UFAM - Engenharia da Computação

MEDMOB – APLICATIVO DE CONSULTAS MÉDICAS UTILIZANDO REACT NATIVE E NODE.JS

Rodrigo Soares dos Santos

Monografia de Graduação apresentada à Coordenação de Engenharia da Computação, UFAM, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Engenheiro da Computação.

Orientador: Eduardo James Pereira Souto

Manaus

Novembro de 2021

MEDMOB – APLICATIVO DE CONSULTAS MÉDICAS UTILIZANDO REACT NATIVE E NODE.JS

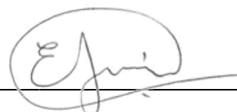
Rodrigo Soares dos Santos

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO CURSO DE ENGENHARIA DA COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DO AMAZONAS COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO.

Aprovada por:

Prof. Eduardo Luzeiro Feitosa.

Prof. David Braga Fernandes de Oliveira.



Prof. Eduardo James Pereira Souto.

Manaus

Novembro de 2021

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S237m

Santos, Rodrigo Soares dos
MedMob - aplicativo de consultas médicas utilizando React Native e Node.JS / Rodrigo Soares dos Santos . 2021
62 f.: il. color; 31 cm.

Orientador: Eduardo James Pereira Souto
TCC de Graduação (Engenharia da Computação) - Universidade Federal do Amazonas.

1. Consultas médicas. 2. React Native. 3. Node.JS. 4. MongoDB. I. Souto, Eduardo James Pereira. II. Universidade Federal do Amazonas III. Título

Agradecimentos

Agradeço a Deus e a todos os meus familiares que me ajudaram a disponibilizar a mim o melhor ambiente possível para desenvolver o projeto em casa, bem como terem me dado apoio durante a minha caminhada como estudante universitário. Agradeço em especial a minha mãe Maria Regina, que sempre tentou oferecer a mim todas as condições necessárias para que eu fosse o primeiro membro da família a ter um diploma de ensino superior, mas principalmente por nunca ter desistido de mim, até quando eu mesmo já havia desistido.

Também agradeço a meus parceiros da primeira versão do MedMob – Leonam Bernardo, Elaine Soares e Andrezza Bonfim, que me apresentaram ao *React Native* e permitiram que eu pudesse continuar desenvolvendo o aplicativo.

Agradeço aos meus amigos que me ajudaram a entrar no mercado de trabalho e a lidar com as pressões que o mesmo pode nos dar.

Agradeço também ao meu orientador Eduardo Souto, que pode me ajudar a tomar as melhores decisões a respeito do projeto, tanto em relação a qualidade de vida, quanto ao desenvolvimento.

Por fim, agradeço as minhas gatas de estimação, Marie e Estrela, que sempre chamavam a minha atenção quando eu estava cansado ou desanimado.

Resumo da Monografia apresentada à UFAM como parte dos requisitos necessários para a obtenção do grau de Engenheiro

MEDMOB – APLICATIVO DE CONSULTAS UTILIZANDO REACT NATIVE E NODE.JS

Rodrigo Soares dos Santos

Novembro/2021

Orientador: Eduardo James Pereira Souto

Curso: Engenharia da Computação

Este trabalho mostra o desenvolvimento de um aplicativo para solicitação de consultas médicas. O sistema deve permitir tanto a criação de consulta pelo paciente, quanto a busca por consultas pelo médico. O aplicativo MedMob tem como objetivo principal permitir a um usuário solicitar uma consulta, definir um local para que a mesma ocorra, e permitir a um médico buscar, aceitar ou rejeitar consultas. No desenvolvimento do *frontend, f*oi utilizada a linguagem de programação *JavaScript*, com o uso do *framework React Native*. Para a implementação do *backend*, foi utilizada a ferramenta *Node.JS*. Foi utilizada a biblioteca *Mongoose* para modelar o banco de dados para o *MongoDB*.

Palavras-chave: Consultas médicas, React Native, Node.JS, MongoDB.

Abstract of Monograph presented to UFAM as a partial fulfillment of the requirements for the degree of Engineer

MEDMOB – MEDICAL CONSULTATION APP USING REACT NATIVE AND NODE.JS

Rodrigo Soares dos Santos

November/2021

Advisor: Eduardo James Pereira Souto

Course: Computer Engineering

This work shows the development of an application for requesting medical appointments. The system should allow both the creation of an appointment by the patient and the search for appointments by doctor. The MedMob application's main purpose is to allow a user to request an appointment, define a place for it to occur, and allow a doctor to seek, accept and reject appointments. In the development of frontend, the application use JavaScript programming language, using the React Native framework. For the backend implementation, the system use the Node.JS tool. The application uses the Mongoose framework to model the database for MongoDB.

Keywords: Medical appointments, React Native, Node.JS, MongoDB.

Sumário

1. Introdução 12
1.1 Objetivos 13
1.1.1 Objetivo Geral 13
1.1.2 Objetivos Específicos 13
1.2 Organização da Monografia 13
2. Conceitos teóricos 15
2.1 JavaScript 15
2.2 React Native 15
2.3 Redux e Redux-Saga 17
2.4 Node.JS 19
2.5 MongoDB 20
2.6 Mongoose 20
2.7 Socket.IO 21
3. Desenvolvimento do MedMob 22
3.1 Desenvolvimento do *frontend* 25
3.1.1 Design e User Experience 25
3.1.2 Tela de login e de cadastro de usuário 28
3.1.3 Tela inicial 30
3.1.4 Histórico 31
3.1.5 Tela de configurações 32
3.1.6 Tela de perfil 34
3.1.7 Tela de métodos de pagamento 35
3.1.8 Criação de forma de pagamento 36
3.1.9 Edição de método de pagamento 37
3.1.10 Tela de informações gerais da consulta 37
3.1.11 Tela de seleção de local de consulta 38
3.1.12 Tela de seleção de pagamento da consulta 39
3.1.13 Tela de espera de consulta 40
3.1.14 Tela de procura de consultas 40
3.1.15 Tela de espera de início de consulta 41
3.1.16 Tela de consulta em andamento 42
3.1.17 AsyncStorage 43
3.2 Desenvolvimento do *backend* 44
3.2.1 Especialidades 44

3.2.2 CRUD de especialidades 44
3.2.3 Usuário 45
3.2.4 CRUD de usuários 46
3.2.5 Pagamentos 46
3.2.6 CRUD de pagamentos 47
3.2.7 Consultas 48
3.2.8 CRUD de consultas 49
3.2.9 Report 52
3.2.10 CRUD de Report 52
3.2.11 Serviço de Socket.io 53
3.2.12 Login 54
3.2.13 Validação de rotas com o token 54
3.2.14 Logout 55
4. Testes e Resultados 56
5. Conclusões 59
5.1 Considerações Finais 59
5.2 Propostas para Trabalhos Futuros 59
6. Referências Bibliográficas 61

Lista de Imagens

Figura 1 - Conjunto de componentes do React.. 16
Figura 2 - Componentes centrais do Redux. 18
Figura 3 - Diagrama das coleções do banco de dados 24
Figura 4 - Casos de uso dos pacientes... 24
Figura 5 - Casos de uso dos médicos 25
Figura 6 - A relação entre cores, sensações transmitidas e aplicativos........................ 26
Figura 7 - Logo do MedMob. 26
Figura 8 - Paleta de cores primária. 27
Figura 9 - Paleta de cores secundária... 27
Figura 10 - Botão para prosseguimento de tarefas. 27
Figura 11 - Botão secundário, para voltar ações/ações secundárias. 27
Figura 12 - Botão de cancelamento de ação... 27
Figura 13 - Caixa de entrada de texto. 27
Figura 14 - Legenda de dados e dados.. 28
Figura 15 - Título da tela... 28
Figura 16 - Janela de opções e botões do tipo texto ("galeria" e "câmera"). 28
Figura 17 - Tela de login. 28
Figura 18 - Tela de novo usuário - 1/5.. 29
Figura 19 - Tela de novo usuário - 2/5.. 29
Figura 20 - Tela de novo usuário (para médicos) - 2/5... 29
Figura 21 - Tela de dados de verificação. 29
Figura 22 - Tela de envio de código para o usuário.. 30
Figura 23 - Tela de envio de código a ser verificado.. 30
Figura 24 - Tela inicial, nenhuma consulta em andamento.. 30
Figura 25 - Tela inicial - paciente procurando consulta.. 30
Figura 26 - Tela inicial - Médico à caminho 31
Figura 27 - Tela inicial - consulta em andamento 31
Figura 28 - Aba de histórico de consultas. 31
Figura 29 - Tela de visualização de consulta... 32
Figura 30 - Tela de configurações 32
Figura 31 - Janela para fazer reports ao MedMob.. 33

Figura 32 - Janela de opções da conta.. 33
Figura 33 - Janela de confirmação de exclusão da conta. 33
Figura 34 - Tela de perfil do usuário. 34
Figura 35 - Janela de visualização e opções para a foto de perfil. 34
Figura 36 - Tela de edição do usuário. 35
Figura 37 - Lista de métodos de pagamentos. 35
Figura 38 - Janela de pagamento selecionado. 36
Figura 39 - Tela de criação de forma de pagamento. 36
Figura 40 - Edição de método de pagamento. 37
Figura 41 - Informações da nova consulta. 38
Figura 42 - Lista de especialidades. 38
Figura 43 - Tela de seleção do local de consulta. 38
Figura 44 - Mapa de seleção do local de consulta. 39
Figura 45 - Tela de seleção de pagamentos. 39
Figura 46 - Tela de espera de consulta. 40
Figura 47 - Tela de procura de consulta para médicos. 41
Figura 48 - Consulta encontrada na tela de procura. 41
Figura 49 - Nenhuma consulta foi encontrada na tela de procura. 41
Figura 50 - Tela de médico à caminho. 41
Figura 51 - Tela de espera do paciente. 41
Figura 52 - Chat da consulta. 42
Figura 53 - Tela de consulta em andamento. 43
Figura 54 - Janela de avaliação do médico/paciente. 43
Figura 55 - Feedback de conclusão de consulta médica 57
Figura 56 - Feedback de erro na tentativa de login.. 57
Figura 57 - Feedback de cadastro de pagamento 57
Figura 58 - Feedback de edição de usuário 57
Figura 59 - Sandbox do serviço de Short Notification Services. 58
Figura 60 - Acesso ao modo de desenvolvimento do SNS. 58

Lista de Tabelas

Tabela 1 - Relação entre nomes de componentes e ambientes de desenvolvimento...... 16
Tabela 2 - Esquema de especialidades. 44
Tabela 3 - Esquema de usuários. 45
Tabela 4 - Esquema de pagamentos. 47
Tabela 5 - Esquema de consultas. 48
Tabela 6 - Esquema de Report. 52

Lista de Equações

Equação 1 - Definição do preenchimento horizontal 27
Equação 2 - Definição do raio de distância de procura de consultas 51

1. Introdução

A comodidade e o conforto são importantes para o bem-estar de qualquer pessoa, ainda mais em tempos corridos como os de hoje em dia. Logo, há um grande investimento por parte de empresas consolidadas e *startups* em criar e modificar seus serviços de modo que o conforto e a praticidade sejam qualidades perceptíveis no uso.

No contexto de alimentação, os serviços de *delivery* cresceram bastante entre os estabelecimentos alimentares, sendo assim uma ótima opção para pessoas com pouco tempo, poucas habilidades gastronômicas, ou que simplesmente não querem cozinhar. Temos hoje como principal exemplo o iFood, que centraliza o serviço de *delivery* para vários restaurantes e lanchonetes.

Olhando para o setor de locomoção de pessoas, se tornou comum o uso de serviços de solicitação de motoristas particulares para o deslocamento ao trabalho, escola, faculdade e locais de lazer. Isto acontece pois o conforto de se locomover para algum lugar utilizando um carro é bem maior do que utilizando o transporte coletivo público, além de ser mais seguro. Como principais exemplos, temos o Uber e o 99app.

Na parte médica, os convênios médicos entregam uma comodidade que infelizmente a rede pública de saúde ainda não pode nos oferecer. Alguns convênios oferecem serviços de marcação de consultas e *check-in online*, onde o paciente, no hospital ou consultório, precisa apenas fazer uma verificação de presença, economizando tempo em filas e recepções.

Uma área pouco explorada em relação a serviços médicos são as consultas no lar do paciente. As consultas médicas ocorrem, na grande maioria das vezes, em hospitais e consultórios particulares. Porém, em um contexto de pandemia, a locomoção de pessoas e aglomerações são limitadas, a demanda de atendimentos em hospitais aumenta, e os hospitais reduzem a quantidade de consultas em várias especialidades para evitar aglomerações e centralizar o foco no tratamento das doenças causadas pelo vírus pandêmico.

Para lidar com tais problemas, este trabalho apresenta uma aplicação capaz de criar uma consulta que será realizada na localidade escolhida pelo paciente, descentralizando consultas de hospitais e consultórios particulares.

