Guilherme Almeida Andrade

Implementação de uma Ferramenta Web para a Automação de Redes IP Utilizando Python

Uberlândia, MG

2024

Guilherme Almeida Andrade

Implementação de uma Ferramenta Web para a Automação de Redes IP Utilizando Python

Trabalho de Conclusão de Curso da Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Campus Santa Mônica, como requisito para a obtenção do título de Graduação em Engenharia de Controle e Automação.

Universidade Federal de Uberlândia - UFU
Faculdade de Engenharia Elétrica - FEELT

Orientador Prof. Dr. Éderson Rosa da Silva

Uberlândia, MG
2024

Guilherme Almeida Andrade

Implementação de uma Ferramenta Web para a Automação de Redes IP Utilizando Python

Trabalho de Conclusão de Curso da Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Campus Santa Mônica, como requisito para a obtenção do título de Graduação em Engenharia de Controle e Automação.

Trabalho aprovado em 13 de novembro de 2024.

COMISSÃO EXAMINADORA

Prof. Dr. Éderson Rosa da Silva
Orientador

Prof. Dr. Fábio Vincenzi Romualdo da Silva
Membro Avaliador

Mestre Luís Ricardo Cândido Côrtes
Membro Avaliador

Uberlândia, MG
2024

Agradecimentos

Agradeço profundamente à minha família, que foi a base e o suporte para todas as minhas conquistas. Aos meus pais, Mary Almeida e Alessandro Andrade, e à minha irmã, Gabriela Almeida, que sempre estiveram ao meu lado em cada etapa da minha jornada acadêmica, oferecendo todo o apoio e motivação que precisei para alcançar meus objetivos. Sem o carinho, a compreensão e o incentivo de vocês, essa conquista não seria possível.

À minha namorada, Giovana Rodrigues, que esteve comigo em cada momento do curso, oferecendo apoio incondicional e sendo um verdadeiro pilar em minha vida. Sua presença e suporte me deram forças para seguir em frente nos momentos mais desafiadores.

Aos amigos que conheci na faculdade, especialmente Lucas Dantas, Rodrigo Santana e Lucas Sleyder, minha gratidão por cada momento compartilhado, pelo apoio mútuo, pelas risadas e pelo companheirismo ao longo dessa jornada. Vocês foram essenciais para tornar essa experiência ainda mais especial e significativa.

Aos professores do curso, minha eterna gratidão. Em especial, ao meu orientador, Éderson Rosa, que me guiou com paciência e conhecimento neste trabalho, e ao professor Luiz Cláudio Theodoro, por suas valiosas lições e por me oferecer oportunidades de crescimento e desenvolvimento pessoal e profissional. A contribuição de cada um de vocês foi essencial para a realização deste trabalho.

"When you ain't got nothing, you got nothing to lose."
(Bob Dylan, 1965)

Resumo

O crescimento da infraestrutura das redes de computadores é um fenômeno observado nos últimos anos, e a grande quantidade de *vendors* e equipamentos torna o provisionamento e a manutenção das redes cada vez mais complexos e trabalhosos. Desta forma, a automação de redes oferece benefícios significativos para as empresas, como a redução de custos com operações manuais, diminuição do tempo de inatividade e aumento da produtividade ao permitir que equipes de tecnologia da informação (TI) foquem em atividades estratégicas. Além disso, facilita o monitoramento contínuo, a rápida detecção de falhas e a execução automática de tarefas repetitivas nos *Network Operation Centers* (NOCs), o que resulta em maior eficiência e precisão nas operações. A linguagem de programação Python se destaca como uma das principais ferramentas para o desenvolvimento de *scripts* de automação de redes, devido às diversas bibliotecas construídas para integração com equipamentos de rede, como Paramiko e Netmiko. Este trabalho utiliza o *framework* Django para desenvolver um portal que utiliza estas ferramentas de acesso à equipamentos de rede integradas com bibliotecas e módulos Python, realizando automações de processos na rede e garantindo seu monitoramento via interface *Web*. A aplicação foi provisionada utilizando um contêiner Docker virtualizado na rede construída utilizando o *software* GNS3. Esta ferramenta de emulação permitiu a montagem de uma topologia com imagens de equipamentos reais Cisco IOS e Cisco IOU, além de máquinas virtuais, a fim de verificar a utilização do código desenvolvido. Esse código permitiu realizar o *discovery* da topologia, *backup* de configurações, verificação de disponibilidade e monitoramento contínuo.

Palavras-chaves: Automação; Django; Docker; GNS3; Netmiko; Paramiko; Python; Redes; Virtualização.

Abstract

The growth of computer network infrastructure has been a phenomenon observed in recent years, and the large number of vendors and equipment makes network provisioning and maintenance increasingly complex and laborious. In this way, network automation offers significant benefits for companies, such as reducing costs with manual operations, minimizing downtime, and increasing productivity by allowing information technology (IT) teams to focus on strategic activities. Additionally, it facilitates continuous monitoring, quick fault detection, and the automatic execution of repetitive tasks in Network Operation Centers (NOCs), resulting in greater efficiency and accuracy in operations. The Python programming language stands out as one of the main tools for developing network automation scripts due to the various libraries built for integration with network equipment, such as Paramiko and Netmiko. This work uses the Django framework to develop a portal that utilizes these network equipment access tools integrated with Python libraries and modules, automating network processes and ensuring its monitoring via a Web interface. The application was provisioned using a Docker container virtualized on a network built using the GNS3 software. This emulation tool allowed the assembly of a topology with real Cisco IOS and Cisco IOU equipment images, as well as virtual machines, to verify the use of the developed code. This code allowed for topology discovery, configuration backup, availability verification, and continuous monitoring.

Key-words: Automation; Django; Docker; GNS3; Netmiko; Paramiko; Python; Networks; Virtualization.

Lista de ilustrações

Figura 1 – Modelo MVT Django. 21
Figura 2 – Exemplo da criação do *model Equipamento* em uma aplicação Django. 22
Figura 3 – Exemplo da criação da *view listaEquipamentos* em uma aplicação Django. 22
Figura 4 – Exemplo da criação de *template* para apresentar a lista de equipamentos da *view listaEquipamentos*. 23
Figura 5 – Arquitetura agendador de tarefas Celery e *message broker* Redis. . . . 24
Figura 6 – Funcionamento protocolo SSH. 25
Figura 7 – Topologia GNS3. 26
Figura 8 – Arquitetura Docker. 27
Figura 9 – Fluxograma de atividades. 29
Figura 10 – Topologia implementada no emulador GNS3. 30
Figura 11 – Exemplo de comando utilizando o terminal Solar Putty. 31
Figura 12 – Configuração de *hardware* no GNS3. 31
Figura 13 – Configuração da VLAN 1 no SW1. 32
Figura 14 – Configuração do protocolo SSH no SW1. 32
Figura 15 – Configuração de interfaces e protocolo OSPF no roteador R2. 33
Figura 16 – Tabela ARP do roteador R1 antes e após um ping no *broadcast* local. . 35
Figura 17 – Processo de conexão SSH do roteador R1 para o switch SW2. 36
Figura 18 – Código presente em *models.py* para a criação do *model* Equipamentos. 37
Figura 19 – Código presente em *views.py* para criação da *view index*. 37
Figura 20 – Página inicial do portal *Web*. 38
Figura 21 – Definição do dicionário *device*. 39
Figura 22 – Definição das variáveis de controle e início do processo de *discovery*. . . 39
Figura 23 – Código de iteração sobre os *hosts* e acesso aos equipamentos. 41
Figura 24 – Utilização do comando *show ip interface brief* no roteador R2. 41
Figura 25 – Código para extração de informações e recursão do acesso aos equipamentos. 42
Figura 26 – Código das funções utilizadas para extrair endereços IP e interfaces utilizando a biblioteca *re*. 43
Figura 27 – Código inicial *view realizarBackupEquipamento* que acessa o *switch* SW1 e o banco de dados do *model* Equipamentos. 43
Figura 28 – Código da *view realizarBackupEquipamento* que acessa os equipamentos e salva sua configuração. 44
Figura 29 – Código da *view realizarBackupEquipamento* que salva em arquivos texto a configuração dos equipamentos. 45
Figura 30 – Página de backups do portal *Web*. 45

Figura 31 – Código da *view backupEquipamento* que permite baixar e deletar os arquivos de configuração dos equipamentos. 46
Figura 32 – Código da *view backupEquipamento* que lista os arquivos de configuração e suas datas de salvamento. 47
Figura 33 – Opção da *view verificarEquipamento* para acessar um equipamento específico da rede. 47
Figura 34 – Código da *view verificarEquipamento* acessa um equipamento e atualizada suas informações banco de dados e arquivo de texto. 48
Figura 35 – Tela inicial do portal *Web* para a interação com topologia desenvolvida. 49
Figura 36 – Código da *view backupEquipamento* que lista os arquivos de configuração e suas datas de salvamento. 50
Figura 37 – Aba administrador do portal *Web*. 51
Figura 38 – Aba de criação de tarefas periódicas do portal *Web*. 51
Figura 39 – Conexão da VM Ubuntu utilizada para a plataforma docker ao *switch* SW1. 52
Figura 40 – Código do arquivo Dockerfile da aplicação *Web*. 52
Figura 41 – Código do arquivo *docker-compose.yml* da aplicação *Web*. 53
Figura 42 – Utilização do comando *sudo docker-compose up –build* no CLI Ubuntu. 55
Figura 43 – Configuração do *gateway* padrão da máquina Ubuntu e acesso ao portal *Web* utilizando o *UbuntuDesktopGuest*. 55
Figura 44 – *Logs* de acesso à página inicial e gerados pela *view discovery*. 56
Figura 45 – Página inicial do portal *Web* após a realização do processo de *discovery*. 57
Figura 46 – Tabela equipamentos do banco de dados SQLite. 57
Figura 47 – *Logs* gerados pela inicialização da *view realizarBackupEquipamentos*. . 58
Figura 48 – Aba para a *view backupEquipamentos*, apresentando o *download* de um arquivo. 59
Figura 49 – Roteador R6 adicionado à topologia. 59
Figura 50 – Utilização da *view verificarEquipamento* para coleta de informações do roteador R6 e *logs* gerados. 60
Figura 51 – Criação da tarefa periódica *TESTAR CONEXÃO*. 60
Figura 52 – Início e fim dos *logs* gerados pela tarefa automatizada *testarConexao*. . 61
Figura 53 – Funcionamento da tarefa *testarConexao* ao não conseguir resposta do roteador R6. 61

Lista de tabelas

Tabela 1 – Imagens utilizadas na topologia. 30
Tabela 2 – Configuração das interfaces do equipamentos. 34

Lista de abreviaturas e siglas

ACL *Access Control List*

API *Application Programming Interface*

CLI *Command Line interface*

GNS3 *Graphical Network Simulator-3*

HTML *Hyper Text Markup Language*

HTTP *Hypertext Transfer Protocol*

IGP *Interior Gateway Protocol*

IOS *Internetwork Operating System*

IOU IOS *on UNIX*

IA Inteligência Artificial

IP *Internet Protocol*

ISP *Internet Service Provider*

LAN *Local Area Network*

LSA *Link-State Advertisement*

LSDB *Link-State Database*

ML *Machine Learning*

MTV *Model-Template-View*

MVC *Model-View-Controller*

NIC *Network Interface Card*

NOC *Network Operation Center*

NVRAM *Non-Volatile Random Access Memory*

OSPF *Open Shortest Path First*

QEMU *Quick Emulator*

RAM	*Random Access Memory*
RSA	Rivest-Shamir-Adleman
SDN	*Software Defined Network*
SSH	*Secure Shell*
TCP	*Transmission Control Protocol*
TI	Tecnologia da Informação
VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
VTY	*Virtual Teletype*

Sumário

1	**INTRODUÇÃO**	**14**
1.1	**Justificativas**	**15**
1.2	**Trabalhos Relacionados**	**16**
1.3	**Objetivos**	**17**
1.3.1	Objetivos Específicos	17
1.4	**Organização do Trabalho**	**17**
2	**REFERENCIAIS TEÓRICOS**	**19**
2.1	**Python e Bibliotecas para Automação de Redes**	**20**
2.1.1	Paramiko	20
2.1.2	Netmiko	20
2.1.3	Django	21
2.1.4	Celery, Celery Beat e Redis	23
2.2	**Protocolo SSH**	**24**
2.3	**Emulador de Redes e Virtualização**	**25**
2.3.1	GNS3	26
2.3.2	Containers Docker	27
2.4	**Considerações Finais**	**27**
3	**METODOLOGIA**	**29**
3.1	**Implementação da Topologia no GNS3**	**29**
3.1.1	Configuração da Rede	30
3.1.2	Testes de conexão	34
3.2	**Automação Python Local**	**36**
3.2.1	Ambiente Virtual e Projeto Django	36
3.2.2	Desenvolvimento dos *Models*, *Views* e *Templates*	36
3.2.2.1	*Discovery*	38
3.2.2.2	*Backup*	42
3.2.2.3	Verificar Equipamento	46
3.2.2.4	Automação de Tarefas utilizando Celery Beat e Redis	48
3.3	**Aplicação em Container Docker**	**50**
3.4	**Considerações Finais**	**54**
4	**RESULTADOS E DISCUSSÕES**	**55**
4.1	**Testes e resultados para a *view discovery***	**56**

4.2 Testes e resultados para as *views realizarBackupEquipamentos* e *backupEquipamentos* 56
4.3 Testes e resultados da *view verificarEquipamento* 58
4.4 Testes e resultados da tarefa *testarConexao* 60

5 CONCLUSÃO 62

REFERÊNCIAS BIBLIOGRÁFICAS 64

APÊNDICE A – SCRIPT PARA A *VIEW DISCOVERY* 69

APÊNDICE B – SCRIPT PARA A *VIEW REALIZARBACKUPEQUIPAMENTOS* 72

APÊNDICE C – SCRIPT PARA A *VIEW BACKUPEQUIPAMENTOS* 74

APÊNDICE D – SCRIPT PARA A *VIEW VERIFICAREQUIPAMENTO* 76

APÊNDICE E – SCRIPT PARA A TAREFA *TESTARCONEXAO* . . 78

1 Introdução

Define-se como rede de computadores um conjunto de computadores autônomos interconectados por uma tecnologia, permitindo o compartilhamento de informações entre eles (TANENBAUM; FEAMSTER; WETHERALL, 2022). Os sistemas finais, dispositivos utilizados para transmitir ou receber dados através da rede, são conectados entre si por meio dos enlaces de comunicação, implementados através dos mais variados meios físicos, como cabos coaxiais, fibras ópticas, dentre outros (KUROSE; ROSS, 2021). Os dados transmitidos na rede por um sistema final são chamados de pacotes, encaminhados por meio dos *switches* aos seus destinatários corretos dentro de uma rede e por roteadores que permitem conectar redes distintas, possibilitando a transmissão de pacotes entre elas. Além disso, os protocolos tem a função de definir como a comunicação na rede será organizada.

O crescimento da infraestrutura das redes de computadores é um fenômeno que tem sido observado nos últimos anos. Como por exemplo, a receita do mercado de infraestrutura de rede está projetada para alcançar 192.8 bilhões de doláres em 2024 (STATISTA, 2023). Nesse sentido, o crescimento constante das redes torna seu provisionamento e manutenção cada vez mais complexos. Segundo (CISCO, 2022) o crescimento de dados e número de dispositivos está começando a superar as capacidades de TI, tornando abordagens manuais praticamente impossíveis, embora cerca de 95% das mudanças em uma rede ainda são feitas de forma manual. Dentro deste contexto, a necessidade por ferramentas de automação de redes cresce cada vez mais.

A automação de redes é o processo de automatizar a configuração, administração, implantação, operação e manutenção dos dispositivos dentro de uma rede(CISCO, 2022), permitindo diminuir ou até mesmo acabar com o esforço e interferência humana nestes tipos de operações repetitivas, morosas e até mesmo complexas. Diante disto, as linguagens de programação desempenham um papel fundamental na automação de redes, permitindo a criação de *scripts* e ferramentas que simplificam tarefas de configuração, gerenciamento e monitoramento de dispositivos de rede. Dentre elas, a linguagem Python terá ênfase neste trabalho.

O Python é uma linguagem de programação utilizada em aplicações da Web, desenvolvimento de *software*, ciência de dados e *Machine Learning* (ML)(AWS, 2024), é uma linguagem altamente flexível e extensível, permitindo a integração com outras tecnologias. Sua grande variedade de bibliotecas e *frameworks* especializados permite uma implementação facilitada de *scripts* para automação de redes. Paramiko, Netmiko, NAPALM e PySNMP são apenas alguns exemplos de bibliotecas que fornecem ao desenvolvedor uma ampla gama de funcionalidades para interagir com dispositivos de rede, como

roteadores, *switches* e *firewalls*.

Neste contexto, este trabalho busca implementar uma topologia de rede, que se assemelha a uma rede real, através do *software* de emulação GNS3 e desenvolver uma ferramenta que utiliza das bibliotecas Python para automação de redes. Desta forma, será desenvolvido um portal *Web* internamente dentro da topologia, que será responsável por realizar rotinas de automação periódicas, documentação e gerenciamento da rede, e permitir ao usuário o monitoramento de dispositivos.

1.1 Justificativas

Através da constante expansão das redes de computadores e o aumento do número de dispositivos conectados, a administração manual dessas infraestruturas torna-se progressivamente inviável. A crescente complexidade e o tamanho do mercado de infraestrutura de redes estão entre os principais fatores que impulsionam o aumento da demanda por automação de redes (REVIEW, 2024). A grande variedade de *vendors* no mercado, cada um com seus comandos de configuração via *Command Line Interface* (CLI) e interfaces diferentes, requerem cada vez mais uma experiência elevada e proficiência por parte dos engenheiros de rede em tecnologias diferentes para seu provisionamento, manutenção e solução de problemas.

Nesse contexto, a crescente complexidade das redes aumenta a propensão a erros humanos durante o processo de configuração, tornando-os cada vez mais frequentes. Tais erros podem gerar impactos significativos na disponibilidade, segurança e desempenho da infraestrutura de rede. Estes problemas podem ser identificados desde a configuração incorreta dos dispositivos, falhas na implementação de controles de acesso robustos, diagnósticos incorretos na resposta a incidentes, dentre inúmeros outros. Segundo (FRANCO; KURITZKY; LUKACS, 2022), 95% dos problemas de cibersegurança derivam de erros humanos.

Um estudo realizado pela a empresa Analysys Mason foi responsável por pesquisar os benefícios da automação de redes IP. O projeto coletou mais de 60 pontos de dados de estratégias de automação da rede de cinco operadoras diferentes, para desenvolver um modelo de benefícios dimensionado para uma operadora regional de grande escala (GOLDMAN; RAO; KILLEEN, 2021). Verificou-se na categoria ”Manutenção do Ciclo de Vida da Rede”, que engloba provisionamento, configuração e manutenção de equipamentos de rede, uma diminuição de 72% do custo e tempo de trabalho de provisionamento, de 70% do custo e do tempo de trabalho em melhorias e 65% de redução do tempo de trabalho e de custos de gerenciamento do ciclo de vida da rede.

Assim, torna-se cada vez mais essencial o desenvolvimento de aplicações no campo da automação de redes, com o objetivo de implementar e aprimorar as práticas e tecnologias

que minimizem erros humanos, aumentem a eficiência operacional e assegurem a segurança e a confiabilidade das infraestruturas de rede.

1.2 Trabalhos Relacionados

No âmbito acadêmico, diversas pesquisas se concentraram em apresentar abordagens para a automação de tarefas como o provisionamento de dispositivos, monitoramento em tempo real, e a integração de redes com ferramentas para predição de falhas e otimização de performance. Esta seção discute algumas abordagens e soluções para automação de redes IP, analisando estudos acadêmicos, ferramentas amplamente utilizadas no setor, e casos de uso relevantes, destacando como esses trabalhos contribuem para a evolução das práticas de automação e monitoramento de redes.

No trabalho proposto em (ROCHIM et al., 2020), destaca-se a criação de uma REST API desenvolvida com o Django, utilizando uma abordagem semelhante à deste trabalho. O estudo discute como, tradicionalmente, os administradores de rede configuram dispositivos manualmente, em um processo demorado e ineficiente. A pesquisa apresenta a automação de redes como solução, realizando um estudo comparativo entre a aplicação *Web* desenvolvida, chamada As-RaD, e as bibliotecas Python Paramiko e NAPALM. Utilizando o roteador Cisco CSR1000V que permite a comunicação via API, o estudo demonstrou que a As-RaD foi 75% mais rápida que o Paramiko e 92% mais rápida que o NAPALM, destacando-se pelo alto desempenho.

O estudo apresentado em (SANTYADIPUTRA; LISTARTHA; SASKARA, 2021) descreve a implementação de uma simulação voltada à automação de redes utilizando uma ferramenta denominada ANA. Assim como As-RaD, a aplicação foi desenvolvida utilizando o *framework* Django para a interface *Web*, mas emprega as bibliotecas de automação de rede Paramiko e Netmiko em um ambiente virtualizado com GNS3 e VirtualBox. Os resultados demonstram que ANA foi eficaz para atingir os objetivos da automação de redes, melhorando a eficiência nas atividades administrativas.

Por fim, o artigo apresentado em (ANDARA; WIDYARTO; RUSDAH, 2023) utiliza métricas obtidas com o modelo Kano de gestão de qualidade e satisfação dos usuários com relação às funcionalidades de automação de redes desenvolvidas. Assim como as demais, a aplicação foi implementada com a biblioteca Paramiko para comunicação via protocolo SSH e Django para o desenvolvimento da interface *Web*. Testes foram realizados e mostraram que a aplicação é eficiente na automação de tarefas como configuração centralizada de roteadores, *switches*, *backups* e restauração de configurações, proporcionando melhor gerenciamento, além de variados graus de satisfação para cada automação.

A análise das diferentes abordagens mencionadas evidencia como a automação de redes tem se tornado uma solução eficaz para os desafios enfrentados pelos administradores

de rede. Assim, a automação de redes representa uma tendência crescente para otimizar processos repetitivos, aumentar a confiabilidade e reduzir erros humanos, configurando-se como uma área de estudo e desenvolvimento crucial para o futuro da administração de redes IP. Com isso em mente, este trabalho se propõe a desenvolver soluções práticas que alinhem as tendências discutidas nas aplicações apresentadas.

1.3 Objetivos

O objetivo deste trabalho é desenvolver uma aplicação *Web* para automação e gerenciamento de redes IP, permitindo a execução de rotinas em uma topologia emulada, monitoramento de equipamentos e documentação de sua infraestrutura, a fim de garantir sua confiabilidade e eficiência de gerenciamento.

