Uma API REST para a contratação de profissionais na aplicação Severino

Pedro Henrique Faria Teixeira

Universidade Federal de Uberlândia
Faculdade de Computação
Trabalho de conclusão de curso em Ciência da Computação

Uberlândia
2021

Pedro Henrique Faria Teixeira

Uma API REST para a contratação de profissionais na aplicação Severino

Trabalho de Conclusão de Curso apresentada ao Programa de conclusão de curso de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Graduação em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Prof. Dr. Marcelo Rodrigues de Sousa

Uberlândia
2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Engenharia Elétrica
Av. João Naves de Ávila, 2121, Bloco 3N - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4701/4702 - www.feelt.ufu.br - feelt@ufu.br

ATA DE DEFESA - GRADUAÇÃO

Curso de Graduação em:	Ciência da Computação				
Defesa de:	GBC082 - Projeto de Graduação 2				
Data:	25/06/2021	Hora de início:	14h	Hora de encerramento:	14h55min
Matrícula do Discente:	11621BCC025				
Nome do Discente:	Pedro Henrique Faria Teixeira				
Título do Trabalho:	Uma API REST para a contratação de profissionais na aplicação Severino.				

Reuniu-se, por meio da plataforma de Conferência Web (Google Meet) - utilizada como alternativa à reunião presencial, considerando as recomendações do Ministério da Saúde e a pandemia COVID-19, a Banca Examinadora, designada pelo Colegiado do Curso de Graduação em Ciência da Computação: Bacharelado - Integral, assim composta: (Professores: Dr. Igor Santos Peretta - FEELT/UFU; Dr. Márcio José da Cunha - FEELT/UFU e Dr. Marcelo Rodrigues de Sousa - FEELT/UFU orientador do candidato.

Iniciando os trabalhos, o presidente da mesa, Dr. Marcelo Rodrigues de Sousa, apresentou a Comissão Examinadora e o candidato(a), agradeceu a presença do público, e concedeu ao discente a palavra, para a exposição do seu trabalho. A duração da apresentação do discente e o tempo de arguição e resposta foram conforme as normas do curso.

A seguir o(a) senhor(a) presidente concedeu a palavra, pela ordem sucessivamente, aos(às) examinadores(as), que passaram a arguir o(a) candidato(a). Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o(a) candidato(a):

(X) Aprovado(a) Nota 85 (Somente números inteiros)

OU

( ) Aprovado(a) sem nota.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Marcelo Rodrigues de Sousa**,

Professor(a) do Magistério Superior, em 29/06/2021, às 19:04, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Igor Santos Peretta**, **Professor(a) do Magistério Superior**, em 29/06/2021, às 19:27, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcio José da Cunha**, **Professor(a) do Magistério Superior**, em 29/06/2021, às 20:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2849272** e o código CRC **685DF471**.

Esta monografia é dedicada aos trabalhadores informais brasileiros que estão em busca de melhora financeira.

Agradecimentos

A Deus, por ter permitido que eu tivesse saúde e determinação para não desanimar durante a realização desse trabalho. Aos meus pais, por nunca terem medido esforços para me proporcionar um ensino de qualidade durante todo o meu período escolar. Ao meu orientador Marcelo Rodrigues de Sousa, que conduziu o trabalho com paciência e dedicação, sempre disponível a compartilhar todo o seu vasto conhecimento. A minha noiva, pelo companheirismo, pela cumplicidade e pelo apoio em todos os momentos delicados da minha vida. Aos meus colegas de curso, cm quem convivi intensamente durante os últimos anos, pelo companheirismo e pela troca de experiências que me permitiram crescer não só como pessoa, mas também como formando. A todos aqueles que contribuíram, de alguma forma, para a realização desse trabalho.

"Aqueles que têm um grande autocontrole ou que estão totalmente absortos no trabalho falam pouco. Palavra e ação juntas não andam bem. Repare na natureza: trabalha continuamente, mas em silêncio."

(Mahatma Gandhi)

Resumo

Quando surge um problema doméstico ou uma demanda de algum serviço a ser realizado na vida das pessoas, por exemplo, imprevistos com encanamento ou rede elétrica, serviços de jardinagem ou pintura, elas tendem a contratar alguém para prestar o serviço, já que, em muita das vezes, não possuem experiência para realizar tal tarefa. Para solucionar esse problema, propõe-se a aplicação Severino, um sistema multiplataformas capaz de fazer uma interação simples e direta entre profissionais e clientes empregadores. Diante disso, a aplicação proporcionará aos usuários uma interface intuitiva e inovadora para que possam se cadastrar e navegar a procura por diversos serviços com seus respectivos profissionais. Assim, para o melhor entendimento do problema dos usuários, essa monografia aborda a construção do *back-end* ao qual foi feito um processo de levantamento de requisitos e a modelagem de banco de dados, além de ter criado uma prototipagem anteriormente, a fim de que o sistema final fosse uma experiência agradável aos usuários para, assim, terem maior adesão a aplicação e auxiliar na implementação do sistema que é composto por uma API REST e por um banco de dados, o qual alimenta os sítios da *web* e o aplicativo *mobile* da aplicação Severino.

Palavras-chave: Sistema *Back-end*, Desenvolvimento, Aplicação Severino, Contratação de serviços.

Abstract

When there is a domestic problem or a demand for some service to be performed in people's lives, for example, unforeseen problems with plumbing or electricity, gardening or painting services, they tend to hire someone to provide the service, since in a lot of sometimes, they do not have the experience to perform such a task. To solve this problem, the Severino application is proposed, a multiplatform system capable of making a simple and direct interaction between professionals and employers. Therefore, the application will provide users with an intuitive and innovative interface so that they can register and browse the search for various services with their respective professionals. Thus, for a better understanding of the users' problem, this monograph addresses the construction of the *back-end*, which was carried out through a requirements gathering process and the database modeling, in addition to having previously created a prototyping, in order for the final system to be a pleasant experience for users to, thus, have greater adherence to the application and assist in the implementation of the system, which is composed of a REST API and a database, which feeds the sites *web* and the *mobile* application of the Severino application.

Keywords: System *Back-end*, Development, Severino Application, Service Contracting.

Lista de ilustrações

Figura 1 – Ilustração da proposta de valor do Severino. 26
Figura 2 – Ilustração do Canvas de Modelo de Negócio do Severino. 27
Figura 3 – Representação da divisão do projeto. 28
Figura 4 – Atividades criadas no Trello para a construção do projeto. 34
Figura 5 – Representação da arquitetura MVC. 44
Figura 6 – Representação da arquitetura relacional. 45
Figura 7 – Caso de uso para o usuário cliente empregador da aplicação Severino. . 52
Figura 8 – Caso de uso para o usuário trabalhador da aplicação Severino. 52
Figura 9 – Fluxograma explicativo do cliente empregador do usuário. 53
Figura 10 – Fluxograma explicativo do prestador de serviço do trabalhador. 53
Figura 11 – Diagrama Entidade-Relacionamento do banco de dados. 58
Figura 12 – Representação do funcionamento da aplicação. 60
Figura 13 – Representação do funcionamento do sistema *back-end*. 61

Lista de tabelas

Tabela 1 – Tabela explicatória para os casos de uso dos usuários. 54

Lista de siglas

ABNT Associação Brasileira de Normas Técnicas

API Application Programming Interface

ACID Atomicity, Consistency, Isolation, Durability

AWS Amazon Web Services

CI Circle Integration

CLI Command-line Interface

CORS Cross-Origin Resource Sharing

CICS Customer Information Control System

CRUD Create Updated Delete

DBA Database Administrator

DDL Data Description Language

DER Diagram Entity Relationship

ECDSA Elliptic Curve Digital Signature Algorithm

HTTP Hypertext Transfer Protocol

HTML Hyper Text Markup Language

HLL High-level Programming Language

iOS iPhone OS

JSON JavaScript Object Notation

JWT JSON Web Token

MVC Model View Controller

NLP Natural Language Processing

NBR Denominação de norma da Associação Brasileira de Normas Técnicas

OS Operating System

ORM Object Relational Mapping

OIT Organização Internacional do Trabalho

OOPS Object Oriented Programming

POO Programação Orientada a Objetos

RSA Rivest–Shamir–Adleman

REST Representational state transfer

RDBMS Relational Database Management System

SOAP Simple Object Access Protocol

SSD Solid-state Drive

SQL Structured Query Language

URL Uniform Resource Locator

WSDL Web Services Description Language

Sumário

1 INTRODUÇÃO . . . 23
1.1 A aplicação Severino . . . 24
1.1.1 Motivação . . . 24
1.1.2 Usuários . . . 24
1.1.3 As dores dos usuários . . . 25
1.1.4 Contribuição da Aplicação . . . 25
1.1.5 Mercado potencial e Concorrência . . . 25
1.2 O desafio de desenvolver a aplicação Severino . . . 27
1.2.1 Arquitetura da solução . . . 27
1.2.2 Divisão de trabalho e gestão de desenvolvimento . . . 28
1.2.3 Integração e testes . . . 29
1.3 Objetivos e Desafios da Pesquisa . . . 29
1.4 Método . . . 29
1.5 Organização do trabalho . . . 30

2 FUNDAMENTAÇÃO TEÓRICA . . . 31
2.1 Conceitos de desenvolvimento de software aplicados . . . 31
2.1.1 Programação Orientada a Objetos . . . 31
2.1.2 Engenharia de Software . . . 32
2.1.3 Engenharia de Requisitos . . . 32

3 METODOLOGIA . . . 33
3.1 Metodologia de gestão aplicada . . . 33
3.1.1 Metodologia Ágil e Kanban . . . 33
3.2 Ferramentas e Tecnologias de Desenvolvimento . . . 35
3.2.1 JSON Server . . . 35
3.2.2 Insominia REST . . . 35
3.3 Banco de dados Relacional . . . 35
3.3.1 PostgreSQL . . . 35
3.3.2 Git e GitHub . . . 36
3.3.3 Yarn . . . 36
3.3.4 Node.js . . . 36
3.3.5 JavaScript . . . 37

3.3.6 TypeScript 37
3.3.7 Docker e Docker Compose 37
3.3.8 TypeORM 38
3.3.9 Express 39
3.3.10 JSON Web Token 39
3.3.11 Faker 40
3.3.12 TSNodeDev 40
3.3.13 CORS 40
3.3.14 BcryptJS 40
3.3.15 Multer 40
3.4 Arquitetura 41
3.4.1 API Rest 41
3.4.2 MVC 43
3.4.3 Arquitetura relacional 44
3.4.4 Principais Características do desenvolvimento *back-end* da aplicação Severino 46
3.4.5 Estrutura de pastas e componentes 47

4 DESENVOLVIMENTO 49
4.1 Levantamento de requisitos 49
4.2 Modelagem do Banco de Dados 55
4.3 Protótipo 58
4.4 Funcionalidades do TypeORM 59
4.4.1 Entidades 59
4.4.2 Migration 59
4.4.3 Seeds 59
4.5 Inicialização do banco de dados 59
4.6 Implementação 60
4.6.1 Back-End 60

5 CONCLUSÃO 63
5.1 Desafios encontrados 63
5.2 Trabalhos Futuros 64

REFERÊNCIAS 65

Capítulo 1

Introdução

O Brasil é um dos países com o maior número de trabalhadores informais ativos, o que se deve a alta taxa de desemprego e a falta de mão de obra qualificada. Em dezembro de 2020 a taxa de informalidade atinge 39,5%, o que significa que cerca de 34,029 milhões de trabalhadores não atuam em um emprego com carteira assinada e não possuem a segurança de uma renda fixa (RIO, 2021). Ainda no ano de 2020, a percentagem do desemprego subiu para 14,7% (IBGE.GOV.BR, 2021), sendo que todos esses aumentos nas estatísticas brasileira está relacionado, dentre outros fatores, ao cenário complexo de pandemia em que estamos vivendo, logo as pessoas estão tendo que se reinventar e, cada vez mais, são incentivados a buscarem formas alternativas de renda. Com isso, é cada vez mais comum encontrar trabalhadores atuando em plataformas como IFood, Uber, Rappid, dentre outros.

Diante desse cenário de intensa migração do ambiente trabalhista e como não há muitas aplicações no mercado brasileiro que possui a mesma entrega, surgiu a ideia de criar o Severino, uma aplicação a qual os usuários podem contratar profissionais, de forma rápida, para prestar diferentes serviços. Assim, observa-se que a plataforma beneficia tanto os trabalhadores informais, de qualquer área de serviços, que encontram clientes e têm, como consequência, um aumento ou obtenção da renda salarial, quanto os usuários que precisam encontrar alguma prestação de serviço em específico para o seu dia a dia. Ademais, nessa aplicação, os diversos profissionais podem, ainda, acessar a plataforma para divulgarem seus trabalhos e experiências como forma de *marketing* pessoal.

Em suma, procurar por profissionais em listas telefônicas, panfletos, ou na *internet* para contratar algum serviço, se tornou obsoleto, assim como buscar por indicações profissionais no meio social, pois, a plataforma fornece uma aplicação simples, sofisticada e intuitiva com todas as informações necessárias para contratar um profissional rapidamente, prestadores de serviços esses que não possuem, na grande maioria dos casos, um meio de comunicação com seus clientes, dado isso os trabalhadores ficam sujeitos a esperarem que os empregadores entrem em contato com eles, para enfim, serem contratados.

Optou-se pelo nome “Severino” em homenagem ao personagem do programa de televisão “Zorra Total”, o qual exercia diversas funcionalidades além da sua profissão, sendo assim, já que a aplicação oferece diferentes serviços a serem contratados foi feita uma analogia com a intenção de que a funcionalidade da plataforma seja intuitiva aos usuários. O Severino se constitui de quatro partes técnicas diferentes, as quais se complementam, sendo elas, uma página *web* para apresentar aos usuários os profissionais, um aplicativo

mobile com o mesmo intuito, outra página *web* responsável por realizar o cadastro dos trabalhadores e, por último, o *back-end* que guarda e fornece informações para as demais partes.

Dessa forma, por ser uma aplicação complexa, para construí-la foi feita a união de três integrantes do curso de Ciência da Computação da Universidade Federal de Uberlândia, sendo o aluno Gabriel Mendes responsável pelas páginas *webs*, João Daniel pelo aplicativo e Pedro Teixeira pelo *back-end* da aplicação *web* e *mobile*.

O foco dessa monografia é abordar os passos para a construção do *back-end*, o qual se constitui de uma API REST que fornece *endpoints* para estabelecer a conexão entre os sistemas por meio de requisições HTTP que tem como finalidade fazer o tráfego de dados entre o banco de dados relacional e os *front-ends*.

1.1 A aplicação Severino

1.1.1 Motivação

A necessidade de criar uma API REST que fornecesse dados para os sítios da *web* e o aplicativo *mobile* da aplicação Severino é o motivo para a construção da monografia. Com essa API espera-se fazer com que ambas as plataformas funcionem perfeitamente, tanto a *web* quanto a *mobile*, pois a partir delas que é feito o tráfego de informações e o armazenamento dos dados no banco de dados relacional. Além disso, a criação desse *software* pode mudar a vida de milhares de pessoas pelo Brasil, visto que o profissional têm a oportunidade de divulgar seu trabalho e aumentar sua renda e os clientes empregadores podem ter um acesso simples e rápido a esses profissionais.

1.1.2 Usuários

A aplicação possui dois tipos de usuários, clientes empregadores e os prestadores de serviços.

1. O cliente empregador é a pessoa que está com alguma atividade em pendência e precisa de algum profissional para realizá-la ou auxiliá-lo. Dessa forma, o cliente usa a aplicação para pesquisar por um serviço em específico que necessita e, então, encontra uma lista de profissionais que realizam tal tarefa, além disso, podem ter acesso ao perfil do profissional, o que permite ao cliente empregador ter uma análise melhor do trabalhador, a fim de escolher o mais capacitado para o serviço.

2. O usuário prestador de serviço pode usar a aplicação como forma de obter uma nova renda salarial, ou então, para a divulgar o seu trabalho e aumentar a sua rentabilidade. O trabalhador possui uma página específica onde pode realizar seu cadastro, editar seu perfil com informações relevantes, tais como, formas de contato, experiência profissional e serviços que realiza. Por fim, após seguir os passos da página *web*, o profissional esta apto a ser contratado por algum cliente empregador.

1.1.3 As dores dos usuários

A dificuldade do cliente empregador surge quando algum imprevisto acontece em seu cotidiano ou ele necessita de algum serviço em específico e possui dificuldade para encontrar algum profissional que possa ajudá-lo. Assim, procurar por profissionais na *internet*, por indicações de amigos, em panfletos, ou até mesmo na lista telefônica, demanda tempo, o que torna o processo trabalhoso ou inviável, além da falta de confiança em pessoas desconhecidas. Com isso, o Severino fornece uma infraestrutura na qual o usuário empregador pode ver os comentários de pessoas que já contratou o profissional, além dos seus trabalhos já realizados anteriormente.

