UNIVERSIDADE EVANGÉLICA DE GOIÁS - UNIEVANGÉLICA
ENGENHARIA DE SOFTWARE

MATHEUS VIEIRA GONÇALVES
JÚNIOR AUGUSTO XAVIER ESBALTAR

Desenvolvimento de um sistema de software para gestão de serviços e produtos de um pet shop

Anápolis
Dezembro, 2021

UNIVERSIDADE EVANGÉLICA DE GOIÁS - UNIEVANGÉLICA
ENGENHARIA DE SOFTWARE

MATHEUS VIEIRA GONÇALVES
JÚNIOR AUGUSTO XAVIER ESBALTAR

Desenvolvimento de um sistema de software para gestão de serviços e produtos de um pet shop

Trabalho apresentado ao Curso de Engenharia de Software da Universidade Evangélica de Goiás – UniEVANGÉLICA, da cidade de Anápolis-GO como requisito parcial para obtenção do Grau de Bacharel em Engenharia de Software.

Orientador (a): Prof. Me. Millys Fabrielle Araujo Carvalhaes.

Anápolis
Dezembro, 2021

UNIVERSIDADE EVANGÉLICA DE GOIÁS - UNIEVANGÉLICA

ENGENHARIA DE COMPUTAÇÃO/ENGENHARIA DE SOFTWARE

MATHEUS VIEIRA GONÇALVES

JÚNIOR AUGUSTO XAVIER ESBALTAR

Desenvolvimento de um sistema de software para gestão de serviços e produtos de um pet shop

Monografia apresentada para Trabalho de Conclusão de Curso de Engenharia de software da Universidade Evangélica de Goiás - UniEVANGÉLICA, da cidade de Anápolis-GO como requisito parcial para obtenção do grau de Engenheiro(a) de software.

Aprovado por:

___

Prof. Me. Millys Fabrielle Araujo Carvalhaes

___

Prof. Me. William Pereira dos Santos Junior

___

Eduarda Mosquera

Anápolis, 08 de dezembro de 2021

FICHA CATALOGRÁFICA

GONÇALVES, Matheus Vieira; ESBALTAR, Júnior Augusto Xavier. **Desenvolvimento de um sistema de software para gestão de serviços e produtos de um pet shop.** [Anápolis] 2021. (Universidade Evangélica de Goiás – UniEVANGÉLICA, Engenheiro(a) ______________, 2021).

Monografia. Universidade Evangélica de Goiás, Curso de Engenharia de Software, da cidade de Anápolis-GO.

1. *Software*; Desenvolvimento; Aplicativo *mobile*; Sistema Web; Pet Shop.

REFERÊNCIA BIBLIOGRÁFICA

GONÇALVES, Matheus Vieira; ESBALTAR, Júnior Augusto Xavier. **Desenvolvimento de um sistema de software para gestão de serviços e produtos de um pet shop.** Anápolis, 2021. 79 p. Monografia - Curso de Engenharia de Software, Universidade Evangélica de Goiás - UniEVANGÉLICA.

CESSÃO DE DIREITOS

NOMES DOS AUTORES: Matheus Vieira Gonçalves e Júnior Augusto Xavier Esbaltar
TÍTULO DO TRABALHO: Desenvolvimento de um sistema de software para gestão de serviços e produtos de um pet shop.
GRAU/ANO: Graduação / 2021

É concedida à Universidade Evangélica de Goiás - UniEVANGÉLICA, permissão para reproduzir cópias deste trabalho, emprestar ou vender tais cópias para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte deste trabalho pode ser reproduzida sem a autorização por escrito do autor.

____
Matheus Vieira Gonçalves

____
Júnior Augusto Xavier Esbaltar

Anápolis, 17 de dezembro de 2021

RESUMO

Devido à constante evolução tecnológica e a busca por melhoria, diversos estabelecimentos comerciais estão aderindo à utilização de tecnologias em suas atividades básicas visando adquirir vantagens no mercado e o aumento de lucros. Este trabalho tem como objetivo descrever as etapas do desenvolvimento de um sistema software que auxilie na gestão de pet shops, proporcionando melhor controle dos produtos e serviços oferecidos pelo estabelecimento, e também aumento na satisfação dos clientes. Para atingir esse objetivo serão desenvolvidas duas aplicações, a primeira será um sistema web responsável pela gestão das informações do pet shop, a segunda será um aplicativo mobile disponibilizado para os clientes do estabelecimento, possibilitando a compra de produtos, solicitação de serviços e acompanhamento dos serviços solicitados de forma remota. Espera-se, com o uso do sistema, uma melhora nos processos de gestão das informações e de comunicação com os clientes.

Palavras-chave: *Software*. Desenvolvimento. Aplicativo *mobile*. Sistema *Web*. *Pet Shop*.

ABSTRACT

Due to constant technological evolution and the search for improvement, several commercial establishments are adhering to the use of technologies in their basic activities in order to acquire market advantages and increase profits. This work aims to describe the stages of development of a software system that helps in the management of pet shops, providing better control of the products and services offered by the establishment, and also an increase in customer satisfaction. To achieve this goal, two applications will be developed, the first will be a web system responsible for managing pet shop information, the second will be a mobile application made available to the establishment's customers, enabling the purchase of products, request for services and monitoring of services requested remotely. The use of the system is expected to improve the information management and communication processes with customers

Key words: Software. Development. Mobile app. Web System. Pet Shop.

LISTA DE ILUSTRAÇÕES

Figura 1 - Fases de teste..19

Figura 2 - Desenvolvimento dirigido a testes..20

Figura 3 - Produtividade X tempo...22

Figura 4 - Exemplo de diagrama de caso de uso..29

Figura 5 - Exemplo de tag HTML..35

Figura 6 - Exemplo de estilização utilizando CSS...36

Figura 7 - Exemplo de diagrama entidade-relacionamento...40

Figura 8 - Exemplo de um modelo lógico..40

Figura 9 - GitFlow…………………...44

Figura 10 - Primeiros wireframes do aplicativo...47

Figura 11 - Primeiros mockups do aplicativo..48

Figura 12 - Projeto no GitHub..50

Figura 13 - Padrões de código com Eslint...51

Figura 14 - Exemplo de padrões do commitlint..52

Figura 15 - Exemplo de erro retornado pelo commitlint..52

Figura 16 - Commitzen: Escolha do tipo do commit...53

Figura 17 - Commitzen: padronização da mensagem de commit..53

Figura 18 - Estrutura de pastas e arquivos na raiz do back-end...54

Figura 19 - Estrutura da pasta “prisma”...54

Figura 20 - Estrutura da pasta “src”...55

Figura 21 - Estrutura da pasta “app”..55

Figura 22 - Estrutura da tabela “users” no arquivo schema.prisma.......................................56

Figura 23 - Estrutura da tabela User no arquivo schema.prisma...57

Figura 24 - Entidade usuário..57

Figura 25 - Teste da funcionalidade responsável por criar novo usuário................................58

Figura 26 - Falha no teste de usuário...59

Figura 27 - Primeira implementação da interface IUsersRepositories...................................60

Figura 28 - Primeira implementação da classe PrismaUsersRepository.................................60

Figura 29 - Primeira implementação da classe UserService..61
Figura 30 - Primeira implementação da classe UserService...62
Figura 31 - Estrutura do arquivo UserFactory...63
Figura 32 - Arquivo de rotas do módulo de usuário...63
Figura 33 - Disponibilizando as rotas de usuário para aplicação..64
Figura 34 - Teste do módulo de usuário aprovado...64
Figura 35 - Estrutura de pastas do front-end...65
Figura 36 - Estrutura de pastas do front-end...66
Figura 37 - Componente de input..67
Figura 38 - Componente de botão..67
Figura 39 - Tela de registro de usuário..68

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface* (Interface de Programação de Aplicativos)
CSS	*Cascading Style Sheets* (Folhas de estilo em Cascata)
CVS	*Concurrent Version System* (Sistema de versão simultânea)
DDL	*Data Definition Language* (Linguagem de definição de dados)
DER	Diagrama Entidade-Relacionamento
DML	*Data Manipulation Language* (Linguagem de manipulação dedados)
DOM	*Document Object Model* (Modelo de Objeto de Documento)
IBGE	Instituto Brasileiro de Geografia e Estatística
IEEE	*Institute of Eletrical and Electronic Engineers* (Instituto de Engenheiros Eletricistas e Eletrônicos)
IDE	*Integrated Development Environment* (Ambiente de Desenvolvimento Integrado)
JS	*JavaScript*
JSON	*JavaScript Object Notation* (Notação de Objeto JavaScript)
HTTP	*HyperText Transfer Protocol* (Protocolo de transferência de Hipertexto)
MVC	*Model-View-Controller* (Modelo-Visão-Controlador)
ORDBMS	Sistema de Gerenciamento de Banco de Dados Relacional de Objetos
RCS	*Revision Control System* (Sistema de Controle de Revisão)
SDK	*Software Development Kit* (Kit de Desenvolvimento de Software)
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
TDD	*Test Driven Development* (Desenvolvimento Guiado Por Testes)
TI	*Information Technology* (Tecnologia da Informação)
TS	*TypeScript*
UI	User Interface (Interface de Usuário)
UML	*Unified Modeling Language* (Linguagem de Modelagem Unificada)
XP	Extreme Programming (Programação Extrema)
US	User Story (História de usuário)

SUMÁRIO

LISTA DE ILUSTRAÇÕES 6

LISTA DE ABREVIATURAS E SIGLAS 8

1 INTRODUÇÃO 11

2 FUNDAMENTAÇÃO TEÓRICA 14

2.1 Software 14

2.2 Engenharia de software 15

2.3 Processos de software 15

2.3.1 Processos dirigidos a planos ou tradicionais 16

2.3.2 Processos ágeis 16

2.3.2.1 Gerenciamento de projeto com Scrum 17

2.4 Qualidade de software 18

2.4.1 Desenvolvimento Orientado a Teste 19

2.4.1.1 Jest 20

2.4.2 Padrões de código 21

2.4.2.1 Código confuso 21

2.4.2.2 Código limpo 22

2.4.2.3 Ferramentas para o controle de padrões de código 23

2.4.2.4 Eslint 23

2.4.3 Gestão de Configuração de Software 23

2.4.3.1 Controle de versões 24

2.4.3.2 Sistemas Locais de Controle de Versão 24

2.4.3.3 Sistemas Centralizados de Controle de Versão 25

2.4.3.4 Sistemas Distribuídos de Controle de Versão 25

2.4.3.5 Git 26

2.4.3.6 Github 27

2.5 Planejamento do sistema 27

2.5.1 Técnicas para levantamento e validação de requisitos de software 27

2.5.1.1 Entrevistas 29

2.5.1.2 Casos de uso 29

2.5.1.3 Brainstorming 30

2.5.1.4 Prototipação 30

2.5.2 Ferramentas para levantamento e validação de requisitos de software 30

2.5.2.1 Figma 31

2.5.2.2 Whimsical 31

2.6 Desenvolvimento do sistema 31

2.6.1 Desenvolvimento web 32

2.6.2 Desenvolvimento mobile 33
2.6.3 React 33
2.6.4 HTML: HiperText Markup Language 35
2.6.5 CSS: Cascading Style Sheets 35
2.6.6 JavaScript 36
2.6.7 TypeScrypt 37
2.6.8 Ambiente de Desenvolvimento Integrado 38
2.6.8.1 Visual Studio Code 38
2.6.9 Node.js 39
2.6.10 Banco de dados 39
2.6.10.1 PostgreSQL 41
2.6.11 Ferramentas de planejamento e gestão 42
2.6.11.1 Trello 42
3. DESENVOLVIMENTO 42
3.1 Processo metodológico 43
3.2 Política de versionamento 44
3.3 Sprints 45
3.3.1 Sprint 01 – Levantamento de requisitos 45
3.3.2 Sprint 02 –Validação de requisitos e prototipação de telas. 46
3.3.3 Sprint 03 – Definição da arquitetura do sistema e modelagem inicial do banco de dados. 48
3.3.4 Sprint 04 - Preparação do ambiente de desenvolvimento 49
3.3.5 Sprint 05 - Estrutura de pastas e arquivos do Back-End 53
3.3.6 Sprint 06 - Implementação do primeiro requisito (US006) 56
3.3.6.1 Primeira implementação Back-End (US006) 56
3.3.6.2 Configuração da estrutura de pastas e arquivos do Front-End (mobile) 65
3.3.6.3 Primeira implementação Front-end (US006) 66
3.3.7 Sprint 07 – Desenvolvimento US005 e US010. 68
3.3.8 Sprint 08 – Desenvolvimento US002, US008 e US011. 69
3.3.9 Sprint 09 – Desenvolvimento US003, US012 e US013. 69
4. RESULTADOS 70
5. CONCLUSÃO E CONSIDERAÇÕES FINAIS 71
REFERÊNCIAS 72

1 INTRODUÇÃO

Na década de 2010 o crescente número de animais de estimação vem proporcionando uma expansão do segmento de mercado voltado para esses animais com fornecimento de produtos e serviços. Pesquisas realizadas pelo Instituto Brasileiro de Geografia e Estatística (IBGE) apontam que em 2013 a população de animais de estimação (comumente chamados de *pet*) no Brasil era cerca de 132,4 milhões de animais (IBGE, 2013), estima-se que em 2018 houve um aumento nos números atingindo 139,3 milhões (INSTITUTO PET BRASIL, 2018). Algumas pesquisas realizadas em 2019, mostram que esse número atingiu aproximadamente 141,6 milhões de pets só no país (INSTITUTO PET BRASIL, 2019).

Dados coletados pelo Instituto Pet Brasil mostram que o varejo *pet* nacional movimentou R$ 34,4 bilhões em 2018, apresentando uma alta de 4,6% em comparação a 2017, quando o faturamento final foi de R$32,9 bilhões (INSTITUTO PET BRASIL, 2018).

Mesmo com a presença da pandemia em 2020, notícias relatadas pelo jornal Folha de São Paulo mostram um crescimento no faturamento da indústria *pet* de 13,5% atingindo cerca de R$40,1 bilhões no ano de 2020 (FOLHA DE SÃO PAULO, 2020).

Graças a esse crescimento acelerado, diversos novos modelos de loja e canais de distribuição estão sendo criados com o tempo. Novos perfis de consumidores mais exigentes e adeptos a mudanças possibilitam a adesão de novas estratégias e ferramentas de *marketing*, gerando novas formas de relacionamento na comercialização de produtos e serviços (LIMA, B. R. et al, 2013).

Diante desse cenário, a gestão de informações para o correto uso dos recursos financeiros da empresa torna-se uma tarefa indispensável, exigindo grande esforço quando realizada de forma manual. Graças a isso, várias empresas têm adotado a utilização da tecnologia da informação, em destaque os *softwares*, utilizando-os para obter uma melhoria na gestão de seus recursos (PRATES; OSPINA, 2004). Os Pet shops não são uma exceção nesse cenário, devido a necessidade de lidar com diversas informações, ligadas a produtos, serviços, clientes e animais.

Devido a problemática apresentada, este trabalho tem por objetivo o desenvolvimento da documentação e de um sistema de *software* que auxilie a gestão dos produtos e serviços oferecidos por um *pet shop* e possibilite uma melhor comunicação com seus clientes.

Para o desenvolvimento do sistema, foi realizado um levantamento de informações utilizando pesquisas sobre o mercado pet na década de 2010 em conjunto com entrevistas utilizando o *pet shop* Riguas Pet como cliente modelo. Além disso, durante todo o processo de desenvolvimento foram utilizadas diferentes ferramentas, artefatos e tecnologias o quais serão abordados posteriormente no trabalho, no capítulo 2.

Diversos fatores favorecem o desenvolvimento de *software* tornando-o tema de alta relevância, pois, com o passar dos anos as empresas têm buscado estratégias que lhe proporcionem destaque no mercado, visando atrair clientes e aumentar os lucros (SANTIAGO; CARDOSO, [entre 2015 e 2021]). Assim, a busca por inovações tornou-se uma necessidade para a permanência das empresas em um mercado competitivo que está em constante transformação (LIMA, B. R. et al, 2013).

Prates e Ospina afirmam que "na maioria das empresas, a adoção da TI surge em função de uma necessidade derivada dos objetivos organizacionais preestabelecidos" (2004, p. 20), e em suas pesquisas para buscar os motivos para implantação de TI em pequenas empresas, apresentam que nas empresas entrevistadas 24% tem como objetivo a melhoria de controle organizacionais, 22% no aumento da participação de mercado e 10% a redução de custos, apontando uma visão estratégica pelo lado das pequenas empresas para utilização de TI em seus negócios para buscar maior número de clientes e uma boa gestão.

Dado o contexto atual de evolução, a tecnologia entra como uma inovação para diversos setores comerciais. Estabelecimentos voltados para venda e prestação de serviço, tais como *pet shops*, a utilizam como uma estratégia para melhorar os processos de gestão e atendimento ao cliente (BAYLÃO; OLIVEIRA, 2012). Dentre as tecnologias utilizadas destaca-se os programas computacionais denominados *softwares*.

Softwares, apesar de apresentarem diversos benefícios quando aplicados nas empresas, também apresentam um fator limitante: seu alto custo. Realizando uma breve pesquisa envolvendo três principais sistemas de *softwares* disponíveis para o mercado de *pet shop*, sendo *SimplesVet*, *Vetus* e *Vetwork*, foi identificado que todos trabalham com um sistema onde o cliente paga mensalmente para ter acesso a um pacote de recursos.

Dentre os três sistemas analisados durante a realização da pesquisa, o pacote de menor valor foi oferecido pelo *Vetwork*, no valor de R$159,90. O pacote de maior custo foi oferecido pelo *SimplesVet*, no valor de R$719,00. Levando em consideração que uma das principais

dificuldades das pequenas empresas está relacionado à escassez de recursos financeiros (SANTOS, P. V. S. et al, 2018), a adesão de tais *softwares* por longos períodos de tempo torna-se inviável.

Um dos fatores que geram esse alto custo é a grande quantidade de recursos que estes sistemas disponibilizam, mesmo nos pacotes mais básicos. Parte dos recursos oferecidos acabam não sendo totalmente aproveitados por pequenas empresas, sendo até dispensáveis.

Assim, a criação de um *software* que apresente um valor reduzido em sua contratação tende a se tornar um forte diferencial entre os demais, mesmo que ele disponibilize os recursos mais básicos para o estabelecimento.

Para que seja possível então a criação do *software* proposto, o capítulo 2 apresentará os principais conceitos, temas e ferramentas necessárias para este projeto. Posteriormente, o capítulo 3 abordará as etapas necessárias para o desenvolvimento do sistema, evidenciando os padrões, técnicas e metodologias utilizadas. Por fim, os capítulos 4 e 5 descrevem os principais resultados, aprendizados e considerações relativas ao trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo visa realizar uma introdução acerca dos conteúdos que serão abordados durante o desenvolvimento do trabalho, através de uma breve revisão bibliográfica que apresentará as principais ferramentas, processos e tecnologias que serão utilizadas no desenvolvimento da solução proposta.

2.1 Software

Segundo Sommerville, *softwares* são definidos como programas de computadores os quais podem ser desenvolvidos de forma amadora ou profissional. *Software* amadores no geral são utilizados para uso próprio do desenvolvedor na resolução de suas tarefas pessoais. *Softwares* desenvolvidos de forma profissional possuem um objetivo de negócio e são, em sua maioria, utilizados por mais pessoas além do próprio desenvolvedor (SOMMERVILLE, 2011).

Ao trabalhar com *softwares* de forma profissional será desenvolvido um produto destinado para um cliente específico ou para o mercado em geral. Na construção de *softwares* generalistas, fica a cargo da empresa desenvolvedora controlar suas especificações, ou seja, definir o que o *software* deverá fazer. *Softwares* feitos sob demanda possuem suas especificações controladas pelo cliente que o está adquirindo (SOMMERVILLE, 2011).

Atualmente, “o *software* distribui o produto mais importante de nossa era – a informação” (PRESSMAN; MAXIM, 2016, p. 03). Através do gerenciamento da informação, dados comerciais que se perdiam com o tempo e não apresentavam grande relevância tornam-se ferramentas úteis para impulsionar a competitividade da empresa no mercado, entretanto, a facilidade no acesso de tais dados pode gerar brechas para que pessoas mal intencionadas cometam crimes (PRESSMAN; MAXIM, 2016).

Assim, para que os *softwares* produzidos possam agregar valor ao cliente de forma segura e com qualidade, torna-se necessário a utilização de ferramentas e processos os quais são fornecidos pela engenharia de *software*.

2.2 Engenharia de software

O termo engenharia de *software* pode ser definido como um conjunto de métodos, padrões, técnicas e ferramentas que quando aplicados em conjunto visam produzir de forma rápida e eficaz *softwares* de alta qualidade e confiabilidade (PONTES; ARTHAUD, 2019).

Ao mencionar a palavra *software,* no senso comum as pessoas pensam somente nos programas computacionais, entretanto ao trabalhar com a engenharia de *software* não se limita apenas ao próprio programa em si, fatores como a documentação associada ao programa e a configuração dos recursos necessários para seu correto funcionamento são etapas indispensáveis (SOMMERVILLE, 2011).

Todas as etapas e atividades necessárias para o desenvolvimento de um *software* são definidas dentro do processo adotado para o projeto.

2.3 Processos de software

Os processos de *software*, assim como é definido por Pressman e Maxim (2016), são um conjunto de atividades, ações e tarefas que podem ser adaptados para uma realidade, sendo aplicadas durante todo o ciclo de desenvolvimento de um *software*, desde a etapa de especificação de requisitos até a fase de implementação e manutenção.

Segundo o Sommerville (2011), existem diferentes processos de *software*, entretanto algumas etapas que são comuns a todos eles, sendo elas:

- **Especificação de *software***: Definição das funcionalidades e restrições do sistema.
- **Projeto e implementação de *software*:** Codificação das funcionalidades identificadas seguindo suas devidas restrições.
- **Validação de software:** Validação do *software* produzido a fim de garantir que atenda às necessidades dos usuários.
- **Evolução de software:** Manutenções evolutivas e corretivas, visando atender as mudanças solicitadas pelo cliente.

De forma geral essas atividades estão presentes em todos os processos de *software,* apresentando apenas pequenas alterações dependendo da empresa que as aplica, isso porque

"não existe um processo ideal, a maioria das organizações desenvolve os próprios processos de desenvolvimento de software" (SOMMERVILLE, 2011, p. 19). Graças a isso os processos de *software* têm evoluído com o tempo, sempre visando tirar o melhor proveito dos recursos disponíveis pela empresa (SOMMERVILLE, 2011).

Os processos desenvolvidos pelas empresas podem ser categorizados de duas formas, processos dirigidos a planos, também conhecidos como processos tradicionais e processos ágeis (SOMMERVILLE, 2011).

2.3.1 Processos dirigidos a planos ou tradicionais

Os processos dirigidos a planos são caracterizados pelo planejamento com antecedência, ou seja, todas as atividades realizadas e artefatos produzidos são previamente definidos e todo o progresso do projeto é avaliado através da comparação com o planejamento inicial (SOMMERVILLE, 2011).

Com sua utilização todas as atividades e tarefas necessárias para o desenvolvimento do *software* são realizadas sequencialmente, sendo necessário um alto nível de detalhamento do sistema e uma baixa probabilidade de mudanças, entretanto, em um cenário de constante transformação envolvendo o mercado de *software*, tais processos apresentam dificuldades em atingir os objetivos propostos (PRESSMAN; MAXIM, 2016).

