Universidade Federal da Bahia
Instituto de Computação
Bacharelado em Ciência da Computação

Trabalho de Conclusão de Curso

Desenvolvimento de uma API baseada no padrão Redfish para Monitoramento Remoto de Raspberry Pi

Alexsandro Leite Figueiredo

Salvador - Bahia
24 de novembro de 2023

Desenvolvimento de uma API baseada no padrão Redfish para Monitoramento Remoto de Raspberry Pi

Alexsandro Leite Figueiredo

Trabalho de Conclusão de curso do título de Bacharel em Ciência da Computação.

Orientador(a): Prof(a). Ph.D. Cassio Serafim Prazeres.

Salvador - Bahia
24 de novembro de 2023

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias Prof. Omar Catunda, SIBI – UFBA.

F475 Figueiredo, Alexsandro Leite

Desenvolvimento de uma API baseada no padrão Redfish para Monitoramento Remoto de Raspberry Pi / Alexsandro Leite Figueiredo. – Salvador, 2023.

69 f.: il.color.

Orientador: Prof. Dr. Cássio Serafim Prazeres.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal da Bahia. Instituto de computação, 2023.

1. Raspberry Pi. 2. Computação. 3. Linguagem de programação. I. Prazeres, Cássio Serafim. II. Universidade Federal da Bahia. III. Título.

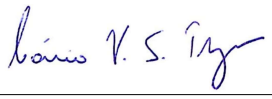
CDU 004

Desenvolvimento de uma API baseada no padrão Redfish para Monitoramento Remoto de Raspberry Pi

Alexsandro Leite Figueiredo

Trabalho de Conclusão de curso do título
de Bacharel em Ciência da Computação.

Trabalho aprovado. Salvador/BA, Brasil, 24 de novembro de 2023.



Prof(a). Ph.D. Cássio Serafim Prazeres
UFBA

Documento assinado digitalmente
gov.br CLAUDIO JUNIOR NASCIMENTO DA SILVA
Data: 17/01/2024 15:38:01-0300
Verifique em https://validar.iti.gov.br

Prof(a). M.e Cláudio Júnior Nascimento Silva
UFBA

Documento assinado digitalmente
gov.br GEORGE PACHECO PINTO
Data: 17/01/2024 08:28:14-0300
Verifique em https://validar.iti.gov.br

Prof(a). M.e George Pacheco Pinto
IFBA

Dedico este trabalho à minha família e minha querida esposa Thainã, sem ela por perto os resultados não seriam os mesmos. Grato pela sua compreensão e presença.

Agradecimentos

A Deus, que fez com que meus objetivos fossem alcançados, durante todos os meus anos de estudos. Aos meus pais, que me ajudaram em todas as dificuldades encontradas, não me deixando desanimar. A minha esposa, que me deu todo o apoio necessário. Ao meu Orientador, pela orientação, suporte e paciência durante a realização deste trabalho. E a Universidade Federal da Bahia, pela oportunidade oferecida.

"Eu prefiro ter perguntas que não
podem ser respondidas a ter respostas
que não podem ser questionadas".
Autor - Richard Feynman

Resumo

Na era da Internet das Coisas (IoT), a necessidade de monitorar os dispositivos de maneira eficaz e padronizada é fundamental. Este Trabalho de Conclusão de Curso se propõe a abordar essa demanda, explorando o desenvolvimento de uma API baseada no padrão Redfish para o monitoramento remoto de dispositivos Raspberry Pi, equipamento que desempenha um papel vital na IoT, e o padrão Redfish oferece uma estrutura unificada para o gerenciamento de dispositivos. Neste projeto, será pesquisado a arquitetura, implementação técnica e integração da API Redfish em dispositivos Raspberry Pi. Este trabalho busca demonstrar como essa abordagem pode otimizar o monitoramento remoto de dispositivos IoT, promovendo a eficiência e a automação.
Palavras-chave: Raspberry. Monitoramento remoto. API Redfish

Abstract

In the era of the Internet of Things (IoT), the need to monitor devices effectively and in a standardized manner is crucial. This Final Course Project aims to address this demand by exploring the development of an API based on the Redfish standard for the remote monitoring of Raspberry Pi devices, which play a vital role in IoT. The Redfish standard provides a unified framework for device management. This project will investigate the architecture, technical implementation, and integration of the Redfish API into Raspberry Pi devices. The goal is to demonstrate how this approach can optimize remote monitoring of IoT devices, promoting efficiency and automation.
Keywords: Raspberry Pi, Remote Monitoring, Redfish API

Lista de ilustrações

Lista de abreviaturas e siglas

API *Application Programming Interface*

BIOS *Basic Input Output System*

DTMF *Distributed Management Task Force*

GLP *General Public License*

HTTP *Hipertext Transfer Protocol*

HTTPS *Hipertext Transfer Protocol Secure*

Json *JavaScript Object Notation*

IoT *Internet of Things*

IPMI *Intelligent Platform Management Interface*

OData *Open Data Protocol*

REST *Representational State Transfer*

SSH *Secure Shell*

TLS *Transport Layer Security*

URI *Uniform Resource Identifier*

URL *Uniform Resource Locator*

SSL *Secure Sockets Layer*

YAML *yet another markup language*

XML *Extensible Markup Language*

Sumário

1 Introdução

Na era da *Internet of Things* (IoT), onde a interconexão e o gerenciamento eficiente de dispositivos são cruciais para o avanço tecnológico, encontramos desafios significativos no âmbito do monitoramento remoto. Dispositivos como o Raspberry Pi, desempenham papéis essenciais em diversos setores, desde automação residencial até monitoramento ambiental e industrial. No entanto, monitorar esses dispositivos de forma remota ainda é uma tarefa complexa e desafiante.

Diante dessa realidade, surge a necessidade premente de um protocolo de gerenciamento padronizado, simples e eficaz, é neste momento que o padrão Redfish entra em cena. Apoiado por organizações líderes da indústria, esse padrão oferece uma estrutura unificada para o gerenciamento remoto de dispositivos, proporcionando interoperabilidade e eficiência em um cenário tão dinâmico.

O propósito deste trabalho é ir além e explorar a implementação de uma *Application Programming Interface* (API), baseada no padrão Redfish para o monitoramento remoto de dispositivos Raspberry Pi. Preenchendo uma lacuna importante, facilitando o monitoramento e gerenciamento destes dispositivos, enquanto se aproveita de todos os benefícios do protocolo Redfish.

Ao longo desta pesquisa, será aprofundado o estudo sobre a arquitetura, desenvolvimento e implementação técnica de uma API Redfish adaptada para o universo do Raspberry Pi. Além disso, será analisado de forma prática como essa aplicação pode ser integrada em um ambiente Raspberry Pi, explorando tanto os benefícios quanto os desafios inerentes a essa abordagem.

Esta iniciativa oferece oportunidade para otimizar a eficiência e segurança no monitoramento remoto. Objetivando desenvolver uma solução que não apenas permita o monitoramento remoto de forma eficiente do Raspberry Pi, mas ofereça simplicidade para os usuários e integração com outras ferramentas, garantindo interoperabilidade e contribuindo para o desenvolvimento de soluções inovadoras.

Ao concluir este estudo, não apenas será demostrado a viabilidade e eficácia de uma API Redfish para Raspberry Pi, mas também ressaltar as implicações práticas dessa abordagem. Certamente essas implicações podem reverberar em uma variedade de aplicações na IoT e na automação, trazendo benefícios tangíveis para diferentes setores.

1.1 Objetivos

Desenvolver um protótipo de API baseado no padrão Redfish, com objetivo de oferecer um conjunto funcional de ferramentas básicas para monitoramento remoto de dispositivos Raspberry Pi.

Como Objetivos específicos temos:

- Elaborar um protótipo funcional de API para monitoração de dispositivos Raspibery Pi;
- Oferecer um pacote básico de funções para monitoração;
- Oferecer um arcabouço para desenvolvimento futuros do trabalho;
- Disponibilizar sua inicialização de forma autônoma.

1.2 Organização do trabalho

Esta monografia está dividida em cinco capítulos. Introdução onde é feito uma contextualização e motivações para elaboração do projeto. O segundo capítulo contempla a fundamentação teórica e os conceitos aplicados no desenvolvimento da aplicação apresentada. Já o terceiro capítulo aborda a arquitetura proposta com a descrição de seu funcionamento, aplicação de conceitos do capítulo anterior e decisões de implementação. O quarto capítulo apresenta o sistema construído, com detalhamento das ferramentas utilizadas. O quinto capítulo conclui a monografia, incluindo considerações finais sobre a solução desenvolvida e propostas de melhorias futuras.

2 Fundamentação Teórica

Este capítulo descreve os principais conceitos explorados durante o trabalho, tratando elementos presentes no desenvolvimento da aplicação, sua arquitetura e tecnologias utilizadas.

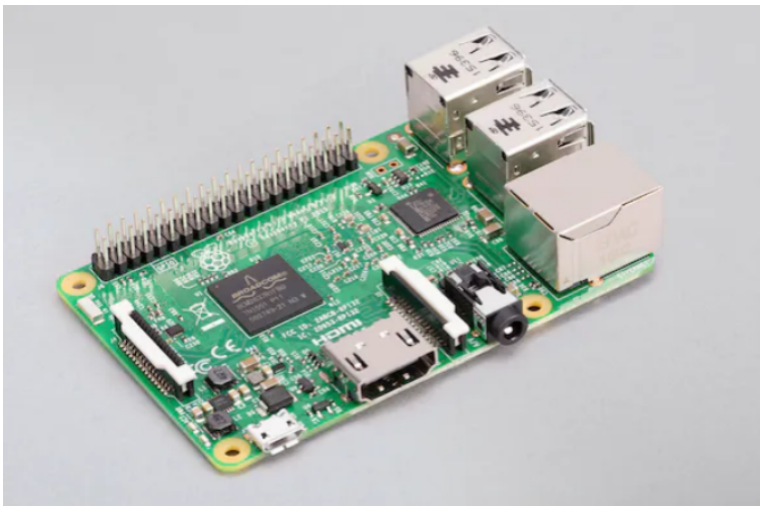
2.1 Raspberry Pi

O Raspberry Pi, um pequeno e simples computador, criado por Eben Upton em 2006. Apesar de seu tamanho compacto, ele é um dispositivo potente e de baixo custo que funciona como qualquer outro computador, necessitando apenas de um teclado, mouse, monitor e fonte de alimentação para operar.

Devido à sua versatilidade e custo acessível, o Raspberry Pi é uma excelente plataforma para servir como central de controle, podendo se conectar a vários dispositivos. Ele é produzido pela Raspberry Pi Foundation, uma organização oriunda do Reino Unido cujo objetivo é promover o interesse em computação, fornecendo computadores de alto desempenho e baixo custo para aprendizado, resolução de problemas e diversão.(FOUNDATION, 2023)

O aparelho possui unidades de processamento central e gráficos, áudio, comunicação com periféricos e outros *hardwares*, além de um chip de memória RAM. Todos esses componentes são construídos sobre um único componente, conforme ilustrado na Figura 1 (FOUNDATION, 2023).

Figura 1 – Raspberry Pi 3 Model B



Fonte:(FOUNDATION, 2023)

.

Na Figura 1, temos o dispositivo de terceira geração do Model B onde dispõe de 1 GB de memória RAM, o processador Broadcom de quatro núcleos com 1.2 GHz e 64 bits, possui uma entrada de cartões SD para ser usado como disco rígido (FOUNDATION, 2023).
Como principais vantagens do dispositivos temos:

- Suporte de 4 portas USB aumentando a quantidade de periféricos conectados;
- Conexões via Wi-Fi, bluetooth, e cabo de rede expandindo os métodos de comunicação com outros aparelhos na mesma rede;
- Devido ao seu tamanho pequeno e por usar um Sistema operacional (S.O) independente pode ser programado para diversas necessidades.

Como principais desvantagens:

- Não possui uma *Basic Input Output System* (BIOS) sempre é necessário inicializar através do cartão SD;
- Não possui bateria no circuito, consequentemente não possui relógio em tempo real;
- Para suprir o circuito necessita de uma fonte de 2.5 amperes, consequentemente alguns carregadores são incompatíveis (CHACOS, 2016).

2.2 Conceito de IoT

A IoT é um conceito que se refere à interconexão de objetos físicos e dispositivos através da internet, permitindo que eles coletem e compartilhem dados em tempo real. Esses objetos podem variar de dispositivos simples, como sensores e atuadores, a dispositivos mais complexos, como *smartfones* e veículos (ORACLE, 2023). A IoT possibilita a coleta, monitoramento e controle de informações de maneira automatizada e remota, gerando benefícios em diversas áreas, incluindo agricultura, saúde, indústria, transporte e muito mais.

2.3 Protocolo HTTP

O *Hipertext Transfer Protocol* (HTTP), protocolo mais importante da internet, é usado para distribuir objetos de hipermídia referenciados por uma *Uniform Resource Identifier* (URI). Ele funciona com base em solicitações feitas por um cliente e respostas recebidas de um servidor.(OLIVEIRA, 2018).

Para enviar uma requisição ao servidor, é essencial que a mensagem inclua, entre outros detalhes, a especificação do método a ser empregado para enviar a requesição. Os métodos HTTP determinam qual ação será executada no servidor. Os mais utilizados são, DELETE, POST, GET e PUT (MASSE, 2011).

Esses métodos são fundamentais para a comunicação entre o cliente e o servidor na web. O método DELETE remove um recurso específico. O método POST envia dados para serem processados por um recurso específico. O método GET solicita dados de um recurso específico. E o PUT atualiza um recurso específico que já existe ou cria um novo se ele não existir.

Existe um código de status para toda resposta do protocolo HTTP, que servem para identificar se ocorreu algum erro no sistema ou se a requisição foi realizada com sucesso, os códigos estão divididos em classes conforme a Tabela 1.

Tabela 1 – Categorias dos códigos de estado HTTP

CATEGORIA	DESCRIÇÃO
1xx:Informativo	Sinaliza informações do protocolo de transferência.
2xx:Sucesso	Sinaliza que a requisição do cliente foi aceita com êxito.
3xx:Redirecionamento	Sinaliza a necessidade de uma ação do cliente, para completar a ação.
4xx:Erro de Cliente	Comunica erros que o cliente pode ter cometido.
5xx:Erro de Servidor	Informa um erro do servidor durante o processamento da requisição.

Fonte:(MASSE, 2011)

2.4 *Web Services*

Um *Web Service* pode ser definido como um sistema de software que provê a comunicação entre aplicativos e sistemas diferentes. Isso permite que essas plataformas troquem dados em diferentes formatos. Dado que cada uma dessas partes possui sua própria linguagem, é essencial o uso de uma "linguagem universal", como *Extensible Markup Language* (XML) e *JavaScript Object Notation* (Json)(MORO; DORNELES; REBONATTO, 2011).

