Web/Asynchronous Server Gateway Interface

Live de Python # 301

Não vamos implementar um servidor de aplicação e também não vamos criar um framework web.

Avisos!

O objetivo dessa live é entender a camada de SGI e as tecnologias envolvidas.

Avisos!

Roteiro

1. Desmembrando

Uma aplicação web

2. Servidor de aplicação

Pra que? Por que? Quando?

3. A camada de aplicação

Implementando as coisas

4. Usando de fato

A parte prática

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

55adriano, Albano Maywitz, Alexandre Costa, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Neto, Alynnefs, Alysson Oliveira, Andre Makoski, André Oliveira, Andre Paula, Antonio Filho, Apolo Ferreira, Aurelio Costa, Belisa Arnhold, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Canibasami, Carlos Gonçalves, Carlos Henrique, Carol Souza, Cauã Oliveira, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Silva, Darcio Alberico, Darcioalberico_sp, David Couto, David Frazao, Dh44s, Diego Guimarães, Dilan Nery, Dunossauro, Edgar, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fábio Belotto, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Ferrabras, Fichele Marias, Fightorfall, Francisco Aclima, Franklin Sousa, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Gfmaionese, Glauber Duma, Gnomo Nimp, Grinaode, Guilherme Felitti, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Herian Cavalcante, Hiago Couto, Idlelive, Iuri Kintschev, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, Joarez Wernke, Jonas Araujo, Jonatas Leon, Jonatas_silva11, Jose Barroso, Joseíto Júnior, José Predo), Josir Gomes, Jota_lugs, Jplay, Jrborba, Ju14x, Juan Felipe, Juli4xpy, Juliana Machado, Julio Cesar, Julio Franco, Julio Gazeta, Julio Silva, Kaio Peixoto, Knaka, Krisquee, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucasgcode, Lucas Mendes, Lucas Nascimento, Lucas Polo, Lucas Schneider, Luciano_ratamero, Luciano Ratamero, Lúcia Silva, Luidduarte, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Marina Passos, Mateusamorim96, Matheus Mendez, Matheus Vian, Medalutadorespacialx, Michael Santos, Mlevi Lsantos, Murilo Carvalho, Nhambu, Odenirgomesdev, Oopaze, Otávio Carneiro, Patrick Felipe, Programming, Pytonyc, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Ramon Lobo, Renan, Renan Sebastião, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Costa, Richard Sousa, Rinaldo Magalhaes, Robsonpiere, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Ronaldocostadev, Rui Jr, Rwallan, Samael Picoli, Samanta Cicilia, Santhiago Cristiano, Scrimf00x, Scrimfx, Shinodinho, Shirakawa, Tarcísio Miranda, Tenorio, Téo Calvo, Teomewhy, Thamires Betin, Tharles Andrade, Thiago, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tomás Tamantini, Valdir, Vinícius Areias, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vitoria Trindade, Vladimir Lemos, Vonrroker, Williamslews, Willian Lopes, Xxxxxxxxx, Xzerow22, Zero! Studio