Por outro lado, a dificuldade do prestador de serviço é encontrar cliente, visto que não possui uma ferramenta de comunicação eficaz para fazer a divulgação do seu serviço e, assim, ter mais chances de ser contratado por alguém. Ademais, o trabalhador não sabe como o seu serviço é interpretado pelos clientes, se o seu trabalho é de boa qualidade ou não, o que impacta no momento de uma recontratação. Com isso, o Severino fornece uma estrutura de *feedbacks* ao profissional para que ele tenha mais confiança em seu trabalho.

1.1.4 Contribuição da Aplicação

A API foi criada para conectar três outras plataformas, sendo um aplicativo *mobile* e duas páginas *web* que tem como intuito facilitar a busca e a contratação de diversos profissionais. Assim, além de contribuir com as outras plataformas, espera-se ajudar os trabalhadores brasileiros que utilizarem a aplicação, visto que podem ter mais chances de encontrar um emprego e aumentar sua rentabilidade. Em relação aos clientes empregadores, almeja-se agilizar seu dia a dia com uma lista de serviços e seus respectivos profissionais, já que isso esta a poucos cliques de uma tela. Vale ressaltar, ainda, que na era tecnológica, a qual vivemos atualmente, é comum se deparar com pessoas que vivem do trabalho fornecido ou intermediado por aplicações, tais como, iFood e Uber. Diante disso, o Severino vem para se juntar ao mercado digital e logo gerar mais empregos para a sociedade brasileira.

1.1.5 Mercado potencial e Concorrência

O número de trabalhadores informais no Brasil é cerca de 34,029 milhões de pessoas, isso é 39,5% da taxa de informalidade no Brasil (RIO, 2021) no ano de 2021. Visto que a cada dia essa taxa cresce mais, junto a ela está a taxa de desemprego, a qual chegou na percentagem de 14,7% no ano de 2021 (IBGE.GOV.BR, 2021). Com isso, pode-se ter uma noção dos possíveis novos trabalhadores que optariam pela aplicação Severino como nova forma de renda salarial. Além das estatísticas apresentadas, pode-se avaliar a quantidade de profissionais cadastrados em outras aplicações concorrentes do Severino, tais como o GetNinjas e o Triider, sendo o GetNinjas a maior do mercado atual, já que possui mais de 500 mil trabalhadores cadastrados em sua base (MOBILETIME, 2018), já o Triider conta com cerca de 100 mil (MOBILETIME, 2019). Assim, com esses números percebe-se que a quantidade de profissionais é alta e possui crescimento contínuo.

Em contrapartida, é mais complexo mensurar os usuários que serão clientes empregadores, visto que não se tem uma forma de avaliação, porém é possível olhar para os dados de crescimento das aplicações existentes e já citadas acima, tais como, o GetNinjas que

demonstrou um crescimento de 102% no mês de junho de 2020 comparado ao ano anterior, devido ao aumento da demanda na pandemia, assim, todos os registros que foram comparados com o mesmo mês do ano de 2019 tiveram aumento na taxa de contratação da aplicação (GASPARIN, 2020), exemplo disso é que se teve uma média de 2 milhões de contratos entre profissionais e clientes em um ano. Já o Triider, em meio a pandemia, no ano de 2020, registrou um crescimento de 157% na demanda de contratações, realizou mais de 50 mil serviços nas regiões em que atua e pretende aumentar ainda mais a sua expansão dos negócios (JORNALDOCOMERCIO, 2021).

Diante das análises feitas acima, pode-se concluir que o mercado é amplo e possui espaço para o Severino atuar, dado que as aplicações existentes hoje no mercado não atendem em todas as regiões do país. Além disso, é deficitária as divulgações das aplicações citadas, como o GetNinjas e o Triider, visto que o conhecimento sobre elas foi adquirido durante pesquisas bibliográficas para essa monografia. Assim, para diferenciar o Severino das demais plataformas, primeiramente será feita sua divulgação voltada estritamente para a região em que irá atuar, tendo o foco os profissionais e depois os clientes empregadores. Ademais, será feito inicialmente promoções para atrair os trabalhadores para o Severino, sendo a primeira delas a isenção de pagamento do uso da aplicação no primeiro mês de cadastro, já que a aplicação irá trabalhar com o método de pagamento mensal por parte dos profissionais, a fim de que tenham mais liberdade na hora de cobrar os preços dos seus serviços. Por outro lado, para os usuários empregadores não haverá custo para usar o programa.

Por fim, para avaliar o que o Severino pode proporcionar aos seus usuários foi feito uma análise de proposta de valor e, também, foi criado o modelo de negócios para dar introdução da aplicação no mercado, sendo que os balões azuis pertencem aos clientes empregadores, os vermelhos aos profissionais e os em amarelo pertencem à ambos. A figura 1 referencia a proposta de valor e a figura 2 o modelo de negócios.

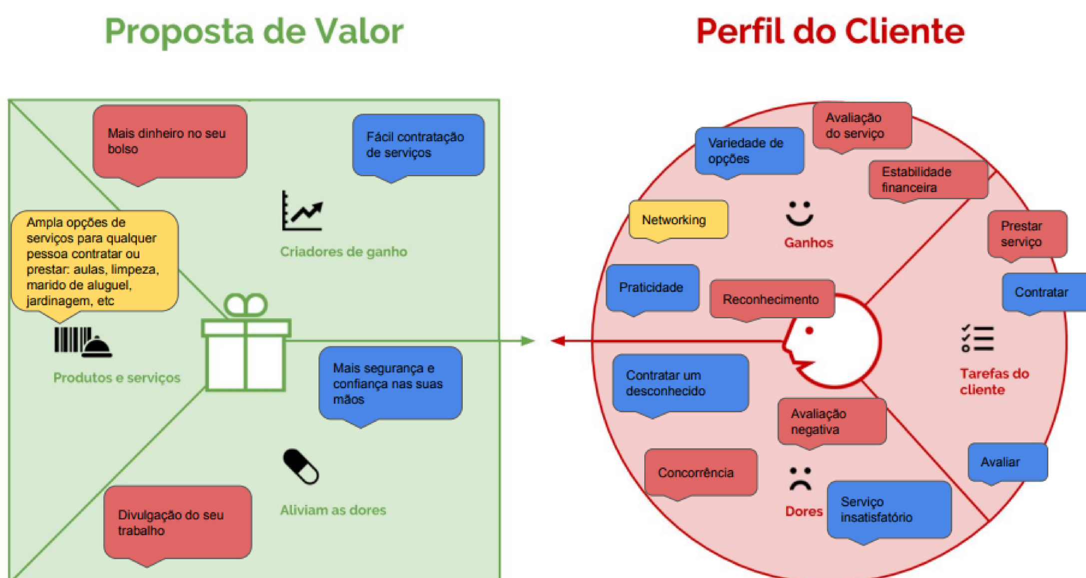

Figura 1 – Ilustração da proposta de valor do Severino.

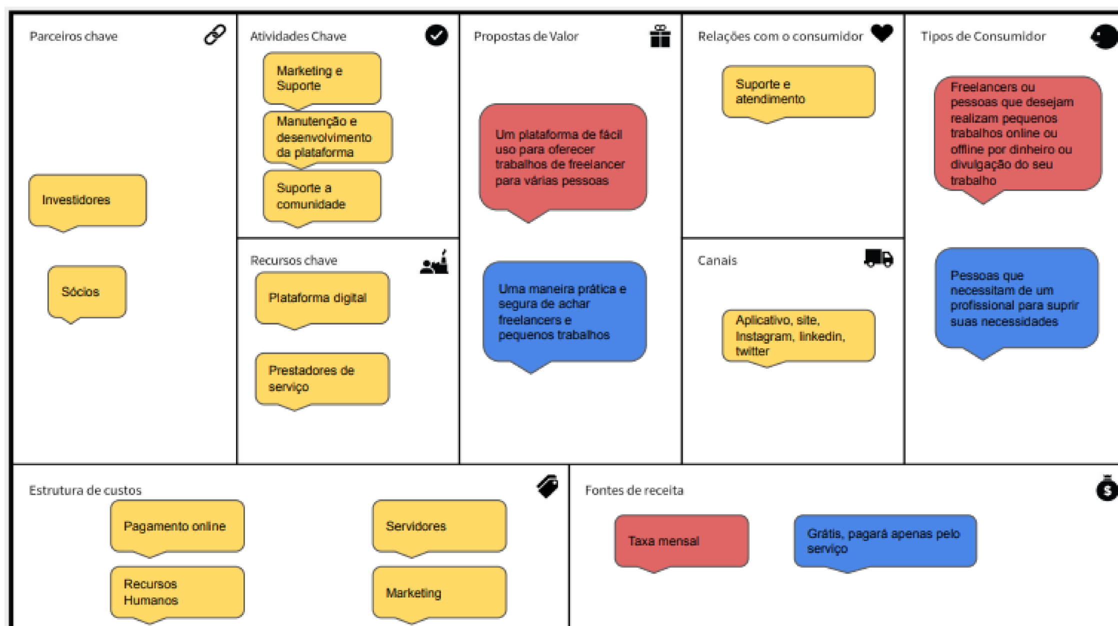

Figura 2 – Ilustração do Canvas de Modelo de Negócio do Severino.

1.2 O desafio de desenvolver a aplicação Severino

1.2.1 Arquitetura da solução

A aplicação Severino é formada por quatro componentes: uma página *web* para os clientes empregadores, um aplicativo *mobile* com o mesmo intuito, uma página *web* para o cadastro dos profissionais e por fim o *back-end* que fornece os dados para as páginas *web* e o aplicativo *mobile*.

1. A página *web* para os clientes empregadores: o *site* é focado na experiência do usuário que está em busca de trabalhadores para contratação e foi construído com um *design* inovador e responsivo para atender aos diversos tamanhos de telas, inclusive de *smartphones*. No site o usuário pode pesquisar por inúmeros serviços, listar em sua tela um conglomerado de diferentes tipos de serviços com seus respectivos profissionais, aplicar filtros por especificidade do serviço, localidade e por ranque dos profissionais. Ademais, é possível, também, ver o perfil do profissional para que, assim, o cliente tenha uma melhor avaliação do trabalhador que pode estar contratando.

2. O aplicativo *mobile*: tem o mesmo intuito que a página *web* do cliente empregador, sendo que ele foi construído com uma ferramenta híbrida, a qual atende tanto *smartphones* com o sistema operacional *Android* quanto *iOS*.

3. A página *web* para o cadastro dos profissionais: é o sítio no qual o profissional prestador de serviço segue um fluxo para a criação do seu cadastro. A página apresenta todas as informações necessárias para alguém que queira trabalhar na aplicação Severino e, assim, consiga realizar com sucesso seu cadastro e edição do seu perfil adicionando informações relevantes, tais como, formas de contato, localização, experiência de trabalho e serviços prestados.

4. O *back-end*: foi feito uma API REST para manipular e fornecer dados para as páginas *web* e para o aplicativo *mobile*. Todos os dados transacionados pelas plataformas é armazenado em um banco de dados relacional para fazer a comunicação entre os sistemas, o que é feito com requisições por meio de *endpoints*, buscando os dados do banco e levando para os *front-ends* e vice versa.

A figura 3 representa a divisão dos sistemas entre os integrantes e como cada usuário realiza seu acesso.

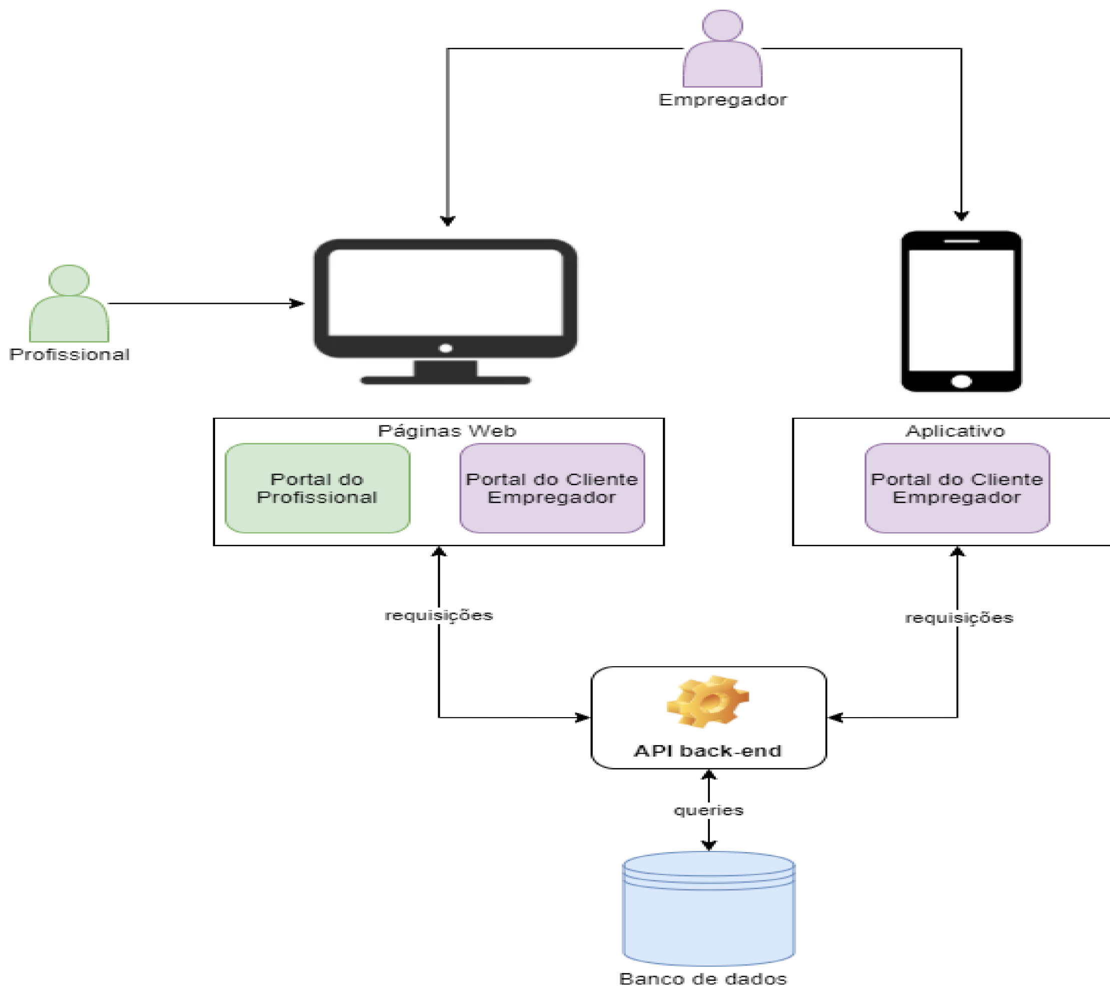

Figura 3 – Representação da divisão do projeto.

1.2.2 Divisão de trabalho e gestão de desenvolvimento

Foi feita uma junção de três alunos da Faculdade de Ciência da Computação da Universidade Federal de Uberlândia para a realização da aplicação, sendo eles Pedro Henrique Faria Teixeira, João Daniel Aquino Rufino e Gabriel Mendes, devido a complexidade da aplicação. Assim, a divisão das atividades de cada parte do projeto foi de crucial importância para que cada uma das partes tivesse a melhor performance possível.

A divisão foi feita da seguinte forma: o integrante Gabriel Mendes ficou responsável pelos sítios na *web*, João Daniel com o aplicativo *mobile* e, por último, Pedro Henrique com o *back-end*.

Para a gestão das atividades foi decidido utilizar o método ágil Kanban, explicado na seção 3.1.1. O método foi escolhido por ser muito utilizado em grandes empresas que atuam na área tecnológica, além de ser objetivo e fácil de gerenciar, dando uma visão clara do que cada integrante faz no projeto.

1.2.3 Integração e testes

A integração dos sítios da *web* com o *back-end* foi por meio de requisições HTTP utilizando um modelo REST de arquitetura, o qual é explicado na seção 3.4.1. A API fornece *endpoints* para que a comunicação entre os sistemas seja possível, a partir de métodos como PUSH, GET, PUT e DELETE.

Por fim, serão aplicados testes unitários para cada módulo e componente do *back-end*, a fim de testá-los separadamente e garantir o seu funcionamento. Para testar as *features* dos *sites* e aplicativo, foi pedido a amigos e familiares para fazerem o uso dos sistemas com a intenção de testar a usabilidade e conseguir *feedbacks*, para que, assim, seja feita alterações e no final do desenvolvimento o usuário tenha a melhor experiência possível.

