Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

RAFAEL PERBELINE

PROTÓTIPO DE SMART HOME COM TECNOLOGIAS OPEN SOURCE

Assis/SP
2020

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

RAFAEL PERBELINE

PROTÓTIPO DE SMART HOME COM TECNOLOGIAS OPEN SOURCE

Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando: Rafael Perbeline
Orientador: Prof. MSc. Guilherme de Cleva Farto

Assis/SP
2020

FICHA CATALOGRÁFICA

P427p PERBELINE, Rafael
Protótipo de smart home com tecnologias open source / Rafael Perbeline. – Assis, 2020.

99p.

Trabalho de conclusão do curso (Ciência da Computação). – Fundação Educacional do Município de Assis - FEMA

Orientador: Me. Guilherme de Cleva Farto

1.Smart home 2.Automação residencial 3.Python

CDD001.61

PROTÓTIPO DE SMART HOME COM TECNOLOGIAS OPEN SOURCE

RAFAEL PERBELINE

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: Prof. MSc. Guilherme de Cleva Farto

Avaliador: Prof. Esp. Célio Desiró

Assis/SP
2020

RESUMO

Com o avanço da tecnologia, possibilitou-se o desenvolvimento de aplicações que fazem uso de microcontroladores para controlar sensores que monitoram e controlam o ambiente em que são implementados. Uma destas aplicações são as Smart Homes que por meio da tecnologia permitem que o morador da residência monitore a atividade dos sensores distribuídos pela casa e controle objetos conectados à rede, como lâmpadas e tomadas com a tecnologia de wifi ou bluetooth. A proposta deste trabalho foi de desenvolver um protótipo de Smart Home controlado por um aplicativo móvel, oportunizando que o usuário utilize as funcionalidades disponíveis na aplicação e consulte o status dos sensores que armazenam e manipulam informações no banco de dados por meio de uma API.

Palavras-chave: Smart Home; Internet das Coisas; Python; Raspberry Pi; Automação Residencial; Aplicativo Móvel.

ABSTRACT

With the advancement of technology, it was possible to develop applications that use microcontrollers to control sensors that monitor and control the environment in which they are implemented. One of these applications are Smart Homes that, through technology, allow the resident of the residence to monitor the activity of sensors distributed throughout the house and control objects connected to the network, such as lamps and sockets with wifi or bluetooth technology. The purpose of this work was to develop a Smart Home prototype controlled by a mobile application, allowing the user to use the features available in the application and see the status of the sensors that store and manipulate information in the database through an API.

Keywords: Smart home; Internet of Things; Python; Raspberry Pi; Home automation; Mobile Application.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Com o avanço da tecnologia e a capacidade de minimização dos componentes eletrônicos ao longo dos anos, surgiu a possibilidade de implementação de dispositivos cada vez mais inteligentes que utilizam cada vez menos espaço, além de automatizar o cotidiano das pessoas. Esse avanço deu origem ao termo *Smart Homes*, ou casas inteligentes em uma tradução literal.

Neste contexto se tem o objetivo de criar uma automação residencial que possa ajudar o morador com coisas simples ou até mesmo complexas de forma semiautomática ou completamente automatizada.

Essas residências inteligentes são compostas de dispositivos simples como computadores, *tablets* e demais máquinas domésticas interligadas com sensores, *wearables*, *smart appliances* que se comunicam entre si por meio de uma rede sem fio (PONTE, 2018).

A inteligência artificial aplicada a segmentos da sociedade remete ao conceito de internet das coisas (do inglês, *Internet of Things* ou *IoT*) onde se tem como objetivo conectar todos os objetos físicos em uma rede e permitir que um fique interligado com outro. Esse termo é muito usado quando se fala em *Smart Home*.

Segundo Martins (2017), as casas inteligentes possuem um papel importante para os moradores, pois beneficiam uma melhor qualidade de vida, já que dispõem de controles automáticos de aplicativos e serviços de apoio, além de otimizar o conforto do morador, possibilitar controlar remotamente as aplicações, e reduzir o gasto de energia elétrica.

De acordo com Oliveira et al. (2017) para poder obter otimização dos gastos energéticos, costuma-se usar a estratégia de juntar a técnica de *Smart Home* com a de *Smart Grid*. Para esse mesmo autor, a técnica de *Smart Grid* em termos gerais pode ser definida como um meio de gerenciar dispositivos elétricos de vários domínios e ao mesmo tempo proporcionar eficiência nessa gerência, além de confiabilidade, segurança e qualidade dos serviços, possuindo um conjunto de sete domínios interconectados.

Geralmente com a implementação das *Smart Homes* nas residências são utilizados diversos protocolos diferentes para conseguir uma conexão segura e prática entre todos os dispositivos e sensores interconectados e ainda possibilitar ao usuário usar remotamente algumas das aplicações.

Dentre os diversos protocolos utilizados, existem alguns que se tornam destaque, como o *Bluetooth*, *wi-fi* e um desenvolvido pela *ZigBee Alliance* denominado *ZigBee* que é um protocolo de rede aberto, onde faz uso dos serviços de transporte com os padrões IEEE 802.15.4.

Segundo Ponte (2018) o *ZigBee* é usado normalmente em aplicativos que possuem uma taxa de transferência baixa e necessitam que a bateria tenha uma vida útil longa. Diversos desenvolvedores de *Smart Homes* costumam usar esse tipo de protocolo, por ser algo leve e possuir essa vantagem de não desgastar em excesso a bateria, além de consumir pouca energia o que ajuda na otimização do consumo energético.

Porém o protocolo possui um lado negativo, já que tem baixa taxa de transferência não é recomendado ser usado em aplicações que transmitem faixas grandes de mídias, como áudios, vídeos e imagens.

Em alguns casos, desenvolvedores de *Smart Homes* costumam usar uma tecnologia bastante recorrente nesses últimos anos, o *Raspberry Pi*, que é um computador em escala diminuída, do tamanho de um cartão de crédito, porém com uma capacidade e potência incríveis.

Uma utilização dessa tecnologia nas residências inteligentes foi apresentada por Silva (2014) que fez uso para desenvolver o *eaZy*, que segundo o autor é um sistema de Automação Residencial com objetivo de automatizar o dia a dia do residente da casa, ter informações do ambiente domiciliar e possibilitar controlar até mesmo lâmpadas para que desliguem remotamente sem o esforço do usuário, também foi usado no projeto uma tecnologia que é baseada no *ZigBee* denominada *Zwave* e o próprio *ZigBee*.

Outro caso do desenvolvimento de um ambiente residencial automatizado usando essa tecnologia é o projeto de Patchava et al. (2015), onde foi criado um sistema que controla as luzes da residência e outros dispositivos. Esse controle é realizado por um web site que envia as informações para o dispositivo *Raspberry*.

Mais um projeto desenvolvido em *Raspberry* foi criado por Bhaskar Rao et al. (2015), que desenvolveu um projeto onde o usuário conseguiria controlar as luzes da sua residência remotamente. Também foi criado um monitoramento por sensores de portas e janelas que permitiria verificar se houve alguma invasão na casa, e por meio de uma webcam monitorar mesmo estando longe.

No entanto, existem algumas complicações em relação ao desenvolvimento de uma *Smart Home*. Um desses problemas seria a coexistência de diferentes dispositivos com a mesma frequência, o que pode acarretar interferências entre os aparelhos e inconsistência na rede.

Segundo Ponte (2018), é comum isso acontecer na WHN (*Wireless Home Network*) de uma *Smart Home* por causa da quantidade de dispositivos diversos que possuem a mesma frequência. O autor também cita que tal problema afeta a qualidade de serviços, principalmente a confiabilidade e a latência da rede, e esse problema pode ser agravado por causa da proximidade entre os aparelhos.

1.1 OBJETIVOS GERAIS

O projeto apresentado tem como objetivo principal desenvolver um aplicativo de automação residencial baseado nos conceitos de *Smart Home* e *IoT* que auxilie no cotidiano das pessoas em suas casas e possa promover facilidade em ações básicas realizadas no dia a dia, que seja de fácil utilidade e intuitivo.

A partir dos artefatos desse projeto, foi possível criar uma simulação de uma residencial em escala menor, para poder ser testado o aplicativo desenvolvido e verificar o nível de automação alcançado.

1.2 OBJETIVOS ESPECÍFICOS

Pretende-se com esse projeto desenvolver um aplicativo de automação residencial utilizando os recursos como *Raspberry Pi 3 B,* o *Framework Ionic* 6, a linguagem de

programação Python 3, o interpretador de JavaScript Node Js e as *databases* MongoDB e SQLite, visando melhorar o dia a dia do residente da casa que foi implementado.

Os tópicos abaixo representam os objetivos específicos seguidos ao decorrer do desenvolvimento do trabalho de conclusão de curso:

- Pesquisar e analisar minuciosamente a utilização de *Smart Homes* com as tecnologias *Open Source* citadas, junto com os protocolos *HTTP*, *Wi-fi*, *Bluetooth, etc*;
- Buscar e analisar os melhores sensores utilizados junto com o *Raspberry*, além de verificar como utilizar tais sensores de modo apropriado e por fim adquiri-los;
- Pesquisar algoritmos usados para desenvolver as casas automatizadas e analisar qual a melhor forma de implementá-los sem muitas complicações;
- Implementar os sensores e a lógica da *Smart Home* no *Raspberry*;

1.3 JUSTIFICATIVAS

Com a grande evolução tecnológica que tem acontecido nos últimos anos, as pessoas começaram cada vez mais adequar a tecnologia as suas vidas, tornando quase que parte essencial do seu dia. Essa imersão no mundo tecnológico fez com que buscassem cada vez mais facilidade e conforto, principalmente nas suas próprias casas.

Visando essa busca por conforto que surgiram as *Smart Homes*, que podem facilitar o dia a dia do residente da casa com tarefas simples de ligar uma lâmpada ou até mesmo complexas como sistemas de alarmes e técnicas de detecção de invasões, além de poder ajudar no cuidado de pessoas que tenham dificuldade de fazer coisas por conta própria como indivíduos com idade avançada ou com algum tipo de déficit físico ou mental.

1.4 MOTIVAÇÃO

O desenvolvimento deste projeto consiste no fato da tecnologia estar cada vez mais presente no cotidiano da população, tanto pela facilidade que a mesma proporciona para

as pessoas quanto pela popularidade ao longo dos anos, de tal modo que isso acarretou em uma alta demanda em relação implementações para automação residencial. Isso fez com que o desenvolvimento de *Smart Homes* para automação residencial se torne mais requisitado, pois isso facilitaria a vida dos residentes da casa, além de possibilitar que controlem algumas funcionalidades remotamente sem precisarem se locomover até as suas casas.

Outra motivação para o desenvolvimento desse projeto e talvez a principal, é a criação da Smart Home para auxílio de cuidados de pessoas que não conseguem realizar algumas tarefas por si só, como pessoas com alguma deficiência que dificulte fazer tarefas simples ou até mesmo complexas ou pessoas com idade mais avançada que necessitam de mais cuidados em comparação às demais pessoas.

1.5 PERSPECTIVAS DE CONTRIBUIÇÃO

O protótipo do *Smart Home* desenvolvido nesta pesquisa possibilitou uma comparação com outras aplicações relacionadas a residência automatizada e com tecnologias convencionais, com o objetivo de contribuir com projetos futuros relacionados a essa área.

1.6 METODOLOGIA DE PESQUISA

As propostas e objetivos deste trabalho foram realizados por meio de pesquisas teóricas minuciosas, com o intuito de adquirir conhecimento por meio da análise de artigos científicos, livros, monografias, dissertações, teses, guias práticos e técnicos, livros e fontes digitais confiáveis. Dessa forma foi possível a elaboração e implementação de um projeto de *Smart Home* funcional.

Foi utilizado para o desenvolvimento do projeto a tecnologia de *Raspberry Pi 3 B* que possibilita a utilização de vários módulos e sensores acoplados em conjunto com a linguagem *Python 3* para se comunicar com os pinos que conecta os módulos.

O *web server* foi desenvolvido em uma máquina virtual com o sistema operacional Zorin OS Lite, para poder simular um servidor real. Nele está localizado o *backup* do banco de dados armazenado na database SQLite, junto com a *API* implementada em *Node JS*.

O banco de dados principal foi desenvolvido em *MongoDB* e hospedada na plataforma *online Atlas*, para ser melhor gerenciado, além de diminuir o processamento usado dentro da máquina virtual. Utilizando essa plataforma *online* também pode-se evitar que ocorra perda de dados que poderia acontecer caso fosse utilizado uma versão local do banco.

Os módulos e sensores utilizados no projeto foram escolhidos pensando em obter os melhores dados do ambiente com um custo mínimo. Foram utilizados os sensores de temperatura de ambiente *DHT22* e *DS18B20 waterproof*, interruptores magnéticos *Reed Switch* para detectar se portas e janelas estão abertas e o sensores de movimiento *HC-SR501 PIR* para detectar presenças no cômodo.

A comunicação entre dispositivos foi feita por protocolos *HTTP*, *Wi-fi* e a tecnologia de *Sockets.io e WebSockets* utilizada para parear o cômodo ao raspberry e poder enviar as informações dos sensores.

Foi desenvolvimento em conjunto um aplicativo utilizando o *framework Ionic 6*, o mesmo se comunica com a *API* e retorna as informações do banco de dados, tal aplicativo pode ser utilizado por *Smartphones* (*Android* e *IOS*), *tablets* e computadores pessoais.

1.7 ESTRUTURA DO TRABALHO

A estrutura deste trabalho é composta das seguintes partes:

- **Capítulo 1 – Introdução:** Neste capítulo é contextualizada a área de estudo e apresenta os objetivos, justificativas, motivação, perspectivas de contribuição e metodologia de pesquisa para o desenvolvimento deste trabalho.
- **Capítulo 2 – Tecnologias:** Neste capítulo são especificadas as tecnologias usadas no trabalho, como o microcontrolador utilizado, os sensores e demais métodos.

- **Capítulo 3 – Proposta do Trabalho**: São apresentadas neste capítulo a arquitetura usada no projeto, a prototipação de tela do aplicativo móvel e a ideia de *Smart Home* que será usada no trabalho.
- **Capítulo 4 – Conclusão:** Neste Capítulo são apresentadas as vantagens da *Smart Home* e os objetivos idealizados para o projeto e implementações futuras.
- **Referências**

2. TECNOLOGIAS

2.1 INTERNET DAS COISAS (IOT)

Internet das Coisas ou do inglês *Internet of Things (IoT)* pode ser definido como a interconexão de objetos diversos, onde possibilita a comunicação e compartilhamento de dados e informações de forma inteligente entre si.

Porém o *IoT* está além da conexão de somente objetos, a ideia geral do *IoT* é conectar tudo que se é conhecido em uma única rede, permitindo a comunicação de pessoas entre pessoas, pessoas entre coisas e coisas entre coisas, e até animais com coisas e pessoas de forma automática e transparente, onde cada objeto teria sua identidade única na rede, como pode ser visto na figura 1.

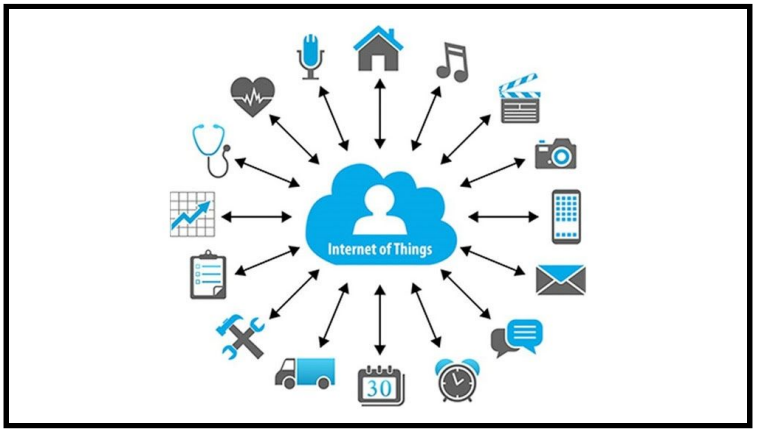


Figura 1: *Internet of Things* (In: Cass, 2018, https://justcreative.com/2018/11/19/internet-of-things-explained/)

Essa conexão seria feita geralmente por meio de redes com e sem fio, onde cada item conectado a rede teria um código *IP*, possibilitando um sistema de rastreamento de fácil acesso, onde tais objetos, pessoas e animais irão gerar um enorme fluxo de dados, que serão analisados por máquinas voltadas para isso, e assim criar informações úteis.

Madakan et al (2015) diz que tais fluxos de dados seriam gerados por sensores e atuadores que são colocados em objetos físicos, e que quando começarem a entender o

ambiente que está ao seu redor, se tornarão mais eficientes e responderão rapidamente ao estímulo ou requisição de dados.

Porém, como todas as tecnologias recentes, existem diversos desafios no quesito de implementação, como explica Gubbi et al (2013). Em seu artigo o autor cita que podem existir problemas em relação da privacidade, análise de dados, eficiência energética, segurança, protocolos usados, entre diversos outros que o mesmo cita em seu trabalho.

2.1.1 Utilização e Aplicações

Essa tecnologia pode ser implementada em diversos áreas, já que é uma metodologia muito versátil e abrangente. A tecnologia pode ser implementada na agropecuária e agronomia com as chamadas *smart farming*, onde o dono da fazenda poderia cuidar do gado e das plantações facilmente e com uma grande precisão, e dependendo da tecnologia usada, seria possível prever pragas e secas.

Outra forma de implementar as técnicas de *IoT* é nas *Smart Homes*, que são residências inteligentes automatizadas que prometem facilitar o dia a dia do morador, de forma que o mesmo possa controlar todas as funções da casa pelo celular, como apagar luzes e desligar aparelhos ligados na tomada, o que pode ser útil para pessoas com problemas de locomoção.

Segundo Gubbi et al (2013) pode-se dividir os aplicativos desenvolvidos seguindo a *IoT* em 4 categorias: aplicativos para uso pessoal e uso na residência, aplicativos empresariais, aplicativos utilitários e aplicativos móveis. Essas categorias foram definidas pelos autores seguindo alguns filtros como tipo de disponibilidade de rede, cobertura, escala, dentre outros.

Um exemplo que pode ser citado de um uso do *IoT* é o trabalho de Song et al (2020) que desenvolveu uma pesquisa para utilizar o *IoT* na medicina com o intuito de criar uma forma para prevenir o vírus SARis, o qual é da mesma cepa que o coronavírus encontrado em Wuhan em 2019.

Song et al (2020) diz que com o *mIoT* (*Medical Internet of Things*) poderia auxiliar os médicos no diagnóstico de novos casos do SARis, usando inteligências artificiais que se

comunicarão uma com as outras, compartilhando dados sobre pacientes e seria usado também dispositivos e sensores conectados nessa rede e Identificadores por Frequência (RFIDs).

Outro exemplo de utilização de *IoT* foi no desenvolvimento do artigo de Mao et al (2017). Em seu trabalho o autor descreve que foi utilizado *IoT* e *Cloud Computing* com o protocolo de rede *ZigBee* para desenvolver uma *Smart Home* funcional com todas as técnicas avançadas dessas tecnologias citadas, sendo aplicação de controle de residência, monitoramento de ambiente, vigilância em vídeo, acesso de multi serviços e ajuste no conforto da residência.

2.2 RASPBERRY PI

Raspberry pi é um minicomputador contendo todos os componentes necessários para a utilização tanto como desktop básico quanto para uso de automação junto com *IoT*. Este microcontrolador possui uma configuração relativamente básica e de baixo consumo de energia, além de ser relativamente pequeno comparado a outras máquinas, sendo do tamanho de um cartão de crédito. A figura 2 abaixo representa uma das versões mais recentes do microcontrolador.

Figura 2: *RaspBerry Pi* 4 B (In: Guse, 2019, https://www.filipeflop.com/blog/lancamento-raspberry-pi-4-model-b/)

Por ter esse tamanho compacto é possível o desenvolvimento de inúmeros projetos, tanto para automação como já citado, como para desenvolvimento de uma plataforma de jogos e protótipos de robôs.

De acordo com Carvalho (2019) o dispositivo foi desenvolvido pela fundação *Raspberry Pi* no reino unido. A organização não possui fins lucrativos e tem como objetivo desenvolver dispositivos de baixo custo que sejam acessíveis para quase todos, com o foco na educação básica da computação para jovens e universidades.

A ideia da fundação e do desenvolvimento do *raspberry* teve início no ano de 2006 na universidade de Cambridge no Reino Unido por um grupo de universitários do laboratório da computação, que usaram como base o microcontrolador Atmel ATmega644 (Carvalho, 2019).

2.2.1 Sensores e Módulos

O *Raspberry* em si é um ambiente muito poderoso para desenvolvimento de projetos simples que não necessitam de muitos recursos para o funcionamento, porém em projetos mais complexos, sozinho não tem a capacidade de suprir a demanda de recursos. Para isso são usados os sensores e módulo. Esses equipamentos adicionais estão cada vez mais abrangentes no mercado ao decorrer da evolução dos computadores em miniatura como *Raspberry* e *Arduino*.

Os sensores são conectados nos pinos que estão do lado do dispositivo, onde cada pino possui uma voltagem e um uma numeração no sistema para poder ser manipulado ou monitorado por meio de programação.

Existem inúmeros sensores para as mais diversas utilizações, como medidores de umidade e temperatura do ar que são a família DHT11 e DHT22, sensores de umidade da terra usados em projetos de agricultura ou até mesmo sensores de gás que possibilitam medir também a temperatura do ambiente como o MQ-2 que é possível detectar butano, GLP e fumaça, também possuindo uma versão melhorada do mesmo, denominado MQ-3 que consegue detectar álcool e etanol.

2.2.2 Arquitetura

Nos últimos anos desde o surgimento do primeiro modelo do *Raspberry*, foram desenvolvidos diversos tipos de dispositivos para os mais variados quesitos e aplicações

mesmo que algumas das novas versões tiveram poucas ou quase nenhuma modificação de um modelo para outro.

De acordo com Carvalho (2019) modelos mais importantes que foram lançados e que estão disponíveis no site para venda são *Raspberry Pi Zero*, *Raspberry Pi Zero W*, *Raspberry Pi 1 Model A*+, *Raspberry Pi 1 Model B*+, *Raspberry Pi 2 Model B*, *Raspberry 3 Model B*, *Raspberry Pi 3 Model B*+. Também existe um novo modelo lançado no período que esse artigo foi feito, o *Raspberry 4 Model B*, mais voltado para *desktop* do que os seus modelos anteriores.

2.2.2.1 Raspberry Pi Zero E Raspberry Pi Zero W

A versão *Raspberry Pi Zero* e *Zero W* possuem quase a mesma especificação de hardware. Ambos possuem uma CPU com 1 GHZ *single-core*, a memória volátil ou memória RAM é equivalente a 512MB em ambas as versões, entradas Mini HDMI, USB OTG e uma entrada USB Power. Também possui um compartimento para conectar os módulos e sensores de 40 pinos e conector de câmera CSI. O que diferencia os dois são os módulos *wireless* 802.11 b/g/n e *bluetooth* 4.1 acoplados no modelo *Zero W*.

2.2.2.2 Raspberry Pi A+

De acordo com Lima (2014) a versão *Raspberry Pi A+* é a sucessora do modelo A, possuindo o mesmo processador de 700MHz, o *Broadcom* BCM2835, e tendo um tamanho de 5,5 cm por 6,5 cm, sendo bem menor que as versões anteriores. Além disso Lima explica que a quantidades de pinos para conectar os sensores é razoavelmente superior quanto a versão A que possuía 26 pinos e nessa foram colocados 40, um crescimento significativo.

Existe somente uma entrada USB, o que dificulta se algum projeto precisar de mais entradas para conectar periféricos, mas Lima (2014) fala que esse problema é facilmente solucionado com a utilização de um *hub* USB. Esse modelo não possui um conector de *Ethernet* acoplado na placa, porém tem uma economia de energia 20% a 25% superior a versões A e B+ como explica Lima (2014). Outras duas coisas que possuem em sua

arquitetura é uma entrada hdmi para conectar em dispositivos de vídeo e uma alimentação de energia feita por uma entrada mini USB com 5V.

2.2.2.3 Raspberry Pi B+

De acordo com Lima (2014) versão B+ do modelo Pi 1 conta com o mesmo processador da versão B, o Broadcom BCM2835 e uma memória volátil igual da versão anterior de 512MB. Além disso comparando a versão anterior, teve um acréscimo de duas entradas UBSs totalizando 4 entradas 2.0.

Como na versão A+, o modelo B+ teve um aumento significativo de pinos para conectar periféricos e sensores em comparação a sua versão mais antiga, passando de 26 para 40, e também nessa versão foi trocada o tipo de entrada para cartão SD, onde deixaram de usar a entrada de SD para colocarem a de MicroSD (LIMA, 2014).

2.2.2.4 Raspberry Pi 2 B

Comparado ao modelo pi 1 essa versão teve uma modificação no processador e na memória ram, como pode ser visto no site oficial da fundação para venda do minicomputador. Foi alterado para um processador *quad-core* de 900 MHz ARM Cortex-A7 e uma memória RAM de 1GB, o que aumentou significamente a potência do PI 2 em relação ao PI 1, com quase o dobro de memória RAM.

Os demais componentes não foram alterados ou tiveram alterações mínimas, possuindo uma conexão *Ethernet*, 4 portas USB, a mesma quantidades de pinos que a versão B+ e A+ de 40 pinos, uma conexão HDMI para ligar em telas e televisões, interface de câmera, entrada micro SD e um processador multimídia de baixa potência *VideoCore IV 3D graphics core*.

2.2.2.5 Raspberry Pi 3 B

Essa seria uma das versões mais recentes que foi lançada junto com o modelo PI 3 B+ e PI 4 B. Pode-se ver que houve um salto na tecnologia usada em relação ao processador

do microcomputador. De acordo com o site oficial da *raspberry pi*, foi colocado um *quad core* 1.2GHz do tipo *Broadcom* BCM2837 64 bit baseado no Cortex-A53. Também foi adicionada a tecnologia *wireless* BCM43438, além de uma saída estéreo de 4 polos e porta de vídeo composta.

Os demais componentes continuaram relativamente o que era na sua versão anterior, com 1GB de memória RAM, 40 pinos para conexão de periféricos, entrada de HDMI, conexão de câmera, entrada micro SD, e conexão de *Ethernet*.

2.2.2.6 Raspberry Pi 3 B+

A versão *Raspberry PI 3 B+* foi o modelo de final da franquia PI 3 como está descrito na página oficial do produto. Nesta página pode-se ver que essa versão contém muitas especificações relativamente melhores do que a sua versão anterior, começando por seu processador, sendo um *Broadcom* BCM2837B0 com Cortex-A53 (ARMv8) 64bit SoC de 1.4 GHz e uma conexão *wireless* de 2.4GHZ e 5 GHZ IEEE 802.11.b/g/n/ac e conexão *bluetooth* 4.2. Os demais componentes continuaram o mesmo de sua versão anterior.

2.2.2.7 Raspberry Pi 4 B

Esse seria a versão mais recente já desenvolvida pela fundação até o momento.Tal modelo possui um diferencial em relação as suas versões anteriores, foi lançado com três variantes do mesmo produto, sendo a de 1GB, 2GB e 4GB de memória volátil ou RAM, com a possibilidade de ser usado como desktop a versão com 4GB de memória RAM.

O salto de tecnologia da linha PI 3 para PI 4 foi realmente incrível, onde o tipo de memória RAM passou de LPDDR2 para LPDDR4-3200, alem de mudar o processador para um *Broadcom* BCM2711 Quad core Cortex-A72 de 1.5GHz que de acordo com o site do processador escrito pela fundação *raspberry pi*, é até 50% mais rápido do que a versão *Raspberry PI 3 B+*.