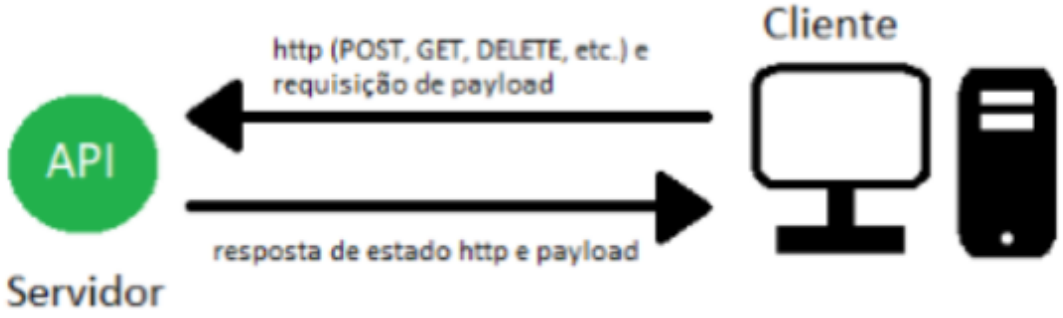
Consequentemente, um Web Service tem como finalidade envolver funcionalidades disponíveis para acesso por meio de protocolos. O protocolo predominantemente empregado na comunicação entre clientes e servidores desses serviços é o HTTP. O uso de tecnologias fundamentadas nesses serviços tem crescido substancialmente, impulsionado pelo aumento na utilização de API para a integração entre diversas plataformas. Quando

uma API é acessada por meio do protocolo HTTP, ela pode ser classificada como um webservice (OLIVEIRA, 2018).

2.5 *Application Programming Interface*

Application Programming Interface API, termo em inglês, ou simplesmente API, é uma interface desenvolvida por meio de código que segue um conjunto de padrões e procedimentos, capaz de conectar diferentes tipos de aplicações e sistemas. Nesse sentido, uma API possibilita que qualquer utilizador que tenha familiaridade com a documentação correspondente a ela possa utilizá-la, eliminando a necessidade de criar um aplicativo que tenha a mesma lógica. Desta maneira, uma API contém um ou vários pontos de acesso *endpoints* que disponibilizam um serviço a ser utilizado por outras aplicações clientes (OLIVEIRA, 2018).

Figura 2 – Comunicação Cliente Servidor.



Fonte: adaptado (GROUP., 2020)

A Figura 2 é um exemplo de comunicação da API com o cliente utilizando os métodos de comunicação via protocolo HTTP. O cliente envia uma requisição de *payload* acompanhada de um método get, post e delete, então é retornado resposta de estado HTTP e a API responde o *payload* (GROUP., 2020).

2.6 *Representational State Transfer*

Representational State Transfer (REST) , termo em inglês para Transferência de Estado Representacional, é um modelo de arquitetura baseado no protocolo HTTP, que foi apresentado pela primeira vez no ano de 2000 na tese de doutorado de Roy Thomas Fielding, REST é um modelo arquitetônico para sistemas distribuídos que estabelece normas e traz características como escalabilidade, uso de cache e identificação de recursos, possibilitando a criação de uma aplicação distribuída seguindo as melhores práticas. Quando

uma API é criada baseada na arquitetura REST, podemos dizer que ela é RESTful (FIELDING, 2000). *Constrains* são restrições obrigatórias que fazem parte da arquitetura:

1 - Cliente e Servidor: A ideia central é que o usuário de uma aplicação REST não precisa ter conhecimento sobre as implementações de banco de dados, cache, entre outros. Isso permite que cada componente se desenvolva de maneira autônoma, tornando a manutenção mais fácil (FIELDING, 2000);

2 -*Stateless*: Em cada solicitação do cliente ao servidor, ela é tratada como uma entidade única e não depende de outras solicitações. Assim, ela deve conter todas as informações necessárias para que o servidor possa interpretá-la corretamente. Em aplicações REST, não há retenção de informações do usuário através de sessões e cookies (FIELDING, 2000);

3 - Cache: Para melhorar a velocidade das respostas do servidor e evitar processamento desnecessário, especialmente quando há um aumento no número de solicitações, o cache pode ser empregado para otimizar o desempenho da aplicação. (FIELDING, 2000);

4 - Interface Uniforme: A utilização de uma interface uniforme é a restrição principal do REST. Isso o diferencia de todos os outros serviços web. A interface uniforme se concentra na simplicidade, acessibilidade, interoperabilidade e na habilidade de descobrir recursos (COSTA, 2014).

2.7 JavaScript Object Notation

Trata-se de um padrão aberto utilizado para transferência de dados entre servidor e uma aplicação web de forma estruturada em formato simples e de rápida compreensão por humanos, facilitando a leitura e escrita e por esse motivo é amplamente utilizada. Json é uma alternativa a outro padrão já muito conhecido, o XML, que é baseado em linguagens C, C++, C, Java, JavaScript, Perl, Python e outras (JSON, 2023).

O Json é composto de duas estruturas:

- Chave-valor: organizados em uma coleção de pares, em várias linguagens, isso é percebido como um objeto, registro, estrutura, dicionário, tabela de hash;
- Uma lista ordenada de valores. Na maioria das linguagens, isso é percebido como um array, vetor, lista ou sequência.

Os arquivos Json diferem do XML ao empregarem pares de atributos e valores em vez de marcadores. Um arquivo Json típico possui uma estrutura organizada com delimitadores, usando chaves e colchetes em sequências para definir sua estrutura (JSON, 2023).

Devido à sua estrutura e notação simplificada, os arquivos Json são mais fáceis de compreender. Isso torna a organização dos dados mais clara para o usuário, o que, por sua vez, facilita o desenvolvimento e a integração de aplicativos. Além disso, esses arquivos são mais leves e compactos, ocupando menos espaço na memória. Como resultado, os dados podem ser transmitidos de maneira mais rápida e interpretados com facilidade pela aplicação (JSON, 2023).

2.8 *Yet Another Markup Language*

O *yet another markup language* (YAML) é uma linguagem de serialização de dados amplamente empregada na criação de arquivos de configuração. Em termos de acrônimos em inglês, pode tanto significar "*yet another markup language*" (ou seja, "mais uma linguagem de marcação") quanto "YAML ain't markup language" indicando que "YAML não é uma linguagem de marcação". Ambos deixam bem claro que a linguagem YAML é especificamente voltada para o tratamento de dados, não para utilização em documentos (YAML, 2023).

O YAML é conhecida por sua legibilidade e facilidade de compreensão. Além disso, ela pode ser integrada com outras linguagens de programação. Devido à sua flexibilidade e acessibilidade, essa linguagem é amplamente utilizado pela ferramenta Ansible para a criação de processos de automação, na forma de Ansible playbooks (YAML, 2023).

2.9 *Open Data Protocol*

Open Data Protocol (OData) é um protocolo de comunicação e padrão web que tem como objetivo facilitar a exposição e o consumo de dados por meio de serviços web RESTful. Ele foi desenvolvido pela Oasis Open, com o objetivo de tornar mais fácil o compartilhamento de dados e informações entre sistemas, aplicativos e dispositivos (OASIS, 2023).

Trata-se de uma série de convenções e regras que padronizam a forma como os dados são expostos e acessados por meio de serviços web RESTful. Isso inclui práticas para criar, recuperar, atualizar e excluir dados, bem como para consultar informações, além disso facilita a exposição de dados, permitindo que sistemas e aplicativos possam acessar informações por meio de *Uniform Resource Locator* (URL) padronizadas, tornando os dados acessíveis por meio de chamadas HTTP (OASIS, 2023).

O Padrão OData permite a interação com informações por meio de operações de consulta, filtragem, ordenação e paginação. Isso torna possível buscar dados específicos, atualizá-los e executar cálculos nos dados expostos. O padrão inclui metadados descritivos

que definem a estrutura dos dados expostos, essa definição de metadados é frequentemente realizada usando a linguagem CSDL (Common Schema Definition Language). Esses metadados facilitam a compreensão da estrutura dos dados disponíveis e a criação de consultas apropriadas (OASIS, 2023).

Para representar as informações, diferentes formatos de dados podem ser utilizados com OData, sendo o formato Json um dos mais comuns. Isso torna a comunicação de dados flexível e adaptável às necessidades dos aplicativos consumidores, tornando esse padrão amplamente utilizado em uma variedade de domínios, incluindo integração de sistemas, desenvolvimento de APIs, acesso a dados em serviços web e em casos de uso em que a padronização e a interoperabilidade são importantes. Ele é suportado por várias tecnologias e serviços, incluindo Microsoft Azure, SAP Gateway e muitos outros (OASIS, 2023).

2.10 Redfish

O Redfish é um protocolo aberto com conjunto de especificações que define uma forma simples e padronizada de gerenciar sistemas de computação e infraestrutura de *data centers*. Ele é fruto de uma iniciativa da *Distributed Management Task Force* (DTMF), uma organização de padrões de tecnologia da informação que se dedica ao desenvolvimento, definição e promoção de padrões de gerenciamento de sistemas de computação e infraestrutura.

O Redfish define uma API web que permite uma interação consistente e segura com servidores, sistemas de armazenamento, *switches* de rede e outros dispositivos de infraestrutura, tem o objetivo de substituir padrões de gerenciamento mais antigos, como o *Intelligent Platform Management Interface* (IPMI), que podem ser complexos e limitados em funcionalidade (DTMF, 2015).
Destaca-se algumas características que definem o modelo de dados do protocolo que são:

- Segue as convenções Odata Protocol;
- Utiliza *schemas* com texto legível a humanos;
- Pode ser utilizado tanto por aplicações ou *scripts*;
- Interoperabilidade através do uso de APIs em *Ain't Markup Language* (YAML) e *Common Schema Definition Language Odata* (CSDL Odata)
- Uso do protocolo *Hipertext Transfer Protocol Secure* (HTTPS) Json ou YAML.

No Padrão, cada URL corresponde a um recurso, serviço ou conjunto de recursos, representando uma funcionalidade ou outros elementos. Esses recursos são disponibilizados por meio de *schemas* que os clientes podem empregar para compreender a semântica de recursos específicos. Portanto, todos os recursos estão interconectados por meio de um serviço, que é representado por um ponto de entrada no *schema*, conhecido como o "*root*", definido pela URL: /redfish/v1 (DTMF, 2015). Por se tratar de uma API podemos chamar essas URLs de *endpoints*.

O Redfish define uma série de *endpoints* que representam recursos e funcionalidades específicas ao longo de sua hierarquia. Abaixo estão listados os principais *endpoints* do padrão:

- Systems: `.../redfish/v1/Systems`

  Fornece informações detalhadas sobre o sistema e recursos de *hardware*. São detalhes e dados, sobre Processador, Memoria, dispositivos de armazenamento, interfaces de rede, entre outros.

- Chassis: `.../redfish/v1/Chassis`

  Informa detalhes sobre o estado dos componentes contidos no chassis, como fontes de alimentação, coolers, unidades de armazenamento e outros elementos físicos relacionados a estrutura do chassis.

- Managers: `.../redfish/v1/Managers`

  Este *endpoint* representa os componentes de gerenciamento, como controladores de gerenciamento remoto, e fornece recursos relacionados à administração.

- Sessions: `.../redfish/v1/SessionService/Sessions`

  Gerencia as sessões de usuários ativos, incluindo detalhes de autenticação e estado das sessões.

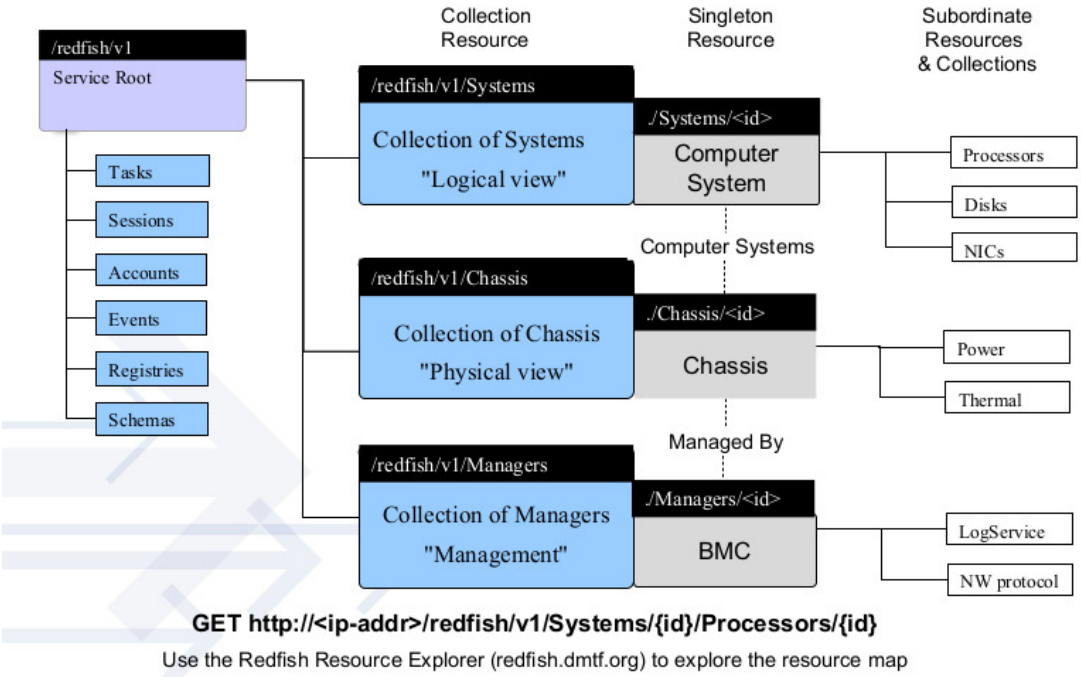
- Event Service: `.../redfish/v1/EventService`

  Acesso a eventos e notificações relacionados ao *hardware*, permitindo o monitoramento em tempo real de mudanças no sistema.

Os fabricantes de *hardware* podem personalizar criando extensões específicas e pode haver variações em alguns serviços, assim fornecendo alguma funcionalidade específica. A hierarquia exata e os endereços para acesso aos serviços podem variar dependendo do *hardware* específico e da implementação do Redfish. Cada *endpoint* fornece informações e funcionalidades específicas relacionadas ao gerenciamento de *hardware* e sistemas de computação.

A Figura 3 demostra um diagrama com representação *Resource map* desses principais *endpoints* do padrão.

Figura 3 – *Resource map.*



Fonte: Adaptado Redfish Specification:(DTMF, 2015)

2.10.1 Segurança

O esquema de segurança da comunicação do protocolo Redfish é projetado para garantir a confidencialidade, a autenticidade e a integridade das informações trocadas entre o cliente e o servidor Redfish. O protocolo utiliza uma série de mecanismos de segurança para alcançar esses objetivos (DTMF, 2015). Aqui estão alguns dos principais componentes do esquema de segurança da comunicação do Redfish:

Autenticação: O Redfish exige autenticação para verificar a identidade do cliente que está tentando acessar recursos. Isso pode ser feito por meio de vários métodos, como senhas, *tokens*, certificados digitais ou autenticação baseada em *hardware* (DTMF, 2015).

Autorização: Após a autenticação, o Redfish utiliza sistemas de autorização para determinar quais recursos e operações o cliente tem permissão para acessar e executar. Isso garante que os clientes tenham acesso apenas aos recursos apropriados com base em suas funções e permissões (DTMF, 2015).

Criptografia: A comunicação entre o cliente e o servidor Redfish é normalmente criptografada usando protocolos seguros, como HTTPS com *Transport Layer Security* (TLS) e *Secure Sockets Layer* (SSL). Isso protege os dados durante a transmissão,

garantindo que não possam ser interceptados ou lidos por terceiros não autorizados (DTMF, 2015).