1.3 Objetivos e Desafios da Pesquisa

O principal objetivo desse projeto de graduação foi criar uma API *back-end* para conectar a duas aplicações *web* e a outra *mobile* que foram construídas para complementar a aplicação Severino, sendo que o principal intuito da conexão é agilizar a vida de milhares de brasileiros, pois quando tiverem a necessidade de contratar um profissional para realizar algum trabalho, terão duas plataformas com diversos profissionais para contratação.

A aplicação Severino fornece dados dos prestadores de serviços, coleta a localidade dos clientes empregadores para garantir a apresentação de trabalhadores mais próximos geograficamente, oferece filtros para a restrição de chamadas ao serviço que o cliente especificar, além de informações dos trabalhadores nas aplicações *front-ends*, tais como número de contato, seu ranque, tipos de trabalhos prestados, comentários dos clientes que já o contrataram, dentre outras. Além disso, o sistema guarda todas as informações de cadastros, tanto do usuário empregador quanto do profissional, sendo que tudo isso fica armazenado em um banco de dados relacional.

Em relação aos desafios que foram encontrados na construção do projeto, pode-se citar o estudo aprofundado das tecnologias de desenvolvimento junto com os métodos de organização e gerenciamento durante a construção da aplicação, o estudo e entendimento dos procedimentos adotados para manter a conexão das duas aplicações *front-ends*, o planejamento de como iria ser criada as *queries* para as requisições HTTP e, por fim, a compreensão da estrutura relacional das tabelas do banco de dados.

1.4 Método

Para a realização dos objetivos apresentados na seção 1.3 e para a construção da aplicação *back-end*, foi seguido os seguintes passos:

- Criar protótipos para o problema especificado:

1. Estudo do mercado ao qual a aplicação irá participar;
2. Levantamento de requisitos do *back-end*;
3. Definir assinatura dos *endpoints* da aplicação;
4. Definição e estudo das ferramentas que serão utilizadas para construção do *back-end*;
5. Fazer a modelagem do banco de dados em um *software* especializado;
6. Designar as tarefas a serem realizadas no quadro kanban.

- Desenvolvimento da solução:

1. Subir um servidor em Node.js com Express;
2. Exportar as APIS com as especificações da *url* e assinatura dos *endpoints*;
3. Controlar as versões dos códigos com Git e GitHub;
4. Implementar os serviços utilizando TypeScript;
5. Esquematizar as tabelas do banco de dados no PostgreSQL;
6. Fazer as *migrations* usando o TypeORM para subir e alterar as tabelas quando necessário;

1.5 Organização do trabalho

A apresentação do conteúdo dessa monografia está separada da seguinte forma:

- O segundo capítulo abrange as principais tecnologias que foram utilizadas para a organização e construção do projeto;
- O terceiro capítulo apresenta o desenvolvimento do *back-end* para o tema proposto levantando os requisitos e descrevendo como foi realizado a construção dos serviços para a conexão dos *front-ends*;
- O quarto capítulo aborda a conclusão, os desafios encontrados durante o desenvolvimento e os futuros trabalhos a serem desenvolvidos a partir do mesmo.

Capítulo 2

Fundamentação Teórica

Esse capítulo contempla parte do conceito aplicado para a construção da API, os quais foram aprendidos durante a graduação de Ciência da Computação. Além disso, uma disciplina essencial foi a de "Programação Orientada a Objetos", na qual foi apresentada a arquitetura MVC que foi aplicada no código do *back-end*, junto com os paradigmas de programação orientada a objetos. Por fim, os aprendizados em programação orientada a objetos junto com os conhecimentos adquiridos na matéria de "Engenharia de Software" e "Modelagem de Software" foram um diferencial para a construção do projeto.

2.1 Conceitos de desenvolvimento de software aplicados

2.1.1 Programação Orientada a Objetos

A programação orientada a objetos (POO) é um paradigma de programação que se baseia no conceito de classes e objetos. É usado para estruturar um programa de *software* em pedaços simples e reutilizáveis de projetos de código (geralmente chamados de classes), que são usados para criar instâncias individuais de objetos (DOHERTY, 2020).

Uma classe é um projeto abstrato usado para criar objetos concretos mais específicos. As classes geralmente representam categorias amplas, como Carro ou Cachorro, que compartilham atributos. Essas classes definem quais atributos uma instância desse tipo terá, como cor, mas não o valor desses atributos para um objeto específico.

As classes também podem conter funções, chamadas métodos, disponíveis apenas para objetos desse tipo. Essas funções são definidas dentro da classe e executam alguma ação útil para aquele tipo específico de objeto (DOHERTY, 2020).

Benefícios da OOP:

- OOP modela coisas complexas como estruturas simples e reproduzíveis;
- Reutilizáveis, objetos OOP podem ser usados em programas;
- Permite comportamento específico de classe por meio de polimorfismo;
- Mais fáceis de depurar, as classes geralmente contêm todas as informações aplicáveis a eles;

- Seguro, protege as informações por meio do encapsulamento (DOHERTY, 2020).

2.1.2 Engenharia de Software

Os princípios da engenharia de *software*, quando executados de forma consistente e adequada, garantem que o processo de desenvolvimento do *software* seja executado de maneira contínua e eficiente com a entrega de *softwares* de alta qualidade. Quando os princípios são aplicados, garantem uma melhor compreensão de como está sendo construído e como cada pessoa contribui para o processo de construção (CASTSOFTWARE, 2015).

Há alguns passos para serem seguidos na aplicação da engenharia de *software*, sendo que geralmente os projetos os seguem para que tenham um padrão na etapa da construção e que a equipe fique ciente das etapas por onde o *software* percorrerá e, também, a situação atual naquele momento. Dessa forma, facilita para os desenvolvedores darem continuidade ao projeto, ou então, fazerem modificações. Diante disso, as etapas são “análise”, “projeto”, “codificação”, “testando” e “manutenção”. Porém, há outras fundamentações relacionadas a engenharia de *software* (PRESSMAN, 2014), sendo que as atividades relacionadas com a construção de um bom *software* segue os passos de comunicação, planejamento, modelagem, construção e emprego. Vale ressaltar que essas atividades podem ser realizadas em diferentes ordens.

2.1.3 Engenharia de Requisitos

Para garantir um *software* de alta qualidade é necessário utilizar a técnica de Engenharia de Requisitos que é indispensável para o sucesso do desenvolvimento, já que oferece padrões e exigências para o projeto (MONITORA, 2020). O método consiste em algumas etapas, tais como levantamento de requisitos, documentação, validação, verificação, gerência de requisitos e garantia de qualidade, as quais padronizam o gerenciamento do projeto e garantem maior produtividade durante o desenvolvimento, manutenção e operação. Todavia, para que o processo citado tenha sucesso, toda a equipe do projeto tem que estar totalmente alinhada com os objetivos a serem alcançados. Assim, tais etapas padronizam e garantem o sucesso da construção do *software*.

Capítulo 3

Metodologia

Esse capítulo apresenta como foi realizado a gestão das atividades do projeto durante a construção da API REST, ferramentas utilizadas e arquitetura de código. Em primeiro lugar será listada a metodologia de gerenciamento do projeto seguido das ferramentas e padrões usados para a construção do projeto. As ferramentas técnicas para a construção do *back-end* foram, em grande parte, aprendidas em estudos extra-curriculares, porém a parte arquitetural do projeto foi apresentado durante a graduação de Ciência da Computação. Ferramentas que gerenciam projetos, como métodos ágeis, foram escolhidas por fazer parte do cotidiano de um desenvolvedor, assim como fez parte do meu dia a dia quando atuei com desenvolvimento em cenário laboral. Enfim, sempre que vou desenvolver faço uso dos métodos ágeis por ser extremamente satisfatório quanto a sua função. Sobre o versionamento de código, outra ferramenta que está no cotidiano de um *dev*, são o Git e o Github, os quais são utilizados como portfólio dos programadores

3.1 Metodologia de gestão aplicada

3.1.1 Metodologia Ágil e Kanban

As metodologias ágeis são abordagens destinadas para o desenvolvimento de produtos que estão alinhados com os valores e princípios descritos no "Manifesto Ágil" (AGILEMANIFESTO, 2001) para o desenvolvimento de *software*. Visam entregar o produto certo, com entrega incremental e frequente de pequenos pedaços de funcionalidade, por meio de pequenas equipes multifuncionais auto-organizadas, permitindo *feedback* frequente do cliente e correção de curso conforme necessário.

Ao fazer isso, o *Agile* visa corrigir os desafios enfrentados pelas abordagens tradicionais de entrega em "cascata" de produtos grandes ou em longos períodos de tempo, durante os quais os requisitos do cliente mudam com frequência e resultam na entrega de produtos errôneos.

No presente projeto utilizou-se a metodologia Kanban, mesmo existindo inúmeras outras, pois é um dos métodos mais aplicados em empresas de tecnologia da informação, além de ser a metodologia que mais tive contato até o momento.

Kanban é uma estrutura popular usada para implementar o desenvolvimento de *software* ágil e DevOps. Requer comunicação em tempo real da capacidade e total transparência do trabalho. Os itens de trabalho são representados visualmente em um quadro

kanban, o que permite que os membros da equipe vejam a situação de cada trabalho em qualquer momento.

Sendo assim, o trabalho de todas as equipes que usam o método Kanban gira em torno de um quadro kanban. Embora os quadros físicos sejam populares entre algumas equipes, os quadros virtuais são um recurso crucial em qualquer ferramenta ágil de desenvolvimento de *software*, devido a sua rastreabilidade, colaboração facilitada e por ser facilmente acessível. Em suma, a metodologia depende da total transparência do trabalho e comunicação em tempo real, portanto, o quadro deve ser visto como a única fonte de verdade para a equipe.

Entretanto, independentemente do quadro da equipe ser físico ou digital, sua função é garantir que o trabalho seja visualizado, o fluxo do trabalho seja padronizado e todos os bloqueadores ou pendências sejam imediatamente identificados e resolvidos(ATLASSIAN, 2015). Um quadro kanban básico tem um fluxo de trabalho de três etapas, sendo elas “A Fazer”, “Em Andamento” e “Concluído”. No entanto, a depender do tamanho, estrutura e objetivos de uma equipe, o fluxo de trabalho pode ser mapeado para atender o time em específico com um processo exclusivo.

Na figura 4, tem-se o modelo do quadro utilizado durante a construção do projeto. Não é preciso se limitar aos fundamentos pregados pelo método kanban, sendo possível adicionar novas colunas para que o trabalho tenha uma melhor análise e desempenho. Por exemplo, pode-se adicionar colunas como "TESTANDO", "AGUARDANDO", "*BACKLOG*". No presente trabalho foi adicionado uma lista de ideias para ficarem registradas como novas possíveis funcionalidades ao projeto e outra coluna de *backlog*, onde deixava-se algumas funcionalidades já definidas e aptas para execução.

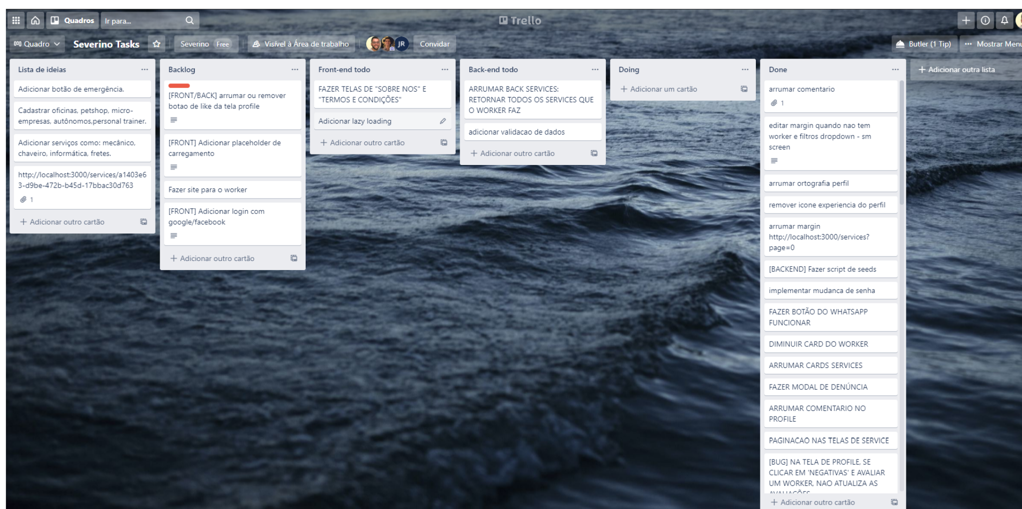

Figura 4 – Atividades criadas no Trello para a construção do projeto.

Para visualizar o quadro kanban foi utilizado o Trello, uma ferramenta *on-line*, a qual consiste em um quadro digital com os conceitos da metodologia Kanban. A figura 4 foi retirada dessa mesma plataforma.

3.2 Ferramentas e Tecnologias de Desenvolvimento

3.2.1 JSON Server

JSON Server é um pacote do Node.js que permite criar uma API REST completa e com dados falsos que possui como finalidade apenas realizar testes de integração com o *front-end* (NPMJS, 2020). Sua principal vantagem é que não precisa escrever praticamente nenhum código, apenas criar um arquivo JSON com os dados desejados e executar o comando, *json-server –watch nome-arquivo.json*. Contudo, é possível escrever códigos para customizar as requisições.

3.2.2 Insominia REST

Insomnia é um cliente API REST poderoso com gerenciamento de *cookies*, variáveis de ambiente, geração de código e autenticação disponível para Mac, Windows e Linux. Além disso, Insomnia REST *Client* é uma ferramenta na categoria API *Tools* de uma pilha de tecnologia (STACKSHARE, 2016).

3.3 Banco de dados Relacional

3.3.1 PostgreSQL

PostgreSQL é um poderoso sistema de banco de dados relacional de código aberto que usa e estende a linguagem SQL combinada com muitos recursos que armazenam e escalam com segurança as cargas de trabalho de dados mais complicadas. As origens do PostgreSQL remontam a 1986 como parte do projeto POSTGRES na Universidade da Califórnia em Berkeley e tem mais de 30 anos de desenvolvimento ativo na plataforma central (POSTGRESQL, 2021).

Além disso, PostgreSQL ganhou uma forte reputação por sua arquitetura, confiabilidade, integridade de dados, conjunto de recursos robustos, extensibilidade e dedicação da comunidade de código aberto por trás do *software* para fornecer soluções inovadoras e de alto desempenho de maneira consistente. O PostgreSQL é executado em todos os principais sistemas operacionais, é compatível com ACID desde 2001 e possui complementos poderosos, como o popular extensor de banco de dados geoespacial PostGIS. Não é nenhuma surpresa que o PostgreSQL se tornou o banco de dados relacional de código aberto preferido de muitas pessoas e organizações (POSTGRESQL, 2021).

Diante disso, o PostgreSQL foi usado no presente projeto, já que se adequa ao padrão SQL onde tal conformidade não confronta os recursos tradicionais ou possa levar a decisões arquitetônicas inadequadas. Muitos recursos exigidos pelo padrão SQL são suportados, embora às vezes com sintaxe ou função ligeiramente diferente. Outros movimentos em direção à conformidade podem ser esperados ao longo do tempo. A partir do lançamento da versão 13 em setembro de 2020, o PostgreSQL está em conformidade com pelo menos 170 dos 179 recursos obrigatórios para os padrões do SQL 2016 *Core*. Assim, até o momento em que essa monografia foi escrita, nenhum banco de dados relacional, além do PostgreSQL estava em grande conformidade com o padrão SQL (POSTGRESQL, 2021), o que justificou a escolha desse banco de dados. Além disso, foi o banco utilizado nas

disciplinas cursadas durante o bacharelado e oferece diversos recursos atualizados sem custo para utilizá-lo.

3.3.2 Git e GitHub

Para facilitar o armazenamento do código do *back-end*, foi utilizado a ferramenta de versionamento de código chamada GitHub, que auxiliou no controle de versões criadas durante todo o processo de construção.

GitHub é um *site* que serve como plataforma de hospedagem para códigos diversos que mantém a sua integridade, além disso possibilita ao usuário controlar versões e adicionar colaboradores, guardando um histórico de alteração indicando o que foi alterado e por quem essa alteração foi realizada. A base do GitHub é o Git, por isso é possível que qualquer pessoa que tenha permissão, acesse o código de onde estiver (GITHUB, 2021).