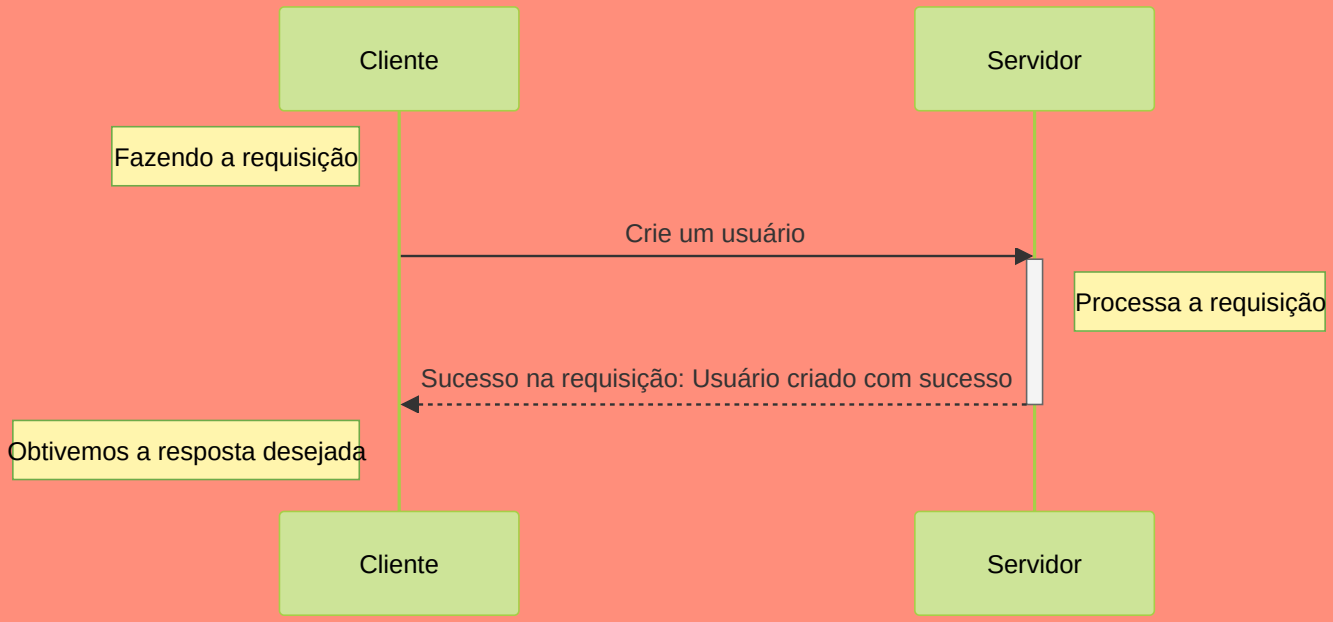
Obrigado você <3

Desmembrando os componentes

da WEB

A arquitetura base da web

O conceito fundamental das aplicações WEB, seja a arquitetura distribuída de **Cliente/Servidor**:



Onde o cliente e o servidor estão distribuídos na rede.

O protocolo base da web

Quando falamos de comunicação em rede, estamos sempre falando sobre o uso de diferentes protocolos em diferentes camadas.

Camadas	Protocolos	O que importa pra nós
Aplicação	HTTP, RTMP, ...	Programas se comunicam usando a rede
Transporte	TCP, UDP, ...	Dados são entregues entre aplicações, com ou sem confiabilidade
Rede	IP, ARP, ICMP...	Pacotes são endereçados e roteados entre redes
Acesso à Rede	Ethernet, ...	Dados são transmitidos fisicamente na rede

HTTP

HTTP (HyperText Transfer Protocol), é um protocolo simples de **solicitação-resposta** utilizado em toda a World Wide Web.

Ele especifica quais mensagens os **clientes** podem enviar aos **servidores** e quais respostas eles recebem. Esse modelo simples foi parcialmente responsável pelo sucesso inicial da Web, pois tornou o desenvolvimento e a implantação simples. [Tanenbaum, 2021]

São um **conjunto de restrições** como:

- **Métodos**: indicam a intenção da requisição (ex: GET, POST, DELETE)
- **Códigos de resposta**: indicam o resultado da requisição (ex: 200, 404, 422)
- **Cabeçalho**: carregam informações adicionais sobre a requisição ou resposta
- **Corpo**: transmite os dados propriamente ditos (opcional)

Versões do HTTP

O protocolo HTTP evoluiu ao longo do tempo para atender às novas demandas da Web:

Versão	Ano	Características principais
0.9	1991	Apenas texto plano, sem cabeçalhos; só suporta `GET`
1.0	1996	Introdução de cabeçalhos e outros métodos (`POST`, etc.)
1.1	1997	Conexões persistentes, chunked encoding, cache avançado
2	2015	Binário, multiplexação, compactação de cabeçalhos (`HPACK`)
3	2022	Baseado em QUIC (em vez de TCP), mais rápido e seguro

As versões mais modernas buscam **eficiência**, **baixa latência** e **segurança** para aplicações em tempo real.

