UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE SISTEMAS PARA INTERNET

LUCAS FERNANDO GERON

MAPEAMENTO COMERCIAL: SISTEMA DE GESTÃO DE RELACIONAMENTO COM O CLIENTE PARA EMPRESAS DE FORMATURAS

MONOGRAFIA DE TRABALHO DE CONCLUSÃO DE CURSO

GUARAPUAVA
2023

LUCAS FERNANDO GERON

MAPEAMENTO COMERCIAL: SISTEMA DE GESTÃO DE RELACIONAMENTO COM O CLIENTE PARA EMPRESAS DE FORMATURAS

Monografia de Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso Superior de Sistemas para Internet – SI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Guarapuava, como requisito parcial para obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Diego Marczal
Universidade Tecnológica Federal do Paraná

Coorientadora: Renata Luiza Stange Carneiro Gomes
Universidade Tecnológica Federal do Paraná

GUARAPUAVA
2023

LUCAS FERNANDO GERON

MAPEAMENTO COMERCIAL: SISTEMA DE GESTÃO DE RELACIONAMENTO COM O CLIENTE PARA EMPRESAS DE FORMATURAS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnológo em Sistemas para Internet do Curso de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná (UTFPR).

Data da aprovação: 3/Julho/2023

Prof. Diego Marczal
Doutor
Universidade Tecnológica Federal do Paraná - Campus Guarapuava

---

Prof. Dênis Lucas Silva
Mestre
Universidade Tecnológica Federal do Paraná - Campus Guarapuava

---

Prof. Luciano Ogiboski
Doutor
Universidade Tecnológica Federal do Paraná - Campus Guarapuava

GUARAPUAVA

2023

Dedico este trabalho a minha amada mãe, Rose.
Por todos os motivos.

AGRADECIMENTOS

Gratidão é expressa através do reconhecimento de uma pessoa por alguém que lhe prestou um benefício. Portanto, expresso aqui o meu orgulho em ter alcançado meu objetivo com a ajuda de vocês.

Rose, minha mãe, a qual sempre insistiu que eu concluísse a graduação, a melhor maneira que posso demonstrar minha gratidão por ti, é tornando real seu sonho de possuir todos os filhos formados.

Bruno Seleme, que por muitos anos possibilitou que eu pudesse me desenvolver como profissional, permitindo que eu me descobrisse e me aperfeiçoasse. O qual proporcionou diversos cursos complementares que foram de bom proveito e através de uma relação de confiança mútua permitiu que eu atuasse em novas áreas colaborando para o crescimento da equipe de trabalho, assumindo grandes projetos e responsabilidades. Sem isto, certamente este projeto não aconteceria.

Diego Marczal e Renata Stange, meus orientadores, que se dispuseram a agregar seus esforços e conhecimentos para que este projeto fosse concluído.

E a todos professores do curso de Tecnologia em Sistemas Para Internet (TSI), pelos vários semestres em que empenharam horas de aulas, compartilhando conhecimento com todos os alunos quantas vezes fosse necessário. Muitos de vocês, além de uma figura de respeito, se tornaram amigos da universidade, tornando a jornada acadêmia mais leve através dos incentivos a programar e permanecer no curso ao longo da graduação.

RESUMO

GERON, Lucas Fernando. Mapeamento Comercial: Sistema de Gestão de Relacionamento com o Cliente para Empresas de Formaturas. 2023. 102 f. Monografia de Trabalho de Conclusão de Curso – Curso de Sistemas para Internet, Universidade Tecnológica Federal do Paraná. Guarapuava, 2023.

Este trabalho propõe o desenvolvimento de uma aplicação web para ser utilizada pelo setor comercial de empresas de formaturas propondo melhorias significativas na forma de gerir as suas negociações através do registro adequado das informações no sistema, o qual além de padronizar a formatação dos dados permite que os usuários acessem os dados de forma online, visualizem e interpretem as informações de um modo intuitivo. Uma das grandes dificuldades entre os gestores das empresas deste segmento diz respeito ao acompanhamento das negociações, onde, por cada representante trabalhar de uma maneira independente e em uma região diferente, a forma de conduzir tais negociações acaba sendo dinâmica, tornando o registro e acompanhamento das informações algo complexo tanto para o representante quanto para o gestor. O sistema em questão busca registrar turmas e alunos de várias cidades e instituições, permitindo ao usuário do sistema documentar seus atendimentos, prospecções e negociações de forma organizada e padronizada. Com isto, a aplicação tem como proposta assumir o papel de um sistema de gerenciamento de relacionamento com o cliente (CRM - *Customer Relationship Management*), a ser utilizado como ferramenta principal para gestão das negociações, viabilizando assim a extração das estatísticas de mercado uma vez que tanto as negociações bem sucedidas quanto as não sucedidas estarão registradas na aplicação, e do desempenho dos representantes baseado nos registros contidos no sistema de forma que seu histórico de atendimentos ficará registrado na aplicação.

Palavras-chave: Aplicações Web. Banco de dados. Planejamento empresarial. Sistemas de Informação.

ABSTRACT

GERON, Lucas Fernando. Commercial Mapping: Customer Relationship Management System for Graduation Companies. 2023. 102 f. Monografia de Trabalho de Conclusão de Curso – Curso de Sistemas para Internet, Universidade Tecnológica Federal do Paraná. Guarapuava, 2023.

This work proposes the development of a web application to be used by the commercial sector of graduation companies, proposing significant improvements in the way of managing their negotiations through the proper registration of information in the system, which, in addition to standardizing the formatting of data, allows users to access data online, view and interpret information in an intuitive way. One of the great difficulties among managers of companies in this segment concerns the monitoring of negotiations, where, since each representative works independently and in a different region, the way of conducting such negotiations ends up being dynamic, making the registration and monitoring of information something complex for both the representative and the manager. The system in question seeks to register classes and students from various cities and institutions, allowing the system user to document their attendances, prospecting and negotiations in an organized and standardized manner. With this, the application aims to assume the role of a customer relationship management (CRM) system, to be used as a main tool for managing negotiations, thus enabling the extraction of market statistics since both successful and unsuccessful negotiations will be registered in the application, and representative performance based on records contained in the system so that their attendance history will be recorded in the application.

Keywords: Web applications. Data bases. Business plannings. Information Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABEFORM	Associação Brasileira de Empresas de Formatura
ABNT	Associação Brasileira de Normas Técnicas
ABRAPE	Associação Brasileira dos Promotores de Eventos
ABRES	Associação Brasileira de Estágios
API	*Application Programming Interface* (Interface de Programação de Aplicação)
AWS	*Amazon Web Services*
BI	*Business Intelligence* (Inteligência Empresarial)
COINT	Coodenação de Sistemas para Internet
CRM	*Customer Relationship Management* (Gestão de relacionamento com o cliente)
CSS	*Cascading Style Sheets* (Folhas de estilo em cascata)
DEED	Diretoria de Estatísticas Educacionais
DOM	*Document Object Model* (Modelo de Documento por Objetos)
DRY	*Don't repeat yourself* (Não repita a si mesmo)
GAS	*Google Apps Scripts*
HTML	*HyperText Markup Language* (Linguagem de Marcação de HiperTexto)
IDE	*Integrated Development Environment* (Ambiente de desenvolvimento integrado)
INEP	Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira
JS	JavaScript
MEC	Ministério da Educação
MVC	*Model-View-Controller* (Modelo - Visão - Controlador)
ORM	Object-Relational Mapping (Mapeamento Relacional de Objetos)
PR	Paraná
SC	Santa Catarina

SISU	Sistema de Seleção Unificada
SSE	*Server-Sent Events* (Eventos Enviados pelo Servidor)
SI	Sistemas para Internet
UML	*Unified Modeling Language* (Linguagem de Modelagem Unificada)
VSCODE	*Visual Studio Code*
WSL2	*Windows Subsystem for Linux* (Subsistema Windows para Linux)

SUMÁRIO

1 INTRODUÇÃO

O setor de vendas de qualquer instituição comercial ou prestadora de serviços é considerado o coração da empresa baseado na premissa de que se não há vendas, então não há receita, e por isto, é comum que os próprios empresários atuem no setor de vendas nas pequenas empresas. Entretanto, segundo Serasa (2020), o principal motivo pelo qual a maioria das empresas declaram falência no Brasil é a falta de recursos ou de lucro, muitas vezes ocasionados pela falta de conhecimento dos gestores do negócio.

Assim como há uma diversidade de empresas que trabalham com vendas, a forma como suas equipes de vendas são organizadas pode variar. No entanto, o conceito de venda é o mesmo, permitindo que empresas distintas em seus segmentos de atuação desenvolvam ferramentas para realizar vendas de maneira genérica, ou seja, ferramentas para venda de vários tipos, como por exemplo, o funil de vendas. Segundo SalesForce (2023), funil de vendas é o acompanhamento de um cliente desde o momento em que ele tem o primeiro contato com os produtos ou serviços da sua empresa até o momento em que a venda é efetivada.

Com a necessidade de possuir sistemas especializados para fazer a gestão das vendas, uma vasta quantidade de softwares surgiu e estão atualmente disponíveis no mercado para este fim. Particularmente, este trabalho enfoca sistemas gestão para o mercado de vendas de formaturas. Tal segmento que possui um modelo de negócio único, podendo ser considerado restrito nacionalmente devido aos costumes e tradições assim como o modelo de educação superior no país. O peso deste segmento é significativo e expressa seu valor de forma financeira. De acordo com o presidente da Associação Brasileira das Empresas de Formatura (ABEFORM), o seguimento de formatura movimentou em 2019 em torno de 7 bilhões de reais baseado nas métricas das empresas associadas (BRUCCE, 2019).

O produto comercializado deste mercado é diferenciado, pois é vendido de forma individual, porém entregue de forma coletiva e muitas vezes possuem valores considerados altos, por isso, o setor comercial de uma empresa de formaturas precisa ser organizado de forma a saber quando, quem, e como abordar o seu cliente. Além disso, uma necessidade na gestão de empresas de vendas de formaturas é poder analisar o desempenho das negociações de forma que a meta de venda seja alcançada gerando receita para empresa. Pois para realizar um atendimento de qualidade, as empresas precisam investir diretamente em seu trabalho, seja promovendo uma festa para que os alunos conheçam o potencial da empresa, levando os formandos à degustação de menus que serão oferecidos ou até mesmo com despesas ocasionadas das reuniões de negócios.

Desta forma, cada empresa estabelece estratégias de venda esperando atingir seus objetivos e obter mais lucro, muitas vezes, sem medir os gastos para angariar novos clientes. Para uma melhor abstração do cenário no qual este trabalho está inserido, considere o contexto

de um curso superior de graduação com uma duração de quatro anos e um *ticket* médio[1] de mil reais por aluno. Considere também um curso de medicina onde o valor de contrato normalmente é dez vezes maior por aluno. O período em que ambas as turmas precisam ser abordadas para oferecer a melhor forma de pagamento é realizada em momentos distintos, uma vez que os valores a serem pagos e o poder aquisitivo dos alunos costumam ser diferentes, assim como o valor gasto por parte da empresa para fechar a negociação também varia de acordo com o *ticket* médio do curso.

Com a necessidade de suprir essa gestão comercial de alto nível de empresas de formatura, este trabalho propõe um recurso tecnológico para gerenciar os dados de maneira simples e intuitiva, com a intenção de reduzir o número de negociações perdidas e auxiliar na rotina de trabalho dos representantes comerciais.

1.1 Objetivo

Nesta sessão é abordado de maneira breve o que foi desenvolvido ao longo da implementação deste trabalho, sendo descritos o objetivo geral e os objetivos específicos que foram alcançados.

1.1.1 Objetivo Geral

Este trabalho de conclusão de curso teve como objetivo geral elaborar um sistema *web* capaz de gerenciar alunos e turmas de diversas instituições e cidades, através de registros de prospecções, atendimentos e negociações. Por meio da utilização do sistema, alimentando a base de dados com as devidas informações, o sistema exibe métricas como quantidade de negociações efetivadas ou perdidas, além de outros filtros, com a finalidade de viabilizar uma análise rápida e interativa da sua empresa e das empresas concorrentes, uma vez que é necessário informar qual empresa foi a ganhadora ao fim de uma negociação, viabilizando assim a elaboração das estatísticas regionais.

1.1.2 Objetivos Específicos

A seguir, os objetivos específicos são elencados na forma de lista, incluindo desde o estudo a ser realizado em uma empresa de formaturas para a compreensão do modelo de trabalho, até os principais objetivos relacionados ao desenvolvimento do sistema como objeto de estudo.

I Compreender o ambiente onde o sistema será utilizado e o modelo de trabalho atual realizado pela empresa de vendas de formaturas;

[1] *Ticket médio* refere-se ao preço médio de contrato individual de prestação de serviços de acordo com a turma em questão.

II Desenvolver um protótipo do sistema de informação afim de validar o modelo de negócio e as principais informações geradas pelo sistema;

III Definir os requisitos funcionais e não funcionais do sistema, as regras de negócios e documentá-los por meio de diagramas de UML e prototipação de telas;

IV Implementar a aplicação *web* com base na documentação elaborada;

V Disponibilizar a aplicação em um ambiente de produção.

2 REVISÃO DE LITERATURA

Este capítulo aborda o contexto de empresas de formatura em relação a estrutura de cursos de graduação superior no país assim como as ferramentas utilizadas para gestão da informação das negociações da empresa.

2.1 Contexto da aplicação

Para o inicio do desenvolvimento deste projeto, é essencial compreender como os cursos superiores de graduação são ofertados no país e a compreensão do ambiente em que o sistema proposto será utilizado, que neste caso, se aplica ao setor comercial de uma empresa de formaturas qualquer.

2.1.1 Educação Superior no Brasil

Segundo dados do BRASIL (2022), de acordo a pesquisa liderada pela Diretoria de Estatísticas Educacionais (DEED) a qual pertence a Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP) que responde diretamente para o Ministério da Educação (MEC), publica que no censo de educação superior realizado no ano de 2020, no Brasil, existiam 2.457 instituições de ensino de educação superior, considerando universidades, centro universitários, faculdades e institutos federais, contabilizando tanto instituições públicas quanto privadas. A partir destas instituições, 41.953 cursos de graduação foram ofertados à população, gerando assim um montante de 8,680 milhões de matrículas durante o ano vigente e alcançando a métrica de quase 1,3 milhão de estudantes que concluíram suas graduações durante o ano de 2020.

Uma maneira em que as instituições públicas e privadas encontram para angariar novos estudantes é ofertar turmas em modalidade semestral e anual, além de turmas com horários alternativos, proporcionando à um maior número de alunos a oportunidade de iniciar os estudos na instituição, gerando mais profissionais preparados para o mercado de trabalho e para a sociedade além de capitar mais recursos para sua rede, no caso das instituições privadas.

2.1.2 Setor Comercial

Apesar do sistema educacional ser bem estruturado e definido, gerenciá-lo é uma tarefa relativamente complexa que cabe as instituições governamentais, porém, perante a perspectiva de uma empresa de formatura, este acompanhamento se torna mais fácil, sendo necessário mapear e acompanhar apenas as cidades de atuação da empresa.

Independente do tamanho do empreendimento, é essencial que todas as empresas mapeiem suas prospecções de forma assertiva e correta, em especial para as empresas de

formatura, visto que o público alvo delas são os alunos que estão próximos de suas graduações, ou seja, um público restrito se considerado os dados publicados pelo censo.

2.2 Trabalhos Correlatos

Considerando uma gama de sistemas disponíveis nos dias atuais, é possível fazer menção de duas plataformas que propõe modelos de *softwares* com soluções de âmbito comercial e que atendem respectivamente empresas de formaturas. Porém, antes e explanar mais sobre tais *softwares*, se faz necessário compreender de forma ampla como um modelo de empresa qualquer de formatura atende o seu público alvo no aspecto de coletar e documentar estas informações internamente.

Considerando uma empresa que atenda mais de 500 cursos, o volume de informações é relativamente alto, pois leva em conta que cada curso possui várias turmas e cada turma possui vários alunos e atendimentos, portanto, excluindo qualquer *software* de gestão de informação, a ferramenta frequentemente adotada costuma ser o Microsft Excel, pois permite planilhar uma série de dados de forma livre, tornando o trabalho de cada representante mais fácil. Contudo, nem todos os representantes estão habituados a usar a ferramenta e muitos se confundem com comandos básicos.

De modo que em dado momento, a necessidade de se realizar uma reunião para apresentação de resultados, é necessário compilar os dados de várias planilhas, afim de validar as atividades registradas e as turmas mapeadas. Além de ser uma tarefa demorada e custosa muitas vezes é falha e gera resultados e imprecisos pois depende diretamente do hábito do colaborador em alimentar as informações e da maneira correta ou do próprio avaliador em auditar a veracidade das informações.

Portanto, visando obter tais resultados de maneira mais rápida, surge a busca por sistemas que sejam capaz de gerir informações referente as negociações de todos os representantes, assim como possam contribuir para os demais setores da empresa no aspecto de oferecer soluções através de aplicações ou aplicativos.

A primeira ferramenta a ser mencionada e que será comparada com aplicação que será desenvolvida ao concluir este projeto é o Control Eventus, represetado pela Figura 1, desenvolvido pela PRONET é um sistema desktop feito para gerenciar os resultados e processos operacionais da empresa (CRITOVÃO, 2023).

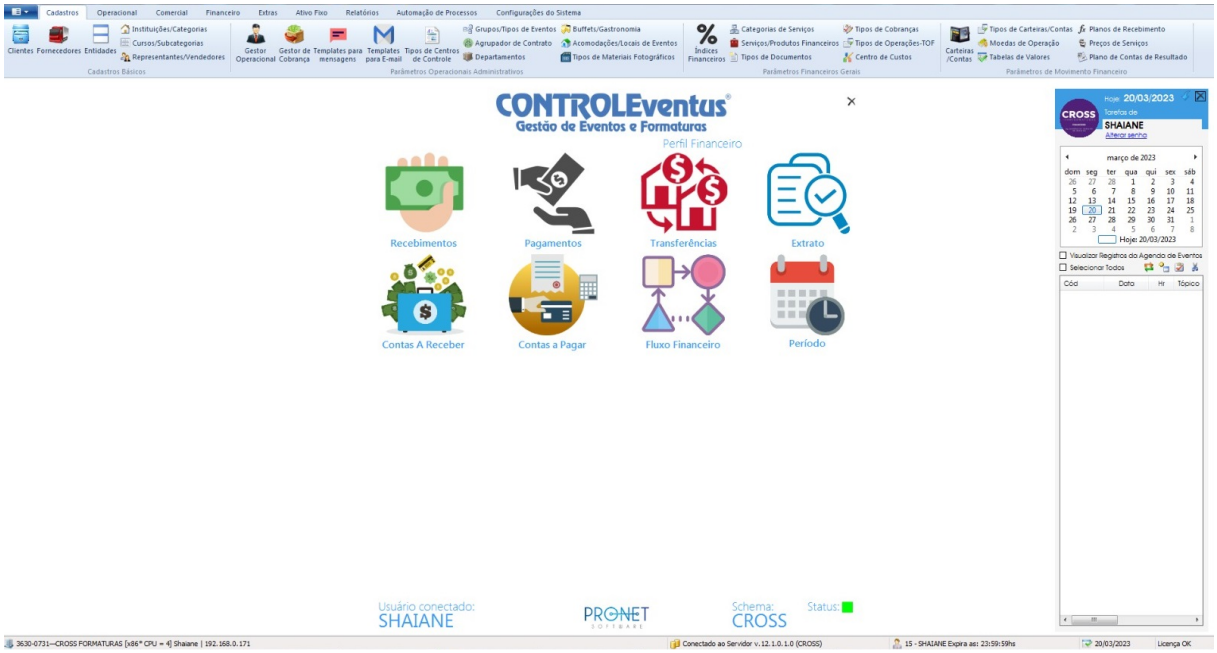

Figura 1 – Control Eventus - Tela Inicial
Fonte: Control Eventus 4.0 - Pronet

O Control Eventus busca atender empresas de formaturas e organizadoras de eventos oferecendo uma cesta de ferramentas robusta, permitindo ao usuário gerenciar todas as modalidades de seu negócio como cadastros de alunos, negociações, orçamentos, relatórios, checklists, calendários, compras e vendas, fluxo de caixa e inúmeros outros recursos. Deste modo, o sistema é considerado uma alternativa para organização destes dados, entretanto, um ponto que precisa ser observado, levanta o questionamento sobre a usabilidade do sistema para os usuários leigos, uma vez que devido as funcionalidades do sistema podem acabar causando confusão, tornando o registro das informações, uma tarefa simples, algo complexo, até mesmo em planilhas do Excel, abrindo assim um espaço para uma observação sobre o projeto proposto e tal sistema.

A segunda ferramenta a ser mencionada se trata do Agendor. Representada pela Figura 2, o Agendor é um sistema que organiza as vendas e permite acompanhar as atividades dos vendedores, além de identificar novas oportunidades de negócio em tempo real (AGENDOR, 2023).

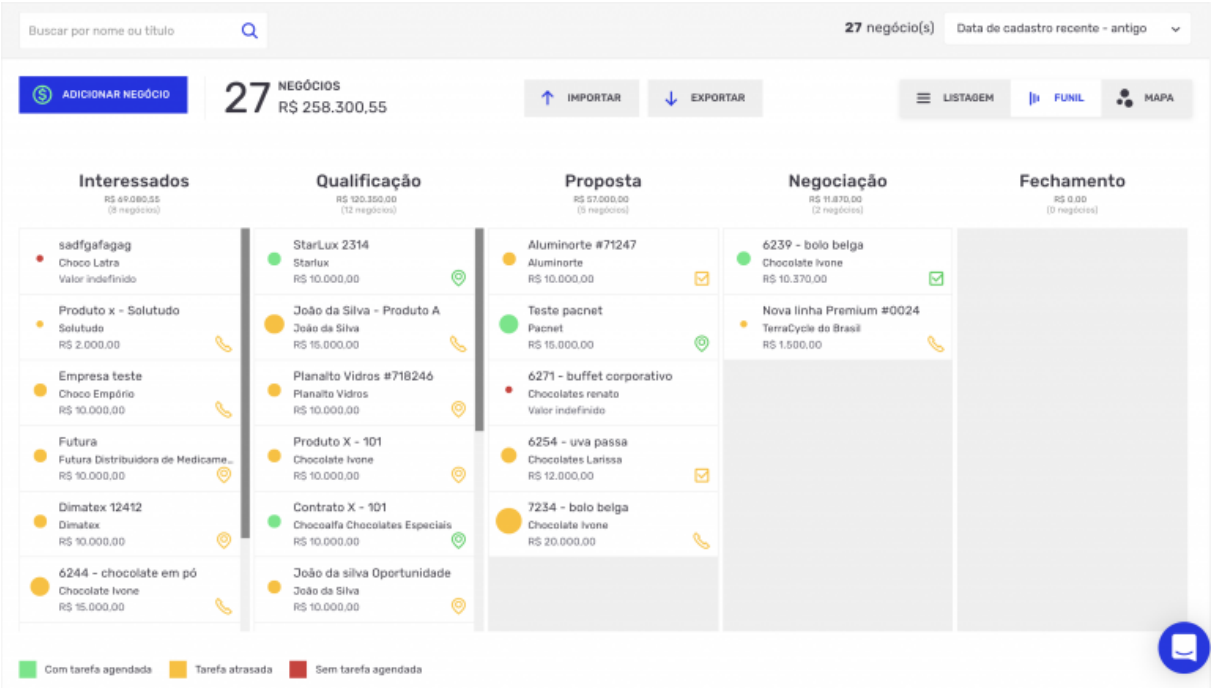

Figura 2 – Funil de Vendas - Agendor
Fonte: Blog Agendor - Matéria: CRM Simples

O Agendor é especializado em gerir negociações de várias maneiras e aspectos, sendo as principais, através do quadro Kanban[1] e calendário.

A plataforma também permite registrar atendimentos de forma automática através de uma extensão específica com que integra o WhatsApp à aplicação, sendo um grande diferencial em relação aos seus concorrentes.

Contudo, apesar de oferecer uma gama de recursos para o usuário, o sistema *web* não possui nenhuma especialidade, e também não permite incluir extensões adicionais para tal, pontuando assim a dificuldade do sistema mediante o volume e dinamismo das informações presente no dia-a-dia de uma empresa de formatura.

É valido destacar que apesar de serem mencionados como objeto de estudo deste projeto, qualquer empresa pode possuir acesso a ambos os sistemas mencionados acima. Neste caso, para a empresa aonde será realizado o estudo descrito anteriormente, o Control Eventus é utilizado como principal *software* da empresa e o Agendor, é utilizado por alguns representantes, podendo assim agregar mais conhecimento ao projeto.

[1]Segundo (MARIOTTI, 2012), Kanban é uma abordagem que funciona como uma ferramenta de sinalização de processos, deixando explícito aos integrantes do projeto, de forma visual, através de um quadro organizado por colunas, o fluxo de trabalho através dos processos atribuídos em cada etapa.

3 MATERIAIS E MÉTODOS

3.1 Metodologia de desenvolvimento

Neste capítulo, será descrito a metodologia de desenvolvimento de *software* que foi adotada para este trabalho, bem como as ferramentas e técnicas escolhidas. A metodologia utilizada foi a **Learn by doing**, que se baseia na ideia de que o aprendizado ocorre através da prática e da experimentação. Essa abordagem permite que habilidades e conhecimentos sejam desenvolvidos enquanto o trabalho do projeto é realizado, testando diferentes soluções e aprendendo com os erros e acertos. É válido destacar que apesar da metodologia ser considerada uma metodologia ativa de ensino e aprendizado e não especificamente de desenvolvimento de *softwares*, utilizá-la para este fim é considerado relevante, pois, durante o processo de desenvolvimento desta aplicação especificamente, se fez necessário realizar pesquisas a respeito das tecnologias utilizadas, sobre o funcionamento do *framework* e de soluções alternativas para alcançar os objetivos propostos.

Conforme Bacich e Moran (2018) aborda em seu livro sobre metodologias ativas para uma educação inovadora, os conceitos do criador da metodologia *Learn by doing*, John Dewey, traz a proposta de que a aprendizagem deve ser feita na prática, pelo envolvimento do estudante nas atividades. Desta forma, o primeiro passo para iniciar o projeto é compreendendo o ambiente onde o sistema será implantado e utilizado. Em seguida, realizar o levantamento de requisitos da aplicação assim como estabelecer corretamente as regras de negócio e por fim estabelecer o cronograma de desenvolvimento e como este será realizado.

Para gestão do projeto, foi aplicado alguns conceitos vindos de outras metodologias que estabelecem diretrizes de grande valia, conforme sugere o autor do *SCRUM*, estabelecendo a premissa de que se for errar, que este erro seja cometido o mais breve possível, para que assim, também seja corrigido o mais rápido possível.

> Fracasse rápido para que possa corrigir o problema o quanto antes. A cultura corporativa costuma dar mais valor a formulários, procedimentos e reuniões do que à criação de valor palpável que pode ser verificado a curtos possibilita de tempo pelos usuários. O trabalho que não resulta em valor real é loucura. Trabalhar em um produto em ciclos curtos possibilita um *feedback* inicial do usuário, permitindo que você possa eliminar imediatamente tudo aquilo que obviamente constitui um desperdício de esforço. (SUTHERLAND, 2014, p. 25).

Para o desenvolvimento deste trabalho foi utilizado o modelo incremental em conjunto com o modelo cascata. A combinação destes modelos pode ser interpretada mediante as etapas de desenvolvimento, onde, desde a concepção do problema até a implantação do sistema, foram realizadas através de etapas, em alguns aspectos consideradas paralelas umas as outras, e ao mesmo tempo, sendo essenciais para o avanço do desenvolvimento do sistema. Considere por tanto que, para abstração do problema, a etapa de estudos precisou ser realizada inicialmente,

após os estudos serem concluídos, a documentação do sistema, assim como o desenvolvimento de um protótipo funcional pode ser realizada, etapas que ocorreram simultaneamente de forma paralela, por sua vez, mediante a documentação estabelecida, foi possível avançar para etapa de desenvolvimento e implantação. Todas estas etapas serão abordadas ao longo deste documento de forma detalhada. Entretanto, para execução dos objetivos propostos pode-se elencar duas etapas principais:

- Construção de um protótipo para validação do modelo de negócio do setor de vendas de formaturas.

- Construção de um sistema *web* que implementa o modelo negócio do setor de vendas de formaturas.

3.1.1 Etapas para Construção do Protótipo Funcional

Com o objetivo de compreender como é o fluxo de trabalho do setor de vendas de formaturas, foi realizado um estudo composto por quatro etapas, sendo respectivamente: **Observação**, **Sugestão**, **Desenvolvimento** e **Implantação de Melhorias**.

I **Observação**: Esta etapa consiste em conhecer o ambiente de trabalho e como suas rotinas funcionam. Para realização desta etapa foram utilizadas duas técnicas principais para elicitação e compreensão dos requisitos do negócio: etnografia e entrevista.

- Etnografia, segundo (GASPAR, 2013), consiste na observação prolongada no local de estudo e conversas abertas com as pessoas que vivenciam as situações.

- A entrevista etnográfica consiste em uma entrevista informal que muitas vezes é resultado da observação participante. Ela é realizada em condições controladas e buscam guiar o entrevistado ao contexto problema abordado.

II **Sugestão**: Esta etapa consiste em propor sugestões diversas e fundamentas para solucionar os problemas encontrados a partir dos pontos observados.

III **Desenvolvimento**: Esta etapa consiste no desenvolvido do protótipo funcional de uma aplicação para gerenciar o modelo de trabalho proposto, assim como corrigir possíveis falhas reportadas na etapa anterior.

IV **Implantação de Melhorias**: Esta etapa consiste em analisar as melhorias da rotina de trabalho dos representantes mediante a implantação das alterações propostas a fim de comparar as vantagens no modelo de trabalho antes e depois das mudanças realizadas.

A partir do protótipo funcional validado, foi possível definir a estrutura da aplicação de forma semelhante a uma aplicação *web*, elaborando a documentação inicial do sistema. Desta forma, serão elaborados os itens:

- **Requisitos do Sistema**: Requisitos funcionais, não funcionais e regras de negócio.
- **Diagramas Auxiliares**: Elucidam os casos de uso do sistema.
- **Estrutura dos Dados**: Elucidam as principais classes e relacionamentos do sistema.
- **Desenho de Telas**: Elucidam o *layout* da interface gráfica do sistema.

3.1.2 Etapas para Construção do Sistema Web

Mediante a conclusão das etapas de construção do protótipo, o sistema *web*, denominado Mapeamento Comercial foi implementado. Para isto, as seguintes etapas foram realizadas:

I **Desenvolvimento Web**: Consiste em implementar o código da aplicação mediante cronograma de desenvolvimento.

II **Implantação em Produção**: Consiste em hospedar a aplicação em servidor real, tornando-a pronto para uso.

III **Validações e Testes**: Consiste em realizar as validações dos formulários, assim como testar os comportamento esperado da aplicação em diversos cenários.

3.2 Ambiente de Desenvolvimento

Para o desenvolvimento desta aplicação, foi utilizado um *Notebook* Samsung NP550XDA-KS1BR, com Intel Core i7 11th Gen 2.80GHz, 16 GB RAM DDR4 e HD de 200GB SSD. Nele, foi instalado o Subsistema Windows para Linux, WSL2.

> O Subsistema do Windows para Linux permite que os desenvolvedores executem um ambiente GNU/Linux, incluindo a maioria das ferramentas de linha de comando, utilitários e aplicativos, diretamente no Windows, sem modificações e sem a sobrecarga de uma máquina virtual tradicional ou instalação *dualboot* Microsoft (2023b, p 1).

Com um ambiente Linux virtualizado em um sistema operacional Windows. Fazendo uso da distribuição Linux Ubuntu 20.04 on Windows, a maquina virtual foi configurada para utilizar uma memória de 8Gb RAM e 2Gb de memória SWAP, configurações suficientes para rodar o sistema sem problemas de desempenho e sem prejudicar o desempenho na aplicação ou o desempenho no ambiente de desenvolvimento. Ainda no Windows, sua versão em relação ao sistema operacional é a *Windows 11 Home Single Language*, distribuição x64 bits, na qual foi instalado um ambiente de desenvolvimento integrado, também conhecido por IDE,

Visual Studio Code[1] e os recursos de integração presente do editor com WSL2 para facilitar na escrita de código e gerenciamento dos contêineres do Docker. Também será utilizado o aplicativo TablePlus[2] que estabelecendo conexão com o banco de dados e possibilita visualizar as informações contidas no aplicação fora do *console* Rails.

O Docker Desktop[3] também foi adicionado ao Windows para realizar o gerenciamento dos contêineres montados e virtualizados. Vale destacar, que inicialmente, o projeto não foi desenvolvido diretamente em Docker, sendo sua virtualização apenas uma das últimas etapas, de modo que a maior parte do desenvolvimento do projeto foi feito através da aplicação rodando diretamente no WSL2. No ambiente de desenvolvimento Linux em WSL2, foi instalado a versão Ruby 3.0.0 e de Rails 7.0.4. Também foi instalado o PostgreSQL [4] 15.2, mas que assim como Docker, também só foi utilizado ao fim do desenvolvimento, sendo em sua maior parte desenvolvido com SQLite3 [5].

3.3 Tecnologias Utilizadas

Nesta seção são descritas as principais tecnologias utilizadas para o desenvolvimento da aplicação assim como suas principais vantagens. Este tópico tem como objetivo proporcionar ao leitor a noção básica das tecnologias e como elas contribuem para o trabalho.

A aplicação *web* foi desenvolvida utilizando a linguagem Ruby e o *framework* Ruby on Rails, seguindo os princípios da metodologia *Learn by doing*. A aplicação *web* permite que os usuários cadastrem e gerenciem as informações das instituições, turmas, alunos, atendimentos e negociações, bem como visualizem métricas sobre o desempenho das negociações.

A aplicação *web* também utiliza recursos como o ActiveStorage, que é um recurso do *framework* Rails utilizado para facilitar o gerenciamento de imagens, a `gem` **Devise**, que é um biblioteca que fornece recursos para a autenticação dos usuários, o Hotwire, que é uma biblioteca *JavaScript* que é utilizado para criação de formulários dinâmicos, o Heroku, que é um serviço utilizado para a hospedar aplicativos em ambiente de produção, o *Amazon Web Service*, AWS, que é um serviço utilizado para realizar o armazenamento de arquivos em nuvem, o Docker, que é uma ferramenta desenvolvida para a criação de um ambiente isolado e padronizado que tem como objetivo manter a aplicação em um contêiner, e por sua, realiza o genericamente deste, e o GitHub Actions, que é uma plataforma utilizada para versionamento

[1] O Visual Studio Code é um editor de código-fonte leve, mas poderoso, executado em sua área de trabalho e disponível para Windows, macOS e Linux. Microsoft (2023a)

[2] TablePlus é uma ferramenta nativa moderna com interface de usuário elegante que permite gerenciar simultaneamente vários bancos de dados, como MySQL, PostgreSQL, SQLite, Microsoft SQL Server e muito mais TablePlus (2023).

[3] O Docker permite que você separe seus aplicativos de sua infraestrutura para que você possa entregar software rapidamente. Docker (2023)

[4] O PostgreSQL é um poderoso sistema de banco de dados objeto-relacional de código aberto que usa e estende a linguagem SQL combinada com muitos recursos que armazenam e dimensionam com segurança as cargas de trabalho de dados mais complicadas PostgreSQL (2023).

[5] SQLite é uma biblioteca em processo que implementa um mecanismo de banco de dados SQL transacional, sem servidor, sem configuração e independente SQLite (2023).

de arquivos, e permite configurar automação dos testes e da implantação da aplicação.

3.3.1 Framework Back-end: Ruby on Rails

O desenvolvimento *back-end*[6] é conhecido por fazer a aplicação funcionar em sua parte lógica, isto é, no processamento das informações da aplicação. Diferente do *front-end* onde sua preocupação principal é a *interface* para o usuário, o *back-end* se preocupa com o desempenho da aplicação assim como a segurança dos dados.

Souza (2013), define Ruby como uma linguagem dinâmica, orientada à objetos e que possui algumas características funcionais, e também afirma que uma das grandes vantagens da linguagem é ser extremamente legível, e de fato, Ruby é uma linguagem que atende a estes atributos.

Ruby sem dúvida é uma linguagem extremamente fácil e intuitiva para se programar, sua sintaxe é muito reduzida e objetiva, o que facilita muito na leitura e interpretação de um algoritmo qualquer. Ruby permite elaborar estruturas de repetições de uma maneira muito mais intuitiva, manipular coleções e textos com uma sintaxe simples além de oferecer vários outros métodos auxiliares que facilitam a utilização da linguagem durante o desenvolvimento da aplicação.

Já o Rails é um *framework* de desenvolvimento de aplicações *web* e foi projetado para que desenvolvedores escrevam menos código e realizem mais tarefas em comparação a muitas outras linguagens. Utilizar o *framework* segundo RailsGuides (2023b), se referindo a expressão "*the rails way*", ou seja, no jeito que deve ser feito em Rails, ao invés de tentar usar padrões aprendidos em outros lugares, pode ser mais eficiente e proporcionar uma experiência melhor ao desenvolver uma aplicação robusta.

Outro motivo que traz destaque ao *framework* deve-se ao fato da comunidade Rails existir e ser bem movimentada. Ao utilizar o *framework* e compartilhar suas dúvidas, problemas e soluções com a comunidade faz com que o *framework* seja aperfeiçoado constantemente. Segundo RubyOnRails.Org (2023), 6.376 desenvolvedores já contribuíram de forma prática, implementando recursos ao *framework* para estimular o crescimento do mesmo e atrair novos desenvolvedores.

Uma vez compreendendo o que a linguagem Ruby permite fazer e como sua sintaxe deve ser escrita, é preciso aprender o que o *framework* Rails faz. Com poucas horas de estudo sobre o *framework* é totalmente compreensível porque ele é considerado um dos melhores *frameworks* de desenvolvimento *web*. Seus recursos como ActiveRecords, ActiveStorage, TurboLinks, Form Helpers, Nested Routes, APIs, Email e inúmeras outros recursos tornam o desenvolvimento e desempenho de qualquer aplicação extremamente mais rápido, desde que implementada corretamente.

