Protótipo, foi-se definido o nome **Noelle** para a plataforma, uma palavra calma e amigável para gerar atração do usuário pelo protótipo e assim atrair um público.

3.1 - Back-end:

Todo backend da aplicação foi feita em NestJS, uma framework para desenvolvimento de aplicações REST API, o NestJS vem ganhando um grande espaço no mercado por sua simplicidade, facilidade de desenvolvimento, por ser uma framework bem estruturada, e por seu suporte competente e documentação recebendo atualizações frequentes, para o desenvolvimento da aplicação foi usado o CLI do NestJS, disponível em seu portal para baixar por meio de *yarn* ou *npm*, com a CLI do projeto se torna mais ágil o ato de iniciar um projeto em NestJS, vindo com pré-configurações prontas, e o mesmo também já contém um grande suporte para TypeScript, o superset de JavaScript usado para o desenvolvimento do protótipo.

Após a inicialização do projeto o primeiro passo foi a criação e validação dos endpoints para criação de usuário e autenticação de login.

3.1.1 Usuário:

A Classe/Documento responsável pelo armazenamento dos dados de cada usuário no protótipo do sistema Noelle, nesta classe ficam armazenados os dados do usuário sendo eles seu username e password as principais fontes necessárias para um login, entretanto na definição e construção da classe foram definidos mais alguns campos para seguirem regras de autenticação, ligação com as demais classes do sistema, e identificador único.

A classe de usuários fica dentro de uma pasta chamada *auth*, nela ficaram todas as classes, objetos, e arquivos responsáveis pela autenticação, criação, exclusão, edição e o que mais for necessário para um usuário. A imagem a seguir mostra como foi montada a classe de usuário do sistema Noelle:

Imagem 1 - Classe de Usuários.

```
import { BaseEntity, Column, Entity, OneToMany, PrimaryGeneratedColumn } from 'typeorm'
import * as bcrypt from 'bcrypt'
import { Problems } from '../problems/problems.entity'

@Entity()
export class User extends BaseEntity {

  @PrimaryGeneratedColumn()
  id: string

  @Column()
  username: string

  @Column()
  password: string

  @Column()
  salt: string

  @OneToMany(type => Problems, problems => problems.user, { eager: true })
  problems: Problems[]

  async validadePassword( password: string ): Promise<boolean> {
    const hash = await bcrypt.hash(password, this.salt)
    return hash === this.password
  }
}
```

fonte: (própria 2020).

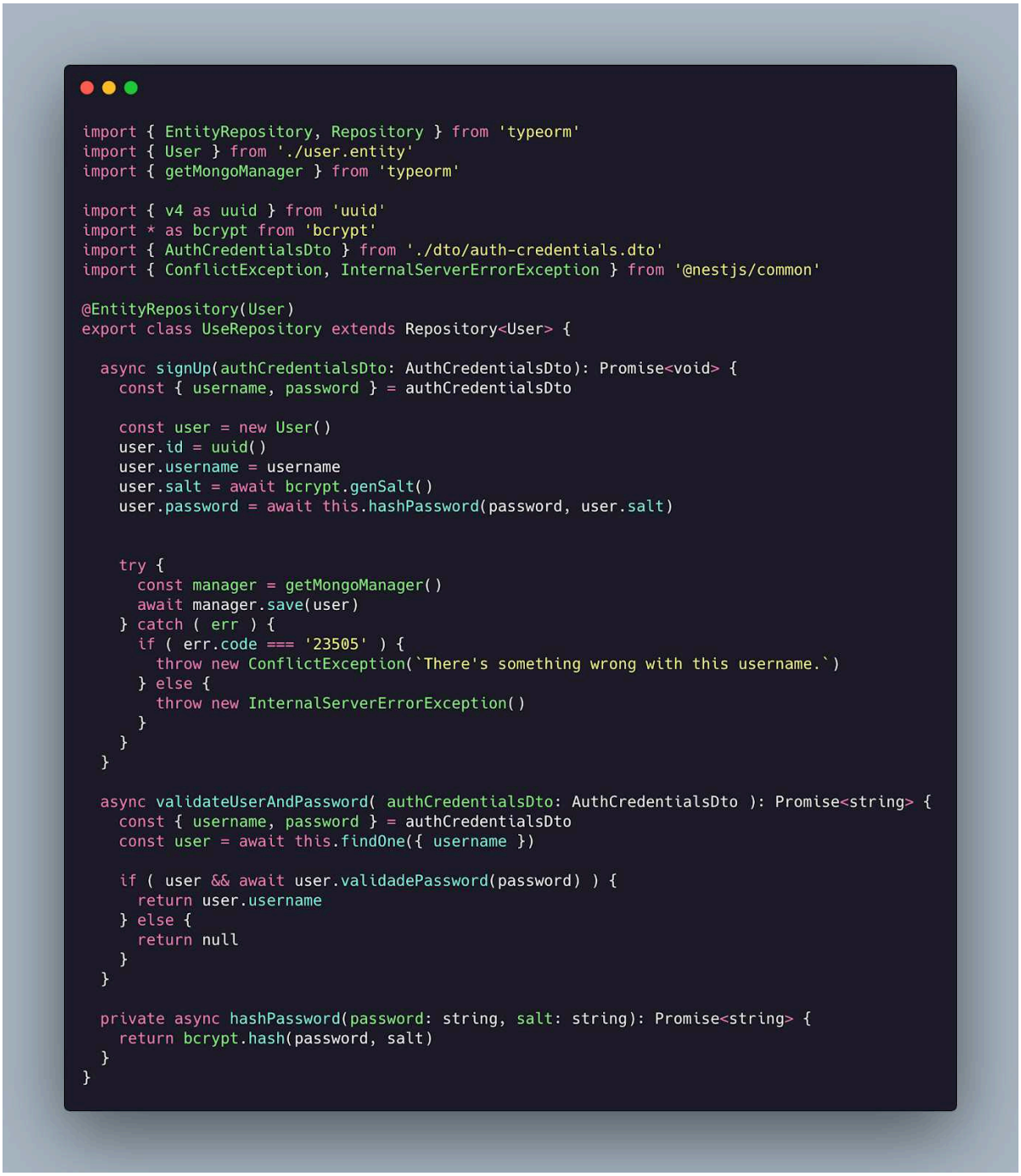
O valor "problem" com a anotação de ManyToOne, é uma prática de quando se usa TypeORM, onde se define uma variável de ligação desta entidade com a possível outra que ela terá relação, no caso teremos diversos problemas para um usuário, no caso um set de problemas, por isso foi criado como um Array de *Problems*, que se trata da entidade de problemas.