Também se teve uma evolução na tecnologia *Bluetooth* usada, mudando da versão 4.2 para versão 5 que seria a versão mais recente atualmente, que segundo Cabral (2018)

possui uma capacidade de 2 MBits por segundo de transferência de arquivos e um alcance entre aparelhos pareados de 240 metros.

Com o avanço das tecnologias de porta USB também foi possível acoplar duas entradas de 3.0, que se trata de uma das versões mais rápidas de transferências e leitura de dados que se tem hoje em dia. De acordo com Valin (2019) chega a ser até dez vezes mais rápido que sua versão anterior, possuindo uma taxa de 600 MB por segundo enquanto o modelo 2.0 tinha somente 60 MB por segundo de passagem de dados. O microcontrolador continua tendo 4 portas USBs, porém as outras duas são da versão 2.0.

Além dessa troca de entradas, de acordo com o site oficial do produto, também houve mudanças na tecnologia de HDMI que agora suporta até 4k a 60 quadros por segundo e na tecnologia de emulação 2D e 3D que foi colocado o *OpenGL ES 3.0*. O tipo de abastecimento de energia também foi trocado. Agora o mesmo utiliza uma entrada tipo C, precisando de uma fonte de 5V de no mínimo 3A.

2.3 SENSORES UTILIZADOS

2.3.1 Sensor de Movimento

Esses tipos de sensores podem ser usados para a parte de economia energética, como por exemplo se não houver ninguém dentro de alguma sala que seja pouco utilizado, a luz desse cômodo será apagada instantaneamente, para poder evitar gastos desnecessários.

Outra utilidade dessa categoria de sensor é na parte de segurança, onde o mesmo ficará monitorando se algo se move perto de um local específico que queira deixar seguro, como portas e janelas, assim o sensor irá acionar um sinal para o usuário poder verificar.

Também é possível usar para detectar se alguém está se aproximando para poder ligar algum aparelho ou tela, ideal para implementações em residências inteligentes automatizadas.

Um exemplo de sensor de proximidade ou de movimento é o *PIR Motion Sensor*, que de acordo com o site *raspberry pi tutorials* é um dispositivo que comparado aos demais no

mercado tem um custo relativamente baixo e funciona enviando um sinal somente quando alguma coisa passe pelo sensor.

Outro exemplo que pode ser citado é o sensor ultrassônico HC-SR04, que não se trata de um sensor de movimento em si, porém segundo Heywood (2018) pode ser usado para medir a distância do objeto que está perto usando ondas sonoras, e de acordo com Lima (2017) consegue medir a distância entre 2 a 4 cm.

2.3.2 Sensor de Temperatura do Ar

Sensores dessa categoria são muito usados em sistema de agricultura inteligente para poder medir a temperatura e umidade do ar e com esses valores é possível construir uma media se está ocorrendo uma seca ou não. Também é possível utilizá-los para verificar se um ambiente está muito quente ou muito frio.

Exemplos de sensores assim são o DHT11 e DHT22, que tem a capacidade de medir a temperatura e umidade do ar em uma precisão alta em comparação ao preço vendido e a outros produtos no mercado.

De acordo com o site *raspberry pi tutorials*, a diferença entre os dois sensores é a faixa de detecção de umidade que ambos medem e a precisão que os mesmos têm, além do preço. O sensor DHT11 que possui a cor azulada tem uma precisão de 5% e só é capaz de medir lugares com umidade de 20% a 90%, além de ser mais barato que a sua versão melhorada. Já o DHT22 possui uma precisão de 2% e pode medir áreas de 0 a 100% de umidade.

Outro exemplo de sensor para medição de temperatura são os sensores DS18B20 e DS18S20. São sensores simples que de acordo com a ficha técnica encontrada no site *maxin integrated* ambos tem a capacidade de medir temperaturas de -55º C a +125º C com uma precisão de mais ou menos 0,5º C. A diferença entre esses sensores e os citados anteriormente é que possuem somente a capacidade de medir temperatura.

Mais um sensor que pode ser usado para medir a temperatura do ambiente é o barômetro BMP180. Porém este sensor é mais usado para estações meteorológicas, pois de acordo com Thomsen (2015), o mesmo é voltado para medir a pressão do ar, mas consegue

medir a temperatura também, além de conseguir medir a altitude por meio da pressão atmosférica, sendo excelente para fazer previsões do tempo com precisão.

2.3.3 Interruptor Magnético

Esse tipo de módulo é muito usado em áreas que precisam detectar se um campo magnético foi ativado perto daquele lugar que está o sensor. É também usado geralmente para detectar se portas e janelas foram abertas e fechadas.

2.4 OUTRAS TECNOLOGIAS UTILIZADAS

2.4.1 Api

Uma *API (Application Programming Interface)* pode ser definida como várias rotinas e serviços que possibilita dispositivos ou programas com tecnologias diferentes se comunicarem entre si, além de permitir receber e enviar informações ou dados.

A *API* é como uma ponte entre os dois aparelhos que independente do tipo de linguagem ou sistema operacional usado ainda será possível passar os dados, como um link para que uma requisição seja passada para o *backend* e depois retorne para o *frontend*.

De acordo com Pearlman (2016) essa tecnologia cria funcionalidades independente das suas implementações, sendo assim as definições e implementações podem variar livremente sem que cause algum problema uma na outra.

Segundo Johnson (2014) geralmente as *APIs* são baseados em *Web Services* residentes em hipertexto-padrão, por exemplo *REST*, *SOAP*, *XML-RPC*, *JSON-RPC*. O autor também explica que os *Web Services* baseados em *SOAP (Simple Object Access Protocol)* tem como base os protocolos *WC3*, e são frequentemente associados a *WSDL* e *UDDI*.

O *WSDL (Web Services Description Language)* é uma linguagem baseada em *XML* mais avançada para descrição de definições de um web service (JOHNSON, 2014). O *UDDI (Universal Description, Discovery and Integration)* definido como uma parte fundamental

da arquitetura orientada a serviços, é um diretório onde os sistemas podem registrar e pesquisar serviços (DEVMEDIA, 2009).

Já *Web Services* baseados na arquitetura *REST (Representation State Transfer)* são chamados de *RESTful* e seguem os princípios dessa arquitetura, sendo do tipo cliente e servidor, sem estado e cada solicitação feita pelo cliente para o servidor precisa conter as informações necessárias para que possa ser processada, devendo ser armazenável em cache, ter uma interface uniforme, e um sistema de camadas (RESTFULAPI, 2020).

Os dados nessa arquitetura são acessados por meio de *URIs (Uniform Resource Identifiers)* que segundo Rouse (2016) são uma sequência de caracteres para poder identificar um recurso lógico ou físico, e também usa os métodos *HTTP GET*, *PUT*, *POST* e *DELETE* para manipular os dados segundo Johnson (2014).

Existem vários tipos de *APIs* desenvolvidas e o grande conglomerado da google é uma das principais desenvolvedoras do mercado, possuindo diversos tipos de *APIs* para as mais variadas áreas.

Um exemplo é o sistema da *Google Maps* que de acordo com o site oficial se trata de um conjunto de *APIs* e *SDKs* para que desenvolvedores de software possam integrar o sistema de geolocalização em suas criações. Uma dessas *APIs* muito útil é o *Maps JavaScript API*, que permite customizar os mapas quando colocado no sistema.

2.4.2 Web Server

Um servidor web ou web service pode ser definido de três maneiras. Pode ser uma máquina que armazena os dados e arquivos de um site ou programa web ou baseado em *HTTP (Hypertext Transfer Protocol)*, que entrega essas informações armazenadas para o usuário final em forma de página web ou alguma informação na tela do navegador como estilo *CSS*, scripts em javascript ou até mesmo páginas *HTML* completas (MDN WEB DOCS, 2019).

Outra definição de web server é um software que possui as regras de negócios que são usadas no site para manipular os dados e gerenciar os usuários cadastrados (MDN WEB DOCS, 2019). Também pode ser ambas as definições, onde na máquina irá conter os

arquivos armazenados e as regras e programações para o site funcionar (MDN WEB DOCS, 2019).

O web server irá se comunicar com o navegador por meio de chamadas HTTP (NGINX, 2020), onde o cliente irá fazer requisições para o servidor (*REQUEST*) e o servidor irá devolver as informações solicitadas de volta para o cliente (*RESPONSE*). O servidor web também pode salvar as informações estáticas que são requisitadas frequentemente em cache para poder agilizar o processo da chamada (NGINX, 2020).

Um web server pode ser feito de duas maneiras, de forma estática, com o servidor enviando as informações do banco de dados sem a possibilidade de alterá-la, consistindo em uma máquina com o servidor *HTTP* (MDN WEB DOCS, 2019). Outro tipo de servidor web é o de forma dinâmica, nesse caso a máquina do servidor contém as configurações do server estático, junto a um servidor de aplicações e um banco de dados (MDN WEB DOCS, 2019).

2.4.3 Ionic

Ionic é um *framework open source* voltado para o desenvolvimento de aplicações móveis híbridas com a aparência de uma aplicação nativa de dispositivos móveis, utilizando tecnologias encontradas na construção de aplicativos e sites WEBs como *HTML*, *JavaScript* e *CSS* e integrações de *Angular* e *React* para escrever o código de uma aplicação Ionic (PASSOS, 2018). Mesmo seu foco sendo aparelhos móveis, esse *framework* possibilita desenvolver aplicativos *desktops* também.

2.4.4 Python

É uma linguagem de programação interpretada e orientada a objeto, de alto nível com semântica dinâmica, que possui uma estrutura de código simples (PYTHON, 2020), permitindo assim que usuários leigos possam ter um aprendizado rápido. Seu foco é a legibilidade de código, o que torna programas desenvolvidos com essa linguagem

facilmente interpretáveis, ou seja, os desenvolvedores de Python podem ler e traduzir os códigos facilmente comparados a outras linguagens (PYTHONFORBEGINNERS, 2018).

O *python* oferece opções de digitação e ligações dinâmicas (PYTHON, 2020), criando uma fácil adaptabilidade em relação a programadores inexperientes, ou que estejam migrando para essa linguagem de programação.

Sua utilização é bem variada, já que se aplica em quase todas as áreas por sua facilidade e simplicidade, tanto para aplicativos complexos quanto para scripts simples. Um exemplo de sua utilização é para o desenvolvimento web, já que possui frameworks excelentes e poderosos para essa área, como *Django* que visa as ferramentas necessárias para desenvolver um aplicativo web padronizado (STXNET, 2020).

Essa linguagem possui inúmeras bibliotecas pré criadas tanto pela comunidade quanto pela desenvolvedora oficial, podendo ser usada para quase tudo que precisar, como é o caso de áreas que estão em crescimento na linguagem *Python* como o *IoT*, Inteligência Artificial e *Machine Learning*.

2.4.5 Node Js

Essa tecnologia pode ser definida como um ambiente que possibilita a execução de códigos criados em *JavaScript*, onde geralmente só navegadores conseguiam fazer isso (OPUS SOFTWARE, 2018), possibilitando que programadores especializados em JavaScript possam criar aplicativos auto suficientes diretamente nas máquinas sem que seja necessário usar um browser para poder interpretar o código, além de possibilitar que sejam executados *scripts* diretamente no servidor (*server-side*).

Essa ferramenta foi desenvolvida por Ryan Dahl em 2009, baseado no motor V8 desenvolvido pela Google e implementado no seu navegador Chrome, com o objetivo de desenvolver uma aplicação que pudesse executar *JavaScript* em server-side, uma vez que só era executado em client-side (NODEBR, 2016).

O *Node JS* possui uma arquitetura baseada em eventos, e uma execução de *single-thread*, ou seja, uma única *thread* executa várias requisições enviadas para o aplicativo. Essa *thread* é chamada de *Event Loop* e cada requisição que passa irá ser

tratada como um evento, criando assim um novo evento para cada um que passar nessa *thread* (OPUS SOFTWARE, 2018).

Por conta do node js poder receber várias requisições simultâneas se tem um enorme ganho de desempenho e velocidade, sendo ideal para desenvolvimento de aplicações em tempo real que necessitam de um tempo de resposta muito baixo, como *chats* ou *APIs*.

2.4.6 Mongodb

MongoDB é um banco de dados lançado em 2009 pela empresa 10gen desenvolvido em C++, de código aberto e gratuito (JUNIOR, 2017). Diferente dos bancos de dados relacionais geralmente usados em muitos sistemas que utilizam uma abordagem bidimensional, onde os dados são armazenados em linhas e colunas com um ID para localizar os dados e relacionar com outras tabelas, o *mongoDB* utiliza um sistema orientado a objetos. ou seja, os dados são manipulados a partir do conceito de dados e documentos como explica Medeiros (2014).

Esse tipo de banco possibilita guardar dados importantes em um único documento, além de ter um sistema de indicadores únicos e universais (*UUID*) e possuir uma consulta de documentos por métodos de agrupamentos e filtragem como explica Medeiros (2014).

De acordo com Junior (2017) em bancos de dados baseados em documentos, se tem coleções de documentos, onde cada um é auto suficiente e contém todos os dados que irá precisar. O objetivo disso seria evitar o uso de *JOINS* que podem causar problemas de performance no sistema.

2.4.7 Sqlite

O Sqlite é uma biblioteca de código aberto e domínio público para fins comerciais e pessoais, e funciona implementando um mecanismo SQL transicional independente que não necessita de configurações e servidores como explica Bhosale et al (2015).

Bhosale el al (2015) também explica que o SQLite não usa processos de servidor separado como outros banco de dados que são vistos no mercado, e para isso a

biblioteca lê e grava diretamente no disco. Por ser multiplataforma e compacto esse banco de dados se torna uma escolha popular em diversos sistemas, como celulares e dispositivos com baixa memória e espaço disponível.

2.4.8 WebSockets e Sockets.Io

O *WebSockets* é um protocolo que segundo Marion et al introduziu no *JavaScript* a tecnologia de *sockets*, que define um único fluxo *full-duplex* entre cliente e servidor, que possibilita enviar dados e mensagens. Desse modo é possível estabelecer conexões com vários usuários com sincronização em tempo real via navegador.

Segundo Oliveira (2019) o *client*, que é o nome dado a uma das partes da conexão de *WebSockets*, estabelece uma ligação por meio de um processo chamado de *handshake* pertencente ao protocolo de *websocket*, e de acordo com o autor inicialmente nessa ação o client envia uma solicitação *HTTP* regular para o servidor e um cabeçalho upgrade é incluído que informa para o servidor que o *client* quer realizar uma conexão *WebSocket*.

Oliveira também fala que se for suportado no lado do servidor então irá aceitar a atualização e avisará pelo cabeçalho *upgrade* na resposta da solicitação. Após estar concluída a conexão, o HTTP inicial irá ser substituído por um WebSocket que usa o protocolo TCP/IP, e assim qualquer um dos lados podem enviar dados.

O socket.io, segundo o site oficial de documentação, é uma biblioteca que permite comunicação entre um cliente e servidor em tempo real e bidirecional baseado em eventos entre um navegador e um servidor, onde tal comunicação é feita através de WebSockets na maior parte das vezes. Se não funcionar, a biblioteca irá iniciar uma conexão HTTP.

3. PROPOSTA DO TRABALHO

O presente trabalho tem como objetivo desenvolver um aplicativo de *smart home* que possua as funções básicas usadas em uma automação residencial, como detecção de presença, controle remotamente de luzes e tomadas, ajuste de temperatura ambiente por meio de ar condicionado e comunicação infravermelho, entre outras funcionalidades.

Pretende-se usar as técnicas de *IoT* junto com o mini computador *Raspberry Pi 3 B* conectados com módulos como sensores de distância, sensores de temperatura, transmissores infravermelhos, detectores de presença, tomadas controladas a distância, luzes controlada remotamente, interruptores magnéticos para detectar abertura de portas e janelas e transmissores de onda de rádio.

Além dessas tecnologias, também será usada a linguagem de programação *python* junto com o interpretador de *JavaScript NodeJS* para poder desenvolver o *backend* do projeto, onde o *python* será usado para controle dos pinos de conexão do *raspberry* e da parte lógica do sistema, como cálculos matemáticos e o *nodejs* será usado para desenvolver uma *API RESTful* que vai possibilitar que vários dispositivos consumam os dados do banco.

Será usado o banco de dados não relacional *mongoDB* na plataforma *Atlas* para poder armazenar as informações do usuário, como o cadastro dele no sistema, e controlar a autenticação garantindo uma segurança maior no controle da casa inteligente. Também será usado como backup da *database* localmente no *raspberry* a biblioteca *SQLite.*

Será desenvolvido também um aplicativo móvel frontend com o *framework Ionic* denominado “*Smart Automation*”, possibilitando assim que o usuário possa se cadastrar no sistema e poder usufruir das funcionalidades implementadas. O aplicativo que foi criado usou a *API* desenvolvida como forma de se comunicar com o servidor e o banco de dados, consumindo dados do banco e salvando novas informações pelo *API*.

3.1 PROTOTIPAGEM DE TELA

Para que o usuário possa utilizar as implementações feitas na *Smart Home*, foi desenvolvido um aplicativo final com as opções de fazer login e cadastro. Quando o usuário estiver cadastrado no sistema vai ter todas as funcionalidades disponíveis, como mostrado nas imagens abaixo que representam a prototipagem do *app*.

A figura 3 representa a prototipagem da tela inicial do aplicativo móvel, tendo os botões para login e de cadastro.

Figura 3: Tela de boas vindas do aplicativo "*Smart Automation*".

Na figura 4 representa a prototipagem da tela de cadastro do aplicativo móvel, com as informações que precisam ser preenchidas.

Figura 4: Tela de cadastro do aplicativo "*Smart Automation*".

A figura 5 representa a prototipagem da tela de login do aplicativo. Pede as informações de login e senha para que possa entrar no aplicativo.

Figura 5: Tela de *login* do aplicativo "*Smart Automation*".

A figura 6 representa a prototipagem da tela inicial do aplicativo, onde o usuário pode ver as informações dos cômodos favoritos e dos cômodos normais.

Figura 6: Tela inicial do aplicativo "*Smart Automation*".

A figura 7 representa a prototipagem da tela de informações detalhadas do comodo, possibilitando ver todas as informações do mesmo.

Figura 7: Tela do cômodo do aplicativo "*Smart Automation*".

Na figura 8 pode-se ver a prototipagem da tela de funções do aplicativo móvel, nela o usuário pode controlar todas as funcionalidades do aplicativo individualmente.

Figura 8: Tela de funções do aplicativo "*Smart Automation*".

Na figura 9 pode-ser ver a prototipagem da tela de temperatura do cômodo. Nela pode ser vista as informações de temperatura e umidade além de controlar a temperatura.

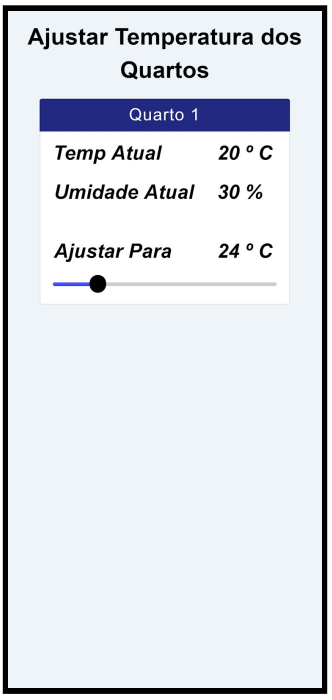


Figura 9: Ajuste de temperatura do aplicativo "*Smart Automation*".

A figura 10 representa a prototipagem de tela de controle de tomadas do cômodo, podendo controlar se as tomadas estão ligadas ou desligadas.

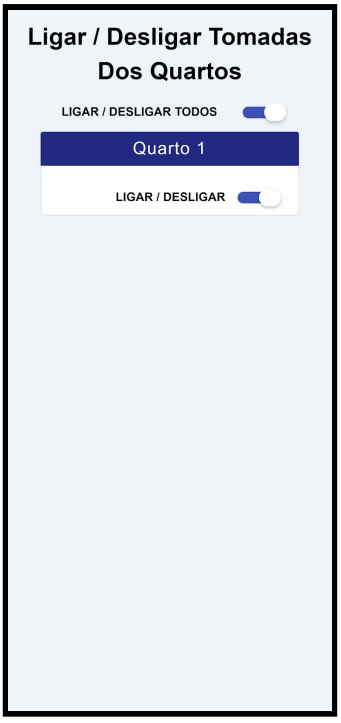


Figura 10: Controle das tomadas do aplicativo "*Smart Automation*".

Pode-se ver na figura 11 a prototipagem da tela de controle de luzes dos cômodos, podendo acender ou apagar uma ou várias luzes.

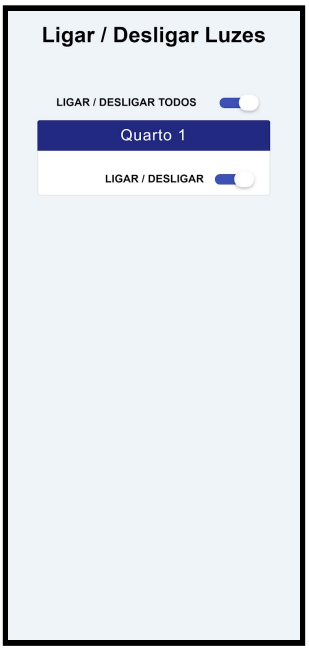


Figura 11: Tela de controle de luzes do aplicativo "*Smart Automation*".

Na figura 12 está a representação da prototipagem da tela de alarmes do aplicativo, onde o usuário pode verificar se detectou a presença de alguém ou se as portas e janelas estão fechadas ou abertas.

Figura 12: Tela de alarmes do aplicativo "*Smart Automation*".

A figura 13 representa a prototipagem da tela de criação de cômodo, onde o usuário precisa colocar o nome do mesmo para cadastrar.

Figura 13: Cadastro de cômodos do aplicativo "*Smart Automation*".

Na figura 14 está a prototipagem da tela de configuração de usuário, onde o mesmo poderá consultar suas informações, mudar dados ou a sua senha

Figura 14: Tela de configurações do aplicativo "*Smart Automation*".

3.2 ARQUITETURA DA SOLUÇÃO

Para que os dados sejam armazenados na *database* principal *MongoDB*, foi desenvolvida uma *API* em *nodejs* que gerencia todas as requisições feitas no aplicativo móvel e no *raspberry*.

Com a finalidade de que o armazenamento funcione, foi necessário criar dois arquivos que serão os *Schema* usado pela biblioteca *Mongoose* na *API* para se conectar ao banco de dados. Esses arquivos serão mapeados pela biblioteca e transformado em coleções do *MongoDB*, além de definir como serão os documentos armazenados.

Nas imagens a seguir (figura 15 e figura 16), pode-se ver os *Schema* do usuário e o *Schema* do cômodo. Ambos possuem uma constante que é a declaração da biblioteca do *Mongoose*, e outra constante que define o tipo *Schema*. Com esse tipo é criado a constante *user* ou cômodo que será definido como um *model* no final do arquivo, com o nome do model e a constante *User* ou Cômodo.

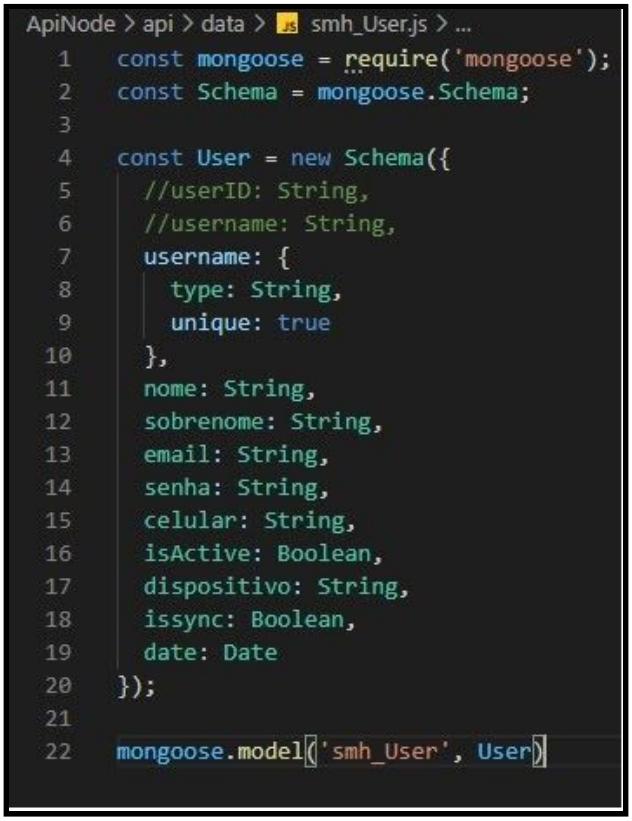


Figura 15: Definição de Schema Usuário na API

Figura 16: Definição de Schema Cômodo na API

Na constante *User* é criado o *username* que foi definido como *unique*, ou seja, não poderá ter um *username* igual a outro, tendo o tipo como *String*. Foi implementado campos do tipo *String* relacionados às informações do usuário, como nome, sobrenome, email, senha, celular e dispositivo. Os campos email, senha e celular são tratados no aplicativo móvel usando expressões regulares.

Também foi implementado os campos *date* que irão conter a data atual de cadastro do usuário. Esse campo é tratado na *API* quando for enviado a requisição *POST* do *user*. Os outros dois campos *isActive* e *issync* do tipo *Boolean* ou verdadeiro / falso são respectivamente para verificar se o usuário está ativo e se será sincronizado com o banco de dados de *backup*.

Na constante Cômodo foi criado um campo para vincular com o usuário, que é o *userID*, sendo do tipo *objectID*, sendo como o *mongoDB* define o *id* dos documentos armazenados, ou seja esse campo so vai poder receber valores *objectID*, se não, irá retornar erro, e como a referência é para a coleção *smh_Users*, se não encontrar nenhum *id* irá retornar erro também.

Foi implementado também nessa constante a definição de 2 campos que irão receber os valores de temperatura e umidade dos sensores conectados no *Raspberry*, sendo respectivamente o *tempComodo* e o *umiComodoF*, ambos do tipo *String*.

Outros campos como *statusJanelaComodo*, *statusPortaComodo*, *statusPresencaComodo* do tipo *Boolean* foram usados respectivamente para determinar se uma janela e porta estão abertos e se o sensor de presença foi acionado.

Os campos isActive, isFavorite, isPareado, issync respectivamente foram usados para determinar se o cômodo está ativo, se é favorito, se está pareado com o *raspberry* e se vai ser sincronizado com a base de dados local *SQLite*.

Para conectar ao banco de dados *MongoDB*, primeiramente precisou-se chamar o método *connect* do *mongoose* e definir a *string* de conexão que o *Atlas* disponibiliza. Para poder usar o novo método de conexão por *string* foi colocado o *useNewUrlParser* como verdadeiro (*true*). Também foi definido *useUnifiedTopology* para usar o novo tipo de topologia, como mostra na figura 17.

Figura 17: Conexão ao banco de dados MongoDB na API

Com os requisitos para conexão prontos, foi adicionado um tratamento de erro com o *then* e *catch* e após isso, realizou-se a criação de duas constantes para resgatar os *Schemas* definidos anteriormente. Pode-se observar isso nas linhas 23 e 24 da figura 17.

Para conectar ao banco de dados *SQLite*, foi implementado primeiramente uma constante que será a localização do banco denominada *DBSOURCE*. Como a constante só possui o nome do banco de dados, então a database é criada na raiz do projeto.