Em resumo, o esquema de segurança da comunicação do protocolo Redfish é abrangente e projetado para proteger a integridade e a confidencialidade das informações, autenticar os usuários e garantir que somente os usuários autorizados tenham acesso aos recursos e operações adequados. A implementação correta e a configuração segura do Redfish são fundamentais

2.11 *Framework*

Um *framework* representa uma estrutura de suporte predefinida na qual outro projeto de software pode ser estruturado e desenvolvido, quando se considera o contexto do desenvolvimento de software. Ele pode compreender programas auxiliares, bibliotecas de código, linguagens de *script* e outros tipos de software que auxiliam na criação e integração de diversos componentes em um projeto de software (AREVALO, 2000).

2.12 Python e suas dependências

A linguagem Python foi concebida por Guido van Rossum em 1991, com a filosofia de valorizar o esforço do programador em detrimento do esforço computacional, priorizando a legibilidade do código por meio de uma sintaxe elegante, concisa e clara. Ela é uma linguagem de alto nível interpretada que combina orientação a objetos, programação procedural e funcional, caracterizando-se pela tipagem dinâmica, tipagem forte e suportando múltiplas plataformas. Uma característica notável da sintaxe única da linguagem Python é a definição de blocos de código por meio de identação, dispensando o uso de delimitadores como BEGIN, END ou { e }(PYTHON, 2023).

Dentre suas principais características, podemos destacar a licença de código aberto que é compatível com a *General Public License* (GLP), não impondo quaisquer restrições quanto ao seu uso e comercialização, além de possuir tipos de dados de alto nível que possibilitam a realização de operações complexas em uma única instrução, dispensando a obrigatoriedade de declarar variáveis ou parâmetros formais. Essa característica, juntamente com as várias outras mencionadas anteriormente, torna a linguagem altamente atraente, incentivando profissionais a estudá-la e usá-la como uma ferramenta essencial em seus trabalhos (PYTHON, 2023).

2.12.1 Flask

O Flask é classificado como um "*microframework*"para desenvolvimento web, devido ao seu suporte mínimo e simples para a operação de uma aplicação básica. Não oferece suporte para funcionalidades mais complexas, como abstração de bancos de dados e validação de formulários. No entanto, o núcleo do Flask é expansível, e há uma ampla variedade de bibliotecas de código aberto disponíveis para adicionar funcionalidades adicionais (FLASK, 2023). O Flask utiliza *decorators* como base para a definição das rotas da aplicação, o que torna a escrita de programas com Flask rápida e simples. No entanto, para aqueles que não estão familiarizados com o uso de decorators ou com o próprio *framework*, isso pode ser um pouco confuso.

2.12.2 Psutil

O PsUtil é uma biblioteca escrita em Python que oferece funções para monitorar status sobre os componentes do computador. Com o PsUtil, é possível obter detalhes sobre o *hardware* da máquina em execução, incluindo informações sobre memória RAM, disco rígido, CPU, rede, sistema operacional, entre outros (PSUTIL, 2023).

2.12.3 Subprocess

O Subprocess é uma biblioteca escrita em Python que fornece uma maneira de interagir com processos externos, permitindo que você inicie, controle a comunicação com eles e capture sua saída. Permitindo a execução de comandos ou programas externos a partir de um código em Python, por meio de comandos de terminal *bash*. A principal função da biblioteca subprocess é criar e gerenciar subprocessos a partir do código Python (SUBPROCESS, 2023).
Permitindo usar as funções e classes fornecidas por essa biblioteca para:

- Executar comandos do sistema operacional;
- Comunicar-se com o processo filho, enviando dados para sua entrada padrão *stdin* e capturando sua saída padrão *stdout* e erro padrão *stderr*;
- Aguardar que o processo filho termine sua execução;
- Lidar com exceções e erros que podem ocorrer durante a execução do processo filho.

A utilização de subprocessos é útil em muitos cenários, como automação de tarefas do sistema, integração com programas externos, execução de *scripts* e muito mais. Por exemplo, você pode usar o módulo subprocess para executar um comando do sistema

operacional, como a criação de um novo processo para rodar um programa em Python a partir do seu *script* Python principal. Isso permite que você integre funcionalidades de outros programas e sistemas em seu próprio código Python (SUBPROCESS, 2023).

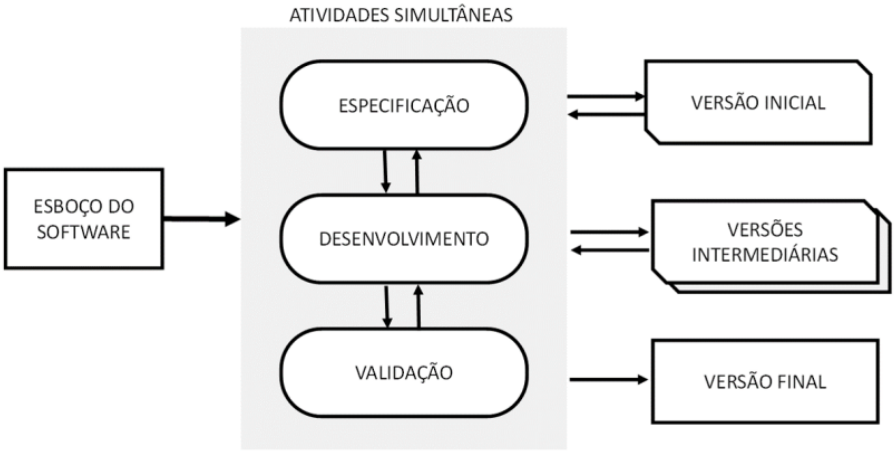
3 Metodologia e Materiais

Este capítulo foi reservado para demostrar como o trabalho foi projetado, a metodologia utilizada no desenvolvimento, bem como apresentar o ambiente de desenvolvimento e outras tecnologias a ser utilizadas.

3.1 Metodologia

Foi utilizado nesse projeto uma adaptação da metodologia de desenvolvimento incremental, com características específicas para este projeto. Consistindo em pequenas entregas de versões que foram sendo avaliadas e revisadas pelo cliente que diferentemente do modelo tradicional, esse papel foi desempenhado pelo professor. A cada versão o projeto era melhorado e incrementando outras funções e modificações. Como se pode ver na Figura 4.

Figura 4 – Modelo Incremental.



Fonte: Adaptado (SOMMERVILLE, 2011)

Esse modelo foi escolhido por conta do pouco tempo para desenvolver uma aplicação deste tipo. Ao invés de especificar e desenvolver tudo de uma só vez, com este modelo foi implementado pequenos pedaços do software, e ao longo do semestre era apresentado o que estava sendo realizado ao professor, a cada consulta era exposto as dificuldades e possíveis soluções eram apresentadas.

A avaliação contínua do projeto permitiu a identificação eficiente dos desvios em relação aos objetivos estabelecidos. Os ajustes foram realizados em resposta a desafios emergentes, garantindo a eficiência e alinhamento com as expectativas do projeto.

3.2 Raspberry Pi

Nesta seção, abordaremos detalhadamente as características do Raspberry Pi utilizado na implementação deste projeto, destacando suas especificações técnicas e funcionalidades de *hardware* e software além dos periféricos utilizados.

Figura 5 – Raspiberry Utilizado.

Fonte:Autoria própria

1. Modelo: Raspberry Pi 3 Model B Rev 1.2
2. *Hardware*:
   - Porcessador: Broadcom BCM2837;
   - Core: ARMv7 rev 4;
   - Memoria: 1 GB de memória RAM (LPDDR2);
   - GPU: `VideoCore IV de 400 MHz`.
3. Sistema Operacional:
   - OS : Raspbian GNU/Linux;
   - versão: 12.1;
   - Codename: bookworm;
   - Kernel: Linux raspberrypi 6.1.0-rpi4-rpi-v7.

4 . Armazenamento:

   - SD Card;
   - Marca: Hikivision;
   - Capacidade: 32 Gb;
   - Classe: 10.

3.3 Desafios e Decisões de projeto

Algumas decisões para projeto foram necessárias durante a fase de planejamento e desenvolvimento da aplicação, várias dificuldades surgiram devido às características singulares do *hardware* do Raspberry Pi, que diferem substancialmente de um computador comum. Algumas dessas dificuldades estão relacionadas à ausência alguns componentes no *hardware* do dispositivo em questão, além dos desafios quanto à captura das métricas. Abordaremos essas questões e as decisões tomadas para superá-las e seguir com a implementação.

3.3.1 Dificuldades decorrentes da ausência de BIOS

.

A primeira das principais dificuldades encontrada no desenvolvimento da API baseada no padrão Redfish foi a ausência de uma BIOS nesse dispositivo. Componente que é parte fundamental em computadores tradicionais, pois fornece uma camada de abstração de *hardware* além de permitir o acesso a informações do sistema, a principal consequência dessa ausência, é a incompatibilidade da utilização dos comandos `DMI DECODE`, assim a obtenção de informações de sistema e métricas torna-se mais desafiadora sendo necessário buscar por outras alternativas, consequentemente dificultando o trabalho e limitando o escopo do projeto, pois alguns *endpoints* do padrão original interagem diretamente com a BIOS, como: `/redfish/v1/Managers`. Para superar essa dificuldade, foi preciso pensar em soluções específicas para o Raspberry Pi, utilizando comandos via terminal como, `lshw`, `vcgencmd` e leitura de arquivos descritores, usando `subprocess` para integrar ao código do projeto, possibilitando assim a obtenção de informações dos sensores, leitura de dados diretamente de componentes do *hardware*.

3.3.2 Dificuldades com o padrão Redfish

.

Além da ausência de BIOS, outra dificuldade encontrada no desenvolvimento da aplicação foi a falta de orientações na documentação do padrão Redfish a respeito da captura das métricas e dados no sistema. O padrão Redfish define um conjunto de *endpoints* e recursos padrão para a gestão de sistemas, mas não especifica exatamente como pode ser realizada a captura dessas métricas no dispositivo, deixando essa responsabilidade a cargo dos desenvolvedores que utilizam o padrão.

Isso levou a um desafio adicional, e para superar essa dificuldade, foi necessário fazer uma extensa pesquisa e realizar experimentos para encontrar métodos e melhores práticas para coletar esses dados e métricas no Raspberry Pi, como utilizar biblioteca

`psutil` e `subprocess` para integrar comandos diretamente do terminal com código fonte do projeto. Os comandos mais importantes foram `lshw`, `vcgencmd` e leitura de arquivos descritores.

3.3.3 Decisões tomadas

Diante das dificuldades, foram necessárias as seguintes adequações:

1 Redução do escopo do projeto, devido estrutura de *hardware* do Raspberry ser reduzida e não possuir alguns componentes como:

1.1 Por não possuir BIOS não foi possível implementar o *endpoint* `.../Managers`, pois o mesmo interage diretamente neste componente para gerenciamento.

1.2 O *endpoint* `.../Chassis` teve grande parte de suas funções removidas do escopo deste projeto. Como não possui um gabinete como os computadores convencionais, não faria sentido implementar leitura das métricas das *fans*, temperatura do chassis, informações sobre montagem, dimensões e localização do chassis geograficamente e em posição de *rack*.

1.3 O *endpoint* `.../AccountService` é o serviço de gerência de credenciais de usuário Redfish, como neste trabalho não implementei a parte de segurança não foi necessário implementá-lo.

2 Limitar o projeto somente para leitura de informações, foi uma decisão que mudou as características da API de administração de dispositivos para somente monitoramento, pois implementar as funções de administração necessitaria de mais tempo, além da necessidade de implementação de um número muito maior de funções. Estas pendências ficaram registradas como sugestões de melhorias futuras;

3 `readings.py` Concentrar todas as funções responsáveis por leitura das métricas e aquisição dos dados, em um único arquivo com objetivo de agilizar o processo de implementação, pois dessa forma deixa o código inteligível. Além de deixar as funções validadas e testadas para mais rapidamente serem implementadas, em um processo de apenas preencher lacunas seguindo os modelos de Json que a DTMF disponibiliza na documentação. Além disso foi inserido *docstrings* nas funções com objetivo de deixar uma breve descrição de sua utilidade.

As adequações realizadas no projeto foram fundamentais para superar as dificuldades inicialmente encontradas e abrir caminho para o desenvolvimento do projeto de uma API para o Raspberry Pi que fosse funcional com um grande número de métricas, tornando-a uma ferramenta eficaz para o monitoramento remoto desses dispositivos.

3.4 Ambiente de Desenvolvimento

O ambiente de desenvolvimento desempenha um papel fundamental na eficácia e eficiência do processo de criação da aplicação. No contexto deste projeto, o ambiente de desenvolvimento foi cuidadosamente configurado para possibilitar trabalhar com a aplicação diretamente no Raspberry Pi 3 a partir de um computador pessoal. A seguir, descreveremos os componentes chave desse ambiente:

1 Visual Studio Code (VS Code): O Visual Studio Code é um ambiente de desenvolvimento integrado (IDE) de código aberto amplamente utilizado, conhecido por sua extensibilidade e capacidade de suportar várias linguagens de programação. Sua interface de usuário amigável e a integração de uma ampla variedade de extensões tornam-no uma escolha popular entre desenvolvedores;

2 Acesso SSH: Para estabelecer uma conexão segura com o Raspberry Pi 3 a partir do ambiente de desenvolvimento, foi utilizado o protocolo SSH. Permite a comunicação criptografada e segura entre o computador pessoal e o Raspberry Pi, proporcionando um meio confiável para acessar e controlar o dispositivo remotamente;

3 Configuração do Raspberry Pi 3: O Raspberry Pi 3 foi devidamente configurado para aceitar conexões SSH. Isso envolve a ativação do SSH no Raspberry Pi, a configuração de senhas ou chaves de autenticação, e a disponibilização de informações de endereço IP para a conexão.

O processo de desenvolvimento envolve a criação, edição e depuração de código-fonte no Visual Studio Code, diretamente na plataforma de destino. Abaixo, descrevemos as etapas envolvidas na configuração do ambiente de desenvolvimento e no processo de programação:

1 Instalação do Visual Studio Code: O Visual Studio Code foi instalado no computador pessoal a partir do site oficial, e extensões relevantes para o projeto foram instaladas, como as extensões para suporte à python;

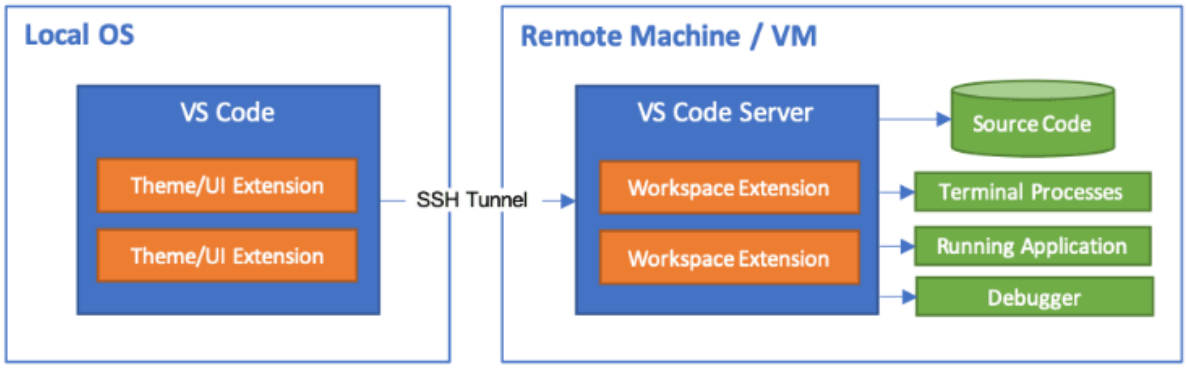
2 Conexão SSH ao Raspberry Pi 3: Foi configurada uma conexão SSH entre o Visual Studio Code e o Raspberry Pi 3, fornecendo as informações de endereço IP e credenciais necessárias usuário e senha para estabelecer a comunicação segura, nos retângulos 1 e 2 da Figura 6 mostra o status da conexão quando estabelecida;

3 Desenvolvimento e Testes: A programação da aplicação ocorreu no Visual Studio Code, aproveitando as funcionalidades de edição de código, depuração e controle de

versão oferecidas pela IDE. Os testes da aplicação foram realizados diretamente no Raspberry Pi 3;

4 Simultaneamente ocorrendo em paralelo a implementação tínhamos os testes: Após o desenvolvimento de cada etapa, a aplicação que já estava implantada no Raspberry Pi 3, era executada no seu ambiente real, assim reduzindo etapas e minimizando tempo do projeto.