2.3.2 Processos ágeis

Processos ágeis surgem como uma alternativa aos modelos tradicionais. Seu foco está na satisfação do cliente, para isso são propostas entregas incrementais constantes para que o cliente possa ver, testar e avaliar o que está sendo feito (ALMEIDA *et al*., 2018). Além disso, processos ágeis são caracterizados por serem flexíveis quanto a mudanças, capazes de reduzir os impactos de tempo e custo, sendo assim uma ótima opção para o mercado de *software* contemporâneo (PRESSMAN; MAXIM, 2016).

Segundo Sommerville (2011), o desenvolvimento ágil teve seu início no final da década de 90 quando as empresas notaram a necessidade de adotar medidas de desenvolvimento rápido, sendo capaz de lidar com mudanças durante o andamento do projeto.

Tal necessidade levou ao surgimento das primeiras metodologias ágeis de desenvolvimento, tais como *Scrum* e *Extreme Programming* (XP).

A utilização de metodologias ágeis é recomendada principalmente para projetos desenvolvidos por equipes pequenas e que não possuam seu escopo totalmente definido, estando sujeito a mudanças. Dentre os benefícios gerados pela sua aplicação, pode-se destacar a melhora na comunicação, maneiras de lidar com mudanças, rápido desenvolvimento e entregas contínuas para o cliente (SOARES, 2004).

Metodologias ágeis seguem os princípios defendidos pelo manifesto ágil, publicado no ano de 2001. Esse manifesto, segundo Soares (2004, p. 03), "[...] não rejeita os processos e ferramentas, a documentação, a negociação de contratos ou o planejamento [...]", seu objetivo é mostrar que essas etapas do processo possuem uma menor importância se comparadas com "[...] indivíduos e interações, com o software estar executável, com a colaboração do cliente e as respostas rápidas a mudanças e alterações" (2004, p. 03).

Tendo como base os benefícios gerados pela utilização de metodologias ágeis no desenvolvimento de *software*, este trabalho também adotará a utilização de uma metodologia ágil, o *Scrum*, visando lidar com o processo de gerenciamento do projeto.

2.3.2.1 Gerenciamento de projeto com Scrum

Criado na década de 90, *Scrum* é um *framework* para o desenvolvimento ágil de *softwares*. Seu objetivo é auxiliar pessoas, equipes e organizações a gerarem valores através de soluções para problemas complexos. Sua estrutura simples permite a integração de vários processos, técnicas e métodos considerados necessários para atingir os objetivos do projeto (SUTHERLAND; SCHWABER, 2020).

Segundo Sutherland e Schwaber (2020), "Scrum é baseado no empirismo e pensamento enxuto". O empirismo define que o conhecimento é proporcionado pela experiência, já o pensamento enxuto visa atingir os objetivos com o menor esforço possível. Dessa forma o *Scrum* visa a resolução de problemas da melhor forma possível através do menor custo, utilizando das experiências passadas pela equipe a fim de melhorar o processo.

O *Scrum* é construído através de três pilares: a transparência, inspeção e adaptação. A transparência permite que todos os envolvidos tenham o mesmo entendimento sobre o estado atual do projeto e seus objetivos. A inspeção, por sua vez, busca por problemas ou variações

indesejáveis. Por fim, a adaptação permite ajustes no projeto para minimizar impactos ocasionados por situações inesperadas (SUTHERLAND; SCHWABER, 2020).

Segundo Sutherland e Schwaber (2020), o Scrum trabalha com equipes pequenas e multifuncionais, não havendo subequipes ou hierarquias. Existem três papéis desempenhados dentro da equipe, sendo eles: Product Owner, Scrum Master e Time de Desenvolvimento.

De forma objetiva o Product Owner é responsável por definir e priorizar as funcionalidades do produto. O Time de Desenvolvimento é composto por um ou mais profissionais responsáveis pelo desenvolvimento das funcionalidades e o Scrum Master é responsável por manter a utilização das boas práticas pregadas pelo Scrum por todos os membros (SUTHERLAND; SCHWABER, 2020).

De acordo com Sutherland e Schwaber (2020), no *Scrum* o projeto é desenvolvido em ciclos, denominados de *Sprint*. Cada ciclo apresenta um período de tempo fixo inferior a um mês. Nesse tempo uma lista de atividades denominada *Sprint Backlog* é desenvolvida. O *Sprint Backlog*, é apenas uma parte da lista de funcionalidades do projeto, conhecida como *Backlog* do produto.

Cada início ou término de Sprint é marcado por reuniões. A "Sprint Planning", marca o início de uma nova Sprint, nela o *Product Owner* define as atividades e artefatos a serem produzidos durante a *Sprint*. A "Daily Scrum" é uma reunião diária com duração de 15 minutos com foco em manter todos os envolvidos cientes do estado atual do projeto. A "Sprint Review" marca o término da *Sprint*, nela o Time Scrum realiza uma avaliação junto com o *Product Owner* sobre os artefatos produzidos. Por fim na "Sprint Retrospective", o time avalia os erros e acertos cometidos durante a *Sprint,* buscando melhorias no processo para a próxima iteração (SUTHERLAND; SCHWABER, 2020).

Apesar de eficaz, a adoção de metodologias ágeis como o *Scrum* é apenas um dos fatores que podem proporcionar qualidade ao produto final obtido. Outras estratégias podem ser adotadas no projeto visando melhorar a qualidade do *software* produzido.

2.4 Qualidade de software

O conceito de qualidade de *software* é definido pelo Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE, 1990) como um conjunto de ações necessárias para definir um nível de confiança do produto em relação à sua conformidade com seus requisitos. Assim,

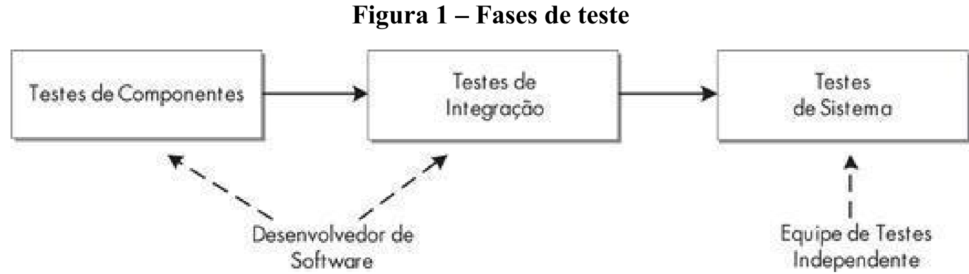
ações que visam garantir a qualidade do projeto demandam uma atitude preventiva em relação aos problemas que podem surgir, sendo necessária a busca por mecanismos capazes de reduzir os defeitos gerados por tais problemas (PÁDUA, 2019).

Os testes de *software* se tornaram indispensáveis na detecção de erros. Segundo Himara (2011, p. 93), as atividades de teste têm como objetivo "verificar os aspectos estruturais e lógicos do software, bem como os seus aspectos sistêmicos, com o intuito de descobrir defeitos no software", buscando comportamentos inesperados do sistema os quais não condizem com as suas especificações.

Ainda segundo Himara (2011), o processo de teste pode ser executado em fases para ser mais eficaz, sendo elas:

- **Testes de componentes (ou de unidades):** é realizada pelo próprio desenvolvedor com objetivo de testar individualmente os módulos ou componentes desenvolvidos.
- **Testes de integração:** também realizada pelo próprio desenvolvedor onde testa a integração dos componentes do sistema.
- **Testes de sistema:** com objetivo de testar o *software* do ponto de vista sistêmico e deve ser realizada por uma equipe independente de testes.

Figura 1 – Fases de teste



Fonte: HIMARA, 2011

2.4.1 Desenvolvimento Orientado a Teste

O *Test Driven Development* (TDD) é uma das práticas sugeridas pelo *Extreme Programing*, onde o desenvolvedor escreve o teste antes da funcionalidade se baseando em um pequeno ciclo de atividades (ANICHE, 2012).

As etapas dos processos fundamentais do TDD, segundo Sommerville (2011) são:

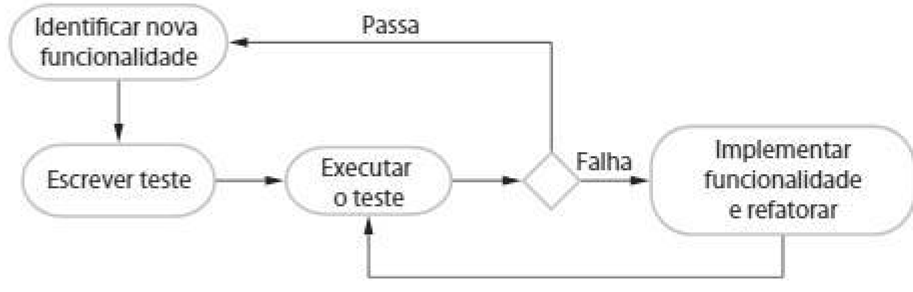
- Identificar o incremento da funcionalidade necessária, sendo um pequeno código.

- Escrever um teste automatizado para a funcionalidade.
- Executar o teste, junto com outros testes já desenvolvidos, onde a funcionalidade falhará, pois ainda não foi implementada, isto é proposital assim mostra que o teste acrescenta algo ao conjunto de testes.
- Implementar a funcionalidade e executar novamente o teste, podendo envolver a refatoração ou adição de mais código.
- Após todos testes serem executados com sucesso, caminhar para próxima parte da funcionalidade.

Com a devida implementação do TDD, diversos benefícios são alcançados. Sua aplicação permite ao desenvolvedor um maior entendimento sobre o problema trabalhado, além de possibilitar a identificação de erros e bugs gerados por uma nova implementação ou por mudanças no sistema, facilitando assim a sua resolução.

A figura 2 apresentada a seguir demonstra o processo de implementação de novas funcionalidades ou mudanças em um sistema utilizando o TDD:

Figura 2 – Desenvolvimento dirigido a testes



Fonte: SOMMERVILLE, 2011 (p. 155)

2.4.1.1 Jest

Existem tarefas de atividade de teste que são executadas de forma mais eficiente pelo computador do que pelo ser humano. A automatização destas atividades se baseia em definir quais entradas devem ser executadas e qual resultado esperado pelo programa. Caso os resultados produzidos sejam os resultados esperados diz-se que o caso de teste “passou”, caso contrário o caso de teste “falhou” (MALDONADO, 2018).

O *Jest* é um *framework* de teste robusto, desenvolvido em *Javascript* para realização de testes automatizados. Com o foco na simplicidade, graças a uma boa documentação, requer poucas configurações e pode ser estendido de acordo com a funcionalidade, tendo suporte também a outras linguagens como *Typescript*. O *Jest* garante que os testes tenham um estado único global e executa-os em paralelo de forma rápida e eficiente (FACEBOOK OPEN SOURCE, 2021a).

2.4.2 Padrões de código

Ainda relacionado a qualidade de *software*, outra estratégia para manter a boa qualidade no produto desenvolvido seria através dos padrões de código. Isso pois, segundo Sommerville (2011), dentre as atividades presentes no desenvolvimento de *softwares* a codificação do sistema ganha destaque, sendo caracterizada com um dos estágios mais críticos.

Com o objetivo de melhorar o processo de codificação, padrões de código e arquiteturas são implementados no projeto, buscando ao máximo manter a alta produtividade da equipe ao longo do tempo. Autores como Martin (2009), descrevem os problemas ocasionados pela falta de padrões e pela presença de códigos confusos e mal estruturados em um projeto. Além disso, também são abordados conceitos que diferem um código considerado bom, de um código ruim.

2.4.2.1 Código confuso

Códigos confusos ou mal estruturados geralmente estão presentes em projetos desenvolvidos, principalmente por equipes inexperientes que não seguem padrões bem definidos. De maneira geral, seu impacto é proporcional ao tempo. Dessa forma, no início do projeto a produtividade da equipe não é afetada, mas à medida que o projeto evolui, cada alteração feita no código pode proporcionar falhas ou comportamentos inesperados (MARTIN, 2009).

Assim, “conforme a confusão aumenta, a produtividade da equipe diminui, assintoticamente aproximando-se de zero” (MARTIN, 2009, p. 04). A figura 3 mostrada

abaixo apresenta um gráfico que demonstra a perda de produtividade ao longo do tempo, ocasionada pela falta de padrões durante a escrita de código.

Figura 3 – Produtividade X tempo

Fonte: MARTIN, 2009 (p. 04)

A fim de evitar a perda de produtividade, diversos padrões foram desenvolvidos por profissionais experientes da área, os quais após aplicados corretamente são capazes de gerar “códigos limpos”.

2.4.2.2 Código limpo

Robert C. Martin em seu livro “Código Limpo”, apresenta a definição de código limpo através do ponto de vista de diversos profissionais da área que possuem vasta experiência no assunto. Para esses profissionais, um código só pode ser classificado como “limpo” se sua legibilidade estiver totalmente clara, permitindo que qualquer profissional da área seja capaz de entendê-lo (MARTIN, 2009).

Além disso, códigos limpos são caracterizados por serem simples e diretos, apresentando variáveis de fácil interpretação e funções que realizam apenas uma única tarefa de forma clara e objetiva. Neles a realização de modificações se torna uma tarefa simples a qual pode ser realizada por qualquer membro da equipe. Com exceção de poucos casos, a utilização de comentários é desnecessária graças à boa legibilidade presente no código (MARTIN, 2009).

2.4.2.3 Ferramentas para o controle de padrões de código

Visando facilitar o controle sobre os padrões de código presentes no projeto, diversas ferramentas foram desenvolvidas ao longo do tempo. Com a sua utilização torna-se possível estabelecer e monitorar padrões de escrita de código a serem seguidos por todos os membros da equipe (OPENJS FOUNDATION, 2021a).

Essas ferramentas são comumente denominadas ferramentas de *lint*, as quais são responsáveis por auxiliar os desenvolvedores a identificarem códigos que estão fora dos padrões estabelecidos para o projeto (OPENJS FOUNDATION, 2021a). Entre as diversas ferramentas disponíveis, podemos destacar o *Eslint* o qual será abordado neste projeto.

2.4.2.4 Eslint

Segundo seus criadores, o *Eslint* é definido como uma ferramenta de *lint* plugável para *JavaScript* e JSX. Sua função é analisar estaticamente o código fonte do projeto na busca por erros e quebras de padrões que foram definidos previamente pela equipe (OPENJS FOUNDATION, 2021a).

A grande maioria das ferramentas de codificação disponíveis no mercado, tais como ambientes de desenvolvimento integrado, são capazes de utilizar os recursos disponibilizados pelo Eslint. Isso possibilita a identificação e correção automática de erros, o que aumenta, a produtividade dos desenvolvedores (OPENJS FOUNDATION, 2021a).

Como o *Typescript* é um superconjunto do *JavaScript*, sua integração com *Eslint* também é possível, adicionando mais uma camada responsável pela identificação e correção de erros durante o desenvolvimento do projeto, visando aumentar a qualidade do produto final obtido (OPENJS FOUNDATION, 2021a).

2.4.3 Gestão de Configuração de Software

Em um ambiente de desenvolvimento contínuo, no qual a tecnologia se encontra, mudanças tornam-se um fator inevitável durante o desenvolvimento de *software.* Quando não

gerenciadas de forma apropriada, podem prejudicar os membros da equipe, através de *bugs* ou ações inesperadas que reduzem a produtividade dos integrantes (PRESSMAN; MAXIM, 2016).

Assim, manter um sistema de histórico de modificações no projeto, principalmente no código, torna-se uma tarefa tão importante quanto desenvolvê-lo de forma organizada e estruturada. Para isso, surge a gestão de configuração.

A gestão de configuração de *software* é um conjunto de atividades responsáveis pelo gerenciamento das alterações realizadas em um sistema computacional durante todo o seu ciclo de vida (PRESSMAN; MAXIM, 2016). Para a sua implementação foram desenvolvidas algumas ferramentas, tais como os controles de versão.

2.4.3.1 Controle de versões

De acordo com Chacon e Straub (2014, p. 10), "controle de versão é um sistema que registra alterações em um arquivo ou conjunto de arquivos ao longo do tempo". Essa prática permite que seus autores possam recuperar versões específicas, caso necessário, independente do intervalo de tempo entre a versão atual e a desejada.

Com o sistema de controle de versões é possível reverter arquivos selecionados para um estado anterior, ou até mesmo reverter todo o projeto de uma única vez. Além disso é possível realizar uma comparação de alterações, identificando os responsáveis por determinadas mudanças, os locais modificados e as datas de modificação, facilitando a identificação de erros e problemas que surgirem no decorrer do projeto (CHACON; STRAUB, 2014).

Segundo Chacon e Straub, existem três principais tipos de controle de versões: sistemas locais, sistemas centralizados e sistemas distribuídos.

2.4.3.2 Sistemas Locais de Controle de Versão

Sistemas locais é uma das primeiras formas de controle de versão, realizada na própria máquina através da cópia dos arquivos que compõem o projeto, armazenando-os em diretórios específicos. Apesar de ser a forma mais simples é a mais sujeita a erros, pois o controle de

versões ficava restrito a máquina do desenvolvedor e seu processo é realizado de forma totalmente manual. (CHACON; STRAUB, 2014).

Apesar da implementação de ferramentas como o *Revision Control System* (RCS) solucionarem o problema de realização manual da cópia dos arquivos, o problema de privatização deles ainda permanece, desfavorecendo o trabalho em equipe (CHACON; STRAUB, 2014).

2.4.3.3 Sistemas Centralizados de Controle de Versão

Diversos projetos voltados para o desenvolvimento de sistemas computacionais envolvem mais de um colaborador, com isso surge a necessidade de compartilhar as versões dos arquivos entre os envolvidos. A fim de suprir essa necessidade foram desenvolvidos sistemas como *Concurrent Version System* (CVS), *Subversion* e *Perforce*, os quais apresentam um servidor que centraliza o armazenamento dos arquivos do controle de versão, disponibilizando-os para todos os colaboradores com acesso ao servidor (CHACON; STRAUB, 2014).

Apesar das diversas vantagens que esse modelo de controle apresenta, sua utilização não é a mais segura. Caso ocorra algum problema com o servidor, todos os colaboradores ficam impedidos de salvar ou buscar por mudanças. Além disso, caso o dispositivo de armazenamento do servidor sofra alguma falha e não haja um backup dos dados, todo o projeto pode ser perdido (CHACON; STRAUB, 2014).

2.4.3.4 Sistemas Distribuídos de Controle de Versão

Sistemas distribuídos, por sua vez, visam sanar os pontos falhos dos modelos anteriores. Esse modelo mantém a ideia de clientes e servidores, adicionando um ponto de segurança a mais, o qual permite aos usuários duplicarem os projetos em suas máquinas locais. Desse modo, as alterações feitas pelos colaboradores são salvas localmente e sincronizadas ao servidor (CHACON; STRAUB, 2014).

A possibilidade de clonar o repositório contendo todos os arquivos do servidor para a máquina local do cliente permite que ele tenha acesso ao histórico de versões do projeto mesmo que o servidor seja desligado da rede durante algum tempo. Após as devidas

mudanças e a retomada do servidor à rede, basta sincronizar as modificações feitas no repositório local com o servidor (CHACON; STRAUB, 2014).

Tal estratégia permite que cada colaborador do projeto também tenha um backup dos arquivos em sua máquina local. Assim, caso ocorra algum problema com o dispositivo de armazenamento contido no servidor e o projeto seja perdido, basta enviá-lo novamente através da máquina de algum colaborador (CHACON; STRAUB, 2014).

Atualmente existem diversas ferramentas que realizam o controle de versão de forma distribuídas, como por exemplo o Git, Mercurial, Bazaar ou Darcs. Várias dessas ferramentas são integradas a repositórios remotos, permitindo que desenvolvedores colaborem em diferentes projetos independente de sua localização (CHACON; STRAUB, 2014).

2.4.3.5 Git

Git é um sistema de controle de versão distribuído, gratuito e de código aberto, que tem por objetivo realizar o controle de versão de qualquer tipo de arquivo. Sua utilização se destaca principalmente no desenvolvimento de *softwares*, sendo capaz de lidar de forma rápida e eficiente com diferentes escalas de projetos, desde os mais simples até os mais complexos (GIT, 2021).

Além da realização do controle de versões, o Git permite que vários membros de uma equipe contribuam de forma simultânea no mesmo projeto, sem que haja os riscos de um membro da equipe sobrepor as alterações de outro membro acidentalmente (GIT, 2021).

A ferramenta também disponibiliza um sistema de ramificações possibilitando a divisão do projeto em diversas linhas temporais, ou seja, caso o projeto já apresente uma série de arquivos e surja uma nova mudança a ser implementada, pode ser criada uma nova ramificação do projeto que será basicamente uma cópia de todos os arquivos e pastas já existentes. A partir dessa cópia, serão realizadas as mudanças necessárias e após a sua conclusão e aprovação essa ramificação será unida novamente ao projeto principal, adicionando ou modificando apenas os arquivos necessários, sem afetar os demais (GIT, 2021).

A estratégia de ramificação oferecida pelo Git permite que novas implementações e modificações em um sistema de *software* possam ser aplicadas e testadas de formas seguras,

em um ambiente controlado antes de serem enviadas para o cliente, evitando falhas e erros que poderiam surgir de forma inesperada, afetando negativamente o projeto (GIT, 2021).

2.4.3.6 Github

Gihub é um serviço baseado em nuvem que hospeda o sistema de controle de versão Git citado anteriormente. Seu objetivo é a criação de uma rede social para desenvolvedores, possibilitando que diversos profissionais espalhados pelo mundo possam acessar e contribuir paralelamente no projeto (GITHUB, 2021).

O *Github* permite aos desenvolvedores uma forma de interação com a comunidade de desenvolvimento de *software*, através da divulgação de projetos de código aberto os quais diferentes profissionais podem contribuir, tanto na correção de problemas como na implementação de melhorias (GITHUB, 2021).

2.5 Planejamento do sistema

Após a definição das principais práticas e ferramentas utilizadas na gestão e preservação da qualidade, é dado início ao planejamento do sistema através da obtenção das principais informações, definindo assim as atividades a serem realizadas e artefatos a serem produzidos. Para isso serão aplicadas técnicas de levantamento e validação das informações pertinentes ao projeto.

2.5.1 Técnicas para levantamento e validação de requisitos de software

Antes de iniciar o desenvolvimento do sistema através da codificação, torna-se necessário o detalhamento e o planejamento do que será feito. Projetos que são iniciados sem o entendimento claro do produto final esperado aumentam significativamente suas chances de fracasso (PRESSMAN; MAXIM, 2016).

A Engenharia de Requisitos surge então, como área responsável por obter e especificar as informações necessárias para o desenvolvimento do *software*. Suas atividades podem ser

divididas em três principais fases: a concepção, levantamento e elaboração (PRESSMAN; MAXIM, 2016).

A fase de concepção define a natureza do problema a ser solucionado. A fase de levantamento define quais ações são necessárias para a resolução do problema identificado. Por fim, a fase de elaboração modifica, modela e documenta as ações definidas anteriormente. Todas as informações pertinentes ao projeto, geradas por essas fases, são denominadas de requisitos (PRESSMAN; MAXIM, 2016).

Segundo Sommerville (2011), os requisitos de um *software* são as descrições das ações que o sistema deve oferecer, e junto delas as restrições para o seu correto funcionamento. Ainda segundo Sommerville (2011), os requisitos podem ser classificados de duas principais formas: requisitos funcionais e não funcionais.

