Documentação do Projeto: Requisições com Node.js e Python

Gabriel Maximino
gabriel.maximino@sga.pucminas.br
Ciência da Computação - Puc Minas

2 de julho de 2025

Conteúdo

1 Introdução e Contexto do Projeto

A gerência de redes em ambientes modernos frequentemente exige a execução de tarefas específicas em diferentes sistemas. A complexidade de realizar operações como verificações de conectividade (ping), inspeção de diretórios ou leitura de logs remotamente, de forma centralizada e eficiente, é um desafio comum.

Este projeto propõe uma arquitetura que integra as capacidades do **Node.js** e do **Python** para criar uma solução de gerência de redes distribuída. O **Node.js**, com seu ecossistema robusto para APIs e manipulação de requisições web, atua como a camada de orquestração. O **Python**, com sua vasta gama de bibliotecas para automação e interação com o sistema operacional, é utilizado para a execução efetiva dos scripts de gerência. Juntos, eles demonstram uma sinergia poderosa para o controle remoto de tarefas.

2 Arquitetura da Solução

A arquitetura implementada é baseada na comunicação **REST (Representational State Transfer)** via HTTP, permitindo que diferentes componentes de software interajam de forma padronizada.

2.1 Visão Geral do Fluxo de Comunicação

O processo de execução remota de um comando Python segue o seguinte fluxo:

1. Um **Cliente** (neste projeto, o **Postman** ou **Insomnia**) envia uma requisição `POST` para a aplicação **Node.js**. Esta requisição contém o código Python a ser executado no corpo, em formato JSON.

2. A aplicação **Node.js** recebe a requisição do cliente. Ela então atua como um intermediário, fazendo uma nova requisição `POST` para a aplicação **Flask** (Python). O código Python recebido é repassado para o Flask.

3. A aplicação **Flask** (escrita em Python) recebe o código Python. Ela salva temporariamente este código em um arquivo e o executa utilizando o módulo `subprocess` do Python. A saída (ou erros) da execução do comando é capturada.

4. O **Flask** retorna o resultado da execução (sucesso/erro, saída e mensagens de erro) para o **Node.js** em formato JSON.

5. O **Node.js** recebe a resposta do Flask e simplesmente a retransmite de volta para o **Cliente** original.

2.2 Papel do Node.js

O Node.js, utilizando o framework **Express**, atua como a interface de API primária do sistema. Seu papel é **receber as requisições externas** do cliente, **validar a estrutura básica** da requisição e **repassar o código** para a aplicação Flask. Ele também é responsável por **receber a resposta** do Flask e enviá-la de volta ao cliente, funcionando como um *proxy* ou orquestrador. A biblioteca **Axios** é empregada para realizar as requisições HTTP do Node.js para o Flask.

2.3 Papel do Flask (Python)

O Flask, um microframework web para Python, serve como o **executor de código**. Ele expõe um endpoint HTTP que recebe requisições `POST` contendo código Python. Sua principal função é **executar esse código de forma segura** (dentro dos limites do ambiente de execução), capturar todos os dados de saída (`stdout` e `stderr`) e retornar esses resultados. Isso permite que a lógica de negócio e as tarefas de gerência de redes sejam implementadas em Python, aproveitando sua vasta gama de bibliotecas.

3 Configuração do Ambiente e Pré-requisitos

Para replicar o ambiente de desenvolvimento e executar o projeto, siga os passos de instalação e configuração abaixo.

3.1 Instalação de Ferramentas Essenciais

1. **Node.js e npm:**
   - Acesse o site oficial: https://nodejs.org/en/download/
   - Baixe e instale a versão LTS (Long Term Support) recomendada para o seu sistema operacional. O **npm** (Node Package Manager) será instalado junto.
2. **Python e pip:**
   - Acesse o site oficial: https://www.python.org/downloads/
   - Baixe e instale a versão mais recente do Python 3. Certifique-se de **marcar a opção "Add Python to PATH"** durante a instalação no Windows. O **pip** (package installer for Python) será instalado automaticamente.