Após definido o caminho do banco, foi desenvolvido 4 constantes denominadas *tableName_User*, *tableName_Comodo*, *smhsqlite_UserCreateTableQuery*, *smhsqlite_ComodoCreateTableQuery*, sendo os dois primeiros são os nomes das tabelas do usuário e do cômodo e os outros dois são para as querys de *create table* do usuário e do cômodo.

Foi desenvolvido uma variável que recebe uma nova instância da *Database SQLite*. Nesta instância foi atribuído a localização do banco e definido uma *function* que recebe erros como parâmetro. Dentro dela existe um teste para tratar a ocorrência de erros, caso não ocorra então será executado as querys de *create table* com o *db.run*, como mostrado nas figuras 18 e 19 abaixo.

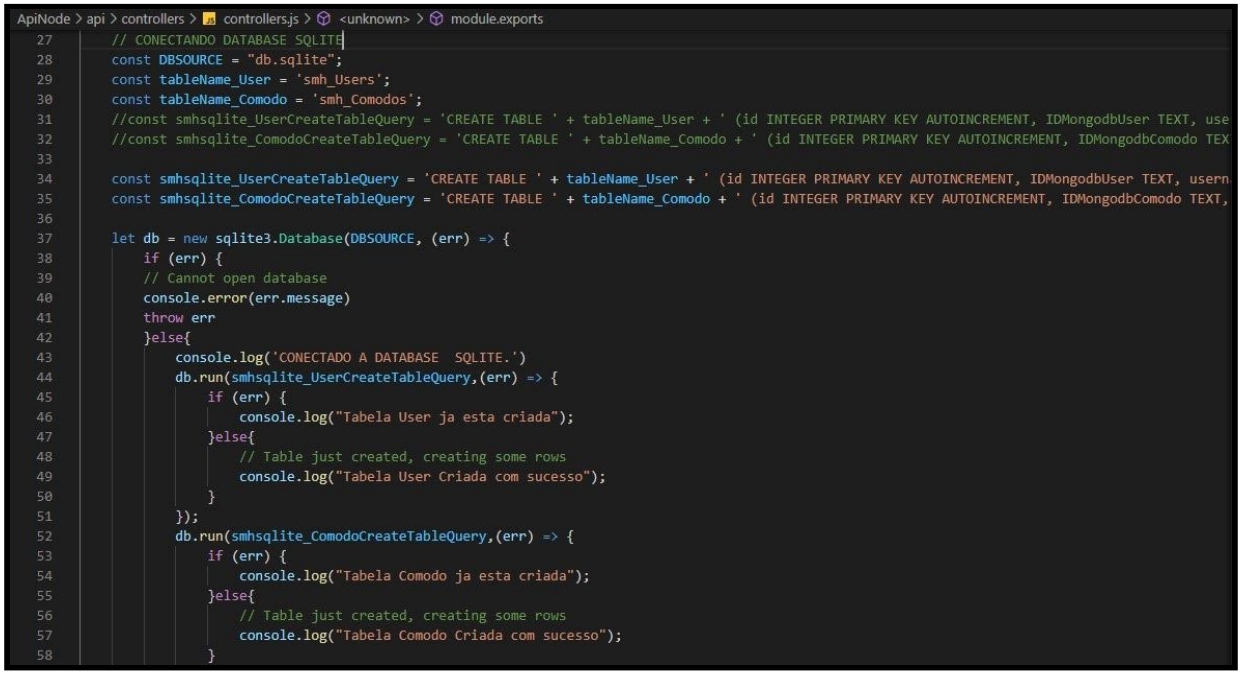


Figura 18: Definição de conexão de database SQLite na API parte 1

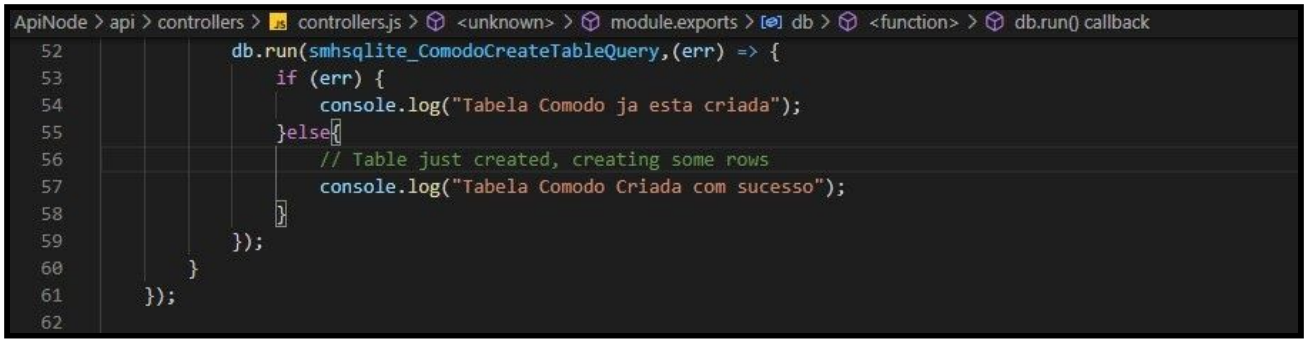


Figura 19: Definição de conexão de database SQLite na API parte 2

Para que os dados fossem buscados nas *databases*, foi usado o método *HTTP GET*. Esse tipo de requisição só possibilita que a API envie dados para o aplicativo móvel quando é chamado.

No banco de dados *MongoDB*, foi utilizado as constantes criadas anteriormente baseadas nos *Schema smh_User* e *smh_Comodo*. Com intenção de buscar todos os dados sem nenhum parâmetro de filtragem foi chamado o método da biblioteca *Mongoose* denominado *find* e atribuindo um objeto vazio, indicando que não terá nenhum filtro de dados.

Foi definido método *then* para receber todos os retornos de sucesso, que serão armazenados no parâmetro comodo e retornado como uma *response* para o aplicativo móvel. Sempre que ocorrer algum erro de busca, entrará no *catch*, sendo armazenado no parâmetro *err* e retornando para o usuário como *response*.

As buscas de dados por meio do SQLite foram implementadas utilizando a constante *db* criada anteriormente. Com essa instância realizou-se a chamada do método *all*, que busca todos os dados definidos pela *query* junto dos parâmetros passados, armazenados na variável *sql* e no vetor *params*.

Foi implementado uma *function* que recebe como parâmetro os erros (*err*) que retornarem e as colunas (*rows*) com os dados. Os erros foram tratados utilizando um condicionamento *if-else*, se o erro for verdadeiro retorna como *response* para o usuário, caso contrário retorna todas as colunas como *response*. Ambos os exemplos para usuário e cômodo podem ser vistos nas figuras 20 e 21 abaixo.

Figura 20: Métodos *HTTP GET* para o cômodo nos bancos SQLite e MongoDB

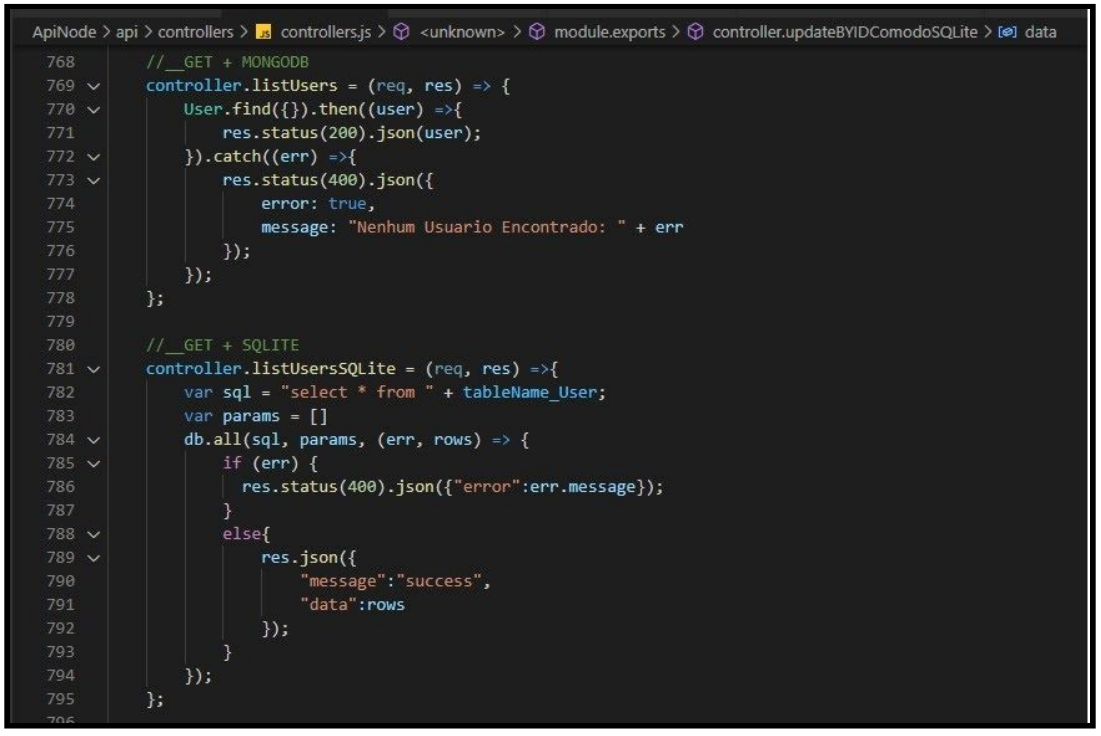


Figura 21: Métodos *HTTP GET* para o usuário nos bancos SQLite e MongoDB

Para buscar dados usando parâmetros no *mongoDB* foram usados os métodos da biblioteca *Mongoose findOne* caso precisar buscar um dado específico ou *find* para vários

dados. Diferente da implementação anterior, foi atribuído um objeto chave-valor como filtro de busca, nesse caso foi utilizado o id do documento.

Com finalidade de realizar a busca com filtros no *SQLite*, foi utilizada a mesma *query* do *GET SQLite*, porém com um acréscimo da cláusula *WHERE* junto com o campo usado como filtro, que no caso foi o *id* da tabela.

O parâmetro passado por *url* foi armazenado na variável *params* e utilizado nas funções *get* da instância *db* caso precisar buscar um item específico ou *all* para caso precisar buscar um conjunto de dados. As figuras 22 e 23 mostram a implementação do cômodo e do usuário

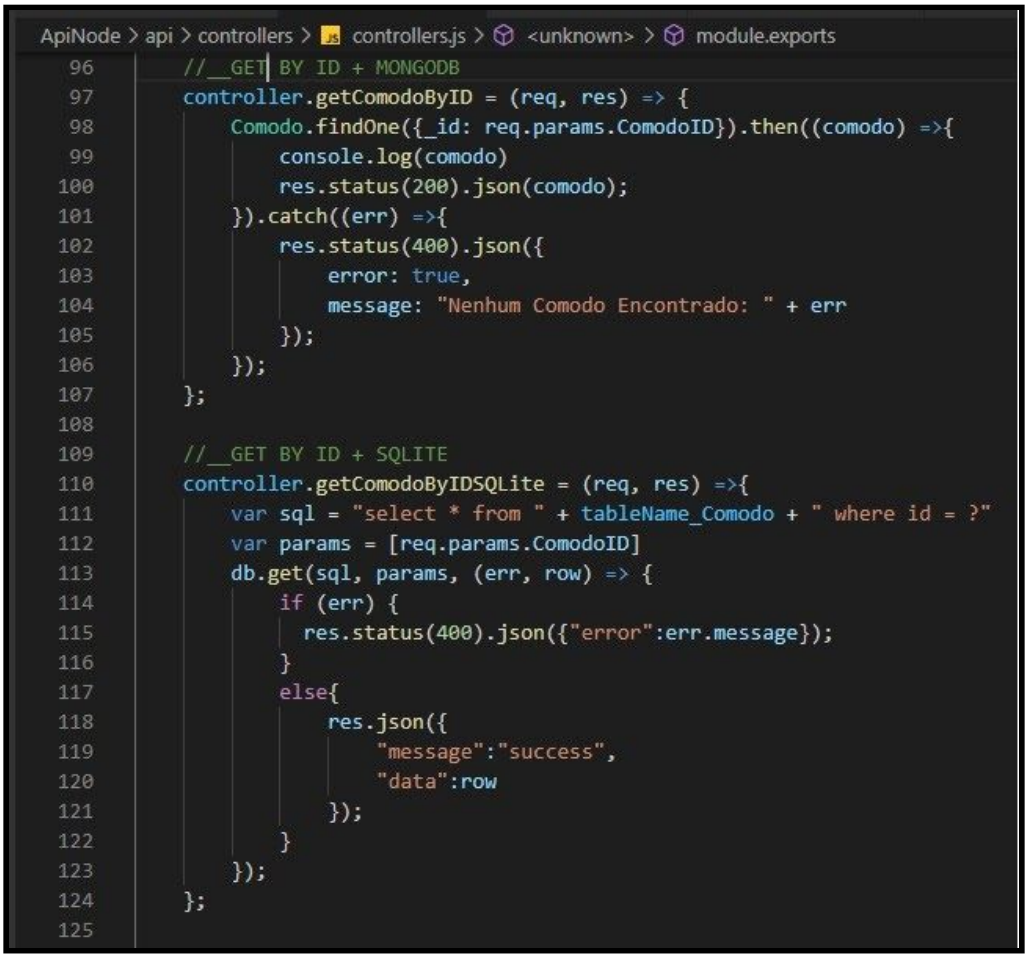


Figura 22: Métodos *HTTP GET* com filtro para o cômodo em SQLite e MongoDB

```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports
797     //__GET BY ID + MONGODB
798     controller.getUserByID = (req, res) => {
799         User.findOne({_id: req.params.userID}).then((user) =>{
800             res.status(200).json(user);
801         }).catch((err) =>{
802             res.status(400).json({
803                 error: true,
804                 message: "Nenhum Usuario Encontrado: " + err
805             });
806         });
807     };
808
809     //__GET BY ID + SQLITE
810     controller.getUserByIDSQLite = (req, res) =>{
811         var sql = "select * from " + tableName_Comodo + " where id = ?"
812         var params = [req.params.ComodoID]
813         db.get(sql, params, (err, row) => {
814             if (err) {
815               res.status(400).json({"error":err.message});
816             }
817             else{
818                 res.json({
819                     "message":"success",
820                     "data":row
821                 });
822             }
823         });
824     };
```

Figura 23: Métodos *HTTP GET* com filtro para o usuário em SQLite e MongoDB

Para implementar o método *POST* com a coleção cômodo foi implementado um objeto de chave-valor com os respectivos campos de um novo documento como chave e os dados passados dentro do body da requisição como valor. Para que o *MongoDB* não ignorasse valores nulos, foi atribuído valores vazios para *Strings* e valores *falses* os *Booleans*.

Foi usado a constante com a instância da coleção de comodo e o método da biblioteca *Mongoose create* para poder adicionar um novo documento não coleção do cômodo, passando o objeto criado anteriormente. Os erros são armazenados no parâmetro *err* e o cômodo criado no parâmetro c.

Para poder salvar os erros e mensagens durante todas as operações, foi implementado dois vetores denominados *messages* e *errors*, possibilitando enviar um *response* com todas as informações que sucederem, como mostrado na figura 24 abaixo.

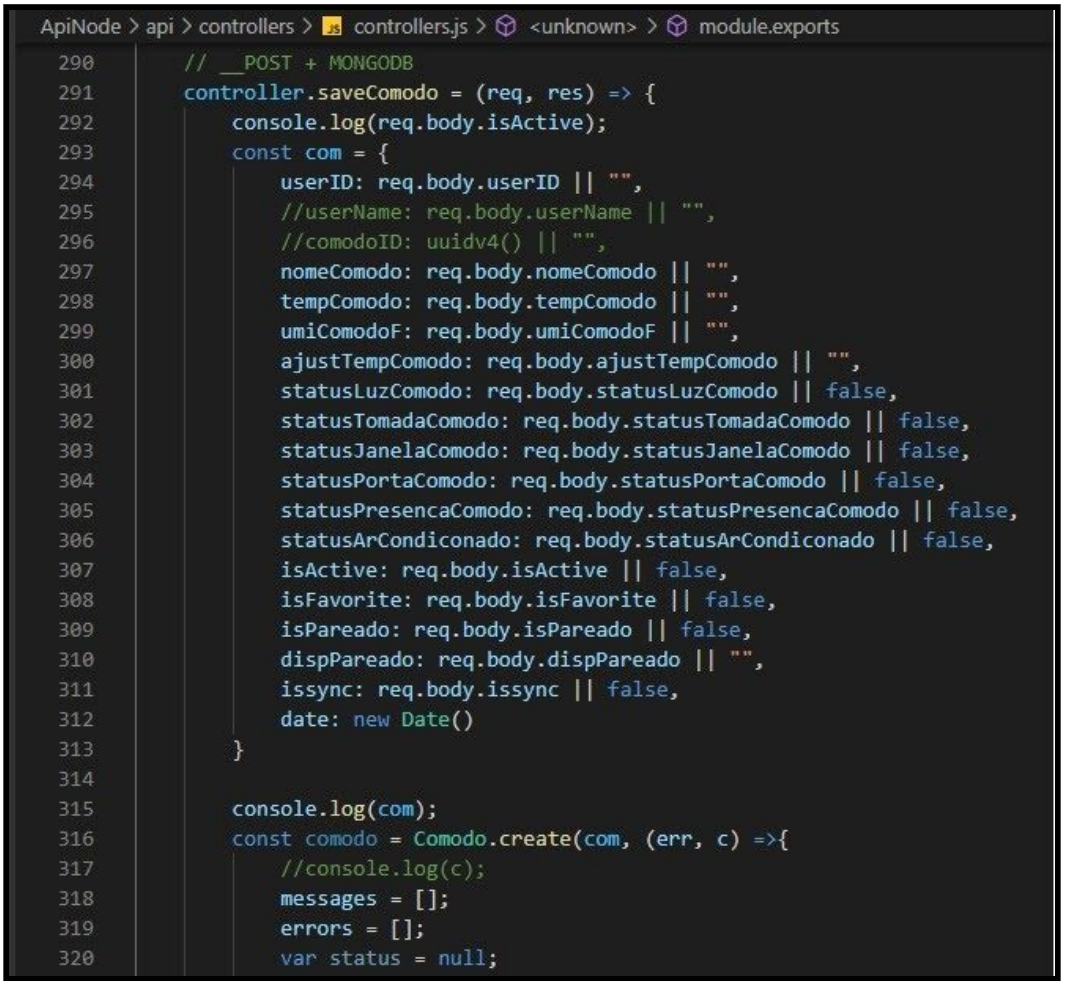


Figura 24: Método Post cômodo MongoDB parte 1

O tratamento de erros dentro do método *create* foi feito utilizando o condicionamento *if-else*. Se o erro for verdadeiro, entra no *if* e armazena no vetor de erros e envia pelo *response* junto com o vetor de mensagens.

Caso contrário entra no *else*, armazenado no vetor de mensagens uma mensagem de sucesso e logo em seguida é verificado se o *issync* do cômodo criado é verdadeiro com *if-else*. Caso for, então inicia o *if* e sincroniza o comodo na *database SQLite*. Se for falso entra no *else* e retorna por *response* os vetores de erro e mensagens.

Dentro do *if* de sincronização, foi feito uma busca na tabela de usuário do *SQLite* se existe um usuário com o id do *mongoDB*, utilizando o método *get* da instância *db*, possibilitando assim vincular a chave primária do usuário com a chave estrangeira do cômodo na tabela de sincronização.

Os erros da busca são armazenados no parâmetro *err* e a coluna encontrada é armazenada no parâmetro *row*. Dentro da *function* do *get* é testado se *row* está vazio com o condicionamento *if-else*. Caso for, entra no *else*, armazena o erro no vetor *errors* e envia por *response* junto com o vetor de mensagens e o código de erro *HTTP* 400.

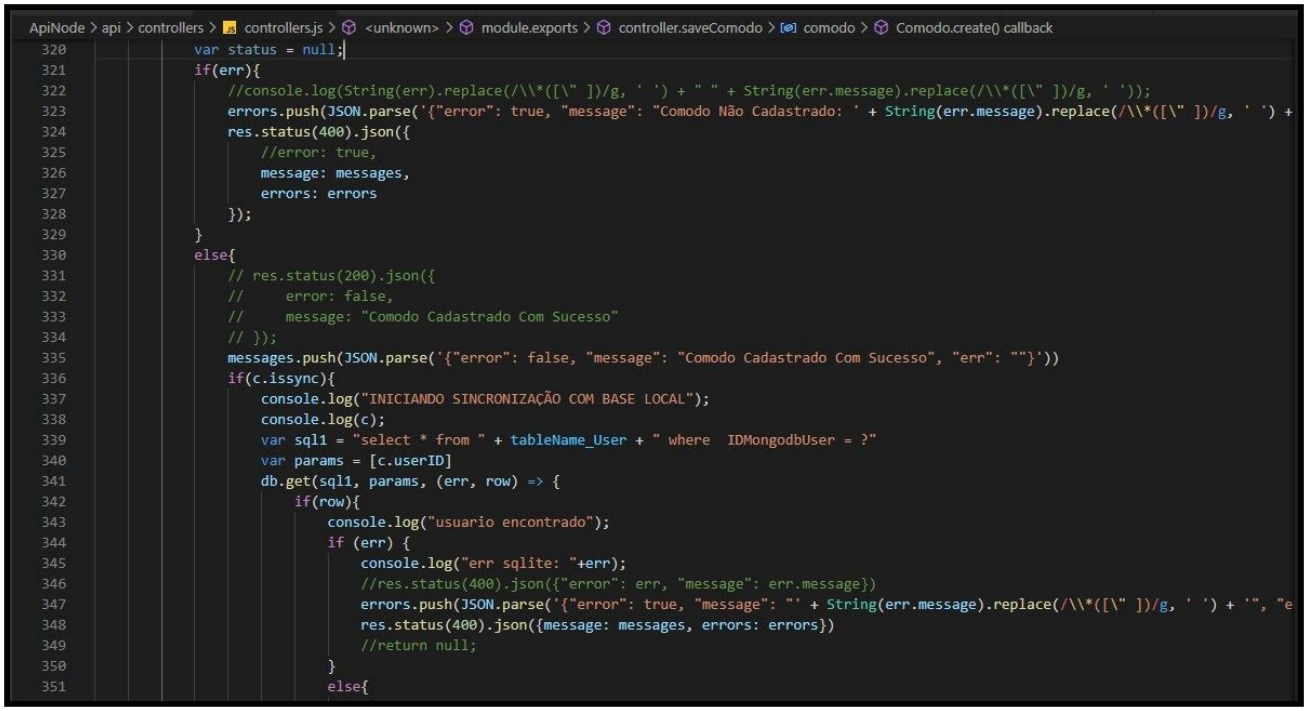
Se a *row* estiver populada com a coluna de usuário, então entra no *if* e trata os erros que sucederam ao executar a query de busca de usuário. Caso ocorrer algum erro, entra no *if(err)*, armazenando o erro no vetor de erros e retornando junto com o vetor de mensagens por meio de um *response*.

Caso o erro for falso, então entra no *else* para tratar o resto da sincronização. É verificado com uma *query* de contador se a coluna a ser criada já existe no *SQLite*, essa query retorna 1 ou mais se existir a coluna na *database* e 0 se não existir.Os erros e sucessos foram tratados como as demais execuções do *SQLite*. O valor do contador é armazenado no parâmetro *test*.

Caso não ocorrer nenhum erro, entra no *else* e verifica se o contador retornado é maior que 0. Caso for verdadeiro a condição, é armazenado no vetor de mensagem que o comodo já existe na tabela de sincronização e retornado como *response* junto com o vetor de erros.