Algumas especialidades e tipos de consultas necessitam de equipamentos e recursos que são inviáveis para o transporte, mesmo nas fases iniciais de inspeção de paciente. Por isso, algumas especialidades não estarão disponíveis no aplicativo, uma vez que o uso dos equipamentos são necessários para garantir a boa condição de trabalho [20].

Além disso, o tipo de consulta a ser pedido no aplicativo deve abranger os processos de inspeção, palpação e ausculta (ouvir os sons e ruídos do corpo), ou seja, fases iniciais de consultas. Sendo assim, o projeto não abrange retorno, avaliação de exames, muito menos procedimentos cirúrgicos ou procedimentos envolvendo equipamentos de grande porte.

1.1 Objetivos

1.1.1 Objetivo Geral

Este projeto tem como objetivo apresentar um sistema de agendamento de consultas médicas que possibilita, por meio da conexão entre paciente e médico, que cada atendimento ocorra no local escolhido durante o agendamento.

1.1.2 Objetivos Específicos

Temos os seguintes objetivos específicos:

- Disponibilizar uma interface para que pacientes possam solicitar e visualizar consultas médicas;
- Oferecer a médicos uma interface de busca e visualização de consultas médicas, de acordo com a especialidade a qual o médico pertence;
- Desenvolver um servidor capaz de controlar os processos envolvendo a conexão entre médico e paciente e o gerenciamento de consultas, usuários e dados auxiliares;
- Gerenciar o armazenamento dos dados disponibilizados por usuários do aplicativo e dados gerados pelo servidor.

1.2 Organização da Monografia

A monografia está organizada da seguinte forma:

- Capítulo 2: são apresentados os fundamentos teóricos, principais bibliotecas e tecnologias utilizadas para desenvolver o projeto;
- Capítulo 3: apresenta o processo de desenvolvimento realizado no projeto, tanto na parte de *design*, quanto nas partes de *frontend* e *backend.* São mostrados os requisitos, diagramas, telas do sistema e funções de operação do servidor*;*
- Capítulo 4: mostra os testes e avaliação de resultados obtidos com o desenvolvimento;
- Capítulo 5: considerações finais sobre o projeto, conclusão e trabalhos futuros a respeito do MedMob.

2. Conceitos teóricos

Este capítulo apresenta e resume as principais linguagens, *frameworks* de desenvolvimento e banco de dados utilizados no desenvolvimento do aplicativo MedMob.

2.1 JavaScript

JavaScript [1] é uma linguagem de programação comumente usada na implementação de aplicações para a *Web*, porém pode ser utilizada para desenvolver aplicações que funcionam fora do navegador, como no *Node.JS* e *Apache*. A linguagem é multiparadigma, ou seja, suporta estilos de programação funcional, imperativa, orientada a objetos e eventos.

Com o *JavaScript* é possível armazenar conteúdo em variáveis sem especificação de tipo, operações com *strings* e *arrays*, execução de funções em resposta a ações realizadas com teclado e *mouse*, como rolagem para cima e para baixo, cliques, teclas pressionadas, entre outros.

Junto com as Interfaces de Programação de Aplicativos (*API*), é possível utilizar vários tipos de códigos de construção para implementar diversos recursos na aplicação, como por exemplo geolocalização, gráficos e reprodução de áudio e vídeo.

Ao carregar uma página da Web, o navegador utiliza os códigos em *JavaScript*, HTML (*HyperText Markup Language*) e CSS (*Cascading Style Sheets*) para mostrar o conteúdo. O *JavaScript* consegue modificar dinamicamente os códigos em HTML e CSS utilizando a API do *Document Object Model*.

A execução do código em *JavaScript* é feita, como em quase todas as linguagens de programação, de cima para baixo. Logo, é preciso ter cuidado com a ordem em que variáveis, importações e funções são adicionados no código.

2.2 React Native

React Native é um *framework* de código aberto do *Facebook* para criação de aplicações em *Android* e *iOS,* que utiliza o *React* e os recursos nativos da plataforma na qual está sendo desenvolvido o aplicativo.

Componentes nativos tanto do *Android* quanto do *iOS* são representados pelos componentes do *React Native*. Podemos ver na Tabela 1 esta relação entre componentes [2].

Tabela 1 - Relação entre nomes de componentes e ambientes de desenvolvimento.

COMPONENTE DO REACT NATIVE	VISUALIZAÇÃO ANDROID	VISUALIZAÇÃO IOS	RELATIVO PARA A WEB	DESCRIÇÃO
<View>	<ViewGroup>	<UIView>	<div> não rolável	Contêiner que suporta *layout* com *flexbox,* estilos, suporte a toques e controles de acessibilidade
<Text>	<TextView>	<UITextView>	<p>	Visualização de texto que suporta estilos, junta sequências de texto e suporte a toques
<Image>	<ImageView>	<UIImageView>	<img>	Mostra diversos tipos de imagens
<ScrollView>	<ScrollView>	<UIScrollView>	<div>	Recipiente rolhável que pode conter múltiplos componentes
<TextInput>	<EditText>	<UITextField>	<input type="text">	Permite o usuário inserir textos

Uma aplicação em *React Native* combina vários desses componentes, criados pelo próprio desenvolvedor e outros criados pela comunidade em sua implementação, como podemos ver no diagrama representado pela Figura 1.

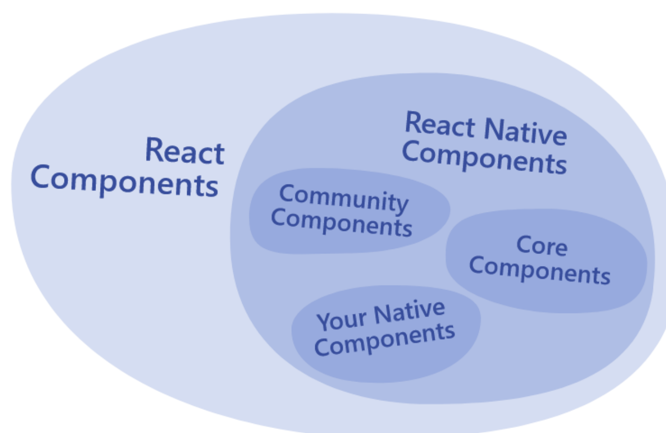


Figura 1 - Conjunto de componentes do React.

A comunidade contribui para que o *React Native* tenha diversas funcionalidades e recursos, transformando-as em bibliotecas. No MedMob, as principais bibliotecas utilizadas são:

- *React navigation* [3] – proporciona ao aplicativo a transição entre telas e gerenciamento do histórico de navegação. É semelhante à navegação pela web –

uma pilha onde é possível tirar e adicionar itens de navegação que resultam na tela que vai ser mostrada na *interface* do usuário. Com o *React navigation,* é possível adicionar gestos e animações de transição às telas. É possível adicionar alguns tipos de navegação comuns em aplicativos para *smartphone*: abas (inferiores ou superiores), telas comuns e menu lateral;

- *AsyncStorage* [4] – biblioteca criada para armazenar dados na memória do próprio *smartphone*, de forma assíncrona e no formato chave-valor. As informações são salvas em formato de cadeia de caracteres (*strings*), além de serem persistentes entre sessões de uso da aplicação.

- *Redux* e *Redux-Saga* – *Redux* é uma biblioteca independente que pode ser usada com qualquer camada de interface do usuário ou *framework* [5]. Então, apesar de serem usados juntos com frequência, *React* e *Redux* são independentes entre si. O *Redux-Saga* [6] é uma ponte entre os dados inseridos na interface do aplicativo e o armazenamento na memória do dispositivo. É possível tratar exceções durante o processo, diferente do que acontece com o *Redux*. As ações realizadas em uma função saga acontecem de forma sequencial – a próxima ação inicia quando a anterior termina;

A biblioteca de Redux e Redux-Saga serão detalhadas no próximo tópico, devido as suas importâncias para o *frontend* do projeto.

2.3 Redux e Redux-Saga

React Native usa o sistema de estados para armazenar dados. Esses dados ficam disponíveis ao usuário enquanto a tela está sendo exibida. Quando o usuário sai da tela, os dados são perdidos.

Com a utilização do ***Redux*** [5], esses estados podem ser armazenados na memória local do *smartphone,* e com isso, as informações utilizadas em uma tela ficam disponíveis para as outras.

Abaixo temos alguns componentes centrais do *Redux* [19]:

- *Store:* é um contêiner que armazena estados da aplicação. É imutável, ou seja, não há nenhuma alteração, apenas evolução dos estados;

- *Actions:* são objetos que descrevem o fluxo de dados, e transitam informação da aplicação para o *"store"*. A descrição é feita com o campo *"type"*, identificando o tipo de ação que deve ser feita pelo *Redux*;
- *Reducers:* são funções que definem como os estados mudam. Essas funções precisam ser puras e não podem ter efeitos colaterais. O *"reducer"* tem dois parâmetros: os estados anteriores e a *"action"* passada;

Um evento executado na interface da aplicação envia uma "*action"* contendo os valores a serem armazenados e uma identificação da *"action"*. Esta ação é passada para o "*reducer"* – este gerencia como tem que ser armazenado os dados que vem da *"action"*. Após o armazenamento no *"store"* ser feito, a interface do usuário é atualizada [7]. Este ciclo é ilustrado pela Figura 2.

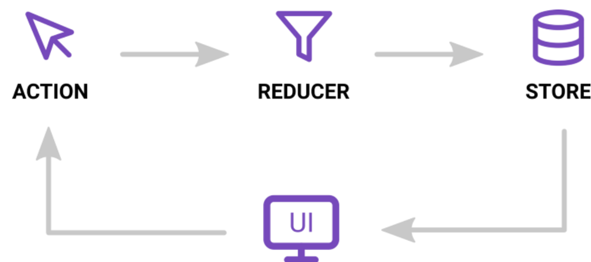


Figura 2 - Componentes centrais do Redux.

Conforme o tamanho da aplicação, o *"reducer"* passa a ter mais estados, dificultando a organização. Para resolver isso, é possível fazer vários *"reducers"* e depois combina-los em um só na hora de criar o *"store"* - isso é feito com a função *combineReducers*.

Os dados armazenados tem como objetivo facilitar o compartilhamento de dados entre componentes do aplicativo. Porém, ao fechar o aplicativo, esses dados são apagados do "*store"* automaticamente. Assim, na próxima vez que o aplicativo for aberto, o *"store"* precisa ser preenchido de novo.

O *Redux* não cobre exceções que possam acontecer durante o processo de armazenamento de dados. A biblioteca ***Redux-Saga*** [6] tem como objetivo ser uma ponte entre a *"action"* e o armazenamento. Nesta ponte, é possível realizar ações que possam retornar erros, como por exemplo, chamadas à API e acesso ao armazenamento local do dispositivo.

O funcionamento do *Redux-Saga* acontece da seguinte forma: uma *"action"* é selecionada para ser observada pelo *Redux-Saga* – caso a *"action"* seja disparada, a

função saga relacionada será executada. A função pode chamar outras *"actions"* dentro de si, que passam por tratamento de erros e são realizadas assincronamente – caso exista uma função que retorne um processamento assíncrono (chamada de *Promise*), a sequência de funções só continua quando o processamento terminar, o que permite que a próxima função tenha acesso ao resultado da função anterior.

2.4 Node.JS

Node.JS é um ambiente de tempo de execução baseado no interpretador V8 do *Google* que permite o desenvolvimento de aplicações escaláveis de rede, em linguagem *JavaScript*. Em uma aplicação *Node.JS*, diversas conexões podem ser controladas ao mesmo tempo. Se houver uma conexão, uma função de retorno é chamada. Caso contrário, o *Node.JS* fica inativo. No *Node*, não existem impasses de processos, pois o *Node.JS* quase nunca utiliza operações diretas de entrada e saída. É projetado para que tenha alta taxa de fluxo e baixa latência ao utilizar o protocolo HTTP (*HyperText Transfer Protocol)* [8].

No MedMob, várias bibliotecas são usadas para construção do *backend*, sendo as mais importantes:

- JWT – Sigla para *JSON Web Token*, a biblioteca é um meio compacto e seguro para representar dados a serem transmitidos entre duas partes. Geralmente é usado em cabeçalhos de autorização HTTP e parâmetros de pesquisa URI (*Uniform Resource Identifier*) [9]. No MedMob, é usado para a geração de *"tokens"* de autenticação e verificação do mesmo para a utilização de rotas do *backend;*
- *Mongoose* – Biblioteca usada na modelagem de banco de dados para o *MongoDB*. Inclui conversão de tipo, validação, construção de busca e outras funcionalidades;
- *Express* [11] – é um *framework* para aplicativos da *web* para *Node.JS,* que oferece recursos robustos de *middleware* e utilitários HTTP. Com o *Express*, é possível ter um sistema de rotas completo, tratamento de exceções na aplicação, gerenciamento de requisições HTTP, além de ser possível ter uma aplicação completa com um conjunto pequeno de arquivos e pastas. Este *framework* é um dos mais usados no mundo por desenvolvedores que utilizam o *Node.JS*;
- *Bcrypt* [12] – é um *framework* para criptografar dados. É usado para evitar que senhas ou outras variáveis sejam salvam integralmente no banco de dados;

- *Cors* – é uma especificação que quando é implementada pelo navegador, permite que um site acesse recursos de outro, mesmo que esteja em um domínio diferente, desde que essa comunicação seja especificada [13]. A especificação é feita de acordo com a API utilizada, onde é incluído o *Access-Control-Allow-Origin* no cabeçalho da resposta da API.

