UNIVERSIDADE ESTADUAL DO MARANHÃO
CENTRO DE CIÊNCIAS TECNOLÓGICAS
CURSO DE ENGENHARIA DA COMPUTAÇÃO

ALEXI LALLAS RIBEIRO PEREIRA

DESENVOLVIMENTO DE UM SISTEMA DE ACOMPANHAMENTO DE PACIENTES PARA O POSTO MÉDICO DA UEMA

São Luís

2019

ALEXI LALLAS RIBEIRO PEREIRA

DESENVOLVIMENTO DE UM SISTEMA DE ACOMPANHAMENTO DE PACIENTES PARA O POSTO MÉDICO DA UEMA

Monografia apresentada ao Curso de Engenharia de Computação da Universidade Estadual do Maranhão, como requisito para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. MSc. Wesley Batista Dominices de Araujo.

São Luís

2019

ALEXI LALLAS RIBEIRO PEREIRA

DESENVOLVIMENTO DE UM SISTEMA DE ACOMPANHAMENTO DE PACIENTES PARA O POSTO MÉDICO DA UEMA

Monografia apresentada ao Curso de Engenharia de Computação da Universidade Estadual do Maranhão, como requisito para obtenção do grau de Bacharel em Engenharia de Computação.

Aprovada em: de dezembro de 2019

BANCA EXAMINADORA

_____

Prof. MSc. Wesley Batista Dominices de Araujo (Orientador)
Universidade Estadual do Maranhão

_____

Prof. MSc. Pedro Brandão Neto (Examinador)
Universidade Estadual do Maranhão

_____

Prof. Dr. Cicero Costa Quarto (Examinador)
Universidade Estadual do Maranhão

Dedico este trabalho à minha querida mãe que sempre se dedicou à minha educação.

AGRADECIMENTOS

Agradeço primeiramente à Deus que me deu sabedoria e forçar para chegar até aqui, também agradeço à minha mãe, Jaquilene Ribeiro pelo absoluto apoio que sempre tive em minha carreira de estudante, também agradeço aos sacrifícios pelos quais passou para que eu pudesse chegar onde estou. Também agradeço à minha irmã Laila Ribeiro pelo suporte que prestou nas muitas vezes que passei por dificuldades fora da Universidade.

Aos meus amigos Lucas Santos, Pedro Fellipe, Marcos Guilhon, Onildo, Eduardo Andrade e Breno Batista pelos bons momentos e boas risadas proporcionadas durante o curso. Em especial gostaria de gradecer à José Pedro Jr e Lorena Tavares que além da amizade e bons momentos, me proporcionaram também um canal de apoio e desabafos durantes os dias difíceis que tivemos, levarei vocês sempre comigo.

Ao analista Paulo César pela orientação na modelagem do banco de dados, ao programador Guilherme de Oliveira pelo auxílio e dicas durante o design das telas do sistema e à Gabriela Raposo por me ajudar a sair da inércia para escrever este trabalho.

Por fim agradeço a Wesley Batista Dominices de Araujo pelas oportunidades dadas e também pela orientação para conclusão deste trabalho.

“As pessoas são sozinhas porque constroem muros ao invés de pontes”

O Pequeno Príncipe

RESUMO

Este trabalho apresenta o desenvolvimento de um sistema de gerenciamento de pacientes para o posto médico da UEMA, tendo em vista que o mesmo não possui qualquer forma digital de gerenciar os dados coletados e, além disso, possui uma grande movimentação de dados anual, o que gera uma dificuldade dos administradores em gerenciar o mesmo. A metodologia escolhida para o desenvolvimento da aplicação foi o RAD (*Rapid Application Development*) e a arquitetura da aplicação foi com base na utilização de uma API (*Application Programming Interface*) para se comunicar com a aplicação *web*. As tecnologias utilizadas foram Laravel para a construção da API e Angular para o desenvolvimento da aplicação *web*. O sistema foi testado pelos funcionários do posto e teve uma boa aprovação, pois possui uma interface amigável e intuitiva, o que proporcionou uma boa experiência aos usuários.

Palavras-chave: Laravel Framwork, Angular Framework, API, Gerenciamento de pacientes.

ABSTRACT

This paper presents the development of a patient management system for the UEMA medical clinic, considering that it had no digital way to manage the collected data and, besides, it has a large annual data movement, which creates a difficulty for administrators to manage it. The methodology chosen for application development was Rapid Application Development (RAD) and the application architecture was based on the use of an Application Programming Interface (API) to communicate with the web application. The technologies used were Laravel for building API and Angular for web application development. The system was tested by the station staff and was well approved because it has a friendly and intuitive interface, which provided a good user experience.

Key Words: Laravel Framework, Angular Framework, API, patient management.

LISTA DE FIGURAS

LISTA DE SIGLAS

API	- Application Programming Interface (Interface de programação de aplicações)
CLI	- Comand-line Interface
CPF	- Certidão de Pessoa Física
CSS	- Cascading Style Sheets
ER	- Entidade Relacionamento
HTML	- Hypertext Markup Language
HTTP	- Hypertext Transfer Protocol
IDE	- Integrated Development Environment
JSON	- JavaScript Object Notation
JWT	- JSON Web Token
MVC	- Model – view – controller
PHP	- Hypertext Preprocessor
PROECAC	- Pró-Reitoria de Extensão, Cultura e Assuntos da Comunidade
RAD	- Rapid Application Development
REST	- Representational State Transfer
SI	- Sistema de Informação
TI	- Tecnologia da Informação
UEMA	- Universidade Estadual do Maranhão
URL	- Uniform Resource Locator
XML	- Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

O serviço social médico do Campus Paulo VI é um órgão ligado à Pró-reitoria de Extensão, Cultura e Assuntos da Comunidade (PROECAC). O órgão foi reativado em 10 de março de 2015 para atender à comunidade universitária (docentes, discentes, servidores e prestadores de serviço) e também à comunidade local, com funcionamento nos horários matutino e vespertino.

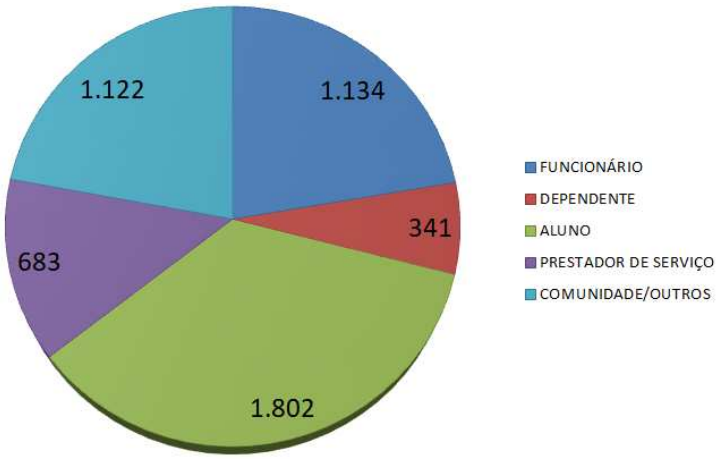
O posto tem três consultórios, uma sala para material biológico, uma farmácia básica, uma enfermaria e uma sala da coordenação. Com uma equipe formada por médicos, enfermeiros, técnicos de enfermagem, assistente social e farmacêutico, tem como atividade principal o atendimento de emergência. Os casos mais complexos são encaminhados para hospitais próximos à Universidade.

Além disso, o Serviço Médico também desenvolve campanhas de prevenção, vacina e atividades educativas de promoção da saúde e atendimentos de serviço social, para tanto, busca parcerias com órgãos ligados ao setor.

Sua capacidade de atendimento é de 10 pacientes diariamente, entretanto é comum dias em que essa quantidade é muito superior, devido à grande demanda existente. Seu atendimento é dividido em duas etapas, a primeira com o enfermeiro, onde são coletados dados do exame físico como temperatura, pressão e glicemia.

Caso haja necessidade, a segunda etapa é realizada com um médico que irá fazer o acompanhamento do caso do paciente. Esses atendimentos, somente no ano de 2017, geraram a movimentação de mais de 5 mil fichas de anamnese, como pode-se ver na Figura 1.

Figura 1 - Atendimento de Enfermagem em 2017



Fonte: Posto Médico da UEMA (2017)

A consulta a dados de um paciente que já frequentou o posto médico gera um grande esforço por parte do atendente em recuperar os dados desejados, já que não é utilizado nenhum meio digital para salvar as informações, dessa forma também dificulta o trabalho do gestor em coletar dados para relatórios.

Por essas razões, este trabalho tem como objetivo o desenvolvimento de um sistema de informação onde se poderão cadastrar pacientes, itens da farmácia, registrar consultas, resgatar informações de qualquer paciente rapidamente e também gerar relatórios quantitativos e gráficos sobre os atendimentos realizados pelo serviço médico.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver um sistema de acompanhamento de pacientes para o posto médico da UEMA utilizando tecnologias gratuitas e de livre acesso.

1.1.2 Objetivos Específicos

- Estudar os melhores *frameworks* para se utilizar no desenvolvimento do sistema;
- Escolher uma arquitetura adequada para a construção de um sistema escalável e reutilizável;
- Desenvolver um sistema com interface amigável e intuitiva.

1.2 Estrutura organizacional do trabalho

O presente trabalho está dividido em cinco capítulos. O primeiro é a introdução, onde é apresentado o problema e a proposta para solução do mesmo, o segundo é a fundamentação teórica, onde serão explicados conceitos importantes para a compreensão do trabalho, como as tecnologias e ferramentas utilizadas durante o desenvolvimento do sistema, assim como conceitos pertinentes ao desenvolvimento do sistema.

No capítulo seguinte será detalhada a metodologia utilizada no desenvolvimento, mostrando o que acontece em cada etapa e como a metodologia converge para a solução final. Em sequência, no capítulo 4, serão abordados os resultados obtidos e Por fim o último capítulo apresentará as conclusões obtidas e propostas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Sistemas de informação

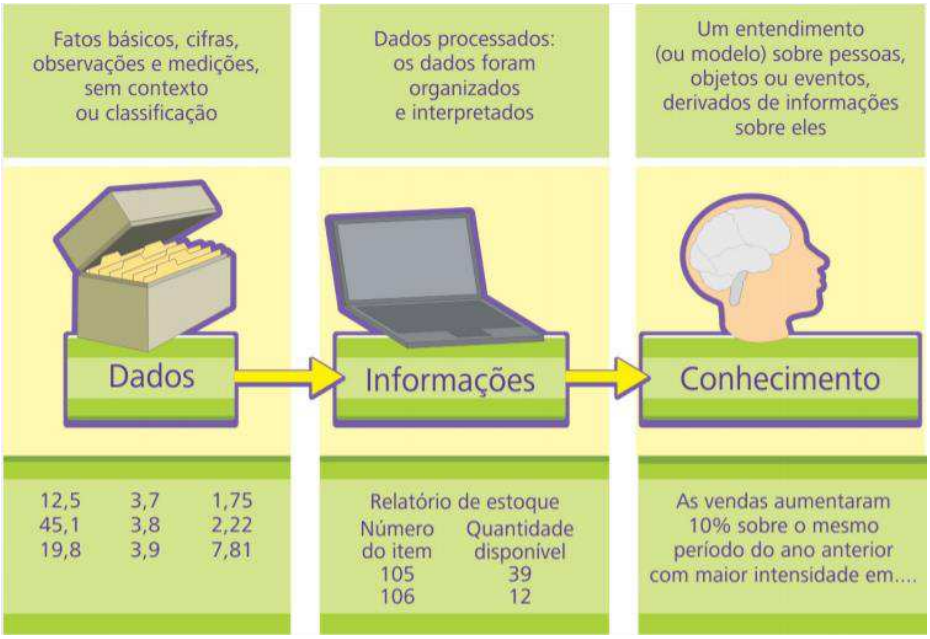
A criação de sistemas remonta ao homem primitivo que sempre teve uma necessidade natural de organização, para tal começou a se compor em sociedade e constituir civilizações (FERNANDES). Um dos primeiros sistemas criado pela civilização foi o numérico (MORAES, 2015) e (IMENES, 1999), que proporcionou um grande avanço à humanidade, facilitando a contagem de suas produções e impulsionando a criação também do calendário e a definição das estações do ano para que se tivesse um maior controle sobre a colheita. Nota-se, portanto que os sistemas sempre se fizeram necessários para um melhor controle do ser humano sobre suas atividades.

Alguns milênios depois e essa necessidade de organização se manteve contínua, entretanto o homem evoluiu muito tecnologicamente. Os períodos de chuva e seca que antes não eram previstos de maneira precisa agora são facilmente obtidos por meio de sistemas meteorológicos.

Como se pode notar, os sistemas são sempre utilizados para fornecer informações, levando em consideração alguns dados fornecidos. Para Chiavennato (2004, p.417), sistema é um conjunto de elementos dinamicamente relacionados, formando uma atividade para atingir um objetivo, operando sobre dados para fornecer informação. É importante destacar que todo sistema, usando ou não recursos de tecnologia da informação (TI), irá gerar informação, portanto, pode-se dizer que são sistemas de informação (SI).

As informações fornecidas pelos SI são a interpretação dos dados, que podem ser números, letras ou dígitos isolados que não representam significado claro. A informação será o resultado desses dados trabalhados, que irá representar um significado lógico para quem a utilizar. Por sua vez o conhecimento será resultado da interpretação da informação. Esse fluxo pode ser observado na Figura 2.

Figura 2 - Dado, Informações e Conhecimento



Fonte: PRATES (2019)

Neste trabalho serão abordados os sistemas de informação computadorizados, tais SI utilizam-se de *hardware*, *software*, redes de telecomunicações, técnicas de administração de dados computadorizadas e outras formas de TI.

Seja para trabalhar nos registros, análises, planejamento e tomada de decisões, em todos os níveis da organização, é imprescindível o uso de um SI, pois o mesmo é o responsável por difundir as informações através da organização.

De acordo com (O' Brien, 2000), pode-se dividir os sistemas de informação em quatro tipos: Sistemas de informação transacionais, gerenciais, de apoio à decisão e executivos.

Os sistemas de informações mais comumente utilizados em pequenas organizações são os transacionais, pois são os mais fáceis e baratos de serem desenvolvidos por tratar-se de um SI relativamente simples, responsável por prover as funções básicas, onde determinada ação do usuário resultará em uma transação, que pode gerar uma reação em cadeia, gerando outras transações. Como por exemplo, ao se excluir um paciente do sistema, todas as fichas de avaliação médica referentes àquele paciente serão removidas também.

Os SI transacionais tem como finalidade colaborar para o melhor atendimento das organizações e monitoramento das atividades diárias como atendimento de pacientes, fluxo de materiais e outros. Seu objetivo é capacitar as organizações a executar suas atividades mais importantes de maneira mais rápida e eficiente.

2.2 Engenharia de *software*

Desenvolver *software* é, muitas vezes, confundido com programação. Essa confusão inicial pode ser atribuída, parcialmente, pela forma como as pessoas são introduzidas nesta área de conhecimento, começando por desenvolver habilidades de raciocínio lógico, através de programação e estruturas de dados. Começa-se resolvendo pequenos problemas que gradativamente vão aumentando a complexidade, requerendo maiores conhecimentos e habilidades, até chegar a um nível em que essa abordagem individual será insuficiente, fazendo-se necessário o uso da engenharia.

A engenharia de *software* é uma abordagem sistemática para a produção de *software*; ela analisa questões práticas de custo, prazo e confiança, assim como as necessidades dos clientes e produtores do *software*. A forma como essa abordagem sistemática é realmente implementada varia drasticamente de acordo com a organização que está a desenvolver o *software*, com o tipo de *software* e com as pessoas envolvidas no processo de desenvolvimento e recursos disponíveis (SOMMERVILLE, 2013).

Não existem técnicas e métodos universais na engenharia de *software* adequados a todos os sistemas e todas as empresas. Em vez disso, um conjunto diverso de métodos e ferramentas de engenharia de *software* tem evoluído nos últimos 50 anos.

2.3 Metodologias de desenvolvimento de *software*

Para uma boa gestão de projeto, é necessário gerir pessoas, ter um bom planejamento e evitar o máximo o desperdício de tempo ou dinheiro. Para tal, faz-se a utilização de metodologias de desenvolvimento, atualmente encontram-se na literatura duas formas. A primeira metodologia criada, chamada de tradicional e as metodologias ágeis. A seguir será abordado com mais detalhes as qualidades e características de cada uma.

2.3.1 Metodologias tradicionais

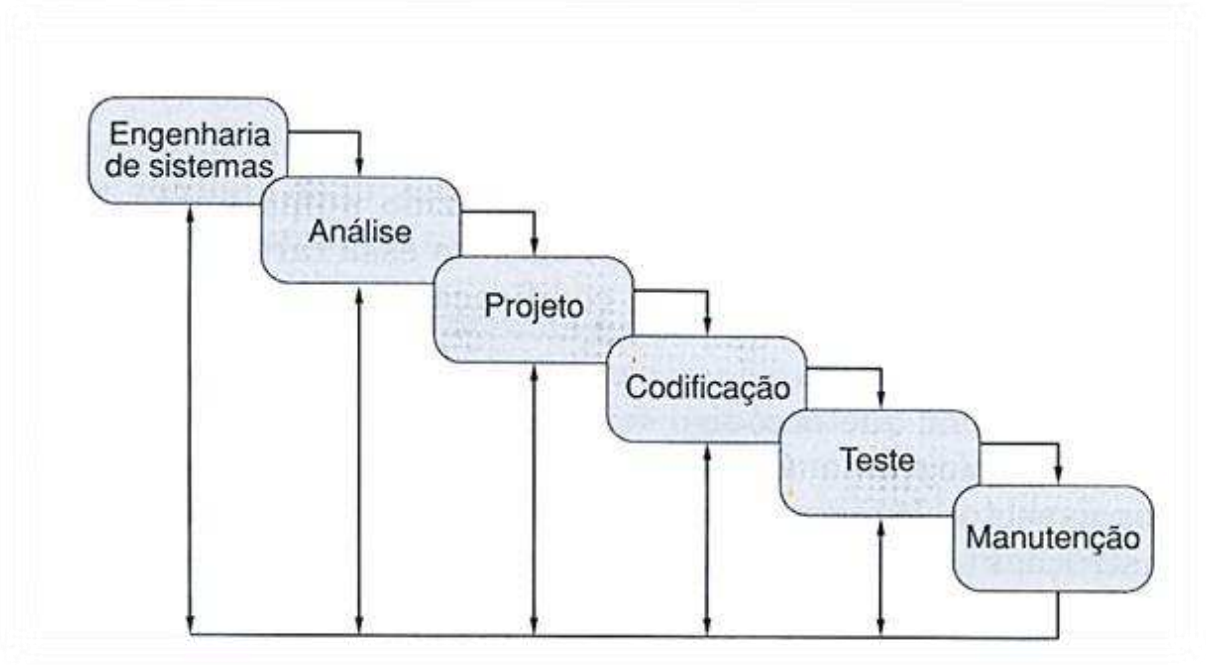
As metodologias tradicionais são chamadas de pesadas ou orientadas a documentação. Elas foram muito utilizadas no passado em um contexto de desenvolvimento de *software* muito diferente do atual, baseado apenas em *mainframes* e terminais.

