PROJETO FINAL DE GRADUAÇÃO

Proposta de um modelo de IOT para gerenciamento e monitoramento de animais

Bruno Brandão Vieira do Norte

Projeto Final de Graduação em Engenharia de Redes de Comunicação

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

PROJETO FINAL DE GRADUAÇÃO

Proposta de um modelo de IOT para gerenciamento e monitoramento de animais

Bruno Brandão Vieira do Norte

Projeto Final de Graduação submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Bacharel em Engenharia de Redes de Comunicação

Banca Examinadora

Prof. Fábio Lúcio Lopes de Mendonça, Ph.D, FT/UnB
Orientador _____

Prof. Georges Daniel Anvame Nze, Ph.D, FT/UnB
Examinador Interno _____

Prof. Daniel Alves da Silva, Ph.D,
Examinador Externo _____

FICHA CATALOGRÁFICA

BRANDÃO, BRUNO
Proposta de um modelo de IOT para gerenciamento e monitoramentode animais [Distrito Federal] 2022.
xvi, 70 p., 210 x 297 mm (ENE/FT/UnB, Bacharel, Engenharia Elétrica, 2022).
Projeto Final de Graduação - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Animais domésticos	2. Internet das Coisas (IoT)
3. Microcontroladores	4. GPRS
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

BRANDÃO, B. (2022). *Proposta de um modelo de IOT para gerenciamento e monitoramentode animais*. Projeto Final de Graduação, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 70 p.

CESSÃO DE DIREITOS

Bruno Brandão Vieira do Norte
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA E AGRADECIMENTOS

Dedico este trabalho de conclusão de curso às seguintes pessoas:

Ao professor Fábio Lúcio Lopes de Mendonça, pelo apoio e orientação no desenvolvimento desta monografia.

À minha mãe, Maria Rita Brandão de Oliveira, pela presença, apoio, suporte, orientação e amor que foi dado a mim durante toda a minha vida.

À minha irmã, Priscila Brandão Vieira do Norte, por me inspirar e ensinar a sempre buscar novos conhecimentos.

À Julia Eloi Cohen, por estar sempre ao meu lado, em momentos bons e ruins, me acalmando e ajudando a encontrar uma melhor solução para os problemas.

À Isabela Lobato Braga, por todo o suporte e companhia durante os quase 7 anos de universidade, passando por diversos desafios e conquistas juntos.

Aos meus padrinhos por me ajudarem a conseguir ter acesso a uma universidade.

Aos meus familiares que sempre me apoiaram e confiaram em potencial.

E por fim, ao Snape, gatinho de estimação que teve bastante paciência como cobaia neste modelo IoT.

RESUMO

Os amantes dos animais de estimação, em especial os de cães e gatos vem aumentando a cada dia. O Brasil ocupa a 4ª maior nação do mundo em população total de animais de estimação, sendo o 2º do mundo em população de cães, gatos e aves, contendo o 2º maior em faturamento no ramo, com cerca de R$ 34,4 bilhões de em 2018. Consequentemente, surgem os cuidados e as preocupações. Estima-se que mais de 1000 animais domésticos são furtados, perdidos ou abandonados diariamente e, por esse motivo, este trabalho apresenta uma proposta de modelo IoT voltado para o gerenciamento e monitoramento de animais. Serão apresentados nesta monografia os conceitos básicos de funcionamento de um dispositivo IoT, a arquitetura de implementação, desenvolvimento dos elementos de hardware e implementação do aplicativo mobile. A partir disso, portanto, será possível realizar o monitoramento e acompanhamento de animais.

Palavras-chave: Animais domésticos, Internet das Coisas (IoT), Microcontroladores, GPS, Middleware.

ABSTRACT

Pet lovers, especially those of dogs and cats, are increasing every day. Brazil occupies the 4th largest nation in the world in total population of pets, being the 2nd in the world in population of dogs, cats and birds, containing the 2nd largest in revenue in the sector, with about R$ 34.4 billion in 2018. Consequently, care and concerns arise. It is estimated that more than 1000 domestic animals are stolen, lost or abandoned daily and, for this reason, this work presents a proposal for an IoT model aimed at the management and monitoring of animals. This monograph will present the basic concepts of an IoT device, the implementation architecture, development of hardware elements and implementation of the mobile application. From this, therefore, it will be possible to carry out the monitoring and follow-up of animals.

Keywords: Pets, Internet of Things (IoT), Microcontrollers, GPS, Middleware.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

IBGE	Instituto Brasileiro de Geografia e Estatística
IoT	*Internet of Things*
UnB	Universidade de Brasília
IEEE	Institute of Electrical and Electronic Engineers
ISO	Internacional Organization for Standardizatio
ABNT	Associação Brasileira de Normas Técnicas
GPS	*Global Positioning System*
IBSG	*Internet Business Solutions Group*
WSN	*Wireless Sensor Network* (variabilidade do volume sistólico)
RSSF	Rede de Sensores Sem Fio
GSM	*Global System for Mobile Communications*
GPRS	Serviço de Rádio de Pacote Geral
TCP/IP	*Transmission Control Protocol/Internet Protocol*
REST	*Representational State Transfer*
HTTP	*Hypertext Transfer Protocol*
SQL	*Structured Query Language*
UART	*Universal Asynchronous Receiver/Transmitter*
I2C	*Inter-Integrated Circuit*
GPIO	*General Purpose Input/Output*
RTC	*Real-Time Clock*
API	*Application Programming Interface*
UI	*User Interface*
CPU	*Central Processing Unit*
RAM	*Random Access Memory*
AWS	*Amazon Web Services*
EC2	*Elastic Compute Cloud*
RDS	*Relational Database Service*
ULP	*Ultra Low Power*
IAM	*Identity and Access Management*
SSH	*Secure Shell*
CI	Integrações Contínua
CD	Entrega Contínua
OLAP	Online Analytical Processing

1 INTRODUÇÃO

Os animais domésticos estão cada vez mais valorizados nas famílias, fazendo com que muitos tutores tenham mais preocupações com os tutelados. Com a crescente na quantidade de furtos e perdas de animais, verificou-se a necessidade de criar mecanismos que minimizasse esta problemática, de forma que pudesse encontrar alguma forma de monitoramento desses animais, haja visto que o Brasil hoje se destaca como um dos maiores países no ramo da pecuária, com grandes proporções em quantidade de investimento em produtos bovinos, ovinos e suínos.

Tabela 1.1: IBGE - População de animais de estimação no Brasil - 2013 - Em milhões [4]

Efeticvos de animais	**IBGE 2013**
Cães	52,2
Aves	37,9
Gatos	22,1
Peixes	18,0

Tabela 1.2: Número de cabeças, por tipologia, espécie da pecuária e condição do produtor em relação às terras [5]

Efeticvos de animais	**Censo 1995 - 1996**	**Censo 2006**	**Censo 2017**
Bovinos	15.3058.275	176.147.501¨	172.719.164
Bubalinos	834.922	885.119	950.173
Caprinos	6.590.646	7.107.613	8.260.607
Ovinos	13.954.555	14.167.504	13.789.345
Suínos	27.811.244	31.189.351	39.346.192
Aves*	718.538	1.143.458	1.362.254
**galinhas, galos, frangas e frangos (1 000 cabeças)*			

De acordo com as Tabela 1.1 e Tabela 1.2, percebe-se que há uma grande quantidade de animais, tanto domésticos, quanto de pecuária. Além disso, para os animais de pecuária, verifica-se que há uma crescente no valor absoluto, identificando que a cada ano, mais animais estão presentes no solo brasileiro.

Com isso, no processo de criação de animais, seja domésticos ou não, existem algumas problemáticas que devem ser levadas em consideração, que estão listadas a seguir:

- Fugas de animais domésticos;
- Furtos de animais domésticos e animais de pecuária;
- Atividades físicas dos animais; e
- Controle de saúde dos animais.

A partir de um monitoramento destes animais com sensores, estas problemáticas podem ser solucionadas. Um exemplo prático pode ser identificado utilizando um sensor de georreferenciamento. A partir deste sensor, é possível obter as coordenadas geográficas de determinado animal, dentro de um intervalo de tempo. Com isso, diversas métricas podem ser encontradas, como a distância percorrida pelo animal e a região em que o mesmo se encontra. Caso ocorra um cenário de furto, com este sensor de posicionamento também é possível determinar o local onde ele se encontra, possibilitando a recuperação do animal.

Porém, para a implementação de um dispositivo de monitoramento de animais, algumas limitações são encontradas. O primeiro ponto de preocupação é referente à alimentação energética do dispositivo. É necessário que exista um consumo equilibrado de bateria, evitando ao máximo de desperdício energético, visto que para situações emergenciais, o dispositivo deve estar em pleno funcionamento. A segunda limitação que deve ser levada em consideração é o tamanho do dispositivo, visto que o mesmo deve ser criado de modo confortável para o animal, evitando com que seja removido pelo animal devido ao desconforto. Por fim, o dispositivo deve estar conectado a uma rede de telecomunicações, fazendo com que seja possível enviar as informações para um servidor de gerenciamento.

A partir destas problemáticas e limitações, motivou-se a ideia de como evitar com que os animais fossem perdidos ou furtados, conseguindo realizar um melhor monitoramento e acompanhamento dos mesmos. Caso os tutores saibam onde estejam os animais e em quais condições os mesmos se encontram, muitos destes problemas seriam evitados, fazendo com que menos complicações existam na criação dos animais. Com isso, foi criada a proposta de um modelo IoT para o gerenciamento e monitoramento destes animais, tentando associar ao máximo os conceitos de Internet das Coisas com o mundo de Cloud Computing, otimizando, portanto, o fluxo de dados e as informações obtidas pelos dispositivos anexados aos animais.

Dessa forma, a proposta deste trabalho é realizar a elaboração de um modelo de monitoramento através de IoT voltado para o gerenciamento e monitoramento de animais, em específico animais domésticos, de forma que serão apresentados os conceitos básicos de funcionamento de um dispositivo IoT, a arquitetura de implementação, desenvolvimento dos elementos de hardware e implementação do aplicativo mobile possibilitando o monitoramento e acompanhamento destes animais.

Em primeiro momento, será apresentada a arquitetura e como será estruturado a comunicação de rede entre o microcontrolador, *middleware* e usuário. Neste momento, será possível observar de maneira global toda a arquitetura do projeto, orientando-o para os microserviços que serão implementados. Em segundo momento, será possível observar como está estruturado o dispositivo IoT, apresentando componentes e módulos de hardware que serão responsáveis pela obtenção dos dados georreferenciados dos animais.

Já em terceiro momento, será apresentado como está estruturado o ambiente *cloud*, explicando o processo de configuração e criação de um *middleware*, utilizando componente de computação em nuvem. Para o quarto momento, será realizada a configuração da interface do usuário, isto é, o aplicativo mobile que vai utilizar os dados capturados pelo dispositivo IoT.

Por fim, em um quinto momento, será apresentado todo o funcionamento da proposta. Portanto, será possível observar os resultados práticos obtidos a partir do monitoramento de um animal de estimação, recuperando os dados georreferenciados do animal, os enviando para uma persistência de dados *cloud* e apresentando estas informações para o dispositivo celular do usuário tutor do animal.

1.1 OBJETIVO GERAL

Este projeto tem por objetivo geral apresentar uma proposta de uma arquitetura baseada em Internet das Coisas - IoT para realizar o monitoramento e localização de animais domésticos, juntamente com o desenvolvimento de um aplicativo mobile para realizar o monitoramento.

1.2 OBJETIVOS ESPECÍFICOS

Para alcançar este objetivo, os seguintes Objetivos Específicos são propostos: Dentre os objetivos específicos, tem-se:

- Análise detalhada da tecnologia estudada;
- Propor uma arquitetura para definição do modelo;
- Propor um método de monitoração, com tecnologias de baixo custo;
- Desenvolvimento do modelo de aplicação para realizar o monitoramento e
- Simular situações de monitoramento através de uma prova de conceito;

1.3 REVISÃO DA LITERATURA

A base da bibliografia referenciada neste trabalho levou em conta a busca por artigos teses, monografias e livro em diversas fontes, especialmente, a UnB (Universidade de Brasília) e IEEE (Institute of Electrical and Electronic Engineers). Além disso foram realizadas pesquisas em bases de dados dos organismos de normatização ISO (Internacional Organization for Standardization) e ABNT (Associação Brasileira de Normas Técnicas). Foram realizadas pesquisas na base bibliográfica da UnB, que é uma instituição de renome.

A ideia deste projeto tem como base a utilização dos modelos já apresentado nos artigos de (Caio Poli SILVA, Pedro Gustavo Ramm et al) através do artigo [6] e (NEVES, M. C.; PEREZ, NAYLOR BASTIANI), [7] entretanto no trabalho de [6] trata do monitoramento por sistemas de GPS com equipamentos com tecnologias de alto custo e não se preocupa com sistemas de controle de energia, já no artigo [7] ele realiza do controle de gado e não consegue uma localização precisa dos animais.

Apesar das pesquisas dos artigos citados acima apresentarem trabalhos relevantes, nenhum deles aborda a análise proposta neste projeto. No entanto o trabalho proposto pretende-se realizar um sistema de monitoramento através de dispositivos IoT de baixo custo, com precisão e um controle energético dos dispositivo.

1.4 ESTRUTURA DO TRABALHO

Este trabalho será organizado entre os próximos capítulos em quatro partes principais: fundamentação teórica, proposta, análise de resultados e conclusões.

O Capítulo 2 apresenta os principais conceitos tecnologias que compõem este trabalho, bem como apresenta a arquitetura, softwares e metodologia dos estudos conduzidos com o funcionamento detalhado do sistema.

O Capítulo 3 apresenta uma seção tem como finalidade promover uma visão detalhada da estrutura do processo experimental do projeto, de forma que a mesma possa ser reproduzida posteriormente em outros trabalhos, a fim de atender a outras propostas de estudo.

O Capítulo 4 mostra de forma detalhada o desenvolvimento do trabalho e uma análise dos resultados obtidos em cada cenário considerado.

O Capítulo 5 contém a discussão final do estudo e conclui o trabalho, apresentando possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 INTERNET DAS COISAS

A Internet das Coisas emergiu a partir dos avanços de diferentes áreas como sistemas embarcados, microeletrônica, comunicação e sensoriamento. Por se tratar de uma tecnologia emergente, ainda não há um conceito único que possa definir de forma completa o termo Internet das Coisas, no entanto, considerando seus aspectos qualitativos é possível descrever como uma extensão da Internet atual [8], pois permite que objetos que fazem parte do cotidiano possam conectar-se à Internet e se comunicarem entre si, a fim de facilitar e beneficiar as pessoas em suas atividades diárias. Por outro lado, considerando uma abordagem mais objetiva em função de aspectos quantitativos, de acordo com o Cisco Internet Business Solutions Group - IBSG, a IoT começou no momento exato em que foram conectados à Internet mais objetos do que pessoas [9].

Apesar das divergências entre as definições já expostas, o que todas têm em comum é a ideia de que a primeira versão da Internet era sobre dados criados por pessoas, enquanto a próxima versão é sobre dados criados por coisas. Desta forma, a melhor definição para a Internet das Coisas (10) seria:

"Uma rede aberta e abrangente de objetos inteligentes que têm a capacidade de se autoorganizar, compartilhar informações, dados e recursos, reagindo e agindo diante de situações e mudanças no ambiente" [11].

Diante desses conceitos, é possível destacar sua importância em uma sociedade de constante desenvolvimento, pelo fato de cumprir com a função de facilitar o dia-a-dia das pessoas, a IoT possui aplicações em diferentes áreas de atuação, na área médica, de segurança, em redes domésticas e empresariais. Nesse sentido, é possível fazer uma pequena análise segundo o Gartner Hype Cycle [12].

2.2 REDES DE SENSORES

Ultimamente, tem havido inúmeros esforços de pesquisa no campo de rápido crescimento de dispositivos de sensores sem fio, especialmente com o advento de muitos aplicativos e cenários de Internet das Coisas (IoT) que utilizam Redes de Sensores Sem Fio (WSNs) para fornecer uma infraestrutura eficiente de detecção e comunicação. . Uma rede de sensores sem fio (WSN) consiste em nós de sensores sem fio distribuídos ou motes que podem ser usados para detectar vários fenômenos físicos e ambientais. Os nós enviam colaborativamente os dados detectados para uma estação base ou coletor para processamento adicional e tomada de decisão. Exemplos de parâmetros detectados incluem temperatura, pressão, umidade, movimento, nível de líquido, etc. As RSSFs são muito úteis em uma ampla gama de aplicações que incluem monitoramento ambiental, agricultura de precisão, IoT, monitoramento da qualidade da água, rastreamento de animais e assim por diante [13], [14]. Ao longo dos anos, vários nós de sensores sem fio práticos ou motes foram desenvolvidos e introduzidos no mercado para facilitar a implementação de vários

cenários de rede de sensores sem fio Avanços nas tecnologias de nós de sensores sem fio [15] facilitam a pesquisa de ponta e a implementação no mundo real de RSSFs em inúmeras aplicações ao redor do mundo.

Um nó sensor normalmente consiste nos subsistemas de detecção, computação, comunicação e fonte de alimentação. As plataformas de nós sensores sem fio são limitadas em recursos quando se trata de fornecimento de energia, capacidade de computação e armazenamento, etc. Alguns dos objetivos de projetar e desenvolver um nó sensor sem fio incluem operação de energia ultrabaixa, baixo custo por nó, tamanho pequeno e software e hardware reconfiguráveis. O projeto de plataformas de nós sensores sem fio cada vez menores e mais acessíveis de ultrabaixo consumo de energia tornou-se um tópico de pesquisa quente para a comunidade de pesquisa [16].

A vida útil de uma RSSF depende fortemente do fornecimento de energia para alimentar os motes do sensor. Tradicionalmente, os nós sensores são alimentados por pequenas baterias de capacidades limitadas. Em vários cenários de aplicação, um grande número de motes de sensores implantados é necessário para durar sem supervisão por uma vida útil ilimitada em campos remotos e severos. Em tais cenários de implantação, a substituição da bateria se torna um grande desafio. O consumo de energia dos motes impacta diretamente na vida útil das RSSFs. Portanto, abordagens de energia ultrabaixa juntamente com técnicas de captação de energia são críticas para a operação onipresente e perpétua de RSSFs [8]. Devido aos diferentes requisitos de várias aplicações de RSSF, diferentes plataformas de nós sensores sem fio estão sendo desenvolvidas. Por exemplo, alguns cenários de aplicativos de rastreamento e monitoramento de RSSFs exigem o uso de camadas próprias.

2.3 GPS

Um dos principais pilares deste projeto é a geolocalização dos animais. Para isso, é necessário utilizar um módulo GPS e associá-lo à ESPWROOM32. O GPS, Sistema Posicionamento Global, é um sistema de navegação por satélite, responsável por fornecer a localização geográfica de um elemento receptor, calculando, portanto, a latitude, longitude, altitude e velocidade de determinado elemento[17].