2.5 MongoDB

MongoDB [14] é um banco de dados no estilo NoSQL, onde os dados são armazenados como objetos no formato JSON (*JavaScript Object Notation*), chamados "documentos", em tabelas que são chamadas de "coleções" - diferentemente do banco de dados de modelo relacional, em que usamos linhas, colunas e tabelas. É conhecido por ser um banco de dados não-relacional, isto é, não é possível fazer *links* entre documentos.

Tem como características [15]:

- Consultas utilizando código *JavaScript* – segundo a documentação do *MongoDB*, é possível fazer consultas com funções dos modelos criados. Em tais funções, objetos do tipo JSON são utilizados como filtros para a pesquisa;
- Escalabilidade horizontal – é a necessidade do banco de dados ser adaptável, aumentando o próprio tamanho de acordo com o que os usuários da aplicação adicionam de informação. Também envolve a carga em servidores (aumentando a capacidade de acordo com o uso);
- Indexação – assim como em banco de dados relacionais, o *MongoDB* tem índices para facilitar as pesquisas no banco. Ao criar um documento em uma coleção, é criado uma outra estrutura que contém o valor do campo e o ponteiro para o documento criado.

Tem como vantagens: custo baixo (por ser de código aberto), perfeito para aplicações em *JavaScript*, e bom para ambientes com poucos recursos.

2.6 Mongoose

Mongoose é uma biblioteca para modelagem de objetos para o *MongoDB* [10]. Gerencia relacionamento entre dados, validação de esquemas, e traduz objetos do código para serem usados pelo *MongoDB.* Tem como objetivo ser um facilitador nas operações

de criação, leitura, edição e remoção de dados, além de ajudar na modelagem do banco e criar esquemas.

Ao criar um esquema, é possível definir um tipo, valores iniciais, formatação dos valores inseridos no banco de dados, validadores, dentre outras propriedades. Também é possível definir o tipo de um item como um *ObjectID,* significando que seu valor é o *id* de um documento em uma outra coleção.

Modelos são construtores baseados em esquemas definidos na aplicação. São responsáveis pela criação, visualização, edição e remoção de dados. Ao criar um modelo, é preciso identificar o nome da coleção e o esquema na qual aquela coleção é definida. A partir disso, o *Mongoose* faz uma cópia do esquema e procura pela versão da coleção no banco de dados.

2.7 Socket.IO

Socket.IO [16] é uma biblioteca do *React* e *Node.JS* que habilita a comunicação bidirecional e com baixa latência entre o servidor e o cliente, baseando-se em eventos e utilizando o protocolo de comunicação *WebSocket*.

O cliente irá tentar estabilizar uma conexão via *WebSocket* se possível, e caso não seja, recorrerá a técnica de *Long Polling* do HTTP. Apesar de utilizar *WebSocket* quando necessário, o *Socket.IO* adiciona *metadados* nos pacotes. Sendo assim, um cliente *WebSocket* não consegue se conectar a um servidor *Socket.IO*, e vice-versa.

O *Socket.IO* oferece:

- Confiabilidade
- Reconexão automática
- Funções de retorno de chamada para saber quando a informação chega ao outro lado da conexão
- Envio de dados para vários clientes ao mesmo tempo

No MedMob, usaremos o *Socket.IO* para envio de mensagens entre o médico e o paciente, saber quando uma ação relacionada a uma consulta é feita entre um dos lados, além de enviar ao paciente o tempo até o médico chegar a sua residência.

3. Desenvolvimento do MedMob

Após um breve resumo das principais bibliotecas e tecnologias usadas no projeto, será apresentado o processo de desenvolvimento do MedMob. Serão definidas as especificações do sistema, o processo de desenvolvimento do *frontend,* e então será demonstrado o funcionamento dos serviços que compõem o *backend*.

Na criação do MedMob, algumas especificações foram levadas em consideração: durante a criação do usuário, é possível escolher entre os tipos de usuário, “paciente” e “médico”, e esta escolha não pode ser modificada; a verificação do telefone do usuário pode ser feita durante a criação de usuário ou na tela de perfil; ao criar uma consulta, o usuário do tipo “paciente” espera por um usuário do tipo “médico” aceitar a consulta; é possível reportar uma consulta durante ou depois de já terminada; o usuário do tipo “paciente” pode marcar a consulta para até 24 horas depois do horário atual.

Os requisitos funcionais (RF) do sistema são:

- O sistema deverá permitir o cadastro de usuários (RF01);
- O sistema deve suportar a verificação de número de telefone (RF02);
- O sistema requer autenticação de usuário para a utilização (RF03);
- O sistema deverá permitir a edição de dados do usuário (RF04);
- O sistema deverá permitir a edição da foto de perfil (RF05);
- O sistema deverá permitir a criação, edição e remoção de formas de pagamento (RF06);
- O sistema deverá permitir a criação de consultas médicas, selecionando o local da consulta, a especialidade, forma de pagamento e especificando o motivo da consulta (RF07);
- O sistema deverá permitir o usuário cancelar a consulta (RF08);
- O sistema deverá permitir o usuário do tipo médico procurar consultas de acordo com a especialidade que o médico escolheu na criação de usuário (RF09);
- O sistema deverá permitir o usuário recusar uma consulta (RF10);

- O sistema deverá permitir o usuário reportar a consulta (RF11);
- O sistema deverá permitir o usuário do tipo “médico” visualizar o caminho para o local de consulta em aplicativos de rotas (RF12);
- O sistema deverá permitir o contato entre paciente e médico antes da consulta via *chat* (RF13);
- O sistema deve garantir que tanto o paciente quanto o médico iniciem a consulta para que a mesma realmente inicie (RF14);
- O sistema deve garantir que tanto o paciente quanto o médico terminem a consulta para que a mesma realmente termine (RF15);
- O sistema deve permitir a visualização da lista de consultas (RF16);
- O sistema deve permitir a visualização de uma consulta específica (RF17);
- O sistema deve permitir que o usuário termine a sua sessão no aplicativo (RF18);
- O sistema deve permitir que o usuário encerre sua conta no MedMob (RF19);

Os requisitos não-funcionais (RNF) são:

- O sistema deve exibir os componentes de acordo com o tipo do usuário – médico ou paciente (RNF01);
- A versão mínima do sistema operacional *Android* suportada é a 5.0;
- Para a implementação, construção, atualização e testes do *frontend* foi utilizada a plataforma *Expo* 42.0.0
- Para o banco de dados, será usada a versão do MongoDB 4.4;
- Para a implementação do servidor, será usado o *Node.JS* 14.17.6.

Em relação a modelagem do banco de dados, a Figura 3 mostra o diagrama de coleções da aplicação.

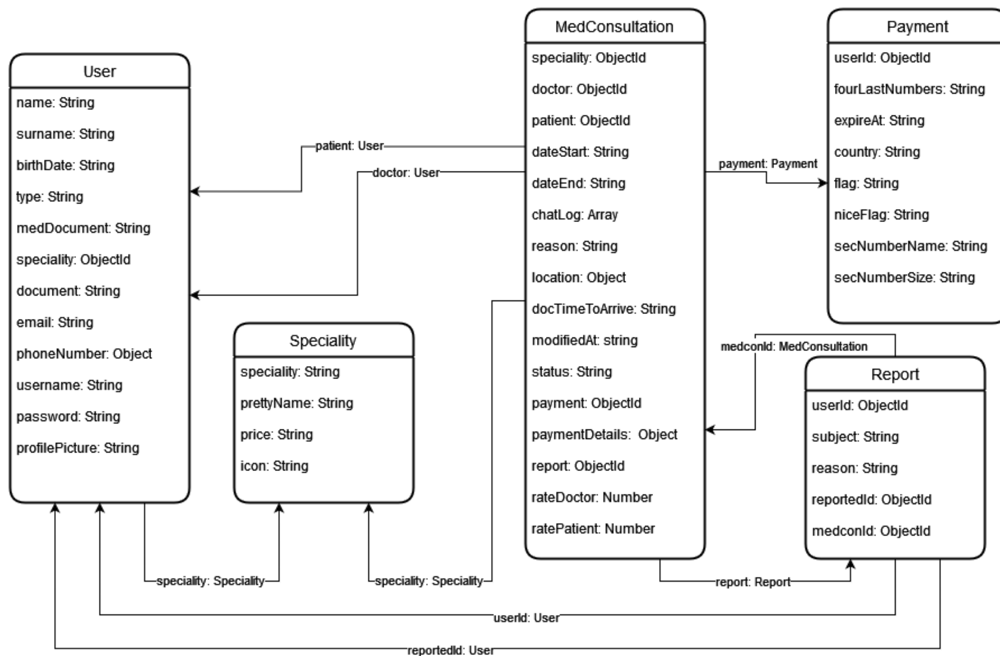


Figura 3 - Diagrama das coleções do banco de dados

A seguir, será apresentado o diagramas de casos de uso, que mostrará uma visão geral do relacionamento dos atores (paciente e médico) com os recursos do aplicativo. Será dividido em dois módulos: módulo paciente, mostrado na Figura 4, e módulo médico, ilustrado na Figura 5.

Figura 4 - Casos de uso dos pacientes

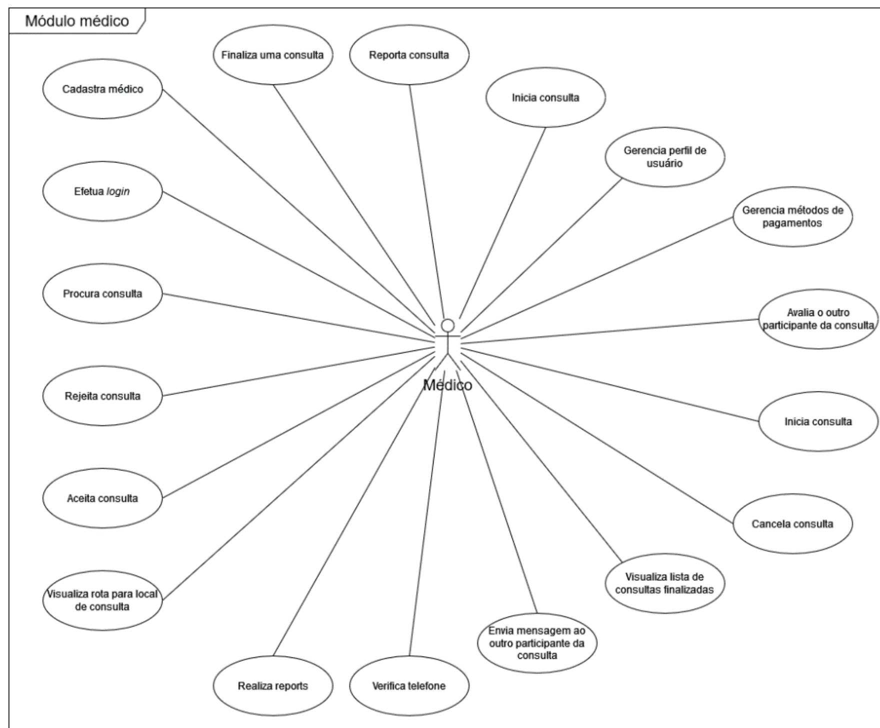


Figura 5 - Casos de uso dos médicos

É possível perceber no diagrama que a maioria dos casos é aplicado a ambos os tipos de usuários. Os casos mais à esquerda em ambos os diagramas são os casos que diferenciam médicos e pacientes.

3.1 Desenvolvimento do *frontend*

A seguir, será apresentado detalhes do desenvolvimento do *frontend*. Primeiramente, as definições de *design* e experiência do usuário. Logo após, serão mostradas as telas do MedMob.

3.1.1 Design e User Experience

As cores influenciam na experiência do usuário e usabilidade do aplicativo. Além disso, as cores transmitem sensações, mesmo que inconscientemente. Por exemplo, as cores vermelha e amarela são remetidas a fome, sendo assim usadas por empresas relacionadas à alimentação. A cor azul, por sua vez, transmite força, dependência e confiança, como podemos ver na Figura 6 [17].

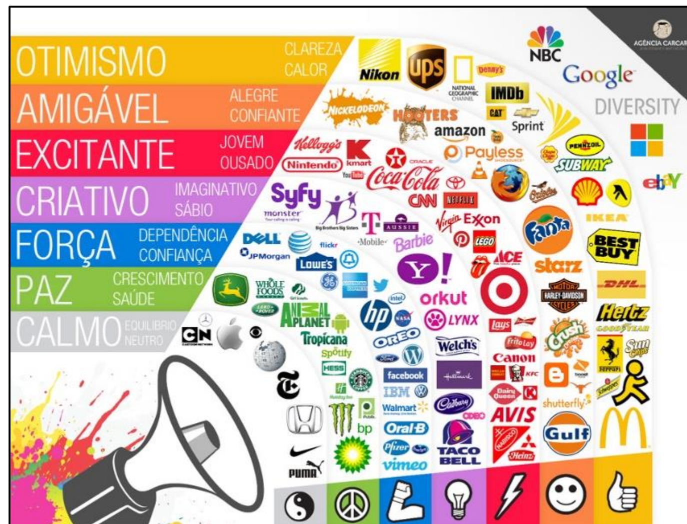


Figura 6 - A relação entre cores, sensações transmitidas e aplicativos.