Qual a cara do http?

Se formos olhar de forma superficial, a mensagem trocada se parece com isso:

```
POST /login HTTP/1.1
Host: api.exemplo.com
Content-Type: application/json
User-Agent: Mozilla/5.0

{
  "email": "joao@exemplo.com",
  "senha": 1234
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 85

{"access_token": "access_token", "token_type": "bearer"}
```

O servidor web

O servidor web é o componente responsável por receber as requisições HTTP enviadas pelos clientes, processá-las e retornar uma resposta adequada.

São as aplicações onde o *protocolo de aplicação* termina.

Algumas opções como:

- **Apache**: Um dos servidores mais antigos e mais utilizados
- **NGINX**: Famoso pela performance e por lidar com múltiplas conexões simultâneas
- **Caddy**: Um servidor moderno, com configuração automática de HTTPS
- **IIS**: Servidor da Microsoft, utilizado principalmente em ambientes Windows
- **Traefik**: Servidor e proxy reverso moderno, ideal para contêineres e microserviços

Responsabilidades do Servidor Web

O servidor web vai além de simplesmente servir conteúdo. Ele desempenha várias funções importantes para garantir o desempenho e segurança das aplicações web.

- **Controle de Cache**: pode armazenar conteúdos em cache para otimizar o desempenho, evitando que o mesmo recurso seja gerado ou recuperado repetidamente.
- **Proxy Reverso**: Atua como intermediário entre o cliente e a rede interna, redirecionando requisições para outros servidores ou serviços, melhorando segurança e escalabilidade.
- **Controle de Acesso**: Gerencia autenticação e autorização de usuários, garantindo que apenas os clientes permitidos tenham acesso a recursos específicos.
- **Rate Limiting**: O servidor pode limitar o número de requisições feitas por um cliente em um determinado período, prevenindo abusos e ataques, como DoS (Denial of Service).

Mas... Temos um problema...

O servidor web, embora seja responsável por receber e enviar requisições, não sabe lidar com a execução de linguagens de programação. Ele serve conteúdo estático como HTML, CSS e JS, mas quando se trata de lógica de aplicação dinâmica, ele precisa de ajuda.

Usamos o servidor web, além das funções listadas, como proxy reverso, encaminhando as requisições para um servidor de aplicação que processa a lógica (como PHP, Node.js, Python) e devolve a resposta ao cliente.

Servidor de Aplicação

Pra que? por que?

Servidores de aplicação

Os servidores de aplicação são componentes essenciais para executar a lógica de negócios de aplicações dinâmicas. Eles processam a parte interativa e dinâmica de uma aplicação, como consultas a bancos de dados, execução de código e interação com APIs.

- **Unicorn / Puma**: Servidores para aplicações **Ruby**
- **Gunicorn / Uvicorn**: Servidores para aplicações **Python**
- **TomCat**: Servidor para **Java** (principalmente para servlets e JSP)
- **ASP.NET Core**: Servidor para **C#**
- **Node.js**: Serve diretamente aplicações **JavaScript/Node.js**

Cada servidor de aplicação é projetado para trabalhar com uma linguagem específica ou um conjunto de frameworks, oferecendo o ambiente ideal para executar a lógica da aplicação.