Naquela época, o custo de fazer alterações e correções no código era muito alto, uma vez que o acesso aos computadores era limitado e não existiam ferramentas robustas de apoio ao desenvolvimento do *software*, tais como depuradores e analisadores de código. Por isso o *software* era todo planejado e documentado antes de ser implementado.

As metodologias tradicionais apresentam, entre outras características, fases bem definidas de ciclo de vida de desenvolvimento de *software*, maior documentação sobre o produto gerado e suas fases de desenvolvimento, maior formalização da comunicação através de documentação, papéis bem definidos para os membros da equipe e busca de padronização das etapas de desenvolvimento (ALMEIDA, 2017). Uma das primeiras metodologias tradicionais criadas foi o modelo clássico ou cascata (SOARES, 2004).

O modelo cascata é caracterizado por possuir uma progressão sequencial entre uma fase e a seguinte. Eventualmente, pode voltar para uma fase anterior, caso seja necessário, mas em suma o modelo cascata segue uma sequência, que pode ser observada na Figura 3.

Figura 3 - Fases da modelo Cascata



Fonte: PRESSMAN (2006)

A figura acima representa o modelo em cascata, também conhecido como sequencial, ou linear, por se basear em uma sucessão de etapas onde uma só é iniciada após o fim da anterior a ela. Como se pode ver, o desenvolvimento flui da parte de cima em direção à etapa de manutenção.

Nessa metodologia, inicialmente procura-se compreender completamente o problema a ser resolvido, seus requisitos e suas restrições; depois projeta-se soluções que atendam a todos os requisitos e restrições. Feito isto se inicia a implementação do projeto e quando toda a etapa de implementação é concluída verifica-se junto ao cliente se a solução atende aos

requisitos estabelecidos e por fim é efetuada a entrega do produto (KROLL e KRUCHTEN, 2003 apud LUIZ, 2011).

2.3.2 Metodologias ágeis

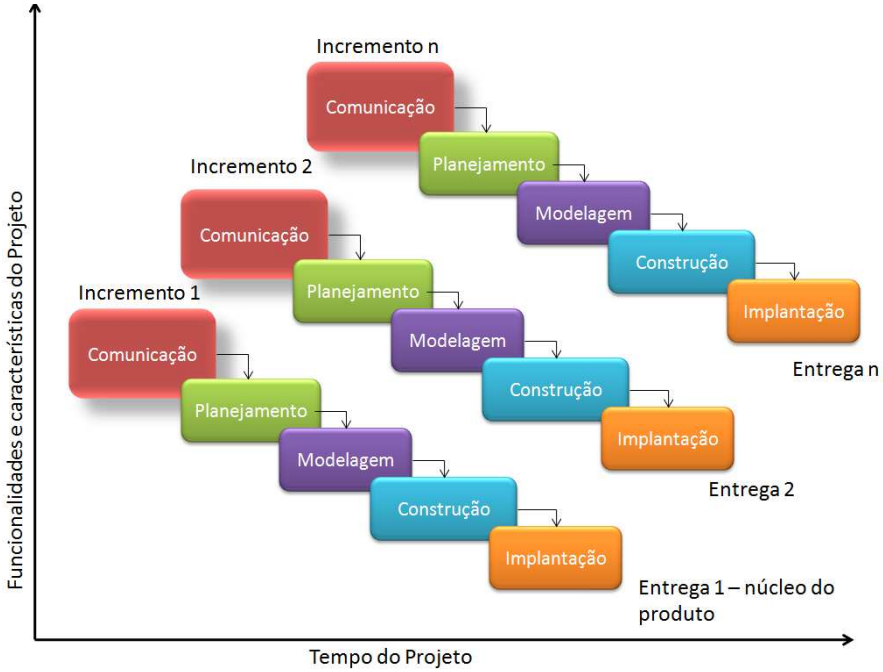
Um dos grandes problemas das metodologias tradicionais é que quase nunca os projetos reais seguem de forma sequencial, e como o cliente sente dificuldades de exprimir suas necessidades e só vê o produto no final do processo, o risco do produto final não ser o esperado é muito alto. Além disso, caso houver erro em uma das etapas, o resultado pode ser caro ou desastroso.

Como as metodologias tradicionais apresentaram uma grande ineficiência, surgiram as metodologias ágeis que possibilitaram o desenvolvimento com mais rapidez e flexibilidade na mudança dos requisitos do longo do processo. Os métodos ágeis são caracterizados por suas etapas interativas (com ciclos incrementais) de desenvolvimento, menor documentação e maior foco no desenvolvimento do sistema, papéis menos definidos, maior aceitação a mudanças e comunicação menos informal entre os membros da equipe.

Um modelo bem utilizado para desenvolvimento ágil é o RAD, trata-se de um modelo linear sequencial que faz com que o ciclo de produção seja bem reduzido, com uma média de 30 a 90 dias. No RAD os requisitos são estabilizados e o projeto é dividido em subprojetos para equipes que irão desenvolver o *software* utilizando uma base de componentes reutilizáveis, que podem ser encontrados prontos ou podem ser desenvolvidos também.

A definição do termo RAD foi cunhada por James Martin, autor e consultor de tecnologias da informação, em 1991. O RAD é dividido em cinco etapas, que se forem seguidas permitem o desenvolvimento muito rápido de aplicações. Suas etapas podem ser vistos na Figura 4.

Figura 4 - Etapas do modelo RAD



Fonte: Elaborado pelo autor (2019).

Como se pode ver na Figura 4, o *software* é desenvolvido de forma incremental, possibilitando o desenvolvimento paralelo de vários componentes e ao final do processo, as entregas serão somadas e integradas ao *software* final. Suas etapas serão descritas detalhadamente a seguir.

- Comunicação: É a etapa de modelagem do negócio, será realizada a análise e negociação do projeto, onde serão esclarecidas as dúvidas e processos que o sistema irá gerir;
- Planejamento: É a etapa de planejamento dos dados que o sistema utilizará, será desenhada a estrutura de dados do *software*, onde terão as tabelas e seus relacionamentos;
- Modelagem: É a fase da modelagem do processo, onde serão estruturadas as regras de modificação, edição, exclusão e qualquer outra mudança de estado do objeto e seus relacionamentos;
- Construção: É a fase de geração da aplicação, onde será desenvolvido o *software* com o foco na ideia principal do RAD que é o reaproveitamento máximo de componentes já existentes para que haja ganho máximo de tempo no projeto;
- Implantação: Fase final do RAD, onde serão realizados os testes de fluxo e modificações, caso seja necessário. Como o RAD é uma metodologia incremental, os

componentes já foram testados anteriormente, então a probabilidade de se encontrar erros é muito baixa, o que torna tal fase bem rápida.

Existem diversas ferramentas que possibilitam o RAD, as mais utilizadas são Zoho Creator, Google App Maker, GitHub, Microsoft Power Apps, Glide Apps entre outros.

2.4 Framework

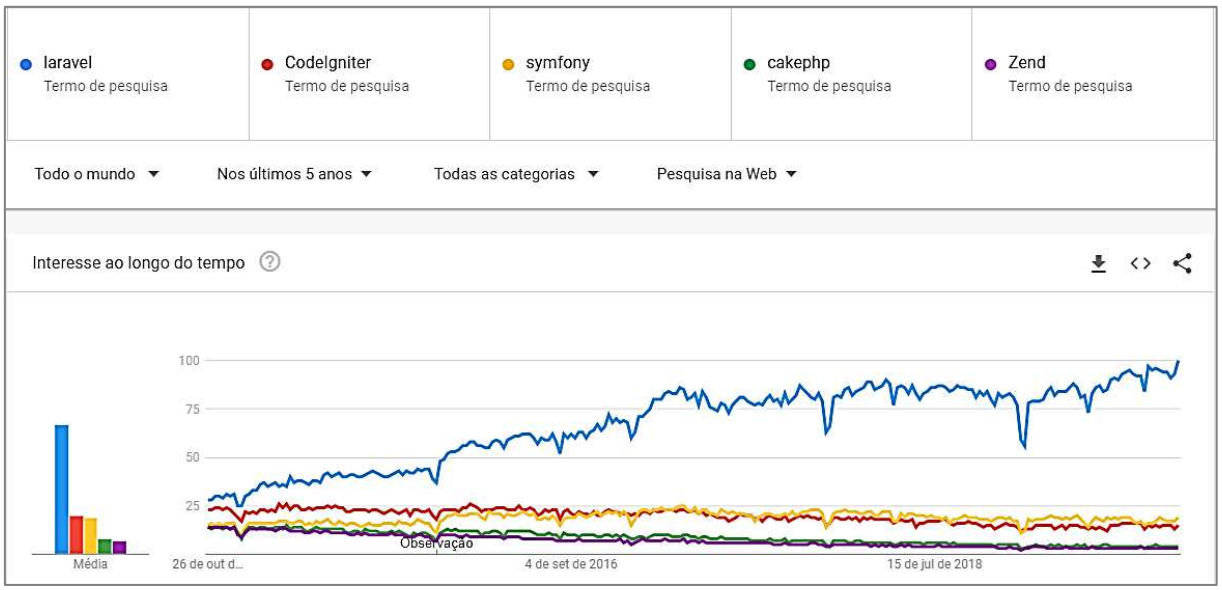
Para fomentar as boas práticas de programação e acelerar o desenvolvimento de sistemas de informação, além do uso de metodologias ágeis, faz-se também o uso de *frameworks*. Segundo Govoni (1999), *framework* é coleção abstrata de classes, interfaces e padrões dedicados a resolver uma classe de problemas através de uma arquitetura flexível e extensível.

Pode-se definir também *framework* como um conjunto de bibliotecas, ferramentas, guias, sistemas e componentes que servem para agilizar o processo de desenvolvimento de soluções. É usada como uma base onde o sistema de informação será construído (COUTO, 2006).

Os *frameworks* ajudam bastante às comunidades de desenvolvimento de *software* porque já trazem, prontas e encapsuladas muitos sistemas que são muito trabalhosos de implementar do zero e que precisam ser refeitos sempre.

A grande diversidade de linguagens e arquiteturas de *software* leva à existência de uma grande quantidade de *frameworks*. Para a linguagem PHP, umas das mais utilizadas no mundo é o Laravel, um *framework* livre e *open-source*. Por fornecer diversas ferramentas para o desenvolvedor e ser possuir uma baixa curva de aprendizado, o Laravel tornou-se extremamente popular, com uma gigantesca comunidade e aceitação no mercado.

Figura 5 - Tendências de alguns frameworks nos últimos 5 anos

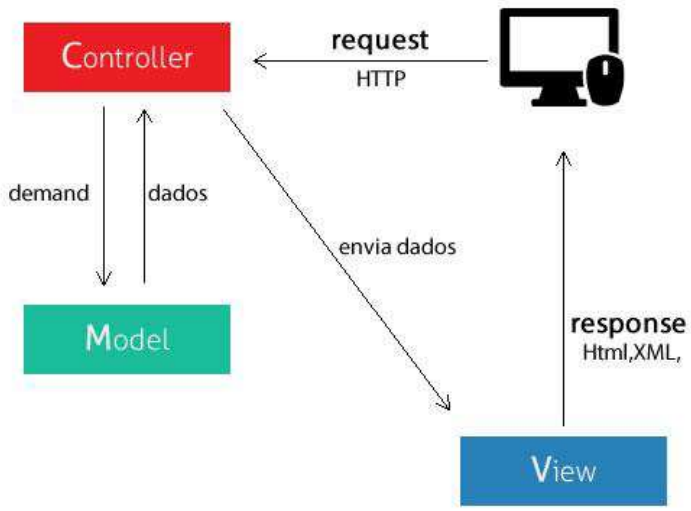


Fonte: Google Trends

O Laravel utiliza o MVC (*Model – view – controller*), padrão arquitetural de *software* para implementar interfaces com o usuário, separando a aplicação em três camadas. A camada de interação do usuário (*view*), a camada de manipulação dos dados (*model*) e a camada de controle (*controller*), esta última é a responsável por fazer a comunicação entre a *model* e a *view*.

Em suma, o funcionamento da arquitetura MVC é iniciado na *view*, onde o usuário envia uma requisição HTTP que será recebida e processada pela camada de controle e retornará uma ação, que geralmente é o resultado de uma consulta em um banco de dados ou retornar uma *view*. Tal relação pode ser observada na Figura 6.

Figura 6 - Arquitetura MVC



Fonte: RAMOS (2015).

É importante notar que o Laravel é um poderoso *framework back-end*, isto é, ele atua "por trás" da aplicação, sendo responsável por realizar todas as operações referentes às regras de negócio do sistema. Dessa forma, também existem *frameworks front-end*, que atuam na interatividade do usuário com o sistema, dando vida à aplicação e ajudando a melhorar a experiência do usuário.

Um *framework front-end* muito utilizado por grandes corporações é o Angular, criada e em constante desenvolvimento pela Google, o Angular é uma poderosa ferramenta para criação de aplicações *web*, *desktop* e *mobile*.

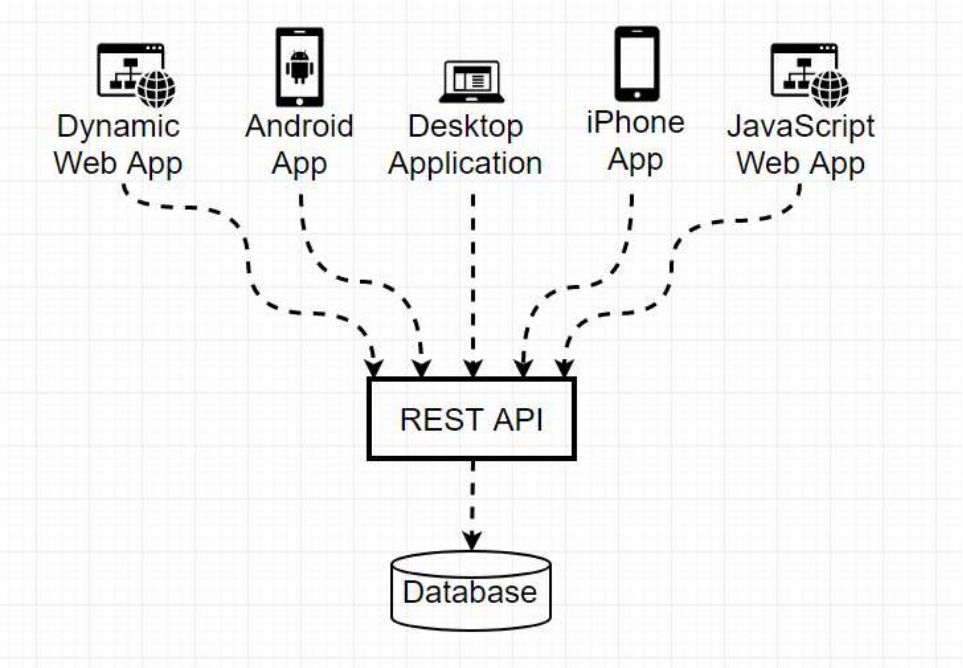
Utilizando-se de HTML, CSS e Typescript o Angular é capaz de prover um alto desempenho e portabilidade entre plataformas ao mesmo tempo em que provê alta produtividade ao desenvolvedor, disponibilizando de *templates*, CLI e IDEs. Em comparação com outros concorrentes, sua vantagem está na quantidade de recursos disponíveis ao desenvolvedor, não necessitando assim de bibliotecas extras durante o desenvolvimento, todos os recursos necessários estarão disponíveis no pacote do Angular.

Entretanto isso também leva a certa desvantagem, pois tal quantidade de recursos requer uma maior quantidade de espaço, tornando o Angular inadequado para criação de aplicações de pequena escala.

2.5 Arquitetura do sistema

REST é um estilo de arquitetura híbrido derivado de vários outros estilos arquiteturais baseados em rede. A REST conta com protocolos de comunicação sem estado, que podem ser armazenados em cache, para facilitar o desenvolvimento de aplicações *web*. Quando os princípios do REST são utilizados juntamente com um protocolo, como o HTTP, pode-se criar *web services* que serão capazes de compartilhar dados para qualquer dispositivo (DIAS, 2016) ou sistema *web* (como se pode observar na Figura 7, onde vários dispositivos acessam um API, que por sua vez realiza operações no banco de dados). Visando a escalabilidade e reuso de recursos, as aplicações com essa arquitetura são chamadas de APIs REST.

Figura 7 – API REST fornecendo dados a diversos dispositivos e aplicações.

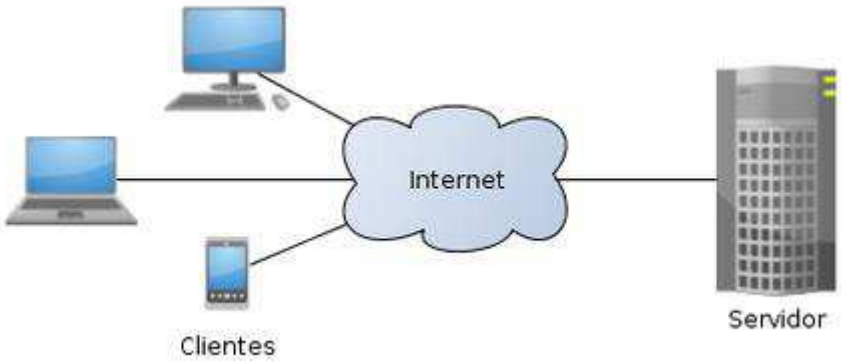


Fonte: Happy Coding (2018)

O estilo arquitetônico REST é equipado com um conjunto de restrições que tentam minimizar a latência e as comunicações em rede e, ao mesmo tempo, maximizam a independência e a escalabilidade de implementações de componentes. Conforme definido na dissertação de Roy Fielding (2000), as seis restrições da REST serão melhor descritas a seguir:

- Arquitetura cliente-servidor: O princípio por trás da restrição cliente-servidor é a divisão das responsabilidades. Como pode ser observado na Figura 8, ao separar as preocupações da interface do usuário das preocupações com o armazenamento de dados, aprimora-se a portabilidade da interface do usuário em várias plataformas e melhora-se a escalabilidade, simplificando os componentes do servidor. O ponto mais interessante dessa divisão de responsabilidades é que ela permite que os componentes evoluam de forma independente, dando suporte ao requisito de escalabilidade na internet em múltiplos domínios organizacionais;

Figura 8 – Arquitetura cliente-servidor.