3.2 Configuração dos Ambientes de Projeto

3.2.1 Criação da Aplicação Flask (Python)

1. Crie uma pasta para o projeto, por exemplo, `seu_projeto_gerencia_redes`.
2. Dentro dela, crie uma subpasta para a aplicação Flask: `flask_app`.
3. Abra um terminal (**PowerShell** ou **CMD**) e navegue até a pasta `flask_app`.
4. **Crie um ambiente virtual:**

```
python -m venv venv

```

Listing 1: Criação do Ambiente Virtual

5. **Ative o ambiente virtual:**

```
.\venv\Scripts\activate

```

Listing 2: Ativação do Ambiente Virtual no PowerShell

```
venv\Scripts\activate

```

Listing 3: Ativação do Ambiente Virtual no CMD

6. **Instale o Flask:**

```
pip install Flask

```

Listing 4: Instalação do Flask

7. Crie o arquivo `app.py` na pasta `flask_app`.

8. **Inicie o servidor Flask:**

```
python app.py

```

Listing 5: Iniciando o Servidor Flask

3.2.2 Criação da Aplicação Node.js

1. Dentro da pasta `seu_projeto_gerencia_redes`, crie uma subpasta para a aplicação Node.js: `node_app`.

2. Abra um terminal (**PowerShell** ou **CMD**) e navegue até a pasta `node_app`.

3. **Inicialize o projeto Node.js:**

```
npm init -y

```

Listing 6: Inicializando o Projeto Node.js

4. **Instale as dependências (Express e Axios):**

```
npm install express axios

```

Listing 7: Instalação de Dependências Node.js

5. Crie o arquivo `server.js` na pasta `node_app`.

6. **Inicie o servidor Node.js:**

```
node server.js

```

Listing 8: Iniciando o Servidor Node.js

4 Implementação do Flask (Backend Python)

O coração da execução de código reside na aplicação Flask (`app.py`).

4.1 Código da Aplicação Flask (`app.py`)

```python
from flask import Flask, request, jsonify
import subprocess
import os

app = Flask(__name__)
UPLOAD_FOLDER = os.path.dirname(os.path.abspath(__file__))
    # Pasta atual (flask_app)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

@app.route('/execute_python_code', methods=['POST'])
def execute_python_code():
    data = request.get_json() # <--- Ponto Chave 1: Obtem
   o JSON da requisicao

    if not data or 'code' not in data:
        return jsonify({"error": "Nenhum codigo Python
   fornecido."}), 400

    python_code = data['code']
    temp_script_path = os.path.join(app.config['
   UPLOAD_FOLDER'], 'temp_script.py')

    try:
        with open(temp_script_path, "w", encoding='utf-8')
    as f: # <--- Ponto Chave 2: Salva o codigo em um
   arquivo temporario
            f.write(python_code)

        # <--- Ponto Chave 3: Executa o script temporario
   usando subprocess
        # text=True para decodificar stdout/stderr, shell=
   True para compatibilidade Windows
        # encoding/errors para lidar com caracteres
        result = subprocess.run(
            ["python", temp_script_path],
            capture_output=True,
            text=True,
            shell=False, # Geralmente False para maior
   seguranca, mas pode ser True dependendo do comando
            timeout=30, # Tempo limite para a execucao do
   script
            encoding='utf-8',
            errors='ignore'
        )

```

```
36          # <--- Ponto Chave 4: Captura e retorna a saida/
    erro
37          output = result.stdout
38          error = result.stderr
39          status = "success" if result.returncode == 0 else
    "error"
40
41          return jsonify({"status": status, "output": output
    , "error": error})
42
43      except subprocess.TimeoutExpired:
44          return jsonify({"status": "error", "output": "", "
    error": "O script excedeu o tempo limite de execucao."})
    , 500
45      except Exception as e:
46          return jsonify({"status": "error", "output": "", "
    error": f"Erro interno do servidor: {e}"}), 500
47      finally:
48          if os.path.exists(temp_script_path):
49              os.remove(temp_script_path) # <--- Ponto Chave
     5: Remove o arquivo temporario
50
51  if __name__ == '__main__':
52      app.run(debug=True, port=5000)
```

Listing 9: Código de app.py (Partes Chave)