O Git foi criado por Linus Torvalds em 2005 e é um projeto *opensource*, construído para facilitar o versionamento e compartilhamento de código entre os desenvolvedores. Git é um exemplo de DVCS (Distributed Version Control System) (ATLASSIAN, 2021). Antes de sua criação era comum criar vários arquivos com nomes diferentes para controlar a versão desses códigos. Assim, o Git inovou trazendo uma ferramenta que guarda os históricos das versões do *software*, o que possibilita separar cada uma em um repositório diferente com todas as alterações posteriores e fazer comparações entre os códigos se necessário.

3.3.3 Yarn

Yarn é um gerenciador de pacotes que serve como gerente de projeto. Essa ferramenta paraleliza as operações para maximizar a utilização de recursos, a fim de minimizar o tempo de instalação, além de armazenar em *cache* cada pacote que baixa para nunca precisar baixá-los uma segunda vez (YARN, 2021). Ademais, ele utiliza somas de verificação para garantir a integridade de cada pacote antes que seu código seja executado. Além disso, o Yarn usa um formato de arquivo de bloqueio detalhado, o qual é mais preciso e possui um algoritmo determinístico para instalações. A proposta do Yarn é garantir que um *software* que esteja instalado em um sistema funcione da mesma forma em qualquer outro. Sendo assim, devido as suas qualidades o Yarn foi escolhido como o gerenciador do projeto, já que possui maior desempenho do que o *npm*, gerenciador padrão do Node.js.

3.3.4 Node.js

Desenvolvido por Ryan Dahl, o Node.js é um ambiente plataforma cruzada de código aberto para o desenvolvimento de aplicativos, do lado do servidor os aplicativos são escritos em JavaScript e podem ser executados no tempo de execução do Node.js, compatível com diversos sistemas como: OS X, Microsoft Windows e Linux (TUTORIALSPOINT, 2020).

Além disso, é uma plataforma construída no tempo de execução do JavaScript do Chrome (motor V8) para construir facilmente aplicativos da *web* rápidos e escaláveis (NODE.JS, 2021). O Node.js usa um modelo orientado por eventos que o torna leve e eficiente, perfeito para aplicativos em tempo real com grande volume de dados que são executados em dispositivos distribuídos. Node.js também fornece uma rica biblioteca de vários módulos

JavaScript que simplifica o desenvolvimento de aplicativos da *web* usando Node.js em grande escala.

Assim, o Node.js foi escolhido para o presente projeto por ser uma ferramenta que utiliza o JavaScript como linguagem de programação principal, o que faz com que toda a linguagem utilizada na aplicação seja unificada tanto no *back-end* quanto nos *front-ends*, tendo a manutenção de ambas as partes facilitada.

3.3.5 JavaScript

JavaScript é uma linguagem de programação leve, interpretada ou compilada *just-in-time* com funções de primeira classe. Embora seja mais conhecida como a linguagem de *script* para páginas da Web, muitos ambientes sem navegador também a usam, como Node.js, Apache CouchDB e Adobe Acrobat (MOZILLA, 2021). Assim, JavaScript é uma linguagem dinâmica baseada em protótipo, multi-paradigma, *thread* único, que suporta estilos orientados a objetos, imperativos e declarativos (por exemplo, programação funcional). Diante disso, foi escolhido devido a dependência do TypeScript com essa linguagem e por ser de extrema utilização tanto no *back-end* quanto nos *front-ends.*

3.3.6 TypeScript

O TypeScript mantém uma relação incomum com o JavaScript. O TypeScript oferece todos os recursos do JavaScript e uma camada adicional sobre ele: o sistema de tipos do TypeScript. Por exemplo, JavaScript fornece primitivos de linguagem como *string* e número, mas não verifica se os atribuiu de forma consistente, TypeScript sim (TYPESCRIPTLANG, 2021). Isso significa que seu código JavaScript de trabalho existente também é código TypeScript. O principal benefício do TypeScript é que ele pode destacar comportamentos inesperados em seu código, reduzindo a chance de *bugs*, o que justifica a escolha dessa linguagem para o desenvolvimento da aplicação.

3.3.7 Docker e Docker Compose

Docker é uma plataforma *open source* aberta para desenvolvimento, envio e execução de aplicativos que permite separar aplicativos de sua infraestrutura para que possa entregar o *software* rapidamente. Com essa plataforma, pode-se gerenciar sua infraestrutura da mesma forma que gerencia os aplicativos, além de poder aproveitar das metodologias do Docker para envio, teste e implantação de código rapidamente, o que reduz significativamente o atraso entre escrever o código e executá-lo na produção (DOCKER, 2017a).

Ademais, o Docker oferece a capacidade de empacotar e executar aplicativos em um ambiente vagamente isolado, denominado contêiner. Diante do isolamento e da segurança, é permitido executar vários contêineres simultaneamente em um determinado servidor. Os contêineres são leves e contém tudo o que é necessário para executar o aplicativo, portanto, não precisa depender do que está instalado atualmente no *host.* Por fim, pode-se compartilhar contêineres facilmente enquanto trabalha com a certificação de que todos com quem compartilha recebam o mesmo contêiner e que funcione da mesma maneira (DOCKER, 2017a).

Docker Compose é uma ferramenta para definir e executar contêineres do Docker. Utiliza-se um arquivo YAML para configurar os serviços e deixá-los aptos para execução.

Dado isso, com um único comando se cria e inicia todos os serviços configurados. O método que o Docker Compose oferece funciona em todos os ambientes como em produção, teste, desenvolvimento e também é possível utilizá-lo em fluxos de trabalho de CI (DOCKER, 2017b). Para isso, é preciso seguir um processo com três etapas:

1. Definir o ambiente do aplicativo com um Dockerfile para que possa ser reproduzido em qualquer lugar;

2. Definir os serviços que compõem seu aplicativo em *docker-compose.yml* para que possam ser executados juntos em um ambiente isolado;

3. Executar *docker-compose up* e o comando Docker *compose*, desse modo iniciará e executará todo os serviços da aplicação. Como alternativa, pode-se executar *docker-compose up* usando o binário *docker-compose*.

Dessa forma, o Docker e o Docker Compose foram usados na aplicação Severino para evitar o *download* de muitas ferramentas, visto que o ambiente fornecido pelo Docker contém os serviços essenciais para o funcionamento do *back-end*, como por exemplo, o servidor do banco de dados estava sendo executado pelo Docker e configurado com Docker Compose.

3.3.8 TypeORM

O TypeORM é um *framework Object Relational Mapping* (ORM) em execução em um servidor Node.js e escrito em TypeScript. *Object* se refere ao domínio e modelo em sua aplicação. *Relational* se refere ao relacionamento entre as tabelas no Sistema de Gerenciamento de Banco de Dados Relacional (por exemplo, Oracle, MySQL, MS-SQL Server, PostgreSQL, SAP, Hana, WebSQL etc.) e *Mapping* se refere ao ato de criar uma ponte entre o modelo e tabelas.

O *framework* suporta múltiplos bancos de dados como os citados acima e é um ORM fácil de usar para criar novos aplicativos que se conectam a bancos de dados. Além disso, suas funcionalidades são compostas por conceitos específicos de RDBMS (Sistema de Gerenciamento de Banco de Dados Relacional), o que permite criar rapidamente novos projetos e microsserviços. Vale ressaltar, que o TypeORM foi inspirado em ferramentas semelhantes de outras linguagens de programação como Hibernate, Doctrine, Entity *framework*, entre outros (TUTORIALSPOINT, 2020).

Ademais, o ORM é um tipo de ferramenta que mapeia entidades com tabelas de banco de dados, fornece processo de desenvolvimento simplificado, o que automatiza a conversão de objetos para tabelas e tabelas para objetos, além de que uma vez escrito o modelo de dados em algum lugar, fica mais fácil atualizar, manter e reutilizar o código.

Em suma, como o modelo é pouco ligado ao resto do aplicativo, pode-se alterá-lo sem nenhuma dependência rígida com outra parte do aplicativo e pode ser facilmente usado em qualquer lugar dentro do micro-serviço. Vale evidenciar, ainda, que o TypeORM é muito flexível, abstrai o sistema de banco de dados da aplicação e permite tirar proveito do uso do conceito OOPS (Programação orientada a objetos) (TUTORIALSPOINT, 2020).

Os principais recursos do TypeORM são:

- Criar esquemas de tabela de banco de dados automaticamente com base nos seus modelos;

- Inserir, atualizar e excluir facilmente objetos no banco de dados.
- Criar mapeamentos (um para um, um para muitos e muitos para muitos) entre as tabelas;
- Fornecer comandos CLI simples.

Benefícios do TypeORM:

- Aplicações de alta qualidade e fracamente acopladas;
- Aplicativos escaláveis;
- Integre-se facilmente com outros módulos;
- Se encaixa perfeitamente em qualquer arquitetura, de aplicativos pequenos a corporativos.

Assim, os pontos principais que levaram para a escolha da utilização do TypeORM, no presente projeto, foram os recursos e os benefícios que o *framework* oferece.

3.3.9 Express

Express é uma estrutura de aplicativo Node.js mínima e flexível que fornece um conjunto robusto de recursos para desenvolver aplicativos móveis e *web*. Além disso, ele agiliza o desenvolvimento de aplicações baseadas em Node.js (TUTORIALSPOINT, 2019). A seguir estão alguns dos principais recursos da estrutura Express:

- Permite configurar *middlewares* para responder a solicitações HTTP.
- Define uma tabela de roteamento que é usada para realizar diferentes ações com base no método HTTP e URL.
- Permite renderizar páginas HTML dinamicamente com base na passagem de argumentos para modelos.

Dessa forma, o Express foi escolhido para ser utilizado no desenvolvimento do projeto por ser uma ferramenta simples de aprender e utilizar, além de fornecer todos os recursos que é preciso para a construção de uma aplicação.

3.3.10 JSON Web Token

JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma forma compacta e independente para transmitir informações com segurança entre as partes como um objeto JSON. Essas informações podem ser verificadas e confiáveis, pois estão assinadas digitalmente. Ademais, os JWTs podem ser assinados usando criptografia (com o algoritmo HMAC) ou um par de chaves pública e privada usando RSA ou ECDSA (JWT.IO, 2018).

Assim, além da segurança que o JWT fornece, a ferramenta foi escolhida para o projeto por ser o padrão mais utilizado no mercado de aplicações que faz a comunicação autenticada.

3.3.11 Faker

Faker é uma ferramenta que serve para gerar grandes quantidades de dados falsos no navegador e no Node.js (MARAK, 2015). Diante disso, no presente projeto alguns dados foram guardados no banco para serem utilizados como exemplos para algumas funcionalidades da aplicação.

Assim, o Faker foi usado no projeto para se ter uma prévia das funcionalidades que as páginas *web* e o aplicativo *mobile* possuem, afim de testar a usabilidade dos serviços e apresentação dos *layouts* aos usuários.

3.3.12 TSNodeDev

TSNodeDev é uma ferramenta usada para reiniciar o processo do nó de destino quando qualquer um dos arquivos do projeto é alterado (como *node-dev* padrão), porém compartilha o processo de compilação do Typecript entre as reinicializações, o que aumenta significativamente a velocidade de re-inicialização em comparação com as variações, pois não há necessidade de instanciar a compilação todas as vezes (WCLR, 2015).

Diante disso, o TSNodeDev foi escolhido no projeto para economizar tempo ao fazer alterações em arquivos, já que sem o uso da ferramenta teria que interromper a execução do servidor e reiniciar manualmente, assim, com o TSNodeDev basta salvar o arquivo e continuar a codificação do código.

3.3.13 CORS

CORS ou Compartilhamento de Recursos de Origem Cruzada é um mecanismo baseado em cabeçalho HTTP que permite a um servidor indicar qualquer outra origem (domínio, esquema ou porta) além da sua própria, a partir da qual um navegador deve permitir o carregamento de recursos (DEVELOPER.MOZILLA.ORG, 2021). O CORS também conta com um mecanismo pelo qual os navegadores fazem uma solicitação de “comprovação” ao servidor hospedeiro do recurso de origem cruzada, a fim de verificar se o servidor permitirá a solicitação real. Nesse "*preflight*", o navegador envia cabeçalhos que indicam o método HTTP e os cabeçalhos que serão usados na solicitação real.

Dessa forma, foi feita a configuração do CORS na aplicação devido a sua obrigatoriedade, pois sem esse mecanismo não é possível fazer a conexão das funcionalidades do *back-end* com o aplicativo *mobile* e os sítios da *web*.

3.3.14 BcryptJS

Bcrypt é uma biblioteca criptográfica que fornece uma maneira segura de armazenar senhas em banco de dados, independentemente da linguagem de *back-end* dos aplicativos (JAVASCRIPT.PLAINENGLISH.IO, 2018).

Dado isso, essa biblioteca foi utilizada no projeto para fazer a criptografia das senhas dos usuários ao guardá-las no banco de dados.

3.3.15 Multer

O Multer é um *middleware* Node.js para lidar com *multipart / form-data* usado principalmente para fazer *upload* de arquivos (EXPRESSJS, 2020).

Assim, o Multer foi utilizado para guardar os *paths* das imagens no banco de dados ao invés de armazená-las com o formato de *base64*, o comum em muitas aplicações. Dessa forma, as imagens fiquem são salvas do lado do servidor, o que aumenta o desempenho da aplicação.

3.4 Arquitetura

3.4.1 API Rest

Criado por Dr. Roy Fielding em sua dissertação de doutorado de 2000, o *design* REST ou RESTful API (Representational State Transfer) é notável por sua incrível camada de flexibilidade e foi projetado para tirar proveito dos protocolos existentes. Embora o REST possa ser usado em quase todos os protocolos, geralmente tira proveito do HTTP quando usado para APIs da *web*, o que significa que os desenvolvedores não precisam instalar bibliotecas ou *software* adicional para aproveitar as vantagens de um *design* de API REST. Além disso, como os dados não estão vinculados a métodos e recursos, o REST tem a capacidade de lidar com vários tipos de chamadas, retornar diferentes formatos de dados e até mesmo mudar estruturalmente com a implementação correta da hipermídia.

A liberdade e flexibilidade inerentes ao *design* da API REST permite construir uma API que atenda às necessidades de um desenvolvedor e, ao mesmo tempo, atenda às necessidades de clientes diversos. Ademais, o REST não é restrito ao XML, mas pode retornar JSON, YAML ou qualquer outro formato a depender do que o cliente solicita. E, ao contrário do RPC, os usuários não precisam saber os nomes dos procedimentos ou parâmetros específicos em uma ordem específica (MULESOFT, 2020).

No entanto, existem desvantagens no *design* da API REST como a possibilidade de perder a capacidade de manter o estado em REST, como dentro das sessões, e pode ser mais difícil para desenvolvedores mais novos usarem. Também é importante entender o que torna uma API REST RESTful e por que essas restrições existem antes de construir sua API.

Embora a maioria das APIs afirme ser RESTful, elas ficam aquém dos requisitos e restrições declarados pelo Dr. Fielding. Existem seis restrições principais, as quais serão apresentadas abaixo, para o *design* da API REST que se deve conhecer ao decidir se esse é o tipo de API correto para o seu projeto.

1. Cliente Servidor: A restrição cliente-servidor trabalha com o conceito de que o cliente e o servidor devem ser separados um do outro e ter permissão para evoluir de forma individual e independente. Em outras palavras, deve-se ser capaz de fazer alterações no aplicativo móvel sem afetar a estrutura de dados ou o *design* do banco de dados no servidor. Ao mesmo tempo, deve-se ser capaz de modificar o banco de dados ou fazer alterações em meu aplicativo de servidor sem impactar o cliente móvel, o que cria uma separação de interesses e permite que cada aplicativo cresça e dimensione de forma independente um do outro (MULESOFT, 2020).

2. Sem estado: As APIs REST são sem estado, o que significa que as chamadas podem ser feitas independentemente umas das outras e cada chamada contém todos os dados necessários para ser concluída com êxito. Uma API REST não deve depender de dados armazenados no servidor ou nas sessões para determinar o que fazer com uma

chamada, mas apenas nos dados fornecidos na própria chamada. As informações de identificação não estão sendo armazenadas no servidor ao fazer chamadas. Em vez disso, cada chamada tem os dados necessários em si, como a chave da API, *token* de acesso, ID do usuário, entre outras. Isso também ajuda a aumentar a confiabilidade da API por ter todos os dados necessários para fazer a chamada, em vez de depender de uma série de chamadas com estado de servidor para criar um objeto, o que pode resultar em falhas parciais. Em vez disso, para reduzir os requisitos de memória e manter seu aplicativo o mais escalonável possível, uma API RESTful requer que qualquer estado seja armazenado no cliente, não no servidor.