Fonte: NEVES (2013)

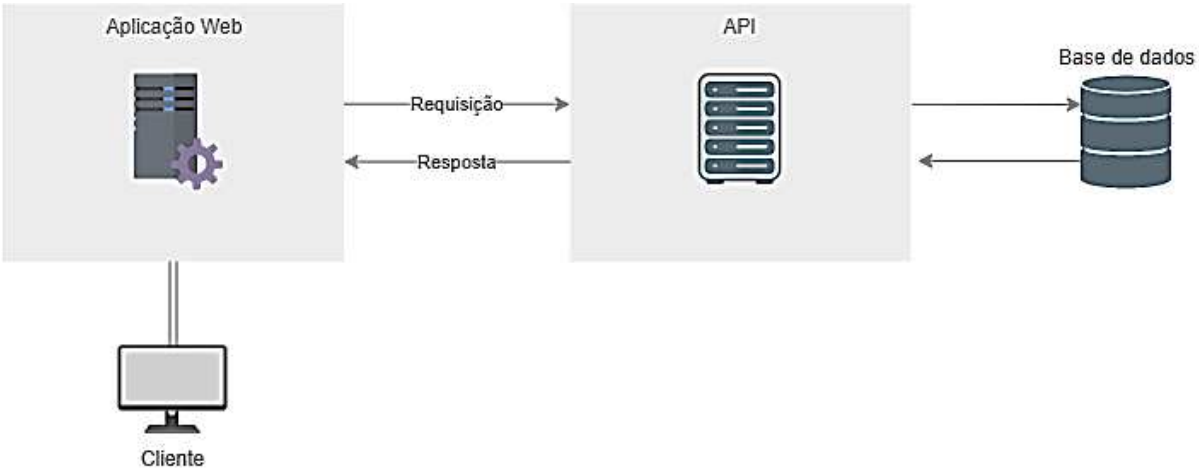
- Comunicação sem estado: Esta restrição é adicionada à comunicação cliente-servidor, onde a comunicação deve ser de natureza sem estado, de modo que cada solicitação do cliente para o servidor deve conter todas as informações necessárias para entender a solicitação e não pode tirar proveito de nenhum contexto armazenado no servidor. O estado da sessão é, portanto, mantido inteiramente no cliente;
- Cache: Para melhorar a eficiência da rede, adicionam-se restrições de cache. As restrições de cache exigem que os dados em uma resposta a uma solicitação sejam implicitamente ou explicitamente rotulados como armazenáveis em cache ou não. Se uma resposta for armazenável em cache, o cache do cliente terá o direito de reutilizar esses dados de resposta para solicitações posteriores equivalentes;
- Sistema em camadas: O estilo do sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas, restringindo o comportamento do componente, de modo que cada componente não possa "ver" além da camada imediata com a qual está interagindo. Ao restringir o conhecimento do sistema a uma única camada, limita-se a complexidade geral do sistema e promove-se a independência do substrato. As camadas podem ser usadas para encapsular serviços herdados e proteger novos serviços de clientes herdados, simplificando os componentes, movendo a funcionalidade usada com pouca frequência para um intermediário compartilhado. Os intermediários também podem ser usados para melhorar a escalabilidade do sistema, permitindo o balanceamento de carga de serviços em várias redes e processadores;
- Código sob demanda: O REST permite que a funcionalidade do cliente seja estendida baixando e executando o código na forma de *applets* ou *scripts*. Isso simplifica os clientes, reduzindo o número de recursos necessários para a pré-implementação.

Permitir o *download* de recursos após a implantação melhora a extensibilidade do sistema. No entanto, também reduz a visibilidade e, portanto, é apenas uma restrição opcional no REST;

- Interface uniforme: O recurso central que distingue o estilo arquitetural REST de outros estilos baseados em rede é a ênfase em uma interface uniforme entre os componentes. Ao aplicar o princípio de generalidade da engenharia de *software* à interface do componente, a arquitetura geral do sistema é simplificada e a visibilidade das interações é aprimorada. As implementações são dissociadas dos serviços que são prestados, o que incentiva a capacidade de evolução independente. A desvantagem, no entanto, é que uma interface uniforme degrada a eficiência, uma vez que as informações são transferidas de forma padronizada, e não específica para as necessidades de um aplicativo. A interface REST foi projetada para ser eficiente para a transferência de dados *hipermídia* de alta granularidade, otimizando para o caso comum da *web*, mas resultando em uma interface que não é ideal para outras formas de interação arquitetural.

Considerando o estilo arquitetônico REST, a aplicação a ser desenvolvida neste trabalho adotará tais restrições, com exceção da restrição de código sob demanda, pois não será necessário executar qualquer código ao lado do cliente. As APIs *web* que adotam as restrições de arquitetura da REST são chamadas de APIs *RESTful* (Red Hat). A Figura 9 monstra a arquitetura do sistema a ser desenvolvido, na qual nota-se o uso da API para a comunicação com a base de dados, e a aplicação *web* para se comunicar com a API e com o usuário final (cliente).

Figura 9 – Arquitetura do sistema



Fonte: Elaborado pelo autor (2019)

3 DESENVOLVIMENTO

O enfoque deste capítulo encontra-se na apresentação de conceitos que ajudarão a compreender o porquê da arquitetura e metodologia utilizada para resolver o problema proposto neste trabalho. Conforme já apresentado, o objetivo deste trabalho consiste em construir um sistema de acompanhamento de pacientes para o posto médico da UEMA.

Nesta seção serão descritos detalhadamente os procedimentos realizados, as ferramentas utilizadas e até a programação utilizada.

3.1 Desenvolvimento da aplicação

Tomando como base a arquitetura apresentada na seção 3.1, observa-se que o sistema do posto médico será desenvolvido em duas etapas:

- Desenvolvimento da aplicação *front-end*: A interface que fará a conexão entre o usuário e a API por meio de requisições, a qual retornará uma resposta com os dados, que serão processados e renderizado para serem apresentados ao usuário;
- Desenvolvimento da aplicação *back-end*: A API, que receberá requisições vindas da aplicação *front-end* e retornará as respostas no formato JSON. Esta aplicação será a responsável por manipular a base de dados.

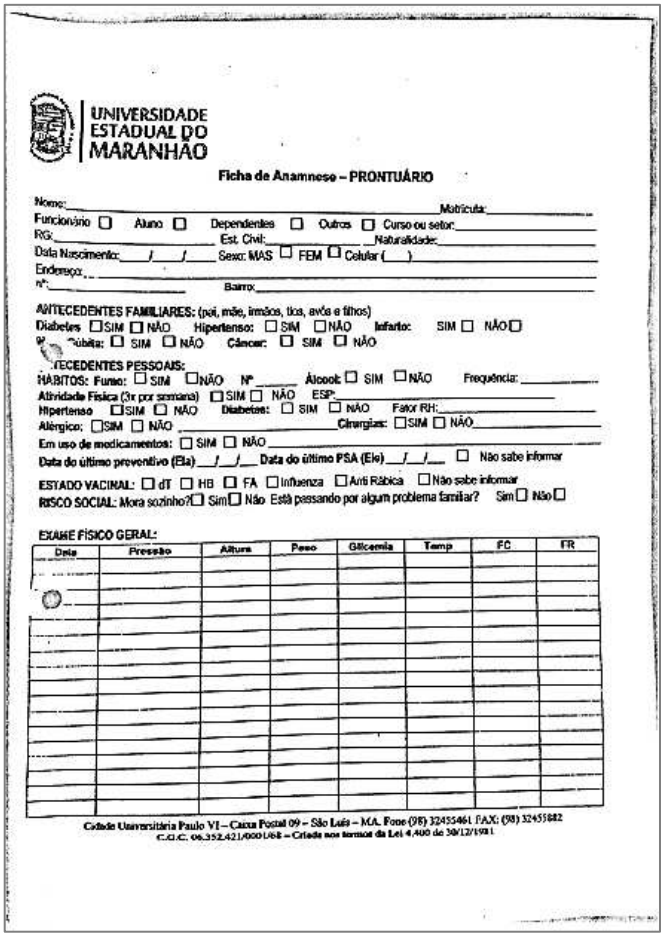
A metodologia apresentada na seção 2.3.2, o RAD foi a metodologia aplicada ao desenvolvimento do sistema proposto neste trabalho, portanto este capítulo tem como objetivo mostrar cada uma das suas etapas aplicadas a seguir.

3.1.1 Comunicação

Esta é a etapa mais importante, pois o completo entendimento sobre as necessidades do usuário é o fator chave para uma boa modelagem do sistema. Esta fase se iniciou com uma longa reunião com os funcionários-alvo do posto médico, que são os médicos, atendente, enfermeiros e coordenador.

Nessa reunião foram levantadas todas as funcionalidades que o sistema deveria apresentar, assim como os dados referentes ao sistema, esta fase foi auxiliada pelo uso da ficha de anamnese que é utilizada para guardar os registros dos pacientes atendidos pelo posto médico (Figura 10 e 11).

Figura 10 – Ficha de Anamnese do posto médico vista de frente.



UNIVERSIDADE ESTADUAL DO MARANHÃO

Ficha de Anamnese – PRONTUÁRIO

Nome:_____ Matrícula:_____

Funcionário ☐ Aluno ☐ Dependentes ☐ Outros ☐ Curso ou setor:_____

RG:_____ Est. Civil:_____ Naturalidade:_____

Data Nascimento:___/___/___ Sexo: MAS ☐ FEM ☐ Celular (___)_____

Endereço:_____

nº:_____ Bairro:_____

ANTECEDENTES FAMILIARES: (pai, mãe, irmãos, tios, avós e filhos)

Diabetes ☐SIM ☐NÃO Hipertenso: ☐SIM ☐NÃO Infarto: SIM ☐ NÃO☐

[illegible] Súbita: ☐ SIM ☐ NÃO Câncer: ☐ SIM ☐ NÃO

ANTECEDENTES PESSOAIS:

HÁBITOS: Fumo: ☐ SIM ☐NÃO Nº_____ Álcool: ☐ SIM ☐NÃO Frequência: _____

Atividade Física (3x por semana) ☐SIM ☐ NÃO ESP:_____

Hipertenso ☐SIM ☐ NÃO Diabetes: ☐ SIM ☐ NÃO Fator RH:_____

Alérgico: ☐SIM ☐ NÃO _____ Cirurgias: ☐SIM ☐ NÃO_____

Em uso de medicamentos: ☐ SIM ☐ NÃO_____

Data do último preventivo (Ela) ___/___/___ Data do último PSA (Ele) ___/___/___ ☐ Não sabe informar

ESTADO VACINAL: ☐ dT ☐ HB ☐ FA ☐Influenza ☐Anti Rábica ☐Não sabe informar

RISCO SOCIAL: Mora sozinho? ☐ Sim☐ Não Está passando por algum problema familiar? Sim☐ Não☐

EXAME FÍSICO GERAL:

Data	Pressão	Altura	Peso	Glicemia	Temp	FC	FR

Cidade Universitária Paulo VI – Caixa Postal 09 – São Luís – MA. Fone (98) 32455461 FAX: (98) 32455882
C.G.C. 06.352.421/0001-68 – Criada nos termos da Lei 4.400 de 30/12/1981

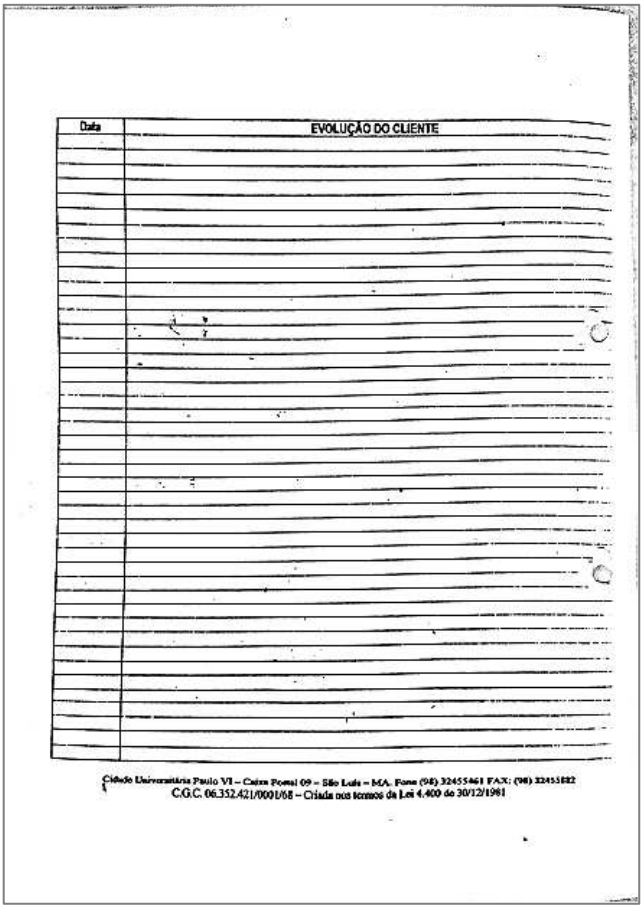
Fonte: Posto Médico – UEMA (2019)

A ficha de Anamnese é dividida em 4 seções que são preenchidas sequencialmente uma após a conclusão da anterior. Na primeira são coletados os dados pessoais do paciente como nome, estado civil, endereço, sexo, etc. A segunda são coletados o histórico familiar e pessoal, onde é verificado se o paciente ou algum familiar próximo já teve algumas possíveis doenças, como hipertensão, diabetes, alergia, etc., também é verificado alguns hábitos do paciente, como praticar atividade física, fumar, consumir bebida alcóolica, etc.

As duas primeiras seções são preenchidas pelo atendente, a terceira seção é preenchida pelo enfermeiro, com base no exame físico a ser realizado, onde são coletados dados do paciente no momento da consulta, esses dados são: data do atendimento, pressão, altura, peso, glicemia, temperatura, frequência cardíaca e frequência respiratória.

A última seção (Figura 11) será preenchida pelo médico, onde o mesmo fará uma avaliação do estado do paciente, registrando a data e a descrição da evolução do mesmo.

Figura 11 – Verso da ficha de Anamnese do posto médico.



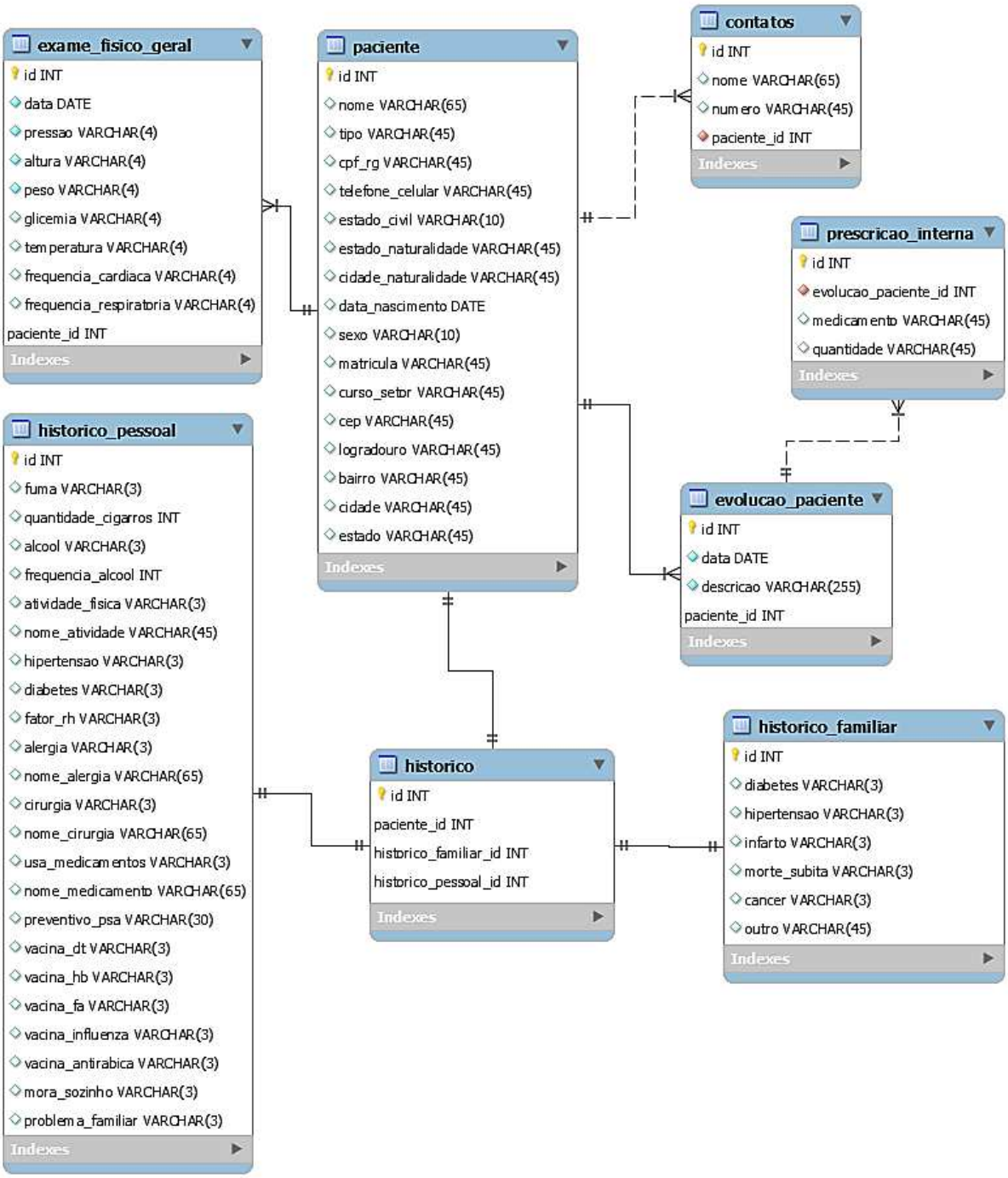
Fonte: Posto Médico – UEMA (2019)

Todo o entendimento que se obteve sobre o sistema nesta fase será utilizado para as próximas etapas.

3.1.2 Planejamento

Esta etapa tem como objetivo fazer a modelagem do banco de dados considerando-se os dados da ficha de anamnese (Figura 10 e 11) assim como os requisitos levantados na etapa anterior, tal modelagem foi realizada utilizando o modelo de dados entidade relacionamento (ER) e a ferramenta para a elaboração do mesmo foi o MySQL Workbench 8.0 CE.

Figura 12 – Modelagem do banco de dados



Fonte: Elaborado pelo autor (2019)

Como se pode observar na Figura 12, a modelagem do banco de dados conta com 8 tabelas, relacionadas entre si, onde a tabela principal é a de paciente, que é onde serão salvos todos os dados pessoais do mesmo, esta está relacionada com a tabela de exame físico geral e contatos em uma relação de um para muitos, ou seja, um paciente poderá ter muitos exames físicos e muitos contatos.

A tabela de paciente também está relacionada com a tabela pivô de histórico, em uma relação de um para um, ou seja, um paciente terá apenas um histórico. Como se pode observar, o histórico médico foi dividido em duas tabelas, são elas: histórico familiar e histórico pessoal, onde se juntando as suas chaves primárias, faz-se uma única correspondência na tabela pivô.