O módulo GPS que será utilizado para este protótipo é o UBLOX NEO-6M. Este módulo possui uma antena para a obtenção dos sinais dos dispositivos IoT.

2.4 GPRS

Para a comunicação celular, será utilizado o padrão de Sistema Global para comunicações Móveis, GSM. Este padrão foi implementado nos anos de 1980, tendo o seu maior sucesso após a implementação em solo europeu, nos anos de 1990[18]. A partir disso, o GSM passou por diversas gerações, evoluindo o processo de comunicação e envio de dados. Na primeira geração, 1G, só era possível o tráfego de dados de voz, evoluindo até os sistemas 5G, em que foi viabilizado o tráfego de grandes volumes de informação.

Dentre estas gerações, a que se destaca para este projeto é a 2,5G do GSM. Nesta geração foi introduzido o Serviço de Pacote de Radio Geral, GPRS. A partir disso, o usuário é capaz de utilizar de forma

dinâmica múltiplos canais de rádio para trafegar dados a partir do modelo TCP/IP[18]. Logo, foi possível realizar a integração do sistema de comunicação móvel para trafegar dados na rede Internet, a taxas de até 115Kbps. Portanto, como a rede celular possui grande cobertura global, ela é a melhor escolha para utilização de transferência de dados para o gerenciamento de animais, pois será raro os casos em que o pet se encontrará fora de uma zona de cobertura.

2.5 CLOUD

2.5.1 Middleware

Em sistemas IoT, é necessário uma ponte responsável por conectar os dados, obtidos pelos dispositivos IoT, com os usuários finais que utilizarão estes dados. A infraestrutura responsável por esta interligação é chamada de middleware. O middleware é uma camada de software que abstrai as complexidades dos sistemas locais ou dos hardwares dos dispositivos. Ele funciona entre a camada física (onde estão os dispositivos, sensores e usuários finais) e a camada de aplicação (ambiente de apresentação dos dados) [1]. A Figura 2.1 apresenta a diagramação de camadas de funcionamento da topologia. A partir desta figura, percebe-se que, os dados emitidos pelos elementos da camada física são encapsulados em pacotes e enviados para o middleware. A partir disso, o middleware será responsável por encaminhá-los à camada de aplicação, onde serão processados estes pacotes. Após o processamento, os dados processados serão encapsulados em novos pacotes, enviados ao middleware e, por fim, encaminhados aos elementos da camada física.

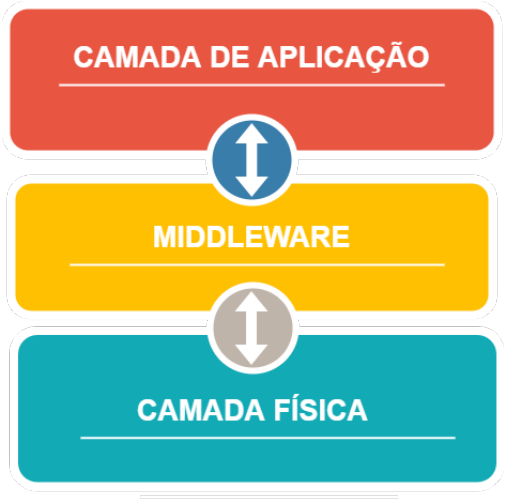

Figura 2.1: Camadas de funcionamento middleware [1]

Olhando novamente a arquitetura tecnológica apresentada na Figura 3.1, percebe-se que o middleware está exatamente no centro da infraestrutura e, por conta disso, é um dos principais serviços para que o sistema IoT funcione corretamente. Para o desenvolvimento do middleware, será implementado uma aplicação em python, utilizando o kit de ferramentas da biblioteca Flask. O middleware implementará o serviço de aplicação REST. Esta aplicação utilizará como base o protocolo HTTP a fim de estruturar a arquitetura REST. O HTTP, *Hypertext Transfer Protocol*, é um protocolo da camada de aplicação, responsável pela

transferência de hypertextos. Com isso, os dados capturados pelos dispositivos IoT serão encapsulados em metadados HTTP, utilizando o método POST e encaminhados para a aplicação REST.

Portanto, o middleware deve conter interfaces de *input* e *output* de dados. Assim, há o envio das informações à camada de aplicação, onde está presente a infraestrutura de banco de dados. O middleware será orientado ao banco de dados[1], atuando como um processador de consultas centralizado, responsável para a implementação das funcionalidades de criação, atualização, leitura e exclusão das informações, abstraindo o processamento dos dados obtidos pelos dispositivo IoT e adicionando-os a uma persistência da informação.

2.5.2 Base de Dados

Por se tratar de um grande volume de informação dos animais, estes dados precisam estar persistidos em um local para associá-los aos usuários, gerando as métricas importantes para o aplicativo. Existem várias maneiras para realizar esta persistência de dados, podendo ser em arquivos de texto, json, implementação de dados estruturados, implementação em dados não estruturados, implementação em dados relacionais e implementação em dados não relacionais.

Cada tipo de persistência é recomendada para determinados tipos de aplicações. Na implementação deste modelo IoT, será utilizado um banco de dados relacional, a partir da tecnologia SQL. Para esta tecnologia, os dados são organizados em tabelas, sendo que as colunas das tabelas são as características ou atribuições do dado, enquanto as linhas são os registros inseridos.

Cada registro desta tabela, possui um sequencial identificador, fazendo com que os mesmos possam ser relacionados à outras tabelas, criando um relacionamento dos dados. Para acessar os registros das tabelas, utiliza-se o SQL, que é uma linguagem de consulta estruturada, criando *queries* que manipulam as informações existentes no banco de dados.

2.6 APLICATIVO MOBILE

Para a apresentação dos dados obtidos pelo o dipositivo IoT, será desenvolvido um aplicativo mobile utilizando o Framework Flutter.

Este aplicativo estará presente nas principais lojas de apps, App Store, para disposivitos que executam o sistema operacional IOS, e Google Play, para dispositivos que executam o sistema operacional Android. Além disso, o aplicativo deverá estar instalado em um dispositivo celular que tenha acesso à internet, podendo ser via 3G, 4G, 801.11 ou outras tecnologias existentes.

O aplicativo se conectará, portanto, ao middleware a partir de requisições HTTP, obtendo os dados dos animais capturados pelo dispositivo IoT.

3 PROCESSO EXPERIMENTAL

3.1 ARQUITETURA TECNOLÓGICA

Para o desenvolvimento deste projeto, como prova de conceitos de seu processo experimental, será necessário a implementação de uma proposta de arquitetura que represente o modelo a ser desenvolvido, essa arquitetura tecnológica é apresentada na Figura 3.1.

A arquitetura está dividida em três sessões: dispositivos, cloud e usuário. A sessão de dispositivos é responsável pela captura de dados a partir dos dispositivos IoT que vão estar associados aos animais. Cada dispositivo contém um microcontrolador de baixo consumo, um módulo de captura de GPS e um módulo de rede GPRS. Além disso, para que cada dispositivo contenha autonomia, será proposto em projetos futuros uma redundância energética que implemente o armazenamento energético em uma bateria, com recarga partir de microplacas fotovoltaicas, obtendo carga a partir da energia solar.

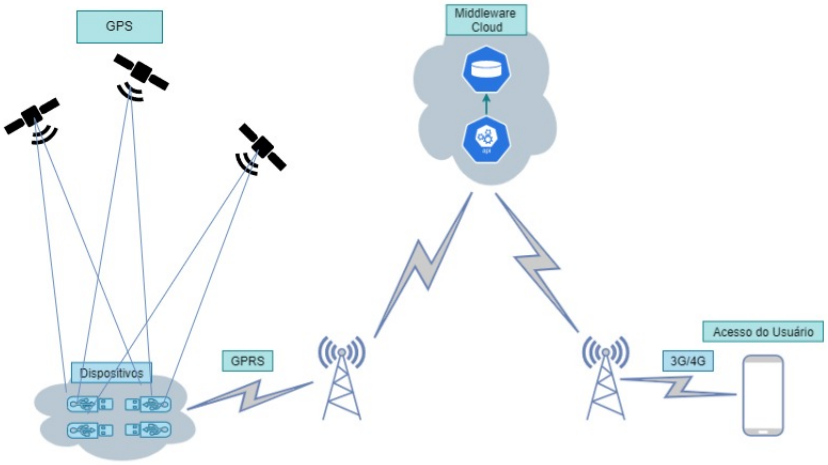

Figura 3.1: Arquitetura Tecnológica.
Fonte: elaborado pelo autor (2022)

A conexão entre a sessão de dispositivos com a sessão de cloud será realizada a partir da rede GPRS. Com isso, cada dispositivo atuará como um hospedeiro móvel, tendo que aplicar conceitos de redes sem fio. Portanto, os dispositivos devem estar em uma região que contenha suporte a rede celular. A sessão de cloud é a responsável pela captura, transformação e armazenamento dos dados capturados pelos dispositivos. Nesta sessão a seguinte infraestrutura é implementada: uma aplicação de serviço REST, uma aplicação dos usuários e uma base de dados SQL.

Por fim, a terceira sessão existente na arquitetura tecnológica é a sessão do usuário. Nesta sessão, será criado um aplicativo mobile que apresentará as informações capturadas pelos dispositivos ao usuário. Logo, o detentor do dispositivo poderá verificar em tempo real a localização do animal, assim como as estatísticas de quanto ele se movimentou e se o mesmo localiza-se dentro de uma zona segura delimitada pelo usuário. Este aplicativo estará disponível para as principais plataformas (Android e IOS), sendo

desenvolvido, portanto, uma aplicação híbrida em Flutter.

3.2 MICROCONTROLADOR DE BAIXO CONSUMO

O microcontrolador utilizado para este projeto é o ESP32. Uma das principais características do ESP32 são os recursos de baixo consumo, sendo ideal para aplicações IoT. Além disso, o ESP32 possui módulos de controle de clock de alta resolução, modos de energia e dimensionamento dinâmico de potência [2]. Por conta disso, é possível minimizar o consumo energético, fazendo com que o dispositivo fique mais tempo executando as funcionalidades propostas.

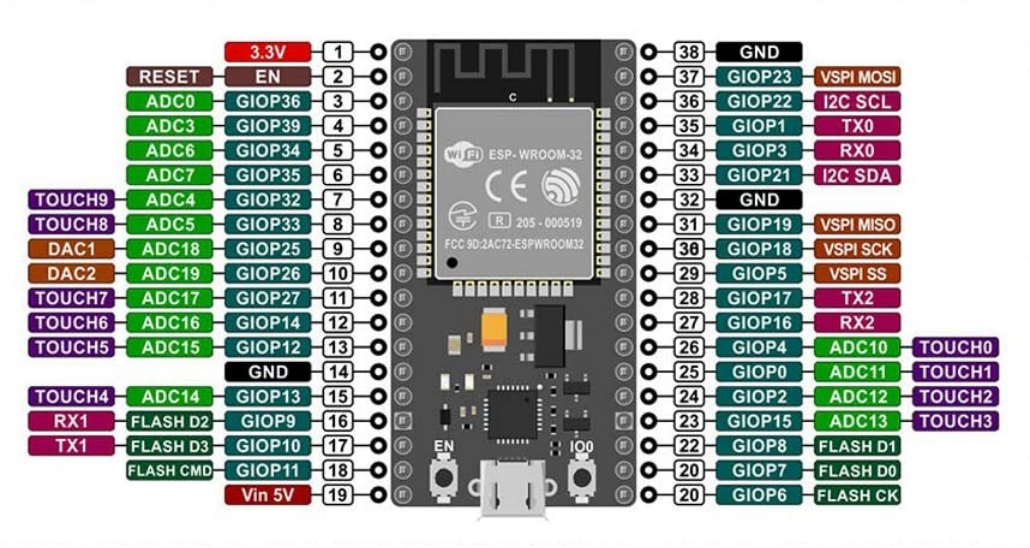

Figura 3.2: Pinagem do ESP-WROOM-32 [2]

A lauchpad utilizada neste projeto para a ESP32 será a ESPWROOM32. Esta launchpad possui diversos componentes internos, como elementos comunicações UART, I2C e GPIO, interfaces WI-FI e Bluetooth, RTC, entre diversos outros módulos. Porém, mesmo que contenha todos estes elementos, nem todos serão utilizados para este projeto. Com a Figura 3.2, é possível observar a pinagem deste dispositivo, determinando quais portas acessam determinados módulos existentes na launchpad. Das portas apresentadas nesta figura, serão utilizadas as portas RX e TX, para a realização de comunicação serial entre o dispositivo e os módulos complementares. Logo, serão utilizadas as portas 16, 17, 27, 28, 35 e 35 da ESPWROOM32.

3.3 FERRAMENTAL UTILIZADO

3.3.1 Python e Flask

Para o desenvolvimento do middleware do dispositivo IoT foi utilizado o kit de ferramentas Flask, disponível para Python. Com o Flask, é possível desenvolver, de maneira rápida e escalável, aplicações que se comunicam via HTTP com um servidor. A partir do Flask, são criadas APIs, que contém rotas de

direcionamento dos serviços. Estes direcionamentos ocorrem a partir do protocolo HTTP executando suas respectivas funcionalidades, como os métodos GET, POST, PUT e DELETE.

O uso de Python [19] como linguagem de programação deve-se a vários motivos, cabendo principalmente citar alguns pontos fortes dessa linguagem que influenciaram na implementação do protótipo de validação deste trabalho.

O primeiro ponto forte é que é uma linguagem altamente portável. Isto quer dizer que o desenvolvimento de eventuais classes ou soluções podem ser migrado para outras plataformas sem grande esforço.

Outro ponto importante é que se trata de uma linguagem de programação dinâmica que pode ser utilizada em diversos tipos de aplicações (*Web Application*, *Win32 App*, redes, aplicações científicas, aplicações móveis, entre outras). Além disso, é uma linguagem de fácil integração com outras linguagens e ferramentas (C++, java, .NET, C, MatLab e Delphi), possuindo uma extensa biblioteca para tais fins.

A linguagem Python foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade da execução, tendo uma sintaxe concisa e clara. Além disso, conta com os recursos poderosos de sua biblioteca padrão e também com módulos e *frameworks* desenvolvidos por terceiros. É assim uma linguagem e ambiente de desenvolvimento bastante adequada para efeito da validação desta tese, por se adaptar à situação com múltiplos módulos de software distribuídos e paralelos que devem atuar de forma coordenada.

A versão utilizada nesta implementação foi a Python 3.7.3 com os módulos ou *plugins* usados no desenvolvimento do protótipo que são descritos nas subseções a seguir.

O Flask [20] é um pequeno *framework web* escrito em Python e baseado na biblioteca WSGI Werkzeug e na biblioteca de Jinja2. Flask está disponível sob os termos da Licença BSD.

Além disso, o Flask tem a flexibilidade da linguagem de programação Python e provê um modelo simples para desenvolvimento web. Uma vez importando no Python, ele pode ser usado para economizar tempo construindo aplicações web. Além disso, ele é chamado de *microframework* porque mantém um núcleo simples, mas extensível. Não provê abstração de banco de dados, validação de formulários, ou qualquer outro componente, mas bibliotecas de terceiros existem para prover as funcionalidade complementares. Assim, o Flask suporta extensões capazes de adicionar tais funcionalidades na aplicação final, o que simplifica o *framework* e torna sua curva de aprendizado mais suave.

3.3.2 Flutter

Flutter é um kit de ferramentas UI do Google open source utilizado para a construção de aplicativos diagramados e compilados nativamente para dispositivos móveis, web, desktop e incorporados a partir de uma única base de código [21]. A linguagem de programação base do framework é o Dart, tornando o desenvolvimento mais fluido e com menos redundâncias. A partir deste kit, é possível desenvolver aplicativos de maneira rápida e eficiente, utilizando componentes nativos de cada sistema operacional, como Android e IOS, sem a necessidade de escrever um código para cada plataforma [22]. Um dos pilares do Flutter é a produtividade, visando um desenvolvimento rápido de aplicações. Uma das principais funcionalidades que incrementa este rápido desenvolvimento é o *hot reload*. Com ela, é possível criar e editar elementos

visuais do aplicativo sem a necessidade de recompilação do projeto, fazendo com que as modificações criadas durante o desenvolvimento sejam apresentadas em tempo real.

3.3.3 Arduino IDE e .ino

Para o desenvolvimento da configuração dos códigos do hardware, será utilizado o software Arduino IDE. Com este software, são desenvolvidos códigos *.ino*, que possuem a sintaxe equivalente à linguagem C. Logo, utilizando a porta serial do computador instalada a partir dos drivers de comunicação serial disponibilizado pelo Arduino IDE, é possível realizar a transferência dos dados que foram programados para a ESPWROOM32.

3.4 DESCRIÇÃO DO HARDWARE UTILIZADO

Para a confecção do dispositivo IoT, foi utilizado o microcontrolador ESPWROOM32. Neste hardware, foi utilizado os modos de operação em baixo consumo para conseguir otimizar o uso de bateria, assim como módulos externos, conectados às portas de transmissão e recepção de dados da ESPWROOM32.

Para a obtenção dos dados georreferenciados, foi utilizado o módulo GPS NEO6M. A partir dele, é possível obter a latitude, longitude, altitude e quantidade de satélites disponíveis para a leitura.

Para a comunicação entre dispositivo e o middleware, foi utilizado o módulo SIM800L. Com este módulo, é possível operar os protocolos de redes sem fio para comunicar o dispositivo com alguma estação-base transceptora. Este mólulo é compatível com GSM, GPRS, 3G, 4G, dentre outros padrões de comunicação.

3.5 DESCRIÇÃO DO AMBIENTE DE SOFTWARE UTILIZADO

Para este projeto, são necessários três ambientes: um para a criação do código que será executado na ESPWROOM32, outro para o middleware e por fim, outro para o banco de dados.

O primeiro ambiente, para a escrita do código executado na ESPWROOM32, foi utilizado um computador Windows 10, com 16GB de RAM e 16 CPUs. A ferramenta utilizada para o desenvolvimento dos códigos foi o Arduino IDE. Vale ressaltar que o computador de desenvolvimento deve conter portas USB que serão utilizadas para a comunicação serial entre o dispositivo e o computador.

Para os ambientes do middleware e banco de dados, foram utilizados serviços cloud da Amazon Web Services - AWS. O primeiro serviço utilizado foi o Amazon Elastic Compute Cloud - EC2. Neste serviço, é contratado máquinas virtuais, podendo escalar as quantidades de CPUs virtuais, memórias RAM e armazenamento a medida em que é requisitado. Neste caso, a máquina contratada utiliza a plataforma Linux/UNIX, Ubuntu Focal 20.04 Server, com 8GB de memória RAM e 4vCPUs. Por ser um processo experimental, a quantidade de requisições ao middleware é baixa, logo, os requisitos estabelecidos são os necessários para o desenvolvimento.