3. Cache: Como uma API sem estado pode aumentar a sobrecarga de solicitação ao lidar com grandes cargas de chamadas de entrada e saída, uma API REST deve ser projetada para incentivar o armazenamento de dados armazenáveis em *cache*. Isso significa que quando os dados podem ser armazenados em *cache*, a resposta deve indicar que os dados podem ser armazenados até um certo tempo (expira em) ou, nos casos em que os dados precisam ser em tempo real, que a resposta não deve ser armazenada em *cache* pelo cliente. Ao habilitar essa restrição crítica não apenas reduzirá muito o número de interações com a API, como reduzirá o uso do servidor interno. Ademais, também fornecerá aos usuários da API as ferramentas necessárias para fornecer os aplicativos mais rápidos e eficientes possíveis. Além disso, é importante lembrar de que o armazenamento em *cache* é feito no lado do cliente e embora possa armazenar em *cache* alguns dados em sua arquitetura para executar o desempenho geral, a intenção é instruir o cliente sobre como ele deve proceder e se o cliente pode ou não armazenar os dados temporariamente.

4. Interface Uniforme: A chave para o desacoplamento do cliente do servidor é ter uma interface uniforme que permite a evolução independente do aplicativo sem ter os serviços, modelos ou ações do aplicativo fortemente acoplados à própria camada da API. A interface uniforme permite que o cliente converse com o servidor em um único idioma, independente do *back-end* arquitetônico de qualquer um. Essa interface deve fornecer um meio de comunicação padronizado e imutável entre o cliente e o servidor, como o uso de HTTP com recursos de URI, CRUD (Criar, Ler, Atualizar, Excluir) e JSON.

5. Sistema em camadas: Como o nome indica, um sistema em camadas é um sistema composto de camadas, com cada camada tendo uma funcionalidade e responsabilidade específicas. Se pensarmos em um *framework* de Model View Controller (MVC), cada camada tem suas próprias responsabilidades, com os modelos compreendendo como os dados devem ser formados, o controlador se concentrando nas ações de entrada e a visualização se concentrando na saída (MULESOFT, 2020). Cada camada é separada, mas também interage com a outra. No *design* da API REST, o mesmo princípio é válido, com diferentes camadas da arquitetura trabalhando juntas para construir uma hierarquia que ajuda a criar um aplicativo mais escalável e modular. Um sistema em camadas também permite encapsular sistemas legados e mover funcionalidades menos acessadas para um intermediário compartilhado, ao mesmo tempo que protege os componentes mais modernos e comumente usados. Além disso, o sistema em camadas oferece a liberdade de mover sistemas para dentro e para fora de sua arquitetura à medida que as tecnologias e serviços

evoluem, aumentando a flexibilidade e a longevidade, contanto que mantenha os diferentes módulos o mais fracamente acoplados possível. Existem benefícios de segurança substanciais em ter um sistema em camadas, uma vez que permite interromper ataques na camada de *proxy* ou dentro de outras camadas, o que evita chegar à arquitetura do servidor real. Ao utilizar um sistema em camadas com um *proxy* ou criar um único ponto de acesso, pode-se manter os aspectos críticos e mais vulneráveis de sua arquitetura atrás de um *firewall*, o que evita a interação direta com eles pelo cliente. Lembre-se de que a segurança não se baseia em uma solução única de "parar tudo", mas em ter várias camadas com o entendimento de que certas verificações de segurança podem falhar ou ser ignoradas. Dessa forma, quanto mais segurança for capaz de implementar em um sistema, maior será a probabilidade de evitar ataques prejudiciais.

6. Código sob demanda: Talvez a menos conhecida das seis restrições, e a única restrição opcional, o código sob demanda permite que o código ou miniaplicativos sejam transmitidos por meio da API para uso no aplicativo. Em essência, ele cria um aplicativo inteligente que não depende mais exclusivamente de sua própria estrutura de código. No entanto, talvez por estar à frente de seu tempo, o código sob demanda tem lutado para ser adotado, já que as APIs da *web* são consumidas em vários idiomas e a transmissão de código levanta questões e preocupações de segurança. Juntas, essas restrições constituem a teoria da Transferência de Estado Representacional, ou REST. Ao olhar para trás, pode-se ver como cada restrição sucessiva é construída sobre a anterior, eventualmente criando uma interface de programa de aplicativo bastante complexa, mas poderosa e flexível. Mas o mais importante é que essas restrições constituem um *design* que opera de maneira semelhante à forma como acessamos as páginas em nossos navegadores na *World Wide Web*. Ele cria uma API que não é ditada por sua arquitetura, mas pelas representações que ela retorna, e uma API que - embora arquitetonicamente sem estado - depende da representação para ditar o estado do aplicativo (MULESOFT, 2020).

3.4.2 MVC

MVC é um padrão de *design* que promove a separação de interesses onde três interesses participantes são:

1. O ***Model*** (modelo) é uma estrutura de dados que contém dados de negócios e são transferidos de uma camada para outra;

2. A ***View*** (visão) que é responsável por mostrar os dados presentes no aplicativo ou pensar neles como uma estrutura de dados (totalmente desacoplada do modelo) usada exclusivamente para fins de apresentação (não é necessária uma saída de apresentação em si). ver modelo no diagrama abaixo;

3. O ***Controller*** (controlador) atua como um mediador e é responsável por aceitar uma solicitação de um usuário, modificar um modelo (se necessário) e convertê-lo na visualização.

Assim, o MVC pode ser utilizado como um padrão, em que pode existir completamente no *back-end* ou completamente no *front-end* ou em sua forma comum de *back-end* e *front-end* combinados.

Toda a ideia por trás do padrão MVC é uma separação muito clara entre objetos de domínio que representam entidades do mundo real e a estrutura de dados da camada de apresentação. Os objetos de domínio devem ser completamente independentes e também funcionar sem uma visão (representação de dados). Outra maneira de pensar sobre isso é, no MVC, as visualizações de contexto são isoladas do modelo, o que permite usar o mesmo modelo com diferentes vistas (STACKOVERFLOW, 2019).

Diante disso, o Spring MVC é um bom exemplo de uma estrutura MVC de *back-end*. No diagrama abaixo 5 descreve como todos os três componentes existem apenas no lado do servidor (dentro do contêiner do aplicativo) (SPRING, 2018).

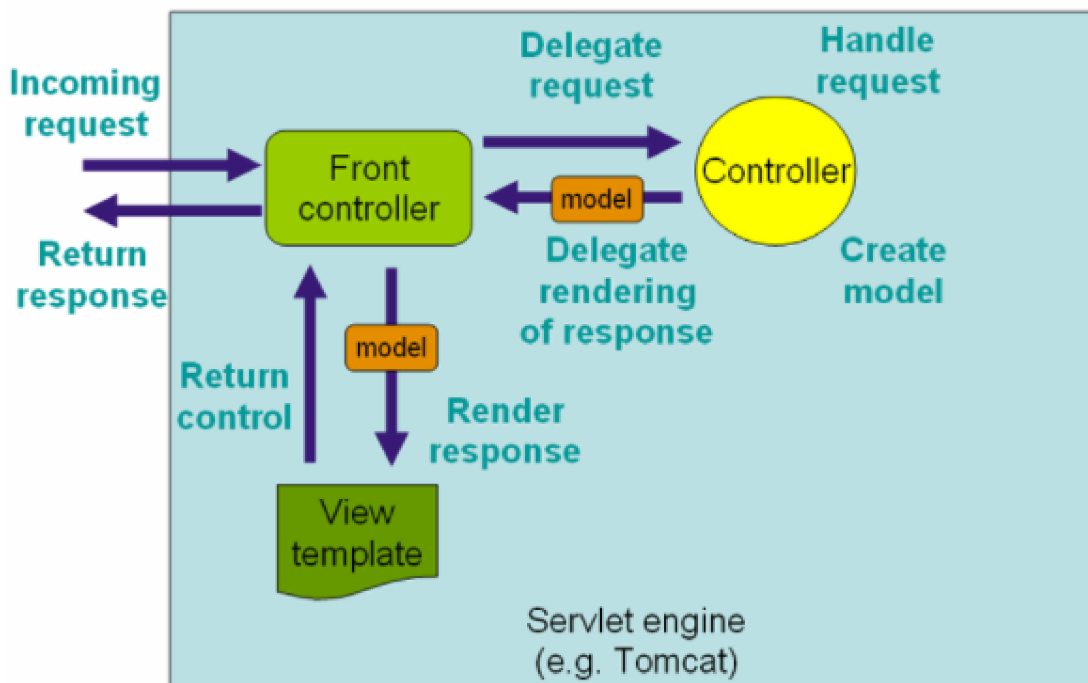

Figura 5 – Representação da arquitetura MVC.

(SPRING, 2018)

Dessa forma, durante a criação da API REST da aplicação Severino, foi escolhida a arquitetura MVC por ser o padrão mais utilizado no desenvolvimento das aplicações, além de ser apresentada em disciplinas da graduação de Ciência da Computação. Vale ressaltar, que a *View* ficou separada do *back-end*, estando presente apenas nas aplicações *web* e *mobile*.

3.4.3 Arquitetura relacional

RDBMS significa *Relational Database Management System* e implementa SQL. No cenário do mundo real, as pessoas usam o RDBMS para coletar informações e processá-las para fornecer serviço. A figura 6 representa a arquitetura RDBMS (GEEKS, 2020).

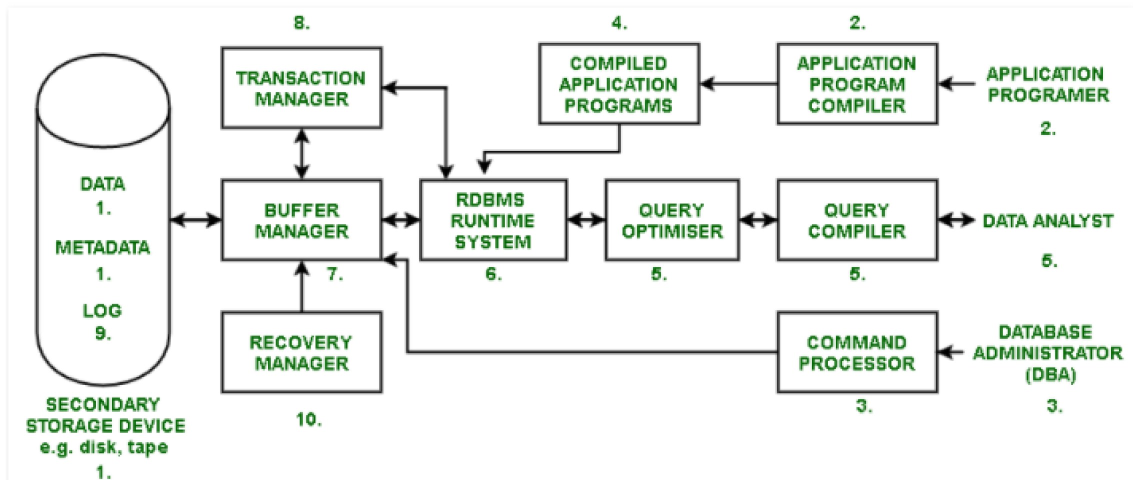

Figura 6 – Representação da arquitetura relacional.

(GEEKS, 2020)

Cada termo no diagrama é explicado abaixo no número do ponto associado ao termo respectivamente:

1. Todos os dados (metadados) e *logs* são armazenados nos dispositivos de armazenamento secundário (SSD), como discos e fitas. Os programas que são usados para fazer as tarefas diárias de uma empresa são chamados de programas aplicativos, os quais fornecem a funcionalidade para as operações diárias da empresa. Eles são escritos em linguagens de alto nível (HLL) como Java, C, dentre outras, que junto com o SQL, são usados para se comunicar com os bancos de dados.

2. O RDBMS possui um compilador que converte os comandos SQL em linguagem de baixo nível, processa e armazena no dispositivo de armazenamento secundário.

3. É função do Administrador do Banco de Dados (DBA) configurar a estrutura do banco de dados usando o processador de comando. O DDL significa Linguagem de Definição de Dados e é usado pelo DBA para criar ou eliminar tabelas, adicionar colunas, etc. O DBA também usa outros comandos que são usados para definir restrições e controles de acesso.

4. Os programadores de aplicativos compilam os aplicativos usando um compilador e criam arquivos executáveis (programas aplicativos compilados) e a seguir, armazenam os dados no dispositivo de armazenamento secundário.

5. A tarefa do Analista de Dados é usar o Compilador de Consultas e o Otimizador de Consultas (usa propriedades relacionais para executar consultas) para manipular os dados no banco de dados (GEEKS, 2020).

6. O RDBMS *Run Time System* executa as consultas compiladas e os programas aplicativos e também interage com o gerenciador de transações e o gerenciador de *buffer*.

7. O *Buffer Manager* armazena temporariamente os dados do banco de dados na memória principal e usa o algoritmo de paginação para que as operações possam ser realizadas mais rapidamente e o espaço em disco possa ser gerenciado.

8. O Gerenciador de transações lida com o princípio de realizar uma tarefa completamente ou não realizá-la (propriedade atomicidade). Por exemplo. Suponha que uma pessoa chamada João queira enviar dinheiro para sua irmã. Ele envia o dinheiro e o sistema falha no meio. Em nenhum caso deve acontecer que ele tenha enviado dinheiro, mas sua irmã não o tenha recebido. Isso é tratado pelo gerenciador de transações. O gerente de transações iria devolver o dinheiro para João ou transferi-lo para sua irmã (GEEKS, 2020).

9. *Log* é um sistema que registra as informações sobre todas as transações, de forma que sempre que ocorrer uma falha no sistema (falha no disco, desligamento do sistema por falta de energia, etc), as transações parciais podem ser desfeitas.

10. O *Recovery Manager* assume o controle do sistema para que ele atinja um estado estável após a falha. O *Recovery Manager* leva em consideração os arquivos de *log* e desfaz as transações parciais e reflete a transação completa no banco de dados (GEEKS, 2020).

Assim, essa arquitetura foi utilizada, pois a ferramenta de banco de dados PostgreSQL a aplica implicitamente.

3.4.4 Principais Características do desenvolvimento *back-end* da aplicação Severino

1. O padrão de nomenclatura adotado para o código foi o *Camel Case* (TECHTERMS, 2020), em que a primeira letra de cada palavra de uma palavra composta é maiúscula, o que não se aplica para a primeira palavra. Por exemplo, *myOneMethod* é mais fácil de ler do que *myonemethod*;

2. O padrão de nomenclatura adotado para o banco de dados foi o *Snake Case*, o qual consiste em letras minúsculas, pontuação e espaços são substituídos por sublinhado simples (_). Exemplo, *myOneMethod* se torna *my_one_method*;

3. As *Queries* foram otimizadas usando TypeORM;

4. A tipagem de variáveis foi feita com Typescript;

5. A API que fornece os dados é a RESTful e a que fornece as respostas é o JSON;

6. Idioma do código: Inglês.

3.4.5 Estrutura de pastas e componentes

```
root
├── src
|   ├── config                (Configurações gerais)
|   ├── controllers           (Todos os Controllers)
|   ├── database              (Configurações do banco de dados)
|   |   ├── migrations        (Todas as migrations)
|   |   ├── fake-data         (Dados fake para as seeds)
|   |   └── seeds             (Todas as seeds)
|   ├── erros                 (Arquivos para tratar erros)
|   ├── models                (Todas as entidades)
|   ├── routes                (Rotas)
|   ├── services              (Serviços para os controllers)
|   └── server.ts             (Entrypoint da aplicação)
├── .env                      (Variáveis de ambiente)
├── .docker-compose.yml       (Configurações do docker para o banco de dados)
├── .ormconfig.json           (Configurações do TypeORM)
└── ...                       (Outros arquivos de configuração)
```

Capítulo 4

Desenvolvimento

Esse capítulo é responsável por apresentar o entendimento do projeto, prototipação, levantamento dos requisitos, modelagem do banco de dados, implementação e, por fim, a apresentação do serviço *back-end*.

4.1 Levantamento de requisitos

O início do desenvolvimento do presente projeto se deu com o levantamento de requisitos, o qual foi feito em acordo com os outros desenvolvedores do projeto para definir as regras de negócio e quais dados as APIs precisariam receber e retornar.

Dado isso, foi iniciado o mapeamento das atividades essenciais a serem acrescentadas no fluxo logístico da aplicação, sendo elas também os principais requisitos. Diante disso, para melhor compreensão dos usuários da aplicação Severino, ela foi dividida em duas partes, uma em que os clientes apenas visualizassem as informações que procuram e outra exclusiva para profissionais que desejam prestar serviços, sendo que podem realizar o cadastro, editar seu perfil para que os clientes possam visualizar suas informações e descrever os serviços que presta.

Logo abaixo serão levantados os requisitos para a construção da API REST das duas partes da aplicação já citada acima.