Dentro do contexto de um aplicativo, as cores representam ações e consequências. Por exemplo, se um aplicativo tem como base a cor azul, e algum botão tem a cor vermelha, o botão indica a realização de uma função irreversível, como por exemplo cancelar um serviço, apagar o próprio usuário, ou excluir um item de uma lista.

O MedMob é um aplicativo relacionado à saúde. A cor verde foi escolhida como a cor base do aplicativo, pois segundo estudos de cores, a cor é relacionada a paz, crescimento, e saúde. Essas cores estão presentes no logo do aplicativo que pode ser visto na Figura 7.

Figura 7 - Logo do MedMob.

A partir da cor verde, foram definidas outras cores que irão compor a paleta primária, mostrada na Figura 8, que serão as cores dos elementos principais do aplicativo. Isso não significa que outras cores não possam ser usadas. Por exemplo, a cor vermelha é usada em componentes que indicam erro ou a exclusão de um item, e a cor laranja é usada em componentes que indicam cancelamento e atenção. Essas cores fazem parte da paleta de cores secundária, mostrada na Figura 9.

Figura 8 - Paleta de cores primária.

Figura 9 - Paleta de cores secundária.

Um preenchimento, da borda da tela para o conteúdo mostrado, é inserido. O tamanho deste preenchimento é definido pela Equação 1.

$$paddingHorizontal = (0.14\ x\ deviceWidth)/1.3$$

Equação 1 - Definição do preenchimento horizontal

A variável *"deviceWidth"* é definida pelo componente *Dimensions*, do *React Native*, que retorna os valores de altura e largura da tela, em *pixels*. Logo, o valor de *"deviceWidth"* varia de acordo com o tamanho da tela do dispositivo, possibilitando que o conteúdo seja mostrado corretamente.

Foram definidos padrões para alguns componentes. Botões que realizam ações de progresso são implementados da forma mostrada na Figura 10. Botões que definem ações secundárias ou direcionamento para a tela anterior são implementados como na Figura 11. A Figura 12 mostra a definição de um botão que realiza cancelamento de ações, ou exclusão de itens. Caixas de entrada de texto são implementadas conforme mostrado na Figura 13. A legenda e o valor de um item são mostrados na Figura 14. A Figura 15 mostra o resultado da implementação do título de uma tela. Por fim, na Figura 16, é mostrado o componente de janela, além dos botões sem preenchimento e borda.

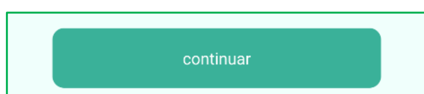


Figura 10 - Botão para prosseguimento de tarefas.

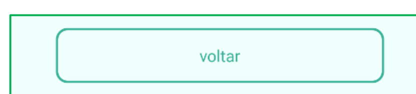


Figura 11 - Botão secundário, para voltar ações/ações secundárias.

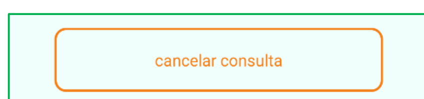


Figura 12 - Botão de cancelamento de ação.

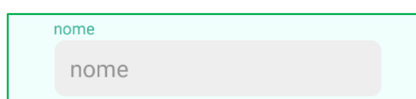


Figura 13 - Caixa de entrada de texto.

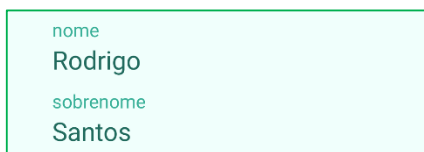


Figura 14 - Legenda de dados e dados.

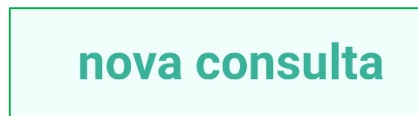


Figura 15 - Título da tela.

Figura 16 - Janela de opções e botões do tipo texto ("galeria" e "câmera").

3.1.2 Tela de login e de cadastro de usuário

A tela de *Login* mostrada na Figura 17 tem dois campos: "usuário" e "senha", onde os valores inseridos são usados para autenticar o usuário no sistema. A tela de cadastro de usuário é acessada pelo texto "Não tenho uma conta!".

Figura 17 - Tela de login.

São quatro etapas para o cadastro de usuário. Primeiro, são informados os dados pessoais e o tipo do usuário, que pode ser "paciente" ou "médico", conforme mostrado na Figura 18. Na etapa dois, caso o tipo escolhido anteriormente seja "paciente", somente o documento de CPF (Cadastro de Pessoa Física) precisa ser informado, conforme vemos

na Figura 19. Caso contrário, além do CPF, devem ser inseridos os campos "número CRM" e "especialidade", igual mostrado na Figura 20.

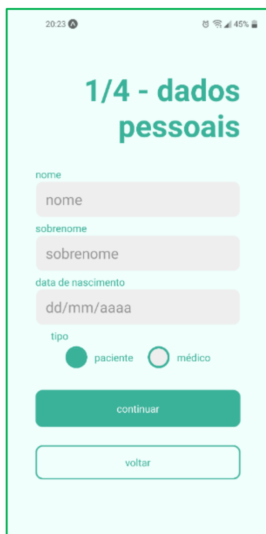


Figura 18 - Tela de novo usuário - 1/5.

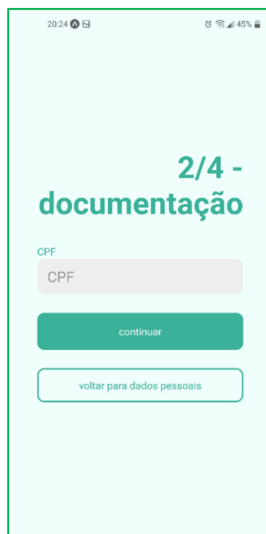


Figura 19 - Tela de novo usuário - 2/5.

Figura 20 - Tela de novo usuário (para médicos) - 2/5.

A tela do terceiro passo é mostrada na Figura 21, onde são informados os dados para autenticação: *"email"*, "telefone celular", "usuário" e "senha". Abaixo do campo de "telefone celular", existe um aviso de que será enviado um SMS (*Short Message Service*) para o número inserido.

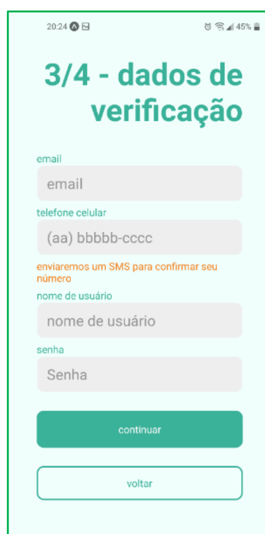


Figura 21 - Tela de dados de verificação.

No quarto passo, ilustrado na Figura 22, o usuário é informado de que um SMS será enviado para o número de telefone informado no passo anterior. Ao prosseguir, a função que realiza o envio da mensagem é acionada, e o usuário é levado à tela de inserção de código de verificação, apresentada na Figura 23, onde existe um único campo para

inserir os seis números que o usuário recebeu no SMS. Assim, o telefone é verificado, e o aplicativo redireciona o usuário à tela de *login*. É possível fazer essa verificação posteriormente na tela de perfil.

Figura 22 - Tela de envio de código para o usuário.

Figura 23 - Tela de envio de código a ser verificado.

3.1.3 Tela inicial

A tela inicial é mostrada na Figura 24. É composta de um contêiner com as duas consultas mais recentes já terminadas (ao apertar no mini histórico, o usuário é levado à aba de histórico de consultas), além do botão de nova consulta. Este botão varia caso exista uma consulta em andamento – caso o paciente tenha criado uma consulta, este botão é substituído pelo componente ilustrado na Figura 25.

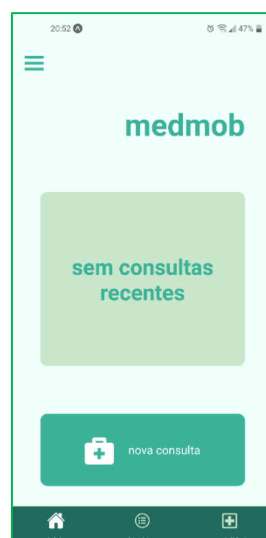


Figura 24 - Tela inicial, nenhuma consulta em andamento.

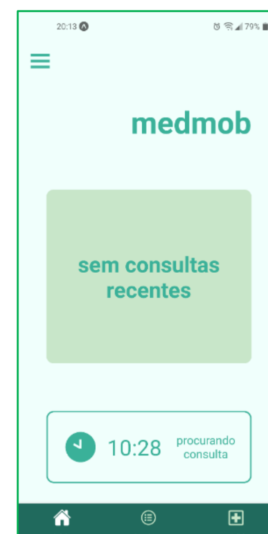


Figura 25 - Tela inicial - paciente procurando consulta.

A Figura 26 mostra o botão da tela inicial com a função de visualizar o andamento da consulta (no caso, com o médico à caminho). A Figura 27 mostra o botão da tela inicial quando a consulta já está em andamento.

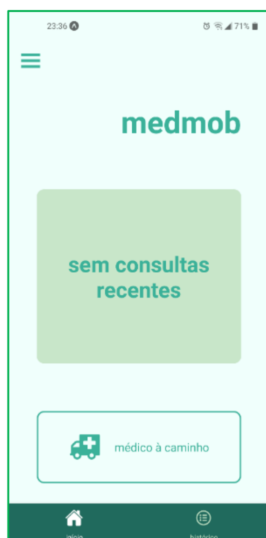


Figura 26 - Tela inicial - Médico à caminho

Figura 27 - Tela inicial - consulta em andamento

3.1.4 Histórico

A Figura 28 mostra a tela de histórico, que contém as consultas do usuário em um componente chamado *FlatList* – este é um componente que carrega a lista conforme a visualização de itens na tela. O *FlatList* evita possível lentidão no aplicativo, caso muitos itens sejam inseridos na tela de uma vez.

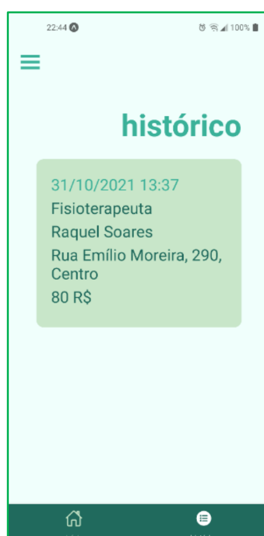


Figura 28 - Aba de histórico de consultas.

Ao pressionar em uma consulta, é possível ver mais detalhes da mesma em uma nova tela, conforme demonstrado na Figura 29. Caso o usuário não tenha feito o *upload* de uma imagem de perfil, aparece um ícone no lugar.

Figura 29 - Tela de visualização de consulta.

3.1.5 Tela de configurações

A tela de configurações desta primeira versão do MedMob está ilustrada conforme a Figura 30.

Figura 30 - Tela de configurações

As opções da tela são:

- Reportar ao MedMob – é aberto uma janela, demonstrado na Figura 31, onde é possível fazer reclamações e sugestões sobre o aplicativo, bem como pedir adição de especialidade não existente no sistema;

Figura 31 - Janela para fazer reports ao MedMob.

- Sobre – informações sobre o aplicativo;
- Modo escuro – habilita/desabilita o modo escuro da aplicação. Não está presente nesta versão do MedMob;
- Opções da conta – a princípio, quando esta opção é pressionada, aparece uma janela, ilustrada na Figura 32, com a opção de excluir a conta. Ao escolher esta opção, a janela muda conforme a Figura 33. É mostrada uma confirmação de exclusão de conta. Esta ação, uma vez que é irreversível, não é feita diretamente na tela de configurações;

Figura 32 - Janela de opções da conta.

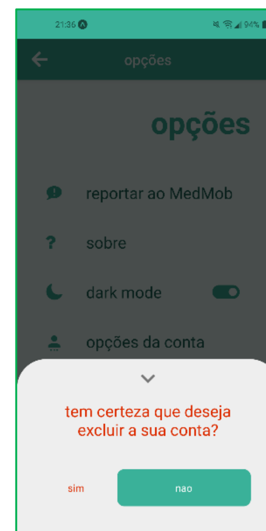


Figura 33 - Janela de confirmação de exclusão da conta.

- Sair – encerrar a sessão da conta e ir para a tela de *login*.

3.1.6 Tela de perfil

Na tela apresentada na Figura 34, são mostradas algumas das informações do usuário que foram informadas no cadastro.