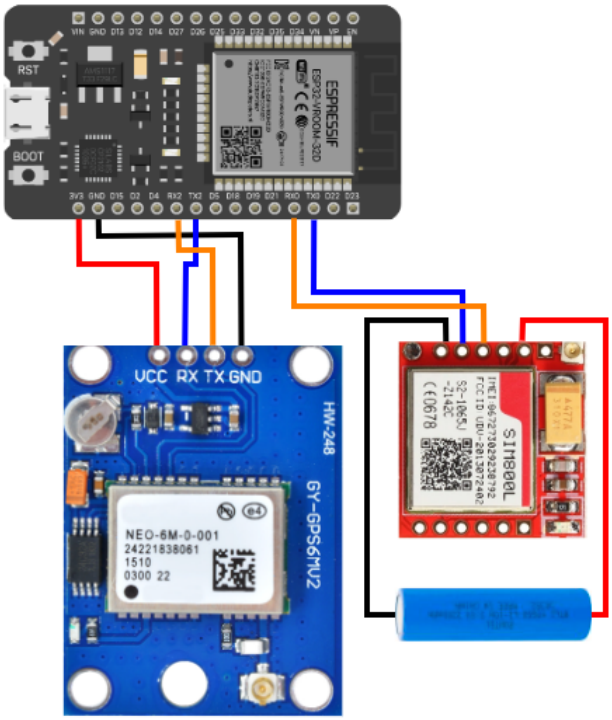

Figura 3.3: Conexão entre os módulos externos e a ESPwroom32.
Fonte: elaborado pelo autor (2022)

Por fim, para o banco de dados foi utilizado o serviço Amazon Relational Database Service - RDS. A partir deste serviço, é possível utilizar máquinas virtuais destinadas para o uso de bancos de dados relacionais. Para este artigo, o banco de dados utiliza a classe *db.t3.micro*, com PostgreSQL, com 1GB de memória RAM, processador Intel Skylake E5 2686 v5 (2.5 GHz) com 2 vCPUS.

Além disso, o servidor em que são executados os serviços tanto do middleware, quanto do banco de dados, são da região sa-east-1a, localizados no estado de São Paulo. A escolha desta região ocorreu para a redução da latência de transferência de dados entre os dispositivos IoT que estarão conectados aos animais, com a persistência de dados.

3.6 CONSTRUÇÃO DO PROTÓTIPO IOT

3.6.1 Proposta de Arquitetura e Camadas do protótipo

O protótipo do dispositivo será dividido em três camadas: a camada de alimentação, a camada de hardware e a camada de módulos. Estas camadas podem ser observadas na Figura 3.4.

Para a camada de alimentação, deverão estar todas a fontes energéticas que alimentarão o dispositivo IoT. Neste cenário, portanto, serão utilizadas duas baterias de lítio de 3.4V. Para projetos futuros, esta camada de alimentação pode ser incrementada, adicionando redundâncias energéticas, como por exemplo, mini painéis fotovoltaicos.

Já a camada de hardware, será responsável pelo cérebro do dispositivo, no caso, a ESPWROOM32.

Todos os códigos programados serão armazenados na memória interna deste microcontrolador, realizando, portanto, as atividades do dispositivo IoT.

Por fim, a camada de módulos será responsável pelos módulos externos ao microcontrolador. Neste caso, serão utilizados os módulos UBLOX NEO-6M, para o GPS, e SIM800L, para o GPRS.

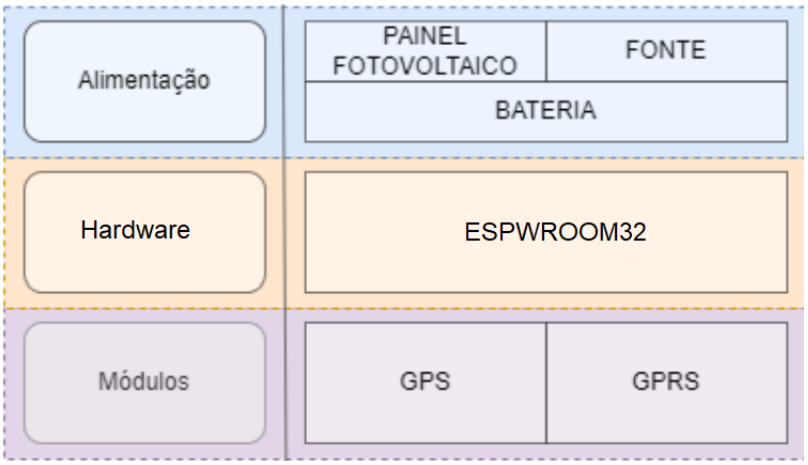

Figura 3.4: Modelo de Arquitetura das Camadas do dispositivo.
Fonte: elaborado pelo autor (2022) adaptado de [1]

3.6.2 Dados a serem recuperados

Neste projeto, serão recuperados e enviados os seguintes dados ao middleware:

- Latitude;
- Longitude;
- Altitude; e
- Nível de Bateria.

Para realizar a captura dos dados georreferenciados, no caso, latitude, longitude e altitude, será utilizado o módulo GPS NEO-6M da empresa U-blox. Já para a determinação do nível da bateria, o valor da tensão resultante será encontrado a partir das portas digitais de GPIO existentes na ESPWROOM32, calculados a partir do valor da tensão de entrada.

3.6.3 Ciclo de vida do dispositivo

A principal preocupação em que deve ser levada em consideração no desenvolvimento do dispositivo é a questão da alimentação energética. Por estar conectado a um animal, o dispositivo deverá funcionar em ambientes remotos e, por conta disso, não poderá ser conectado a uma alimentação energética via cabo. Devido a este fator, o microcontrolador ESP32 será alimentado por uma bateria recarregável. A recarga dessa bateria poderá ocorrer através de mini células fotovoltaicas ou por recarga a partir de uma fonte de

alimentação externa. Para o cenário experimental deste artigo, a recarga da bateria será somente a partir de uma fonte de alimentação.

O nível da bateria é muito importante, pois o dispositivo deve estar disponível em qualquer situação de emergência em que o animal estiver passando, principalmente quando estiver fora da zona de segurança. Para realizar o controle de bateria, o ESP32 utiliza o modo de operação de baixo consumo. Neste modo, o hardware fica em *standby* aguardando um **evento** para realizar uma **ação**. Enquanto estiver em *standby*, todos os módulos (GPS, GPRS entre outros) são "desligados", evitando o uso desnecessário da bateria do dispositivo. O ESPWROOM32 possui as seguintes funcionalidades de baixo consumo: *active*, *modem sleep*, *ligth sleep*, *deep sleep* e *hibernation*.

Cada um destes modos, desativa um ou mais componentes do microcontrolador, sendo que o modo *model sleep* existe menos desativações, enquanto o *hibernation* desativa quase todos os elementos do ESP32. No modo *hibernation*, somente o elemento RTC fica ativado, fazendo com que o microcontrolador "acorde" a partir de determinado tempo definido no código de implementação. Com isso, o consumo energético é muito baixo, fazendo com que a bateria dure por bastante tempo. O cenário consumo energético do *hibernation* é o ideal, porém para este protótipo, será necessária a utilização do módulo GPIO para identificar o nível da bateria. Logo, o modo de baixo consumo utilizado será o *deep sleep*, ou sono profundo. Neste modo, fica ativo tanto o RTC quanto o ULP *Coprocessor*, consumindo apenas $10\mu A$ de corrente.

Com isso, encontra-se outro problema. Em quais momentos o dispositivo IoT sairá do *standby*? A partir desta pergunta, percebe-se o ESP32 contará com alguns estados, que são alternados de maneira cíclica, criando o ciclo de vida do dispositivo IoT. A partir deste ciclo, serão definidos os eventos que alterarão os estados, realizando a ação do dispositivo. Os seguintes estados foram mapeados:

- 000 Standby/ Área não segura/ Baixo nível de bateria;
- 001 Standby/ Área não segura/ Bom nível de bateria;
- 010 Standby/ Área segura/ Baixo nível de bateria;
- 011 Standby/ Área segura/ Bom nível de bateria;
- 100 Ativo/ Área não segura/ Baixo nível de bateria;
- 101 Ativo/ Área não segura/ Bom nível de bateria;
- 110 Ativo/ Área segura/ Baixo nível de bateria;e
- 111 Ativo/Área segura/ Bom nível de bateria.

Com isso, percebe-se três variáveis importantes: estado de espera, zona de segurança e nível de bateria, chamados de S, Z e B, respectivamente. Cada estado apresentado, pode ser representado por uma sequência de três bits, identificando as variáveis existentes entre eles. O bit mais significativo representa S, enquanto o menos significativo representa B. Logo:

- 000 = Não está ativo, não está na zona segura e não possui bom nível de bateria;

- ...
- 111 = Está ativo, está na zona segura e possui um bom nível de bateria.

A partir desta codificação, é possível montar a máquina de estados, implementando o ciclo de vida do dispositivo IoT. A Figura 3.5 apresenta o diagrama de transição de estados do dispositivo. Nela, pode-se perceber alguns fatores. Sempre quando ocorre alguma transição, é alterado somente o bit de uma única variável. Isto é, ou será alterado o bit do modo de espera, ou da zona ou do nível de bateria. Em nenhum momento há transição de dois bits simultaneamente. Além disso, quando o dispositivo estiver em algum estado de espera, ele sempre será alterado para algum estado não-espera, contendo as mesmas variáveis de zona e bateria. Assim, a partir do estado 000 só poderá ir para o estado 100, já para do 001 só poderá transicionar para 101 e assim sucessivamente.

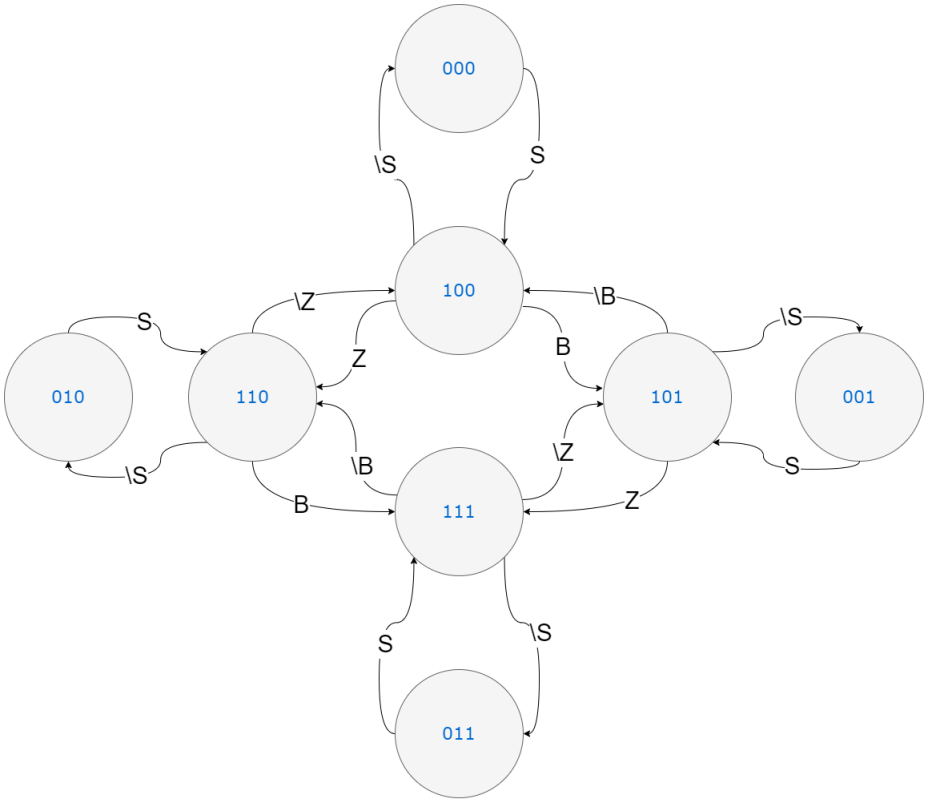

Figura 3.5: Diagrama de transição de estados do dispositivo IoT.
Fonte: elaborado pelo autor (2022)

Para realizar as transições de estados e realização das ações, será necessário implementar funções auxiliares na codificação do dispositivo IoT. Dentre estas funcionalidades estão presentes:

- getBateryLevel();
- setBateryLevel();
- getZone();
- setZone();

- setStandBy();
- getStandBy();
- getState();e
- setState().

Na assinatura destes métodos, existem os prefixos **get** e **set**. Quando um método é iniciado por **get**, significa que ele está recuperando uma informação. Caso ele seja iniciado com **set**, significa que o mesmo estará adicionando alguma informação em determinada variável. Nesta situação, poderão ser recuperados e instanciados o nível da bateria (*baterryLevel*), a zona (*zone*), o estado de espera (*standby*) e o estado (*state*).

O valor da **Safe Zone** será definido pelo usuário e instanciado na variável global **CUT_SAFE_ZONE**. O valor instanciado nesta variável é o raio de segurança do animal, em metros. Por conta disso, este valor é o corte para definir se o animal está ou não na *safe zone*. Caso o animal esteja dentro do raio da safe zone, o bit de estado Z será alterado para 1. Caso contrário, o animal estará fora da zona de segurança, alterando o bit de estado Z para 0. Para a recuperação da zona, utilizará o método **getZone()**.

O nível da bateria poderá conter dois valores: bateria baixa e bateria cheia. O corte para definição destes valores é o nível a partir de **30%**. Isto é, quando a bateria estiver abaixo de 30%, o dispositivo apresentará bateria baixa, alterando o bit de bateria B para 0. A variável global de definição do nível de bateria é chamada de **CUT_BATERY_LEVEL**, instanciado, por padrão, o inteiro 30. Caso contrário, o dispositivo estará com bateria cheia, alterando o bit de estado B para 1. Para a recuperação do nível de bateria, será utilizado o método **getBateryLevel()**.

Por fim, para a mudança de estados, será utilizado o método **setState()**. Este método inicialmente verifica o estado atual da máquina utilizando getState(). Com ele, é possível recuperar os valores das variáveis globais S, Z e B e alterar o estado da máquina. Portanto, para cada caso de estado, serão executadas as devidas funcionalidades. O pseudocódigo de setState() está apresentado no Código 1.

Código 1

```
void setState(){
    unsigned short STATE = getState();
    //Definição do estado
    STATE |= S;
    STATE = STATE << 1;
    STATE |= Z;
    STATE = STATE << 1;
    STATE |= B;

    switch(STATE){
        case 0b000:{
            state000();
```

```
            break;
        }
        case 0b001:{
            state001();
            break;
        }
        ...
        default:{
         println("Estado inexistente.");
            break;
        }
    }
}
```

3.6.3.1 Configuração do GPS

Para realizar a configuração do Módulo GPS NEO-6M, é necessário inicialmente utilizar as bibliotecas **TinyGPS++.h** e **HardwareSerial.h**. A biblioteca TinyGPS++ é responsável pela configuração do módulo GPS, possibilitando, a partir dela, obter as informações capturadas pelo sensor. A partir dos métodos existentes nesta biblioteca, é possível obter com facilidade os atributos de latitude, longitude, altitude e quantidade de satélites capturados. Já a biblioteca HardwareSerial é responsável pela configuração da conexão serial existente entre o microcontrolador ESPWROOM32 com o GPS NEO-6M.

Com isso, deve-se realizar a importação destas bibliotecas, como apresentado no Código 2.

Código 2

```
#include <HardwareSerial.h>
#include <TinyGPS++.h>
```

Em seguida, é necessário realizar a configuração dos pinos do microcontrolador que serão utilizados para a realização da comunicação serial. De acordo com o diagrama de pinagem apresentado na Figura 3.2, para a utilização da comunicação serial 1 **TX1/RX1**, é necessário utilizar os pinos 16 e 17. Para organizar o código, portanto, foram criadas duas variáveis globais, GPS_RX_PIN e GPS_TX_PIN. Logo, para GPS_RX_PIN, o valor para ser configurado é o pino 16, enquanto para GPS_TX_PIN, é o pino 17. Já a variável para a comunicação serial, chamada de GPS_SERIAL_NUM, é configurada com o valor 1. O Código 3 apresenta a etapa de configuração das variáveis globais.

Código 3

```
#define GPS_RX_PIN 16

#define GPS_TX_PIN 17
```

```
#define GPS_SERIAL_NUM 1
```

O próximo passo é a inicialização dos componentes do GPS. Para isso, é necessário definir o objeto **HardwareSerial**, inicializando de acordo com a porta serial que será utilizada, no caso, o valor de GPS_SERIAL_NUM. Além disso, é necessário definir o objeto de gerenciamento do GPS **TinyGPSPlus**. O Código 4 apresenta esta inicialização de variáveis.

Código 4

```
HardwareSerial GPSSerial(GPS_SERIAL_NUM);
TinyGPSPlus gps;
```

A etapa seguinte é inicializar a comunicação serial entre o GPS e o ESP32. Nesta inicialização, será utilizada a frequência de 9600Hz para a comunicação. Além disso, como apresentado anteriormente, serão utilizadas as portas definidas em GPS_RX_PIN e GPS_TX_PIN. O Código 5 indica esta configuração.

Código 5

```
GPSSerial.begin(9600, SERIAL_8N1, GPS_RX_PIN, GPS_TX_PIN);
```

Por fim, a última etapa de configuração e implementação do módulo GPS é realizar a captura dos dados do sensor. Para isso, será definido um método chamado **getGPS**, que terá como retorno um objeto do tipo PackageData. Este objeto armazena todas as informações que serão enviadas pelo pacote HTTP, como a URL do middleware, dados georreferenciados e dados do dispositivo IoT.

Neste método, primeiramente deve-se verificar se a comunicação serial está disponível para uso. Caso esteja, o GPS começará a escrever dados em **GPSSerial**. Em seguida, será verificado se a comunicação GPSSerial está disponível. Caso esteja disponível, o código recuperará o que foi escrito anteriormente, sendo salvo em uma variável chamada **c**. Esta variável contém os dados brutos do sensor, porém, a biblioteca TinyGPSPlus contém o método encode. A partir desse método, é possível acessar esses valores brutos, pois eles foram codificados para serem acessados pelo objeto **TinyGPSPlus gps**, que foi definido anteriormente. Com isso, basta realizar a recuperação dos atributos de **gps**, armazenando as informações obtidas no objeto PackageData, a partir do método setPackageData(PackageData P). Por fim, ao realizar a recuperação dos dados georreferenciados, é necessário ativar o módulo de baixo consumo para reduzir o consumo energético, visto que os dados georreferenciados só serão utilizados novamente após o tempo definido no ciclo de vida do dispositivo IoT. Para isso, é chamado o método **esp_deep_sleep**, fazendo que o dispositivo seja "acordado" novamente depois do tempo definido em TIME_TO_SLEEP. O método **getGPS()** está apresentado no Código 6.