Requisitos funcionais “são declarações de serviços que o sistema deve fornecer” (SOMMERVILLE, 2011, p. 59). Sua utilização possibilita descrever quais ações o usuário poderá ou não realizar dentro do sistema e quais resultados serão gerados através dessas ações (SOMMERVILLE, 2011).

Requisitos não funcionais “são restrições aos serviços ou funções oferecidos pelo sistema” (SOMMERVILLE, 2011, p. 59). Sua utilização possibilita definir regras relacionadas às qualidades do sistema, podendo envolver tempo de resposta, segurança de dados e outros fatores que podem englobar o sistema como um todo (SOMMERVILLE, 2011).

Visto que os requisitos serão responsáveis pela base do resultado obtido, sua identificação e documentação torna-se uma tarefa fundamental para o sucesso do projeto. Afinal, não adianta projetar e construir um programa de computador elegante, se o problema que ele resolve não atende a necessidade de seus usuários (PRESSMAN; MAXIM, 2016).

Autores como Sommerville (2011) e Pressman (2016) abordam diversas técnicas que podem ser utilizadas para o levantamento de requisitos, tais como entrevistas, casos de uso, *brainstorming*, prototipação entre outras. A seguir será realizada uma breve apresentação das principais técnicas para levantamento de requisitos que será abordado neste projeto.

2.5.1.1 Entrevistas

Entrevistas formais ou informais com os *stakeholders* (partes interessadas ao projeto, como clientes, usuários, investidores, entre outros) é uma das abordagens tradicionais para o levantamento de requisitos. Durante as entrevistas a equipe questiona os *stakeholders* sobre o sistema que será desenvolvido, visando identificar requisitos através das respostas (SOMMERVILLE, 2011).

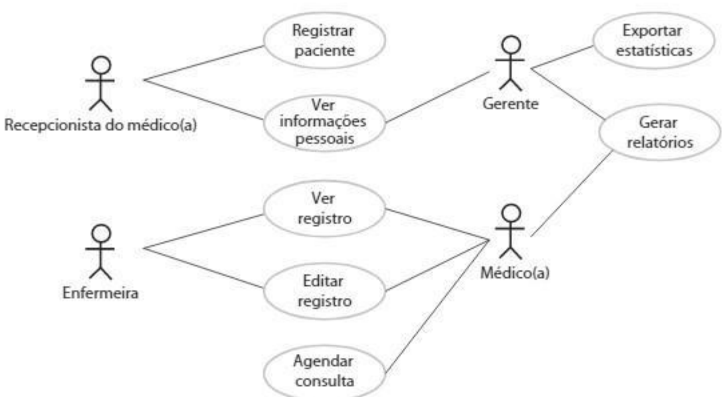
As entrevistas podem ser realizadas de duas principais formas: entrevistas fechadas e abertas. Durante as entrevistas fechadas uma série de perguntas predeterminadas são aplicadas aos *stakeholders*. As entrevistas abertas, por sua vez, não utilizam perguntas predeterminadas, a equipe é livre para explorar o questionamento da forma que preferir (SOMMERVILLE, 2011).

2.5.1.2 Casos de uso

Um caso de uso é um diagrama que visa demonstrar, de forma simples e direta, os atores contidos no sistema e as possíveis interações realizadas por cada ator. Cada possível interação realizada pelo ator pode ser suplementada por informações adicionais que a descrevem, através de artefatos textuais ou gráficos, como diagrama de sequência ou de estados da Linguagem de Modelagem Unificada (UML) (SOMMERVILLE, 2011).

A figura 4 contida abaixo, mostra um exemplo de diagrama de caso de uso, o qual demonstra os possíveis atores e interações contidos em uma clínica:

Figura 4 – Exemplo de diagrama de caso de uso



Fonte: SOMMERVILLE, 2011 (p. 75)

2.5.1.3 Brainstorming

As famosas *brainstormings* ou do inglês, chuvas de ideias, trata-se de uma ótima opção para obter novos requisitos ou até mesmo propor melhorias para os que já foram identificados. Sua aplicação se baseia em debates onde todos os integrantes discutem acerca do problema tratado. Os principais pontos abordados são separados e melhor estudados após o debate. (PRESSMAN; MAXIM, 2016)

2.5.1.4 Prototipação

O processo de prototipação trata-se do desenvolvimento de uma versão inicial do sistema, realizada de forma ilustrativa. Sua utilização permite "demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções" (SOMMERVILLE, 2011, p. 30).

Os protótipos são ótimos recursos nas fases de levantamento e validação dos requisitos. Sua utilização permite experimentar novas propostas de implementação ou mudanças nas partes já codificadas, possibilitando uma visão sobre sua viabilidade. Essa prática auxilia na redução de ações indevidas que poderiam gerar impactos no projeto (SOMMERVILLE, 2011).

2.5.2 Ferramentas para levantamento e validação de requisitos de software

Graças a evolução tecnológica e a busca constante por novos métodos para levantamento e validação de requisitos, diversas ferramentas têm surgido com o intuito de facilitar essas ações. Para esse trabalho, além das ferramentas tradicionais utilizadas para a documentação de requisitos como o *Microsoft Word*, também será utilizado recursos mais atuais como o *Figma* e *Whimsical*.

Como a utilização de ambas apresenta um papel crucial no levantamento e validação de requisitos, será apresentado abaixo uma breve descrição de cada.

2.5.2.1 Figma

Figma é uma ferramenta para design de interface totalmente colaborativa e online que possibilita a manipulação de vetores e criação de protótipos. Por ser uma aplicação *web* não é necessário a realização do *download* e instalação, todos seus recursos podem ser acessados através de um navegador, além disso, o material produzido é salvo automaticamente em seus servidores (FIGMA, 2021).

Os projetos podem ser compartilhados através de *links* ou *e-mails* com diferentes níveis de permissões. *Links* com permissões de edição permitem que mais de um profissional trabalhe ao mesmo tempo no projeto. *Links* com permissões apenas de visualização permitem que clientes acompanhem, em tempo real, as modificações que estão sendo realizadas, possibilitando uma melhor interação e um rápido *feedback* (FIGMA, 2021).

2.5.2.2 Whimsical

Whimsical é uma ferramenta colaborativa e *online* que permite a seus usuários a elaboração visual de ideais através de diversos recursos visuais que a plataforma oferece. A ferramenta apresenta recursos intuitivos e de fácil utilização além de estar acessível através de vários navegadores de *internet* (WHIMSICAL, 2021).

Atualmente a plataforma conta com recursos que possibilitam a criação de fluxogramas, mapas mentais, documentos escritos, *wireframes* e gerenciamento de atividades com *stick notes*, sendo, dessa forma, uma ótima opção na criação, planejamento e gerenciamento de projetos (WHIMSICAL, 2021).

2.6 Desenvolvimento do sistema

Após o planejamento inicial do projeto, com a definição do problema a ser tratado e os principais requisitos definidos, torna-se possível iniciar o processo de implementação, envolvendo etapas de configuração de ferramentas, codificação de requisitos, configuração dos recursos de armazenamento de dados e várias outras necessidades.

A seguir serão apresentadas as principais ferramentas, tecnologias, linguagens e *frameworks* que serão utilizados no processo de desenvolvimento do sistema.

2.6.1 Desenvolvimento *web*

Com o passar dos anos, o desenvolvimento *web* se tornou mais complexo em resposta às grandes mudanças. Loudon (2010) apresenta as principais características que as aplicações *web* devem seguir, a fim de serem mantidas mais facilmente e gerar menos custo ao longo de sua existência, sendo elas:

- **Disponibilidade contínua:** Deve estar disponível 24 horas por dia, 7 dias por semana, com rápidas respostas a todo momento.
- **Grande base de usuários:** Gerenciamento de diversas conexões simultâneas.
- **Diversidade:** Diferentes tipos de negócios, os desenvolvedores devem estar prontos para escrever códigos que possam ser reutilizados em lugares inesperados.
- **Longevidade:** O *software* deve partir do princípio de que ele será capaz de resistir há anos de alterações e manutenção.
- **Múltiplos ambientes:** Inesperado de onde o usuário estará acessando, ou seja, existe a necessidade de dar suporte a velhos navegadores ou outros dispositivos, com telas de tamanho totalmente diferentes.
- **Atualizações em tempo real:** Aplicações *web* estão sempre em evolução, ou seja, deve ser capaz de que essas alterações se adaptem a todos os outros ambientes.

Lazar (2001), apresenta que o desenvolvimento *web* deve ser centralizado no usuário, através de artefatos que apresentem boa usabilidade, ou seja, proporcionar ao usuário facilidade em realizar suas atividades. Tal prática proporciona aos usuários alto nível de satisfação, influenciando-os na adesão ao sistema.

Uma abordagem comum no desenvolvimento *web* é a estruturação das páginas de forma “responsiva”, ou seja, o sistema torna-se adaptável para diferentes tamanhos de tela, assim, sua utilização pode ser realizada tanto através de computadores quanto de dispositivos móveis como *smartphones*, sendo que, o único requisito necessário para sua utilização torna-se a existência de um navegador *web* instalado no dispositivo.

Apesar de ser uma solução simples, existem problemas que desfavorecem a sua prática, tais como usabilidade e desempenho (CARVALHO, 2014). Com o objetivo de evitar tais problemas, o desenvolvimento de uma versão do sistema voltada exclusivamente para o ambiente *mobile* vem se tornando uma prática comum em diferentes projetos.

2.6.2 Desenvolvimento mobile

O desenvolvimento *mobile* pode ser realizado de duas principais formas: o desenvolvimento nativo ou híbrido.

O desenvolvimento nativo remete a codificação do sistema de forma específica para a plataforma desejada. Tal prática proporciona a utilização de todos os recursos disponíveis gerando maior desempenho, por outro lado, o sistema fica limitado para a plataforma escolhida. Assim, caso seja necessária sua utilização em outra plataforma, o código deverá ser reescrito, possivelmente em outra linguagem (TAVARES, 2016).

O desenvolvimento híbrido por sua vez, traz consigo um grande benefício que é a possibilidade da utilização da mesma linguagem de programação para diferentes plataformas, possibilitando que o sistema seja compatível com diversos sistemas operacionais. Esse modelo de desenvolvimento, mesmo que seja de forma limitada comparada ao anterior, também permite o acesso aos recursos nativos da plataforma, pois será compilado na linguagem da plataforma (TAVARES, 2016).

Com a popularização do desenvolvimento *mobile* e *web*, diversos *frameworks* e bibliotecas foram desenvolvidos a fim de otimizar a produtividade dos desenvolvedores. Para esse projeto, será adotada a utilização do *React*, uma biblioteca que possibilita a criação de *softwares* para diferentes plataformas, tanto *web* quanto *mobile* (FACEBOOK OPEN SOURCE, 2021b).

2.6.3 React

O *React*, assim como definido por seus criadores “é uma biblioteca JavaScript declarativa, eficiente e flexível” (FACEBOOK OPEN SOURCE, 2021b), a qual é utilizada para a criação de interfaces de usuário, também conhecidas como User Interface (UI). Com a

sua utilização é possível compor interfaces complexas através de recursos denominados "componentes" (FACEBOOK OPEN SOURCE, 2021b).

Criado em 2011 pelo *Facebook*, o *React* surgiu com o objetivo de otimizar as constantes atualizações que ocorriam nos diversos elementos contidos no *feed* de notícias da rede social, tais como *chats*, *status*, listagens, entre outros elementos. Para isso o *React* implementa o sistema de componentes citado anteriormente (FACEBOOK OPEN SOURCE, 2021b).

Os componentes criados pelo *React* são trechos de códigos isolados responsáveis por construir um elemento presente em tela. Com a sua criação torna-se possível a reutilização desses elementos em diversas partes do projeto, sem a necessidade da repetição de código. Cada componente é responsável por manipular e exibir seus próprios dados. Quando os dados vinculados ao componente são alterados, ele é recarregado de forma independente na tela (FACEBOOK OPEN SOURCE, 2021b).

Assim, a atividade de manipular os diferentes elementos contidos em tela, semelhante ao o que ocorre no *feed* de notícias do *Facebook* torna-se mais simples com a utilização do sistema de componentes, permitindo o isolamento de código e a atualização das informações presentes em tela de forma mais prática e simples (FACEBOOK OPEN SOURCE, 2021b).

Com a utilização do *React* na criação das telas *web* do sistema, parte do código pode ser reutilizada na criação das telas *mobile*, aumentando a produtividade da equipe. Além disso, o *React Native* - biblioteca do *React* - aplica os conceitos de desenvolvimento híbrido citado anteriormente, assim o sistema *mobile* é capaz de ser executado em diferentes plataformas sem a necessidade da reescrita de código para uma linguagem específica (FACEBOOK OPEN SOURCE, 2021c).

Além do *Javascript*, o *React* também trabalha com linguagens como HTML e CSS na estruturação das telas. A utilização combinada dessas linguagens normalmente é utilizada através de arquivos com a extensão JSX, uma extensão de sintaxe para *Javascript* que facilita na leitura e escrita de código. Também é possível a utilização de *Typescript* visando melhorar o controle sobre o código, nesse caso a extensão dos arquivos passa a ser TSX, visando o mesmo objetivo do JSX (FACEBOOK OPEN SOURCE, 2021b).

2.6.4 HTML: HiperText Markup Language

O HTML (*HiperText Markup Language* ou Linguagem de Marcação de Hipertexto) é uma linguagem de marcação responsável por construir a estrutura de uma página *web*, através de códigos que são capazes de inserir e delimitar os conteúdos que constituem a página, tais como títulos, textos, imagens, vídeos e diversos outros elementos (MOZILLA, 2021a).

Os códigos implementados pelo HTML são comumente denominados de *Tags,* os quais apresentam uma simples sintaxe constituída pelo sinal de menor, nome da *Tag* e sinal de maior. Um exemplo de *Tag* para títulos pode ser verificado na figura 5 contida abaixo:

Figura 5 – Exemplo de *tag* HTML

Fonte: Próprios autores

Cada *Tag* apresenta uma função dentro do código, desde a divisão de conteúdos em blocos até a divisão de elementos por tipo, possibilitando assim a identificação e manipulação de cada elemento (MOZILLA, 2021a).

2.6.5 CSS: Cascading Style Sheets

O CSS (*Cascading Style Sheets* ou Folhas de Estilo em Cascata) é uma linguagem de folhas de estilos utilizada para trabalhar a forma como as informações contidas em uma página *web* são exibidas para seus usuários, tornando-as mais atrativas e agradáveis. (MOZILLA, 2021b).

Para a estilização das páginas *web*, o CSS permite que o desenvolver aplique estilos de forma seletiva aos elementos contidos na página que foi previamente estruturada pelo HTML, assim como demonstrado na figura 6 contida a seguir:

Figura 6 – Exemplo de estilização utilizando CSS

Fonte: Próprios autores

Com a utilização do sistema de seletores, torna-se possível diversas formas de estilização, desde as mais genéricas que aumentam a produtividade do projeto, até as mais específicas utilizadas em pequenos detalhes da página (MOZILLA, 2021b).

2.6.6 JavaScript

O *Javascript*, diferentemente do HMTL e CSS, é uma linguagem de programação interpretada, criada inicialmente para ser executada em navegadores e responsável por manipular os elementos contidos em uma página *web*. Por se tratar de uma linguagem multiparadigma e dinâmica, o *Javascript* é capaz de lidar com diferentes estilos de programação, tais como orientação a objetos, imperativos e declarativos (MOZILLA, 2021c).

Com o passar dos anos e o surgimento de novas tecnologias, bibliotecas, API e *frameworks*, o *JavaScript* ganhou novas funcionalidades, não limitando-se mais a apenas páginas *webs* e navegadores. Sua utilização pode ser aplicada na resolução de diversos problemas, como a criação de sistemas *webs*, aplicativos *mobiles*, *softwares* para *desktops* e até mesmo servidores (MOZILLA, 2021c).

Graças aos diversos recursos que o *Javascript* disponibiliza, sua utilização ganhou um aumento de popularidade entre várias empresas, sendo utilizado tanto por pequenas *startups* quanto por grandes empresas já consolidadas no mercado (MOZILLA, 2021c).

Entretanto, apesar de popular, ainda existem fatores que desfavorecem a utilização do *Javascript* em projetos extensos, como por exemplo a falta de definição de tipos. Para resolver esse problema surge então o *Typescript*.

2.6.7 TypeScrypt

A maioria das linguagens de programação existentes são capazes de identificar erros presentes no código durante a sua escrita, ou então, durante a sua compilação. Para isso são utilizados padrões de sintaxe de código que englobam recursos como as declarações de tipos e marcadores que sinalizam a finalização de blocos de códigos (MICROSOFT, 2021).

O *JavaScript* entretanto, por ser criado inicialmente para ser uma linguagem de *script* simples para navegadores apresenta uma sintaxe de código bem flexível, reduzindo assim o número de erros apresentados durante sua escrita. Sua flexibilidade pode gerar benefícios na criação de pequenos programas aumentando a produtividade do desenvolvedor. Entretanto, à medida que os programas desenvolvidos se tornam complexos, comportando centenas ou milhares de linhas de código, tanto a probabilidade de ocorrência de erros e *bugs* quanto a sua dificuldade em solução aumentam de forma proporcional ao tamanho do código (MICROSOFT, 2021).

Dado o cenário de preocupação envolvendo o *Javascript* na criação de sistemas com grandes quantidades de código, surge então o *Typescript*, uma linguagem de programação considerada um superconjunto do *Javascript*, ou seja, o *Typescript* oferece todos os recursos disponíveis no *Javascript*, adicionando sobre eles uma camada adicional de tipos que permite ao desenvolvedor um maior controle sobre o código produzido, auxiliando na resolução e prevenção de erros e *bugs* que poderiam afetar negativamente o projeto (MICROSOFT, 2021).

Com a utilização de editores de códigos apropriados, o *Typescript* realiza a identificação de erros mesmo antes da execução do código, possibilitando uma rápida e precisa correção. Além disso, editores de códigos, tais como os ambientes de desenvolvimento integrado, oferecem diversos recursos que proporcionam ganhos na produtividade do desenvolvedor, assim a sua utilização será adotada neste projeto.

2.6.8 Ambiente de Desenvolvimento Integrado

Os ambientes de desenvolvimento integrado (do inglês, *Integrated Development Environment* - IDE) comumente conhecidos pela sigla IDE, são editores de códigos que fornecem suporte aos desenvolvedores durante a codificação de um sistema através de recursos como interpretadores, compiladores e depuradores (RUSSI; CHARÃO, 2011).

IDEs mais avançadas são capazes de auxiliar o programador na identificação de erros no projeto, ocasionados tanto pelo uso indevido da sintaxe de código da linguagem, quanto por erros ocasionados na lógica implementada (RUSSI; CHARÃO, 2011). Dentre as IDEs de destaque no mercado, podemos citar o *Visual Studio Code*.

2.6.8.1 Visual Studio Code

O *Visual Studio Code*, também chamado de VSCode, é um editor de código-fonte desenvolvido pela *Microsoft*, estando disponível para os sistemas operacionais Windows, macOS e Linux. Além de possuir por padrão suporte integrado para o desenvolvimento com *JavaScript*, *TypeScript* e Node.js, o VSCode apresenta um rico sistema de extensões que possibilitam que o desenvolvedor utilize e gerencie uma série de outras ferramentas como o C, C++, Java, Python, PHP, Docker, entre outras (MICROSOFT, 2021).

O sistema de extensões do VSCode também permite o gerenciamento de recursos utilizados pelo projeto, tais como as ferramentas de versionamento como o Git e ferramentas responsáveis por manter os padrões de código estabelecidos pela equipe, como por exemplo o Eslint, ambos citados anteriormente (MICROSOFT, 2021).

Seus recursos serão amplamente utilizados nesse projeto, envolvendo todas as etapas onde a codificação seja necessária, seja ela no desenvolvimento das telas *web* e *mobile*, utilizando a biblioteca *React* citada anteriormente ou no desenvolvimento do servidor utilizando o Node.js, o qual será explanado posteriormente.

2.6.9 Node.js

Node.js é um ambiente de desenvolvimento de código aberto para o lado do servidor, tratando nativamente das requisições de forma assíncrona, resolvendo assim o problema de desempenho presentes em arquiteturas que trabalham no modelo *single-thread* (PEREIRA, 2014).

Seu desenvolvimento teve início no final de 2009 por Ryan Dahl com o auxílio de 14 colaboradores, visando a criação de uma tecnologia capaz de resolver os problemas de desempenho dos servidores *web*, ocasionado pelos *Blocking Thread,* onde funções do sistema ficam paralisadas enquanto uma requisição é resolvida (PEREIRA, 2014).

Graças a esse objetivo, a arquitetura aplicada pelo Node.js segue o modelo *Non-Blocking Thread,* assim o servidor trabalha de forma paralela às requisições realizadas, evitando que suas funções sejam bloqueadas (PEREIRA, 2014).

Com ambiente assíncrono e orientado a eventos, o Node.js tem como principal benefício ser escalável em desenvolvimento de aplicações para rede. O *Hypertext Transfer Protocol* (HTTP) no Node.js tem alta performance com baixo consumo e latência, o tornando uma ótima escolha para o desenvolvimento *web*. (OPENJS FOUNDATION, 2021b).

2.6.10 Banco de dados

Heuser (2009, p. 22), define um banco de dados como um "[...] conjunto de dados integrados que tem por objetivo atender a uma comunidade de usuários", ou seja, é um conjunto de informações que está disponível para um ou mais usuários. Para o gerenciamento dessas informações são utilizados Sistemas de Gerenciamento de Banco de Dados (SGBD), sendo responsáveis por realizar o armazenamento, acesso, alteração e exclusão das informações (HEUSER, 2009).

Um banco de dados pode ser desenvolvido através de diferentes estratégias, a mais comum é o modelo relacional, onde as informações são armazenadas em formato de tabelas e relações (HEUSER, 2009). Heuser (2009, p. 120) afirma que "uma tabela é um conjunto não ordenado de linhas [...], cada linha é composta por uma série de campos".

Seguindo esse modelo, um banco de dados responsável por armazenar dados dos empregados de uma organização apresentaria em sua estrutura uma tabela reservada apenas

para o registro de empregados. Assim, cada empregado representaria uma nova linha na tabela e cada informação do empregado seria um novo campo contido na linha (HEUSER, 2009).

De forma geral, durante o desenvolvimento de um *software* é comum que a modelagem do banco de dados seja realizada antes mesmo da codificação, com o objetivo de estruturar os diferentes tipos de informações que serão armazenados. Para isso existem três principais modelos que são construídos através das linguagens de modelagem de dados, textuais e gráficas, sendo eles o modelo conceitual, lógico e físico. (HEUSER, 2009).

Heuser (2009, p.25) define o modelo conceitual como “[...] uma descrição do banco de dados de forma independente de implementação em um SGBD”. Sua construção normalmente é realizada através do diagrama denominado diagrama entidade-relacionamento (DER). Um exemplo pode ser observado na figura 7 contida abaixo:

Figura 7 - Exemplo de diagrama entidade-relacionamento

Fonte: HEUSER, 2009 (p.25)

Um modelo lógico por sua vez é uma descrição detalhada da estrutura do banco de forma dependente a um SGBD específico. Nele são descritos quais tabelas que o banco contém e quais colunas existem em cada uma delas (HEUSER, 2009). A figura a seguir mostra um exemplo de modelagem lógica das tabelas de produto e tipo de produto:

Figura 8 - Exemplo de um modelo lógico

```
TipoDeProduto (CodTipoProd,DescrTipoProd)
Produto (CodProd,DescrProd,PrecoProd,CodTipoProd)
        CodTipoProd referencia TipoDeProduto
```

Fonte: HEUSER, 2009 (p.27)

Por fim, o modelo físico se baseia na construção do banco de dados através de linguagens e notações que não são padronizadas entre os SGBD. Na prática, o modelo físico é um processo contínuo que ocorre mesmo após a implementação do banco de dados. Esse processo contínuo de adaptações e melhorias é denominado de sintonia de banco de dados (HEUSER, 2009).