Responsabilidades dos Servidores de Aplicação

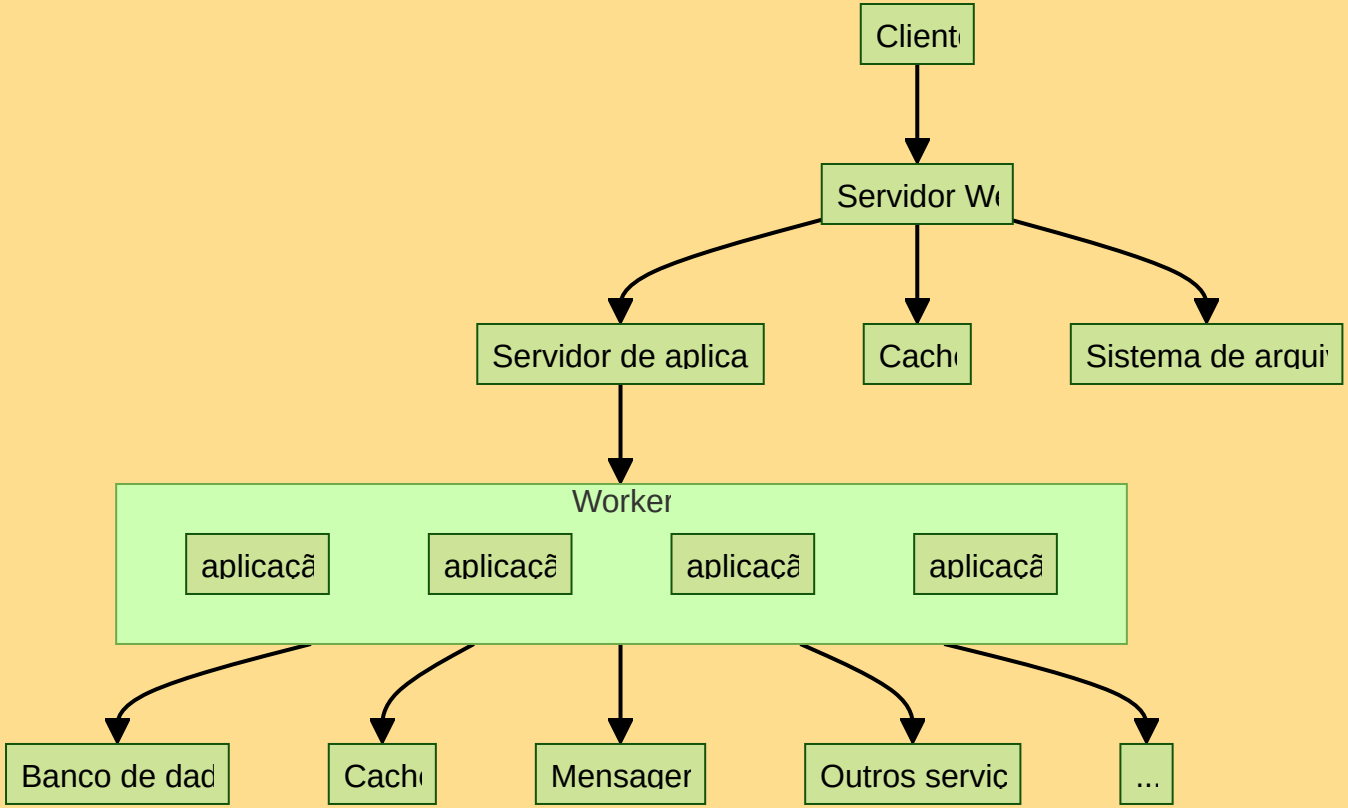
Os servidores de aplicação são responsáveis por:

- **Receber requisições do proxy**: Transformar requisições HTTP em objetos nativos da linguagem.
- **Gerenciar o ciclo de vida da aplicação**: Controlar inicialização, execução e encerramento.
- **Gerenciar workers**: Distribuir requisições entre múltiplos workers para escalabilidade.
- **Gerenciar conexões e sessões**: Controlar sessões de usuários e conexões com o banco de dados.

Essas responsabilidades garantem que a aplicação dinâmica funcione corretamente em um ambiente de produção.

Implicações práticas

O que nos dá algo parecido com isso na vida real de uma aplicação em produção:



Uma pergunta!

Mas por que existem linguagens que não usam servidores de aplicação?