Caso contrário entra no *else* e executa o método *run* com a query de *INSERT* junto com os valores da coluna vindos da variável c retornada pelo método *create* do *Mongoose* e o *id* do usuário do SQLite vindo da variável *row* retornada da pesquisa de usuário feita anteriormente. O tratamento de erros e sucessos serão feitos igual as execuções anteriores

Se não ocorrer nenhum erro, entra no *else*, armazena no vetor de mensagens que o cômodo foi sincronizado com sucesso e retorna esse vetor e o de erros em uma *response*. As figuras 25, 26, 27, 28 e 29 abaixo mostram as implementações explicadas anteriormente.



```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveComodo > comodo > Comodo.create() callback
320             var status = null;
321             if(err){
322                 //console.log(String(err).replace(/\\*([\" ])/g, ' ') + " " + String(err.message).replace(/\\*([\" ])/g, ' '));
323                 errors.push(JSON.parse('{"error": true, "message": "Comodo Não Cadastrado: ' + String(err.message).replace(/\\*([\" ])/g, ' ') +
324                 res.status(400).json({
325                     //error: true,
326                     message: messages,
327                     errors: errors
328                 });
329             }
330             else{
331                 // res.status(200).json({
332                 //     error: false,
333                 //     message: "Comodo Cadastrado Com Sucesso"
334                 // });
335                 messages.push(JSON.parse('{"error": false, "message": "Comodo Cadastrado Com Sucesso", "err": ""}'))
336                 if(c.issync){
337                     console.log("INICIANDO SINCRONIZAÇÃO COM BASE LOCAL");
338                     console.log(c);
339                     var sql1 = "select * from " + tableName_User + " where  IDMongodbUser = ?"
340                     var params = [c.userID]
341                     db.get(sql1, params, (err, row) => {
342                         if(row){
343                             console.log("usuario encontrado");
344                             if (err) {
345                                 console.log("err sqlite: "+err);
346                                 //res.status(400).json({"error": err, "message": err.message})
347                                 errors.push(JSON.parse('{"error": true, "message": "' + String(err.message).replace(/\\*([\" ])/g, ' ') + '", "e
348                                 res.status(400).json({message: messages, errors: errors})
349                                 //return null;
350                             }
351                             else{
```

Figura 25: Método Post cômodo MongoDB parte 2

```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveComodo > comodo > Comodo.create() callback > db.get() callback
351 else{
352     console.log(row)
353     //return row;
354     console.log(c._id)
355     var p = [c._id]
356     db.get("SELECT count(*) as t FROM " + tableName_Comodo + " WHERE IDMongodbComodo=?",p, (err,test) =>{
357         if(err){
358             console.log("deu erro: "+err);
359             //res.status(400).json({"error": err, "message": err.message})
360             errors.push(JSON.parse('{"error": true, "message": "' + String(err.message).replace(/\\*([\" ])/g, ' ') + '", "err": "' + String(err
361             res.status(400).json({message: messages, errors: errors})
362         }
```

Figura 26: Método Post cômodo MongoDB parte 3

```
363         else{
364             console.log(test.t);
365             var count = test.t;
366             if(count > 0){
367                 console.log("Coluna ja Existe na base do servidor, Sincronizado com sucesso")
368                 //res.json({"message": "Coluna ja Existe, Sincronizado com sucesso"})
369                 messages.push(JSON.parse('{"error": false, "message": "Coluna ja Existe na base do servidor, Sincronizado com sucesso", "err": "
370                 res.status(200).json({message: messages, errors: errors})
371             }
372             else{
373                 var sql ="INSERT INTO " + tableName_Comodo +
374                  " (IDMongodbComodo, IDU, IDMongodbUser, nomeComodo, tempComodo, umiComodoF, ajustTempComodo, statusLuzComodo, statusTomadaComod
375                 var params =[c._id, row.id, c.userID, c.nomeComodo, c.tempComodo,
376                     c.umiComodoF, c.ajustTempComodo, c.statusLuzComodo, c.statusTomadaComodo,
377                     c.statusJanelaComodo, c.statusPortaComodo, c.statusPresencaComodo, c.statusArCondiconado,
378                      c.isActive, c.isFavorite, c.isPareado, c.dispPareado, c.issync, c.date]
379                 db.run(sql, params, function (err, result) {
380                     if (err){
381                         console.log(err)
382                         //res.status(400).json({"error": err, "message": err.message})
```

Figura 27: Método Post cômodo MongoDB parte 4

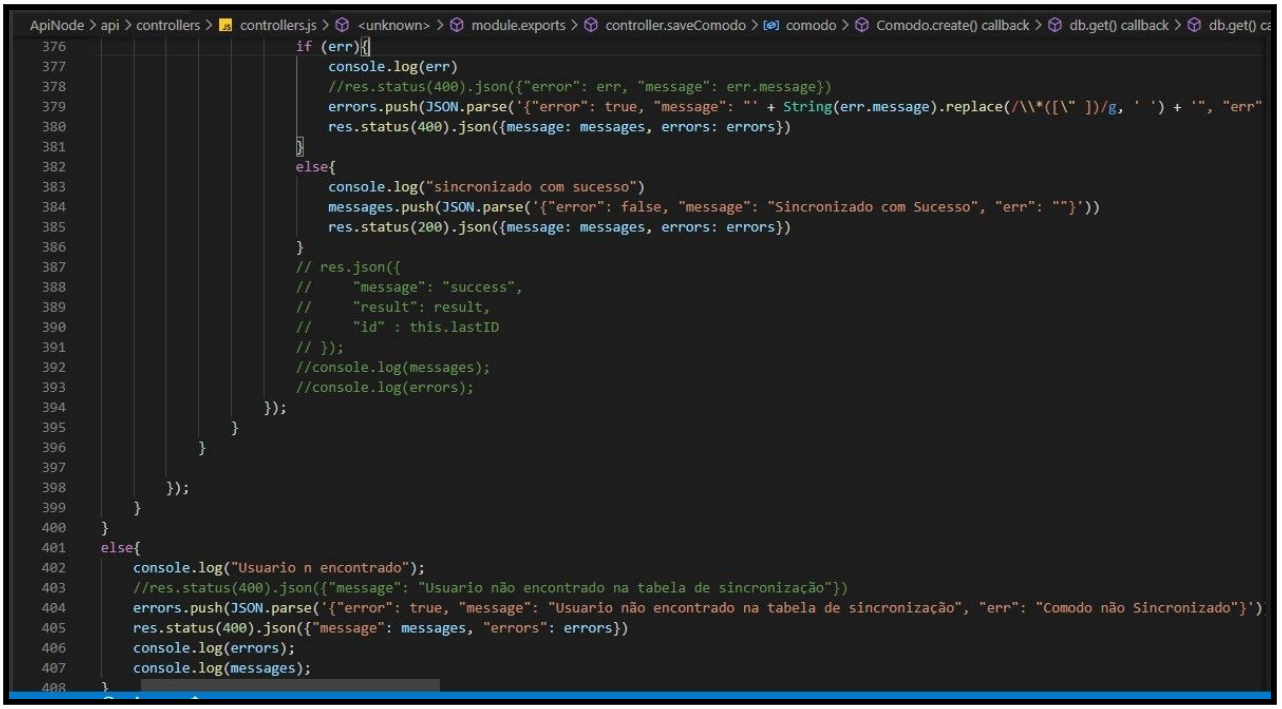


Figura 28: Método Post cômodo MongoDB parte 5

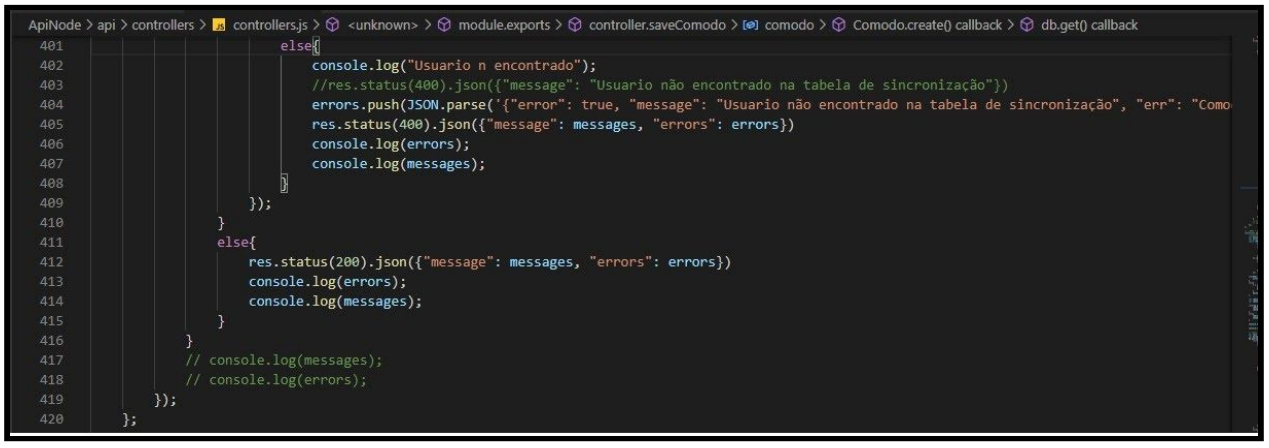


Figura 29: Método Post cômodo MongoDB parte 6

Para que o usuário seja cadastrado no *mongoDB* foi desenvolvido um método *HTTP POST* semelhante ao feito no cômodo, porém com algumas mudanças e remoções de chamadas. Foi criado um objeto de chave-valor que recebe os dados do usuário do *body* da requisição, caso os dados estejam vazios é tratado cada tipo especificamente.

Também foi usado a constante criado com base no *Schema* do usuário para executar o método *create* do *mongoose*, e como no cômodo, os erros e o *user* criado são passados por parâmetro respectivamente nas variáveis *err* e c. Também foi criado os dois vetores de mensagens e erros para poder armazenar os dados para o *response*.

O tratamento de erros foi feito usando o condicionamento *if-else*, como nas aplicações anteriores. Caso não ocorrer nenhum erro entra no *else* para tratar e sincronizar o usuário no *SQLite*. Também é armazenado uma mensagem de sucesso no vetor de mensagens.

Para verificar se o usuário pode ser sincronizado na hora de criar-lo no *mongoDB*, foi usado o *if-else* para determinar se o campo *issync* do usuário que foi retornado no create é verdadeiro ou falso. Caso for verdadeiro, entra no *if*, em caso negativo vai ser direcionado ao else que retorna os erros e mensagens como *response*.

Dentro do *if(c.issync)* é tratado a sincronização do usuário na *database SQLite*. É analisado se existe uma coluna com o mesmo *id* de documento do *MongoDB* que o usuário retornado no *create*, usando o *db.get*, que devolve como parâmetros nas variáveis *err* e *test* respectivamente os erros e o valor em números de tabelas encontradas.

Se der algum erro na pesquisa entra no *if*, tratando como as implementações anteriores. Caso contrário entra no else e executa a *query* de *insert*, realizando o mesmo tratamento de erros que a inserção do cômodo no SQLite. As figuras 30, 31, 32, 33 e 34 abaixo apresentam a implementação explicada anteriormente.

ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.updateBYIDComodoSQLite > data

```
872     // __POST + MONGODB
873     controller.saveUser = (req, res) => {
874         const usr = {
875             //userID: uuidv4() || "",
876             username: req.body.username || "",
877             nome: req.body.nome || "",
878             sobrenome: req.body.sobrenome || "",
879             email: req.body.email || "",
880             senha: req.body.senha || "",
881             celular: req.body.celular || "",
882             isActive: req.body.isActive || false,
883             dispositivo: req.body.dispositivo || "",
884             issync: req.body.issync || false,
885             date: new Date()
886         }
887         const user = User.create(usr, (err, c) =>{
888             errors = [];
889             messages = [];
890             console.log(c);
891             if(err){
892                 errors.push(JSON.parse('{"error": true, "message": "Usuario Não Cadastrado: ' + String(err.message).replace(/\\*([\" ])/g, ' ')
893                 res.status(400).json({
894                     messages: messages,
895                     errors: errors
896                 });
897             }
898             else{
899                 // res.status(200).json({
900                 //     error: false,
901                 //     message: "Usuario Cadastrado Com Sucesso"
902                 // });
903                 messages.push(JSON.parse('{"error": false, "message": "Usuario Cadastrado Com Sucesso", "err": ""}'))
```

Figura 30: Método Post usuario MongoDB parte 1

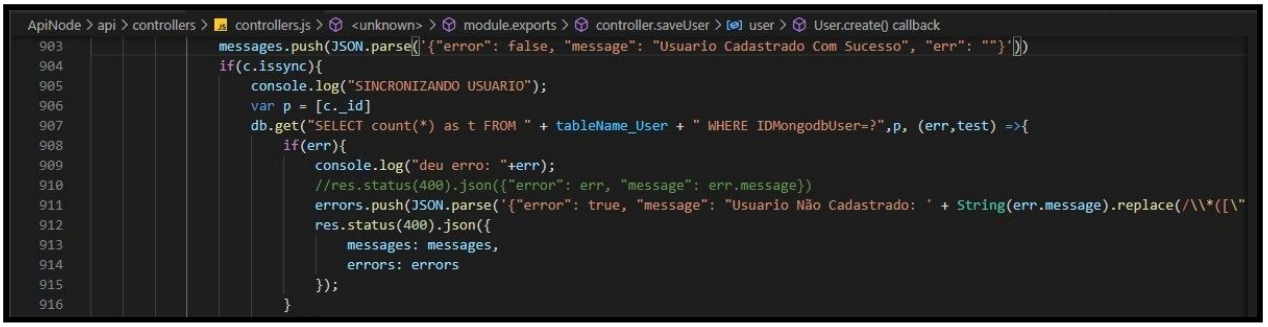


Figura 31: Método Post usuario MongoDB parte 2

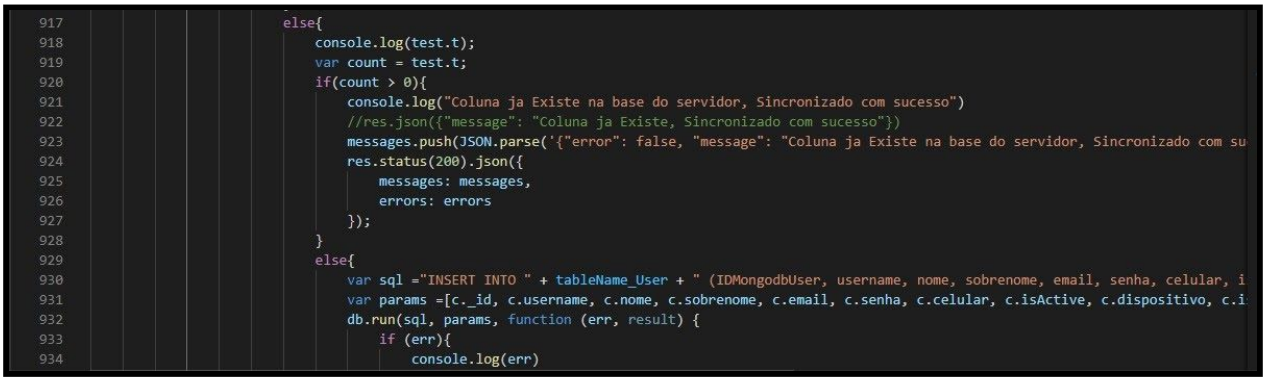


Figura 32: Método Post usuario MongoDB parte 3

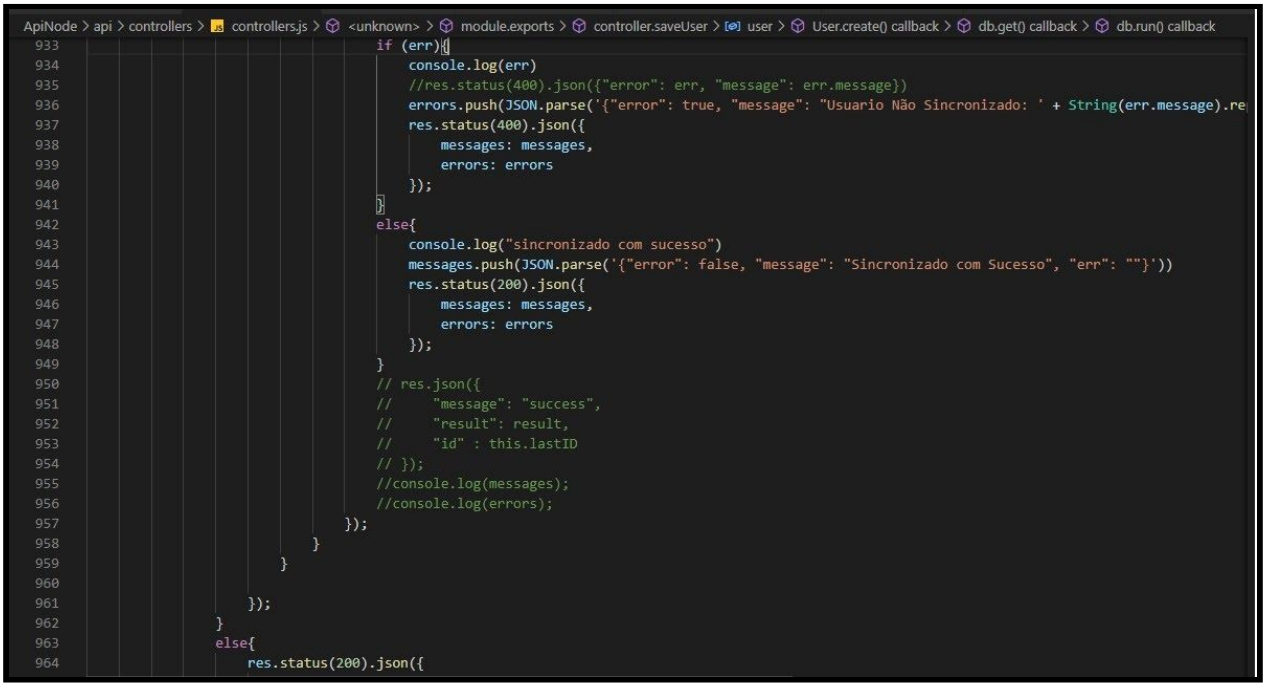


Figura 33: Método Post usuario MongoDB parte 4

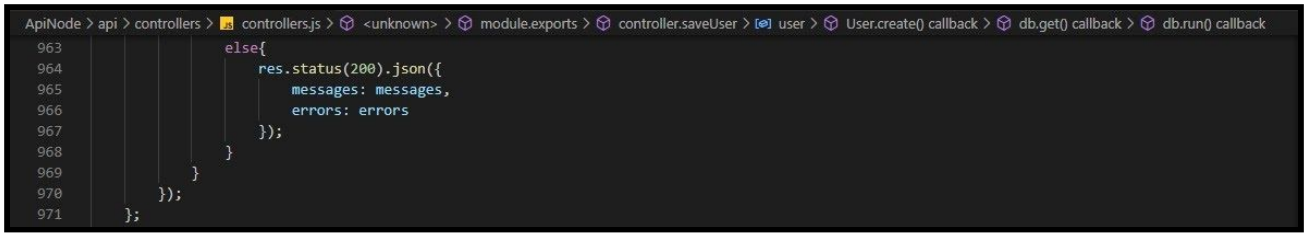


Figura 34: Método Post usuario MongoDB parte 5

Para desenvolver o método *POST* do cômodo no SQLite, foi criado dois vetores que receberão os erros(*errors*) e mensagens(*messages*) emitidos durante o decorrer do processo. Também foi criado uma constante (*IDMgC*) que recebe os valores do cômodo no *MongoDB* de acordo com o id passado no *body* da requisição se nenhum erro acontecer.

Logo em seguida é verificado a constante *IDMgc*, certificando se o campo *error* é verdadeiro ou falso. Caso for verdadeiro entra dentro de um *if* e adicionando uma mensagem de erro no vetor de erros. Se for falso, entra no *else* e adicionar um objeto de chave-valor no vetor de mensagem, contendo a *message* e o *error* retornada em *IDMgC*.

Também foram criados duas variáveis denominadas *idUserMg* e *idComMg* que são usados como parâmetros para filtros de buscas. Para poder popular essas variáveis, foi usado os condicionamentos *if-else*.

Se o campo IDMongodbUser vindo do *body* for vazio e os campos *error* e *comd* de *IDMgC* forem respectivamente false e diferente de vazio, então entra no *if* e preenche as variáveis filtros com o *id* do cômodo e do usuário contidas dentro de *IDMgC*, caso contrário, os filtros são preenchidos com os dados dos *ids* enviados pelo *body*.

Também foi desenvolvido a implementação que possibilita a criação do cômodo no *SQLite*, verificando se existe um usuário cadastrado dentro da tabela de usuário com a variável filtro *idUserMg* nos parâmetros (*params*), passando os erros na variável *err* e a coluna encontrada na variável *row*.

Se a variável *row* estiver vazia, entra no *else*, armazena um erro no vetor de erros e envia o mesmo junto com o vetor de mensagens para o usuário com o código de erro 400. Caso contrário entra no *if* para tratar os erros da execução e realizar a criação do cômodo.

Dentro do *if(row)* será verificado se a variável *err* está vazia com outro condicionamento *if-else*. Caso a condição for positivo, então entra no *if*, armazena o erro no vetor de erros e envia o mesmo junto com o vetor de mensagens, esse tratamento de erros será o mesmo em todo o processo.

Caso o *err* for vazio, então entra no else e inicia uma pesquisa dentro da tabela de cômodos usando a variável *idComMg* como parâmetro de pesquisa. Essa busca retorna um erro se existir e um contador de quantas tabelas foram encontradas, que são armazenados nos respectivos parâmetros *err* e *test*.

Se o erro for verdadeiro, realiza o tratamento de erros padrão. Caso contrário entra no *else* e verifica se existe alguma tabela com o condicionamento *if-else*. Se for maior que zero o contador, então é armazenado no vetor de mensagens que já está sincronizado e retorna o mesmo com o vetor de erros para o usuário.

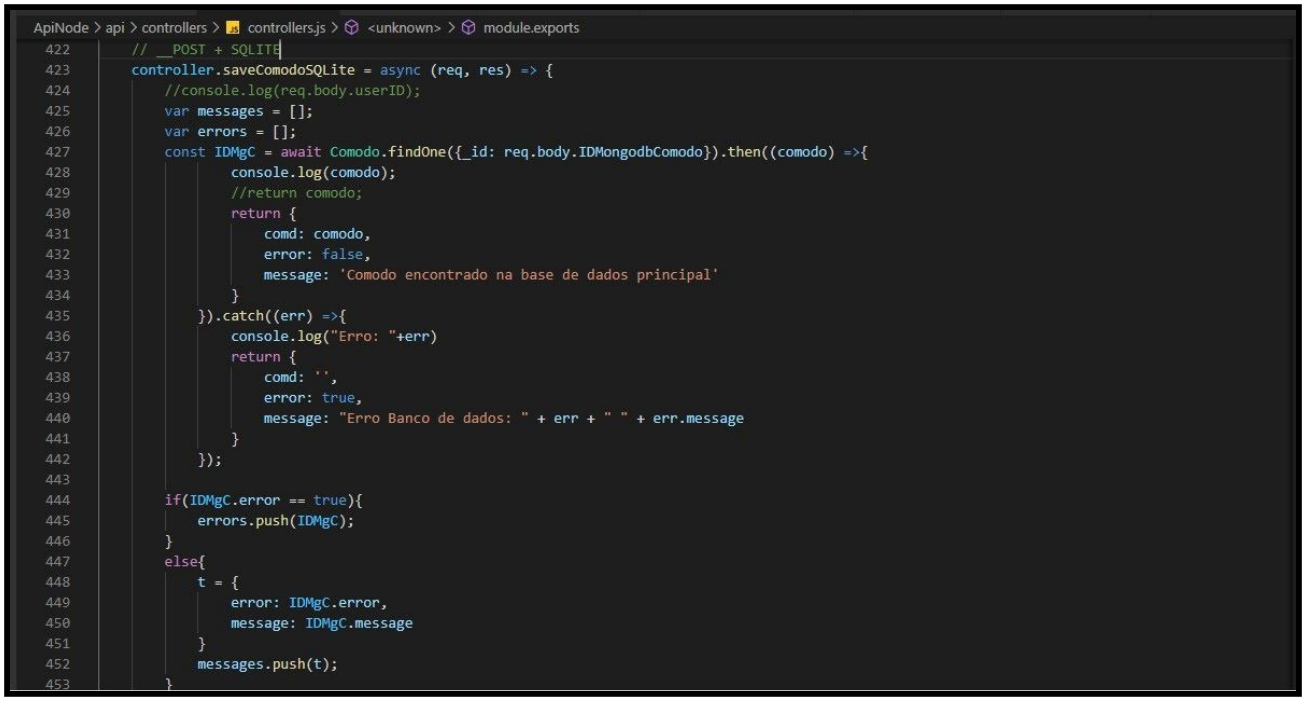
Caso contrário entra no else e chama a função *insert* passando como parâmetro a variável *row*. Dentro dela foi criado um objeto chave-valor que recebe os dados do *body* como valor para cada respectiva coluna da tabela que serão as chaves.

Se os dados forem vazios, é passado as informações do cômodo contidas em *IDMgC*, caso este também esteja vazio, é colocado como vazio para strings e falso para booleanas. Também foi passado o *id* do usuário da variável *row* no campo de chave estrangeira do novo cômodo.

Também foi verificado se a sincronização de dados está ativa, caso não estiver, entra em um *else*, armazena um mensagem que não está ativo a sincronização no vetor de mensagens e envia o mesmo junto com o vetor de erros para o usuário.

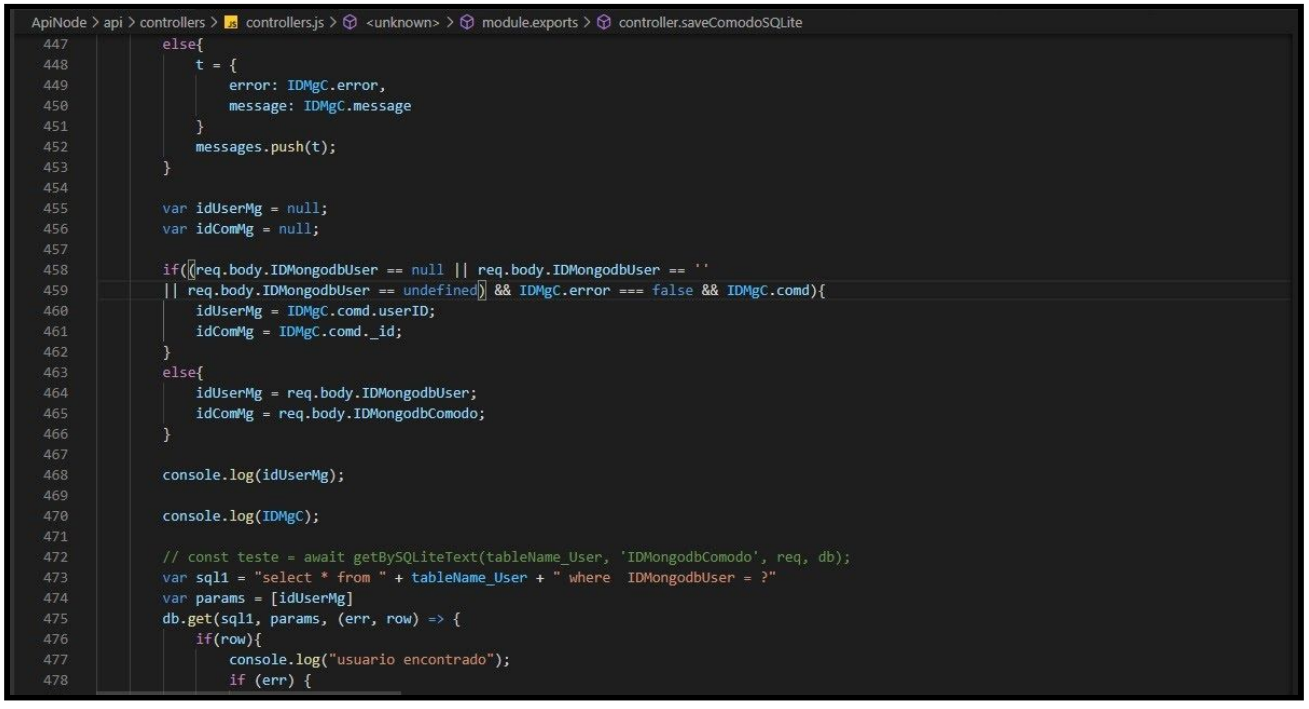
Caso estiver ativa a sincronização, entra no *if* e executa a *query* de inserção, passando como parâmetros os itens do objeto criado anteriormente, retornando um erro se existir e um resultado, que são armazenados nas variáveis *err* e *result*. O erro é tratado de forma padrão e caso não ocorra, entra no else, armazena uma mensagem de sucesso no vetor de mensagens e enviar junto com o vetor de erros para o usuário.

As figuras 35, 36, 37, 38, 39 e 40 abaixo são as implementações das explicações anteriores.