Observação: Os tópicos a seguir são requisições HTTP e todos os dados que serão alocados no banco de dados são resgatados das aplicações *front-ends*.

1. **Cadastro do cliente empregador:** É a requisição que recebe o e-mail, nome e senha do cliente e guarda as informações no banco de dados, já que serão utilizadas futuramente para o cliente realizar o *login* nas aplicações.

2. ***Login* do cliente empregador:** É a requisição que resgata o e-mail e a senha do cliente para fazer a autenticação do usuário nas aplicações *front-ends*, sendo que o serviço checa no banco de dados se as informações resgatadas existem e, então, autoriza o acesso ou não à *Home*.

3. ***Home* do cliente empregador:** Nessa página tem uma requisição referente a barra de pesquisa, na qual o usuário pode buscar por diversos serviços que a aplicação fornece, tais como, limpeza, aulas, jardinagem, encanadores, dentre outros.

Além disso, possui outra requisição, a qual transfere o usuário para a área de serviços gerais e por último existem *cards* com serviços já especificados, em que ao clicar no *card* leva o usuário para uma lista com exemplares daquele tópico.

4. **Área de serviços gerais:** É a página que possui uma requisição de busca de serviços, assim como na *Home*, sendo que há também outra requisição que lista alguns serviços não especificados e, por fim, um mecanismo de paginação.

5. **Área de contratação de serviço:** É a página com maior infraestrutura e regras de negócio da aplicação, sendo que nela contem requisições de filtros, os quais podem ser por avaliações, localidade e serviços em específico. Além disso, possui outra requisição para a listagem de *cards* com as informações dos prestadores de serviços junto com as requisições para as funcionalidades dos *cards* como botões de *like* e favoritar.

6. ***Cards* com informações dos trabalhadores:** Há uma requisição responsável por trazer informações, tais como a imagem do prestador de serviço, número de telefone, nome do trabalhador, uma breve descrição do seu perfil, além da sua avaliação de 0 a 5 estrelas, sendo que se o trabalhador não tiver nenhuma estrela é mostrado como um novo usuário. Além disso, tem outras duas requisições, as quais são o botão de favoritar e o botão que redireciona para o *WthatsApp* do prestador de serviço.

7. **Perfil do trabalhador:** É a página que contém as principais informações dos prestadores de serviços e nela estão as informações cruciais para a sua contratação por um cliente empregador. Abaixo será listado as requisições das funcionalidades que a página possui.

    a) Requisições para resgatar a foto do trabalhador do banco de dados, para dar função aos botões de ações, os quais são o botão para redirecionamento ao *WhatsApp*, o botão de favoritar, o botão que rola a página para seção de avaliação, o botão de avaliar, no qual abre o modal de avaliação e, por fim, o botão de denúncia para abrir o modal correspondente;

    b) Requisições para resgatar do banco de dados o nome do prestador de serviço, avaliação média, número de telefone, localização, serviços prestados e uma descrição escolhida;

    c) Requisição para resgatar as fotos do *carroussel* com legendas de seus trabalhos realizados, as quais serão escolhidas pelo trabalhador e outra requisição para mostrar o seu currículo com experiências profissionais, formação acadêmica e competências.

    d) Por fim, a requisição para resgatar a média de avaliações do prestador de serviço, bem como as avaliações classificadas como positivas, negativas ou ambas, junto com a paginação dos comentários.

8. **Cadastro do prestador de serviço:** Tem a requisição que recebe o e-mail, nome, sobrenome, celular, estado, cidade e senha, sendo que após receber os dados guarda as informações no banco de dados.

9. ***Login* do prestador de serviço:** Possui a requisição que recebe o e-mail e senha do trabalhador, além de checar se esses dados estão presentes na base de dados, caso existam, é gerado um *JSON Web Token* (JWT), o qual contém informações de autenticação e, assim, retorna o usuário para a página de *Login*.

10. ***Home* do prestador de serviço:** Página a qual não precisa de nenhuma requisição.

11. **Perfil do prestador de serviço:** Há a requisição que guarda no banco de dados as informações críticas do trabalhador, como seus dados gerais, fotos dos trabalhos realizados, suas experiências profissionais, formação acadêmica e suas competências, sendo que essa requisição é usada, também, de acordo com a necessidade do trabalhador para a atualização de seus dados.

12. **Troca de senha:** Possui a requisição na qual o trabalhador ou o cliente empregador coloca sua senha atual e a nova de sua escolha para a troca, sendo que após a confirmação é atualizada no banco de dados.

Após levantar todas as requisições necessárias para o funcionamento das aplicações *web* e *mobile*, foi feito a definição dos *stakeholders*, ou seja, o grupo de interesse que utilizarão o sistema. Dentro do contexto da aplicação Severino, há apenas dois tipos de usuários que se beneficiarão, sendo os usuários clientes empregadores que necessitam de um serviço a ser realizado e o usuário profissional que é contratado para a realização do serviço. Tais usuários foram separados em diferentes perfis, porém com os mesmos níveis de permissões, nas quais serão capazes apenas de criar a conta, editar o perfil e consultar as páginas para a contratação de um profissional. Assim, os usuários foram separados nos em dois tipos, os quais serão descritos abaixo.

1. **Usuário cliente empregador:** é o que busca na aplicação um serviço a ser realizado.

2. **Usuário prestador de serviço:** é aquele que realiza o serviço para o qual foi contratado.

Dessa forma, a aplicação garante o atendimento por igual à todos os usuários que a utilizar, sendo que os clientes empregadores visualizam os serviços que se encontram na plataforma e os perfis dos trabalhadores que os prestam. Além disso, os prestadores de serviços terão uma plataforma a parte focada para o cadastro como trabalhadores do Severino e para a edição do perfil, sendo que essa segunda plataforma leva todas as informações preenchidas pelos usuários prestadores de serviços para serem guardadas no banco de dados da plataforma e, assim, alimentar a página em que os clientes empregadores visualizarão as informações dos trabalhadores. Em suma, as figuras **??** e **??** representam o Diagrama de Caso de Uso dos usuários, em que é possível visualizar como funciona o fluxo de uso nas duas plataformas.

Como já descrito nos requisitos acima, foram definidas inúmeras requisições para que as páginas de *front-end web* e *mobile* funcionassem, dado isso foi criado o Diagrama de Caso de Uso para ficar mais compreensível a quem esteja lendo a presente monografia. Desse modo, foi descrito na tabela 1 os fluxos do diagrama que os usuários seguem para conseguirem realizar as atividades desejadas.

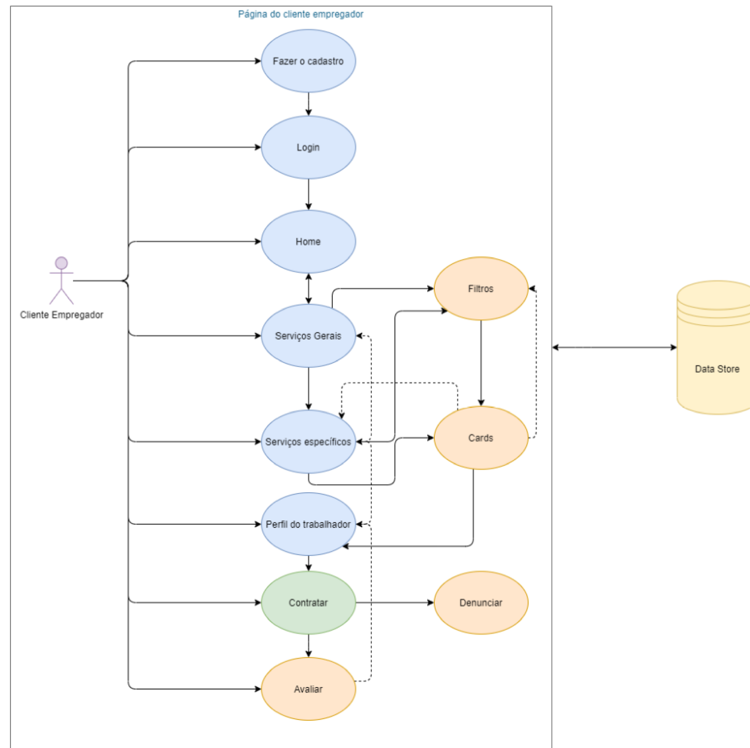

Figura 7 – Caso de uso para o usuário cliente empregador da aplicação Severino.

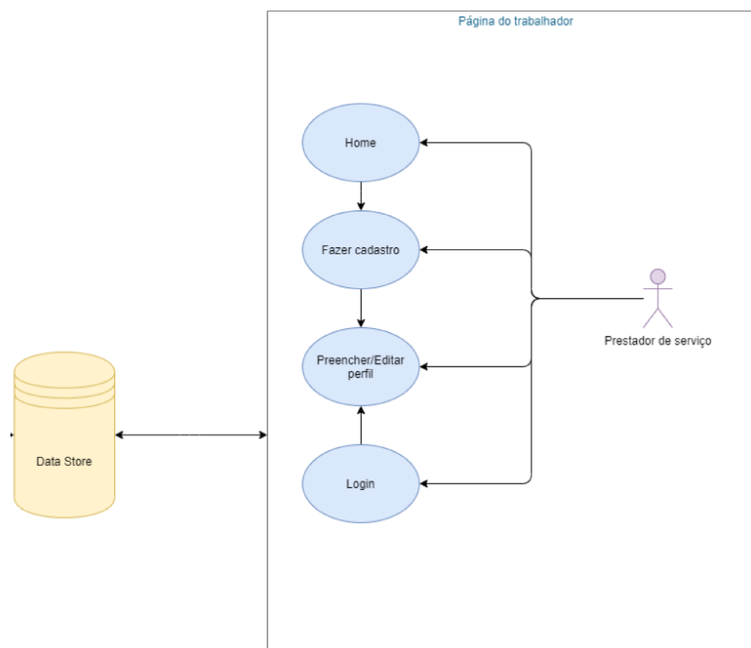

Figura 8 – Caso de uso para o usuário trabalhador da aplicação Severino.

Junto com as figuras 10 e 9, as quais representam os fluxogramas de atividades de ambos usuários para melhor ilustração do caso de uso e, também, para um melhor entendimento do leitor sobre a aplicação, nas figuras é possível visualizar os diferentes papéis que os usuários têm de acordo com seus perfis e interação com o sistema. Assim, esses fluxogramas representam as telas com seus respectivos serviços, sendo as transições que o usuário pode ou não seguir representadas pelas setas.

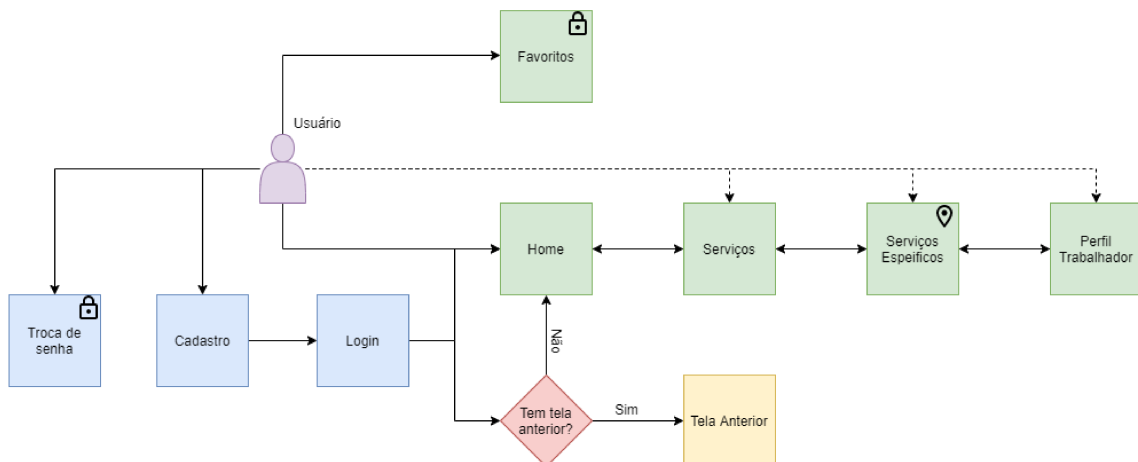

Figura 9 – Fluxograma explicativo do cliente empregador do usuário.

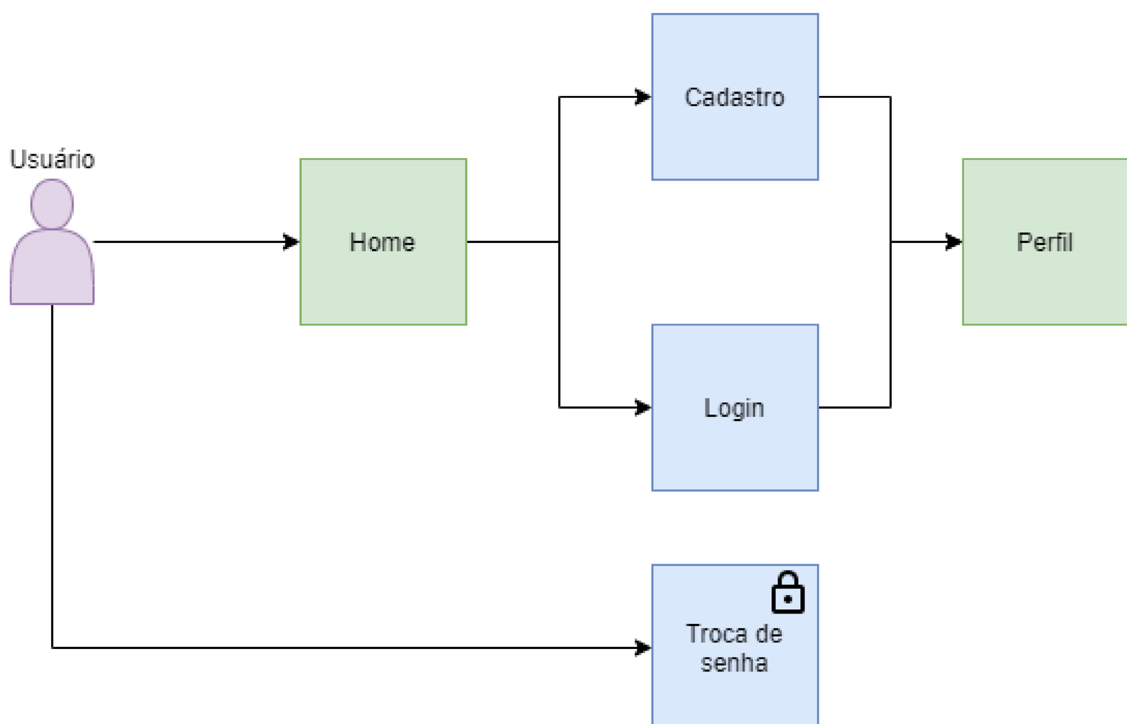

Figura 10 – Fluxograma explicativo do prestador de serviço do trabalhador.

Caso de uso: Cliente empregador	
Escopo	O sistema deve ser capaz de cadastrar o usuário, mostrar todos os serviços existentes na aplicação, além dos perfis dos profissionais.
Pré-condição	O usuário deve ter acesso à *internet*.
Garantia de Sucesso	Usuário deve ser capaz de se cadastrar e interagir com as ferramentas da aplicação.
Cenário de sucesso principal	
Fluxo principal: Cadastro e pesquisa de profissional	O usuário cliente empregador irá se cadastrar, fazer o *login* e após realizado o usuário será direcionado para a *home*, onde já se encontra alguns *cards* com serviços gerais, após isso clicará em algum dos *cards* ou, então, buscará pela barra de pesquisa. Além disso, após clicar em algum *card* ou fazer a busca pela barra de pesquisa, o usuário cliente empregador será direcionado para a página de serviços e, então, poderá aplicar os filtros e achar o serviço necessário. Dessa forma, após clicar no *card* de sua escolha irá para a página do perfil do prestador de serviço, a qual possui as informações críticas para a contratação do profissional.
Fluxo secundário: Contratação e avaliação do profissional	Após ter encontrado um profissional que o cliente empregador escolheu, ele irá contatar ou, então, clicará no botão de redirecionamento para o *WhatApp* do trabalhador para, talvez, contratá-lo. Após o profissional ter realizado o serviço, o cliente empregador poderá avaliá-lo pela aplicação e essa avaliação ficará salva no perfil do profissional para que novos clientes empregadores possam ver os *feedbacks*.

Tabela 1 – Tabela explicatória para os casos de uso dos usuários.