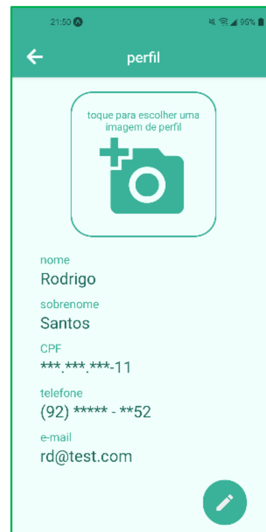


Figura 34 - Tela de perfil do usuário.

Ao clicar no retângulo com um ícone de câmera, uma janela aparece, conforme a Figura 35 – nesta o usuário pode editar a foto de perfil (a janela mostra a foto caso o usuário tenha escolhido uma anteriormente).

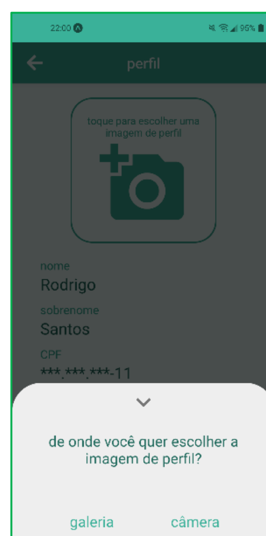


Figura 35 - Janela de visualização e opções para a foto de perfil.

Ao clicar no botão no canto inferior direito da tela mostrada na Figura 34, o usuário é direcionado para a tela de edição, ilustrada da Figura 36. Não é possível editar a especialidade e o CRM (no caso do perfil tipo “médico”).

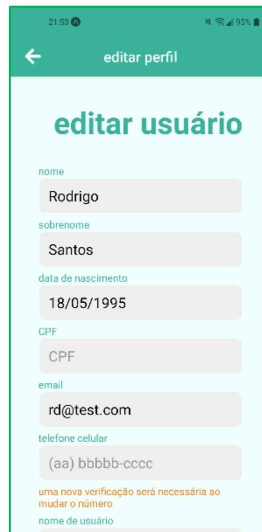


Figura 36 - Tela de edição do usuário.

3.1.7 Tela de métodos de pagamento

A Figura 37 mostra a tela onde são listados os métodos de pagamento cadastrados pelo usuário, além do método de pagamento via dinheiro. É possível ver os quatro últimos números e a bandeira do cartão.

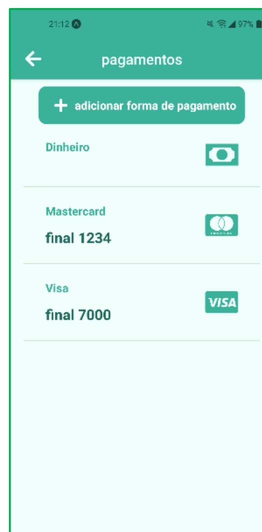


Figura 37 - Lista de métodos de pagamentos.

Ao clicar em um método de pagamento, é possível ver a validade do cartão, além de dois botões – editar e remover, conforme mostrado na Figura 38.

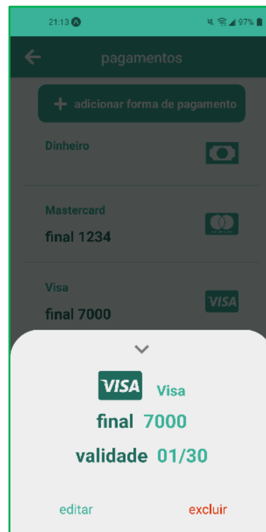


Figura 38 - Janela de pagamento selecionado.

3.1.8 Criação de forma de pagamento

O botão superior da tela apresentada na Figura 37 leva a aplicação à tela ilustrada na Figura 39, onde informações de um novo método de pagamento podem ser adicionados.

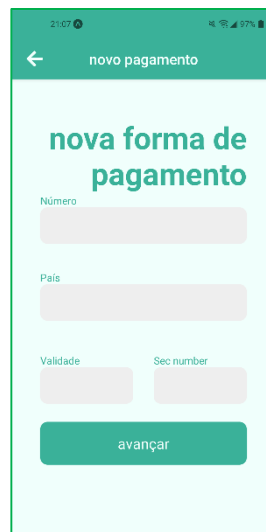


Figura 39 - Tela de criação de forma de pagamento.

O número do cartão define qual a bandeira que irá aparecer embaixo da caixa de inserção de número, além de propriedades do número de segurança – nome de identificação e número de caracteres. A validade é inserida no formato comum de validade de cartão de crédito/débito, MM/AA. Ao apertar em avançar, a forma de pagamento é salva no banco de dados. Nesta versão do MedMob, não é feita uma verificação de método de pagamento válido.

3.1.9 Edição de método de pagamento

Na tela de edição de método de pagamento apresentada na Figura 40, apenas dois campos são passíveis de edição: número de segurança e data de validade (geralmente modificados quando cartões virtuais são atualizados, em alguns serviços de banco *online*).

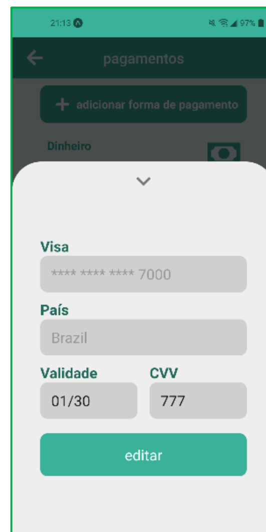


Figura 40 - Edição de método de pagamento.

3.1.10 Tela de informações gerais da consulta

A tela de informações gerais apresentada na Figura 41 tem uma caixa de entrada de texto para que o motivo da consulta seja informado. Ao apertar na caixa de "especialidade", é aberta uma janela com lista de especialidades, conforme ilustrado na Figura 42. O campo "local de consulta" é na verdade um botão que leva à tela de seleção de local de consulta. O botão "continuar" direciona o usuário para a tela de pagamentos.

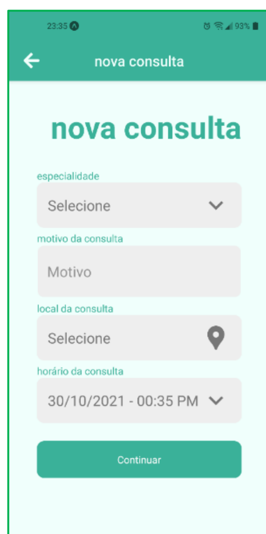


Figura 41 - Informações da nova consulta.

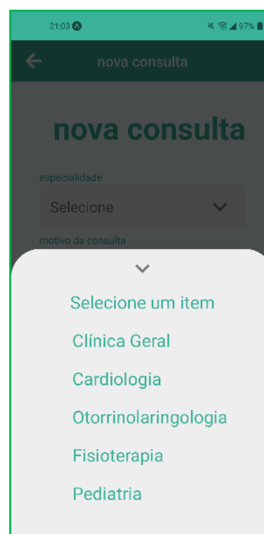


Figura 42 - Lista de especialidades.

3.1.11 Tela de seleção de local de consulta

A figura 43 ilustra a tela de seleção de local de consulta. É possível escolher um local para a realização da consulta de dois jeitos diferentes: pela caixa de texto, onde é possível escolher um local de acordo com o texto inserido (com o uso da API de *Geocoding* do *Google*), e também pelo mapa.

Figura 43 - Tela de seleção do local de consulta.

Ao apertar no botão de "escolher no mapa", é aberta uma janela mostrada na Figura 44, contendo um mapa com a localização atual do dispositivo. Ao mover o mapa, a localização é modificada e indicada por um marcador vermelho. Ao clicar no botão de "concluir", a localização é selecionada e a descrição do local aparece acima da legenda "local de consulta", na tela ilustrada pela Figura 43.

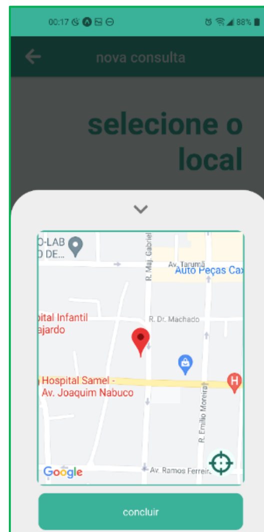


Figura 44 - Mapa de seleção do local de consulta.

3.1.12 Tela de seleção de pagamento da consulta

A tela de seleção de pagamento ilustrada na Figura 45 aparece após todas as entradas na tela de informações gerais serem preenchidas. É possível ver o valor da consulta e a forma de pagamento selecionada.

Figura 45 - Tela de seleção de pagamentos.

O botão "criar consulta" confirma o tipo de pagamento selecionado e as informações anteriormente passadas, e direciona o paciente à tela de espera de consulta. Nesta versão do MedMob, a consulta será criada sem efetuar nenhuma transação com cartões, devido a API de pagamentos não estar implementada ainda. Os valores apresentados são baseados em uma média de valores de consultas por especialidade.

3.1.13 Tela de espera de consulta

Na tela apresentada pela Figura 46, é possível ver quanto tempo o paciente está esperando por uma consulta, além de dados e botões de ações - “cancelar consulta” e “voltar”. É possível cancelar a procura por uma consulta e voltar para a tela inicial. Apenas o usuário do tipo “paciente” consegue ver esta tela.

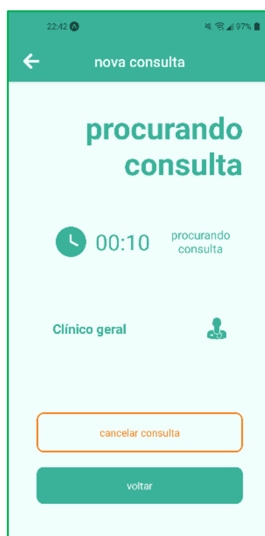


Figura 46 - Tela de espera de consulta.

3.1.14 Tela de procura de consultas

Na tela de procura de consulta aparecem as consultas disponíveis para que o médico possa aceitar ou recusar - ao aceitar, o médico é direcionado para a tela de espera; ao recusar, uma nova consulta aparece (caso exista outra disponível).

Ao procurar uma consulta, aparece um ícone de relógio verde, conforme a Figura 47; ao encontrar pedidos de consultas, aparece um ícone de exclamação laranja, de acordo com a Figura 48. Depois de aproximadamente 200 segundos, caso não seja encontrada nenhuma consulta, este ícone é substituído por um ícone “X” vermelho, com o texto “sem consultas disponíveis” em seguida, conforme mostrado na Figura 49. É possível reiniciar a busca, apertando em “tentar novamente”.

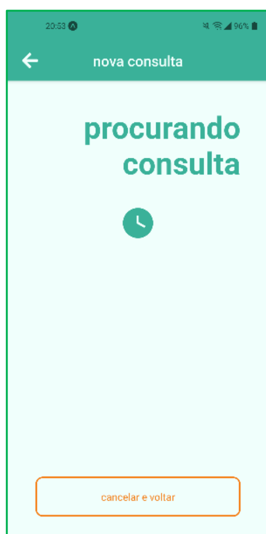


Figura 47 - Tela de procura de consulta para médicos.

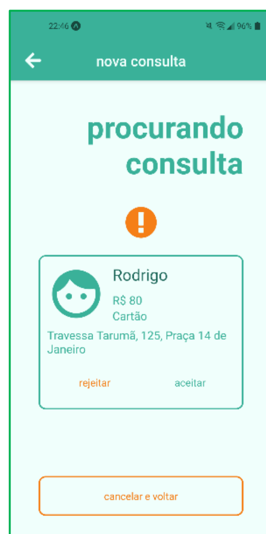


Figura 48 - Consulta encontrada na tela de procura.

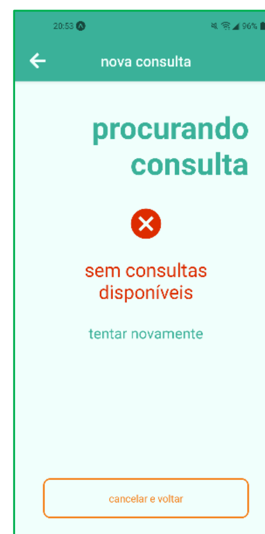


Figura 49 - Nenhuma consulta foi encontrada na tela de procura.

3.1.15 Tela de espera de início de consulta

Nesta tela, o paciente e o médico estão conectados via soquete. A tela é exibida de acordo com o tipo de usuário (paciente ou médico). Para o médico, é possível ver o endereço, especialidade, dois botões para visualizar o trajeto nos aplicativos *Google Maps* e *Waze*, tempo previsto para a chegada ao destino, botão para reportar a consulta, e botões para cancelar a consulta e voltar à tela inicial (ocultos devido ao tamanho da tela), conforme mostrado na Figura 50. A visualização da tela pela visão do paciente é mostrada na Figura 51.

Figura 50 - Tela de médico à caminho.

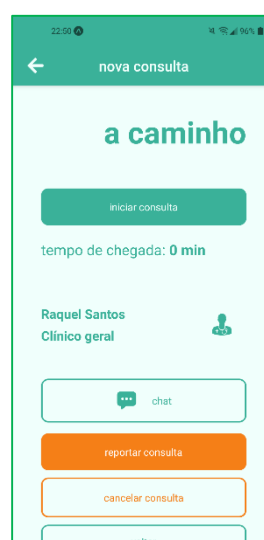


Figura 51 - Tela de espera do paciente.