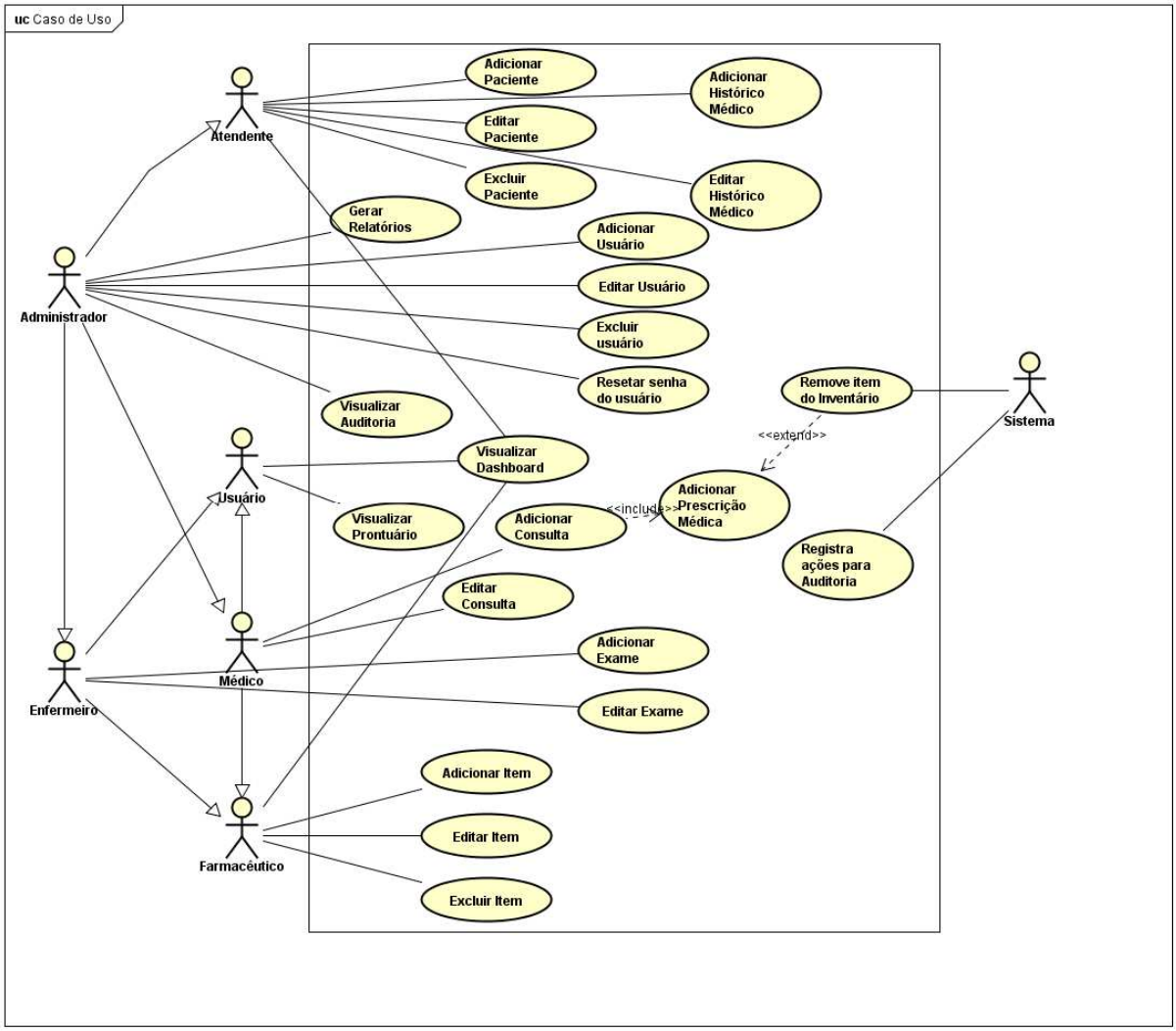
Por sua vez, a tabela de paciente estará relacionada com a tabela de evolução do paciente em uma relação de um para muitos, ou seja, um paciente poderá ter muitas evoluções. Nota-se também que a tabela de evolução possui uma relação facultativa de muitos para muitos com a tabela de prescrição interna, pois nem todo paciente necessitará de uma prescrição interna após a consulta.

3.1.3 Modelagem do processo

Nesta etapa os objetos de dados definidos na fase anterior são transformados para seguir o fluxo necessário para implementar uma função do negócio. As descrições do processamento são criadas para adicionar, modificar, descartar ou recuperar um objeto de dados. Sua representação é feita através de diagramas de atividade, e a ferramenta utilizada foi o Astah Community 7.0.0/846701.

Para auxiliar a modelagem de processo, foi criado também um diagrama de caso de uso (Figura 13), onde é demostrado todas as ações dos usuários no sistema, assim como algumas ações que o sistema realizará automaticamente sem que o usuário solicite.

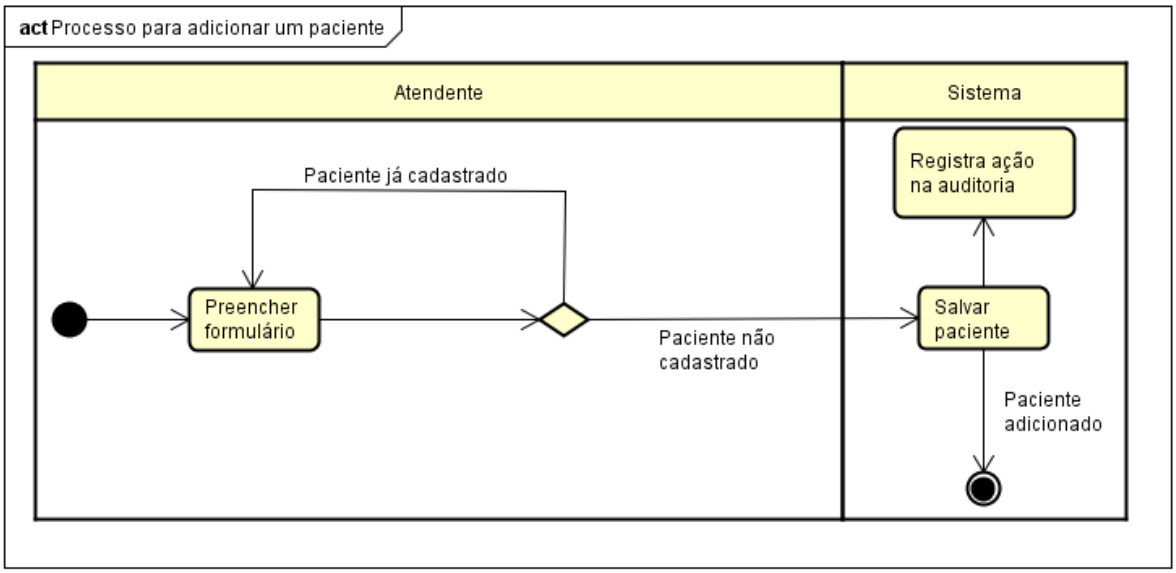
Figura 13 – Diagrama de caso de uso



Fonte: Elaborado pelo autor (2019)

No diagrama de caso de uso (Figura 13) podem-se ver todos os papéis (ações) que os atores (usuários) irão desempenhar no sistema, também se pode notar que há um ator chamado “sistema”, esse é uma representação literal do sistema, que realizará algumas ações sem que o usuário solicite, por exemplo: ao se editar os dados de um paciente, o sistema automaticamente salvará na tabela de auditoria o usuário que realizou a operação, assim como os dados referentes à própria operação (tabela modificada, campos alterados, etc). Todos os processos descritos abaixo levam em consideração que o usuário já está efetuou o *login* no sistema.

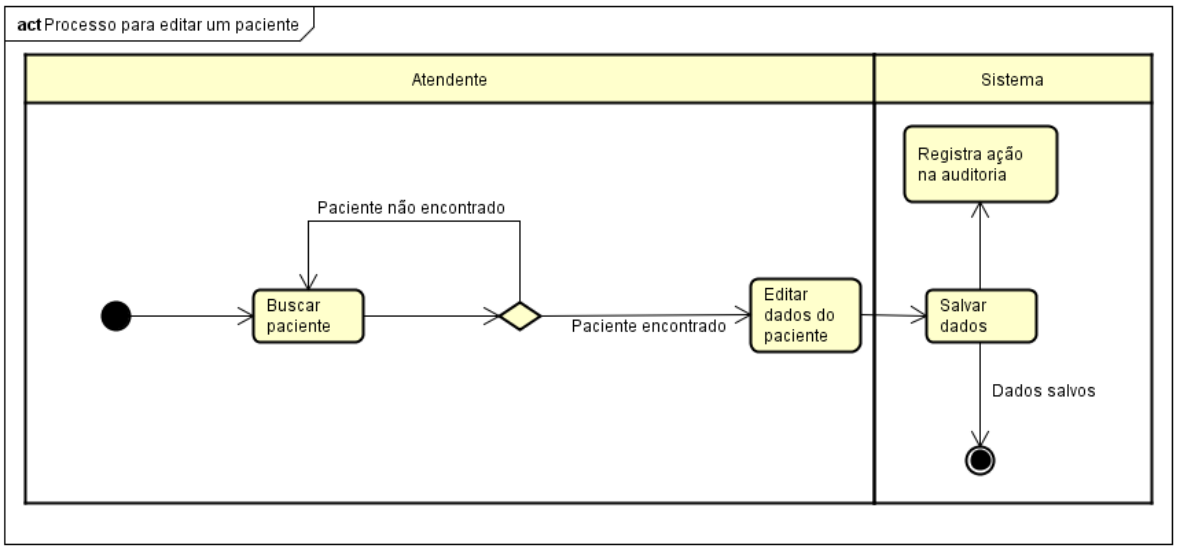
Figura 14 – Processo para adicionar um paciente



Fonte: Elaborado pelo autor (2019)

No diagrama mostrado na Figura 14, o atendente irá preencher o formulário de cadastro de paciente, o campo CPF é único, portanto, caso o atendente preencha o campo com um CPF de um paciente já cadastrado, o mesmo permanecerá na tela de cadastro e o sistema enviará uma mensagem lhe informando do ocorrido. Caso o paciente não esteja cadastrado ainda, os dados serão salvos no banco e o sistema registrará essa ação na tabela de auditoria do sistema.

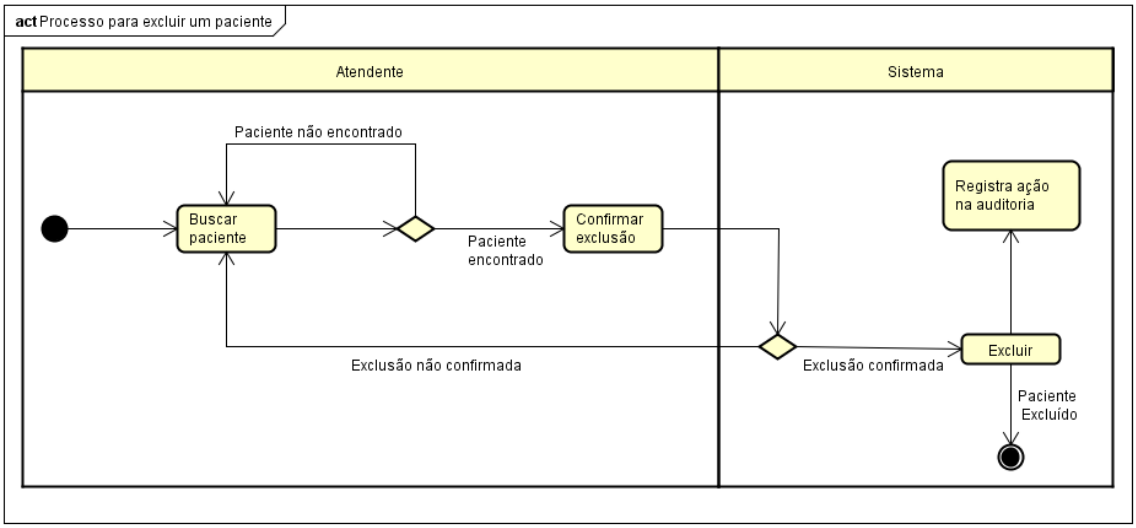
Figura 15 – Processo para editar os dados pessoais de um paciente



Fonte: Elaborado pelo autor (2019)

Para se editar dados de um paciente (Figura 15), o atendente irá pesquisar um usuário, alterar os campos desejados e salvar o formulário, após isso o sistema registrará essa ação na tabela de auditoria do sistema.

Figura 16 – Processo para excluir um paciente

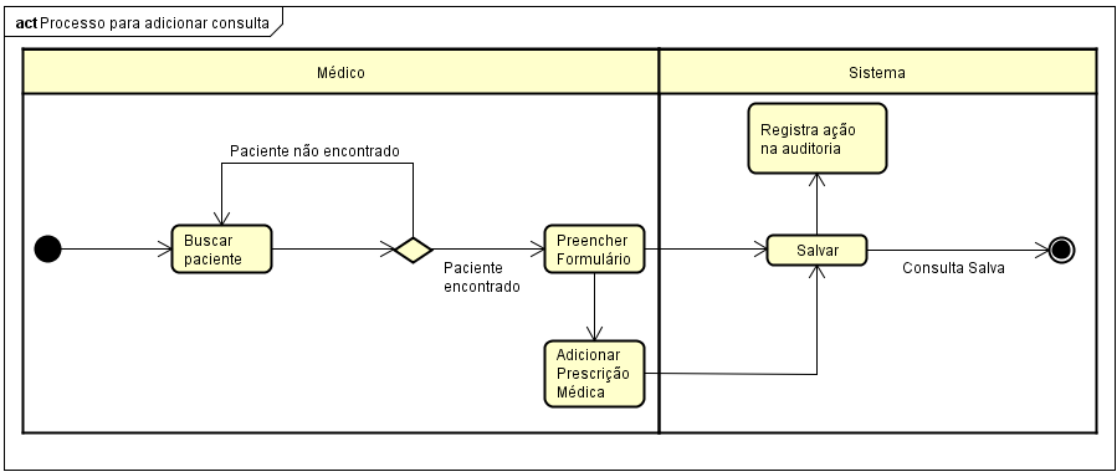


Fonte: Elaborado pelo autor (2019)

O processo para se excluir um paciente se dará pela localização do paciente alvo, logo após será solicitado uma confirmação de exclusão para impedir que o atendente exclua um paciente por acidente, após a confirmação, o sistema removerá o paciente e registrará essa ação na tabela de auditoria.

Todos os outros processos do sistema referentes às ações de se adicionar (Figura 14), editar (Figura 15) e excluir (Figura 16) são de forma análoga aos apresentados acima, portanto, a seguir serão apresentados apenas os processos relacionados às ações que apesar de serem parecidas, se diferenciam em algum ponto das acima.

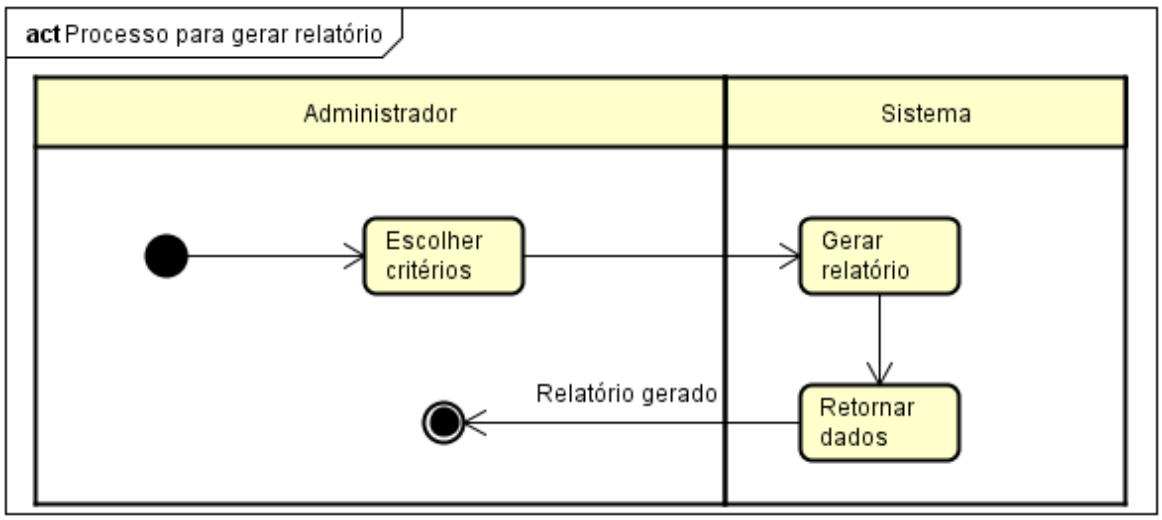
Figura 17 – Processo para se adicionar consulta a um paciente



Fonte: Elaborado pelo autor (2019)

A Figura 17 representa o processo para se adicionar uma consulta à um paciente que já está cadastrado, como pode-se perceber, ao preencher o formulário, o médico também tem a possibilidade de adicionar uma prescrição interna para o paciente. No final do processo o sistema irá salvar a consulta e registrará a ação na tabela de auditoria.

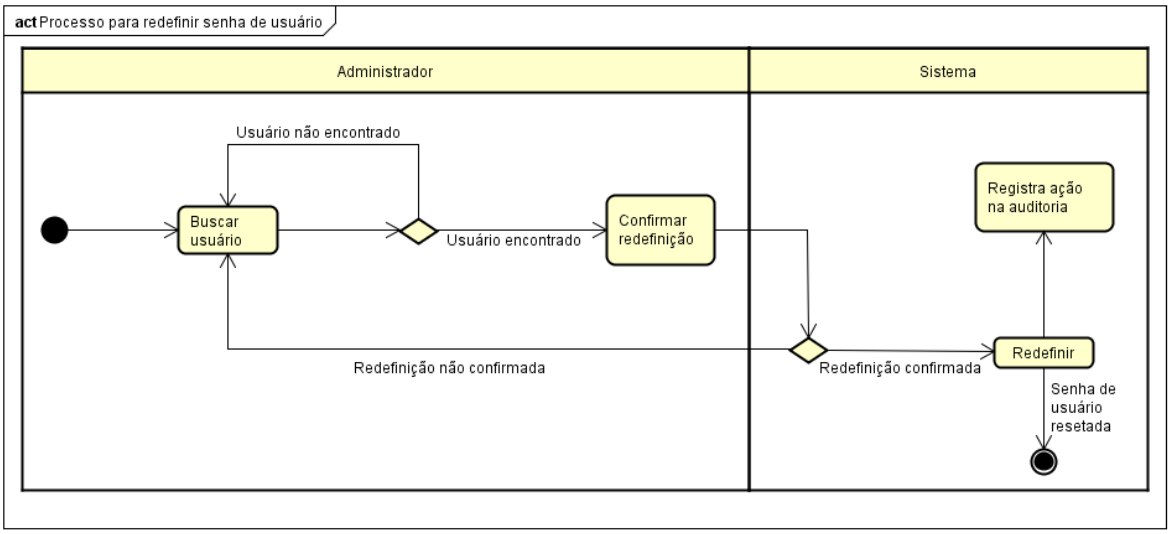
Figura 18 – Processo para gerar relatório



Fonte: Elaborado pelo autor (2019)

O processo para se gerar relatório (Figura 18) é muito simples, o usuário administrador só precisará escolher os critérios que desejar e o sistema irá retornar os dados que serão apresentados na tela para o mesmo. Esta ação não requer que o sistema salve quaisquer dados, apenas serão retornados os dados requisitados pelo usuário.