Tão intuitivo quanto Ruby, criar um projeto Rails é extremamente fácil, com apenas

[6] "O Back-End trabalha em boa parte dos casos fazendo a ponte entre os dados que vem do navegador rumo ao banco de dados e vice-versa Souto (2023)."

alguns comandos é possível ter sua aplicação funcionando, tudo isso sem precisar configurar muitas coisas. Dos recursos mais interessantes que o Rails oferece, é possível destacar o comando `scaffold`, comando utilizado para gerar as ações básicas de um determinado modelo, isto é, criar, listar, atualizar e excluir dados de aplicações de forma rápida, onde através de um único comando, o *framework* gera todas as rotas, controladores, visualizações, testes e migração para o banco de dados. Rails também permite a integração de `gems`, que tornam possível re-utilizar código de bibliotecas externas ao projeto de forma rápida e fácil conforme sera abordado no tópico a seguir.

Desta forma, apenas com o diagrama de classes do projeto, é possível criar todo o banco de dados do sistema e com pouquíssimo código estabelecer a relação entre os objetos, de forma que o próprio *framework* gerará o resto o código para que tudo funcione. Contudo, uma aplicação web não se resume apenas em alguns comandos, de modo que para colocar nossa aplicação nos trilhos, Fuentes (2012) elenca diversas etapas que precisam ser configuradas antes da aplicação estar pronta para uso. As `gems`, são bibliotecas de desenvolvedores que podem ser facilmente acopladas em um projeto Rails. Estas bibliotecas podem implementar diversos comportamentos e em alguns casos podem possuir dependências para funcionarem corretamente. Com uma comunidade Rails emergente, o número de desenvolvedores que contribuem de forma *open source* para as ruby `gems`, torna o ambiente de desenvolvimento como um todo muito melhor.

Vejamos a seguir uma lista das principais `gems` utilizadas no projeto, seguidas de um breve comentário sobre qual o propósito da biblioteca para com o projeto.

```
gem "tailwindcss-Rails", "~> 2.0"    # Tailwind CSS - Framework CSS
gem "font-awesome-sass", "~> 6.2.1"  # Font Awesome - Ícones
gem "jquery-Rails", "~> 4.4"         # Jquery Rails
gem "city-state"                     # Cidades e Estados
gem "faker"                          # Gera dados fictícios
gem "devise"                         # Autenticação
gem "requestjs-Rails"                # requestjs - AJAX
gem "aws-sdk-s3"                     # AWS S3 - Armazenamento de arquivos
gem "will_paginate"                  # Paginação
```

Desta forma, com estas `gems` instaladas no projeto, é possível utilizar métodos e recursos oferecidos por elas durante a implementação do código da aplicação.

3.3.2 Framework Front-end: Tailwind e Flowbite

A parte de *front-end*[7] requer uma atenção especial pois esta atrelado a usabilidade do sistema, isto quer dizer que um usuário só vai conseguir utilizar de fato o sistema se compreender

[7]"Podemos classificar como a parte visual de um *site*, aquilo que conseguimos interagir Souto (2023) ."

suas funcionalidades e comportamentos básicos. Para se obter isto é recomendado oferecer uma aplicação com uma *interface* amigável, bem organizada, transformando informações vindas do banco de dados em telas que facilitem a interpretação do usuário ao realizar a leitura das informações.

Essa responsabilidade requer cuidado com a estrutura do *layout* da aplicação, tanto para manutenção futura quanto para desenvolvimento de novos recursos, de modo que é preciso manter os arquivos do projeto organizados. Outro ponto importante a ser destacado é que, mesmo que já exista um protótipo funcional com várias telas implementadas com a finalidade de serem utilizadas como referência, a utilização de um *framework* CSS pode implicar em mudanças significativas, uma vez que propõe melhorar o *layout* da aplicação. Considere portanto ao invés de manter um *card* conforme o protótipo, a possibilidade de criar um *card* mais adequado, que forneça as informações necessárias de forma mais acessível ao usuário e formate os dados de forma mais eficiente.

É importante destacar que um ponto crucial no desenvolvimento de um *software* é a implantação do mesmo para as pessoas que irão utilizá-lo. De forme que de nada adianta ter um sistema que faça tudo, mas ninguém consiga utilizar devido ao *layout* das informações. Ou seja, o sistema pode até fazer as tarefas que lhe são propostas, contudo, se o usuário não conseguir extrair informações de sua base de forma clara, a aplicação se tornará gradativamente obsoleta.

Para auxiliar no desenvolvimento da parte visual da aplicação será utilizado o *framework* TailwindCSS. Uma biblioteca que define as classes personalizadas com uma sintaxe intuitiva e explicita, permitindo estilizar os elementos HTML com um visual moderno e limpo descrevendo o próprio elemento em sua forma final.

A escolha por este *framework* também foi selecionada baseada em mais dois pontos, o primeiro é que o *framework* é integrado com o *plugin* PostCSS que é uma dependência complementar utilizada para adicionar recursos de pré-processador à aplicação, compilando arquivos CSS mais leves.

> "Uma a das vantagens do Tailwind é de que sua aplicação será mais rápida, pois o arquivos CSS passa a ser processado apelas pelo PostCSS, e não por várias ferramentas, além de que, os recursos de Sass e Less são compatíveis e funcionam bem com a aplicação TailwindLabsInc (2023)."

E o segundo ponto é de que utilizando as chamadas de *render* - comando para renderizar as telas - do Rails da forma correta, estruturar um *layout* da aplicação se torna uma tarefa relativamente mais simples se comparado a fazer tudo isto sem nenhum arquivo de estilo para auxiliar. Além disto, a biblioteca possui uma vários aspectos CSS definidos e disponibiliza uma documentação robusta com bastante exemplo aos novos usuários de Tailwind.

Como complementos ao *framework* CSS, também foi adicionado ao projeto algumas `gems` complementares, como a biblioteca de ícones Font Awesome, que permite adicionar

diversos ícones na aplicação deixando a interface muito mais expressiva e representativa para o usuário.

O Tailwind Flowbite também foi integrado ao projeto, esta biblioteca é complementar a biblioteca principal, combinando de forma prática os principais recursos do Tailwind o Flowbite oferece componentes adicionais pré-configurados, como `tooltips`, `dropdrowns`, `masonry`, `jumbotrons`, formatação de tabelas e vários outros recursos.

3.3.3 Gerenciamento de Versões

Para auxiliar no versionamento dos arquivos durante o desenvolvimento do sistema será utilizado o gerenciador de versões Git. O repositório por sua vez, será hospedado no Github [8], proporcionando assim à jornada de desenvolvimento um *workspace* mais organizado além de documentar todas as implementações desde o começo do projeto. O repositório deste projeto está disponível no Github e pode ser acessado por qualquer pessoa pelo *link* `github.com/lucasgeron/tcc-utfpr`.

[8] "uma aplicação *Web* que possibilita a hospedagem de repositórios Git, além de servir como uma rede social para programadores Aquiles (2014) ."

4 ANÁLISE DE REQUISITOS E PROTOTIPAÇÃO

O presente trabalho se trata do desenvolvimento de um sistema *web* para gestão de informação em um setor de vendas de formaturas. Para realização deste trabalho, o ambiente de trabalho observado foi uma empresa que possui mais de 15 anos de atuação em seu segmento, que possui em média 150 contratos por temporada e realiza mais de 500 eventos por ano.

Esta empresa possui uma estrutura hierárquica composta pelo diretor da empresa, um gerente regional de vendas, e um time de vendas, o qual compõem o setor comercial. A área de atuação desta empresa se situa principalmente no Paraná e em Santa Catarina de forma que ela realiza eventos em mais de 35 cidades e atende mais de 50 instituições de ensino de todos os níveis de escolaridade, contudo, para o estudo em questão, será analisado apenas a abordagem aos cursos superiores de graduação.

4.1 Construção do Protótipo

Esta seção escreve os resultados obtidos que satisfazem aos seguintes objetivos específicos: 1. Compreender o ambiente onde o sistema será utilizado e o modelo de trabalho atual realizado pela empresa de vendas de formaturas; 2. Desenvolver um protótipo do sistema de informação afim de validar o modelo de negócio e as principais informações geradas pelo sistema. Para compreender como é o fluxo de trabalho do setor, foi realizado um estudo composto por quatro etapas, citadas anteriormente, são respectivamente **Observação**, **Sugestão**, **Desenvolvimento** e **Implantação de Melhorias**. Vejamos de forma detalhada os resultados obtidos durante a execução de cada etapa.

4.1.1 Obervação

Durante esta etapa que durou aproximadamente um mês, foi possível elaborar uma relação de pontos a serem considerados pela direção na empresa. Em primeira instância foi observado que o *Customer Relationship Management*, também conhecido por sistema de gestão de relacionamento com o cliente, ou CRM, utilizado pela empresa, não era utilizado pelo setor comercial em nenhum aspecto, tornando assim o acesso as informações totalmente restritas e exclusivas aos representantes que realizavam os atendimentos e negociações.

Deste modo foi possível observar que nesta empresa é culturalmente explícito que existe um sigilo em relação a situação das negociações que estão ocorrendo. Desta forma, ao questionar o gerente regional de vendas sobre como era realizada a autoria no desempenho dos colaboradores e porque se mantinha sigilo em relação a estes tipos de dados, o mesmo relatou que existia um controle interno realizado através de várias planilhas, sendo uma para cada vendedor, nas quais eram listados os prospectos, isto é, clientes à serem atendidos, atendimentos

realizados e situação das negociações de todas as turmas da instituição, sendo uma planilha por cidade.

Estas planilhas de mapeamento eram compartilhadas semanalmente apenas entre o representante e seu supervisor, entretanto, ao analisar as planilhas em questão, se tornou evidente a falta de padrão, validação e integridade dos dados, pois o documento não apresentava qualquer tipo de segurança ou regra de validação, sendo totalmente editável e público a qualquer usuário. Em relação ao sigilo destas informações, é possível afirmar que a empresa opta por manter os dados sigilosos até mesmo entre os próprios representantes, evitando que quando um colaborador se desligue da empresa ele tenha acesso as demais negociações.

O mesmo vale para representantes distintos, ainda que trabalhem para a mesma empresa e na mesma equipe, considerando que cada representante atenda regiões diferentes, a venda de informações privilegiadas à empresas concorrentes é sem dúvidas uma janela de oportunidade, que mediante as planilhas atuais, permanecem abertas, expondo facilmente o sigilo das informações.

Também foi possível constatar que cada representante comercial utiliza seu próprio celular pessoal em vez de um dispositivo separado próprio para uso profissional, omitindo assim as informações tratadas com os alunos durantes a negociação. Além disto, a forma como os representantes se organizavam para atender as regiões e instituições não era bem definido, isto é, os vendedores mais experientes atendiam mais turmas, independente de qualquer outra regra já pré estabelecida pela direção da empresa, influenciando diretamente nos resultados e em suas comissões. Ou seja, apesar da empresa evidenciar uma forte restrição aos dados do setor comercial, internamente, tais dados não possuem segurança alguma, ainda que sejam utilizados de forma local, não existe nenhuma camada adicional de segurança para tais arquivos, expondo de modo frágil os dados de inteligência do setor.

Ainda durante a etapa de observação do ambiente de trabalho, foi realizado uma entrevista com os representantes comerciais onde os quais foram questionados sobre as ferramentas e rotinas de trabalho e como isso auxilia eles na realização de suas responsabilidades. De forma mutua, todos reportaram uma enorme dificuldade em se organizar de forma correta perante as normas da empresa, isto é, fazendo os devidos registros e anotações em suas planilhas. O principal motivo disto segundo eles é o número elevado de atendimentos realizados durante a jornada de trabalho e além de não possuir um documento exato do que deve ser preenchido, muitas vezes, ficando a cargo deles formatar a própria planilha.

É correto afirmar que a planilha de mapeamento, representada pelo Figura 3, ainda que com diversas irregularidades, é a principal ferramenta utilizada pelo setor. Portanto, o estudo posterior a etapa de observação foi compreender como cada planilha é alimentada e quais são de fato os principais campos utilizados, visto que cada representante se organizava de maneira diferente.

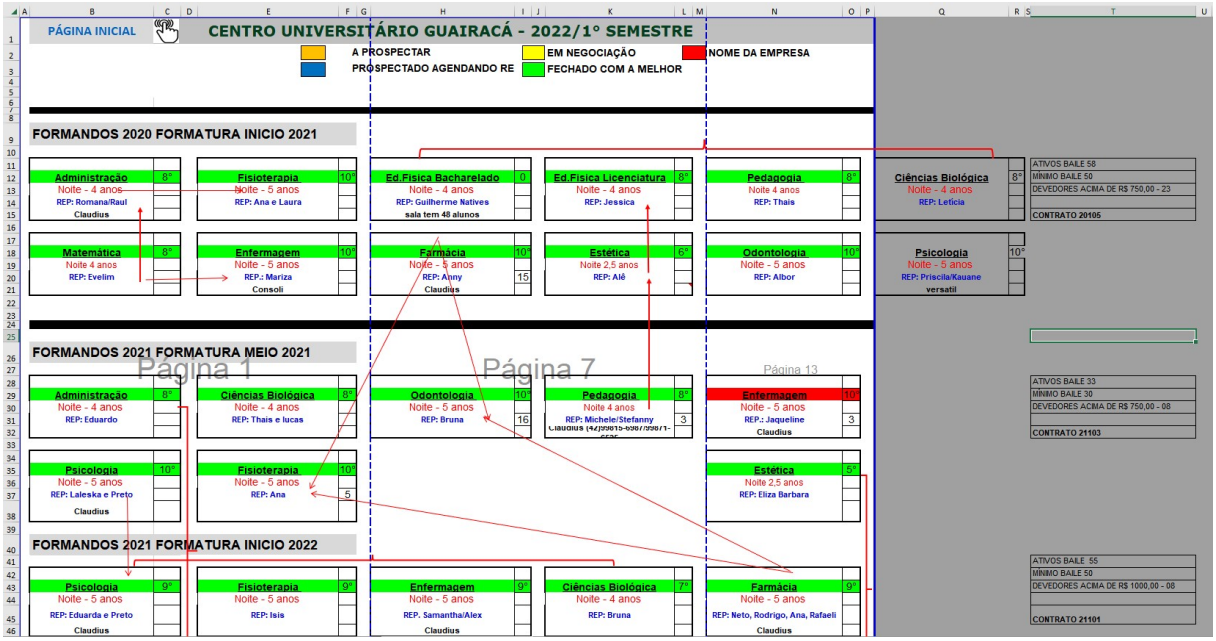

Figura 3 – Planilha de Mapeamento Comercial
Fonte: Empresa Colaboradora

É possível constatar que além da falta de padrão e formatação, o documento apresenta diversas setas direcionais, as quais correspondem ao entendimento do representante que alimentou esta planilha, a ponto de nem mesmo o gerente de vendas saber exatamente se possuem algum significado e o que elas representam.

Em um estudo detalhado dos diversos modelos de planilha, onde cada uma correspondia a uma cidade, foi possível estabelecer os principais campos utilizados para documentar a rotina de trabalho, sendo eles respectivamente:

- **Curso** - Nome do curso.
- **Período** - Período da turma em relação ao número de períodos do curso;
- **Representantes** - Nome e Telefone do Prospecto ou Representante;
- **Comissão** - Nome e Telefone da Comissão da Turma;
- **Status** - Situação da Negociação;
- **Empresa** - Empresa que fechou contrato com a turma.

Por sua vez, estes campos eram replicados para cada semestre, de cada curso, de cada instituição, de cada cidade, gerando assim uma planilha em formato de 'tabuleiro', nome familiarizado pelo setor comercial, o qual se referia a planilha de mapeamento comercial. A maneira como os dados eram gerenciados pelos representantes e pelo setor era considerada caótica e defasada, pois não ofereciam recursos padronizados de segurança a todos os representantes, de modo que não era possível se obter uma métrica de desempenho dos colaboradores em relação ao número de negociações fechadas ou perdidas mediante seus atendimentos realizados.

Outro ponto à ser elencado neste etapa, se trata do histórico de atendimentos, que

por não possuir um lugar correto a ser armazenado, acaba não existindo, ficando a mercê dos representantes e suas anotações, muitas vezes tendo o histórico apenas no dispositivo móvel do representante. Considere que, na planilha em questão, até existia uma campo determinado para registrar tais atendimentos, entretanto, os poucos representantes que o utilizavam, para manter a formatação do documento, substituíam a informação da célula, perdendo o histórico das negociações, mantendo assim apenas a última atualização registrada.

Uma última observação considerada uma dificuldade trivial, mas que fazia a negociação ser influenciada, era a atualização tardia da planilha, visto que após um semestre passar, os alunos também deveriam passar de período, porém, devido a falta de atualização no documento, era comum entrar em contato com os alunos de forma tardia, muitas vezes perdendo a negociação por falta de contato no momento adequado, tornando o parcelamento do contrato inviável para os possíveis contratantes. É importante destacar que a planilha poderia ter uma `macro` implementada para realizar tal comportamento, contudo, isto eventualmente era feito de forma parcial e manual.

Outros assuntos estudados durante a etapa de observação que não são pertinentes ao escopo deste trabalho não serão abordados, visto que, contemplam assuntos como estratégia de venda, marketing e contabilidade, assuntos os quais fazem parte do setor comercial da empresa, contudo, fogem do escopo deste projeto.

4.1.2 Sugestão

Como forma de contribuir no processo de melhoria com a empresa colaboradora do estudo, uma reunião com os gestores e diretor da empresa foi realizada nas quais os itens observados foram apresentados juntos de melhorias e boas práticas relativos ao item elencados. Diante das observações listadas, propostas e sugestões de melhorias foram feitas, podendo-se destacar a de desenvolver um aplicativo *web* para realizar o mapeamento comercial, facilitando assim as atualizações de planilhas de formas irregulares e tornando o preenchimento das informações e atendimentos muito mais rápida, além de permitir emitir notificações e receber alertas sobre novas prospecções.

Contudo isto levaria alguns meses para ser desenvolvido e durante este tempo a empresa ainda estaria trabalhando de forma caótica, portanto, antes de qualquer busca alternativa de soluções para o desenvolvimento da aplicação, uma nova planilha de mapeamento comercial foi elaborada de forma padronizada, com validações de dados e formatação condicional com o objetivo de ser uma solução provisória para o problema principal.

Por fim, tal sugestão ganhou destaque durante a reunião, e acabou sendo proposto como resultado da mesma o desenvolvimento da aplicação, listando apenas as objeções de possuir protótipos de telas semelhante ao modelo atual, conforme o mapeamento comercial, e um cronograma de desenvolvimento do projeto. Apesar da ideia do projeto ter sido aprovada, nenhuma especificação sobre como a aplicação deveria funcionar, suas funcionalidades ou aparência foi definida, tendo como contexto da aplicação, o problema abordado durante a

reunião, e a resposta, uma aplicação que solucione tal problema.

4.1.3 Desenvolvimento do Protótipo

Com a o objetivo de suprir uma demanda urgente na gestão da informação e explicitamente temporária, com regras de validação de dados e padrão em sua formatação de campos, abas e formatação condicional, uma nova planilha, representada na Figura 4, foi desenvolvida com já pensando na possibilidade de se importar tais dados para a aplicação futuramente.

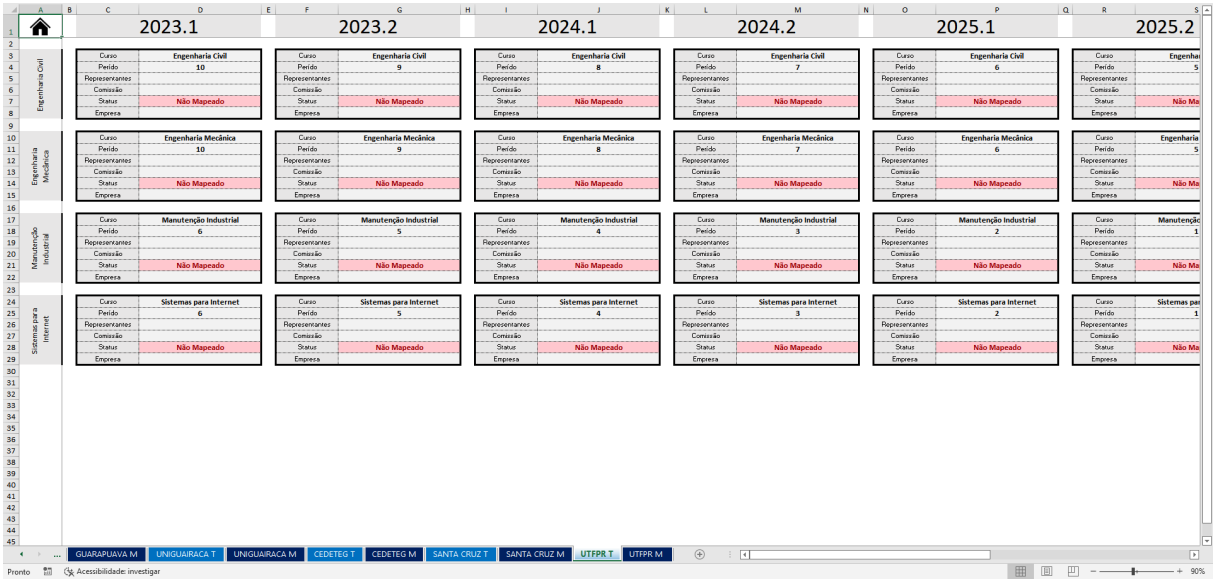

Figura 4 – Planilha de Mapeamento Comercia Reestruturada
Fonte: Autoria Própria.

Esta nova planilha ainda possuía a capacidade de registrar novos cursos para as instituições, ou abrir novos semestres de forma prática, bastando apenas copiar e colar as colunas, que as formulas pré-configuradas faziam o resto da configuração.

Em busca de soluções rápidas para o desenvolvimento deste projeto que ainda não possuía seus requisitos de sistema, protótipos de tela e banco de dados bem definidos, foi possível conhecer o AppSheet[1], um produto GOOGLE (2023a), listado na sessão de *No-code Development*.

> Na tradução direta para o português, *no-code* quer dizer "sem código", e o termo é usado para descrever plataformas que permitem o desenvolvimento de programas a partir de comandos simples, como arrastar e soltar. Além disso, esses sistemas têm interfaces acessíveis e são bastante simples de operar, já que são feitos pensando exatamente em quem não sabe nada de programação (LIMA, 2022, p. 1).

[1]Permite criar um app para uso em computadores, smartphones e tablets de modo que todas as pessoas possam criar e estender aplicativos através de planilhas e fonte de dados, sem programação.

Com o auxílio desta plataforma, foi possível transformar a planilha de mapeamento comercial, agora reestruturada, em uma Planilha Google[2], para ser utilizado como banco de dados da aplicação, gerenciando níveis de segurança da aplicação através da segurança Google O-Auth2.

Segundo GOOGLE (2023c), a tecnologia Google O-Auth2 permite uma experiência de vinculação perfeita para os usuários do Google, pois com uma conta ativa, é possível que o usuário crie uma nova conta no seu serviço usando a Conta Google, desta forma, por ser um produto integrado ao Google Workspace, o AppSheet faz uso dos recursos de segurança de forma integrada, utilizando as credenciais da conta para definir permissões de leitura, edição, uso e compartilhamento do aplicativo hospedado na plataforma, de forma semelhante ao compartilhamento de arquivos do Google Drive. Desta forma, com as regras básicas da tipagem dos dados e a configuração de proteção dos dados, a plataforma realiza uma auto configuração, gerando assim as telas, rotas, formulários e botões de navegação de acordo o banco de dados, isto é, a planilha google que foi criada.

É importante citar que todas as configurações realizadas pela plataforma são pré-configuradas e qualquer comportamento diferente do esperado deve ser configurado utilizando o *framework* como *interface* de código. Vale destacar que o *framework* de desenvolvimento, possui recursos para executar consultas a diversas `API` e `scripts` de terceiros ou a rotina de atividades através de eventos ou de regras pré estabelecidas, além de permitir que o usuário estabeleça fórmulas semelhante a uma planilha. Desta forma, de uma maneira muitíssima simplificada, foi possível realizar o desenvolvimento de uma aplicação completamente responsiva hospedada no AppSheet apenas utilizando os recursos já disponíveis na plataforma.

Durante o desenvolvimento prático da aplicação, é possível citar algumas etapas manuais, como a personalização das telas, o relacionamento entre os objetos, a programação de automações, regras de notificações, regras de validação de dados, formatação condicional e integração um `script` externo que viabilizava adicionar métodos personalizados a aplicação, permitindo a geração automática dos registros na tabela 'Tabuleiros' como fora denominada, assim que um novo registro na tabela 'Semestres' fosse adicionado, comportamento qual a plataforma AppSheet não dispunha de recursos nativos para realizar tal ação.

É valido destacar que a plataforma AppSheet por sí poderia fazer o aplicativo sem a necessidade de código com maestria, contudo, o comportamento personalizado devido ao relacionamento entre as classes era algo que precisava ser implementado. No exemplo acima por exemplo, era possível criar apenas um novo registro na tabela 'Tabuleiros' quando um novo semestre fosse criado, porém, o comportamento correto seria criar de forma dinâmica, um tabuleiro para cada curso aberto do sistema.

Para vincular esta funcionalidade ao sistema, um `script` foi escrito através do Google Apps Script (GAS) contendo os métodos que implementavam tal logica. Segundo GOOGLE

[2]Planilhas colaborativas, inteligentes e seguras para organizações que precisam de agilidade, e que trabalham de modo integrado à infraestrutura do Google. GOOGLE (2023b)

(2023d), o Google Apps Script é uma plataforma de desenvolvimento rápido de aplicativos, que permite criar aplicativos empresariais que se integrem ao Google Workspace de forma fácil, pois o editor online permite que sem a instalação ou configuração de nada, código `Javascript` seja executado diretamente nos servidores da Google realizando tarefas em todo *workspace* de maneira eficiente.

O GAS é uma ferramenta poderosa que permite desenvolver dois tipos de *scripts*, `Standalone Script`, considerados *scripts* autônomos e `Container-bound Scripts`, *scripts* vinculados a um arquivo do Planilhas, Documentos, Slides ou Formulários Google. Em ambos os tipos de *script* é possível desenvolver menus, caixas de diálogos e barras laterais para o serem utilizadas no *workspace*. Além disto, é possível criar tarefas automatizadas e hospedar aplicações *web* que podem ser muito úteis no dia-a-dia.

Neste projeto, o uso de `scripts`, permitiu integrar ao AppSheet uma função para executar um método que altera os registros de uma Planilha Google, que por sua vez é utilizada como banco de dados da aplicação, finalizando o ciclo de integração. Os métodos definidos tinham como objetivo implementar a sincronização de dados, e a criação de registros de forma automatizada, contudo, não será abordado detalhes sobre as implementações do código, ou em sua lógica de manipulação de dados, no entanto, é possível afirmar que para realizar diversas tarefas o `script` escrito possuí aproximadamente 210 linhas de código e realiza uma série de consultas cruzadas nos dados da planilha, tornando o processo mais lento mediante o volume de dados registrados.

É correto afirmar que o maior viés facilitador de utilizar uma plataforma *No-code Development* como alternativa para o desenvolvimento de aplicações robustas é a agilidade e praticidade devido aos recursos disponíveis para estruturar sua aplicação, entretanto, devido ao modelo de plataforma modularizado e genérico, ao utilizar recursos de formas específicas que fujam a regra pré-estabelecida, a aplicação começa a se comportar de maneiras inadequadas. Para exemplificar, durante uma rotina de testes, foi constatado que era possível adicionar um aluno, diretamente da tela de prospecções, comportamento que funcionava corretamente no dispositivo móvel, mas quando realizado em um navegador através de um computador, apresentava falha.

A falha consistia na ausência das informações ao fazer o redirecionamento da página, uma vez que a versão *mobile*, te redirecionava para páginas diferentes a cada critério do filtro, e a aplicação *desktop* apenas filtrava os dados presente, sem realizar uma nova consulta baseada nos critérios selecionados, erro que era ocasionado pelas rotas da aplicação e que não poderiam serem alteradas. Será possível ter uma compreensão melhor deste exemplo ao analisarmos os protótipos de tela.

Ou seja, assim como esta falha que foram detectada e reportada ao time de desenvolvimento do AppSheet, a comunidade da aplicação também reporta inúmeras inconsistências da plataforma, que por mais madura e preparada que esteja, ainda tem muito a ser desenvolvida e melhorada.

Ainda se tratando do desenvolvimento desta aplicação embarcada, é possível afirmar que toda a elaboração de sua estrutura, como banco de dados, telas específicas e comportamentos da aplicação foram realizados com embasamento nos conceitos de programação aprendidos durante a graduação, já cogitando a real possibilidade desta aplicação se tornar uma aplicação *web* independente, sem estar hospedado em um serviço terceirizado.

É relevante ter conhecimento de que alguns recursos, como o registro de atendimentos, lista de cidades, instituições, cursos e diversas outras informações, precisaram ser adaptadas da planilha de mapeamento comercial, sendo organizadas por sua vez tabelas relacionadas, permitindo assim o registro correto das informações, de forma ordenada e definida, possuindo suas respectivas abas:

- **Prospecções** - Tabela que armazenava detalhes das prospecções da empresa.
- **Representantes** - Tabela com o cadastro de todos os representantes comerciais.
- **Cidades** - Tabela de cidades e estados do Paraná e Santa Catarina.
- **Instituições** - Tabela de instituições atendidas pela empresa.
- **Cursos** - Tabela de cursos das instituições.
- **Semestres** - Tabela de semestres listados no sistema.
- **Tabuleiros** - Tabela onde cada registro correspondia a uma turma de cada curso.
- **Alunos** - Tabela com o cadastro de todos os alunos.
- **Atendimentos** - Tabela com o registro de todos os atendimentos realizados.

Também é relevante destacar que como forma de padronizar os dados contidos no sistema, uma pesquisa foi realizada na qual os representantes informaram uma relação de cidades e instituições atendidas, e por sua vez, uma busca por todos os cursos, modalidades e períodos em que eram ofertados e a duração do curso foi realizada manualmente. Dentre as tabelas listadas é possível afirmar que cada uma tabelas apresenta uma estrutura de colunas própria que representa seus respectivos atributos e muitas delas apresentavam colunas virtuais.

> Uma coluna virtual é calculada automaticamente usando uma expressão de fórmula de aplicativo. Os valores da coluna virtual não são realmente armazenados em nenhum lugar, então eles não aparecerão em sua planilha depois que você os criar. No entanto, eles afetam seu aplicativo, comportando-se como uma coluna comum AppSheet (2023, p 1).

As colunas virtuais neste *framework* são fundamentais e possibilitam estabelecer vínculo as pastas do Google Drive de forma segura, uma vez que a segurança do Google realiza a listagem e acesso aos documentos dos arquivos através da aplicação. As colunas virtuais também podem ser utilizadas para agregar dados ou concatenar textos, devido as restrições da aplicação.

Os detalhes deste projeto, como detalhes do protótipo de telas, estrutura do banco de dados, requisitos de sistema, entre outros tópicos serão descritos ao decorrer da metodologia com o objetivo de situar o leitor a uma melhor interpretação do conteúdo.

4.1.4 Implantação do Protótipo

Com a aplicação desenvolvida através de um *framework no-code* implementada com algumas restrições entre as versões *desktop* e *mobile*, o acesso a aplicação foi liberado aos representantes da empresa colaboradora, juntamente com um treinamento e a explicação de todos os recursos disponíveis na mesma.

Durante este processo de adaptação ao sistema foi possível perceber a utilização da aplicação e registrar com sucesso diversos atendimentos e alunos, mas o mais inovador para o setor, era conseguir saber de forma clara em qual turma haviam necessidade de atuar, uma vez que o sistema permitia filtrar os dados da tabela 'Tabuleiros' que representavam as turmas, de cada semestre, de cada curso, por diversos parâmetros.

Como resultado, a aplicação permitiu que os representantes de venda tivessem sua lista de clientes em potencial de forma clara, e também tornou possível acompanhar suas negociações baseado nos status das mesmas. Além disto, o protótipo também permitiu a direção da empresa estabelece-se metas aos representantes, com a intenção de melhorar o desempenho do time de vendas. A aplicação também entrega de maneira prática notificações sobre novas prospecções assim que eram registradas e enviava um relatório de uso semanal de modo automatizado.

A representação visual do protótipo implementado está disposto na seção 4.2.4. Tal seção define as telas da aplicação AppSheet como modelo de referência para os protótipos de tela que serão utilizados na aplicação *web*.

Portanto, com a conclusão do estudo e implantação da aplicação desenvolvida, ficou evidente a melhora significativa das principais dificuldades do setor, uma vez que poderia ser utilizado de forma mais simplificadas do que as agendas manuscritas e planilhas complexas, com isto, a empresa também possuía acesso aos registros dos usuários da aplicação e poderia acompanhar as métricas de desempenho de cada representante, coisa que antes não era possível.

Com a conclusão do modelo funcional, ainda que simplificado e limitado, o próximo passo consiste no desenvolvimento de uma aplicação *web* que apresente um comportamento semelhante, mas que seja acessível para edição e customização.

4.2 Mapeamento Comercial

Os resultados apresentados nesta seção buscam satisfazer ao seguinte objetivo específicos: 3. Definir os requisitos funcionais e não funcionais do sistema, as regras de negócios e documentá-los por meio de diagramas de UML e prototipação de telas.

Como objeto de estudo desta monografia, a aplicação *web* à ser desenvolvida, denominada como Mapeamento Comercial, originada da demanda do mercado de formaturas com alto potencial de captação de receita e criado através do estudo realizado em uma empresa credibilizada para tal, demonstrou através da implantação de um protótipo funcional, o potencial em proporcionar uma melhor organização e visão estratégica ao setor comercial.

É importante destacar que o escopo deste projeto pode ser resumido em elaborar um sistema capaz de filtrar dados com múltiplos critérios, que possua gerenciamento de cidades, instituições, cursos, turmas e alunos, assim como possuir representantes que possam interagir com o sistema registrando atendimentos, negociando com turmas e assumindo prospecções de novos clientes.

Desta forma, além de possuir uma base de dados bem estruturada a aplicação deve exibir tais dados de forma representativa, afim de facilitar a interpretação das informações ao usuário do sistema, utilizando as próprias planilhas e o protótipo como referência de interface.

Ao decorrer desta sessão, o projeto será descrito em detalhes de acordo com os tópicos, sendo eles respectivamente: **Requisitos do Sistema**, **Diagramas Auxiliares**, **Estrutura dos Dados**, **Desenho de Telas**.

Os tópicos elencados definem o processo de criação e especificação da aplicação que será desenvolvida ao longo deste projeto. Portanto, mediante a conclusão do estudo realizado anteriormente e com o protótipo funcional da aplicação em funcionamento, é possível analisar o uso do mesmo afim de estabelecer de fato o modelo estrutural da aplicação. É importante considerar da mesma forma que as planilhas foram reestruturadas e passaram por uma série de adaptações até se tornar o protótipo, o mesmo deve ocorrer ao adaptar o protótipo em um produto final, sendo mudanças tanto na estrutura de dados, quanto aos recursos que o *framework* oferece.

Vejamos a seguir os primeiros documentos elaborados para descrever a aplicação *web* que será implementada.

4.2.1 Requisitos do sistema

Afim de aprofundar as especificações técnicas do projeto, mediante aos resultados obtidos até o presente momento, isto é, com uma base de dados sólida e robusta assim como um protótipo funcional, é possível elaborar a seguinte lista de requisitos de sistema, descrita de forma agrupada por requisitos funcionas, requisito não funcionais e regras de negócio. Vejamos a relação elaborada na Tabela 1.

Tipo	**Requisitos**
Requisito Funcional	A Aplicação precisa armazenar coleções de Representantes, Cidades, Instituições, Cursos, Alunos, Atendimentos, Turma, Prospecções e Semestres.
Requisito Funcional	Uma Cidade tem muitas Instituições.
Requisito Funcional	Uma Instituição tem muitos Cursos.
Requisito Funcional	Um Curso tem uma Turma por Semestre.
Requisito Funcional	Uma Turma tem muitos Alunos.
Requisito Funcional	Um Representante deve possuir uma atribuição que pode ser "Prospectador"ou "Representante de Vendas".
Requisito Funcional	Um Aluno deve possuir uma atribuição que deve informar se o mesmo pertence a comissão de formatura
Requisito Funcional	Um Alunos deve possuir uma atribuição que deve informar se o mesmo é considerado informante
Requisito Funcional	Um Atendimento deve possuir uma atribuição que pode ser "WhatsApp", "Ligação", "Reunião Presencial", "Reunião Online", "Assembléia"ou "Email".
Requisito Funcional	Uma Negociação, deve ser representada pela Turma e deve possuir uma atribuição que pode ser "Não Iniciada", "Em Prospecção", "Em Negociação" ou "Fechada".
Requisito Não Funcional	O sistema deve ser protegido por Login e Senha.
Requisito Não Funcional	O sistema deve possuir uma interface intuitiva e de fácil usabilidade.
Requisito Não Funcional	O sistema deve permitir a listagem dos dados através de agrupamentos e filtros específicos.
Regra de Negócio	Uma Prospecção deve ser registrada através de um formulário público.
Regra de Negócio	Um Atendimento deve possuir obrigatoriamente turma atribuída ao registro, e pode incluir um ou vários alunos no atendimento.
Regra de Negócio	As turmas devem ser geradas pelo sistema de acordo com os cursos ativos e semestres ativos.
Regra de Negócio	Uma Negociação, ao possuir o atributo "Fechada", deve possuir a empresa informada.

Tabela 1 – Requisitos do Sistema.

Fonte: Autoria Própria.

É importante considerar que apesar de pretender ser um sistema capaz de atender qualquer empresa de formatura, devido ao contexto do projeto, alguns requisitos trazem especificações do ambiente de trabalho como exigências, como por exemplo os possíveis cargos de um representante ou os possíveis status de uma negociação, de forma que tornam o comportamento da aplicação limitada ao modelo de trabalho estudado.