Código 6

```
PackageData getGPS() {
  PackageData data;
  if (Serial.available()) {
    char c = Serial.read();
```

```
    GPSSerial.write(c);
  }
  if (GPSSerial.available()) {
    char c = GPSSerial.read();
    gps.encode(c);
    if (gps.location.isUpdated()) {
      DynamicJsonDocument doc(2048);

      setPackageData(
        &data,
        gps.location.lat(),
        gps.location.lng(),
        gps.altitude.meters(),
        MAC_ADDRESS,
        getBattery(),
        getSafeZone()
      );
    }
  }
  esp_deep_sleep(TIME_TO_SLEEP * uS_TO_S_FACTOR);
  return data;
}
```

3.6.3.2 Configuração do GPRS

O segundo módulo a ser configurado é o SIM800L. Para este módulo, será necessário a importação da biblioteca **TinyGsmClient.h**. Esta biblioteca é responsável pela configuração e acesso aos recursos existentes para o módulo SIM800L. A partir dela é possível definir um modem GSM, que será responsável pelo envio e recebimento de pacotes entre o dispositivo IoT e o middleware. O Código 7 apresenta a importação desta biblioteca.

Código 7

```
#include <TinyGsmClient.h>
```

Para funcionamento do módulo SIM800L, é necessário um chip de telefone com plano de dados. Com isso, realiza-se a configuração de um Access Point Name - APN, da rede telefônica por onde os dados serão trafegados.

Cada operadora móvel possui suas credenciais de autenticação de APN. Para isso, é necessário indicar qual o *gateway* que será utilizado, o usuário e a senha. Neste contexto, portanto, três variáveis são definidas,

APN, **USER** e **PASSWORD**. No código 8 ocorre a instanciação das variáveis de configuração da APN.

Código 8

```
const char* APN = "<gateway_apn>";
const char* USER = "<username_apn>";
const char* PASSWORD = "<passwork_apn>";
```

Além disso, é necessário realizar a inicialização das portas seriais do ESPWROOM32 para associá-las ao SIM800L. Portanto, define-se **GSM_TX_PIN** para a porta de transmissão de dados e **GSM_RX_PIN** para a porta de recepção de dados. De acordo com Figura 3.2, as portas utilizadas para TX e RX são, respectivamente, 28 e 27. Logo, são definidas as seguintes variáveis globais no código 9.

Código 9

```
#define GSM_TX_PIN 28
#define GSM_RX_PIN 27
```

Em seguida, é necessário realizar a inicialização do modem GSM. Utilizando, portanto, a classe TinyGsm, existente na biblioteca **TinyGsmClient.h**, é possível realizar a instanciação do modem GSM. O Código 10 apresenta a definição deste modem.

Código 10

```
HardwareSerial GSMSerial(GSM_SERIAL_NUM);
TinyGsm modemGSM(GSMSerial);
TinyGsmClient client(modemGSM);
HttpClient http(client,serverName,port);
```

Assim, deve-se realizar a inicialização da comunicação serial. Logo, com o GSMSerial, é utilizada a frequência de 9600Hz para a comunicação. Além disso, os pinos de GSM_TX_PIN e GSM_RX_PIN devem ser configurados nesta inicialização. Com o código 11 é possível observar esta configuração.

Código 11

```
GSMSerial.begin(9600, SERIAL_N1, GSM_RX_PIN, GSM_TX_PIN);
```

A última etapa é a realização da configuração do modem GSM. Será definido, portanto, um método chamado *gsmConfiguration*, que será executado no *setup* do código. A primeira etapa deste método é verificar se o modem está sendo reiniciado ou não. Caso ele não esteja, é solicitada a reinicialização para executar o reset das configurações. Logo em seguida, será verificado se existe alguma conexão com a rede celular. Por fim, caso exista a conexão, será realizada a configuração com a rede GPRS, associando à APN ao usuário e à senha da conexão. O Código 12 apresenta esta configuração.

Código 12

```
void gsmConfiguration() {
```

```
  if (!modemGSM.restart())
  {
    Serial.println("Reinicializando GSM");
    ESP.restart();
    return;
  }
  Serial.println("Modem restart OK");

  if (!modemGSM.waitForNetwork())
  {
    Serial.println("Erro de conexão na rede.");
    return;
  }
  Serial.println("Modem network OK");

  if (!modemGSM.gprsConnect(APN, USER, PASSWORD))
  {
    Serial.println("Falha na conexão GPRS");
    return;
  }
}
```

3.6.3.3 Envio dos dados para o middleware

A última etapa a ser configurada no dispositivo é o envio dos dados coletados para o middleware. Para isso, é necessário realizar a configuração do envio do pacote HTTP ao servidor. O código 13 apresenta esta implementação.

Código 13

```
void sendData(PackageData json) {
  String serverPath = "/api/evento/";
  String contentType =  "application/json";
  String body = transformPackageDataToJson(json);
  http.post(serverPath, contentType, body);
  int statusCode = http.responseStatusCode();
  String response = http.responseBody();
  Serial.print("Status code: ");
  Serial.println(statusCode);
  Serial.print("Response: ");
  Serial.println(response);
  http.stop();
```

}

O método sendData(PabackageData json) definido no Código 13, em primeiro momento, define a variável serverPath. Esta variável é responsável por associar qual rota do middleware que será utilizada para a consulta HTTP. Ao acionar esta rota, os dados que serão enviados e processados pelo middleware, finalizarão a comunicação. Logo em seguida, é definido o *contentType* do pacote http. Neste caso, como será enviado uma informação em json o tipo de conteúdo a ser definido é o **applcation/json**.

Já o método transformPackageDataToJson() é um método auxiliar que transforma o objeto Package-Data, definido anteriormente, em uma *string* no formato *json*. Com isso, a partir dela, é possível realizar a inserção dos dados coletados ao corpo do pacote HTTP. Logo, com todas estas informações, é possível realizar a comunicação com o servidor, enviando os dados coletados ao middleware.

Por fim, os métodos seguintes, responseStatusCode() e responseBody() indicam, respectivamente, o status da resposta da requisição enviada pelo servidor e o conteúdo da resposta. A partir disso, portanto, é possível verificar se a comunicação com o servidor ocorreu corretamente ou não.

3.7 CONSTRUÇÃO DO MIDDLEWARE

Para a construção do middleware, foi utilizada a linguagem de programação Python, em conjunto com o *framework* Flask. Este *framework* é bastante modularizado e por conta disso, é possível desenvolver pequenas e grandes aplicações escaláveis. Para auxiliar a implementação de serviços web e REST, junto ao Flask, foi adicionado ao projeto uma interface Swagger, podendo realizar testes e criar pacotes HTTP para a ferramenta. A Figura 3.6 apresenta a interface do middleware do dispositivo IoT.

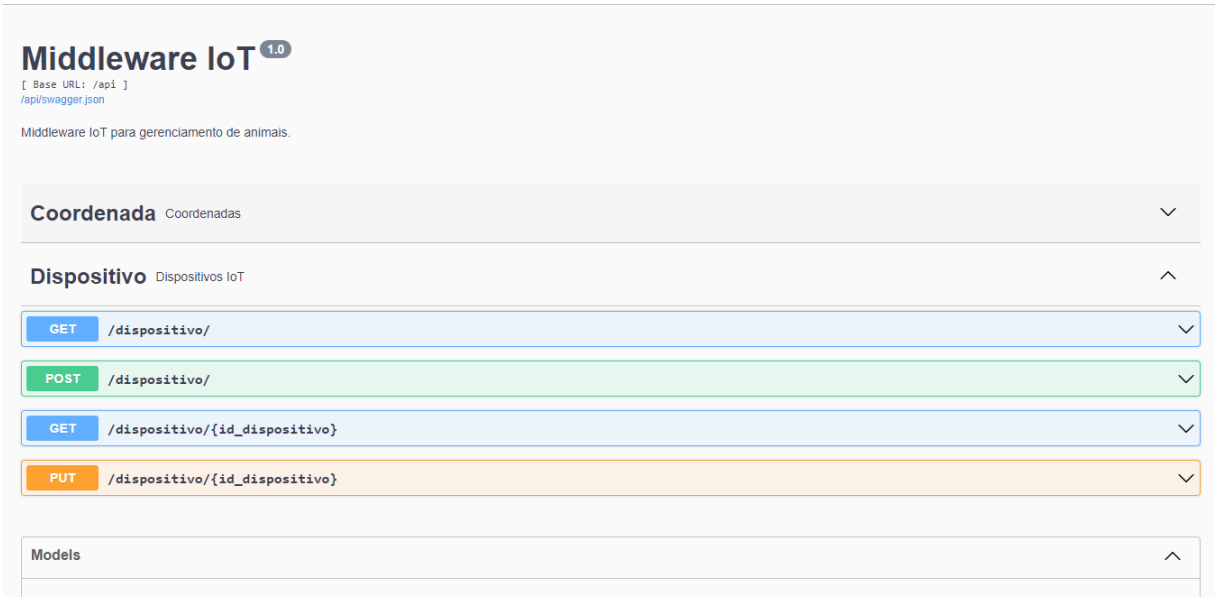

Figura 3.6: Página inicial do Middleware.
Fonte: elaborado pelo autor (2022)

A codificação do middleware está organizada da seguinte maneira:

- Modelos
- Controladores
- Rotas

Os **modelos** são os objetos que serão inseridos ao banco de dados. Estes objetos podem ser os dispositivos, animais, tutores, coordenadas e até mesmo a zona de segurança do animal. Estes modelos são todos os elementos que geram entradas de dados para a aplicação IoT.

Já os **controladores**, são os elementos responsáveis pela conexão entre o middleware e o banco de dados. Este controlador recuperá os dados de entrada e os encaminhará para serem inseridos no banco de dados. Nesta etapa, podem ocorrer alguns processamentos e otimizações de dados.

Por fim, as **rotas** representam o caminho de entrada de dados no middleware. A partir das configurações de rotas, os usuários podem encontrar o endereço do middleware a partir do seu IP público ou do DNS e, com isso, conseguir realizar as solicitações HTTP.

Além disso, algumas entidades foram criadas para o funcionamento integral da ferramenta. Dentre estas entidades, estão:

- Coordenada
- Dispositivo
- Evento
- Pet
- Usuário
- Zona de Segurança

Para o desenvolvimento, cada entidade possui seus próprios modelos, controladores e rotas, fazendo com que tenha um padrão de desenvolvimento para o middleware. Cada entidade, possui dois arquivos-chave, que engloba os três componentes de modelos, controladores e rotas. O primeiro arquivo é o **<nome_da_entidade>_controller.py**. Este arquivo possui o mapeamento da rota, definições dos atributos de modelo e os métodos responsáveis pelo controle de atividades da entidade. Já o segundo arquivo, é o **<nome_da_entidade>_db.py**. Este arquivo é responsável por fazer as ações de comunicação com o banco de dados, podendo inserir, editar, excluir ou atualizar novos registros.

O primeiro arquivo, **<nome_da_entidade>_controller.py**, possui uma estrutura padrão. Inicialmente, é necessário realizar as importações das bibliotecas python que serão utilizadas por esta entidade. Cada entidade utilizará, no mínimo, as seguintes bibliotecas:

- flask, utilizando o componente request;
- flask_restx, utilizando os componentes Resource, Namespace, fields; e

- <nome_da_entidade_db>, utilizando o componente CoordenadaDb.

A primeira biblioteca, flask, é a responsável pelo código do framework flask que estamos utilizando. A partir dela, é possível criar a ferramenta REST, possibilitando executar requisições HTTP entre um cliente e um servidor.

Já a segunda biblioteca, Flask-RESTX, é uma extensão para Flask que adiciona suporte para a construção rápida de APIs REST[23]. Além disso, ela possui componentes que auxiliam bastante na implementação de uma documentação da API, possibilitando também na integração com o Swagger, uma ferramenta com suporte a documentações de API de grande escala.

Por fim, a terceira biblioteca de importação, é o <nome_da_entidade_db>. Esta biblioteca é a responsável pela comunicação do middlweare com o banco de dados, implementado no segundo arquivo-chave da entidade.

Com isso, as três primeiras linhas deste código devem ser as importações destas bibliotecas, apresentadas na sessão de Código 14.

Código 14

```
from flask_restx import Resource, Namespace, fields
from flask import request
from app.main.coordenada.nome_da_entidade_db import NomeDaEntidade
```

Em seguida, é necessário realizar a criação de um namespace. Um namespace é um objeto responsável pelo armazenamento de contexto de uma entidade. Com isso, a partir dele, é possível realizar as principais configurações da rota e de modelos que serão implementados. Logo, com ele, aplica-se o nome e descrição da rota, definição dos atributos de modelo e métodos que serão utilizados pelo controlador. Para iniciar um namespace, basta realizar a implementação apresentada no Código 15.

Código 15

```
api = Namespace('NomeDaEntidade', description='Descricao da Entidade')
```

Neste código, foi instanciada à variável api o objeto Namespace, contendo como atributos "Nome da Entidade" e "Descrição da entidade". A partir disso, todos os novos valores configuráveis, como os modelos e os métodos dos controladores e rota, serão associados a esta variável **api**.

Para configurar os atributos de modelo, por exemplo, basta chamar a variável api e utilizar o método existente ao Namespace, chamado de model. Este método recebe como atributo de entrada um objeto json e, a partir deste json, todos os atributos e tipagens existentes são transformados nos elementos da entidade que será documentada no middleware. O Código 16 é possível observar o processo de criação do modelo do objeto CoordenadaModel.

Código 16

```
modelo = api.model('CoordenadaModel', {
```

```
    'id_coordenada': fields.Integer,
    'vl_longitude': fields.String,
    'vl_latitude': fields.String,
    'vl_altitude': fields.String,
    'dta_create': fields.DateTime,
    'dta_update': fields.DateTime
})
```

Por fim, a última etapa para configurar o arquivo-chave controlador é a definição das rotas e quais funcionalidades serão executadas ao ser chamada a rota. Para isso, cada atribuição de rota deverá ser associado a uma classe e esta classe deverá conter os métodos http a serem executados. Portanto, utilizando um decorador python, é possível utilizar esta funcionalidade codificada. O método a ser "decorado" à classe é o **route**. Este método recebe como atributo a rota que vai ser utilizada pelo middleware para conseguir executar o método existente na classe. O Código 17 apresenta o procedimento para executar esta decoração, para o caso de CoordenadaController.

Código 17

```
@api.route('/')
class CoordenadaController(Resource):
    @api.response(200, "Busca realizada com sucesso")
    def get(self):
        return CoordenadaDb.get(), 200

    @api.expect(modelo)
    def post(self):
        return CoordenadaDb.insert(request.json), 201
```

Para realizar uma decoração, basta utilizar o parâmetro '@', como apresentado na linha 1 do Código 17. Com isso, o método `@api.route('/')` é associado à classe `CoordenadaController`. Ao verificar esta classe, é possível identificar que existem dois métodos implementados, o método get e o método post. A funcionalidade get, é responsável por recuperar todas as coordenadas do banco de dados, portanto, ao utilizar o método GET para realizar requisição na url *http://ip_da_api:porta/api/coordenada/*, todas as coordenadas serão retornadas na requisição realizada, com o código 200.

Já para a funcionalidade post, está 'decorado' que é esperado um objeto de modelo ao método HTTP, que foi definido anteriormente. Com isso, ao realizar uma requisição POST para a mesma URL, ao passar o modelo no corpo da requisição, este objeto é inserido no banco de dados. Caso tudo ocorra com sucesso, a requisição HTTP retornará o código 201. Para criar novas rotas para uma entidade, basta criar uma nova anotação `@api.route(path)` associada a uma nova classe. O arquivo completo está presente nos anexos deste artigo.

A próxima etapa, portanto, é a definição do arquivo-chave **<nome_da_entidade>_db.py**. Para este arquivo, será definida uma classe chamada de nomeDaEntidadeDb. Os métodos existentes nesta classe são os responsáveis por realizar o Create, Read, Update, Delete - CRUD, com o banco de dados. Estes

métodos seguem um padrão, independentemente se serão para inserção ou leitura de dados. Inicialmente, deve-se criar a conexão com o banco de dados, definindo a variável *conn*. Com a conexão iniciada, deve-se recuperar o cursor do banco de dados. O cursor é um indicativo de qual apontamento de memória de conexão está sendo executado no banco de dados.

A partir deste cursor, portanto, é possível realizar as consultas com o banco de dados, utilizando o método **cursor.execute(query)**. Com isso, o resultado deste cursor pode ser tratado e apresentado para obter o retorno do middleware. Por fim, deve-se realizar um commit no cursor, caso seja realizada alguma inserção ou atualização e encerrar a conexão com banco de dados. O Código 18 apresenta um exemplo de um método existente neste arquivo-chave. Neste código, é possível verificar a configuração de inserção de um novo dispositivo no banco de dados, na tabela tb_dispositivos.

Código 18

```
import json
from app.conf.config import Config
from datetime import datetime as dt


class DispositivoDb:

    @classmethod
    def insert(item):
        conn = Config.config()
        if conn != None:
            try:
                cur = conn.cursor()
                id = cur.execute('''
                    INSERT INTO tb_dispositivos (cd_mac_address)
                    VALUES (%s)
                    RETURNING id_dispositivo
                ''', (item['cd_mac_address'],))

                id = cur.fetchone()[0]
                conn.commit()
                cur.close()
                return json.dumps(id, default=str)
            except Exception as ex:
                print(ex)
                conn.rollback()
```

3.7.1 Modelagem conceitual do Banco de Dados

Para a realização da modelagem conceitual do banco de dados, foi utilizado o software SQL Power Architect. Este software possui as ferramentas suficientes para gerar tanto a modelagem conceitual, quanto modelagem física das informações.

A partir dela é possível criar as tabelas, criar chaves, relacionamentos e códigos SQL, proporcionando uma melhor documentação e manutenção do banco de dados.

Além disso, como apresentado anteriormente, será utilizado o serviço cloud RDS da Amazon AWS, utilizando como software PostgreSQL. Logo, será desenvolvido um banco de dados relacional, responsável pela carga de todos os dados coletados dos animais, dados dos usuários e dados de zonas de segurança do animal.

A Figura 3.7 apresenta a modelagem conceitual do banco de dados relacional que será utilizado para esta aplicação IoT.