Já na **Imagem 2** será demonstrado onde o salt será usado e porque. E o método *validadePassword* é definido para validação do *password* digitado pelo usuário na hora de realizar o login na aplicação, se trata de uma classe responsável pela validação.

Acima temos as importações do TypeORM, biblioteca utilizada para criação de relações das entidades assim, criando classes que serão refletidas no banco de dados com precisão, seguindo as ordens de Colunas usando a *annotation* de ***Column*** para chave-primária usamos as *annotations* ***PrimaryGeneratedColumn***. A importação da biblioteca bcrypt e seu uso no método de validação do password será explicado mais tarde no decorrer do projeto.

A imagem a seguir é a representação da classe de *user.repository.ts,* que se trata da classe do sistema onde ficam armazenadas as lógicas de cadastro e autenticação do usuário digitado no sistema. O processo de lógica por trás de toda aplicação que usa NestJS vem do Controller e Service, e no caso da classe User, o Repository que é a classe responsável por salvar os dados no banco, o *repository* é onde todas as transações entre aplicação e banco de dados ficam armazenadas. Então, a seguir, mostrada a classe de repositório, responsável pela interação com o banco de dados diretamente:

Imagem 2 - Classe Repository.



```
import { EntityRepository, Repository } from 'typeorm'
import { User } from './user.entity'
import { getMongoManager } from 'typeorm'

import { v4 as uuid } from 'uuid'
import * as bcrypt from 'bcrypt'
import { AuthCredentialsDto } from './dto/auth-credentials.dto'
import { ConflictException, InternalServerErrorException } from '@nestjs/common'

@EntityRepository(User)
export class UseRepository extends Repository<User> {

  async signUp(authCredentialsDto: AuthCredentialsDto): Promise<void> {
    const { username, password } = authCredentialsDto

    const user = new User()
    user.id = uuid()
    user.username = username
    user.salt = await bcrypt.genSalt()
    user.password = await this.hashPassword(password, user.salt)


    try {
      const manager = getMongoManager()
      await manager.save(user)
    } catch ( err ) {
      if ( err.code === '23505' ) {
        throw new ConflictException(`There's something wrong with this username.`)
      } else {
        throw new InternalServerErrorException()
      }
    }
  }

  async validateUserAndPassword( authCredentialsDto: AuthCredentialsDto ): Promise<string> {
    const { username, password } = authCredentialsDto
    const user = await this.findOne({ username })

    if ( user && await user.validadePassword(password) ) {
      return user.username
    } else {
      return null
    }
  }

  private async hashPassword(password: string, salt: string): Promise<string> {
    return bcrypt.hash(password, salt)
  }
}
```

fonte: (própria 2020)

A classe de repository tem sua declaração com a annotation de ***EntityRepository*** pois define que a mesma se trata de um repositório de uma entidade do banco de dados, está annotation é um padrão obrigatório nas classes de repositório quando se usa TypeORM.

Os métodos definidos com a palavra ***async*** em seu início são definidos assim por se tratarem de métodos assíncronos, ou seja, o programa só irá continuar com seu processo quando o método assíncrono for finalizado, seja ele com êxito ou não. A classe de signUp, ou seja, a classe de registrar um usuário recebe como parâmetro uma DTO, data transfer object (objeto de transferência de dados) se trata de um padrão de desenvolvimento de software onde é definido um padrão de objeto que será recebido para transferência de informações. Na imagem a seguir é mostrado o Objeto de Transferência de Dados responsável pelo recebimento dos valores para criação e autenticação de um usuário.

Imagem 3 - DTO para criação e transação de usuários.

```
import { IsString, Matches, MaxLength, MinLength } from 'class-validator'

export class AuthCredentialsDto {
  @IsString()
  @MinLength(4)
  @MaxLength(20)
  username: string

  @IsString()
  @MinLength(6)
  @MaxLength(20)
  @Matches(/((?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$/, {
    message: 'Password too weak.'
  })
  password: string
}
```

fonte: (própria 2020)

O DTO de usuário importa algumas classes importantes da biblioteca *class-validator,* adicionada no projeto por meio do gerenciador de pacotes ***yarn,*** o class-validator trás consigo uma variedade de annotations para validação de um objeto em uma classe, com essas annotations utilizadas na **Imagem 3** verificamos e validamos se o objeto feito por meio de AuthCredentialsDto recebe um username do tipo String, com no mínimo 4 caracteres e máximo de 20, e o mesmo para o

password (senha), mas no mesmo definimos uma expressão regular para que o password se encaixe na mesma, definindo assim um padrão para senha forte, onde seja necessário caractere maiúsculos, minúsculos, especial e numeral.

A função de *signUp* recebe os dados do DTO digitados pelo usuário e assim cria um novo objeto do tipo usuário, assimilando os valores digitados pelo mesmo nos campos correspondentes, com a função de genSalt do bcrypt, é gerado uma sequência de dados encriptados que serão salvos no objeto *salt* para validação de password, e o password acaba sendo o resultado da função *hashPassword* que gera um password encriptado com os dados de salt, tornando a aplicação muito mais segura, e a senha do usuário mais difícil de ser descoberta por hackers.