Um botão com um ícone de balão de diálogo pode ser visto e tem como ação direcionar o usuário para a tela de chat, que é apresentada na Figura 52. Paciente e médico podem trocar informações e tirar dúvidas sobre a consulta, localização, entre outras coisas.

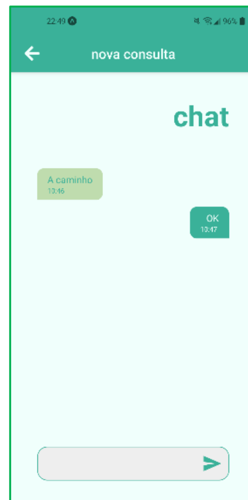


Figura 52 - Chat da consulta

Quando o tempo de chegada é menor que trinta segundos, aparece na tela um botão com o texto “iniciar a consulta” – a consulta só começa quando ambos os usuários pressionam o botão (conforme visto na Figura 51).

3.1.16 Tela de consulta em andamento

A Figura 53 mostra a tela de consulta em andamento. Após paciente e médico aceitarem a consulta, um timer aparece – agora indicando o tempo de andamento da consulta. Além disso, aparecem informações do paciente/médico, um botão para reportar algum problema, e o botão de finalizar consulta. Assim como no botão de iniciar consulta, a finalização da consulta só acontece quando ambos os usuários apertam o botão de finalizar.

Ao apertar em finalizar, aparece uma janela pedindo para que o usuário avalie a consulta, conforme mostrado na Figura 54.

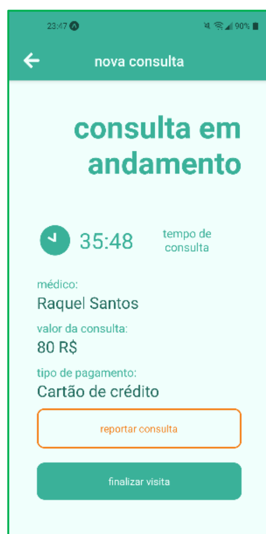


Figura 53 - Tela de consulta em andamento.

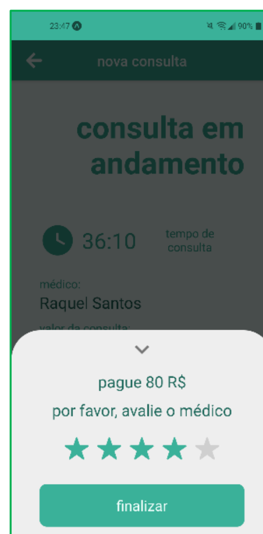


Figura 54 - Janela de avaliação do médico/paciente.

Ao terminar a consulta, o usuário é direcionado para a tela inicial.

3.1.17 AsyncStorage

Na maioria das aplicações em que uma autenticação inicial é necessária para entrar no sistema, é gerado um "*token*" que indica que as credenciais apresentadas pelo usuário são válidas. Este "*token*" pode ou não ter data de validade, dependendo do tipo da aplicação. Geralmente, em aplicativos bancários, a data de validade do *"token"* é curta, e caso o usuário feche o aplicativo, o *"token"* é destruído. Em outros aplicativos em que o *"token"* não tem data de expiração, é necessário que o mesmo seja armazenado no dispositivo.

A biblioteca *AsyncStorage* tem como objetivo guardar informações no armazenamento do telefone celular. Os dados são salvos como cadeia de caracteres (*strings*), identificadas pela biblioteca por palavras-chave - este método pode dificultar o armazenamento de grandes arquivos ou objetos. Neste projeto utilizamos o *AsyncStorage* para armazenar o "*token*" de autenticação, dados do usuário, métodos de pagamentos do usuário e configurações.

A biblioteca *AsyncStorage* armazena dados sem fazer nenhum tratamento ou criptografia – a única exigência é que o dado a ser armazenado seja uma "*string*". Caso o usuário queira que a informação seja protegida, é necessário o uso de outra biblioteca semelhante – *SecureStore*. É uma biblioteca do Expo que realiza a criptografia dos dados antes de armazenar na memória dispositivo.

3.2 Desenvolvimento do *backend*

Este tópico apresenta o desenvolvimento do *backend* – esquemas do banco de dados, funções de CRUDs, funções de serviços de autenticação, validação de variáveis e conexão entre usuários, utilizando diversas bibliotecas do *Node.JS*. As funções do CRUD (*Create, Read, Update, Delete*) são identificadas em **negrito**.

3.2.1 Especialidades

O esquema de banco de dados da coleção de especialidades tem quatro campos, mostrados na Tabela 2.

Tabela 2 - Esquema de especialidades.

Propriedade	**Tipo**	**Descrição**
speciality	*String*	Nome da especialidade
prettyName	*String*	Nome dado a quem realiza tal especialidade
price	*String*	Valor de uma consulta
icon	*String*	Ícone da especialidade

A propriedade "*icon"* é um objeto transformado em sequência de caracteres, cujos campos são: "*iconName"* e "*iconLibrary"* – variáveis utilizadas pelo componente de ícone para saber qual o ícone utilizado e a biblioteca na qual o ícone está presente.

As especialidades, inicialmente, são adicionadas na primeira inicialização do banco de dados.

3.2.2 CRUD de especialidades

A função ***create***, verifica se os campos *"speciality"*, *"nameToDoctor"* e "*icon*" estão presentes. Caso positivo, uma instância do modelo *Speciality* é criada, e então usada a função *save* do *Mongoose* para salvar o documento no banco de dados.

A função ***update*** utiliza a função *findByIdAndUpdate* do *Mongoose* para fazer as mudanças no documento, usando como parâmetros o *id* do documento e os dados que serão modificados.

A função ***find*** do CRUD usa a função *find* do *Mongoose* sem nenhum objeto como parâmetro, para assim obter todas as especialidades.

A função ***delete*** utiliza a função *findByIdAndDelete* do *Mongoose* para realizar a exclusão da especialidade de acordo com o *id* especificado.

3.2.3 Usuário

O esquema de banco da coleção de usuários é demonstrado na Tabela 3.

Tabela 3 - Esquema de usuários.

Propriedade	**Tipo**	**Descrição**
name	*String*	Nome do usuário
surname	*String*	Sobrenome do usuário
birthdate	*String*	Data de nascimento
type	*String*	Tipo de usuário (médico ou paciente)
medDocument	*String*	Número do documento médico
speciality	*ObjectId*	Especialidade do médico
document	*String*	Número de documento
email	*String*	*Email* do usuário
phoneNumber	*Object*	Objeto para o número de telefone
username	*String*	Nome de *login* do usuário
password	*String*	Senha do usuário
profilePicture	*String*	Nome do arquivo relacionado a imagem de perfil do usuário

As propriedades "*speciality*" e "*medDocument*" não são obrigatórios, pois estes são preenchidos apenas se o tipo escolhido no momento da criação do usuário for "médico". A propriedade *"password"* é criptografada com o auxílio da biblioteca *Bcrypt*.

Os campos "*medDocument*", "*document*", "*email", "phoneNumber"* e "*username*" são únicos, ou seja, cada documento da coleção precisa ter um valor único nesses campos.

A propriedade *"phoneNumber"* é um objeto com os campos *"number"* e *"verified"*. No campo *"verified"* é armazenado o código de seis dígitos gerado para a

verificação. Caso a verificação seja feita, o valor de "*phoneNumber.verified*" muda para "*verified*".

O campo *"profilePicture"* tem como título o *id* do usuário junto com a extensão da imagem, para ajudar na diferenciação entre arquivos no diretório de imagens do *backend.*

3.2.4 CRUD de usuários

Ao utilizar a função ***create***, são verificados os campos cujas propriedades são definidas como obrigatórias. Caso todos os campos estejam presentes, a senha apresentada é criptografada pela função *hashSync* do *Bcrypt*, para então ser salva no banco de dados. Em seguida, o número de telefone é inscrito no tópico do MedMob no server de SNS (*Short Notification Service*) do AWS (*Amazon Web Services*). Caso nenhum erro aconteça, é criada uma instância do modelo *User* com os dados a serem criados, e assim, a função *save* do *Mongoose* é utilizada para criar o documento no banco de dados.

A função ***find*** utiliza a função *findById* do *Mongoose*. Apenas o id do usuário é passado como parâmetro. A função tem como objetivo retornar o usuário com o *id* apresentado. Já a função ***findByUsername*** tem como objetivo encontrar um usuário que tenha o mesmo valor de "*username*" que foi passado como parâmetro. É utilizada quando um usuário realiza a autenticação de credenciais no aplicativo, apresentando o "*username*" e "*password*".

A função ***update*** modifica dados de um documento de acordo com o *id* apresentado. Primeiramente é verificado se a senha foi alterada. Caso positivo, a nova senha é criptografada. Em seguida, é usada a função *findByIdAndUpdate* do *Mongoose* para fazer as mudanças. O método ***update*** retorna o documento atualizado.

O método ***delete*** requer um *id* para remover um documento do banco de dados. A função *findByIdAndDelete* do *Mongoose* usa o *id* apresentado para encontrar o documento e removê-lo.

3.2.5 Pagamentos

As propriedades contidas no esquema de pagamentos são mostradas na Tabela 4.

Tabela 4 - Esquema de pagamentos.

Propriedade	**Tipo**	**Descrição**
userId	*ObjectId*	Usuário que é dono do método de pagamento
number	*String*	Número do cartão de crédito
fourLastNumbers	*String*	Os quatro últimos dígitos do cartão
expireAt	*String*	Mês e ano de validade do cartão
country	*String*	País de emissão do cartão
flag	*String*	Bandeira do cartão
niceFlag	*String*	Bandeira do cartão comumente conhecida
secNumber	*String*	Número de segurança do cartão
secNumberName	*String*	Nome do número de segurança (varia de bandeira a bandeira)
secNumberSize	*String*	Quantidade de caracteres do número de segurança

Os campos *"secNumberName"* e *"secNumberSize"* são adquiridos utilizando a biblioteca *credit-card-type,* a partir do número do cartão. O campo *"fourLastNumbers"* é adquirido também pelo número do cartão.

3.2.6 CRUD de pagamentos

A função ***create*** verifica os dados inseridos pelo usuário que correspondem a propriedades obrigatórias do esquema, e a partir daí, é criada uma instância do esquema de pagamento, para então ser usada a função *save* do *Mongoose*.

O método ***update*** verifica se o objeto passado como parâmetro não é vazio, e se existe um campo *"_id"* presente. Caso positivo, é usada a função *findByIdAndUpdate* do *Mongoose* para modificar o documento encontrado.

A função ***find*** retorna todos os métodos de pagamentos do usuário que tenham o campo *"userId"* igual ao *id* passado como parâmetro. A função *find* do *Mongoose* retorna os dados em forma de objeto JSON. É preciso transformar este objeto em um *array* para então ser retornado pela função ***find***.

Duas funções tem como objetivo excluir dados: ***delete*** e ***deleteAll***. A primeira remove apenas um método de pagamento, de acordo com o *id* de pagamento passado como parâmetro. Utiliza a função *findByIdAndDelete* do *Mongoose*, e retorna o *id* do documento excluído. Já a função ***deleteAll*** remove todos os pagamentos de um respectivo usuário, usando a função *findByIdAndDelete* do *Mongoose*, porém o filtro aplicado é a propriedade *"userId"*. Esta função é chamada quando o usuário decide encerrar a conta no aplicativo.

3.2.7 Consultas

As propriedades no esquema de consultas estão listadas na Tabela 5.

Tabela 5 - Esquema de consultas.

Propriedade	**Tipo**	**Descrição**
speciality	*ObjectId*	*Id* da especialidade remetida a consulta
doctor	*ObjectId*	*Id* do médico responsável pela consulta
patient	*ObjectId*	*Id* do paciente responsável pela consulta
dateStart	*String*	Data de início da consulta, escolhida pelo paciente
dateEnd	*String*	Data de fim da consulta, atribuída pelo *backend* quando ambos os envolvidos terminam a consulta, ou a consulta é cancelada
chatLog	*Array*	Mensagens enviadas entre paciente e médico
reason	*String*	Detalhes da consulta fornecida pelo paciente
location	*Object*	Dados de localização do local da consulta
docTimeToArrive	*String*	O tempo necessário para o médico chegar ao local de consulta
modifiedAt	*String*	Data em que modificações na consulta são feitas
status	*String*	Variável que indica o andamento da consulta
payment	*ObjectId*	*Id* do método de pagamento usado

paymentDetails	*Object*	Objeto com detalhes do pagamento
report	*Array*	*Id* de algum *report* feito relacionado à consulta
rateDoctor	*Number*	Nota de avaliação dada pelo paciente
ratePatient	*Number*	Nota de avaliação dada pelo médico

As propriedades *"patient"*, *"doctor"*, *"speciality"* e *"payment"* recebem como valores *ids* de outras coleções. Enquanto as propriedades *"patient"* e *"doctor"* são relacionados a *ids* de usuários, a propriedade *"speciality"* recebe como valor um *id* relacionado à uma especialidade, e a propriedade *"payment"* recebe como valor um *id* de um método de pagamento.