1.3.1 Objetivos Específicos

A fim de atingir o objetivo principal deste trabalho, foram definidos os seguintes objetivos específicos:

- Implementação de uma topologia utilizando o *software* GNS3, um emulador de redes que torna possível a utilização de equipamentos que se assemelham a equipamentos reais;
- Configuração inicial nos equipamentos que permita seu acesso e gerenciamento através de uma plataforma dentro da rede;
- Desenvolvimento de um portal *Web*, *backend* Django e *frontend* JavaScript e CSS. Esse portal terá a função de gerenciar os equipamentos da rede, implementar rotinas de automações periódicas e proporcionar ao usuário a capacidade de monitorar e administrar os dispositivos;
- Virtualização do portal na rede através de um contêiner Docker;
- Realização de testes dos *scripts* na topologia emulada.

1.4 Organização do Trabalho

Este trabalho está apresentado a seguir em quatro capítulos, cada um abordando aspectos essenciais do desenvolvimento e implementação de uma aplicação *Web* para automação e gerenciamento de redes IP.

No Capítulo 2, Referenciais Teóricos, são exploradas as principais tecnologias e bibliotecas utilizadas na automação de redes, incluindo Python, Paramiko, Netmiko, Celery

e Django. Além disso, discute-se também o protocolo SSH e as ferramentas de emulação e virtualização, como GNS3 e Docker.

Por meio do Capítulo 3, Metodologia, são detalhados os passos para a implementação da topologia de rede utilizando o *software* GNS3, incluindo a configuração da rede, testes de conexão e a automação local utilizando Python. Este capítulo também abrange a implementação o desenvolvimento local da aplicação Django, a tarefa periódica e *views* desenvolvidas para automação e monitoramento da rede e a utilização de contêineres Docker para *deploy* da aplicação.

O Capítulo 4, Resultados e Discussões, contém os testes realizados nas funcionalidades implementadas, com uma análise dos resultados obtidos para cada uma das *views* e tarefa periódica, discutindo os *logs* apresentados nos contêineres Docker.

Por fim, o Capítulo 5, Conclusão, sintetiza as principais implementações e contribuições do trabalho, refletindo sobre os resultados alcançados e as implicações para futuras pesquisas e desenvolvimentos na área de automação de redes.

2 Referenciais Teóricos

Os protocolos TCP/IP são a base em que se formou a Internet, a rede de computadores que interconecta centenas de milhões de dispositivos de computação ao redor do mundo (KUROSE; ROSS, 2021), permitindo a transferência de dados entre tais dispositivos em escala local e global. O *Internet Protocol* (IP) é responsável pelo endereçamento e roteamento, fornecendo um formato universal de pacote que todos os roteadores reconhecem e que pode ser transmitido por quase todas as redes (TANENBAUM; FEAMSTER; WETHERALL, 2022), utilizando endereços IP para identificar dispositivos únicos.

Atualmente, o procotolo IP possui duas versões: IPv4 e IPv6, e estes protocolos fazem parte do conjunto de protocolos de comunicação de dados TCP/IP. O *Transmission Control Protocol* (TCP) garante a entrega ordenada e confiável dos dados entre sistemas finais. A Internet abrange centenas de milhares de redes em todo o mundo, essas redes são construídas por provedores comerciais, provedores de redes nacionais e provedores de redes regionais, que juntos criam a infraestrutura, compartilhados através do acesso local fornecido pelos *Internet Service Providers* (ISPs)(HUNT, 2002).

O aumento da popularidade da Internet têm sido exponencial, alcançando hoje 5,35 bilhões de usuários no mundo todo, cerca de 66% da população mundial (PELCHEN, 2024). Esse grande aumento no número de usuários gera, consequentemente, a necessidade de expansão das redes para garantir que a entrega eficiente dos serviços esteja disponível para todos os usuários. O crescimento no número de dispositivos conectados e na quantidade de dados trafegados, tornou a gestão manual dessas redes cada vez mais desafiadora, verificando assim a necessidade de soluções automatizadas para manter a eficiência e a segurança.

Segundo (JUNIPER, 2023) a automação de redes IP consegue realizar uma ampla variedade de tarefas como: redução do número de erros ao utilizar fluxos de trabalho pré definidos evitando erros manuais, diminuição de custos ao simplificar e tornar operações eficientes, garantia de maior controle da rede tornando-a controlável e adaptável, dentre várias outras melhorias. É dentro deste contexto que o Python entra como uma das principais ferramentas utilizadas por engenheiros para desenvolver aplicações de automação de redes. Esta linguagem de programação é simples comparada a linguagens como Java e Ruby e possui uma grande variedade de bibliotecas que facilitam a criação de códigos e integração com *Application Programming Interfaces* (APIs) (KUBADE, 2019).

2.1 Python e Bibliotecas para Automação de Redes

O desenvolvimento de ferramentas de automação de redes em Python é recomendável por oferecer uma base sólida para escalabilidade, melhoria contínua e aumento de funcionalidades de acordo com as necessidades do usuário. A ampla gama de bibliotecas e *frameworks* disponíveis em Python garante um desenvolvimento especializado e adaptável, permitindo a criação de ferramentas de automação de redes, como Paramiko e Netmiko, além de possibilitar sua integração com módulos nativos ou externos.

Como verificado por (MIHăILă et al., 2017), estas bibliotecas permitem a criação de novos métodos na configuração de dispositivos de rede usando automação e, deste modo, permitem a redução do tempo de configuração de equipamentos, facilita sua manutenção e melhora a segurança da rede identificando e eliminando vulnerabilidades de segurança. A análise de performance realizada na automação de uma rede por (MAZIN et al., 2021) demonstrou que a automação melhorou significavelmente a eficiência da configuração da rede, aumentando sua velocidade e evitando erros humanos.

Um cenário semelhante foi descrito por (JAYASEKARA, 2022), que buscou simplificar a complexidade da configuração de redes por meio de automações Python. O resultado deste projeto foi semelhante aos trabalhos citados, observando que as automações desenvolvidas contribuíram significativamente na economia de tempo e prevenção de erros. Assim, destaca-se a relevância do desenvolvimento de ferramentas de automação para redes IP, um processo facilitado e viabilizado pelo uso das bibliotecas Python.

2.1.1 Paramiko

Paramiko é uma biblioteca Python de baixo nível que implementa a comunicação *Secure Shell* (SSH) versão 2. Embora seja menos orientada para automação de rede, permite a execução de comandos remotos via SSH e sua integração com outros módulos permite uma flexibilidade para a criação de automações complexas de forma simples (FORCIER, 2024). Além disso, pode-se verificar a utilização desta biblioteca até mesmo em conjunto com algoritmos de ML para a detecção de anomalias em redes, como é mostrado em (BUDIATI et al., 2022). Esta biblioteca é a base estrutural da biblioteca Netmiko.

2.1.2 Netmiko

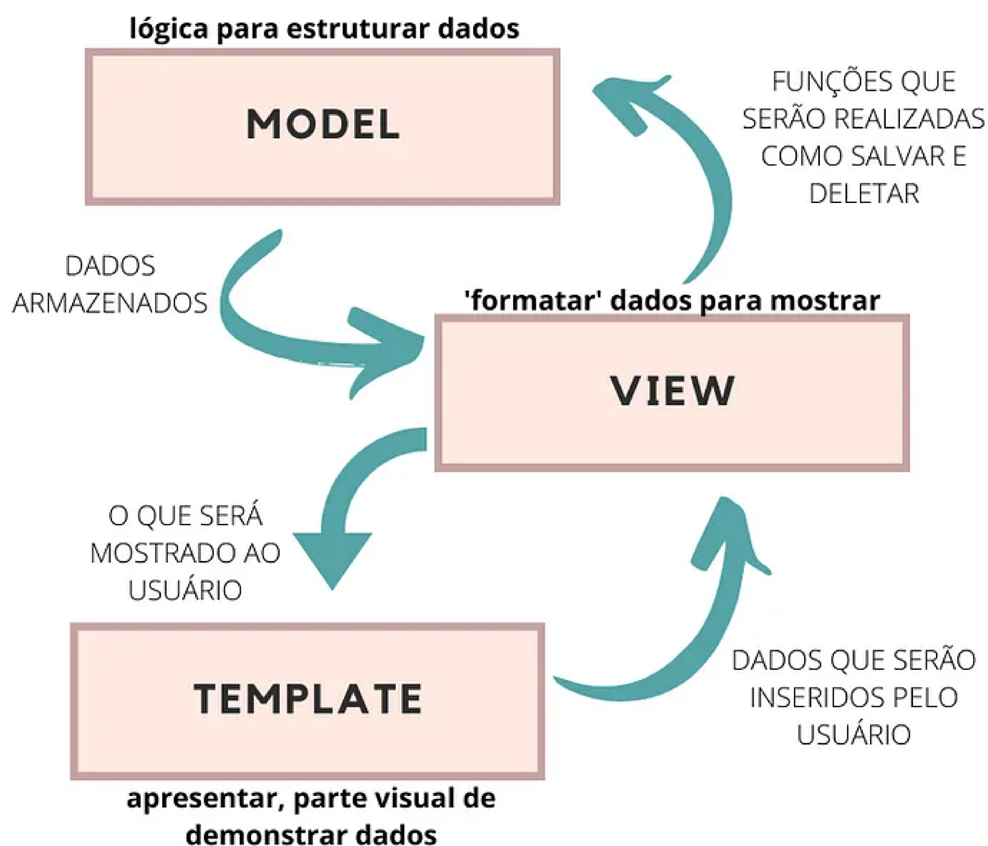
Netmiko é uma biblioteca Python que simplifica a automação de dispositivos de rede via SSH. Desenvolvida pelo engenheiro Kirk Byers, Netmiko oferece suporte a uma variedade de dispositivos de diferentes fabricantes, como Cisco, Juniper, Arista, Huawei, entre outros (BYERS, 2024a). Baseada na biblioteca Paramiko, dentre suas funções principais pode-se citar: a conexão SSH aos equipamentos de forma simplificada, coleta de

dados eficiente, abstração de ferramentas de baixo nível dos equipamentos, entre outras (BYERS, 2021).

2.1.3 Django

O Django é um *framework*, escrito em Python, para desenvolvimento *Web* de alto nível, projetado para facilitar a criação de aplicações web rapidamente, seguras contra *scripts* maliciosos, altamente gerenciáveis e escaláveis de maneira rápida (DJANGO, 2024c). O *framework* segue o padrão de arquitetura de design *Model-Template-View* (MTV), uma variação do modelo *Model-View-Controller* (MVC) (URANO, 2023), que pode ser observado na Figura 1.

Figura 1 – Modelo MVT Django.



Fonte: Retirado de (SILVA, 2023).

A arquitetura apresentada na Figura 1 evidencia a divisão base do *framework* entre:

- *Model*: Responsável por salvar as informações da aplicação, representando uma tabela no banco de dados utilizado, onde cada atributo da classe corresponde a um campo

nessa tabela. Um *model* define os campos essenciais e comportamentos dos dados que estão sendo armazenados, fornecendo automaticamente uma API para interagir com o banco de dados. É definido por uma lista de campos, especificados como atributos da classe, representando as colunas da tabela criada no banco de dados e seus tipos de dados, que podem ser manipulados por meio dos métodos fornecidos pelo Django e *scripts* criados pelos desenvolvedores da aplicação. A Figura 2 mostra um exemplo de código para a criação do *model Equipamento*.

Figura 2 – Exemplo da criação do *model Equipamento* em uma aplicação Django.

```
from django.db import models

class Equipamento(models.Model):
    hostname =models.CharField(max_length=100)
    status =models.BooleanField(default=True)
```

Fonte: Autoria própria (2024).

- *View*: É uma função Python que recebe uma requisição *Web* e retorna uma resposta para tal requisição (DJANGO, 2024d), seguindo a lógica implementada por seu desenvolvedor. As *Views* processam as requisições *Hypertext Transfer Protocol* (HTTP), que chegam ao servidor da aplicação, interagem com os *Models* criados para acessar ou manipular dados, renderizam *Templates* e outros tipos de respostas. As *Views* são implementadas utilizando funções ou baseadas em classes, facilitando a reutilização de código e a organização da aplicação. Deste modo, conectam os dados do bando de dados com as interfaces de usuário, garantindo que a lógica da aplicação seja executada e exibida corretamente. A Figura 3 mostra um exemplo de *view listaEquipamentos* que busca todos os valores do banco de dados para o *model Equipamento*, enviando esta informação ao *template*.

Figura 3 – Exemplo da criação da *view listaEquipamentos* em uma aplicação Django.

```
from django.shortcuts import render
from .models import Equipamento

def listaEquipamentos(request):
    equipamentos =Equipamento.objects.all()
    return render(request, 'equipamentos/lista.html', {'equipamentos': equipamentos})
```

Fonte: Autoria própria (2024).

- *Template*: Forma pela qual a aplicação Django utiliza para gerar o *Hyper Text Markup Language* (HTML) de maneira dinâmica (DJANGO, 2024b). Um *template* contém as partes estáticas do HTML e o conteúdo dinâmico a ser inserido pela

lógica desenvolvida por meio da *view*. Utilizando sua *template engine* o Django consegue reconhecer variáveis, *tags*, filtros e comentários, interpretando e criando o *template* final. A Figura 4 mostra um exemplo de *template* para mostrar uma lista dos equipamentos apresentados pela *view listaEquipamentos*.

Figura 4 – Exemplo da criação de *template* para apresentar a lista de equipamentos da *view listaEquipamentos*.

```
<h1>Lista de Equipamentos</h1>
<ul>
    {% for equipamento in equipamentos %}
        <li>{{ equipamento.hostname }}:
        {% if equipamento.status %}
        Ativo
        {% else %}
        Inativo
        {% endif %}</li>
    {% endfor %}
</ul>
```

Fonte: Autoria própria (2024).

2.1.4 Celery, Celery Beat e Redis

O Celery é uma biblioteca muito utilizada na criação de automações em aplicações Django, pois permite a criação de filas de tarefas com processamento em tempo real, permitindo o suporte ao agendamento destas tarefas (DOCUMENTATION, 2023a). Em contraste com a arquitetura de multiprocessamento, o Celery fornece o agendamento e o encadeamento de sub tarefas (STIGLER; BURDACK, 2020). A flexibilidade dos componentes desta biblioteca permite aos desenvolvedores a execução de tarefas de longa duração ou tarefas que não precisam ser concluídas imediatamente, sem que ocorra um bloqueio do fluxo da aplicação principal, rodando os *scripts* automatizados em segundo plano.

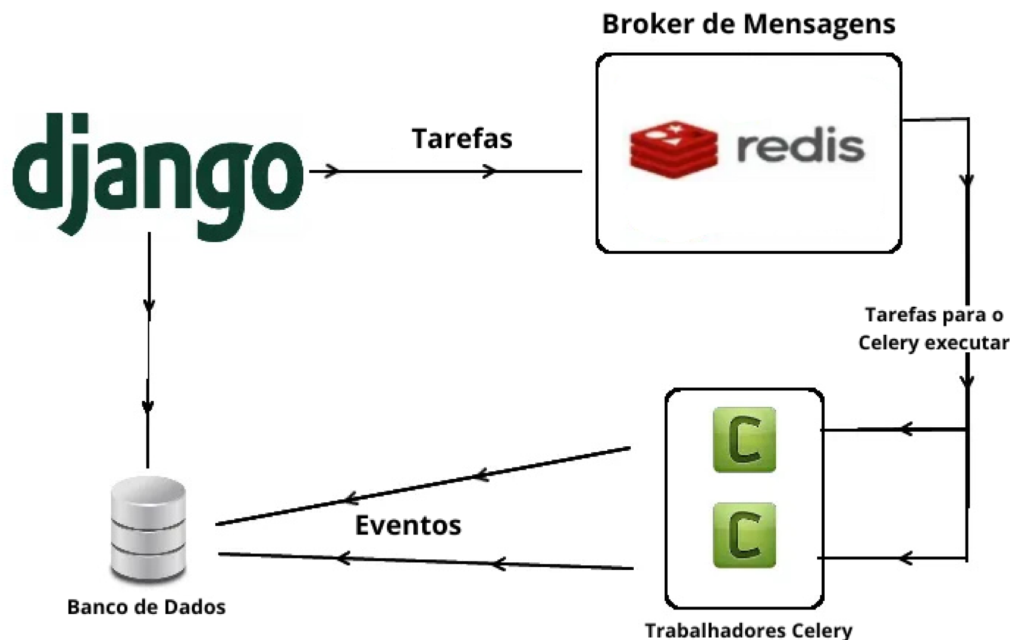
A fim de realizar tarefas periódicas, utiliza-se o Celery Beat, um agendador que inicia tarefas em intervalos regulares, executadas pelos *worker nodes* disponíveis em seu *cluster* (DOCUMENTATION, 2023b). No intuito de obter maior escalabilidade de uma solução utilizando o Celery Beat, este pacote deve ser utilizado junto a um *message broker*, neste trabalho, o Redis. Um *message broker* é um software que possibilita que aplicativos, sistemas e serviços se comuniquem e troquem informações (IBM, 2024), atuando como o intermediário na troca de informações entre as partes. Ao utilizar o Redis para tratar as mensagens das tarefa, os *workers* do Celery podem se concentrar em executá-las, podendo

escalar a execução para mais *workers* com eficiência no gerenciamento das filas(PRASAD, 2024).

A arquitetura da automação de tarefas utilizando os componentes citados está apresentada na Figura 5. O fluxograma mostra que, inicialmente, o Django envia tarefas para o *broker* de mensagens sempre que há tarefas assíncronas a serem executadas. O *broker* de mensagens atua como um intermediário, recebendo as tarefas enviadas pelo Django, organizando-as e encaminhando-as para os trabalhadores do Celery.

Os trabalhadores do Celery são processos responsáveis pela execução das tarefas recebidas enviadas pelo *broker* de mensagens, permitindo sua realização sem bloquear o fluxo central do servidor *Web* do Django. Após a conclusão das tarefas, eles registram eventos no banco de dados responsável por armazenar informações sobre a aplicação.

Figura 5 – Arquitetura agendador de tarefas Celery e *message broker* Redis.



Fonte: Adaptado de (PATEL, 2022).

2.2 Protocolo SSH

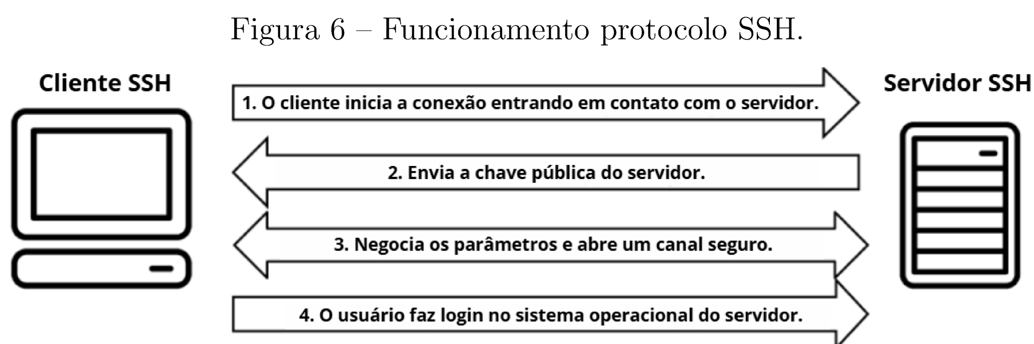
O SSH é um protocolo de rede criptográfico usado para criptografar dados enviados de um computador à rede e realizar sua descriptografia quando chegam ao seu destino final (SILVERMAN; BARRETT, 2001). Como observado, um dos seus principais usos é evidenciado no acesso remoto a equipamento e sistemas, assim como a execução remota de comandos, transferência segura de arquivos e tunelamento de serviços de rede. O SSH fornece uma alternativa segura a protocolos como Telnet (MATHEUS, 2018), Rlogin e FTP, que transmitem dados sensíveis sem criptografia pela rede. É através deste protocolo que

grande parte dos *scripts* para automação de redes conseguem realizar seu acesso remoto a equipamentos.

A Figura 6 mostra como é realizada a conexão entre dois equipamentos via SSH. É evidenciado que o cliente SSH envia inicialmente uma solicitação de conexão para o servidor SSH, geralmente realizado ao especificar o endereço IP ou nome do *host* do servidor e a porta 22. Assim que o servidor recebe a solicitação de conexão, ele responde enviando sua chave pública para o cliente. Essa chave é usada para estabelecer uma conexão segura, dado que o cliente a utiliza para verificar a identidade do servidor e garantir que está se conectando ao *host* correto.

Após o cliente verificar a identidade do servidor, tanto o cliente quanto o servidor negociam os parâmetros de criptografia e compressão a serem usados durante a sessão, estabelecendo um canal seguro usando criptografia para toda a comunicação subsequente. Por fim, o usuário insere suas credenciais de acesso ao sistema operacional do servidor. Uma vez autenticado, o usuário pode executar comandos e gerenciar o servidor de forma segura remotamente.

Figura 6 – Funcionamento protocolo SSH.



Fonte: Adaptado de (SSH, 2023).

A fim de realizar a autenticação criptográfica, é necessário primeiro gerar um par de chaves, composto por uma chave privada representando a identificação digital armazenada no *host* e uma chave pública, armazenada no servidor(SILVERMAN; BARRETT, 2001). Este par de chaves é criado pelo algoritmo de encriptação Rivest-Shamir-Adleman (RSA).

2.3 Emulador de Redes e Virtualização

A emulação de redes e a virtualização são técnicas essenciais no campo da administração e engenharia de redes, pois permitem a criação de topologias que se assemelham à realidade e a execução de múltiplos sistemas operacionais em um único *hardware* físico, respectivamente. Essas tecnologias são amplamente utilizadas em conjunto para fins de teste, desenvolvimento de rede, treinamento de algoritmos e experimentação, proporcionando um ambiente controlado e seguro para validar configurações e comportamentos de redes sem impactar os sistemas de produção.