Um sistema de banco de dados trabalha com dois principais tipos de linguagens, as linguagens de definição de dados (DDL) e as linguagens de manipulação de dados (DML). As DLL são responsáveis por especificar o esquema de banco de dados, já as DML são responsáveis por realizar consultas e manipulação dos dados armazenados (KORTH, H.F et al, 2020).

Assim como explicam os autores Korth, Silberschatz e Sudarshan (2020), na prática as DML e DLL não são duas linguagens separadas, elas constituem uma única linguagem de banco de dados, como por exemplo a *Structured Query Language* (SQL), amplamente utilizada por diversos bancos de dados.

Para o desenvolvimento desse projeto será abordado a utilização do PostgreSQL, um banco de dados gratuito e de fácil utilização.

2.6.10.1 PostgreSQL

PostgreSQL é um sistema de gerenciamento de banco de dados relacional de objetos (ORDBMS). Sua origem é derivada do pacote POSTGRES, desenvolvido pelo Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley na década de 80. Seu código é totalmente aberto, com mais de 30 anos de desenvolvimento ativo, fator que lhe proporcionou uma reputação de confiabilidade, robustez de recursos e desempenho (THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2021).

Graças a sua licença gratuita, o PostgreSQL pode ser usado, modificado e distribuído por qualquer pessoa de forma livre, independente do propósito, seja ele privado, comercial ou acadêmico. Com isso, nos últimos anos sua utilização tornou-se popular entre empresas e acadêmicos, sendo amplamente utilizado nos mais variados tipos de projetos (THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2021).

2.6.11 Ferramentas de planejamento e gestão

Além da definição de ferramentas e processos voltados para o desenvolvimento do projeto, torna-se necessário também a utilização de ferramentas que auxiliem no monitoramento e gestão das atividades a serem realizadas, com o objetivo de manter o controle sobre o cronograma planejado. Dentre as ferramentas utilizadas para esse projeto podemos destacar o Trello.

2.6.11.1 Trello

Trello é uma ferramenta online de gerenciamento de projetos e atividades que permite diversos usuários trabalharem de forma colaborativa no gerenciamento das informações do projeto. Sua interface é semelhante a um quadro Kanban, apresentando assim colunas que simbolizam as etapas do processo e cartões que simbolizam as atividades a serem realizadas (TRELLO, 2021).

Cada cartão possibilita ao usuário inserir diversos elementos que auxiliam no controle da atividade que o cartão representa, tais como, vídeos, imagens, *checklists*, comentários, responsáveis e diversos outros recursos (TRELLO, 2021).

Assim, o Trello permite de forma fácil e eficiente que os membros da equipe identifiquem o atual andamento do projeto e colaborem para a constante atualização do mesmo, evitando dessa forma problemas que poderiam ser ocasionados por uma precária comunicação entre os integrantes da equipe (TRELLO, 2021).

3. DESENVOLVIMENTO

Esse capítulo apresentará os passos realizados para o desenvolvimento do software proposto, utilizando as metodologias e ferramentas apresentadas anteriormente. Assim, será apresentado as etapas de versionamento de código, arquiteturas desenvolvidas, padrões de documentação, prototipação de telas, implementação dos requisitos e outros resultados desenvolvidos durante o projeto.

3.1 Processo metodológico

Como já citado anteriormente no trabalho, a definição inicial das informações do projeto é uma das principais etapas que influenciam para o sucesso do mesmo (PRESSMAN; MAXIM, 2016). Assim, inicialmente foi necessário definir o escopo do projeto, através de um levantamento dos requisitos que são indispensáveis para o seu funcionamento. Nessa etapa, as informações foram levantadas principalmente através de entrevistas com os *stakeholders* e também com a realização de *brainstormings*.

Com as informações obtidas na primeira etapa do projeto foi possível a criação do Documento de Visão (APÊNDICE A), responsável por apresentar as principais informações do sistema, entre elas o backlog, o qual pode ser definido como uma lista de funcionalidades a serem implementadas e que gerem valor para o cliente (PRESSMAN; MAXIM, 2016).

Após a definição do problema que será tratado através da criação do documento de visão, foi dado início ao desenvolvimento do projeto adotando a metodologia Scrum para o seu gerenciamento, visando melhorar o controle do desenvolvimento das atividades planejadas através da utilização das *Sprints* apresentadas anteriormente,

Todo o desenvolvimento do sistema foi guiado pela metodologia Test Driven Development (TDD), ou em português, desenvolvimento guiado por testes. Com a sua utilização todas as partes do sistemas são desenvolvidas através de ciclos de atividades que possibilitam uma grande variedade de testes, tornando o sistema menos suscetível a falhas (ANICHE, 2012).

De forma simples, o desenvolvedor ao implementar uma nova funcionalidade (*feature*) ao projeto deve seguir uma sequência de passos, começando pela criação do teste para a nova *feature*, execução de todos os testes do sistema, verificação do resultado do teste o qual deverá apresentar uma falha devido a ausência da funcionalidade, a implementação da funcionalidade ausente e por fim uma nova execução de todos os testes com o objetivo que nenhum deles falhe. Para a implementação dos testes, foi adotada a utilização da ferramenta *Jest* citada anteriormente.

Por fim, após a finalização do desenvolvimento do sistema será realizada a homologação do mesmo, sendo necessário a disponibilização de um servidor para o consumo de dados e a criação de uma conta na plataforma da Google Play Console para a publicação do aplicativo na loja Android.

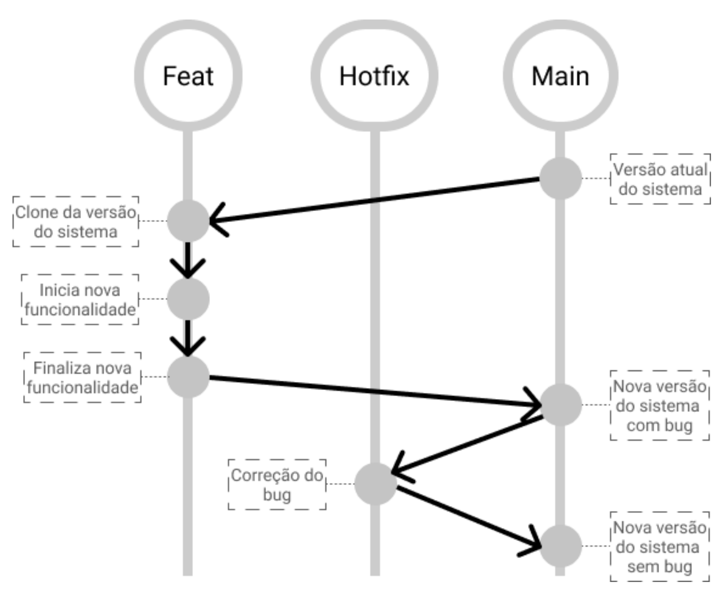
3.2 Política de versionamento

Antes de iniciar o desenvolvimento foi decidido pela equipe que todo e qualquer artefato produzido seria armazenado através da ferramenta de controle de versão Git, visando uma maior segurança sobre a integridade dos artefatos produzidos independente do número de atualizações necessárias.

Para que o sistema de controle de versões fosse utilizado de maneira correta, foi definido uma política de versionamento que norteou o processo de desenvolvimento evitando problemas comuns relacionados a modificações paralelas nos artefatos, tais como: conflitos de código, versões duplicadas e/ou desatualizadas e a liberação do código.

A seguir, na figura 9, será apresentado uma representação gráfica do fluxo de versionamento utilizado, com uma breve descrição textual logo após a figura.

Figura 9 - GitFlow



Fonte: Próprios autores

Inicialmente decidiu-se que a política de versionamento conteria três principais tipos de branchs, sendo elas as branches *Feat*, *HotFix* e *Main*. Para esse projeto, a branch principal é denominada *Main*, nela será contido todo o código estável da aplicação. As branches do tipo *Feat* são responsáveis pela implementação ou alteração de uma funcionalidade do sistema,

enquanto que as branches do tipo *Hotfix* são responsáveis pela correção dos bugs urgentes que surgirem.

Desta forma foi possível adicionar e reparar módulos do sistema com o mínimo de problemas possíveis, melhorando a produtividade e evitando a ocorrência de problemas inesperados.

É válido ressaltar que essa política de versionamento do projeto será adotada apenas durante a construção da primeira versão do sistema, a qual deve ser realizada de forma rápida e objetiva. Com a primeira versão concluída e entregue, mudanças serão aplicadas sobre a estrutura de versionamento existente, adicionando novas camadas de branche responsáveis por validar modificações feitas ao software antes de atualizá-lo para novas versões.

3.3 Sprints

Como definido anteriormente o projeto adotará a utilização da metodologia *Scrum* para o seu gerenciamento. Assim, para manter uma constante entrega de artefatos e feedback foi definido pela equipe um time box (tempo de duração) semanal para as Sprints, não sendo necessário a realização de reuniões diárias, mas com a utilização constante dos meios de comunicação sempre que necessário relatar algo relacionado ao projeto.

3.3.1 *Sprint* 01 – Levantamento de requisitos

A primeira *Sprint* do projeto foi responsável pelo levantamento das principais informações relacionadas ao sistema, dando origem aos documentos de visão (APÊNDICE A), diagrama dos casos de uso (APÊNDICE B) e histórias de usuário (APÊNDICE C). Esses foram os principais artefatos que guiaram o desenvolvimento do software.

Para a obtenção das informações contidas nos documentos criados na primeira *Sprint*, foram realizadas entrevistas com os possíveis *stakeholders* do projeto em conjunto com a realização das *brainstormings*. Tais técnicas possibilitaram a criação do Documento de Visão, evidenciando o problema que o sistema solucionará juntamente com uma visão macro dos principais requisitos e estrutura do projeto.

Após a definição do Documento de Visão, foi criado então o Diagrama de Caso de Uso, que demonstra de forma simples os atores que interagem com o sistema e suas possíveis

ações. Para melhor descrever cada ação que um determinado ator poderá realizar, foi criado então o documento de Especificação de Caso de Uso, o qual descreve através das Histórias de Usuário o funcionamento de cada caso de uso contido no diagrama.

Foram levantadas as seguintes histórias de usuário:

- US001 - Manter Produtos
- US002 - Manter Serviços
- US003 - Gerenciar Solicitações de Agendamento de Serviços
- US004 - Gerar Relatório de Serviços e/ou Produto
- US005 - Efetuar *Login*
- US006 - Registrar-se
- US007 - Visualizar Produto Disponível
- US008 - Visualizar Serviço Disponível
- US009 - Acompanhar Andamento de Serviço
- US0010 - Manter Dados do Cliente
- US0011 - Manter Dados do Pet
- US0012 - Solicitar Agendamento de Serviço
- US0013 - Cancelar Agendamento de Serviço

3.3.2 *Sprint* 02 –Validação de requisitos e prototipação de telas.

Com a visão macro do sistema definido, foi dado início o desenvolvimento dos primeiros protótipos através da ferramenta *Figma*, com o objetivo de validar as informações levantadas até o momento através da criação gráfica de telas que proporcionasse uma visualização inicial do software juntamente com seu fluxo de trabalho.

O processo de prototipação foi dividido por etapas, onde a primeira etapa foi responsável pela ideia inicial das telas, definindo as estruturas e elementos que a constituíram. Para isso foram criados pequenos esboços denominados *wireframes*. Os *wireframes* são versões "primitivas" das telas, sendo formados por estruturas simples e geométricas, sem a presença de cores, ícones ou identificação visual como um todo, podendo ser produzidos através de *softwares* ou até mesmo com lápis e papel.

A figura 10 contida abaixo retrata os primeiros *wireframes* criados para o projeto. Nela estão presentes as estruturas iniciais das telas de “login”, “registrar-se”, “lista de serviço” e “detalhes do serviço”.

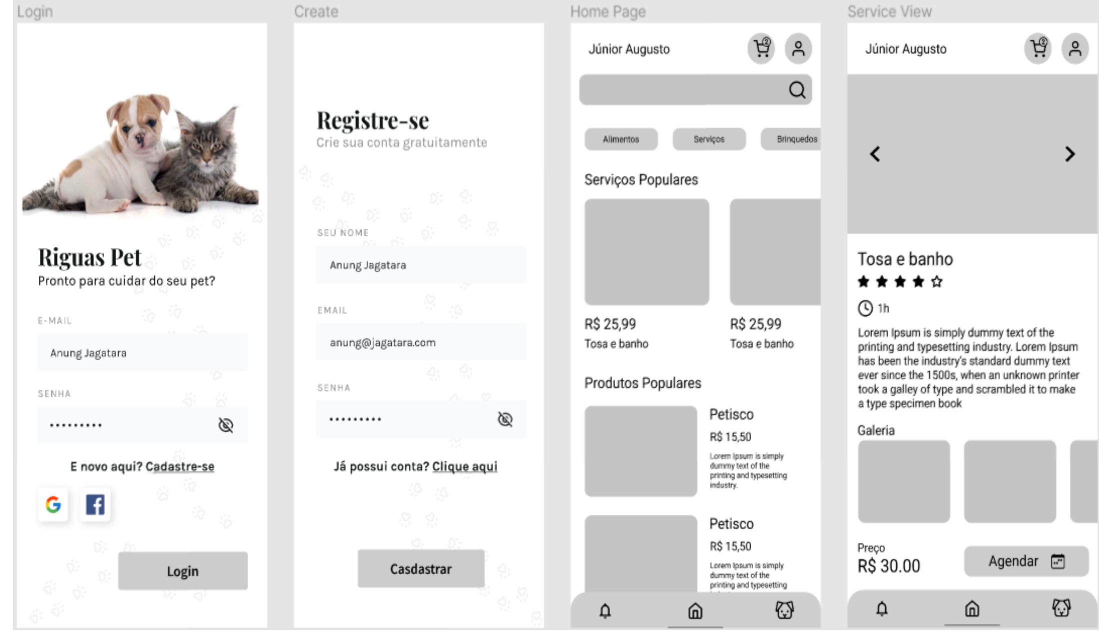
Figura 10 - Primeiros *wireframes* do aplicativo.

Fonte: Próprios autores

A criação dos primeiros *wireframes* foi realizada da forma mais simples possível, utilizando apenas lápis e papel. Tal escolha teve como objetivo evitar qualquer distração que um *software* especializado na construção de telas pudesse oferecer, como por exemplo ícones, imagens, cores e outros elementos visuais. Isso pois, a primeira etapa buscava apenas identificar as estruturas das páginas e validar seu fluxo de navegação.

A segunda etapa do processo de prototipação, deu-se pela criação dos primeiros *mockups* (modelos de telas) do sistema com a ferramenta *figma* (Figura 11), utilizando como base os *wireframes* criados anteriormente.

Figura 11 - Primeiros *mockups* do aplicativo.



Fonte: Próprios autores

Por fim, a terceira etapa do processo de prototipação foi responsável pela criação das demais telas do sistema (APÊNDICE XX), aplicando também a implementação de recursos visuais que gerassem a identidade visual do aplicativo, assim como ícones, cores, imagens, entre outros. Após a etapa de prototipação, foi dado início ao processo de implementação, começando pela definição da arquitetura do sistema e modelagem do banco de dados.

3.3.3 *Sprint* 03 – Definição da arquitetura do sistema e modelagem inicial do banco de dados.

Com o objetivo de melhorar o desenvolvimento dos recursos implementados pelo sistema, foi definido pela equipe de desenvolvimento um modelo de arquitetura em camadas que possibilite um padrão na implementação das funcionalidades. Tal prática visa isolar de forma eficiente as quatro principais camadas presentes na aplicação, sendo elas a camada de interação do usuário (*View*), a camada de acesso e resposta da aplicação (*Controller*), a camada de regras de negócio (*Service*) e a camada de acesso ao banco de dados (*Model*).

A separação das camadas pode ser visualizada de forma gráfica através do Diagrama de Componentes (APÊNDICE D). Uma breve descrição de cada camada será apresentada abaixo.

Basicamente, a primeira camada (*View*) é responsável pela interação do usuário, através da disponibilização de interfaces gráficas que possibilitem o acesso às funcionalidades do sistema.

A segunda camada (*Controller*), responsável pelo acesso e resposta da aplicação, possibilitará que o usuário solicite ações de busca, cadastro, alteração ou exclusão de dados no sistema, validando se as informações necessárias para a ação solicitada foram informadas corretamente e retornando o valor de resposta, seja ele o valor desejado pelo usuário no caso de sucesso ou um valor de erro especificando o motivo da solicitação ter falhado.

A camada de regra de negócios (Service), por sua vez, visa implementar qualquer validação referente às solicitações requisitadas pelos usuários, realizando por exemplo validação nos dados durante as buscas, cadastros e exclusões ou limitando as ações para diferentes tipos de usuários e permissões.

Por fim, a camada de acesso ao banco de dados (*Model*) torna-se responsável por gerenciar o armazenamento dos dados do sistema, realizando dessa forma as opções de busca, cadastro, alteração ou exclusão de dados.

Após a definição da arquitetura do sistema foi desenvolvido o diagrama do primeiro modelo conceitual do banco de dados (APÊNDICE E), através da utilização das informações obtidas nos documentos de visão, casos de uso e histórias de usuário. O desenvolvimento deste diagrama visava identificar todas as principais informações que o sistema necessitaria armazenar, possibilitando assim uma visão inicial de como seria desenvolvidos as interfaces gráficas para que tais informações fossem manipuladas pelos usuários.

3.3.4 *Sprint* 04 - Preparação do ambiente de desenvolvimento

A quarta *Sprint* teve como objetivo a preparação do ambiente de desenvolvimento, sendo dividido por etapas. A primeira etapa foi responsável pela criação dos repositórios Git na plataforma do Github. Eles foram divididos de três formas, sendo elas o repositório para o código *back-end* (typet-backend), repositório para o código *front-end web* (typet-web) e o repositório para o código *front-end mobile* (typet-mobile). Todos os três repositórios foram agrupados no projeto “Typet" a fim de manter a organização, assim como demonstrado na figura 12, contida abaixo.

Figura 12 - Projeto no GitHub

Fonte: Próprios autores

A segunda etapa foi responsável pela inicialização dos projetos, configurando dessa forma as estruturas de pastas, instalação de dependências e criação dos arquivos de configurações referentes aos padrões de código e *commits*.

Com exceção da estrutura de pastas e dependências, as configurações de padrões de código e *commits* são semelhantes para os três repositórios e será explicada logo abaixo.

Começando pela padronização de código, ela será gerenciada pela biblioteca *Eslint*, para isso é necessário realizar os procedimentos de instalação contidos em sua documentação oficial. Após a instalação, foi criado então o arquivo ".eslintrc.js", nele estão contidas todas as regras relacionadas aos padrões de código presentes no projeto, sendo possível customizá-lo conforme necessário. A figura 13 contida abaixo demonstra as regras de código adotadas para o sistema.

Figura 13 - Padrões de código com *Eslint*

```
1   module.exports = {
2     env: {
3       es2021: true,
4       node: true,
5     },
6     extends: ['eslint:recommended', 'plugin:@typescript-eslint/recommended'],
7     parser: '@typescript-eslint/parser',
8     parserOptions: {
9       ecmaVersion: 12,
10      sourceType: 'module',
11    },
12    plugins: ['@typescript-eslint'],
13    rules: {
14      semi: ['error', 'always'], // ; No final é obrigatório.
15
16      'eol-last': ['error', 'always'], // Linha extra no final de cada arquivo.
17      'spaced-comment': ['error', 'always'], // Espaço antes do comentário.
18      'array-bracket-spacing': ['error', 'never'], // Espaço antes e depois dos colchetes do array.
19      'object-curly-spacing': ['error', 'always'], // Espaço antes e depois das chaves do objeto.
20
21      // Vírgula no final de arrays, objetos, importações, exportações e funções.
22      'comma-dangle': [
23        'error',
24        {
25          arrays: 'always-multiline',
26          objects: 'always-multiline',
27          imports: 'always-multiline',
28          exports: 'always-multiline',
29          functions: 'only-multiline',
30        },
31      ],
32
33      // Espaço entre a função e seus parenteses.
34      'space-before-function-paren': [
35        'error',
36        {
37          anonymous: 'ignore',
38          named: 'ignore',
39          asyncArrow: 'ignore',
40        },
41      ],
42
43      camelcase: ['error', { properties: 'never', ignoreDestructuring: true }],
44
45      '@typescript-eslint/explicit-module-boundary-types': 'off',
46      '@typescript-eslint/no-var-requires': 'off',
47    },
48  };
```

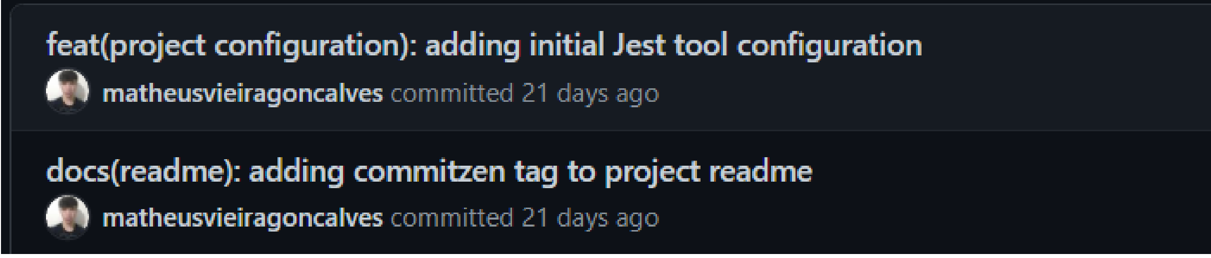
Fonte: próprios autores

A padronização das mensagens de *commit* por sua vez é gerenciada por duas bibliotecas: "*commitlint*" e "*commitzen*". A biblioteca de *commitlint* é responsável por validar a mensagem de *commit*. Com a sua utilização é possível determinar um padrão a ser seguido contendo assim palavras chaves, tamanho máximo de caracteres ou qualquer outra limitação.

O *commitlint* após instalado já oferece um conjunto de regras adotado pela comunidade de desenvolvimento. Nele a mensagem de commit deve seguir o seguinte padrão:

"palavra chave:(contexto) Descrição". Um exemplo de mensagem de *commit* pode ser verificado na figura 14 logo abaixo.