A propriedade *"status"* pode receber os seguintes valores: *"search"*, *"ongoing"*, *"pendingPatientConfirm"*, *"pendingDoctorConfirm"*, *"progress"*, "*pendingPatientFinish*", "*pendingDoctorFinish*", *"ended"* e *"canceled"*. Os "*status"* que contém a palavra "*pending*" indicam que uma ação (ação que precisa que ambos os usuários da consulta médica a efetuem para o prosseguimento do processo) foi realizada apenas pelo paciente ou apenas pelo médico.

A propriedade *"chatLog"* é um *array* onde cada item é um objeto com campos "*message"* (mensagem enviada), *"userId"* (*id* do remetente da mensagem), *"date"* (data de envio da mensagem) e *"reported"* (identifica se a mensagem foi reportada ou não).

Por fim, a propriedade *"paymentDetails"* é um objeto de campos *"paymentType"*, que representa o tipo do pagamento (cartão ou dinheiro), *"datePayment"*, que mostra a data e a hora em que o pagamento foi realizado, e *"value"*, que indica o valor a ser pago.

3.2.8 CRUD de consultas

A função ***create*** verifica os dados passados como parâmetro, adiciona dois novos campos ao objeto – *"rateDoctor"* e *"ratePatient"*, cria uma instância de um modelo de consulta, e então usa a função *save* do *Mongoose*.

As funções que realizam ***update*** recebem como parâmetros três valores: *id* da consulta, tipo de ação e um objeto chamado *"data"*, que contém os dados a serem atualizados em um documento. As funções relacionadas as ações estão listadas abaixo:

- ***accept***: tem como parâmetros o *id* da consulta e o *id* do médico contido no objeto *"data"*. Modifica os seguintes campos: *"doctor",* que recebe valor do *id* do médico, *"modifiedAt"*, que recebe a data atual, e o *"status"*, que recebe o valor *"ongoing"*;

- ***cancel:*** recebe apenas o *id* da consulta como parâmetro. A função *findByIdAndUpdate* modifica dois campos: o campo *"status"* para "*canceled*", e o campo *"dateEnd"* para a data atual;

- ***start***: tem como parâmetros o *id* da consulta, o *"status"* atual da consulta a ser modificada, e o *id* do médico relacionado a consulta. Modifica quatro valores: o *"status",* que pode receber três valores: *"progress"*, *"pendingPatientConfirm"* ou *"pendingDoctorConfirm"*, o campo *"docTimeToArrive"*, que recebe o valor *"arrived"*, e os campos *"modifiedAt"* e *"dateStart"*, que recebem a data atual.

- ***finish***: recebe como parâmetros o *id* da consulta, *"status"* da consulta que será modificada, *id* do usuário que realizou a ação de finalizar a consulta (paciente ou médico) e uma das variáveis a seguir: *"rateDoctor"* ou *"ratePatient"*, que são as avaliações dos usuários. A consulta recebe modificações nos seguintes campos: *"status"* – este é atualizado para *"ended", "pendingDoctorFinish"* ou *"pendingPatientFinish"*, dependendo do *"status"* atual da consulta; *"dateEnd"* – é atualizado com a data atual; *"rateDoctor"* ou *"ratePatient"* – depende de qual dos dois campos apresentar valor diferente de nulo. O usuário cujo *id* é recebido como parâmetro tem o campo *"medconCount"* incrementado, além do campo *"rate"*.

- ***update***: recebe como parâmetros o *id* da consulta e o objeto com os dados que serão atualizados. Esta função tem como objetivo fazer modificações na consulta que as outras funções não cobrem, utilizando a função *findByIdAndUpdate* do *Mongoose*.

As funções apresentadas acima – ***accept, cancel, start, finish*** e ***update*** – retornam o documento da consulta médica que foi atualizado. Além disso, temos a função ***chatLog***, que é responsável por adicionar mensagens entre usuários à consulta, utilizando a função *findByIdAndUpdate* da biblioteca *Mongoose*.

As funções que tem como objetivo o retorno de dados estão apresentadas abaixo:

- ***find***: tem como objetivo encontrar uma consulta específica, utilizando o *id* passado como parâmetro. É usada a função *findById* do *Mongoose*, com a função *populate*, para retornar os dados de especialidade, paciente, médico e pagamento;
- ***findAll***: função que encontra todas as consultas finalizadas ou canceladas que um usuário participou. Os parâmetros passados são: o *id* do usuário, usado para a verificação do tipo do mesmo (paciente ou médico), e então direcionar a procura por consultas pelas propriedades *"doctor" ou "patient"*; número *"n"*, que indica o limite de quantas consultas devem ser retornadas. Caso este valor seja zero ou não definido, a função retorna todas as consultas;
- ***findBySpeciality***: quatro parâmetros são passados a esta função: *"speciality"*, *"reject"* (conjunto de *ids* que devem ser ignorados na pesquisa), *"attempt"* (quantas vezes a rota foi chamada pelo médico, máximo de vinte) e *"location"* (objeto contendo a cidade e as coordenadas do médico). A função *find* do *Mongoose* é utilizada para encontrar as consultas, utilizando os seguintes filtros: *"speciality", "location.city"*, *"reject"* e *"status"* com valor *"search"*. Após encontrar as consultas, é usada a API de *Distance Matrix* do *Google* para obter a distância entre o local da consulta e o local atual do médico (chamaremos este valor de *Delta*), em metros. Esta distância é comparada com o resultado da Equação 2.

  $$Raio = (1000\ x\ attempt) + 1000$$

  Equação 2 - Definição do raio de distância de procura de consultas

  Caso *Delta* seja menor que o valor de *Raio* (em metros), a consulta médica correspondente a localização utilizada pela API de *Geomatrix* é adicionada no retorno da função *findBySpeciality* do *Mongoose*. O raio máximo de procura por consultas é de 21 quilômetros.
- ***findInProgress***: procura uma consulta que esteja em progresso, de acordo com o *id* fornecido pelo cliente. Procura consultas médicas em que o campo *"status"* tenha valores diferentes de *"ended"* ou *"canceled"*.

A função ***delete*** do CRUD remove a consulta do banco de dados. Isso acontece apenas quando o paciente decide cancelar a consulta sem um médico ter aceitado a mesma.

3.2.9 Report

Temos os seguintes campos no esquema de *Report* do MedMob mostrados na Tabela 6.

Tabela 6 - Esquema de Report.

Propriedade	**Tipo**	**Descrição**
userId	*ObjectId*	*Id* do usuário que fez o *report*
subject	*String*	Tema do *report*
reason	*String*	Descrição e motivos para o *report*
reportedId	*ObjectId*	*Id* do usuário que levou o *report,* caso seja feito em uma consulta
medconId	*ObjectId*	*Id* da consulta relacionada ao *report*

É possível fazer dois tipos de *report*:

- Relacionado a uma consulta: é possível relatar algo errado que houve na mesma. Neste caso, *"reportedId"* e *"medconId"* são preenchidos, além dos campos *"userId"*, "*subject*" e "*reason*";
- Reclamações e sugestões a respeito do MedMob: por exemplo: pedir adição de especialidade, reportar *bug*, etc. Só os campos "*subject*", "*reason*" e "*userId*" são preenchidos;

Logo, os campos "*subject*", "*reason*" e "*userId"* são obrigatórios ao criar um *report*, significando que um *report* necessariamente precisa de um tema e uma descrição.

3.2.10 CRUD de Report

Não é possível editar ou excluir um *report*. Logo, temos apenas duas funções no CRUD de *Report*.

A função ***create*** recebe como parâmetro um objeto contendo todos as propriedades presentes no esquema de *Report*. Uma instância do modelo de *Report* é criada, e em seguida é usada a função *save* do *Mongoose*. Logo após, é usada a resposta da função *save* para obter o *id* da consulta criada. A partir disso, é adicionado o *id* ao campo *report* da consulta cujo *id* é igual ao *medconId*, caso o mesmo exista.

A função ***userReports*** tem como objetivo retornar todos os *reports* de um usuário, utilizando como filtro o *id* do usuário passado como parâmetro.

3.2.11 Serviço de Socket.io

As funções de *Socket.IO* tem como objetivo gerenciar e organizar as entradas em salas e emissões de dados. São listadas no arquivo principal do *backend*, dentro da função que inicia o soquete. Então, são passados como parâmetros a instância do *server* de soquete e a variável de soquete criada na inicialização.

Neste projeto, foram criados alguns eventos:

- *medcon* – responsável por conectar um usuário a uma sala cujo nome será o *id* da consulta que é passada como parâmetro;
- *accept* – recebe como parâmetro o *id* da consulta e envia o mesmo *id* ao outro usuário conectado na sala utilizando a função *emit* do soquete, indicando que a consulta foi aceita;
- *cancel* – recebe os seguintes parâmetros: "*id"* da consulta e *"type"*, que é o tipo do usuário (paciente ou médico). Em seguida, chama a função *emit* do soquete para enviar os dois valores recebidos como parâmetros ao outro usuário conectado na sala, indicando que a consulta foi cancelada;
- *endMedcon* – recebe apenas o *id* da consulta como parâmetro. O *id* é enviado para o outro usuário conectado na sala pela função *emit* do soquete, indicando que a consulta foi terminada. Por fim, é usada a função *leave* do soquete para encerrar a conexão entre os dois usuários envolvidos na consulta;
- *start* – recebe dois parâmetros: o *id* e o *"status"* atual da consulta. A função *emit* do soquete é usada para indicar o início da consulta, enviando o *id* e o *"status"* para o outro usuário conectado no soquete;
- *finish* – utiliza a mesma lógica da função *start*, porém com a finalidade de indicar o fim da consulta;
- *chat* – recebe como parâmetros o *id* da consulta, a mensagem enviada (pela variável *"message"*) e o *id* da consulta. Estes parâmetros são enviados ao outro usuário conectado no soquete pela função *emit,* junto com a data atual. Logo após, a mensagem é adicionada no documento relacionado à consulta.

- *timeToArrive* – recebe como parâmetros o *id* da consulta e a variável que indica o tempo que falta para o médico chegar na consulta, chamada de *"docTimeToArrive"*. Utilizando a função *emit,* a variável *"docTimeToArrive"* é enviada ao outro usuário conectado, e logo após, é feita a atualização da propriedade *"docTimeToArrive"* da consulta.

3.2.12 Login

A função de autenticação recebe dois parâmetros: *"username"* e *"password"*. O *Mongoose* faz a procura do usuário pelo *"username"*, usando a função ***findByUsername*** que se encontra no CRUD de usuários.

Caso o usuário seja encontrado, é feita a comparação da senha inserida e da senha que está no banco de dados, usando a função *compare* da biblioteca *Bcrypt.* Se a função *compare* retorna dados, é gerado um *"token"* de autenticação, que é retornado junto com os dados do usuário e uma variável demonstrando que a autenticação foi bem sucedida.

3.2.13 Validação de rotas com o token

Ao usar as rotas da API criada, é enviado no cabeçalho de requisições do *Express* o "*token"* que é gerado na função de *login*. Este *"token"* é verificado pela função *verify* da biblioteca *JsonWebToken* [18]. Caso a verificação seja malsucedida, é retornada uma mensagem de erro; caso contrário, é retornado o *id* do usuário que detém o *"token"*.

Este *id* é usado na função *find* do modelo de usuários criado. Caso o *"status"* da chamada seja igual a 200, a verificação é concluída, e o *backend* pode realizar a função pretendida pelo *frontend*.

Algumas rotas não necessariamente precisam receber o *"token"* no cabeçalho da chamada. Isso por que algumas rotas são executadas antes do *login* do usuário ser feito. Estas rotas são definidas em uma "lista branca", e estão listadas abaixo:

- "*/data*" - rota usada para obter o endereço da imagem de perfil do usuário);
- "*/user/send*" - usada para indicar ao *backend* que é preciso enviar um SMS ao usuário com o código para verificar o telefone;
- "*user/verify*" – rota usada para verificar o código de seis dígitos enviado pelo cliente, com o objetivo de verificar o telefone do usuário;

- "*speciality/all*" – rota usada para obter as lista de especialidades. Essa lista é usada na criação de um usuário cujo tipo escolhido é médico;
- "*/auth/login*" – rota usada para fazer o *login* do usuário. É a partir dessa rota que é gerado o *"token"*, logo essa rota não poderia receber nenhum *"token"* no cabeçalho;

Antes de verificar se o *"token"* apresentado é válido, é preciso verificar se a rota que foi chamada está contida na "lista branca" de rotas. Caso positivo, a função *verify* não é executada. Caso negativo, a verificação do "*token"* é realizada.

3.2.14 Logout

Uma vez que o que define que o usuário está autenticado no aplicativo é o *"token"*, quando o usuário realiza a ação de *logout*, o *"token",* os dados do usuário e os outros dados armazenados no dispositivo (exceto as configurações) são apagados. Isso significa que a ação de *logout* não depende de uma chamada à API.