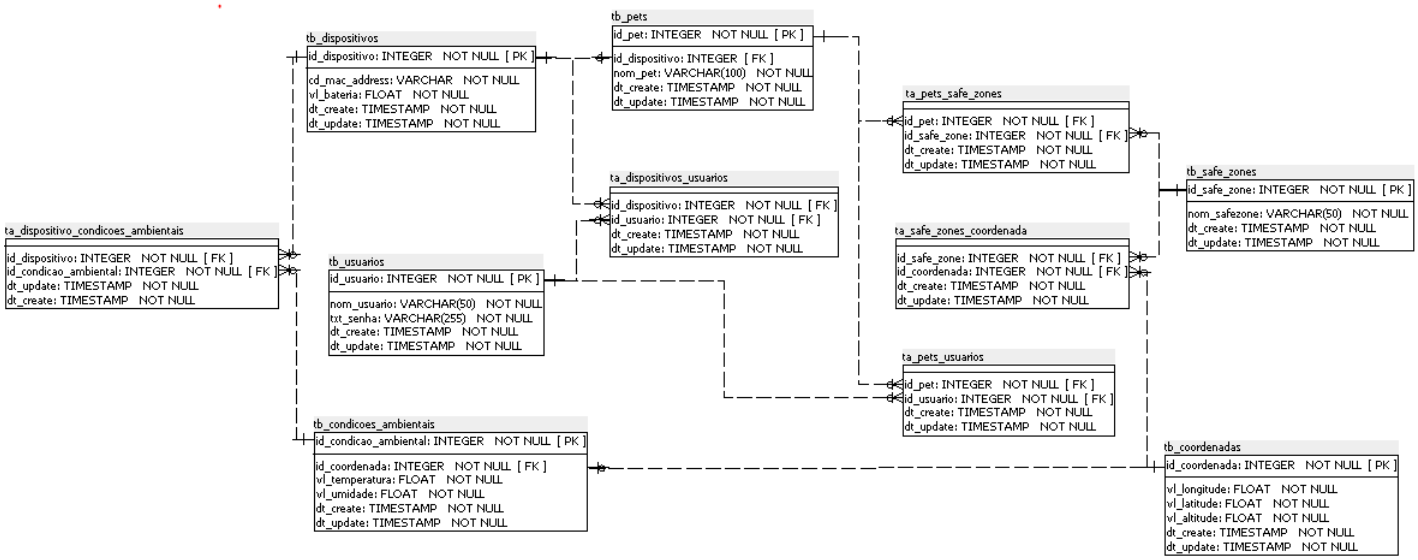

Figura 3.7: Modelagem conceitual para a aplicação.
Fonte: elaborado pelo autor (2022)

3.7.1.1 Tabelas do banco de dados

Para o banco de dados, as tabelas serão dividias em tabelas de entidade e tabelas associativas. As tabelas de dados apresentam as características de cada objeto cadastrado. Já as tabelas associativas, apresentam a associação um para “n” (1:n) entre os objetos.

As seguintes tabelas de entidade de dados compõem o banco de dados relacional:

- tb_pets
- tb_usuarios

- tb_safe_zones
- tb_condicoes_ambientais
- tb_dispositivos
- tb_coordenadas

Já as tabelas associativas existentes são:

- ta_pets_usuarios
- ta_pets_safe_zones
- ta_safe_zones_coordenada
- ta_dispositivos_usuarios

3.8 IMPLEMENTAÇÃO DOS SERVIÇOS EM CLOUD

Para realizar a hospedagem dos serviços utilizados neste protótipo, foram empregados elementos de infraestrutura *cloud* da Amazon AWS. Para o banco de dados, utilizou-se o RDS, enquanto para o middleware, utilizou-se o EC2. Para a realização desta arquitetura cloud, é necessária a criação de uma conta Identity and Access Management - IAM - na Amazon AWS. Com esta conta, é possível utilizar os serviços a nível de gestor. Logo, é possível autenticar, gerenciar e instanciar qualquer serviço provido pela Amazon, no caso, o RDS e EC2. Além disso, é possível também realizar as operações do usuário IAM utilizando uma camada superior de autenticação a partir do usuário root.

Para este protótipo IoT, por utilizar um ambiente *cloud*, é extremamente importante a melhor escolha para a localização do datacenter. Existem diversas possibilidades de escolha, em todos os continentes. Porém, para cada tipo de datacenter, podem ser ocasionados preços diferentes na implementação. Por exemplo, no ano de 2022, o uso de um serviço de um datacenter localizado em São Paulo possui um custo operacional superior do que um datacenter localizado em Oregon, nos Estados Unidos. Porém, existe uma relação de *trade-off* entre a localização e o custo operacional.

Na situação deste protótipo, por lidar com informações em tempo real, é de extrema importância os tempos de requisições aos serviços, pois podem impactar diretamente na localização do animal doméstico. Com isso, um servidor localizado no Brasil é a melhor escolha para este cenário, pois o protótipo inicialmente é voltado para animais do Brasil. Portanto, o datacenter em que será implementado os serviços, por mais que contenha um custo operacional superior, é o localizado em São Paulo, denominado de *sa-east-1*.

3.8.1 Relational Database Service

O banco de dados modelado na Figura 3.7, será implementado em um PostgreSQL do RDS. Para realizar isso, inicialmente deve-se acessar o portal da Amazon AWS e selecionar o serviço de RDS Management

Console. Com isso, é possível iniciar diversas instâncias de bancos de dados, como SQL Server, Oracle Database, MySQL, PostgreSQL, entre outros. Neste caso, por ser uma ferramenta open source, escolheu-se o PostgreSQL.

Existem duas maneiras de iniciar uma instância de banco de dados no RDS Management Console: *standard create* e *easy create*. Utilizando o método de *easy create*, os bancos são pré-configurados de acordo com boas práticas de desenvolvimento, necessitando somente a escolha do tipo do banco de dados e as capacidades da instância que será utilizada. Para esta proposta, por ser ambiente prototípico, foi selecionada as seguintes configurações de ambientes: banco de dados da classe *db.t3.micro*, com PostgreSQL, com 1GB de memória RAM, processador Intel Skylake E5 2686 v5 (2.5 GHz) com 2 vCPUS.

Com a instância criada, algumas informações importantes são disponibilizadas, apresentadas na Tabela 3.1. A primeira atribuição importante a ser observada é o **endpoint**. O endpoint é o endereço público de acesso ao servidor de banco de dados. A partir dele, é possível realizar a string de conexão ao banco de dados, fazendo com que o banco hospedado na AWS possa ser utilizado por outras ferramentas.

A segunda atribuição apresentada nesta tabela é a porta do banco de dados. A porta que é utilizada para realizar o tráfego de pacotes de rede é a 5432, a porta padrão do PostgreSQL. Esta porta pode ser alterada no console da AWS, caso seja necessário.

A terceira atribuição apresentada é a Zona de disponibilidade. Esta zona indica a localização física em que o banco de dados está instanciado. O código sa-east-la indica que está sendo utilizado o Servidor Leste da América do Sul, localizado em São Paulo. Este é o servidor da AWS mais próximo de Brasília, por isso esta escolha.

Por fim, a quarta atribuição indica se o banco de dados está publicamente acessível ou não. Caso o banco esteja publicamente acessível, são criados grupos de segurança, chamados de Virtual Private Cloud, ou simplesmente VPC, possibilitando a criação de regras de *inbound* e *outbound* de pacotes na instância em que está instalado o banco de dados. Neste cenário, por ser publicamente acessível, a instância aceita como elementos de entrada todos os protocolos TCP do tipo PostgreSQL, que utilize a porta 5432. Todo o restante do tráfego é eliminado. Como regra de *outbound*, todo o tráfego é liberado.

Tabela 3.1: Informações de Instância de Banco de Dados PostgreSQL

Atribuição	**Valor**
Endpoint	tcc.XXXXXXXXXX.sa-east-1.rds.amazonaws.com
Porta	5432
Zona de Disponibilidade	sa-east-1a
Publicamente acessível	Sim

Para acessar ao banco de dados e implementar o modelo definido na Figura 3.7, pode-se utilizar a ferramenta de linha de comando **psql**. Para instalar esta ferramenta, em um ambiente Debian, deve-se realizar os seguintes comandos:

Código 19

```
sudo apt-get update
```

```
sudo apt install postgresql postgresql-contrib
```

Estes comandos instalam um cliente PostgreSQL no ambiente, sendo possível realizar o acesso ao *psql*. Logo, em um terminal, para acessar ao banco de dados instanciado no RDS da Amazon AWS, executa-se:

Código 20

```
psql \
   --host=tcc.XXXXXXXXXXXX.sa-east-1.rds.amazonaws.com \
   --port=5432 \
   --username=tcc \
   --dbname=TCC
```

A partir deste comando, alguns argumentos são passados ao psql. O argumento **–host** é o endereço do banco de dados na AWS. Já o argumento **–port** é a porta do banco de dados obtida anteriormente ao criar a instancia na AWS. Já o argumento **–username** é o usuário de autenticação do banco de dados e **–dbname** é o nome do banco de dados que será acessado. Com isso, executando-o, será solicitada a senha de autenticação do usuário e, caso seja passada corretamente, o usuário terá acesso a toda a interface de comandos do PostgreSQL.

Logo, utilizando Códigos de Banco de dados apresentado no Anexo 3, é possível realizar a implementação das tabelas do banco de dados para o ambiente RDS. Para isso, basta utilizar:

Código 21

```
 psql --host=tcc.XXXXXXXXXXXX.sa-east-1.rds.amazonaws.com \
--port=5432 \
--username=tcc \
--dbname=TCC < backup.sql
```

Realizando este comando, todos os códigos SQL existentes dentro do arquivo *backup.sql* são executados. A Figura 3.8 apresenta o resultado da implementação deste comando, listando todas as tabelas existentes dentro do banco de dados. Finalizadas as configurações, portanto, é realizada a implementação da persistência de dados do protótipo IoT, utilizando o serviço cloud da AWS.

3.8.2 Elastic Compute Cloud - EC2

Instâncias EC2 são máquinas virtuais no ambiente cloud da Amazon AWS. A partir destas instâncias, é possível realizar a implementação de diversos tipos de serviços computacionais, como por exemplo, o middleware da proposta IoT. Assim, é possível realizar a implementação do código-fonte do middleware na instância EC2.

Para estes passos, no painel da AWS, ao selecionar o serviço de instâncias EC2, pode-se iniciar a criação de uma máquina virtual. As configurações de criação da instância estão apresentadas na Tabela 3.8.2.

```
                    List of relations
Schema |                Name                 | Type  | Owner
-------+-------------------------------------+-------+-------
public | ta_dispositivo_condicoes_ambientais | table | tcc
public | ta_dispositivos_coordenadas         | table | tcc
public | ta_dispositivos_usuarios            | table | tcc
public | ta_pets_safe_zones                  | table | tcc
public | ta_pets_usuarios                    | table | tcc
public | ta_safe_zones_coordenada            | table | tcc
public | tb_condicoes_ambientais             | table | tcc
public | tb_coordenadas                      | table | tcc
public | tb_dispositivos                     | table | tcc
public | tb_pets                             | table | tcc
public | tb_safe_zones                       | table | tcc
public | tb_usuarios                         | table | tcc
(12 rows)
```

Figura 3.8: Tabelas criadas no banco de dados PostgreSQL
Fonte: elaborado pelo autor (2022)

Tabela 3.2: Configurações da Instância EC2 AWS

Configuração	**Valor**
Endpoint	tcc.XXXXXXXXXX.sa-east-1.rds.amazonaws.com
Sistema Operacional	Ubuntu Focal 20.04 Server
Memória	8GB de RAM
CPU	4 vCPU
DNS público	ec2-34-220-156-154.us-west-2.compute.amazonaws.com
IP Público	34.220.156.154
Porta	5501

Outro ponto importante em que deve ser configurado na criação da instância, é a determinação de um par de chaves públicas e privadas para realizar a autenticação SSH da máquina virtual. Para isso, na própria interface de criação de instâncias, é possível criar um novo par de chaves ou utilizar um par de chaves existentes. Com isso, ao realizar a criação, será gerado uma chave pública e outra chave privada, do tipo RSA ou do tipo ED25519. Portanto, com essas chaves públicas e privadas criadas, é possível realizar o acesso remoto à instância, a partir de ferramentas como Putty, openSSH e WinSCP.

Utilizando o Putty como exemplo, é necessário recuperar a chave privada e utilizá-la como autenticação no instante da conexão. Para isso, com o Putty aberto, na sessão *category*, deve-se selecionar *Connection -> SSH -> Auth* e associar a chave privada ao campo *Authentication parameters*. Assim, utilizando o DNS público ou IP público apresentado na Tabela 3.2, a partir do protocolo SSH, é possível realizar a conexão com a instância EC2 a partir de um acesso remoto. A Figura 3.9 apresenta a linha de comando da instância EC2 após a autenticação de chaves.

3.8.3 *Continuous Integration e Continuous Delivery* - CI/CD

Comumente usado em DevOps, o paradigma de CI/CD é utilizado para otimizar entregas de *releases* para ambientes de homologação e produção. Neste processo, o foco principal são as mesclas entre integrações contínuas (CI) e entregas contínuas (CD). O processo de CI, consiste em 4 etapas principais: planejamento, desenvolvimento de código, *build* do código e testes contínuos. Já para o processo de CD

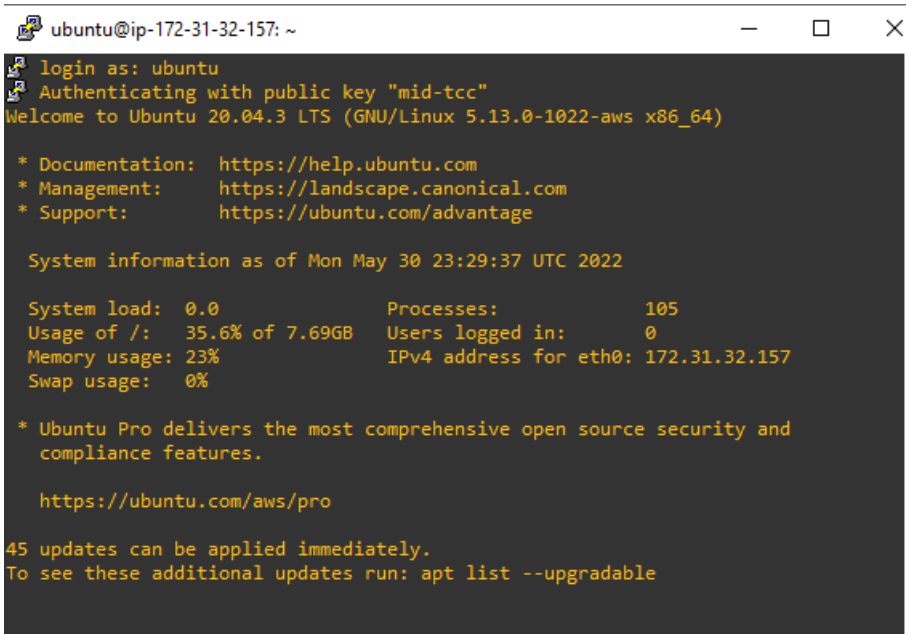

Figura 3.9: Instância EC2 do Middleware.
Fonte: elaborado pelo autor (2022)

as cinco etapas existentes são as seguintes: testes contínuos, *release*, *deploy*, operações e monitoramentos [3].

Realizando, portanto, a etapa de CI e após ela, a de CD, percebe-se que dá início a um ciclo de atividades, chamado de Ciclo de DevOps, ou simplesmente, CI/CD.

3.8.3.1 *Continuous Integration* - CI

A primeira etapa do ciclo de CI, é o planejamento do código que será implementado, isto é, planejamento do middleware da proposta IoT. Nesta etapa são definidos os padrões de códigos, linguagens de programação, ferramentas de desenvolvimento, etc. No caso, como apresentado anteriormente, o código do middleware será desenvolvido em python, utilizando o framework Flask, versionado pelo GitHub e implementado em uma instância EC2 na Amazon Web Services.

Para a segunda etapa, todo o código desenvolvido para o middleware foi versionado pelo GitHub. Este é um ponto de extrema importância, pois, no processo de CI/CD, a cada versionamento gerado pelo git, é possível gerar uma nova compilação para iniciar o processo de entrega. O código existente no repositório deste projeto estará apresentado nos Anexos I, II e III deste artigo.

Com o código preparado, a terceira etapa é a realização da compilação do código, verificando se todo o código foi programado corretamente, corrigindo possíveis erros de sintaxe. A partir desta compilação, é gerado um executável, responsável pela execução da aplicação desenvolvida. Neste cenário, o código `run.py` do middleware é o responsável pela inicialização da ferramenta.

Por fim, para a quarta etapa, são realizados testes contínuos na ferramenta, identificando inconsistências e possíveis pontos de melhoria de funcionamento e de código. Além disso, caso a compilação passe por

todos os testes de integridade de código, esta compilação estará disponível para ser utilizada no processo de CD.

3.8.3.2 *Continuous Delivery* - CD

A primeira etapa do ciclo de CD é iniciada no instante dos testes de integridades de códigos, assim como no processo de CI. Esta é uma etapa transitória, responsável por identificar se a compilação está disponível ou não para ser inserida em um ambiente de produção.

Com isso, caso ocorram as aprovações dos testes, inicia-se a segunda etapa do ciclo de CD. Nesta etapa é gerada uma versão de compilação de *release*, a documentando e a deixando disponível para a implementação em diversos ambientes, podendo ser de homologação ou de produção. Após a criação da versão de *release*, esta compilação deverá ser disponibilizada em algum ambiente, no caso do Middleware IoT, na instância EC2 da AWS. Este processo de disponibilização da versão em um ambiente produtivo é chamado de *deploy*.

Com a compilação em ambiente produtivo, são iniciadas as duas últimas etapas do ciclo de CD, que é de operações e monitoramentos. Nestas etapas, são coletados insumos da ferramenta, para verificar se está funcionando operacionalmente bem, assim como o monitoramento de performances e coleta de erros. Nesta etapa, é bastante importante a obtenção de relatórios de erros, pois, a partir deles, deverão ocorrer novas análises e planejamentos de como corrigir estes problemas. Assim, a etapa de CI é iniciada novamente. A Figura 3.10 esquematiza o ciclo de CI/CD de maneira visual.

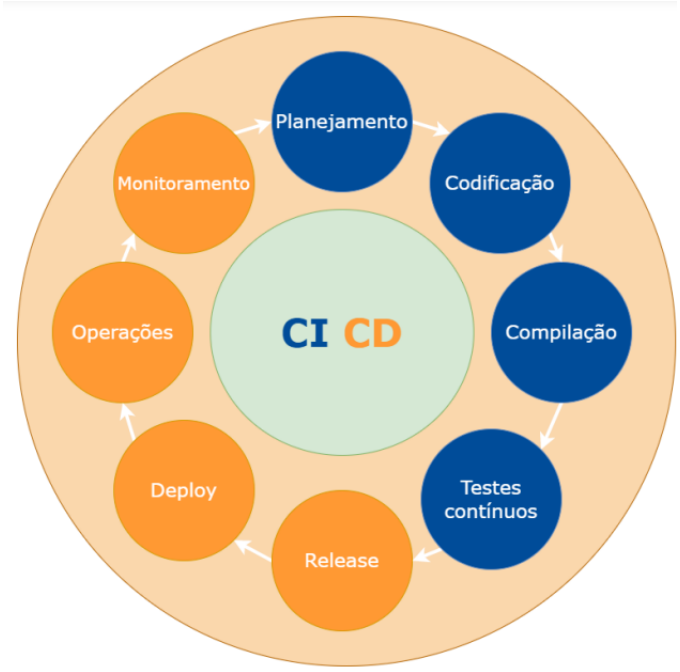

Figura 3.10: Ciclo de CI/CD.
Fonte: elaborado pelo autor (2022) adaptado de [3]

3.8.4 *Pipeline* do CI/CD

Para a realização do CI/CD das ferramentas *cloud*, utilizaram-se os seguintes serviços AWS: AWS Code Deploy e AWS Code Pipeline. Além disso, para o versionamento dos códigos, foi utilizado o GitHub. A escolha para o uso das ferramentas da Amazon AWS para fazer o CI/CD ocorreu-se por conta da facilidade de integração com os demais serviços que estão implementados. Além isso, com o AWS Code Pipeline é possível começar a desenvolver rapidamente, com interface amigável e com integração facilitada com outras ferramentas de pipeline *open source*, como o Jenkins.