Enfim, temos a função de validação de password, que é usada para verificação de login do usuário, a mesma procura o usuário no banco a partir do username e verifica se o password digitado no momento do login é equivalente ao presente no banco de dados.

Na imagem a seguir é apresentado o arquivo de controller de usuários. Nele é definido os end-points da api e seus tipos de dados de entrada, em uma request a métodos HTTP usamos métodos Post para enviar dados, assim como Put e Patch, Delete e Get, cada um com suas propriedades e funções, no caso do Controller de autenticação são utilizadas somente métodos Post para envio de dados no formato definido no DTO de autenticação, tanto para criação de de um usuário, quanto para validação de um usuário.

Imagem 4 - Controller de usuários.

```
import { Body, Controller, Post, ValidationPipe } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthCredentialsDto } from './dto/auth-credentials.dto';

@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService) {}

  @Post('/sign-up')
  signUp(@Body(ValidationPipe) authCredentialsDto: AuthCredentialsDto ): Promise<void> {
    return this.authService.signUp(authCredentialsDto)
  }

  @Post('/sign-in')
  signIn(@Body(ValidationPipe) authCredentialsDto: AuthCredentialsDto): Promise<{accessToken: string}>
  {
    return this.authService.signIn(authCredentialsDto)
  }
}
```

fonte: (própria 2020)

Os endpoints de registro de um usuário, como de validação de um usuário são acessíveis por meio dos links com finais "/auth/sign-up" e "/auth/sign-in", no caso da aplicação, como foi feita em ambiente de desenvolvimento são acessados por meio do localhost:3001, sendo 3001 a porta escolhida para que a aplicação acesse na máquina que a mesma esteja rodando.

Já a autenticação do usuário muito parecido com o método para criar, requer que o mesmo envie para o endpoint de "/auth/sign-up" o username e password do mesmo, para que o back-end da aplicação o valida. A validação é enviada para o repository, onde o mesmo procura um usuário com aquele username, se existir, irá enviar o password digitado pelo usuário para a função de validação de password como podemos ver nos processos da **Imagem 2** e a função de validação do password é feito no entity (**Imagem 1**), o mesmo irá criar um hash com o valor digitado pelo usuário, e se o mesmo bater com o password salvo no banco de dados, que é encriptado para proteção do próprio usuário significa que o mesmo está autorizado e digitou as credenciais de maneira correta.

Se o usuário digitou suas credênciais corretas, entra em ação o método de *sign* do ***jwtService*** que irá criar um token a partir do método JWT, para proteção e validação do usuário dentro das rotas.

O JWT (Json Web Token) é um método RCT 7519 padrão das indústrias atuais para autenticação de um usuário, por meio de requisições feitas a todo momento para os endpoints do back-end, o JWT cria um token onde o browser irá enviar a todo momento que precisar fazer uma requisição para o back-end em alguma rota segura, ou seja, uma rota trancada que só é liberada por meio de login feito pelo usuário, assim protegendo seus dados. Sempre que o usuário enviar um token para o backend ele será validado, caso conste no sistema, o resultado esperado será retornado, se não, ele receberá uma resposta HTTP 401, o código referente a usuário não autorizado para determinada rota.

Na **figura 5** fica exposta a classe service de usuários, onde é criado o token e na **figura 6** é mostrado a estratégia do Json Web Token para autenticação do usuário no sistema.

Imagem 5 - Service de usuários.

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt'
import { InjectRepository } from '@nestjs/typeorm'
import { UseRepository } from './user.repository'
import { AuthCredentialsDto } from './dto/auth-credentials.dto';

@Injectable()
export class AuthService {
  constructor(
    @InjectRepository(UseRepository) private userRepository: UseRepository,
    private jwtService: JwtService
  ) {}

  async signUp( authCredentialsDto: AuthCredentialsDto ): Promise<void> {
    return this.userRepository.signUp(authCredentialsDto)
  }

  async signIn( authCredentialsDto: AuthCredentialsDto ): Promise<{accessToken: string}> {
    const username = await this.userRepository.validateUserAndPassword(authCredentialsDto)

    if ( !username ) {
      throw new UnauthorizedException(`Invalid Credentials...`)
    }

    const payload = { username }
    const accessToken = await this.jwtService.sign(payload)

    return { accessToken }
  }

}
```

fonte: (própria 2020)

Após ser encontrado um usuário e validado no banco de dados é enviado para o JWTService o username deste usuário, para criação do token que será enviado para o browser, geralmente este token é armazenado no localStorage do browser, ou em algum local de armazenamento de dados, como redux ou algo do genêro.

Já a classe a seguir define a estratégia usada pelo JWT para achar o usuário e também a secretKey do mesmo, no caso, por se tratar de um protótipo foi usado o nome 'noelleProject', mas, para projetos em produção esta chave-secreta fica localizada em algum arquivo de configuração do aplicativo.

Imagem 6 - Strategy JWT.

```
import { Injectable, UnauthorizedException } from '@nestjs/common'
import { PassportStrategy } from '@nestjs/passport'
import { User } from './user.entity'
import { Strategy, ExtractJwt } from 'passport-jwt'
import { InjectRepository } from '@nestjs/typeorm'
import { UseRepository } from './user.repository';

export interface JwtPayload {
  username: string
}

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    @InjectRepository(UseRepository) private userRepository: UseRepository
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: 'noelleProject'
    })
  }

  async validate( payload: JwtPayload ): Promise<User> {
    const { username } = payload
    const query = this.userRepository.createQueryBuilder('user')
    query.where('user.username =  :usernameFind', { usernameFind: username })

    const userFinded = await query.getOne()

    if( !userFinded ) {
      throw new UnauthorizedException()
    }

    return userFinded
  }
}
```

fonte: (própria 2020)

3.1.2 - Problema:

A entidade de Problema é onde ficam salvos os problemas que a matrizGUT irá armazenar, ou seja, é na mesma em que ficará salvo o problema que possui certa, gravidade, urgência e tendência. A Entidade foi definida da seguinte maneira como descrita na **Imagem 7**:

Imagem 7 - Entidade de problema.