Figura 14 - Exemplo de padrões do *commitlint*



Fonte: próprios autores

É válido ressaltar que as regras contidas no *commitlint* são totalmente flexíveis, ou seja, é possível adaptá-las conforme a preferência da equipe. Caso o desenvolvedor tente realizar um *commit* que não siga os padrões especificados, a biblioteca bloqueará a mensagem e solicitará a sua correção, assim como demonstrado na figura 15, listada a seguir:

Figura 15 - Exemplo de erro retornado pelo *commitlint*

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL

PS D:\Repositorios\typet\typet-backend> git commit -m "mensagem fora do padrão"
⧗   input: mensagem fora do padrão
✖   subject may not be empty [subject-empty]
✖   type may not be empty [type-empty]

✖   found 2 problems, 0 warnings
ⓘ   Get help: https://github.com/conventional-changelog/commitlint/#what-is-commitlint

husky - commit-msg hook exited with code 1 (error)
PS D:\Repositorios\typet\typet-backend>
```

Fonte: próprios autores

A biblioteca *commitzen* por sua vez é responsável pela criação das mensagens de commit. Sua utilização não é obrigatória, mas proporciona uma maior facilidade em seguir os padrões descritos pela *commitlint*. A seguir, nas figuras 16 e 17, será mostrado um exemplo de como são criadas as mensagens de commit utilizando a biblioteca *commitzen*.

figuras 16 - *Commitzen*: Escolha do tipo do *commit*

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL                                  node
PS D:\Repositorios\typet\typet-backend> yarn cz
yarn run v1.22.10
$ D:\Repositorios\typet\typet-backend\node_modules\.bin\cz
cz-cli@4.2.4, cz-conventional-changelog@3.3.0

? Select the type of change that you're committing: (Use arrow keys)
> feat:     A new feature
  fix:      A bug fix
  docs:     Documentation only changes
  style:    Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
  refactor: A code change that neither fixes a bug nor adds a feature
  perf:     A code change that improves performance
  test:     Adding missing tests or correcting existing tests
(Move up and down to reveal more choices)
```

Fonte: próprios autores

figuras 17 - *Commitzen*: padronização da mensagem de *commit*

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL
PS D:\Repositorios\typet\typet-backend> yarn cz
yarn run v1.22.10
$ D:\Repositorios\typet\typet-backend\node_modules\.bin\cz
cz-cli@4.2.4, cz-conventional-changelog@3.3.0

? Select the type of change that you're committing: feat:     A new feature
? What is the scope of this change (e.g. component or file name): (press enter to skip) users module
? Write a short, imperative tense description of the change (max 80 chars):
 (28) adding logging functionality
? Provide a longer description of the change: (press enter to skip)

? Are there any breaking changes? No
? Does this change affect any open issues? No
```

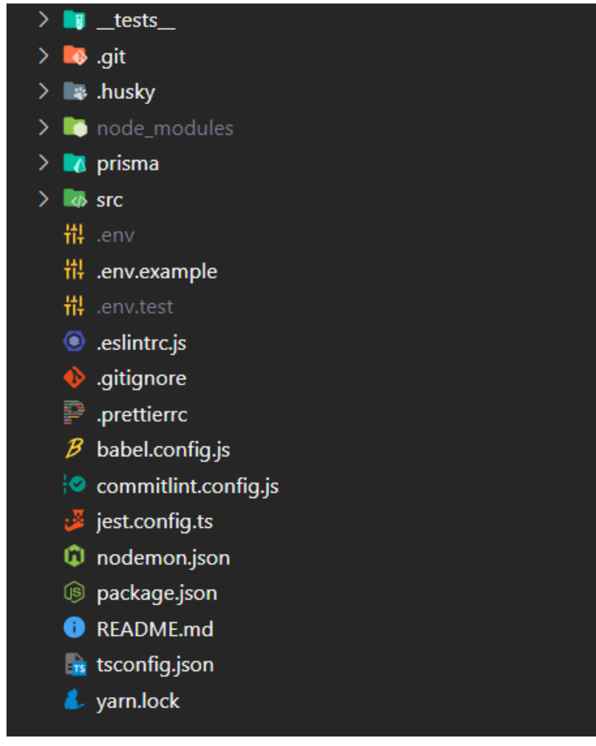
Fonte: próprios autores

Como mostrado nas figuras acima, a criação de *commits* utilizando o *commitzen* permite ao desenvolvedor uma breve visualização das informações que devem estar contidas na mensagem, facilitando a padronização da mesma.

3.3.5 *Sprint* 05 - Estrutura de pastas e arquivos do *Back-End*

Após a configuração dos arquivos responsáveis pela padronização de código e mensagens de *commit* nos três repositórios, foi então criada a estrutura inicial do projeto *back-end*. Começando pela pasta raiz do projeto, nela estão presentes todos os principais arquivos de configuração necessários, assim como os arquivos do *Eslint* e *Commitlint* criados no tópico anterior. A estrutura de pastas e arquivos contidos na raiz do projeto pode ser vista na figura 18.

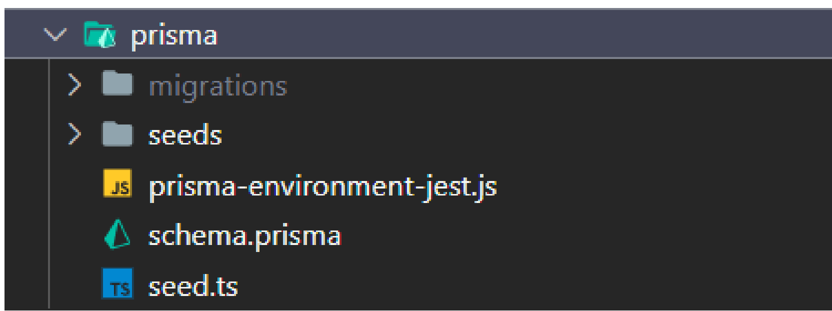
figura 18 - Estrutura de pastas e arquivos na raiz do *back-end*



Fonte: próprios autores

A pasta "*prisma*" localizada na raíz do projeto conterá todas as informações relacionadas a estrutura do banco de dados, tais como os arquivos de migrações (*migrations*) e as sementes (*seeds*), responsáveis respectivamente por criar as tabelas do banco de dados e realizar a inserção dos dados pré-cadastrados. A figura 19 mostra a estrutura contida nesta pasta.

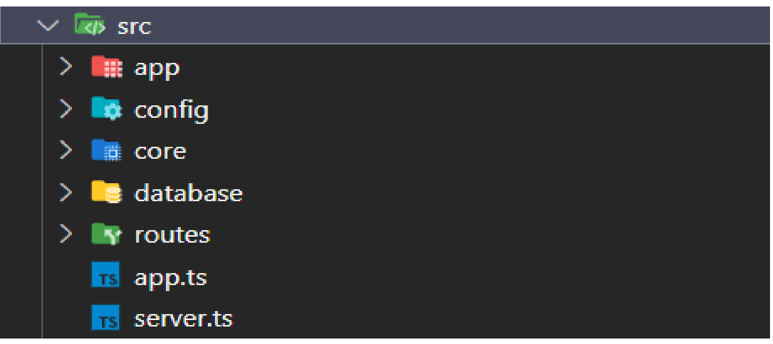
figura 19 - Estrutura da pasta "*prisma*"



Fonte: próprios autores

A pasta “src” por sua vez conterá todo o código relacionado a aplicação do projeto (figura 20). Nela haverá as configurações do servidor (app.ts e server.ts), os arquivos de rotas (*routes*), as configurações relacionadas ao acesso do banco de dados (*database*), o arquivos presentes no núcleo da aplicação (*core*), configurações gerais do sistema como a autenticação (*config*) e por fim os arquivos que constituem as funcionalidades da aplicação (*app*).

figura 20 - Estrutura da pasta “src”



Fonte: próprios autores

A pasta "app", como citado anteriormente, armazenará todo o código relacionado às funcionalidades da aplicação (figura 21). Nela haverá as principais interfaces do sistema responsáveis por modelar o formato dos dados trabalhados (*@types*), funções responsáveis pelas tratativa e retorno de erros (*CustomError*), as entidades responsáveis por armazenar os dados do projeto (*entities*), funções que realizarão as tratativas de dados presentes nas requisições a API, tais como autenticação (*middlewares*), os módulos do sistema que constituem as funcionalidades implementadas (*modules*) e por fim as classes responsáveis pelo acesso e gerenciamento do banco de dados (*repositories*).

figura 21 - Estrutura da pasta “app”

Fonte: próprios autores

3.3.6 Sprint 06 - Implementação do primeiro requisito (US006)

Após a definição da arquitetura a ser utilizada e a configuração adequada do ambiente de desenvolvimento torna-se possível realizar a implementação da primeira funcionalidade do sistema que servirá como modelo base para o desenvolvimento dos demais requisitos. Assim, a seguir será explanado o processo de desenvolvimento do requisito manter usuário.

3.3.6.1 Primeira implementação *Back-End* (US006)

Ao iniciar o desenvolvimento da funcionalidade, primeiramente é necessário a criação da estrutura no banco de dados que armazenará as informações do requisito, para isso é criado então a migração através da ferramenta *Prisma*. A figura 22 retrata a criação da migração da funcionalidade manter usuário.

Figura 22 - Estrutura da tabela "*users*" no arquivo schema.prisma

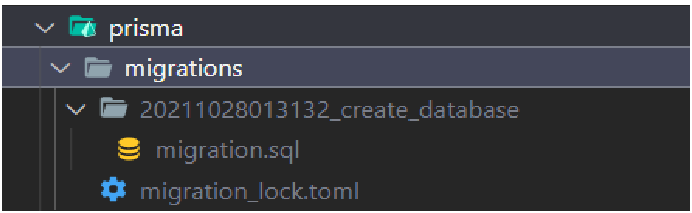
```
1  // This is your Prisma schema file,
2  // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4  datasource db {
5    provider = "postgresql"
6    url      = env("DATABASE_URL")
7  }
8
9  generator client {
10   provider = "prisma-client-js"
11 }
12
13 model User {
14   id           Int    @id @default(autoincrement())
15   email        String @unique
16   passwordHash String
17
18   createdAt DateTime @default(now())
19   updatedAt DateTime @updatedAt
20
21   Client Client[]
22
23   @@map("users")
24 }
```

Fonte: próprios autores

Todas as tabelas serão construídas no arquivo *schema.prisma* contido na pasta *prisma* na raiz do projeto, seguindo as instruções presentes na documentação da ferramenta.

Uma das vantagens da utilização da ferramenta *prisma*, é a facilidade na criação e gestão do banco de dados. Todas as tabelas podem ser criadas de forma simples, sendo especificado o nome da tabela, seus campos com seus respectivos tipos e por fim os relacionamentos caso existam. Após a inserção de todas as informações basta executar o comando responsável por criar as migrações, sua execução atualizará o banco de dados com a nova tabela e armazenará o registro da migração nos arquivos do projeto (figura 23).

Figura 23 - Estrutura da tabela *User* no arquivo *schema.prisma*



Fonte: próprios autores

Com a implementação da migração de usuário concluída, é criado então a sua classe modelo, utilizada no instanciamento de novos usuários na aplicação sempre que necessário (figura 22).

Figura 24 -Entidade usuário

```
1  class User {
2    id?: number;
3    email!: string;
4    passwordHash!: string;
5    password?: string;
6    oldPassword?: string;
7
8    private constructor({ email, passwordHash }: User) {
9      return Object.assign(this, {
10       passwordHash,
11       email,
12     });
13   }
14
15   static create({ email, passwordHash }: User): User {
16     const user = new User({ email, passwordHash });
17     return user;
18   }
19 }
20
21 export { User };
```

Fonte: próprios autores

Após a implementação dos arquivos necessários para o armazenamento dos dados contidos pelos usuários (classe modelo e migração), é iniciada a implementação de suas funcionalidades, seguindo a estratégia de desenvolvimento estabelecida pela metodologia TDD, onde o primeiro arquivo criado é responsável pelos testes de funcionalidades (figura 25).

Figura 25 - Teste da funcionalidade responsável por criar novo usuário

```
/**
 * @jest-environment ./prisma/prisma-environment-jest
 */

import request from 'supertest';
import App from '../../../app';
import { prisma } from '../../../database/client';

describe('UserController', () => {
  describe('Create', () => {
    beforeAll(async () => {
      await prisma.user.deleteMany({});
    });

    it('Should be able to create a new user', async () => {
      const response = await request(App).post('/users').send({
        email: 'user@email.com',
        password: '123456',
      });

      expect(response.status).toBe(200);
      expect(response.body).toHaveProperty('id');
    });

    it('Should not be able to create a new user with invalid fields', async () => {
      const response = await request(App).post('/users').send({
        email: 'email invalido teste',
        password: '123456',
      });

      expect(response.status).toBe(400);
    });
  });
});
```

Fonte: próprios autores

Com a execução do arquivo de teste implementado é normal que seja gerado uma mensagem de falha, isso pois, como a funcionalidade ainda não foi desenvolvida é impossível que a aplicação a execute de forma apropriada e retorne o valor desejado, desta forma é retornado uma mensagem de erro como demonstrado na figura 26 contida abaixo:

Figura 26 - Falha no teste de usuário

```
● UserController › Create › Should be able to create a new user

  expect(received).toBe(expected) // Object.is equality

  Expected: 200
  Received: 404

    19 |       });
    20 |
  > 21 |       expect(response.status).toBe(200);
       |                               ^
    22 |       expect(response.body).toHaveProperty('id');

    at src/app/modules/user/UserController.spec.ts:21:31
    at fulfilled (src/app/modules/user/UserController.spec.ts:8:58)

● UserController › Create › Should not be able to create a new user with invalid fields

  expect(received).toBe(expected) // Object.is equality

  Expected: 400
  Received: 404

    29 |       });
    30 |
  > 31 |       expect(response.status).toBe(400);
       |                               ^
    32 |     });
    33 |   });
    34 | });

    at src/app/modules/user/UserController.spec.ts:31:31
    at fulfilled (src/app/modules/user/UserController.spec.ts:8:58)

A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests
 leaking due to improper teardown. Try running with --detectOpenHandles to find leaks.
Test Suites: 1 failed, 1 total
Tests:       2 failed, 2 total
Snapshots:   0 total
Time:        17.529 s
Test run was interrupted.

Active Filters: filename /Userc/
 › Press c to clear filters.
```

Fonte: próprios autores

Após a criação do arquivo de teste é realizada a implementação da funcionalidade referente ao teste. Os primeiros arquivos criados são responsáveis pelo acesso ao banco de dados (*Models*), sendo eles a interface *IUsersRepositories* (figura 27) e a classe *PrismaUsersRepository* (figura 28). A interface *IUsersRepositories* é responsável por mapear todos os métodos que deverão estar presentes na classe *PrismaUsersRepository*, enquanto que a classe por sua vez faz a implementação dos métodos especificados.

Figura 27 - Primeira implementação da interface *IUsersRepositories*

```
import { User } from '../entities/User';

interface IUsersRepository {
  create(user: User): Promise<User>;
  existingEmail(email: string): Promise<boolean>;
}

export { IUsersRepository };
```

Fonte: próprios autores

Figura 28 - Primeira implementação da classe *PrismaUsersRepository*

```
import { prisma } from '../../../database/client';
import { User } from '../../entities/User';
import { IUsersRepository } from '../IUsersRepositories';

class PrismaUsersRepository implements IUsersRepository {
  async existingEmail(email: string): Promise<boolean> {
    const user = await prisma.user.findUnique({
      where: {
        email,
      },
    });

    return !!user;
  }

  async create({ email, passwordHash }: User): Promise<User> {
    const user = await prisma.user.create({
      data: {
        email,
        passwordHash,
      },
    });

    return user;
  }
}

export { PrismaUsersRepository };
```

Fonte: próprios autores

A princípio tanto a interface quanto a classe só implementam os métodos "*create*" e *existingEmail*", pois são os únicos necessários para a primeira funcionalidade do módulo de usuário (criação de usuário). A medida que novas funcionalidades sejam implementadas ao módulo, novos métodos serão criados e adicionados a esses arquivos.

Em seguida, será criado o arquivo *UserService* (*service*) responsável pelas regras de negócio do módulo. Nele serão implementadas todas as validações necessárias para o cadastro de usuário, como por exemplo a verificação se já existe um usuário cadastrado com o e-mail informado (figura 29).

Figura 29 - Primeira implementação da classe *UserService*

```
1  import bcrypt from 'bcryptjs';
2  import { CustomError } from '../../CustomError/CustomError';
3  import { User } from '../../entities/User';
4  import { IUsersRepository } from '../../repositories/IUsersRepositories';
5
6  interface IUserRequest {
7    id?: number;
8    email: string;
9    password: string;
10   oldPassword?: string;
11 }
12
13 class UserService {
14   constructor(private usersRepository: IUsersRepository) {}
15
16   public async create({ email, password }: IUserRequest) {
17     const userEmailAlreadyExists = await this.usersRepository.existingEmail(
18       email
19     );
20
21     if (userEmailAlreadyExists) {
22       throw new CustomError(401, { email: 'E-mail já cadastrado!' });
23     }
24
25     const passwordHash = await bcrypt.hash(password, 8);
26
27     const userCreate = User.create({ email, passwordHash });
28
29     const user = await this.usersRepository.create(userCreate);
30
31     return user;
32   }
33 }
34
35 export { UserService };
36
```

Fonte: próprios autores

Posteriormente será desenvolvido o arquivo responsável pelas requisições ao módulo (*Controller*), verificando se os dados necessários para seu funcionamento foram devidamente informados pelo cliente. A figura 30 demonstra a implementação da classe *UserController*.

Figura 30 - Primeira implementação da classe *UserController*

```
1  import { NextFunction, Request, Response } from 'express';
2  import * as Yup from 'yup';
3  import AbstractController from '../../../core/AbstractController';
4  import { UserService } from './UserService';
5
6  interface AuthRequest extends Request {
7    userId?: number;
8    error?: {
9      email?: string;
10     password?: string;
11   };
12 }
13
14 class UserController extends AbstractController {
15   constructor(private userService: UserService) {
16     super();
17   }
18
19   public async create(
20     request: AuthRequest,
21     response: Response,
22     next: NextFunction
23   ): Promise<Response | void> {
24     const schema = Yup.object().shape({
25       email: Yup.string().email().required(),
26       password: Yup.string().required().min(6),
27     });
28
29     await this.validateData(request, response, schema);
30
31     const { email, password } = request.body;
32
33     try {
34       const user = await this.userService.create({ email, password });
35       return response.json(user);
36     } catch (err) {
37       return next(err);
38     }
39   }
40 }
41
42 export { UserController };
43
```

Fonte: próprios autores

Por fim, para que a primeira funcionalidade do módulo seja finalizada é necessário a criação das rotas de acesso, às quais serão utilizadas pelos usuário do sistema através do sistema de requisições HTTP. A criação das rotas segue dois passos, sendo o primeiro passo a criação do arquivo *UserFactory* (figura 31). Esse arquivo possui a responsabilidade de unir as três camadas criadas para o módulo anteriormente (*model, service* e *controller*) em apenas um único objeto utilizado nas rotas.

Figur 31 - Estrutura do arquivo *UserFactory*

```
1  import { PrismaUsersRepository } from '../../repositories/prisma/PrismaUsersRepository';
2  import { UserController } from './UserController';
3  import { UserService } from './UserService';
4
5  export const userFactory = () ⇒ {
6    const usersRepository = new PrismaUsersRepository();
7    const userService = new UserService(usersRepository);
8    const userController = new UserController(userService);
9    return userController;
10 };
```

Fonte: próprios autores

O segundo passo é a criação do arquivo de rotas para o módulo de usuário (figura 32) utilizando a biblioteca *express* e adicioná-lo ao método "*routes*" contido no arquivo app.ts (figura 33).

Figura 32 - Arquivo de rotas do módulo de usuário

```
1  import { Router } from 'express';
2  import { userFactory } from '../app/modules/user/UserFactory';
3
4  const routes = Router();
5
6  routes.post('', (request, response, next) ⇒
7    userFactory().create(request, response, next)
8  );
9
10 export default routes;
```

Fonte: próprios autores

Figura 33 -Disponibilizando as rotas de usuário para aplicação.

```
1
2  private routes(): void {
3    this.express.use('/users', UserRoutes);
4  }
```

Fonte: próprios autores

Após a implementação das rotas é realizado um novo teste com o arquivo de testes criado anteriormente, o resultado pode ser analisado na figura 34 logo abaixo:

Figura 34 -Teste do módulo de usuário aprovado.

```
schemas code_schema_8e8afa64-ce93-4378-98e7-cfd70e9e3f3b
 PASS  src/app/modules/user/UserController.spec.ts (15.201 s)
  UserController
    Create
      √ Should be able to create a new user (101 ms)
      √ Should not be able to create a new user with invalid fields (10 ms)

A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests
 leaking due to improper teardown. Try running with --detectOpenHandles to find leaks.
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        16.317 s, estimated 17 s
Ran all test suites matching /Userc/i.