Figura 19 – Processo para se redefinir a senha de um usuário



Fonte: Elaborado pelo autor (2019)

Para se redefinir uma senha (Figura 19), o usuário administrador irá encontrar o usuário desejado e redefinir sua senha, mas para isso o sistema irá pedir uma confirmação, para que diminua a possiblidade do administrador resetar a senha de um usuário por engano.

Caso o administrador confirme, a senha do usuário será resetada, e o sistema irá registrar na tabela de auditoria tal ação.

3.1.4 Construção

Esta fase tem como objetivo a construção da aplicação. Como foi citado na seção 3.2.1, a aplicação é dividida em *front-end* e *back-end*, portanto, esta seção tem como objetivo mostrar a construção de cada uma delas, sendo dividida em: (1) Desenvolvimento da aplicação *front-end*; (2) Desenvolvimento da aplicação *back-end* e (3) Integração.

3.2.4.1 Desenvolvimento da aplicação *front-end*

3.2.4.1.1 Arquitetura do Angular

Como explicado na seção 2.4, o Angular é uma plataforma e estrutura para a criação de aplicativos *front-end* em HTML e TypeScript. O Angular é escrito em TypeScript e implementa as funcionalidades principais e opcionais como um conjunto de bibliotecas TypeScript importadas para o projeto.

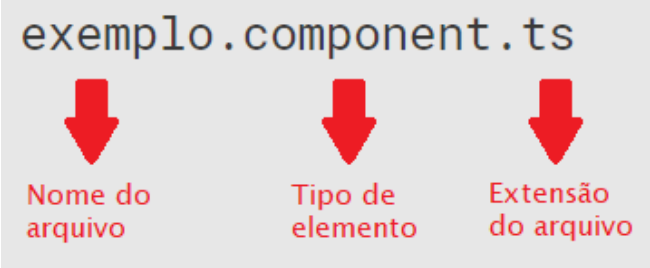
Os blocos de construção básicos de um aplicativo Angular são módulos NgModules, que fornecem um contexto de compilação para os componentes. Tais módulos coletam código relacionado em conjuntos funcionais; um aplicativo Angular é definido por um conjunto de NgModules. Um aplicativo sempre tem pelo menos um módulo raiz que habilita a inicialização.

Tanto os componentes quanto os serviços são simplesmente classes, com *decorators* que marcam seu tipo e fornecem metadados que informam ao Angular como usá-los.

Os componentes de um aplicativo geralmente definem muitas visualizações, organizadas hierarquicamente. O Angular fornece o *router service* (serviço de roteamento) para ajudar a definir as rotas de navegação entre as páginas.

Os elementos estruturais do Angular seguem um padrão, onde o elemento se inicia com seu nome, seguido do nome do tipo de elemento do Angular e finalizado com a extensão do arquivo, como pode-se observar na Figura 20.

Figura 20 – Padrão para nomenclatura de elementos do Angular



Fonte: Elaborado pelo autor (2019)

Para melhor organização, os elementos do tipo componente podem ser segmentados em 3 arquivos (Figura 21), onde a extensão irá indicar sua natureza.

Figura 21 – Exemplo de um componente segmentado

```
|-- exemplo
    |-- exemplo.component.html
    |-- exemplo.component.scss
    |-- exemplo.component.ts
    |-- exemplo.service.ts
    |-- exemplo.ts
```

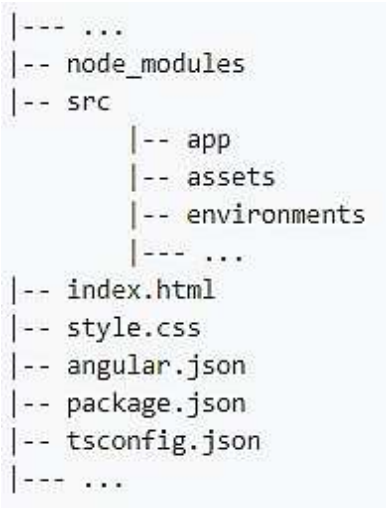
Fonte: Elaborado pelo autor (2019)

3.2.4.1.2 Projeto

Como o objetivo principal da metodologia adotada neste trabalho é o desenvolvimento rápido, além do uso de um *framework* (Angular), esta aplicação também foi projetada com a contribuição de um *admin template*. As tecnologias utilizadas nesta etapa foram:

- NodeJS v 10.2;
- Angular 8.02;
- Angular CLI 8.0.4;
- Paper Dashboard Angular 2.0.0.

Figura 22 - Estrutura do Projeto

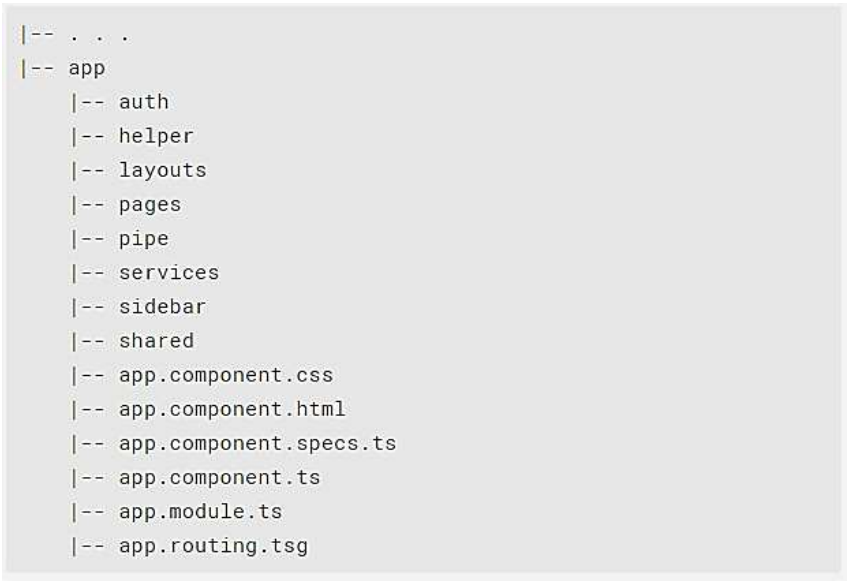


Fonte: Elaborado pelo autor (2019)

A estrutura do projeto (Figura 22) representa como os diretórios e arquivos importantes estão dispostos ao longo do projeto, nota-se também a presença de pontilhados (...), os mesmos representam os diretórios e arquivos que não são relevantes ao desenvolvimento, pois fazem parte do padrão do *framework* adotado e não foram alterados durante todo o processo de desenvolvimento.

Nesta seção será comentado apenas o núcleo da aplicação, que são os códigos-fonte que estão dentro do diretório **app**, entretanto, a documentação completa desta aplicação estará disponível em: https://github.com/alexilallas/Posto-Medico-com-Angular-8. A estrutura do diretório **app** é monstrada na figura a seguir.

Figura 23 – Estrutura do diretório app



Fonte: Elaborado pelo autor (2019)

O núcleo da aplicação foi organizado em vários diretórios, o primeiro a ser descrito é o **auth,** neste diretório se encontram outros dois: **guard** e **login** (Figura 24).

O diretório **guard** é o responsável por conter a classe responsável pela guarda de rotas (auth.guard.ts) , sua classe faz a verificação se o usuário está logado, e caso esteja, verifica também se o usuário tem permissão para acessar a rota solicitada. Caso o usuário esteja logado e tenha permissão para acessar a rota, será direcionado para a tela de início, caso contrário, será direcionado para a tela de *login*.

Figura 24 – Organização do diretório auth

```
|-- . . .
|-- app
    |-- auth
        |-- guard
            |-- auth.guard.ts
        |-- login
            |-- login.component.html
            |-- login.component.scss
            |-- login.component.ts
            |-- login.service.ts
            |-- login.ts
            |-- reset-password.ts
    |-- . . .
```

Fonte: Elaborado pelo autor (2019)

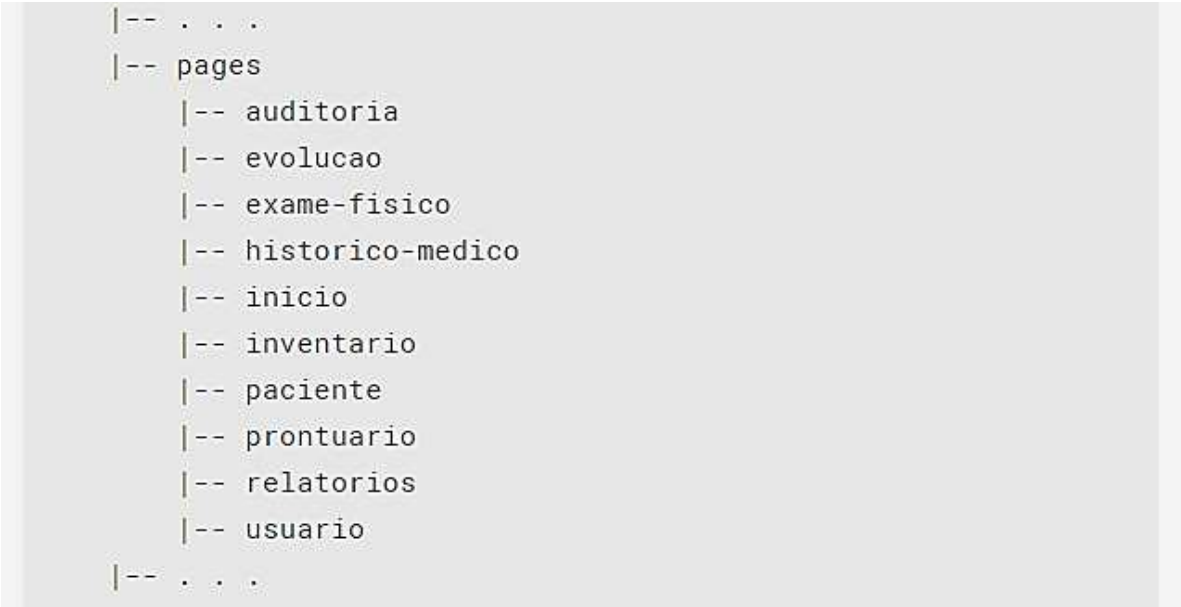
Já o diretório **login** irá conter todo o código fonte referente à tela de *login*.

- login.component.html**:** Contém a marcação HTML;
- login.component.scss**:** Contém os estilos CSS personalizados para a página HTML de seu componente;
- login.component.ts: É o componente de *login*, contém todo o código-fonte que será responsável pelo *login* do usuário no sistema, assim como a redefinição de senha de usuário;
- login.service.ts: É o serviço que será consumido à priori pelo componente login. Esta é a classe que fará as requisições HTTP referentes ao *login* para a API;
- login.ts: Este é o modelo de dados, utilizado pelo formulário de *login*;

- reset-password.ts: Este é o modelo de dados, utilizado pelo formulário de redefinição de senha.

O próximo diretório é o **helper**, este contém apenas um serviço com alguns métodos que serão reutilizados em várias partes do projeto, portanto é um serviço apenas de ajuda.

Figura 25 – Organização do diretório pages



O diretório **pages** (Figura 25) contém todos os componentes que serão utilizados como páginas do sistema, cada subdiretório contém um componente que representará uma página no sistema, e cada um segue o padrão monstrado na Figura 21, no qual os elementos exemplo.ts, exemplo.component.html e exemplo.component.scss serão facultativos, os componentes nesse diretório serão descritos a seguir.

O componente **auditoria** irá mostrar os dados da tabela de auditoria do sistema, através de requisições realizadas com o serviço de auditoria.

Figura 26 - Componente auditoria

Fonte: Autor (2019)

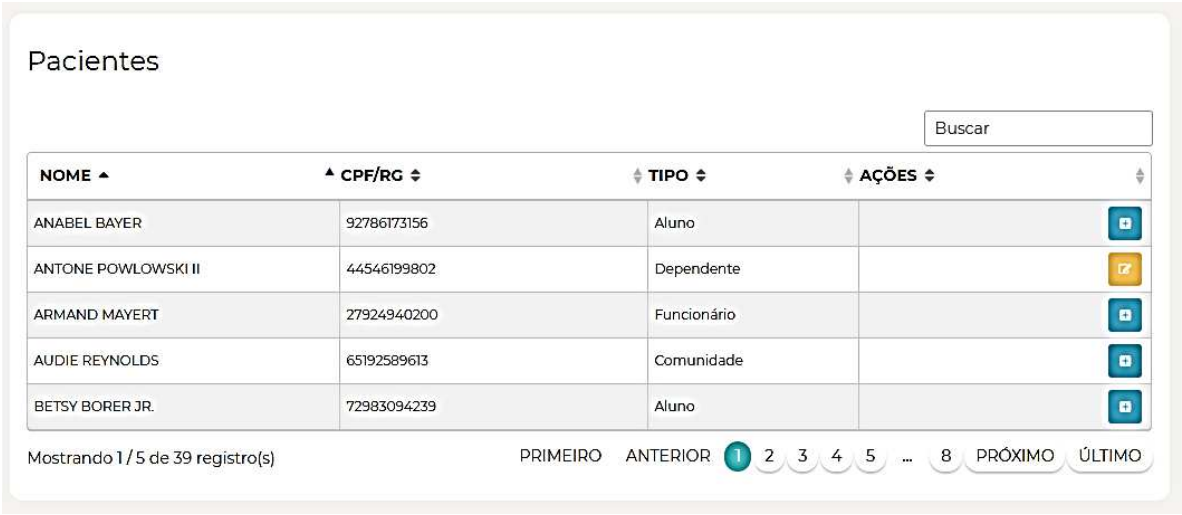
O componente **evolucao** irá mostrar as consultas realizadas pelos médicos do posto, a qual terá opções na tabela de se adicionar uma nova consulta ou editar uma existente, como se pode observar na Figura 27.

Figura 27 – Componente evolucao

Fonte: Autor (2019)

O componente **exame-fisico** também será da mesma forma da Figura 27, com a diferença de que os botões de adicionar e editar serão voltados para o exame físico realizado pelo enfermeiro. E o componente **histórico-medico** também será de forma similar ao componente anterior, com a diferença que não se pode criar mais de um histórico, apenas editar, caso já exista (Figura 28).

Figura 28 – Componente histórico-medico



Fonte: Autor (2019)

O componente **inicio** apenas mostrará alguns resumos para o usuário em forma de gráfico e quantitativamente (Figura 29), será a primeira tela do sistema após o usuário realizar *login*.

Figura 29 – Componente inicio

Fonte: Autor (2019)

O componente **inventario** mostrará os itens cadastrados e terá opções de cadastrar um item novo, editar ou excluir um existente (Figura 30).

Figura 30 – Componente inventario

Fonte: Autor (2019)

O componente **paciente** será o responsável pelo cadastro de novos pacientes, edição e exclusão de seus dados (Figura 31).

Figura 31 – Componente paciente

Fonte: Autor (2019)

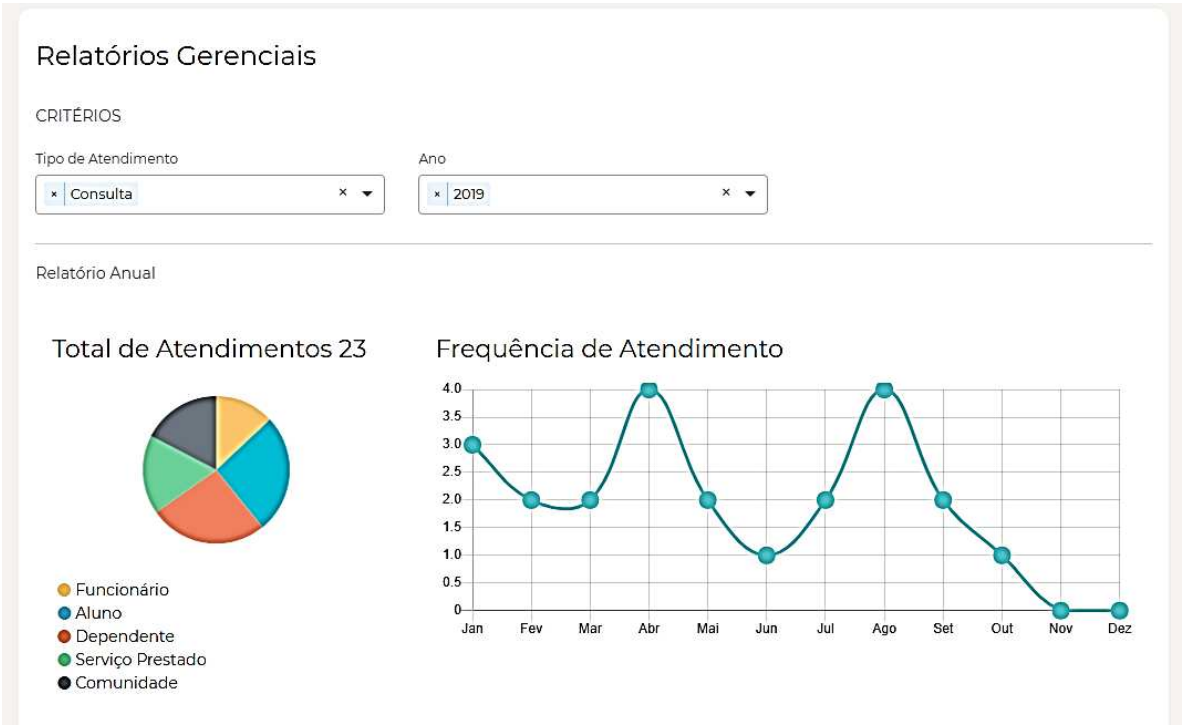
O componente **prontuario** mostrará apenas os pacientes cadastrados e um botão para se visualizar a ficha de cada paciente (Figura 32). Se o paciente possuir consultas ou exames, serão mostrados no prontuário.

Figura 32 – Componente prontuario

Fonte: Autor (2019)

O componente **relatorios** será o responsável por gerar gráficos e tabelas em forma de relatórios com base nos critérios escolhidos pelo usuário (Figura 33).

Figura 33 – Componente relatorios



Fonte: Autor (2019)

O componente **usuario** será o responsável pelo gerenciamento de usuários do sistema, onde se poderá cadastrar, editar, excluir ou redefinir a senha de um usuário (Figura 34).

Figura 34 – Componente usuario

Fonte: Autor (2019)

Voltando à estrutura do diretório **app** (Figura 23), o diretório **sidebar** conterá um componente e um módulo para exportá-lo, este componente será o responsável pelo menu lateral do sistema (Figura 35).

Figura 35 – Componente sidebar

Fonte: Autor (2019)

O diretório **pipe** conterá o filtro que será utilizado para filtrar os itens do componente **sidebar** de acordo com as permissões de cada usuário, por exemplo: o usuário administrador poderá ver todas as telas do sistema, mas o usuário com perfil de farmacêutico só poderá ver a

tela inicial e de inventário, portanto apenas os itens “início” e “inventário” deverão aparecer para o mesmo na **sidebar**.