Contudo, a metodologia *Learn by doing* sugere que tais requisitos deveriam ser considerado um problema real, de forma que a interação com o problema e suas variáveis façam o papel de motivar o aprendizado na busca de uma solução para atingir as exigências solicitadas.

Também é importante considerar que tais mudanças podem influenciar no produto final, porém, tais especificações são um bom ponto de partida, assumindo o papel de um exemplo a ser reproduzido no contexto apropriado para colocar a prova o aprendizado obtido durante a graduação.

4.2.2 Diagramas Auxiliares

Conforme DEVMEDIA (2012) recomenda, o uso de diagramas UML podem ser considerados artefato para desenvolvimentos de *software*, sendo que o diagrama de casos de uso, documenta o que o sistema faz do ponto de vista do usuário, facilitando a compreensão do sistema para todos os leitores.

O diagrama de caso de uso exibido, representado pela Figura 5, busca elucidar o principal ator do sistema, o representante comercial, representado por `Vendor`.

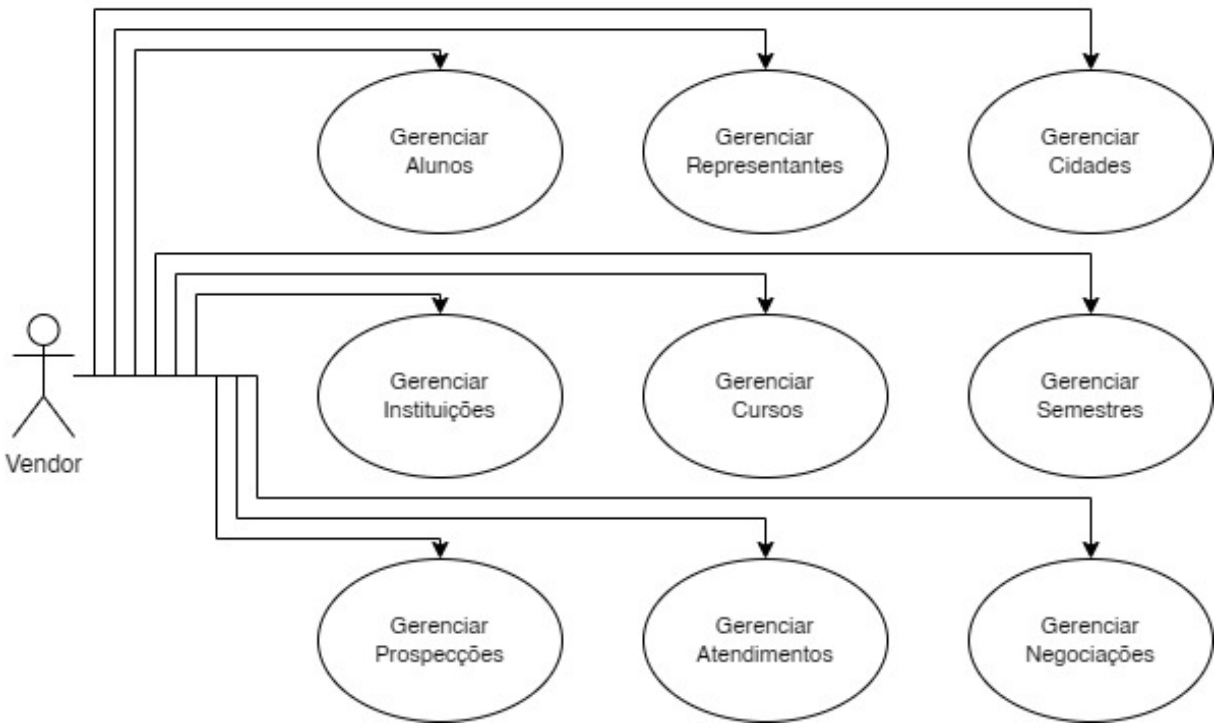

Figura 5 – Diagrama de Casos de Uso
Fonte: Autoria Própria

Apesar do diagrama parecer simples e representar que o ator `Vendor` pode fazer praticamente tudo, existem algumas considerações que precisam ser abordadas. Note, que ao se referir ao termo 'Negociação', o diagrama esta se referindo a uma turma, uma vez que a própria turma representa a negociação. Note também, que o `Vendor` não possui permissão para criar novas turmas. Isto deve-se ao fato de que as turmas são geradas automaticamente, baseadas nos semestres disponíveis e seus respectivos cursos ativos, comportamento que será detalhado nos tópicos de desenvolvimento *web*. Desta forma, o gerenciamento da turma fica sob responsabilidade da aplicação, de forma que um representante pode alterar as informações contidas na turma, mas não alterar a turma como objeto.

Outro detalhe importante que devido sua simplicidade não é representada no diagrama, consiste na regra de negócio em que qualquer pessoa poder realizar o cadastro de uma nova prospecção. Este requisito tem como objetivo facilitar a geração de novos clientes, visto que os próprios alunos poderiam acessar o formulário e solicitar atendimento, contudo, apenas o representante pode gerenciá-la.

É importante frisar que os comportamentos de um `Vendor`, são prioritariamente os representados no diagrama, entretanto, eles também assumem o papel de usuário do sistema, etapa que será abordada nos próximos tópicos.

4.2.3 Estrutura de Dados

O diagrama de classes, segundo DEVMEDIA (2016), é útil para classificação dos objetos que serão usados na abstração do problema. Para uma melhor compreensão do sistema proposto, o diagrama a seguir, representado pela Figura 6, representa a estrutura de dados do sistema, assim como seu relacionamento e comportamentos de determinadas entre as classes.

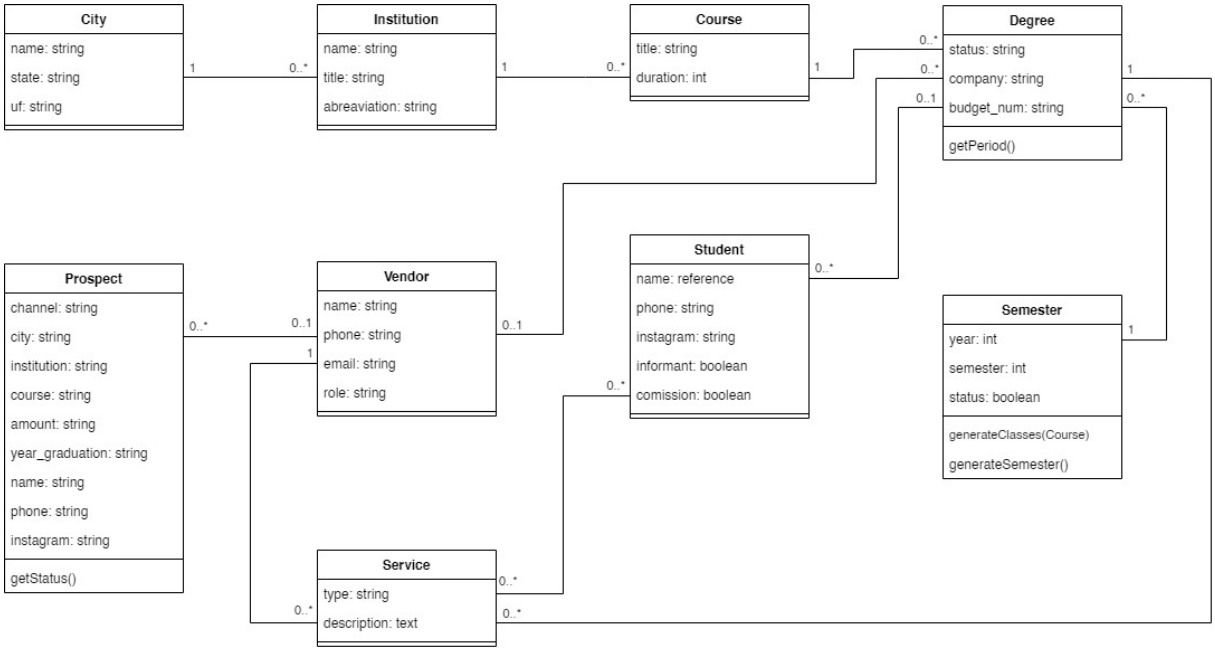

Figura 6 – Diagrama de Classes
Fonte: Autoria Própria

Ao interpretarmos o diagrama, é possível contemplar a forte agregação das classes `Vendor`, `Degree` e `Student` com as demais classes. Isto se deve ao fato de que estas classes são relacionadas por referências diretas de múltiplas instancias das demais classes, de modo que uma turma, possui diversos alunos e é negociada com um vendedor, assim como um vendedor pode registrar o atendimento à uma turma, e vincular vários alunos ao mesmo.

Também é possível notar que algumas classes possuem métodos próprios como é o caso das classes `Prospect`, `Degree` e `Semester`, funções que representam o comportamento da classe quando requisitado pelo servidor. Conforme citado anteriormente, o detalhamento destas funções, serão listadas no tópico de desenvolvimento *web*.

De acordo com o diagrama de classes representado na Figura 6 é possível estabelecer as diferenças em relação ao protótipo funcional desenvolvido anteriormente como por exemplo a ausência do vínculo direto que listava os arquivos do Google Drive dentro do aplicação, a remoção de algumas colunas específicas como as que armazenavam o caminho das imagens contidas no protótipo e várias colunas virtuais que eram utilizadas para concatenar textos, agregar números ou referênciar imagens na aplicação. Estas mudanças se devem ao fato da ausência de um *framework*, no caso, a remoção do AppSheet.

4.2.4 Desenho de Telas

Quando se trata em questões de *layout*, existem uma infinidades de variações possíveis de se exibir as informações, assim como critérios de listagem e ordenação de dados, principalmente quando estes dados representam o mapa de atuação do setor de vendas de uma empresa de formaturas.

Por exemplo, qual a melhor maneira de se elaborar uma campanha de venda, listando as turmas agrupados por semestres ou agrupados por status da negociação ou ainda por cidade? É possível que cada pessoa considere um modo melhor que outro, portanto, é natural que o sistema seja dinâmico e permita ser utilizado de acordo com a necessidade do usuário de forma intuitiva.

Protótipos são versões interativas dos *wireframes*[3], e conforme afirma Teixeira (2014), com o protótipo em mãos, é muito mais rápido conseguir colocá-lo na frente de usuários reais para testar o produto. Em alguns casos, você pode até mandar o link do protótipo para alguém testar remotamente, coletando *feedback* quase imediato sobre aquilo que está sendo desenhado.

Conforme a Figura 7 demonstra, o usuário através do menu 'Tabuleiro', que representa todas as turmas de cada curso, de acordo com os semestres listados, podendo-se realizar um filtro através de uma barra de navegação, a qual apresenta os dados agrupados por cidade, instituições, e semestres de formatura. Na disposição principal, é possível perceber que o padrão da planilha de mapeamento comercial foi adaptado para o *layout* de *cards*[4].

[3]Wireframe podem ser definidos como um esqueleto, um rascunho, um protótipo ou uma versão bastante primitiva do visual de um projeto

[4]Cards são representados por conteúdo individual altamente coerente, divididos em pequenos blocos Serradas (2018).

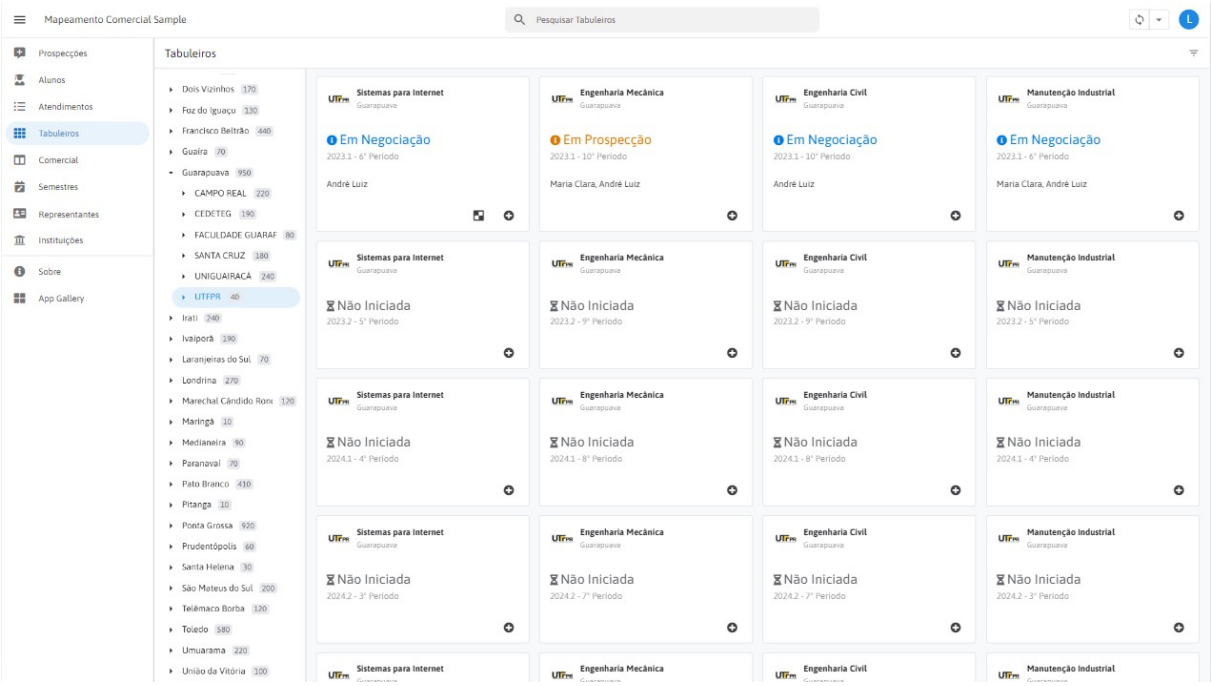

Figura 7 – AppSheet *desktop* - Tela de exibição de "Tabuleiros"
Fonte: Empresa Colaboradora

As figuras listadas neste tópico, foram capturadas a partir do protótipo funcional desenvolvido com AppSheet, onde a partir dele, conforme representado na Figura 8, é possível constatar que existe uma grande limitação de informação, considerando seu principais aspectos, como por exemplo os elementos de cada cartão, isto é, ao topo, possui apenas três variáveis, imagem, título e subtítulo, em seu conteúdo principal, novamente três variáveis, título, subtitulo e descrição, e ao rodapé, o menu de botões limitado a quatro opções apenas, desta forma, desenvolver uma aplicação robusta em um *framework* limitado pode ser considerado um ponto importante a medida que a aplicação cresce.

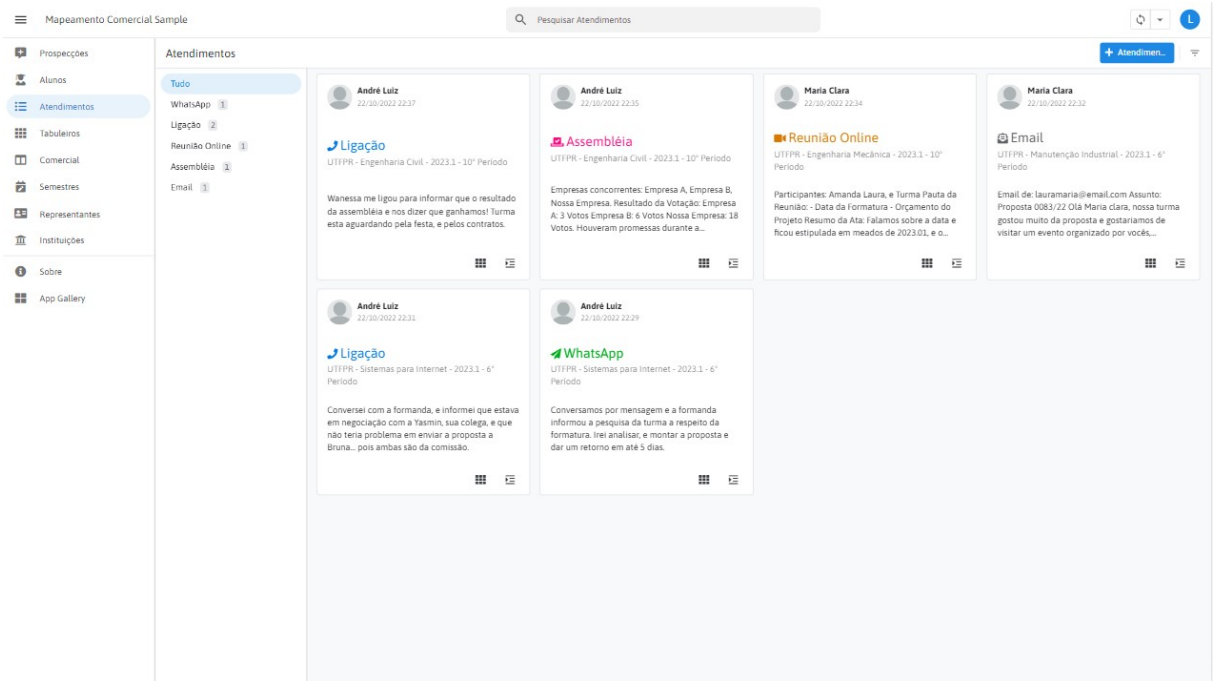

Figura 8 – AppSheet *desktop* - Tela de exibição de "Atendimentos"
Fonte: Empresa Colaboradora

É possível afirmar que ainda se utilizando a mesma estrutura de *layout*, porém em uma aplicação *web* de código aberto, seria possível incluir mais campos, além de ser possível a utilização de `hovers` e `tool-tips` como recursos para facilitar a leitura das informações, recursos que a plataforma utilizada em sua versão de protótipo funcional não oferece.

Ainda na Figura 8 é possível perceber a semelhança entre o *layout* da tela e seus filtros, isto se deve ao fato da plataforma oferecer layouts modulares, onde, caso não fosse formatado condicionalmente tornaria o uso da aplicação mais difícil, confundindo facilmente o usuário devido a semelhança das telas.

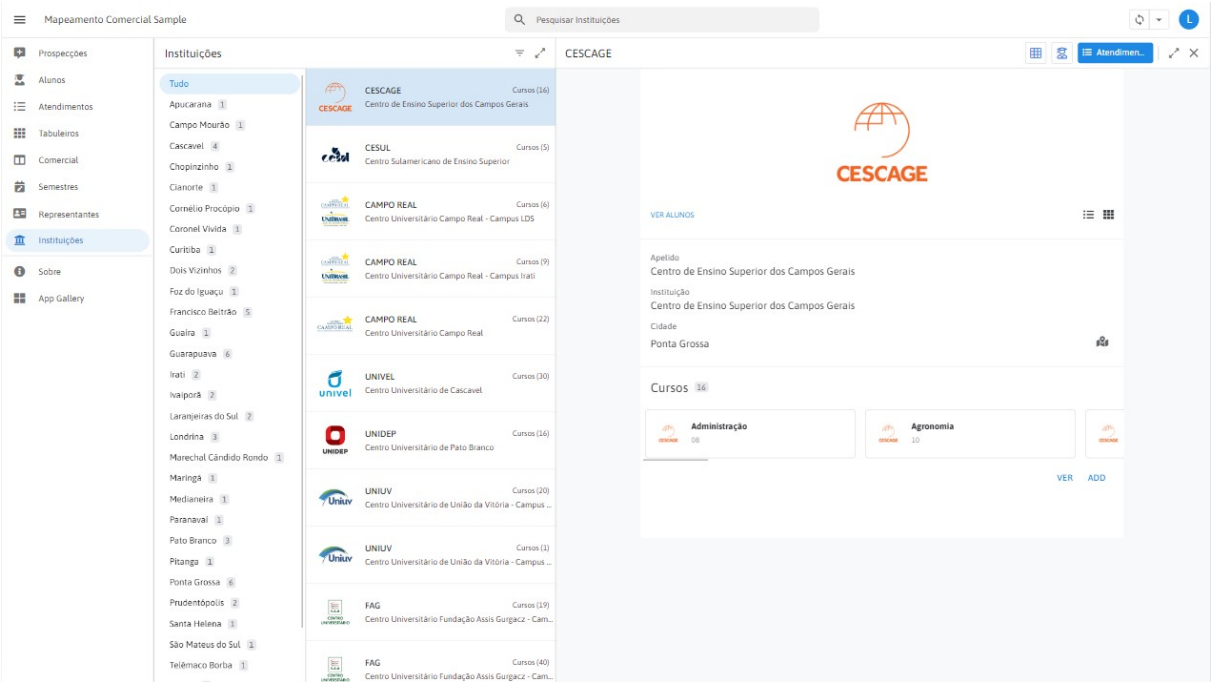

Figura 9 – AppSheet *desktop* - Tela de exibição de "Instituições"
Fonte: Empresa Colaboradora

Um aspecto interessante da aplicação, representado pela Figura 9 disponibilizada pela plataforma é a concatenação de telas, podendo exibir detalhes de registros filtrados sem ser redirecionado para outra página, algo semelhante a uma caixa de e-mail, onde os e-mails são exibidos sem que você seja redirecionado da sua caixa de entrada.

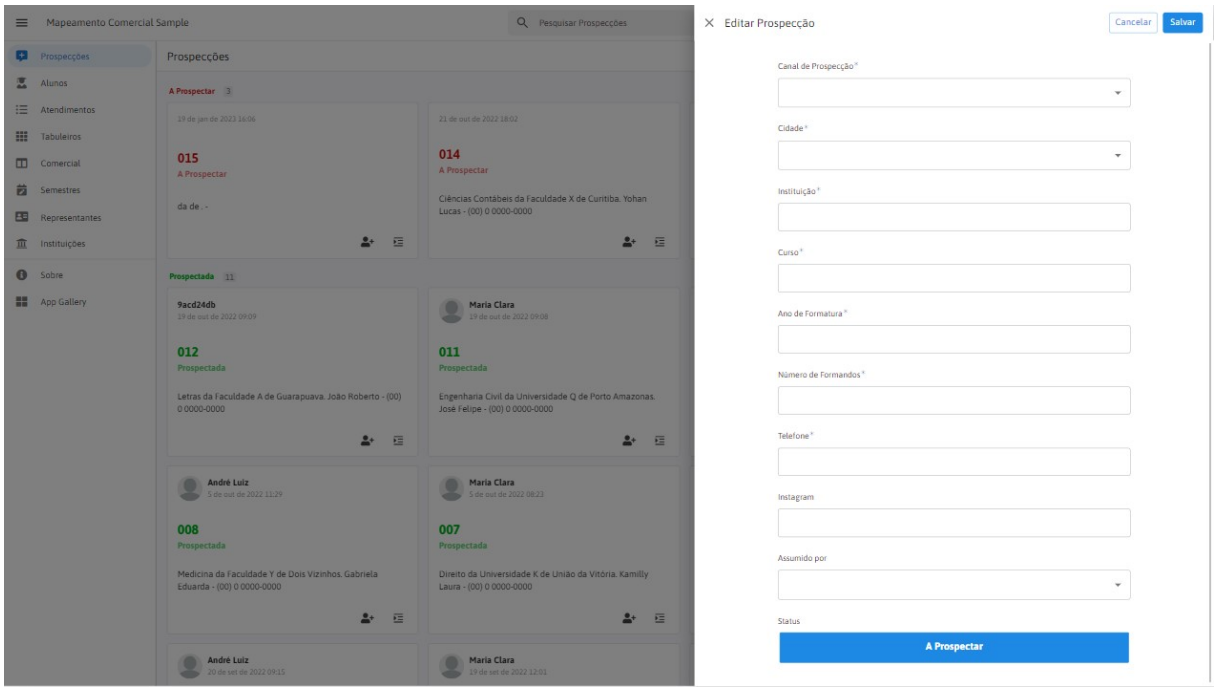

Figura 10 – AppSheet *desktop* - Tela de Exibição de "Formulário de Nova Prospecção"
Fonte: Empresa Colaboradora

O mesmo acontece com o formulário de nova prospecção sendo exibida em um

componente sobreposto, conforme representado na Figura 10, perceba, que o formulário está em branco, mas já existe um cartão (prospecção 015) criado, vinculado diretamente com o formulário.

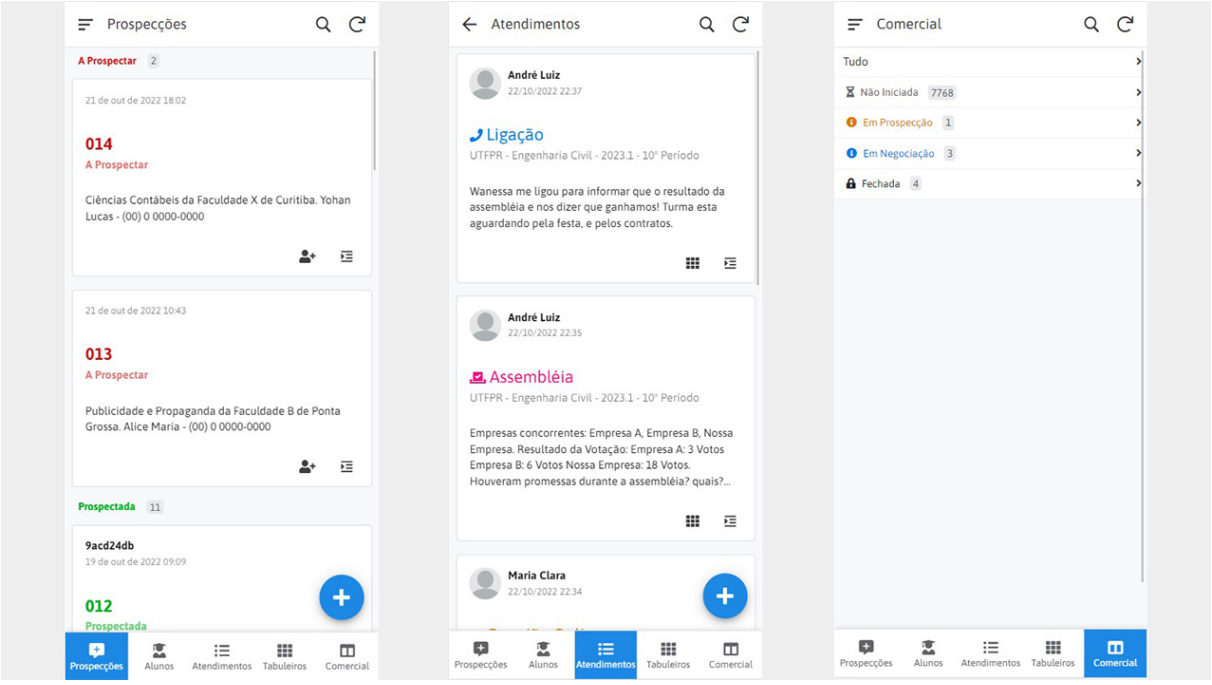

Figura 11 – AppSheet *mobile* - Tela de Exibição de "Prospecção", "Atendimentos", "Comercial"
Fonte: Empresa Colaboradora

É possível observar que tanto a Figura 11 quanto a Figura 12, representam a navegação da aplicação porém na *interface mobile*, demonstrando a versão responsiva da aplicação.

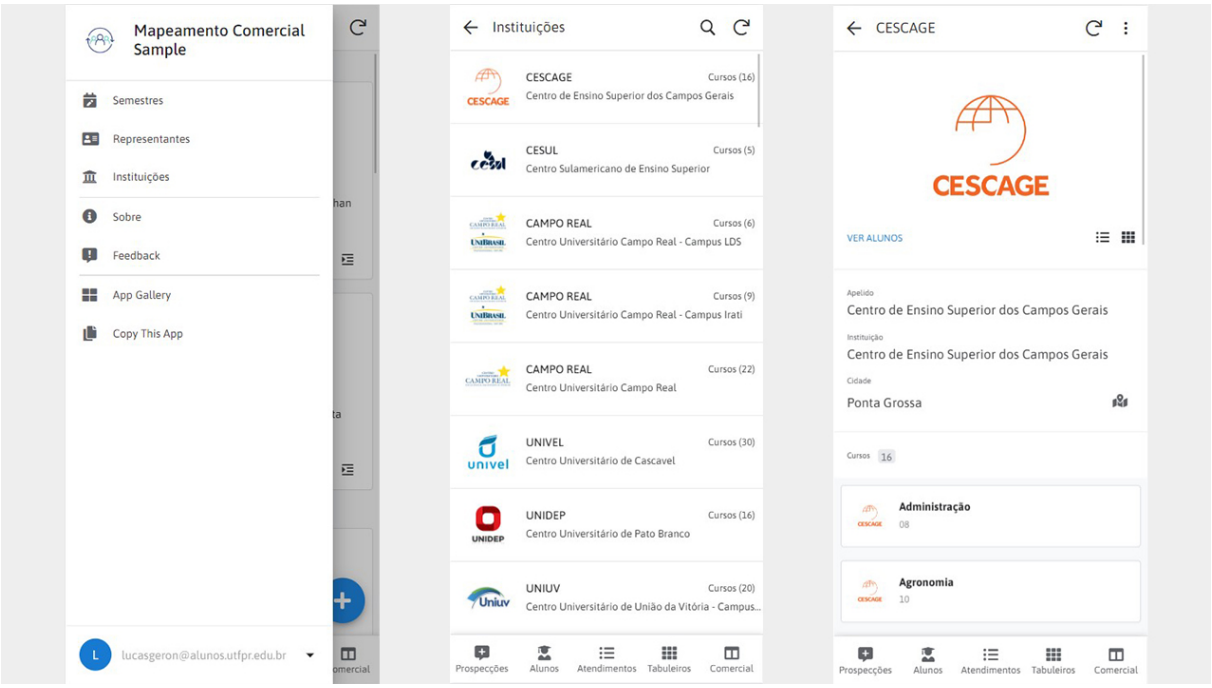

Figura 12 – AppSheet *mobile* - Tela de Exibição de "Menu Geral", "Instituições", "Detalhe da Instituição"

Fonte: Empresa Colaboradora

Portanto, a partir das telas geradas através do protótipo funcional da aplicação desenvolvido na plataforma AppSheet, o mesmo conceito presente nas telas tentará ser mantido, porém, com aperfeiçoamentos que a plataforma não permite, como o exemplo citado a pouco. O sistema a desenvolvido irá replicar os mesmos padrões de *layout* nas telas que forem pertinentes e irá elaborar novas telas caso considere necessário para o melhor uso da plataforma.

5 DESENVOLVIMENTO DA APLICAÇÃO WEB

Neste capítulo, será descrito o processo de desenvolvimento da aplicação *web* que implementa o sistema de gestão de relacionamento com o cliente para empresas de formaturas. É importante destacar que o conteúdo está descrito em ordem cronológica e tem como intenção contextualizar o leitor a respeito da evolução das etapas de desenvolvimento do projeto.

Ao longo do capítulo citado também serão abordadas as mudanças que ocorreram durante a implementação do código e os principais desafios durante o processo de desenvolvimento. Desta forma, é possível elencar quatro principais etapas em relação ao desenvolvimento da aplicação, sendo elas respectivamente: **Implementação do Código**, **Implantação em Produção**, **Alterações do Projeto** e **Validações e Testes**.

5.1 Implementação do Código

O sistema em questão foi desenvolvido do zero, com pouco tempo de estudo sobre Ruby on Rails, mas com um conhecimento sólido em programação, a adaptação aos padrões do Rails foi fácil. Com o diagrama de classes, o protótipo funcional da aplicação e um ambiente de desenvolvimento adequado, o desenvolvimento da aplicação teve início.

O projeto de desenvolvimento foi criado com as configurações padrões, ou seja, sem Docker e sem PostgreSQL, ferramentas que seriam necessárias futuramente para poder implantar e compartilhar o projeto. Desta forma, a primeira etapa do desenvolvimento do projeto foi instalar todas as dependências de um novo projeto Rails e prosseguir para a criação das classes do sistema baseando-se do diagrama de classes elaborado na primeira etapa do desenvolvimento.

5.1.1 Primeiros Passos com Scaffold

O *scaffold* consiste basicamente na geração de arquivos padronizados que facilitam a realização de criação, exibição, atualização e remoção dos dados de uma aplicação, isto é, ações também conhecidas por CRUD, dos termos em ingles: *create, read, update, delete*, de modo que ao executar o comando, o *framework* irá criar os arquivos necessários de acordo com os parâmetros informados, gerando por padrão arquivos de `migrations`, `model`, `controller`, `tests` e todas as `views`.

De modo explícito, o modelo *Model- View-Controller*, MVC, especifica como é a arquitetura de dados é utilizada pelo *framework*.

> O MVC é utilizado em muitos projetos devido a arquitetura que possui, o que possibilita a divisão do projeto em camadas muito bem definidas. Cada uma delas, o Model, o Controller e a View, executa o que lhe é definido e nada mais do que isso DEVMEDIA (2013, p 1).

Vale destacar que o comando *scaffold* age de forma rápida, mas que ainda assim exige manutenção posterior, visto que os campos dos formulários são gerados todos em formato `text`, contudo, quando se faz necessário exibir algum campo de tipo diferente no formulário, é necessário configurar de forma manual. Também é possível realizar configurações adicionais antes de executar o comando *scaffold* para definir os aspectos do conteúdo que será gerado, reduzindo assim a manutenção posterior. De certa forma, utilizar o *framework* a seu favor ainda que necessite de pequenas manutenções torna o processo de desenvolvimento *web* realmente eficiente e ágil. Logo após fazer isto, foi realizado a associação dos relacionamentos dos objetos conforme representados no diagrama de classes, de modo que as classes foram associadas através do ActiveRecord[1], através de relações, `has_one`, `has_many`, `has_and_belongs_to_many` e `belongs_to`.

O ActiveRecord é descrito como um padrão de design denominado ORM (*Object-Relational Mapping*) o qual insere uma camada de abstração na aplicação que permite interagir com o banco de dados através de objetos, uma vez que a classe destes objetos representam as tabelas, e uma instância deste objeto representa uma ou mais linhas da tabela, dependendo da interação, fazendo isto de forma prática e intuitiva e sem escrever código `SQL` de forma direta.

Em seguida foi realizado as migrações geradas pelo *scaffold* utilizando o comando `rails db:migrate`. Tal comando é utilizado para gerar o esquema das tabelas no banco de dados da aplicação. Desta forma, a aplicação já possui a maioria de seus controladores, rotas de acesso pré-definidas, formulários e páginas de criação, visualização e edição criadas, além do banco de dados com suas respectivas tabelas. É valido destacar que o cronograma de desenvolvimento presente na sessão A de anexos deste documento, prevê o refinamento de toda estrutura criada, adicionando formatações personalizadas, métodos personalizados e diversas outras ações para funcionar conforme o esperado.

Antes de iniciar o refinamento do sistema e configurar o *layout* básico da aplicação, foi necessário criar um menu de navegação para tornar a aplicação usual. De acordo com o protótipo de telas, este menu deveria ficar ao lado esquerdo da tela na versão para *desktop*. É importante destacar que todo o projeto foi desenvolvido primeiramente para ser utilizado em *desktop* e, posteriormente, poderá ser adaptado para um formato responsivo. No entanto, toda a aplicação foi planejada levando em conta como se comportaria quando renderizada em um dispositivo móvel.

O menu lateral da aplicação foi criado como uma `partial`[2], dentro da pasta `app/views/layouts` e é renderizado através da requisição explicita no arquivo `application. html.erb`. Uma vez que o menu de navegação da aplicação estava pronto para uso, o próximo passo foi o desenvolvimento do *layout* base da aplicação. De acordo com a reco-

[1]"ActiveRecord é o M em MVC, que é a camada do sistema responsável pela representação da lógica e dados de negócio. O ActiveRecord facilita a criação e uso de objetos de negócio cujos dados precisam ser persistidos num banco RailsGuides (2023a) ".

[2]"Templates parciais - normalmente chamados de "partials- são outro dispositivo para quebrar o processo de renderização em pedaços menores. Com uma partial, você pode mover o código para renderizar um pedaço específico de uma resposta para um arquivo próprio" (RAILSGUIDES, 2023c).

mendação do protótipo, a aplicação deveria ser renderizada em um contêiner dividido em três segmentos horizontais sendo o segmento da esquerda reservado para o menu de navegação, o segmento do meio para o menu de filtros e segmento da direita, o principal, com a característica de se expandir dentro de seus limites e renderizar o máximo de conteúdo possível.

Com o *layout* definido, foi possível começar o refinamento dos funcionalidades do sistema conforme o cronograma. Partindo da premissa de que uma cidade é necessária para criar uma instituição e também por ser mais simples da aplicação, a seção de cidades foi o primeiro *layout* a ser refinado e em seguida foi replicado para as demais telas do sistema.

A proposta de trabalhar com *cards* busca tornar a experiência do usuário com a aplicação responsiva o mais semelhante possível, sem fazer grandes alterações de *layout* quando renderizado em diferentes dimensões. Desta forma, fazer uso de cartões que representam registros foi a base para elaborar a listagem das cidades e consequentemente dos demais modelos da aplicação. No entanto, ao incluir o menu de filtros, foi necessário realizar a primeira alteração do projeto, pois, o menu de navegação indicava qual rota o usuário estava acessando e o menu de filtro listava todas as cidades, fazendo com que a aplicação parecesse possuir um sub-menu de navegação, ao invés de parecer um filtro de cidades conforme representada a Figura 13.

Figura 13 – Layout Base - Semelhante ao Protótipo
Fonte: Autoria Própria

Desta forma a mudança proposta foi simples e consistia em alterar a posição dos segmentos para que a informação principal ficasse ao meio, mantendo a capacidade de se expandir para mostrar o conteúdo, e o menu de filtros passaria a ser disposto no último segmento, isto é, ao lado direito da tela, com uma proporção fixa, semelhante ao do menu de navegação. A alteração é elucidada na Figura 14.