Figura 6 – Ambiente de desenvolvimento.



Fonte:(MICROSOFT, 2023)

Este ambiente de desenvolvimento permitiu uma abordagem eficaz e conveniente para o desenvolvimento da aplicação do projeto no Raspberry Pi 3, garantindo que a programação e os testes pudessem ser realizados de forma consistente, rápida e segura. A combinação do Visual Studio Code com a conexão SSH ofereceu um conjunto de ferramentas necessárias para criar, programar e editar as funções da aplicação com sucesso. Ocorrendo diretamente no Raspberry Pi 3, esse processo foi muito simples agilizando o trabalho, assim demonstrando que foi uma peça fundamental para um processo robusto e eficiente ao longo deste projeto.

3.4.1 Instalação das dependências da API

Nesta seção, será descrito as etapas necessárias para a configuração e instalação das dependências utilizadas no desenvolvimento do projeto bem como a descrição de sua finalidade, versão e compatibilidade.

Para o desenvolvimento do projeto foi utilizado Python na versão 3.11.2, envolvendo a utilização de diversas bibliotecas e pacotes essenciais para o seu funcionamento

adequado. Algumas dependências já fazem parte do pacote original Python ou já estão inclusas no sistema operacional, no entanto houve a necessidade de instalação de alguns pacotes, e para facilitar esse processo foi utilizado o Pip versão 23.0.1, que é gerenciador de pacotes Python. Segue a lista das principais dependências e a descrição de como configurá-las:

- Flask, foi o framework escolhido para o desenvolvimento da API. Para instalá-lo, foi utilizado a ferramenta Pip, que gerencia as bibliotecas Python. A instalação do Flask é bem simples e pode ser realizada com o seguinte comando:

  ```
  pip install Flask
  ```

  Digitando `flask` no terminal, verificamos se a instalação foi realizada corretamente.

A coleta de informações de *hardware* é fundamental para o monitoramento remoto do Raspberry Pi, pois fornece dados e métricas essenciais para a exposição de informações da API, para isso foi necessário utilizar algumas ferramentas como:

- lshw é uma ferramenta de linha de comando que fornece informações detalhadas sobre o *hardware* do sistema. Ela é utilizada para coletar dados sobre os componentes do Raspberry Pi, como processador, memória, dispositivos de armazenamento e outros. A instalação da ferramenta Lshw pode ser realizada com o seguinte comando:

  ```
  sudo apt-get install lshw
  ```

  a versão utilizada no projeto é `02.19.git.2021.06.19.996aaad9c7-2`, com o comando lshw no terminal.

- Psutil é uma biblioteca Python que fornece uma interface simples e consistente para acessar informações do sistema e gerenciar processos. Ela foi essencial para a coleta de dados em tempo real sobre o desempenho do sistema, incluindo informações sobre CPU, memória, disco e rede. A instalação da biblioteca Psutil pode ser realizada com o seguinte comando:

  ```
  pip install psutil
  ```

  A biblioteca Psutil desempenha um papel crucial na implementação de recursos de monitoramento da API, permitindo a coleta e a exposição de métricas de sistema em tempo real. Foi utilizado neste projeto a versão 5.9.4.

Módulo subprocess em Python, que nesse projeto foi muito importante pois com ele foi permitindo a execução de comandos do *shell* a partir do código Python. Resultando na captura de muitos dados do sistema, não foi necessário a sua instalação pois já faz parte do pacote python instalado. Suas principais funcionalidades serão descritas a seguir:

1 check_output:

1.1 Função: Executa um comando no *shell* e captura a saída desse comando.

1.2 Utilização: necessário para executar um comando e obter a saída resultante no contexto do programa Python.

2 `Popen`:

2.1 Classe: Permite a criação de objetos que representam processos em execução.

2.2 Utilização: foi útil para execução assíncrona de comandos, interação com a entrada/saída do processo e controle mais avançado sobre o fluxo do programa.

3 `call`:

3.1 Função: Executa um comando e espera até que a execução seja concluída.

3.2 Utilização: Simplesmente executa um comando e aguarda seu término, retornando o código de retorno do processo.

4 `DEVNULL`:

4.1 Função: Representa um descritor de arquivo para o dispositivo nulo (no qual os dados são descartados).

4.2 Utilização: Pode ser usado como argumento para redirecionar a saída (stdout) ou entrada (stdin) de um comando para o nada.

5 `STDOUT`:

5.1 Função: Representa um descritor de arquivo para a saída padrão.

5.2 Utilização: Pode ser usado como argumento para redirecionar a saída (stderr) de um comando para a saída padrão.

6 PIPE:

6.1 Função: Permite a comunicação entre o processo Python e o processo filho através de uma tubulação. Constante usada como valor para indicar que uma tubulação `pipe` deve ser criada ao abrir um novo processo.

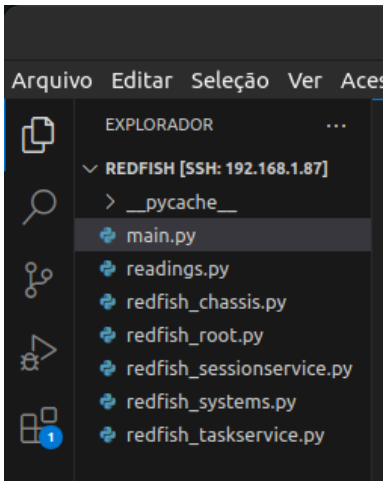
Nesta seção, foi detalhado todas as etapas de configuração e instalação das dependências necessárias para o desenvolvimento do projeto de uma API Redfish baseada

em Python com o framework Flask. Além disso, explicamos como instalar algumas das ferramentas nescessarias como Lshw, biblioteca Psutil entre outras, que são fundamentais para a obtenção dos dados e captura das métricas do monitoramento remoto de dispositivos Raspberry Pi. Com essas dependências devidamente configuradas e testadas, estará tudo pronto para implementação da aplicação.

3.5 Desenvolvimento

O Desenvolvimento da API foi cuidadosamente organizado para garantir inteligibilidade, manutenibilidade e escalabilidade do código. Começando pela a escolha do Python para a implementação do projeto, e como já descritos na sessão anterior o framework Flask foi utilizado para criar a estrutura da aplicação. O projeto está dividido em sete arquivos, cada um com funções e responsabilidades específicas, conforme mostra na Figura 7. Essa divisão ajuda a manter o código organizado e a facilitar o desenvolvimento colaborativo. A seguir, apresentamos uma breve descrição de cada arquivo:

Figura 7 – Estrutura de arquivos do projeto.



Fonte: Autoria própria

3.5.1 Arquivo `main.py`

Principal arquivo do projeto, é responsável por definir o roteamento dos *endpoints* para respectivas funções específicas, e cada uma dessas funções vai retornar o Json correspondente, ou seja cada serviço vai ter um Json associado, que são definidos nos arquivos auxiliares redfish_root, redfish_systems, redfish_chassis, redfish_taskservice e redfish_sessionservice. Será descrito as principais funcionalidades e rotas da aplicação.

Primeiramente é definida uma rota ou *endpoint* padrão http://localhost:5003/ quando acessado apenas aparece uma mensagem de boas vindas para o usuário, "Bem Vindo a RedfishPi", serve apenas para mostrar que aquela URL direciona para a API, a partir dela são derivadas todas as outras rotas relacionadas à API, conforme mostra o trecho de código da Figura 8.

Figura 8 – Rota Inicial.

```
12  @app.route('/')
13  def index():
14      return'Bem Vindo a RedfishPi'
15
16  @app.route('/redfish/v1/', methods=['GET'])
17  def get_v1():
18      return redfish_root.get_redfish_v1()
```

Fonte: Autoria própria

Assim como no padrão Redfish a rota de partida na API é `/redfish/v1/`, rota para acessar a raiz do serviço da API. Conforme a tabela abaixo onde está os principais `endpoints` e suas descrições :

Tabela 2 – Principais `endpoints` e Descrição

Endpoint	Descrição
/redfish/v1/	Acessa a raiz do serviço Redfish.
/redfish/v1/Chassis	Acessa informações do chassi.
/redfish/v1/TaskService	Acessa informações tarefas do sistema.
/redfish/v1/SessionService	Acesso a informações sessões de usuários ativas.
/redfish/v1/Systems	Acesso a informações específicas do sistema

Fonte:Autoria própria

Dependendo do *endpoint* ele pode dar acesso a outro *endpoint*, cada um com sua rota definida no seu respectivo arquivo, existindo rotas estáticas e rotas dinâmicas, isso por conta de serviços ou recursos que não são fixos, como interfaces ethernet, dispositivos de armazenamento e as sessões de usuários ativas. As rotas dinâmicas são capazes de lidar com vários sub recursos, como diferentes interfaces ethernet e dispositivos de armazenamento, usando funções dinâmicas,

Em resumo, no arquivo `main.py` além de criar a aplicação, Ele fornece rotas para os *endpoints*, e como algumas rotas dinâmicas para lidar com sub recursos específicos, cada *endpoints* presente na tabela 02 corresponde a uma função que é chamada chamada para fornecer os dados e respostas adequadas que nesse caso está no formato Json, cada módulos está separado em seus respectivos arquivos `redfish_chassis.py`, `redfish_systems.py` , `redfish_sessionservice.py`, `redfish_taskservice.py`.

3.5.2 Arquivo `root.py`

Este é o módulo mais simples da aplicação, porem é de extrema importância para o seu funcionamento, ele retorna um Json contendo os principais *endpoint*, conforme mostra na Figura 9.

Figura 9 – `redfish_root.py`

```python
import readings

def get_redfish_v1():
    redfish_v1 = {
        "@odata.context": "/redfish/v1/$metadata#ServiceRoot",
        "@odata.type": "#ServiceRoot.1.0.0.ServiceRoot",
        "@odata.id": "/redfish/v1/",
        "Id": "RootService",
        "Name": "Root Service",
        "RedfishVersion": "1.0.0",
        "UUID": readings.system_uuid(),
        "Chassis": {
            "@odata.id": "/redfish/v1/Chassis"
        },
        "TaskService": {
            "@odata.id": "/redfish/v1/TaskService"
        },
        "SessionService": {
            "@odata.id":  "/redfish/v1/SessionService"
        },
        "Systems": {
            "@odata.id": "/redfish/v1/Systems"
        },
    }

    return redfish_v1
```

Fonte: Autoria própria

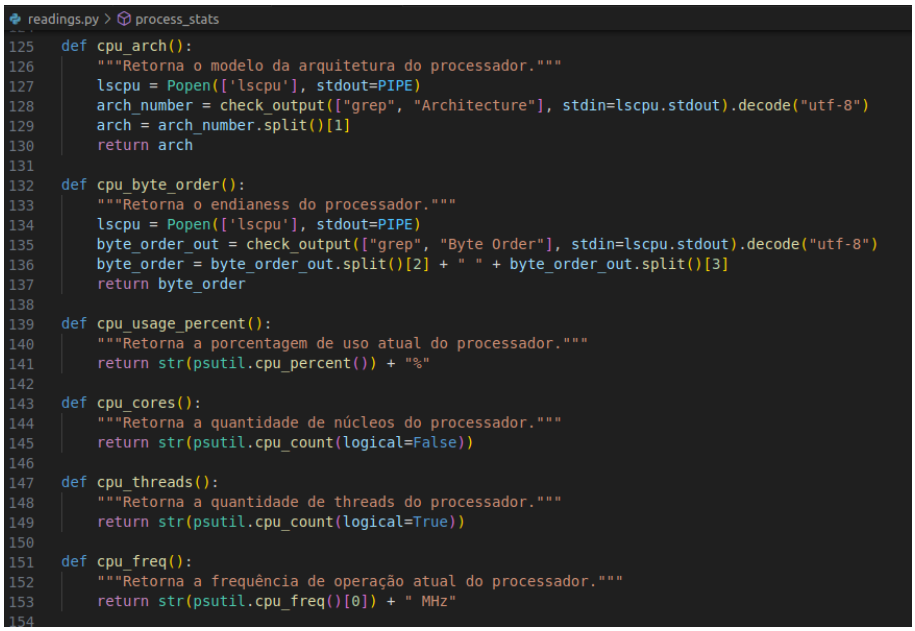
- "Id": Uma identificação para o recurso, neste caso, "RootService".
- "Name": O nome do serviço, neste caso, "Root Service".
- "RedfishVersion": A versão do Redfish utilizada, que é "1.0.0".
- "UUID": O identificador exclusivo do sistema, obtido chamando a função, `readings.system_uuid()` que é uma função pertencente ao modulo `readings.py`.
- "Chassis:", "TaskService:", "SessionService:", e "Systems": São links para outros recursos da API Redfish, representados por seus respectivos *endpoints*.

Por fim temos `get_redfish_v1_compositionService()`: A função simplesmente chama `get_redfish_v1()` e retorna o mesmo objeto Json representando o serviço raiz Redfish.

3.5.3 Arquivo `readings.py`

Esse é o maior modulo da aplicação com 568 linhas de código, ele concentra 65 funções, sendo que destas 60 são responsáveis por realizar as capturas das métricas da aplicação, todas as funções possuem uma descrição em formato de `docstrings` aumentando a inteligibilidade do código, facilitando a vida de quem pretende utilizar as funções no futuro. A lista com todas as funções e suas respectivas descrições está no apêndice desse trabalho.

Figura 10 – `readings.py`



Fonte: Autoria própria

Na Figura 10 podemos ver como a captura das métricas e dados do *hardware* e os diversos artifícios como comandos via terminal utilizando Subprocess e lshw, bibliotecas como psutil e outras dependências já apresentadas na sessão anterior.

3.5.4 Arquivo `redfish_taskservice.py`

O arquivo taskservice.py encapsula lógica relacionada ao serviço de tarefas, fornecendo uma representação básica do serviço `get_taskService()`, no geral ele cria e retorna um Json representando o serviço de tarefas do *hardware* monitorado. na Figura 11 segue a sua descrição.

Figura 11 – Json task service


```
      },
      {
        "@odata.id": "/redfish/v1/TaskService/2379",
        "Process Name": "[kworker/2:2-events]"
      },
      {
        "@odata.id": "/redfish/v1/TaskService/2384",
        "Process Name": "[kworker/u8:0-events_unbound]"
      },
      {
        "@odata.id": "/redfish/v1/TaskService/2389",
        "Process Name": "[kworker/3:2-events_power_efficient]"
      }
    ],
    "Members@odata.count": 148
  },
  "Modified": "2012-03-07T14:44",
  "Name": "Task Collection"
}
```


Fonte: Autoria própria

- `"@odata.id":` "`/redfish/v1/TaskService/2384`"
  Endpoint contendo mais informações sobre processo em questão, é acessado através do ID, que é o PID do processo, neste exemplo é 2384, nesta tela é possível ver uma URL para cada processo.