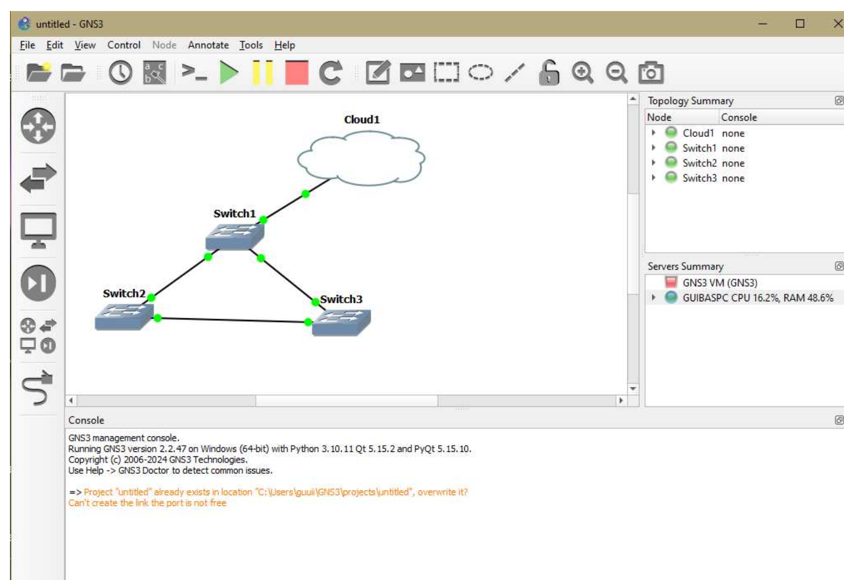
Os emuladores de redes são ferramentas que permitem a criação de topologias complexas, incluindo equipamentos, *links* e políticas de roteamento, em um ambiente virtual. Ou seja, ao realizar a emulação de uma rede cria-se uma *Software Defined Network* (SDN) para testes, que replica uma rede real. Um dos emuladores de rede mais populares é o *Graphical Network Simulator-3* (GNS3).

A virtualização é a tecnologia que permite a criação de instâncias virtuais de sistemas operacionais de diversos equipamentos em um único *hardware* físico (AWS, 2023). Por meio dos hipervisores, *softwares* utilizados para a execução de Máquinas Virtuais (VMs), o *hardware* ou sistema operacional de um equipamento consegue separar os recursos físicos dos ambientes virtuais que os utilizam (REDHAT, 2018). Dentre as várias plataformas de virtualização existentes, pode-se citar VMware, VirtualBox e Docker.

2.3.1 GNS3

O GNS3 é uma plataforma de emulação de rede que permite ao usuário emular dispositivos de rede reais e virtuais. Assim é possível a integração entre imagens de *Internetwork Operating System* (IOS) da Cisco, Cisco IOS *on UNIX* (IOU), virtualização via *Quick Emulator* (QEMU), emulador de sistemas operacionais, VMs, acesso à Internet e rede local (LAN). Portanto, apresenta-se como a ferramenta adequada para desenvolver um ambiente de testes para automações. O GNS3 é um *software* gratuito e *Open Source*, utilizado por centenas de milhares de engenheiros. Um exemplo de topologia criada no GNS3 pode ser observado na Figura 7.

Figura 7 – Topologia GNS3.



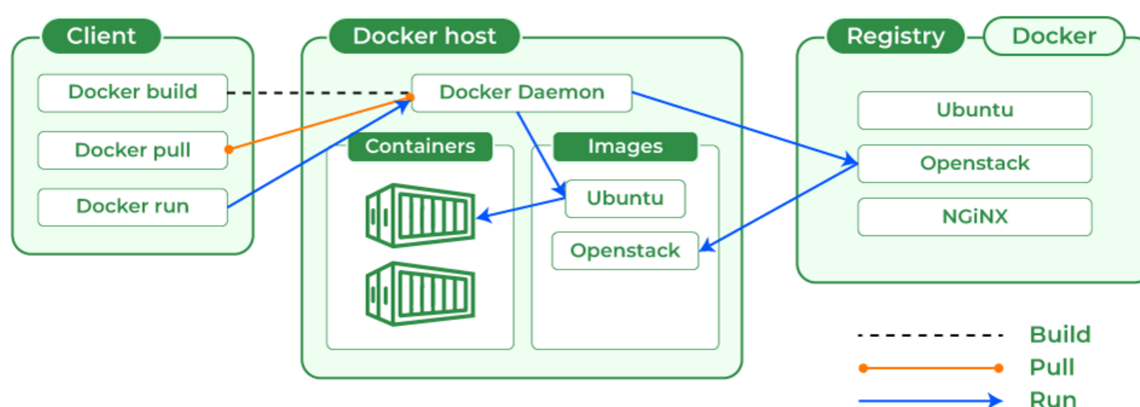
Fonte: Autoria própria (2024).

2.3.2 Containers Docker

Um contêiner é uma unidade padronizada de *software* que empacota um código e todas as suas dependências, garantindo que a aplicação desenvolvida execute de forma rápida e confiável em diferentes ambientes de computação (DOCKER, 2024b). Ou seja, um contêiner Docker é um pacote de *software* que agrega tudo que é necessário para a aplicação funcionar, permitindo que esta possa ser implementada de forma padronizada através da *engine* do Docker.

A Figura 8 mostra como funciona a arquitetura do Docker. Por meio dela, é possível notar sua estrutura cliente-servidor, onde o cliente se conecta ao *daemon* Docker a fim de gerenciar a criação, execução e distribuição de contêineres. Essa comunicação pode ser realizada localmente ou remotamente, empregando uma API REST que opera através de um socket UNIX ou de uma rede (GEEKSFORGEEKS, 2020).

Figura 8 – Arquitetura Docker.



Fonte: Retirado de (GEEKSFORGEEKS, 2020).

Como cita (BOETTIGER, 2015), o grande sucesso de mercado do Docker se dá, principalmente, das necessidades das empresas na implantação de aplicações *Web* e de seu grande potencial de uma alternativa leve à virtualização completa. Desta forma, pode-se unir o Docker ao código desenvolvido em Python, a fim de criar a aplicação *Web* que realizará automações na rede.

2.4 Considerações Finais

Neste capítulo, foram abordados os princípios teóricos utilizados no desenvolvimento da aplicação *Web* para automação de redes. Foi discutido o uso da linguagem Python e suas bibliotecas, como Paramiko e Netmiko, que facilitam a interação com dispositivos de rede por meio de protocolos como SSH. Além disso, foi explorada a integração do *framework* Django, que fornece a estrutura necessária para construir a aplicação, e do

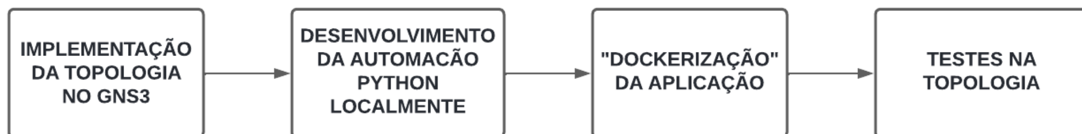
Celery, que permite a execução assíncrona de tarefas automatizadas, essencial para garantir a eficiência e a responsividade do sistema.

O uso de ferramentas como GNS3 e contêineres Docker também foi destacado, pois elas possibilitam a emulação de redes e a virtualização, fundamentais para testes e desenvolvimento. Essa integração entre as ferramentas e tecnologias discutidas é crucial para a implementação eficaz da solução proposta. No próximo capítulo, Metodologia, é detalhado o processo de desenvolvimento da aplicação, incluindo as etapas e abordagens utilizadas em sua construção.

3 Metodologia

A fim de obter os resultados esperados neste trabalho, os desenvolvimentos realizados estão representados no fluxograma da Figura 9, que detalha o passo a passo necessário para a implementação da automação aplicada a redes IP.

Figura 9 – Fluxograma de atividades.



Fonte: Autoria própria (2024).

3.1 Implementação da Topologia no GNS3

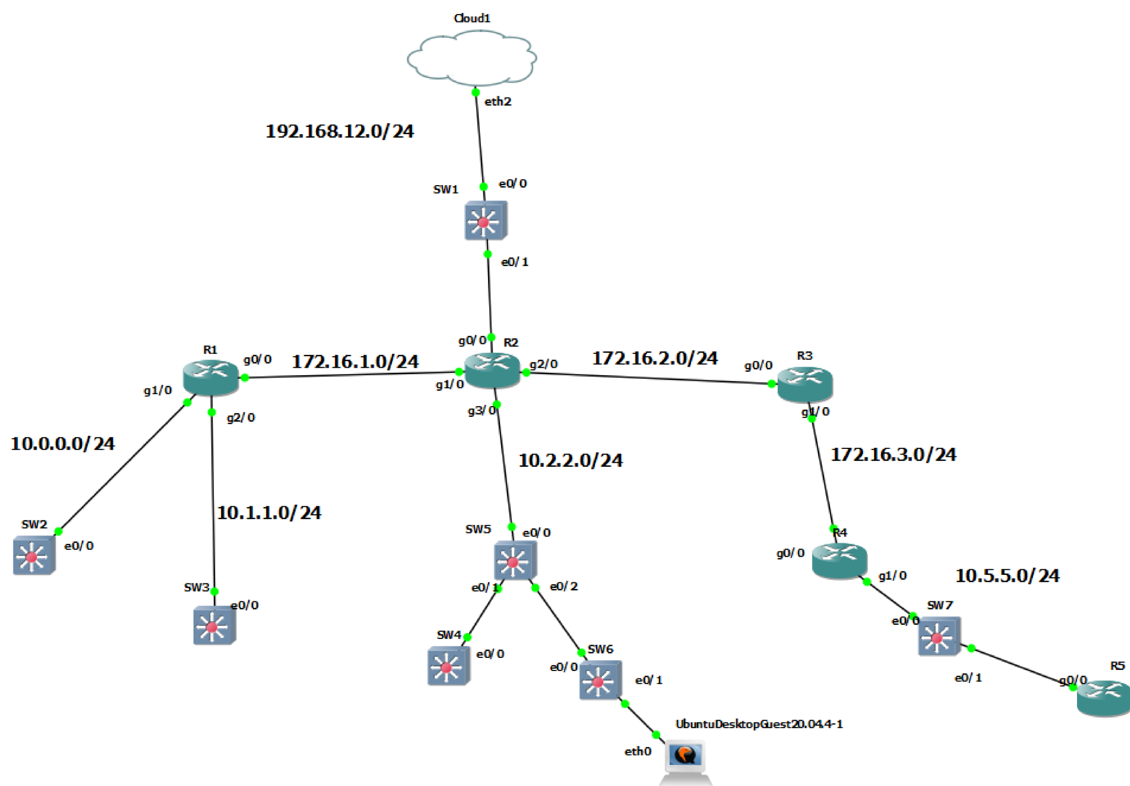
A arquitetura de uma rede é composta por diversas camadas que conectam dispositivos e controlam o tráfego de dados, além de garantir sua segurança. Esta arquitetura se diferencia de acordo com a necessidade e seu propósito de criação, sendo uma rede corporativa, governamental, *data center*, *ISP*, dentre inúmeros outros tipos. Generalizando, pode-se citar que os principais equipamentos que compõem uma rede são *switches* para conectar dispositivos locais, roteadores para gerenciar o tráfego entre redes diferentes, e *firewalls* para proteger a rede contra acessos não autorizados.

Dentro do contexto da criação de sua arquitetura, a segmentação da rede é essencial para manter uma topologia eficiente, segura e garantir seu monitoramento (FORTINET, 2023). Essa segmentação pode ser física, por meio de *firewalls* discretos, *switches*, cabeamento de interfaces específicas, separação por *sub-nets*, como também pode ser lógica, utilizando *Virtual Local Area Networks* (VLANs), *Access Control List* (ACLs) para controle de acesso, dentre outras tecnologias (ZSCALER, 2024).

Utilizando o *software* de emulação GNS3, foi desenvolvida a topologia de rede apresentada na Figura 10, composta por diversos equipamentos, criando um ambiente de laboratório que simula uma rede real. Esse laboratório pode ser segmentado tanto fisicamente quanto logicamente, permitindo o monitoramento e a implementação de automações. As imagens IOS, sistema operacional de rede desenvolvido pela Cisco, foram utilizadas para configurar a rede em conjunto com o software Dynamips, o que possibilitou a emulação do sistema operacional virtualizado IOSv (BLACKWELL, 2014). Adicionalmente,

foram utilizados QEMU e IOUs para a virtualização de outros sistemas dentro dessa topologia.

Figura 10 – Topologia implementada no emulador GNS3.



Fonte: Autoria própria (2024).

A versão inicial da rede é composta por roteadores (R1 ao R5) IOS Cisco 7200, os *switches* (SW1 ao SW7) IOUs e uma QEMU de uma VM Ubuntu 20.04.4, "UbuntuDesktopGuest". As especificações das *appliances* estão disponibilizadas na Tabela 1.

Tabela 1 – Imagens utilizadas na topologia.

Equipamento	**Tipo**	**Imagem**
Cisco 7200	Cisco IOSv	c7200-adventerprisek9-mz.124-24.T5.image
Cisco SW L2	Cisco IOU	i86bi-linux-l2-upk9-15.0b.bin
Ubuntu 20.04.4	QEMU	Ubuntu 20.04.4 (64bit).vmdk

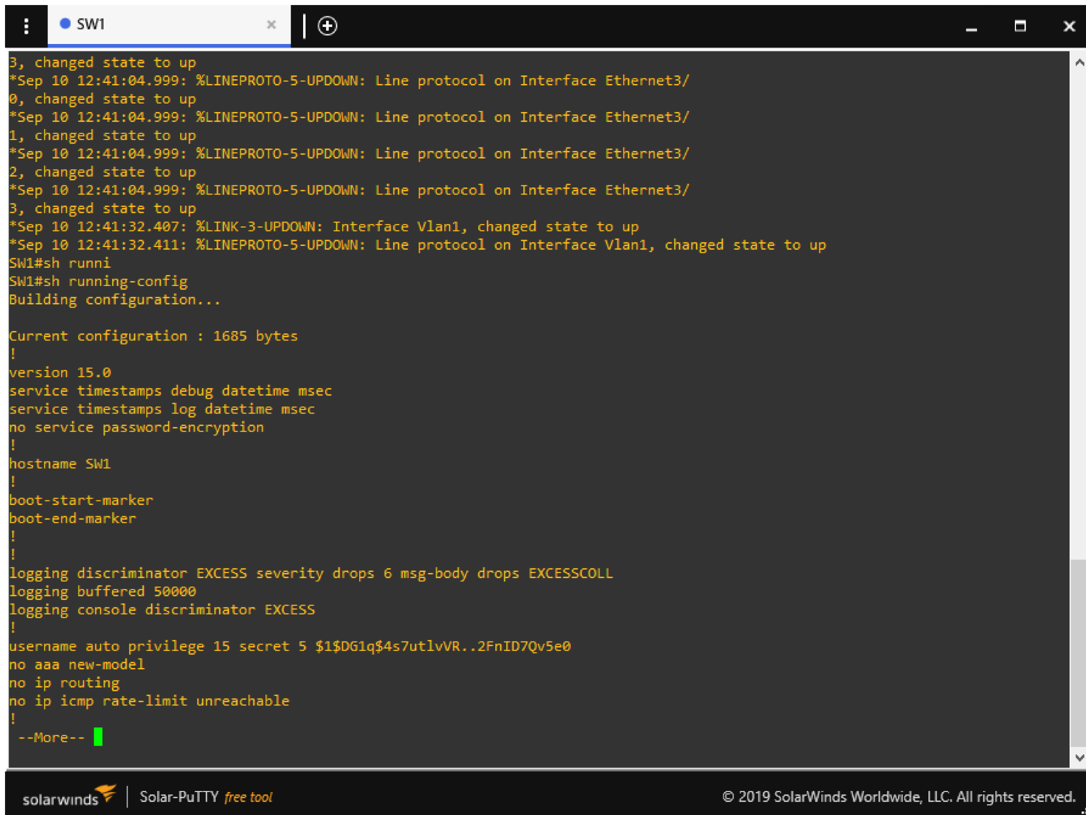
Fonte: Autoria própria (2024).

3.1.1 Configuração da Rede

As configurações de rede são realizadas utilizando o Solar Putty, uma ferramenta que utiliza um terminal CLI para acessar os equipamentos, neste caso, via protocolo *telnet*. Um exemplo de comando utilizado no *switch* SW1 via terminal pode ser observado

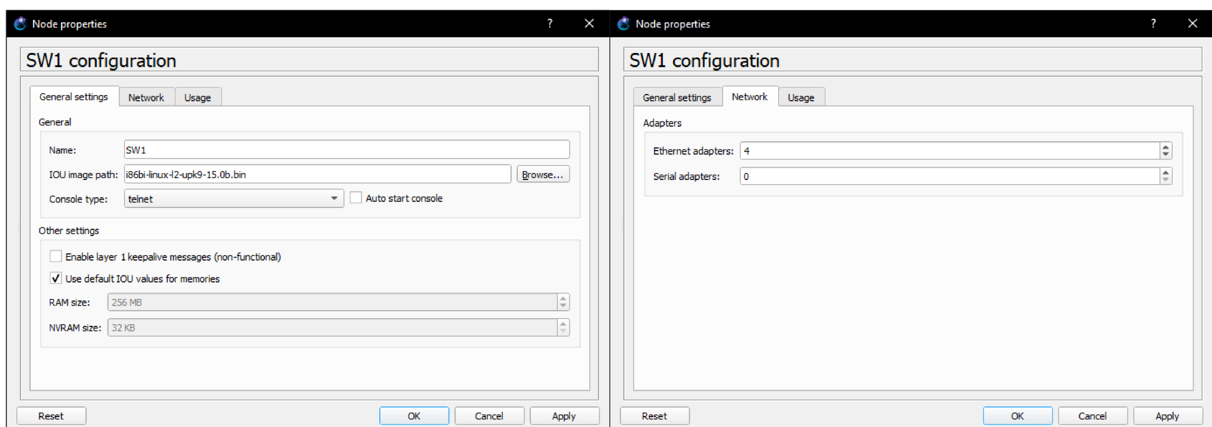
na Figura 11. A Figura 12 mostra o meio pelo qual o GNS3 permite a modificação da configuração de *hardware* dos equipamentos, valores de *Random Access Memory RAM* (RAM) e *Non-Volatile Random Access Memory* (NVRAM), assim como o número de adaptadores *Ethernet* que ele possui.

Figura 11 – Exemplo de comando utilizando o terminal Solar Putty.



Fonte: Autoria própria (2024).

Figura 12 – Configuração de *hardware* no GNS3.



Fonte: Autoria própria (2024).

A rede 192.168.12.0/24, onde estão localizados SW1 e R2 via interface g0/0 possui a Cloud1, uma *bridge* à rede local LAN, permitindo um acesso aos equipamentos utilizando

a máquina local. Foram utilizados endereços IP privados para os equipamentos, todos com máscaras de rede /24, possibilitando aumentar o número de equipamentos para testes sem que seja necessário grandes mudanças nas configurações de rede.

A fim de desenvolver uma ferramenta de automação, deve-se seguir dois princípios: os equipamentos devem ser capazes de comunicar-se entre si e a aplicação de automação deve ser capaz de acessá-los via SSH. O *switch* SW1 é utilizado como o acesso inicial ao resto da topologia, desta forma a interface VLAN 1 foi configurada com um endereço IP da rede local, e esta será utilizada como *management* VLAN, utilizada para controle remoto e monitoramento dos equipamentos (CISCO, 2020). Esta configuração pode ser observada na Figura 13.

Figura 13 – Configuração da VLAN 1 no SW1.

```
SW1(config)#int vlan 1
SW1(config-if)#ip address 192.168.12.100 255.255.255.0
```

Fonte: Autoria própria (2024).

Agora, para o acesso ao equipamento deve-se configurar primeiramente uma chave SSH, a credencial de acesso ao protocolo. Utilizando um domínio fictício *auto* para este laboratório, foi gerado um par de chaves RSA de 768 *bits*, por meio dos comandos verificados em (CISCO, 2023). Foram configuradas as linhas *Virtual Teletype* (VTY), que permitem o acesso remoto ao equipamento, para trabalhar somente com o protocolo SSH na versão 2 utilizando a autenticação local. Por meio da Figura 14 pode-se visualizar como esta configuração foi desenvolvida para permitir o login remoto utilizando o usuário *auto*, com o mais alto nível de privilégio. Uma configuração equivalente foi utilizada para o resto dos *switches* presentes na rede.

Figura 14 – Configuração do protocolo SSH no SW1.

```
SW1(config)#ip domain-name auto
SW1(config)#crypto key generate rsa 768
SW1(config)#ip ssh version 2
SW1(config)#line vty 0 4
SW1(config-line)#transport input ssh
SW1(config-line)#login local
SW1(config)#username auto privilege 15 secret auto
```

Fonte: Autoria própria (2024).

Nesta topologia foram utilizados *switches Layer* 2, que operam apenas na camada de enlace comutando pacotes em uma mesma LAN por meio dos endereços *Media Access Control* (MAC), atributos únicos dos equipamentos que permitem sua identificação. Já os roteadores operam na camada de rede *Layer* 3, sendo responsáveis por encaminhar

pacotes entre as redes. Desta forma, se faz necessário configurar o roteamento entre as redes, sendo escolhido o protocolo *Open Shortest Path First* (OSPF) para este propósito, por ser dominante em redes intra-domínio, amplamente utilizado em redes corporativas (TANENBAUM; FEAMSTER; WETHERALL, 2022).

OSPF é um protocolo do tipo *link-state*, este tipo de protocolo é baseado em cinco princípios: descobrir equipamentos vizinhos e seus endereços de rede por meio de seus *link-state advertisements* (LSA), definir uma métrica de distância ou custo para cada vizinho, construir um pacote com as informações aprendidas, enviar e receber estes pacotes para os vizinhos e calcular o menor caminho para os outros roteadores (TANENBAUM; FEAMSTER; WETHERALL, 2022).

O OSPF faz parte dos *Interior Gateway Protocols* (IGPs), protocolos de roteamento utilizados dentro de um mesmo sistema autônomo (AS). O roteador mantém em seu banco de dados *link-sate* (LSDB) as informações de *link-state* e adjacência de seus vizinhos, sendo capaz de invalidar essas informações e recalcular as rotas caso necessário. Ele utiliza o algoritmo de Dijkistra para calcular o caminho mais curto para os próximos roteadores (NETWORKS, 2024), armazenando as informações em suas tabelas de roteamento. Os bancos topológicos de roteadores que estão na mesma área possuem exatamente as mesmas informações, como é o caso dos roteadores criados na topologia desenvolvida.

A Figura 15 mostra a configuração das interfaces e roteamento do roteador R2, vale ressaltar que a interface g0/0 utiliza um endereço IP da rede local fornecido pelo *Dynamic Host Configuration Protocol* (DHCP) configurado. O processo OSPF de identificador 1 foi populado com redes de área 0, considerada por padrão como a *backbone* da rede. Este padrão foi seguido pelos demais roteadores utilizados.