4.2 Principais Pontos do Código Flask

- `request.get_json()`: Responsável por parsear o corpo da requisição HTTP (JSON) e extrair o código Python enviado.
- **Criação de Arquivo Temporário**: O código Python recebido é salvo em `temp_script.py` na pasta da aplicação Flask. Isso permite que o interpretador Python o execute como um script regular.
- `subprocess.run()`: É a função central para a execução do código Python. Ela invoca um novo processo Python que executa `temp_script.py`.
  - `capture_output=True`: Captura a saída padrão (`stdout`) e a saída de erro (`stderr`).
  - `text=True`: Decodifica `stdout` e `stderr` como texto.
  - `shell=False`: Embora tenhamos usado `shell=True` para comandos específicos como `ping` (devido a problemas de threading no Windows), para a execução de scripts Python diretamente, `shell=False` é geralmente mais seguro e preferível, pois evita a dependência do shell e possíveis injeções de comando.
  - `timeout`: Define um tempo limite para a execução do script, prevenindo que scripts maliciosos ou com erros travem o servidor.

- `encoding='utf-8'`, `errors='ignore'`: Crucial para garantir que a saída de comandos externos seja decodificada corretamente, evitando erros de codificação.

- **Remoção do Arquivo Temporário**: Garante a limpeza do sistema após a execução do script.

5 Implementação do Node.js (Frontend API)

A aplicação Node.js (`server.js`) atua como a interface de comunicação principal com o cliente.

5.1 Código da Aplicação Node.js (`server.js`)

```
const express = require('express');
const axios = require('axios'); // Importa a biblioteca Axios
const app = express();
const PORT = 3000;

app.use(express.json()); // <--- Ponto Chave 1: Middleware para parsear JSON no corpo da requisicao

// <--- Ponto Chave 2: Endpoint que o cliente externo ira chamar
app.post('/run_python_script', async (req, res) => {
    const pythonCode = req.body.code; // Obtem o codigo Python do corpo da requisicao

    if (!pythonCode) {
        return res.status(400).json({ error: 'Nenhum codigo Python fornecido.' });
    }

    try {
        // <--- Ponto Chave 3: Faz a requisicao POST para o Flask
        const flaskResponse = await axios.post('http://127.0.0.1:5000/execute_python_code', {
            code: pythonCode
        });

        // <--- Ponto Chave 4: Retorna a resposta do Flask para o cliente
        res.json(flaskResponse.data);
    } catch (error) {
        console.error('Erro ao comunicar com o Flask:', error.message);
        // Retorna o erro do Flask se disponivel, ou um erro generico
```

```
27          if (error.response) {
28              res.status(error.response.status).json(error.
    response.data);
29          } else {
30              res.status(500).json({ status: "error", output
    : "", error: "Erro interno ao processar a requisicao."
    });
31          }
32      }
33  });
34
35  app.listen(PORT, () => {
36      console.log(`Servidor Node.js rodando em http://
    localhost:${PORT}`);
37  });
```

Listing 10: Código de server.js (Partes Chave)

5.2 Principais Pontos do Código Node.js

- `app.use(express.json())`: Middleware que permite ao Express parsear automaticamente o corpo das requisições JSON. Essencial para receber o código Python do cliente.
- **Endpoint** `/run_python_script`: O caminho (*route*) que a aplicação Node.js expõe para o cliente externo fazer as requisições.
- `axios.post()`: A biblioteca Axios é utilizada para fazer a requisição HTTP `POST` do Node.js para o endpoint do Flask. Isso é onde a "ponte"entre as duas aplicações é estabelecida.
- **Tratamento de Respostas e Erros**: O Node.js captura a resposta do Flask (seja sucesso ou erro) e a repassa para o cliente original. Isso garante que o cliente tenha visibilidade do resultado da execução do código Python.

6 Desafios Superados e Aprendizados Chave

O desenvolvimento e teste desta solução apresentaram desafios notáveis, especialmente na interação com o ambiente Windows e na manipulação de dados entre diferentes tecnologias.