- "Process Name:" "`[kworker/u8:0-events_unbound]`"
  Nome do processo em questão.

- "Members@odata.count": "148"
  Total de processos na maquina naquele instante.

- "`Modified:`" "`2012-03-07T14:44`"
  Tempo de funcionamento do sistema.

- "Name": "`Task Collection`"
  Nome do serviço.

Acessando `/redfish/v1/TaskService/2099` além das informações mostradas anteriormente sobre esse processo, temos informações especificas deste processo, como:

- "StartTime": "2023-11-09T23:44:07"

  hora que o processo foi iniciado.

- "TaskState": "Idle kernel thread"

  estado do processo.

Como os processos são voláteis , os *endpoints* em questão são dinâmicos.

3.5.5 Arquivo `redfish_Chassis.py`

Assim como arquivo no anterior o `arquivo redfis_chasis.py` encapsula lógica relacionada ao informações sobre o chassis, como se trata de um raspberry pi, essas informações foram bastante restritas, uma vez que esse dispositivo não tem chassis ou o convencional gabinete como em computadores e servidores, assim foi necessário limitar e adaptar as funcionalidades para o contexto deste dispositivo. Fornecendo algumas respostas básicas do sistema como veremos a seguir na Figura 12, Json com as informações acessadas através desse *endpoint*.

Figura 12 – Json Chassis


```
{
  "@Redfish.Copyright": "Copyright 2014-2021 DMTF. For the full DMTF copyright policy, se
  "@odata.id": "/redfish/v1/Chassis/f97532826de34a7893931d1a591b3865",
  "@odata.type": "#Chassis.v1_15_0.Chassis",
  "Id": "f97532826de34a7893931d1a591b3865",
  "IndicatorLED": "On",
  "Links": {
    "ComputerSystems": [
      {
        "@odata.id": "/redfish/v1/Systems/f97532826de34a7893931d1a591b3865"
      }
    ]
  },
  "Manufacturer": "Embest",
  "Model": "Raspberry Pi 3 Model B Rev 1.2",
  "Name": "Raspberry Pi 3",
  "PowerState": "On",
  "Sensors": {
    "@odata.id": "/redfish/v1/Chassis/f97532826de34a7893931d1a591b3865/Sensors"
  },
  "SerialNumber": "0000000061afc4aa",
  "Status": {
    "Health": "OK"
  },
  "ThermalSubsystem": {
    "@odata.id": "/redfish/v1/Chassis/f97532826de34a7893931d1a591b3865/ThermalSubsystem"
  }
}
```


Fonte: Autoria própria

Este formato de resposta Json tem uma visualização bem simples para compreensão do usuário, oferecendo uma estrutura organizada e hierárquica para informações sobre o chassis, nele temos a informação para acesso as subdivisões dessa categoria, chamada de Sensors onde são extraídas as informações dos poucos sensores que o dispositivo possui como de temperatura, e tensões da placa, também temos as informações de ThermalMetrics que foi bastante limitada pois o dispositivo em questão não possui coolers no equipamento e por não se tratar de um chassis de verdade onde naturalmente haveria vários. Logo abaixo temos algumas das principais informações coletadas por esse *endpoint*.

- Id: “f97532826de34a7893931d1a591b3865”
  Número de série do Raspberry coletado utilizando a função.

- IndicatorLED: “On”
  indica se o led de funcionamento do equipamento está ligado, essa informação é coletada utilizando a função `readings.power_led()` que faz a leitura através do arquivo `/sys/class/leds/PWR/brightness` utilizando subprocess para coletar essa informação.

- “Manufacturer: “Embest”
  Informação sobre quem é fabricante do *hardware*, é coletada utilizando a função `readings.manufacturer()` que lê o arquivo `/proc/cpuinfo` essa tarefa é realizada utilizando `subprocess`.

- Model: “Raspberry Pi 3 Model B Rev 1.2”
  Modelo e revisão do Raspberry, que é coletado pela da função `board_name()`, essa função acessa coleta essa informação direto do arquivo `/sys/firmware/devicetree/base/mode`
  utilizando `subprocess`.

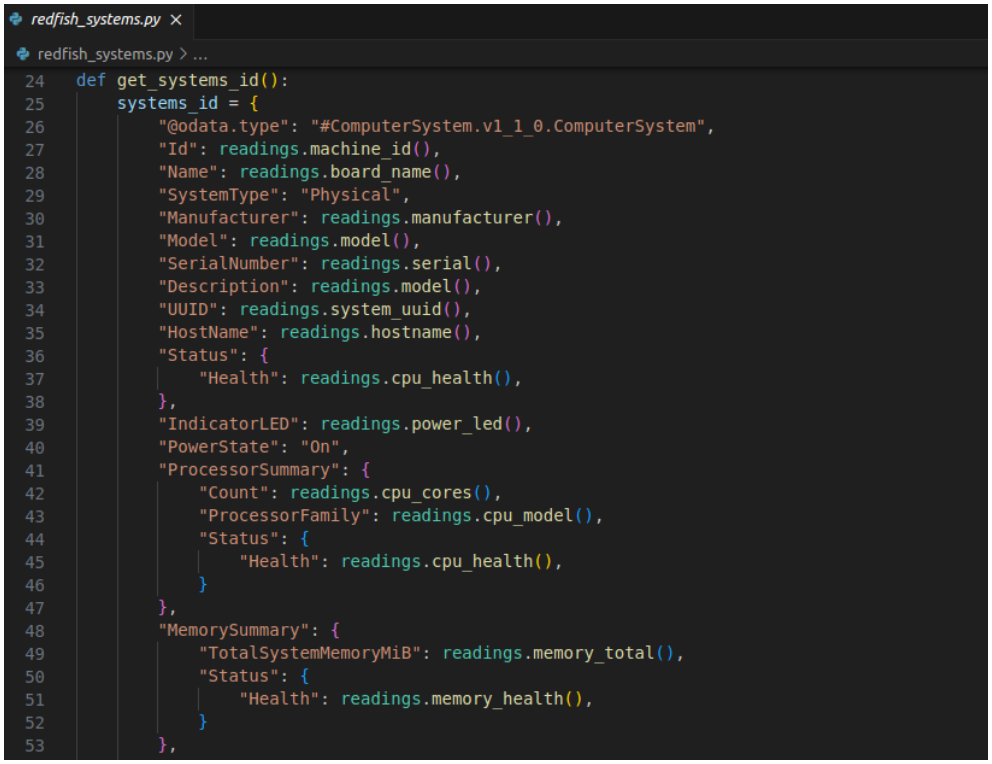
- SerialNumber: “0000000061afc4aa”
  Numero de série do equipamento que é coletado pela da função `readings.serial()`,
  essa função acessa coleta essa informação direto do arquivo `/sys/firmware/devicetree/base/`
  isso utilizando `subprocess`.

- Status: “Health”: “OK”
  Com base nas informações de tensão da placa indica se o equipamento está “saudável”, essa informação é gerada na função `readings.cpu_health()` que faz uma análise da tensão do processador que utilizando o comando`measure_volts` do `vcgencmd`, é feita uma série de comparações e se o valor estiver dentro do aceitável em relação aos dados do fabricante e retornada a informação “`ok`”ou caso contrario a informação “`WARRING`”.

3.5.6 Arquivo `redfish_Systems.py`

Arquivo que concentra maior número de informações coletadas do *hardware* do Raspberry, consequentemente o que possui uma dependência da utilização das funções do modulo `readings.py`, conforme podemos ver em um pequeno trecho do código mostrado na Figura 13, além disso ele é responsável pelas implementações do *endpoint* “`.../Systems/`” gerando uma ramificação com 4 outros importantes *endpoints* são eles:

Figura 13 – Json systems



Fonte: autoria própria

1 "/redfish/v1/Systems/id/EthernetInterfaces"
endpoint dinamico que apresenta informações dos dispositivos de rede ativos no momento;

2 "/redfish/v1/Systems/id/Processors"
endpoint que exibe informações do processador como, modelo, arquitetura, número de núcleos, clock, temperatura, tensão de consumo, estado de saúde;

3 "/redfish/v1/Systems/id/Memory"
endpoint que exibe informações relativas a memória, que no caso dessa API foram direcionadas para dar um enfoque nas estatísticas de uso. Como "*Available*", "*Buffers*", "*Cached*", "*Clock*", "*Free*", "*GPUMiB*", "*Memory Used*", "*Percent Used*", "*SystemMiB*", "*TotalMiB*";

4 "/redfish/v1/Systems/id/SimpleStorage"
endpoint dinâmico que apresenta informações dos dispositivos de armazenamento em razão da sua disponibilidade.

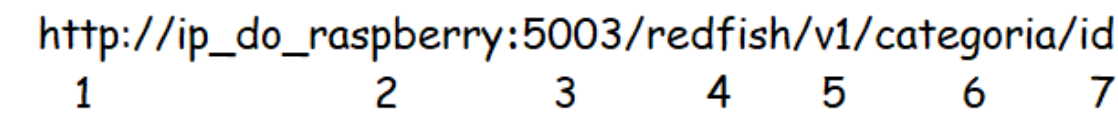
3.6 Composição da API

Nesta sessão vamos mostrar as características da API implementada, além de fazer as devidas considerações em relação ao padrão Redfish original, mostrando os principais aspectos e vantagens da utilização de cada característica.

3.6.1 Composição dos *endpoints*

Os *endpoints* desta aplicação são compostos de uma forma bem intuitiva. Sendo está a principal característica adotada do padrão redfish original. Com base no esquema representado pela Figura 14 vamos demostrar como são compostos os *endpoint* e o significado de cada elemento:

Figura 14 – Composição dos *Endpoints*



Fonte: Autoria própria

1 HTTP Protocolo utilizado pela API para enviar as solicitações;

2 ip_do_Raspberry, endereço atribuido ao dispositivo, essa configuração é realizada diretamente no aparelho, também existe a opção de ser acessado por `localhost` caso esteja utilizando o navegador do próprio Raspberry;

3 Porta que a aplicação está utilizando, essa porta é definida no arquivo `main.py`;

4 Identifica o padrão Redfish, é uma especificação do padrão;

5 Indica a versão do padrão Redfish utilizada;

6 Categorias que podem ser Systems, TaskService, SessionService, Chassis. Além das subcategorias existentes;

7 ID, é o número de série do aparelho, item muito importante pois é único para cada dispositivo, assim em um cenário de gerência de vários dispositivos ele permitirá identificar o aparelho desejado.

3.6.2 Composição do Json

Segundo (DTMF, 2015) o padrão Redfish utiliza representações do padrão OData que são convertidos para o Json. OData é um padrão do setor que encapsula as práticas recomendadas para serviços RESTful e fornece interoperabilidade entre serviços de diferentes tipos. Nesta sessão vamos demostrar como é composta a estrutura básica de um Json e suas diferenças para padrão original, a partir do exemplo da FIGURA 15.

Figura 15 – Json resposta *endpoint* root `/redfish/v1`.

```
{
    "@odata.context": "/redfish/v1/$metadata#ServiceRoot",
    "@odata.id": "/redfish/v1/",
    "@odata.type": "#ServiceRoot.1.0.0.ServiceRoot",
    "Chassis": {
        "@odata.id": "/redfish/v1/Chassis"
    },
    "Id": "RootService",
    "Name": "Root Service",
    "RedfishVersion": "1.0.0",
    "SessionService": {
        "@odata.id": "/redfish/v1/SessionService"
    },
    "Systems": {
        "@odata.id": "/redfish/v1/Systems"
    },
    "TaskService": {
        "@odata.id": "/redfish/v1/TaskService"
    },
    "UUID": "3c215345-c59c-4a79-941a-ab121e090a42"
}
```

Fonte: Autoria própria

1 “@odata.context”: Fornece o contexto do documento json, indicando a localização do esquema de metadados que define a estrutura dos dados no documento. Importante para interpretar corretamente a estrutura e os tipos de dados utilizados no documento Redfish.

2 “@odata id ”: Indica a URL do recurso ao qual o Json se refere. Neste caso, refere-se ao serviço raiz Redfish Importante para navegar e acessar diretamente o recurso associado a este Json.

3 “@odata.type ”: Indica o tipo do recurso Redfish ao qual Json pertence. Neste caso, serviço raiz. Permite identificar e validar o tipo de recurso sendo manipulado.

4 “Chassis”: Contém a URL do recurso de chassis associado ao serviço raiz. Facilita a navegação para informações específicas sobre os chassis do sistema.

5 “ID”: Um identificador único para o serviço raiz. Pode ser usado para referenciar o serviço raiz de forma única.

6 “Name”: O nome atribuído ao serviço raiz. Fornece uma descrição legível para identificar o serviço raiz.

7 “ RedfishVersion”: Indica a versão do protocolo Redfish que está sendo usada. Permite verificar a conformidade e a compatibilidade da implementação Redfish;

8 “SessionService”: Contém a URL do recurso do serviço de sessão associado ao serviço raiz. Permite acesso e manipulação de sessões dentro do contexto do serviço raiz.

9 “Systems”: Contém a URL do recurso de sistemas associado ao serviço raiz. Facilita a navegação para informações específicas sobre os sistemas do sistema.

10 “TaskService:”: Contém a URL do recurso do serviço de tarefas associado ao serviço raiz. Permite o gerenciamento de tarefas relacionadas ao serviço raiz.

11 “UUID ”: Um identificador único universal (UUID) associado ao serviço raiz. Pode ser usado como um identificador único global para referenciar o serviço raiz.

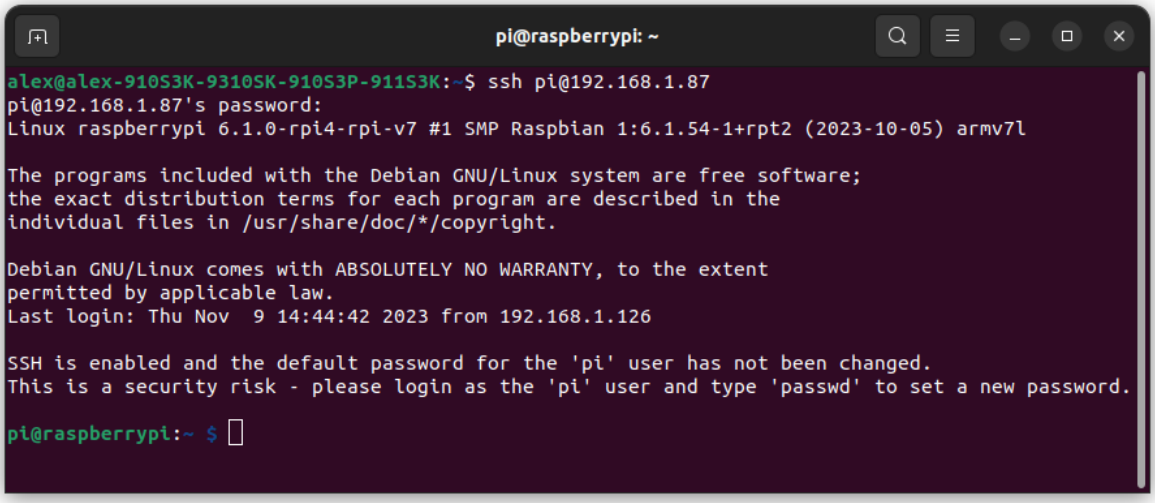
Como padrão Redfish baseia-se em princípios e convenções definidos pela OData, neste projeto a estrutura do Json de saída da API reflete diretamente essas influências, incorporando as marcações @odata em conformidade com as especificações originais do Redfish.
Embora tenha mantido a estrutura do Json alinhada com o padrão original do Redfish, é importante destacar que, neste projeto, as funcionalidades específicas relacionadas ao padrão OData não foram implementadas. As chaves @odata foram mantidas no Json de saída por razões de consistência e para garantir que a estrutura do documento seja reconhecida de acordo com os princípios do Redfish. Ao aderir a essa abordagem, buscamos assegurar que nossa API seja facilmente compreensível e utilizável por sistemas que seguem as mesmas convenções. Isso não apenas promove a interoperabilidade, mas também reflete o objetivo de tentar seguir as melhores práticas estabelecidas além de deixar possibilidade futura de implementação de todas as funcionalidades do padrão OData.