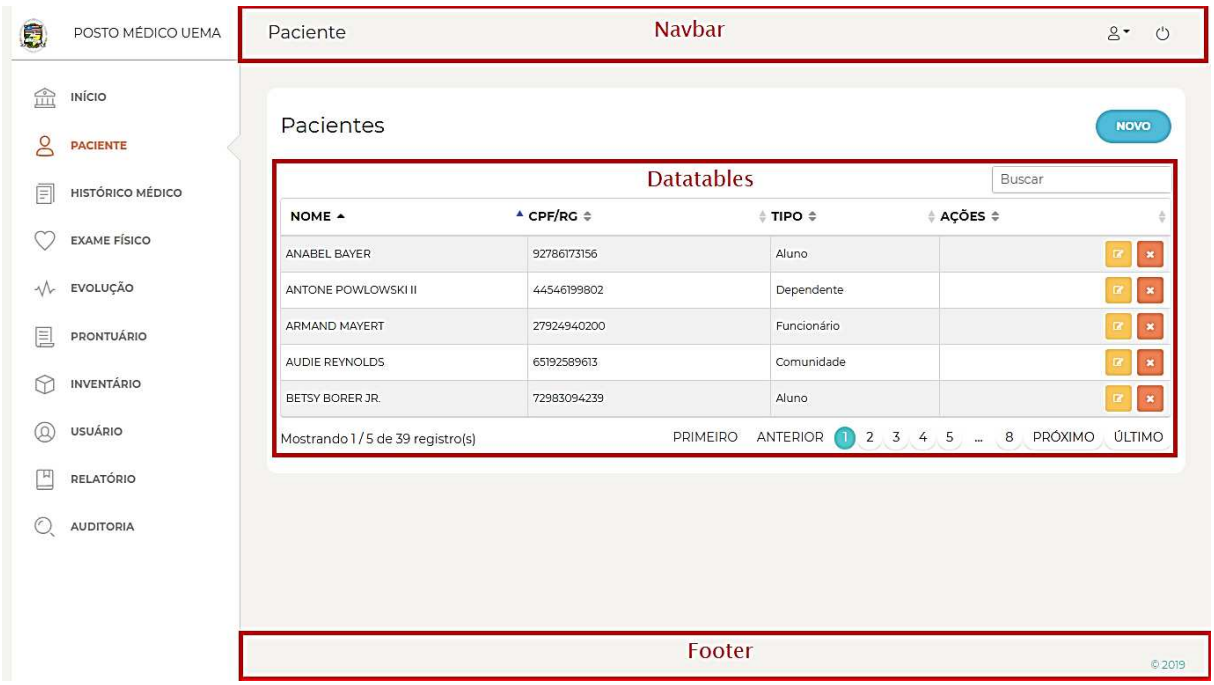
O diretório **shared** (Figura 36) contém os componentes que serão utilizados por praticamente todos os outros componentes, são eles o componente datatables, utilizado para renderização de todas as tabelas; footer, o rodapé de todas as páginas e navbar, a barra de navegação que é utilizada em todas as páginas também Figura (37).

Figura 36 – Diretório shared

```
|-- . . .
|-- shared
    |-- datatables
    |-- . . .
    |-- footer
    |-- navbar
```

Fonte: Autor (2019)

Figura 37 – Componentes do diretório shared dispostos na tela



Fonte: Autor (2019)

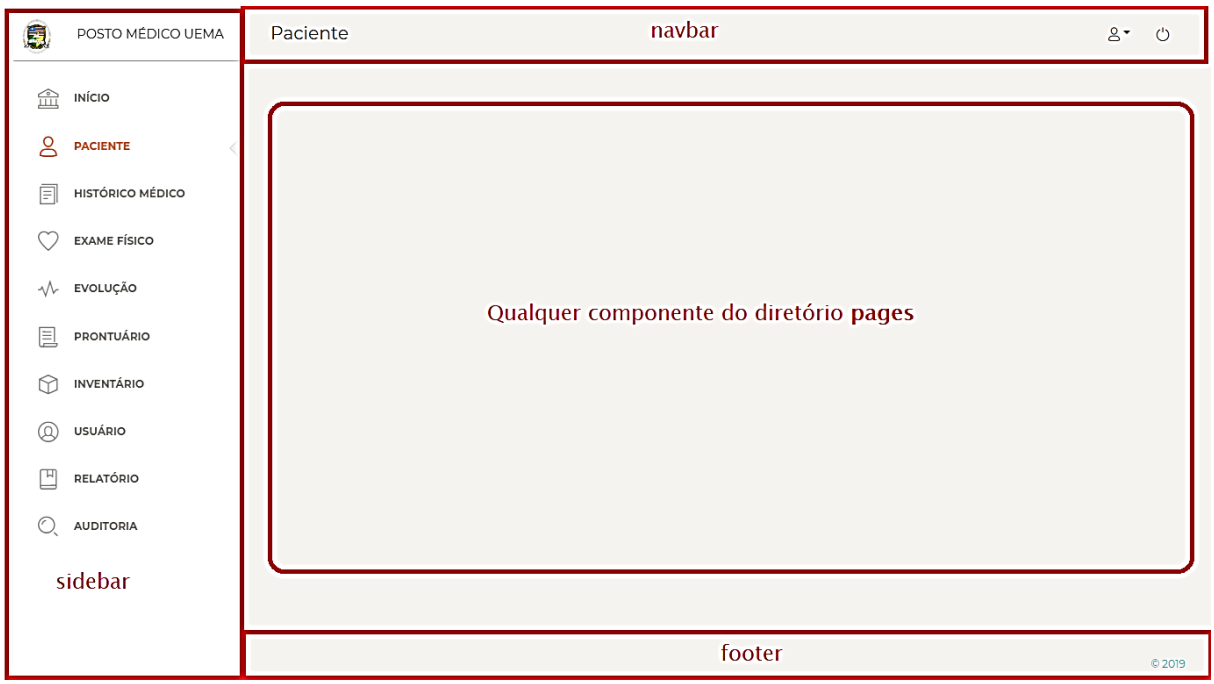
O diretório **services** irá conter alguns serviços que serão utilizados por todos os componentes, são eles o serviço de mensagem (Figura 38) e o serviço de interceptação, o qual todas as requisições HTTP, antes de irem para a API, serão tratadas de acordo com sua natureza.

Figura 38 – Serviço de mensagens

Fonte: Autor (2019)

Por último, o diretório **layouts** será o componente responsável por agregar os componentes do diretório **pages**, **shared** e **sidebar** para que sejam mostrados como uma página única, como pode-se observar na Figura 39.

Figura 39 – Disposição dos componentes no sistema



Fonte: Autor (2019)

Como se pode perceber, o componente **sidebar** será o menu lateral, alinhado à esquerda da página onde terão todos os links para acessar os componentes que estão em **pages**. O componente **navbar** será a barra de navegação, onde ficará o nome do componente que está sendo acessado à esquerda, e os botões para informações do usuário logado e o botão de *logout* do sistema. O componente **footer** mostra apenas o ano de desenvolvimento do sistema. E por último, a área central da Figura 39 é onde será mostrado o conteúdo de cada componente que será acessado através da **sidebar**.

Dentro do diretório **layouts** também contém um arquivo de rotas (admin-layouts.routing.ts) responsável por atribuir um caminho (path) à cada componente de **pages** (Figura 40).

Figura 40 – Rotas presentes em admin-layouts.routing.ts

```
export const AdminLayoutRoutes: Routes = [
    { path: 'inicio',           component: InicioComponent,          canActivate: [AuthGuard] },
    { path: 'paciente',         component: PacienteComponent,        canActivate: [AuthGuard] },
    { path: 'historico-medico', component: HistoricoMedicoComponent, canActivate: [AuthGuard] },
    { path: 'exame-fisico',     component: ExameFisicoComponent,     canActivate: [AuthGuard] },
    { path: 'evolucao',         component: EvolucaoComponent,        canActivate: [AuthGuard] },
    { path: 'prontuario',       component: ProntuarioComponent,      canActivate: [AuthGuard] },
    { path: 'inventario',       component: InventarioComponent,      canActivate: [AuthGuard] },
    { path: 'usuario',          component: UsuarioComponent,         canActivate: [AuthGuard] },
    { path: 'auditoria',        component: AuditoriaComponent,       canActivate: [AuthGuard] },
    { path: 'relatorio',        component: RelatorioComponent,       canActivate: [AuthGuard] },
];
```

Fonte: autor (2019)

Como se pode ver, a união desses 4 componentes individuais (sidebar, navbar, footer e qualquer componente do diretório pages) formarão uma página única a ser visualizada para o usuário que está acessando o sistema. O diretório **layout** também conta com um módulo para que esses componentes possam ser exportados como um componente único.

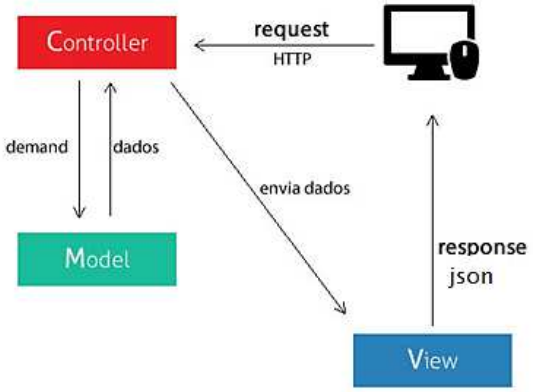
Voltando à estrutura do diretório mostrado na Figura 23, nota-se que o diretório raiz **app** contém um módulo (app.module.ts), este é o módulo raiz que responsável por importar todos os outros módulos presentes dentro dos subdiretórios em **app** para que sejam acessados em index.html, onde por sua vez, serão exportados hierarquicamente para que sejam renderizados na tela em forma de página *web*.

3.2.4.2 Desenvolvimento da aplicação *back-end*

3.2.4.2.1 Arquitetura do Laravel

Como já foi explicado na seção 2.4, o Laravel utiliza-se da arquitetura MVC (Figura 6) para a construção de suas aplicações, entretanto será realizada uma pequena mudança com relação ao que se pode ver na Figura 6, o qual as respostas enviadas da *view* para o usuário será todas no formado JSON, como se pode observar na Figura 41.

Figura 41 – Arquitetura MVC adaptada para API



Fonte: Adaptado de RAMOS, (2015)

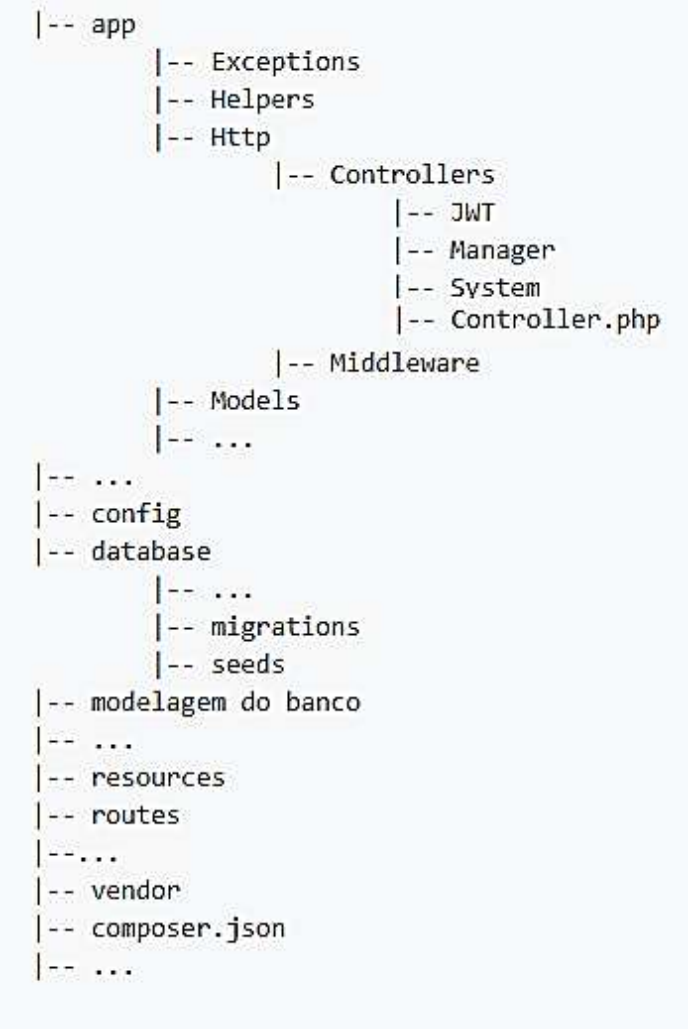
3.2.4.2.2 Projeto

Ainda com o foco no desenvolvimento rápido, além do uso do *framework* Laravel, nesta fase também foi utilizada uma ferramenta RAD muito popular no mundo todo, o GitHub, que forneceu várias bibliotecas que foram largamente utilizadas no projeto. As tecnologias utilizadas nessa etapa foram:

- PHP 7.3;
- Laravel 5.7;
- MySQL 5.7.19.

Nesta seção será comentado apenas o núcleo da aplicação, que são os códigos-fonte que estão dentro do diretório **app**, entretanto, a documentação completa desta aplicação estará disponível em: https://github.com/alexilallas/API-RESTful-com-Laravel5.7. A estrutura do diretório **app** é monstrada na figura a seguir.

Figura 42 – Estrutura do projeto



Fonte: Autor (2019)

A estrutura do projeto (Figura 42) representa como os diretórios e arquivos importantes estão dispostos ao longo do projeto, nota-se também a presença de pontilhados (...) que representam os diretórios e arquivos que não são relevantes ao desenvolvimento, pois fazem parte do padrão do framework adotado e não foram alterados durante todo o processo de desenvolvimento.

Todos os subdiretórios de **app** serão listados abaixo e explicados o que cada um contém e qual sua função no sistema:

- **Exceptions:** Possui a classe responsável por retornar as exceções da aplicação, esta classe foi modificada para capturar exceções do tipo JWT e retornar a resposta do tipo JSON. As exceções mais retornadas por esta classe serão de *token* expirado ou inválido, tendo em vista que o mesmo tem uma vida útil de apenas 8 horas, após esse tempo, caso o usuário faça qualquer requisição para a API, a aplicação identificará que o *token* expirou e retornará uma exceção no formato JSON para o usuário;

- **Helpers**: Possui funções que servirão como auxílio em algumas partes do código. As funções criadas nesse diretório podem ser utilizadas em qualquer parte do projeto pois são globais;
- **Http**: Este diretório possui 2 subdiretórios, são eles:

o **Controllers**: Este diretório é onde estão todos os *controllers* referentes à aplicação, pode-se dizer que é o núcleo do projeto. Neste diretório houve uma segmentação de forma que os *controllers* foram organizados em subdiretórios de acordo com a natureza de sua utilização, são eles:

- **JWT**: Nesse diretório está o *controller* responsável pela autenticação de usuários, o qual retornará um *token* após o usuário efetuar *login* no sistema e também invalidará o *token* do mesmo após efetuar o *logout*.
- **Manager**: Nesse diretório estão todos os *controllers* responsáveis pelo gerenciamento de usuários, perfis e permissões do sistema.
- **System**: Neste diretório estão todos os *controllers* responsáveis pelo gerenciamento do paciente, relatório e auditoria.
- **Controller.php**: Este é o *controller* base, no qual todos os outros irão herdar, ele contém os métodos-base que são utilizados por todos os *contollers* que estão nos diretórios **System** e **Manager**.

o **Middleware**: Neste diretório estão todos os *middleware* utilizados pelo framework, grande já vem por padrão, e outros foram desenvolvidos ou reutilizados de alguma biblioteca importada pelo projeto.

o **Kernel.php**: No Laravel, toda requisição HTTP, por questão de segurança passa por um conjunto de *middlewares*, nesta classe é onde se define quais os *middlewares* serão chamados a cada requisição.

- **Models**: É o diretório que irá conter todos os modelos do sistema, onde cada modelo representa uma tabela no banco.

Voltando à Figura 42, pode-se ver na raiz do projeto o diretório **routes**, este é o responsável por relacionar cada URL a um método de um *controller*.

Figura 43 – Algumas rotas para paciente

```
/**
 * Rotas para Paciente
 */
Route::get(
    'paciente',
    [
        'middleware' => 'acl:criarPaciente',
        'uses' => 'PacienteController@find'
    ]
);
Route::get(
    'paciente/{id}',
    [
        'middleware' => 'acl:criarPaciente',
        'uses' => 'PacienteController@findById'
    ]
);
```

Fonte: Autor (2019)

Como está descrito na figura acima, quando o usuário acessar a rota **paciente** via HTTP (http://localhost/api/paciente), o método **find** do *controller* **PacienteController** será chamado, o resultado pode ser visto na Figura 44.

Figura 44 – Resultado ra requisição da Figura 44


```
{
    message: "Pacientes cadastrados",
  - data: {
      - pacientes: [
          - {
                id: 1,
                nome: "STEPHAN HANE",
                telefone_celular: "176311396",
                cpf_rg: "52698061556",
                estado_civil: "Casado(a)",
                estado_naturalidade: "MA",
                cidade_naturalidade: "São Luís",
                data_nascimento: "1995-01-08",
                observacao: "Et assumenda expedita ullam",
                sexo: "Masculino",
                tipo: "Serviço Prestado",
                cep: 65044239,
                estado: "MA",
                cidade: "Lake Genovevaborough",
                bairro: "Mara Harbors",
                logradouro: "Cary Centers",
                numero: "49",
                matricula: null,
                curso_setor: null,
                ativo: 1,
                versao: 1,
                deleted_at: null,
                created_at: "2019-10-17 20:43:00",
                updated_at: "2019-10-17 20:43:00"
            }
        ]
    },
    status: 200
}
```


Fonte: Autor (2019)

3.2.4.3 Integração

Como se pode observar nas duas seções anteriores, as aplicações *front* e *back-end* foram desenvolvidas separadamente, pois são independentes, podendo até mesmo ser executadas separadamente, entretanto, somente quando as duas aplicações são integradas que desempenham o papel do sistema proposto neste trabalho.

A integração foi realizada com uma simples configuração na aplicação *front-end*, onde apenas se configurou o endereço da API (http://localhost:8000/api/) na qual a aplicação fará todas as requisições. Esta configuração é realizada no arquivo **environment.ts** que fica no diretório **src/environments** e pode ser vista na figura a seguir.

Figura 45 – Configuração para integração da aplicação *front-end* com a API

```
export const environment = {
  production: false,
  baseAPI: 'http://localhost:8000/api/'
  }
```

Fonte: Autor (2019)

3.1.5 Implantação

Como alguns dos componentes utilizados já foram testados anteriormente pelos desenvolvedores, só foi necessário realizar o teste com os componentes que foram criados pelo autor deste trabalho.

Para a realização dos testes foi necessário primeiro popular as tabelas com dados, para que se pudesse verificar se as ações do usuário no sistema estavam retornando o resultado esperado. Para isso fez-se o uso da biblioteca *faker* do PHP, que conta com uma grande variedade de dados, tais como nome de pessoas, lugares, números, CEP e outros dados falsos que são gerados randomicamente.

O *seed* é um recurso nativo do Laravel para inserção massiva de dados no banco, portanto, para popular as tabelas, foram escritos os *seeders* (integrados à biblioteca *faker*) para cada tabela desejada. O resultado foram 40 pacientes cadastrados, com dados pessoais, consultas, exames e históricos médicos com informações criadas para testes.