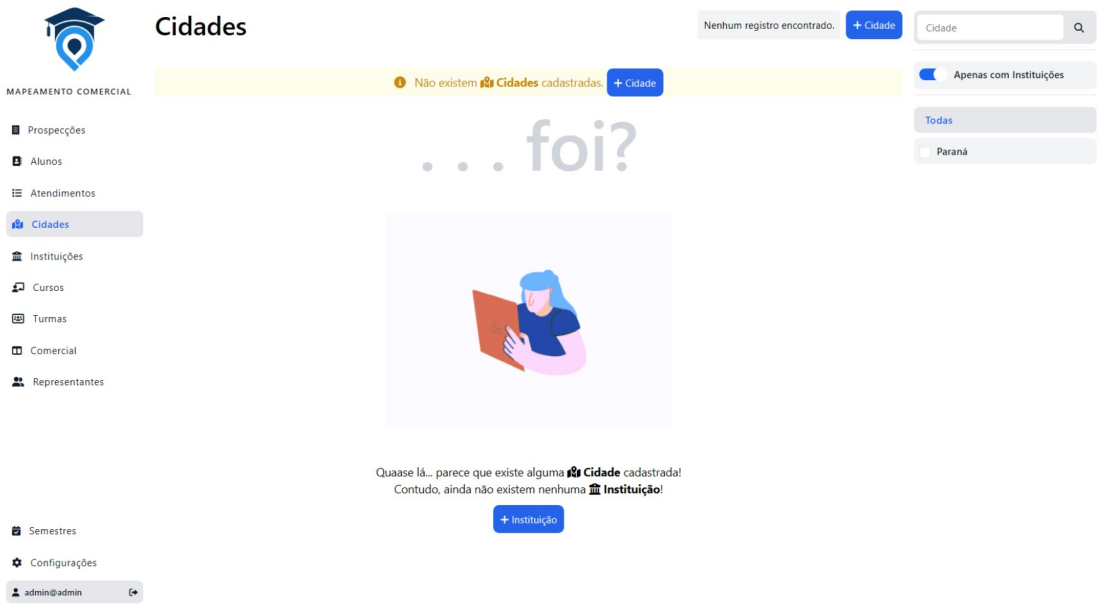

Figura 14 – Layout Base - Após Alteração
Fonte: Autoria Própria

Ainda que parece uma tarefa trivial, como simplesmente alterar a ordem das linhas em `application.html.erb`, organizar os menus para que ficassem em seus devidos lugares, respeitando seus limites sem quebrar a formatação, foi uma tarefa que levou várias horas.

Uma vez que o *layout* base da aplicação estava estabelecido, para prosseguir com o refinamento da tela de cidades era necessário adicionar registros de cidades e verificar se o *layout* estava se comportando conforme o esperado, desta forma, o próximo passo foi realizar a configuração dos `seeds`, ou seja, registros que são gerados de forma programada e populam as informações essenciais da aplicação.

5.1.2 Populando a Aplicação com Seeds

Os `seeds` de uma aplicação são considerados os registros fundamentais para que a mesma funcione. Em Rails, com a `gem` **Faker**, é possível utilizar os recursos de `seed` para criar dados fictícios no banco de dados, como nomes de pessoas, telefones, entre outros. Para manter os arquivos de `seed` deste trabalho organizados, foi criado um subdiretório `db/seeds` que armazena o `seed` responsável por cada modelo do sistema. Já o arquivo `db/seed.rb`, foi alterado para fazer as requisições dos `seed` de forma ordenada através da diretiva `load "#{Rails.root}/db/seeds/city.rb"`. O mesmo se repete para os demais modelos, entretanto, cada modelo possui suas próprias especificações e nem todos podem ser gerados de forma randômica ou com dados aleatórios. Sendo assim, para popular o banco de dados inicial desta aplicação, foi utilizado a mesma base de dados da empresa colaboradora a qual fora previamente elaborada através da pesquisa de informações ainda na etapa de desenvolvimento do protótipo.

Por meio de uma lista de cidades em que a empresa colaboradora prestava atendimento, foi realizada uma pesquisa para mapear todas as instituições de ensino superior de educação presencial, bem como seus cursos e duração. Durante o levantamento de informações, instituições com mais de uma unidade e que atendiam em cidades vizinhas, que não pertenciam à lista inicial, foram adicionadas, mas as demais instituições da cidade não. Também foi levantada uma lista de todas as cidades do Paraná e Santa Catarina, visto que inicialmente o sistema teria a intenção de atuar apenas nesses estados, as cidades portanto deveriam estar disponíveis na sessão de cadastro de novas cidades em formato de lista de seleção. Com os resultados desta pesquisa em uma planilha de Excel, preparar o arquivo de `seed` foi uma tarefa fácil. No entanto, devido ao relacionamento entre as classes, ao elaborar o arquivo `seed` das instituições, onde era preciso informar a cidade, foi necessário utilizar os recursos de ActiveRecord para localizar o registro das cidades criadas anteriormente. O mesmo processo se repetiu para o `seed` de cursos, onde era necessário encontrar o registro da instituições mantedora do curso.

Já o `seed` de semestres, por ser dinâmico, foi gerado através de um laço duplo de repetição, sendo que o laço de repetição externo representava o ano, e o laço interno os semestres. Essa mesma lógica também foi replicada para o `seed` de turmas, utilizando o laço de repetição externo para representar os semestres e o laço de repetição interno para representar os cursos de todas as instituições. Desta forma, devido a dependência de modelos associados aos demais arquivos, os `seeds` precisam ser executados de forma ordenada baseados na lista:

```
db/seeds/city.rb          # não possui relação direta
db/seeds/institution.rb   # pertence obrigatoriamente a uma cidade
db/seeds/course.rb        # pertence obrigatoriamente a uma instituição
db/seeds/semester.rb      # não possui relação direta
db/seeds/vendor.rb        # não possui relação direta
db/seeds/prospect.rb      # pode ou não possuir um representante
db/seeds/degree.rb        # pertence obrigatoriamente a um
                          # curso e semestre,  pode ou não
db/seeds/student.rb       # pode ou não pertencer a uma turma
db/seeds/service.rb       # pertence obrigatoriamente a uma turma,
                          # pode ou não possuir vários alunos
```

É valido destacar que para as demais classes como `Vendor`, `Prospect`, `Student` e `Service`, o arquivo de `seed` foi gerado fazendo uso da `gem` **Faker**, gerando dados fictícios para a maioria dos campos. Desta forma, a aplicação já apresentava registros em todas as suas telas. Para confirmar se tudo realmente estava organizado, mesmo sabendo como acessar a base de dados diretamente pelo console do Rails, o uso da ferramenta TablePlus facilitou a conferência e revisão dos dados criados.

Na edição do formulário de cidades, como o arquivo `seed` já possuía todas as cidades da região do PR e SC, foi necessário apenas adaptar o formulário alterando os parâmetros do Form

Helpers, recurso do *framework* Rails que auxilia na criação de formulários, de `form.text_field` para `form.select` para todas as cidades cadastradas. No entanto, um *input* do tipo `select` com muitas opções não é considerado uma boa prática, pois além de limitar o uso do sistema apenas nos estados do PR e SC, deixando o usuário selecionar apenas uma cidade que estivesse pré-cadastrada, tornava a busca por uma cidade difícil, pois apresentava muitos registros em único campo. Além disso, esta prática pode implicar em uma tabela com vários registros, mas que provavelmente seja pouco explorada, pois dificilmente serão criados dados relacionados para cada registro desta tabela. Desta forma, para evitar o atraso do cronograma, esta situação foi adicionada à uma lista de correções e o desenvolvimento das demais telas continuou conforme o previsto.

Com o avanço do desenvolvimento, o refinamento das outras telas foi aprimorado, adicionando informações importantes na visualização, como a adição de ícones, cores e informações ordenadas e categorizadas. É importante destacar que, até o momento presente, a aplicação apresenta todos os resultados do respectivo modelo em uma única tela. Apesar disto, a maioria das páginas não apresenta lentidão para listar as informações, com exceção de turmas, que apenas com a base de dados atual, com os cursos cadastrados e apenas dois semestres mapeados, somam mais de 1.400 registros, os quais realizavam um série de consultas complementares para exibir os dados de forma legível ao usuário. Desta forma, para carregar os registros de Turmas, a aplicação realizava consultas complementares as tabelas de Cidades, Representantes e Cursos, ocasionando a lentidão relatada. Portanto, as consultas eram agrupadas com outras requisições, gerando uma infinidade de consultas ao banco de dados e refletindo diretamente na velocidade de carregamento das informações na *view*.

Além disso, algumas funções como `time_ago_in_words` ou a função `getPeriod()` da classe `Degree` afetavam diretamente o desempenho da aplicação durante o carregamento dos dados. Esses pontos, assim como o formulário das cidades, também foram adicionados a uma lista de correção junto com situações de alto consumo de memória para serem revisados em um segundo momento. Vale ressaltar que até este momento, a aplicação apresenta todas as telas pré-configuradas e uma base de dados robusta gerada propositalmente para o ambiente de desenvolvimento, contudo, a aplicação ainda não possuí nenhum comportamento específico como filtragem de dados, validação dos campos do formulário ou dos atributos do modelo ou recursos de segurança como autenticação.

As outras classes seguiram o mesmo processo de desenvolvimento, sendo replicadas do modelo construído nas cidades, onde os cartões eram adaptados para exibir dados relevantes de acordo com o modelo. Em seguida, alterações na página de visualização detalhada, criação e edição eram realizadas. Ajustes do formulário que ainda era utilizado tanto na página de detalhes quanto na página de criação e edição e por fim, a elaboração da barra de filtros baseada no contexto do modelo em questão.

5.1.3 Filtragem de Dados de Modo Funcional

Uma das principais utilidades de um sistema *web* deve ser a capacidade de listar registros baseado em critérios especificados, contudo, até este momento, o menu de filtros implementado no projeto atual era capaz de filtrar apenas os registros baseado em seus atributos de forma exata, como por exemplo, filtrar as cidades que possuam o estado desejado, contudo, ainda não é possível pesquisar uma cidade através de um texto específico.

Portanto, para implementar os recursos de pesquisa de modo funcional, foi necessário declarar a rota aninhada à instrução `recoures :cities` no arquivo `routes.rb`, incluindo a rota `search` dentro da coleção. Desta forma, o Rails cria e direciona a requisição do usuário para o `city_controller.rb` na ação `search`, que por sua vez filtra os dados de acordo com os parâmetros informados e pode renderizar uma página própria ou redirecionar o usuário para a mesma página que fez a requisição, porém com os dados atualizados.

Baseado nesta estrutura, o Rails permite reutilizar códigos HTML de forma adequada e nos permite replicar o mesmo comportamento para os demais menus de filtros do sistema, possibilitando a edição do método `search` para cada modelo. Além disso, cada menu possui critérios distintos, onde alguns devem ser agrupados, outros devem ser de critérios exclusivos e outros devem possibilitar pesquisar com múltiplos parâmetros. Por exemplo, as prospecções com status prospectadas de dois ou mais representantes específicos.

Neste ponto, é interessante destacar um comportamento implementado nos filtros, onde ao fazer uma busca apenas por um texto, todas as caixas de seleção são marcadas, indicando que a pesquisa está sendo feito com base em todos os critérios. O que parece ser um comportamento esperado, se não implementado, torna o uso da aplicação estranha a ponto de ser considerada uma falha de sistema.

O mesmo comportamento é replicado se a pesquisa não encontrar nenhum registro como resultado. É importante notar que esta pesquisa pode ser vista de duas formas: o usuário pode selecionar um filtro e, em seguida, remover o mesmo filtro, desejando voltar para a página inicial de sua rota e não necessariamente consultar todos os registros, que é o que acontece.

O inverso também pode ocorrer, tornando-se algo incoerente ter que marcar todas as caixas de seleção do filtro para consultar todos os registros. Por exemplo, considere uma empresa com mais de 50 representantes, selecionar manualmente todas as opções seria inviável. Portanto, a ação de selecionar um critério e em seguida remove-lo para filtrar os dados é considerado uma ação intencional do usuário e não exibe nenhuma mensagem de aviso, pois não é considerado um problema, mas sim o resultado de uma busca.

5.1.4 Mudança Estrutural: Criação de Empresas

Com o desenvolvimento da aplicação, uma percepção não prevista se fez presente: no protótipo, a classe `Degree` possuía um campo para informar qual empresa havia fechado a negociação , porém, as empresas eram definidas diretamente no AppSheet, assim como a

lista de status para as negociações e os tipos de atendimentos. Estes dados não eram alocados na planilha pois eram considerados informações estáticas, eles eram especificados em uma coleção de valores, também conhecidas pelo termo em inglês `Enum`, a tradução desta abreviação significa "enumerado" , e consiste em uma lista de valores predefinidos. Porém, em uma aplicação *web*, essas opções poderiam ser personalizadas de cliente para cliente. Contudo, apesar destas sugestões, apenas as empresas foram de fato abstraídas como uma nova classe da aplicação, sendo posteriormente gerada através do comando scaffold.

O status das negociações e as categorias de atendimentos foram considerados padrões do sistema até o momento, não podendo ser alterados, pois serão utilizados em diversas views com formatação condicional. Alterar tal estrutura implicaria na alteração do escopo do projeto, sendo necessário criar estruturas para gerenciar ícones, cores, ordem de listagem, e diversos comportamentos que poderia atrasar o cronograma de desenvolvimento estabelecido.

Quanto ao menu de navegação, não seria viável adicionar o menu de 'Empresas' de forma fixa, afinal, esta informação pertence mais ao grupo de configurações do sistema do que a uma funcionalidade do sistema. Desta forma, um novo controlador denominado `settings_controller.rb` foi criado com uma única ação definida, que exibia a página de configurações da aplicação, localizada em `views/settings/index.html.erb`. Esta, por sua vez, realizava chamadas de renderização de `Company`, classe que representa as empresas, aproveitando novamente a reutilização de código de uma aplicação Rails de forma eficaz e com pouco código.

5.1.5 Corrigindo Comportamentos Inadequados

Com a aplicação parcialmente desenvolvida, a lista de correções e melhorias de desempenho começou a ser implementada. Em um primeiro momento, foi necessário corrigir a situação das cidades, que ou deixava o usuário cadastrar uma cidade sem validação alguma ou carregava uma lista com todas as cidades registradas no banco de dados, neste caso, cidades dos estados do PR e SC, contudo, sem permitir a adição de novos municípios à base de dados.

Para auxiliar com este problema a `gem` **thecodecrate/city-state** foi integrada ao projeto. A biblioteca traz em forma de métodos a consulta de regiões do mundo todo através de parâmetros simples. Desta forma, após a gem estar devidamente configurada, foi estabelecida a comunicação entre a ação do `select` do formulário com uma requisição AJAX que atualizava o campo de cidades, exibindo apenas as cidades do estado selecionado. Inicialmente, tudo parecia funcionar conforme esperado. Havia várias cidades e estados do Brasil. Durante os testes manuais, tudo correu bem, sem apresentar falhas ou comportamento indevido.

Dando sequência a lista de correção, o próximo problema tratava do relacionamento de `Service` com `Degree` e `Student`, visto que um atendimento precisa ser realizado para uma turma e um aluno poderia estar relacionado com o atendimento em questão. Entretanto, para o ActiveRecord, isso representa apenas um campo de `ID`, ou seja, no formulário que o usuário preenchia, este campo era representado por um `select` onde cada opção representava

um aluno, gerando um problema semelhante ao descrito nas cidades, e também dificultando a seleção de múltiplos alunos para o mesmo atendimento uma vez que o `select` carregava todos os alunos da base de dados e não apenas os alunos da turma.

Semelhante a solução das cidades, a alternativa foi elaborar uma pré-filtragem de dados através de mais três campos, sendo eles respectivamente cidade, instituição e curso, cada um populando respectivamente seu sucessor através de requisições AJAX escritas em JavaScript a fim de trazer uma lista de opções adequadas e com dados filtrados. Desta maneira, as opções disponíveis no formulário seriam organizadas de forma coerente e por sua vez, mais assertivas.

5.1.6 Formulários Dinâmicos com Hotwire

Diferente da `gem` utilizada para o gerenciamento das cidades que foi integrada ao projeto, o comportamento esperado para filtrar os alunos de uma determinada turma em um atendimento precisou de implementado manualmente. Para isto, foi implementado uma rota para consultas que retornam dados em `JSON`, simulando uma `API` dentro da própria aplicação. As rotas estipuladas como `/ajax/:model/:value` foram inseridas e direcionavam cada rota para o `controller` do seu modelo respectivo, evitando assim a criação de um `controller` único para AJAX. Vejamos o trecho de código elucidade na Figura 15.

```javascript
import "@hotwired/turbo-rails"
import "controllers"
import "@rails/request.js"

$(document).ready(function () {

    $("#cities_select").change(function () {
        $.ajax("/ajax/institutions/" + this.value)
            .done(function (data) {
                var htmlOptions = "<option value=''>Selecione uma instituição </option>"

                for (let i = 0; i < data.length; i++) {
                    const element = data[i];
                    htmlOptions += "<option value='" + element.id + "'>" + element.abreviation + "</option>"
                }

                $("#institutions_select").html(htmlOptions)

                // CODE TO RESET NESTED SELECTIONS (INSTITUTIONS AND COURSES)
                $("#courses_select").html("<option value=''>Selecione um Curso</option>")
                $("#degrees_select").html("<option value=''>Selecione uma Turma'</option>")

            })
            .fail(function () {
                alert("error on cities_select.change()");
            });
    });

    $("#institutions_select").change(function () {···
    });

    $("#courses_select").change(function () {···
    });

    $("#degrees_select").change(function () {···
    });

});
```

Figura 15 – app/javascript/application.js
Fonte: Autoria Própria

Toda vez que uma opção fosse atualizada, todos as demais sofreriam alterações, seja gerando opções válidas para o campo a ser filtrado ou removendo todas as opções disponíveis. Isso novamente pareceu funcionar normalmente, com uma única ressalva: só funcionava corretamente caso o usuário recarregasse a página do formulário, forçando a página a recarregar todo o seu conteúdo.

O funcionamento do *framework* Rails trabalha de uma maneira diferente com arquivos JavaScript, de forma que, através do Turbo Links, recurso que melhora o desempenho da aplicação através do comportamento onde o cabeçalho das páginas é carregado apenas uma vez e ao navegar na aplicação, o conteúdo é alterado de forma dinâmica através de requisições turbo, tornando o desempenho da aplicação mais leve e rápida uma vez que não necessita recarregar toda estrutura de estilos e funções JavaScripts. Contudo, em casos específicos como este, onde um determinado código JavaScript precisa ser carregado, é recomendado fazer isto com os próprios recursos do *framework*.

Desta forma, mediante pesquisas a respeito do *framework*, foi possível compreender que o Rails já possui recursos AJAX implementados, não sendo necessário realizar toda a

configuração manualmente conforme havia sido feito. Além do mais, o jeito recomendado em Rails 7 de se fazer um formulário com `selects` dinâmicos era utilizar o Hotwire, através dos recursos de Turbo.

O Hotwire usa uma abordagem que, em vez de enviar `JSON` para a aplicação renderizar, envia o código HTML pronto para a aplicação, tornando a renderização dos dados muito mais fluida e automática. Atualmente, o Hotwire é composto por dois pacotes principais: o Turbo, que é considerado o núcleo da aplicação, e uma biblioteca JavaScript denominada Stimulus, que busca complementar a biblioteca com comportamentos originados de ações executadas na aplicação.

O Hotwire Turbo, por sua vez, é segmentado em quatro modelos, sendo respectivamente Turbo Drive, Turbo Frames, Turbo Streams e Turbo Native. Desses, o Turbo Drive acelera links de envios de formulários, atualizando apenas o que é necessário e também fazendo o controle de navegação, como por exemplo o usuário voltar à página e ainda possuir as informações preenchidas. O Turbo Frames segmenta as páginas em contextos diferentes, permitindo que a aplicação carregue a página em pedaços, otimizando o desempenho do carregamento de páginas com muitos dados e permitindo que os elementos se comportem como componentes, sofrendo atualizações de maneira separada. O Turbo Stream, o qual iremos abordar mais a fundo, permite retornar HTML diretamente às respostas dos formulários, eventos enviados pelo servidor, também conhecido como *Server-Sent Events* (SSE), que é o caso, ou através de *Websocket*. E o Turbo Native permite realizar transições perfeitas em sua aplicação *web* para aplicações IOS ou Android.

Além destes a biblioteca Stiumulus, a qual também será abordada, conecta o código HTML ao código JavaScript através de um `data-controller`, que faz uso de `data-actions`, `data-targets` e `data-values` para se comunicar e interagir de forma direta com o `DOM` da aplicação.

Portanto, o Hotwire consegue agregar na aplicação o recurso de *selects* dinâmicos e usa o Stimulus para enviar estímulos ao controlador, que por sua vez, retornam código HTML para o elemento alvo. Vejamos com detalhe como ele funciona neste projeto e como ele elimina praticamente toda a implementação JavaScript apresentada na Figura 15.

Para o HotWire funcionar corretamente, é necessário criar um `controller` que irá gerenciar os métodos a serem implementados. O arquivo padrão gerado pelo Rails situado no caminho `app/javascripts/controllers/hello_controler.js` é um ótimo exemplo de como o HotWire deve ser implementado, desta forma, foi possível alterar o arquivo existente em vez de criar um novo.

Após isto, é preciso determinar qual elemento HTML terá conexão com este `controller`, com a importância de que todos os elementos filhos do elemento que representará o `controller`, terão acesso ao mesmo. Caso estejam fora do escopo do elemento, os mesmos não conseguiram acessar e nem serem visto pelo `data-controller`.

Para testar os recursos do HotWire, o primeiro foi atualizado o formulário de cidades

por ser o mais simples, em seguida, foi atualizado o formulário de Atendimentos e Alunos, formulários que empregavam o filtro dos alunos através de filtros pré-liminares. Vejamos a seguir como foi elaborado o comportamento das cidades.

O `hello_controler.js` foi renomeado para `country_controler.js`, e nele implementado a função `change()`, escrita em JavaScript que com a *gem* **requestjs**, simplifica a escrita de uma requisição AJAX e faz uma consulta à rota desejada, neste caso, `/cities/cities_filter` passando os parâmetros `target` e `state`. Esta rota por sua vez, esta declarada na coleção de rotas de `cities`, e leva diretamente ao `cities_controller.rb` na ação `cities_filter`, a qual através do ActiveRecord pesquisa os elementos baseados nos parâmetros recebidos e tem como saída a renderização de um `turbo_stream`, localizado em `views/cities/cities_filter.turbo_stream.erb`, que por sua vez, retorna a lista de cidades em HTML formatado para dentro do `target` indicado.

No código HTML, é importante associar uma `data-action` a algum elemento que esteja dentro do `data-controller="contry"`, neste caso, indicando a ação os parâmetros como `evento->controller#action`, neste caso: `data-action: "change->country#change"`. Deste modo, fica vinculado os eventos enviados pelo servidos (SSE) a ação `change` do `select`, ou seja, quando o campo em questão tiver sua opção alterada, a ação `change` do *contry_controller* será executada, que por sua vez desencadeará a ação descrita no parágrafo anterior.

Ainda no formulário, foi necessário atribuir o `data-target`, que é o alvo que irá receber o HTML gerado pelo `turbo-stream`. Para isto, a diretiva `data: {country_target: "citiesSelect"}` foi adicionada ao select de cidades, tornando o elemento um alvo que receberá os dados filtrados. Com isto, tudo funciona conforme o esperado, sem a necessidade de recarregar a página conforme reportado no problema inicialmente. O mesmo comportamento foi aplicado para os formulários de serviços e alunos, entretanto, devido ao seu comportamento ser semelhante para duas classes, não se fez necessário a criação de um controlador para cada classe, visto que os métodos serão os mesmos.

Em ambos os formulários, seria necessário informar uma turma, a qual precisava ser previamente filtrada por cidade, instituição e curso. No caso dos serviços, após definir a turma, a lista de alunos poderia ser atualizada automaticamente.Portanto, ao selecionar a cidade, a aplicação irá buscar pelas instituições presente nesta cidade. Ao selecionar a instituição, o mesmo se repetirá para os cursos da mesma. Novamente, ao selecionar o curso, a aplicação retornará as turmas disponíveis do curso. Portanto, *degree_controller.js* foi definido e implementado de forma semelhante o *country_controller.js*, porém, com seu conteúdo alusivo ao contexto do formulário em questão.

Seguindo com as correções, algumas consultas ao banco de dados foram reformuladas utilizando as funções `joins` e em alguns casos, `includes`, recursos disponibilizados pelo Rails que reduzem drasticamente o número de requisições enviadas ao banco de dados, fazendo a mesma busca de uma maneira mais eficiente ao trazer todos os dados de uma só vez, ao invés

de fazer isto em várias requisições. É importante destacar que preocupações com desempenho são fundamentais para uma boa aplicação. Conseguir otimizar o sistema com componentes dinâmicos através de Hotwire e outras bibliotecas sem dúvidas traz benefícios tanto em agilidade de desenvolvimento quanto em desempenho da aplicação.

Além dessas alterações, a lista de correções e melhorias incluía algumas tarefas simples como a tradução do sistema e algumas tarefas complexas, como por exemplo, na tela de listagem dos modelos que estavam relacionados diretamente com cidades, ao invés de exibir todos os registros na tela de listagem, uma mensagem foi adicionada solicitando que o usuário selecionasse uma cidade para então exibir os registros buscados, melhorando significativamente o desempenho da aplicação, mas implicando de forma negativa diretamente no uso da aplicação. Neste contexto, considerando que ainda que a empresa só atuasse em duas ou três cidades, seria necessário selecionar as mesmas para ver os dados, mas ainda assim era melhor do que causar lentidão na aplicação devido à consulta de todos os registros existentes na base de dados. Além desses, também foi finalizado os ajustes da página de configurações e o refinamento das páginas referente as Empresas, fazendo seus filtros funcionarem e exibindo os resultados por semestres.

A implementação da página 'Comercial' também foi realizada durante este período, visto que a mesma trabalha com os mesmos dados de `Degree`, apenas classificando-os de maneira diferente. O menu Turmas tem por objetivo exibir todas as turmas de uma determinada instituição, enquanto o menu Comercial tem por objetivo mostrar as turmas atendidas por um determinado representante e os respectivos status de suas negociações.

Outro ponto válido a destacar é que, como a listagem de empresas é feita diretamente em `../views/settings/index.html.erb`, de forma que o arquivo `index.html.erb` da pasta `views/companies` não é necessário, sendo excluído de seu diretório. Diferente do arquivo que é utilizado para renderizar uma empresa em uma coleção de empresas, `../view/companies/_company.html.erb`, que é utilizado para renderizar uma empresa em uma coleção de empresas.

Ainda durantes testes de uso do sistema e correções de forma manual, mais um problema foi identificado. A `gem` **city-state** estava desatualizada e o banco de estados e cidades do Brasil estava defasado. Também haviam muitas cidades com o nome incorreto, sem acentuação ou com a tradução indevida, como era o caso de 'District Federal', ao invés de 'Distrito Federal', ou 'Maranhao' ao invés de 'Maranhão'. Devido a isso, cidades cadastradas através do `seed` não eram localizadas na página de edição de cidades, isto ocorria pelas cidades registradas através do `seed` não estarem na relação das cidades contidas na biblioteca. Um exemplo prático disto que implicava diretamente na usabilidade do sistema era o caso 'Francisco Beltrão', que, por não possuir acento, não era carregado no formulário e, ao ser salvo, gerava uma série de erros nos filtros devido a sua acentuação.

Em uma pesquisa mais ampla, foi possível encontrar a `gem` **ConectaAddressBr** que funciona de uma maneira um pouco diferente, trazendo em sua especificação uma coleção com

todas as cidades do Brasil associadas e fornecendo métodos para extrair listagens desta matriz a partir da sigla do estado.

A substituição da `gem` foi tranquila e não gerou transtorno, necessitando apenas implementar um método complementar para remapear as opções do `select`, visto que esta `gem` retornava uma lista de **uf**, **cidade**, quando o esperado era retornar uma lista de **id**, **cidade**.

Para escrever esta função bastou utilizar os recursos da linguagem Ruby, em conjunto com as diretivas da `gem`. Portanto, para gerar uma `array` de **id**, **cidade**, foi necessário utilizar os métodos `each_with_index`, que atua em cada elemento da `array`, seguido da método `map`, que realizar uma interação em cada elemento do `array`, retornando ao fim de sua execução, uma nova `array` de elementos e seus índices. Veja o código abaixo:

```
states_for_select = ConectaAddressBr::States.all_full_names.each_with_in-
dex.map {|x, index| [x, x + ConectaAddressBr::States.all_initials[index]]}
```

É valido destacar que utilizar o índice de um `array` como id de um objeto não é uma boa pratica, pois a ordem dos elementos podem ser facilmente alteradas, entretanto, como a `gem` não recebe atualização e esta configurada para isto, não há problema em utiliza lá desta forma. Além do mais, se em um futuro fosse adicionadas mais cidades à base da `gem`, bastaria adicioná-las ao fim da `array`, em seguida, ordenar os elementos diretamente no `controller` da aplicação, evitando erros de referência. Considerando que os os dados são de 2019, para este projeto, é considerado atualizado o suficiente para ser utilizado.

Até este momento, o desenvolvimento da aplicação se encontra em fase intermediária, com toda a estrutura feita, os modelos interligados e navegação da aplicação, listagem e filtragem dos dados funcionando. O projeto poderia ser tido como pronto para uso a partir deste momento, entretanto, ainda é necessário concluir os requisitos do protótipo, adicionando a logos das instituições ao projeto e todo o painel de segurança da aplicação, tornando o acesso a informação restrita apenas aos usuários autorizados. Com isto, a aplicação também passará por uma série de refinamentos que tem por objetivo melhorar a experiência do usuário.

5.1.7 Gerenciamento de Imagens com ActiveStorage

Para realizar o *upload* de imagens na aplicação com o objetivo de incluir as imagens das instituições, a melhor opção oferecida pelo *framework* é utilizar os recursos de ActiveStorage, que semelhante ao ActiveRecord, fornece uma série de facilidade na hora de vincular imagens na sua aplicação Rails.

Para habilitar o ActiveStorage na aplicação foi necessário apenas executar o comando `bin/Rails active_storage:install` seguido do comando `rails db:migrate`, que cria as tabelas no banco de dados conforme especificações da migração gerada. Fazendo isto, por padrão, a aplicação estará direcionando o armazenamento dos arquivos para o diretório `/storage` da aplicação. Desta forma, foi necessário configurar o modelo `Institution.rb` e

informar seu relacionamento com imagens através da diretiva `has_one_attached :image`, estabelecendo que cada instituição possua apenas uma imagem associada.

Mediante esta implementação, se fez necessário mudar o formulário de criação e edição, assim como as visualizações da aplicação, para que caso exista, a imagem enviada fosse exibida. Para realizar isto, uma verificação, descrita por `institution.image.attached?` foi adicionado no arquivo `/views/institutions/_form.html.erb`. Esta verificação retorna verdadeiro apenas se houver uma imagem associada ao objeto, ou falso caso não exista vínculo. A partir do resultado desta verificação, a aplicação pode se comportar de maneiras distintas, exibindo a imagem existente e o link para exclusão da imagem caso a resposta seja verdadeira, ou solicitando ao usuário o envio de novo arquivo em um formato especifico, neste caso, aceitando apenas os formatos: `.png`, `.gif`, `.jpeg`.

É importante destacar que foi necessário criar uma rota exclusiva para realizar exclusão da imagem, a qual passa através do método `DELETE` o próprio elemento, neste caso, `institution`. Este parâmetro representa a própria instituição e a ação por sua vez, remove a sua imagem associada do elemento em questão, em seguida, redireciona o usuário para a página de detalhes, encerrando a edição da instituição, sem salvar os demais dados, apenas realizando a exclusão da imagem. Sendo assim, o envio de imagens, bem como a atualização e remoção das logos das instituições já estava configurado. Além de replicar os ajustes das imagens nas visualizações que as utilizavam, foi necessário incluir a pasta e as imagens nos `assets` do projeto, possibilitando que os arquivos de `seeds` da aplicação também incluíssem o registro das imagens definidas de forma automatizada, isto é, sem ser necessário fazer o *upload* manual das imagens ao criar os registros definidos no `seed`, portanto, no arquivo de `seed` de instituições, uma diretiva foi adicionada ao fim de cada linha, conforme o código abaixo representa.

```
Institution.create([...]).image.attach(io: File.open('../images/
seed_images/institutions/utfpr.png'), filename: 'utfpr.png')
```

Neste exemplo, a logo da instituição UTFPR é adicionada após a criação de uma determinada instituição. Perceba que as imagens referenciadas estão no diretório da aplicação em `app/assets/ images/seed_images/institutions`, bastando apenas serem referenciados aos registros que irão ser associados. Outro detalhe é que nas instruções definidas no `seed`, cada instituição precisou ter seu `filepath` e `filename` alterados manualmente, pois nem todas as instituições que são do mesmo grupo possuem a mesma logo, como é o caso da Unicentro Cedeteg e demais campus, e do Centro Universitário Campo Real de Laranjeiras do Sul com as demais unidades da rede.

Com a funcionalidade implementada para a instituição, foi possível replicar o mesmo para as empresas, atribuindo assim uma imagem para ser utilizada como logo para cada empresa. Isto torna o sistema mais representativo para o usuário. Tal ajuste foi feito rapidamente seguindo a mesma lógica de programação. Vale lembrar que o ActiveStorage ainda esta configurado para

armazenar seus arquivos localmente. Essa configuração será alterada posteriormente, antes da implantação do projeto.

5.1.8 Integração com Amazon Web Services

Um medo comum entre os desenvolvedores é colocar a aplicação em produção, pois além da mudança das variáveis de ambiente, muitas outras coisas podem acontecer durante esse processo. A aplicação também pode apresentar erros que só ocorrem em ambiente de produção. No entanto, como parte deste projeto e da qualificação profissional para realizá-lo, a implementação com os serviços de hospedagem de arquivos da Amazon é algo trivial que qualquer programador deveria saber fazer nos dias de hoje.

Para realizar esta etapa, após compreender como o serviço funciona, foi necessário abrir uma conta nos servidores de hospedagem e realizar as configurações necessárias para criar um *Bucket*, termo utilizado para se referir a uma pasta de arquivos de uma determinada aplicação. Também foi realizado a configuração das credenciais de um novo usuário com acesso às políticas de privacidade que permitiam acesso total aos arquivos do *bucket*.

Voltando ao código da aplicação foi necessário alterar o arquivo `config/storage.yml`, para incluir as variáveis de ambiente, conforme as instruções explicitas. As credencias por sua vez foram adicionadas através do próprio *framework*, sendo definidas portanto no arquivo `credentials.yml.enc` que por sua vez, é criptografado para o arquivo `master.key`, de modo que o arquivo ser acessado e lido corretamente apenas pelo *framework*.

Uma última configuração para a aplicação Rails funcionar com a nova configuração, era alterar o arquivo `config/environments/development.rb`, alterando a instrução `config.active_storage.service = :local` para `:amazon`. O mesmo se aplica ao arquivo `production.rb`. Estas instruções definem tanto no ambiente de desenvolvimento quanto de produção o destino das imagens enviadas pelo formulário.

Portanto, após reiniciar o servidor e realizar um teste manual, tudo ocorreu conforme o esperado. É importante destacar que a conta AWS precisou ser ativada para que o bucket realizasse o credenciamento dos usuários, permitindo a adição de arquivos remotamente.

5.1.9 Restrição dos Dados: Autenticação com Devise

Quase em sua etapa final de desenvolvimento, com vários recursos implementados, foi necessário realizar a configuração de segurança da aplicação para restringir o acesso das informações apenas a usuários cadastrados e autenticados.

Em uma busca rápida pela comunidade Rails, a *gem* **Devise**, fortemente recomendada para realizar o controle de autenticação do usuário foi integrada ao projeto. De fácil instalação e utilização, os primeiros passos recomendados na documentação da biblioteca foram realizados e, logo após realizar as migrações, a `gem` parecia mais complexa do que parecia, visto que seus arquivos fontes ainda não haviam sido criados dentro do projeto e estavam sendo fornecidos diretamente pela `gem`.

Executando o comando `rails generate devise:views`, todas as telas utilizadas pelo Devise são criadas dentro do projeto, permitindo personalizar as mesmas conforme for necessário. Desta maneira, com os arquivos visíveis, tudo ficou mais organizado, permitindo editar o *layout* das telas geradas pela Devise. Entretanto, ainda não estava claro como os controladores funcionavam e nem onde estavam, de modo que a forma mais fácil de compreender o funcionamento da autenticação foi através das rotas da aplicação.

Após realizar a implementação de uma classe denominada `User` conforme recomendado pela `gem`, dois questionamentos vieram à tona e puderam contribuir imensamente para o projeto. Considerando que a estrutura do diagrama de classes do projeto havia sido alterada, onde inicialmente o acesso ao sistema deveria estar atribuído a classe `Vendor`, agora estaria sendo atribuído a classe `Users`.

A partir disto uma das perguntas levantadas foi: devemos vincular obrigatoriamente um representante à cada usuário do sistema? A resposta para esta pergunta foi negativa baseada nos argumentos de que ao associar automaticamente um representante a um usuário do sistema, implicaria na quebra de várias páginas, uma vez que trariam resultados filtrados apenas pelo usuário em questão, impossibilitando assim visualizar dados de outros usuário, e também tornaria a base de dados do sistema incoerente, visto que nem todos que possuem acesso ao sistema são de fato representantes, como é seria o caso do diretor da empresa, que obviamente, também teria acesso a plataforma. A segunda pergunta, mas não menos importante, aborda qual seria exatamente a função de um usuário do sistema sem possuir atribuições exceto a de acessar as informações?

Ainda sem ter uma resposta concreta, vincular um usuário do sistema à um representante parecia algo coerente visto por outra perspectiva. Uma situação havia sido notada ao testar o sistema inúmeras vezes: a de sempre ter que selecionar uma cidade para exibir os dados iniciais. Sim, tal funcionalidade busca melhorar o desempenho da aplicação, contudo, implica diretamente na usabilidade do sistema. Portanto, a partir disto, uma possível solução seria associar uma cidade padrão à um representante para que então, o representante pudesse ser vinculado corretamente com uma conta de usuário, desta forma, os registros que dependem de uma cidade para serem exibidos, já viriam selecionados de forma automática com base na cidade do representante.