4.2 Modelagem do Banco de Dados

Após o levantamento de requisitos do sistema, a modelagem do banco de dados já está apta a ser produzida, sendo que nela é armazenada as informações do sistema e o os dados que o alimenta. A arquitetura do banco de dados foi definida e manipulada por meio do PostgreSQL e da expansão do VSCode, denominada PostgreSQLExplorer, a qual foi utilizada para a visualização dos dados. Além disso, as entidades são as tabelas referentes aos dados do cliente empregador, trabalhador, serviços, portfólio do trabalhador, experiência do trabalhador, avaliações dos clientes aos trabalhadores e trabalhadores favoritados pelo cliente, sendo que algumas dessas entidades possuem outras tabelas para armazenar informações secundárias, como serviços específicos e gerais. Na figura 11 se encontra o Diagrama Entidade-Relacionamento (DER) da aplicação.

Na tabela de cor violeta do diagrama abaixo temos dados que pertencem aos **usuários**, os quais foram divididos em dois tipos já citados anteriormente, sendo cliente empregador e prestador de serviço. Nessa tabela contém as seguintes informações:

- **id:** número de identificação do usuário.
- **e-mail:** e-mail;
- **password:** senha;
- **name:** nome;
- **created-at:** data da criação da conta;
- **updated-at:** data da última atualização da conta.

Nas tabelas de cor lilás temos os dados relacionados ao cliente empregador, sendo que a primeira tabela é a ***customer***, a qual serve apenas para fazer o relacionamento com as tabelas de avaliações (***reviews***) e profissionais favoritos (***customer-worker-favorite***), com o seguinte campo:

- **customer-id**: número de identificação.

Dando continuidade, tem-se a tabela ***customer-worker-favorite***, a qual guarda informações dos profissionais favoritos dos clientes com os seguintes dados:

- **customer-id:** número de identificação do cliente;
- **worker-id:** número de identificação do trabalhador;
- **created-at:** data de criação;
- **updated-at:** data da última atualização.

Por fim, a última tabela relacionada aos clientes empregadores é a ***reviews***, a qual contém as informações das avaliações do trabalhador.

- **id:** número de identificação;
- **worker-id:** número de identificação do trabalhador avaliado;

- **customer-id:** número de identificação do cliente que avaliou;
- **title:** título;
- **comment:** comentários;
- **rating:** número de estrelas;
- **is-good:** campo *booleano* indicando se a avaliação é boa ou não;
- **created-at:** data de criação;
- **updated-at:** data de atualização

Em azul tem-se as tabelas relacionadas aos **trabalhadores** (prestadores de serviço), sendo que a primeira é a tabela **worker** que contém as seguintes informações:

- **portfolio-id:** número de identificação do portfólio;
- **worker-id:** número de identificação;
- **description:** uma descrição breve sobre seu perfil;
- **avatar-url:** imagem de perfil;
- **phone:** telefone de contato;
- **state:** estado;
- **city:** cidade.

Em seguida, tem-se a tabela do **portfólio**, na qual está guardada as informações dos serviços já realizados pelos profissionais na aplicação, sendo que a entidade possui as seguintes informações:

- **id:** número de identificação do portfólio;
- **worker-id**: número de identificação a qual trabalhador o portfólio pertence;
- **fotos:** é uma lista de fotos;
- **academic-graduation:** escolaridade do trabalhador;
- **skills:** habilidades, pode ser entendida também como capacitação para um serviço.
- **created-at:** data da criação;
- **updated-at:** data da última atualização.

Seguindo com as tabelas do usuário trabalhador, tem-se a entidade de **experiência**, a qual se refere a trabalhos já realizados ou a algumas de suas capacitações como profissional, além de onde já trabalhou anteriormente. Nessa tabela está contida os seguintes campos:

- **id**: número de identificação.

- **potfolio-id:** número de identificação do portfólio;
- **service-name:** nome do serviço em que já trabalhou;
- **company:** nome da companhia em que trabalhou;
- **city:** cidade;
- **state:** estado;
- **time-worked:** quantidade de tempo trabalhada;
- **created-at:** data de criação;
- **updated-at:** data da última atualização.

Após a explanação sobre as entidades do usuário prestador de serviço, será abordada na tabela de cor verde a parte dos serviços que são prestados pelos profissionais. A primeira tabela dessa cor é a que correlaciona o trabalhador com o serviço e foi denominada por ***worker-service***. Nela contém as seguintes informações:

- **id:** número de identificação;
- **worker-id:** número de identificação do trabalhador;
- **service-id:** número de identificação do serviço.

Seguindo com as tabelas de serviços prestados, tem-se a tabela dos serviços **específicos** com as seguintes informações:

- **id:** número de identificação;
- **service-id:** número de identificação do serviço;
- **name:** nome específico;
- **created-at:** data de criação;
- **updated-at:** data da última atualização.

Por fim, tem-se a tabela de **serviço** contendo as seguintes informações:

- **id:** número de identificação.
- **name:** nome;
- **image-url:** imagem do serviço;
- **created-at:** data de criação;
- **updated-at:** data da última atualização.

Em suma, finaliza-se toda a infraestrutura modelada para a criação do banco de dados e, assim, o banco está pronto para ser alimentado com as informações que vêm das aplicações *web* e *mobile*.

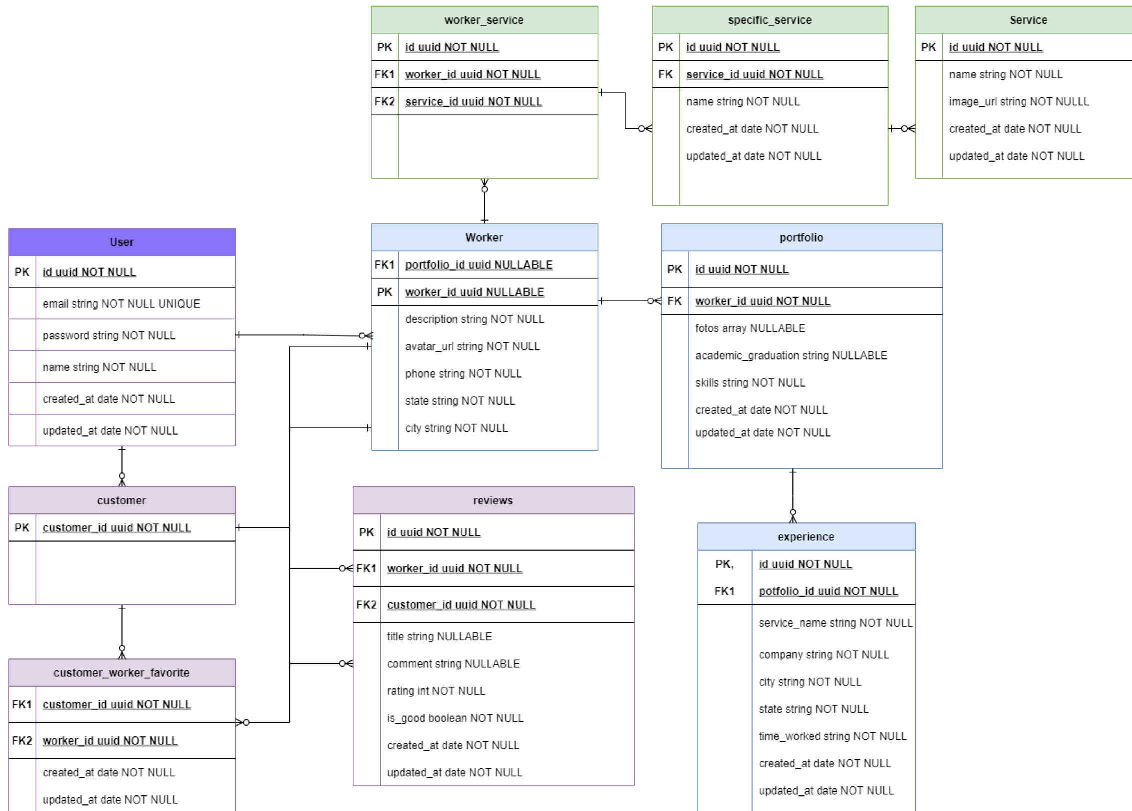

Figura 11 – Diagrama Entidade-Relacionamento do banco de dados.

4.3 Protótipo

Para a prototipação do *back-end* foi utilizado algumas ferramentas que simulam requisições REST com dados falsos, sendo elas o Faker referenciado em 3.3.11 e o JSON *Server* no capítulo 3.2.1.

O JSON abaixo representa como foi feito prioritariamente a listagem dos profissionais nos *front-ends* usando dados falsos. O JSON *Server* cria automaticamente os *endpoints* com as rotas com a primeira chave encontrada no aquivo. O código disponível no arquivo worker.json, a chave é *workers*, por exemplo, para listar todos os trabalhadores utiliza-se o método GET para o seguinte *endpoint*: *http://localhost:3000/workers*.

É possível realizar *queries* com o JSON *Server*. Um exemplo de *query* possível seria listar apenas o trabalhador de *id* igual a 1, sendo que para realizar é só aplicar o método GET para o *endpoint*: *http://localhost:3000/workers?id=1*. Além disso, essa ferramenta contempla todos os métodos HTTP, sendo eles: POST, GET, PUT, PATCH, DELETE. Por fim, é possível, ainda, executar outras atividades, tais como criar um usuário e atualizar os perfis dos profissionais.

4.4 Funcionalidades do TypeORM

Está seção tem como intuito explicar melhor o funcionamento do TypeORM e como foram aplicadas todas as funcionalidades no decorrer da construção do *back-end*.

4.4.1 Entidades

As entidades correspondem às tabelas do banco de dados no código em TypeScript, sendo coleções de campos usados para mapear as tabelas do banco de dados. Nelas são caracterizadas todas as variáveis junto com os seus tipos. Como exemplo, temos a entidade do usuário disponível no arquivo user-entity.ts.

4.4.2 Migration

Migrations permitem a criação e a manipulação de bancos de dados, tendo como objetivo fornecer uma série de recursos, como por exemplo, manter um histórico de alterações que a base de dados vai tendo ao longo do tempo, sendo que com esses históricos é possível reverter qualquer alteração feita e tornar melhor o gerenciamento das mudanças realizadas ao banco de dados, de modo que funcione como um controle de versão.

No TypeORM, para se rodar uma *migration* é preciso antes ter a configuração citada no arquivo orm-config.json, semelhante a que se encontra do código fonte da aplicação Severino:

Após essa configuração, para se criar uma *migration* utilizando o TypeORM basta usar comando *typeorm migration:create -n nome-migration*. Após o término do comando, é criado o arquivo onde se encontra a *migration*, a qual já pode ser editada, assim como no código disponível no arquivo migration.ts:

Na função *up*, na qual ocorre as mudanças que serão realizadas ao executar a *migration*, já na função *down* é ao contrário, ela desfaz as mudanças que a função *up* realizou. Para se executar a *migration* é preciso usar o seguinte comando *typeorm migration:run* e para reverter usa-se *typeorm migration:revert*.

4.4.3 Seeds

Seeds é o conceito utilizado para popular o banco de dados com valores falsos, a fim de testar os *front-ends* antes de estarem com dados reais em produção. Para isso, foi utilizada a ferramenta Faker, abordada no capítulo 3.3.11, para gerar dados aleatórios para preencher as colunas das tabelas no banco de dados. No código disponível no arquivo seed.ts, tem-se a *seed* utilizada para preencher a tabela dos trabalhadores.

4.5 Inicialização do banco de dados

Para subir a instancia do banco de dados durante a criação do *back-end* foi utilizado o Docker Compose, citado no capítulo 3.3.7. Para isso, basta estar situado na pasta em que o arquivo se encontra e usar o comando *docker-compose up*, o que faz com que o arquivo *docker-compose.yaml* seja executado automaticamente construindo toda a infraestrutura do banco de dados. O código disponível no arquivo docker-compose.yaml representa o arquivo.

4.6 Implementação

Após o levantamento de requisitos, foi feita a construção de alguns protótipos e a modelagem do banco de dados. Em seguida, começou-se a etapa de implementação do sistema, a qual consiste em fazer uma *API*, a qual é a aplicação *back-end*, a fim de realizar a comunicação entre os sistemas *web* e *mobile* com o banco de dados para a manipulação e armazenamento dos dados. Essa comunicação é feita por meio de protocolos HTTP e *queries* do TypeORM. Além disso, os *front-ends* são as partes responsáveis pelas interfaces visuais e pela interação do usuário com o sistema, sendo que é a partir delas que as requisições são ativadas para chegar ao *back-end* e retornarem com os dados. Na figura 12 temos um fluxograma que representa tal comunicação que permite aos usuários do sistema manipular os dados para executar suas tarefas.

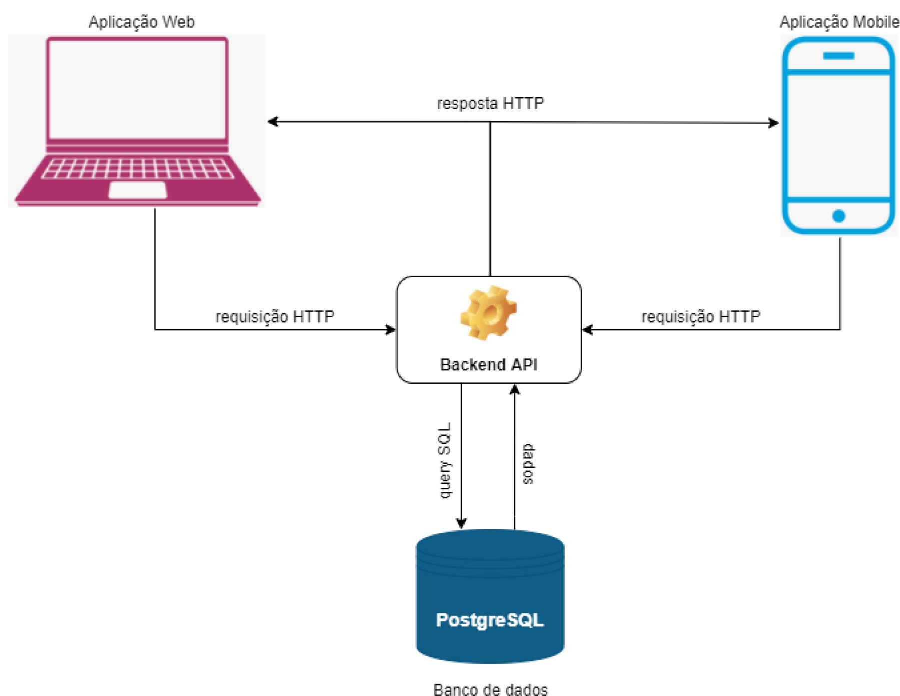

Figura 12 – Representação do funcionamento da aplicação.

Vale ressaltar, que o foco da atual monografia é a construção do sistema *back-end*, o qual contempla inúmeras requisições, assim como demonstrado na área de levantamento de requisitos, sendo que cada requisição foi construída separadamente de acordo com seus níveis hierárquicos, já que algumas demandam dados que apenas outras requisições conseguiriam obtê-los. Por exemplo, para que o cliente empregador possa consultar os *cards* de serviço é preciso primeiro a construção da requisição do cadastro do prestador de serviço e da requisição que preenche as informações nos *front-ends*, para que assim possam aparecer os dados nos *cards*.

4.6.1 Back-End

Para construir o *back-end* do sistema foi implementada uma API REST usando Node.js com TypeScript. O principal objetivo da aplicação é fornecer e manipular os dados do sistema para as aplicações de *front-ends*, as quais ficam armazenadas no banco de dados,

sendo que para efetuar tal tarefa, o sistema fez uso do TypeORM, o qual executa consultas ao banco de dados usando linguagem de alto nível e de melhor desempenho. Além disso, a aplicação contém dois principais grupos de arquivos: "Modelos"e "Controladores". Diante disso, os "Modelos"definem a estrutura dos dados, sendo que tais estruturas são representações similares às tabelas do banco de dados. Já os "Controladores", definem e executam os métodos que manipulam os dados com a realização de ações, tais como ler, criar, atualizar e excluir. Para acessar esses métodos, o Node.js em conjunto com o Express mapeia as requisições disponíveis pela aplicação com o uso de rotas no formato URI e métodos do padrão HTTP como GET, POST, PUT e DELETE. Desse modo, quando uma requisição HTTP é feita para a API, o mapeamento de rotas é usado para chamar o controlador responsável pelo serviço e o método que o executa. Em consequência disso, quando a API receber uma requisição para realizar um serviço ela efetua os passos abaixo, os quais estão representados pela figura 13:

1. Reconhecer o método HTTP da requisição;
2. Usar o *path* no URI para identificar o tipo de serviço que é executado e chamar o método responsável por ele;
3. Localizado no controlador, o método usa o TypeORM e os modelos já definidos para gerar uma *query* e executá-la no banco de dados;
4. Retornar o resultado da requisição.

