INSTITUTO FEDERAL
Norte de Minas Gerais

API Rest com Flask + WSGI

João Marcos Guimarães Coutinho

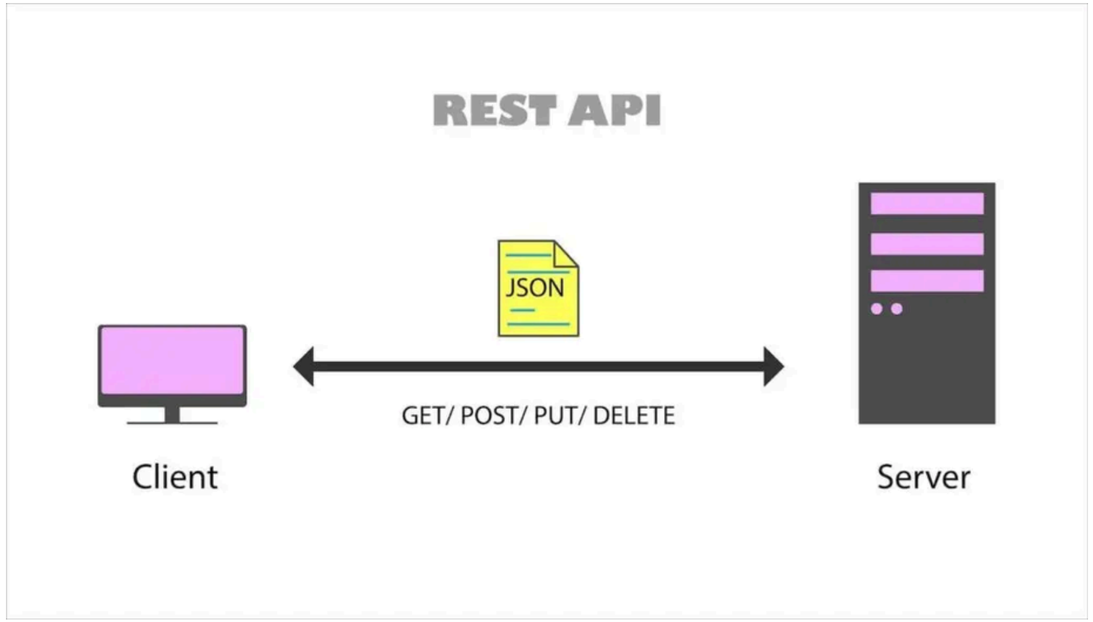
Sistemas Distribuídos

Instituto Federal do Norte de Minas Gerais - Campus Januária

REST

REST (Representational State Transfer) é um estilo arquitetural que define como sistemas distribuídos podem se comunicar de forma

- Simples
- Stateless
- Escalável
- Independente (desacoplado)
- Usando protocolos web (HTTP)



Padrão Arquitetural REST

REST (Representational State Transfer) não é um protocolo ou uma tecnologia, mas sim um estilo arquitetural (um padrão) para projetar aplicações distribuídas

O principal objetivo do REST é garantir que os sistemas sejam escaláveis, performáticos e modificáveis, usando os protocolos padrão da web, principalmente o HTTP.

Uma arquitetura deve aderir a seis restrições (princípios) para ser considerada “RESTful”

- Interface Uniforme (Uniform Interface)
- Cliente-Servidor (Client-Server)
- Stateless
- Cacheável (Cacheable)
- Sistema em Camadas (Layered System)
- Code on Demand (Optional)

Flask

Flask é um **microframework** escrito em Python, utilizado para desenvolver aplicações web e APIs.

app.py ●

app.py > ...

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'hello world!'