Linguagens como Go e Rust não dependem de servidores de aplicação porque:

- **Execução direta**: Essas linguagens têm bibliotecas nativas (como o pacote net/http no Go e frameworks como Rocket no Rust) que permitem criar servidores web diretamente na aplicação.
- **Desempenho e simplicidade**: O código é compilado para máquinas específicas, tornando a execução muito eficiente. Isso elimina a necessidade de uma camada intermediária, como o servidor de aplicação.

Essas linguagens podem gerenciar o servidor HTTP internamente, sem precisar de servidores de aplicação como intermediários.

Falando especificamente sobre python

Bom... Estamos na live de Python, afinal...

Uma questão que precisava ser resolvida é como criar essa "meiota" entre qualquer proxy reverso e qualquer aplicação python.

Nesse caso, nasceu o **WSGI (Web Server Gateway Interface)** [uísgui], proposto em 2003 pela PEP 333 e expandido em 2010 pela PEP 3333

As PEPs descrevem um padrão simples e comum para que servidores web possam se comunicar com aplicações Python. Ele define como o servidor de aplicação e o servidor web devem interagir, criando uma interface clara entre os dois.

Junto disso, foi introduzido na biblioteca padrão o `wsgiref`. Uma implementação de referência.

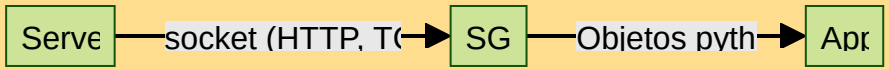
Gateway?

A palavra *gateway* pode ser traduzida como "porta de entrada" ou "ponto de ligação". Em redes de telecomunicação, um gateway é um dispositivo físico ou lógico que conecta duas redes diferentes, permitindo a comunicação entre elas mesmo que usem protocolos distintos.

- **Fisicamente**, a função de gateway pode ser desempenhada por um `roteador`, especialmente em redes domésticas, onde ele conecta a rede local à internet.
- **Logicamente**, essa função pode ser implementada por sistemas como `firewalls` ou `proxies`, que filtram ou redirecionam o tráfego entre redes.

Gateway?

Com esse conceito em mente, no contexto do **WSGI**, o "gateway" atua como um componente que converte a "rede" (HTTP, TCP, sockets etc.) em objetos Python que são passados para a aplicação. Ele faz a ponte entre o servidor web e a lógica da aplicação, permitindo que você escreva código Python independente dos detalhes de rede ou protocolo.



WSGI

A proposta do WSGI é garantir:

- **Desacoplamento**: O servidor web não precisa saber nada sobre a lógica da aplicação, e a aplicação não precisa saber como lidar com os detalhes do servidor web.
- **Interoperabilidade**: Qualquer servidor web WSGI compatível pode trabalhar com qualquer aplicação Python que siga o padrão WSGI, proporcionando flexibilidade e portabilidade.
- **Simplicidade e desempenho**: Embora simples, o WSGI garante que o ciclo de requisição/resposta seja eficaz, sem sobrecarga.

Em resumo, o WSGI resolve o problema de como o servidor web e a aplicação Python podem se comunicar de forma eficiente e padronizada.

A aplicação [ref.py]

Basicamente, com essa especificação, bibliotecas, frameworks e qualquer código podem ser servidos por um servidor de aplicação de forma simples:

```
def app(environ: dict, start_response: callable):
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return [b"Hello world!\n"]
```

- `environ`: Um `dict` contendo as informações da requisição
- `start_respose`: Uma função fornecida pelo servidor de aplicação que recebe os:
  - `status_code`
  - `headers`

O retorno da função `app` deve ser um iterável de `bytes`

Usando o servidor de referência [ref.py]