3.7 Configuração para execução automática

Como solução para viabilizar e facilitar o uso da aplicação, a mesma está configurada para inicialização automática utilizando o *s*cript de inicialização de superusuário `rc.local` já existente por padrão no Raspibian, além disso é utilizado um utilitário chamado screen para a execução da aplicação em segundo plano no Raspberry Pi 3.
Nesta seção será demostrado como é feita essa configuração, de forma muito simples e envolve alguns passos, começando pelo processo de instalação da aplicação "Screen"que será responsável por permitir executar em segundo plano nossa API, segue as etapas:

1. Acesso ao Raspberry:
   precisamos ter acesso ao Raspberry, seja por meio de um monitor e teclado conectados diretamente ao dispositivo, SSH a partir de outro computador ou qualquer outro método de acesso, na Figura 16 mostra o acesso via SSH.

Figura 16 – Acessando via SSH.



Fonte: autoria própria

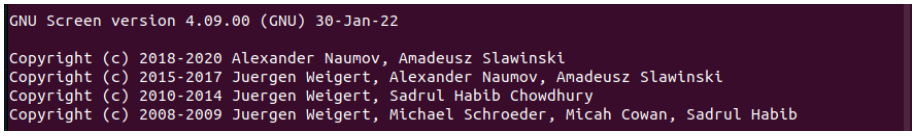
2. Instalação do "Screen": Caso ainda não tenha precisamos instalar o utilitário "screen"no Raspberry, você pode instalá-los com os seguintes comandos:

```
sudo apt-get update
```

```
sudo apt-get install screen
```

   Após a instalação executando o comando Screen pode-se verificar se a instalação ocorreu com sucesso, conforme mostra na Figura 17.

Figura 17 – `Screen` Instalado.

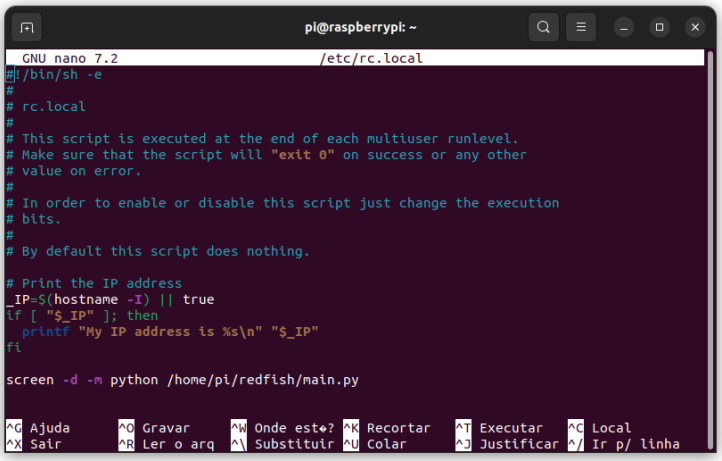


Fonte: autoria própria

3. Configuração da Inicialização Automática:
   Esse passo é o mais importante, configurar o sistema para executar o Screen na inicialização. Além de apontar o caminho para o arquivo `main.py`, precedidos dos parâmetros `-d` e `-m`, para fazer isso, basta adicionar uma linha no arquivo `/etc/rc.local`.

   Abra o arquivo para edição com o comando `sudo nano /etc/rc.local`, conforme mostra na Figura 18, e antes da linha `exit 0:`, adicione a seguinte linha:
   `screen -d -m python /caminho/para/arquivo/main.py`, salve o arquivo e saia.
   OBS: Os parâmetros `-d` e `-m`, faz o screen funcionar no modo "detached"assim é

Figura 18 – `rc.local`.



Fonte: autoria própria

   criada uma nova sessão, mas não o anexa.

4. Reinicialização do equipamento, para que as alterações tenham efeito:

   ```
   sudo reboot
   ```

Lembre-se de substituir "`/caminho/para/arquivo/main.py`"pelo caminho real da sua aplicação. Este processo permite que você configure a inicialização automática para executar sua aplicação em segundo plano sempre que o Raspberry Pi 3 for inicializado.

4 Resultados e Discussão

Ao ligar o Raspberry a aplicação é inicializada juntamente com sistema operacional e seu funcionamento ocorre em segundo plano até o desligamento do dispositivo, a aplicação funciona a partir de qualquer dispositivo com acesso por rede ao raspberry, para isso basta acessar a rota "ip-raspberry:5003/redfish/v1", a resposta é um Json com as rotas para os outros 4 principais *endpoints* como mostra na Figura19 a seguir:

Figura 19 – `Root`.

Fonte: autoria própria

4.1 Dimensão da aplicação

A respeito do trabalho desenvolvido a partir do projeto, se faz necessário apresentar uma visão geral da dimensão da API após o termino de sua implementação. Segue abaixo os dados sobre o que foi desenvolvido até a finalização do projeto.

- 27 rotas desenvolvidas;
- 60 métricas ou dados captados;
- Tratamento do comportamento de *endpoints* de comportamento dinâmico;
- Execução em segundo plano além da inicialização automática junto com sistema.

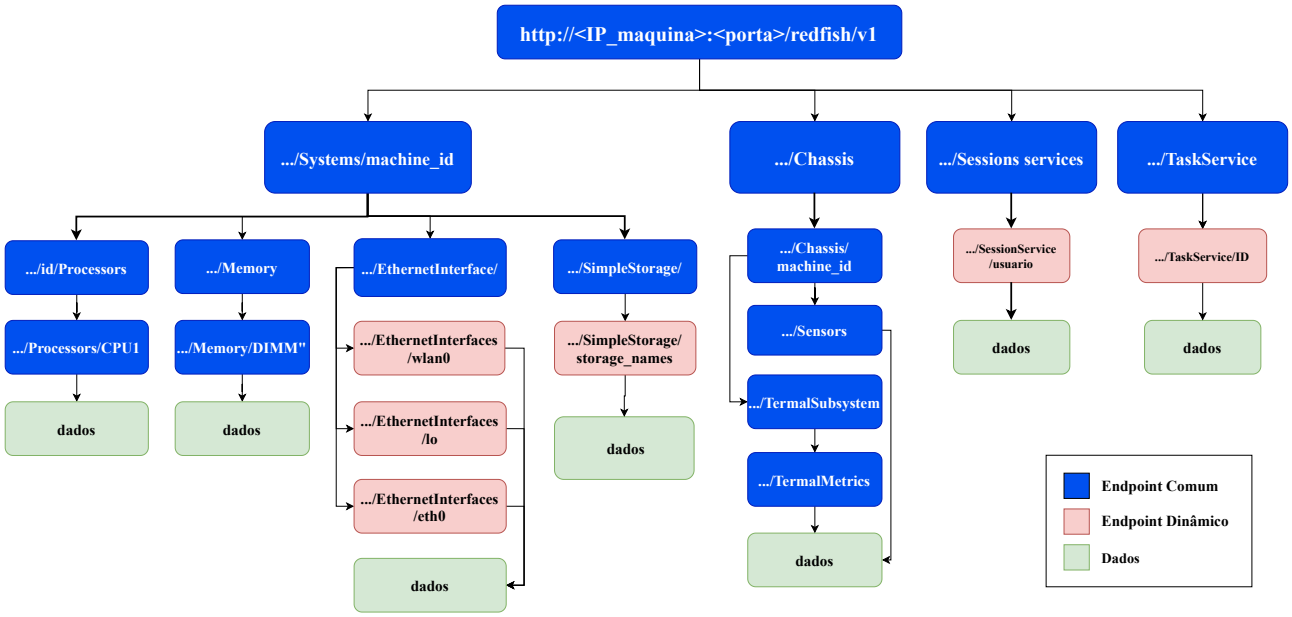
4.2 Validação dos *endpoints*

Com a API em execução, utilizando o browser e uma extensão chamada Json-viewer percorri todas as 27 diferentes combinações de endpoits, e todos estão funcionando, além disso foi observando todos os campos do Json, verificando se os dados estavam dentro do padrão esperado, também foi analisado a execução das operações realizadas pela aplicação e não foi observado travamentos.

4.3 Mapa dos *endpoints* da API

Nesta sessão temos o mapa com todos os `endpoints` da API trazendo um panorama geral de todas as divisões e ramificações da API, desde o *endpoint* `raiz` até retorno de apenas o Json com apenas informações.

Figura 20 – Mapa dos *endpoints* da API



Fonte: Autoria própria

Conforme informa a legenda do mapa da Figura 20 temos: caixas na cor verde são *endpoints* comuns estáticos e levam a outros *endpoints* ou aos Json com apenas informações. Caixas na cor vermelha, representam *endpoints* dinâmicos ou seja são voláteis e dependem se o serviço que eles representam está ativo ou não. Caixas na cor azul, representa um Json com apenas informações não possui uma URL para outro lugar.

4.4 Principais Saídas da API

Nesta sessão será demostrado todas as principais saídas da aplicação, onde será mostrado o Json, e seus detalhes, as chaves com os respectivos valores de métricas ou dados coletados do dispositivo, objetivando o melhor entendimento da API.

4.4.1 Json Memory

Para acessar as informações contidas no Json *memory* precisamos acessar o *endpoint*: "`..Systems/ID/Memory/DIMM`". Como mostra na Figura 21. Vamos destacar apenas os campos que retornam métricas captadas no raspberry, uma vez que os outros elementos do Json são dados padronizados.

Figura 21 – Json *endpoint* "Memory"


```
{
    "@Redfish.Copyright": "Copyright 2014-2016 DMTF. For the full DMTF copyright po
    "@odata.context": "/redfish/v1/$metadata#Memory.Memory",
    "@odata.id": "/redfish/v1/Systems/f97532826de34a7893931d1a591b3865/Memory/DIMM"
    "@odata.type": "#Memory.v1_0_0.Memory",
    "Id": "DIMM1",
    "Memory": {
        "Available": "679M",
        "Buffers": "20M",
        "Cached": "483M",
        "Clock": "450 MHz",
        "Free": "240M",
        "GPUMiB": "76M",
        "Memory Used": "178M",
        "Percent Used": "26.3%",
        "SystemMiB": "948M",
        "TotalMiB": "1024M"
    },
    "Name": "DIMM Slot 1",
    "Status": {
        "Health": "OK"
    },
    "Swap": {
        "Free": "98M",
        "Memory Used": "1M",
        "Percent Used": "1.3%",
        "TotalMiB": "99M"
    }
}
```


Fonte: Autoria própria

1 "Id": Identificador único para a memória, neste caso, "DIMM1".

2 "*Memory*": Objeto que contém informações detalhadas sobre a memória.

2.1 "Available": Quantidade de memória disponível (679 megabytes).

2.2 "*Buffers*": Quantidade de memória utilizada como buffers (20 megabytes).

2.3 "*Cached*": Quantidade de memória em cache (483 megabytes).

2.4 "*Clock*: Velocidade do *clock* da memória (450 megahertz).

2.5 "*Free*":Quantidade de memória livre (240 megabytes).

2.6 "GPUMiB": Quantidade de memória usada pela GPU (76 megabytes).

2.7 "*Memory Used*": Quantidade total de memória usada (178 megabytes).

2.8 "*Percent Used*": Percentual de memória utilizada (26.3%).

2.9 “*SystemMiB*”: Quantidade total de memória do sistema (948 megabytes).

2.10 “*TotalMiB*”: Quantidade total de memória (1024 megabytes).

3 “*Status”*, Objeto que fornece informações sobre o status da memória:

3.1 “*Health*”: Indica o estado de saúde da memória, neste caso, “OK”.

4 “*Swap*”, Contém informações sobre a memória *swap*:

4.1 “*Free*”: Quantidade de memória de *swap* livre (98 megabytes).

4.2 “*Memory Used*”: Quantidade total de memória *swap* (1 megabyte).

4.3 “*Percent Used*”: Percentual de uso da memória *swap* (1.3%).

4.4 “*TotalMiB*”: Quantidade total de memória *swap* disponível (99 megabytes).

4.4.2 Json Processors

Para acessar as informações contidas no Json Processors precisamos acessar o *endpoint*: “`..ID/Processors/CPU1`”. Vamos destacar apenas os campos que retornam métricas captadas no Raspberry, conforme a Figura 22.

Figura 22 – Json *endpoint* “Processors”


```
{
    "@Redfish.Copyright": "Copyright 2014-2016 DMTF. For the full DMTF copyright policy,
    "@odata.context": "/redfish/v1/$metadata#Systems/Members/f97532826de34a7893931d1a591b
    "@odata.id": "/redfish/v1/Systems/f97532826de34a7893931d1a591b3865/Processors/CPU1",
    "@odata.type": "#Processor.v1_0_2.Processor",
    "Id": "CPU1",
    "InstructionSet": "armv7l",
    "Manufacturer": "ARM",
    "MaxSpeedMHz": "800.0 MHz",
    "Model": "BCM2835",
    "ProcessorArchitecture": "armv7l",
    "ProcessorID": {
        "VendorID": "ARM"
    },
    "ProcessorType": "CPU",
    "Socket": "CPU 1",
    "Status": {
        "Health": "OK"
    },
    "TotalCores": "4",
    "TotalThreads": "4"
}
```


Fonte: Autoria própria

1 “*ID”*: Identificador único para o processador, neste caso, “CPU1”.

2 “*InstructionSet*”: Define o conjunto de instruções suportado pelo processador, aqui “armv7l”(arquitetura ARM).

3 “*Manufacturer*”: Indica o fabricante do processador, neste caso, “ARM”.

4 “*MaxSpeedMHz*”: Indica a velocidade máxima do processador em megahertz, neste caso, “800.0 MHz”.

5 “*Model*”: Indica o modelo do processador, aqui “BCM2835”

6 “*ProcessorArchitecture*”: Indica a arquitetura do processador, neste caso, “armv7l”.

7 “*ProcessorID*”: Objeto que contém informações sobre a identificação do processador.

7.1 “*VendorID*”: Indica o ID do fabricante, aqui “ARM”.

9 “*Status*”: Objeto que fornece informações sobre o status do processador.

9.1 “*Health*”: Indica o estado de saúde do processador, neste caso, “ok”.

10 “*TotalCores*”: Indica o número total de núcleos no processador, neste caso, “4”.

11 “*TotalThreads*”: Indica o número total de *threads* (tarefas simultâneas) suportadas pelo processador, neste caso, “4”.

4.4.3 Json SimpleStorage

Para visualizar as informações como na Figura 23 precisamos acessar o *endpoint*: “`../redfish/v1/Systems/ID/SimpleStorage/storageID`”. Como o número de unidades de armazenamento podem variar, esse *endpoint* é dinâmico.