Durante o processo de desenvolvimento desta funcionalidade, novamente houve uma alteração, desta vez uma que fazia mais sentido do que a ideia original. Por que em vez de guardar apenas uma cidade padrão, não fazemos o representante possuir vínculo com várias cidades? Afinal, na prática, um representante atende mais de uma cidade.

Com este questionamento, a resposta ficou evidente: ao vincular um usuário à um representante, todos os filtros serão associados às configurações do representante, de modo que um usuário só pode ter um representante vinculado, mas o representante poderia atender várias cidades. Com isso, as cidades do representante passam a ser associadas ao usuário do sistema, tornando assim a usabilidade do sistema muito mais eficiente, uma vez que a seleção

das cidades em várias telas agora viria filtrada automaticamente através do relacionamento de representantes e suas cidades atendidas.

Implementar esta funcionalidade por sua vez requereu alterações em diversos arquivos. Inicialmente, foram alterados os controles de todas as telas que possuíam cidades em seu menu de filtros, como Instituição, Cursos e Comercial. Após realizar os ajustes nos controladores, foi necessário arrumar as validações do menu de filtros para que, além dos dados filtrados, os critérios do filtro já viessem de acordo com as configurações do usuário, levando em consideração o fato de existir cidades vinculadas ao representante, ou não.

Por fim, ainda por se tratar da área de autenticação do usuário, uma última consideração também foi colocada em desenvolvimento: a de abrir a rota de cadastro de nova prospecção, tornando assim o formulário público para os demais setores da empresa ou até mesmo para ser utilizado nas redes sociais da mesma. Tal requisito era essencial para que qualquer aluno ou empregado da empresa que soubesse de uma nova prospecção pudesse cadastrar essa informação e direcioná-la ao setor responsável.

Para restringir o acesso de um `controller` qualquer, a documentação da `gem` **Devise** recomenda adicionar a validação de autenticação através da diretiva `before_action :authenticate_user!` em cada controlador que possuir conteúdo restrito à autenticação, entretanto, esta diretiva foi adicionada diretamente no `application_controller.rb` como um todo, visto toda a aplicação, exceto as prospecções, seria restrita ao publico. Deste modo, o método `authenticate_user!` foi sobrescrito conforme a figura 16.

app > controllers > application_controller.rb > ApplicationController > authenticate_user!

```
1   class ApplicationController < ActionController::Base
2
3       before_action  :authenticate_user!
4
5       AUTHENTICATE_USER_EXCEPT = [{controller: 'prospects', action: 'new'}, {controller: 'prospects', action: 'create'}]
6
7       def authenticate_user!(opts = {})
8           # LOCK ALL CONTROLLERS AND ACTIONS EXCEPT THE ONES LISTED ABOVE
9           unless AUTHENTICATE_USER_EXCEPT.include?(controller: params[:controller], action: params[:action])
10              super
11          end
12      end
13
14      def after_sign_in_path_for(resource)
15          prospects_path # redirect to prospects page after sign in
16      end
17
```

Figura 16 – app/controllers/application_controler.rb
Fonte: Autoria Própria

O método `after_sign_in_path_for` também foi sobrescrito, redirecionando o usuário para tela prospecções após realizar o login. Desta forma a `gem` **Devise** foi integrada de forma eficaz e útil para o sistema como um todo permitindo proteger o acesso aos dados da aplicação de forma rápida e fácil, além de fornecer configurações de usuários de forma personalizada através da associação dos representantes da aplicação.

É importante destacar que a `gem` em questão permite que qualquer usuário realize o cadastro na plataforma de forma semelhante a uma rede social. Este ponto é considerado crítico

pois influencia diretamente na segurança dos dados e será abordado nos capítulos seguintes deste documento.

5.2 Implantação em Produção

Mediante o desenvolvimento atual da aplicação, a qual apresenta sua estrutura básica para funcionamento implementada, contando com o gerenciamento das coleções do sistema, gerenciamento de imagens e controle de usuário através de autenticação, é possível avançarmos para o próximo passo, a de implantação do projeto em ambiente de produção.

Nesta etapa, o mais adequado antes da implantação - também conhecido pelo termo *deploy* - seria realizar os testes e revisar o código como um todo a fim de realizar os ajustes necessários para colocar a aplicação em produção. Entretanto, conforme mencionado no cronograma, isto será feito em um segundo momento, visto que tais conceitos ainda estão sendo aprendidos no último semestre da graduação. Isso torna este problema uma grande oportunidade de fazer uma atualização da aplicação em produção no futuro.

É válido complementar este contexto com a informação de que a aplicação, mesmo em produção, não tem o objetivo de ser utilizada oficialmente neste momento. Além disso, os arquivos `seed` estão populando parcialmente o banco de dados com informações falsas. Fazer a implantação da aplicação neste momento sem realizar os testes tem como objetivo validar o desenvolvimento até o momento e verificar se tudo funcionaria conforme o planejado quando acessível para toda internet.

5.2.1 Hospedando a aplicação com Heroku

Para hospedar esta aplicação em um ambiente de produção real, uma das alternativas poderia ser a aquisição de um servidor VPS (*Virtual Private Server*) com os requisitos mínimos para rodar a aplicação, onde nele deveriam ser instalados todas as dependências do projeto, contudo, uma solução mais prática e barata acaba sendo o Heroku, uma plataforma que possui suporte para hospedar aplicações Rails diretamente de repositórios GitHub.

Sendo assim, foi realizado o cadastro na plataforma e configurado o ambiente de desenvolvimento da aplicação. Vale destacar que assim como os serviços da AWS, o serviço de hospedagem na plataforma e o banco de dados PostgreSQL são pagos, mas todos possuem valores acessíveis ao público, ainda mais quando pouco utilizados, como é este caso. Após isso, foi realizada dentro do repositório da aplicação a criação de um remoto na plataforma através do comando `heroku apps:create tcc-utfpr`, reservando o endereço **tcc-utfpr.herokuapp.com** como domínio da aplicação. Em seguida, bastou enviar os arquivos da aplicação ao Heroku através do comando `git push heroku main`. Durante estas etapas, dois problemas foram encontrados, o primeiro foi a de que o Heroku não possui suporte ao SQLite3, banco de dados padrão do Rails e a segunda referente as credencias em ambiente de produção.

5.2.2 Alteração de Banco de Dados: SQLite para PostgreSQL

Realizar a alteração de um banco de dados uma aplicação *web* manualmente seria algo custoso, entretanto, com o *framework* Rails, isto é feito de uma forma simples, bastando apenas alterar o arquivo `database.yml`, substituindo o adapter de **sqlite3** para **postgresql** e substituir o nome da base de dados por um nome qualquer, neste caso, `db/development.sqlite3` para `tcc-utfpr_development`. Também é necessário atualizar o arquivo `GemFile` substituindo a linha `gem "sqlite3"` por `gem "pg"`.

Em seguida, foi realizado a instalação das novas gems, através do comando `bundle install`. Desta forma, a aplicação já estava configurada corretamente, sendo realizado em seguida a configuração do PostgreSQL no WSL2 e suas configuração para se conectar com o TablePlus. Com a conexão bem-sucedida, bastou rodar os comandos Rails para criar, migrar e inserir os `seeds` no novo banco de dados.

Tudo ocorreu conforme o planejado de forma rápida. Após realizar a migração e acessar diversas telas, tudo parecia estar igual. A forma como o Rails permite alterar todas as configurações da aplicação de forma fácil é impressionante, de modo que organizar o *framework* de forma inteligente torna o desenvolvimento de aplicações muito mais eficiente. Vale destacar neste momento que, apesar do Rails fazer isso de forma simples, ainda foi necessário instalar o Postgres e as devidas configurações de portas de acesso para se comunicar com o Windows, para que os dados pudessem ser lidos pelo TablePlus.

Considerando portanto que qualquer outro desenvolvedor que queira contribuir para o projeto, o mesmo teria que fazer a instalação do Postgres em sua máquina local, além de gerencias as versões de Ruby e Rails para serem iguais ao da aplicação. Desta forma, o compartilhamento do projeto se torna uma tarefa complexa e complicada, portanto, a partir desta premissa, uma boa alternativa para preparar o projeto atual para ser facilmente compartilhado, seria integra-lo ao ambiente Docker. Tal proposta apresenta uma prioridade média nesta etapa de desenvolvimento do projeto, sendo adicionada a lista de pendências para futuras implementações.

Uma vez finalizada a configuração do novo banco de dados, seria possível realizar o *upload* da aplicação para o Heroku, processo que correu bem até a parte dos `seeds` das instituições. Ao ser executado, a aplicação tentou enviar requisições ao *bucket* da AWS para registrar as imagens das instituições e empresas, porém, devido a um detalhe importante, uma falha estava ocorrendo e a aplicação não estava conseguindo realizar esta ação.

5.2.3 Credenciais em Ambiente de Produção

Em um projeto Rails, um dos arquivos mais importantes é o arquivo `master.key`, que guarda as credenciais de forma criptografada. Devido à sua importância, tal arquivo é ignorado do restante dos arquivos através das especificações de `.gitignore`, ou seja, o problema com o `seed` da aplicação estava sendo ocasionado pela ausência das credenciais corretas no repositório

remoto do Heroku. Desta forma, foi necessário criar o arquivo `master.key`, que atualmente sequer existia na plataforma Heroku executado o comando a seguir.

```
heroku config:set RAILS_MASTER_KEY = "cat config/master.key"
```

Tal comando cria a variável `RAILS_MASTER_KEY` com as mesmas informações contidas no arquivo `master.key`. Após isso, mais um problema se fez presente. Parte dos `seeds` já haviam sido migrados, como as cidades por exemplo, ou seja, caso o banco de dados não fosse resetado, os registros seriam duplicados. Entretanto, é contraintuitivo resetar um banco de dados de uma aplicação em produção. Para realizar tal operação, foi necessário editá-la no painel de configurações do banco de dados da aplicação do Heroku, afirmando que estava ciente do que esta fazendo.

Após isso, com a base de dados limpa, as migrações foram realizadas novamente e ocorreram com sucesso. As imagens estavam sendo direcionadas corretamente para o *bucket* e os registros sendo salvos corretamente no banco de dados Postgres. Sendo assim, já era possível considerar que a aplicação estava oficialmente em produção.

Para testar todos os recursos, foi realizado uma série de ações, como criar registros, filtrar dados, realizar ações permitidas e não permitidas e fazer requisições a URLs não existentes. Com isso, foi identificada uma série de inconsistências, sendo a maioria provenientes da alteração de banco de dados, principalmente nos controladores onde os registros eram filtrados por um `array` de valores que incluíam algum elemento vazio. Essa ação apresenta comportamento diferente entre o SQLite e o PostgreSQL. Para solucionar este problema, bastou adicionar a função `.compact` ao parâmetro, removendo assim os elementos vazios.

Até este momento, com tudo definitivamente funcionando, o projeto pôde ser considerado oficialmente em produção. Contudo, essas alterações ainda não foram enviadas ao Heroku, sendo desenvolvidas apenas localmente para evitar o número excessivo de `commits` na aplicação em produção. É importante citar que essas alterações serão implantadas futuramente através de *pipelines* de integração contínua e implantação contínua, sendo abordadas em suas respectivas seções deste documento.

5.3 Alterações do Projeto

Ainda com um vasto universo de possíveis melhorias, diversas alterações foram surgindo durante a implementação do projeto. Neste tópico será abordado um pouco sobre quais foram as principais mudanças e como elas alteraram a aplicação desde sua especificação inicial.

Vejamos algumas alterações que foram realizadas ao longo da jornada de desenvolvimento da aplicação. A primeira grande alteração a ser considerada, foi a criação de `Company`, que representa as Empresas do sistema. As empresas por sua vez possuem como objetivo, ser associado a uma negociação após o status dela ser alterada para fechada, portanto, seu modelo é bem simples e possui apenas nome e status para caso a empresa não seja mais listada

nas opções ainda possa possua vínculo com as negociações nas quais esteja associada. Com essa migração, os objetos passam a se relacionar de forma direta podendo gerar consultas de informações cruzadas de quais negociação estão fechadas com quais empresas, desta forma, a visualização de detalhes de uma empresa, não se resumia em apenas exibir seu nome, mas sim em listar todas as suas negociações relacionadas.

Com a inclusão da `gem` **Devise**, o modelo `User` também foi criado, possuindo as configurações padrões definidas pela biblioteca. O modelo `User` foi alterado incluindo-se um campo de referência para o id de um representante. Desta forma, era possível associar um usuário do sistema à um representante de forma dinâmica. Este recurso tem como objetivo filtrar as principais telas do sistema através das cidades atendidas pelo representante, ou filtrando pelo próprio representante, implicando assim na estrutura de `Vendor`, que passaria a incluir um novo relacionamento `has_and_belongs_to_many` com cidades, e cidades com representantes, de modo que o diagrama de classes após as atualização poderiam ser representado pela Figura 17.

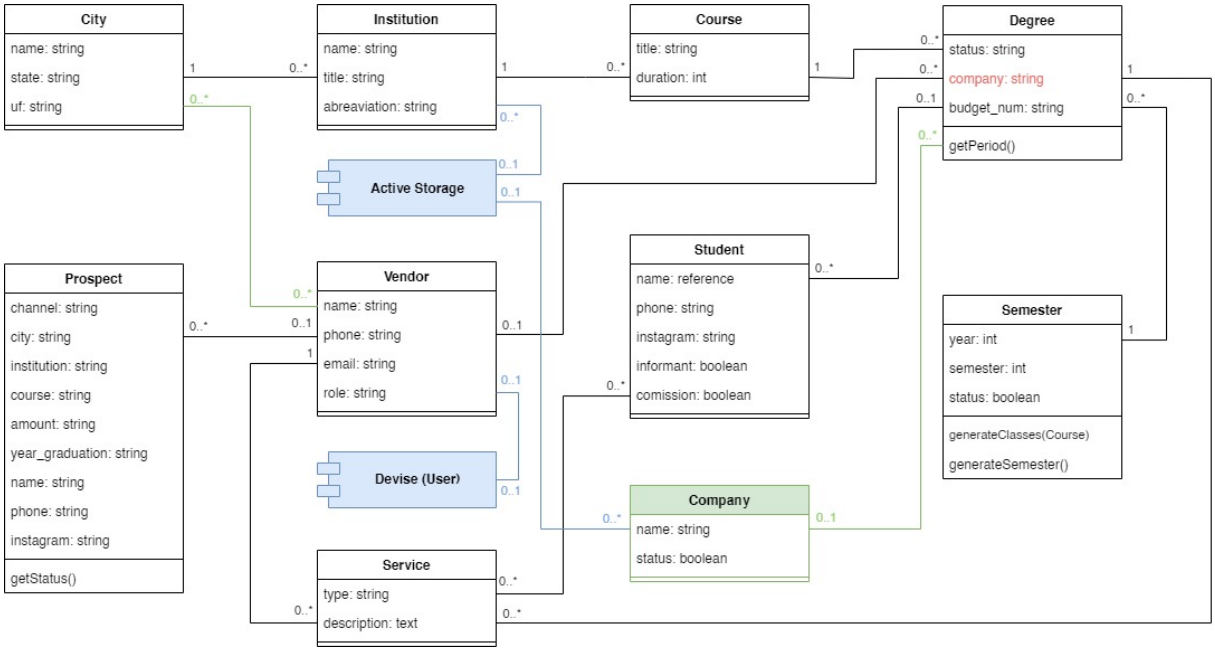

Figura 17 – Diagrama de Classes Atualizado
Fonte: Autoria Própria

Se tratando das alterações no banco de dados, também foi necessário realizar a configuração adequada aos parâmetros permitidos nos modelos já existentes uma vez que estes sofreram alterações. Isto também ocorreu com o modelo `User`, visto que o mesmo precisou ser vinculado a um modelo `Vendor`. Para fazer isto, foi necessário incluir o parâmetro `vendor_id` na função `configure_permitted_parameters` modificando as configurações do devise através do método `devise_parameter_sanitizer.permit(...)`. Além destas alterações feitas no *back-end* da aplicação, o ajuste das consultas SQLite3 para PostgreSQL nos demais controladores foi realizada, corrigindo o erro ocasionado por consultar dados em uma lista de `arrays` que continham algum valor em banco.

Outro ponto a ser considerado como uma alteração de projeto ainda no âmbito *back-end* é a melhoria de desempenho ocasionado pela obrigatoriedade de um valor inicial para listar os dados buscados. Esta mudança ocasionou a alteração do projeto através do relacionamento de `User` e `Vendor`, e como estas classes se comportam, trazendo de forma padrão nas páginas de listagem de dados as cidades que o represente atua, conciliando tanto o desempenho da aplicação quando na usabilidade do mesmo pelo usuário. Com isto, abriu-se uma nova possibilidade, a do representante vinculado ao usuário não possuir nenhuma cidade associada. Desta forma, a página de listagem de dados por mais que fosse renderizada sem erro algum, não apresentava nenhuma informação, ou seja, quando um usuário vinculado a um representante que não atendesse nenhuma cidade, nenhum dado era exibido, causando uma sensação de confusão ao usuário, por não saber se página estava carregando, se havia travado, ou o que exatamente havia acontecido visto que nenhuma mensagem e registro era retornado.

A partir disto, uma tela com sugestão de primeiros passos foi incluída, conforme representando na Figura 18, dando assim uma orientação inicial ao usuário, recomendo-lhe que associe um representante ao seu usuário para obter um melhor experiência ao usar o sistema. Seguindo o mesmo conceito, a tela listagem de Turmas, a qual exige uma instituição para exibir os dados, também passou a exibir uma instrução de orientação, além destas, mais algumas telas onde a situação se repetia de maneira semelhante também foram implementadas.

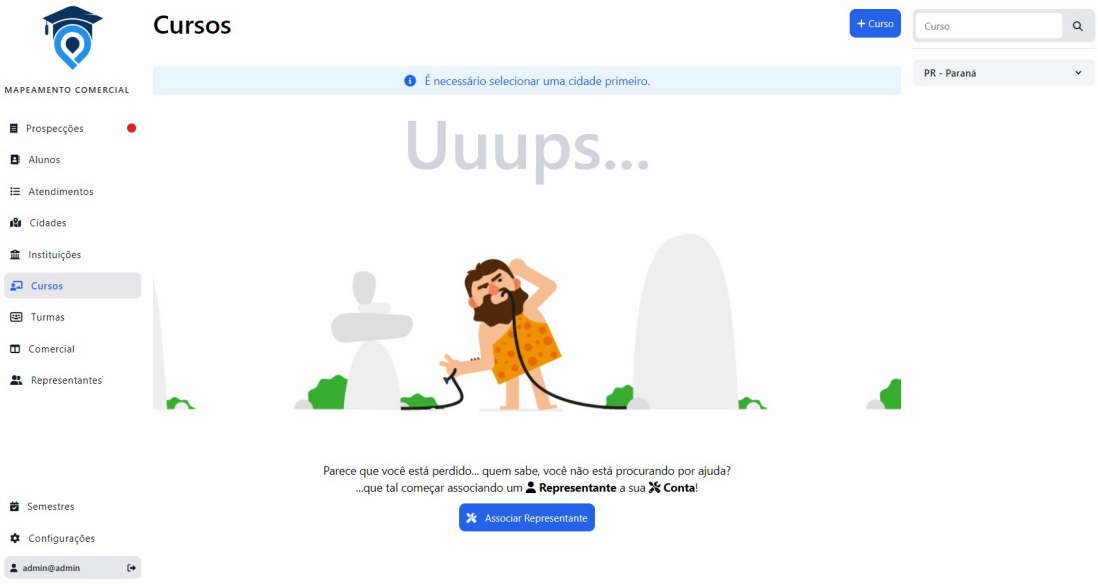

Figura 18 – Tela de Instruções - Associar um Representante
Fonte: Autoria Própria

Ainda neste quesito, outra alteração significativa que se fez presente durante o processo de refinamento foi da necessidade de possuir todos os registros na mesma página e a possibilidade de segmentar os dados através de paginação.

Fazendo uma analogia simples, se um mapa for dado a duas pessoas, cada uma utilizará pontos de referências distintos para se localizar por uma rota qualquer. Dito isto, a

forma com que cada usuário prefere ver as informações é relativa, tanto em critérios de filtros, como critérios de ordenação ou *layout* de visualização. É comum que um usuário considere melhor a tela de listagem de prospecções exibindo todas as prospecções, enquanto outro pode preferir considerar apenas as prospecções abertas. Contudo, um segundo critério precisa ser levado em conta, o desempenho da aplicação.

Considere que está empresa que irá utilizar o sistema possui mais de 10 mil prospecções abertas, e mais de 30 Mil prospecções encerradas. Filtrar todas essas informações assim como renderizar isto na tela tornaria a aplicação extremamente lenta e pouco usual. Por outro lado, ao exibir os detalhes de uma instituição, com mais de 40 cursos, é intencional que o usuário queira ver todos os cursos em uma página apenas. Ao fim de uma analise sobre este assunto, ficou estabelecido que as telas de listagem de prospecções e atendimentos, poderiam ser modificadas para que tivessem paginação, possibilitando assim manter o *layout* da aplicação sem rolagem vertical destas páginas. Quanto as páginas, Turmas, Cidades e Comercial, telas nas quais exibiam dados de acordo com as configurações do usuário, e que poderiam conter pouca ou muita informação, ficou-se estabelecido que não seria adequado separar tais dados por páginas, mas sim por critérios como semestres por exemplo.

Com isto, a gem **will_paginate** foi integrada ao projeto para facilitar na paginação dos dados um ajuste de *layout* foi feito, buscando manter a aplicação sem rolagem vertical, e para as telas não iriam receber paginação a visualização dos dados foi segmentada de acordo com os semestres. Um trecho de código CSS foi adicionado para ocular a barra de rolagem do navegador, permitindo que a rolagem do conteúdo seja renderizado apenas dentro do espaço estipulado, tornando a rolagem vertical possível, sem que a barra de rolagem seja exibida e sem quebrar os menus de navegação ou de filtros ao rolar a tela.

Buscando melhorar a aplicação afim de facilitar a interpretação à todos os usuários, era possível filtrar os dados do sistema da maneira como o usuário preferisse, uma vez que agora as informações eram ordenadas de forma lógica e coerente com os filtros determinados. Deste modo, é possível filtrar os emails enviados por um determinado representante que continha determinado termo, ou ver as turmas de uma determinada instituição separada por semestres e ordenadas por nome de curso, ou ainda, todas as negociações de um determinado representante com um determinando status e/ou fechadas com uma determinada empresa. Sendo que todas estas consultas traziam agora os resultados de forma paginada ou de forma única, conforme cada modelo determinava, sem prejudicar o desempenho da aplicação com grandes consultas.

Também trazendo uma melhor usabilidade dos recursos da `gem` **Flowbite**, a simplificação nas informações dos cards, abstraindo tais informações para `tooltips` interativos foi um grande diferencial na fase de refinamento da aplicação, pois permite que o usuário acesse a informação com todos os detalhes necessários sem ser redirecionado para uma página específica para consultar as informações, ou seja, tudo o que o usuário precisa esta pronto para ser acessado de maneira intuitiva na página de listagem e mantendo o *layout* limpo e organizado conforme mostra a Figura 19.

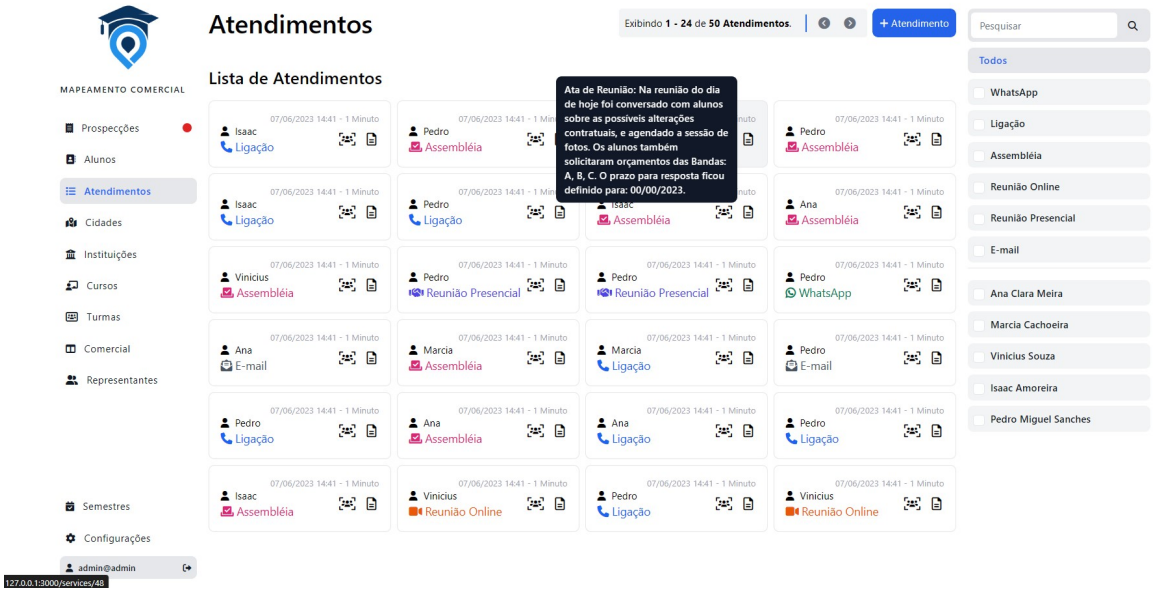

Figura 19 – Tooltips e Paginação
Fonte: Autoria Própria

Abordando um pouco sobre a facilidade em escrever código Ruby em Rails, existem dois métodos criados que merecem ser descritos neste documento, `generateSemester()` e `generateClasses(course)`.

É preciso voltarmos ao protótipo funcional baseado em planilhas para compreender o que estes métodos representam. Um comportamento esperado do sistema é que as turmas sejam geradas automaticamente pelo sistema, de forma que existem dois gatilhos que ativam este comportamento, o primeiro é a criação de um novo curso, onde é preciso abrir uma turma para cada semestre ativo, e o segundo é a criação de um novo semestre, onde é preciso abrir uma turma para cada curso registrado no sistema. O *framework* AppSheet que foi utilizada para o desenvolvimento do protótipo, apesar de oferecer uma série de automações, sejam elas acionadas por eventos, agendamentos ou até mesmo de forma intencional, como era o caso destas situações, não oferecia suporte nativo que permitisse realizar tal comportamento, o de se replicar de forma dinâmica ao longo dos registros da planilha, entretanto, era possível vincular arquivos do Google Apps Script (GAS) para executarem funções personalizadas. Para realização a criação de turmas para todo semestre aberto quando um curso fosse criado, foi necessário criar de um `script` que recebia a turma criada como parâmetro, e criava os demais dados na planilha de forma assíncrona. O mesmo acontecia para um Semestre, muitas vezes levando um tempo de resposta de quase 3 minutos para finalizar a execução do `script`, ou seja, caso mais cursos fossem registrados, corria risco de estourar o tempo limite de execução de 5 minutos e gerar inconsistência nos dados da aplicação a medida que a base de dados fosse crescendo.

O protótipo também possuía uma funcionalidade que foi desabilitada em sua versão final, a que era possível relacionar duas turmas, unificando-as uma única negociação. Considere o fato de que negociar todas as turmas de uma instituições pode implicar em uma unificação de eventos. Tal comportamento funcionava no protótipo e era realizada através do mesmo

`script` anexado a ferramenta. A complexidade desta função é absurda, pois implica em série de consequências. Considere por exemplo, três turmas vinculadas a mesma negociação, e uma destas desiste da negociação. Isto deve ser replicada as demais? O mesmo deve ocorrer para um atendimento? Caso uma turma saia da negociação, o que deve acontecer com seu histórico?

Diversas dúvidas como estas não era possível responder, pois cada caso era tratado individualmente, entretanto, era possível fazer com que os representantes fossem replicados nas negociações vinculadas. A função para adicionar representantes as turmas vinculadas foi simples de ser implementadas, entretanto, o oposto dessa função, isto é, quando um representante não atendente mais esta turma, foi bastante complexa. Ao interpretar os dados da planilha, o `script` não sabe exatamente 'quem' precisa sair 'daonde', sendo necessário analisar uma série de critérios para concluir sua lógica da forma esperada.

Feito em JavaScript, com 210 linhas para realizar tais tarefas, desde gerar as turmas corretamente, assim como replicar as informações dos tabuleiros vinculados, tal recurso foi desabilitado da versão *web* justamente por não possuir um comportamento claro de como a aplicação deve se comportar. Como alternativa a esta situação, o campo 'Orçamento Número' presente na Turma foi consolidado para realizar a associação das negociações, permitindo que qualquer empresa utilize seu próprio padrão de código ao nominar suas propostas. É valido destacar que este sistema não possuí módulo de propostas ou orçamentos, sendo vinculado com seus respectivos documentos apenas através deste campo informativo. Portanto, os comportamentos que foram implementados no `script` do protótipo funcional, são replicados de forma mais eficiente no *framework* Rails, de forma que a função que gerava as turmas ao ser criado um novo semestre ou um novo curso, foi escrita facilmente com 2 linhas de código dentro dos controladores de semestre e cursos, adicionando um laço de repetição dentro do método `create`.

Outro comportamento que é feito de forma automática pelo *framework* é o gerenciamento de alunos dentro de um atendimento, onde todo atendimento é obrigatoriamente vinculado a uma turma, entretanto, o atendimento pode ser associado a um ou mais integrantes da turma, de modo que replicar o comportamento em todas as 210 linhas escritas no protótipos foram abstraídas pelo próprio *framework* através do ActiveRecord. Concluir está série de alterações na aplicação foi uma tarefa essencial para melhorar a usabilidade do sistema.

5.3.1 Implementação de Tarefas Rake

Se tratando de alterações de projeto também é importante destacar o desenvolvimento de duas tarefas `rake`. `Rake` é o nome atribuído a um executor de tarefas, e uma tarefa pode ser descrita de várias formas como por exemplo fazendo um *backup* do banco de dados, executar uma série de testes com configurações distintas e até mesmo popular o banco de dados com informações em lote.

Para implementar de forma mais eficiente o ambiente de desenvolvimento, permitindo popular a aplicação de varias formas, devido as dependências entre suas classes, os arquivos

`lib/tasks/custom_seed.rake` e `lib/tasks/populate.rake` foram criados. Considere o fato inicial de que a maior parte das informações contidas no sistema vinham de um estudo previamente realizado, entretanto, algumas informações eram geradas de forma randômica como o caso das prospecções, atendimentos, alunos e representantes.

O conceito correto da função `seed`, que seria adicionar no sistema apenas dados essenciais para seu funcionamento, não estava sendo respeitado, pois gerava dados fictícios e supérfluos. Desta forma o código descrito em `populate.rake` separou a geração de dados fictícios da geração de dados considerados essenciais, deixando o projeto mais organizado. Este código basicamente implementa o comando `rails db:populate` de forma que quando executado pode se comportar de diversas maneiras mediante seu parâmetros informados.

Se executado apenas `rails db:populate`, por não estar informando nenhuma especificação, a tarefa irá excluir toda base de dados da aplicação e criar um registro de cada modelo do sistema, incluindo as turmas, que são geradas pelo sistema.

Contudo, é possível estabelecer até três parâmetros que tornam a tarefa muito mais dinâmica, sendo eles respectivamente `model_name`, `num_of_records` e `reset_records`, tornando possível realizar a chamada da tarefa de forma mais objetiva, como por exemplo, adicione 5 representantes sem excluir os já presentes no sistema, em seguida, adicione 15 prospecções excluindo todos as prospecções do sistema.

```
docker compose exec web rails db:populate[vendor,5]
docker compose exec web rails db:populate[prospect,15,true]
```

Também é possível chamar a tarefa de um modo diferente, de modo que implica na geração de `N` registros de todos os objetos do sistema.

```
docker compose exec web rails db:populate[all,5]
```

Para esta tarefa funcionar corretamente, diversos arquivos foram criados no diretório `lib/populators` de modo que um arquivo denominado `base_populator.rb` é utilizado como pai das demais classes filhas, que utilizam da herança dos métodos para utilizar e implementar comportamentos próprios referente a seus respectivos atributos. A partir destes arquivos específicos, é possível estabelecer as dependência entre classes, retornando uma mensagem do que precisa ser cadastrado previamente caso o usuário esteja tentando realizar uma ação indevida, como por exemplo, tentar adicionar uma instituição sem que exista ao menos uma cidade. Além disto, foi desenvolvido uma função de *log* mais detalhada, que exibe no terminal os registros que foram criados com detalhes dos seus objetos relacionados.

Portanto, o comando `rails db:seed` se encarrega apenas de popular o usuário de desenvolvimento do sistema (administrador), enquanto `rails db:populate` cuida de popular a aplicação com dados fictícios, incluindo imagens, para fins de testes da aplicação. Contudo, parte

dos dados como citado anteriormente, faziam parte de um estudo prévio e estavam preparados para o sistema, sendo considerados a base inicial de uma empresa qualquer de formaturas. Desta forma, foi possível fazer a importação de informações customizadas, consideradas essências para tal base, mas de forma mais controlado, através das instruções contidas no arquivo `lib/tasks/custom_seed.rake`.

Este `script` por sua vez permite executar o comando `rails db:seed:city` especificando um lista de cidade personalizadas, buscando o arquivo `city.rb` no diretório `db/seed/`. O mesmo se replica para as demais classes do sistema, sendo utilizadas principalmente para criação de instituições e cursos, pois já possuem as informações conferidas e as logos das instituições configuradas corretamente. É importante destacar que estes `custom_seeds` podem facilmente ser adaptados com informações de outras empresas, vindas por exemplo de um banco de dados qualquer ou até mesmo de uma planilha, facilitando assim a importação e configuração prévia da plataforma para implantação do sistema.

Ao longo deste percurso de desenvolvimento, a aplicação também foi sendo refinada aos poucos, editando detalhes simples como `badges`, legendas, tamanho dos cards e espaçamentos, além de cuidados com o *front-end* de forma geral.

5.3.2 Expandindo o Projeto com Docker

O Docker é uma solução para desenvolvedores, que possibilita configurar diversas aplicações com variáveis de ambiente diferentes com pacotes, bibliotecas e dependências distintas, pois realiza o gerenciamento de ambientes virtuais através de contêineres. Desta forma, é possível que com o projeto configurado para Docker, o mesmo seja instanciado em qualquer sistema operacional, compatível com Docker, de forma fácil e rápida.

Além de integrar este recurso ao ambiente de desenvolvimento local outra grande vantagem incluir sua aplicação em um contêiner Docker é facilitar o compartilhamento do seu projeto de forma que qualquer usuário consiga acessar seu sistema com suas próprias variáveis de ambiente. Baseado nas suas próprias configurações, os desenvolvedores podem contribuir com a manutenção e desenvolvimento da aplicação sem precisar fazer alterações em seus ambientes de desenvolvimento, entretanto, podem customizar suas credenciais de acesso aos serviços integrados da mesma.

Docker atualmente é essencial para grandes projetos ou projetos compartilhados com um time de desenvolvimento. Portanto, tornar esta aplicação compatível com o Docker foi uma grande alterações do projeto. Vejamos abaixo quais foram os passos realizados.

No `Dockerfile`, arquivo onde são descritos as etapas necessárias para configurar o ambiente que sera virtualizado e suas configurações. Durante o processo de criação da imagem, o Docker lê este arquivo e realiza a configuração do sistema operacional, pacotes, dependências e demais configurações pertinentes ao servidor onde a aplicação será instanciada.

As diretivas `ENV`, `RUN`, `FROM`, `ARG`, `WORKDIR` entre outras, são instruções que definem a imagem base a ser usada como ponto de partida para instalar as demais dependências

necessárias. Ao ler tais instruções, copiam arquivos para o contêiner criado e configuram o ambiente. Estas instruções portanto definem o comando a ser executado quando o contêiner é iniciado, garantindo que a configuração sempre seja a mesma. Conforme descrito no arquivo DockerFile, a imagem base a utilizada é Ruby 3.0.0, e alguns passos de sua configuração incluem instalar alguns pacotes como `curl` e `git`, a instalação do NodeJS, a instalação das dependências por Yarn e por fim realiza as definições das variáveis de ambientes do projeto para o contêiner Docker. A partir disto, o Docker realiza as configurações do usuário, incluindo-o no grupo de administradores, permitindo que o mesmo execute comando como administrador sem a necessidade de uma senha. Por fim, instala o `Bundler` e a versão de Rails utilizada no projeto.

O DockerFile apesar de ser de fácil compreensão possui uma complexidade de implementação alta, sendo escrito originalmente pelo orientador desta monografia, e adaptado as necessidades desta aplicação. Já os recursos de Docker Compose se baseia em um arquivo denominado `Docker-compose.yml`, que também deve ficar localizado no *root* do projeto. Seu objetivo é criar, configurar e executar todos os contêineres necessários para executar o aplicativo Docker definido no arquivo, permitindo que você configure o servidor do aplicativo com um único comando. O arquivo `Docker-compose.yml` possui sua estrutura semelhante ao `DockerFile`, também sendo um arquivo de texto. Para esta aplicação, o ambiente de desenvolvimento foi composto por quatro contêineres: `web`, `db`, `tests` e `selenium`, cada um recebendo seus respectivos atributos.

Além destes dois arquivos que serão utilizados pelo Docker, um terceiro arquivo precisar ser modificado, desta vez, para funcionar adequadamente com o PostgreSQL. No arquivo `config/database.yml` é necessário adicionar os parâmetros `host`, `username` e `password`, para que quando interpretados pelo Docker, apresentem a configuração válida para acessar o banco de dados agora instanciando em um ambiente virtualizado. Desta forma, o arquivo denominado `config/application.yml`, define as variáveis de ambiente de forma explicita, as quais são utilizadas em `database.yml`. Vale destacar que as informações de acesso são as padrões fornecidas pelo Postgres.