Watch Usage: Press w to show more.
```

Fonte: próprios autores

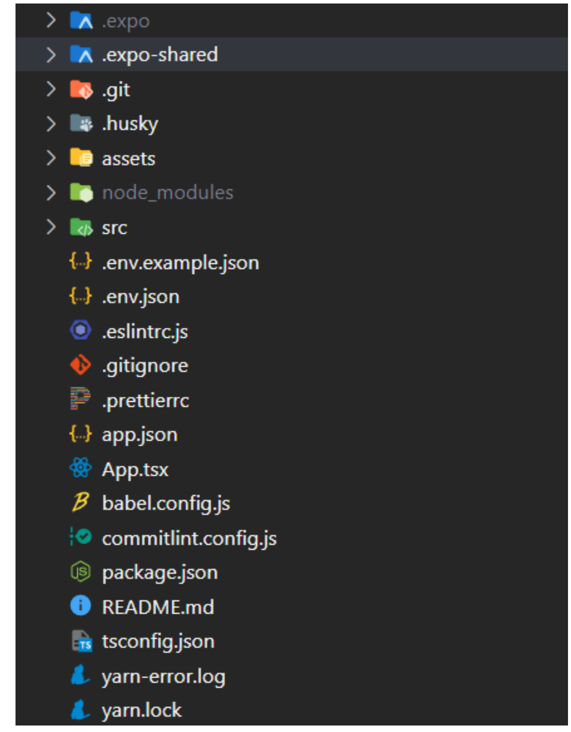
Como a funcionalidade testada foi implementada corretamente é esperado que o teste seja aprovado e a funcionalidade seja classificada como finalizada, encerrando-se o ciclo de desenvolvimento de uma das funcionalidades presentes no módulo do usuário presente no *back-end*.

3.3.6.2 Configuração da estrutura de pastas e arquivos do *Front-End* (*mobile*)

Com a finalização da implementação da primeira funcionalidade de usuário no *back-end* da aplicação, inicia-se então o desenvolvimento da estrutura *front-end* responsável pelo cadastro de usuário, começando pela criação inicial do projeto utilizando a ferramenta *Expo*.

Assim como configurado no *back-end*, o *front-end* (aplicativo) também segue uma estrutura de pastas e arquivos. A seguir será apresentado uma breve explicação sobre a estrutura contida no *front-end*, começando pela pasta raiz do projeto (figura 35).

figura 35 - Estrutura de pastas do *front-end*



Fonte: próprios autores

Primeiramente, na raiz do projeto existirá todos os arquivos de configuração, assim como os arquivos do *Eslint* e *Commitlint* configurados anteriormente. Também contém as pastas "*assets*" e "src" . A pasta "*assets*" armazenará os arquivos relacionados à imagem do aplicativo, tais como sua logo e sua *Splash Screen* (tela de carregamento).

A pasta "src" por sua vez conterá todo o código relacionado a aplicação do projeto (figura 36). Nela haverá as principais interfaces do sistema responsáveis por modelar o formato dos dados trabalhados (*@types*), os arquivos mídia (*assets*), componentes que são utilizados em mais de uma parte do sistema (*components*), as telas do aplicativo (*screens*), os arquivos responsáveis pelo acesso a API (*services*), o contexto do aplicativo que permite a troca de informações (*store*) e por fim os estilos globais (*styles*) e as rotas (*routes*.tsx).

figura 36 - Estrutura de pastas do front-end

Fonte: próprios autores

3.3.6.3 Primeira implementação *Front-end* (US006)

Com os arquivos e pastas devidamente configurados é iniciado então a implementação da funcionalidade de cadastro de usuário no *front-end*. O primeiro passo é a identificação e codificação dos componentes que constituem a tela e poderiam ser reaproveitados em outros locais, como por exemplo os campos de inserção de dados (Figura 37) e o botão de cadastro (Figura 38).

Figura 37 - Componente de input

```
const InputTextLogin: ForwardRefRenderFunction<unknown, IProps> = (
  props: IProps,
  ref?: React.Ref<any>
) => {
  const { label, error, value, Icon, onChangeText, onIconPress, ...rest } =
    props;

  return (
    <View>
      {label ? <Text style={styles.label}>{label}</Text> : null}

      <View
        style={[
          styles.inputBox,
          !!error && { borderWidth: 1, borderColor: 'red' },
        ]}
      >
        <TextInput
          style={styles.input}
          value={value}
          autoCorrect={false}
          onChangeText={onChangeText}
          {...rest}
          ref={ref}
          placeholderTextColor={theme.colors.primary60}
        />