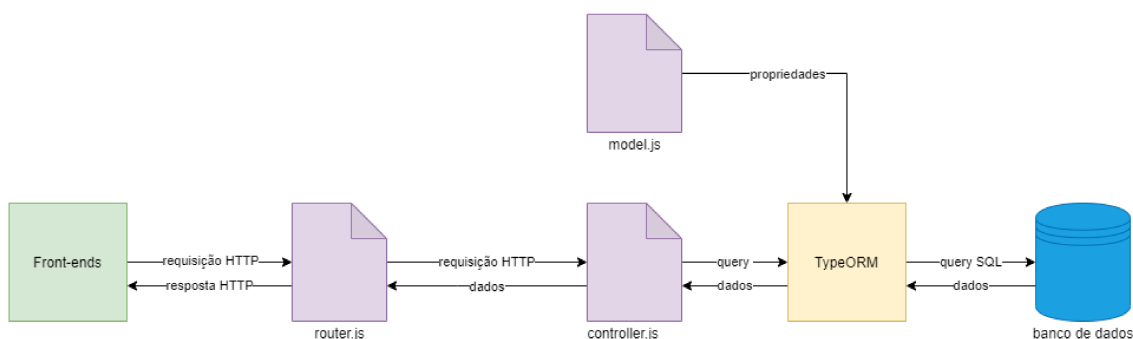

Figura 13 – Representação do funcionamento do sistema *back-end*.

Para demonstrar o fluxo acima foi exemplificado o processo para efetuar duas requisições: criar e ler um serviço. Para criar um serviço é usado o método HTTP POST, o *path* “/service” e as informações dos serviços a serem criados, as quais são mandadas para o *body*, ou corpo da requisição. Já no caso de ler um serviço, é usado o método GET e o *path*, os quais são compostos pela *string* “/service/:id” já com o ID do serviço em questão concatenado ao *endpoint*. As duas requisições estão exemplificadas no código disponível no arquivo create-service.json, sendo que ambas as requisições, em caso de sucesso na execução, retornam os dados dos serviços em formato JSON (JavaScript Object Notation).

Diante disso, assim que o servidor recebe as requisições, é feito um mapeamento das rotas com o método HTTP e o *path* para identificar o serviço que precisa ser realizado com a denominação do controlador e o método responsáveis por ele. A seguir, o trecho

de código disponível no service-router.ts mostra como foram mapeadas as rotas usadas no exemplo.

Desse modo, quando a requisição vinda do *front-end* chega no *back-end* é feito o repasse para o controlador responsável por aquele serviço. No exemplo abaixo, os métodos executados pelo controlador de dados de *services* são *create*, *update*, *delete*, *show* e *index*, sendo que primeiro tem-se o "CRUD"completo, o qual é responsável por criar, atualizar e deletar os serviços. Além disso, o método *show* lista o serviço específico, assim que passado o seu identificador e, por último, tem-se a função *index*, a qual lista todos os serviços presentes no banco de dados. Os métodos citados estão apresentados no fragmento de código disponível no arquivo service-controller.ts:

No segmento do código do controlador citado acima, é possível ver dois métodos do modelo *Service*, os quais são chamados de *index* e *show*. Tais métodos utilizam as duas *queries* representadas por *serviceRepository.findAndCount()* e *serviceRepository.findOneOrFail()*, respectivamente.

Ademais, o TypeORM só consegue gerar essas *queries*, pois existe a definição do modelo dos dados da entidade *Service* na aplicação. Tal modelo segue exatamente o mesmo padrão da tabela definida no banco de dados e, assim, é possível que o TypeORM consiga mapear os dados do banco em objetos manipuláveis, além de gerar tais *queries*. No fragmento de código disponível no arquivo service-entity.ts, tem-se a definição da entidade *Service*.

Capítulo 5

Conclusão

Devido a inclusão tecnológica vivenciada atualmente, em que milhares de brasileiros tem acesso à internet e a algum dispositivo tecnológico, tais como computadores, *tablets* ou *smartphones*, surgiu-se a ideia de criar um canal de comunicação entre pessoas que precisão contratar algum profissional para realizar ou auxiliar em tarefas do dia a dia e pessoas que prestam o serviço. Diante disso, deu-se início a criação da aplicação Severino, a qual foi divida em duas partes, uma em que o usuário consiga contratar, com poucos *clicks*, um profissional para realizar as suas pendências e outra onde um trabalhador que estivesse interessado em trabalhar para a aplicação pudesse realizar o seu cadastro de forma descomplicada, em que não necessite de inúmeras etapas de confirmações, o que poderia tirar o interesse do usuário prestador de serviço.

Em suma, diante da sociedade atual, na qual o tempo disponível dos indivíduos se torna cada vez menor, os métodos convencionais de procura por profissionais estão gradativamente mais obsoletos, tais como pesquisar em agendas telefônicas, anúncios, panfletos ou por indicação. Assim, fica inviável procurar por trabalhadores nesses meios, já que demandam um maior tempo, se comparado aos recursos digitais. Sendo assim, o Severino foi criado, por meio de um *software* multiplataforma, com a intenção de agilizar a vida das pessoas, em que qualquer indivíduo possa acessar uma tela contendo um aglomerado de profissionais e uma diversidade de serviços a serem prestados, sendo que tudo isso com a utilização de um visual inovador e simples aos olhos dos usuários.

5.1 Desafios encontrados

Durante o desenvolvimento do *back-end* muitas dificuldades foram encontradas, desde as consolidações das requisições para atender às regras de negócios da aplicação, até a conexão com o banco de dados, sendo que o conhecimento obtido nas disciplinas de "Programação Orientada a Objetos"e "Sistema de Banco de Dados"foram relevantes durante a construção do código e modelagem do banco relacional.

A substituição do SQL pelas *queries* de alto nível do TypeORM foi outro grande desafio durante a construção, pois utilizar o TypeORM traz benefícios, mas não é fácil de ser aplicado. Além disso, é uma ferramenta confusa de entender, em que a sua documentação não contempla uma boa explicação e possui poucos exemplos para se basear. Dado isso, teve-se retrabalho nas construções de inúmeras *queries*, já que a cada vez que havia algum problema ou impedimento com a *querie*, era preciso reestrutura-la.

Ademais, aplicar a arquitetura MVC para criar um código de alta performance, por meio da utilização de bibliotecas de diversos *frameworks*, em que fosse ao mesmo tempo escalável e de fácil entendimento, foi outro desafio encontrado.

Contudo, experiências vivenciadas tanto nos estudos extra-curriculares, quanto na realização do estágio facultativo e obrigatório, trouxeram uma bagagem de conteúdo e *insights* que ajudaram no desenvolvimento da API REST.

5.2 Trabalhos Futuros

A presente monografia representa a versão inicial do *back-end* da aplicação Severino, sendo que já pode ser exposta para os *front-ends* e receber, em larga escala, informações dos usuários de todo o Brasil. Entretanto, para buscar a excelência e melhor interação do *software* com o usuário, a API passará por novas versões com a adição de novas *features*, a fim de levar a melhor experiência para os clientes empregadores e profissionais da aplicação. Diante disso, as *features* estão listadas abaixo:

- Construção da requisição que será responsável pela contagem de acessos ao perfil do trabalhador, a fim de mostrar para o profissional que o seu perfil está sendo notado pelos clientes empregadores.
- Criar um serviço para salvar os erros que ocorrerem no banco de dados e nos serviços da API, para que, assim, seja possível identificar e corrigir os erros.
- Criar uma camada interna desacoplada do banco de dados que é responsável por guardar informações de denúncias e sugestões de profissionais, com a finalidade de que o cliente empregador possa consultar rapidamente as informações relevantes dos prestadores de serviços.
- Fazer com que o cadastro do profissional seja mais confiável, a fim de que coloque fotos de seus documentos pessoais para que a aplicação faça uma checagem do seu perfil social.
- Desenvolver um algoritmo de predição NLP destinado ao treinamento da base de dados de avaliações, para que, assim, possa separar os comentários em categorias como ruim, bom e ótimo. Diante disso, o cliente empregador poderá ter uma opinião confiável do trabalhador, além da aplicação estar ciente sobre os profissionais que nela trabalham.
- Criar uma tática promocional para que os clientes empregadores e os prestadores de serviço tenham maior adesão à aplicação.
- Estabelecer a aplicação Severino em algum servidor na nuvem.
- Criar contrato de não violação de dados voltado para o *marketing*, afim de atrair empresas de ferramentas para fazer parcerias com a aplicação Severino, a fim de divulgar e vender materiais para os usuários prestadores de serviço.

Referências

AGILEMANIFESTO. **Manifesto for Agile Software Development**. 2001. Agilemanifesto. Disponível em: <http://agilemanifesto.org/>. Acesso em: 21 abr 2021.

ATLASSIAN. **A brief introduction to Kanban**. 2015. Www.atlassian.com. Disponível em: <https://www.atlassian.com/agile/kanban>. Acesso em: 27 abr 2021.

_____. **What is Git**. 2021. Atlassian. Disponível em: <https://www.atlassian.com/git/tutorials/what-is-git>. Acesso em: 15 abr 2021.

CASTSOFTWARE. **Software Engineering Principles**. 2015. Https://www.castsoftware.com. Disponível em: <https://www.castsoftware.com/glossary/software-engineering-principles-characteristics-process-methodologies>. Acesso em: 31 may 2021.

DEVELOPER.MOZILLA.ORG. **Cross-Origin Resource Sharing (CORS)**. 2021. Developer.mozilla.org. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>. Acesso em: 27 abr 2021.

DOCKER. **Overview Docker**. 2017. Docs.docker.com. Disponível em: <https://docs.docker.com/get-started/overview/>. Acesso em: 27 abr 2021.

_____. **Overview of Docker Compose**. 2017. Docs.docker.com. Disponível em: <https://docs.docker.com/compose/>. Acesso em: 27 abr 2021.

DOHERTY, E. **What is Object Oriented Programming? OOP Explained in Depth**. 2020. Www.educative.io. Disponível em: <https://www.educative.io/blog/object-oriented-programming>. Acesso em: 29 jun 2021.

EXPRESSJS. **Express multer middleware**. 2020. Http://expressjs.com. Disponível em: <http://expressjs.com/en/resources/middleware/multer.html>. Acesso em: 12 jun 2021.

GASPARIN, M. **Demanda por serviços cresce 102%**. 2020. Miriangasparin.com.br. Disponível em: <https://miriangasparin.com.br/2020/07/demanda-por-servicos-cresce-102-em-junho/>. Acesso em: 10 jun 2021.

GEEKS, G. F. **What is Git**. 2020. Dev Media. Disponível em: <https://www.devmedia.com.br/introducao-ao-visual-studio-code/34418>. Acesso em: 12 jun 2021.

GITHUB. **Git Hub**. 2021. GitHub. Disponível em: <https://github.com/>. Acesso em: 16 abr 2021.

IBGE.GOV.BR. **Desemprego**. 2021. Ibge.gov.br. Disponível em: <https://www.ibge.gov.br/explica/desemprego.php>. Acesso em: 27 abr 2021.

JAVASCRIPT.PLAINENGLISH.IO. **How Bcryptjs Works**. 2018. Javascript.plainenglish.io. Disponível em: <https://javascript.plainenglish.io/how-bcryptjs-works-90ef4cb85bf4>. Acesso em: 27 abr 2021.

JORNALDOCOMERCIO. **Demanda por serviços cresce 102%**. 2021. Jornaldocomercio.com. Disponível em: <https://www.jornaldocomercio.com/_conteudo/cadernos/empresas_e_negocios/2021/05/792649-triider-registra-crescimento-de-157-em-meio-a-pandemia.html>. Acesso em: 10 jun 2021.

JWT.IO. **Introduction**. 2018. Jwt.io. Disponível em: <https://jwt.io/introduction>. Acesso em: 27 abr 2021.

MARAK. **Faker.js**. 2015. Github.com. Disponível em: <https://github.com/marak/Faker.js/>. Acesso em: 27 abr 2021.

MOBILETIME. **GetNinjas chega a 500 mil profissionais**. 2018. Mobiletime.com. Disponível em: <https://www.mobiletime.com.br/noticias/14/09/2018/getninjas-chega-a-500-mil-profissionais-cadastrados/>. Acesso em: 10 jun 2021.

______. **Triider: app que conecta cliente se profissionais**. 2019. Mobiletime.com. Disponível em: <https://www.mobiletime.com.br/noticias/06/02/2020/triider-app-que-conecta-clientes-a-profissionais-de-servicos-domesticos-chega-a-sao-paulo/>. Acesso em: 10 jun 2021.

MONITORA. **Serviço de engenharia de requisitos: entenda como funciona**. 2020. Monitora. Disponível em: <https://www.monitoratec.com.br/blog/servico-de-engenharia-de-requisitos/>. Acesso em: 22 abr 2021.

MOZILLA, D. **JavaScript|MDN**. 2021. Https://developer.mozilla.org. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Acesso em: 31 may 2021.

MULESOFT. **What is a REST API design?** 2020. Https://www.mulesoft.com. Disponível em: <https://www.mulesoft.com/resources/api/what-is-rest-api-design>. Acesso em: 22 abr 2021.

NODE.JS. **Intro to Node.js**. 2021. Node.js. Disponível em: <https://www.w3schools.com/nodejs/nodejs_intro.asp>. Acesso em: 14 abr 2021.

NPMJS. **json-server, npm**. 2020. Npmjs.com. Disponível em: <https://www.npmjs.com/package/json-server>. Acesso em: 29 may 2021.

POSTGRESQL. **PostgreSQL: About**. 2021. PostgreSQL. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 28 abr 2021.

PRESSMAN, R. S. **Engenharia de Software. Uma abordagem profissional**. 2014. Roger S. Pressman. Disponível em: <https://fateczlads.files.wordpress.com/2014/08/engenharia-de-software-7c2b0-edic3a7c3a3o-roger-s-pressman-capc3adtulo-1.pdf>. Acesso em: 22 abr 2021.

RIO, D. A. **País tem taxa de informalidade de 39,5% no trimestre até dezembro, mostra IBGE... - Veja mais em https://economia.uol.com.br/noticias/estadao-conteudo/2021/02/26/pais-tem-taxa-de-informalidade-de-395-no-trimestre-ate-dezembro-mostra-ibge.htm?cmpid=copiaecola**. 2021. Uol. Disponível em: <https://economia.uol.com.br/noticias/estadao-conteudo/2021/02/26/pais-tem-taxa-de-informalidade-de-395-no-trimestre-ate-dezembro-mostra-ibge.htm>. Acesso em: 09 jun 2021.

SPRING. **Web MCV framework**. 2018. Docs.spring.io. Disponível em: <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-servlet>. Acesso em: 30 abr 2021.

STACKOVERFLOW. **Web MCV framework**. 2019. Model View Controller. Disponível em: <https://stackoverflow.com/questions/55172456/what-is-mvc-in-the-context-of-backend#:~:text=MVC%20as%20a%20pattern%20can,of%20backend%20and%20frontend%20combined.&text=The%20whole%20idea%20behind%20MVC,the%20presentation%20layer%20data%20structure.> Acesso em: 30 abr 2021.

STACKSHARE. **Insomnia REST Client**. 2016. Stackshare.io. Disponível em: <https://stackshare.io/insomnia-rest-client>. Acesso em: 28 abr 2021.

TECHTERMS. **CalmeCase Definition**. 2020. Techterms. Disponível em: <https://techterms.com/definition/camelcase>. Acesso em: 26 may 2021.

TUTORIALSPOINT. **TypeORM - Quick Guide**. 2019. Tutorialspoint. Disponível em: <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm#:~:text=Express%20is%20a%20minimal%20and,core%20features%20of%20Express%20framework%20%E2%88%92>. Acesso em: 27 abr 2021.

_____. **Node.js-Introduction**. 2020. Www.tutorialspoint.com. Disponível em: <https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm>. Acesso em: 31 may 2021.

_____. **TypeORM - Quick Guide**. 2020. Tutorialspoint. Disponível em: <https://www.tutorialspoint.com/typeorm/typeorm_quick_guide.htm#:~:text=TypeORM%20is%20an%20Object%20Relational,js%20and%20written%20in%20TypeScript.&text=js).,apps%20that%20connect%20to%20databases.> Acesso em: 27 abr 2021.

TYPESCRIPTLANG. **TypeScript: Documentation**. 2021. Typescriptlang. Disponível em: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>. Acesso em: 31 may 2021.

WCLR. **TSNodeDev**. 2015. Github.com. Disponível em: <https://github.com/wclr/ts-node-dev>. Acesso em: 27 abr 2021.

YARN, C. **FAST, RELIABLE, AND SECURE DEPENDENCY MANAGEMENT.** 2021. Classic Yarn. Disponível em: <https://classic.yarnpkg.com/en/>. Acesso em: 15 abr 2021.