Figura 23 – Json *endpoint* “SimpleStorage”


```
{
    "@Redfish.Copyright": "Copyright 2014-2016 DMTF. For the full DMTF copyright policy, see h
    "@odata.context": "/redfish/v1/$metadata#Systems/Members/f97532826de34a7893931d1a591b3865/
    "@odata.id": "/redfish/v1/Systems/f97532826de34a7893931d1a591b3865/SimpleStorage/mmcblk0",
    "@odata.type": "#SimpleStorage.v1_0_2.SimpleStorage",
    "Description": "SD Card",
    "Devices": [
        {
            "CapacityBytes": "31609323520",
            "Manufacturer": "Unknown (18)",
            "Model": "SDU1",
            "Name": "/dev/mmcblk0"
        }
    ],
    "Id": "mmcblk0",
    "Name": "/dev/mmcblk0"
}
```


Fonte: Autoria própria

1 “`@odata.id`”: Fornece o identificador único para o recurso de armazenamento simples associado ao SD Card.

2 “*Description*”: Descreve o tipo de dispositivo de armazenamento, aqui “SD Card”.

3 “*Devices*”: Lista os dispositivos de armazenamento associados ao recurso de armazenamento simples. Neste caso, há um dispositivo na lista.

3.1 “*CapacityBytes*”: Indica a capacidade total do dispositivo em bytes, aqui “31609323520”(aproximadamente 31.6 gigabytes).

3.2 “*Manufacturer*”:Indica o fabricante do dispositivo, “*Unknown* (18)”(Desconhecido com código 18).

3.3 “*Model*”:Indica o modelo do dispositivo, “SDU1”.

3.4 “*Name*”: Indica o nome do dispositivo, aqui “`/dev/mmcblk0"`

4 “`ID`”: Identificador único para o dispositivo de armazenamento, neste caso, “mmcblk0”.

4.4.4 Json Sensors

Para acesso as informações demostradas na Figura 24 precisamos acessar o *endpoint*: “`..Chassis/ID/Sensors`”. Percebemos que é um subnível da categoria Chassis.

Figura 24 – Json *endpoint* “Sensors”


```
{
    "@Redfish.Copyright": "Copyright 2014-2021 DMTF. For the full DMTF copyright
    "@odata.id": "/redfish/v1/Chassis/f97532826de34a7893931d1a591b3865/Sensors",
    "@odata.type": "#SensorCollection.SensorCollection",
    "CPU Temperature": "60.7'C",
    "CPU Voltage": "1.2688V",
    "Name": "Chassis sensors",
    "SDRAM_C Voltage": "1.2000V",
    "SDRAM_I Voltage": "1.2000V",
    "SDRAM_P Voltage": "1.2250V"
}
```


Fonte: Autoria Própria

1 “`@odata.id`”: Fornece o identificador único para o recurso de sensores.

2 “*CPU Temperature*”: Indica a temperatura da CPU, aqui “60.7°C”.

3 “*CPU Voltage*”: Indica a voltagem da CPU, neste exemplo “1.2688V”

4 “*SDRAM_C Voltage*”: Indica a voltagem do barramento SDRAM_C, aqui “1.2000V”.

5 “*SDRAM_I Voltage*”: Indica a voltagem do barramento SDRAM_I, aqui “1.2000V”.

6 “*SDRAM_P Voltage*”: Indica a voltagem do barramento SDRAM_P, aqui “1.2250V”.

4.4.5 Json EthernetInterfaces

Para acesso as informações demostradas na Figura 25 precisamos acessar o *endpoint*: "`..Systems/ID/EthernetInterfaces`". Como o número de dispositivos pode variar, então esse *endpoint* é dinâmico. Como os dados obtidos são iguais para todas as interfaces vamos tomar como exemplo a `WLAN0`:

Figura 25 – Json *endpoint* "EthernetInterfaces"


```
{
    "@Redfish.Copyright": "Copyright 2014-2016 DMTF. For the full DM
    "@odata.context": "/redfish/v1/$metadata#Systems/Members/f975328
    "@odata.id": "/redfish/v1/Systems/f97532826de34a7893931d1a591b38
    "@odata.type": "#EthernetInterface.v1_0_2.EthernetInterface",
    "Description": "System NIC 2",
    "FactoryMacAddress": "b8:27:eb:fa:91:ff",
    "FullDuplex": "False",
    "IPv4Addresses": [
        {
            "Address": "192.168.1.87",
            "AddressOrigin": "Static",
            "Gateway": "192.168.1.1",
            "SubnetMask": "255.255.255.0"
        }
    ],
    "IPv6Addresses": [
        {
            "Address": "fe80::d692:e920:9f35:4d06%wlan0",
            "AddressOrigin": "Static",
            "AddressState": "Preferred",
            "PrefixLength": 64
        }
    ],
    "IPv6DefaultGateway": "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff",
    "Id": "wlan0",
    "MacAddress": "b8:27:eb:fa:91:ff",
    "Name": "Ethernet Interface",
    "NameServers": [
        "192.168.1.1",
        "fe80::a2b5:3cff:fee7:a996"
    ],
    "SpeedMbps": "0",
    "Status": {
        "State": "Enabled"
    }
}
```


Fonte: Autoria própria

1 "`@odata.id`": Fornece o identificador único para o recurso de interface Ethernet associado à WLAN0.

2 "*Description*": Descreve a interface Ethernet, aqui denominada "System NIC 2".

3 "*FactoryMacAddress*": Indica o endereço MAC de fábrica associado à interface Ethernet.

4 "*FullDuplex*": Indica se a comunicação é em modo full duplex ou não. Neste caso, é "False", indicando que não está em modo full duplex.

5 "*IPv4Address*": Lista informações sobre os endereços IPv4 associados à interface Ethernet. Neste exemplo, é endereço IPv4 estático.

   5.1 "*Address*": Informa o IP do dispositivo.

   5.2 "*AddressesOrigin*": Informa se é estático ou DHCP.

   5.3 "*SubnetMask*": Informa a máscara de sub-rede.

6 "`IPv6`": Lista informações sobre os endereços IPv6 associados à interface Ethernet.

   6.1 "*Address*": Informa o IPv6 do dispositivo.

   6.2 "*AddressesOrigin*": Informa se é estatico com estado "*Preferred*"ou DHCP.

   6.3 "*PrefixLength*": prefixo de comprimento 64.

7 "*IPv6DefaultGateway*": Indica o gateway IPv6 padrão associado à interface Ethernet.

8 "*ID*": Identificador único para a interface Ethernet, neste caso, "wlan0".

9 "*MacAddress*": Indica o endereço MAC associado à interface Ethernet.

10 "*Name*": Nome atribuído à interface Ethernet, aqui "Ethernet Interface".

11 "*NameServers*": Lista de servidores de nomes associados à interface Ethernet.

12 "*NameServers*": Indica a velocidade da interface Ethernet em megabits por segundo. Neste caso, é "0", indicando velocidade desconhecida ou não aplicável.

13 "*Status*": Objeto que fornece informações sobre o status da interface Ethernet.

   13.1 "*State*": Indica o estado da interface, "*Enabled*", significa ativo.

4.4.6 Json ThermalMetrics

Para acesso as informações demostradas na Figura 26 precisamos acessar o *endpoint*: "`..Chassis/ID/ThermalSubsystem/ThermalMetrics`". É um subnível da categoria Chassis.

1 "`@odata.id`": Fornece o identificador único para o recurso de métricas térmicas associado ao chassi.

2 "*DeviceName*": Nome do dispositivo relacionado à leitura de temperatura, aqui "CPU".

Figura 26 – Json *endpoint* "ThermalMetrics"


```
{
    "@Redfish.Copyright": "Copyright 2014-2021 DMTF. For the full DMTF copyright policy, see http://www.
    "@odata.id": "/redfish/v1/Chassis/f97532826de34a7893931d1a591b3865/ThermalSubsystem/ThermalMetrics",
    "@odata.type": "#ThermalMetrics.v1_0_0.ThermalMetrics",
    "Id": "ThermalMetrics",
    "Name": "Chassis Thermal Metrics",
    "Oem": {},
    "TemperatureReadingsCelsius": [
        {
            "DataSourceUri": "/redfish/v1/Chassis/f97532826de34a7893931d1a591b3865/Sensors",
            "DeviceName": "CPU",
            "Reading": "60.7'C"
        }
    ]
}
```


Fonte: Autoria própria

3 "*Reading*": Valor da leitura de temperatura, aqui "60.7°C".

4.4.7 Json TaskService

Para acesso as informações demostradas na Figura 27 precisamos acessar o *endpoint*: "`../redfish/v1/Chassis/ID/TaskService`". assim mostrará a lista com todos os processos que estão rodando naquele momento, os processos são dinamicos assim o numero de *endpoints* exibidos também será.

Figura 27 – Json *endpoint* "TaskService"


```
{
    "@odata.context": "/rest/v1/$metadata#TaskService/$entity",
    "@odata.type": "#Tasks.0.94.0.TaskCollection",
    "Links": {
        "Members": [
            {
                "@odata.id": "/redfish/v1/TaskService/1",
                "Process Name": "/sbin/init"
            },
            {
                "@odata.id": "/redfish/v1/TaskService/5962",
                "Process Name": "[kworker/0:2-events]"
            }
        ],
        "Members@odata.count": 150
    },
    "Modified": "2012-03-07T14:44",
    "Name": "Task Collection"
}
```


Fonte: Autoria própria

1 "*Links*": Fornece o identificador único para o recurso de métricas TaskService.

2 “*Members*”: Uma lista de tarefas individuais, cada uma representada por um objeto Json que inclui sua identificação (@odata.id) e nome do processo associado, denominado como *Process Name*.

3 “Members@odata.count”: Indica o número total de membros na coleção de tarefas.

4 “*Modified*”: Indica a data e hora da última modificação na coleção de tarefas.

4.4.8 Json SessionService

Para acesso as informações demostradas na Figura 28 precisamos acessar o *endpoint*: “`../redfish/v1/SessionService`”. Assim mostrará a lista de todas as sessões de usuários logados naquele momento. Também trata-se de um recurso dinâmico pois pode haver mais de um usuário logado ou não.

Figura 28 – Json *endpoint* “SessionService”

Fonte: Autoria própria

1 “`@odata.id`”: Indica a URL específica do recurso de serviço de sessão ao qual o Json se refere. Permite a navegação direta até o recurso de serviço de sessão associado;

2 “*Members*”: Uma lista de membros da coleção, onde cada membro é identificado por um objeto Json contendo @odata.id como a URL da sessão.

3 “*Members@odata.count*”: Indica o número total de membros sessões;

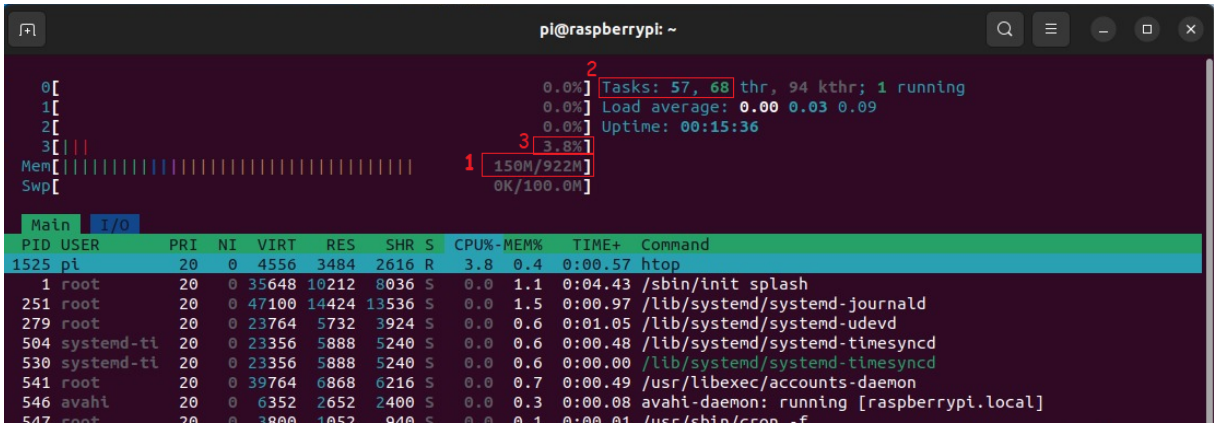
4 “*Modified*”: Indica a data e hora da última modificação na coleção de sessões. Fornece informações sobre quando foi atualizado pela última vez.

Neste Json da Figura 28 indica que há duas sessões de usuário (*Members*) ativos, cada um identificado por uma URL única (@odata.id).

4.5 Avaliação de desempenho

A avaliação de desempenho é um aspecto crítico na determinação da eficiência e adequação de um projeto em um ambiente específico. Nesta seção, examinaremos o desempenho de nossa aplicação em execução no Raspberry Pi 3, com foco no consumo de memória e uso do processador. Para capturar essas métricas, foi utilizado no terminal o comando `htop`, realizando o teste em duas etapas: antes da execução da aplicação e durante a execução da aplicação. Na segunda etapa, foram avaliados diferentes cenários, analisando o desempenho não apenas da execução pura da aplicação, mas conforme as chamadas dos *endpoints*. Na Figura 29 temos as métricas da primeira etapa do teste.

Figura 29 – cenário antes de executar a aplicação



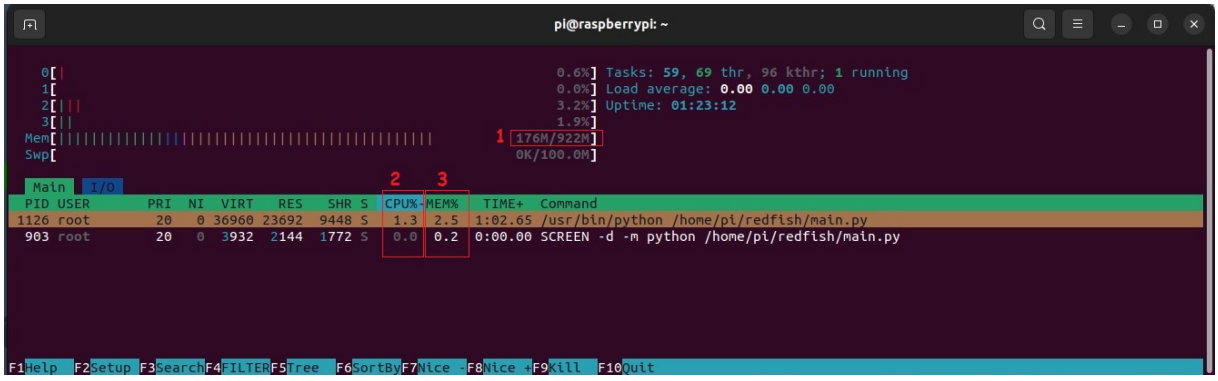
Fonte:Autoria própria

Na figura supracitada constam resultados da primeira etapa de avaliação. Podemos observar três retângulos em vermelho, que sinalizam os recursos disponíveis antes da execução da aplicação. Podemos observar na área sinalizada pelo retângulo 1 o consumo de memória naquele momento que é de apenas 150 MB de um total de 922 MB. No retângulo 2 podemos observar 57 processos em execução com 68 *threads*, já no retângulo 3 o uso total de 3.8% do terceiro núcleo do processador.