Figura 15 – Configuração de interfaces e protocolo OSPF no roteador R2.

```
R2(config)#int gigabitEthernet 0/0
R2(config-if)#ip address dhcp
R2(config-if)#no sh
R2(config)#int gigabitEthernet 1/0
R2(config-if)#ip address 172.16.1.2 255.255.255.0
R2(config-if)#no sh
R2(config)#int gigabitEthernet 2/0
R2(config-if)#ip address 172.16.2.2 255.255.255.0
R2(config-if)#no sh
R2(config)#interface gigabitEthernet 3/0
R2(config-if)#ip address 10.2.2.2 255.255.255.0
R2(config-if)#no sh
R2(config)#router ospf 1
R2(config-router)#network 192.168.12.0 0.0.0.255 area 0
R2(config-router)#network 172.16.1.0 0.0.0.255 area 0
R2(config-router)#network 172.16.2.0 0.0.0.255 area 0
R2(config-router)#network 10.2.2.0 0.0.0.255 area 0
```

Fonte: Autoria própria (2024).

Os demais dispositivos possuem uma configuração semelhante. Foi atribuída a

mesma configuração SSH à todos os equipamentos, as interfaces utilizadas pelos roteadores receberam endereços IPs e foram ativadas por meio do comando "*no shutdown*". Os *switches* receberam *management* VLANs para acesso remoto, o OSPF foi configurado em cada roteador para contemplar cada rede de suas interfaces, todas em área 0. Para a interface *Ethernet* do UbuntuDesktopGuest foi designado um endereço IP na rede pertencente à interface g3/0 do R2, utilizado como seu *gateway* padrão. A Tabela 2 apresenta todas as configurações de interface implementadas nos equipamentos.

Tabela 2 – Configuração das interfaces do equipamentos.

<table>
<tr><th>Equipamento</th><th>Interfaces</th><th>Máscara</th></tr>
<tr><td>R1</td><td>GigabitEthernet0/0: 172.16.1.1
GigabitEthernet1/0: 10.0.0.1
GigabitEthernet2/0: 10.1.1.1</td><td rowspan="13">/24</td></tr>
<tr><td>R2</td><td>GigabitEthernet0/0: DHCP
GigabitEthernet1/0: 172.16.1.2
GigabitEthernet2/0: 172.16.2.2
GigabitEthernet3/0: 10.2.2.2</td></tr>
<tr><td>R3</td><td>GigabitEthernet0/0: 172.16.2.3
GigabitEthernet1/0: 172.16.3.3</td></tr>
<tr><td>R4</td><td>GigabitEthernet0/0: 172.16.3.4
GigabitEthernet1/0: 10.5.5.4</td></tr>
<tr><td>R5</td><td>GigabitEthernet0/0: 10.5.5.5</td></tr>
<tr><td>SW1</td><td>Vlan1: 192.168.12.100</td></tr>
<tr><td>SW2</td><td>Vlan1: 10.0.0.2</td></tr>
<tr><td>SW3</td><td>Vlan1: 10.1.1.3</td></tr>
<tr><td>SW4</td><td>Vlan1: 10.2.2.4</td></tr>
<tr><td>SW5</td><td>Vlan1: 10.2.2.5</td></tr>
<tr><td>SW6</td><td>Vlan1: 10.2.2.6</td></tr>
<tr><td>SW7</td><td>Vlan1: 10.5.5.7</td></tr>
</table>

Fonte: Autoria própria (2024).

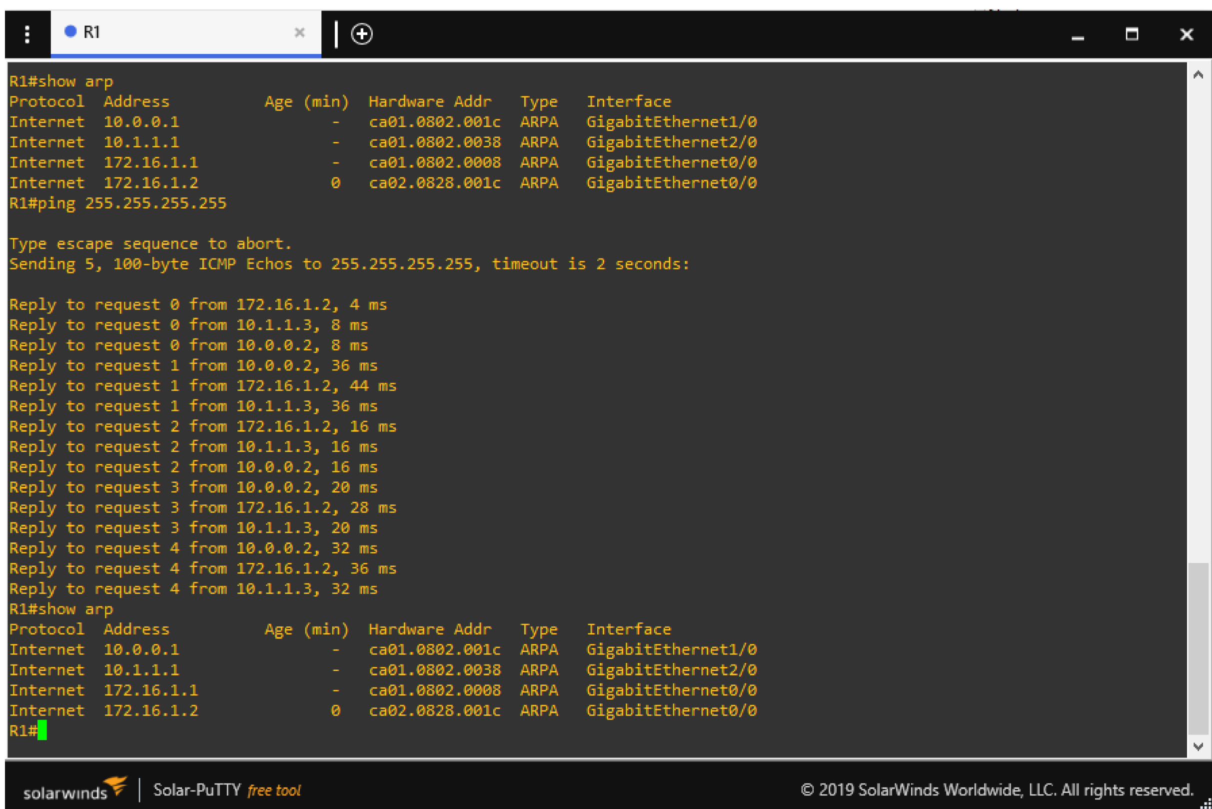
3.1.2 Testes de conexão

Ainda seguindo os princípios de funcionamento da ferramenta de automação, após o provisionamento inicial dos equipamentos é necessário popular a tabela *Address Resolution Protocol* (ARP) dos roteadores. Responsável por mapear os endereços IP de uma rede LAN aos respectivos endereços MAC de cada equipamento, o protocolo ARP é responsável por questionar e receber a resposta de qual equipamento possui um determinado endereço IP(TANENBAUM; FEAMSTER; WETHERALL, 2022). Os mapeamentos realizados pelo protocolo são guardados na tabela, ou *cache*, ARP. Uma maneira simples de atualizar os registros deste *cache* com um novo equipamento é utilizar um simples comando de *ping* a uma de suas interfaces ou ao *broadcast* local 255.255.255.255, enviando uma mensagem

limitada a todos os equipamentos da mesma sub-rede do roteador. Em um cenário real, a tabela ARP dos equipamentos seria populada inevitavelmente ao verificar essa comunicação entre elas, já que o teste de conexão é uma prática comum a fim de verificar o funcionamento das configurações e necessidade de correções.

Ao observar a Figura 16, pode-se notar os endereços IP e endereços MAC correspondentes antes e após um *ping* ao *broadcast* local. O roteador R1 apresenta na tabela inicial os endereços IP atrelados às interfaces do próprio equipamento, apresentando o endereço MAC da Placa de Interface de Rede (NIC). Após a utilização do *ping*, o roteador recebe respostas de todos os equipamentos das redes em que está conectado, atualizando o *cache*.

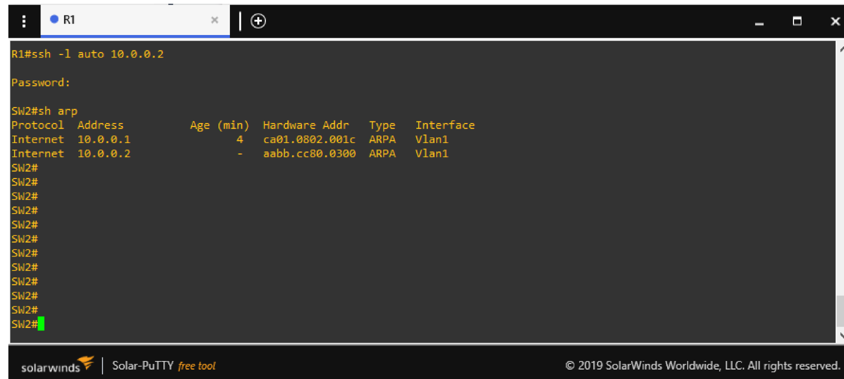
Figura 16 – Tabela ARP do roteador R1 antes e após um ping no *broadcast* local.



Fonte: Autoria própria (2024).

É possível ainda verificar que a conexão SSH aos equipamentos foi configurada com sucesso. Utilizando como exemplo uma conexão SSH do roteador R1 ao *switch* SW2, como observado na Figura 17, vê-se que no comando ssh é passado o parâmetro -l para indicar o nome de usuário que será utilizado na conexão, que o dispositivo foi configurado para utilizar credenciais presentes localmente. Colocando a senha do usuário, é possível concluir o acesso ao dispositivo remotamente.

Figura 17 – Processo de conexão SSH do roteador R1 para o switch SW2.



Fonte: Autoria própria (2024).

3.2 Automação Python Local

A fim de desenvolver o *script* de automação utilizou-se o *Visual Studio Code*, editor de código desenvolvido pela Microsoft, que garante suporte a versionamento e *debbuging*, que são de grande ajuda para a formulação de aplicações robustas. Pode-se destacar também os terminais CLI disponibilizados pelo *software*, cruciais para o desenvolvimento do projeto.

3.2.1 Ambiente Virtual e Projeto Django

A fim de manter o versionamento da aplicação e sua reprodutibilidade, um ambiente virtual é iniciado no *Visual Studio Code*, em uma pasta dedicada ao projeto. Utilizando o terminal *shell*, a biblioteca Django é instalada utilizando o instalador de pacotes python *pip*, que será utilizado para obter todos os demais pacotes necessários. O projeto Django é iniciado através do comando *django-admin startproject portal* e a aplicação onde o código será desenvolvido é iniciada utilizando o comando *django-admin startapp app_redes*, devidamente indicada nas configurações de *INSTALLED_APPS*.

3.2.2 Desenvolvimento dos *Models*, *Views* e *Templates*

Inicialmente, é necessário criar um *model* no arquivo *models.py* para guardar as informações dos equipamentos da topologia no GNS3, deste modo o código da Figura 18 foi desenvolvido. Para esta tabela são guardados três tipos de informações diferentes: *hostname* do equipamento, suas interfaces utilizadas e o seu *status* atual, ativo ou não. Ao utilizar os comandos *python manage.py makemigrations* e *python manage.py migrate*, o Django gerencia as mudanças e criações realizadas nos *models*, aplicando-as ao banco de dados. Por padrão, o SQLite vem em conjunto com o Django, garantindo um banco leve e de fácil utilização para projetos simples.

Figura 18 – Código presente em *models.py* para a criação do *model* Equipamentos.

```
from django.db import models

class Equipamentos(models.Model):
    hostname =models.CharField(max_length=255, unique=True)
    interfaces =models.CharField(max_length=255)
    status =models.CharField(max_length=255)
    class Meta:
        app_label ='app_redes'
        db_table ='equipamentos'
        ordering=['id']
```

Fonte: Autoria própria (2024).

Após a criação da tabela de dados, pode-se começar a desenvolver as *views*. A Figura 19 mostra a criação da *view* inicial do portal *Web*. Utilizando o arquivo *views.py*, a função *index* foi criada, dentro dela o *model* equipamentos é acessado e todo seu conteúdo é passado para renderização na página. Além disso, foi desenvolvida uma função de excluir um registro da tabela no banco de dados por meio de seu *id*, utilizando um botão na página. A estrutura visual da página está presente no *template index.html*, localizados na pasta *templates* do diretório do projeto. Para que a aplicação reconheça as *views*, seus caminhos são passados no arquivo *urls.py*.

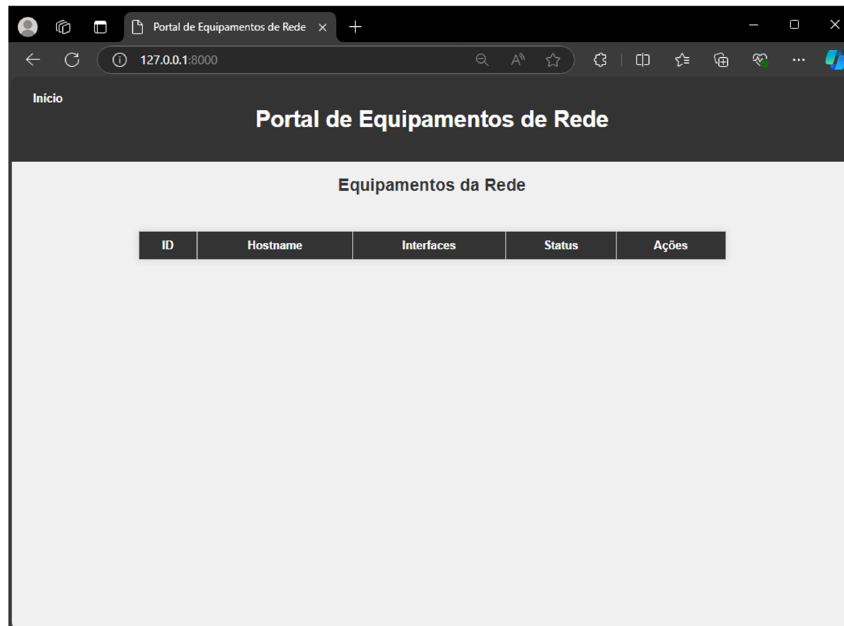
Figura 19 – Código presente em *views.py* para criação da *view index*.

```
def index(request):
    if request.method =='POST':
        equipamento_id =request.POST.get('equipamento_id')
        if equipamento_id:
            Equipamentos.objects.filter(id=equipamento_id).delete()
            return redirect('index')

    equipamentos =Equipamentos.objects.all().order_by('hostname')
    return render(request, 'index.html', {'equipamentos': equipamentos})
```

Fonte: Autoria própria (2024).

A fim de visualizar o portal é necessário acessar, utilizando o navegador, o *link* para o servidor de desenvolvimento *localhost*: http://127.0.0.1:8000/, gerado através do comando *python manage.py runserver*. A Figura 20 mostra a página inicial do portal, onde as informações dos equipamentos são mostradas por meio de uma tabela e as *views* acessadas por meio de botões. É necessário seguir este mesmo padrão de desenvolvimento no *framework* para desenvolver as demais *views* com *scripts* de automação.

Figura 20 – Página inicial do portal *Web*.



Fonte: Autoria própria (2024).

3.2.2.1 *Discovery*

A fim de descobrir os equipamentos que compõem a topologia, a *view discovery* foi desenvolvida, cujo código completo está apresentado no apêndice A. O objetivo é conectar-se ao *switch* SW1 e, a partir dele, identificar outros dispositivos conectados, coletando informações como IPs, interfaces e status destes dispositivos. Para este propósito, utiliza-se a biblioteca Netmiko, que facilita a conexão com dispositivos de rede via SSH, permitindo a execução de comandos. A documentação completa contendo cada uma das funções e métodos utilizados neste trabalho estão disponíveis em (BYERS, 2024b).

Inicialmente, é necessário definir o dicionário *device*, apresentado na Figura 21. Este armazena os parâmetros de conexão essenciais, como o endereço IP do *host*, 192.168.12.100, o tipo de dispositivo, *cisco_ios*, e as credenciais de autenticação, usuário e senha, definidos na configuração realizada previamente. O dicionário é passado como argumento para a função *ConnectHandler* da Netmiko, que utiliza essas informações para estabelecer a sessão SSH.

A variáveis e estruturas de dados que auxiliam no controle e monitoramento dos dispositivos que já foram acessados estão apresentadas na Figura 22. O contador *co* controla se o equipamento acessado é o *switch* SW1, ou seja, a conexão inicial. A definição da topologia e forma de configuração dos equipamentos requer a realização de saltos SSH a fim de acessar os demais dispositivos. A variável *listaHostsVisitados* é um *set* utilizado para armazenar os IPs dos dispositivos que foram visitados na rede. O *set* é utilizado dado a sua característica intrínseca de que seus valores armazenados não podem se repetir. A lista *infoEqs* guarda as informações extraídas das interfaces de cada equipamento: os

Figura 21 – Definição do dicionário *device*.

```
def discovery(request):
    device ={
        'host': '192.168.12.100',
        'device_type': 'cisco_ios',
        'username': 'auto',
        'password': 'auto',
    }
```

Fonte: Autoria própria (2024).

endereços IP atribuídos e o estado das interfaces, *UP* ou *DOWN*.

Figura 22 – Definição das variáveis de controle e início do processo de *discovery*.

```
def discovery(request):
        ###
        co =0
        listaHostsVisitados =set()
        infoEqs =[]
        visitedDevices =[]
        hosts =['192.168.12.100']

        connection =ConnectHandler(**device)
        Equipamentos.objects.all().update(status="DOWN")
        access_hosts(connection, hosts, co, listaHostsVisitados, visitedDevices,
                                                infoEqs)
        ###
```

Fonte: Autoria própria (2024).

O registro dos dispositivos que já foram acessados anteriormente são guardados na lista *visitedDevices*, visando evitar tentativas de reconexão desnecessárias. Por fim, a lista contendo os IPs que devem ser acessados está armazenada na variável *hosts*, iniciada apenas com o endereço IP do *switch* SW1.

A função *ConnectHandler* seleciona a classe apropriada e cria um objeto baseado no *device_type* (BYERS, 2024b), sendo utilizada para a primeira conexão. Antes de iniciar a descoberta, o valor para *status* de todos os equipamentos que estejam no banco de dados é atualizado para *DOWN*, presumindo que nenhum equipamento esteja ativo até que sejam confirmados como *UP* durante a descoberta. Em seguida, a função recursiva *access_hosts* é chamada para iniciar o processo de descoberta, iniciando com as variáveis de controle definidas, sendo utilizada repetidas vezes até o final do processo de *discovery*.

A função *access_hosts()* é o núcleo do processo de descoberta de dispositivos de rede. Ela é responsável por iterar sobre os endereços IP encontrados, conectar-se aos equipamentos que carregam os IPs descobertos, extrair informações de suas interfaces,

atualizar o status dos dispositivos no banco de dados e, por fim, propagar a descoberta de novos dispositivos conectados para novas conexões SSH.

O *set* novaListasIPs é inicializado com o intuito de armazenar novos IPs descobertos durante a interação com o dispositivo atual. Esse conjunto garante que a função possa chamar a si mesma recursivamente para acessar dispositivos recém-descobertos. A função percorre cada IP presente na lista *listaIPs*, verificando se o dispositivo já foi visitado por meio da lista *listaHostsVisitados*, caso o *host* tenha sido acessado anteriormente, ele é ignorado para evitar reconexões desnecessárias.

O acesso aos equipamentos é realizado dentro de um bloco *try except* para que caso o *script* falhe ao acessar um dispositivo, ele possa continuar percorrendo a *listaIPs*. O *hostname* do dispositivo é identificado por meio do método *find_prompt()*, pois ele retorna o texto que aparece no terminal atual do equipamento. Com o objetivo de acessar os equipamentos da topologia, o método *write_channel()* é utilizado, pois permite escrever dados diretamente no terminal (BYERS, 2024b). Nele, é escrito o comando para realizar uma conexão SSH por meio das credenciais locais do dispositivo, anteriormente comentado, junto ao método *redispatch()*, que muda dinamicamente a classe do objeto Netmiko para a classe apropriada (BYERS, 2024b). A biblioteca *time*, nativa do Python, é importante neste processo, pois permite a resolução do comando para seguir com a autenticação no dispositivo. Esta composição inicial do código que começa o acesso à topologia está apresentada na Figura 23

No dispositivo acessado, são utilizados comandos a fim de obter informações de configuração. Inicialmente o comando *show ip interface brief* é enviado utilizando o método *send_command()*, ele mostra o *status* de usabilidade das interfaces configuradas para vários endereços IP (CISCO, 2017). Um exemplo da saída deste comando quando utilizado no roteador R2 é observado na Figura 24. Desta forma, estes endereços IP são capturados utilizando a função *extract_ips()* e guardados na lista *listaHostsVisitados*, evitando o retorno ao equipamento. As informações de interface e IP são agrupadas utilizando a função *extract_ips_with_interfaces()*, permitindo que elas sejam salvas no banco de dados utilizando o método *update_or_create()*, que atualiza um objeto utilizando os argumentos fornecidos (DJANGO, 2024a).

É importante salientar que o processo de extrair informações, cujo código está presente na Figura 25, só atua quando o equipamento não foi visitado, este controle é realizado pela lista *visitedDevices*, que guarda os *hostnames* já identificados. Ainda dentro do equipamento, o comando *show arp* é utilizado para obter a tabela ARP com os endereços IP de equipamentos, vizinhos ou não, como foi mostrado na Figura 16. Deste modo, a recursão da função *access_hosts()* é realizada para acessar a lista *novaListaIPs*, que guarda os endereços IP identificados. Caso não sejam detectados novos IPs, o método *disconnect()* é ativado fechando a conexão SSH (BYERS, 2024b), liberando os recursos do sistema e

Figura 23 – Código de iteração sobre os *hosts* e acesso aos equipamentos.

```
def discovery(request):
    ###
        def access_hosts(connection, listaIPs, co, listaHostsVisitados, visitedDevices,
                                                          infoEqs):
            novaListaIPs =set()
            for host in listaIPs:
                if host in listaHostsVisitados:
                    continue
                device['host'] =host
                try:
                    hostname =""
                    print(f"Conectando a {host}")
                    if co ==0:
                        hostname =connection.find_prompt()
                    else:
                        connection.write_channel(f'ssh -l auto {host}\n')
                        time.sleep(2)
                        connection.write_channel(f'auto\n')
                        time.sleep(2)
                        redispatch(connection, device_type='cisco_ios')
                    ###
            except Exception as e:
                print(f"Falha...")
                listaHostsVisitados.add(host)
        ###
```

Fonte: Autoria própria (2024).