Com os dados inseridos, foram feitos os testes de integração e de sistema, onde foi verificado se cada componente estava respondendo conforme ao esperado e também se os componentes estavam cumprindo com seu papel quando integrados.

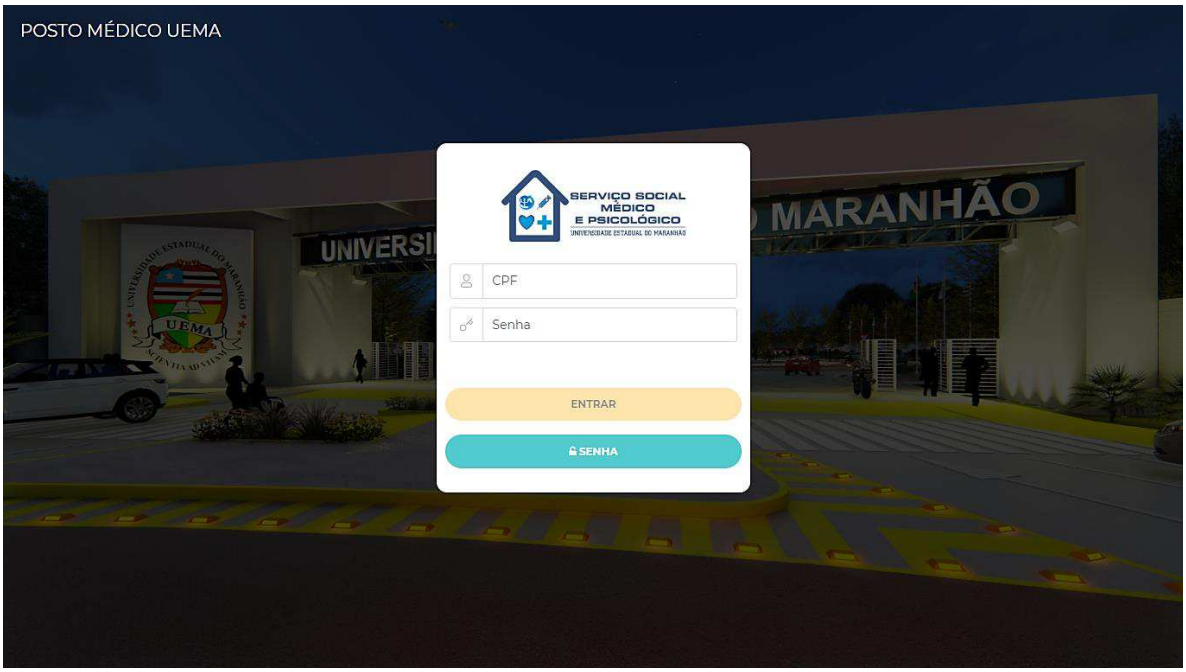
No final dos testes, foi realizada a validação junto aos funcionários do posto médico para ver se o sistema estava atendendo às suas expectativas. Após algumas validações, houve algumas correções e melhorias no sistema até que se chegou ao produto final.

4 RESULTADOS

Após a aprovação dos funcionários, o sistema foi para a fase de testes, antes de entrar para o ambiente de produção, para que pudessem ir se familiarizando ou talvez sugerir possíveis alterações. Algumas das telas do sistema podem ser observadas nas figuras a seguir:

A Figura 46 mostra a tela de *login* do sistema, a qual o usuário irá digitar seu CPF e senha para ter acesso ao sistema, nessa tela também é possível resetar a senha.

Figura 46 – Página de *login* do sistema



Fonte: Autor (2019)

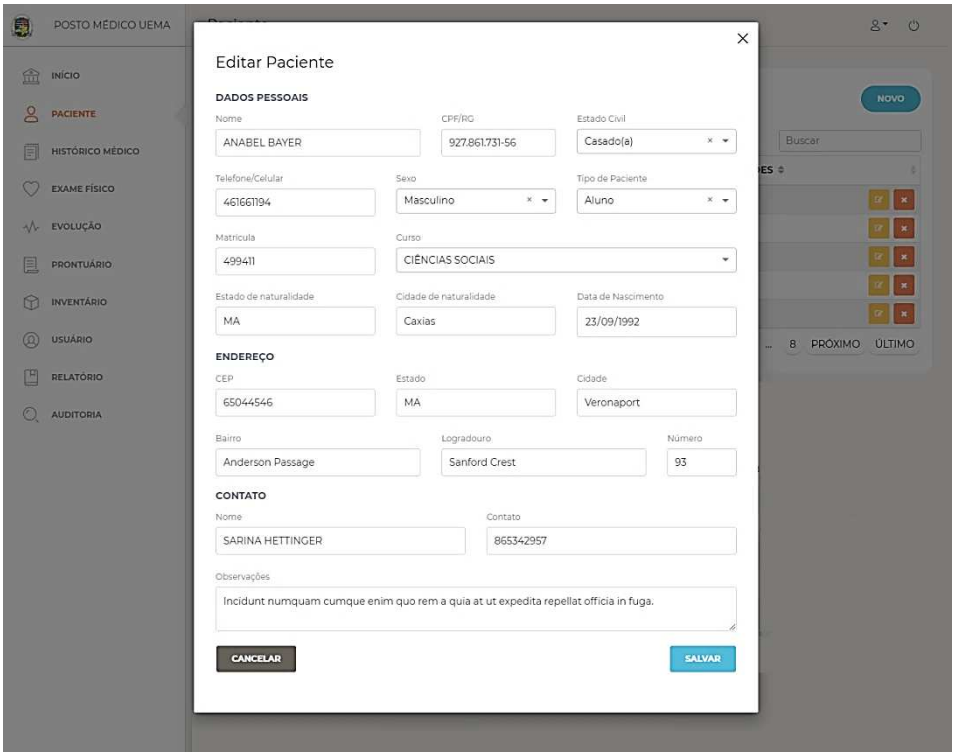
A Figura 47 monstra a tela inicial do sistema. É a tela que o usuário é direcionado após o *login*, nela podem-se visualizar pequenos resumos quantitativos e em forma de gráfico a cerca dos dados cadastrados no sistema.

Figura 47 - Página Inicial ou *Dashboard*

Fonte: Autor (2019)

A Figura 48 monstra a tela de edição de dados pessoais do paciente. Todas as figuras mostradas abaixo apresentam apenas a tela de edição de dados, entretanto estas telas são a mesma usada para cadastro.

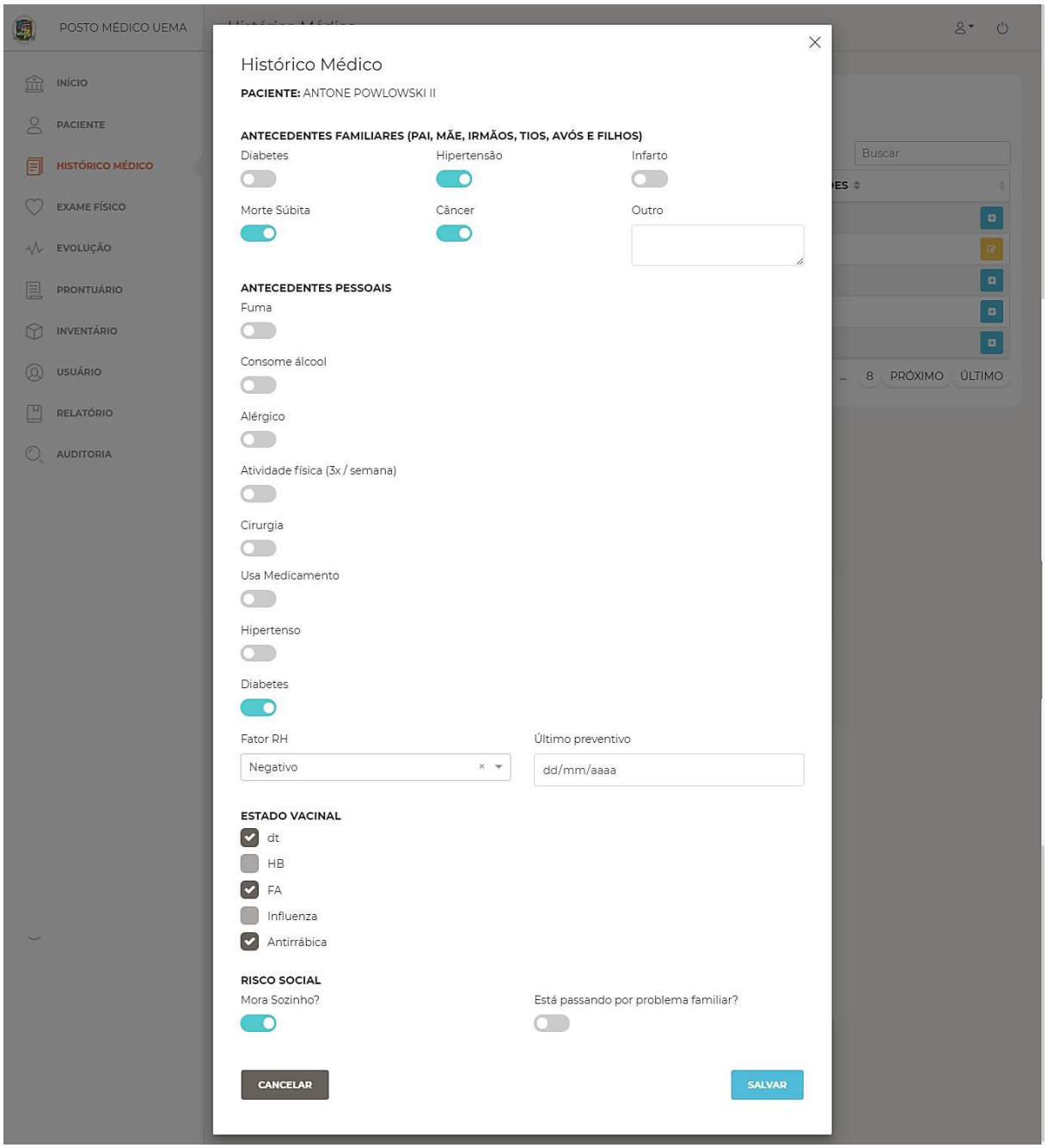
Figura 48 - Tela de edição de pacientes.



Fonte: Autor (2019)

A Figura 49 monstra a tela de histórico do paciente, a qual o atendente irá preencher os dados de acordo com as respostas do paciente.

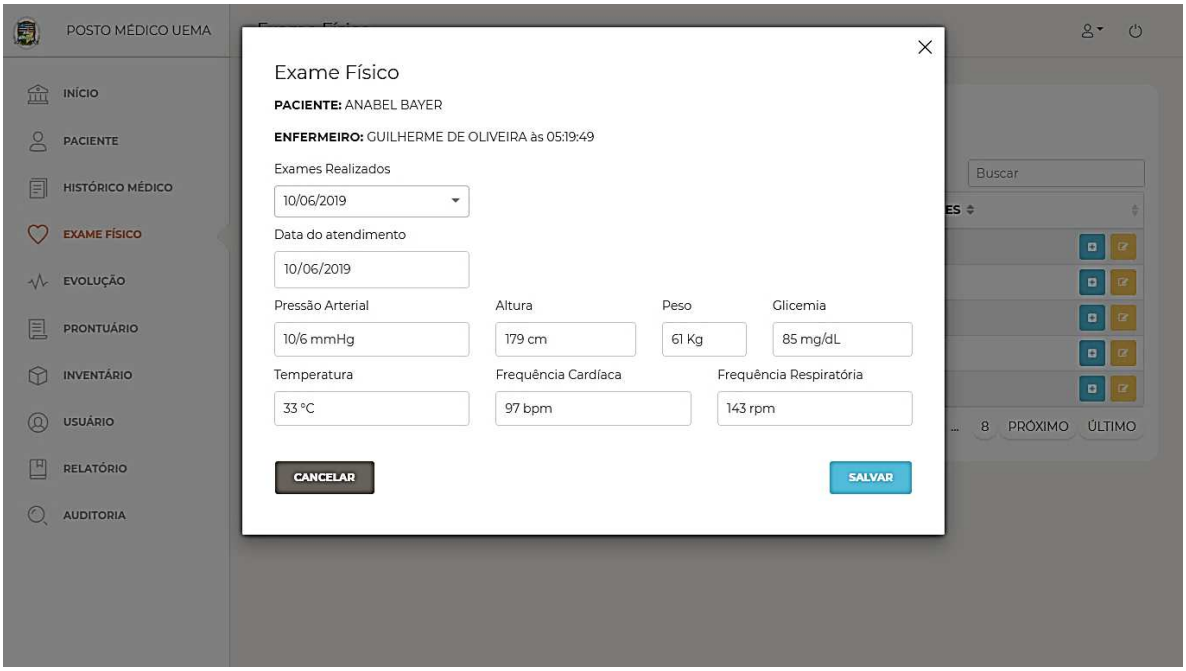
Figura 49 – Tela de histórico do paciente



Fonte: Autor (2019)

A Figura 50 monstra a tela de edição de exame físico, nela o enfermeiro irá preencher os dados com base no exame realizado, é importante notar também que nesta tela também se pode observar informações sobre o atendimento (enfermeiro que realizou o exame e horário de atendimento).

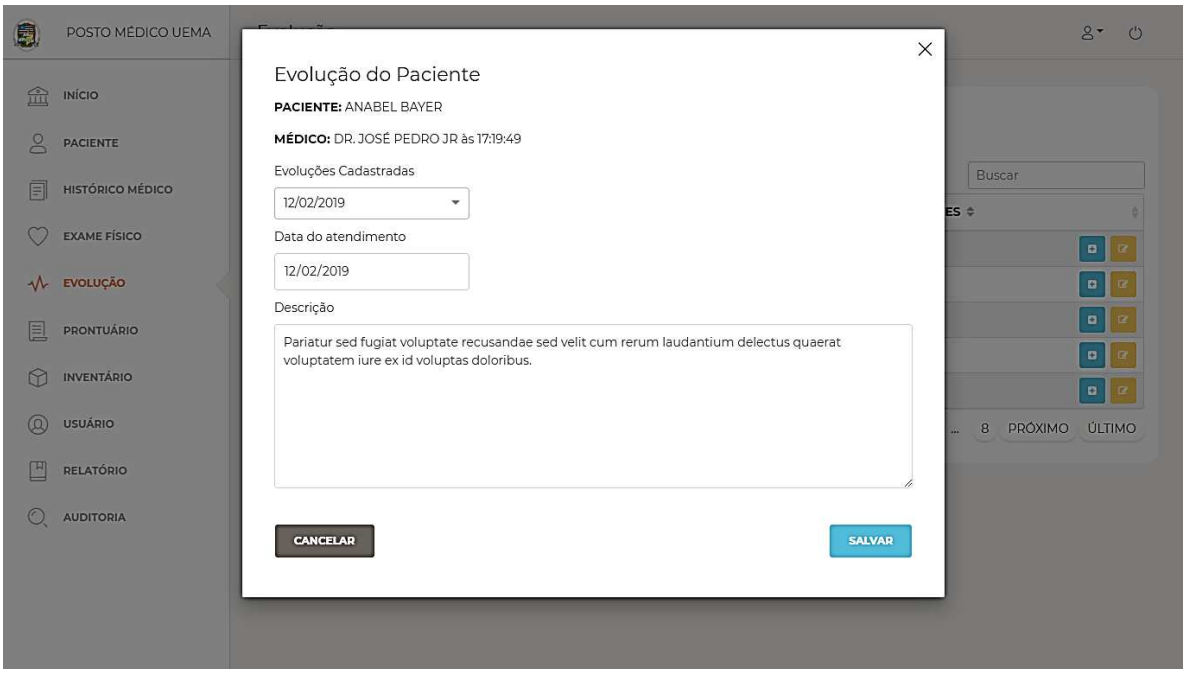
Figura 50 – Tela de exame físico



Fonte: Autor (2019)

A Figura 51 monstra a tela de edição de evolução do paciente, e análoga à tela anterior, também conta com informações sobre o atendimento.

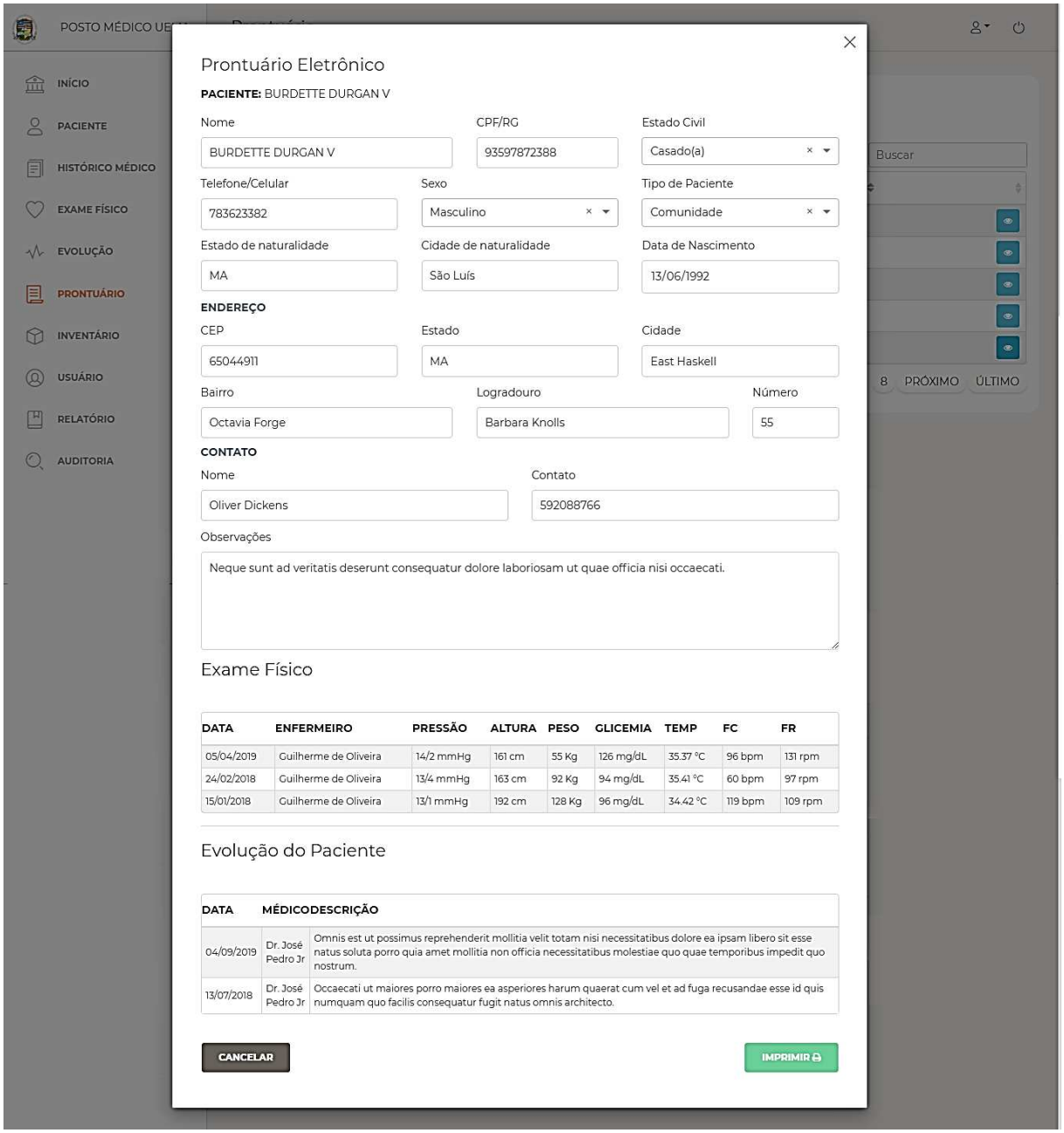
Figura 51 – Tela de evolução do paciente



Fonte: Autor (2019)

A Figura 52 monstra o prontuário eletrônico de um paciente, nele é possível visualizar os dados pessoais do paciente assim como seus exames e consultas.