        {Icon ? (
          onIconPress ? (
            <TouchableOpacity onPress={onIconPress}>{Icon}</TouchableOpacity>
          ) : (
            Icon
          )
        ) : null}
      </View>
      {error && <Text style={styles.errorText}>{error}</Text>}
    </View>
  );
};
```

Fonte: próprios autores

Figura 38 - Componente de botão

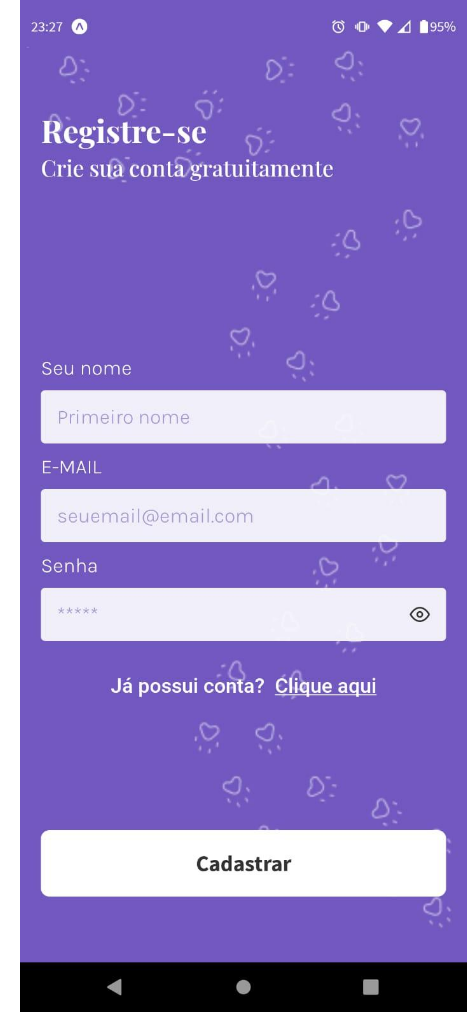
```
const ButtonRequest: React.FC<IProps> = ({
  title,
  loading,
  backgroundColor,
  color,
  styleCustom,
  ...rest
}) => {
  return (
    <View>
      <TouchableOpacity
        activeOpacity={0.5}
        style={[
          styles.button,
          !!backgroundColor && { backgroundColor },
          styleCustom,
        ]}
        {...rest}
      >
        {!loading ? (
          <Text style={[styles.buttonText, !!color && { color }]}>{title}</Text>
        ) : (
          <ActivityIndicator
            color={color || theme.colors.secondary100}
            size={theme.fontSizes.titleBig}
          />
        )}
      </TouchableOpacity>
    </View>
  );
};
```

Fonte: próprios autores

A criação da tela responsável pela funcionalidade é realizada através da junção dos recursos disponíveis pelo *React Native* juntamente com os componentes criados anteriormente.

Por fim é realizado testes manuais na funcionalidade a fim de garantir o correto funcionamento e a qualidade da mesma, concluindo o desenvolvimento da funcionalidade. A figura 39 demonstra o resultado da tela de registro de usuário.

Figura 39 - Tela de registro de usuário



Fonte: próprios autores

3.3.7 *Sprint* 07 – Desenvolvimento US005 e US010.

A *Sprint* 07 deu-se início na reunião de planejamento na qual foram selecionados os seguintes requisitos para o desenvolvimento, de acordo com as suas prioridades: Realizar

Login (US005) e Manter Dados do Cliente (US010). Posteriormente foi realizada a revisão da *Sprint* juntamente com os *stakeholders*. A *Sprint* fechou com a retrospectiva.

3.3.8 *Sprint* 08 – Desenvolvimento US002, US008 e US011.

A *Sprint* 08 deu-se início na reunião de planejamento na qual foram selecionados os seguintes requisitos para o desenvolvimento, de acordo com as suas prioridades: Manter Dados do Pet (US002), Visualizar Serviço Disponível (US008) e Manter Serviços (US011). Posteriormente foi realizada a revisão da *Sprint* juntamente com os *stakeholders*. A *Sprint* fechou com a retrospectiva.

3.3.9 Sprint 09 – Desenvolvimento US003, US012 e US013.

A *Sprint* 09 deu-se início na reunião de planejamento na qual foram selecionados os seguintes requisitos para o desenvolvimento, de acordo com as suas prioridades: Gerenciar Solicitações de Agendamento de Serviços (US003), Solicitar Agendamento de Serviço (US012) e Cancelar Agendamento de Serviço (US013). Posteriormente foi realizada a revisão da *Sprint* juntamente com os *stakeholders*. A *Sprint* fechou com a retrospectiva.

4. RESULTADOS

Com a construção do projeto proposto foi possível atingir diversos objetivos estabelecidos, começando pelo desenvolvimento dos documentos referentes aos requisitos do sistema, tais como o Documento de Visão (APÊNDICE A), Diagrama de Casos de Uso (APÊNDICE B) e Especificação dos Casos de Uso (APÊNDICE C). Neles estão presentes as principais informações que regem o desenvolvimento do sistema proposto.

Após a finalização dos documentos responsáveis por mapear os requisitos do sistema, foi então produzido o Diagrama Entidade Relacionamento (APÊNDICE E) juntamente com a escolha da arquitetura a ser utilizada (APÊNDICE D). Respectivamente, esses arquivos visam informar à equipe de desenvolvimento quais dados o sistema deverá armazenar e quais ferramentas e camadas deverão ser aplicadas para que haja o gerenciamento de tais dados.

Com os documentos de requisitos desenvolvidos e com a arquitetura e banco montados foi possível então realizar o desenvolvimento das principais funcionalidades do sistema. Dentre as funcionalidades destacam-se: Registrar-se (US006), Efetuar Login (US005), Manter Dados do Cliente (US010), Manter Dados do Pet (US011), Solicitar Agendamento de Serviço (US012), Cancelar Agendamento de Serviço (US013), Manter Serviços (US002), Gerenciar Solicitações de Agendamento de Serviços (US003) e Visualizar Serviço Disponível (US008). Todas as telas das funcionalidades desenvolvidas estão presentes no apêndice F.

Por fim, é válido ressaltar o crescimento profissional gerado pelo projeto para seus autores, através da aplicação de conhecimentos adquiridos ao longo da graduação.

5. CONCLUSÃO E CONSIDERAÇÕES FINAIS

A realização deste projeto foi de extrema relevância para o crescimento profissional de seus autores. Através da resolução do problema proposto, foi possível aplicar diversas técnicas e conhecimentos adquiridos ao longo da graduação, nos processos de pesquisa e levantamento de requisitos, na escolha e utilização das ferramentas, na definição e utilização de metodologias e processos, e em diversos outros cenários envolvidos no projeto.

Alguns pontos podem ser citados como destaque para essa pesquisa, sendo eles as estratégias adotadas para o versionamento, gestão e desenvolvimento do projeto.

Começando pela estratégia de versionamento, sua utilização proporcionou maior eficiência no trabalho em dupla dos autores durante o desenvolvimento dos artefatos. Graças a sua aplicação não houve erros relacionados a sobreposição de versões ou perda de arquivos os quais poderiam afetar negativamente o cronograma.

A estratégia de gestão por sua vez, utilizando o *Scrum*, possibilitou maior controle sobre o desenvolvimento das atividades necessárias para a conclusão do projeto. Com a realização das *Sprints* foi possível desenvolver de forma rápida e eficiente os incrementos necessários para gerar valor ao sistema.

Por fim, a estratégia de desenvolvimento utilizando o TDD que nos deu a garantia de que um código bem definido precisa ter os principais cenários com suas regras de negócio asseguradas de futuras alterações no código, outro ganho com a utilização do TDD está relacionado ao próprio desenvolvimento, dando uma visão melhor de suas funcionalidades.

É válido ressaltar que graças ao poder na linguagem *Javascript* juntamente com seu super set, o *Typescript*, tivemos um ganho na produtividade por trabalhar com três ambientes diferentes e aproveitando os conhecimentos entre eles, facilitando no desenvolvimento entre os ambientes, principalmente entre *web* e *mobile* por utilizar *frameworks* semelhantes fornecidos pelo *facebook*, *React* e *React Native*.

REFERÊNCIAS

ALMEIDA, S. L. F. *et al*. **Aplicação e análise de processos de desenvolvimento de software: um estudo de caso no GPES-IFPB.** Principia: Divulgação científica e tecnológica do IFPB, João Pessoa, n. 43, p. 152-165, abr./2018.

ANICHE, M. F. **Como a prática de TDD influencia o projeto de classes em sistemas orientados a objetos.** 2012

BAYLÃO, Andre Luis da Silva; OLIVEIRA, Victor Miranda. **Impacto da evolução tecnológica na gestão empresarial.** *In*: SEGeT – Simpósio de Excelência em Gestão e Tecnologia, XII, 2015, Resende - RJ, Associação Educacional Dom Bosco, 2012**.** Disponível em: <https://www.aedb.br/seget/arquivos/artigos15/14922205.pdf>. Acesso em: 02 maio 2021.

CARVALHO, Fabiano et al. **Desenvolvimento web para dispositivos móveis com Grails**. Revista de trabalhos acadêmicos-campus Niterói. 2014.

CHACON, Scott; STRAUB, Ben; **Pro Git**: Everything you need to know about git. 2. ed. [S.l.]: apress, 2014. p. 1-516.

FACEBOOK OPEN SOURCE. Jest, 2021a. **Jest é um poderoso Framework de Testes em JavaScript com um foco na simplicidade.** Disponível em: <https://jestjs.io/pt-BR />. Acesso em: 24 mar. 2021

FACEBOOK OPEN SOURCE. React, 2021b. **Uma biblioteca JavaScript para criar interfaces de usuário**. Disponível em: <https://pt-br.reactjs.org/>. Acesso em: 25 abr. 2021.

FACEBOOK OPEN SOURCE. React Native, 2021c. **Crie aplicativos nativos para Android e iOS usando React.** Disponível em: < https://reactnative.dev/>. Acesso em: 25 abr. 2021.

FIGMA. figma, 2021. **Figma helps teams create, test, and ship better designs from start to finish.** Disponível em: <https://www.figma.com/>. Acesso em: 21 mar. 2021.

FOLHA DE SÃO PAULO. **Faturamento do mercado pet no país aumenta 13,5% em 2020**. Disponível em: <https://www1.folha.uol.com.br/mpme/2021/03/faturamento-do-mercado-pet-no-pais-aumenta-135-em-2020.shtml>. Acesso em: 2 mai. 2021.

GIT. Git, 2021. **Sistema de controle de versão distribuído gratuito e de código aberto.** Disponível em: <https://git-scm.com/>. Acesso em: 28 mar. 2021.

GITHUB. GitHub, 2021. **Where the world builds software.** Disponível em: <https://github.com//>. Acesso em: 28 mar. 2021.

HEUSER, C.A. **Projeto de banco de dados**. 6. ed. Porto Alegre: Bookman, 2009.

HIMARA, K. **Engenharia de Software: Qualidade e Produtividade com Tecnologia**. Grupo GEN, 2011. 9788595155404. Disponível em:

<https://integrada.minhabiblioteca.com.br/#/books/9788595155404/>. Acesso em: 25 Apr 2021.

IEEE Standard Glossary of Software Engineering Terminology .IEEEStd 610.12-1990, vol., no., pp.1-84, 31 Dec. 1990, doi: 10.1109/IEEESTD.1990.101064.

IBGE – INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA. População de Animais de Estimação no Brasil. **Governo Federal – Governo do Brasil**. 2013. Disponível em: <https://www.gov.br/agricultura/pt-br/assuntos/camaras-setoriais-tematicas/ documentos/cama ras-tematicas/insumos-agropecuarios/anos-anteriores/ibge-populacao -de-animais-de-estimaca o-no-brasil-2013-abinpet-79.pdf/view>. Acesso em: 19 fev. 2021.

IPB, **Instituto Pet Brasil.** 2018. Disponível em: <http://institutopetbrasil.com>. Acesso em: 28 mar. 2021.

KORTH, H.F; SILBERSCHATZ, A; SUDARSHAN, S. **Sistema de Banco de Dados**. Grupo GEN,2020.9788595157552. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788595157552/>. Acesso em: 12 maio 2021.

LAZAR, Jonathan. **User-centered web development.** Jones and Bartlet Publishers, Inc. 2001

LIMA, B. R. *et al*. **Inovação no mercado de pet shops**: subtítulo do artigo. **RAI**: Revista de Administração e Inovação, São Paulo, v. 10, n. 1, p. 6-26, mar./2013. Disponível em: <https://www.redalyc.org/pdf/973/97325715001.pdf>. Acesso em: 2 mai. 2021.

LOUDON, Kyle. **Desenvolvimento de grandes aplicações web**. Novatec Editora Ltda.2010. Disponível em: <https://telematicafractal.com.br/revista/index.php/telfract/article/view/9/4>. Acesso em: 27 mar. 2021.

MALDONADO, J. et al. **Automatização de Teste de Software com Ferramentas de Software Livre.** Grupo GEN, 2018. 9788595155305. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788595155305/>. Acesso em: 25 Apr 2021.

MARTIN, Robert C.; **Código limpo**: habilidades práticas do Agile Software. 1. ed.[S.l.]: Alta Books, 2009. p. 1-456.

MICROSOFT. Visual Studio Code, 2021. **Code editor redefined and optimized for building and debugging modern web and cloud applications.** Disponível em: <https://code.visualstudio.com/docs>. Acesso em: 21 mar. 2021.

MICROSOFT. Typescriptlang, 2021. **WhatisTypeScript?**. Disponível em: <https://www.typescriptlang.org/>. Acesso em: 1 abr. 2021.

MOZILLA. MDN Web Docs, 2021a. **Recursos para desenvolvedores, por desenvolvedores**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML>. Acesso em: 28 mar. 2021.

MOZILLA. MDN Web Docs, 2021b. **Recursos para desenvolvedores, por desenvolvedores**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/CSS>. Acesso em: 05 abr. 2021.

MOZILLA. MDN Web Docs, 2021c. **Recursos para desenvolvedores, por desenvolvedores**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 13 abr. 2021.

OPENJS FOUNDATION. ESLint, 2021a. **Find and fix problems in your JavaScript code.** Disponível em: <https://eslint.org/>. Acesso em: 30, abr. 2021.

OPENJS FOUNDATION. Node.js, 2021b. **Um runtime JavaScript desenvolvido com Chrome's v8 JavaScript engine.** Disponível em: <https://nodejs.org/pt-br/>. Acesso em: 25 abr. 2021.

PÁDUA, P.F.W. D. **Engenharia de Software - Projetos e Processos** - Vol. 2. Grupo GEN, 2019. 9788521636748. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788521636748/>. Acesso em: 17 mai. 2021.

PEREIRA, Caio. **Aplicações web real-time com Node.js.** Casa do Código, abr. 2014

PONTES, T. B.; ARTHAUD, D. D. B. **Metodologias ágeis para o desenvolvimento de softwares**. Ciência e Sustentabilidade, v. 4, n. 2, 2019.

PRESSMAN,R. ;MAXIM, B. **Engenharia de Software**: Uma Abordagem Profissional. 8. Ed. Porto Alegre: McGraw-Hill, 2016.

PRATES, Gláucia Aparecida; OSPINA, Marco Túlio. Tecnologia da informação em pequenas empresas: fatores de êxito, restrições e benefícios. **RAC**, v. 8, n. 2, p. 9-26, jun./2004.

RUSSI, Davi Felipe; CHARÃO, Andrea Schwertner. **Ambientes de Desenvolvimento Integrado no Apoio ao Ensino da Linguagem de Programação Haskell**. RENOTE: Novas Tecnologias na Educação, Porto Alegre, v. 9, n. 2, dez./2011. Disponível em: <https://seer.ufrgs.br/renote/article/view/25077/14767>. Acesso em: 30 abr. 2021.

SANTIAGO ,Hewerton Luís P.; CARDOSO, Nicanor Davi. **Desenvolvimento do Pet-Software**: sistema para controle de vacinas e agendas de Pet-Shop. [entre 2015 e 2021] Disponível em: <http://www.atenas.edu.br/uniatenas/assets/files/magazines/DESENVOLVIMENTO_DO_PET_SOFTWARE__sistema_para_controle_de_vacinas_e_agendas_de_Pet_Shop_.pdf>. Acesso em: 20 fev. 2021.

SANTOS, P. V. S. et al. **Um estudo acerca da sobrevivência de micro e pequenas empresas (mpes)**. In: Simpósio de Engenharia de Produção da Região Nordeste (SEPRONE) & Simpósio de Engenharia de Produção do Vale do São Francisco (SEPVASF) - Juazeiro-BA, 2018. Disponível em: <https://www.doity.com.br/anais/seprone/trabalho/43605>. Acesso em: 16 de mai. de 2021.

SOARES, Michel dos Santos. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. 2004. Disponível em:

<http://infocomp.dcc.ufla.br/index.php/INFOCOMP/article/view/68>. Acesso em: 04 mai. 2021.

SOMMERVILLE, Ian; **Engenharia de Software**: subtítulo do livro. 9. ed. São Paulo: Pearson Education, 2011.

SUTHERLAND, Meet Jeff; SCHWABER, Meet Ken. **The Scrum Guide:** The Definitive Guide to Scrum. 2020. Disponível em <https://scrumguides.org/scrum-guide.html >. Acesso em: 30 abr. 2021.

TAVARRES, H. L. Introdução ao desenvolvimento de aplicações híbridas. v. 6 n. 1 (2016):RevistaEletrônicae-F@tec. Disponível em: <https://fatecgarca.edu.br/ojs/index.php/ efatec/article/view/113>Acesso em: 30 abr. 2021.

THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL, 2021. **What Is PostgreSQL?** Disponível em: <https://www.postgresql.org/docs/>. Acesso em: 30, abr. 2021.

TRELLO. Trello, 2021. **Colabore, gerencie projetos e alcance novos picos de produtividade.** Disponível em: <https://trello.com/pt-BR>. Acesso em: 21 mar. 2021.

WHIMSICAL. **Whimsical:** Think together**.** Disponível em: <https://whimsical.com/>. Acesso em: 28 mar. 2021.

APÊNDICE A - Documento de visão

Documento de Visão

Pet Shop

Versão 0.3

Autores:

MATHEUS VIEIRA GONÇALVES

JÚNIOR AUGUSTO XAVIER ESBALTAR

Histórico da Revisão

Versão	Data	Descrição	Autor
0.1	20/04/2021	Criação do documento inserindo as principais informações relacionadas ao projeto.	Matheus Vieira e Júnior Augusto
0.2	03/05/2021	Adicionando e modificando requisitos funcionais	Matheus Vieira e Júnior Augusto
0.3	15/05/2021	Remodelando a escrita do problema e escopo do produto	Matheus Vieira e Júnior Augusto
0.4	17/06/2021	Correções solicitadas pela banca de TCC	Júnior Augusto e Matheus Vieira
0.5	06/09/2021	Alteração na tabela de siglas	Júnior Augusto e Matheus Vieira
0.6	07/09/2021	Correção ortográfica	Júnior Augusto e Matheus Vieira
0.7	14/10/2021	Modificação nas prioridades dos requisitos	Júnior Augusto e Matheus Vieira

Sumário

1. Introdução....4
2. Problema....4
3. Escopo do Projeto....4
4. Escopo do Produto....4
5. Abreviaturas, convenções e siglas....4
5.1. Prioridade dos requisitos....5
6. Stakeholders....5
6.1. Equipe....6
6.2. Usuários do sistema....6
7. Requisitos do sistema....6
7.1. Requisitos Funcionais....6
7.2 Requisitos Não Funcionais....8
8. Restrições gerais....9
9. Arquitetura de Software....9
9.1 Metas e Restrições da Arquitetura....9
9.1.1 Segurança....9
9.1.2. Estrutura de transferência de dados....10
9.1.3 Estrutura de código....10
9.1.4 Linguagens e paradigmas....10
9.2 Visão Lógica....10

1. Introdução

O presente documento tem por objetivo expor as principais informações referentes ao projeto. Assim será apresentado uma breve descrição do problema tratado e posteriormente será apresentada a solução proposta. Para a solução, será abordado suas principais necessidades e funcionalidades, apresentando também os principais envolvidos no projeto e suas devidas atuações.

2. Problema

O problema de dificuldade no gerenciamento manual dos serviços, produtos e clientes de um Pet Shop, **afeta** funcionários e clientes do estabelecimento. **Seu impacto** inclui o mau gerenciamento das informações pertinentes ao pet shop, dificultando a realização de vendas e prestação de serviços, gerando insatisfação aos clientes, perda de informações e principalmente perda nos lucros. **Uma boa solução** incluiria a utilização de um sistema de software que possibilitasse aos funcionários do pet shop um melhor controle sobre as informações do estabelecimento e uma melhor comunicação com os seus clientes.

3. Escopo do Projeto

Este trabalho tem por objetivo o desenvolvimento de um sistema de software que auxilie Pets Shops a realizarem o controle de gestão do estabelecimento, melhorando o controle sobre os produtos e serviços oferecidos, automatizando o processo de atendimento ao cliente.

4. Escopo do Produto

Para clientes e funcionários de um pet shop, **que necessitam** realizar as atividades de compra, venda e gestão dos serviços e produtos oferecidos pelo estabelecimento, o produto desenvolvido **será um** sistema de software constituído por uma plataforma *web* e um aplicativo *mobile*. Sua utilização **possibilita** que clientes realizem solicitações de produtos e serviços ao pet shop, e permite aos funcionários e gestores o controle sobre os serviços e produtos solicitados. **Diferente** do sistema manual de gerenciamento, **nosso produto** proporcionará aos usuários agilidade nas atividades de solicitação e gestão dos produtos e serviços oferecidos pelo pet shop.

5. Abreviaturas, convenções e siglas

Todas as abreviaturas, convenções e siglas existentes nos documentos do projeto serão descritas abaixo na tabela:

Sigla	Descrição
Manter	Desempenha as seguintes ações dentro do sistema: Inserir, Alterar, Listar, Visualizar, Ativar e Inativar.
Pets	Animais de estimação.
RF	Requisito Funcional
RNF	Requisito não funcional
US	História de Usuário
CA	Critério de Aceitação
UC	Caso de Uso

5.1. Prioridade dos requisitos

As prioridades dos requisitos foram divididas em quatro categorias, definidas através da utilização do método **MoSCoW,** onde:

- **Essencial (Must):** Refere-se aos requisitos que necessitam ser implementados prioritariamente, sem eles não há o funcionamento do sistema;
- **Importante (Should):** Refere-se aos requisitos que deveriam ser implementados, entretanto o sistema pode funcionar sem eles, ainda que de forma não satisfatória.
- **Poderia (Could):** Refere-se aos requisitos que poderiam ser implementados, entretanto, mesmo sem eles, o sistema pode funcionar de forma satisfatória;
- **Interessante (Want):** Refere-se aos requisitos que seriam interessantes para o sistema, mas a princípio não serão implementados.

6. Stakeholders

As tabelas abaixo descrevem os stakeholders envolvidos no projeto. A primeira tabela faz uma listagem de todos os membros da equipe, informando seu papel e uma descrição de suas ações. A segunda tabela faz uma listagem dos usuários do sistema e informa uma breve descrição sobre suas possíveis ações.

6.1. Equipe

Papel	Responsável	Descrição
Parte do time de desenvolvimento	Matheus Vieira e Júnior Augusto	Responsável pela documentação e implementação das funcionalidades do sistema.
Product Owner	Matheus Vieira	Responsável por definir e priorizar o backlog do projeto, informando dessa forma o que será ou não abordado.
Scrum Master	Júnior Augusto	Responsável por gerenciar as atividades e os prazos trabalhados dentro da sprint, garantido a utilização das boas práticas do Scrum.

6.2. Usuários do sistema

Papel	Descrição
Funcionário do pet shop	Possui acesso ao sistema para realizar o gerenciamento das informações contidas no Pet Shop: produtos, serviços, clientes, pets e agendamentos.
Cliente do pet shop	Possui acesso ao sistema para realizar a solicitação e o acompanhamento de compra de produtos ou agendamentos de serviços.

7. Requisitos do sistema

Abaixo, será apresentado as principais funcionalidades do sistema e suas devidas restrições para o seu correto funcionamento.

7.1. Requisitos Funcionais

7.1.1. Manter funcionário

Código	Nome	Descrição	Prioridade
RF01	Manter funcionário	Permite ao ator realizar todas as ações de manter funcionários do estabelecimento	Essencial

7.1.2. Manter cliente

Código	Nome	Descrição	Prioridade
RF02	Manter Cliente	Permite ao ator realizar todas as ações de manter clientes do estabelecimento	Essencial

7.1.3. Autenticar usuário

Código	Nome	Descrição	Prioridade
RF03	Autenticar Usuário	O sistema deve permitir que apenas os clientes e funcionários cadastrados e devidamente autenticados realizem o acesso as funcionalidades do sistema.	Essencial

7.1.4. Manter produto

Código	Nome	Descrição	Prioridade
RF04	Manter Produto	Permite ao ator realizar todas as ações de manter os produtos do estabelecimento	Interessante

7.1.5. Manter serviço

Código	Nome	Descrição	Prioridade
RF05	Manter Serviço	Permite ao ator realizar todas as ações de manter os serviços do estabelecimento	Essencial

7.1.6. Manter pets do cliente

Código	Nome	Descrição	Prioridade
RF06	Manter pets do cliente	Permite ao ator realizar todas as ações de manter os pets do cliente cadastrado no sistema	Importante