Figura 24 – Utilização do comando *show ip interface brief* no roteador R2.

```
R2#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                unassigned      YES NVRAM  administratively down down
GigabitEthernet0/0         192.168.12.16   YES DHCP   up                    up
GigabitEthernet1/0         172.16.1.2      YES NVRAM  up                    up
GigabitEthernet2/0         172.16.2.2      YES NVRAM  up                    up
GigabitEthernet3/0         10.2.2.2        YES NVRAM  up                    up
```

Fonte: Autoria própria (2024).

redirecionando o usuário à página inicial.

As funções *extract_ips()* e *extract_ips_with_interfaces()*, indicadas na Figura 26, exemplificam como a interação com as demais bibliotecas Python são de grande ajuda no desenvolvimento de *scripts* para automações. O módulo *re* utiliza expressões regulares, uma sequência de caracteres que formam um padrão de busca (W3SCHOOLS, 2024c), sendo utilizado neste código para buscar o padrão para endereços IP. A função *extract_ips()* é responsável por extrair todos os endereços IP presentes na saída do comando *show ip interface brief* ou comando *show arp*, já a função *extract_ips_with_interfaces()* é mais específica: além de extrair endereços IP, ela associa esses IPs à suas respectivas interfaces

Figura 25 – Código para extração de informações e recursão do acesso aos equipamentos.

```
def discovery(request):
                ###
                showIp =connection.send_command('show ip interface brief')
                for i in set(extract_ips(showIp)):
                    listaHostsVisitados.add(i)
                listaHostsVisitados.add(host)
                infoEqs.append(extract_ips_with_interfaces(hostname, showIp))
                if hostname in visitedDevices:
                    print(f"{hostname} ja foi acessado anteriormente. Pulando para o
                                                               proximo equipamento."
                                                               )
                    continue
                hostname_obj, interfaces =extract_ips_with_interfaces(hostname,
                                                           showIp)
                equipamento, created =Equipamentos.objects.update_or_create(
                    hostname=hostname_obj,
                    defaults={
                        'interfaces': '\n'.join(interfaces),
                        'status': "UP",
                    }
                )


                visitedDevices.append(hostname)
                showArp =connection.send_command('show arp')
                ips =extract_ips(showArp)
                for ip in ips:
                    if ip not in listaHostsVisitados:
                        novaListaIPs.add(ip)
            ###
        if novaListaIPs:
            co=1
            access_hosts(connection, list(novaListaIPs), co, listaHostsVisitados,
                                                   visitedDevices, infoEqs)
        connection.disconnect()
        return visitedDevices
    ###
```

Fonte: Autoria própria (2024).

e seu status, caso esteja *UP* ou *DOWN*.

3.2.2.2 *Backup*

A fim de salvar um *backup* dos equipamentos presentes na topologia, a *view realizarBackupEquipamentos* foi desenvolvida, apresentada por completo no apêndice B. Assim como na *view discovery*, o código inicia realizando a conexão via *ConnectHandler()* ao *switch* SW1, definido pelo dicionário *device*, como é observado Figura 27. Logo após, o banco de dados é acessado e são identificados do *model* Equipamentos todos dispositivos que encontram-se ativos na rede. Esta informação é guardada na variável *equipamentosUp*,

Figura 26 – Código das funções utilizadas para extrair endereços IP e interfaces utilizando a biblioteca *re*.

```
def discovery(request):
    ###
        def extract_ips(show_ip_output):
            ipPattern =r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
            ipAddresses =re.findall(ipPattern, show_ip_output)

            return ipAddresses

        def extract_ips_with_interfaces(hostname, show_ip_output):
            ipInterfacePattern =r'^(\S+)\s+([\d.]+)\s+\w+\s+\w+\s+(\w+)'
            matches =re.findall(ipInterfacePattern, show_ip_output, re.IGNORECASE | re.
                                                        MULTILINE)

            ipInterfaceList =[]

            for match in matches:
                interface =match[0]
                ipAddress =match[1]
                status =match[2]
                if status.lower() =='up':
                    ipInterfaceList.append(f"Interface {interface}: {ipAddress}")

            info =ipInterfaceList
            return hostname, info
        ###
```

Fonte: Autoria própria (2024).

um *QuerySet*, que representa uma coleção de objetos do banco de dados (DJANGO, 2024a).

Figura 27 – Código inicial *view realizarBackupEquipamento* que acessa o *switch* SW1 e o banco de dados do *model* Equipamentos.

```
def realizarBackupEquipamentos(request):
    def extract_ips(show_ip_output):
        ip_pattern =r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
        ip_addresses =re.findall(ip_pattern, show_ip_output)
        return ip_addresses
    device ={
        'host': '192.168.12.100',
        'device_type': 'cisco_ios',
        'username': 'auto',
        'password': 'auto',
    }
    connection =ConnectHandler(**device)
    equipamentosUp =Equipamentos.objects.filter(status="UP").order_by('hostname')
    ###
```

Fonte: Autoria própria (2024).

De posse dos equipamentos ativos, um *loop* é criado para acessar cada um deles. A tupla *equiInt* armazena o *hostname* e endereços IP de suas interfaces, obtidas por meio da função *extract_ips()*, idêntica à utilizada na *view*, aplicada ao atributo *interfaces*, campo do *model* Equipamentos. Assim, a conexão SSH aos dispositivos tenta ser realizada por meio de cada uma de suas interfaces, pulando para o próximo equipamento assim que obtêm-se comunicação com alguma delas ou nenhuma. Dentro do equipamento, o comando *show running-config* é utilizado, pois ele exibe o conteúdo do arquivo de configuração em execução no momento (CISCO, 2010). Esta saída é salva no dicionário *backups*, associada ao *hostname* de seu equipamento, como pode ser observado na Figura 28, que mostra a parte do código de acesso e *backup*.

Figura 28 – Código da *view realizarBackupEquipamento* que acessa os equipamentos e salva sua configuração.

```
def realizarBackupEquipamentos(request):
    ###
    backups ={}
    for equipamento in equipamentosUp:
        equiInt =equipamento.hostname, equipamento.interfaces
        listaIntEq =extract_ips(equiInt[1])

        for host in listaIntEq:
            try:
                print(f'Conectando ao {equiInt[0]}')
                connection.write_channel(f'ssh -l auto {host}\n')
                time.sleep(2)
                connection.write_channel(f'auto\n')
                time.sleep(2)
                redispatch(connection, device_type='cisco_ios')
                configBackup =connection.send_command('show running-config')
                backups[equiInt[0]] =configBackup
                break
            except:
                pass
    ###
```

Fonte: Autoria própria (2024).

A fim de salvar os arquivos de configuração atualizados, o módulo *Python os* é utilizado, pois ele possui métodos para a interação com o sistema operacional, como a criação de arquivos e diretórios (W3SCHOOLS, 2024b). Através do código observado na Figura 29, o diretório *backups_equipamentos* é criado, caso ainda não exista, e para cada *hostname* no dicionário *backups*, é gerado um arquivo de texto que contém a configuração do equipamento. sobrescrevendo arquivos já existentes. Após sua execução, o usuário é redirecionado para a *view backupEquipamentos*, criada para mostrar os arquivos de configuração já salvos.

A partir da *view backupEquipamentos*, cuja interface está apresentada na Figura

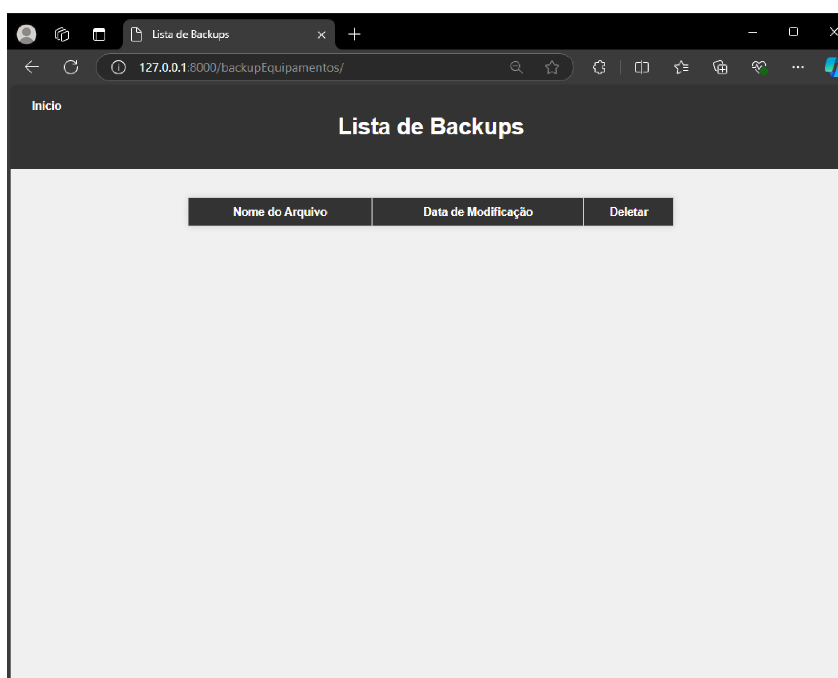
Figura 29 – Código da *view realizarBackupEquipamento* que salva em arquivos texto a configuração dos equipamentos.

```
def realizarBackupEquipamentos(request):
    ###
    backupDir =os.path.join(os.path.dirname(__file__), 'backups_equipamentos')
    os.makedirs(backupDir, exist_ok=True)

    for hostname, config in backups.items():
        file_path =os.path.join(backupDir, f'{hostname}_backup.txt')

        if os.path.exists(file_path):
            os.remove(file_path)

        with open(file_path, 'w') as backup_file:
            backup_file.write(config)

    connection.disconnect()
    return redirect('backupEquipamentos')
```

Fonte: Autoria própria (2024).

30 e o código apresentado no apêndice C, os arquivos de configuração dos equipamentos são listados, permitindo ao usuário baixá-los e deletá-los por meio de *links* personalizados para cada arquivo presente no diretório, como observado na Figura 31. Caso o arquivo não seja encontrado, o portal retorna um erro ao usuário.

Figura 30 – Página de backups do portal *Web*.



Fonte: Autoria própria (2024).

O método do módulo *os listdir()* utiliza o caminho presente na variável*backupFolder*

Figura 31 – Código da *view backupEquipamento* que permite baixar e deletar os arquivos de configuração dos equipamentos.

```
def backupEquipamentos(request):
    backupFolder =os.path.join(os.path.dirname(__file__), 'backups_equipamentos')

    if 'delete' in request.GET:
        fileName =urllib.parse.unquote(request.GET['delete'])
        filePath =os.path.join(backupFolder, fileName)

        if os.path.exists(filePath):
            os.remove(filePath)
            return redirect('backupEquipamentos')
        else:
            raise Http404("Arquivo nao encontrado")

    if 'file' in request.GET:
        fileName =urllib.parse.unquote(request.GET['file'])
        filePath =os.path.join(backupFolder, fileName)

        if os.path.exists(filePath):
            with open(filePath, 'rb') as f:
                response =HttpResponse(f.read(), content_type="application/octet-stream"
                                                )
                response['Content-Disposition'] =f'attachment; filename="{fileName}"'
                return response
        else:
            raise Http404("Arquivo nao encontrado")
    ###
```

Fonte: Autoria própria (2024).

para identificar os arquivos, percorrendo cada um deles, obtendo sua data de modificação. Esta data é formatada para um padrão acessível ao usuário por meio do módulo *datetime*, que trabalha com datas como objetos (W3SCHOOLS, 2024a). A biblioteca *urllib* é utilizada neste caso para codificar o nome do arquivo de forma que ele possa ser incluído em uma URL, isto é realizado por meio da função *urllib.parse.quote()*, deste modo os arquivos podem ser salvos sem problemas.

Em suma, para cada arquivo encontrado, é armazenada na lista *backups* uma tupla com o nome do arquivo, seu nome codificado e a data formatada, como pode ser observado na Figura 32. Esta lista é tratada no *template* da *view*, garantindo ao usuário a manipulação dos dados referentes a configuração dos equipamentos.

3.2.2.3 Verificar Equipamento

A *view verificarEquipamento*, apresentada no apêndice D, foi desenvolvida no intuito de facilitar a inclusão de novos dispositivos na topologia de rede. Para que não seja necessário realizar o *discovery* de todos os equipamentos, utilizando esta *view* o usuário

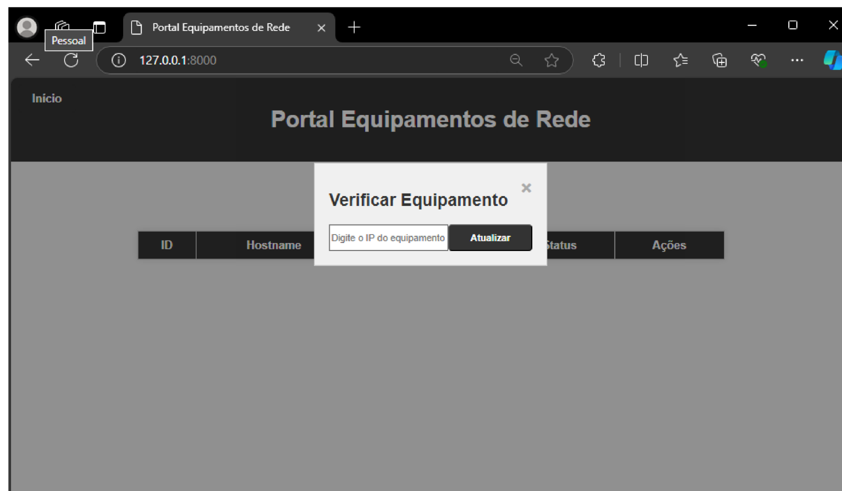
Figura 32 – Código da *view backupEquipamento* que lista os arquivos de configuração e suas datas de salvamento.

```
def backupEquipamentos(request):
    ###
    backups =[]
    for fileName in os.listdir(backupFolder):
        filePath =os.path.join(backupFolder, fileName)
        if os.path.isfile(filePath):
            timestamp =os.path.getmtime(filePath)
            formatted_time =datetime.fromtimestamp(timestamp).strftime('%d/%m/%Y %H:%M:%
                                                                        S')
            backups.append((fileName, urllib.parse.quote(fileName), formatted_time))

    return render(request, 'lista_backups.html', {'backups': sorted(backups)})
```

Fonte: Autoria própria (2024).

consegue atualizar ou adicionar um novo equipamento por meio do endereço IP de uma de suas interfaces, como mostra a Figura 33.

Figura 33 – Opção da *view verificarEquipamento* para acessar um equipamento específico da rede.



Fonte: Autoria própria (2024).

Como é demonstrado no código da Figura 34, a *view* inicia-se assim que o endereço IP do equipamento é digitado e o formulário enviado, o *script* então acessa o *switch* de gerência SW1 e então o equipamento desejado. Por meio dos comandos *show ip interface brief* e *show running-config*, utilizados nas *views* anteriores, obtém-se as informações necessárias para popular o banco de dados com as informações do equipamento, assim como para a criação do arquivo texto de configuração.

A Figura 35 mostra a página inicial do portal *Web* finalizada. Ao centro, localiza-se a tabela para a visualização dos equipamentos presentes no banco de dados do *model Equipamentos* e, ao canto inferior esquerdo, os botões que permitem o acesso às *views*

Figura 34 – Código da *view verificarEquipamento* acessa um equipamento e atualizada suas informações banco de dados e arquivo de texto.

```
def verificarEquipamento(request):
    ###
    if request.method =='POST':
        ip =request.POST.get('ip')
        try:
            connection =ConnectHandler(**device)
            print(f'Conectando ao {ip}')
            connection.write_channel(f'ssh -l auto {ip}\n')
            time.sleep(2)
            connection.write_channel(f'auto\n')
            time.sleep(2)
            redispatch(connection, device_type='cisco_ios')
            hostname =connection.find_prompt()
            print(f"Conectado a {hostname} {ip}")
            showIp =connection.send_command('show ip interface brief')

            hostname_obj, interfaces =extract_ips_with_interfaces(hostname, showIp)
            equipamento, created =Equipamentos.objects.update_or_create(
                hostname=hostname_obj,
                defaults={
                    'interfaces': '\n'.join(interfaces),
                    'status': "UP",
                }
            )
            configBackup =connection.send_command('show running-config')
            backup_dir =os.path.join(os.path.dirname(__file__), 'backups_equipamentos')
            os.makedirs(backup_dir, exist_ok=True)

            file_path =os.path.join(backup_dir, f'{hostname}_backup.txt')
            with open(file_path, 'w') as backup_file:
                backup_file.write(configBackup)
            connection.disconnect()
            ###
```

Fonte: Autoria própria (2024).

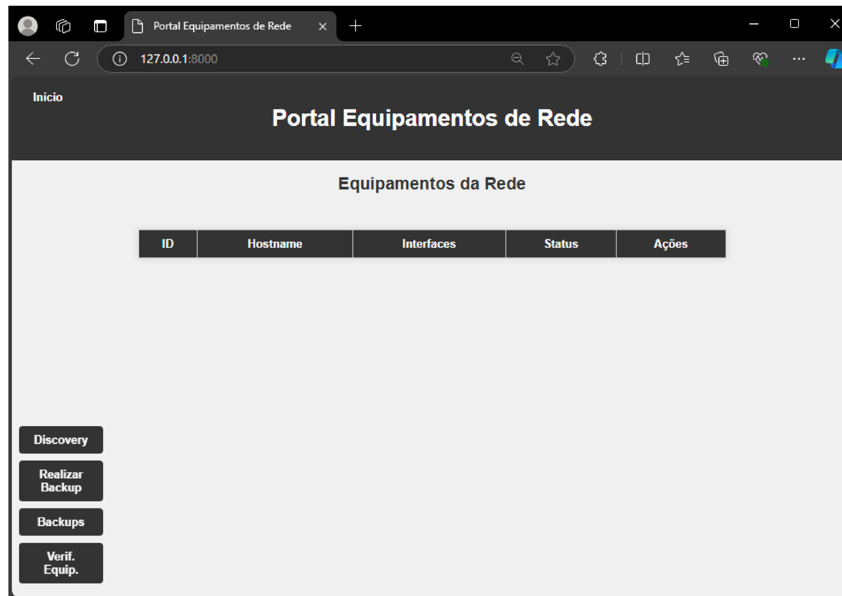
desenvolvidas. Estes componentes formam a interface em que o usuário consegue realizar o monitoramento da rede.

3.2.2.4 Automação de Tarefas utilizando Celery Beat e Redis

Utilizando a biblioteca Celery Beat e Redis como *message broker*, um *script* para um teste automatizado de conexão aos equipamentos da topologia foi desenvolvido, garantindo um monitoramento constante do *status* dos dispositivos presentes na infraestrutura da rede. Deste modo, as bibliotecas foram instaladas, o pacote *django-celery-beat* indicado nas configurações de *INSTALLED_APPS*. Por fim, o arquivo *celery.py* é utilizado para desenvolver o código da tarefa *testarConexao()*.

A função *testarConexao()*, apresentada no apêndice E, é registrada como uma

Figura 35 – Tela inicial do portal *Web* para a interação com topologia desenvolvida.



Fonte: Autoria própria (2024).

tarefa do Celery e marcada para ser executada periodicamente através do Celery Beat. Assim que a automação inicia, a função identifica todos os equipamentos registrados no banco de dados através do *model* Equipamentos. Esses equipamentos são armazenados no dicionário *listaEq*, onde a chave é o *hostname* do equipamento e os valores são os endereços IP de suas interfaces, obtidos pela função *extract_ips()* utilizada anteriormente.

A conexão inicial é realizada ao *switch* SW1, e por meio dele o comando *ping* é enviado a uma interface dos demais equipamentos presentes na *listaEq*, verificando assim se eles estão ativos ou não. Utilizando o método *send_command()*, um parâmetro de *delay* de dois segundos é utilizado para aguardar a saída do equipamento, esperando na saída o caracter # para considerar a execução do comando concluída. Caso a saída apresente o caracter , significa que a mensagem de sucesso foi exibida e o equipamento respondeu ao ping, o *script* então atualiza seu *status* no banco de dados para UP. Caso contrário, o *status* é atualizado para *DOWN*. A referida parte do código desta tarefa está apresentada na Figura 36, onde ainda é possível notar a utilização de *logs* para verificar o funcionamento do código em execução.

O pacote *django-celery-beat* facilita o agendamento das tarefas por meio do próprio painel de administração do Django. A fim de acessar este painel é necessário criar um usuário administrador por meio do comando *python manage.py createsuperuser*, indicando as credenciais de entrada. Através da Figura 37 é possível observar o acesso à esta aba do portal, nela a aba *Periodic Tasks* foi adicionada, permitindo adicionar a tarefa *testarConexao()* e agendar sua execução, como mostra a Figura 38.