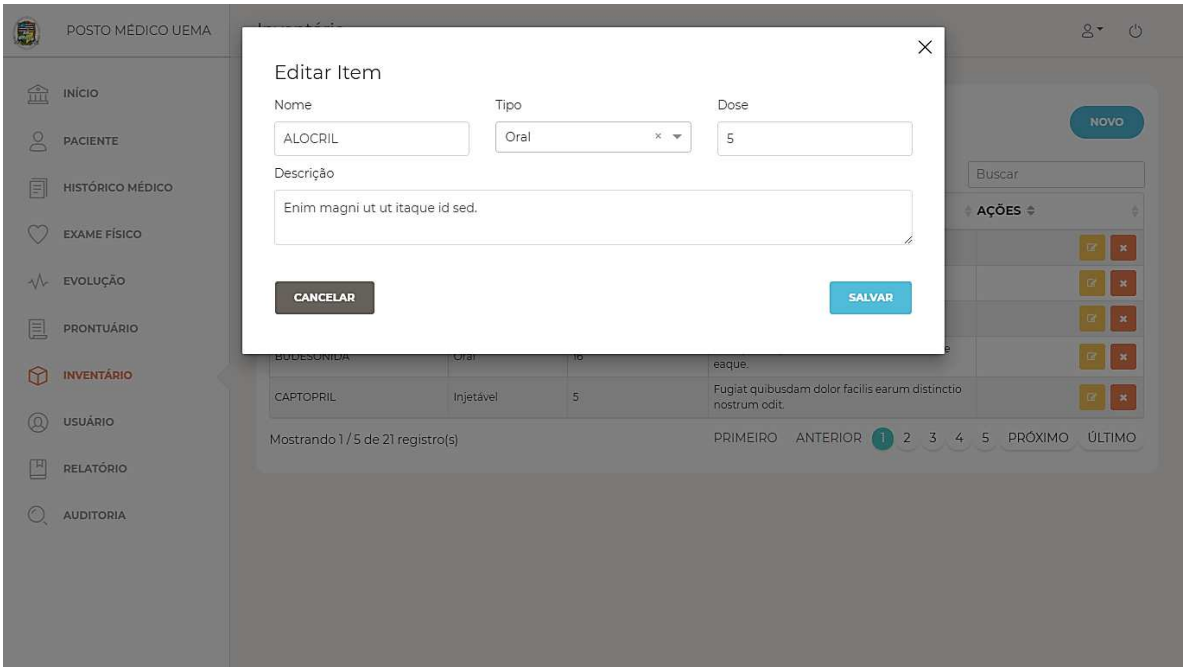
Figura 52 – Prontuário eletrônico de um paciente



Fonte: Autor (2019)

A Figura 53 monstra a tela de edição de item do inventário.

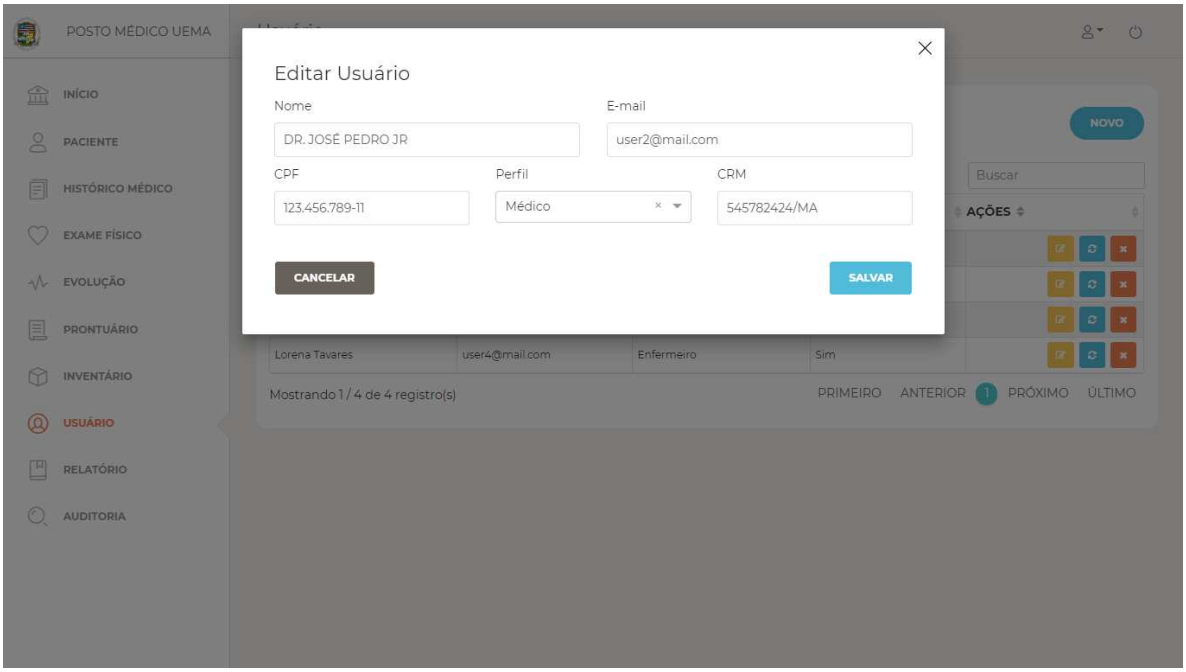
Figura 53 – Tela de edição de item do inventário



Fonte: Autor (2019)

A Figura 54 monstra a tela de edição de usuário do sistema, nela é possível alterar dados do usuário e também o perfil.

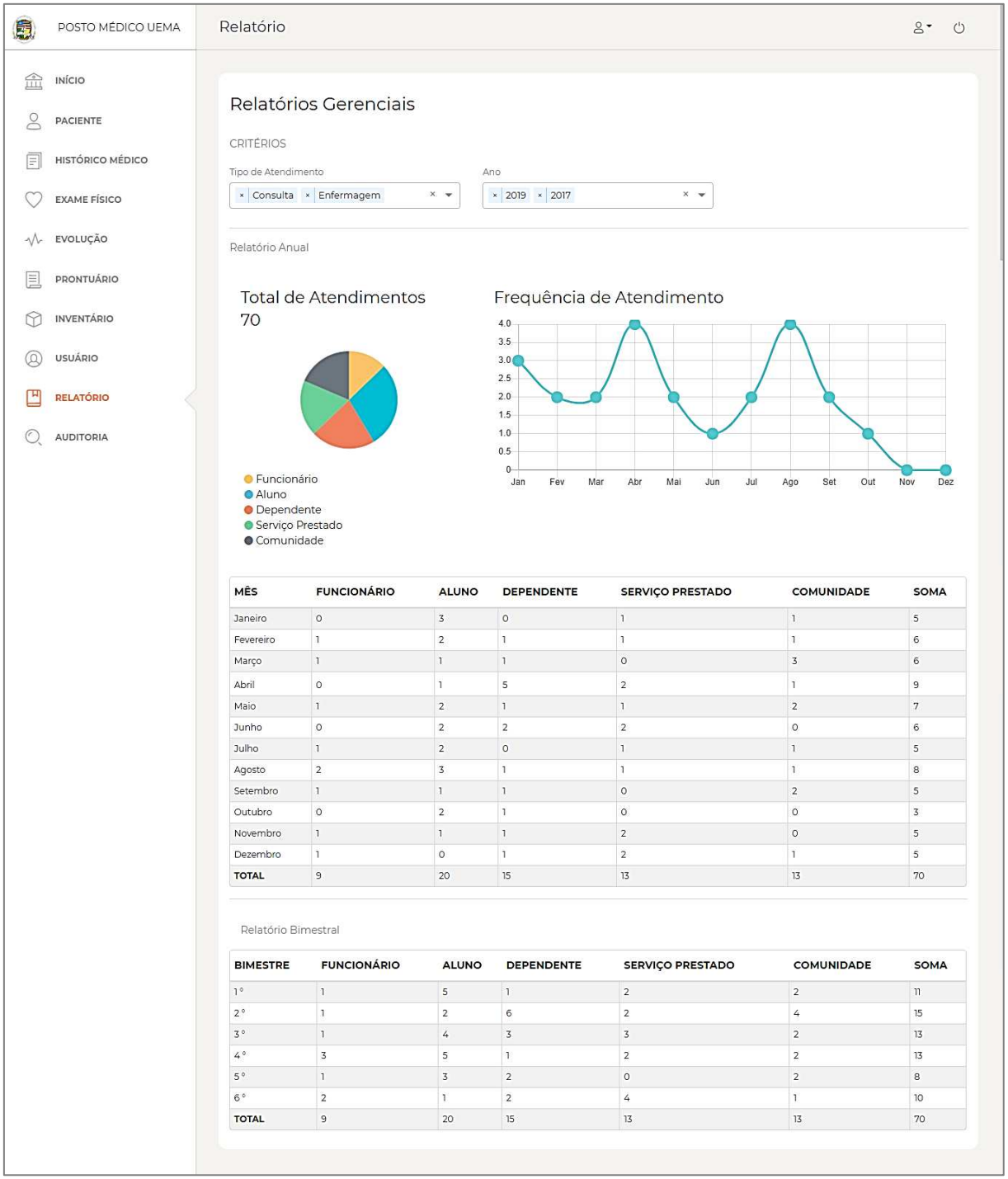
Figura 54 – Tela de edição de usuário do sistema



Fonte: Autor (2019)

A Figura 55 monstra a tela de relatório do sistema, nela é possível visualizar informações sobre os atendimentos realizados. Para gerar as informações, o usuário precisa escolher os critérios (Tipo de atendimento e ano).

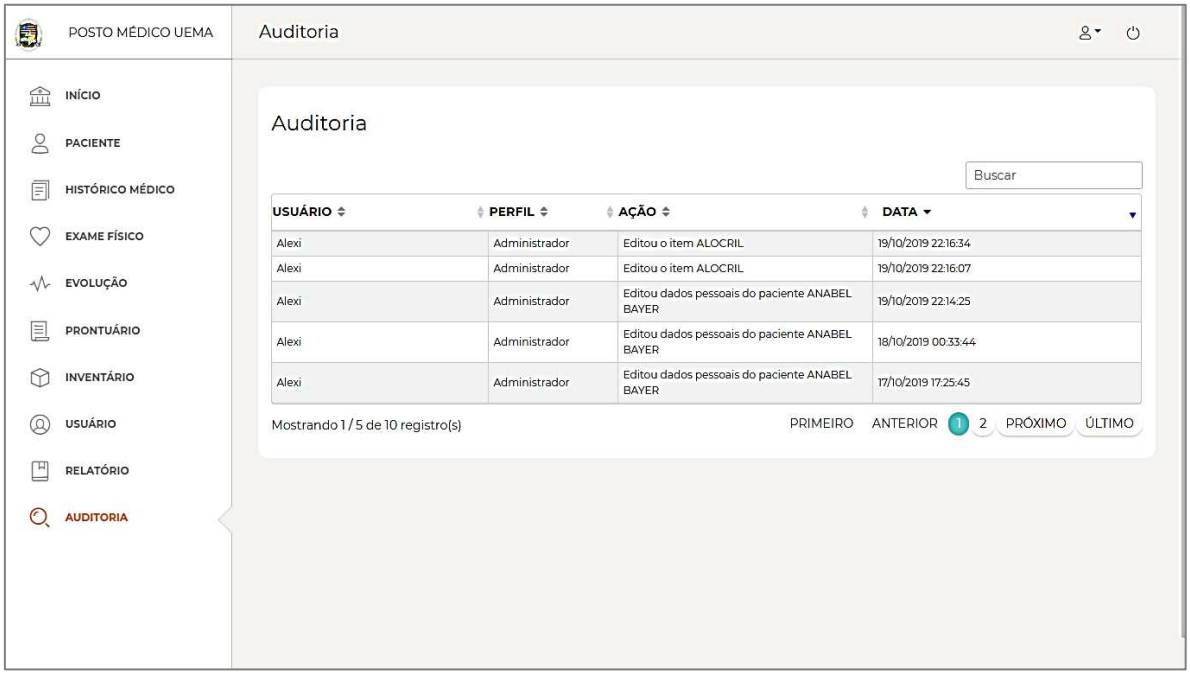
Figura 55 – Tela de relatório do sistema



Fonte: Autor (2019)

A Figura 56 monstra a tela de auditoria, a qual o usuário administrador poderá visualizar todas as ações realizadas por outros usuários.

Figura 56 – Tela de auditoria do sistema



Fonte: Autor (2019)

5 CONCLUSÃO

Neste trabalho foi desenvolvido um sistema de gerenciamento de pacientes para o posto médico da UEMA, tendo em vista que o mesmo não possuía e necessitava para uma melhor gerência dos recursos e melhor visão sobre o trabalho realizado.

Levando em consideração os objetivos descritos na seção 1.1, o sistema desenvolvido neste trabalho utilizou-se de tecnologias gratuitas, de código livre e que possuem ótima aceitação no mercado, foram elas o Laravel para a construção da API e o Angular para o desenvolvimento da aplicação *web*.

A arquitetura de software REST também se mostrou muito satisfatória por ser escalável e também permitir que qualquer aplicação possa consumir seus dados, caso necessário. Além disso, a metodologia utilizada, o RAD se mostrou também muito eficiente, pois o uso desta metodologia ágil permitiu a criação de componentes individuais, que por sua vez possibilitou a integração rápida do sistema e *feedback* contínuo do cliente para mudanças.

O sistema desenvolvido também teve grande aprovação dos interessados, por possuir uma interface amigável e intuitiva, o que resultou em uma experiência agradável aos usuários. Portanto, pode-se concluir que este trabalho cumpriu como os objetivos estabelecidos e obteve êxito total em sua execução.

5.1 Trabalhos futuros

Para o desenvolvimento de trabalhos futuros são sugeridos os seguintes:

- Criar um aplicativo *mobile* para o posto médico para melhor comodidade dos usuários, tendo em vista que na arquitetura escolhida para a construção deste sistema, a interface do usuário é totalmente independente e os dados podem ser acessados por qualquer dispositivo que seja capaz de enviar requisições HTTP.
- A construção de componentes para o gerenciamento do atendimento psicológico, que se iniciou recentemente, mas já possui uma grande demanda.

REFERÊNCIAS

ALMEIDA, Guilherme Augusto Machado de. **Fatores de escolha entre metodologias de desenvolvimento de software tradicionais e ágeis.** 2017. Dissertação (Mestrado em Ciências) – Escola Politécnica da Universidade de São Paulo, São Paulo, 2017.

CHIAVENATO, Idalberto. **Gestão de Pessoas; e o novo papel dos recursos humanos nas organizações**. Rio de Janeiro:2.ed. Campus, 2004 4ª Reimpressão.

COUTO**,** Leonardo Matriciano. **Sistemas de Informação Geográfica Adaptativos Baseados em Modelos.** 2006. Dissertação (Mestrado em Informática) – Pontífica Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2006.

Creating a REST API. Happy Coding. 2018. Disponível em: https://happycoding.io/tutorials/java-server/rest-api**.** Acesso em: 09 nov. 2019.

DIAS, Emílio. **4 Conceitos sobre REST que Qualquer Desenvolvedor Precisa Conhecer.** Algaworks. 2016. Disponível em: https://blog.algaworks.com/4-conceitos-sobre-rest-que-qualquer-desenvolvedor-precisa-conhecer/. Acesso em: 03 nov. 2019.

FERNANDES, Cláudio. **Civilização Mesopotâmica.** Mundo Educação. Disponível em: https://mundoeducacao.bol.uol.com.br/historiageral/civilizacao-mesopotamica.htm. Acesso em: 15 jun. 2019.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Dissertação de Doutorado, Universidade da California, Irvine, 2000.

FIELDING, Roy. **Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Section 4**. Internet Engineering Task Force (IETF). Disponível em: https://tools.ietf.org/html/rfc7231#section-4. Acesso em: 03 nov. 2019.

GOVONI, D. **Java Application Frameworks**. John Wiley & sons, 1999.

IMENES, Luiz Márcio Pereira. Os números na história da civilização. São Paulo: Scipione, 1999. (Coleção Vivendo a matemática).

LUIZ, Ronaldo Rezende Vilela. **Obtendo Qualidade de Software com o RUP**. Disponível em: http://qualidade-de-software.blogspot.com/2010/03/obtendo-qualidade-de-software-com-o-rup.html. Acesso em: 05 out. 2019.

MORAES, Denise. **Conheça a história dos números.** EBC. 2015. Disponível em: http://www.ebc.com.br/infantil/voce-sabia/2015/05/conheca-historia-dos-numeros. Acesso em: 15 jun. 2019.

NEVES, Tiago de Jesus. **Diagrama descrevendo o modelo cliente-servidor**. 2013. Disponível em: https://pt.wikipedia.org/wiki/Modelo_cliente%E2%80%93servidor#/media/Ficheiro:Cliente-Servidor.png**.** Acesso em: 09 nov. 2019.

O que é uma API?. Red Hat. Disponível em: https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces**.** Acesso em: 09 nov. 2019.

O'BRIEN, J. A. **Introduction to information systems**. 9 th edition. New York: Irwin McGraw-Hill, 2000.

PRATES, Thiago Silva. **Sistemas de Informação**. Disponível em: http://tsprates.com/gestao/Sistemas%20de%20Informa%C3%A7%C3%A3o.pdf. Acesso em: 05 out. 2019.

PRESSMAN, Roger S. **Engenharia de Software** : 6 ed. São Paulo: McGraw Hill/Nacional, 2006.

RAMOS, Allan. **O que é MVC?**. Tableless. 2015. Disponível em: https://tableless.com.br/mvc-afinal-e-o-que/. Acesso em: 15 out. 2019.

SOARES, Michel dos Santos. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. INFOCOMP Journal of Computer Science, [S.l.], v. 3, n. 2, p. 8-13, nov. 2004. ISSN 1982-3363. Disponível em: http://www.dcc.ufla.br/infocomp/index.php/INFOCOMP/article/view/68. Acesso em: 10 out. 2019.

SOMMERVILLE, Ian, **Engenharia de Software**. 9.ed. São Paulo: Prentice-Hall, 2013.