```
import { BaseEntity, Column, Entity, ManyToOne, PrimaryGeneratedColumn } from 'typeorm'
import { Problems } from '../problems/problems.entity';

@Entity()
export class Problem extends BaseEntity {

  @PrimaryGeneratedColumn()
  id: number

  @Column()
  title: string

  @Column()
  description: string

  @Column()
  gravity: number

  @Column()
  urgency: number

  @Column()
  trend: number

  @Column()
  total: number

  @ManyToOne(type => Problems, problem => problem.problems, { eager: false })
  problem: Problems
}
```

fonte: (própria 2020)

A entidade de problem contém seu identificador único automaticamente incrementado pelo banco de dados, os dados de título e descrição do problema, que são definidos para serem do tipo string, ou seja, do tipo de texto, os respectivos valores referentes a matriz GUT, de gravidade, urgência e tendência, definidos como numerais, o total, que se trata da multiplicação dos 3 valores que o usuário irá fornecer e uma variável chamada **problem**, uma nova entidade criada no banco de dados chamada Problems que se tem como intuito agrupar todos os problemas de um determinado "grupo" ou "set" de problemas.

Assim como no usuário, a entidade de Problema possui seu controller com seus endpoints, sendo somente o endpoint do método Post, ou seja, um método HTTP que recebe e armazena algo no banco-de-dados, este endpoint abriga a classe "*createProblem*" responsável por criar o problema no banco de dados e

salvá-lo atrelado ao set de *problemas,* e ao usuário, a **Imagem 8** contém a representação do controller de problema.

Imagem 8 - Controller de problema.

```
import { Body, Controller, Post, UseGuards, ValidationPipe, UsePipes, Param, ParseIntPipe } from
'@nestjs/common';
import { AuthGuard } from '@nestjs/passport'
import { CreateProblemDto } from './dto/create-problem.dto'
import { Problem } from './problem.entity'
import { ProblemService } from './problem.service'

@Controller('problem')
@UseGuards(AuthGuard())
export class ProblemController {
  constructor(private problemService: ProblemService) {}

  @Post()
  @UsePipes(ValidationPipe)
  createProblem(@Body() createProblemDto: CreateProblemDto, @Param('id', ParseIntPipe) id: number):
Promise<Problem> {
    return this.problemService.createProblem(createProblemDto, id)
  }
}
```

fonte: (própria 2020)

O controller de problema também é caracterizado por usar da annotation "*UseGuards"* que se trata de uma anotação para validação do usuário e proteção das rotas que ficam armazenadas neste controller, está anotação também será vista no controller de problemas.

Na **imagem 9** é mostrado o service de problem, nele fica somente a chamada para o service de problema onde o mesmo é armazenado no banco de dados, e antes disto ocorrer é feito um método chamado findOne, que irá procurar o set de problemas do id enviado pelo usuário, e o mesmo será retornado, assim, todos os problemas criados serão salvos no referido set.

Imagem 9 - Service de problema.

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm'
import { Connection } from 'typeorm'
import { ProblemsRepository } from '../problems/problems.repository'
import { CreateProblemDto } from './dto/create-problem.dto'
import { Problem } from './problem.entity'
import { ProblemRepository } from './problem.repository'

@Injectable()
export class ProblemService {
  private problemRepository: ProblemRepository
  private problemsRepository: ProblemsRepository
  constructor( private readonly connection: Connection ) {
    this.problemRepository = this.connection.getCustomRepository(ProblemRepository)
    this.problemsRepository = this.connection.getCustomRepository(ProblemsRepository)
  }

  async createProblem(createProblemDto: CreateProblemDto, id: number): Promise<Problem> {
    const problems = await this.problemsRepository.findOne({ where: { id } })
    return this.problemRepository.createProblem(createProblemDto, problems)
  }
}
```

fonte: (própria 2020).

O repository de problema, assim como todos os outros repository, é onde toda lógica e manipulação de dados diretamente do banco de dados, ou os salvando no banco, é realizado, neste caso a função *createProblem*, irá receber os dados que o usuário digitou e fazer o cálculo de total da matriz GUT, este cálculo é realizado por meio da multiplicação dos valores de Gravidade, Urgência e Tendência.

Na **Imagem 10** é demonstrado como este cálculo foi feito, a partir da função *totalOfProblem* e da criação do problema, e logo em seguida, dentro de um bloco de *try catch* é salvo este dado no banco-de-dados, caso ocorra algum problema, o catch irá capturá-lo e retornar para o usuário um problema interno do servidor.

Imagem 10 - Repository de problema.

```
import { EntityRepository, Repository } from 'typeorm'
import { Problem } from './problem.entity'
import { CreateProblemDto } from './dto/create-problem.dto'
import { User } from '../auth/user.entity'
import { Problems } from '../problems/problems.entity'
import { InternalServerErrorException } from '@nestjs/common'


@EntityRepository(Problem)
export class ProblemRepository extends Repository<Problem> {
  async createProblem(createProblemDto: CreateProblemDto, problemsArray: Problems): Promise<Problem> {
    const { title, description, gravity, urgency, trend } = createProblemDto

    const problem = new Problem()
    problem.title = title
    problem.description = description
    problem.gravity = gravity
    problem.urgency = urgency
    problem.trend = trend
    problem.total = this.totalOfProblem(gravity, urgency, trend)
    problem.problem = problemsArray

    try {
      await problem.save()
    } catch (error) {
      throw new InternalServerErrorException()
    }

    delete problem.problem
    delete problem.id
    return problem
  }

  totalOfProblem(gravity: number, urgency: number, trend: number): number {
    return gravity * urgency * trend
  }
}
```

fonte: (própria 2020)

3.1.3 - Problemas:

Foi criado também a entidade de problemas, inicialmente se trata de um entidade básica, onde o banco salva um título e uma descrição para o que se trata aquele problema, ou aquela situação em que o usuário quer usar da metodologia GUT para organizar seus processos.