Para adicionar estas variáveis ao ambiente da aplicação, a linha abaixo foi incluída em `config/application.rb`.

```
ENV.update YAML.load_file('config/application.yml')[Rails.env] rescue {}
```

Desta forma, as informações contidas em `application.yml` são carregadas para ao ambiente Rails, e podem ser utilizadas de forma compartilhada, sem a necessidade de realizar a configuração das credenciais em cada atualização. Para facilitar o processo de instalação do projeto, uma cópia de `application.yml` foi feita para ser utilizada como exemplo, desta forma, a configuração padrão do projeto poderá ser utilizada sem problemas, ou poderá ser personalizada de acordo com o necessidade. Novas instruções em `.gitignore` para ignorar o

arquivo que não é exemplo, por conter as credenciais de forma explicita, ainda que sejam as padrões do banco de dados.

Para facilitar o uso do Docker através do terminal, é possível configurar um `ALIAS`, isto é, um apelido, ou facilitador, tornando possível atribuir uma `string` à uma variável. Isto é feito no `~/.bashrc` em WSL2, através da instrução `alias dc = "docker compose"`, deste modo, basta digitar `dc` para o comando ser interpretado como `docker compose` reduzindo assim a sintaxe ao trabalhar com Docker. Também é possível criar novas funções no arquivo de *bash* que aceitem parâmetros mais complexos ficando a critério próprio desenvolvedor assim faze-lo se desejar.

O comando `build .` compila a imagem a partir das especificações da `DockerFile`, assim como o comando `up` utiliza os valores contidos em `database.yml` para realização ações dos contêineres associados ao banco de dados. Executando o comando `dc build .` é possível acompanhar a montagem do contêiner da aplicação. Logo após isto, o comando `dc run \textit{web} bundle install` instala todas as dependências da aplicação no contêiner `web`, tornando a aplicação acessível através da instalação de seus recursos e `gems`. Para melhorar a utilização do Docker é recomendado adicionar ao comando a ser executado o atributo `--rm` que força o Docker a remover o contêiner após ser utilizado, evitando lixo de memória.

Para finalizar a configuração da aplicação no ambiente Docker, é necessário realizar a configuração do banco de dados e das tabelas da aplicação, sendo necessário executar os comandos `create`, `migrate` e `seed`. Portanto, para configurar o banco de dados, os comandos `dc run --rm web rails db:create`, `dc run --rm web rails db:migrate` e `dc run --rm web rails db:seed` precisaram ser executados. Com isto feito, bastou solicitar ao Docker que inicialize o contêiner `web` da aplicação através do comando `dc up web`. Este comando deverá inicializar o contêiner `web` e `db`, que são consequentemente a aplicação e o banco de dados, de forma que o contêiner `web` depende diretamente do contêiner `db` para funcionar corretamente.

Feito isto, a aplicação esta pronta para uso e compartilhamento através do ambiente Docker, não sendo mais necessário configurar manualmente todo o ambiente de desenvolvimento. É valido citar que o mesmo pode ser feito para o ambiente de produção, agilizando desta forma o processo de configuração do servidor VPS através do uso do Docker em produção.

5.4 Testes e Validações

Conforme previsto pelo cronograma estabelecido, a etapa de testes e validações acabou sendo a última a ser implementada. Esta etapa pode ser considerada a mais relevante do projeto pois agrega de forma prática uma série de novas abordagens e enriquece a percepção sobre a organização do *framework*.

As validações garantem a segurança e o comportamento esperado da aplicação, enquanto os testes são capazes de simular a maioria, se não todos, os cenários possíveis para a

aplicação de forma rápida e eficiente, tornando os testes manuais algo relativamente antigo e defasado.

5.4.1 Ambiente de Testes

O *framework* Ruby on Rails trabalha com Minitest e traz em seu modelo nativo de testes a pasta `fixtures`. Os `fixtures` nada mais são do que especificações de modelos pré-definidos para serem utilizados nos testes, contudo, há uma forma mais interessante de se utilizar modelos gerados para testes. É possível utilizar a `gem` **FactoryBot** para criar modelos de objetos para testes, e além disto, a `gem` é compatível com a *gem* **Faker**, utilizada para gerar dados aleatórios, tornando a criação de informações ainda mais dinâmica. Isto possibilita que dados mais realista sejam testados na sua aplicação, e permite simular até mesmo o envio de imagens ao sistema.

Neste projeto, os `factories` que possuem imagem, como `Companies` e `Institutions` utilizam a `gem` **open-uri**, a qual permite salvar uma imagem temporariamente através de uma URL específica, realizando a criação de modelos de forma completa e com imagens anexadas.

Como será utilizado os recursos do FactoryBot, a pasta `fixtures` pode ser removida da aplicação e substituída pela pasta `factories` e seus respectivos arquivos. Ainda com esta mudança, mais algumas `gems` foram incluídas na aplicação para que os testes pudessem ser realizados de uma maneiras mais eficiente. São elas a `gem` **Shoulda Context** que permite organizar os testes em contexto separados, dando um melhor entendimento do código que esta sendo testado, facilitando assim a configuração e interpretação do teste, e a `gem` **Shoulda Matchers** que permite realizar validações e afirmações com uma sintaxe reduzida e intuitiva. Além destas `gems`, as configurações necessárias para fazer a `gem` **Capybara** encontrar a aplicação no ambiente Docker também foi realizada. Todas as configurações por sua vez foram realizadas seguindo a documentação sugerida pelos desenvolvedores das bibliotecas.

Nesta etapa foi possível classificar os testes em dois aspectos, os testes de sistemas que se referem as `views` da aplicação e os testes de unidade que se referem testes de modelos, métodos e controladores da mesma. Assim como na etapa de desenvolvimento do projeto onde foi-se refinando a aplicação das telas menos complexas para as mais complexas, os testes também seguiriam a mesma lógica de implementação. Antes de iniciar de fato as validações e testes, se fez necessário compreender também o que cada testes deveria de fato testar, de forma a prevenir um erro comum de vários programadores no cenário dos testes, o de escrever testes que não testam nada, ou não testam o que deveriam testar.

Uma pequena mudança que foi realizada para se objetar uma melhor compreensão dos testes foi a estruturação da pasta `tests/system`, sendo organizada com uma pasta para cada `controller` e um arquivo para cada ação do controlador, tornando a reprodução de testes mais semelhante a estrutura do projeto e com arquivos de testes mais objetivos.

Antes de abordar os resultados desta etapa é interessante destacar que assim como o *framework* era algo abstrato mediante meus conhecimentos, o processo de testes também era e

não representava com clareza a sua importância em uma aplicação. Um dos grandes debates no mundo de programação, o de que sistemas não precisam ser testados, vem sendo desconstruindo mediante a compreensão do assunto. Inicialmente, questionamentos como: por que testar se o que eu programei para estar lá, vai estar lá? ou por que testar se o botão de ir para tal link, vai para lá mesmo? eram dúvidas comuns, contudo, ao longo do aprendizado sobre os testes, foi possível além de encontrar várias falhas de código, perceber a importância de realmente garantir que a informação vai estar lá, em todos os cenários possíveis. Vejamos a seguir de forma mais detalhada como e etapa de testes e validações influenciaram no comportamento da aplicação.

5.4.2 Teste de Unidade

Para começarmos a nos aprofundar no assunto, considere por exemplo a simples validação de um campo do tipo `int`, ao receber um dado do tipo `string`, certamente resultará em algum erro. Desta forma, uma simples validação de tipo de dados garante que o sistema esteja se comportando corretamente caso receba uma `string`, ou qualquer outro tipo de dados, ficando a critério do desenvolvedor do sistema, implementar a melhor forma de como tratar esta informação, seja ela qual for.

Tal comportamento abre o debate sobre o tempo correto em se escrever os testes da aplicação, sendo antes da implementação, ou depois da implementação do código. Um assunto muito interessante a ser explorado. Entretanto, neste caso, não havia outra alternativa se não posterior a implementação. Independente desta etapa ser realizada antes ou depois do desenvolvimento do projeto, é importante que ela seja desenvolvida, pois acaba sendo uma dupla conferência do que foi ou será programado.

Vejamos portanto os primeiros testes de modelos de cidade, um modelo que não possui dependências externas ou requisitos para ser criado, mas possuem uma particularidade um pouco diferente da esperada. Em seus atributos uma cidade pode possuir `name`, `state` e `uf`, entretanto, no formulário de cadastro de novas cidades não faria sentido ser necessário informar o campo estado e uf duas vezes, ou até mesmo, mediante existir ambos os campos, permitir que o usuário informe valores distintos fora da convenção geográfica. Desta forma, o formulário faz uma manobra, passando o nome do estado junto com o uf do mesmo e realiza o tratamento deste parâmetro em seu controlador antes de proceder com a ação devida. Portanto o formulário apresenta apena dois campos, estado e cidade, sendo que o modelo possui três atributos. Os testes deste modelo por sua vez estabelecem as possíveis associações de cidade com outras classes, a presença dos atributos presente no formulário, neste caso, somente para `name` e `uf`, assim como o tamanho do texto informado e se a cidade é única, evitando duplicar registros do banco de dados.

Em seguida, dois testes descrevem os contexto, o que deveria funcionar e o que não deveria funcionar. Desta forma, ao executar os testes, o *framework* busca realizar suas asserções, e espera validar que o modelo possua suas especificações conforme foram esta-

belecidas no teste. Para que o teste funcione como o esperado, o próximo passo foi incluir no modelo de cidades as validações conforme elencadas pelo teste, mas utilizando a sintaxe correta para que o teste seja validado. Após incluir as devidas validações no arquivo na classe de cidades, foi possível levantar o servidor de testes pelo Docker e executando o comando:

```
docker compose run tests rails test test/models/cities_test.rb
```

Este comando, executas os testes e verifica se tudo estava funcionando conforme esperado. Note que até o presente momento, apenas três contêineres foram utilizados, sendo eles: `web`, `db`, e `tests`. O último, `selenium`, será utilizado nos testes de sistema.

Com o teste de modelo devidamente concluído, a próxima etapa foi realizar os testes de controladores, onde estes possuem apenas o objetivo de garantir o tipo de resposta obtida ao acessar as rotas solicitadas. Este teste inicialmente se mostrou mais interessante do que o anterior, visto que a camada de segurança adicionada pelo Devise, redirecionou o teste ao tentar acessar a rota solicitada, de modo que o comportamento da aplicação era de fato o esperado, e o teste por sua vez havia pulado a etapa de fazer login no sistema para conseguir visualizar as informações.

É importante levar em consideração a ação em questão que esta sendo testada, de forma que no `cities_controller_test.rb` não deve testar especificamente se o usuário em questão possui um conta válida para acessar o sistema, mas sim garantir que ao acessar a rota desejada, o conteúdo desejado seja entregue, desde que o usuário possa realizar tal ação, caso contrário, o teste deve validar se o usuário foi redirecionado corretamente para página esperada. Desta forma, foi estabelecido os contextos autenticados e não autenticados para maioria dos controladores, com exceção das prospecções, a qual permite acessar a tela de nova prospecção e efetuar um registro diretamente no banco de dados sem estar logado no sistema, possibilitando que este formulário seja facilmente compartilhado e arrecade assim novas prospecções e futuros clientes de forma integrada ao sistema.

Apenas com o teste de modelos e controladores de apenas um objeto do sistema, é possível garantir que em todos os cenários possíveis pertinentes ao que esta sendo testado, a aplicação irá se comportar conforme planejado, seja redirecionando o usuário para uma tela, exibindo notificações e mensagens de erros, assim como garantindo assertividade aos links acessados quando esperado, e encaminhado o usuário à telas de instruções em caso de falhas quando a aplicação se comportar fora do esperado, garantindo assim a integridade da mesma, sem quebrar a experiência do usuário.

Nas etapas seguintes, os mesmos testes foram replicados para os demais modelos e controladores, sendo cada um implementado mediante seus comportamento esperado. Esta etapa revelaram durante a execução dos testes uma série de inconsistências nos relacionamentos existentes, sendo alguns sem correspondentes diretos, isto é, possuíam `has_one`, mas o relacionamento alvo não possuía a diretiva `belong_to`, e em outros casos, as referências

estavam escritas de forma não pluralizada, evidenciando assim a importância real dos testes de sistema, pois ainda que este projeto seja para fins educacionais, tal erro poderia implicar em grandes problemas em um cenário real, reforçando a boa pratica de realizar testes antes de implantar a aplicação.

Como parte da metodologia, foi realizado primeiro o desenvolvimento de todos os testes referente aos modelos da aplicação, seguidos de todos os testes de seus respectivos controladores e ações, onde ao fim da série de testes foi possível constatar 199 execuções que totalizam 289 asserções até o presente momento.

5.4.3 Testes de Sistema

Com todos os modelos do sistema validados e seus respectivos controladores também, a parte que demandou mais atenção foi a parte de testes de sistema, os quais buscam garantir se as informações solicitadas estão sendo renderizada corretamente ao usuário. Este teste por sua vez pode ser considerado um dos mais demorados, pois além do sistema apresentar várias telas, existem um infinidade de situações que podem ocorrer, resultando em ações realizadas ou ações não realizadas, como por exemplo tentar salvar um registro sem suas informações obrigatórias ou tentar fazer uma atualização com um nome que já exista na base de dados, ocasionando assim em ambos os casos uma mensagem de erro, ou simplesmente concluindo um cadastro, gerando assim a mensagem de um objeto cadastrado com sucesso.

Os testes de sistema neste caso podem ser específicos e testar em determinado cenários se a mensagem desejada é exibida corretamente, ou conforme elaborado nesta aplicação, podem ser mais genéricos, testando apenas se um elemento que representa a caixa de dialogo de erros esta presente e sendo renderizada, não necessariamente a sua mensagem. Vale destacar que como uma boa prática, quanto mais assertivos os testes forem, a previsibilidade do sistema possuir erros se torna menor, ficando a cargo do time de desenvolvimento considerar o teste desenvolvido como suficiente ou não para tal parte do sistema. Nesta etapa de implementação de testes de sistema, é possível afirmar que houveram várias mudanças nos arquivos do projeto, além de uma pequena reestruturação nos diretórios contidos em `tests` para que os arquivos ficassem mais organizados. É importante levar em conta que assim como os testes de controladores foram separados em pastas e segmentado em arquivos correspondentes as ações contidas no mesmos, os testes de sistema também seguiram a mesma estrutura.

Um ponto muito relevante a ser considerado neste tópico, é de que o sistema foi desenvolvido contendo um banco de dados populado com registros fictícios, facilitando assim a implementação do *layout* em seu aspecto final, contudo, os testes de sistema trazem a tona uma parte não muito explorada durante a codificação, a dos primeiros passo de um novo usuário do sistema com um base limpa. Como consequência disto, em seu estado atual, com uma base de dados limpa, a maioria das telas não renderizaram informação alguma, isto é, nem mensagem de erros, nem informações, e outras, as quais dependiam de fatores cruzados, como por exemplo, um menu de filtro de cidades, em um cenário onde não existem cidades

cadastradas, implicava na quebra da aplicação, ou na má renderização dos dados, gerando confusão ao usuário.

Mediante a evolução dos testes de sistema, a refatoração do código tanto na `view` quanto no `controller` foi essencial para otimização do sistema e para que os testes de sistema fossem concluídos com sucesso, uma vez que os testes abordavam a aplicação tanto sem dados, quanto com informações em sua base dados.

É possível elucidar a perspectiva de alguns testes de forma prática considerando os cenários em seus respectivos contextos. Considere portanto o contexto da ação `index` de `cities_controller` é possível descrever os cenários onde não há cidades cadastradas, onde há pelo menos uma cidade cadastrada e onde há pelo menos uma cidade cadastrada que possua ao menos uma instituição cadastrada. Observe que tais situações ocorrem de forma sequencial no que diz respeito a usabilidade do sistema, quase que tornando o primeiro uso da aplicação um passo a passo ao usuário. É importante destacar que não adianta possuir um sistema que seja capaz de fazer tudo, se os usuários não souberem por onde começar, ou melhor, como alimentar o sistema da forma correta. A partir desta premissa, diversas telas parciais de instruções foram criadas no diretório `layouts/shared` e por sua vez, a refatoração do código no arquivo `index.html.rb` para renderizar a mensagem de instrução correta conforme a devida situação.

A medida que esta mesma perspectiva de testes era aplicada as demais visualização do sistema, os diversos possíveis cenários se tornam mais complexos e exigem uma série de características para serem realizados, coisa que sem dúvida, fazê-los de forma manual seriam muitíssimos custosos.

Outro caso que é valido de destacar devido a sua complexidade, é o que diz respeito aos cenários que dependem de cidades especificamente. Vejamos a situação onde um novo usuário do sistema, que ainda não possui vínculo com nenhum representante, ao acessar uma tela que precise de ao menos uma cidade para funcionar corretamente. Inicialmente, o sistema informará o primeiro cenário, o de que não há cidades cadastradas. Logo após cadastrar uma cidade e retornar a tela, o usuário verá a tela de que é necessário filtrar a cidade manualmente, ou conforme o segundo cenários demonstra, podendo atribuir um representante a sua conta para fazer isso de forma automática.

Uma dúvida pode ter surgido neste momento, o que exatamente um representante tem a ver com as cidades? E isto deve-se ao fato de um representante poder atender várias cidades, o que nos leva ao terceiro cenário, o qual considera que usuário tenha em sua base de dados, ao menos uma cidade cadastrada e também possua um representante vinculado a sua conta, contudo, neste caso, este representante não atendente nenhuma cidade. Observe que neste momento, pode-se obter duas abstrações, uma de que o usuário é um administrador e não atende nenhuma cidade, de forma que nenhuma mensagem de erros ou alerta deveria ser renderizada, ou a de que o usuário esqueceu de associar as cidades atendidas no representante.

Um último cenário é possível ser descrito, a de se existir um usuário que possua vínculo

a um representante que atenda uma ou mais cidades, onde deste modo, o sistema atende as solicitações esperadas e mostra os dados filtrados de forma mais rápida e eficaz ao usuário, sem a necessidade, mas ainda com a possibilidade, do usuário filtrar os registros manualmente. De forma resumida, apenas neste exemplo, existem quatro possíveis comportamentos esperado para aplicação, sendo cada um referente ao contexto a seguir:

I Usuário, sem cidades, sem representante.
II Usuário, com cidades, sem representante.
III Usuário, com cidades, com representante sem cidade atendidas.
IV Usuário, com cidades, com representante com cidades Atendidas.

Tais contextos se estendem ao longo dos demais testes dos sistema e seus respectivos comportamentos de forma que os cenários podem depender de muitas informações para poder renderizar com sucesso apenas uma mensagem. Reproduzir todos estes cenários com Rails pode ser tido como algo relativamente simples, mas um tarefa relativa extensa mediante o tamanho e complexidade da aplicação.

5.4.4 Alterações do Projeto após Etapa de Testes e Validações

Vejamos neste tópico algumas alterações do projeto em decorrência dos testes gerados. Estas características podem ser descritas como forma de enriquecimento educacional para este projeto, buscando abordar de forma mais detalhada o comportamento do sistema em algumas situações específicas.

Durante a etapa de testes na parte de prospecções, onde é esperado que seja possível um usuário sem autenticação realizar o cadastro de um novo registro, foi possível constatar que no cenário onde o usuário estava autenticado, ao criar uma nova prospecção o sistema o redirecionava para a mesma com a a mensagem de sucesso, porém, o usuário sem autenticação não possuía acesso a esta tela e era redirecionado para área de *login*. Contudo, a mensagem referente ao envio do formulário, seja ela sucesso ou falha, não era renderizada corretamente. Desta forma, a solução para este comportamento foram duas alterações, adicionar a `tag` de renderização de notícias e alertas nas visualizações do Devise, e alterar o `prospects_controller` para que redirecionar o usuário mediante sua autenticação. Durante esta correção, também foi possível identificar que alguns *links* do sistema possuíam atributos duplicados, como em botões com classes extensas, nos quais o atributo `target` estava duplicado, uma antes da definição do atributo de classe, e outra após, bastando apenas apagar o código duplicado.

Já na etapa de testagem de modelos, foi constatados que alguns relacionamentos não estavam estabelecidos da forma correta, conforme citado anteriormente, alguns por falta de pluralização, outros por falta de referência explicita ao modelo em questão, contudo, o fato de reescrever os relacionamentos nos testes, gerou uma segunda revisão sobre a estrutura relacional da aplicação. Objetos relacionados que ainda não foram utilizados no sistema de forma

direta, como por exemplo o relacionamento de `Vendors` com `Services`, uma vez que nenhum momento da aplicação era solicitado a instruções `vendor.services` ou `service.vendors`, acabaram passando por revisões, de forma que a configuração do aplicação estava parcialmente errada, contudo, sem gerar erros, já que não era utilizada em lugar algum. Ainda durante a etapa de revisão de relacionamentos alguns modelos tiveram seus relacionamentos alterados, excluindo assim a obrigatoriedade de se pertencer a um objeto, ou possibilitando o registro de várias instâncias do mesmo, como por exemplo as cidades atendidas por um representante.

Os testes além de abordar uma perspectiva diferente da visão de desenvolvedor, buscando se assemelhar mais a um perspectiva do usuário, possibilita que o novas camadas de abstrações sejam implementadas. Vejamos de forma mais detalhada o cenário a seguir.

Os testes voltados a parte de *back-end* da aplicação normalmente acabam sendo mais simples e tendem a não passar mensagem de erros como no ambiente de desenvolvimento ao usuário do sistema como cliente, tão como não ocasionar a quebra da aplicação. Em outras palavras, o usuário cliente, não espera ver uma mensagem de erro da mesma forma que o desenvolvedor, sendo necessário informar de uma maneira amigável ao usuário o motivo da sua requisição ter falhado, contudo, em alguns casos, é possível que o usuário force algum tipo de erro, como é o caso do teste de turmas. Ao acessar esta tela, independente de possuir um representante vinculado ou não, é necessário selecionar uma instituição no menu de filtros lateral. Ao fazer esta seleção, dois parâmetros são enviados, `institution_id` e `city_id` sendo respectivamente utilizado para filtrar a instituição desejada e exibir as negociações/turmas da mesma, e para expandir automaticamente a cidade selecionada no menu lateral, mantendo o menu de filtros expandido na cidade que fora selecionada. Ambas as informações ao chegar no controlador são processadas em seguida retornadas para a visualização de forma que, mediante o nível de segurança esperado da aplicação, pode-se considerar aceitável que caso o usuário informe uma `city_id` inexistente, implicando desta forma, que o menu lateral de filtros fique comprimido, e não quebre a experiência do usuário. Contudo, pode ser que exista esta exigência de segurança, fazendo com que caso o usuário quebra a busca, forçando algum parâmetro indesejado ou inexistente, a aplicação realmente precise interromper a ação, como neste exemplo, pode ser comparada a alteração do parâmetro `instituion_id`, uma vez que se os dados a serem renderizados esperam de fato uma encontrar uma instituição, quando o usuário força a busca por um número que não exista na base de dados, a aplicação deve agir, tratando este comportamento da maneira mais adequada. Neste caso, a solução foi retornando o usuário a tela turmas, com um alerta de de que algo deu errado durante a consulta de dados e solicita ao usuário que verifique o endereço digitado ou tente novamente, em ambos os casos, sem quebrar a aplicação. É valido lembrar que esta abordagem não necessariamente é considerada correta, uma vez que não existem regras que estabeleçam diretrizes ideais, apenas conceitos sobre o comportamento esperado da aplicação, e que diferente de seu protótipo, pode ser alterada a qualquer momento, conforme assim for necessário.

Outra abordagem importante que os testes nos trazem diz respeito sobre o o contexto

do menu comercial e sua barra de filtros, onde considerando o caso de não havia representante associado a conta de usuário, é necessário realizar uma seleção inicial para se buscar informações, entretanto, após o resultado da consulta ser exibido, se o usuário apenas remover a seleção do filtro, o sistema tentaria buscar por todas as negociações em sua base de dados.

Por mais intuitivo que este comportamento seja, isso pode implicar diretamente no desempenho da aplicação e dos servidores que há hospedam, desta forma, os testes permitem que caso o usuário force uma entrada vazia, o sistema o redirecione para a tela inicial o informando do que ocorreu, em vez de renderizar todos os arquivos sobrecarregando o servidor. Note que é importante que neste momento, caso o usuário realmente queira ver todas as informações, coisa que não é frequentemente usual, o usuário poderá carregar os dados de forma parcial ao utilizar os filtros, tornando as buscas mas leves para servidor, e até mesmo renderizando dados mais rápidos com o HotWire.

Uma alternativa para replicar o comportamento esperado, sem precisar marcar opção por opção, seria adicionar uma opção de 'todos' para considerar todos os registros. Recurso que não esta presente atualmente no escopo do projeto, porém foi adicionado a relação de trabalhos futuros. Além destes erros mais complexos, alguns erros ortográficos também foram corrigidos, como por exemplo o menu de navegação que continha o texto 'comerial' e não 'comercial' como esperado. Por mais simples que pareça, sem os testes isto certamente passaria por muitas pessoas até ser notado, e pode sim ser considerado um erro grotesco no caso de um aplicativo em produção.

Outros erros que podem ser ocasionados de maneira forçada, é um situação reproduzida com as cidades, onde quando utilizado corretamente pela *interface* do sistema, não conduz o usuário ao erro, entretanto, se usada de maneira abstrata, isto é, sem a utilização da *interface* da aplicação renderizada por um navegador, pode-se ocasionar comportamentos inadequados ao sistema. Considere que um usuário qualquer tenha forçado o parâmetro da cidade de Guarapuava, estado de PR, para Guarapuava, estado de SC. Note, que ao utilizar a *interface* do sistema esta situação não acontece devido a implementação do Stimulus com o Turbo, contudo, se feita sem a *interface* recomendada, tal ação resultaria num menu lateral de filtros com registros duplicados, sendo um para Guarapuava-SC, e outro para Guarapuava-PR. É importante se compreender o cenário em que tal situação ocorre e como isto pode impactar a aplicação de tal modo que se necessário as devidas correções para manter a aplicação estável devem ser tomadas de forma breve. Neste caso, ainda que sabendo de tal comportamento, por não implicar diretamente no comportamento da aplicação como um todo, sendo considerado apenas um registro errado, e que poderia ser alterado futuramente, é esperado que o teste em questão não resulte em erro, permitindo que tal comportamento seja aceito, e inclusive, que ambas as cidades sejam renderizadas no menu de filtros, funcionando corretamente, garantindo assim que o teste esperado realmente seja concluído com sucesso, e que a aplicação não quebre.

Note que o intuito deste teste por sua vez não é validar se a cidade de Guarapuava realmente pertence ao estado do Paraná, mas sim se o comportamento da aplicação corresponde

ao esperado, registrando os dados no banco de dados da aplicação, e exibindo-os conforme o esperado, mesmo que sejam dados incorretos.

Outra funcionalidade interessante que os testes permitiram implementar no sistema foi a de perceber a ausência de alguns registros e sugerir ao usuário um botão para adicionar um novo registro do modelo em questão. Desta forma, os testes tem como objetivo garantir que tais botão existam e sejam recomendados apenas quando realmente for adequado ao usuário, e que por sua vez o link contido nele o direcione para o endereço correto. Neste contexto de redirecionamentos, os testes também buscam garantir que os botões de ações com parâmetros exclusivos, como por exemplo, o de cadastrar um novo aluno com os dados contidos em uma prospecção, ou o de cadastrar um novo atendimento para uma determinada turma, já trazendo os dados pré-selecionados, de forma que os testes neste caso, inspecionam os elementos HTML para validar se os parâmetros estão montados da forma esperada e guiam o usuário a ação esperada.

É valido destacar que a aplicação utiliza uma linguagem informal e faz uso de *gifs* de forma descontraída como forma de se aproximar do usuário, buscando conduzir o uso da aplicação em seus passos iniciais de forma mais visual, contudo, mediante a seriedade das informações e por se tratar de um sistema comercial, tal linguagem pode não ser adequada por muito tempo, onde, após o usuário realizar as primeiras configurações e já estar pronto para utilizar o sistema, tais recomendações não são mais exibidas, deixando assim a aplicação com um visual mais adequado para o ambiente de trabalho.

Outra percepção que a visão de teste nos traz, diz respeito a interface guiada ao usuário, onde através de recomendações, guiam o usuário a fazer fazer tal ação pela primeira vez, contudo, após a primeira vez feito, o botão deixa de ser exibido. Nesta situação em particular é possível afirmar que o teste age conforme o esperado, contudo, para um usuário novato, após executar a ação pela primeira vez acaba se familiarizando com o botão no local indicado, porém, em segundo momento, quando já existem registros cadastrados, a falta deste botão, por mais simples que pareça, pode dificultar o uso do sistema.

Um exemplo prático deste caso é o de tentar adicionar um novo curso. Sendo inicialmente a fazer isto a partir da tela de instituições, para adicionar um segundo novo curso, o usuário precisa preencher todo o formulário novamente, ao invés de apenas clicar em um botão que o levaria a mesma ação de forma mais rápida.

Durante os testes de sistema uma outra particularidade pode ser notada e alterada, a que a maioria das visualização possuía um trecho de código para renderizar as noticias vindas do controlador, contudo, após as noticias terem sido refatoradas para o `layouts/application.html.erb`, as notificações acabavam sendo renderizadas duas vez, como consequência disto, foi necessário remover os trechos duplicados para simplificar o código e não exibir as notificações de forma duplicada.

5.4.5 Estabelecendo Relações de Dependência

Os testes portanto contribuem muito para a manutenção futura da aplicação, garantindo que tudo esteja funcionando, ou que o que não esteja funcionando seja facilmente identificado. Ao longo da criação dos testes uma parte crítica foi identifica, a que diz respeito a exclusão de registros de forma destrutiva. O comportamento que o *framework* Rails emprega permite estabelecer a diretiva *dependent* nos modelos da aplicação, e utilizá-la para definir o que acontece com os registros que possuem dependência com o objeto a ser modificado ou excluído. Um exemplo simplificado e paralelo ao projeto atual, contido na documentação do *framework* nos mostra que é possível fazer com que um Autor, ao ser excluído, tenha todos os seus lixos excluídos juntos, conforme representado no trecho de código abaixo:

```
class Author < ApplicationRecord
  has_many :books, dependent: :destroy
end

class Book < ApplicationRecord
  belongs_to :author
end
```

Tal comportamento é muito debatido em vários cenários, levantando o questionamento sobre a real necessidade de se excluir uma informação e suas devidas implicações, tão como a apenas de se apenas inativar um registro, tornando-o oculto para a aplicação. Desta forma, o sistema de mapeamento comercial por possuir um forte relacionamento de dependências entre suas classes, não deve apresentar um comportamento destrutivo, mas sim, restritivo em relação a exclusão dos dados.

O sistema implementa em seus controladores a verificação dos relacionamentos entre as classes, e caso possuam dependências, impossibilitam a ação de exclusão solicitada indicando o motivo por sua vez, é possível garantir que não sejam removidos por exemplo uma instituição que possua cursos ou turmas abertas, da mesma forma que não é possível excluir um representante que possuí atendimentos realizados e/ou prospecções assumidas. Entretanto, caso o representante não possua vínculo algum, ou ainda, que o curso ainda não possua turmas, é possível realizar a exclusão destes registros, já que por não possuir dependências, não influenciariam nos demais registros do sistema. Todos estes cenários são testados e garantem o funcionamento da aplicação conforme esperado.

Contudo, abre-se espaço para uma alteração importante na estrutura do projeto e na forma de como as turmas se comportam no sistema. Considere o seguinte aspecto, as turmas não são geradas de forma manual, de modo que não é possível criar uma nova turma manualmente. Para se criar uma nova turma são necessários duas coisas, um curso e um semestre. Dependendo da ordem em que forem criados, o resultado poderá ser diferente, entretanto, ambos resultaram na geração de novas turmas. Desta forma, por não ser possível

criar uma turma de forma semelhante aos demais itens do sistema, o questionamento de ser possível excluir uma turma vem a tona.

Considere portanto que tal comportamento implica em um série de ações, desde como falhas em estatísticas, melhorias de desempenho dos representantes, exclusão de histórico de atendimentos e alunos e vários outras consciências. Portanto, não seria viável possibilitar a exclusão de registros que são gerados automaticamente e que possuem relacionamento com grande parte da aplicação, além do mais, não é nítido a real motivação de um usuário excluir uma turma, ao invés de registra-la com o `status`: 'não existe'.

Desta forma, vamos supor, que o usuário por algum motivo não queira ver aquela turma nas telas de negociações, a maneira encontrada para solucionar tal comportamento foi relativamente grande para algo trivial. Foi necessário criar uma nova migração adicionando uma coluna booliana nomeada como `visible` para `Degrees`, desta forma, o controlador também foi alterado para que quando recebesse uma requisição de exclusão, o comportamento seria apenas alterar o valor de `visible` para `false`. Assim, a turma não seria listada nas negociações, contudo, ainda seria possível acessar o registro através dos cursos, e ao acessar um curso inativo, seria possível torna-lo ativo novamente.

Com isto, foi possível estabelecer a funcionalidade de exclusão, agora sendo interpretada como 'esconder' as turmas do sistema, sem influenciar diretamente na sua estrutura ou na perda de informações devido as dependências da classe. Vale destacar que para isto, tanto a rota quanto o método `destroy` foram removidos, sendo substituídos pelas respectivas implementações de `hide` e `unhide`.

Outro comportamento importante de ser levado em conta, diz respeito a exclusão de representantes e/ou contas de usuários, de forma que caso possuam vínculo não poderão ser apagados, abrindo assim debate para segurança do sistema, em relação ao acesso as informações. Até o momento, o registro de usuário estava sendo realizado pela `gem` **Devise**, de modo semelhante a uma rede social, isto é, onde qualquer usuário pode se cadastrar ou excluir sua conta, contudo, para um sistema compartilhado e privado, isto não é o correto. Portanto, em relação ao gerenciamento dos usuários do sistema, abre-se debate para uma futura implementação, que seria a de possuir um usuário administrador capaz de gerenciar todos os usuários, visto que, mediante as configurações atuais, caso um usuário que possua acesso a aplicação, por algum motivo se desligasse da empresa, ainda assim possuiria acesso a sua conta. Além do mais, para excluir sua conta é necessário logar no sistema, e uma vez que esteja logado no sistema para fazer tal ação, o usuário ainda tem acesso a base de dados, expondo assim uma vulnerabilidade perante um dos medos relatados pela empresa colabora durante a fase inicial deste projeto.

Para corrigir tal comportamento, foi necessário implementar o gerenciamento de contas, assim como adicionar atributos de administrador as contas, de forma que usuários do tipo administrador pudessem gerenciar contas e conceder acessos de administrador aos demais usuários do sistema. Com isto, os recursos de Devise foram restringidos apenas a

autenticação de contas. A criação de contas por sua vez, passou a ser integrada ao menu de configurações da aplicação, sendo visível apenas para usuários administradores. Por fim, ainda por se tratar de relacionamentos e dependências entre objetos do sistema, uma dúvida inicial, abordada pela empresa colabora, no que diz respeito a cada usuário possuir acesso apenas as suas informações e um usuário administrador possuir acesso a tudo, não foram de fato consideradas relevantes suficiente para este projeto, pois além de não possuir regras de negócio bem definidas, foram consideradas regras de negócio particulares da empresa mediante a sua cultura de segurança com os dados. Outro motivo que reforça tal escolha, corresponde ao fato do protótipo desenvolvido durante o estudo realizado na empresa e de que o mesmo não possuía tais características e listava todos os dados do sistema a todos os usuários, e mesmo assim, não houve *feedback* negativos em relação a esta questão.

Ao fim destas alterações é possível se obter uma aplicação *web* pronta para uso, com testes que garantem o bom funcionamento, e com recursos testados em diversos cenários. É importante destacar que até o presente momento, todas estas alterações ainda encontram-se apenas no ambiente de desenvolvimento local, possuindo assim uma grande atualização a ser realizada em produção, a qual sera descrita no tópico a seguir, através de fluxos de trabalhos automatizados.

5.5 Pipelines de Automação

No desenvolvimento *web*, uma prática comum para acelerar a entrega de novas versões de software é o uso de *pipelines*, ou *workflows* como também são conhecidos. Essas configurações permitem automatizar fluxos de trabalho que constroem, testam e implantam novas versões do projeto em desenvolvimento.

Uma maneira de adicionar mais segurança ao projeto e adequá-lo para manutenções futuras de forma organizada, foi realizado a configurações dos processos de Integração Contínua, CI (*Continuous Integration*) e Implantação Contínua, CD (*Continuous Delivery*). Tais pipelines estabelecem uma série de testes para garantir a integridade do sistema assim como o envio automático da nova versão desenvolvida para o servidor de hospedagem da aplicação, neste caso Heroku.

5.5.1 Integração Contínua

O fluxo de trabalho definido no processo de CI é descrito no arquivo `.github/workflows/ci.yml`. É importante manter tal estrutura para o Github execute corretamente os pipeline ao realizar o versionamento do código. De forma simples e intuitiva o arquivo precisa especificar alguns parâmetros essenciais para seu funcionamento. São eles a definição do nome do pipeline, os tipos de ações que irão acionar a execução deste arquivo e em quais ramificações do repositório ele sera executado. Uma vez definidos é possível especificar as tarefas que serão executadas no fluxo de trabalho e todos os detalhes sobre o ambiente de execução.

Para este projeto configurações semelhantes as do Docker foram definidas, de forma que definem o ambiente como a última versão do Ubuntu, o uso da versão Ruby 3.0.0 e o banco de dados em Postgres.

É importante compreender antes de abordarmos os detalhes do fluxo de trabalho que o *pipeline* de Integração Contínua tem como objetivo garantir que a aplicação esteja funcionando como o esperado, realizando assim a configuração da aplicação em um ambiente independente e em seguida, realizando todos os testes da aplicação. Uma Vez que tudo ocorra conforme o esperado o *pipeline* é dado como concluído com sucesso e deve permitir que a atualização seja integrada ao projeto sem problemas.

5.5.2 Implantação Contínua

Já o termo *Continous Delivery*, referente a implantação contínua, trata a respeito a replicar as atualização implementadas no ambiente de desenvolvimento para o ambiente de produção, fazendo isto de forma ágil e praticamente automática. Para fazer isto, de forma semelhante ao *workflow* de integração continua, bastou definir o nome do *pipeline*, seus gatilhos, e por fim suas ações.