Com algumas poucas linhas, temos nossa aplicação rodando:

```
from wsgiref.simple_server import make_server

with make_server('', 8000, app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

Por via disso, os dados serão passados para o `environ`:

- https://wsgi.readthedocs.io/en/latest/definitions.html#standard-environ-keys
- https://wsgi.readthedocs.io/en/latest/definitions.html#wsgi-environ-keys

Rodando [ref.py]

Vamos entender o que acontece na execução:

```
from pprint import pp


def app(environ: dict, start_response: callable):
    environ_data = {
        var: environ[var]
        for var in environ
        if var.startswith('HTTP') or var.startswith('wsgi')
    }
    pp(environ_data)

    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return [b"Hello world!\n"]
```

Vamos aos fontes...

- flask
- django
- bottle
- ...

É possível notar que todos os frameworks aplicam o padrão.

ASGI, o "novo" padrão

O tempo foi passando, o asyncio chegou (2012~2015)... E agora? Houve uma evolução natural na especificação. Tanto pensando na evolução da linguagem quanto nas versões mais recentes do HTTP.

Assim nasce o **ASGI (Asynchronous Server Gateway Interface)** [asgui] por volta de 2015*.

A principal motivação de uma nova especificação se dá por conta do WSGI ser *single-callable model*, o que faz com que uma chamada seja feita e somente uma resposta seja dada.

Impossibilitando a utilização de chamadas persistentes. Como WebSockets, já suportados no HTTP/1.1 (1997).

---

*: Não consigo precisar a data, pois essa especificação não foi criada em uma PEP.

A especificação

A ideia aqui é bastante parecida com o WSGI, porém, async:

```
async def app(scope, receive, send):
    await send({
        "type": "http.response.start",
        "status": 200,
        "headers": [(b"Content-Type", b"text/plain")],
    })
    await send({
        "type": "http.response.body",
        "body": b"Ola mundo :)",
    })
```

- `scope`: um dicionário que contém informações sobre a solicitação recebida
- `send`: uma função assíncrona usada para enviar mensagens
- `receive`: uma função assíncrona usada para receber mensagens

(multiplexação)

Vendo em ação [aref.py]

Infelizmente, o **`asgiref`** não fornece uma implementação de servidor, então vamos usar o uvicorn para fazer isso:

```
# /// script
# dependencies = ["uvicorn"]
# ///
import uvicorn
uvicorn.run(app)
```

Dessa forma, podemos ver os dados passando e o contexto sendo exposto para as requisições.

Os servidores de aplicação

O python tem diversos servidores de aplicação.

Uma das formas de olhar pra eles é tentando entender quais especificações eles implementam e também qual versão do protocolo HTTP eles suportam:

Server	WSGI	ASGI	HTTP/1	HTTP/2	HTTP/3
Gunicorn	✅	❌	✅	❌	❌
Uvicorn	❌	✅	✅	❌	❌
Hypercorn	❌	✅	✅	✅	⚠
Draphne	❌	✅	✅	❌	❌
Granian	✅	✅	✅	✅	❌

Para explicar alguns conceitos, vou usar alguns deles:

```
pip install gunicorn uvicorn granian
```

Forma de uso

É bastante simples usá-los pela CLI:

```
gunicorn nome_do_arquivo:wsgi_app
# ou
uvicorn nome_do_arquivo:asgi_app
```

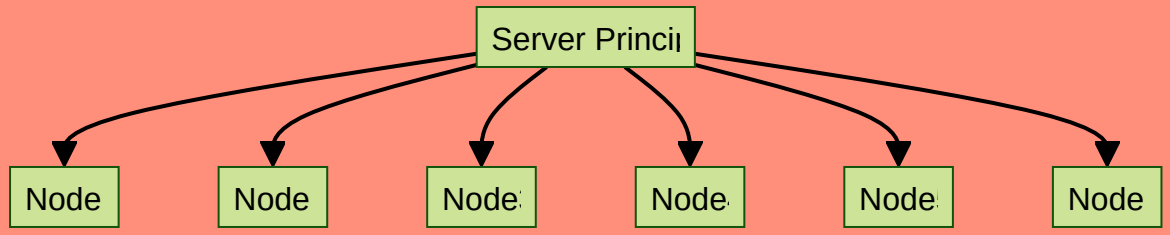
Dessa forma bastante simples, o servidor de aplicação inicia a nossa aplicação.

Workers

Uma característica comum entre os servidores de aplicação é a capacidade de trabalhar com workers, que possibilita executar diversas requisições em paralelo.