6.1 Desafios de Configuração Inicial no Windows

- **Problemas com o PATH de Variáveis de Ambiente**: Inicialmente, ferramentas como `npm` não eram reconhecidas no terminal, exigindo a verificação e, em alguns casos, a adição manual dos diretórios de instalação do Node.js e Python ao `PATH` do sistema.
- **Políticas de Execução do PowerShell**: O PowerShell, por padrão, restringe a execução de scripts, o que causou erros ao tentar ativar ambientes virtuais ou

executar scripts via `npm`. A solução envolveu ajustar a política de execução, geralmente para `RemoteSigned` com escopo `CurrentUser` (`Set-ExecutionPolicy RemoteSigned -Scope CurrentUser`).

6.2 A Complexidade do Escape de Caracteres em Terminais (CMD/PowerShell)

Um dos maiores obstáculos foi a necessidade de escapar corretamente caracteres como aspas (" e ') e barras invertidas (\) dentro da string JSON que continha o código Python.

- **CMD (`curl`)**: Exigiu um escape rigoroso de aspas duplas internas (\") e o uso de \n para quebras de linha. Em casos complexos, a estratégia de carregar o JSON de um arquivo (`curl -d @arquivo.json`) mostrou-se a mais confiável para evitar problemas de interpretação do shell.
- **PowerShell (`Invoke-RestMethod`)**: Apresentou desafios específicos na interpretação de strings passadas para o parâmetro `-Body`. Aspas simples internas a uma string delimitada por aspas simples no PowerShell precisaram ser escapadas com *duas aspas simples consecutivas* (''), e aspas duplas internas necessitaram de barra invertida (\"). A ambiguidade de certas sequências com nomes de parâmetros do PowerShell também foi uma fonte de erros.

A superação desses desafios de escape foi fundamental para o sucesso da comunicação, demonstrando a importância de entender como cada ambiente de terminal lida com strings literais.

6.3 Método `Invoke-RestMethod` no PowerShell

Para a comunicação via PowerShell, o cmdlet `Invoke-RestMethod` foi empregado. Este cmdlet é a ferramenta recomendada no PowerShell para interações com APIs REST, pois além de enviar requisições, ele tenta automaticamente parsear a resposta como JSON ou XML.

```
Invoke-RestMethod -Uri http://localhost:3000/
   run_python_script -Method POST -Headers @{"Content-Type"
   ="application/json"} -Body '{"code": "import subprocess\
   nimport platform\n\ntarget_ip = \"8.8.8.8\"\nparam = \"-
   n 1\" if platform.system().lower() == \"windows\" else
   \"-c 1\"\n\nif platform.system().lower() == \"windows
   \":\n    command = f\"ping {param} {target_ip}\"\n
   shell_mode = True\nelse:\n    command = [\"ping\", param
   , target_ip]\n    shell_mode = False\n\ntry:\n    result
    = subprocess.run(\n        command,\n
   capture_output=True,\n        text=True,\n        shell=
   shell_mode,\n        timeout=10,\n        encoding=\"utf
   -8\",\n        errors=\"ignore\"\n    )\n    \n    if
   result.returncode == 0:\n        print(f\"Ping para {
   target_ip} (SUCESSO):\\n{result.stdout}\")\n    else:\n
          print(f\"Ping para {target_ip} (FALHA - C digo
   de sa da: {result.returncode}):\\n{result.stderr}\\n{
```

```
result.stdout}\")\n\nexcept subprocess.TimeoutExpired:\n
    print(f\"Erro: O comando ping para {target_ip}
excedeu o tempo limite.\")\nexcept FileNotFoundError:\n
   print(f\"Erro: O comando ''ping'' n o foi encontrado
. Verifique se est  no PATH.\")\nexcept Exception as e
:\n    print(f\"Um erro inesperado ocorreu ao executar o
 ping: {e}\")"}'
```

Listing 11: Exemplo de Uso de Invoke-RestMethod (Comando Ping)

Apesar de ser uma ferramenta poderosa, sua sintaxe rigorosa para o parâmetro `-Body` exigiu atenção especial para o correto escape de todos os caracteres especiais e aspas aninhadas, como detalhado na seção 6.2.

6.4 Problemas de Codificação de Caracteres (`charmap`)

- **Contexto**: Erros do tipo `'charmap' codec can't encode character '\ufffd'` surgiram ao lidar com a saída de scripts Python que continham caracteres acentuados ou que interagiam com programas externos que usavam codificações diferentes. Isso ocorre quando o Python tenta converter uma string entre codificações incompatíveis, comum em ambientes Windows.