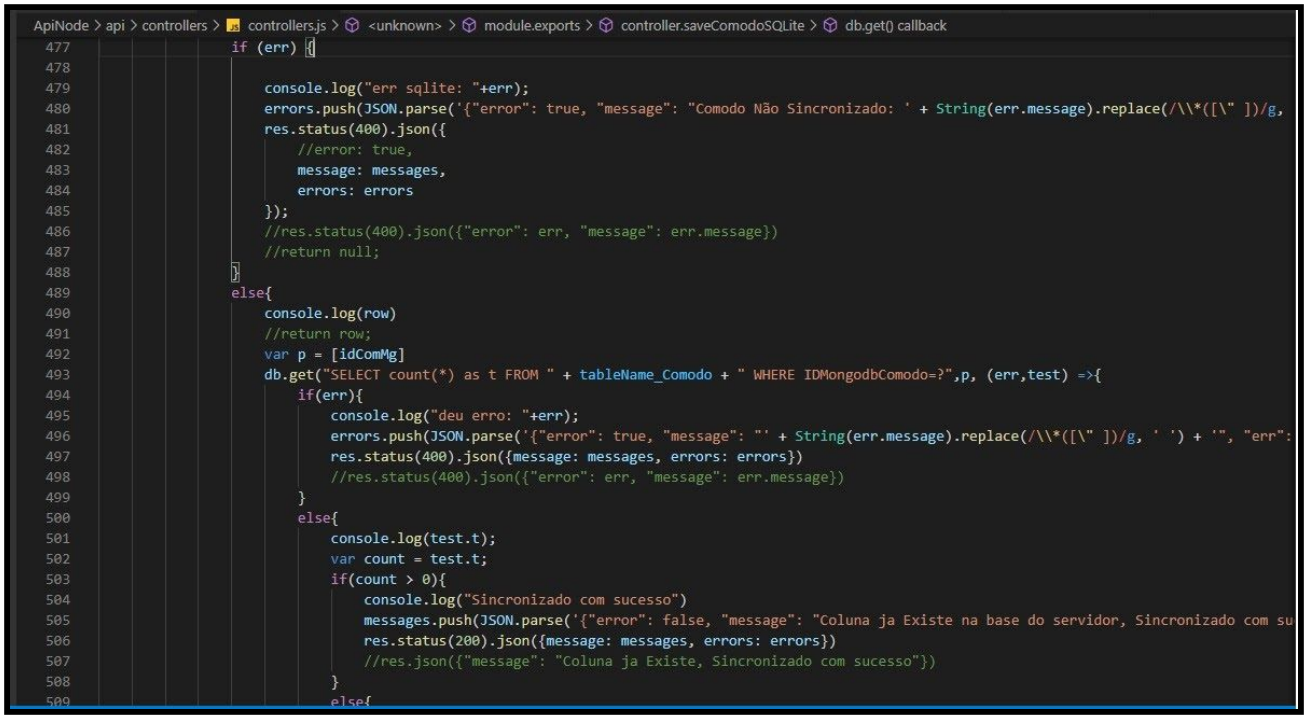
```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports
422     // __POST + SQLITE
423     controller.saveComodoSQLite = async (req, res) => {
424         //console.log(req.body.userID);
425         var messages = [];
426         var errors = [];
427         const IDMgC = await Comodo.findOne({_id: req.body.IDMongodbComodo}).then((comodo) =>{
428                 console.log(comodo);
429                 //return comodo;
430                 return {
431                     comd: comodo,
432                     error: false,
433                     message: 'Comodo encontrado na base de dados principal'
434                 }
435             }).catch((err) =>{
436                 console.log("Erro: "+err)
437                 return {
438                     comd: '',
439                     error: true,
440                     message: "Erro Banco de dados: " + err + " " + err.message
441                 }
442             });
443
444         if(IDMgC.error == true){
445             errors.push(IDMgC);
446         }
447         else{
448             t = {
449                 error: IDMgC.error,
450                 message: IDMgC.message
451             }
452             messages.push(t);
453         }
```

Figura 35: Implementação do método POST no cômodo no SQLite parte 1



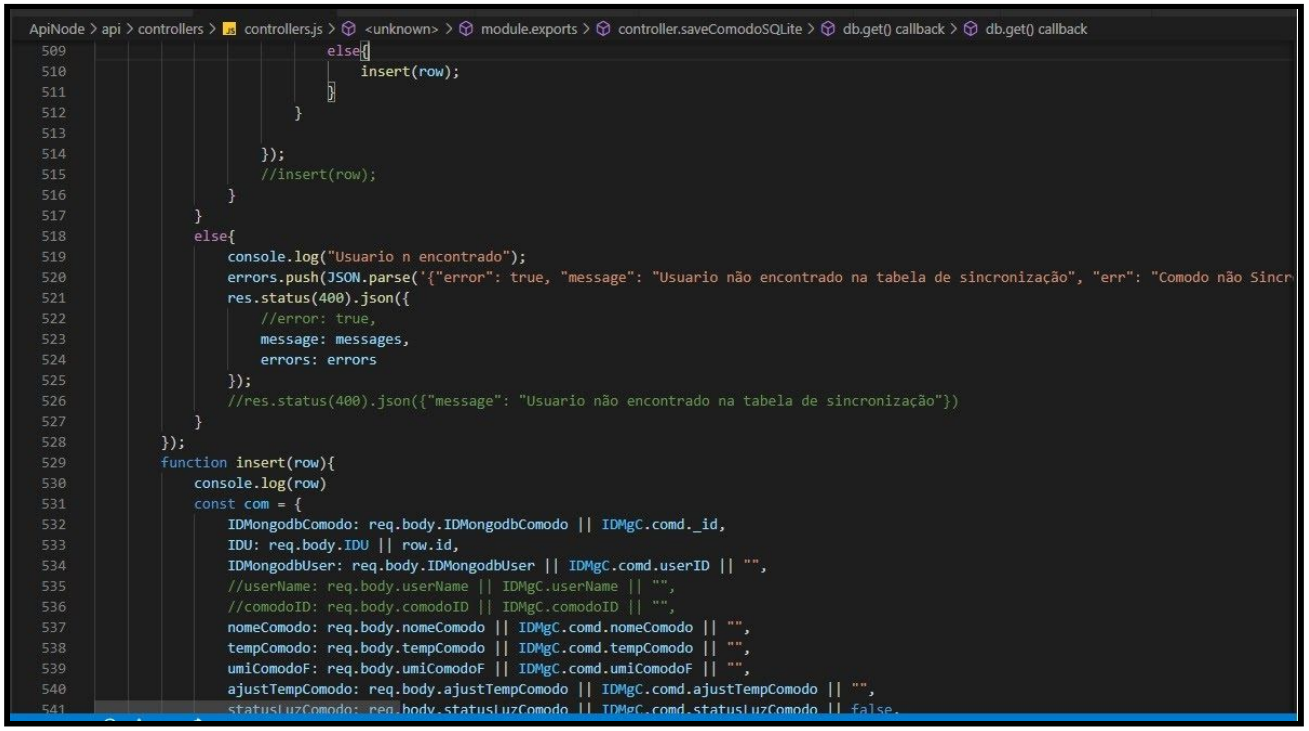
```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveComodoSQLite
447         else{
448             t = {
449                 error: IDMgC.error,
450                 message: IDMgC.message
451             }
452             messages.push(t);
453         }
454
455         var idUserMg = null;
456         var idComMg = null;
457
458         if((req.body.IDMongodbUser == null || req.body.IDMongodbUser == ''
459         || req.body.IDMongodbUser == undefined) && IDMgC.error === false && IDMgC.comd){
460             idUserMg = IDMgC.comd.userID;
461             idComMg = IDMgC.comd._id;
462         }
463         else{
464             idUserMg = req.body.IDMongodbUser;
465             idComMg = req.body.IDMongodbComodo;
466         }
467
468         console.log(idUserMg);
469
470         console.log(IDMgC);
471
472         // const teste = await getBySQLiteText(tableName_User, 'IDMongodbComodo', req, db);
473         var sql1 = "select * from " + tableName_User + " where  IDMongodbUser = ?"
474         var params = [idUserMg]
475         db.get(sql1, params, (err, row) => {
476             if(row){
477                 console.log("usuario encontrado");
478                 if (err) {
```

Figura 36: Implementação do método POST no cômodo no SQLite parte 2



```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveComodoSQLite > db.get() callback
477                 if (err) {
478
479                     console.log("err sqlite: "+err);
480                     errors.push(JSON.parse('{"error": true, "message": "Comodo Não Sincronizado: ' + String(err.message).replace(/\\*([\" ])/g,
481                     res.status(400).json({
482                         //error: true,
483                         message: messages,
484                         errors: errors
485                     });
486                     //res.status(400).json({"error": err, "message": err.message})
487                     //return null;
488                 }
489                 else{
490                     console.log(row)
491                     //return row;
492                     var p = [idComMg]
493                     db.get("SELECT count(*) as t FROM " + tableName_Comodo + " WHERE IDMongodbComodo=?",p, (err,test) =>{
494                         if(err){
495                             console.log("deu erro: "+err);
496                             errors.push(JSON.parse('{"error": true, "message": "' + String(err.message).replace(/\\*([\" ])/g, ' ') + '", "err":
497                             res.status(400).json({message: messages, errors: errors})
498                             //res.status(400).json({"error": err, "message": err.message})
499                         }
500                         else{
501                             console.log(test.t);
502                             var count = test.t;
503                             if(count > 0){
504                                 console.log("Sincronizado com sucesso")
505                                 messages.push(JSON.parse('{"error": false, "message": "Coluna ja Existe na base do servidor, Sincronizado com su
506                                 res.status(200).json({message: messages, errors: errors})
507                                 //res.json({"message": "Coluna ja Existe, Sincronizado com sucesso"})
508                             }
509                             else{
```

Figura 37: Implementação do método POST no cômodo no SQLite parte 3



```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveComodoSQLite > db.get() callback > db.get() callback
509                             else{
510                                 insert(row);
511                             }
512                         }
513
514                     });
515                     //insert(row);
516                 }
517             }
518             else{
519                 console.log("Usuario n encontrado");
520                 errors.push(JSON.parse('{"error": true, "message": "Usuario não encontrado na tabela de sincronização", "err": "Comodo não Sincr
521                 res.status(400).json({
522                     //error: true,
523                     message: messages,
524                     errors: errors
525                 });
526                 //res.status(400).json({"message": "Usuario não encontrado na tabela de sincronização"})
527             }
528         });
529         function insert(row){
530             console.log(row)
531             const com = {
532                 IDMongodbComodo: req.body.IDMongodbComodo || IDMgC.comd._id,
533                 IDU: req.body.IDU || row.id,
534                 IDMongodbUser: req.body.IDMongodbUser || IDMgC.comd.userID || "",
535                 //userName: req.body.userName || IDMgC.userName || "",
536                 //comodoID: req.body.comodoID || IDMgC.comodoID || "",
537                 nomeComodo: req.body.nomeComodo || IDMgC.comd.nomeComodo || "",
538                 tempComodo: req.body.tempComodo || IDMgC.comd.tempComodo || "",
539                 umiComodoF: req.body.umiComodoF || IDMgC.comd.umiComodoF || "",
540                 ajustTempComodo: req.body.ajustTempComodo || IDMgC.comd.ajustTempComodo || "",
541                 statusLuzComodo: req.body.statusLuzComodo || IDMgC.comd.statusLuzComodo || false,
```

Figura 38: Implementação do método POST no cômodo no SQLite parte 4

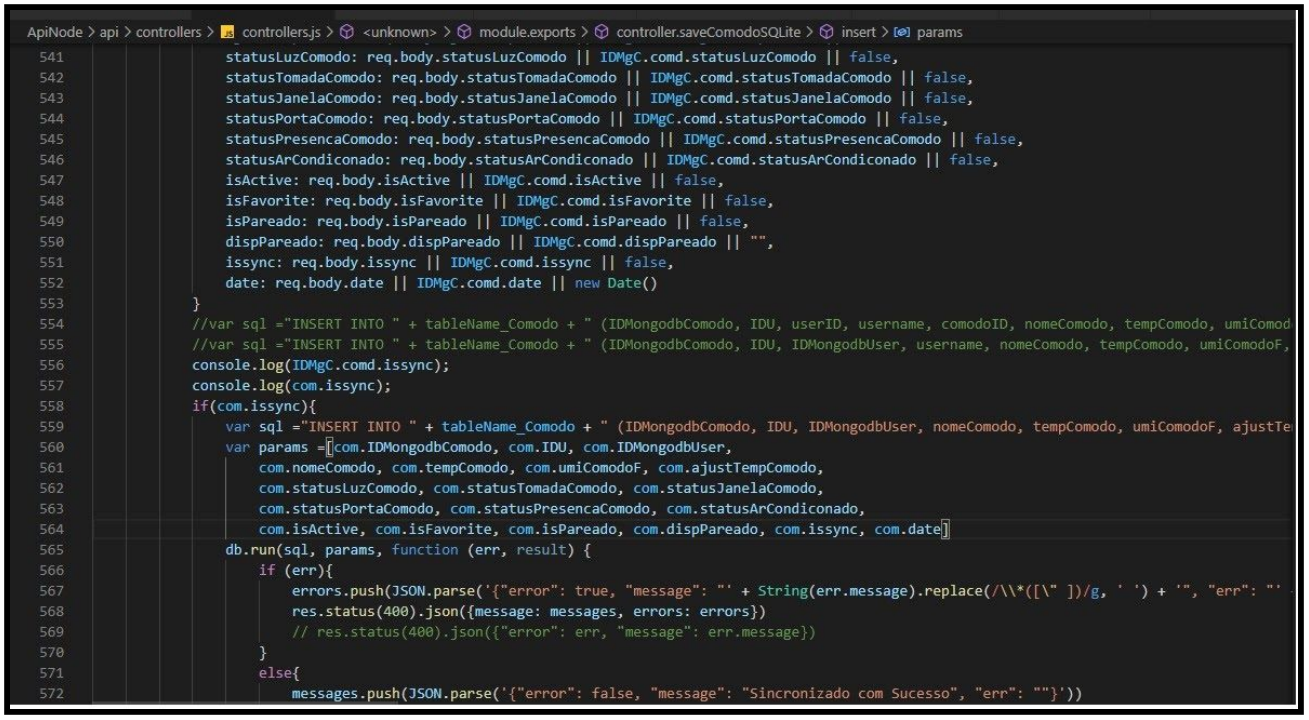


Figura 39: Implementação do método POST no cômodo no SQLite parte 5

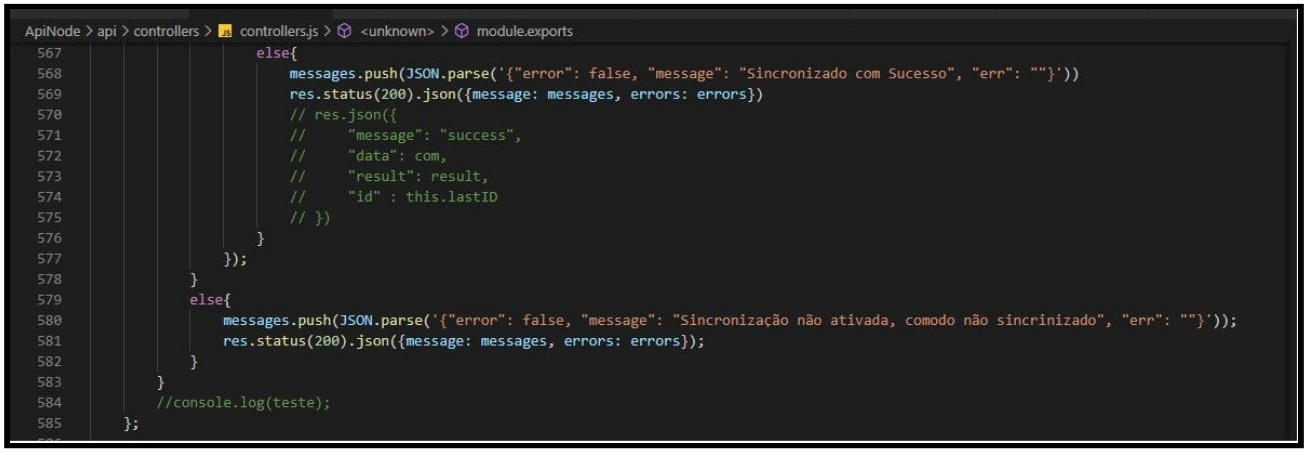


Figura 40: Implementação do método POST no cômodo no SQLite parte 6

Para que a API possa adicionar um novo usuário na *database SQLite*, foi utilizado quase os mesmos meios que a implementação anterior do cômodo, possuindo algumas alterações nos processos.

A parte inicial foi semelhante ao da criação do cômodo no *SQLite*, onde foi criado as variáveis de mensagens e erros, a constante que recebe um objeto com o usuário do *MongoDB* e a verificação para poder popular a variável filtro.

Já que não será utilizado o código do usuário para preencher a chave estrangeira, então foi iniciado o processo de criação pela pesquisa para verificar se é encontrado um usuário

com o mesmo *id* do *MongoDB* que está na variável *idUserMg* passada como parâmetro de busca. Essa pesquisa retorna um erro se existir e retorna um valor de colunas encontradas.

Se o erro for verdadeiro, então é tratado como a implementação anterior, caso contrário entra no else, Dentro é verificado a quantidade de colunas retornadas pela execução da *query*, caso for maior que zero, então entra no *if,* armazena no vetor de mensagens que já existe uma coluna com o mesmo *id* e envia o mesmo junto com o vetor de erros para o usuário por *response*.

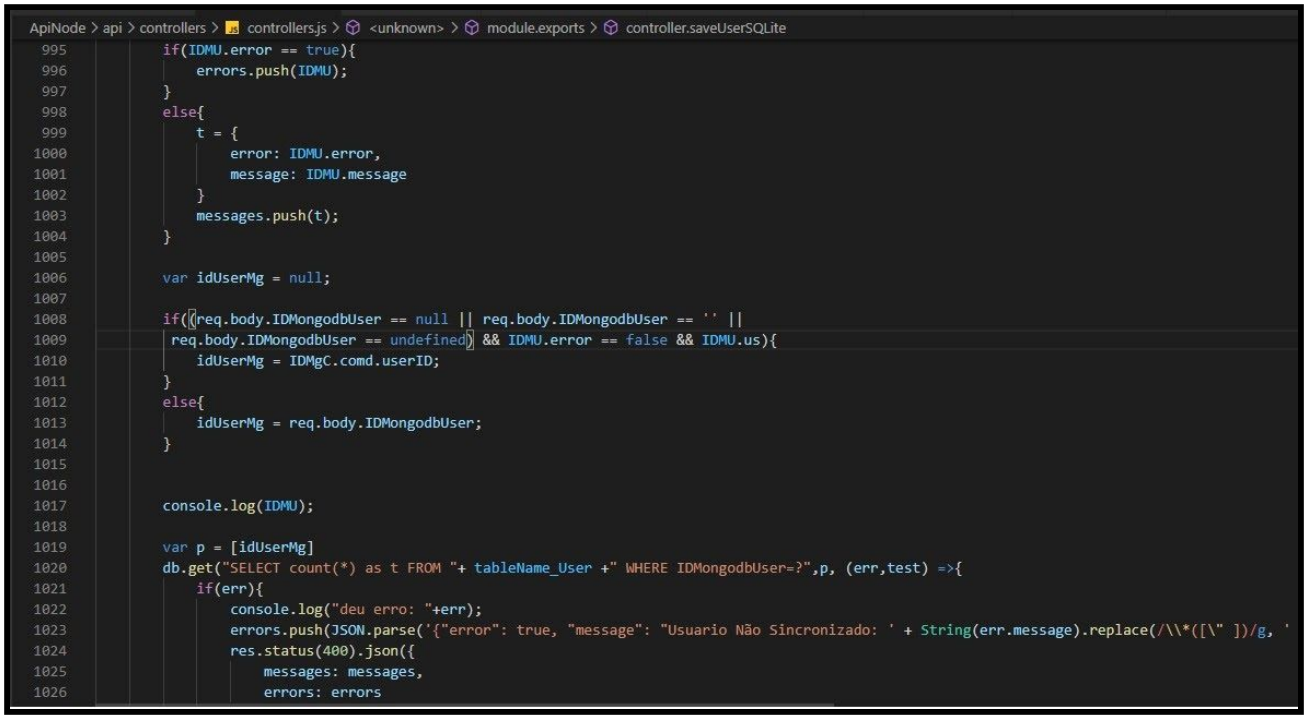
Caso contrário, entra no *else* e chama a função *insert*, que possui a definição de um objeto chave-valor, contendo os campos que serão inseridos na tabela como chave e os valores do body como valor caso não estiver vazio.

Se estiverem vazios os campos do body, são inseridos os valores contidos na constante *IDMU*, caso esses valores também estiverem vazios, são colocados como “” os campos *String* e *false* os campos booleanos.

Foi feito da mesma maneira que a inserção anterior do cômodo, verificando se está ativo a sincronização, tratando se não estiver e caso estiver realiza a inserção, tratando os erro e sucessos. As figuras 41, 42, 43, 44 e 45 abaixo são a representação da implementação explicada anteriormente.

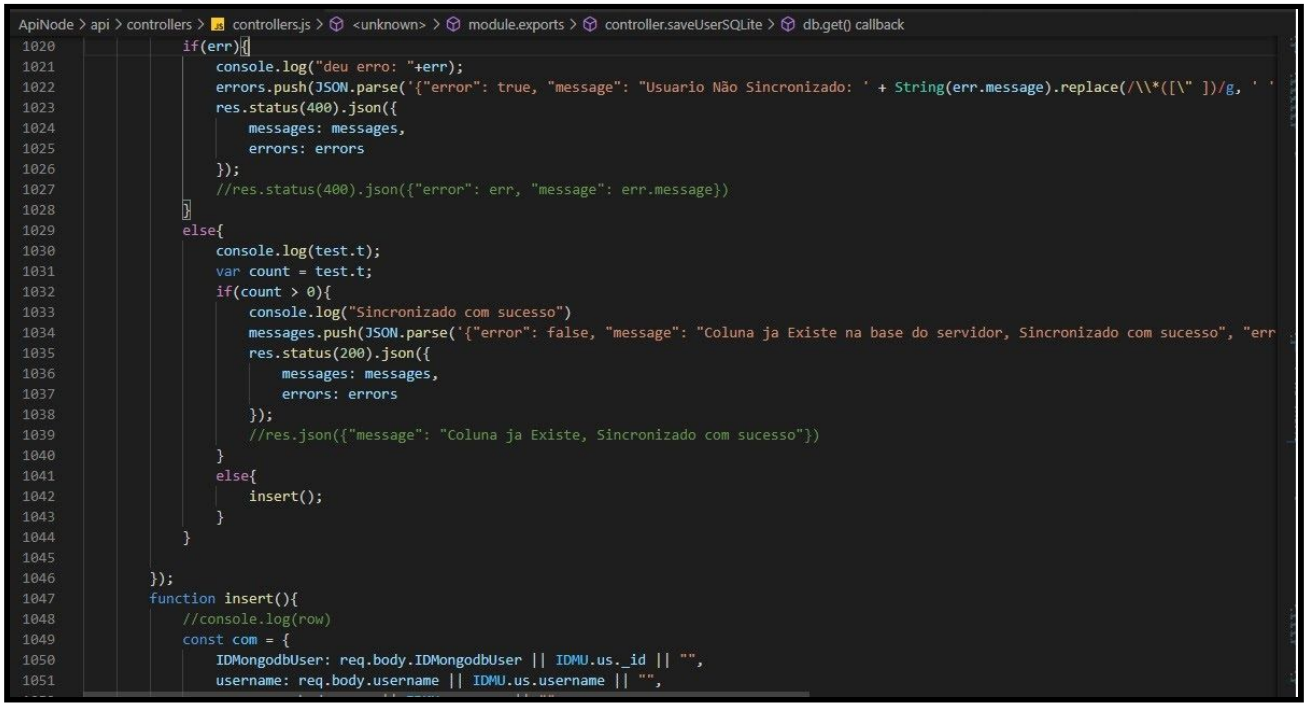
```
973     // __POST + SQLITE
974     controller.saveUserSQLite = async (req, res) => {
975         //console.log(req.body.userID);
976         var messages = [];
977         var errors = [];
978         const IDMU = await User.findOne({_id: req.body.IDMongodbUser}).then((usr) =>{
979             console.log(usr);
980             //return usr;
981             return {
982                 us: usr,
983                 error: false,
984                 message: 'Usuario encontrado para sicronizar'
985             }
986         }).catch((err) =>{
987             console.log("Erro: "+err)
988             return {
989                 us: '',
990                 error: true,
991                 message: "Erro Banco de dados: " + err + " " + err.message
992             }
993         });
```

Figura 41: Método POST com SQLite na tabela usuario parte 1



```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveUserSQLite
995         if(IDMU.error == true){
996             errors.push(IDMU);
997         }
998         else{
999             t = {
1000                error: IDMU.error,
1001                message: IDMU.message
1002            }
1003            messages.push(t);
1004        }
1005
1006        var idUserMg = null;
1007
1008        if((req.body.IDMongodbUser == null || req.body.IDMongodbUser == '' ||
1009         req.body.IDMongodbUser == undefined) && IDMU.error == false && IDMU.us){
1010            idUserMg = IDMgC.comd.userID;
1011        }
1012        else{
1013            idUserMg = req.body.IDMongodbUser;
1014        }
1015
1016
1017        console.log(IDMU);
1018
1019        var p = [idUserMg]
1020        db.get("SELECT count(*) as t FROM "+ tableName_User +" WHERE IDMongodbUser=?",p, (err,test) =>{
1021            if(err){
1022                console.log("deu erro: "+err);
1023                errors.push(JSON.parse('{"error": true, "message": "Usuario Não Sincronizado: ' + String(err.message).replace(/\\*([\" ])/g, '
1024                res.status(400).json({
1025                    messages: messages,
1026                    errors: errors
```

Figura 42: Método POST com SQLite na tabela usuario parte 2



```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.saveUserSQLite > db.get() callback
1020            if(err){
1021                console.log("deu erro: "+err);
1022                errors.push(JSON.parse('{"error": true, "message": "Usuario Não Sincronizado: ' + String(err.message).replace(/\\*([\" ])/g, ' '
1023                res.status(400).json({
1024                    messages: messages,
1025                    errors: errors
1026                });
1027                //res.status(400).json({"error": err, "message": err.message})
1028            }
1029            else{
1030                console.log(test.t);
1031                var count = test.t;
1032                if(count > 0){
1033                    console.log("Sincronizado com sucesso")
1034                    messages.push(JSON.parse('{"error": false, "message": "Coluna ja Existe na base do servidor, Sincronizado com sucesso", "err
1035                    res.status(200).json({
1036                        messages: messages,
1037                        errors: errors
1038                    });
1039                    //res.json({"message": "Coluna ja Existe, Sincronizado com sucesso"})
1040                }
1041                else{
1042                    insert();
1043                }
1044            }
1045
1046        });
1047        function insert(){
1048            //console.log(row)
1049            const com = {
1050                IDMongodbUser: req.body.IDMongodbUser || IDMU.us._id || "",
1051                username: req.body.username || IDMU.us.username || "",
```

Figura 43: Método POST com SQLite na tabela usuario parte 3

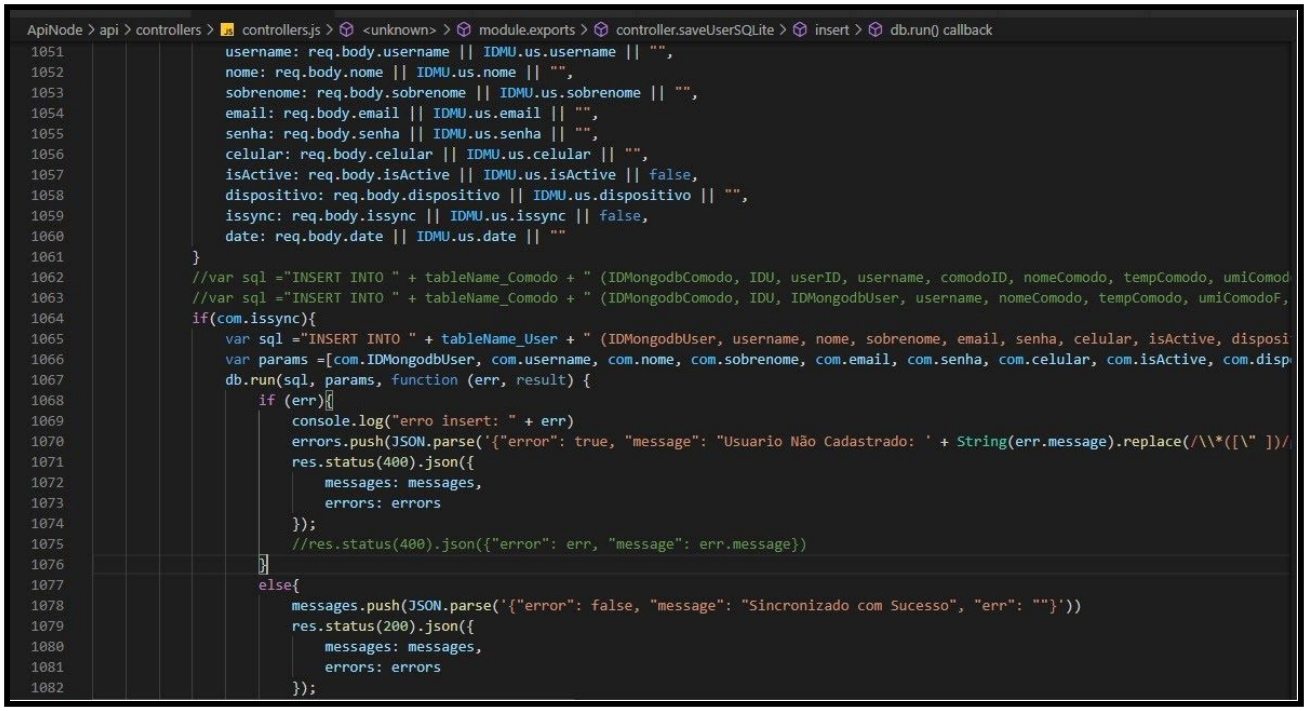


Figura 44: Método POST com SQLite na tabela usuario parte 4

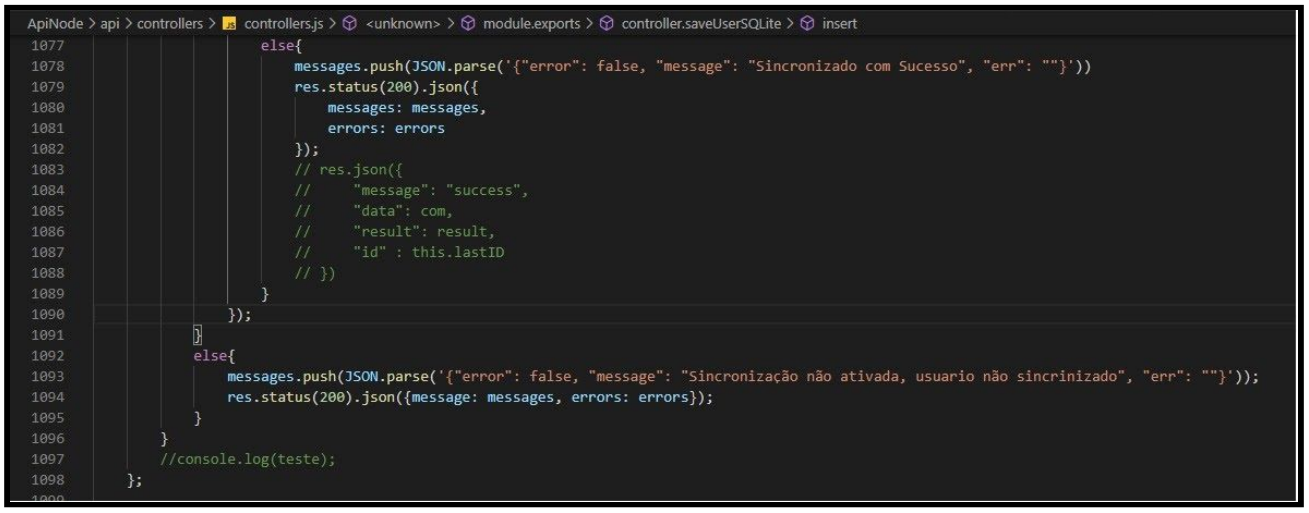


Figura 45: Método POST com SQLite na tabela usuario parte 5

Para que a API possibilite a alteração dos dados dos cômodos e dos usuários tanto no MongoDB, quanto no SQLite, foi utilizado o método *HTTP PUT*. Geralmente é utilizado para substituir as informações antigas de um determinado banco de dados pelas novas que vieram do *body* da requisição.