O *deploy* do código é o coração do pipeline, portanto, de extrema importância para que tudo funcione corretamente. É no *deploy* que todo o código desenvolvido é carregado em um ambiente produtivo. Para esta proposta IoT, o ambiente produtivo selecionado é a instância EC2, configurada anteriormente. Com o AWS Code Deploy, será possível migrar todo o código presente em ambiente de desenvolvimento, a partir de um ramo do GitHub.

Para a utilização do serviço AWS Code Deploy, utilizando um usuário IAM com permissões necessárias, deve-se executar os seguintes passos:

1. Abrir o painel AWS Code Deploy;
2. Criar um novo aplicativo;
3. Associar a instância computacional; e
4. Criar grupo de implantação.

O nome criado para o aplicativo foi *Middleware IoT*, indicado no ponto 2. O ponto 3 é responsável pela associação do aplicativo criado à instância EC2 configurada anteriormente. Já o grupo de implantação, indicado no ponto 4, é um recurso que define configurações relacionadas à implantação, como em quais instâncias implantar e com que rapidez implantá-las [24].

Com a criação do serviço AWC Code Deploy, é possível iniciar a configuração do pipeline. Inicialmente, deve-se conter um usuário IAM, com as permissões para utilização do AWS Code Pipeline. A partir deste pré-requisito, é necessário seguir uma série de etapas para a implementação. Estas etapas estão apresentadas a seguir:

1. Selecionar região AWS;
2. Abrir painel AWS Code Pipeline;
3. Criar nova função de serviço;
4. Adicionar estágio de origem;
5. Adicionar estágios de complicação; e
6. Adicionar estágio de deploy.

Seguindo estes passos, indicados no formulário da AWS, é possível criar o pipeline. Para o ponto 1, selecionou-se a região sa-east-1, localizada em São Paulo, por ser mais próximo fisicamente do local de desenvolvimento do pipeline. Já para o ponto 3, selecionou-se o nome **CICDTCC** para a representação do pipeline que está sendo desenvolvido. Para o ponto 4, selecionou-se o repositório de versionamento de código do GitHub, configurando a funcionalidade de execução do pipeline para cada mudança de versão do código, isto é, a cada *commit*.

Já o ponto 5, para este cenário, não foi necessário implementar, visto que o pipeline está entregando um código python, desenvolvido em Flask. Por conta disso, não é necessário utilizar um estágio de compilação. Por fim, no ponto 6, é necessária a implementação do serviço AWS Code Deploy e associá-lo ao pipeline. Assim, associando a aplicação criada em AWS Code Deploy ao pipeline, pode-se finalizar a configuração. A Figura 3.11 apresenta o resultado da execução do pipeline no AWS Code Pipeline, executando, portanto, todo o processo de obtenção de código, construção e disponibilização de uma ferramenta em ambiente de produção.

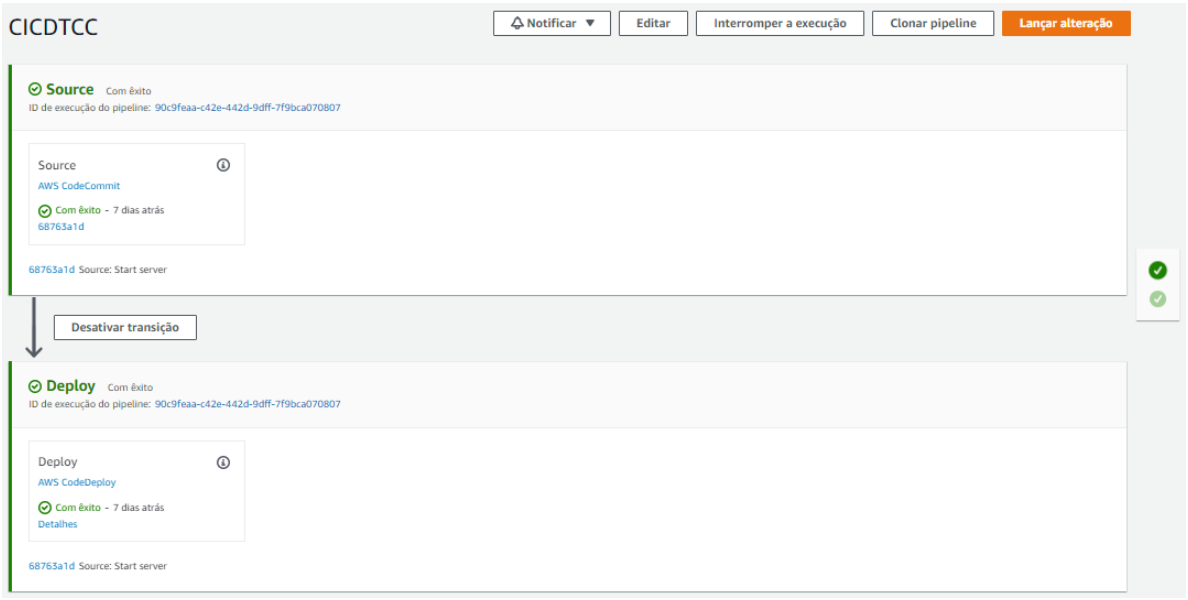

Figura 3.11: Pipeline de CI/CD.
Fonte: elaborado pelo autor (2022)

3.9 CONSTRUÇÃO DO APLICATIVO MOBILE

Para o desenvolvimento do aplicativo será utilizada a linguagem Dart com o framework Flutter. Com o Flutter, é possível o desenvolvimento de uma aplicação para várias plataformas, tanto para IOS, quanto para Android, tornando a programação híbrida. O aplicativo mobile deve conter os seguintes pré-requisitos para conseguir atingir o o propósito da aplicação:

- Conectividade à Internet;
- Acesso a ferramentas de mapas.

Para conseguir conectar o aplicativo ao middleware, deve-se utilizar alguma interface de conexão, como a biblioteca **Dio**. O Dio é um cliente HTTP para Dart, que suporta diversas ferramentas de transporte de dados, como dados de formulários, transporte de imagens, downloads de arquivos, etc [25].

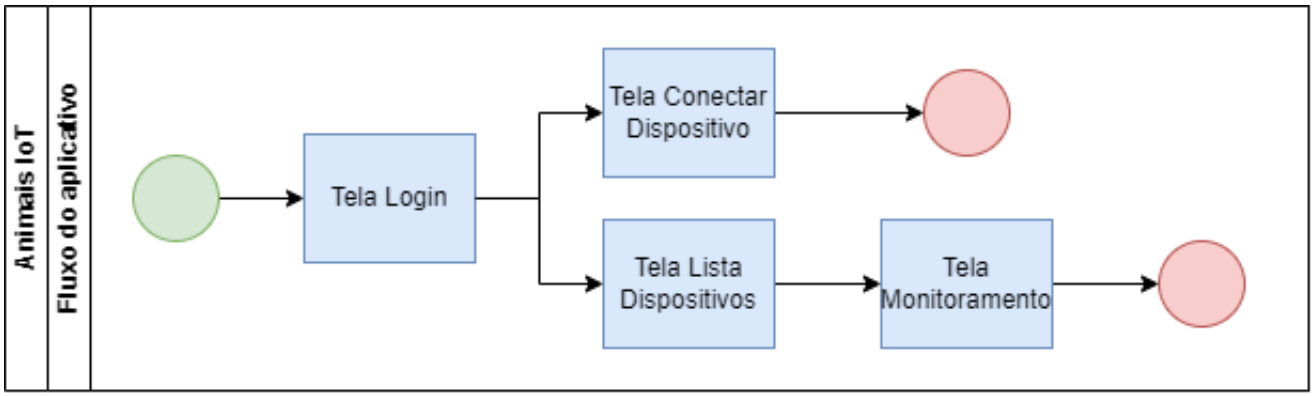

Figura 3.12: Processo de atividades do aplicativo.
Fonte: elaborado pelo autor (2022)

Já para apresentar os dados gerenciados do dispositivo IoT, é necessário de um suporte a uma ferramenta de mapas. Para isso, uma ferramenta open source de ótimo desempenho é o **Open Street Map**. Esta ferramenta possui dados abertos e podem ser editados por qualquer pessoa. Além disso, o uso é livre, podendo ser utilizado em diversos tipos de aplicações, tanto web quando mobiles [26]. Existe também uma biblioteca Dart que realiza a implementação dos serviços da OpenStreetMap para Flutter, chamada de *flutter_osm_plugin*.

No processo apresentado na Figura 3.12 é possível identificar os caminhos que o usuário poderá seguir pelo aplicativo. Olhando esta figura, percebe-se que o aplicativo possui quatro telas, podendo ser acessados por dois caminhos.

- Tela de login;
- Tela Conexão ao dispositivo IoT;
- Tela Lista de Dispositivos; e
- Tela Monitoramento.

Na Tela de Login, o usuário poderá apresentar as suas credenciais de acesso, como usuário e senha. Como o foco do artigo é na arquitetura IoT, não foi estruturado um sistema de segurança de autenticação, apresentando esta tela somente para fins de navegação do usuário.

Já na tela de conexão ao dispositivo IoT, é apresentado um formulário que deverá ser preenchido o *MAC Address* do dispositivo IoT, assim como a localização da zona de segurança. Já a tela de lista de dispositivos apresenta todos os dispositivos que foram associados ao usuário autenticado.

Por fim, a tela de monitoramento, é a tela responsável por apresentar a localização do animal e se o mesmo se encontra em uma zona de segurança. Neste aplicativo, existem dois cenários de apresentação do animal: quando o mesmo se encontra dentro da zona de segurança e quando ele se encontra fora da zona de

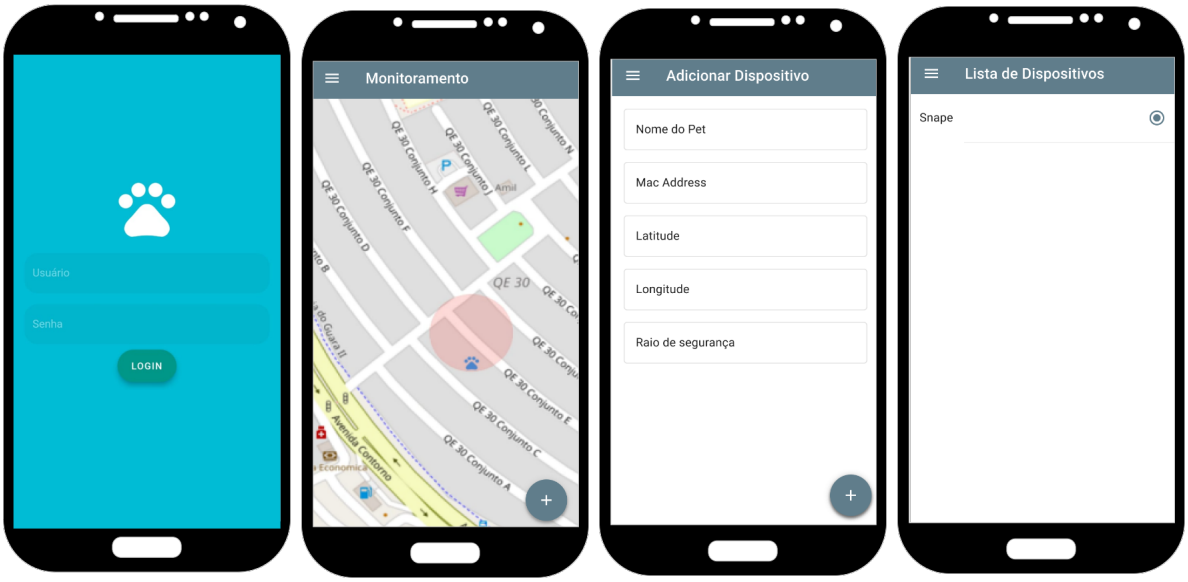

Figura 3.13: Telas do Aplicativo Mobile.
Fonte: elaborado pelo autor (2022)

segurança. Quando o animal se encontra dentro da zona de segurança, o ícone de indicação estará da cor azul. Já quando o animal se encontra em uma região não segura, a cor do ícone é alterada para vermelha.

A Figura 3.13 apresenta as quatro telas do aplicativo mobile.

4 RESULTADOS EXPERIMENTAIS

Para os resultados experimentais, foram necessários uma protoboard, um microcontrador ESPWROOM32, os módulos SIM800L e NEO6M, jumpers, duas baterias de 3.4V e uma roupa de animal. Para conseguir unir a placa de ensaio à roupa do animal foram utilizados velcros. Com isso, a partir do esquemático da Figura 3.3, o protótipo foi montado, possibilitando o início dos experimentos. A Figura 4.1 apresenta o dispositivo como um todo, com um animal de estimação utilizando o protótipo IoT. Por ser um protótipo, foi utilizado uma placa de ensaio, porém posteriormente, todos estes componentes podem ser condensados em uma placa de circuito impresso, reduzindo o tamanho do dispositivo a ser associado a uma roupa ou coleira do animal.

Figura 4.1: Uso do dispositivo em um animal.
Fonte: elaborado pelo autor (2022)

Assim, foi associado um chip de celular, com acesso a rede celular, ao dispositivo IoT. Este chip será o responsável pela transformação do dispositivo IoT em um usuário móvel, conseguindo conectar à Internet em qualquer local onde possua cobertura 4G, 3G e 2,5G. Este chip deverá ser anexado ao módulo SIM800L.

Com o chip conectado e o GPS em funcionamento, foram gerados os seguintes testes de implementação para obter os resultados do dispositivo:

1. Consulta da última localização do animal localizado dentro de uma zona de segurança;

2. Consulta das 100 últimas localizações do animal, localizados dentro de uma zona de segurança;

3. Última localização de um animal fora da zona de segurança.

4.1 ÚLTIMA LOCALIZAÇÃO EM ZONA DE SEGURANÇA

Para a realização deste teste, o dispositivo foi colocado no animal de estimação presente na Figura 4.1. Com isso, de acordo com os horários definidos pelo ciclo de vida do dispositivo de hardware, todas as coordenadas geográficas do animal são capturadas pelo módulo GPS.

Em seguida, os dados georreferenciados são encapsulados em um pacote HTTP e encaminhados para o middleware, a partir do módulo GPRS, utilizando a rede 4G. Portanto, todos estes pacotes são entregues ao middleware e carregados no banco de dados.

Para este primeiro cenário, foi recuperado somente a última localização do animal em zona de segurança. Para isso, o código do aplicativo para a recuperação da última localização está apresentado no Código 22.

Código 22

```
static Future<Coordenada> getLastPosition() async {
    try {
      Response response;
      String url =
          MiddlewareService.SERVER_URL +
          MiddlewareService.GET_POSITION;

      response = await Dio().get(url);
      return Coordenada.fromArrayList(response.data).last;
    } catch (e) {
      rethrow;
    }
}
```

A partir deste código, é possível observar a utilização de um método estático, que retorna um objeto futuro, que no caso, é a última coordenada capturada pelo animal que esteja com o dispositivo IoT associado. Logo, é recupera-se a URL do middleware responsável pela obtenção das coordenadas. Com isso, o protocolo HTTP é encapsulado a partir da utilização da classe Dio, criando a requisição com o servidor. Assim, este método recupera uma lista de coordenadas. Como para este problema só é necessária a última localização, chama-se o atributo *last*, fazendo com que seja retornado somente o último elemento da lista.

Este último elemento da lista, é a representação geográfica do último elemento capturado pelo dispositivo IoT conectado ao animal. A partir disso, esta coordenada geográfica é apresentada em um mapa no dispositivo celular. Neste mapa, é relatada a localização da zona de segurança, assim como o animal.

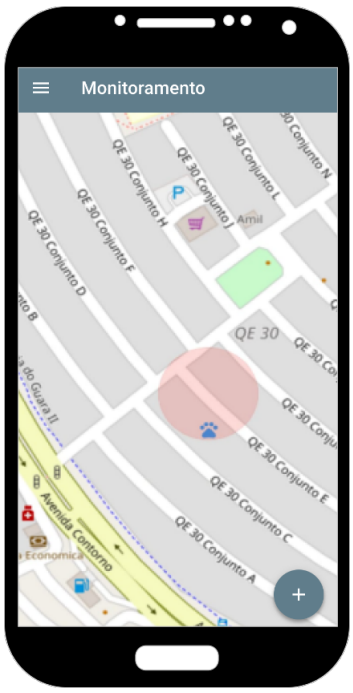

Figura 4.2: Última localização do animal doméstico.
Fonte: elaborado pelo autor (2022)

Na Figura 4.2 é possível identificar a última localização do animal, indicado por uma pata azul. Também é apresentada a zona de segurança, representada pela cor vermelha. Com isso, a partir deste resultado, percebe-se que o animal se encontra dentro da região de segurança cadastrada pelo tutor.

Para verificar se estes valores estão corretos, foi realizado uma consulta no banco de dados para obter o último registro de determinado animal. Logo, utilizando um comando SQL, foi possível recuperar este valor. A consulta SQL executada está apresentada no Código 2.

Código 23

```
select * from tb_coordenadas order by dt_create desc limit 10;
```

Este código acessa a tabela **tb_coordenadas** e recupera todas as colunas dela, com um limite de 10 registros, ordenados de maneira decrescente de acordo com a data de criação. Neste caso, somente o primeiro registro é importante, sendo grifado na Figura 4.3.

4.2 100 ÚLTIMAS LOCALIZAÇÕES DO ANIMAL

Para verificar a acurácia e dos dados georreferenciados coletados pelo dispositivo IoT, foram representados e plotados no aplicativo as 100 últimas localizações capturadas. Para este cenário, o dispositivo IoT ficou em uma localização estática por 2 horas. A partir disso, de acordo com este intervalo de tempo, foram recuperadas as 100 últimas amostras, obtendo o resultado apresentado na Figura 4.4.

Ao observar estes resultados, percebe-se que há um pequeno erro experimental, visto que diversos

```
TCC=> select * from tb_coordenadas order by dt_create desc limit 10;
 id_coordenada | vl_longitude | lv_latitude | vl_altitude |         dt_create          |         dt_update
---------------+--------------+-------------+-------------+----------------------------+----------------------------
           195 |   -47.980835 |  -15.842998 |      1067.2 | 2022-04-23 22:42:31.049082 | 2022-04-23 22:42:31.049082
           194 |   -47.980835 |  -15.842998 |      1069.6 | 2022-04-23 22:42:25.933842 | 2022-04-23 22:42:25.933842
           193 |   -47.980827 |  -15.842999 |      1069.6 | 2022-04-23 22:42:20.3876   | 2022-04-23 22:42:20.3876
           192 |   -47.980827 |  -15.842999 |      1073.1 | 2022-04-23 22:42:13.857937 | 2022-04-23 22:42:13.857937
           191 |    -47.98083 |  -15.842992 |      1073.1 | 2022-04-23 22:42:08.106277 | 2022-04-23 22:42:08.106277
           190 |    -47.98083 |  -15.842992 |      1073.6 | 2022-04-23 22:42:02.892147 | 2022-04-23 22:42:02.892147
           189 |   -47.980835 |  -15.842983 |      1073.6 | 2022-04-23 22:41:56.388746 | 2022-04-23 22:41:56.388746
           188 |   -47.980835 |  -15.842983 |      1072.4 | 2022-04-23 22:41:50.900589 | 2022-04-23 22:41:50.900589
           187 |    -47.98084 |  -15.842974 |      1072.4 | 2022-04-23 22:41:45.059055 | 2022-04-23 22:41:45.059055
           186 |    -47.98084 |  -15.842974 |        1070 | 2022-04-23 22:41:39.878244 | 2022-04-23 22:41:39.878244
```

Figura 4.3: Últimas coordenadas recuperadas diretamente do banco de dados.
Fonte: elaborado pelo autor (2022)

pontos do gráfico não se encontram em uma mesma posição. Isso ocorre por conta dos cálculos realizados entre os satélites para determinar a posição do dispositivo. Como o número de satélites no instante da coleta influencia na localização final capturada, em amostras que foram calculadas com menos satélites foram encontrados valores diferentes da geolocalização.