- **Solução**: A solução principal foi configurar a variável de ambiente `PYTHONUTF8=1` no terminal onde o Flask é executado, forçando o Python a usar UTF-8 para todas as operações de E/S. Adicionalmente, especificar `encoding='utf-8'` e `errors='ignore'` nas chamadas de `subprocess.run` garantiu que a saída dos processos externos fosse decodificada corretamente, prevenindo falhas na captura e exibição de resultados.

6.5 A Importância de Ferramentas Gráficas (Postman/Insomnia)

Diante dos desafios de escape e interpretação de comandos em terminais, a adoção de ferramentas gráficas como **Postman** ou **Insomnia** provou ser fundamental.

- **Abstração de Escapes**: Essas ferramentas abstraem completamente a necessidade de se preocupar com o escape de aspas e caracteres especiais na linha de comando. Ao selecionar o tipo de corpo como JSON e colar o conteúdo, a ferramenta se encarrega de formatar a requisição HTTP corretamente.

- **Depuração Visual**: Oferecem uma interface clara para configurar headers, body e visualizar as respostas da API, tornando o processo de depuração muito mais ágil e intuitivo do que analisar saídas de terminal.

- **Reutilização de Requisições**: Permitem salvar e organizar requisições, facilitando a reutilização de testes complexos sem a necessidade de reescrever comandos longos repetidamente.

O uso de Postman/Insomnia foi um divisor de águas, agilizando drasticamente a fase de testes e validação da comunicação entre os serviços.

7 Exemplos de Uso e Demonstração

Esta seção apresenta exemplos práticos de como a API pode ser utilizada para realizar tarefas de gerência de redes, demonstrando a capacidade de execução remota de código. Todos os exemplos devem ser enviados utilizando Postman ou Insomnia.

7.1 Comando Ping Simples

Demonstra a capacidade de verificar a conectividade de rede para um endereço IP específico.

```
import subprocess
import platform

target_ip = "8.8.8.8"
param = "-n 1" if platform.system().lower() == "windows"
    else "-c 1"

if platform.system().lower() == "windows":
    command = f"ping {param} {target_ip}"
    shell_mode = True
else:
    command = ["ping", param, target_ip]
    shell_mode = False

try:
    result = subprocess.run(
        command,
        capture_output=True,
        text=True,
        shell=shell_mode,
        timeout=10,
        encoding="utf-8",
        errors="ignore"
    )

    if result.returncode == 0:
        print(f"Ping para {target_ip} (SUCESSO):\\n{result
    .stdout}")
    else:
        print(f"Ping para {target_ip} (FALHA - Codigo de
    saida: {result.returncode}):\\n{result.stderr}\\n{result
    .stdout}")

except subprocess.TimeoutExpired:
    print(f"Erro: O comando ping para {target_ip} excedeu
    o tempo limite.")
except FileNotFoundError:
    print(f"Erro: O comando 'ping' nao foi encontrado.
    Verifique se esta no PATH.")
except Exception as e:
```

```
    print(f"Um erro inesperado ocorreu ao executar o ping:
    {e}")
```

Listing 12: Código Python para Ping

```
{
    "code": "import subprocess\\nimport platform\\n\\
   ntarget_ip = \\"8.8.8.8\\"\\nparam = \\"-n 1\\" if
   platform.system().lower() == \\"windows\\" else \\"-c
   1\\"\\n\\nif platform.system().lower() == \\"windows
   \\":\\n    command = f\\"ping {param} {target_ip}\\\\"\\n
       shell_mode = True\\nelse:\\n    command = [\\"ping
   \\", param, target_ip]\\n    shell_mode = False\\n\\ntry
   :\\n    result = subprocess.run(\\n        command,\\n
         capture_output=True,\\n        text=True,\\n
     shell=shell_mode,\\n        timeout=10,\\n
   encoding=\\"utf-8\\",\\n        errors=\\"ignore\\"\\n
     )\\n    \\n    if result.returncode == 0:\\n
   print(f\\"Ping para {target_ip} (SUCESSO):\\\\n{result.
   stdout}\\\\n\\")\\n    else:\\n        print(f\\"Ping
   para {target_ip} (FALHA - C digo de sa da: {result.
   returncode}):\\\\n{result.stderr}\\\\n{result.stdout
   }\\\\n\\")\\n\\nexcept subprocess.TimeoutExpired:\\n
   print(f\\"Erro: O comando ping para {target_ip} excedeu
   o tempo limite.\\")\\nexcept FileNotFoundError:\\n
   print(f\\"Erro: O comando \\'ping\\' n o foi encontrado
   . Verifique se est  no PATH.\\")\\nexcept Exception as
   e:\\n    print(f\\"Um erro inesperado ocorreu ao
   executar o ping: {e}\\\\n\\")"
}
```