Está entidade serve como uma amarração de dados, além de possuir o título e a descrição daquele set de "*problemas"* que o usuário quer resolver, nela também é atrelado o usuário que criou este set, ou seja, é uma das entidades mais importantes, se não a mais importante de todo o sistema. Na **imagem 11** é mostrado o resultado da mesma:

Imagem 11 - Entidade de problemas.

```
import { BaseEntity, Column, Entity, ManyToOne, OneToMany, PrimaryGeneratedColumn } from 'typeorm'
import { User } from '../auth/user.entity'
import { Problem } from '../problem/problem.entity'

@Entity()
export class Problems extends BaseEntity{

  @PrimaryGeneratedColumn()
  id: number

  @Column()
  title: string

  @Column()
  description: string

  @ManyToOne(type => User, user => user.problems, { eager: false })
  user: User

  @OneToMany(type => Problem, problem => problem.problem, { eager: true })
  problems: Problem[]
}
```

fonte: (própria 2020)

Como pode ser visto, a entidade de problemas, além de possuir seu identificador único, título e descrição, possui a amarração com usuário, nela é usado a annotation *ManyToOne*, ou seja, somente um usuário para vários sets de problemas que podem ser criados no sistema. E também um Array de problema, onde aquele set, ou seja, esta entidade, objeto de *problems* pode ter mais de 1 problema, ou mais de 1 objeto a ser resolvido e organizado com ajuda da GUT, e para isso é usado a annotation *OneToMany*.

Está classe possui diversos módulos, além de também usar a anotação de *UseGuards* para proteção das rotas criadas, à **imagem 12** representa o controller da entidade de problemas.

Imagem 12 - Controller de problemas.

```
import { Body, Controller, Post, Get, UseGuards, ValidationPipe, UsePipes, Param, ParseIntPipe } from
'@nestjs/common';
import { AuthGuard } from '@nestjs/passport'
import { GetUser } from '../auth/get-user.decorator'
import { User } from '../auth/user.entity'
import { CreateProblemsDto } from './dto/create-problems.dto'
import { Problems } from './problems.entity'
import { ProblemsService } from './problems.service'

@Controller('problems')
@UseGuards(AuthGuard('jwt'))
export class ProblemsController {
  constructor(private problemsService: ProblemsService) {}

  @Get()
  getProblemListTotal(@GetUser() user: User): Promise<Problems[]> {
    return this.problemsService.getAllProblems(user)
  }

  @Get('/:id')
  getProblemsList(@Param('id', ParseIntPipe) id: number): Promise<Problems> {
    return this.problemsService.getProblemsById(id)
  }

  @Post()
  @UsePipes(ValidationPipe)
  createProblems(@Body() createProblemsDto: CreateProblemsDto, @GetUser() user: User):
Promise<Problems> {
    console.log(createProblemsDto)
    return this.problemsService.createProblems(createProblemsDto, user)
  }
}
```

fonte: (própria 2020)

O controller dos problemas possui 3 endpoints, sendo o controller com mais endpoints de toda aplicação, nele possuímos o controller que pega todos os sets de problemas do usuário, a partir da anotação getUser é possível receber o token recebido pela requisição do browser e assim fazemos uma assimilação com todos os sets de problemas que o usuário que possui aquele token tem, retornando para o browser todos os sets. Existe também ao método Get que recebe um id, assim retornando não só um único set, como todos os problemas atrelado ao mesmo.

O método GET se trata de um tipo de requisição feita pelo browser, as chamadas requisições HTTP, o GET é o método que espera algum tipo de retorno, ele não salva nada no banco-de-dados, ou seja, não é um método com retorno de CREATED ou algo do gênero, sempre retornando algo para o browser possivelmente renderizar.

E por último e método POST, responsável por salvar não só aquele set de problema, como todos os problemas atrelados a ele, na **imagem 13** é mostrado o DTO responsável pela criação e padronização de como serão recebidos os dados pelo endpoint de POST de problemas.

Imagem 13 - DTO de Problemas.

```
import { IsNotEmpty } from 'class-validator'
import { CreateProblemDto } from '../../problem/dto/create-problem.dto'

export class CreateProblemsDto {
  @IsNotEmpty()
  title: string

  @IsNotEmpty()
  description: string

  @IsNotEmpty()
  problems: CreateProblemDto[]
}
```

fonte: (própria 2020)

No DTO é definido que os dados de título, descrição e também os problemas não podem ser vazios, ou seja, é esperado que o sistema receba estes dados fornecidos pelo browser, para serem salvos no banco.

A service de problemas também é uma das maiores do sistema, contando com função que chama o método de criar um set de problemas no banco-de-dados, e duas funções que retornam os problemas, um onde somente retorna o id, título e descrição daquele set, e outra que retorna tanto estes dados como todos os outros dados do set, ou seja, todos os problemas atrelados ao mesmo. Lembrando todos os métodos que retornam dados do banco, retornam somente os dados referentes aquele usuário, ou id. A **figura 14** ilustra como foi construído arquivo de service de problemas.

Figura 14 - Service de problemas.