A segunda etapa do teste consistiu em avaliar a utilização de recursos durante a execução da aplicação, por sua vez representada pelo processo `main.py` e `screen` que é o processo utilizado para execução da aplicação em segundo plano. Inicialmente foi avaliada

apenas a execução da aplicação sem nenhuma chamada de *endpoint* analisando o consumo de memória e uso do processador. Nesta etapa o processo `htop` foi utilizado com uma taxa de atualização mais rápida e portanto consumindo mais recursos, numa média de 25% de processamento de um núcleo e com consumo irrisório de memória; no entanto o consumo do `htop` não reflete na análise da aplicação desenvolvida e portanto foi desconsiderado dos cálculos. O processo `main.py` apresentou o uso do processador variando em média 1.0% a 1.8%, e consumo de memória de 2.5%, já o processo `screen` apresentou um consumo de memória de 0.2%. como podemos ver na Figura 30.

Figura 30 – cenário durante a execução da aplicação



Fonte:Autoria própria

Ainda nesta etapa, foi analisado o consumo dos recursos a partir das requisição dos *endpoints* dos principais serviços, avaliando o consumo de memória e uso do processador. Os dados estão dispostos na Tabela 3, onde as colunas memória e processador representam consumo instantâneo.

Tabela 3 – Analise dos *endpoints*

Endpoint	Memória	Processador
`.../Sensors`	2.5%	4.5%
`.../ThermalMetrics`	2.5%	8.2%
`.../EthernetInterfaces`	2.4%	4.0%
`.../Memory/DIMM`	2.5%	18.2%
`.../Processors/CPU1`	2.5%	11.5%
`.../SimpleStorage`	3.4%	100.0%
`.../TaskService`	2.7%	8.0%
`.../SessionService`	2.6%	8.2%

Fonte: Autoria própria

Analisando os resultados dessa última etapa, podemos notar que a chamada do *endpoint* `.../SimpleStorage` apresentou o maior consumo de recursos entre os demais, com medias de 3.4% de consumo de memoria e um pico de 100.0% de uso em um core

do processador, isso por conta da leitura das unidades de armazenamento e da utilização do comando `lshw` acarretando em um uso maior de processamento. No geral os outros *endpoints* consumiram valores bem próximas.

4.6 Análise dos resultados

Durante a validação dos *endpoints* observei a fluidez da aplicação, isso devido a forma que a mesma foi desenvolvida diretamente no raspberry isso reduziu bastante possíveis problemas de incompatibilidade entre plataformas, destacando este fato como uma decisão acertiva.
Em relação a análise desempenho realizada, foi utilizado uma metodologia prática e rápida, observando o consumo de recursos do dispositivo antes e depois da execução da aplicação, realizando testes mais aprofundados executando diversas chamadas de *endpoints* simulando assim uma utilização severa da aplicação. Considerando os resultados apresentados como satisfatórios diante de um cenário de recursos limitados, por se tratar de um computador de porte extremamente reduzido, assim diante dessa análise posso considerar que o trabalho foi satisfatório e cumpriu com esperado.

4.7 Código Fonte do Projeto

O código fonte do projeto foi disponibilizado na íntegra no GitHub, por meio do repositório ⟨https://github.com/alexlfig/TCC_API_redfishPi⟩.

5 Conclusões e Perspectivas

Este capítulo apresenta as conclusões finais deste trabalho em relação ao estudo de caso de uma API baseada no padrão Redfish. Além disso, são apresentados sugestões de trabalhos futuros para o mesmo.

5.1 Conclusões

Ao finalizar este projeto, é inegável que me deparei com desafios inerentes às características específicas do *hardware* utilizado, os quais impactaram diretamente na implementação de uma API baseada no padrão Redfish. Realizei adaptações necessárias, moldando a aplicação às nuances do Raspberry Pi, resultando em um modelo que, embora não seja uma cópia fiel do padrão original, apresenta suas principais funcionalidades e princípios.

A decisão consciente de focar exclusivamente nas funções de monitoramento, utilizando modelos de *endpoints* e Json do padrão Redfish, resultou em uma aplicação que, embora não abranja todas as funcionalidades previstas no padrão Redfish original, oferece uma experiência simples e funcional do ponto de vista do usuário. É importante ressaltar que, devido a limitações de tempo e objetivos do projeto, decidi por não implementar de forma mais aprofundada funcionalidades de segurança e questões relacionadas ao padrão OData.

Apesar dessas considerações, pode-se pontuar que o produto final responde com eficiência às demandas de monitoramento remoto em dispositivos Raspberry Pi, consumindo poucos recursos do dispositivo, não comprometendo o seu desempenho e mantendo-se alinhado com as características mais essenciais e interessantes do padrão Redfish.

No âmbito do usuário final, acredito que a abordagem simplificada e focada nas necessidades práticas de monitoramento será bem recebida, especialmente por aqueles familiarizados com o padrão Redfish original. Ao destacar as características mais importantes do padrão e adaptá-las ao contexto específico do Raspberry Pi, a aplicação busca proporcionar uma experiência que seja não apenas funcional, mas também simples e intuitiva.

Dessa forma, conclui-se que mesmo diante das limitações do cenário apresentado e dos ajustes necessários, o trabalho atingiu seus objetivos. Apresentando um potencial considerável para uma solução de integração do Raspberry Pi com contexto do Redfish, proporcionando aos usuários uma solução simples porém eficaz e bem próxima das especificações do padrão original.

5.2 Considerações finais e Melhorias futuras

De maneira geral, este trabalho proporcionou a aplicação dos conhecimentos adquiridos ao longo do curso, bem como conhecer novas tecnologias. Também evidenciou ser uma oportunidade para investigar e aprimorar conhecimentos na área de monitoramento remoto.

Entre as possibilidades para continuação deste trabalho posso destacar a implementação de funções para administração remota, outra melhoria será a geração de uma imagem Docker da aplicação, possibilitando a criação de um *contêiner*, que consequentemente irá facilitar a utilização da aplicação, além disso será necessário implementar um modulo de autenticação, para criação de credenciais para a aplicação podendo assim implementar toda a parte de segurança.

Referências

AREVALO, G. B. *Architectural Description of Object Oriented Frameworks. Tese (Doutorado) - Ecole des Mines de Nantes and Vrije Universiteit Brussels,*. 2000.

CHACOS, B. *Raspberry Pi 3 review: The revolutionary $35 mini-PC cures its biggest headaches. 2016. Publicado em PC World from IDG.* 2016. Disponível em: ⟨https://www.pcworld.com/article/414513/raspberry-pi-3-review-the-revolutionary-35-mini-pc-cures-its-biggest-headaches.html#tk.rss_reviews⟩. Acesso em: 28 setembro. 2023.

COSTA, B. *et al. Evaluating a representational state transfer (rest) architecture: What is the impact of rest in my architecture?* [S.l.]: In: IEEE.*Software Architecture (WICSA), 2014 IEEE/IFIP Conference on.* [S.l.], 2014.

DTMF. *Redfish Specification. Versão 1.12.0.* [S.l.], 2015.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures. Tese (Doutorado) - University of California, Irvine.* 2000.

FLASK. *flask documentation.* 2023. Disponível em: ⟨https://flask-ptbr.readthedocs.io/en/latest/foreword.html#qual-o-significado-de-micro⟩ Acessado em: 31 out. 2023.

FOUNDATION, R. *Raspberry Pi.* 2023. Raspberry Pi Foundation. Disponível em: ⟨https://www.raspberrypi.com/products/raspberry-pi-3-model-b/⟩. Acesso em: 28 set. 2023.

GROUP., T. O. *O-PAS Standard, Version 2.0: Part 1 – Techinical architeture overview (informative).Berkshire UK, 2020a. 49 p.* 2020.

JSON. *Introdução ao JSON.* 2023. Disponível em: ⟨https://www.json.org/json-pt.html⟩ Acessado em: 28 out. 2023.

MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. [S.l.].* [S.l.]: "O'Reilly Media, Inc.", 2011.

MICROSOFT. *Visual studio code.* 2023. Developing on remote machines using SSH and visual studio code . Disponível em: ⟨https://code.visualstudio.com/docs/remote/ssh⟩. Acesso em: 15 Nov. 2023.

MORO, T. D.; DORNELES, C.; REBONATTO, M. T. Web services ws-* versus web services rest. *Instituto de Ciências Exatas e Geociências, Universidade de Passo Fundo (UPF).*, 2011.

OASIS, O. *odata4, standards.* 2023. OASIS Open Data Protocolo (OData) TC — OASIS: ⟨https://www.oasis-open.org/standards/#odatav4.0⟩. Acesso em: 02 Nov. 2023.

OLIVEIRA, A. G. G. de. *Construção o de Aplicações Distribuídas Utilizando-se de APIs REST.* 2018. Disponivel em: ⟨https://di.uern.br/tccs2019/html/ltr/PDF/014006456.pdf⟩ Acesso em 20 Setembro 2023.

ORACLE. *O que é IoT (internet das coisas)?* 2023. Oracle Brasil: ⟨https://www.oracle.com/br/internet-of-things/what-is-iot/⟩. Acesso em: 03 Nov. 2023.

PSUTIL. *psutil documentation*. 2023. Disponível em: ⟨https://psutil.readthedocs.io/en/latest/⟩ Acessado em: 01 Nov. 2023.

PYTHON. *Python Programming Language*. 2023. Disponível em: ⟨http://www.python.org/⟩ Acessado em: 31 out. 2023.

SOMMERVILLE, I. *Engenharia de Software*. [S.l.]: pearson Pretice Hall, 2011. v. 9. ed.

SUBPROCESS. *subprocess — Gerenciamento de subprocessos-Documentação-Python 3.13.0a1*. 2023. Disponível em: ⟨https://docs.python.org/pt-br/dev/library/subprocess.html⟩ Acessado em: 02 Nov. 2023.

YAML. *YAML Ain't Markup Language. version 1.2*. 2023. Disponível em: ⟨https://yaml.org/spec/1.2.2/⟩ Acessado em: 02 Nov. 2023.

6 APÊNDICES

6.1 Funções e *Docsstrings*

Documentação com lista de todas as funções do modulo `readings.py`, acompanhadas de uma breve descrição em formato *docstring*.

- `serial()`:

  "Retorna a serial do Raspberry."

- `machine_id()`:

  "Retorna o Machine ID do Raspberry."

- `boot_id()`:

  "Retorna o Boot ID do Raspberry."

- `hostname()`:

  "Retorna o hostname do Raspberry."

- `board_name()`:

  "Retorna o nome do Raspberry."

- `model()`:

  "Retorna o modelo do Raspberry."

- `system_uuid()`:

  "Retorna a UUID do sistema."

- `power_led()`:

  "Retorna o status do LED de Power do Raspberry Pi."

- `manufacturer()`:

  "Retorna o fabricante da placa Raspberry Pi."

- `power_health()`:

  "Retorna a saúde do sistema de alimentação baseado na tensão"dos cores e da memória RAM."

- `temp_health()`:

  “Retorna o status da temperatura; caso maior que 95, alerta.”

- `cpu_model()`:

  Retorna o modelo do processador.”

- `cpu_vendor()`:

  “Retorna o modelo dos núcleos.”

- `cpu_core_model()`:

  “Retorna o modelo dos núcleos.”

- `cpu_arch()`:

  “Retorna o modelo da arquitetura do processador.”

- `cpu_byte_order()`:

  “ ”Retorna o endianess do processador.”

- `cpu_usage_percent()`:

  “Retorna a porcentagem de uso atual do processador.”

- `cpu_cores()`:

  “Retorna a quantidade de núcleos do processador.”

- `cpu_threads()`:

  “Retorna a quantidade de threads do processador.”

- `cpu_freq()`:

  “Retorna a frequência de operação atual do processador.”

- `cpu_min_freq()`:

  “Retorna a frequência de operação mínima do processador.”

- `cpu_max_freq()`:

  “Retorna a frequência de operação máxima do processador.”

- `cpu_cache_l1d()`:

  “Retorna a capacidade de memória cache L1d do processador.”

- `cpu_cache_l1i()`:

  “Retorna a capacidade de memória cache L1i do processador.”

- `cpu_cache_l2()`:

  "Retorna a capacidade de memória cache L2 do processador."

- `cpu_voltage()`:

  "Retorna a tensão de alimentação lida pelo processador."

- `cpu_health()`:

  "Retorna a saúde do processador baseado na tensão dos cores."

- `cpu_temp()`:

  "Retorna a temperatura do processador."

- `memory_total()`:

  "Retorna a memória total do Raspberry."

- `memory_arm()`:

  'Retorna a memória do Raspberry alocada para CPU geral."

- `memory_gpu()`:

  "Retorna a memória do Raspberry alocada para GPU."

- `memory_freq()`:

  "Retorna a velocidade de clock da memória."

- `memory_used()`:

  "Retorna a quantidade de memória utilizada."

- `memory_percent_used()`:

  "Retorna a porcentagem de memória utilizada."

- `memory_available()`:

  "Retorna a quantidade de memória disponível."

- `memory_free()`:

  "Retorna a quantidade de memória livre."

- `memory_voltage()`:

  "Retorna a tensão de alimentação lida pela memória SDRAM."

- `memory_voltage_c()`:

  "Retorna a tensão de alimentação lida pela memória SDRAM."

- `memory_voltage_p()`:

  "Retorna a tensão de alimentação lida pela memória SDRAM."

- `memory_buffers()`:

  'Retorna a quantidade de memória de buffers."

- `memory_cached()`:

  "Retorna a quantidade de memória em cache."

- `memory_health()`:

  "Retorna a saúde da memória baseado nas tensões de alimentação."

- `swap_total()`:

  "Retorna a memória de swap total."

- `swap_used()`:

  " "Retorna a memória de swap utilizada no momento."

- `swap_free()`:

  "Retorna a memória de swap livre no momento."

- swap_percent():

  "Retorna a porcentagem de uso da memória de swap."

- `os_name()`:

  "Retorna o nome do sistema operacional."

- `os_version()`:

  "Retorna a versão do sistema operacional."

- `os_kernel_version()`:

  "Retorna a versão do Kernel do sistema."

- `eth_count()`:

  "Retorna a quantidade de interfaces de rede do sistema."

- `eth_names()`:

  "Retorna o nome das interfaces de rede do sistema."

- eth_members():

  "Retorna a versão do Kernel do sistema."

- `eth_stats(iface: str)`:

  "Retorna estatísitcas de uma determinada interface de rede, cujo nome lógico é passado como parâmetro."

- `storage_count()`:

  "Retorna a quantidade de dispositivos de armazenamento conectados."

- `storage_members()`:

  "Retorna as URLs dos *endpoints* da API para dispositivos de armazenamento conectados."

- `storage_names()`:

  "Retorna os nomes lógicos dos dispositivos de armazenamento conectados."

- `storage_stats(device)`:

  "Retorna estatísitcas de um determinado dispositivo de armazenamento, cujo nome lógico é passado como parâmetro."

- `session_count()`:

  "Retorna a quantidade de sessões ativas."

- `session`$_{m}embers()$:

  "Retorna *endpoints* relativos a cada sessão ativa."

- `session`$_{l}ogin_{t}ime(user)$:

  "Retorna data e hora de login do usuário especificado."

- `process`$_{c}ounter()$:

  "Retorna a quantidade de processos alocados."

- `process`$_{p}ids()$:

  "Retorna a lista de PIDs dos processos alocados."

- `process`$_{m}embers()$:

  "Retorna as URLs dos *endpoints* referentes a cada processo."

- `process`$_{s}tats(pid)$:

  "Retorna status de monitoramento de um processo especificado."