Para fazer isto de forma simples a biblioteca **akhileshns/heroku-deploy@v3.12.14** disponível em GitHub Actions foi utilizada. Para fazer a configurações corretas, de modo que após uma ramificação for mesclada à ramificação principal do projeto, estas atualizações sejam replicadas no Heroku, foi necessário associar `heroku_api_key`, `heroku_app_name` e `heroku_email` nas configurações do repositório do github, para quando o *workflow* seja executado, interprete estas variáveis corretamente e tenha acesso permitido a plataforma Heroku.

Com isto configurado, foi possível realizar a implantação de todas as correções que haviam sido realizadas desde a primeira implantação. Ou seja, muitas alterações de uma única vez, contudo, mediante o funcionamento do CI/CD é possível tornar o processo de desenvolvimento mais ágil para as futuras implementações. Uma vez configurados, estes *pipelines* garantem que todas as futuras implementações passem pelos testes e caso sejam bem sucedidas sejam implantadas sem dificuldades no ambiente de produção.

6 ANÁLISE E DISCUSSÃO DOS RESULTADOS

O processo de desenvolvimento do sistema exigiu muito conhecimento, tempo e dedicação para ser implementado. Sem dúvida o *framework* Ruby on Rails foi a escolha certa para desenvolver este projeto, considerando sua facilidade de integração entre os serviços, `gems` e banco de dados e toda a aplicação de maneira intuitiva torna o desenvolvimento uma tarefa prática extremamente satisfatória. Vejamos nesta sessão algum resultados obtidos através dos conceitos empregados no desenvolvimento deste projeto.

6.1 Desempenho com Metodologias

Assim como os conceitos que John Dewey defende em sua metodologia, o aprendizado baseado na interação do usuário com o ambiente em que ele se encontra surge através da adaptação, muitas vezes sendo guiados por perguntas que os levam a vários caminhos, de forma que o desenvolvimento de todo o processo de criação desta monografia, seja ela em abstrair um problema real, compreender o cenário e os dados envolvidos e desenvolver uma solução para esta situação.

A adaptação em que o autor se refere, pode ser interpretada como o ambiente em que se esta sendo estudado, sendo em seu aspecto de pesquisa, de testes, de experimentos ou qualquer outra abordagem. Vale destacar que mediante tais experiências vivenciadas ao longo do desenvolvimento deste projeto, tão como do aprendizado obtido, muitas perspectivas sobre como melhorar o código da aplicação, ou como estruturar os diretórios e arquivos do ambientes de desenvolvimento evoluíram ao longo do processo. É possível afirmar que mediante tal aprendizado, a sensação de se sentir preparado para avançar em um próximo projeto é plena.

Considerando a ideia da metodologia de aprender fazendo, a maneira de abordar o contudo foi partindo de princípios básicos para mais complexos, conforme descrito ao longo deste documento, criando interfaces primeiramente para cidades, que é um dos modelos mais simples da aplicação, e evoluindo gradativamente até chegar a páginas mais complexas como as de turmas, progredindo gradualmente ao longo do período de desenvolvimento.

Desta maneira, o processo de aprendizado se tornava mitigado, uma vez que os avanços da aplicação eram realizados de modo curto e rápido, nos quais eram desenvolvidos pequenos recursos de modo individual, em seguida, replicados e adaptados para as demais classes que fariam uso das implementações em questão. Em resumo, aprender um pouco por dia, com objetivos pequenos e com todo conteúdo disponível atualmente, trouxe resultados concretos a aplicação.

Também é possível afirmar de que caso todo este projeto fosse refeito, porém com o conhecimento atual, sem dúvidas iriam ser feitas inúmeras mudanças em relação a sua estrutura e como ele foi implementado, porém, isto não é atribuído ao fato deste projeto ser ruim ou

ter sido mal planejado, mas sim pelo fato de ter aprendido a como fazer as coisas da forma mais adequada no *framework* de forma pratica. Os testes e validações certamente não seriam a última etapa de implementação do projeto e que a utilização de algumas `gems` assim como a implementação de determinados formulários e modelos de objetos do sistema seriam revisadas com mais cautela.

O método como o trabalho foi desenvolvido, se baseando em implementações mais básicas para interfaces mais complexas, sendo guiado por objetivos tangíveis e perguntas que instigam a produtividade, se mostraram uma boa forma de se desenvolver, fazendo a ressalva de que este trabalho foi desenvolvido de forma individual. Sendo totalmente diferente para um cenário de desenvolvimento em time.

6.2 Resultados Obtidos

Como resultado deste projeto uma aplicação *web* surge com a proposta de ajudar empresas de formaturas a mapearem seu clientes de uma forma inteligente e eficiente, listando instituições por regiões e turmas por semestres e tornando possível traçar uma melhor estratégia de venda, beneficiando assim a empresa mediante ao uso regular do sistema e suas informações cadastradas na base de dados. Em âmbito comercial, ainda é possível ver no sistema uma ferramenta capaz de ranquear instituições de ensino, visto que podem oferecer uma certa vantagem comercial baseada nas estimativas de formandos por semestre influenciando diretamente na qualidade final dos serviços entregue aos formandos.

Considerando que aplicação se originou de uma planilha, ainda que agora estivesse sendo gerenciado por um banco de dados robusto, tornar a integração com ferramentas de análise de dados como o Microsoft Power BI poderia ser total viável e fácil de fazer, seja através uma API ou até mesmo conectado diretamente com o banco de dados da aplicação em real-time, proporcionando ao usuário final *dashboards* com estatísticas de forma analítica e rica em detalhes.

Por seguir a convenção do `framework` a aplicação apresenta um nível de abstração fácil para qualquer programador iniciante em *Rails*, tornando a manutenção ou contribuição algo prático de ser feito. Além disto, a documentação da aplicação apresentada no repositório da mesma, explica como realizar a instalação e configurar o projeto em seu computador, além de proporcionar as orientações iniciais ao usuário através tanto de forma descritiva, quanto na própria aplicação após estar devidamente configurada.

Mediante aos objetivos propostos inicialmente para este trabalho de conclusão de curso, é possível afirmar que tais objetivos foram alcançados com sucesso, onde, durante a etapa desempenhada durante a disciplina de Trabalho de Conclusão de Curso I, foi possível compreender o ambiente proposto, propor a atualização das ferramentas defasadas com um protótipo, com este, analisar as informações contidos em sua base de dados para então definir a modelagem do sistema, estruturando toda a documentação pertinente ao mesmo.

Já durante a segunda etapa deste projeto, realizada durante a disciplina de Trabalho de

Conclusão de Curso II, também é possível se afirmar que os objetivos propostos foram alcançados com sucesso, visto que se resumiam em desenvolver o sistema mediante a documentação elaborada, e como forma de concluir tal projeto, realizar a implantação em um ambiente produção real, deixando-a pronto para uso ao público.

Ainda é possível afirmar que ainda além destes objetivos, o projeto evoluiu muito além do esperando, conforme descrito ao longo deste documento sendo possível elencar as várias mudanças, alterações, refatorações e percepções a respeito da aplicação e seu *framework*. Ao transformar um protótipo funcional desenvolvido inicialmente em uma aplicação *web* capaz de gerenciar as informações de forma mais eficaz, é possível compreender o que sistema em sua natureza se propõe a fazer de forma clara.

Tendo em vista o escopo deste projeto, onde as planilhas sem validação alguma inicialmente foram se transformando gradativamente em uma aplicação *web* robusta, que auxiliar a solucionar as principais dificuldades relatadas pelo setor comercial de uma empresa de formaturas, é possível afirmar que a aplicação desenvolvida é sem dúvidas uma ferramenta adequada para o ambiente de trabalho.

Outra vantagem é que o sistema dispõe de um formulário público de prospecções que pode ser compartilhado facilmente, assim como possui sua área restrita, na qual é possível cadastrar representantes que tem o papel de assumir tais prospecções. Na área logada do sistema é possível que o usuário realize o cadastro de cidades em que a instituição presta serviço, cadastrar manualmente as instituições atendidas pela sua empresa assim como os respectivos cursos. A partir disto, é possível registrar Alunos em seus devidos cursos, e por sua vez registrar atendimentos as turmas e/ou aos alunos.

O sistema tem como proposta exibir as informações contidas em sua base de dados de forma intuitiva para que o usuário consiga interpretar as mesmas de forma fácil. Como resultado desta proposta, tal aplicação *web* acabou apresentando uma interface mais completa do que a planilha, e do que a do protótipo funcional, exibindo as informações categorizadas por semestres ou cidades, ordenando os dados de forma alfabética e tornando a interação do usuário mais fácil através da rolagem horizontal e vertical, e também das paginações presentes na aplicação.

Desta forma, o objetivo geral descrito por *Efetivar o aprendizado obtido durante a graduação de forma prática, desenvolvendo uma aplicação web, que seja capaz de gerenciar Alunos e Turmas de diversas Instituições e Cidades, através de registros de Prospecções, Atendimentos e Negociações* pode ser tido como concluído.

Após toda série de testes, refatorações e refinamento da aplicação, o sistema por vez, esta apto para ser implantado em uma empresa de forma real, pode ser comparado aos demais softwares disponíveis no mercado.

6.3 Conhecendo a Aplicação Web

Considerando que a aplicação *web* esteja devidamente implementada, é possível que qualquer pessoa possa acessar a aplicação para conhece-la de forma prática, bastando acessar o endereço: `tcc-utfpr.herokuapp.com`, e acessando a plataforma com as credencias: `admin@admin`, senha: `admin`.

Vejamos a seguir algumas imagens capturadas da aplicação em sua versão final com o objetivo de conhecer os principais recursos e como eles são exibidos na aplicação. Este seção pode ser utilizada como ponto de comparação em relação ao protótipo funcional.

Na Figura 20 é possível visualizar a tela de Prospecções com uma base de dados limpa, isto é, sem nenhuma prospecção registrada.

Figura 20 – Tela de Prospecção - Sem Registros
Fonte: Autoria Própria

Na Figura 21 é possível observar o comportamento da mesma tela, porém com registros de prospecções salvos na base de dados.

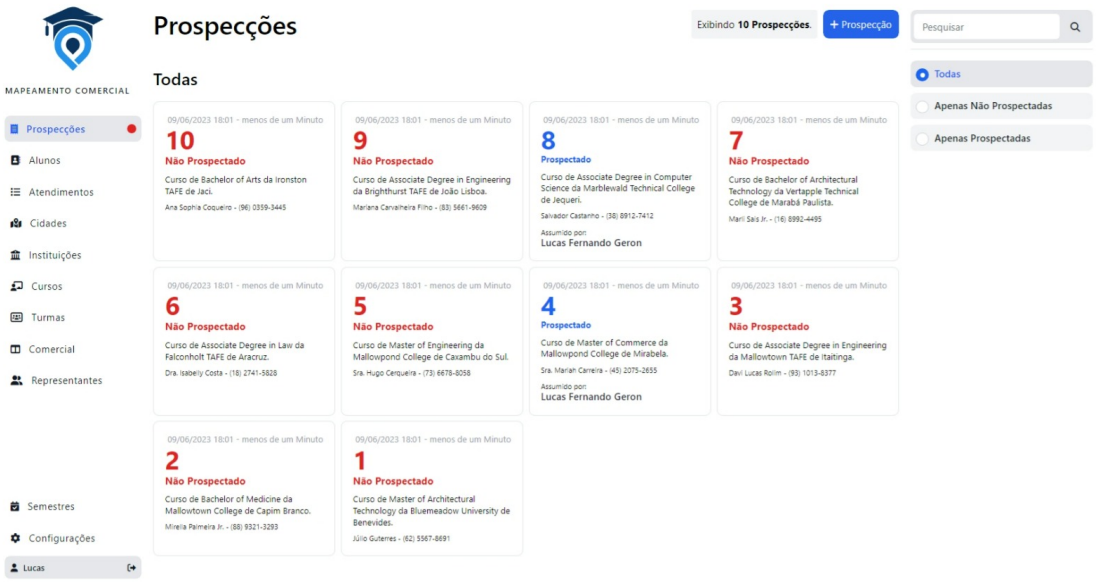

Figura 21 – Tela de Prospecção - Com Registros
Fonte: Autoria Própria

As Figuras 22 e 23 evidenciam o mesmo comportamento, porém, exibindo os Alunos registrados na base de dados.

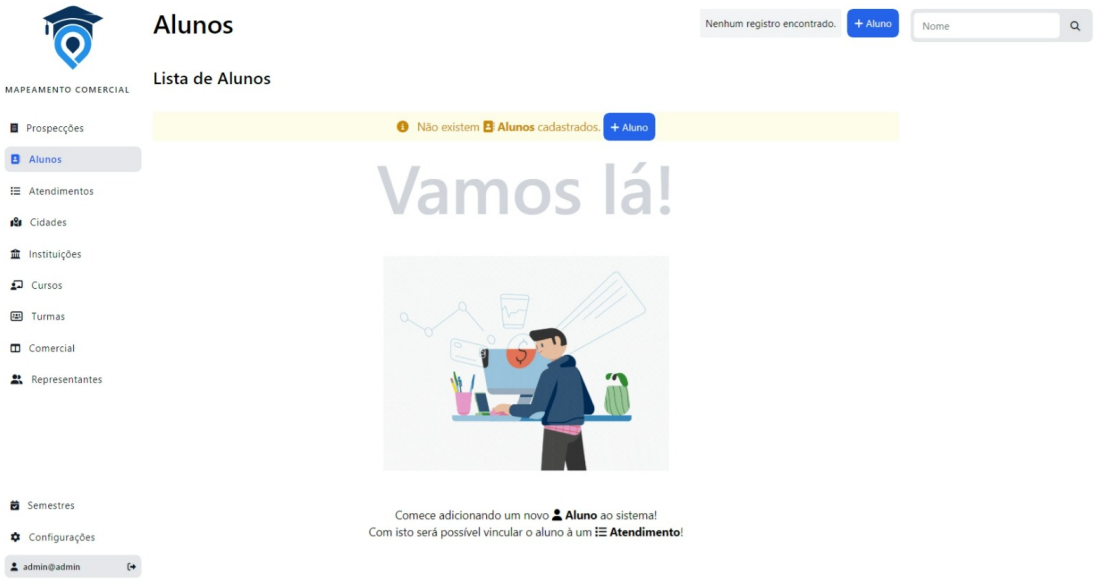

Figura 22 – Tela de Alunos - Sem Registros
Fonte: Autoria Própria

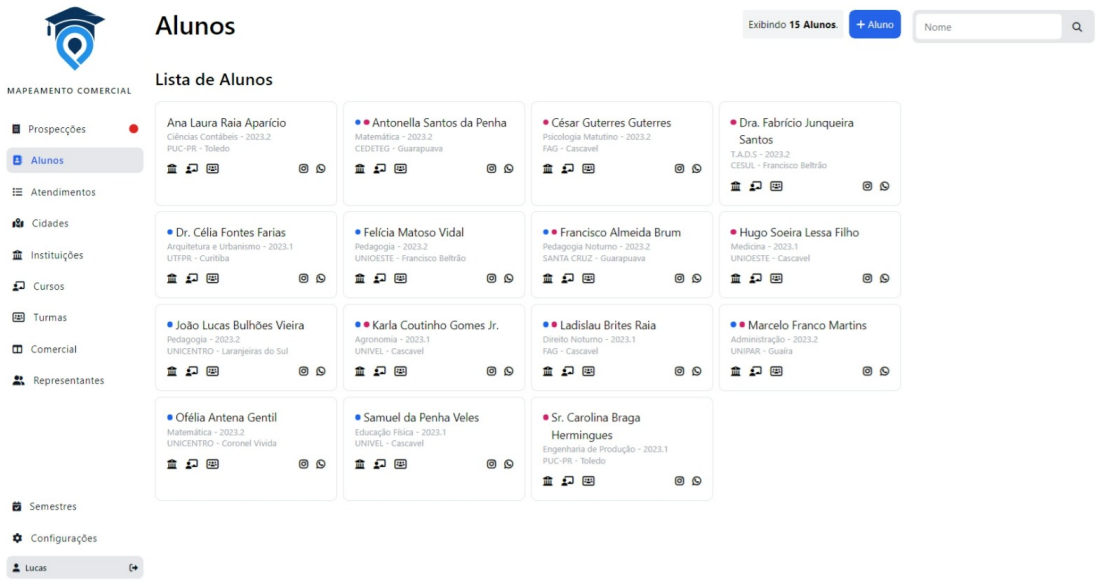

Figura 23 – Tela de Alunos - Com Registros
Fonte: Autoria Própria

Na Figura 24 é possível observar a tela de Atendimentos de uma base de dados vazia.

Figura 24 – Tela de Atendimentos - Sem Registros
Fonte: Autoria Própria

Já na Figura 25 é possível observar a mesma tela porém com atendimentos realizados.

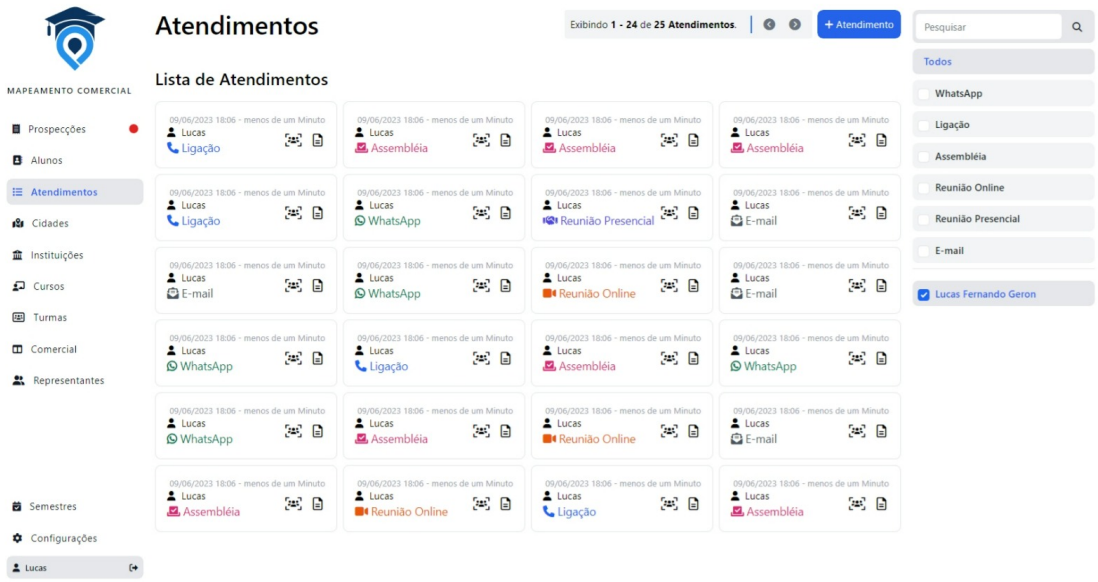

Figura 25 – Tela de Atendimentos - Com Registros
Fonte: Autoria Própria

As próximas figuras evidenciam a aplicação com os dados préviamente cadastrados, informações que foram originadas do estudo previamente realizada e importadas em pelos arquivos de seed personalizados. A Figura 26 mostra o registro das cidades.

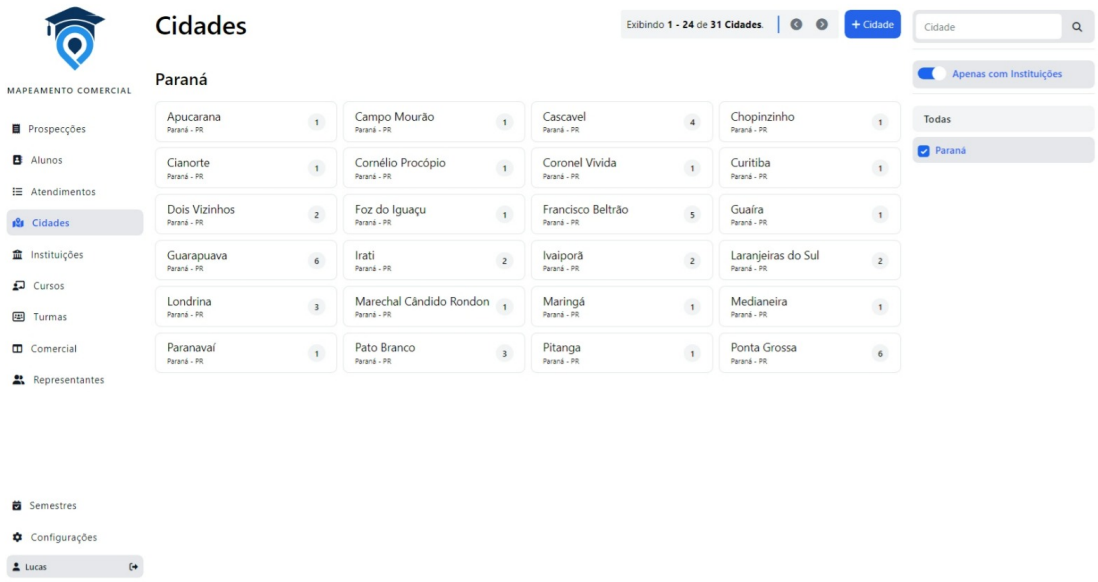

Figura 26 – Tela de Cidades - Com Registros
Fonte: Autoria Própria

Na Figura 27 é possível visualizar a lista de instituições cadastradas do sistema com sua respectivas logos.

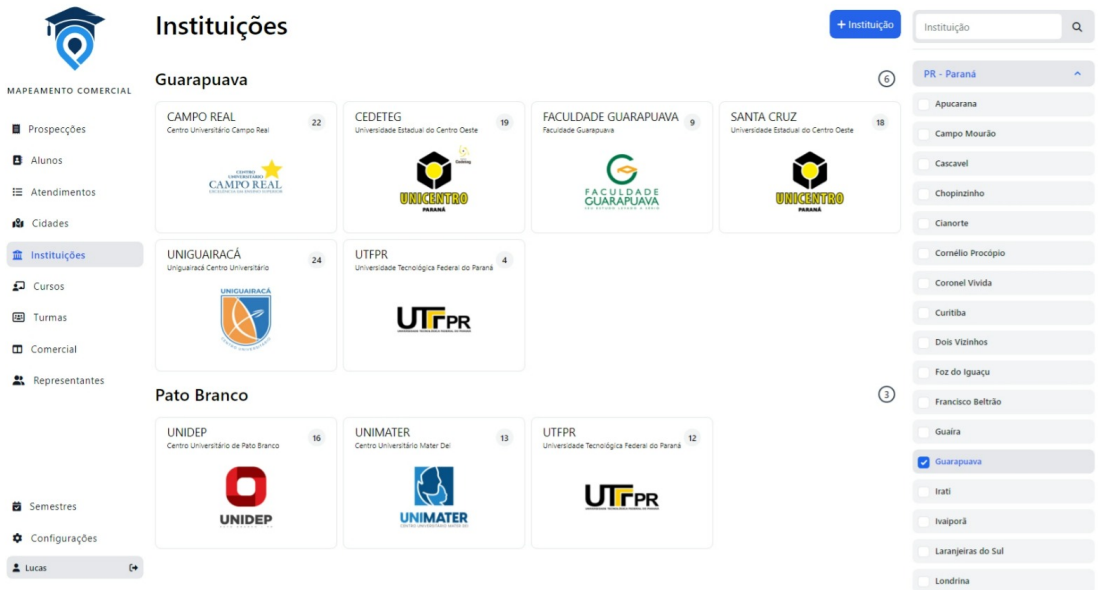

Figura 27 – Tela de Instituição - Com Registros
Fonte: Autoria Própria

Na Figura 28 são listados os cursos das instituições.

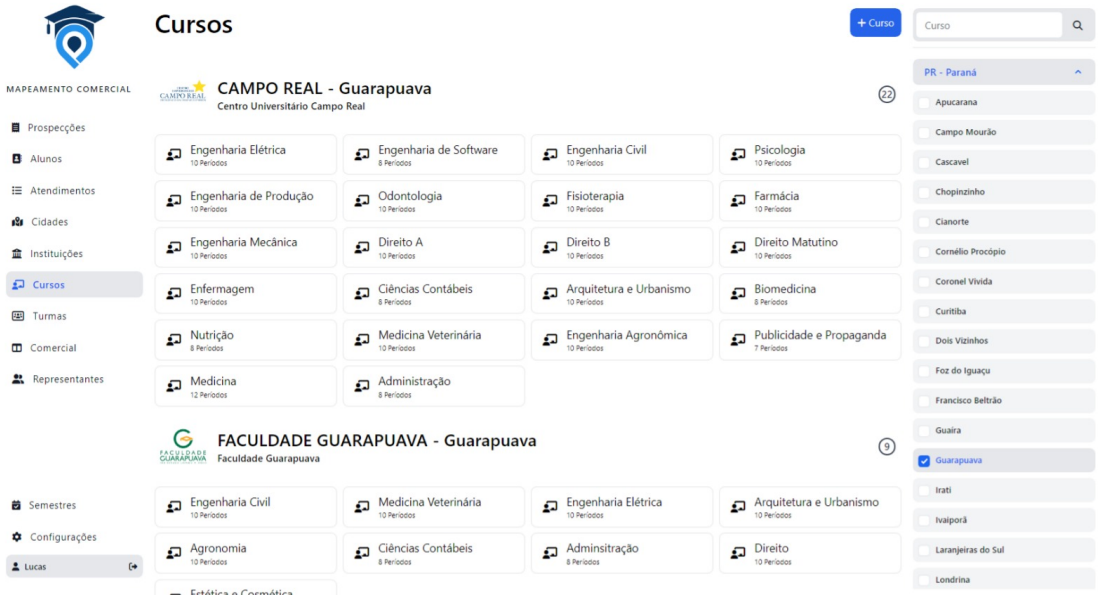

Figura 28 – Tela de Cursos - Com Registros
Fonte: Autoria Própria

Na Figura 29 é possível observar as turmas de uma determinada instituição, um ponto a ser observado nesta imagem diz respeito a rolagem horizontal, que permite ao usuário ter uma melhor visualização das informações uma vez que os *cards* são ordenados em ordem alfabética e seguem a mesma ordem baseando-se nos semestres abertos registrados no sistema.

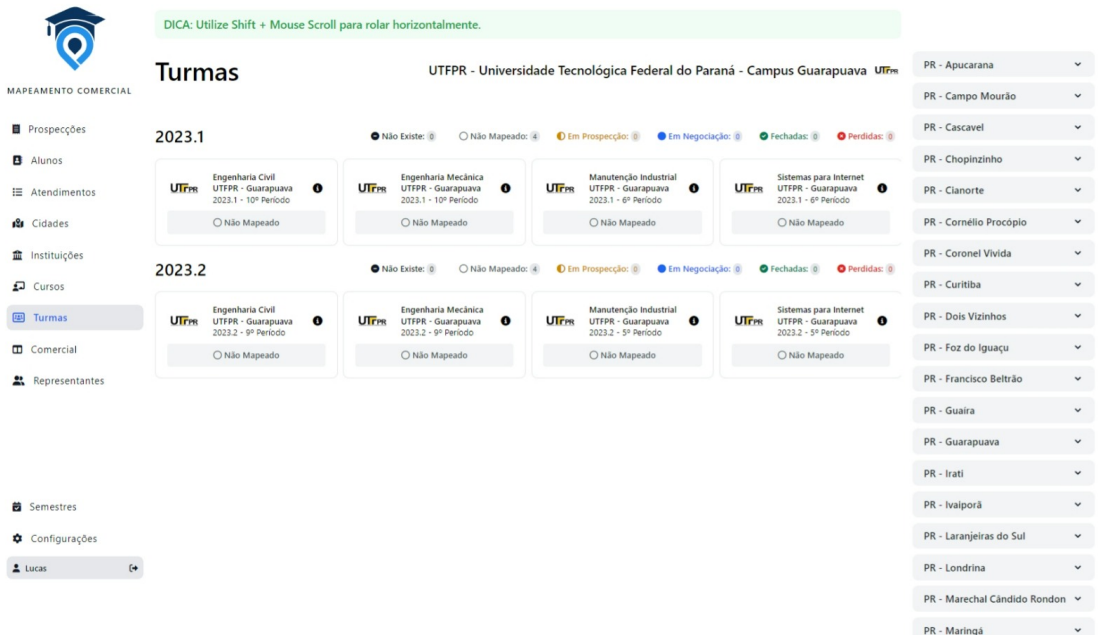

Figura 29 – Tela de Turmas - Com Registros
Fonte: Autoria Própria

Já a Figura 30 evidencia as negociações baseadas em seu status, neste caso, filtrando todas as turmas que se encontram 'Não Mapeadas'. Esta tela também apresenta as métricas e estatísticas de cada semestre, facilitando a interpretação das informações.

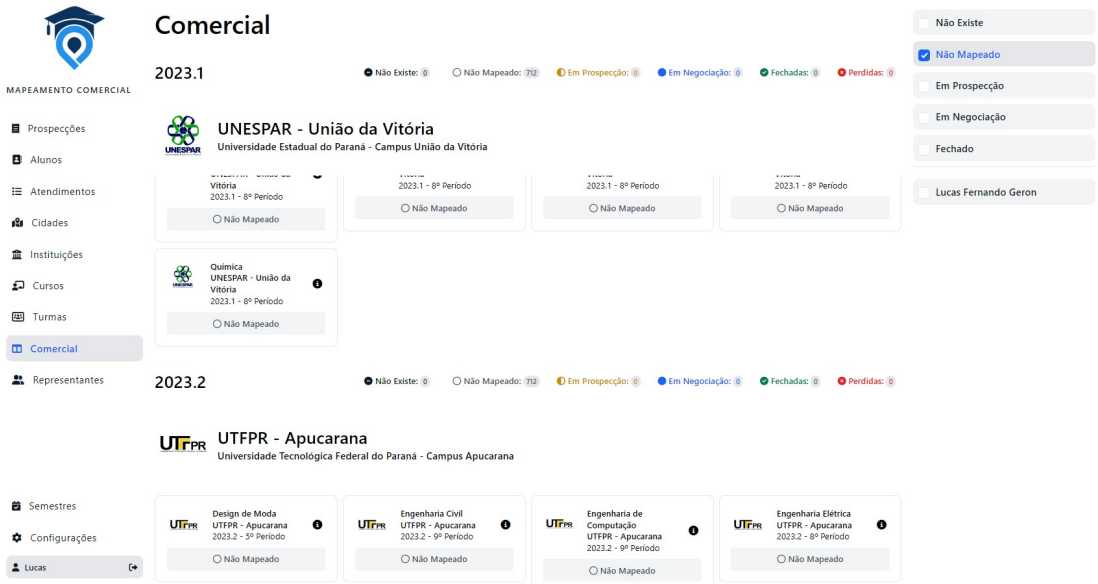

Figura 30 – Tela de Comercial - Com Registros
Fonte: Autoria Própria

A Figura 31 exibe tanto as empresas concorrentes cadastradas, quanto as configurações básicas do sistema.

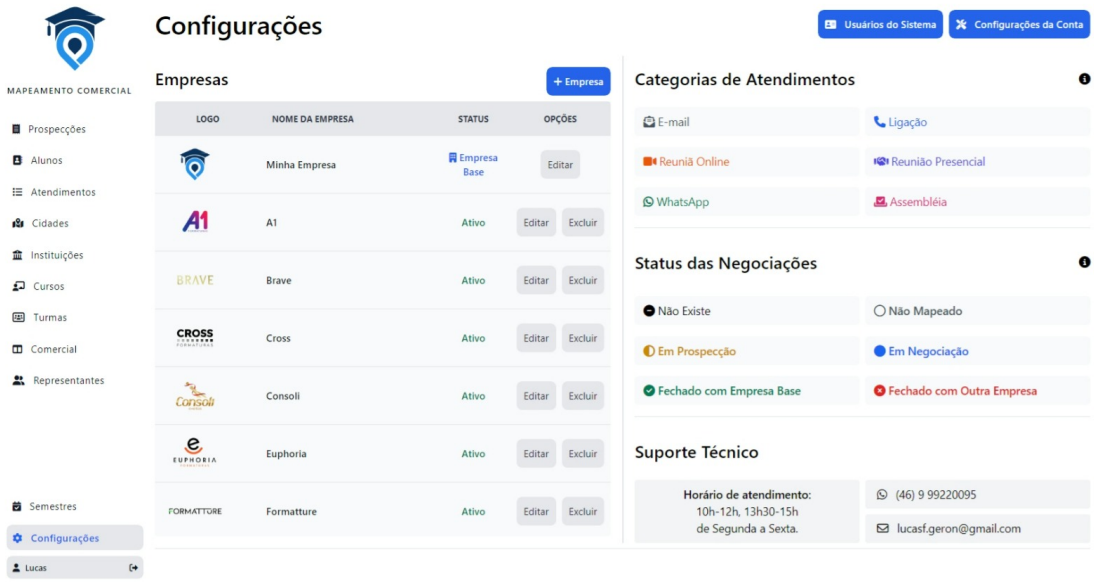

Figura 31 – Tela de Configurações - Com Registros
Fonte: Autoria Própria

Na Figura 32 é possível observar a tela de edição de contas de usuário, a partir de um usuário que possui acesso administrador habilitado.

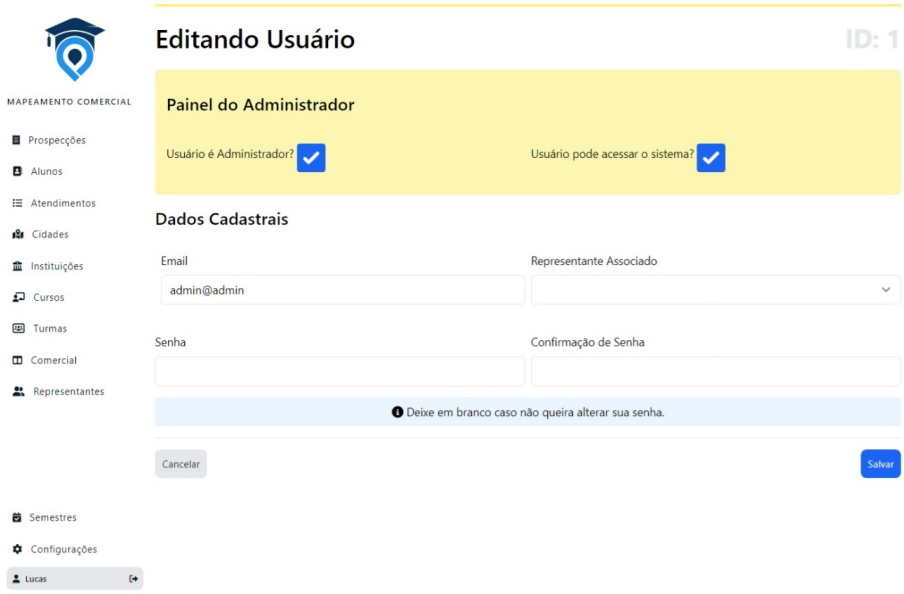

Figura 32 – Tela de Edição de Usuário - Acesso Administrador
Fonte: Autoria Própria

Uma vez que através destas imagens é possível ter um *overview* sobre as principais telas de aplicação, é possível avançarmos para o comparativo de seus recursos com os demais sistemas disponíveis.

Vejamos no tópico a seguir se a aplicação desenvolvida durante este projeto apresenta vantagens ou desvantagens perante aos demais produtos e se ao fim desta comparação, pode ser

de fato tida como ferramenta concorrente ou como ferramenta complementar a infraestrutura de uma empresa que atua na segmento de formaturas.

6.4 Comparativo entre sistemas

Considerando seus recursos implementados, é possível comparar o sistema aos demais softwares disponíveis no mercado. Os dois softwares, previamente citados neste documento que serão abordados como principais concorrentes no segmento são originalmente o sistema utilizado pela empresa colaboradora, denominado ControlEventus, desenvolvido pela PRONET e o outro software do com uma proposta semelhante que se destaca no mercado é o Agendor.

6.4.1 Control Eventus - Pronet

A proposta principal do ControlEventus é declaradamente realizar a gestão de todos os dados de uma empresa de formatura. Com viés fortíssimo em aspectos contábeis, o sistema já vem recebendo atualizações a mais de 15 anos. O ControlEventus é um sistema *desktop* que oferece *interface web* para alguns recursos básicos do sistema, como cadastro de formando e função de e-comerce para produtos e eventos.

O sistema possui diversos módulos que abrangem todo o segmento de atuação de uma empresa de formatura, isto é, gerenciando dados de fornecedores, contratos, contratações, calendário de eventos, contas a pagar e receber, romaneio de vendas, controle de estoque e uma série de ferramentas desenvolvidas para serem utilizadas no cotidiano, porém, o comparativo em relação ao sistema Control Eventus, busca abordar apenas os aspectos desenvolvido apenas no módulo comercial.

Se tratando das prospecções, o sistema permite registrar em seu módulo de alunos, todos os alunos, tanto contratantes, quanto prospecções. Por sua vez, as informações do aluno como, aonde ele estuda, qual sua turma e instituição, precisam ser digitadas manualmente ou precisam estar previamente cadastradas pelo próprio aluno, em ambos os casos, para registrar uma prospecção é necessário selecionar o aluno manualmente em um menu de seleção, buscando-o em uma base de dados que contem todos os registros, sem filtros de pré-seleção.

O módulo aluno permite que qualquer usuário do sistema registre um *follow-up*, recurso semelhante a uma anotação, ou um atendimento, possuindo assim o módulo de 'Notes' em dois ambientes, um sendo acessado diretamente pelo cadastro do aluno e o outro pelo centro de controle, nome utilizado para se referir a um contrato dentro do sistema. Desta forma, é possível registrar *follow-ups* para alunos, ou para contratos (centro de controles).

Considerando os demais módulos do sistema como financeiro, pacotes contratados e itens adquiridos, o cadastro do aluno é bastante robusto, sendo composto por vários campos e várias abas conforme é exibido na Figura 33.