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm'
import { Connection } from 'typeorm'
import { User } from '../auth/user.entity'
import { ProblemRepository } from '../problem/problem.repository'
import { CreateProblemsDto } from './dto/create-problems.dto'
import { Problems } from './problems.entity'
import { ProblemsRepository } from './problems.repository'

@Injectable()
export class ProblemsService {
  private problemRepository: ProblemRepository
  private problemsRepository: ProblemsRepository
  constructor( private readonly connection: Connection ) {
    this.problemRepository = this.connection.getCustomRepository(ProblemRepository)
    this.problemsRepository = this.connection.getCustomRepository(ProblemsRepository)
  }

  async getAllProblems(user: User): Promise<Problems[]> {
    const found = await this.problemsRepository.find({ where: { user } , select: ['id', 'title',
'description']})

    if (!found) {
      throw new NotFoundException()
    }

    return found
  }

  async getProblemsById(id: number): Promise<Problems> {
    const found = await this.problemsRepository.findOne({ where: { id } })

    if (!found) {
      throw new NotFoundException()
    }

    return found
  }

  async createProblems(createProblemsDto: CreateProblemsDto, user: User): Promise<Problems> {
    return this.problemsRepository.createListProblems(createProblemsDto, user)
  }
}
```

fonte: (própria 2020).

O service de problems consiste nos 3 métodos apresentados, o *getAllProblems* que por meio de um select usando a função *find* fornecida pela ferramenta TypeORM, é retornado os valores de id, título e descrição daquele set de problemas. Possui também a função *getProblemById* que além de retornar estes mesmos dados, retorna todos os problemas atrelados ao set definido com tal id. E por fim a função de criação de set de problemas, que recebe o DTO mostrado na **imagem 14** e passa tanto ele, quanto o usuário recebido por meio do token usando a annotation de *GetUser* para o repository de problemas, que irá fazer o trabalho de salvar este set, e os problemas do mesmo, atrelando-os.

A **imagem 15**, a seguir, ilustra como está constituída a o arquivo de *problems repository*.

Imagem 15 - Problemas repository.

```
import { InternalServerErrorException } from '@nestjs/common'
import { EntityRepository, Repository } from 'typeorm'
import { User } from '../auth/user.entity'
import { Problem } from '../problem/problem.entity'
import { CreateProblemsDto } from './dto/create-problems.dto'
import { Problems } from './problems.entity'

@EntityRepository(Problems)
export class ProblemsRepository extends Repository<Problems> {
  async createListProblems(createProblemsDto: CreateProblemsDto, user: User): Promise<Problems> {
    const { title, description, problems } = createProblemsDto

    const problemsServer = new Problems()
    problemsServer.title = title
    problemsServer.description = description
    problemsServer.user = user


    try {
      await problemsServer.save()
    } catch (error) {
      throw new InternalServerErrorException()
    }

    const problemsList = problems
    const problemListServer: Problem[] = []
    let i = 0
    for ( i; i <= problemsList.length - 1 ; i++ ) {
      const problem = new Problem()
      problem.title = problemsList[i].title
      problem.description = problemsList[i].description
      problem.gravity = problemsList[i].gravity
      problem.urgency = problemsList[i].urgency
      problem.trend = problemsList[i].trend
      problem.total = problemsList[i].gravity * problemsList[i].urgency * problemsList[i].trend
      problem.problem = problemsServer

      try {
        await problem.save()
      } catch(error) {
        throw new InternalServerErrorException()
      }

      problemListServer.push(problem)
    }



    delete problemsServer.user
    return problemsServer
  }
}
```

fonte: (própria 2020).

O problems repository fica responsável por salvar os dados do problema definido, e por meio de um laço *for* vai criando os de problema e salvando os

mesmos no banco de dados e os mesmos já estando atrelados ao set seu respectivo set de problemas.

Por meio destas funções, arquivos, classes, bibliotecas e dados foi construído todo o back-end da aplicação Noelle.

Agora será abordado o front-end de toda a aplicação Noelle.

3.2 - Front-end:

Todo front-end da aplicação foi feito com auxílio da biblioteca React, uma das mais famosas bibliotecas em JavaScript feitas para criação de aplicação SPA (single page application), aplicações sem reload inteiro de página, apenas componentes sendo recarregados em seu browser, assim tornando web-sites mais bonitos, acessíveis e rápidos.

Além do uso da biblioteca React, foi usado o mais novo **Next.js,** um framework React com foco em produção e eficiência criado e mantido pela equipe da Vercel, uma companhia de criação e desenvolvimento de ferramentas para desenvolvimento web de maneira open-source, ou seja, gratuíto. O **Next.js** se destaca por sua velocidade, praticidade em construção da arquitetura do código e do projeto todo. Ele também traz grande facilidade de edição do produto a ser desenvolvido. Uma grande *mão na roda* do **Next.js** é sua total compatibilidade com a linguagem de programação utilizada no desenvolvimento do projeto, o **typescript**.

Além destas duas ferramentas, foi utilizado o **styled-component,** uma biblioteca para criação de componentes react estilizados com folhas de estilo SCSS, e a biblioteca **material-ui** que fornece uma série de componentes prontos gratuitos e responsivos para que os desenvolvedores não tenham tantas dores de cabeça no desenvolvimento de suas aplicações front-end. E por fim, teve o uso do **axios** para requisições HTTP entre front-end e back-end da aplicação.

3.2.1 - Tela de Login:

Na **imagem 16** é demonstrado o resultado final da tela de Login e Registro de usuário da aplicação, por se tratar de um protótipo, todas as telas foram feitas de maneiras que sejam, simples, intuitivas e de fácil manuseio para com o usuário.

Imagem 16 - Tela de login e registro de usuário.

fonte: (própria 2020).

A tela de Login da aplicação fica junto da tela de cadastrar um novo usuário, a mesma foi escolhida para ser simples, usando as cores básicas da aplicação, que é o branco-e-preto, cores selecionadas para trazerem seriedade ao protótipo que pode ser usado por times de planejamento de empresas para organização de funções e atividades.

As requisições de dados para o backend foram todas feitas com ajuda da biblioteca axios, uma biblioteca que facilita em todo processo de requisições feitas por meio de HTTP, o axios é usado em toda a aplicação tanto para envio de dados a serem cadastrados, como um set de problemas, ou um usuário, ou login de um usuário no sistema. Esta biblioteca além de facilitar, torna todo o processo de uma requisição mais rápido, e o código mais limpo. Primeiro foi criado uma classe que exporta uma constante chamada API, está constante cria uma instância de Axios, com a URL para o back-end, o tipo de dados a serem enviados, que no caso são em

formato JSON, e o token de autenticação, se a URL da requisição pedir uma autenticação, como nos casos das URL de problemas. Na **imagem 17** é ilustrado a criação da constante de API, que será sempre chamada para todas as requisições HTTP.

Imagem 17 - Constante de API.