Para atualizar no *MongoDB* foi usado as constantes criadas no início da *API* que contém os *models* baseados nos *Schemas* do usuário e do cômodo. Por meio delas foi chamado o método da biblioteca *Mongoose updateOne* que atualiza o documento, com um filtro

baseado em um objeto chave-valor, contendo os campos que desejam ser usados, nesse caso é o *id* de documento do *MongoDB.*

Também foi passado os itens que vão ser atualizados, que no caso são os valores contidos no *body* da requisição. Esse método retorna um erro se existir, e um status da atualização, armazenado-os respectivamente no parâmetro erro e t.

Para poder ser tratado o erro, foi usado o condicionamento *if-else*. Se o erro for verdadeiro, entra no *if* e o retorna para o usuário por meio de uma *response*. Caso contrário, entra no *else* e retorna a mensagem de sucesso por meio de *response*.

As figuras 46 e 47 abaixo representam a implementação explicada acima para o usuário e para o cômodo no MongoDB.

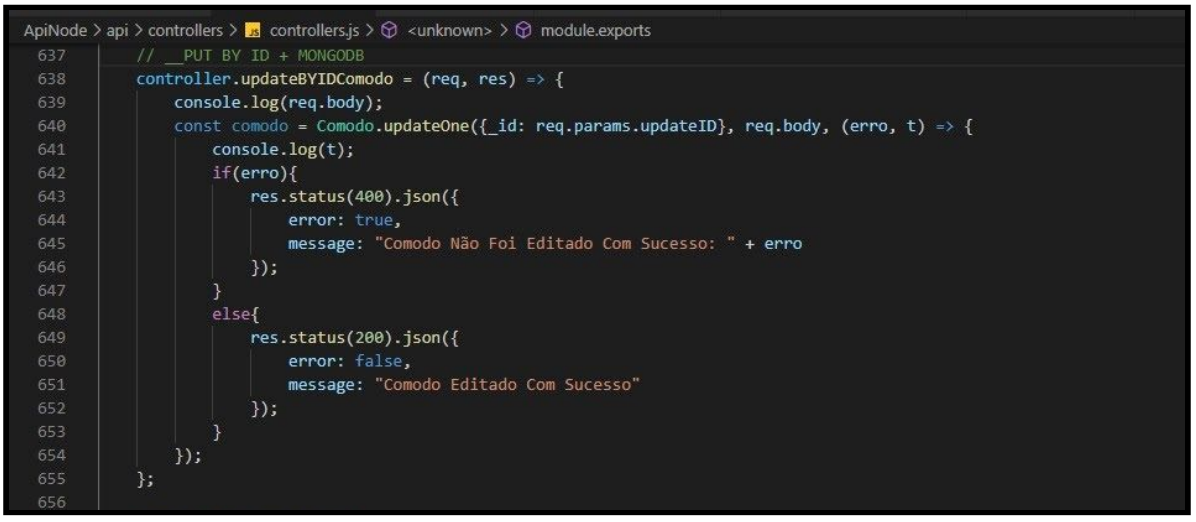


Figura 46: Método PUT com MongoDB no cômodo

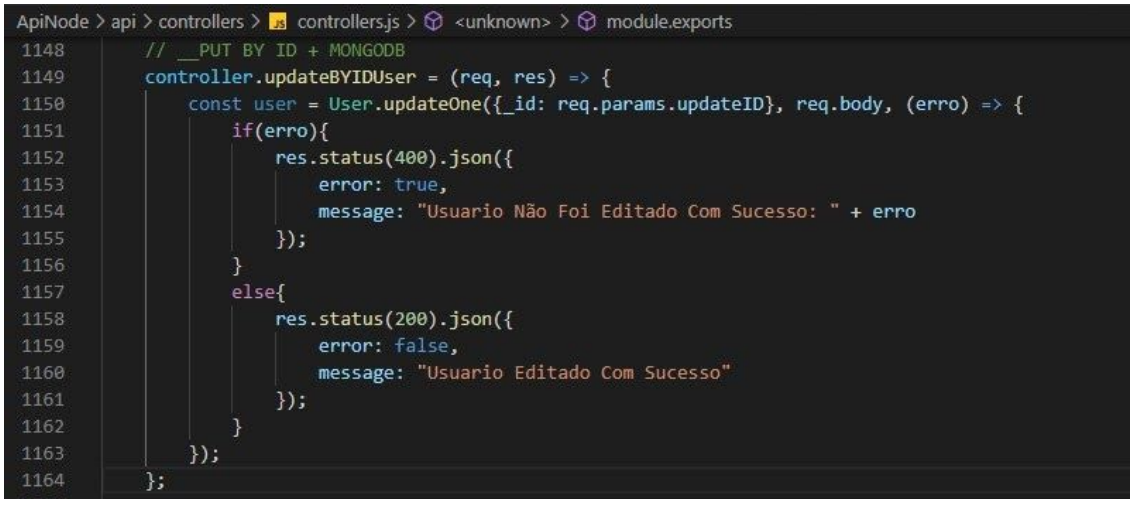


Figura 47: Método PUT com MongoDB no usuário

Para se implementar o *PUT* utilizando o banco de dados *SQLite*, foi criado um objeto chave-valor contendo os campos a serem alterados que serão as chaves e os dados vindos do *body* da requisição que serão os valores.

Então é usado a *query* de *UPDATE* para poder atualizar os dados com a cláusula *WHERE* para filtrar as colunas. Com o intuito de preencher os dados foi usando o *COALESCE* para verificar se o valor passado é vazio, se for, então mantém o valor atual que está na coluna, caso contrário é alterado o valor do campo.

Os dados do objeto criado anteriormente são passados como parâmetros para a alteração da coluna. O campo usado para filtrar foi passado no *WHERE*, retornando um erro se existir e um resultado de status, que serão armazenados nas variáveis *err* e *result*.

Para tratar os erros foi usado o mesmo método das outras implementações anteriores de testar se é falso e retornar para o usuário como *response*, e caso não der nenhum problema entra no *else* e envia a mensagem de sucesso para o usuário

As figuras 48, 49, 50, 51 e 52 abaixo representam as implementações explicadas anteriormente para a tabela cômodo e usuário.

```
656
657     // __PUT BY ID + SQLITE
658     controller.updateBYIDComodoSQLite = (req, res) => {
659         var data = {
660                 nomeComodo: req.body.nomeComodo,
661                 tempComodo: req.body.tempComodo,
662                 umiComodoF: req.body.umiComodoF,
663                 ajustTempComodo: req.body.ajustTempComodo,
664                 statusLuzComodo: req.body.statusLuzComodo,
665                 statusTomadaComodo: req.body.statusTomadaComodo,
666                 statusJanelaComodo: req.body.statusJanelaComodo,
667                 statusPortaComodo: req.body.statusPortaComodo,
668                 statusPresencaComodo: req.body.statusPresencaComodo,
```

Figura 48: Método PUT com SQLite no cômodo parte 1

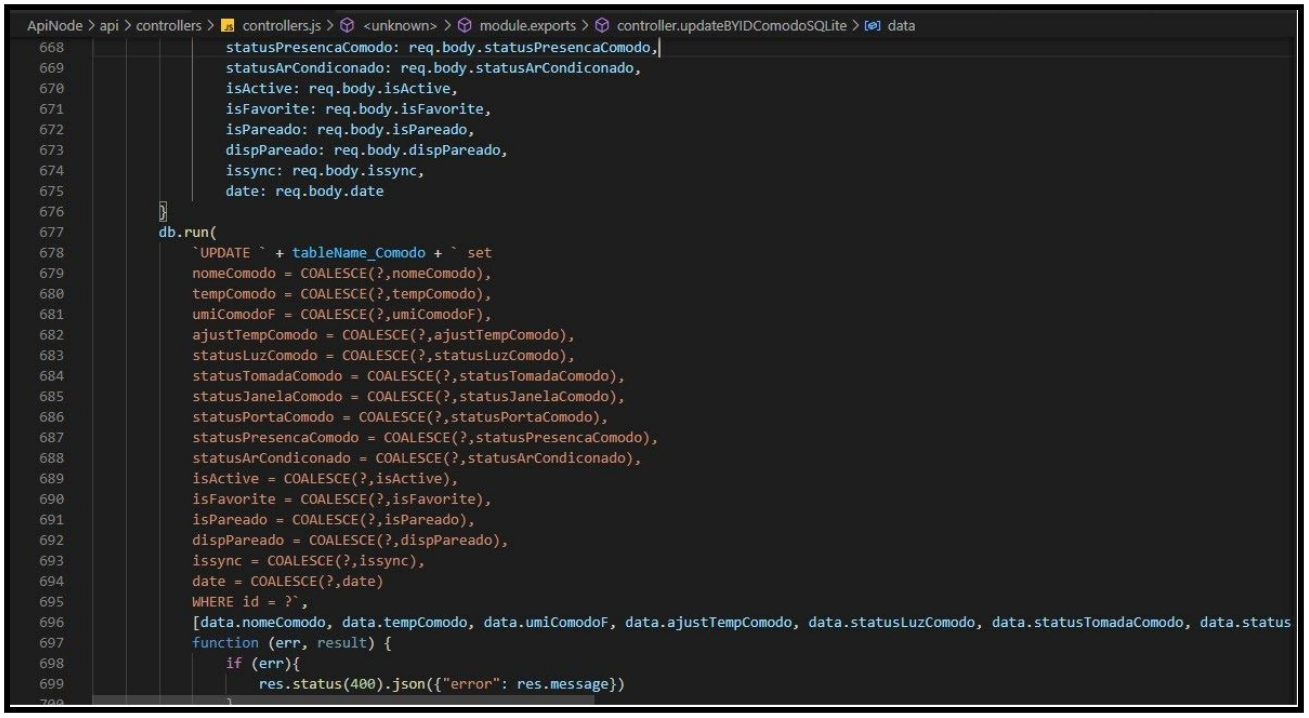


Figura 49: Método PUT com SQLite no cômodo parte 2

Figura 50: Método PUT com SQLite no cômodo parte 3

Figura 51: Método PUT com SQLite no usuário parte 1

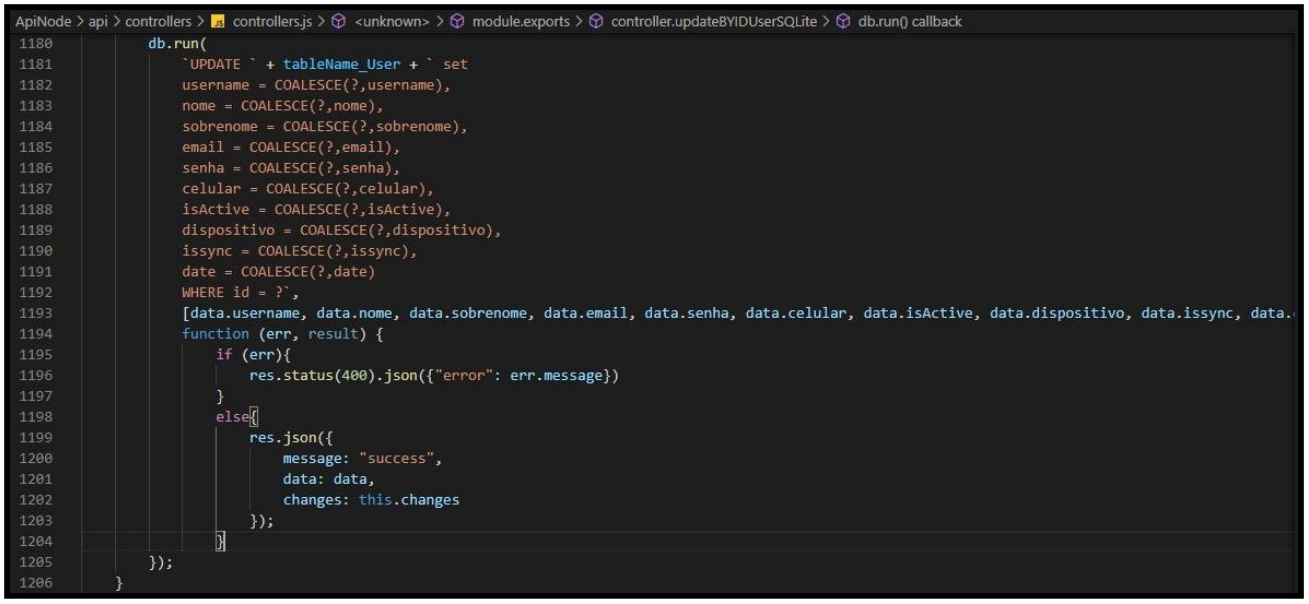


Figura 52: Método PUT com SQLite no usuário parte 2

Com o propósito de possibilitar que a *API* realize a remoção de um documento ou coluna armazenado nas *databases*, foi implementado o método *HTTP DELETE*, que como o nome diz é usado para remover elementos ou recursos específicos.

Para utilizá-lo com a biblioteca *Mongoose* com o intuito de remover documentos dentro de uma coleção, foi usado as constantes criadas com base nos *Schemas* de usuário e cômodo e o método *deleteOne*, que deleta um documento baseado em um objeto chave-valor com o campo para filtragem, que nesse caso foi o *id* do *MongoDB* que foi passado por *url*.

Esse método retorna um erro se existir, armazenando-o na variável erro e tratando com o condicionamento *if-else*, se ocorrer alguma falha então entra o *if* e envira o mesmo como *response*, caso contrário entra no *else* e enviá que o documento foi deletado com sucesso.

No *SQLite*, para deletar uma coluna foi usado a constante com a instância do banco de dados(*db*) junto com o método *run* para poder executar a query *DELETE*, que irá deletar uma coluna de uma tabela específica. A coluna desejada com foi filtrada com a cláusula *WHERE* e o valor passado pela *url*, retornando um erro se existir e um status da exclusão.

Para tratar o erro foi utilizado o condicionamento *if-else*, onde se o erro for verdadeiro, entra no if e retorna o mesmo como *response*, caso contrário, entra no *else* e envia um

response avisando que a coluna foi deletada com sucesso. As figuras 53, 54 e 55 abaixo representam as implementações para o usuário e cômodo que foram explicadas acima.

Figura 53: Método DELETE com MongoDB e SQLite no cômodo parte 1

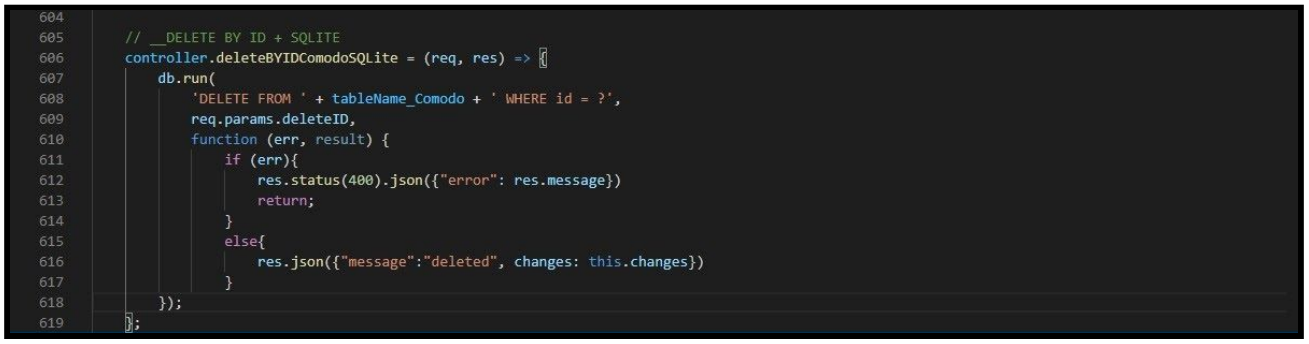


Figura 54: Método DELETE com MongoDB e SQLite no cômodo parte 2

Figura 55: Método DELETE com MongoDB e SQLite no usuário

Para que o usuário possa realizar o login no aplicativo móvel, foi desenvolvido na *API* um método de autenticação por tokens que utiliza a biblioteca *jsonwebtoken* baseado na tecnologia de *web tokens*. De acordo com o *website* jwt.io, é um padrão aberto RFC7519 que possibilita transmitir informações com segurança e de forma compacta como um objeto *json*.

Inicialmente foi necessário realizar uma busca na coleção de usuário do *MongoDB* para encontrar o documento que possui o mesmo *username* que foi enviado quando o usuário fez *login* no aplicativo, armazenando o resultado da busca na constante *user*.

Foi utilizado um condicionamento i*f-else* para verificar o status do erro e objeto *us* da constante *user*. Se for respectivamente falso e diferente de vazio os valores, então entra

no *if* para tratar o login do usuário, caso contrário entra no *else* e retorna como *response* que o usuário não foi encontrado.

Dentro do *if* foi verificado com outro condicionamento *if-else* se os campos de *username* e senha enviados pelo *body* são iguais aos encontrados no banco de dados. Caso forem, então entra no *if* para poder tratar o *token* de *login*. Caso contrário entra no *else* e envia como *response* que o login é inválido.

Dentro do if que trata o *token*, foi realizado a criação do token com a biblioteca citada acima. Foi criado uma constante *id* que recebe o ObjectID do usuário e uma variável *token*, que recebe um valor gerado com a instância da biblioteca(*jwt*) junto com o método *sign,* passando a constante *id* e uma data de expiração de 366 dias.

Esse token é enviado por *response* para o aplicativo móvel e tratado no mesmo, armazenando-o localmente no celular do usuário, assim não precisará ser chamado a *API* sempre quando precisar do *token*.

As figuras 56, 57 e 58 abaixo representam as implementações explicadas anteriormente.

```
1257    // LOGIN
1258    controller.login = async (req, res, next) => {
1259
1260        console.log(req.body);
1261        //console.log(req.body.senha)
1262
1263        var user = await User.findOne({username: req.body.username}).then((usr) =>{
1264            return {
1265                us: usr,
1266                error: !usr ? true : false,
1267                errorMessage: ''
1268            }
1269        }).catch((err) =>{
```

Figura 56: Processo de login parte 1

```
1269        }).catch((err) =>{
1270            console.log("Erro: "+err)
1271            return {
1272                us: '',
1273                error: true,
1274                errorMessage: "Erro Banco de dados: " + err + " " + err.message
1275            }
1276        });
1277
1278        console.log(user);
1279        // console.log("username req: " + req.body.username);
1280        // console.log("username mongo: " + user.us.username);
1281        // console.log("senha req: " + req.body.senha);
1282        // console.log("senha mongo: " + user.us.senha);
1283        if(user.us != null && user.error === false){
1284
1285            if(req.body.username == user.us.username && req.body.senha == user.us.senha ){
1286                //auth ok
1287                const id = user.us._id; //esse id viria do banco de dados
1288                var token = jwt.sign({ id }, process.env.SECRET, {
```

Figura 57: Processo de login parte 2

```
ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.login
1285            if(req.body.username == user.us.username && req.body.senha == user.us.senha ){
1286                //auth ok
1287                const id = user.us._id; //esse id viria do banco de dados
1288                var token = jwt.sign({ id }, process.env.SECRET, {
1289                expiresIn: "366d" // expires in 21 horas
1290                });
1291                res.json({ auth: true, token: token, expiresTime: "366 dias", idUser: user.us._id });
1292            }
1293            else{
1294                res.status(400).json({ auth: false, message: 'Login inválido!' });
1295            }
1296
1297        }
1298        else{
1299            res.status(400).json({ auth: false, message: 'Usuario não encontrado, Login inválido!: ', usr: user });
1300        }
1301    }
1302
1303
```

Figura 58: Processo de login parte 3

Para que o usuário possa alterar a senha na aplicação, foi desenvolvido na API um processo que permita isso com segurança, necessitando que o usuário digite a senha atual e a nova senha no aplicativo móvel. A senha atual é usada para verificar a autenticidade da operação.

Foi criado dois vetores, que foram usados para receber os erros e mensagens ao decorrer do processo. Também foi realizado uma busca semelhante a realizada no login, armazenando o resultado na constante *user*.

Antes de realizar o processo de troca de senhas, foi verificado com um condicionamento se o erro de *user* é verdadeiro ou falso, para que possa ser armazenado no vetor de erros ou de mensagens dependendo do resultado.

Logo em seguida foi implementado o processo para alterar a senha, iniciando com uma verificação *if-else* para saber se o objeto contendo os dados do usuário está vazio e se o erro dentro da constante *user* é falso.

Caso for bem sucedido, realiza mais uma verificação para determinar se a senha atual e *username* passados são iguais aos encontrados no *MongoDB*. Se forem, entra no *if* para realizar a operação de troca. Caso contrário, entra no *else* e retorna um erro como *response*.

Dentro do *if* de verificação de *username* e senha, foi chamado o método *updateOne*, que atualizará um documento da coleção de usuário baseado num objeto chave-valor passado como filtro e um outro que será o item a ser atualizado, nesse caso respectivamente são o *id* do documento dentro da constante *user* e o campo *newPasswd* passado pelo *body* da requisição.

O *updateOne* retornar um erro se existir e armazena na variável *err*, caso for verdadeiro o erro será tratado como mostrado nas implementações anteriores. Se não ocorrer nenhum erro, entra no *else* e armazena no vetor de mensagens que foi atualizado com sucesso.

Também foi verificado se o *issync* dentro de *user* é verdadeiro. Caso for, entra no *if* e executa a *query* de *UPDATE* na tabela do usuário no campo senha com a cláusula *WHERE* filtrando por id do *MongoDB*. Os tratamentos de erros e sucessos são os mesmos feitos na implementação do *PUT* no *SQLite*.

As figuras 59, 60, 61, 62 e 63 abaixo representam a implementação explicada anteriormente.

ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.alterPassWd

```
1309
1310    // ALTER PASSWD MONGO
1311    controller.alterPassWd = async (req, res, next) => {
1312
1313        var errors = [];
1314        var messages = [];
1315
1316        var user = await User.findOne({username: req.body.username}).then((usr) =>{
1317            return {
1318                us: usr,
1319                error: false,
1320                message: 'User Encontrado'
1321            }
1322        }).catch((err) =>{
1323            console.log("Erro: "+err)
1324            return {
1325                us: '',
1326                error: true,
1327                message: "Erro Banco de dados: " + err + " " + err.message
1328            }
1329        });
1330
1331        console.log(user.error);
1332        console.log(user.us.username);
1333
1334        if(user.error == true){
1335            errors.push(user)
1336        }
1337        else{
1338            t = {
1339                error: user.error,
1340                message: user.message
```

Figura 59: Processo para alterar a senha do usuário parte 1

ApiNode > api > controllers > controllers.js > <unknown> > module.exports > controller.alterPassWd > usr > User.updateOne() callback

```
1337        else{
1338            t = {
1339                error: user.error,
1340                message: user.message
1341            }
1342            messages.push(t)
1343        }
1344
1345        console.log(user);
1346
1347        if(user.us != null && user.error == false){
```

Figura 60: Processo para alterar a senha do usuário parte 2

```
1348        if(req.body.username == user.us.username && req.body.senha == user.us.senha){
1349            const usr = User.updateOne({_id: user.us._id}, {senha: req.body.newPasswd}, (err) => {
1350                if(err){
1351                    errors.push(JSON.parse('{"error": true, "message": "Senha Não Editada: ' + String(err.message).replace(/\\*([\" ])/g, '
1352                    // res.status(400).json({
1353                    //     error: true,
1354                    //     message: "Usuario Não Foi Editado Com Sucesso: " + erro
1355                    // });
1356                    res.status(400).json({
1357                        errors: errors,
1358                        messages: messages
1359                    });
1360                }
1361                else{
1362                    // res.status(200).json({
1363                    //     error: false,
1364                    //     message: "Usuario Editado Com Sucesso"
1365                    // });
1366                    messages.push(JSON.parse('{"error": false, "message": "Senha Atualizada Com Sucesso", "err": ""}'))
1367                    if(user.us.issync){
```

Figura 61: Processo para alterar a senha do usuário parte 3

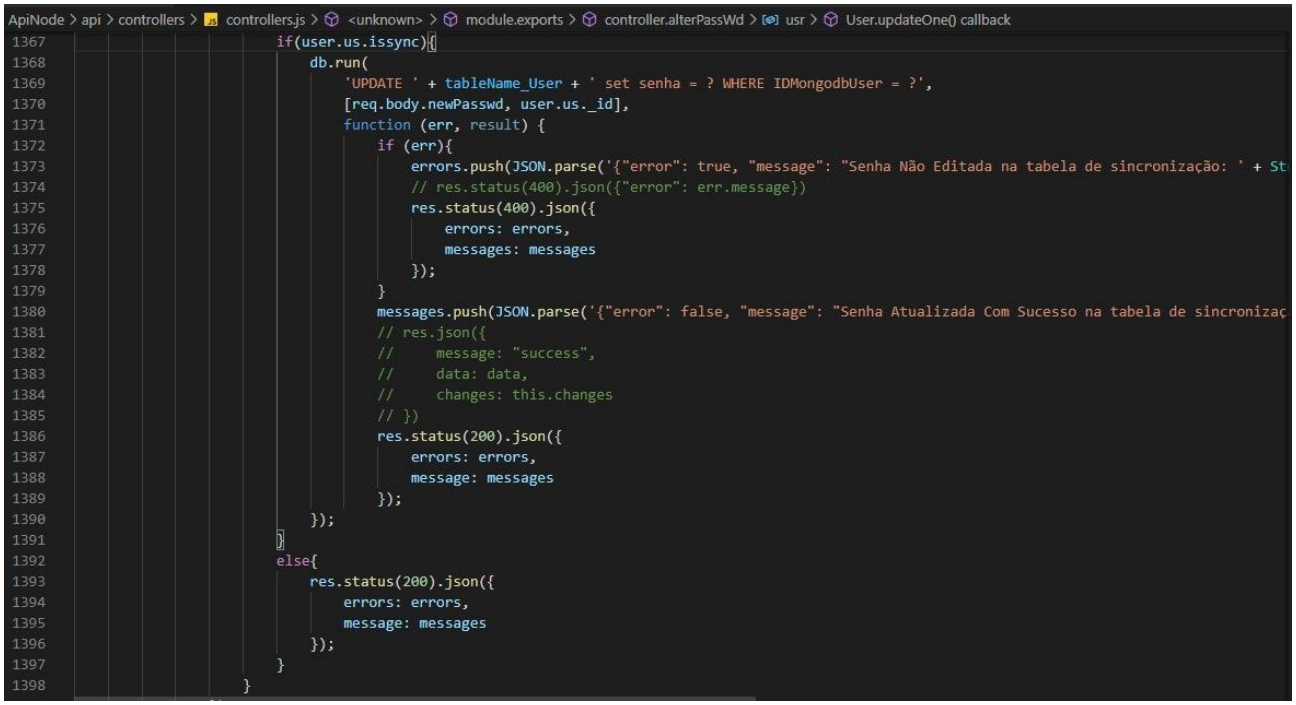


Figura 62: Processo para alterar a senha do usuário parte 4

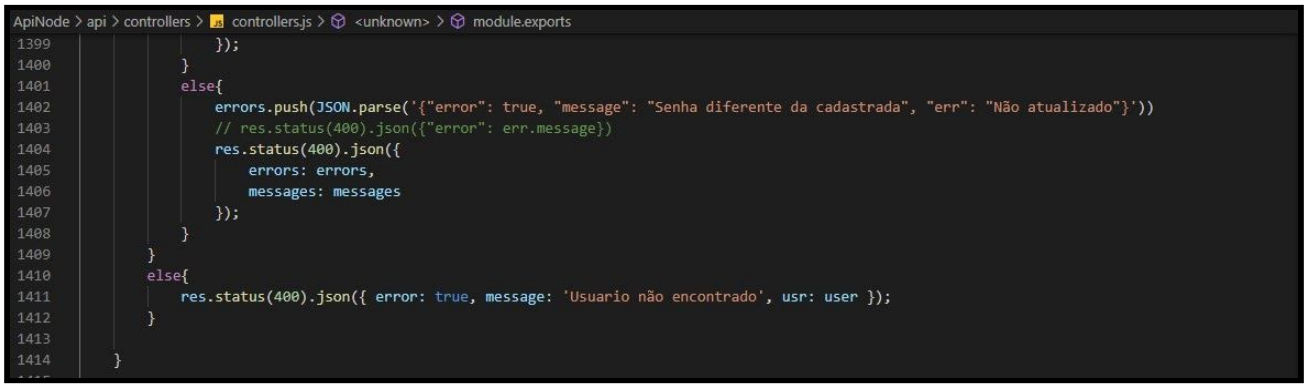


Figura 63: Processo para alterar a senha do usuário parte 5

Com o intuito de evitar que pessoas não autorizadas possam acessar as requisições da *API*, foi criado um verificador de token, que possibilita analisar se em um *request HTTP* existe no cabeçalho um *token* de acesso, ou se o mesmo é válido.

Para isso, foi criado uma função que recebe como parâmetro, toda a estrutura de requisição *HTTP*. Dentro dessa *function* foi criado uma variável chamada *token* que recebe o cabeçalho com o nome de *x-access-token* da requisição. Logo em seguida foi verificado se essa variável é vazia. Caso for, entra no if e retorna como response que não foi encontrado *token*.

Caso a variável *token* não for vazia, então continua a execução normalmente, realizando o processo de verificação de token com a biblioteca *jsonwebtoken* e o método *verify*.

É passado para o *verify* a variável *token* e a localização da senha que é usada para criptografar ou descriptografar o *token*, retornado um erro se existir e um valor descriptografado do *token*. Também foi verificado se ocorreu algum erro, em caso positivo, entra no *if* e enviá que teve uma falha ao autenticar o *token*, caso contrário continua normalmente para a lógica da *function*.

Essa função é colocada junto a parte de *routers* chamando a mesma antes de chamar a lógica da requisição, como mostrado no exemplo, antes de chamar o listar cômodo, é chamado a function *verifyJWT*. As figuras 64 e 65 abaixo representam a explicação anterior.

```
7    function verifyJWT(req, res, next){
8        var token = req.headers['x-access-token'];
9        if (!token) return res.status(401).json({ auth: false, message: 'Nenhum token providenciado' });
10
11       jwt.verify(token, process.env.SECRET, function(err, decoded) {
12           //console.log(err)
13           //console.log(decoded)
14         if (err) return res.status(401).json({ auth: false, message: 'Falha em autentificar o token' });
15
16         // se tudo estiver ok, salva no request para uso posterior
17         //req.userId = decoded.id;
18         next();
19       });
20   }
```

Figura 64: Função para autenticar token de login

```
//__GET + MONGODB
app.route('/api/getAllComodos').get(verifyJWT, controller.listComodos);

//__GET + SQLITE
app.route('/api/getAllComodosSQLite').get(verifyJWT, controller.listComodosSQLite);
```

Figura 65: Exemplo routers

Para que todo esse sistema funcione apropriadamente, foi montado uma arquitetura de conexão que engloba cada parte desde a requisição do usuário na hora de criar um cômodo até a atualização dos valores enviada pelo raspberry, visando segurança e praticidade. Como mostrado na figura 66 abaixo.

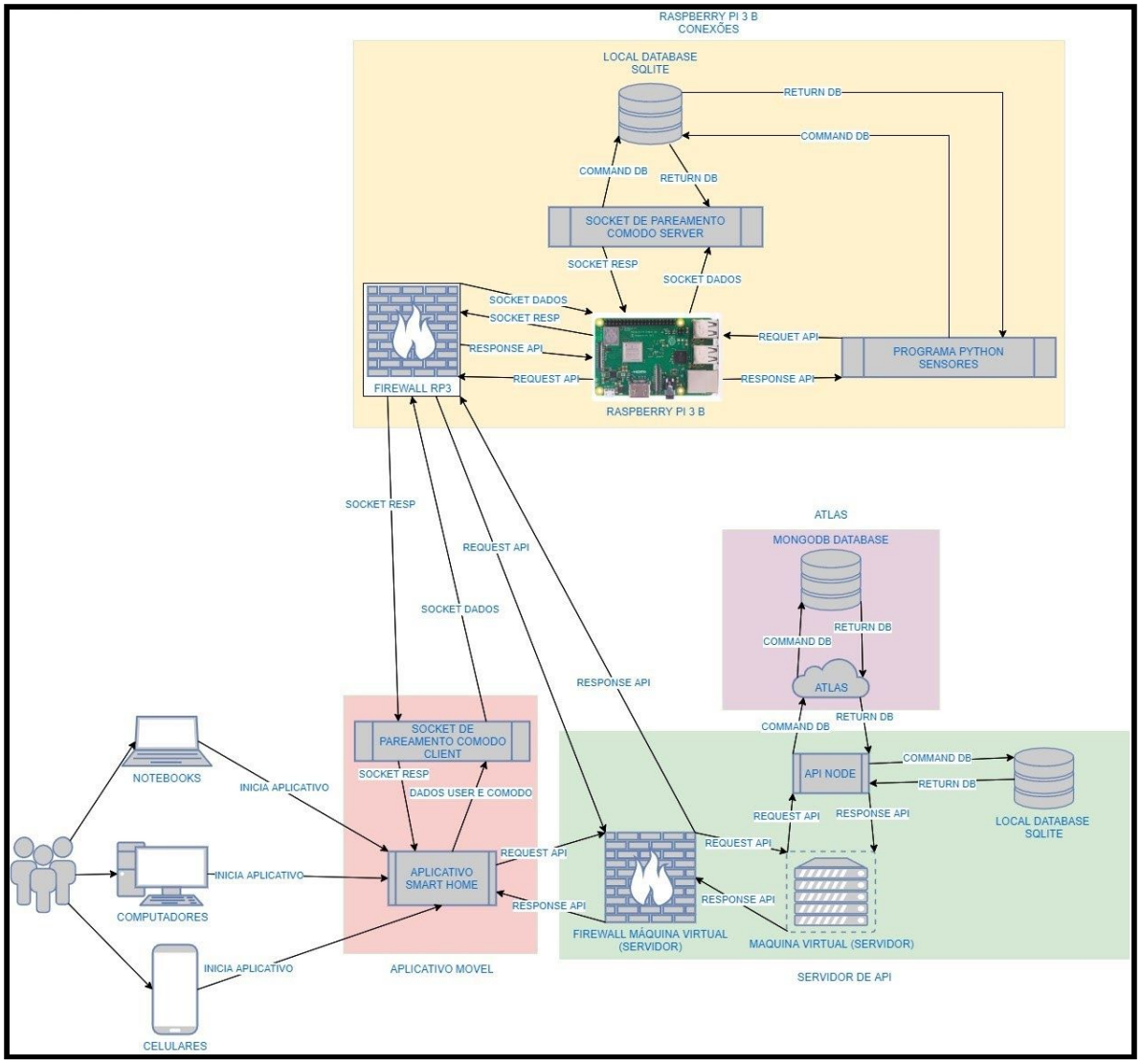


Figura 66: Esquematização de Conexões do Projeto

O processo tem início quando o usuário entra no aplicativo móvel por algum dispositivo e realiza alguma requisição que manipule alguma dado, como realizar login na conta, criar um novo cômodo, entre outros disponíveis na aplicação. Essas ações enviam uma *request* diretamente para a *API* que está armazenada em uma máquina virtual para simular um servidor e evitar sobrecarregamento do *raspberry*.

Dentro da *API*, inicialmente as requisições vão passar pelo *firewall* da máquina virtual que está configurado para permitir acessos na porta 9999 vindos de todos as máquinas. Após passar pelo *firewall*, a requisição vai para a *API* feita em *Node Js*, que executará a regra de negócio que foi requisitada e interagir com os bancos de dados.

A *database* principal usada foi o banco não relacional *MongoDB* que é gerenciada e armazenada pela plataforma *online Atlas*, armazenando todos os dados de usuários e cômodos. Também foi utilizado o banco de dados *SQLite* armazenado localmente junto com a *API* para *backup* de todos os dados do banco principal, sendo salvo dentro dele somente se a sincronização estiver ativada.

Para que o *raspberry* possa enviar os dados para o cômodo certo, foi desenvolvido um sistema usando *websockets* com a biblioteca *socket.io* e o ambiente *Node JS*, assim quando o usuário clicar no botão para parear o cômodo no aplicativo móvel, será enviado uma mensagem com essa biblioteca para o *raspberry*.

Essa aplicação aguarda mensagens com dados de *id* do usuário, do cômodo e *token* de login atual. Esses dados serão armazenado num banco de dados local *SQLite* para usos futuros.

No *raspberry* também possui um programa em *Python* que fica verificando os dados que os sensores enviam, e cada vez que houver alguma atualização, vai ser chamado um *request* de *update* das *databases*.

As conexões dos sensores no raspberry podem ser vistas no diagrama de circuito abaixo.

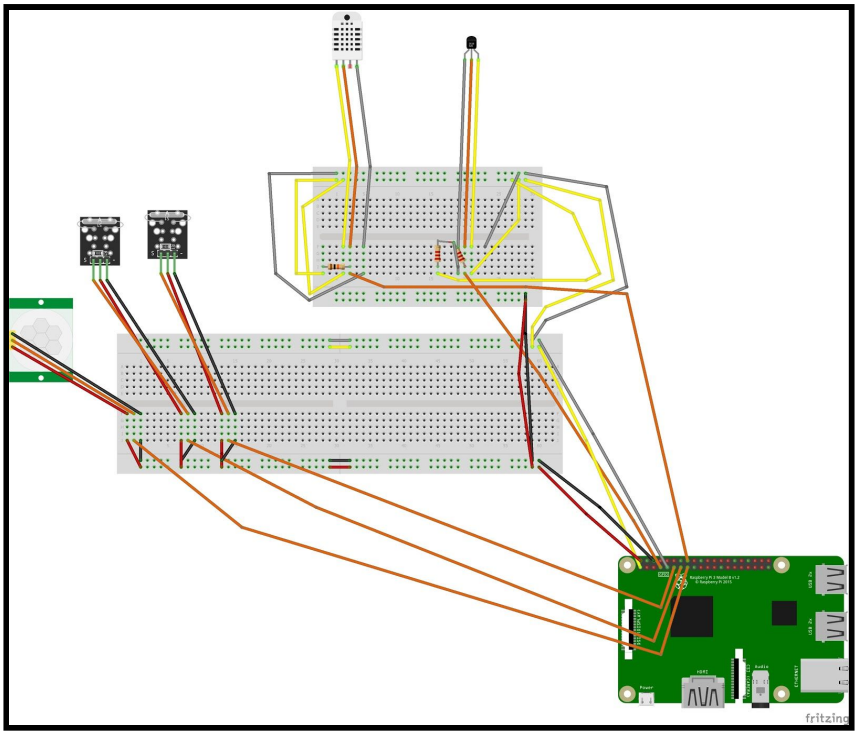


Figura 67: Conexões de Sensores do Projeto

Foi usado duas *protoboards* e os sensores *hc-sr501 pir* usado para detectar presenças no cômodo, dois sensores *reed switches* para detectar se as portas e janelas estão abertas ou fechadas, um sensor *DHT22* para detectar a temperatura e umidade e o sensor de temperatura *ds18b20 waterproof*.

O sensor *hc-sr501 pir* foi conectado a um cabo ligado aos 5v e um *gnd* (terra) para que fosse ligado, e com o intuito de passar os dados de presença detectado foi conectado a ele um cabo ligado a *GPIO22* do *raspberry*.

Os sensores *reed switches* também foram conectados a um cabo de 5v e um *gnd*, e com o objetivo de transferir os dados da porta e janela, foram conectados nos GPIO27 e GPIO17 do raspberry.

Já o sensor *DHT22* foi conectado em um cabo de 3v e um *gnd*, e para que fosse passados os dados, precisou ser um pouco diferente do usado anteriormente, para esse sensor foi preciso conectar um cabo de 3v ligado a um resistor de 10k *ohm* no pino de dados antes de conectá-lo ao cabo ligado ao *GPIO23* do *raspberry*.

No sensor *ds18b20 waterproof* foi conectado também a um cabo de 3v e um *gnd*, e antes de conectar o cabo de dados ao *raspberry* foi conectado um cabo de 3v ligado a dois resistores de 2.2k *ohm* interligados, e como esse sensor funciona com a tecnologia de *1-wire bus* para passar os dados, foi conectado no pino *GPIO4* do raspberry que usa essa mesma tecnologia.

As figuras 68 a 89 representam como o aplicativo móvel está atualmente, contendo a tela mostrada quando o usuário entrar na no aplicativo, além de ter as telas de registo e *login*.

Está sendo demonstrado também alguns exemplos de cômodos cadastrados na tela de *homepage*, na tela de informações do cômodo e na tela de apresentação das informações enviadas pelos sensores.

Também está sendo representado as telas relacionado às configurações do usuário, que mostra as informações do mesmo, além de demonstrar as funcionalidades de alteração de dados e senha do usuário.