Listing 13: JSON de Requisição para Ping

7.2 Listagem de Conteúdo de Diretório

Demonstra a capacidade de inspecionar remotamente o sistema de arquivos do servidor.

```
import os

print("Conte do do diret rio atual onde o script Python
    est  sendo executado:")
print(os.listdir("."))
```

Listing 14: Código Python para Listar Diretório

```
{
    "code": "import os\\n\\nprint(\\"Conte do do
   diret rio atual onde o script Python est  sendo
   executado:\\")\\nprint(os.listdir(\\".\\"))"
}
```

Listing 15: JSON de Requisição para Listar Diretório

7.3 Leitura de Arquivo de Log (`testelogs.txt`)

Simula a coleta de informações importantes de um arquivo de log ou configuração no servidor remoto. Lembre-se de criar o arquivo `testelogs.txt` na pasta `flask_app` com algum conteúdo.

```
try:
    with open("testelogs.txt", "r") as f:
        content = f.read()
        print("Conte do de testelogs.txt:\\n" + content)
except FileNotFoundError:
    print("Erro: testelogs.txt n o encontrado na pasta do
    aplicativo Flask.")
except Exception as e:
    print(f"Erro ao ler o arquivo testelogs.txt: {e}")
```

Listing 16: Código Python para Ler testelogs.txt

```
{
    "code": "try:\\n    with open(\\"testelogs.txt\\", \\"
   r\\") as f:\\n        content = f.read()\\n        print
   (\\"Conte do de testelogs.txt:\\\\n\\" + content)\\
   nexcept FileNotFoundError:\\n    print(\\"Erro:
   testelogs.txt n o encontrado na pasta do aplicativo
   Flask.\\")\\nexcept Exception as e:\\n    print(f\\"Erro
    ao ler o arquivo testelogs.txt: {e}\\")"
}
```

Listing 17: JSON de Requisição para Ler testelogs.txt

8 Considerações Finais

Este projeto demonstra com sucesso a viabilidade de uma arquitetura distribuída para gerência de redes, utilizando a força do Node.js na orquestração de APIs e a versatilidade do Python na execução de scripts de sistema. Os desafios enfrentados durante o desenvolvimento e a depuração, especialmente no que tange ao escape de caracteres e à interação entre processos em diferentes sistemas operacionais, foram cruciais para um aprendizado aprofundado sobre interoperabilidade de sistemas e robustez de aplicações.

8.1 Implicações de Segurança

É vital ressaltar que a capacidade de executar código arbitrário em um servidor remoto, embora poderosa para automação, apresenta riscos de segurança significativos. Em um ambiente de produção real, tal sistema exigiria robustas camadas de:

- **Autenticação**: Para verificar a identidade dos usuários que podem enviar comandos.
- **Autorização**: Para controlar quais comandos cada usuário ou sistema pode executar.

- **Validação de Entrada**: Para sanear e restringir estritamente o tipo de código que pode ser enviado, evitando injeções maliciosas.
- **Ambientes Isolados**: Execução de códigos em contêineres (Docker) ou máquinas virtuais leves para isolar falhas e ameaças.

8.2 Potenciais Melhorias

Para expandir o projeto, futuras melhorias poderiam incluir:

- Desenvolvimento de uma interface de usuário (front-end web) para interagir com a API Node.js.
- Implementação de mais comandos e scripts Python para cenários específicos de gerência de redes (e.g., configuração de interfaces, verificação de serviços).
- Adição de mecanismos de autenticação e autorização para proteger a API.
- Integração com sistemas de monitoramento para feedback em tempo real.