Todos os pacotes e bibliotecas Python necessários para o funcionamento do portal

Figura 36 – Código da *view backupEquipamento* que lista os arquivos de configuração e suas datas de salvamento.

```
@app.task(bind=True)
def testarConexao(self):
    ###
    listaEq ={}
        model =apps.get_model(app_label='app_redes', model_name='Equipamentos')
        equipamentos =model.objects.all()
        for equipamento in equipamentos:
            listaEq[equipamento.hostname] =equipamento.interfaces

        listaHost =[]
        logger.info('Conectando ao dispositivo...')
        try:
            with ConnectHandler(**device) as connection:
                logger.info('Conexao estabelecida com sucesso.')
                for eq in listaEq.items():
                    ip =extract_ips(eq[1])[0]
                    try:
                        logger.info(f'Pingando IP {ip} do dispositivo {eq[0]}...')
                        pingTest =connection.send_command(f'ping {ip}', expect_string=r'#
                                                                ', delay_factor=2)
                        logger.info(f'Resposta do ping: {pingTest}')

                        if '!' in pingTest:
                            listaHost.append(eq[0])
                            model.objects.filter(hostname=eq[0]).update(status='UP')
                            logger.info(f'{eq[0]} esta UP.')
                        else:
                            model.objects.filter(hostname=eq[0]).update(status='DOWN')
                            logger.info(f'{eq[0]} esta DOWN.')
                    except Exception as e:
                        connection.write_channel("\036")
                        logger.error(f'Erro ao pingar {ip}: {str(e)}')
                        model.objects.filter(hostname=eq[0]).update(status='DOWN')
    ###
```

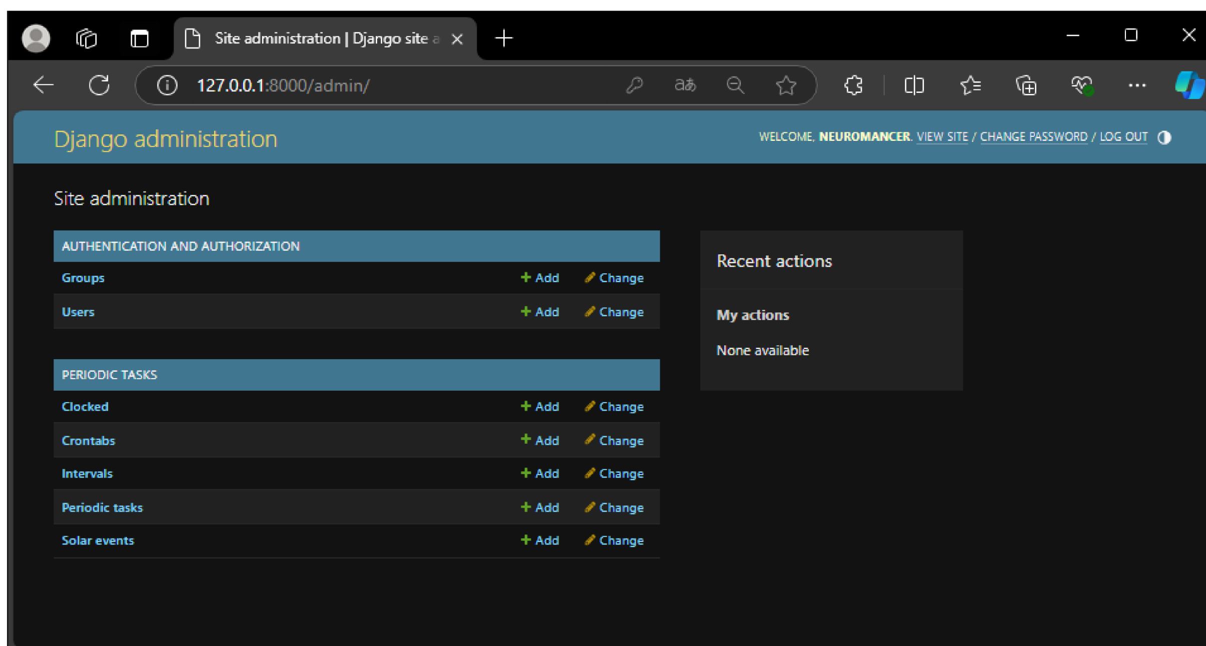
Fonte: Autoria própria (2024).

Web foram instalados. Desta forma, utilizando o comando *pip freeze > requirements.txt* cria-se o arquivo texto que contém os componentes necessários para a aplicação, que serão utilizados para o funcionamento do código em contêiner Docker.

3.3 Aplicação em Container Docker

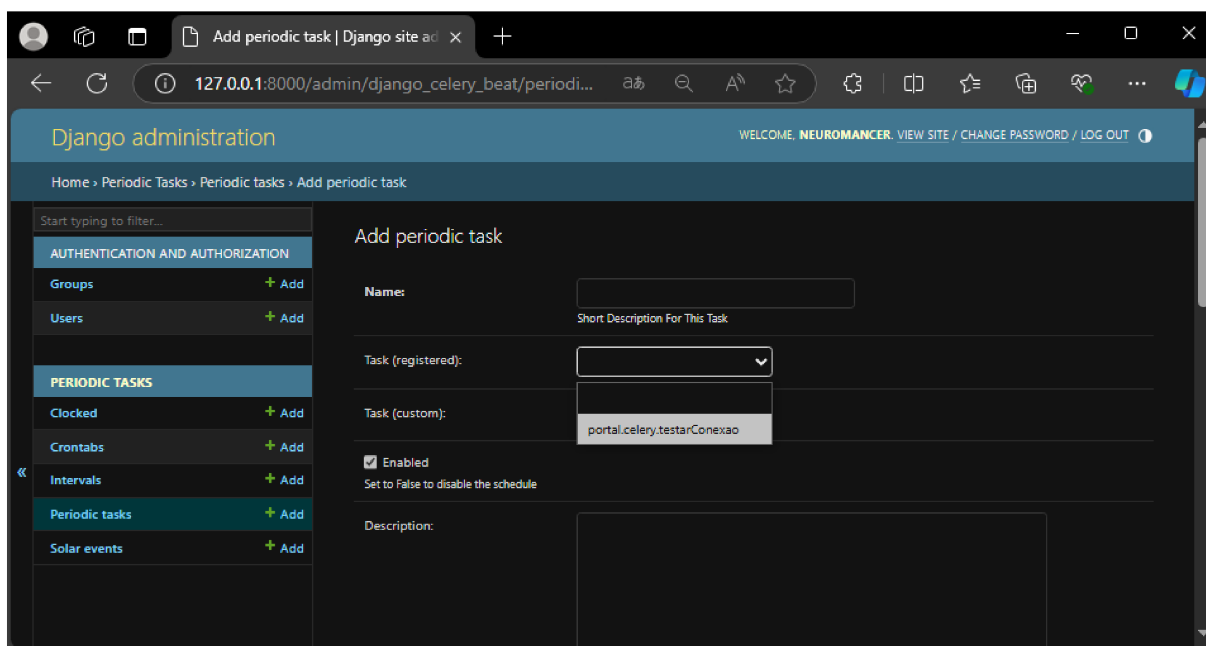
Uma máquina virtual Ubuntu utilizando o endereço IP 192.168.12.17 foi adicionada à rede conectada em modo *bridge* ao *switch* SW1, como mostra a Figura 39. Esta máquina foi instalada utilizando a imagem *ubuntu-22.04.2-desktop-amd64*, através do *software* de virtualização *Virtual Box*, utilizada na topologia por meio do *template* para VMs do GNS3. Este computador será utilizado para a execução da plataforma Docker, portanto, sua

Figura 37 – Aba administrador do portal *Web*.



Fonte: Autoria própria (2024).

Figura 38 – Aba de criação de tarefas periódicas do portal *Web*.

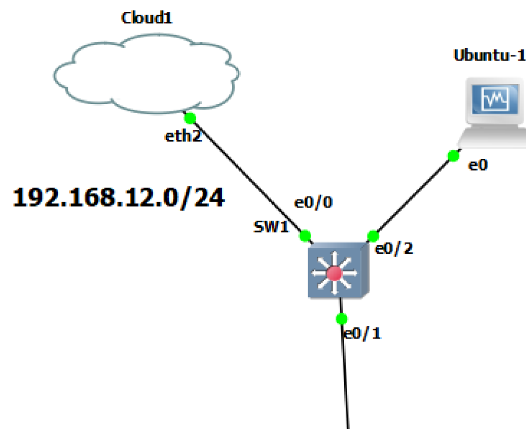


Fonte: Autoria própria (2024).

engine foi instalada via CLI.

A "dockerização" da aplicação *Web* desenvolvida é realizada por meio de quatro contêineres: django, celery, celery-beat e redis. Portanto, dois arquivos devem ser adicionados ao diretório da aplicação, a saber, o *Dockerfile* e *dockercompose.yml*. O *Dockerfile* é um documento texto que contém os comandos utilizados pelo usuário para construir uma imagem (DOCKER, 2024a), definindo o que será instalado e como será configurado o

Figura 39 – Conexão da VM Ubuntu utilizada para a plataforma docker ao *switch* SW1.



Fonte: Autoria própria (2024).

ambiente de um contêiner.

O código do Dockerfile, apresentado na Figura 40, executa as seguintes tarefas no contêiner: define a imagem base como Python 3.9 *slim*, atualiza a lista de pacotes, instala as dependências necessárias para a aplicação e remove pacotes previamente baixados. Em seguida, define o diretório de trabalho como */usr/src/app*, copia o arquivo *requirements.txt* para instalar os pacotes e, por fim, copia o código da aplicação para o diretório de trabalho.

Figura 40 – Código do arquivo Dockerfile da aplicação *Web*.

```
FROM python:3.9-slim

RUN apt-get update && apt-get install -y \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /usr/src/app

COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r requirements.txt

COPY . /usr/src/app/
```

Fonte: Autoria própria (2024).

Em seguida, é necessário criar o arquivo *dockercompose.yml* para definir e gerar os múltiplos contêineres da aplicação, pois isto simplifica a tarefa complexa de coordenar a execução destes componentes, facilitando a replicação do ambiente de aplicação (DOCKER, 2024c). Seu código pode ser observado na Figura 41, que contém os quatro serviços a serem utilizados na aplicação.

O serviço *django*, é responsável por rodar o servidor da aplicação Django, ele

Figura 41 – Código do arquivo *docker-compose.yml* da aplicação *Web*.

```
version: "3.8"
services:
    django:
        build: .
        container_name: django
        command: python manage.py runserver 0.0.0.0:8000
        volumes:
            -.:/usr/src/app/
        ports:
            -"8000:8000"
        environment:
            -CELERY_BROKER=redis://redis:6379/0
            -CELERY_BACKEND=redis://redis:6379/0
        depends_on:
            -redis
    celery:
        build: .
        container_name: celery
        command: celery -A portal worker -l INFO
        volumes:
            -.:/usr/src/app
        environment:
            -CELERY_BROKER=redis://redis:6379/0
            -CELERY_BACKEND=redis://redis:6379/0
        depends_on:
            -django
            -redis
    celery-beat:
        build: .
        container_name: celery-beat
        command: celery -A portal beat -l INFO
        volumes:
            -.:/usr/src/app
        environment:
            -CELERY_BROKER=redis://redis:6379/0
            -CELERY_BACKEND=redis://redis:6379/0
        depends_on:
            -django
            -redis
    redis:
        image: "redis:alpine"
        container_name: redis
```

Fonte: Autoria própria (2024).

constrói a aplicação a partir do arquivo Dockerfile e utiliza um comando para iniciar o servidor de desenvolvimento Django, tornando-o acessível na porta 8000. Além disso, define variáveis de ambiente para que o Django saiba onde encontrar o *broker* e *backend* do Celery, ambos configurados para usar o Redis, indicando ainda que o contêiner Django depende do contêiner Redis para funcionar corretamente.

Logo após, o contêiner *celery*, responsável por executar as tarefas assíncronas da

aplicação, é iniciado. O comando *celery -A portal worker -l INFO* inicia uma instância *worker*, definindo pela *tag* -A a aplicação portal e pela *tag* -l o nível de *log* INFO. O worker monitora as filas de tarefas gerenciadas pelo Redis, utilizando-o como *broker* e *backend*, executando as tarefas conforme elas são atribuídas, funcionando como um sistema de processamento em segundo plano. Estas tarefas são enviadas pelo terceiro serviço, *celery-beat*, cujo trabalho é iniciado por meio do comando *celery -A portal beat -l INFO.*

Por fim, o serviço redis é criado como o núcleo da comunicação entre o Django e o Celery, atuando como *broker* de mensagens ao gerenciar as filas de tarefas e *backend* possibilitando o armazenamento dos resultados das tarefas processadas pelo Celery. O Redis utiliza uma imagem Alpine Linux, instalada assim que o arquivo *docker-compose.yml* é executado.

Desta forma, a aplicação está completa, podendo ser utilizada em contêineres Docker. O diretório contendo todos os arquivos desenvolvidos foi transferido à máquina Linux, para que fossem iniciados os testes de funcionamento e automações.

3.4 Considerações Finais

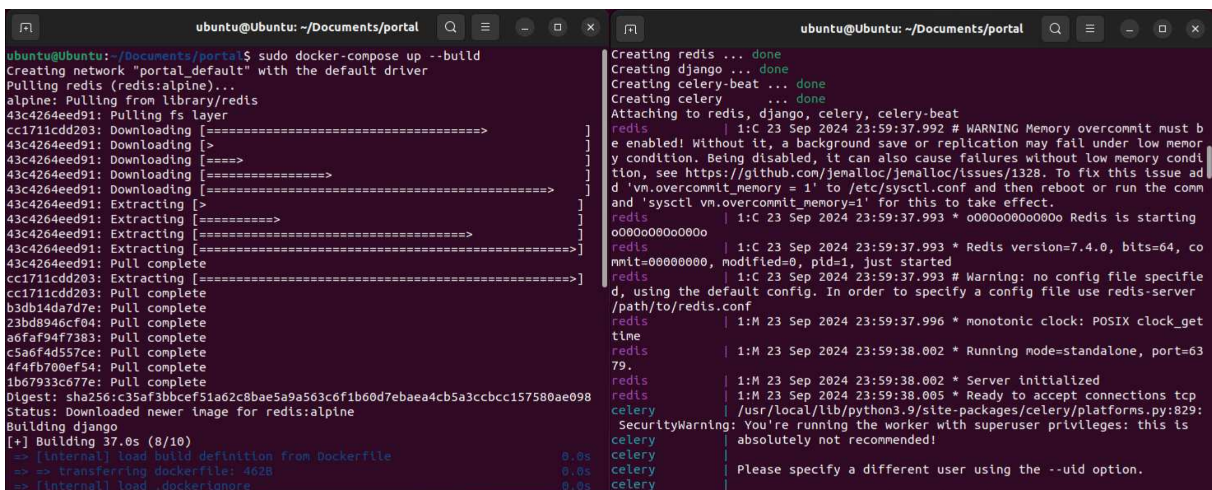
Este capítulo abordou a implementação da topologia de rede no GNS3, o código da aplicação Django e seu *deploy* utilizando contêineres Docker. Foram desenvolvidas as *views* principais: *discovery*, que realiza a descoberta dos dispositivos na rede; *realizarBackupEquipamentos*, responsável por fazer o *backup* das configurações dos equipamentos; *backupEquipamentos*, que permite o acesso aos *backups* obtidos e *verificarEquipamento*, que permite acessar e atualizar o status de um equipamento individualmente. Além disso, o Celery Beat foi integrado para a automação da tarefa periódica de monitoramento do status dos dispositivos, utilizando Redis como *broker* de mensagens.

O código desenvolvido foi portado para os contêineres Docker por meio do *Dockerfile* e *docker-compose.yml*, que foram configurados na máquina Ubuntu adicionada à topologia. No próximo capítulo, Resultados e Discussões, é analisada a eficácia dessas implementações, por meio de testes na aplicação, destacando os resultados obtidos no processo de automação e monitoramento da rede.

4 Resultados e Discussões

Utilizando a interface CLI do Ubuntu, o diretório contendo os arquivos da aplicação é acessado. Para montar os contêineres docker utiliza-se o comando *sudo docker-compose up –build*, iniciando assim a *engine* Docker. Por meio da Figura 42, observa-se a execução deste comando, assim como a inicialização dos contêineres. Como o modo *DEBUG* está ativo, é possível observar as requisições realizadas no portal através do CLI.

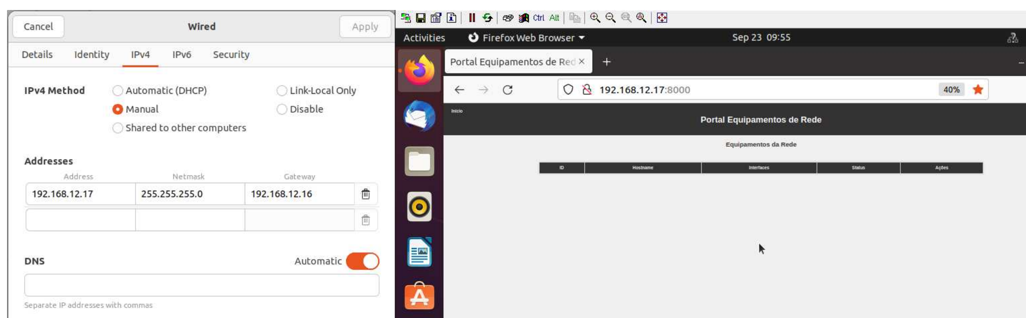
Figura 42 – Utilização do comando *sudo docker-compose up –build* no CLI Ubuntu.



Fonte: Autoria própria (2024).

Em um cenário real o usuário estaria conectado à topologia desenvolvida, acessando por meio de um computador, que será representado por meio do *UbuntuDesktopGuest*. Deste modo é necessário configurar a VM Ubuntu em que o servidor Docker encontra-se para o IP fixo 192.168.12.17 e utilizar a interface g0/0 do roteador R2 como seu *gateway* padrão. A Figura 43 apresenta esta configuração, assim como o acesso ao portal *Web* por meio do *UbuntuDesktopGuest*.

Figura 43 – Configuração do *gateway* padrão da máquina Ubuntu e acesso ao portal *Web* utilizando o *UbuntuDesktopGuest*.



Fonte: Autoria própria (2024).

4.1 Testes e resultados para a *view discovery*

Como visto na Figura 43, o portal *Web* encontra-se ativo e o acesso à sua página inicial gerou o *log* da Figura 44. Nele, pode-se observar a data e hora em que a requisição foi feita, que o método utilizado para obter dados foi o GET e que foi bem-sucedida, pois apresentou o código 200 (DOCS, 2022). Utilizando o botão que inicia a *view discovery*, são gerados os *logs* apresentados também na Figura 44. Estas mensagens são obtidas por meio dos comandos *print* utilizados no desenvolvimento do código e indicam que os equipamentos estão sendo acessados com sucesso. Nota-se também que os dispositivos acessados anteriormente estão sendo pulados, garantindo maior eficiência ao *script*.

Figura 44 – *Logs* de acesso à página inicial e gerados pela *view discovery*.

```
ubuntu@Ubuntu: ~/Documents/portal
//redis:6379/0
celery         | [2024-09-24 00:03:25,030: WARNING/MainProcess] /usr/local/lib/p
ython3.9/site-packages/celery/worker/consumer/consumer.py:508: CPendingDeprecati
onWarning: The broker_connection_retry configuration setting will no longer dete
rmine
celery         | whether broker connection retries are made during startup in Ce
lery 6.0 and above.
celery         | If you wish to retain the existing behavior for retrying connec
tions on startup,
celery         | you should set broker_connection_retry_on_startup to True.
celery         |   warnings.warn(
celery         |
celery         | [2024-09-24 00:03:25,081: INFO/MainProcess] mingle: searching f
or neighbors
django         | Watching for file changes with StatReloader
django         | INFO:django.utils.autoreload:Watching for file changes with Sta
tReloader
django         | Performing system checks...
django         |
django         | System check identified no issues (0 silenced).
django         | September 24, 2024 - 00:03:25
django         | Django version 4.2.16, using settings 'portal.settings'
django         | Starting development server at http://0.0.0.0:8000/
django         | Quit the server with CONTROL-C.
django         |
celery         | [2024-09-24 00:03:26,178: INFO/MainProcess] mingle: all alone
celery         | [2024-09-24 00:03:26,228: INFO/MainProcess] celery@5bbb40e7fa48
 ready.
django         | [24/Sep/2024 00:03:36] "GET / HTTP/1.1" 200 22134
```

```
ubuntu@Ubuntu: ~/Documents/portal
django         | [24/Sep/2024 00:03:36] "GET / HTTP/1.1" 200 22134
django         | INFO:paramiko.transport:Connected (version 2.0, client Cisco-1.
25)
django         | INFO:paramiko.transport:Authentication (password) successful!
django         | Conectando a 192.168.12.100
django         | Conectado a SW1# 192.168.12.100
django         | Conectando a 192.168.12.16
django         | Conectado a R2# 192.168.12.16
django         | Conectando a 192.168.12.17
django         | Conectado a R2# 192.168.12.17
django         | R2# já foi acessado anteriormente. Pulando para o próximo equip
amento.
django         | Conectando a 10.2.2.6
django         | Conectado a SW6# 10.2.2.6
django         | Conectando a 10.2.2.5
django         | Conectado a SW5# 10.2.2.5
django         | Conectando a 172.16.1.1
django         | Conectado a R1# 172.16.1.1
django         | Conectando a 172.16.2.3
django         | Conectado a R3# 172.16.2.3
django         | Conectando a 10.2.2.4
django         | Conectado a SW4# 10.2.2.4
django         | Conectando a 192.168.12.1
django         | Conectado a SW4# 192.168.12.1
django         | SW4# já foi acessado anteriormente. Pulando para o próximo equi
pamento.
django         | Conectando a 10.0.0.2
django         | Conectado a SW2# 10.0.0.2
django         | Conectando a 172.16.3.4
django         | Conectado a R4# 172.16.3.4
```

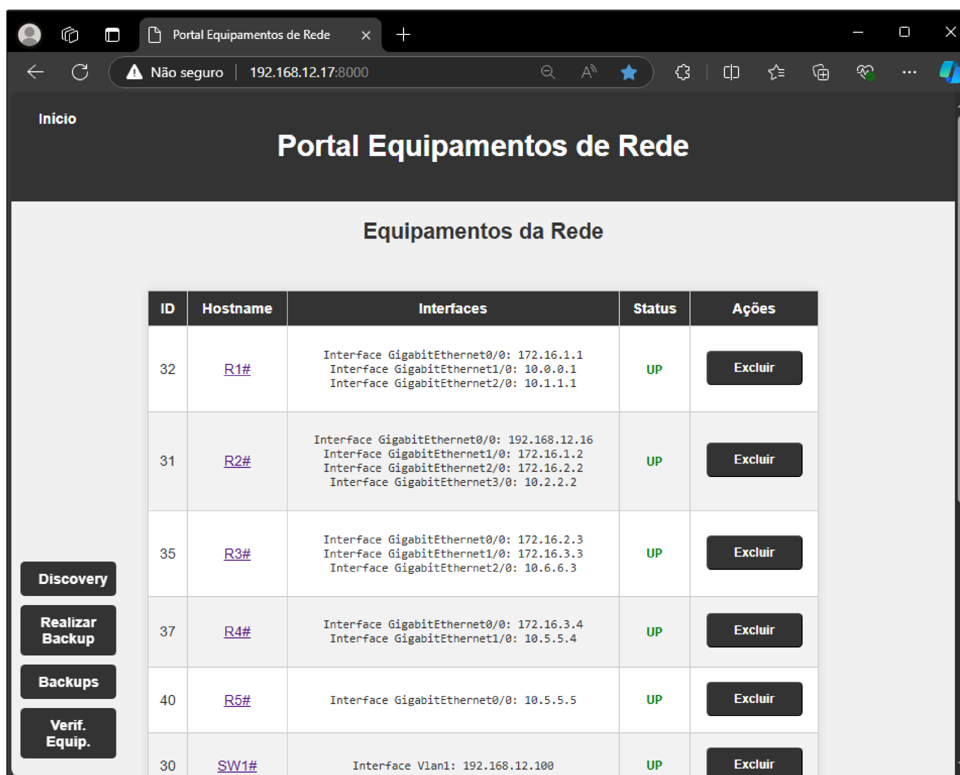
Fonte: Autoria própria (2024).