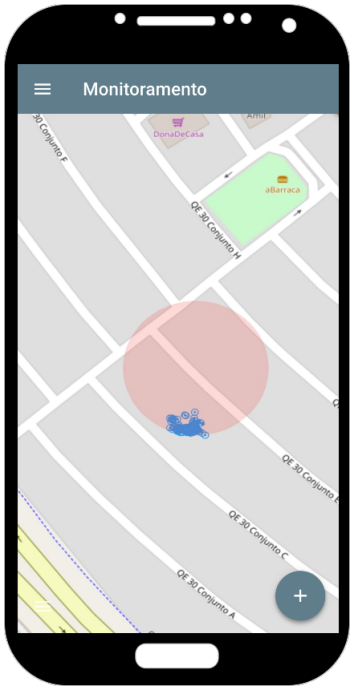

Figura 4.4: Últimas 100 amostras georreferenciadas.
Fonte: elaborado pelo autor (2022)

O erro médio encontrado a partir destas amostras é de um raio de aproximadamente 9,75 metros, a partir da coordenada mais concêntrica entre as amostras. Com isso, é recomendado, portanto, uma zona de segurança que tenha no mínimo 10 metros de raio, para que o animal esteja dentro de uma área confiável.

4.3 ÚLTIMA LOCALIZAÇÃO FORA DA ZONA DE SEGURANÇA

Por fim, o último experimento a ser realizado é referente a última localização do animal, quando este se encontra fora da zona de segurança.

Para este cenário, o objetivo é que o usuário do aplicativo saiba quando seu animal está seguro ou não. Assim, caso o animal esteja fora do posicionamento do círculo de zona de segurança, a cor do ícone do animal será alterada, apresentando um tom de alerta. Logo, para a alteração da identificação da cor do animal, a seguinte lógica é obtida: se a distância entre a última localização e o centro da zona segura for maior que o raio de segurança, altere a cor da indicação do animal. A Figura 4.5 apresenta o resultado do instante em que o animal se encontra fora da zona definida pelo usuário. Nesta figura, é possível observar a zona de segurança, indicada pelo círculo vermelho no mapa, assim como a localização do animal, também indicada pela pata vermelha.

Figura 4.5: Última localização do animal doméstico fora da zona de segurança.
Fonte: elaborado pelo autor (2022)

5 CONCLUSÕES

Por conta da crescente na quantidade de animais domésticos e furtos de animais, foi pensada em uma forma de realizar o gerenciamento de animais, de maneira automatizada e tecnológica. Com isso, este trabalho apresentou uma proposta para implementação deste sistema de gerenciamento, utilizando o conceito de Internet das Coisas, obtendo as informações dos pets a partir de um microcontrolador e os enviando para uma base de informação.

Foram apresentados, portanto, uma proposta de arquitetura tecnológica, elencando tanto a infraestrutura de rede a ser criada e configurada, assim como a codificação dos sistemas middleware e aplicativos finais onde serão apresentadas as informações.

A partir deste desenvolvimento, muitas ideias foram implementadas. Ocorreu a mescla de diversos conceitos de Engenharia de Redes de Comunicação com técnicas de Engenharia de Software, Engenharia de Dados e Devops, a fim de gerar a proposta de modelo IoT para gerenciamento e monitoramento de animais.

A partir destes estudos, foi possível realizar a aplicação do protótipo em um caso real de uso, associando o dispositivo implementado a uma roupa de animal e capturando os dados para o monitoramento. Assim, estes dados capturados foram persistidos em uma infraestrutura cloud, utilizando conceitos de cloud computing, redes e devops, para por fim, os apresentá-los ao usuário em um dispositivo celular. Nesta apresentação, foi possível identificar alguns casos de uso deste gerenciamento, apresentando se o animal encontra-se ou não em uma zona segura pré-estabelecida pelo usuário.

5.1 TRABALHOS FUTUROS

Durante o desenvolvimento da proposta, foram encontrados ainda pontos de melhorias que possam ser implementados em trabalhos futuros. Nestes trabalhos, portanto, podem ser otimizados tanto o fluxo de informação quanto a performance do modelo. Além disso, podem ser melhorados também a maneira de alimentação energética, fazendo com que o dispositivo fique mais tempo sem necessidade de recarga de bateria.

Porém mesmo que existam melhorias futuras a serem implementadas neste modelo, a proposta apresentada é capaz de solucionar todas as problemáticas apresentadas na sessão 1 deste artigo. A partir desta proposta, por fim, é possível contornar as fugas e furtos de animais, apresentando a geolocalização dos mesmos, assim como gerar métricas de monitoramento a partir dos dados capturados e armazenados no banco de dados implementado, melhorando a qualidade de vida tanto do animal, quanto do tutor que o agregou em sua família.

No decorrer deste projeto, muitas ideias foram surgindo, assim como possíveis melhorias da implementação IoT. Para a apresentação destes trabalhos futuros, serão divididos em dois tópicos:

- Big Data
- Data Streaming

A medida que novos usuários forem utilizar a arquitetura apresentada neste trabalho, a quantidade de dados aumentará consideravelmente, ainda mais por se tratar de dados georreferenciados e em tempo real. Com isso, ao deixar de olhar o projeto na visão de protótipo e observar para um caso de uso real, o volume de informação a ser inserido nas bases de dados passa a ser bastante considerável. Por conta disso, deve-se pensar em uma arquitetura de *bigdata*, voltado principalmente para uma organização e apresentação dos dados.

Logo, pode-se realizar a divisão do modelo de dados apresentado em duas partes: uma voltada para o negócio, isto é, para as entidades, e outra voltada para métricas e resultados. Com esta divisão, pode-se, portanto, realizar uma outra modelagem de dados, migrando o modelo relacional apresentado anteriormente para um modelo dimensional. Com o modelo dimensional, pode-se criar um Data Warehouse com as informações coletadas pelos dispositivos, fazendo com que as métricas resultantes possam ser melhor apresentadas e performadas.

Além disso, pode-se ocorrer uma adaptação ao middleware, atualizando o modelo de API REST, que utiliza cargas batch, para um modelo de streaming de dados. Neste caso, o middleware atuaria como um gateway em tempo real entre o dispositivo celular e o dispositivo IoT. Os dados de localização em tempo real, portanto, seriam apresentados diretamente ao celular, sem necessariamente ser salva em um banco de dados. Já os dados históricos, seriam encaminhados para o Data Warehouse.

Existem algumas ferramentas interessantes que são capazes de executar estas alterações. Para *streaming* de dados, uma ferramenta Open Source muito robusta é o Apache Kafka. Esta ferramenta atua enfileirando mensagens em tópicos, existindo um produtor e um consumidor[27]. O produtor é o responsável por enviar a informação ao tópico, enquanto o consumidor responsável por recuperar esta mensagem e apresentar da maneira que o for conveniente. Já para realizar a persistência do Data Warehouse, uma ferramenta interessante e que está em bastante ascendência é o Apache Pinot. Com o Apache Pinot é possível utilizar a comunicação com os tópicos Kafka e criar uma estrutura de tabelas a partir destes dados. Além disso, como o Apache Pinot é uma ferramenta voltada para Big Data, com ele é possível realizar processamento de uma grande volumetria de informação a uma latência muito baixa, o otimizando para apresentação de dados e resultados de métricas. Por fim, o Apache Pinot é uma ferramenta estruturada para funcionar com melhor performance utilizando uma modelagem dimensional, por ser um banco de dados Online Analytical Processing - OLAP [28].

REFERÊNCIAS BIBLIOGRÁFICAS

1 MENDONÇA, F. L. L. D. Proposição de um modelo de interoperação peer-to-peer para internet das coisas - p2piot. 2019.

2 SYSTEMS, E. *ESP32 Datasheet*. 2016.

3 SERVICES, A. W. *Configure um pipeline de integração e entrega contínuas (CI/CD) na AWS*. 2022. <https://aws.amazon.com/pt/getting-started/projects/set-up-ci-cd-pipeline/>. Accessed: 2022-04-30.

4 IBGE, A. População de animais de estimação no brasil - 2013 - em milhões. 2013.

5 IBGE. Censo 2017 - número de estabelecimentos agropecuários com efetivo da pecuária e número de cabeças, por tipologia, espécie da pecuária e condição do produtor em relação às terras. 2017.

6 SILVA, C. P.; FEYH, P. G. R.; ROLAND, C. E. de F. Pets: Desenvolvimento de sistema de geolocalização para o monitoramento de animais de estimação. *Revista Eletrônica de Sistemas de Informação e Gestão Tecnológica*, v. 9, n. 3, 2018.

7 NEVES, M.; PEREZ, N. B.; SISTI, R. N. Análise exploratória de dados de monitoramento dos animais em um sistema de integração lavoura-pecuária. In: IN: CONGRESSO BRASILEIRO DE AGRICULTURA DE PRECISÃO, 2014, SÃO PEDRO, SP .... [S.l.], 2014.

8 SANTOS, B. P.; SILVA, L. A.; CELES, C. S.; NETO, J. B. B.; PERES, B. S.; VIEIRA, M. A. M.; VIEIRA, L. F. M.; GOUSSEVSKAIA, O. N.; LOUREIRO, A. A. Internet das coisas: da teoria à prática. 2016.

9 EVANS, D. *A Internet das Coisas Como a próxima evolução da Internet está mudando tudo. Abril de 2011*.

10 SANTAELLA, L.; GALA, A.; POLICARPO, C.; GAZONI, R. Desvelando a internet das coisas. *Revista GEMInIS*, v. 4, n. 2, p. 19–32, 2013.

11 MADAKAM, S.; LAKE, V.; LAKE, V.; LAKE, V. et al. Internet of things (iot): A literature review. *Journal of Computer and Communications*, Scientific Research Publishing, v. 3, n. 05, p. 164, 2015.

12 GARTNER. *Gartner Hype Cycle*. 2021. <https://www.gartner.com/en/research/methodologies/gartner-hype-cycle>. Accessed: 2022-08-03.

13 AKYILDIZ, I. F.; SU, W.; SANKARASUBRAMANIAM, Y.; CAYIRCI, E. Wireless sensor networks: a survey. *Computer networks*, Elsevier, v. 38, n. 4, p. 393–422, 2002.

14 ABO-ZAHHAD, M.; AMIN, O.; FARRAG, M.; ALI, A. A survey on protocols, platforms and simulation tools for wireless sensor networks. *Int. J. Energy Inf. Commun*, v. 5, p. 17–34, 2014.

15 ALGAMILI, A. S.; KHIR, M. H. M.; DENNIS, J. O.; AHMED, A. Y.; ALABSI, S. S.; HASHWAN, S. S. B.; JUNAID, M. M. A review of actuation and sensing mechanisms in mems-based sensor devices. *Nanoscale research letters*, SpringerOpen, v. 16, n. 1, p. 1–21, 2021.

16 CHUO, L.-X.; FENG, Z.; KIM, Y.; CHIOTELLIS, N.; YASUDA, M.; MIYOSHI, S.; KAWAMINAMI, M.; GRBIC, A.; WENTZLOFF, D.; BLAAUW, D. et al. Millimeter-scale node-to-node radio using a carrier frequency-interlocking if receiver for a fully integrated 3 wireless sensor node. *IEEE Journal of Solid-State Circuits*, IEEE, v. 55, n. 5, p. 1128–1138, 2019.

17 WELLS, D.; BECK, N.; KLEUSBERG, A.; KRAKIWSKY, E. J.; LACHAPELLE, G.; LANGLEY, R. B.; SCHWARZ, K. peter; TRANQUILLA, J. M.; VANICEK, P.; WELLS, D.; DELIKARAOGLOU, D.; KLEUSBERG, A.; KRAKIWSKY, E. J.; LACHAPELLE, G.; LANGLEY, R. B.; SCHWARZ, K. peter; TRANQUILLA, J. M.; VANICEK, P. Guide to gps positioning. In: *Canadian GPS Assoc*. [S.l.: s.n.], 1987.

18 KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach*. 7. ed. [S.l.: s.n.], 2016. ISBN 978-0-13-359414-0.

19 BRAINWY. *PyDev*. 2014–2018. <https://www.pydev.org/>. Accessed: 2022-03.

20 LORD, D. Flask is developed. janeiro 2010. Accessed: 2022-05-18.

21 FLUTTER. *Flutter*. 2021. <https://flutter.dev/>. Accessed: 2021-09-23.

22 ZHANG, Y.; FU, S.-C.; CHAN, K. C.; SHIN, D.-M.; CHAO, C. Y. Boosting power output of flutter-driven triboelectric nanogenerator by flexible flagpole. *Nano Energy*, Elsevier, v. 88, p. 106284, 2021.

23 RESTX. *Documentação Flask RestX*. 2020. <https://flask-restx.readthedocs.io/en/latest/>. Accessed: 2021-04-30.

24 SERVICES, A. W. *Criar um aplicativo no Code Deploy*. 2021. <https://docs.aws.amazon.com/pt_br/codepipeline/latest/userguide/tutorials-simple-codecommit.html#codecommit-create-codedeploy-app>. Accessed: 2021-04-30.