```
import axios from 'axios'

const api = (token?: string) => {

  return axios.create({
    baseURL: 'http://localhost:3001/',
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${token}`
    }
  })
}

export default api
```

fonte: (própria 2020).

Já na **imagem 18,** há uma representação do código utilizado para envio dos dados para requisição HTTP de Login.

Imagem 18 - Código de Login.

```
const submitLogin = async (values: FormLoginValues) => {
  const caller = api()
  try {

    const call = await caller.post('auth/sign-in', values)
    localStorage.setItem('token', call.data.accessToken) //Set the tokenof the user in localStorage
    router.push(`/home`)//Redirect the user to the home.

  } catch (error) {
    alert(error.message)
  }
}
```

fonte: (própria 2020).

O exemplo acima é como foi feita a chamada para API, após inicializar a instância do axios, por meio de um POST foi enviado os dados, que estão armazenados na variável *value,* estes dados ficam dentro de uma função assíncrona, funções assíncronas em **typescript** ou javascript, são funções que precisam ser resolvidas para que então se tenha o resultado, o código espera que as mesmas sejam finalizadas para dar continuidade no processo. No caso, a função fica dentro de um bloco de *try catch,* que se trata de um bloco que irá tentar realizar o que está dentro das chaves de *try* e caso ocorra algum erro, como o usuário não tiver digitado o valor real, o erro será pego pelo bloco *catch* que retornará a mensagem de erro por meio de um modal de alerta do browser.

3.2.2 - Home:

A **imagem 19**, a seguir, representa a página de Home de um usuário que acabou de se cadastrar, ou seja, um usuário que não possui nenhum set de problemas para que o aplicativo o ajude a se organizar e trabalhar com os mesmos, com isso, ele é apresentado a uma série de textos explicando o que se trata a matriz GUT, e o que o projeto Noelle se propõe a fazer para com o usuário. Além disso, a tela conta com um botão para criar uma lista de problemas, como também uma

navbar, com links para o registro de set de problemas, home ou logout do usuário do sistema.

Imagem 19 - Tela de Home inicial.

fonte: (própria 2020).

Caso o usuário já seja cadastrado no sistema, o mesmo é levado para uma Home contendo os cards que o mesmo registrou no sistema como seus sets de problema, como mostra na **imagem 20**.

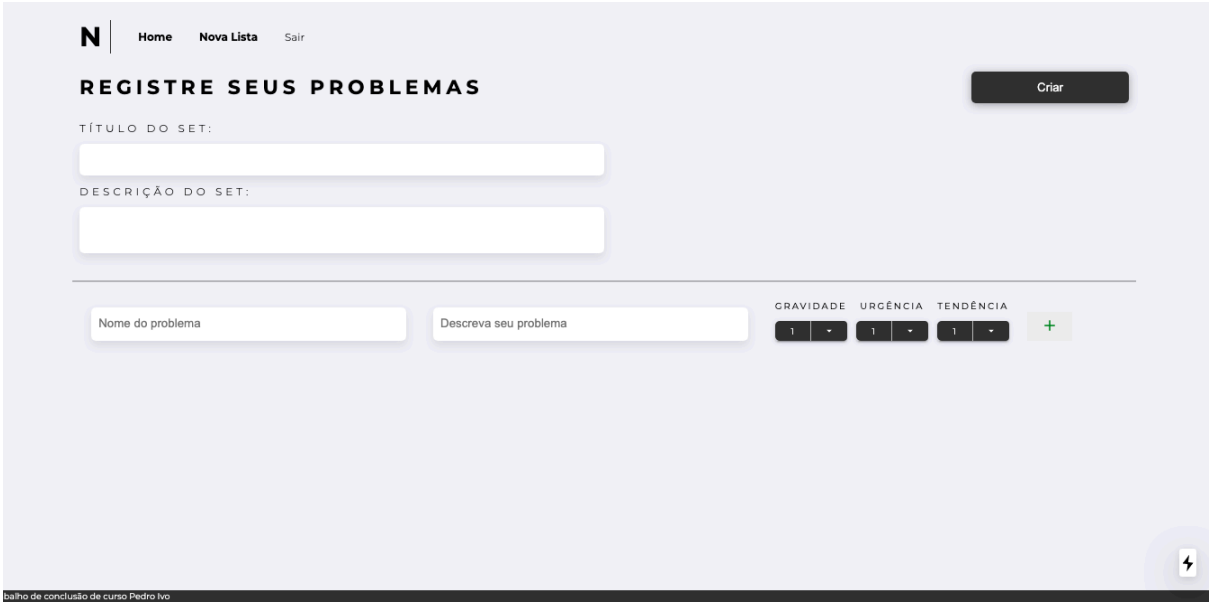
Imagem 20 - Exemplo de tela já com dados cadastrados.

fonte: (própria 2020).

3.2.3 - Cadastro de Set de Problemas:

A tela de cadastro de problemas, **imagem 21**, é onde o usuário será designado para cadastrar o título e a descrição do que se trata a situação que o mesmo está envolvido, em seguida, o mesmo pode adicionar, ou remover as etapas a serem seguidas, e avaliá-las conforme os valores definidos pela matriz GUT.

Imagem 21 - Tela de Registro de set de problemas.



fonte: (própria 2020).

Ao clicar no botão de criar o usuário dispara uma requisição para o banco de dados, com o seu array de problemas, título e descrição do set, e assim o mesmo é redirecionado para a tela com os detalhes, e o resultado do cálculo do GUT, podendo ser guiado a partir das a partir das cores, para saber o que priorizar.

No botão verde com sinal de mais o usuário pode adicionar mais um problema à lista, e no vermelho ele pode remover o mesmo problema. Como mostra na **imagem 22**.

Imagem 22 - Representação de form para cadastro de problemas.

fonte: (própria 2020).

3.2.4 - Tela de detalhes:

Por fim temos a tela de detalhes da situação apresentada pelo usuário, nela além de ter a descrição de toda a situação, o título, id, e dos problemas, temos o total de pontos de cada problema a partir do cálculo de multiplicação dos valores apresentados pelo mesmo, seguindo o cálculo proposto pela metodologia GUT, e partir das cores da tabela, o usuário fica de maneira mais intuitiva a par dos problemas a serem tratados com mais antecedência e urgência pelo mesmo, e dos que podem aguardar para serem resolvidos, pois não possuem a mesma urgência que os demais de acordo com a metodologia GUT. A **imagem 23** finaliza todo o processo de telas do protótipo.

Imagem 23 - Tela de detalhes do set de problemas.

fonte: (própria 2020).