if __name__ == '__main__':
    app.run(debug=True)

```

app.run()

Segurança

Debugger Interativo: Se você deixar debug=True (o que é comum em desenvolvimento), qualquer pessoa que encontrar um erro no seu site verá um debugger interativo no navegador.

Execução Remota de Código: Esse debugger permite que o usuário execute qualquer código Python no seu servidor. Um invasor pode usá-lo para roubar seus dados, apagar seus arquivos ou tomar controle total da sua máquina.

Servidor Não "Blindado": Mesmo com debug=False, o servidor de desenvolvimento não é "blindado" (hardened). Ele não foi projetado para se defender contra ataques comuns da web, como ataques de Negação de Serviço (DDoS).

Performance

Multi-Threaded (dentro de 1 Processo); Limitado a 1 Núcleo (devido ao GIL); Se o processo trava, o site cai
O Python só permite que uma thread execute código Python por vez dentro desse processo.

WSGI Web Server Gateway Interface

WSGI é um **tradutor universal**

Servidores web (Nginx/Apache) só falam “HTTP”

Seu código (Flask/Django) só fala “Python”

Eles não se entendem sozinhos

Ponte: WSGI fica no meio, ele pega o pedido do servidor, traduz para Python para sua aplicação processar, e depois traduz a resposta do Python de volta para HTTP para o servidor entregar.

O WSGI (Web Server Gateway Interface), formalizado na PEP 3333, é uma especificação de interface que padroniza a comunicação entre servidores web (como Nginx, Apache) e aplicações web Python.

O objetivo técnico do WSGI é desacoplar o servidor da aplicação, resolvendo o problema de escalabilidade e compatibilidade.

Gunicorn Green Unicorn

O Gunicorn é um servidor de aplicação que implementa o lado do servidor da interface WSGI

Ele é baseado em um modelo chamado Pre-fork Worker Model (Modelo de Trabalhadores Pré-bifurcados). Isso significa que, quando você inicia o Gunicorn, ele cria um processo Master que, por sua vez, cria várias cópias de processos Workers prontos para receber requisições.

Por que ele é indispensável?

Concorrência (Paralelismo)
O Gunicorn contorna o GIL, criando múltiplos processos separados.

Robustez e Estabilidade
Se houver um erro fatal no seu código que faça o programa fechar, o app.run() derrubaria o site inteiro. Com o Gunicorn, se uma requisição específica causar um erro fatal, apenas aquele worker morre.

Gestão de Recursos
Você pode configurar o Gunicorn para matar workers que estão consumindo memória demais ou que demoram muito tempo para responder (timeout), evitando que um script malfeito trave todo o servidor.

Gunicorn - Gestão de Concorrência

Estrutura Hierárquica (Master & Workers)

Processo **Master**: O "Gerente". Não processa requisições web. Sua única função é manter o número correto de trabalhadores ativos.
Processos **Workers**: Os "Operários". São processos Python independentes (clonados via fork) que carregam a aplicação (Flask/Django) e executam a lógica.

Distribuição de Carga (O Socket Compartilhado)

O Master cria o Socket de Rede e o compartilha com todos os Workers.
O Master não atua como roteador.
O Sistema Operacional (Linux) recebe a requisição e escolhe qual Worker livre irá atendê-la (Load Balancing nativo do SO).

Superando Limitações

Bypass do GIL: Como cada Worker é um processo isolado com sua própria memória, o Python consegue usar Múltiplos Núcleos de CPU simultaneamente (Paralelismo Real).

Resiliência: Se uma requisição mata um Worker, o Master instantaneamente cria um novo (Hot Swap). Os outros usuários não são afetados.

Vantagens

Simplicidade e Facilidade de Configuração

É um servidor WSGI Python puro com uma configuração simples, tornando-o fácil de usar para iniciantes e para integração com plataformas de hospedagem.

Flexibilidade

Oferece vários tipos de workers, incluindo síncronos (padrão) e assíncronos (eventlet/gevent), permitindo ajustar o desempenho para diferentes necessidades de aplicação.

Estabilidade

O modelo Master/Worker é resiliente.
Recuperação Automática; Gestão de Vazamento de Memória; Contorno Eficiente do GIL
O Gunicorn resolve isso via força bruta: criando processos totalmente independentes.

Compatibilidade

O Gunicorn funciona com praticamente qualquer framework Python (Flask, Django, Pyramid, Bottle). Se segue o padrão WSGI, o Gunicorn roda. Você não precisa reescrever código para mudar de servidor.

Limitações

Consumo de Memória (RAM)

É um preço a se pagar pelo modelo de processos (Pre-fork). Como cada Worker é um processo isolado, ele precisa carregar sua aplicação inteira na memória RAM separadamente.

Vulnerabilidade a "Slow Clients" (Clientes Lentos)

O Gunicorn é otimizado para processar rápido e liberar o worker.

O Worker do Gunicorn ficará preso esperando esses dados chegarem bit a bit. Durante esse tempo, ele não atende mais ninguém. Isso torna o Gunicorn vulnerável a ataques de Negação de Serviço (DoS) do tipo Slowloris.

Workers Gunicorn

Sync Workers (Síncronos)

1 Processo = 1 Requisição.
Processa uma requisição do início ao fim antes de aceitar outra.
Máxima estabilidade e isolamento de erros.

Threaded Workers (gthread)

1 Processo gerencia um Pool de Threads.
Aceita múltiplas requisições por processo. Se uma thread bloqueia (esperando DB), outras continuam rodando.
Menor consumo de memória que o Sync; melhor performance para aplicações web comuns.

Async Workers (gevent)

Greenlets (Cooperative Multitasking).
Pausa a execução em esperas de I/O (rede/disco) para atender outros clientes instantaneamente.
Aguenta milhares de conexões simultâneas com pouquíssima CPU/RAM.

Gevent (greenlet)

A Solução para o "Custo da Espera" (I/O Bound)

Sem Gevent: Um worker ou thread fica bloqueado gastando memória RAM à toa enquanto espera.
Com Gevent: Durante a espera, o greenlet cede o lugar. Isso permite que um único processo atenda milhares de requisições que estão esperando.

Eficiência de Recursos

Threads do SO: Pesadas (~MBs de RAM cada). Limite de centenas/milhares.
Greenlets: Extremamente leves (~KBs de RAM cada). Você pode criar dezenas de milhares sem estourar a memória.

A "Mágica" da Compatibilidade (Sem reescrever código)

Esta é a importância prática: O Gevent permite transformar um código Flask síncrono (que foi escrito de forma simples, linha após linha) em um código assíncrono de alta performance sem precisar usar async/await e reescrever a aplicação toda. Isso é feito através do Monkey Patching, que substitui as bibliotecas padrão do Python em tempo de execução.

OBRIGADO

INSTITUTO FEDERAL
Norte de Minas Gerais