25 FLUTTERCHINA.CLUB. *dio 4.0.6*. 2021. <https://pub.dev/packages/dio>. Accessed: 2022-04-22.

26 OPENSTREETMAP. *OpenStreetMap*. 2021. <https://www.openstreetmap.org/about>. Accessed: 2022-04-22.

27 KAFKA, A. *Documentação Apache Kafka*. 2022. <https://docs.pinot.apache.org/>. Accessed: 2022-06-30.

28 PINOT, A. *Documentação Apache Pinot*. 2022. <https://docs.pinot.apache.org/>. Accessed: 2022-06-30.

Anexo 1 - Código Fonte do hardware

Código 24

```
#define TINY_GSM_MODEM_SIM800

#include <HardwareSerial.h>
#include <TinyGPS++.h>
#include <ArduinoJson.h>
#include <ArduinoHttpClient.h>

#include <TinyGsmClient.h>
#include <HTTPClient.h>

#define GPS_SERIAL_NUM 1
#define GPS_RX_PIN 34
#define GPS_TX_PIN 35
#define LED_DEBUG 23
#define uS_TO_S_FACTOR 1000000ULL
#define TIME_TO_SLEEP  3
#define  TINY_GSM_MODEM_SIM800
#define GSM_TX_PIN 28
#define GSM_RX_PIN 27
#define GSM_SERIAL_NUM 2
#define TINY_GSM_MODEM_SIM800
#define CUT_BATTERY_LEVEL 30
#define CUT_SAFE_ZONE 15
#define BATTERY 13

const char serverName[] =
    "http://ec2-34-221-129-164.us-west-2.compute.amazonaws.com/";
const int  port = 5501;

TinyGPSPlus gps;
HardwareSerial GPSSerial(GPS_SERIAL_NUM);

HardwareSerial GSMSerial(GSM_SERIAL_NUM);
TinyGsm modemGSM(GSMSerial);
TinyGsmClient client(modemGSM);
HttpClient http(client, serverName, port);
```

```
String MAC_ADDRESS = "FF:FF:FF:FF:FF:FF";

const int BAUD_RATE = 9600;

const char* APN = "timbrasil.br";
const char* USER = "tim";
const char* PASSWORD = "tim";

int S = 0;
int Z = 0;
int B = 0;

unsigned short STATE = 000;

typedef struct
{
  float latitude;
  float longitude;
  float altitude;
  String macAddress;
  bool battery;
  bool safeZone;
} PackageData;

PackageData safezoneData;
PackageData locationPetData;

void setup() {
  Serial.begin(115200);
  GPSSerial.begin(9600, SERIAL_8N1, GPS_RX_PIN, GPS_TX_PIN);
  GSMSerial.begin(9600, SERIAL_8N1, GSM_RX_PIN, GSM_TX_PIN);

  Serial.print("Inicializando máquina.\n");
  gsmConfiguration();
}

void gsmConfiguration() {
  if (!modemGSM.restart())
  {
    Serial.println("Restarting GSM\nModem failed");
    ESP.restart();
```

```
    return;
  }
  Serial.println("Modem restart OK");

  // aguarda network
  if (!modemGSM.waitForNetwork())
  {
    Serial.println("Failed to connect\nto network");
    return;
  }
  Serial.println("Modem network OK");

  // conecta na rede (tecnologia GPRS)
  if (!modemGSM.gprsConnect(APN, USER, PASSWORD))
  {
    Serial.println("GPRS Connection\nFailed");
    return;
  }
}

void loop() {
  getSafeZoneUser();
  setState();
  getGPS();
}

void sendData(PackageData json) {
  String serverPath = "/api/evento/";
  String contentType =  "application/json";
  String body = transformPackageDataToJson(json);
  http.post(serverPath, contentType, body);
  int statusCode = http.responseStatusCode();
  String response = http.responseBody();
  Serial.print("Status code: ");
  Serial.println(statusCode);
  Serial.print("Response: ");
  Serial.println(response);
  http.stop();
}

void setPackageData(PackageData* P, float latitude, float longitude,
    float altitude, String macAddress, bool battery, bool safeZone) {
```

```
  P->latitude = latitude;
  P->longitude = longitude;
  P->altitude = altitude;
  P->macAddress = macAddress;
  P->battery = battery;
  P->safeZone = safeZone;
}

void getGPS() {
  PackageData data;
  if (Serial.available()) {
    char c = Serial.read();
    GPSSerial.write(c);
  }
  if (GPSSerial.available()) {
    char c = GPSSerial.read();
    gps.encode(c);
    if (gps.location.isUpdated()) {
      setPackageData(
        &data,
        gps.location.lat(),
        gps.location.lng(),
        gps.altitude.meters(),
        MAC_ADDRESS,
        getBattery(),
        getSafeZone()
      );
    }
  }
  locationPetData = data;
}

bool getBattery() {
  return false;
}

bool getSafeZone() {
  return true;
}

PackageData getSafeZoneUser(){
  DynamicJsonDocument doc(2048);
```

```
  String serverPath = "/api/ZonaSeguranca/" + MAC_ADDRESS;
  http.get(serverPath);
  int statusCode = http.responseStatusCode();
  String response = http.responseBody();
  Serial.println(response);
  http.stop();
  PackageData data;
  deserializeJson(doc, response);
  JsonObject obj = doc.as<JsonObject>();
  setPackageData(
        &data,
        obj[String("latitude")],
        obj[String("longitude")],
        obj[String("altitude")],
        MAC_ADDRESS,
        "",
        obj[String("safe-zone")]
  );
  safezoneData = data;
}

String transformPackageDataToJson(PackageData P) {
  DynamicJsonDocument doc(2048);
  doc["lv_latitude"] = P.latitude;
  doc["vl_longitude"] = P.longitude;
  doc["vl_altitude"] = P.altitude;
  doc["cd_mac_address"] = P.macAddress;
  doc["vl_bateria"] = P.battery;
  doc["vl_safe_zone"] = P.safeZone;
  String json;
  serializeJson(doc, json);
  return json;
}

unsigned short getState() {
  return STATE;
}

void setStandBy(int i){
  S = i;
}
```

```
void setSecurityZone(int i){
  Z = i;
}

void setBateryLevel(int i){
  B = i;
}

void state000() {
  STATE = 0;
  setStandBy(0);
  setSecurityZone(0);
  setBateryLevel(0);
  //000 -> 100
}

void state001() {
  STATE = 1;
  setStandBy(0);
  setSecurityZone(0);
  setBateryLevel(1);
}

void state010() {
  STATE = 2;
  setStandBy(0);
  setSecurityZone(1);
  setBateryLevel(0);
}

void state011() {
  STATE = 3;
  setStandBy(0);
  setSecurityZone(1);
  setBateryLevel(1);

}

void state100() {
  STATE = 4;
  setStandBy(1);
  setSecurityZone(0);
```

```
  setBateryLevel(0);
  sendData(locationPetData);
  esp_deep_sleep(TIME_TO_SLEEP * uS_TO_S_FACTOR);
}

void state101() {
  STATE = 5;
   setStandBy(1);
  setSecurityZone(0);
  setBateryLevel(1);
  sendData(locationPetData);
  esp_deep_sleep(TIME_TO_SLEEP * uS_TO_S_FACTOR);
}

void state110() {
  STATE = 6;
  setStandBy(1);
  setSecurityZone(1);
  setBateryLevel(0);
  sendData(locationPetData);
  esp_deep_sleep(TIME_TO_SLEEP * uS_TO_S_FACTOR);
}

void state111() {
  STATE = 7;
  setStandBy(1);
  setSecurityZone(1);
  setBateryLevel(1);
  sendData(locationPetData);
  esp_deep_sleep(TIME_TO_SLEEP * uS_TO_S_FACTOR);
}

int getBatery() {
  float tensao = analogRead(BATTERY);

  float ref = (1024*CUT_BATTERY_LEVEL)/100;

  if(tensao < ref){
    return 0;
  }else{
    return 1;
  }
```

```
}

int getZone(PackageData pet, PackageData safezoneCenter) {
  double distance = TinyGPSPlus::distanceBetween(pet.latitude,
  pet.longitude, safezoneCenter.latitude, safezoneCenter.longitude);
  if(distance < CUT_SAFE_ZONE){
    return 1;
  }else{
    return 0;
  }
}

void setState() {
  int b = getBatery();
  int z = getZone(locationPetData, safezoneData);
  int s = 1;

  //Definição do estado
  STATE |= s;
  STATE = STATE << 1;
  STATE |= z;
  STATE = STATE << 1;
  STATE |= b;

  switch (STATE) {
    case 0b000: {
        if (s == 1) {
          //change for 100
          state100();
        }
        //state000();
        break;
      }
    case 0b001: {
        if (s == 1) {
          //change for 101
          state101();
        }
        break;
      }
    case 0b010: {
        if (s == 1) {
```

```
      //change for 100
      state110();
    }
    break;
  }
case 0b011: {
    if (s == 1) {
      //change for 100
      state111();
    }
    break;
  }
case 0b100: {
    if (s == 0) {
      //change for 100
      state000();
    }
    if (z == 1) {
      //change for 100
      state110();
    }
    if (b == 1) {
      //change for 100
      state101();
    }
    break;
  }
case 0b101: {
    if (s == 0) {
      state001();
    }
    if (z == 1) {
      state111();
    }
    if (b == 0) {
      state100();
    }
    break;
  }
case 0b110: {
    if (b == 1) {
      state111();
```

```
        }
        if (s == 0) {
          state010();
        }
        if (z == 0) {
          state100();
        }
        break;
      }
    case 0b111: {
        if (s == 0) {
          state011();
        }

        if (z == 0) {
          state101();
        }
        if (b == 0) {
          state110();
        }
        break;
      }
    default: {
        Serial.print("Estado inexistente.");
        break;
      }
  }
  Serial.print(STATE);
}
```

Anexo 2 - Código Fonte do Middleware

COORDENADA_DB.PY

Código 25

```
import json
from datetime import datetime as dt
from app.conf.config import Config


class CoordenadaDb:
    items = [
        {
            'id_coordenada': 1,
            'vl_longitude': '-47.9713212,15',
            'vl_latitude': '-15.8478124',
            'vl_altitude': '1000',
            'dta_create': '2022-04-14T04:09:41.654Z',
            'dta_update': '2022-04-14T04:09:41.654Z'
        },
        {
            'id_coordenada': 2,
            'vl_longitude': '-47.9713212,15',
            'vl_latitude': '-15.8478124',
            'vl_altitude': '1000',
            'dta_create': '2022-04-14T04:09:41.654Z',
            'dta_update': '2022-04-14T04:09:41.654Z'
        }
    ]

    @classmethod
    def insert(cls, item):
        conn = Config.config()
        if conn != None:
            try:
                cur = conn.cursor()
                id = cur.execute('''
                    INSERT INTO tb_coordenadas
                    (lv_latitude,vl_longitude,vl_altitude)
```

```
                VALUES (%s, %s, %s)
                RETURNING id_coordenada
            ''', (item['lv_latitude'], item['vl_longitude'],
            item['vl_altitude'],))
            id = cur.fetchone()[0]
            conn.commit()
            cur.close()
            return json.dumps(id, default=str)
        except Exception as ex:
            print(ex)
            conn.rollback()

@classmethod
def insertCoordenadaDispositivo(cls, id_dispositivo,
id_coordenada):
    conn = Config.config()
    if conn != None:
        try:
            cur = conn.cursor()
            cur.execute('''
                INSERT INTO ta_dispositivos_coordenadas
                (id_dispositivo,id_coordenada)
                VALUES (%s, %s)
            ''', (id_dispositivo, id_coordenada,))
            conn.commit()
            cur.close()
            return True
        except Exception as ex:
            print(ex)
            conn.rollback()

@classmethod
def get(cls, id_coordenada=None):
    conn = Config.config()
    if id_coordenada != None:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_coordenada, lv_latitude, vl_longitude,
                vl_altitude, dt_create, dt_update
                FROM tb_coordenadas
                WHERE tb_coordenadas = %s
```

```
            ''', (id_coordenada,))
            dispositivo = cur.fetchone()[0]
            conn.commit()
            cur.close()
            return {
                "id_coordenada": dispositivo[0],
                "vl_latitude": dispositivo[1],
                "vl_longitude": dispositivo[2],
                "vl_altitude": dispositivo[3]}
        except Exception as ex:
            conn.rollback()
            raise ex
    else:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_coordenada, lv_latitude, vl_longitude,
                vl_altitude, dt_create, dt_update
                FROM tb_coordenadas
            ''')
            coordenadas = cur.fetchall()
            coordenadas_arr = []
            for dispositivo in coordenadas:
                coordenadas_arr.append(
                    {
                        "id_coordenada": dispositivo[0],
                        "vl_latitude": dispositivo[1],
                        "vl_longitude": dispositivo[2],
                        "vl_altitude": dispositivo[3]}
                )
            conn.commit()
            cur.close()
            return coordenadas_arr
        except Exception as ex:
            conn.rollback()
            raise ex
```

COORDENADA_CONTROLLER.PY

Código 26

```
from flask_restx import Resource, Namespace, fields
from flask import request
from app.main.coordenada.coordenada_db import CoordenadaDb

api = Namespace('Coordenada', description='Coordenadas')

modelo = api.model('CoordenadaModel', {
    'id_coordenada': fields.Integer,
    'vl_longitude': fields.String,
    'vl_latitude': fields.String,
    'vl_altitude': fields.String,
    'dta_create': fields.DateTime,
    'dta_update': fields.DateTime
})


@api.route('/')
class CoordenadaController(Resource):
    # documentação para tipo de respostas
    @api.response(200, "Busca realizada com sucesso")
    def get(self):
        return CoordenadaDb.get(), 200

    @api.expect(modelo)
    def post(self):
        return CoordenadaDb.insert(request.json), 201


@api.route('/<id_coordenada>')
class CoordenadaIdController(Resource):

    @api.response(200, "Busca realizada com sucesso")
    def get(self, id_coordenada: int):

        return CoordenadaDb.get(int(id_coordenada)), 200
```

DISPOSITIVO_CONTROLLER.PY

Código 27

```
from email.policy import default
from flask_restx import Resource, Namespace, fields
from flask import request
from itsdangerous import json
from app.main.dispositivo.dispositivo_db import DispositivoDb

api = Namespace('Dispositivo', description='Dispositivos IoT')
# definição de modelo que será validado ao receber post
modelo = api.model('DispositivoModel', {
    'id_dispositivo': fields.Integer,
    'cd_mac_address': fields.String,
    'vl_bateria': fields.String,
    'dta_create': fields.String,
    'dta_update': fields.String
})


@api.route('/')
class DispositivoController(Resource):
    # documentação para tipo de respostas
    @api.response(200, "Busca realizada com sucesso")
    def get(self):
        return DispositivoDb.get(), 200

    @api.expect(modelo)  # espera modelo ao criar novo objeto
    def post(self):
        return DispositivoDb.insert(request.json), 201


@api.route('/<id_dispositivo>')
class DispositivoIdController(Resource):
    # documentação para tipo de respostas
    @api.response(200, "Busca realizada com sucesso")
    def get(self, id_dispositivo: int):
        # return CoordenadaDb.obter(), 200
        return DispositivoDb.get(int(id_dispositivo)), 200

    @api.response(200, "Busca realizada com sucesso")
    @api.param('vl_bateria', 'Valor da bateria')
    def put(self, id_dispositivo: int):
        return DispositivoDb.update(int(id_dispositivo),
        request.json), 201
```

```
    @api.response(200, "Exclusão realizada com sucesso")
    @api.param('id_dispositivo', 'Identificador do dispositivo')
    def delete(self, id_dispositivo: int):
        return DispositivoDb.delete(int(id_dispositivo)), 202
```

DISPOSITIVO_DB.PY

Código 28

```
import json
from app.conf.config import Config
from datetime import datetime as dt


class DispositivoDb:
    items = [
        {
            'id_dispositivo': 1,
            'cd_mac_address': 'FF:FF:FF:FF:FF:FF:FF',
            'vl_bateria': '50',
            'dta_create': '2022-04-14T04:09:41.654Z',
            'dta_update': '2022-04-14T04:09:41.654Z'
        }
    ]

    @classmethod
    def insert(cls, item):
        conn = Config.config()
        if conn != None:
            try:
                cur = conn.cursor()
                id = cur.execute('''
                    INSERT INTO tb_dispositivos (cd_mac_address)
                    VALUES (%s)
                    RETURNING id_dispositivo
                ''', (item['cd_mac_address'],))

                id = cur.fetchone()[0]
                conn.commit()
                cur.close()
```

```
            return json.dumps(id, default=str)
        except Exception as ex:
            print(ex)
            conn.rollback()

@classmethod
def get(cls, id_dispositivo=None):
    conn = Config.config()
    if id_dispositivo != None:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_dispositivo, cd_mac_address,
                vl_bateria, dt_create, dt_update
                FROM tb_dispositivos
                WHERE id_dispositivo = %s
            ''', (id_dispositivo,))
            dispositivo = cur.fetchone()[0]
            conn.commit()
            cur.close()
            return {
                "id_dispositivo": dispositivo[0],
                "cd_mac_address": dispositivo[1],
                "vl_bateria": dispositivo[2],
                "dt_create": dt.strftime(dispositivo[3]),
                "dta_update": dt.strftime(dispositivo[4])}
        except Exception as ex:
            conn.rollback()
            raise ex
    else:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_dispositivo, cd_mac_address,
                vl_bateria, dt_create, dt_update
                FROM tb_dispositivos
            ''')
            dispositivos = cur.fetchall()
            dispositivos_arr = []
            for dispositivo in dispositivos:
                dispositivos_arr.append(
                    {
```

```
                        "id_dispositivo": dispositivo[0],
                        "cd_mac_address": dispositivo[1],
                        "vl_bateria": dispositivo[2],
                        "dt_create":
                        dt.strftime(dispositivo[3],
                        "%Y-%m-%dT%H:%M:%S%z"),
                        "dta_update":
                        dt.strftime(dispositivo[4],
                        "%Y-%m-%dT%H:%M:%S%z")
                    }
                )
            conn.commit()
            cur.close()
            return dispositivos_arr
        except Exception as ex:
            conn.rollback()
            raise ex

@classmethod
def getByMac(cls, cd_mac_address=None):
    conn = Config.config()
    if cd_mac_address != None:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_dispositivo, cd_mac_address,
                vl_bateria, dt_create, dt_update
                FROM tb_dispositivos
                WHERE cd_mac_address = %s
            ''', (cd_mac_address,))
            dispositivo = cur.fetchone()
            conn.commit()
            cur.close()
            return {
                "id_dispositivo": dispositivo[0],
                "cd_mac_address": dispositivo[1],
                "vl_bateria": dispositivo[2]
            }
        except Exception as ex:
            conn.rollback()
            raise ex
```

```
    @classmethod
    def delete(cls, id_dispositivo):
        conn = Config.config()
        if id_dispositivo != None:
            try:
                cur = conn.cursor()
                cur.execute('''
                    DELETE FROM tb_dispositivos where id_dispositivo
                    = %s
                ''', (id_dispositivo,))
                conn.commit()
                cur.close()
                return {"mensagem": f"id_dispositivo
                {id_dispositivo} deletado com sucesso"}
            except Exception as ex:
                conn.rollback()
                raise ex
```

EVENTO_CONTROLLER.PY

Código 29

```
from email.policy import default
from flask_restx import Resource, Namespace, fields
from flask import request
from itsdangerous import json

from app.main.evento.evento_db import EventoDb

api = Namespace('Evento', description='Eventos IoT')
# definição de modelo que será validado ao receber post
modelo = api.model('EventoModel', {
    'cd_mac_address': fields.String,
    'vl_bateria': fields.Float,
    'vl_latitiude': fields.Float,
    'vl_longitude': fields.Float,
    'vl_altitude': fields.Float,
    'vl_safe_zone': fields.Float,
    'dta_create': fields.DateTime,
    'dta_update': fields.DateTime
})
```

```
@api.route('/')
class EventoController(Resource):
    @api.expect(modelo)  # espera modelo ao criar novo objeto
    def post(self):
        return EventoDb.insert(request.json), 201
```

EVENTO_DB.PY

Código 30

```
import json
from app.conf.config import Config
from datetime import datetime as dt

from app.main.coordenada.coordenada_db import CoordenadaDb
from app.main.dispositivo.dispositivo_db import DispositivoDb


class EventoDb:

    @classmethod
    def insert(cls, item):
        conn = Config.config()
        if conn != None:
            try:
                coordenada = {}
                coordenada['lv_latitude'] = item['lv_latitude']
                coordenada['vl_longitude'] = item['vl_longitude']
                coordenada['vl_altitude'] = item['vl_altitude']
                dispositivo = DispositivoDb.getByMac(item['cd_mac_address'])
                id_dispositivo = dispositivo['id_dispositivo']
                #DispositivoDb.update(id_dispositivo, dispositivo)
                id_coordenada = CoordenadaDb.insert(coordenada)
                inserted = CoordenadaDb.insertCoordenadaDispositivo(
                    id_dispositivo, id_coordenada)
                if(inserted):
                    return json.dumps({'response': 'Ok'}, default=str)
                else:
                    return json.dumps({'response': 'Erro'}, default=str)
```

```
        except Exception as ex:
            print(ex)
            conn.rollback()

@classmethod
def get(cls, id_dispositivo=None):
    conn = Config.config()
    if id_dispositivo != None:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_dispositivo, cd_mac_address,
                vl_bateria, dt_create, dt_update
                FROM tb_dispositivos
                WHERE id_dispositivo = %s
            ''', (id_dispositivo,))
            dispositivo = cur.fetchone()[0]
            conn.commit()
            cur.close()
            return {
                "id_dispositivo": dispositivo[0],
                "cd_mac_address": dispositivo[1],
                "vl_bateria": dispositivo[2],
                "dt_create": dt.strftime(dispositivo[3]),
                "dta_update": dt.strftime(dispositivo[4])}
        except Exception as ex:
            conn.rollback()
            raise ex
    else:
        try:
            cur = conn.cursor()
            cur.execute('''
                SELECT id_dispositivo, cd_mac_address,
                vl_bateria, dt_create, dt_update
                FROM tb_dispositivos
            ''')
            dispositivos = cur.fetchall()
            dispositivos_arr = []
            for dispositivo in dispositivos:
                dispositivos_arr.append(
                    {
                        "id_dispositivo": dispositivo[0],
```

```
                        "cd_mac_address": dispositivo[1],
                        "vl_bateria": dispositivo[2],
                        "dt_create":
                        dt.strftime(dispositivo[3],
                        "%Y-%m-%dT%H:%M:%S%z"),
                        "dta_update":
                        dt.strftime(dispositivo[4],
                        "%Y-%m-%dT%H:%M:%S%z")
                    }
                )
            conn.commit()
            cur.close()
            return dispositivos_arr
        except Exception as ex:
            conn.rollback()
            raise ex

@classmethod
def delete(cls, id_dispositivo):
    conn = Config.config()
    if id_dispositivo != None:
        try:
            cur = conn.cursor()
            cur.execute('''
                DELETE FROM tb_dispositivos where id_dispositivo = %s
            ''', (id_dispositivo,))
            conn.commit()
            cur.close()
            return {"mensagem": f"id_dispositivo
            {id_dispositivo} deletado com sucesso"}
        except Exception as ex:
            conn.rollback()
            raise ex

@classmethod
def update(cls, id_dispositivo, novo_item: dict):
    item = next(
        filter(
        lambda x: x['id_dispositivo'] == id_dispositivo,
        cls.items), {})
    index = cls.items.index(item)
```

```
if novo_item.get('cd_mac_address'):
    item['cd_mac_address'] = novo_item.get('cd_mac_address')

if novo_item.get('vl_bateria'):
    item['vl_bateria'] = novo_item.get('vl_bateria')

if novo_item.get('dta_update'):
    item['dta_update'] = novo_item.get('dta_update')

cls.items[index] = item
return item
```