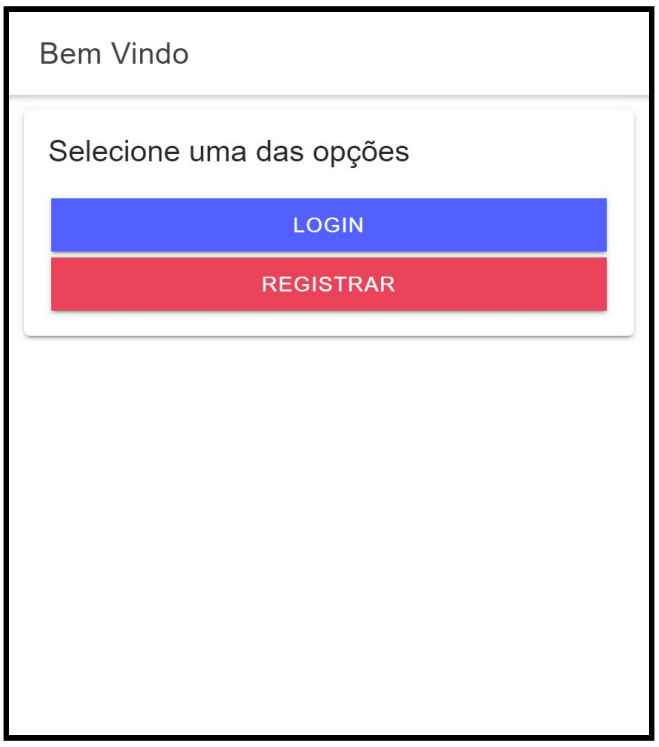


Figura 68: Tela inicial Aplicativo Móvel final

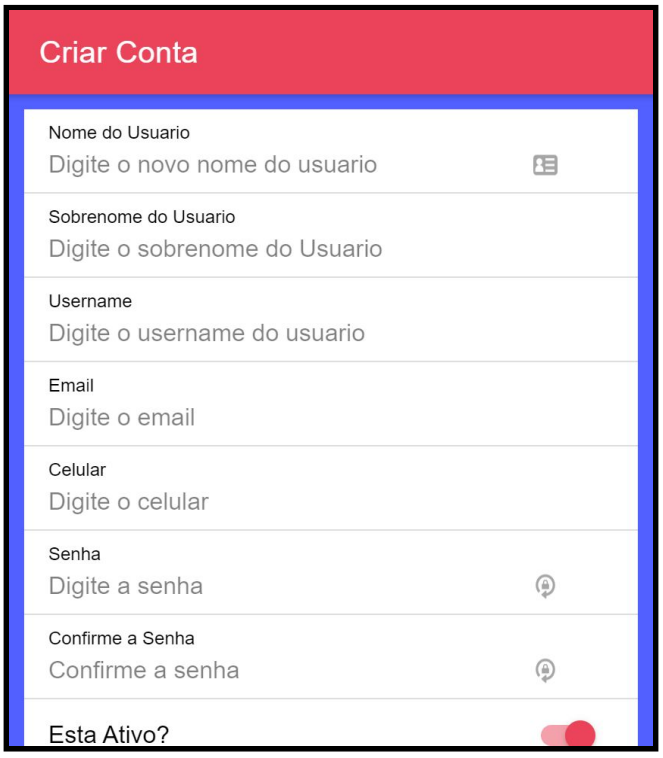


Figura 69: Tela de registro de usuário Aplicativo Móvel final parte 1

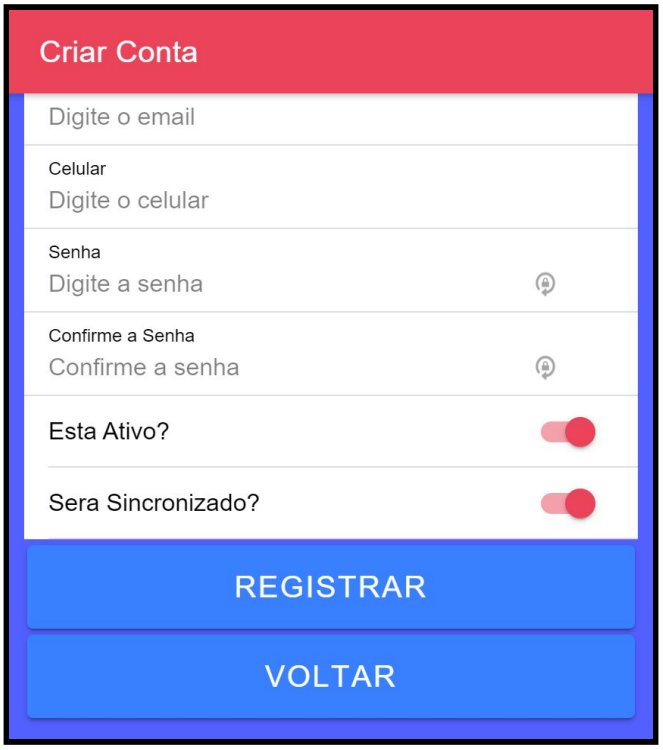


Figura 70: Tela de registro de usuário Aplicativo Móvel final parte 2

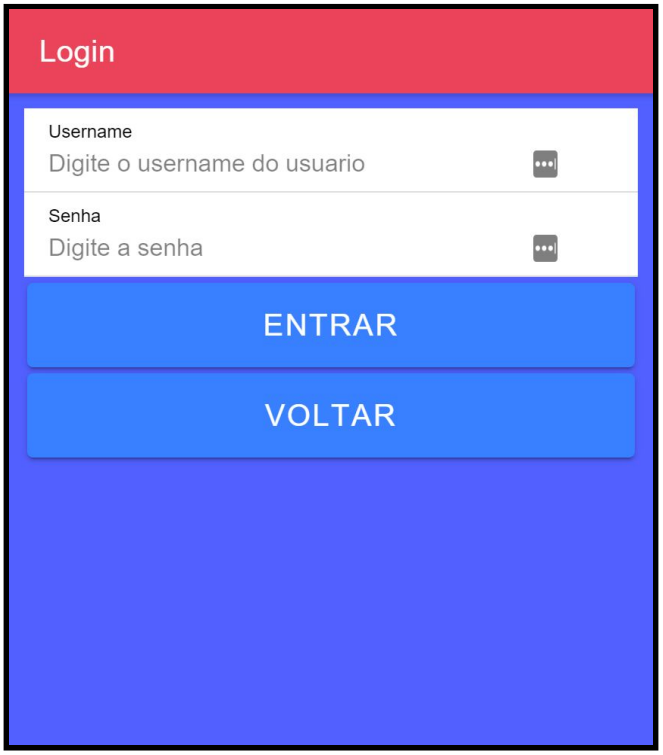


Figura 71: Tela de login de usuário Aplicativo Móvel final

Figura 72: Tela de homepage Aplicativo Móvel final parte 1

Figura 73: Tela de homepage Aplicativo Móvel final parte 2

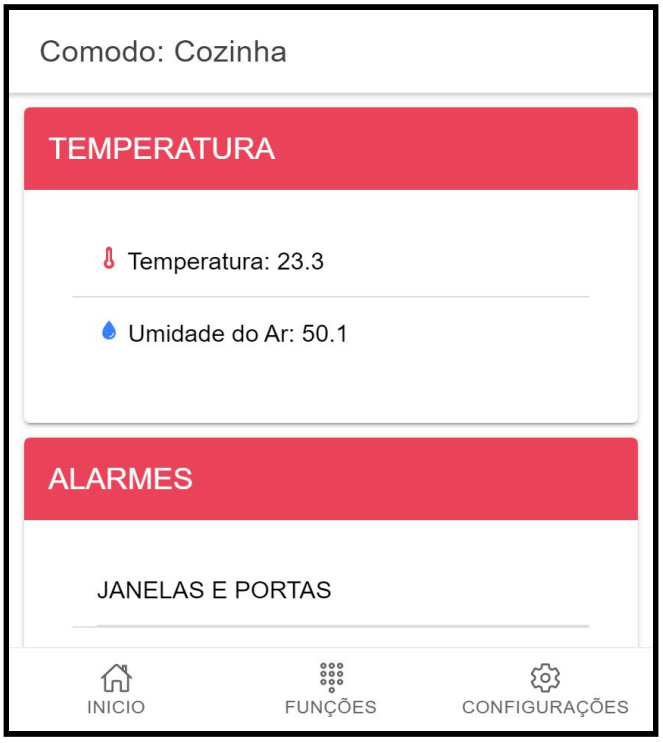


Figura 74: Tela de informação do cômodo Aplicativo Móvel final parte 1

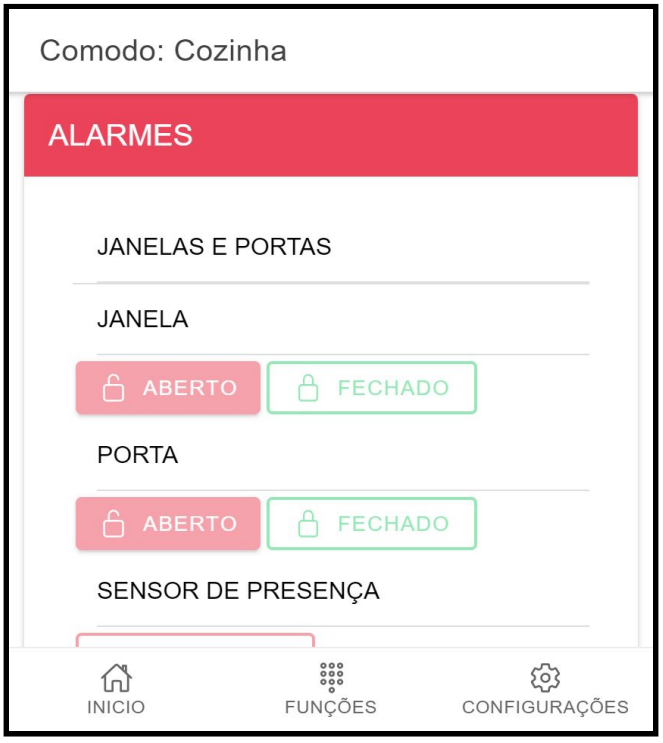


Figura 75: Tela de informação do cômodo Aplicativo Móvel final parte 2

Figura 76: Tela de informação do cômodo Aplicativo Móvel final parte 3

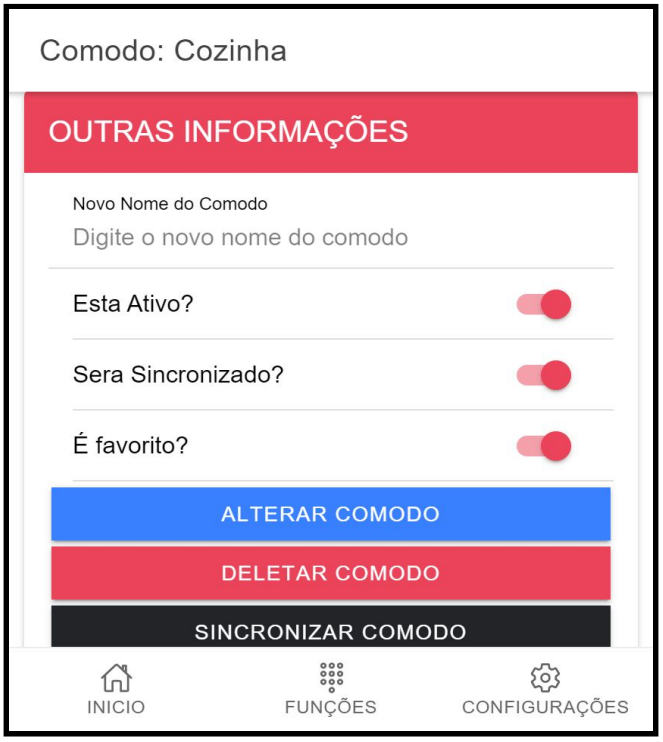


Figura 77: Tela de informação do cômodo Aplicativo Móvel final parte 4

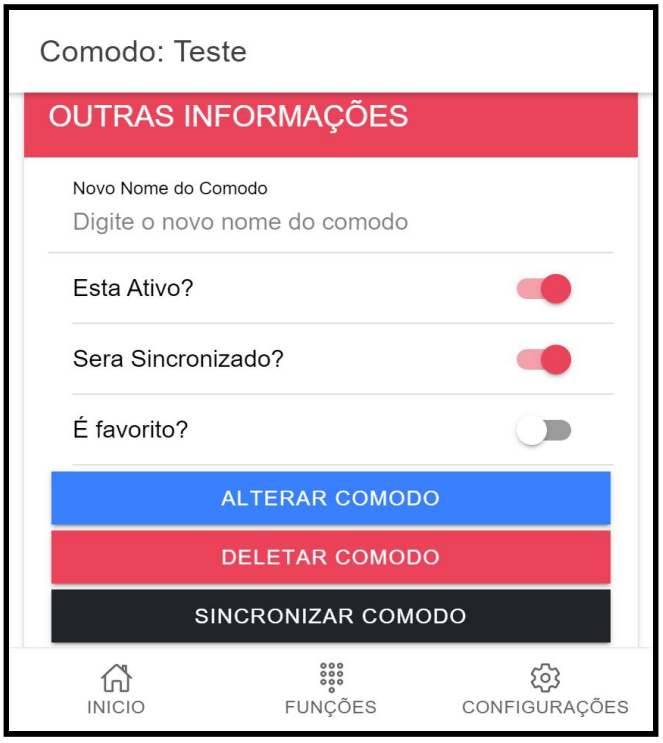


Figura 78: Tela de informação do cômodo Aplicativo Móvel final parte 5

Figura 79: Tela de temperatura Aplicativo Móvel final parte 1

Figura 80: Tela de temperatura Aplicativo Móvel final parte 2

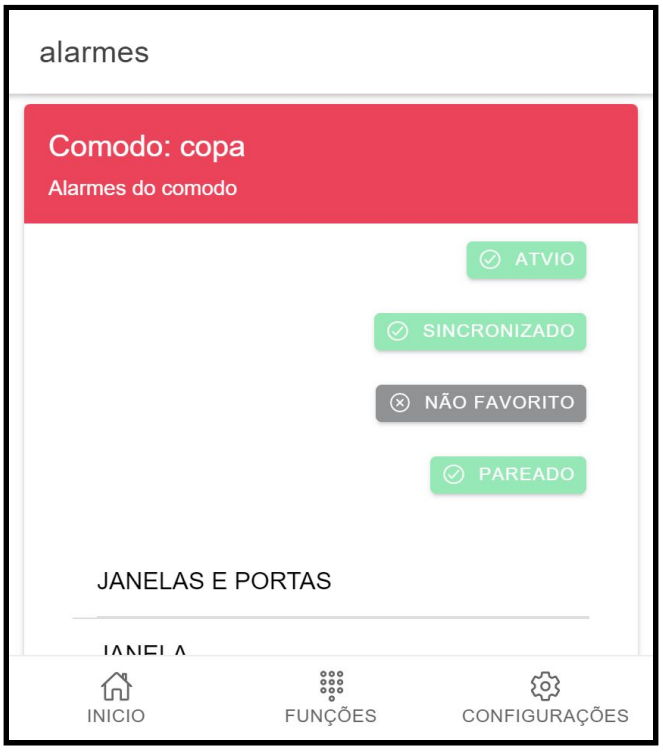


Figura 81: Tela de alarmes Aplicativo Móvel final parte 1

Figura 82: Tela de alarmes Aplicativo Móvel final parte 2

Figura 83: Tela de alarmes Aplicativo Móvel final parte 1

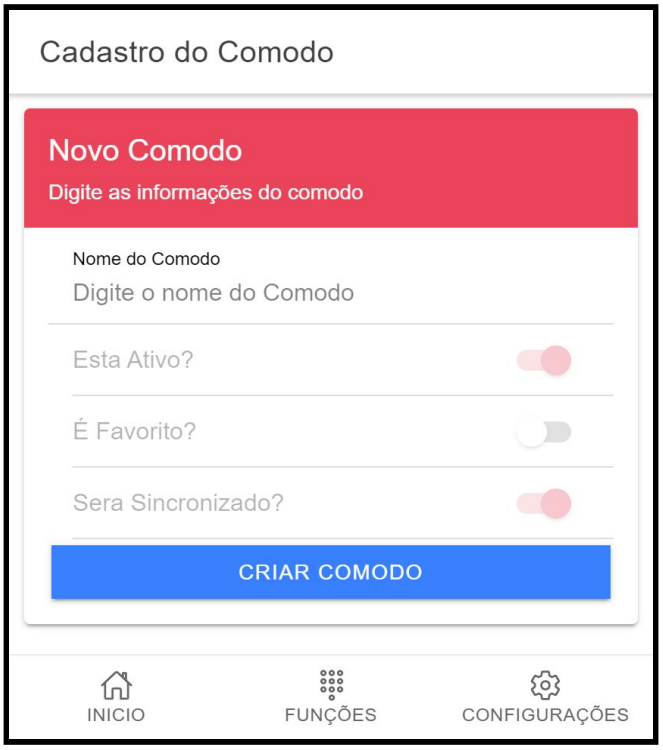


Figura 84: Tela de cadastros de cômodos Aplicativo Móvel final

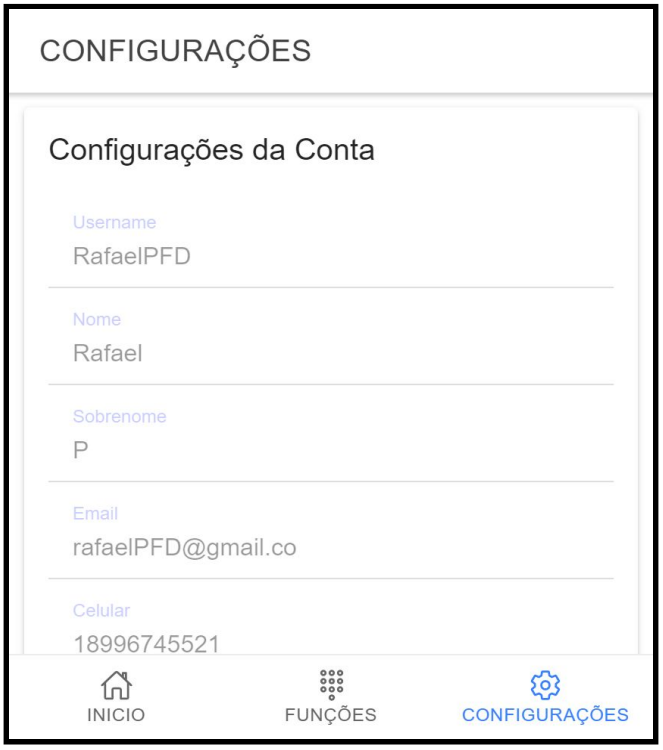


Figura 85: Tela de configurações de usuário Aplicativo Móvel final parte 1

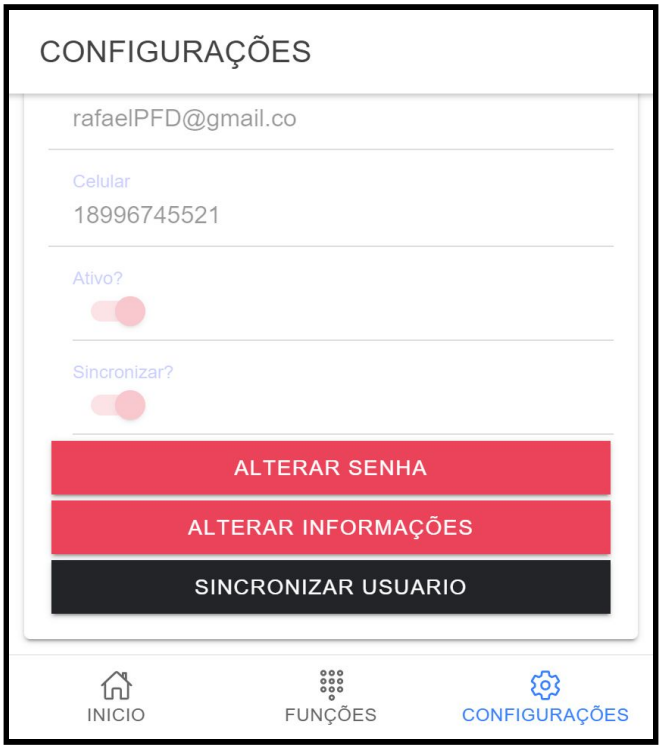


Figura 86: Tela de configurações de usuário Aplicativo Móvel final parte 2

Figura 87: Tela de alteração de informações do usuário Aplicativo Móvel final parte 1

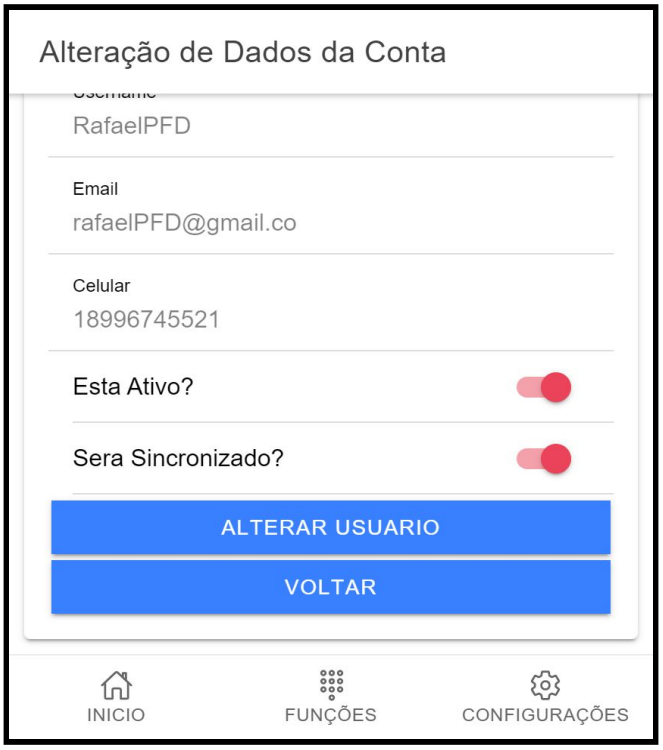


Figura 88: Tela de alteração de informações do usuário Aplicativo Móvel final parte 2

Figura 89: Tela de alteração de senha do usuário Aplicativo Móvel final

4. CONCLUSÃO

Com o desenvolvimento deste trabalho, foi possível perceber que as *smart homes* possuem uma capacidade enorme de se desenvolver no mercado atualmente, por conta de tecnologias que auxiliam sua versatilidade, interatividade e capacidade de processamento dos dados coletados dentro da residência.

Com o intuito de desenvolver uma aplicação de baixo custo e que seja acessível para a maioria das pessoas para automação de suas residências e auxiliar pessoas com problemas fisiológicos, foram escolhidas tecnologias de baixo custo e eficazes para projetos que dependam de cálculos lógicos e velocidade de resposta, assim como na busca de dados.

A metodologia de *IoT (Internet of Things)* ajudou a projetar e desenvolver a arquitetura de rede presente neste trabalho, possibilitando interligar diversos dispositivos e sensores além de permitir transmitir dados do ambiente entre eles para monitoramento do usuário no aplicativo móvel. Por ser uma metodologia versátil e simples, não foi difícil implementá-la no projeto.

Com a linguagem de programação *Python*, foi possível desenvolver um *script* para receber os dados vindos dos sensores com facilidade, por ser muito intuitiva, além de fácil aprendizagem, assim em poucos meses foi possível aprender partes importantes para o desenvolvimento do projeto.

Por ser uma linguagem que ganhou popularidade ao decorrer dos anos, foi possível encontrar diversas bibliotecas e conteúdos na internet para auxiliar ao longo do desenvolvimento do projeto.

No desenvolvimento do Web Server para hospedar a API foi usado uma máquina virtual com o sistema operacional Zorin OS Lite. Inicialmente se teve dificuldade na sua preparação, pois precisou-se preparar todo o ambiente, abrir portas para acessos externos e instalar todas as ferramentas necessárias, consumindo tempo e necessitando buscar treinamentos para configurá-lo.

O desenvolvimento da API foi feito sem muitas dificuldades, por se tratar de uma API Restful, sua arquitetura foi simples de ser entendido e implementado, além de possuir um grande conteúdo em sites e artigos, o que ajudou nas dúvidas e em desenvolver de forma bem estruturada a aplicação. As únicas dificuldades foram na hora de criar o sistema de sincronização na API.

O ambiente *Node JS* possibilitou desenvolver a *API* e o sistema de pareamento raspberry-cômodo utilizando *JavaScript*. Por ser uma tecnologia abrangente no mercado, foi encontra um conteúdo de consulta abrangente na internet, o que facilitou no desenvolvimento de algumas partes da *API*, como o sistema para *requests* e a estrutura da aplicação, além de ajudar na criação do sistema de pareamento com o *socket.io*.

Porém algumas partes foram dificultosas ao decorrer do projeto por ser uma ambiente *single thread*. Algumas das implementações não funcionavam inicialmente por precisar esperar um evento terminar para poder obter os dados dos mesmo. Foi preciso realizar algumas improvisações para conseguir resolver esses empecilhos.

O desenvolvimento do aplicativo móvel foi relativamente fácil, por conta de já conhecer o framework Ionic e já ter desenvolvidos outras aplicações com o mesmo, além de já possuir familiaridade com aplicações Webs. Mesmo só conhecendo versões anteriores do Ionic, foi fácil aprender as novas funcionalidades que surgiram com a versão 6, pois existem vários exemplos em sites e artigos.

Os bancos de dados utilizados foram simples e de fácil utilização. Por conta do *MongoDB* ser *NOSQL*, foi simples implementar as coleções e documentos da *database*, utilizando estruturas em *JSON* simples e a biblioteca *Mongoose* para realizar todas as funções de *GET*, *POST*, *PUT*, *DELETE*.

Com a biblioteca de *SQLite* se obteve um resultado satisfatório, por conta de ser compacto e leve, proporcionou que fosse instalado na máquina virtual sem precisar se preocupar com falta de espaço ou muito uso de processamento. A única dificuldade foi a conversão de dados vindos de uma *database NOSQL* para uma *SQL*.

O raspberry e os sensores também não deixaram a desejar, foram extremamente precisos e práticos com relação a sua utilização no projeto. O minicomputador proporcionou um ambiente ideal para projetar *scripts*, enquanto os sensores retornaram

valores em tempo real. A única desvantagens do sensores foi ter que regula-los para que se adequem às necessidades, alem de que o *DHT22* possui pinos de conexão finos, ocorrendo de esporadicamente se desconectar e causar falha de leitura.

4.1. TRABALHOS FUTUROS

A partir desse projeto é possível dar continuidade, acrescentando novas tecnologias, como um sistema de docker que iria deixar a aplicação mais segura e permitir que os serviços do raspberry sejam melhor gerenciados.

Outro quesito a melhorar seria o sistema de pareamento de cômodos com os sensores, deixando mais seguro e mais polido em relação a bugs e demais imprevistos. Também pretende-se migrar a aplicação móvel para uma linguagem completamente nativa a dispositivos móveis, como *Flutter*, fazendo com que o desempenho em celulares aumente, além de possuir funções que não existem em aplicações Ionic.

Outras contribuições que podem ser feitas no projeto seria adicionar a API em uma máquina gerenciada pela Amazon para melhorar o desempenho e segurança, além de adicionar uma versão local do MongoDB junto com a API, tirando a necessidade de ter que acessar sempre ao Atlas para ter os dados.

Também pretende-se implementar futuramente um sensor de emissão infravermelho para que possa ser controlado o ar condicionado dentro de um cômodo, porém será necessário decodificar os códigos enviados pelo controle do eletrodoméstico para que possa ser enviado pelo emissor.

Além disso pretende-se adicionar relés para controle das luzes do cômodo e uma tomada que usa a tecnologia *wi-fi* para controlar algumas tomadas existentes dentro do mesmo, porém para que seja utilizado essa tomada *wi-fi* irá ser preciso mudar o *firmware* do equipamento para conectar-se ao aplicativo desenvolvido.

REFERÊNCIAS

BHOSALE, Satish Tanaji; Patil, Tejaswini; Patil, Pooja. SQLite: Light Database System. **International Journal of Computer Science and Mobile Computing**, v.4, n.4, abril, 2015. pg. 882-885.

CABRAL, Isabela. **Como funciona o Bluetooth 5.0 no celular**. Disponível em: <https://www.techtudo.com.br/noticias/2018/02/como-funciona-o-bluetooth-50-no-celular.ghtml>. Acesso em: 18 fev. 2020.

CARVALHO, Lucas. **Raspberry Pi: o que é, para que serve e como comprar**. Olhar Digital. Disponível em: <https://olhardigital.com.br/noticia/raspberry-pi-o-que-e-para-que-serve-e-como-comprar/82921>. Acesso em: 01 mar. 2020.

ERL, Thomas. **Web Services**. Disponível em: <https://www.devmedia.com.br/web-services/2873#UDDI>. Acesso em: 15 fev. 2020.

FAROOQ, M.U; WASEEM, Muhammad; MAZHAR, Sadia; KHAIRI, Anjum; KAMAL, Talha. A Review on Internet of Things (IoT). **International Journal of Computer Applications**, v.113, n.1, março, 2015.

GOOGLE MAPS PLATFORM. **Google Maps Platform FAQ**. Disponível em: <https://developers.google.com/maps/faq#whatis>. Acesso em: 15 fev. 2020.

GOOGLE MAPS PLATFORM. **Maps JavaScript API**. Disponível em: <https://developers.google.com/maps/documentation/javascript/tutorial> Acesso em: 15 fev. 2020.

GUBBI, Jayavardhana; BUYYA, Rajkumar; MARUSIC, Slaven; PALANISWAMI, Marimuthu. Internet of Things (IoT): A vision architectural elements and future directions. **Future Generation Computer Systems**, Janeiro, 2013.

HEYWOOD, Mark. **Ultrasonic HC-SR04 Sensor Python Library for Raspberry Pi GPIO**. Disponível em: <https://www.bluetin.io/sensors/python-library-ultrasonic-hc-sr04/>. Acesso em: 19 fev. 2020.

JOHNSON, Glenn. **Application Programming Interface (API) Definition and Classifications by Type**. Disponível em: <https://it.toolbox.com/blogs/glennjohnson/application-programming-interface-api-definition-and-classification-by-types-040214>. Acesso em: 15 fev. 2020.

JUNIOR, Luiz Fernando Duarte. **MongoDB para iniciantes em NoSQL**. Disponível em: <https://imasters.com.br/banco-de-dados/mongodb-para-iniciantes-em-nosql>. Acesso em: 18 fev. 2020.

JWT. **Introduction to JSON Web Tokens**. Disponível em: <https://jwt.io/introduction/>. Acesso em: 08 ago. 2020.

LIMA, Thiago. **Raspberry Pi A+**. Disponível em: <https://www.embarcados.com.br/raspberry-pi-a/>. Acesso em: 08 fev. 2020.

LIMA, Thiago. **Raspberry Pi B+**. Disponível em: <https://www.embarcados.com.br/raspberry-pi-b-plus/>. Acesso em 08 fev. 2020.

LIMA, Izabelle. **Aprenda a utilizar o Sensor de Distância Ultrassônico HC-SR04 com Arduino**. Disponível em: <https://autocorerobotica.blog.br/aprenda-utilizar-o-sensor-de-distancia-ultrassonico-hc-sr04-com-arduino/>. Acesso em: 19 fev. 2020.

MADAKAM, Somayya; RAMASWAMY, R; TRIPATHI, Siddharth. Internet of Things (IoT): A Literature Review. **Journal of Computer and Communications**, março, 2015. p.164-173.

MAO, Xiaobo; LI, Keqiang; ZHANG, Zhiqiang; LIANG, Jing. **Design and Implementation of a New Smart Home Control System Based on Internet of Things**. 2017. 5p. School of electrical engineering - Zhengzhou University, Henan, Zhengzhou, 2017.

MARION, Charles; JOMIER, Julien. Real-time Collaborative Scientific WebGL Visualization with WebSocket. In: PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON 3D WEB TECHNOLOGY, 17, 2012, New York, USA. **Web3D '12**, 47–50.

MARTINS, Flávio de Oliveira Coelho. **Projetos de casas inteligentes e Design Thinking: geração e seleção de concepções baseadas em soluções tecnológicas inovadoras**. 2017. 158p. Dissertação de Mestrado - Programa de Pós-Graduação em Metrologia (Área de concentração: Metrologia para Qualidade e Inovação) - PUC-Rio, Rio de Janeiro, Rio de Janeiro, 2017.

MAXIM INTEGRATED. **DS18S20**. Disponível em: <https://www.maximintegrated.com/en/products/sensors/DS18S20.html>. Acesso em: 19 fev. 2020.

MAXIM INTEGRATED. **DS18B20**. Disponível em: <https://www.maximintegrated.com/en/products/sensors/healthcare-sensor-ics/electrochemical-sensor-afe-ics/DS18B20.html>. Acesso em: 20 fev. 2020.

MDN WEB DOCS. **O que é um servidor web (web server)?**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Common_questions/o_que_e_um_web_server>. Acesso em: 16 fev. 2020.

MEDEIROS, Higor. **Introdução ao MongoDB**. Disponível em: <https://www.devmedia.com.br/introducao-ao-mongodb/30792>. Acesso em: 18 fev. 2020.

NGINX. **What Is a Web Server?**. Disponível em: <https://www.nginx.com/resources/glossary/web-server/>. Acesso em: 16 fev. 2020.

NODEBR. **O que é Node.js?**. Disponível em: <http://nodebr.com/o-que-e-node-js/>. Acesso em: 17 fev. 2020.

OLIVEIRA, Edvar da L.; ALFAIA, Rodrigo D.; SOUTO, Anderson V. F.; SILVA, Marcelino S.; FRANCÊS, Carlos Renato L.; VIJAYKUMAR, N. L. **SmartCoM: Smart Consumption Management Architecture for Providing a User-Friendly Smart Home based on Metering and Computational Intelligence**. Journal of Microwaves, Optoelectronics and Electromagnetic Applications. Disponível em <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S2179-10742017000300736&lang=pt#aff1>. Acesso em: 24 out. 2019.

OLIVEIRA, Robisson. **Como o JavaScript funciona: Aprofundando em WebSockets e HTTP/2 com SSE + como escolher o caminho certo**. Disponível em: <https://medium.com/reactbrasil/como-o-javascript-funciona-aprofundando-em-websockets-e-http-2-com-sse-como-escolher-o-caminho-d4639995ef85>. Acesso em: 03 ago. 2020.

OPUS-SOFTWARE. **Node.js – O que é, como funciona e quais as vantagens**. Disponível em: <https://www.opus-software.com.br/node-js/>. Acesso em: 17 fev. 2020.

PASSOS, Clayton K. N. **O que é o IONIC ?**. Disponível em: <https://medium.com/codigorefinado/oque-%C3%A9-o-ionic-4f8c7b94c51b>. Acesso em: 18 fev. 2020.

PATCHAVA, Vamsikrishna; KANDALA, Hari Babu; BABU, P Ravi. A Smart Home Automation Technique with Raspberry Pi using IoT. In: International Conference on Smart

Sensors and Systems (IC-SSS), 2015, **Centre for Advanced Studies in Electronics Science & Technology**.

PEARLMAN, Shana. **What are APIs and how do APIs work?**. Disponível em: <https://blogs.mulesoft.com/biz/tech-ramblings-biz/what-are-apis-how-do-apis-work/>. Acesso em: 15 fev. 2020.

PONTE, Francisco Rodrigo Parente da. **MECANISMOS DE ATRIBUIÇÃO DE CANAIS DE COMUNICAÇÃO PARA DISPOSITIVOS HETEROGÊNEOS EM SMART HOMES**. 2018. 61p. Programa De Pós-Graduação Em Ciência Da Computação Mestrado Acadêmico Em Ciência Da Computação - Centro De Ciências E Tecnologia - Universidade Estadual Do Ceará.

PYTHON. **What is Python? Executive Summary**. Disponível em: <https://www.python.org/doc/essays/blurb/>. Acesso em: 17 fev. 2020.

PYTHONFOBEGINNERS . **What is Python?**. Disponível em: <https://www.pythonforbeginners.com/learn-python/what-is-python/>. Acesso em: 16 fev. 2020.

RAO, P Bhaskar. RASPBERRY PI HOME AUTOMATION WITH WIRELESS SENSORS USING SMART PHONE. **International Journal of Computer Science and Mobile Computing**, v.4 n.5, maio, 2015, pg. 797-803.

RASPBERRY PI. **BCM2711**. Disponível em: <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/README.md>. Acesso em: 13 fev. 2020.

RASPBERRY PI. **Raspberry Pi Zero**. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-zero/>. Acesso em: 10 fev. 2020.

RASPBERRY PI. **Raspberry Pi Zero W**. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-zero-w/>. Acesso em: 10 fev. 2020.

RASPBERRY PI. **Raspberry Pi 2 Model B** Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>. Acesso em: 08 fev. 2020.

RASPBERRY PI. **Raspberry Pi 3 Model A+**. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-3-model-a-plus/>. Acesso em: 08 fev. 2020.

RASPBERRY PI. **Raspberry Pi 3 Model B**. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>. Acesso em: 11 fev. 2020.

RASPBERRY PI. **Raspberry Pi 3 Model B+**. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>. Acesso em 12 fev. 2020.

RASPBERRY PI. **Raspberry Pi 4 Tech Specs**. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>. Acesso em: 12 fev. 2020.

RASPBERRY PI TUTORIALS. **50 of the most important Raspberry Pi Sensors and Components**. Raspberry Pi Tutorials. Disponível em: <https://tutorials-raspberrypi.com/raspberry-pi-sensors-overview-50-important-components/>. Acesso em: 10 fev. 2020.

RESTFULAPI. **REST API Tutoria**l. Disponível em: <https://restfulapi.net/>. Acesso em: 16 fev. 2020.

ROUSE, Margaret. **URI (Uniform Resource Identifier)**. Disponível em: <https://whatis.techtarget.com/definition/URI-Uniform-Resource-Identifier>. Acesso em: 16 fev. 2020.

SILVA, Matheus Hoffmann. **Sistema de Automação Residencial baseado em Raspberry Pi e Open ZWave**. 2014. 102p. Trabalho de Conclusão de Curso Graduação em Sistemas de Informação - Departamento de Informática e Estatística - Universidade Federal de Santa Catarina.

SOCKET.IO. **Introduction | Socket.io**. Disponível em: <https://socket.io/docs/>. Acesso em: 03 ago. 2020.

SONG, Yuanlin; JIANG, Jinjun; WANG, Xun; YANG, Dawei; BAI, Chunxue. Prospect and application of Internet of Things technology for prevention of SARIs. **Clinical eHealth**, Fevereiro, 2020.

STXNEXT. **What Is Python Used for?**. Disponível em: <https://stxnext.com/what-is-python-used-for/>. Acesso em: 17 fev. 2020.

THOMSEN, Adilson. **Controlando temperatura e pressão com o BMP180**. Disponível em: <https://www.filipeflop.com/blog/temperatura-pressao-bmp180-arduino/>. Acesso em: 20 fev. 2020.

VALIN, Allan. **O que mudará na USB 3.0 em relação a 2.0?**. Disponível em: <https://www.tecmundo.com.br/microsoft/2719-o-que-mudara-na-usb-3-0-em-relacao-a-2-0-.htm>. Acesso em: 12 fev. 2020.