Nesta tela, a tabela apresenta todos os problemas que o usuário digitou, e seus detalhes, a última coluna da tabela representa o resultado total proposto pela matriz GUT.

4 - Conclusão e Visões futuras para o protótipo:

Conforme apresentado durante a descrição da Matriz GUT, por se tratar de uma ferramenta de priorização, acaba se tornando uma grande ferramenta para ajuda de times de planejamento, ou pessoas com trabalhos onde requer certa responsabilidade, ou até mesmo um usuário comum que deseja usar destes critérios de priorização para se organizar em seu dia-à-dia. O protótipo criado vem com esse intuito de trazer a idéia de algo simples, e fácil de ser utilizado, de maneira gratuita e intuitiva para com que o usuário tenha a melhor experiência possível durante seu uso. Conforme foi-se desenvolvendo o protótipo foi analisado que se o projeto crescer, é possível a migração do mesmo para plataformas móveis, trazendo mais acessibilidade e usabilidade da ferramenta no dia-à-dia de times e empresas, trazendo a Matriz GUT de maneira tecnológica, e gratuita para todos que tiverem acesso a internet em suas mãos.

O protótipo apesar de simples, foi desenvolvido com as tecnologias mais atuais do mercado, se destacando por sua velocidade, interação e design atual. Assim, tendo grandes chances de crescer cada vez mais, deixando de ser um protótipo no futuro para se tornar uma plataforma de priorização que sirva de braço direito para usuários em suas organizações, assim como as plataformas Trello e Notion.

Além destas, mais tarde o Noelle pode trabalhar com outras metodologias de priorização de situações problema podendo ser aplicadas outras e o usuário podendo selecionar qual vai ser a metodologia usada. Ou seja, tornando o Noelle numa ferramenta completa de priorização onde o usuário pode escolher qual metodologia seu set de problemas irá usar para resolver. Tornando uma aplicação grande, e também podendo entrar no mobile.

5 - Bibliografia:

Andreia Silva Justo, Euax, 2020. Matriz GUT: entenda o que é e como aplicá-la na priorização dos seus projetos.
Disponível em: <https://www.euax.com.br/2019/04/matriz-gut/> Acesso em: 18 de junho de 2020.

Axios. Promise based HTTP client for the browser and node.js.
Disponível em <https://github.com/axios/axios> Acesso em: 20 de setembro de 2020.

DevMedia. Como o JWT funciona.
Disponível em <https://www.devmedia.com.br/como-o-jwt-funciona/40265> Acesso em: 18 de agosto de 2020.

Facebook Open Source Inc. ReactJs, 2020, Introduction to reactjs
Disponível em <https://pt-br.reactjs.org/> . Acesso em: 20 de julho de 2020

Facebook Open Source. ReactJS, 2020, Getting Started React JS
Disponível em <https://pt-br.reactjs.org/docs/getting-started.html/> Acesso em: 21 de julho de 2020.

JWT. JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.
Disponível em <https://jwt.io/> Acesso em: 18 de agosto de 2020.

Material-UI. React components for faster and easier web development. Build your own design system, or start with Material Design.
Disponível em <https://material-ui.com/> Acesso em: 10 de outubro de 2020.

Microsoft. TypeScript, 2020, TypeScript handbook
Disponível em <https://www.typescriptlang.org/docs/handbook/intro.html/> Acesso em: 5 de agosto de 2020.

Next.JS. The React Framework for Production.
Disponível em <https://nextjs.org/> Acesso em: 20 de setembro de 2020.

NestJS. A progressive Node.js framework for building efficient, reliable and scalable server-side application.
Disponível em <https://nestjs.com/> Acesso em: 15 de agosto de 2020.

NestJS. Documentation os NestJS.
Disponível em <https://docs.nestjs.com/> Acesso em: 15 de agosto de 2020.

Public. BCrypt, 2020, Bcrypt lib
Disponível em <https://www.npmjs.com/package/bcrypt/> Acesso em: 5 de agosto de 2020.

PostgreSQL. PostgreSQL: The World's Most Advanced Open Source Relational Database.
Disponível em: <https://www.postgresql.org/> Acesso em: 20 de agosto de 2020.

Renata Freitas de Camargo, Treasy, 2018. Como fazer a Matriz GUT para a resolução de problemas? Conheça a Matriz de Prioridades.
Disponível em: <https://www.treasy.com.br/blog/matriz-gut/> Acesso em: 17 de junho de 2020.

Styled-Components. Visual primitives for the component age.
Disponível em <https://styled-components.com/> Acesso em: 8 de outubro de 2020.