A Figura 45 mostra a página inicial após a *view discovery* ser finalizada e a Figura 46 os registros dos equipamentos realizados no banco de dados SQLite. É observado por meio da tabela criada que foi possível identificar todos os dispositivos da rede, assim como a configuração de suas interfaces, adicionando estes registros ao banco de dados e atualizando seu *status* para *UP*. Verifica-se assim a identificação e o mapeamento automatizado dos dispositivos conectados, que facilita a gestão e a manutenção da infraestrutura de rede. O acesso torna-se simples e pouco laborioso por meio da utilização de um único botão para realizar toda a descoberta da rede e facilitando a atualização de informações.

4.2 Testes e resultados para as *views realizarBackupEquipamentos* e *backupEquipamentos*

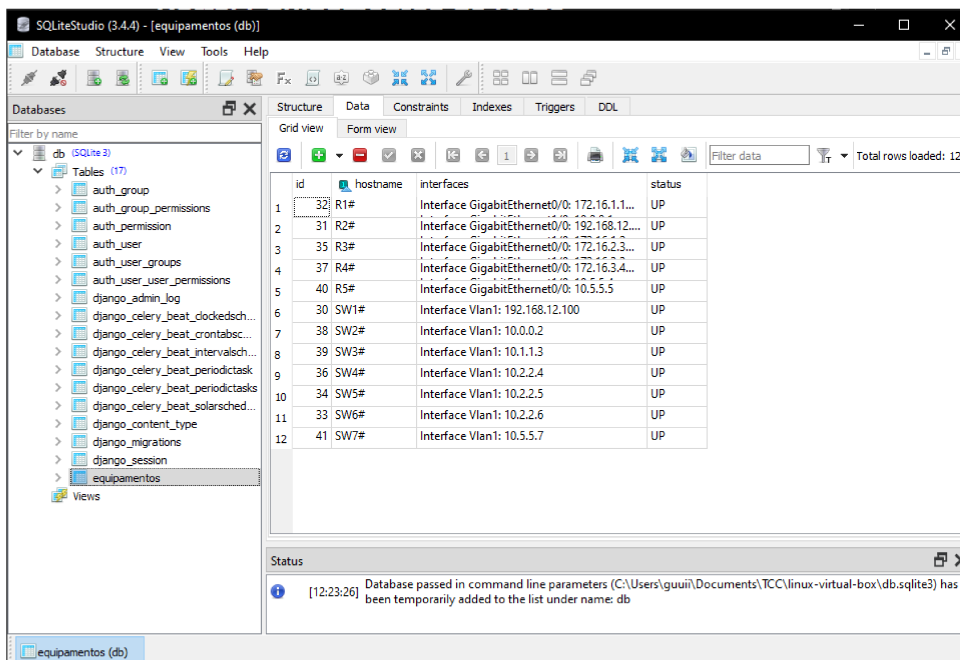
Ao utilizar o botão que realiza a requisição da *view realizarBackupEquipamentos*, os *logs* da Figura 47 serão gerados. Nota-se que o acesso aos equipamentos e aquisição dos arquivos texto contendo suas configurações foram realizadas com sucesso. Esta

Figura 45 – Página inicial do portal *Web* após a realização do processo de *discovery*.



Fonte: Autoria própria (2024).

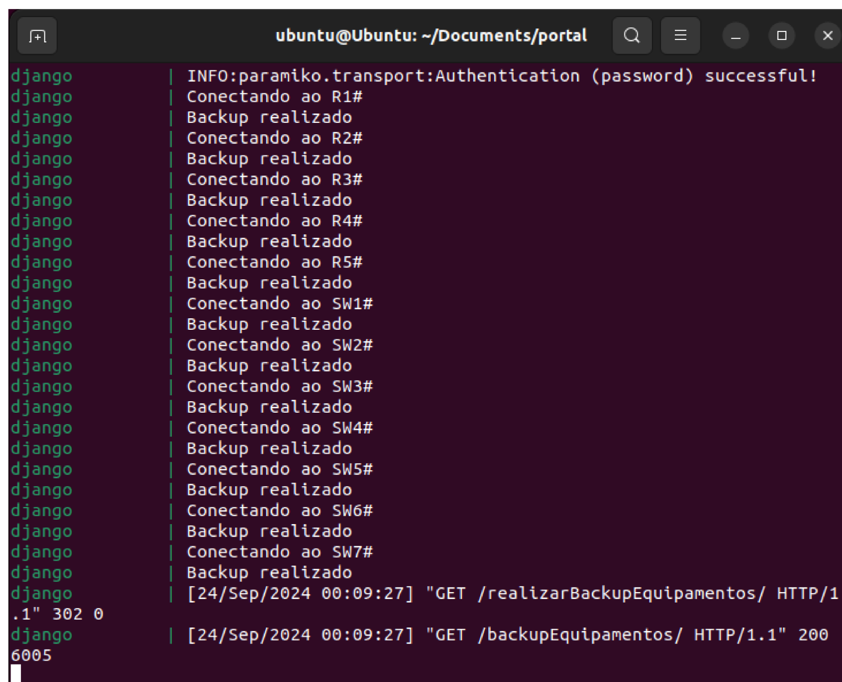
Figura 46 – Tabela equipamentos do banco de dados SQLite.



Fonte: Autoria própria (2024).

automação do *backup* é crucial para manter uma cópia de segurança atualizada das configurações de rede, possibilitando a rápida recuperação de um equipamento ao seu estado de funcionamento em caso de falhas, mudanças não planejadas ou erros de configuração, reduzindo assim o que seria um tempo de provisionamento prolongado. A Figura 48 mostra o redirecionamento para a *view backupEquipamentos*, apresentando os arquivos em uma tabela que contém a data em que foram gerados. É possível notar também o *download* de um dos arquivos gerados, verificando o sucesso do desenvolvimento.

Figura 47 – *Logs* gerados pela inicialização da *view realizarBackupEquipamentos*.



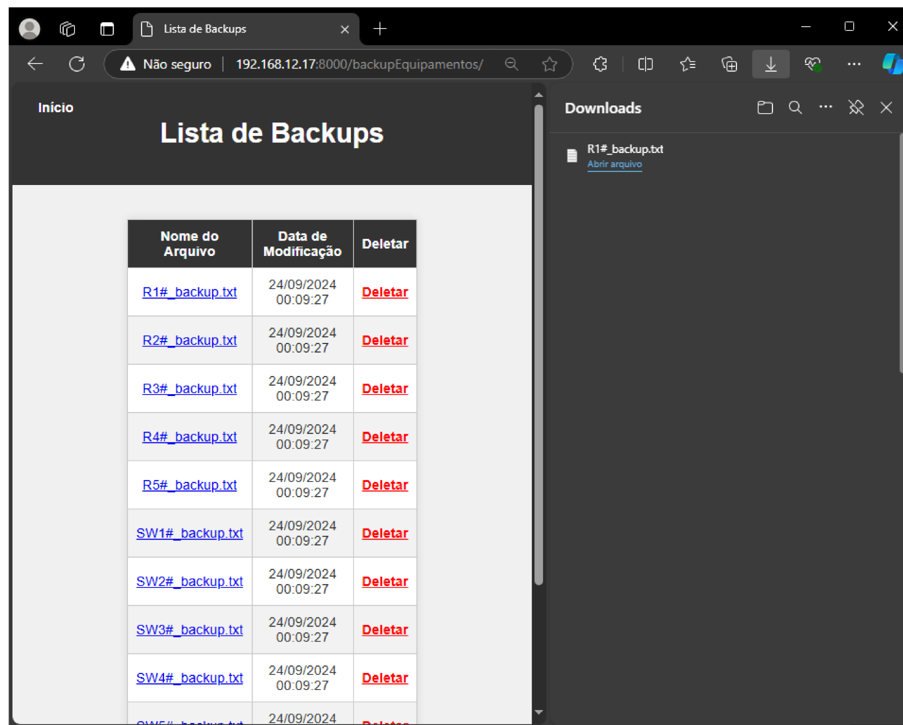
Fonte: Autoria própria (2024).

4.3 Testes e resultados da *view verificarEquipamento*

A fim de utilizar a *view verificarEquipamento* um novo roteador R6 foi adicionado à rede, como observado na Figura 49. Assim como os demais equipamentos, o roteador recebeu as configurações iniciais que garantem seu acesso: interface f0/0 configurada com endereço IP 10.6.6.6/24 e os protocolo OSPF e SSH foram habilitados. Foi configurado ao roteador R3, interface g2/0, o endereço IP 10.6.6.3/24 e OSPF para esta sub-rede, a fim de comunicar com o novo roteador R6. Por meio do botão que inicia a *view* e preenchimento do formulário com o IP da interface, os *logs* de acesso e coleta de dados foram gerados, este processo está apresentado na Figura 50.

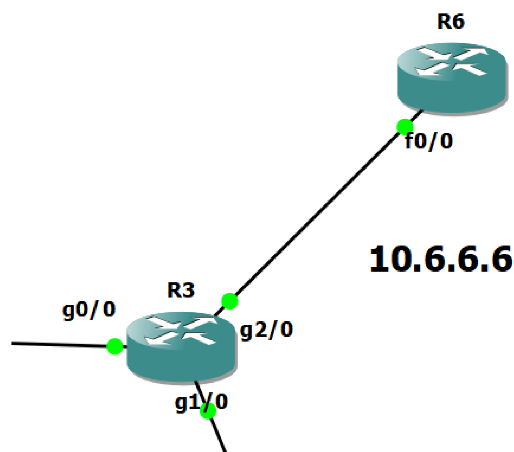
O *script* desenvolvido oferece uma abordagem mais direcionada, permitindo a interação com equipamentos de forma individualizada. Desta forma, torna-se ideal para cenários em que não é necessário coletar informações de todos os dispositivos, mas apenas

Figura 48 – Aba para a *view backupEquipamentos*, apresentando o *download* de um arquivo.



Fonte: Autoria própria (2024).

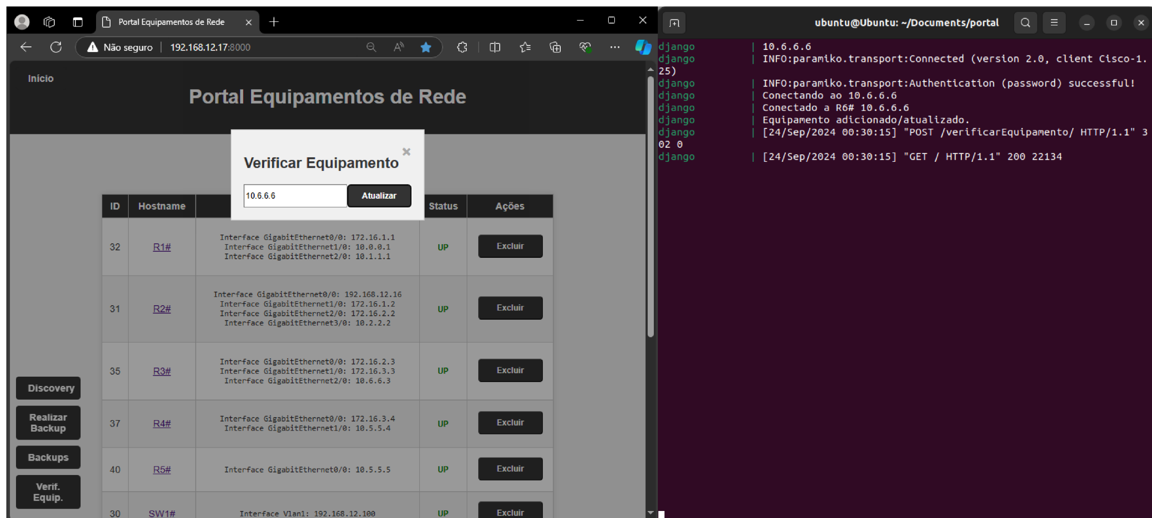
Figura 49 – Roteador R6 adicionado à topologia.



Fonte: Autoria própria (2024).

realizar a atualização ou adição de um equipamento específico. Essa abordagem otimiza o tempo de acesso e reduz a carga no sistema, garantindo maior eficiência ao tratar de ajustes pontuais na rede. Além disso, facilita a gestão de dispositivos, permitindo correções ou atualizações em tempo real sem impactar a operação global da infraestrutura.

Figura 50 – Utilização da *view verificarEquipamento* para coleta de informações do roteador R6 e *logs* gerados.

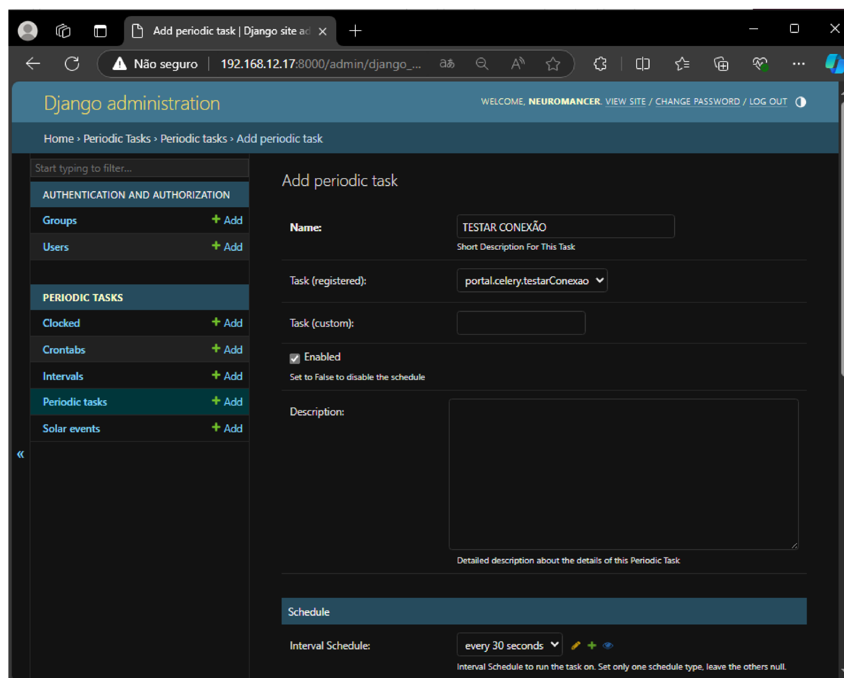


Fonte: Autoria própria (2024).

4.4 Testes e resultados da tarefa *testarConexao*

A Figura 51 mostra a criação da tarefa que utiliza o código desenvolvido *testarConexao*, utilizando a aba de administração do portal *Web*. A tarefa foi configurada para rodar a cada 30 segundos, garantindo uma atualização constante do *STATUS* dos equipamentos. Os *logs* apresentados na Figura 52 mostram que todos os equipamentos foram acessados por meio da saída apresentando sucesso ao *ping*.

Figura 51 – Criação da tarefa periódica *TESTAR CONEXÃO*.



Fonte: Autoria própria (2024).

Figura 52 – Início e fim dos *logs* gerados pela tarefa automatizada *testarConexao*.

Fonte: Autoria própria (2024).

Ao desligar o roteador R6, simulando assim um problema no dispositivo, os logs da Figura 53 são gerados quando o *script* tenta acessar este equipamento. O *log* mostra um erro do Netmiko, pois ele não conseguiu encontrar o carácter # na saída, indicando que o *prompt* não foi retornado no tempo limite de dois segundos. O *worker* então muda o *STATUS* para *DOWN*, apresentado na tabela da página inicial. O responsável pela manutenção da rede pode então identificar o problema e tomar as medidas necessárias para restabelecer a conectividade.

Figura 53 – Funcionamento da tarefa *testarConexao* ao não conseguir resposta do roteador R6.

Fonte: Autoria própria (2024).

Desta forma, a automação criada utilizando o Celery demonstra-se como função essencial no monitoramento contínuo e proativo da topologia, tornando-se útil para equipes de rede que precisam garantir que os dispositivos estejam operacionais, permitindo que a gestão de disponibilidade seja unificada, sem que nenhum comando seja utilizado manualmente.

5 Conclusão

A proposta deste trabalho foi desenvolver uma ferramenta *Web* para monitoramento e gerenciamento de equipamentos de rede em uma topologia, permitindo a criação de *scripts* de automação em Python. O emulador GNS3 mostrou-se a escolha ideal para simular uma topologia realista, utilizando imagens autênticas de *switches* e roteadores Cisco via IOU e IOS, respectivamente, além de facilitar a integração com VMs para a implementação e utilização da aplicação. A configuração dos equipamentos permitiu o acesso remoto à eles via SSH neste ambiente laboratorial, cenário semlhante ao que é encontrar num contexto real, onde uma VLAN de gerência, comum aos equipamentos, é utilizada.

Após a implementação da rede, o *framework* Django foi empregado no desenvolvimento da solução *Web*, permitindo um fluxo de elaboração de código intuitivo e altamente eficiente, ao utilizar o modelo MVT em sua arquitetura. Em suma, o *framework* permite ao usuário compor uma ferramenta utilizando Python, JavaScript, CSS e banco de dados SQLite de maneira simplificada, além de garantir ao usuário um servidor de desenvolvimento que auxilia na adequação e depuração do código antes de ser implementado em produção.

A integração entre as bibliotecas Python mostrou-se como peça fundamental para a execução do projeto. A utilização as bibliotecas de rede *Paramiko* e *Netmiko* para acesso, configuração e execução de comando nos equipamentos trabalhou em conjunto com módulos nativos e bibliotecas do Python. Como exemplo, têm-se a utilização de expressões regulares para a extração de endereços IP utilizando o módulo *re*, criação de diretório e manipulação de arquivos utilizando o módulo *os*, dentre outras aplicações. A união dessas ferramentas demonstrou a versatilidade e o poder do ecossistema Python no desenvolvimento de soluções complexas, garantindo que o projeto fosse não apenas funcional, mas também eficiente e extensível.

A virtualização da aplicação utilizando contêineres Docker trouxe agilidade e eficiência na implantação do código desenvolvido. A modularidade do Docker, que permite a execução isolada de múltiplos processos, garantiu a operação contínua do portal *Web* em tempo real, juntamente com as automações. A arquitetura flexível do Docker permite que a aplicação seja facilmente migrada para diferentes ambientes, facilitando o processo de desenvolvimento e implantação em diversas plataformas.

Por meio dos testes realizados, foi possível automatizar tarefas que tradicionalmente consomem muito tempo, como a coleta de informações e a realização de backups. O sistema automatizado economiza recursos e tempo de trabalho, garantindo respostas rápidas e

monitoramento constante da rede. Além disso, o uso do Celery Beat para a execução de tarefas periódicas de testes de conexão, mostrou-se eficiente na manutenção da integridade da infraestrutura de rede, permitindo que possíveis falhas sejam detectadas de forma rápida, proativa e automatizada. Através da interface do portal, é possível monitorar quase em tempo real o *status* dos dispositivos, um instrumento particularmente útil aos responsáveis pela manutenção da rede, que podem atuar de maneira ágil e precisa, intervindo apenas nos dispositivos que necessitam de atenção, otimizando o tempo de resposta a incidentes e garantindo a disponibilidade dos serviços de rede.

O ambiente desenvolvido e as ferramentas utilizadas garantem aos desenvolvedores a possibilidade de expandir a aplicação com novas automações, adaptando-a aos diferentes ambientes em que será utilizada. As possibilidades de aprimoramento são inúmeras, considerando a vasta quantidade de bibliotecas disponíveis e em constante desenvolvimento pela comunidade Python.

Dentre as possíveis melhorias, pode-se destacar: a integração do sistema com plataformas de comunicação, como e-mails ou APIs de mensagens, para fornecer atualizações constantes sobre o estado da rede; desenvolvimento de um sistema de *login* para acesso ao portal *Web*, aumentando a segurança do sistema; utilização o parâmetro *snmp_autodetect()* para identificar automaticamente o tipo de equipamento acessado, permitindo a automação para uma variedade ainda maior de dispositivos. Pode-se também considerar o uso de Inteligência Artificial (IA) para fornecer recomendações de ações ou até mesmo a solução de problemas de forma proativa, antecipando falhas na rede e sugerindo ajustes automáticos, o que ampliaria significativamente a eficiência e a autonomia do sistema.

Portanto, o projeto atingiu com sucesso sua proposta, proporcionando uma solução eficaz para a automação e monitoramento de redes, ao mesmo tempo em que oferece uma base sólida para futuras expansões e adequações. Através de sua implementação, a aplicação torna-se uma alternativa viável para reduzir o tempo gasto em tarefas repetitivas, mas essenciais, oferecendo suporte direto ao gerenciamento de redes em um ambiente unificado. A automação traz agilidade e eficiência, liberando recursos valiosos para que os responsáveis pela rede possam focar em problemas mais complexos e estratégicos.

Referências Bibliográficas

ANDARA, A.; WIDYARTO, S.; RUSDAH. Development of web-based network automation applications using the kano method and paramiko library to simplify the configuration of multivendor network devices at pt. digital vision nusantara. *International Journal of Science and Society*, v. 5, n. 5, Dec. 2023. Disponível em: <https://www.ijsoc.goacademica.com/index.php/ijsoc/article/view/965>. Citado na página 16.

AWS. *O que é virtualização? – Explicação sobre virtualização da computação em nuvem – AWS*. 2023. Disponível em: <https://aws.amazon.com/pt/what-is/virtualization/>. Citado na página 26.

AWS. *O que é Python?* 2024. Disponível em: <https://aws.amazon.com/pt/what-is/python/>. Citado na página 14.

BLACKWELL, M. *Dynamips*. 2014. Disponível em: <https://www.gns3.com/dynamips>. Citado na página 29.

BOETTIGER, C. An introduction to docker for reproducible research, with examples from the r environment. *ACM SIGOPS Operating Systems Review*, v. 49, n. 11, jan. 2015. ISSN 0163-5980. ArXiv:1410.0846 [cs]. Disponível em: <http://arxiv.org/abs/1410.0846>. Citado na página 27.

BUDIATI, H. et al. Implementation of k-means clustering method for network traffic anomaly detection. *Jurnal Mantik*, v. 6, n. 33, p. 3499–3504, nov. 2022. ISSN 2685-4236. Disponível em: <https://www.ejournal.iocscience.org/index.php/mantik/article/view/3218>. Citado na página 20.

BYERS, K. *Python for Network Engineers - Netmiko Library*. 2021. Disponível em: <https://pynet.twb-tech.com/blog/netmiko-python-library.html>. Citado na página 21.