4. Testes e Resultados

Apesar da grande quantidade de *“features”* presentes no MedMob, alguns recursos precisaram ser testados com mais atenção: *layout*, soquetes, *feedback* de atividades, chamadas à API e verificação de número de telefone celular.

Foram utilizados dois *smartphones*, ambos com sistema operacional *Android*, com tamanhos de telas diferentes: 6,2 polegadas e 5 polegadas. A grande maioria das configurações de componentes tem largura e altura definidas com o auxílio da propriedade *Dimensions* do *React Native*, que obtém os valores de altura e largura da tela. Portanto, os componentes não sofreram problemas quanto a visualização.

Na parte de testes envolvendo soquetes, temos alguns eventos cujos funcionamentos foram testados:

1. Envio de mensagens entre o médico e o paciente;
2. Saber quando um médico aceita uma consulta;
3. Saber quando o paciente ou o médico cancela a consulta;
4. Saber quando um dos lados iniciou a consulta;
5. Saber quando ambos os lados iniciaram a consulta;
6. Saber quando um dos lados finalizou a consulta
7. Saber quando ambos os usuários finalizam a consulta;
8. Enviar ao paciente o tempo até o médico chegar a sua residência

Algumas funções do soquete são utilizadas dentro de sagas – tais funções são realizadas após as chamadas à API. Apesar de acontecer um atraso devido à espera das chamadas à API serem concluídas, isto foi necessário para que as funções do soquete sempre aconteçam depois que o banco de dados já foi atualizado. Logo, o sistema não sinaliza completude de ações utilizando soquetes sem que tais ações sejam completamente concluídas.

O MedMob utiliza um sistema de *feedback* semelhante ao de vários aplicativos: mostra uma mensagem de operação bem-sucedida ou erro no rodapé da tela. Este método é utilizado principalmente em edições de dados, criação de itens e na finalização de

consulta. A Figura 55 mostra o componente sendo mostrado após a finalização de uma consulta, e a Figura 56 mostra o componente de *feedback* quando um erro acontece.

Figura 55 - Feedback de conclusão de consulta médica

Figura 56 - Feedback de erro na tentativa de login

As chamadas de API foram testadas, com atenção especial aos dados de retorno e ações para exceções. O *console* do aplicativo de edição de código *Visual Studio Code* foi usado para verificar os dados retornados pela chamada à API, enquanto as exceções foram tratadas nas funções saga cuja chamada era realizada. Por exemplo, ao cadastrar um método de pagamento, é chamada uma função saga que contém a chamada à API que realiza o cadastro no banco de dados. Caso a função saga não detecte erros, o componente de *feedback* é mostrado conforme a Figura 57 – a variável que define a visualização do componente é gerenciada na própria função saga. A Figura 58 mostra o mesmo componente, porém após o processo de edição de usuário ter sido finalizado.

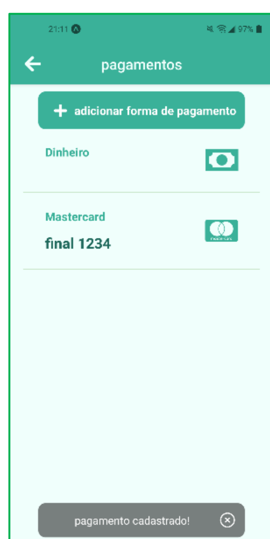


Figura 57 - Feedback de cadastro de pagamento

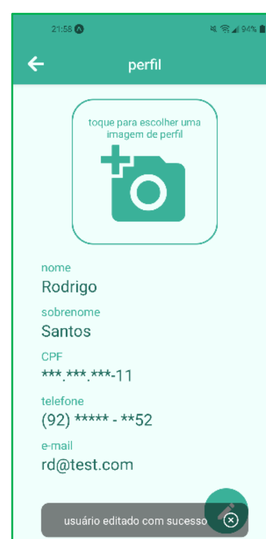


Figura 58 - Feedback de edição de usuário

Ao testar o SNS (*Short Notification Services*) da AWS (*Amazon Web Services*), foi verificado que as mensagens só eram enviadas caso o número de telefone estiver adicionado na lista de telefones de destino da *"sandbox"*. Quando um tópico de

mensagens de texto é criado, o mesmo é definido em modo limitado, conforme ilustrado na Figura 59.

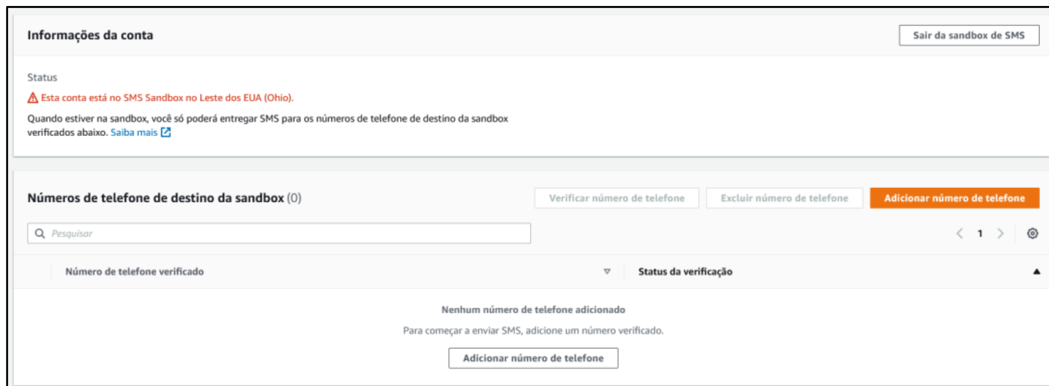


Figura 59 - Sandbox do serviço de Short Notification Services.

É uma política do serviço de SNS para evitar que uma pessoa qualquer envie mensagens para qualquer destino deliberadamente. Este problema foi resolvido ao criar um pedido no suporte da AWS para sair da "*sandbox*" e assim entrar no modo de desenvolvimento. O pedido foi aceito sem grandes problemas, conforme mostrado na Figura 60.

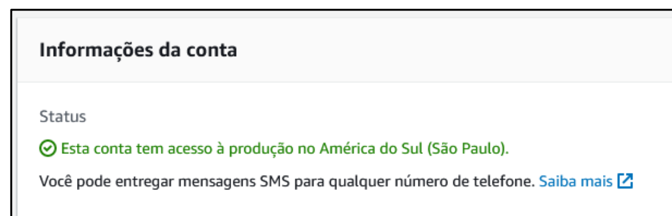


Figura 60 - Acesso ao modo de desenvolvimento do SNS.

5. Conclusões

5.1 Considerações Finais

O objetivo do trabalho foi atingido, que consiste em desenvolver uma aplicação que possibilite a solicitação de uma consulta médica que ocorrerá no local escolhido pelo paciente. A aplicação permite que um paciente crie uma consulta médica e escolha uma especialidade, onde a partir daí, poderá ser encontrada por um médico que utiliza o sistema, e então, caso o médico aceite a consulta, será realizada uma conexão entre o médico e o paciente.

O médico poderá ver o caminho até o local da consulta, tanto pelo MedMob, quanto por aplicativos especializados em rotas de trânsito. Durante todo o processo de consulta, é possível realizar *reports* a respeito do outro usuário envolvido. Ambos os usuários tem que decidir iniciar ou terminar uma consulta para que a mesma inicie ou termine de fato.

Um usuário pode gerenciar o próprio perfil, métodos de pagamentos, foto de identificação, além de visualizar o histórico de consultas já concluídas.

Em relação a desempenho, o MedMob não apresentou problemas de lentidão e paradas devido ao número de bibliotecas utilizadas, imagens apresentadas no aplicativo, ou erros. Ainda assim, é um sistema que tem bastante espaço para melhoramentos.

Quanto ao processo de desenvolvimento, foi tomado o cuidado para que o entendimento do código seja fácil, afim de permitir manutenção rápida e que outros desenvolvedores no futuro possam realizar mudanças no sistema. Para isso, foram usadas as ferramentas *Prettier* e *ESLint*, que definem regras de formatação e codificação de acordo com a linguagem utilizada pelo desenvolvedor.

5.2 Propostas para Trabalhos Futuros

Existe ainda muito espaço para melhoramentos e adições de novos recursos ao MedMob, bem como realizar modificações no *layout* do aplicativo, afim de melhorar a experiência do usuário e deixa-lo mais moderno.

Algumas das mudanças/melhoramentos/adições são:

- Sistema de transações por cartões para pagamento de consultas;

- Sistema de valor de consulta baseado em exigências do paciente;
- Adicionar funções a serem realizadas em *background* (isto é, enquanto o aplicativo não está em evidência, ou não está sendo usado);
    - Considerar usar o serviço de *Cloud Firebase* para envio de mensagens e informações de consultas ao invés do *Socket.IO;*
- Modo escuro no *layout* geral do aplicativo.

6. Referências Bibliográficas

[1]. JAVASCRIPT por MDN WebDocs. Disponível em <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acessado em 27 de maio de 2021.

[2]. CORE COMPONENTS AND NATIVE COMPONENTS, por Facebook Open Source. Disponível em <https://reactnative.dev/docs/intro-react-native-components>. Acessado em 27 de maio de 2021.

[3]. HELLO REACT NAVIGATION por Expo & Software Mansion. Disponível em <https://reactnavigation.org/docs/hello-react-navigation>. Acessado em 30 de maio de 2021

[4]. TREMÜN: Plataforma para entrenamiento cognitivo aplicado al deporte, por Federica Delgado De Leon. Disponível em <http://sedici.unlp.edu.ar/handle/10915/118513>, página 37, tópico 4.1.3.5. Acessado em 16 de novembro de 2021.

[5]. WHY USE REACT REDUX? Por React Redux. Disponível em <https://react-redux.js.org/introduction/why-use-react-redux>. Acessado em 06 de outubro de 2021.

[6]. INTRODUCTION TO REDUX SAGA por Nick Chim, LoginRadius. Disponível em <https://www.loginradius.com/blog/async/introduction-to-redux-saga/>. Acessado em 06 de outubro de 2021.

[7]. CORE CONCEPTS OF REDUX por Dan Abramov. Disponível em <https://redux.js.org/introduction/core-concepts>. Acessado em 30 de maio de 2021.

[8]. ABOUT NODEJS por NodeJS. Disponível em <https://nodejs.org/en/about/>. Acessado em 31 de maio de 2021.

[9]. JSON WEB TOKEN por Michael B. Jones, John Bradley, Nat Sakimura. Disponível em <https://datatracker.ietf.org/doc/html/rfc7519>. Acessado em 31 de maio de 2021.

[10]. MONGOOSE – elegant mongodb object modeling for node.js. Disponível em <https://mongoosejs.com/>. Acessado em 31 de maio de 2021.

[11]. O QUE É EXPRESS.JS? por Ana Paula de Andrade, para Treinaweb. Disponível em <https://www.treinaweb.com.br/blog/o-que-e-o-express-js>. Acessado em 31 de maio de 2021.

[12]. NODE.BCRYPT.JS por Niels Provos and David Mazières. Disponível em <https://www.npmjs.com/package/bcrypt>. Acessado em 31 de maio de 2021.

[13]. ENTENDENDO O CORS por Alexandre Jacques. Disponível em <https://medium.com/@alexandremjacques/entendendo-o-cors-parte-8331d0a777e1>. Acessado em 31 de maio de 2021.

[14]. INTRODUÇÃO AO MONGODB por DevMedia. Disponível em <https://www.devmedia.com.br/introducao-ao-mongodb/30792>. Acessado em 21 de outubro de 2021.

[15]. O QUE É MONGODB? Por Marylene Guedes, Treinaweb. Disponível em <https://www.treinaweb.com.br/blog/o-que-e-mongodb>. Acessado em 21 de outubro de 2021.

[16]. SOCKET.IO - INTRODUCTION por Socket.IO. Disponível em <https://socket.io/docs/v4/>. Acessado em 09 de novembro de 2021.

[17]. A PSICOLOGIA DAS CORES INFLUENCIA O SUCESSO DO SEU APP por Agência Carcará. Disponível em <https://agenciacarcara.com.br/psicologia-das-cores-influencia-o-sucesso-do-seu-app/>. Acessado em 02 de novembro de 2021.

[18]. JSONWEBTOKEN por Auth0. Disponível em <https://www.npmjs.com/package/jsonwebtoken>. Acessado em 31 de maio de 2021.

[19]. THE ONLY INTRODUCTION TO REDUX (AND REACT-REDUX) YOU'LL EVER NEED por Hristijan Stevanoski. Disponível em <https://javascript.plainenglish.io/the-only-introduction-to-redux-and-react-redux-youll-ever-need-8ce5da9e53c6>. Acessado em 11 de novembro de 2021.

[20]. CÓDIGO DE ÉTICA MÉDICA por Conselho Federal de Medicina, Cap. I, Artigo III. Disponível em <https://portal.cfm.org.br/images/PDF/cem2019.pdf>. Acessado em 24 de novembro de 2021.