7.1.7. Abrir solicitação de agendamento de serviço

Código	Nome	Descrição	Prioridade
RF07	Abrir solicitação de agendamento de serviço	O sistema deve permitir que o cliente solicite um serviço para seu pet	Importante

7.1.8. Cancelar agendamento de serviço

Código	Nome	Descrição	Prioridade
RF08	Cancelar agendamento de serviço	O sistema deve permitir que o cliente cancele o serviço.	Importante

7.1.9. Gerenciar solicitações de agendamento de serviço

Código	Nome	Descrição	Prioridade
RF09	Gerenciar solicitações de	O sistema deve permitir que o funcionário aceite ou rejeite as solicitações de serviços pendentes. Também deve	Importante

	agendamento de serviço	permitir o cancelamento de uma solicitação já aceita.	

7.1.10 Emitir relatórios de serviços e/ou produtos

Código	Nome	Descrição	Prioridade
RF10	Emitir relatórios de serviços e/ou produtos	O sistema deve permitir que o ator emita relatórios de serviços prestados e/ou produtos vendidos em um determinado período de tempo.	Interessante

7.2 Requisitos Não Funcionais

01 Interoperabilidade

Código	Descrição
RNF01.01	O sistema deverá ter uma comunicação estável com o banco de dados.

02 Segurança

Código	Descrição
RNF02.01	As senhas dos usuários devem conter no mínimo 6 caracteres, podendo conter letras maiúsculas, letras minúsculas, números e caracteres especiais.

Código	Descrição
RNF02.02	Um e-mail não poderá estar cadastrado em duas ou mais contas.

Código	Descrição
RNF02.03	O sistema deve permitir que apenas funcionários autorizados gerenciem as informações de produtos e serviços no sistema.

Código	Descrição
RNF02.04	O acesso as funcionalidades do sistema só serão permitidas à usuários devidamente cadastrados e autenticados.

03 Confiabilidade

Código	Descrição
RNF03.01	O sistema deve estar disponível 24 horas por dia e sete dias por semana, exceto em dias de manutenção agendadas e avisadas previamente aos usuários

04 Usabilidade

Código	Descrição

RNF04.01	O sistema deverá ser intuitivo, utilizando-se de elementos visuais como ícones e imagens capazes de auxiliar no entendimento das funcionalidades contidas no sistema.

05 Portabilidade

Código	Descrição
RNF05.01	O sistema deverá ser responsivo, adaptando-se para os diferentes tamanhos de tela.

Código	Descrição
RNF05.02	O aplicativo mobile deverá estar disponível para os sistemas operacionais Android e IOS.

8. Restrições gerais

- O sistema só funcionará mediante uma conexão de rede estável e será mantido em um servidor Node.js.
- As telas do aplicativo *mobile* deverão ser desenvolvidas utilizando a tecnologia React Native.
- As telas do sistema *web*, deverão ser desenvolvidas utilizando a tecnologia ReactJS.
- O banco de dados utilizado será o PostgreSQL
- A documentação do sistema será escrita em sua maioria na língua portuguesa, restringindo o entendimento da documentação apenas para pessoas com conhecimento na língua, salvo partes mais técnicas relacionadas diretamente ao código da aplicação.

9. Arquitetura de Software

9.1 Metas e Restrições da Arquitetura

9.1.1 Segurança

Com exceção das rotas de acesso públicas destinadas para a apresentação dos serviços e produtos oferecidos pelo pet shop ao cliente, todo o acesso do aplicativo e site será restrito a somente usuários cadastrados no sistema, sejam eles funcionários ou clientes.

Qualquer pessoa pode se cadastrar de forma independente como cliente, mas somente o administrador do sistema será capaz de cadastrar novos funcionários.

O sistema de acesso será desenvolvido seguindo o padrão JSON Web Token (JWT), o qual permite que o usuário acesse o sistema através de uma chave (token) criptografada que é gerada durante o processo de login e possui prazo de validade, afim de aumentar a segurança.

9.1.2. Estrutura de transferência de dados

O acesso e manipulação dos dados contido no servidor será modelado através do padrão de arquitetura *model–view–controller* (MVC), o qual divide a aplicação em camadas. Dessa forma, o sistema será dividido da seguinte maneira:

- **Model**: Responsável pelo acesso ao banco de dados da aplicação;
- **View**: Camada de interação com o usuário. Responsável pela exibição das informações;
- **Controller**: Responsável por tratar as requisições de dados realizadas pelo usuário, aplicar as regras de negócio e realizar as tratativas de dados.

9.1.3 Estrutura de código

Para o desenvolvimento do código serão seguidos os padrões de arquitetura limpa, descritos no livro "Código Limpo", escrito por Robert Cecil Martin.

A fim de manter a mesma estrutura de código entre todos os integrantes da equipe de desenvolvimento, durante a codificação será obrigatório a utilização do Visual Studio Code, devidamente configurado com as extensões: Eslint, EditorConfig for VS code e Prettier.

9.1.4 Linguagens e paradigmas

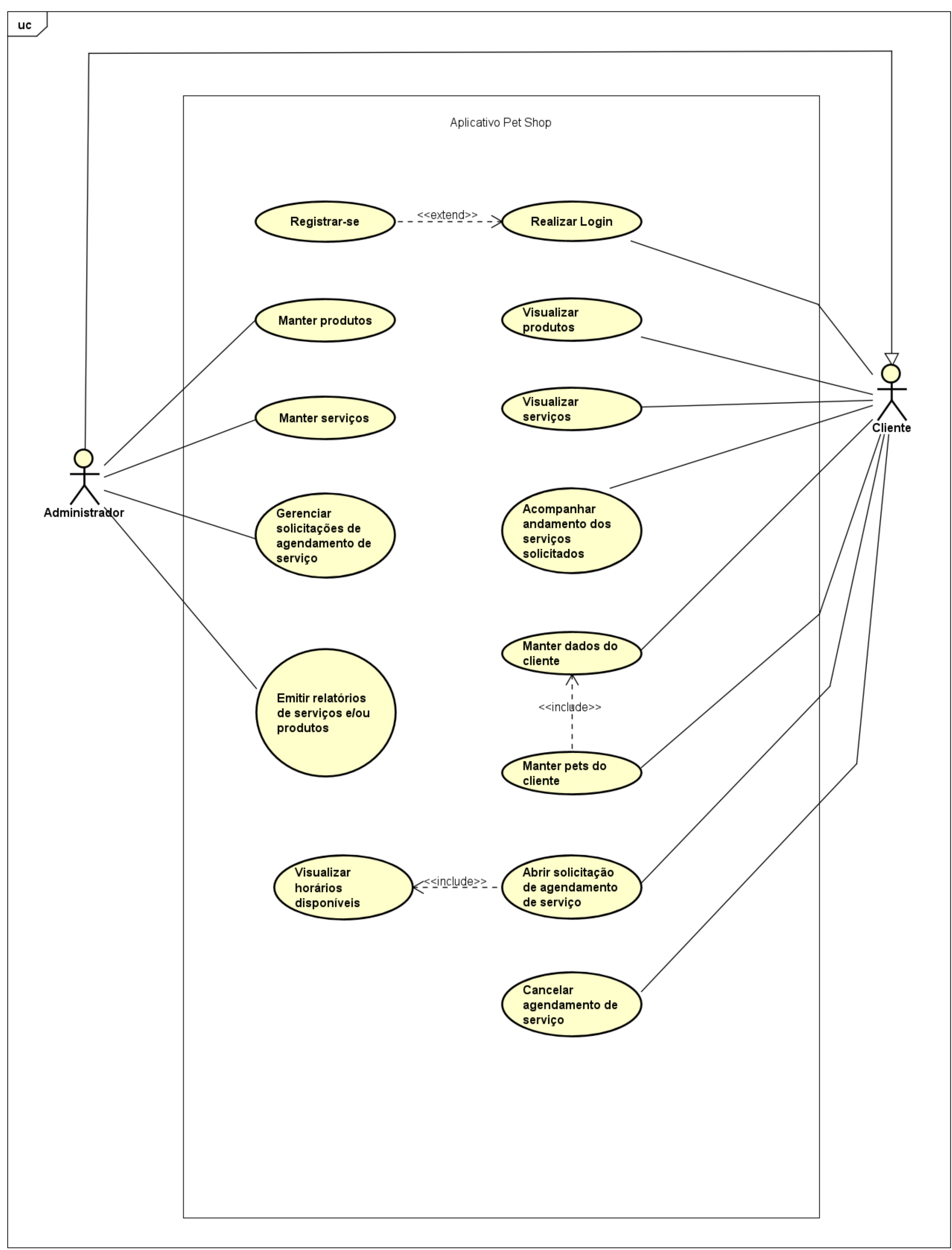
- O sistema terá como linguagem de programação principal o **Typescript**, podendo haver a utilização de outras linguagens de forma complementar em casos específicos;
- O desenvolvimento seguirá os padrões do paradigma de **programação orientada a objetos**;

9.2 Visão Lógica

O projeto será dividido em três partes, sendo elas o sistema web, o aplicativo mobile e a API.

O aplicativo mobile e o sistema web serão responsáveis por estabelecer a interface de comunicação com o usuário, através deles o usuário poderá acessar, manipular e gerar dados no sistema. A API será responsável pelo fornecimento e tratativa de dados do sistema, nela serão aplicadas as regras de negócio e a conexão com banco de dados. Todas as partes do sistema terão seus arquivos organizados através de pastas e classes.

APÊNDICE B - Diagrama de Caso de Uso



APÊNDICE C - Especificação dos Casos de Uso (História de Usuário)

US - História de usuário

CA - Critério de aceitação

US.001	**Manter produtos**
História de Usuário:	**COMO** funcionário **QUERO** gerenciar produtos **PARA** poder adicionar, visualizar, alterar e excluir produtos do sistema.
CA.001.01	**Cadastrar produto** **DADO** que estou na tela de cadastro de produto; **E** estou logado como funcionário; **QUANDO** preencho todos os campos obrigatórios do cadastro; **E** clico em cadastrar; **ENTÃO** ocorre a validação dos dados; **E** visualizo uma mensagem "Produto cadastrado com sucesso";
CA.001.02	**Visualizar produto** **DADO** que estou na tela de produtos; **E** estou logado como funcionário ou cliente; **QUANDO** seleciono um produto; **E** clico para visualizar; **ENTÃO** sou direcionado para uma nova tela; **E** visualizo os dados relacionado ao produto;
CA.001.03	**Alterar produto** **DADO** que estou na tela de alterar produto; **E** estou logado como funcionário; **E** há um produto previamente cadastrado; **QUANDO** preencho todos os campos que desejo alterar; **E** clico em salvar; **ENTÃO** ocorre a validação dos dados; **E** visualizo uma mensagem "Alteração realizada com sucesso";
CA.001.04	**Excluir produto** **DADO** que estou na tela de produtos;

	E estou logado como funcionário; **QUANDO** clico em excluir produto; **E** clico em confirmar exclusão; **ENTÃO** o produto é retirado da tela de produtos; **E** visualizo a mensagem "Produto excluído com sucesso";

US.002	**Manter serviços**
História de Usuário:	**COMO** funcionário **QUERO** gerenciar serviços **PARA** poder adicionar, visualizar, alterar e inativar serviço do sistema.
CA.002.01	**Cadastrar serviços**
	DADO que estou na tela de cadastro de serviços; **E** estou logado como funcionário; **QUANDO** preencho todos os campos obrigatórios do cadastro; **E** clico em cadastrar; **ENTÃO** ocorre a validação dos dados; **E** visualizo uma mensagem "Serviço cadastrado com sucesso";
CA.002.02	**Visualizar serviço**
	DADO que estou na tela de serviços; **E** estou logado como funcionário ou cliente; **QUANDO** seleciono um serviço; **E** clico para visualizar; **ENTÃO** sou direcionado para uma nova tela; **E** visualizo os dados relacionado ao serviço;
CA.002.03	**Alterar serviço**
	DADO que estou na tela de alterar serviço; **E** estou logado como funcionário; **E** há um serviço previamente cadastrado; **QUANDO** preencho todos os campos que desejo alterar; **E** clico em salvar; **ENTÃO** ocorre a validação dos dados; **E** visualizo uma mensagem "Alteração realizada com sucesso";
CA.002.04	**Inativar serviço**

	DADO que estou na tela de serviços; **E** estou logado como funcionário; **QUANDO** clico em inativar serviço; **ENTÃO** o serviço é inativado; **E** visualizo a mensagem "Serviço inativado com sucesso";

US.003	**Gerenciar solicitações de agendamento de serviço**
História de Usuário:	**COMO** funcionário **QUERO** confirmar, recusar ou cancelar uma solicitação de serviço **PARA** controlar os serviços solicitados pelos clientes ao pet shop
	Confirmar solicitação de agendamento de serviço
CA.003.01	**DADO** que estou na tela de solicitação de serviços; **E** que estou logado como funcionário; **E** quero realizar a confirmação de um serviço **QUANDO** clico em confirmar serviço; **ENTÃO** o sistema confirma a solicitação de agendamento do serviço; **E** envia uma notificação ao cliente informando a confirmação do agendamento; **E** visualizo a mensagem "Agendamento confirmado com sucesso";
	Recusar solicitação de agendamento de serviço
CA.003.02	**DADO** que estou na tela de solicitação de serviços; **E** que estou logado como funcionário; **E** quero recusar a solicitação de um serviço; **QUANDO** clico em recusar serviço; **E** o sistema abre um modal solicitando o motivo; **E** escrevo o motivo; **E** clico em enviar motivo; **ENTÃO** o sistema envia uma notificação ao cliente informando o motivo de sua solicitação não ter sido aceita **E** visualizo a mensagem "Agendamento recusado com sucesso";

	Cancelar agendamento de serviço
CA.003.03	**DADO** que estou na tela de listagem dos agendamentos de serviços confirmados; **E** estou logado no sistema; **E** quero realizar o cancelamento do serviço **QUANDO** clico em cancelar serviço; **E** o sistema abre um modal solicitando o motivo; **E** escrevo o motivo; **E** clico em confirmar cancelamento; **ENTÃO** o sistema cancela o agendamento; **E** envia uma notificação ao cliente informando o cancelamento do agendamento; **E** visualizo a mensagem “Agendamento cancelado com sucesso”;

US.004	**Gerar relatório de serviço e/ou produtos**
História de Usuário:	**COMO** funcionário **QUERO** visualizar uma tabela de serviços do mês **PARA** analisar qual período teve maior demanda de serviços e/ou produtos.
	Solicitar um período customizado do relatório de serviço
CA.004.01	**DADO** que estou na tela de gerar o relatório de serviço e/ou produtos; **E** estou logado no sistema; **QUANDO** escolho um período de tempo entre um dia de início e um dia final; **E** clico em continuar e enviar solicitação; **ENTÃO** recebo um relatório específico do período de tempo solicitado; **E** visualizo os serviços prestados e/ou produtos vendidos no período solicitado.

US.005	**Efetuar Login**

História de Usuário:	**COMO** cliente e funcionário **QUERO** efetuar login **PARA** ter a autorização necessária para executar as atividades dentro do sistema.
CA.005.01	**Acesso ao sistema**
	DADO que estou na tela de entrada do sistema; **E** estou cadastrado no sistema; **QUANDO** informo os dados para a autenticação; **E** clico em entrar; **ENTÃO** ocorre a validação dos dados; **E** sou enviado para a tela inicial do sistema.
CA.005.02	**Problema na validação dos dados de login**
	DADO que estou na tela de entrada do sistema; **E** estou cadastrado no sistema; **QUANDO** cliquei em entrar; **E** ocorreu erro de validação de dados; **ENTÃO** os dados de login e senha são apagados; **E** visualizo a mensagem “Login ou senha incorretos!”.
CA.005.03	**Usuário foi bloqueado pelo administrador**
	DADO que estou na tela de entrada do sistema; **E** estou cadastrado no sistema; **QUANDO** cliquei em entrar; **E** usuário tem status bloqueado. **ENTÃO** todos os dados de login e senha são apagados; **E** visualizo a mensagem “Usuário bloqueado! Favor entrar em contato com o administrador do sistema.”

US.006	Registrar-se
História de Usuário:	**COMO** cliente **QUERO** cadastrar-me **PARA** ter a autorização necessária para executar as atividades dentro do sistema.
CA.006.01	**Cadastro no sistema**
	DADO que estou na tela de cadastro do sistema;

	E não estou cadastrado no sistema; **QUANDO** preenche todos os campos obrigatórios do cadastro; **E** clico em cadastrar; **ENTÃO** ocorre a validação dos dados; **E** tenho meu cadastro realizado com sucesso; **E** sou direcionado para a tela inicial do sistema.
CA.006.02	**Problema na validação dos dados de cadastro**
	DADO que estou na tela de cadastro do sistema; **E** não estou cadastrado no sistema; **QUANDO** cliquei em cadastrar; **E** ocorreu erro de validação de dados; **ENTÃO** os campos que são obrigatórios e não foram preenchidos tornam-se inválidos; **E** apresentam uma borda em vermelho; **E** abaixo de cada campo inválido visualizo a mensagem "Preencha este campo".
CA.006.03	**E-mail já cadastrado**
	DADO que estou na tela de cadastro do sistema; **E** não estou cadastrado no sistema; **QUANDO** cliquei em cadastrar; **E** o e-mail que inseri já esteja cadastrado no sistema; **ENTÃO** o campo que está incorreto apresenta uma borda em vermelho; **E** visualizo a mensagem "E-mail já cadastrado no sistema."

US.007	Visualizar produtos disponíveis
História de Usuário:	**COMO** cliente **QUERO** visualizar um produto disponível **PARA** saber quais os produtos que estão à disposição.
CA.007.01	**Visualizar produto**
	DADO que estou na tela de produtos disponíveis; **QUANDO** clico para visualizar produto; **ENTAO** recebo os dados do produto selecionado; **E** visualizo todos os dados disponíveis do produto selecionado.

	Filtrar produtos disponíveis
CA.007.02	**DADO** que estou na tela de produtos disponíveis; **QUANDO** escolho os filtros do produto; **E** clico em filtrar; **ENTÃO** o sistema faz uma filtragem nos produtos; **E** visualizo os produtos que foram resultado da filtragem;

US.008	Visualizar serviços disponíveis
História de Usuário:	**COMO** cliente **QUERO** visualizar um serviço disponível **PARA** saber quais os serviços que estão à disposição.
	Visualizar serviço
CA.008.01	**DADO** que estou na tela de serviços disponíveis; **QUANDO** seleciono um serviço; **ENTÃO** recebo os dados do serviço selecionado; **E** visualizo todos os dados disponíveis do serviço selecionado.
	Filtrar serviços disponíveis
CA.008.02	**DADO** que estou na tela de serviços disponíveis; **QUANDO** escolho os filtros do serviço; **E** clico em filtrar; **ENTÃO** o sistema faz uma filtragem nos serviços; **E** visualizo os serviços que foram resultados da filtragem;

US.009	Acompanhar andamento dos serviços
História de Usuário:	**COMO** cliente **QUERO** visualizar os serviços solicitados **PARA** consultar todos os meus serviços solicitados.
	Visualizar andamento dos serviços
CA.009.01	**DADO** que estou na tela de serviços solicitados; **E** estou logado no sistema; **QUANDO** seleciono o um serviço solicitado; **E** clico em detalhes; **ENTAO** o sistema apresentará o serviço selecionado; **E** visualizo todos os dados relacionados com o serviço.

	Notificar cliente sobre finalização do serviço
CA.009.02	**DADO** que o serviço acaba de ser finalizado; **QUANDO** o funcionário logado no sistema altera o status do serviço para “Finalizado”; **ENTÃO** o sistema apresentará envia uma notificação para o cliente que agendou o serviço informando sobre a sua finalização.

US.010	Manter dados do cliente
História de Usuário:	**COMO** usuário **QUERO** visualizar meus dados no sistema **PARA** confirmar se os dados estão corretos.
	Editar dados do usuário
CA.010.01	**DADO** que estou na tela dados do usuário; **E** estou logado no sistema como usuário da conta ou administrador do sistema; **QUANDO** seleciono um campo específico; **E** modifico os dados contidos nele; **ENTÃO** faço a confirmação da alteração dos dados do campo; **E** recebo a mensagem de que os dados foram alterados com sucesso.
	Desativar conta de usuário
CA.010.02	**DADO** que estou na tela de dados do usuário; **E** estou logado no sistema como usuário da conta ou administrador do sistema; **QUANDO** escolho a opção de desativar conta; **E** clico em confirmar; **ENTÃO** o sistema faz uma confirmação da desativação da conta; **E** retorna a mensagem de “Conta desativada com sucesso.” **E** redirecionar o usuário para tela de login.

US.011	Manter dados do pet
História de Usuário:	**COMO** usuário **QUERO** visualizar dados do pet no sistema **PARA** confirmar se os dados estão corretos.

	Cadastrar Pet
CA.011.01	**DADO** que estou na tela de cadastro de pet de um cliente; **E** estando logado no sistema como cliente dono do pet ou funcionário; **QUANDO** preencho todos os campos obrigatórios do cadastro; **E** clico em cadastrar; **ENTÃO** ocorre a validação dos dados; **E** visualizo uma mensagem “Pet cadastrado com sucesso”;
	Editar dados do pet
CA.011.02	**DADO** que estou na tela dados do pet; **E** estando logado no sistema como cliente dono do pet ou funcionário; **QUANDO** seleciono um campo específico; **E** modifico os dados contidos nele; **ENTÃO** faço a confirmação da alteração dos dados do campo; **E** recebo a mensagem de que os dados foram alterados com sucesso.
	Desativar pet cadastrado
CA.011.02	**DADO** que estou na tela de dados do pet; **E** estando no sistema; **QUANDO** escolho a opção de desativar pet; **E** clico em confirmar; **ENTÃO** o sistema faz uma confirmação da desativação do pet; **E** retorna a mensagem de “Pet desativado com sucesso.”

US.012	Solicitar agendamento de serviço
História de Usuário:	**COMO** cliente **QUERO** realizar uma solicitação de agendamento de serviço **PARA** marca um horário e um dia para a realização dos serviços.
	Solicitando agendamento de serviço
CA.012.01	**DADO** que estou na tela de solicitação de serviço; **E** estou logado no sistema; **QUANDO** escolho uma hora e um dia;

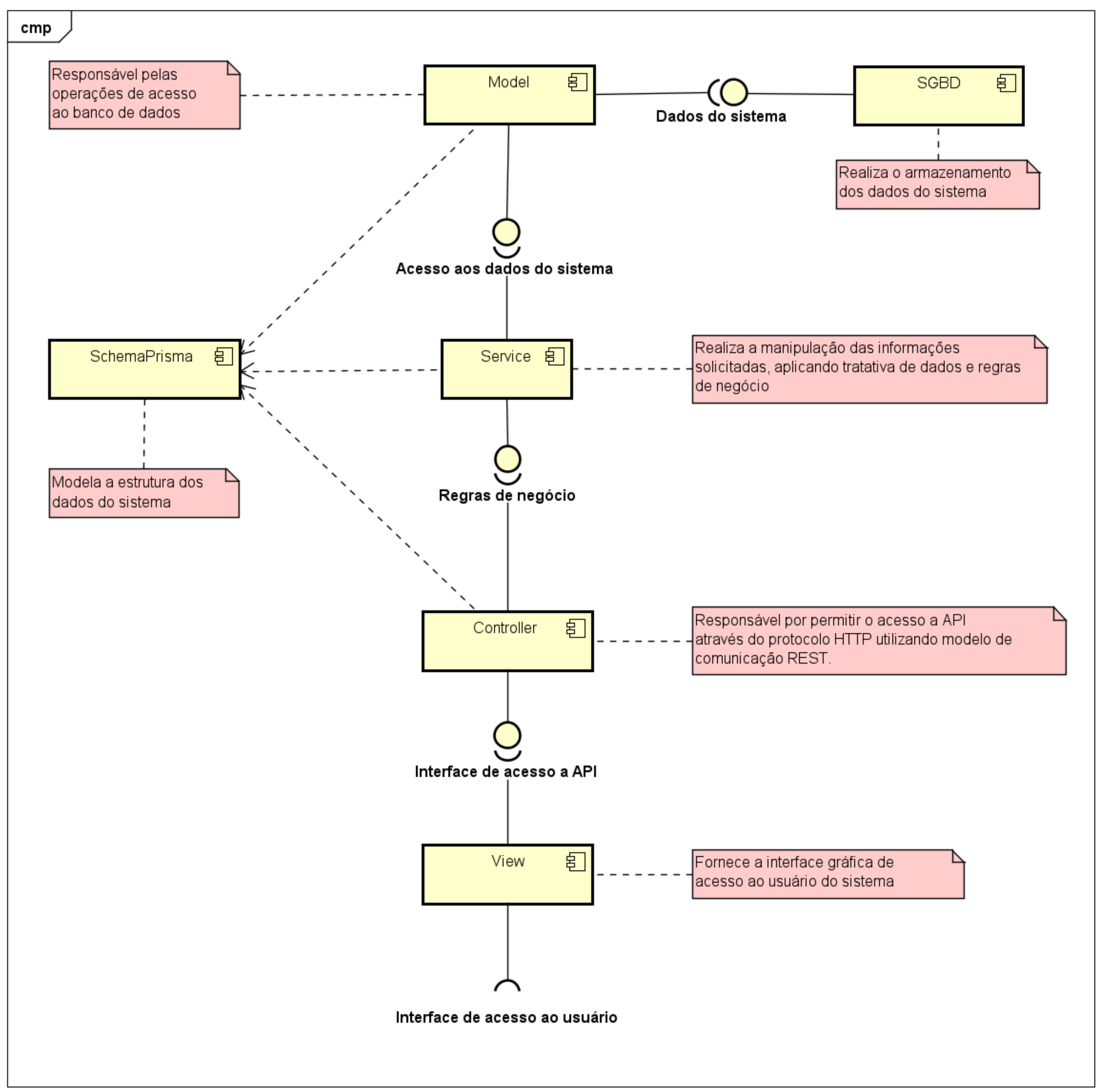
	E clico em continuar e enviar solicitação; **ENTAO** recebo a confirmação de horário; **E** visualizo a hora e o dia que foram agendados.
CA.012.02	**Solicitar agendamento de serviço sem informar todos os campos obrigatórios**
	DADO que estou na tela de solicitação de serviço; **E** estou logado no sistema; **QUANDO** cliquei em solicitar serviço; **E** não preenchi todos os campos marcados como obrigatórios; **E** ocorreu erro de validação de dados; **ENTÃO** os campos que são obrigatórios e não foram preenchidos tornam-se inválidos; **E** apresentam uma borda em vermelho; **E** abaixo de cada campo inválido visualizo a mensagem "Campo Obrigatório".

US.0013	Visualizar horários disponíveis
História de Usuário:	**COMO** cliente **QUERO** visualizar os horários disponíveis **PARA** saber quais os horários que estão à disposição para o agendamento de serviço.
CA.013.01	**Visualizar horários**
	DADO que estou na tela de solicitação de serviço; **E** estou logado no sistema; **QUANDO** seleciono um dia disponível; **E** clico no dia; **ENTÃO** o sistema apresentará os horários disponíveis; **E** visualizo todos os horários que se encontram disponíveis no dia que foi selecionado.

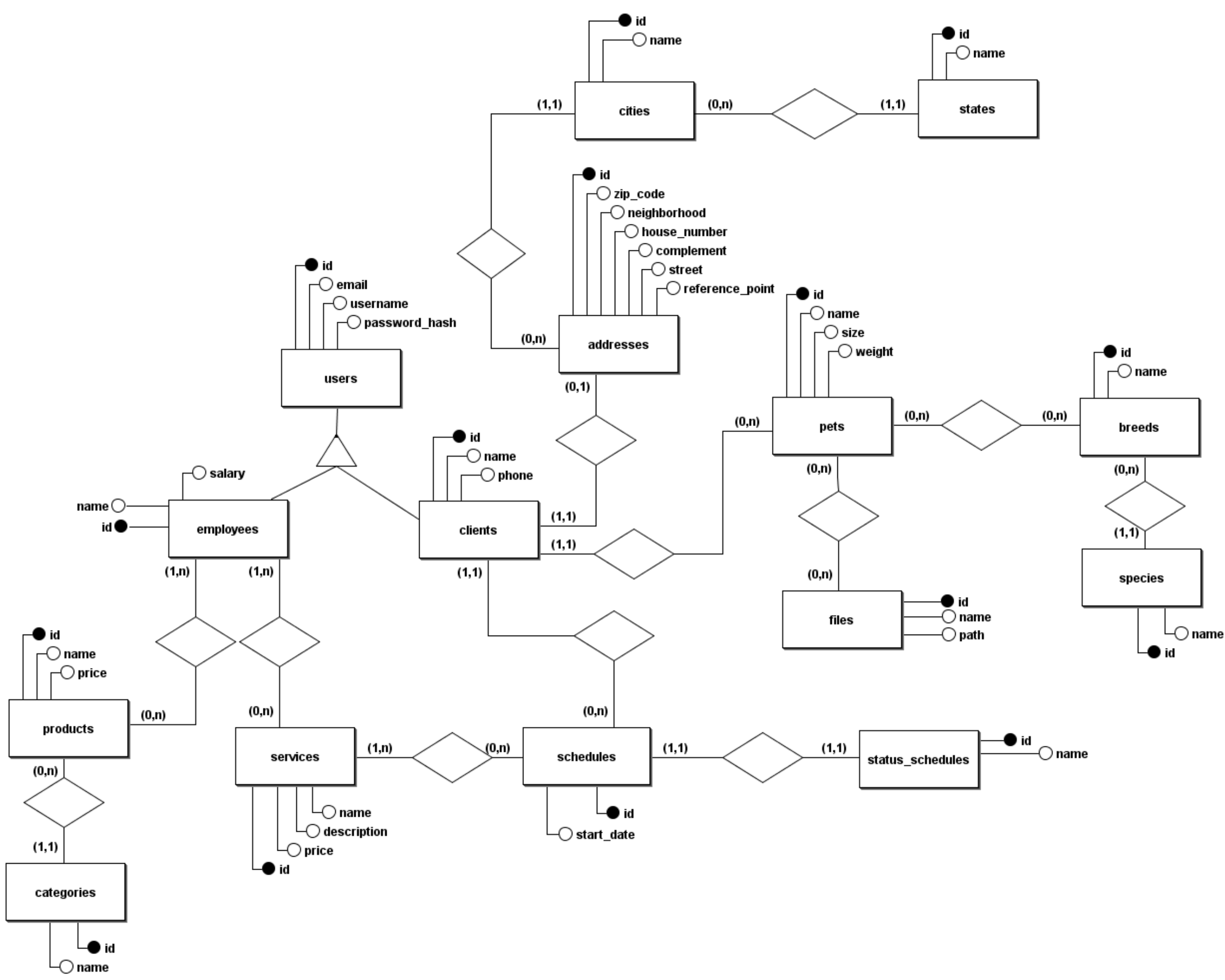
US.014	Cancelar agendamento de serviço
História de Usuário:	**COMO** cliente **QUERO** solicitar o cancelamento de um serviço **PARA** cancelar o agendamento do serviço.

	Cancelar agendamento de serviço
CA.014.01	**DADO** que estou na tela de um serviço; **E** estou logado no sistema; **E** estou solicitando o cancelamento 2 horas antes do início do serviço; **QUANDO** cliquei em cancelar serviço; **E** clico em confirmar cancelamento; **ENTÃO** o sistema informa que o agendamento foi cancelado com sucesso; **E** visualizo a mensagem “Agendamento cancelado com sucesso”;
	Falha ao cancelar agendamento de serviço
CA.014.02	**DADO** que estou na tela de um serviço; **E** estou logado como cliente no sistema; **E** estou solicitando o cancelamento estando a menos de 2 horas para do início do serviço; **QUANDO** cliquei em cancelar serviço; **E** clico em confirmar cancelamento; **ENTÃO** o sistema informa que ocorreu uma falha; **E** visualizo a mensagem “Erro ao cancelar! Só é possível o cancelamento de serviços com 2 Horas de antecedência. Entre em contato com o Pet Shop”;

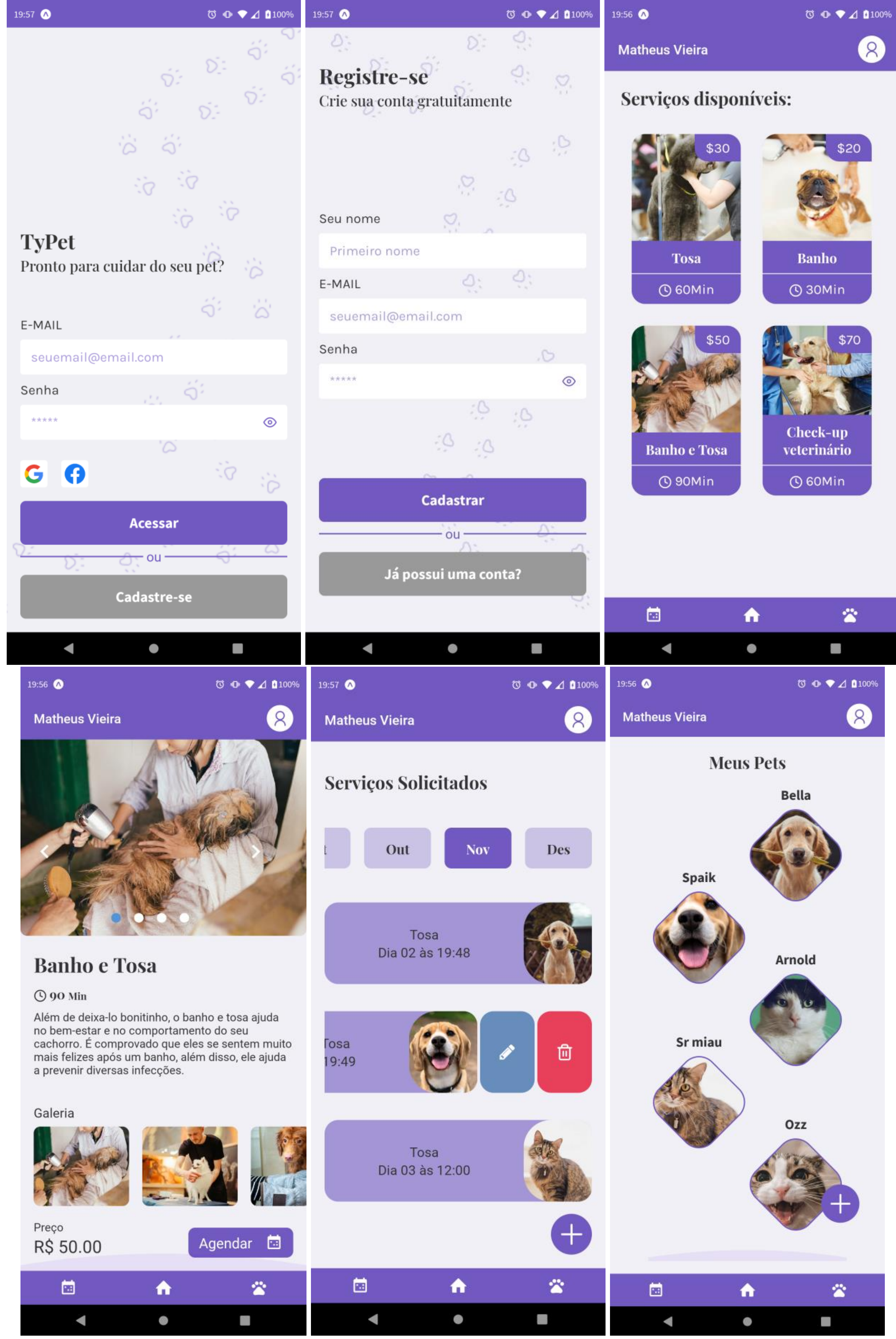
APÊNDICE D - Diagrama de Componentes



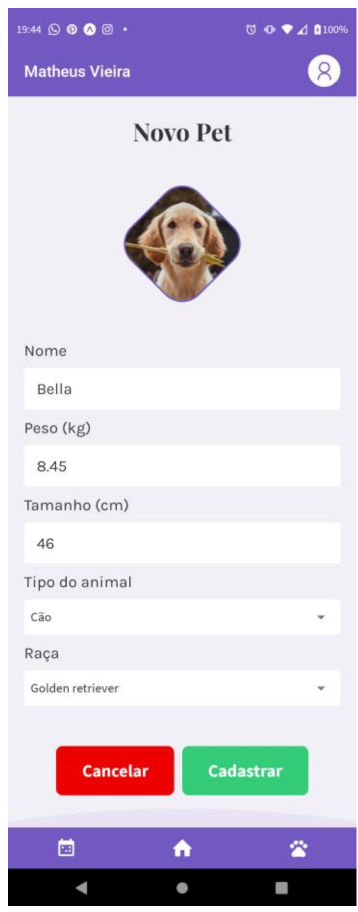
APÊNDICE E - Diagrama Entidade Relacionamento

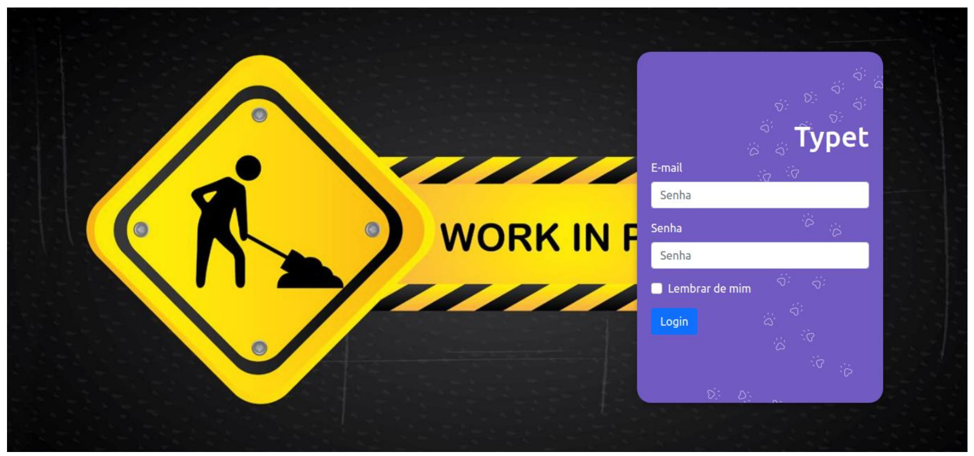


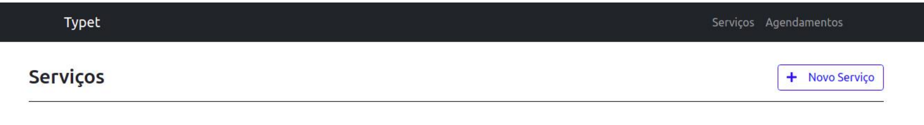
APÊNDICE F – Telas do sistema

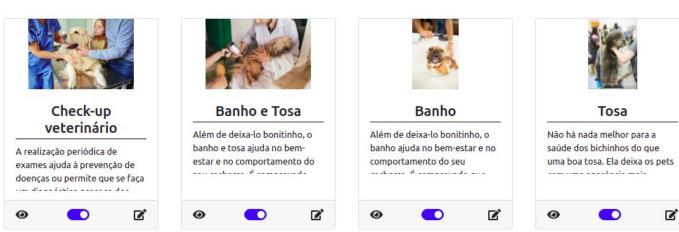


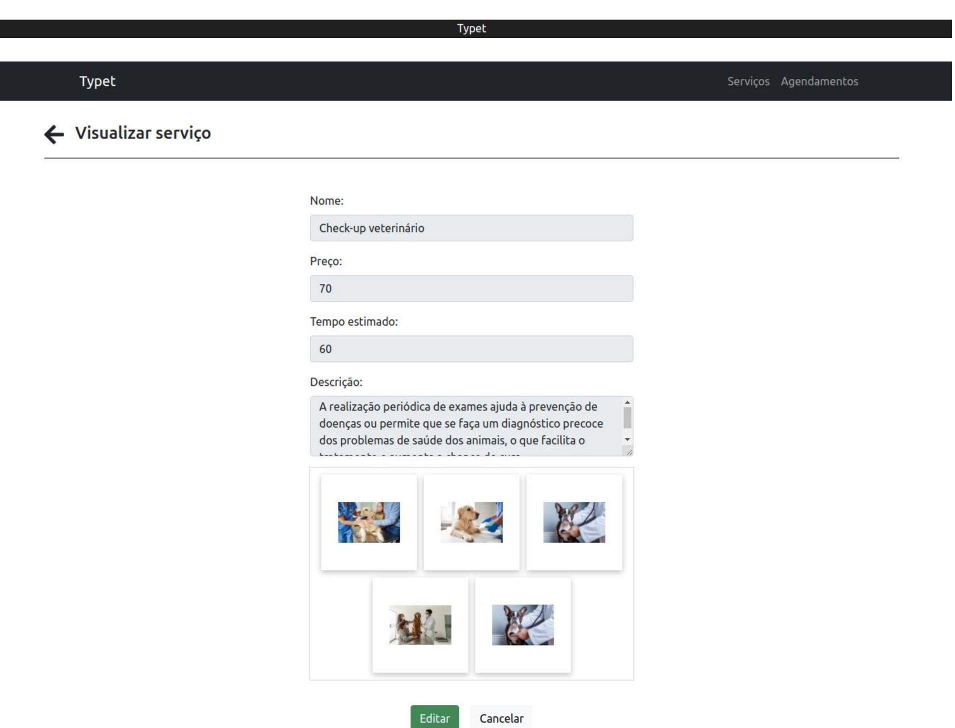
TyPet
Pronto para cuidar do seu pet?
E-MAIL
seuemail@email.com
Senha
Acessar
ou
Cadastre-se
Registre-se
Crie sua conta gratuitamente
Seu nome
Primeiro nome
E-MAIL
seuemail@email.com
Senha
Cadastrar
ou
Já possui uma conta?
Matheus Vieira
Serviços disponíveis:
$30
Tosa
60Min
$20
Banho
30Min
$50
Banho e Tosa
90Min
$70
Check-up veterinário
60Min
Matheus Vieira
Banho e Tosa
90 Min
Além de deixa-lo bonitinho, o banho e tosa ajuda no bem-estar e no comportamento do seu cachorro. É comprovado que eles se sentem muito mais felizes após um banho, além disso, ele ajuda a prevenir diversas infecções.
Galeria
Preço
R$ 50.00
Agendar
Matheus Vieira
Serviços Solicitados
Out
Nov
Des
Tosa
Dia 02 às 19:48
Tosa
19:49
Tosa
Dia 03 às 12:00
Matheus Vieira
Meus Pets
Bella
Spaik
Arnold
Sr miau
Ozz


19:44
100%
Matheus Vieira
Novo Pet
Nome
Bella
Peso (kg)
8.45
Tamanho (cm)
46
Tipo do animal
Cão
Raça
Golden retriever
Cancelar
Cadastrar


WORK IN P
Typet
E-mail
Senha
Senha
Senha
Lembrar de mim
Login







Typet

Typet

Serviços Agendamentos

Novo serviço

Nome:

Preço:

Tempo estimado:

Descrição:

Adiconar nova imagem

Nenhuma imagem selecionada!

Salvar Cancelar

Typet