É importante nesta abordagem, considerarmos que, por utilizar o mesmo módulo para prospectos e clientes, o sistema se é tido como de difícil utilização devido a extensão dos

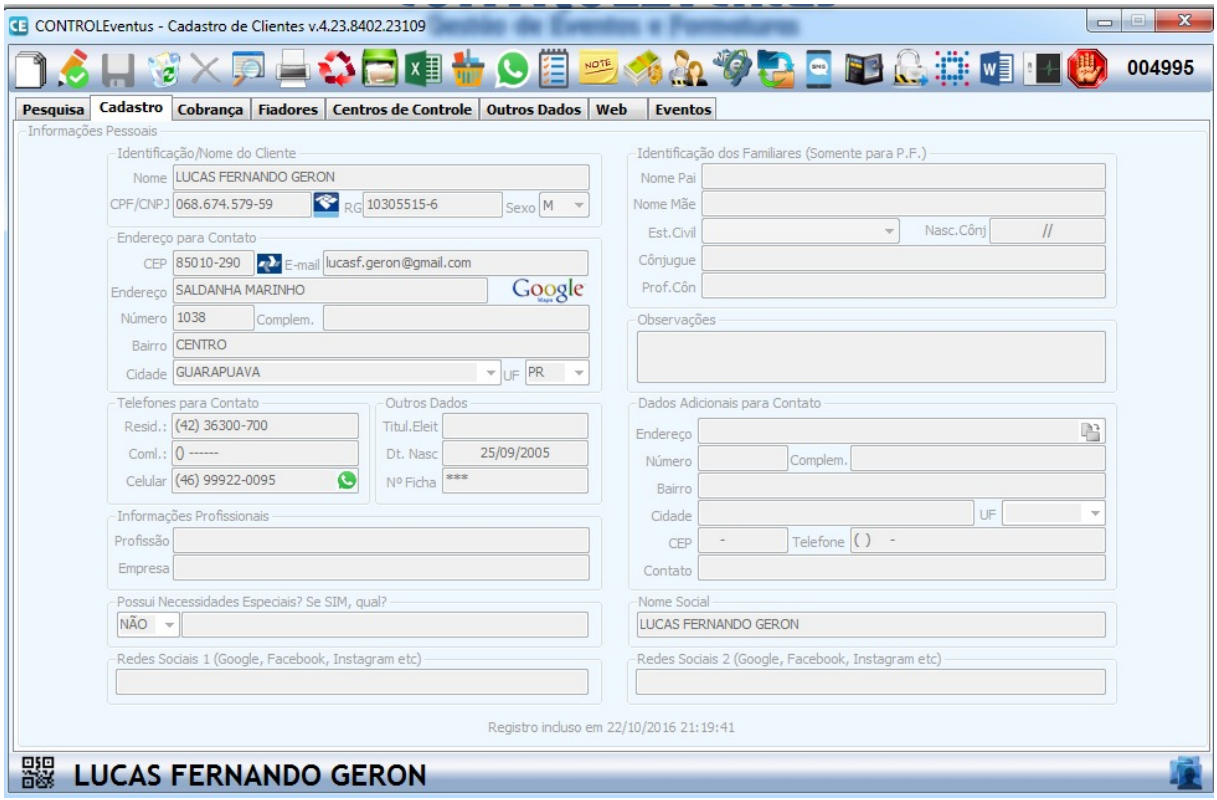

Figura 33 – Tela de Cadastro - ControlEventus
Fonte: Control Eventus 4.0 - Pronet

formulários e a da quantidade de campos que precisam ser preenchidos por um usuário do sistema, não pelo próprio prospecto, tornando a maioria dos cadastros incompletos ou com preenchimentos errados.

Também é valido citar que devido o módulo de notas ser uma extensão da ferramenta de cadastro do aluno, as notas são listadas de acordo com o cadastro que esta sendo exibido, de modo que não é possível visualizar uma lista de todos as notas registradas por um representante por exemplo, apenas todas as notas daquele aluno. Para se obter este tipo de relatório, o permite a geração de relatórios, que trazem os dados em formato de lista, muitas vezes, quebrando a formatação da página devido ao conteúdo contido na anotação.

Conforme a Figura 34 demonstra, a tela das notas lista todas as notas do aluno e possui uma *interface* compacta e é possível perceber que o sistema também cria notas automáticas baseado nas ações do usuário, como enviar uma mensagem automática de cobrança ao aluno, mantendo assim os dados ordenados de forma cronológica, mas dificultando muito a busca de informações realizadas a partir de um usuário do sistema.

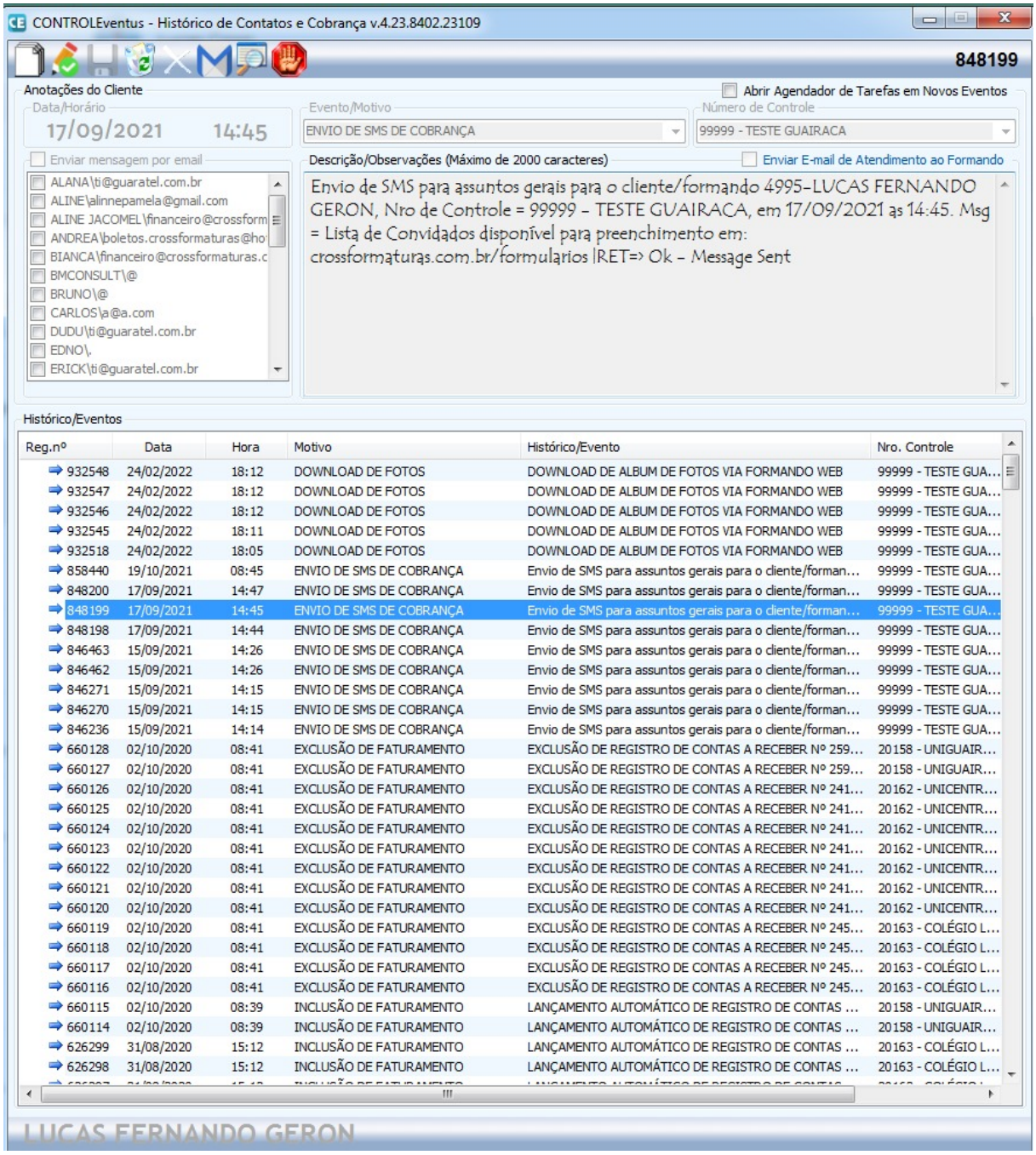

Figura 34 – Tela de Notas do Aluno - ControlEventus

Fonte: Control Eventus 4.0 - Pronet

Concluí-se então que a proposta do Mapeamento Comercial, que por sua vez, não possui módulos complementares tão robustos quanto aos do Control Eventus, é menor, registrados apenas dados simples, porém, permitindo mapear as informações de uma forma mais dinâmica, sendo possível abstrair a utilização de ambos os softwares como sistemas complementares, e não concorrentes do mesmo segmento.

Ainda é possível afirmar que ambas as aplicações possuem o mesmo objetivo, o de registrar os alunos e suas respectivas negociações com suas devidas turmas, assim como mapear as novas prospecções. Também é possível afirmar que o Pronet possui um escopo muito mais

amplo do que aplicação desenvolvida e muito mais tempo de desenvolvimento, possuindo vários recursos complementares para realizar a gestão empresarial com mais facilidade, coisas que o sistema atual não possui. Sob tal perspectiva é possível afirmar que ambos as aplicações, apesar de suas diferenças, atendem o que lhes é proposto.

6.4.2 Agendor - Agendor

Em relação a aplicação *web* desenvolvida pela Agendor, é possível afirmar que a ferramenta se enquadra no modelo de aplicação genérica, atendendo assim uma gama maior de usuários, porém, sem especificações técnicas de nenhuma área. Desta forma, a plataforma consegue empregar maiores esforços em soluções integradas, como site responsivo, aplicativo para dispositivos móveis, conexão direta com WhatsApp[1] para registrar o histórico de mensagens, funil de vendas e acompanhamento da negociação ordenados e agrupados por diversas condições. A Figura 35, ilustrada a seguir mostra a tela de Relatórios do sistema.

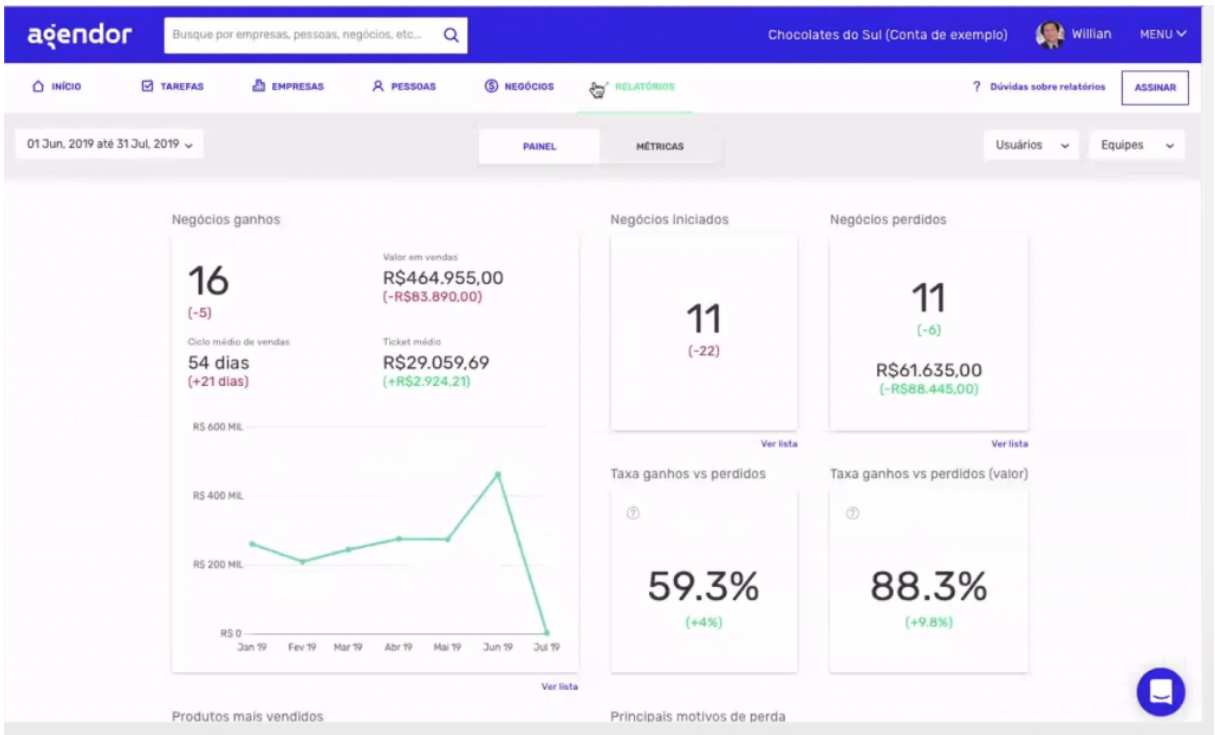

Figura 35 – Agendor - Painel de Relatórios

Fonte: Blog B2B Stack - Matéria: Saiba tudo sobre o CRM brasileiro ideal para quem não pode gastar muito.

Portanto, mediante a comparação de recursos implementados no sistema, é possível afirmar que o Agendor é um software muito mais completo, com um objetivo focado em vendas para todo segmento, tornando assim o uso do Mapeamento Comercial um software com recursos simplificados e especializados no mercado de formaturas, podendo assim serem utilizados de forma complementar, pois considera que o Agendor pode facilmente gerenciar

[1]WhatsApp é um aplicativo multiplataforma de mensagens instantâneas e chamadas de voz para smartphones.

informações específicas de uma negociação, como pode exemplo o andamento da emissão do projeto orçamentário, se a reunião de negócios já foi agendada, ou os valores investidos com as negociações em andamento, tarefas simples que o mapeamento comercial não faz, contudo, acompanhar o andamento das turmas ao passar dos semestres assim como possuir as estatísticas de mercado são recursos que a aplicação entrega e não podem ser gerenciadas pelo Agendor.

É valido lembrar que o Mapeamento Comercial já pensando em ser utilizado como *software* complementar à infra-estrutura de uma empresa, possui o campo denominado `budget_num` atribuído à um `Degree`. Sendo do tipo texto, esta coluna permite o usuário atribuir a turma em questão com uma referência externa, seja ela qual for. Na Figura 36 é possível observar a tela Histórico de Negociações.

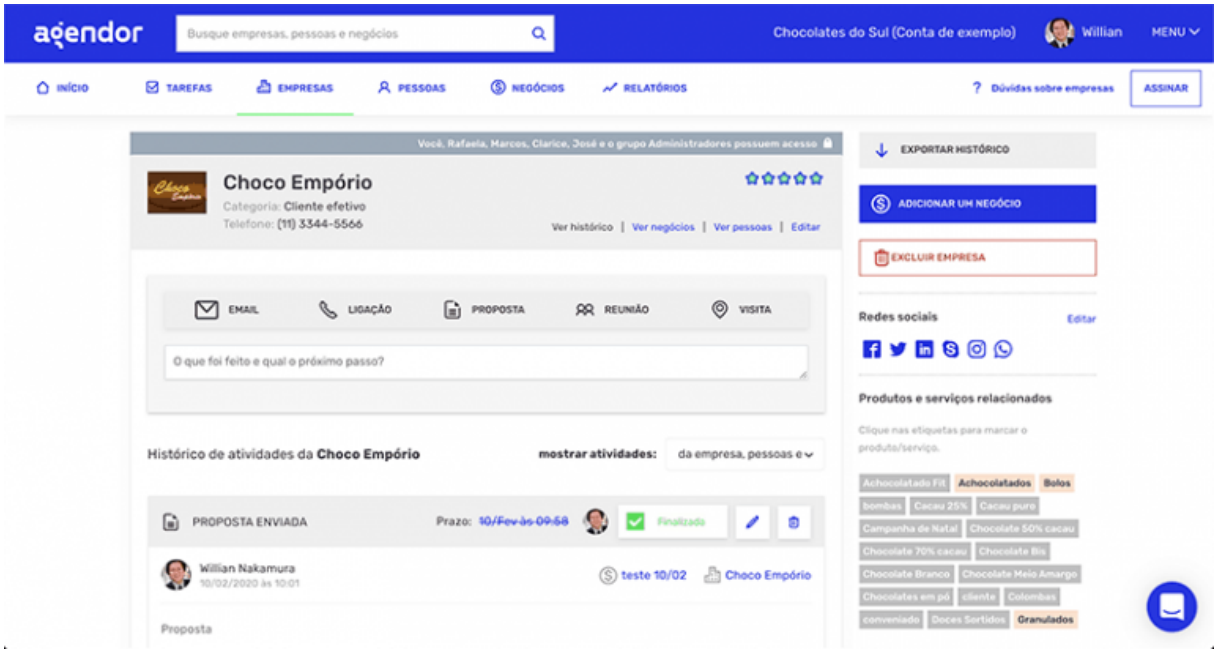

Figura 36 – Histórico de Negociação - Agendor
Fonte: Blog Agendor - Matéria: CRM Simples

Ainda sob uma visão comparativa, podemos afirmar que devido a capacidade de especialização do sistema desenvolvido, é possível se obter informações mais assertivas sobre o percentual de mercado das principais empresas da região, assim como as possíveis unificações entre turmas da mesma instituição, viabilizando assim contratos com valores maiores, recursos que as demais plataformas não oferecerem por possuírem mais recursos genéricos.

A comparação entre sistemas portanto não busca elencar qual sistema é melhor ou pior baseado em um série de critérios, apenas comparar seus recursos disponibilizados e como eles são utilizados, afim de evidenciar a inúmera variedade de *softwares* disponíveis atualmente, e quais as soluções oferecidas pelos mesmo, visto que cada um possui um objetivo como ideal de solução para um determinado problema.

7 CONCLUSÃO

O aprendizado obtido durante a jornada acadêmica e o aprendizado durante o desenvolvimento deste projeto foram etapas fundamentais para conclusão deste trabalho de conclusão de curso. Mediante ao atributos de um estudante, é possível afirmar que os objetivos propostos para se alcançar a graduação, o de desenvolver uma aplicação *web* e realizar a implantação da mesma em ambiente produção fora atingido.

Sem dúvida este projeto se estendeu por um longo período de tempo, no qual foi possível se obter muito conhecimento através da interação dos ambientes de trabalho e estudo, sendo fruto deste tempo investido uma aplicação *web* capaz de beneficiar empresas do ramo de formaturas, organizando suas negociações e mapeando suas turmas de formas intuitivas conforme descritos neste projeto e também disponíveis no repositório do GitHub.

Contudo, nos dias atuais, onde as aplicações precisam receber atualizações frequentes para garantir sua segurança, correções e novas implementações, é praticamente impossível se dizer que uma aplicação esta concluída. Portanto, pode-se afirmar que a aplicação desenvolvida esta em sua primeira versão, sendo dado como concluído apenas perante seu escopo inicial, porém viabilizando um universo de implementações futuras que podem vir a agregar recursos na aplicação. Vejamos no tópico a seguir algumas das situações que abordam melhorias ou novos recursos ao sistema.

7.1 Trabalhos Futuros

Este tópico tem como objetivo descrever o levantamento de novos requisitos que foram concebidos ao longo do processo de desenvolvimento, os quais por fugirem do escopo inicial do projeto, foram listadas para que possam ser implementadas futuramente, vejamos algumas delas.

- **Em Negociação com Representante durante X Dias**: Tal informação deve ser exibida no cartão de todas as Turma, mediante a variável em questão e pode possuir uma formatação condicional para informar se a negociação esta sem atualização a muito tempo;
- **Alunos associados a um Representante / Alunos são filtrados por cidades atendidas do Representante**: Não é muito coerente que um representante tenha o acesso ao contato de um aluno que não pertence a região na qual ele atenda, portanto, uma possibilidade seria exibir os alunos baseado apenas nas cidades atendidas do representante, ou ao realizar o cadastro do aluno, registrar o usuário que o cadastrou, tornando-o visível apenas o mesmo;
- **Log de Atividades do Representante**: Consiste na ideia de criar uma tela que exiba o *log* do usuário baseado no horário de alteração das atividades. É sugerido ainda que a rota

utilizada seja `/vendors/:id/log` e possua uma visualização mais interativa baseada mais nas ações do sistema pelo usuário, do que pelas atividades desenvolvidas em sí, de forma que estas não precisam ser exibidas nesta tela;

- **Permitir gerenciar Status de Negociações**: Este requisito sugere a implementação dos recursos que permitam adicionar mais 'Status' as Turmas. Semelhante as empresas, seria necessário criar todo o contexto da aplicação para tal funcionalidade. Tal implementação é considerada difícil e complexa, pois influencia diretamente nos filtros que a aplicação pode realizar, sendo necessário validar tais implementações em toda aplicação;
- **Permitir gerenciar Categorias nos Atendimentos**: Este requisito sugere a implementação dos recursos que permitam adicionar mais 'Categorias' nos Atendimentos. Semelhante as empresas, seria necessário criar todo o contexto da aplicação para tal funcionalidade. Tal implementação é considerada difícil e complexa, pois influencia diretamente nos filtros que a aplicação pode realizar, sendo necessário validar tais implementações em toda aplicação;
- **Permitir importar Alunos por .CSV**: Este requisito sugere a implementação dos recursos que permita importar os alunos através do edital de aprovação do SISU e das principais universidades que não utilizam o sistema como forma de ingresso, cadastrando todos os alunos em suas respectivas turmas;
- **Permitir filtrar por Todos**: Este requisito sugere que uma opção 'todos' seja adicionada aos filtros da aplicação, facilitando na busca por informações em uma base de dados com muitos registros. O grande desafio desta implementação é otimizar o código em uma base de dados robusta.

Além desta lista com requisitos específicos, é possível descrever algumas outras implementações de forma mais ampla devido ao seu contexto. Fato é que ainda que a aplicação já tinha sido pensada em como se comportar de forma responsiva, realizar a alteração do código ainda não foi um tarefa realizada devido as demais demandas pertinentes a monografia como um todo, contudo, tais implementações só dependem de tempo, visto que já sabemos o que precisa ser feito e como fazer.

Também é possível realizar uma grande alteração no projeto, que caso houvesse sido realizada desde sua criação, seria de complexidade irrelevante, contudo, devido a esta negligencia por ignorância, ou em outras palavras, falta de conhecimento, seria a utilização adequada dos recursos de internacionalização, também conhecido como *I18n*.

A internacionalização tem como principal objetivo reunir em um único arquivo todas as mensagens do sistema assim com as nomenclaturas de seus modelos e seus atributos, tornando o uso da aplicação mais legíveis ao usuário e viabilizando a tradução da aplicação para outros idiomas. Outra grande vantagem que este recurso oferece diz respeito a poder ser instanciado na aplicação em qualquer momento, o que facilita muito as alterações e implementações futuras. Contudo, fazer esta mudança após o projeto ter sido construído sem este recurso, pode tomar um tempo relativamente longo.

Durante a etapa de revisão e orientação do projeto é também é possível destacar algumas sugestões relevantes, pertinentes e complexas, como a de fazer uma alteração significa na forma em que o sistema calcula os períodos.

Considere que os períodos sofrem alteração apenas duas vezes por ano, na data em que se iniciam os semestres, desta forma, calcular estas informações toda vez que o usuário solicitar pode ser uma tarefa custosa, de modo que uma possível refatoração de código poderia implementar uma coluna adicional na tabela *Degrees* que armazenasse esta informação. Desta forma, não seria necessário calcular o período toda vez pois o mesmo seria informado em forma de uma consulta simples de dados, reduzindo assim significativamente o consumo de memoria do servidor e otimizando o desempenho da aplicação. A complexidade desta tarefa por sua vez diz repeito a formatação contida nos campos, informando quando as turmas disponíveis ou não, entretanto, o desempenho da aplicação certamente seria mais eficiente em relação ao modelo atual.

Outra sugestão que também envolve uma certa complexidade, diz respeito a criação de um serviço na aplicação que seja capaz de processar o comportamento da criação de turmas ocasionado pela criação de um novo curso, ou um novo semestre em segundo plano, deixando o usuário livre para navegar pela aplicação enquanto o serviço é executado. Para exemplificar este cenário, é preciso compreender como exatamente o sistema cria as turmas, sendo estas apenas de duas maneiras.

A primeira maneira seria criando um novo curso, a uma base de dados que já contenha semestres abertos. Desta forma, ao adicionar um novo curso, o sistema irá gerar uma nova turma para cada semestre aberto no momento. É valido lembrar que esta situação ocorre apenas quando as instituições resolvem ofertar novos cursos, ou novas modalidades dos cursos existentes. A segunda forma seria criando um novo semestre a uma base de dados que já possua cursos cadastrados. Desta forma, o sistema irá gerar uma nova turma para cada curso cadastrado no sistema.

Esta função por sua vez deverá ser realizada periodicamente, uma vez por semestre, ou por ano, conforme a abordagem da empresa, contudo, caso a empresa atenda muitos cursos, é possível que o sistema demore para realizar esta tarefa. No modelo atual do sistema, o usuário precisa esperar este processamento ocorrer para então ser redirecionado e receber a notificação de que o semestre foi criado. Certamente seria mais interessante que o sistema permitisse que este serviço realize esta tarefa em segundo plano, de forma assíncrona, possibilitando que o usuário possa utilizar a aplicação enquanto esta tarefa é processada pelo servidor.

A plataforma Heroku oferece algumas métricas interessantes referente ao consumo dos dados e desempenho da aplicação em seu ambiente de produção, ficando como uma possível área de pesquisa e/ou desenvolvimento futuro, otimizar ainda mais o desempenho da aplicação, fazendo isto através da implementação do HotWire para todo o contexto da aplicação ou com abordagens alternativas para os problemas da aplicação.

7.2 Considerações Finais

Fazer tudo isso na aprendendo na prática em sem dúvida muito gratificante, pois de fato cumpre os requisitos do que se propõe o título deste documento, um trabalho de conclusão de curso, no qual agrega todo o conhecimento aprendido durante a graduação na prática e no cenário real.

Portanto, é importante destacar que muito mais do que saber utilizar um *framework* moderno, a graduação proporciona uma base de conhecimento ampla a ponto de compreender que um sistema para internet é feito de uma maneira robusta, tendo assim a percepção de que um projeto bem estruturado precisa ser de fato estudado, debatido, analisado e implementado mediante metodologias de desenvolvimento e ao ambiente em que será utilizado, sendo otimizado com as possíveis tecnologias e recursos que a mesma oferece.

Ao longo do processo de implementação, ainda que houvessem falhas que tomaram bastante tempo, como a questão do JavaScript ou HotWire fazendo com que o código feito tivesse que ser descartado, assim como, ao perceber o `script` que complementava o funcionamento do protótipo com 210 linhas, o qual havia demandado um grande esforço para ser elaborado, era reescrito em Ruby em apenas 2 linhas de código e as associações com ActiveRecord, contudo, isto não era desanimador, pelo contrário, ver como trabalhar com Rails era muito mais eficiente agregava muito mais conhecimento do que fazer do jeito antigo.

Outro grande deslize que certamente fez o projeto perder um pouco seu brilho, foi o fato de não possui conhecimento adequado para trabalhar com `git`. Desta forma, o projeto foi feito fora da convenção de `commits`, sendo desenvolvido praticamente inteiro na ramificação principal, e tendo uma série de `commits` com várias alterações resumidos em poucas palavras, assim como `commits` com quase nenhuma alterações com mensagens abstratas, como pontos ou mensagem duplicadas. Tal comportamento fora adequado ao longo das implementações, onde após o CI/CD estarem operando, trabalhar com ramificações adequadas se tornaram obrigatórias.

Este histórico poderia ser refeito recriando o repositório do projeto, entretanto, possuir o histórico de como toda a aplicação foi construída é algo importante. Sendo assim, por mais que isto possa ser considerado uma prática ruim, é possível defender a ideia de que as falhas são parte do sucesso, e não o inverso dele. Ainda para tentar justificar tal situação, como estudante, não tive a oportunidade de participar de nenhum grande projeto que houvesse um uso compartilhado e uma padronização de `commits` como referência.

Apesar destas situações descritas neste documento, a capacidade de deter conhecimento sobre a aplicação é algo da qual posso me orgulhar, uma vez que demonstra real imersão no processo de aprendizado e comprometimento com desenvolvimento deste sistema, contemplando assim a capacidade de transformar uma ideia, em um projeto, e um projeto em algo tangível.

Referências

AGENDOR. **Soluções para gestão comercial B2B**. 2023. Disponível em: <https://www.agendor.com.br/solucoes>. Acesso em: 12 de março de 2023. Citado na página 6.

APPSHEET. **Usar colunas virtuais**. 2023. Disponível em: <https://support.google.com/appsheet/answer/10106758?hl=pt-br>. Acesso em: 15 de março de 2023. Citado na página 23.

AQUILES, A. **Controlando Versões com Git e GitHub**. [S.l.]: Casa do Código, 2014. ISBN 978-8566250534. Citado na página 15.

BACICH, L.; MORAN, J. **Metodologias ativas para uma educação inovadora**. Penso Editora Ltda, 2018. ISBN 978-85-8429-116-8. Disponível em: <https://curitiba.ifpr.edu.br/wp-content/uploads/2020/08/Metodologias-Ativas-para-uma-Educacao-Inovadora-Bacich-e-Moran.pdf>. Acesso em: 09 de março de 2023. Citado na página 8.

BRASIL. **Censo da Educação Superior 2020: notas estatísticas.** 2022. Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira INEP, Brasília, DF. Disponível em: <https://download.inep.gov.br/publicacoes/institucionais/estatisticas_e_indicadores/notas_estatisticas_censo_da_educacao_superior_2020.pdf>. Acesso em: 15 de março de 2023. Citado na página 4.

BRUCCE, M. **ABERFORM, - Associação Brasileira de Empresas de Formatura**. 2019. Disponível em: <https://www.abeform.com.br/>. Acesso em: 11 de março de 2023. Citado na página 1.

CRITOVÃO, F. **Pronet Software - Maringá**. 2023. Disponível em: <https://www.pronetsoftware.com.br/>. Acesso em: 14 de março de 2023. Citado na página 5.

DEVMEDIA, D. **Orientações básicas na elaboração de um diagrama de classes**. 2016. Disponível em: <https://www.devmedia.com.br/orientacoes-basicas-na-elaboracao-de-um-diagrama-de-classes/37224>. Citado na página 28.

DEVMEDIA, H. **Introdução ao Padrão MVC**. 2013. Disponível em: <https://www.devmedia.com.br/introducao-ao-padrao-mvc/29308>. Citado na página 35.

DEVMEDIA, L. **O que é UML e Diagramas de Caso de Uso: Introdução Prática à UML**. 2012. Disponível em: <https://www.devmedia.com.br/o-que-e-uml-e-diagramas-de-caso-de-uso-introducao-pratica-a-uml/23408>. Citado na página 27.

DOCKER. **Visão geral do Docker**. 2023. Disponível em: <https://docs.docker.com/get-started/overview/>. Citado na página 11.

FUENTES, V. B. **Ruby on Rails. Coloque Sua Aplicação Web nos Trilhos**. [S.l.]: Casa do Código, 2012. ISBN 978-8566250039. Citado na página 13.

GASPAR, C. D. O design na valorizaÇÃo de objetos produzidos pelo adulto deficiente. p. 189, 2013. Citado na página 9.

GOOGLE. **AppSheet: desenvolvimento de apps sem código**. 2023. Disponível em: <https://cloud.google.com/appsheet?hl=pt-br>. Acesso em: 15 de março de 2023. Citado na página 20.

GOOGLE. **Planilhas Google: planilhas on-line para empresas | Google Workspace**. 2023. Disponível em: <https://workspace.google.com/intl/pt-BR/products/sheets/>. Acesso em: 15 de março de 2023. Citado na página 21.

GOOGLE. **Vinculação de Conta do Google com OAuth**. 2023. Disponível em: <https://developers.google.com/identity/account-linking/oauth-linking?hl=pt-br>. Acesso em: 15 de março de 2023. Citado na página 21.

GOOGLE. **Visão geral do Google Apps Script**. 2023. Disponível em: <https://developers.google.com/apps-script/overview?hl=pt-br>. Acesso em: 15 de março de 2023. Citado na página 22.

LIMA, E. K. **No code: o que é, vantagens e usos**. 2022. Disponível em: <https://olhardigital.com.br/2022/12/26/tira-duvidas/no-code-o-que-e-vantagens-e-usos/>. Acesso em: 15 de março de 2023. Citado na página 20.

MARIOTTI, F. S. **Kanban: o ágil adaptativo**. 45. ed. [S.l.]: Engenharia de Software Magazine, 2012. Engenharia de Software Magazine. Citado na página 7.

MICROSOFT. **Documentação para Visual Studio Code**. 2023. Disponível em: <https://code.visualstudio.com/docs>. Acesso em: 15 de março de 2023. Citado na página 11.

MICROSOFT. **O que é o Subsistema do Windows para Linux?** 2023. Disponível em: <https://learn.microsoft.com/pt-br/windows/wsl/about>. Acesso em: 15 de março de 2023. Citado na página 10.

POSTGRESQL. **O que é o PostgreSQL?** 2023. Disponível em: <https://www.postgresql.org/about/>. Citado na página 11.

RAILSGUIDES. **O que é Active Record?** 2023. Disponível em: <https://guiarails.com.br/active_record_basics.html>. Citado na página 36.

RAILSGUIDES. **O que é o Rails?** 2023. Disponível em: <https://guiarails.com.br/getting_started.html#o-que-e-o-rails-questionmark>. Citado na página 12.

RAILSGUIDES. **Usando Partials**. 2023. Disponível em: <https://guiarails.com.br/layouts_and_rendering.html#usando-partials>. Citado na página 36.

RUBYONRAILS.ORG. **Rails Contributors - All time**. 2023. Disponível em: <https://contributors.rubyonrails.org/contributors>. Citado na página 12.

SALESFORCE. **Funil de vendas: saiba o que é e conheça suas etapas**. 2023. Disponível em: <https://www.salesforce.com/br/crm/funil-de-vendas/#oque>. Citado na página 1.

SERASA, E. **Ir à falência: veja os principais motivos e como evitá-los**. 2020. Online. Disponível em: <https://empresas.serasaexperian.com.br/blog/ir-a-falencia-veja-os-principais-motivos-e-como-evita-los/>. Acesso em: 15 de março de 2023. Citado na página 1.

SERRADAS, V. **Desenhando cards: boas práticas**. 2018. UX Collective BR. Disponível em: <https://brasil.uxdesign.cc/https-brasil-uxdesign-cc-cards-boas-praticas-6ae813acf8cf>. Acesso em: 15 de março de 2023. Citado na página 29.

SOUTO, M. **Front-end, Back-end e Full Stack**. 2023. Allura. Disponível em: <https://www.alura.com.br/artigos/o-que-e-front-end-e-back-end>. Citado 2 vezes nas páginas 12 e 13.

SOUZA, L. **Ruby - Aprenda a programar na linguagem mais divertida**. E-book. [S.l.]: Casa do Código, 2013. ISBN 978-8566250244. Citado na página 12.

SQLITE. **About SQLite**. 2023. Disponível em: <https://www.sqlite.org/about.html>. Citado na página 11.

SUTHERLAND, J. **Scrum : a arte de fazer o dobro do trabalho na metade do tempo**. São Paulo: LeYa, 2014. Tradução de Natalie Gerhardt. ISBN 9788544100882. Citado na página 8.

TABLEPLUS. **Documentação do Table Plus**. 2023. Disponível em: <https://docs.tableplus.com/>. Citado na página 11.

TAILWINDLABSINC. **Using PostCSS as your preprocessor**. 2023. Disponível em: <https://tailwindcss.com/docs/using-with-preprocessors#using-post-css-as-your-preprocessor>. Citado na página 14.

TEIXEIRA, F. **Introdução a boas práticas em UX Design**. [S.l.]: Casa do Código, 2014. ISBN 978-8566250480. Citado na página 29.

Anexos

ANEXO A – Cronograma de Desenvolvimento Detalhado

Previsão	Requisito - Descrição
01/01 a 14/01	Definição do Projeto - Escolha do objeto de estudo da monografia.
08/01 a 21/01	Coleta de Dados - Estudo da área aonde a aplicação irá ser desenvolvida.
15/01 a 04/02	Elaboração de Diagramas - Diagramas de Casos de Uso, Diagrama de Classe.
22/01 a 04/02	Definição do Cronograma - Estipular tarefas e prazos para execução do projeto.
05/02 a 18/02	Levantamento de Requisitos - Elaborar lista de Requisitos Funcionais, Não Funcionais e Regras de Negócio.
05/02 a 25/02	Prototipagem de Telas - Desenvolver conceito das principais telas da aplicação.
12/12 a 04/03	Framework de Desenvolvimento - Estudo aprofundado no framework Ruby on Rails.
26/03 a 11/03	Cidades : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
26/03 a 11/03	Cidades : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
05/03 a 18/03	Instituições : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
05/03 a 18/03	Instituições : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
05/03 a 18/03	Cursos : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
05/03 a 18/03	Cursos : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
12/03 a 01/04	Turmas : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
12/03 a 01/04	Turmas : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
26/03 a 08/04	Semestres : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
26/03 a 08/04	Semestres : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
02/04 a 15/04	Representantes : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.

Previsão	**Requisito - Descrição**
02/04 a 15/04	Representantes : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
02/04 a 15/04	Alunos : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
02/04 a 15/04	Alunos : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
09/04 a 22/04	Atendimentos : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
09/04 a 22/04	Atendimentos : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
16/04 a 29/04	Proteção do Sistema : Front-End - Desenvolvimento da interface de criação, listagem, edição e exclusão dos dados, componentes complementares e formatação condicional.
16/04 a 29/04	Proteção do Sistema : Back-End - Implementação do comportamento da aplicação e suas regras de negócio.
16/04 a 29/04	Validação e Testes - Implementação de testes para todos os objetos necessários.
30/04 a 20/05	Revisão da Monografia - Realizar a revisão deste documento mediante orientação.
07/05 a 20/05	Ajustes Finais - Correções e elaboração do material para defesa de banca.
14/05 a 03/06	Entrega do Projeto.

Tabela 2 – Cronograma de Desenvolvimento Detalhado.

Fonte: Autoria Própria.