BYERS, K. *ktbyers/netmiko*. 2024. Disponível em: <https://github.com/ktbyers/netmiko>. Citado na página 20.

BYERS, K. *Package netmiko*. 2024. Disponível em: <https://ktbyers.github.io/netmiko/docs/netmiko/index.html>. Citado 3 vezes nas páginas 38, 39 e 40.

CISCO. *Cisco IOS Configuration Fundamentals Command Reference*. 2010. Disponível em: <https://www.cisco.com/c/en/us/td/docs/ios/fundamentals/command/reference/cf_book/cf_s1.html>. Citado na página 44.

CISCO. *Cisco CPT Command Reference Guide CTC and Documentation Release 9.3 and Cisco IOS Release 15.1(01)SA*. 2017. Disponível em: <https://www.cisco.com/c/en/us/td/docs/optical/cpt/r9_3/command/reference/cpt93_cr/cpt93_cr_chapter_01110.html>. Citado na página 40.

CISCO. *VLAN Best Practices and Security Tips for Cisco Business Routers*. 2020. Disponível em: <https://www.cisco.com/c/en/us/support/docs/smb/routers/cisco-rv-series-small-business-routers/

1778-tz-VLAN-Best-Practices-and-Security-Tips-for-Cisco-Business-Routers.html>. Citado na página 32.

CISCO. *What Is Network Automation?* 2022. Disponível em: <https://www.cisco.com/c/en/us/solutions/automation/network-automation.html>. Citado na página 14.

CISCO. *Configure SSH on Routers and Switches*. 2023. Disponível em: <https://www.cisco.com/c/en/us/support/docs/security-vpn/secure-shell-ssh/4145-ssh.html>. Citado na página 32.

DJANGO. *QuerySet API reference*. 2024. Disponível em: <https://docs.djangoproject.com/en/5.1/ref/models/querysets/>. Citado 2 vezes nas páginas 40 e 43.

DJANGO. *Templates*. 2024. Disponível em: <https://docs.djangoproject.com/en/5.1/topics/templates/>. Citado na página 22.

DJANGO. *Why Django?* 2024. Disponível em: <https://www.djangoproject.com/start/overview/>. Citado na página 21.

DJANGO. *Writing views*. 2024. Disponível em: <https://docs.djangoproject.com/en/5.1/topics/http/views/>. Citado na página 22.

DOCKER. *Dockerfile reference*. 2024. Disponível em: <https://docs.docker.com/reference/dockerfile/#:~:text=A%20Dockerfile%20is%20a%20text,line%20to%20assemble%20an%20image.> Citado na página 51.

DOCKER. *What is a Container?* 2024. Disponível em: <https://www.docker.com/resources/what-container/>. Citado na página 27.

DOCKER. *Why use Compose?* 2024. Disponível em: <https://docs.docker.com/compose/intro/features-uses/#:~:text=Key%20benefits%20of%20Docker%20Compose,-Using%20Docker%20Compose&text=Simplified%20control%3A%20Docker%20Compose%20allows,and%20replicate%20your%20application%20environment.> Citado na página 52.

DOCS, M. W. *Códigos de status de respostas HTTP*. 2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>. Citado na página 56.

DOCUMENTATION, C. . *Celery - Distributed Task Queue*. 2023. Disponível em: <https://docs.celeryq.dev/en/stable/>. Citado na página 23.

DOCUMENTATION, C. . *Periodic Tasks*. 2023. Disponível em: <https://docs.celeryq.dev/en/stable/userguide/periodic-tasks.html>. Citado na página 23.

FORCIER, J. *Paramiko - A Python implementation of SSHv2*. 2024. Disponível em: <https://www.paramiko.org/>. Citado na página 20.

FORTINET. *Network Segmentation*. 2023. Disponível em: <https://www.fortinet.com/resources/cyberglossary/network-segmentation#:~:text=Network%20segmentation%20is%20an%20architecture,that%20flows%20into%20their%20systems.> Citado na página 29.

FRANCO, E. G.; KURITZKY, M.; LUKACS, R. *Global risks report 2022*. 2022. Disponível em: <https://www.preventionweb.net/publication/global-risks-report-2022>. Citado na página 15.

GEEKSFORGEEKS. *What is Docker?* 2020. Disponível em: <https://www.geeksforgeeks.org/introduction-to-docker/>. Citado na página 27.

GOLDMAN, L.; RAO, A.; KILLEEN, A. *Nokia: Operator benefits from the automation of IP networks*. 2021. Disponível em: <https://onestore.nokia.com/asset/210799>. Citado na página 15.

HUNT, C. *TCP/IP Network Administration*. [S.l.]: O'Reilly Media, Inc., 2002. ISBN 978-0-596-00297-8. Citado na página 19.

IBM. *O que são message brokers?* 2024. Disponível em: <https://www.ibm.com/br-pt/topics/message-brokers>. Citado na página 23.

JAYASEKARA, C. M. Franpycisco 2022: Network automation & abstraction solutions to simplify configuration complexity. *SSRN Electronic Journal*, 2022. Disponível em: <https://ssrn.com/abstract=4176096>. Citado na página 20.

JUNIPER. 2023. Disponível em: <https://www.juniper.net/us/en/research-topics/what-is-network-automation.html>. Citado na página 19.

KUBADE, O. *Network Automation using Python Programming*. 2019. Disponível em: <https://www.sevenmentor.com/network-automation-using-python-programming>. Citado na página 19.

KUROSE, J. F.; ROSS, K. W. *Redes de computadores e a Internet (coedição Bookman e Pearson)*. [S.l.]: Bookman Editora, 2021. ISBN 978-85-8260-559-2. Citado 2 vezes nas páginas 14 e 19.

MATHEUS, Y. *SSH, Telnet e as diferenças para conectar em um servidor*. 2018. Disponível em: <https://www.alura.com.br/artigos/entendendo-as-diferencas-entre-telnet-e-ssh?srsltid=AfmBOoqhLefJUImsaxAVSlmDO3t2i2cbIQhTBFfUJDGMDZtBk3lfYNjp>. Citado na página 24.

MAZIN, A. M. et al. Performance analysis on network automation interaction with network devices using python. In: *2021 IEEE 11th IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. [S.l.: s.n.], 2021. p. 360–366. Citado na página 20.

MIHăILă, P. et al. Network automation and abstraction using python programming methods. *MACRo 2015*, v. 2, 10 2017. Citado na página 20.

NETWORKS, J. *Guia de usuário do OSPF*. 2024. Disponível em: <https://www.juniper.net/documentation/br/pt/software/junos/ospf/topics/topic-map/ospf-overview.html#:~:text=O%20OSPF%20usa%20o%20algoritmo,bases%20de%20dados%20topol%C3%B3gicas%20individuais.> Citado na página 33.

PATEL, V. *Sending Email Using Django Celery*. 2022. Disponível em: <https://awstip.com/do-background-job-using-django-celery-5aae1b3e8a3a>. Citado na página 24.

PELCHEN, L. *Internet Usage Statistics In 2024*. 2024. Disponível em: <https://www.forbes.com/home-improvement/internet/internet-statistics/>. Citado na página 19.

PRASAD, N. *Redis as a Message Broker: Deep Dive*. 2024. Disponível em: <https://dev.to/nileshprasad137/redis-as-a-message-broker-deep-dive-3oek#:~:text=Task%20Execution%3A%20With%20Redis%20handling,queue%20for%20all%20these%20workers.> Citado na página 24.

REDHAT. *Virtualização*. 2018. Disponível em: <https://www.redhat.com/pt-br/topics/virtualization>. Citado na página 26.

REVIEW, T. *The Potential of Network Automation in Accelerating Business Growth in Asia - Telecom Review Asia Pacific*. 2024. Disponível em: <https://www.telecomreviewasia.com/news/featured-articles/4260-the-potential-of-network-automation-in-accelerating-business-growth-in-asia>. Citado na página 15.

ROCHIM, A. F. et al. As-rad system as a design model of the network automation configuration system based on the rest-api and django framework. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, v. 5, n. 4, Nov. 2020. Disponível em: <https://kinetik.umm.ac.id/index.php/kinetik/article/view/1093>. Citado na página 16.

SANTYADIPUTRA, G. S.; LISTARTHA, I. M. E.; SASKARA, G. A. J. The effectiveness of automatic network administration (ana) in network automation simulation at universitas pendidikan ganesha. *Journal of Physics: Conference Series*, IOP Publishing, v. 1810, n. 1, p. 012028, mar 2021. Disponível em: <https://dx.doi.org/10.1088/1742-6596/1810/1/012028>. Citado na página 16.

SILVA, D. A. e. *Como funciona a arquitetura MTV (Django)*. 2023. Disponível em: <https://diandrasilva.medium.com/como-funciona-a-arquitetura-mtv-django-86af916f1f63>. Citado na página 21.

SILVERMAN, R.; BARRETT, D. *Ssh, The Secure Shell. The Definitive Guide*. O'Reilly, 2001. ISBN 9786920000115. Disponível em: <https://books.google.com.br/books?id=2-uPtgAACAAJ>. Citado 2 vezes nas páginas 24 e 25.

SSH. 2023. Disponível em: <https://www.ssh.com/academy/ssh>. Citado na página 25.

STATISTA. *Network Infrastructure - Worldwide*. 2023. Disponível em: <https://www.statista.com/outlook/tmo/data-center/network-infrastructure/worldwide>. Citado na página 14.

STIGLER, S.; BURDACK, M. A Practical Approach of Different Programming Techniques to Implement a Real-time Application using Django. *ATHENS JOURNAL OF SCIENCES*, v. 7, n. 1, p. 43–66, March 2020. Citado na página 23.

TANENBAUM, A. S.; FEAMSTER, N.; WETHERALL, D. J. *Computer Networks*. [S.l.]: Pearson, 2022. (6TH EDITION). ISBN 9781292374062. Citado 4 vezes nas páginas 14, 19, 33 e 34.

URANO, L. *Django: o que é, para que serve e um Guia desse framework Python*. 2023. Disponível em: <https://www.alura.com.br/artigos/django-framework?srsltid=AfmBOopCvgpp-1Yq-Dl0KDW18mSQRKVeunf_K9-c6eS_cHMoLwHiLtrr>. Citado na página 21.

W3SCHOOLS. *Python Datetime*. 2024. Disponível em: <https://www.w3schools.com/python/python_datetime.asp>. Citado na página 46.

W3SCHOOLS. *Python os Module*. 2024. Disponível em: <https://www.w3schools.com/python/module_os.asp?ref=escape.tech>. Citado na página 44.

W3SCHOOLS. *Python RegEx*. 2024. Disponível em: <https://www.w3schools.com/python/python_regex.asp#:~:text=A%20RegEx%2C%20or%20Regular%20Expression,contains%20the%20specified%20search%20pattern.> Citado na página 41.

ZSCALER. *What Is Network Segmentation?* 2024. Disponível em: <https://www.zscaler.com/resources/security-terms-glossary/what-is-network-segmentation>. Citado na página 29.

APÊNDICE A – Script para a *view discovery*

O script apresentado é responsável pela criação da *view discovery*, que permite ao usuário automatizar a coleta de informações dos equipamentos presentes na topologia, acessando-os remotamente via SSH.

```
def discovery(request):
    device ={
        'host': '192.168.12.100',
        'device_type': 'cisco_ios',
        'username': 'auto',
        'password': 'auto',
    }
    try:
        def extract_ips(show_ip_output):
            ipPattern =r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
            ipAddresses =re.findall(ipPattern, show_ip_output)

            return ipAddresses

        def extract_ips_with_interfaces(hostname, show_ip_output):
            ipInterfacePattern =r'^(\S+)\s+([\d.]+)\s+\w+\s+\w+\s+(\w+)'
            matches =re.findall(ipInterfacePattern, show_ip_output, re.IGNORECASE | re.
                                                                  MULTILINE)

            ipInterfaceList =[]

            for match in matches:
                interface =match[0]
                ipAddress =match[1]
                status =match[2]
                if status.lower() =='up':
                    ipInterfaceList.append(f"Interface {interface}: {ipAddress}")

            info =ipInterfaceList
            return hostname, info

        def access_hosts(connection, listaIPs, co, listaHostsVisitados, visitedDevices,
                                                              infoEqs):
            novaListaIPs =set()
            for host in listaIPs:
```

```
        if host in listaHostsVisitados:
            continue
        device['host'] =host
        try:
            hostname =""
            print(f"Conectando a {host}")
            if co ==0:
                #connection = ConnectHandler(**device)
                hostname =connection.find_prompt()
            else:
                connection.write_channel(f'ssh -l auto {host}\n')
                time.sleep(2)
                connection.write_channel(f'auto\n')
                time.sleep(2)
                redispatch(connection, device_type='cisco_ios')
                hostname =connection.find_prompt()
            print(f"Conectado a {hostname} {host}")
            showIp =connection.send_command('show ip interface brief')
            for i in set(extract_ips(showIp)):
                listaHostsVisitados.add(i)
            listaHostsVisitados.add(host)
            infoEqs.append(extract_ips_with_interfaces(hostname, showIp))
            if hostname in visitedDevices:
                print(f"{hostname} foi acessado anteriormente. Pulando para o
                                                        proximo equipamento."
                                                        )
                continue
            hostname_obj, interfaces =extract_ips_with_interfaces(hostname,
                                                        showIp)
            equipamento, created =Equipamentos.objects.update_or_create(
                hostname=hostname_obj,
                defaults={
                    'interfaces': '\n'.join(interfaces),
                    'status': "UP",
                }
            )


            visitedDevices.append(hostname)
            showArp =connection.send_command('show arp')
            ips =extract_ips(showArp)
            for ip in ips:
                if ip not in listaHostsVisitados:
                    novaListaIPs.add(ip)

        except Exception as e:
            print(f"Falha...")
```

```
                listaHostsVisitados.add(host)
            if novaListaIPs:
                co=1
                access_hosts(connection, list(novaListaIPs), co, listaHostsVisitados,
                                                        visitedDevices, infoEqs)
            connection.disconnect()
            return visitedDevices
        co =0
        listaHostsVisitados =set()
        infoEqs =[]
        visitedDevices =[]
        hosts =['192.168.12.100']

        connection =ConnectHandler(**device)
        Equipamentos.objects.all().update(status="DOWN")
        access_hosts(connection, hosts, co, listaHostsVisitados, visitedDevices,
                                                infoEqs)
        print(sorted(visitedDevices))
        connection.disconnect()
        for i in infoEqs:
            print(i)
        return redirect('index')
    except:
        return redirect('index')
```

APÊNDICE B – Script para a *view* *realizarBackupEquipamentos*

O script apresentado é responsável pela criação da *view realizarBackupEquipamentos*, que permite ao usuário automatizar a coleta dos arquivos de configuração dos equipamentos presentes na topologia, acessando-os remotamente via SSH.

```
def realizarBackupEquipamentos(request):
    def extract_ips(show_ip_output):
        ipPattern =r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
        ipAddresses =re.findall(ipPattern, show_ip_output)
        return ipAddresses
    device ={
        'host': '192.168.12.100',
        'device_type': 'cisco_ios',
        'username': 'auto',
        'password': 'auto',
    }

    connection =ConnectHandler(**device)

    equipamentosUp =Equipamentos.objects.filter(status="UP").order_by('hostname')
    backups ={}
    for equipamento in equipamentosUp:
        equiInt =equipamento.hostname, equipamento.interfaces
        listaIntEq =extract_ips(equiInt[1])

        for host in listaIntEq:
            try:
                print(f'Conectando ao {equiInt[0]}')
                connection.write_channel(f'ssh -l auto {host}\n')
                time.sleep(2)
                connection.write_channel(f'auto\n')
                time.sleep(2)
                redispatch(connection, device_type='cisco_ios')
                configBackup =connection.send_command('show running-config')
                backups[equiInt[0]] =configBackup
                print('Backup realizado')
                break
            except:
                pass
```

```
backupDir =os.path.join(os.path.dirname(__file__), 'backups_equipamentos')
os.makedirs(backupDir, exist_ok=True)

for hostname, config in backups.items():
    file_path =os.path.join(backupDir, f'{hostname}_backup.txt')

    if os.path.exists(file_path):
        os.remove(file_path)

    with open(file_path, 'w') as backup_file:
        backup_file.write(config)

connection.disconnect()
return redirect('backupEquipamentos')
```

APÊNDICE C – Script para a *view* *backupEquipamentos*

O script apresentado é responsável pela criação da *view backupEquipamentos*, que permite ao usuário visualizar, baixar e excluir do diretório *backup_equipamentos* o arquivo de configuração dos equipamentos presentes na topologia.

```
def backupEquipamentos(request):
    backupFolder =os.path.join(os.path.dirname(__file__), 'backups_equipamentos')

    if 'delete' in request.GET:
        fileName =urllib.parse.unquote(request.GET['delete'])
        filePath =os.path.join(backupFolder, fileName)

        if os.path.exists(filePath):
            os.remove(filePath)
            return redirect('backupEquipamentos')
        else:
            raise Http404("Arquivo nao encontrado")

    if 'file' in request.GET:
        fileName =urllib.parse.unquote(request.GET['file'])
        filePath =os.path.join(backupFolder, fileName)

        if os.path.exists(filePath):
            with open(filePath, 'rb') as f:
                response =HttpResponse(f.read(), content_type="application/octet-stream"
                                                        )
                response['Content-Disposition'] =f'attachment; filename="{fileName}"'
                return response
        else:
            raise Http404("Arquivo nao encontrado")

    backups =[]
    for fileName in os.listdir(backupFolder):
        filePath =os.path.join(backupFolder, fileName)
        if os.path.isfile(filePath):
            timestamp =os.path.getmtime(filePath)
            formatted_time =datetime.fromtimestamp(timestamp).strftime('%d/%m/%Y %H:%M:%
                                                            S')
            backups.append((fileName, urllib.parse.quote(fileName), formatted_time))
```

```
    return render(request, 'lista_backups.html', {'backups': sorted(backups)})
```

APÊNDICE D – Script para a *view* *verificarEquipamento*

O script apresentado é responsável pela criação da *view verificarEquipamento*, que permite ao usuário acessar um equipamento remotamente via SSH, realizando a coleta de arquivos de configuração e atualização de informações no banco de dados.

```
def verificarEquipamento(request):
    def extract_ips_with_interfaces(hostname, show_ip_output):
            ipInterfacePattern =r'^(\S+)\s+([\d.]+)\s+\w+\s+\w+\s+(\w+)'
            matches =re.findall(ipInterfacePattern, show_ip_output, re.IGNORECASE | re.
                                                                    MULTILINE)

            ipInterfaceList =[]

            for match in matches:
                interface =match[0]
                ipAddress =match[1]
                status =match[2]
                if status.lower() =='up':
                    ipInterfaceList.append(f"Interface {interface}: {ipAddress}")

            info =ipInterfaceList
            return hostname, info

    device ={
        'host': '192.168.12.100',
        'device_type': 'cisco_ios',
        'username': 'auto',
        'password': 'auto',
    }
    if request.method =='POST':
        ip =request.POST.get('ip')
        try:
            connection =ConnectHandler(**device)
            print(f'Conectando ao {ip}')
            connection.write_channel(f'ssh -l auto {ip}\n')
            time.sleep(2)
            connection.write_channel(f'auto\n')
            time.sleep(2)
            redispatch(connection, device_type='cisco_ios')
            hostname =connection.find_prompt()
```

```
            print(f"Conectado a {hostname} {ip}")
            showIp =connection.send_command('show ip interface brief')

            hostname_obj, interfaces =extract_ips_with_interfaces(hostname, showIp)
            equipamento, created =Equipamentos.objects.update_or_create(
                hostname=hostname_obj,
                defaults={
                    'interfaces': '\n'.join(interfaces),
                    'status': "UP",
                }
            )
            configBackup =connection.send_command('show running-config')
            backup_dir =os.path.join(os.path.dirname(__file__), 'backups_equipamentos')
            os.makedirs(backup_dir, exist_ok=True)

            file_path =os.path.join(backup_dir, f'{hostname}_backup.txt')
            with open(file_path, 'w') as backup_file:
                backup_file.write(configBackup)
            connection.disconnect()
            print('Equipamento adicionado/atualizado.')
            return redirect('index')

        except Exception as e:
            print(f"Erro ao acessar o equipamento {ip}: {e}")
            return redirect('index')
```

APÊNDICE E – Script para a tarefa *testarConexao*

O script apresentado é responsável pela criação da tarefa *testarConexao*, que permite ao criar uma tarefa que realiza a verificação de conectividade para com os equipamentos, permitindo o agendamento desta automação via Celery.

```
@app.task(bind=True)
def testarConexao(self):
    logger.info('Inicio da tarefa testarConexao')
    def extract_ips(show_ip_output):
        ipPattern =r'\b(?:\d{1,3}\.){3}\d{1,3}\b'
        ipAddresses =re.findall(ipPattern, show_ip_output)
        return ipAddresses

    device ={
        'host': '192.168.12.100',
        'device_type': 'cisco_ios',
        'username': 'auto',
        'password': 'auto',
    }

    listaEq ={}
    model =apps.get_model(app_label='app_redes', model_name='Equipamentos')
    equipamentos =model.objects.all()
    for equipamento in equipamentos:
        listaEq[equipamento.hostname] =equipamento.interfaces

    listaHost =[]
    logger.info('Conectando ao dispositivo...')
    try:
        with ConnectHandler(**device) as connection:
            logger.info('Conexao estabelecida com sucesso.')
            for eq in listaEq.items():
                ip =extract_ips(eq[1])[0]
                try:
                    logger.info(f'Pingando IP {ip} do dispositivo {eq[0]}...')
                    pingTest =connection.send_command(f'ping {ip}', expect_string=r'#',
                                                    delay_factor=2)
                    logger.info(f'Resposta do ping: {pingTest}')

                    if '!' in pingTest:
```

```
                    listaHost.append(eq[0])
                    model.objects.filter(hostname=eq[0]).update(status='UP')
                    logger.info(f'{eq[0]} esta UP.')
                else:
                    model.objects.filter(hostname=eq[0]).update(status='DOWN')
                    logger.info(f'{eq[0]} esta DOWN.')
            except Exception as e:
                connection.write_channel("\036")
                logger.error(f'Erro ao pingar {ip}: {str(e)}')
                model.objects.filter(hostname=eq[0]).update(status='DOWN')

    connection.disconnect()
except Exception as e:
    logger.error(f'Erro ao conectar ao dispositivo: {str(e)}')
    model.objects.filter(hostname='SW1#').update(status='DOWN')

logger.info(f'Hosts ativos: {sorted(listaHost)}')
```