```
server nome_do_arquivo:sgi_app --workers 6
```

Cada worker é um processo separado, o que permite ao servidor processar várias requisições simultaneamente.



Quando você utiliza múltiplos workers, o servidor principal distribui as requisições entre os workers, cada um processando as requisições de forma independente.

Vantagens do formato

Só para recapitular e também ficar marcado:

- Independência e Padronização Servidor / App
  - Qualquer servidor web pode ser usado
  - Qualquer servidor de aplicação pode ser usado
  - Qualquer framework pode ser usado
- Escalabilidade e Concorrência: Com a padronização de interfaces, é fácil rodar múltiplos workers para distribuir o tráfego.
- Deploy: Você não precisa se preocupar em configurar servidores diferentes para diferentes frameworks ou plataformas.

Um middleware é apenas "algo" que **envolve** a aplicação e a modifica ou estende sem alterar seu funcionamento interno.

Um exemplo [aref_mid.py]

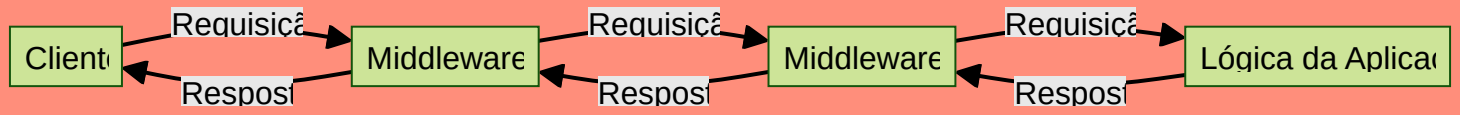
Um middleware bastante simples (e útil) é o `timing-asgi`. Ele mede o tempo de execução da aplicação e imprime ou envia para algum sistema de métricas:

```
# pip install timing-asgi
from timing_asgi import TimingMiddleware, TimingClient

class PrintTimings(TimingClient):
    def timing(self, metric_name, timing, tags):
        print(metric_name, timing, tags)

app = TimingMiddleware(asgi_app, PrintTimings())
```

Agora, toda requisição que chega na aplicação tem seu tempo monitorado.

O Ciclo de Vida da Requisição

Em uma aplicação **WSGI/ASGI**, o ciclo de vida de uma requisição passa por diversas etapas, envolvendo middlewares e a lógica da aplicação. O fluxo é o seguinte:



- **Cadeia de responsabilidades**: Cada componente tem a chance de processar ou modificar a requisição e a resposta.
- **Interceptação de requisições**: Um middleware pode interromper o fluxo
- **Ponto único de entrada e saída**: Toda a requisição entra por essa cadeia e toda resposta sai por ela.
- **Composição e reutilização**: Middlewares facilitam a adição de comportamentos transversais (como logging, métricas, compressão etc.).

Testes em Aplicações SGI [aref_test.py]

Graças à padronização da interface, testar aplicações SGI é simples e direto.

```python
# pip install httpx pytest pytest-asyncio
from httpx import AsyncClient, ASGITransport
import pytest

@pytest.mark.asyncio
async def test_homepage():
    async with AsyncClient(
        transport=ASGITransport(app=app), base_url="http://testserver"
    ) as client:
        response = await client.get('/')

    assert response.status_code == 200
    assert 'ola' in response.text
```

Para WSGI, somente importar `WSGITransport`.

Os frameworks

Se olhamos para a padronização em relação aos frameworks, eles se adaptam aos padrões:

Framework	Interface
Bottle	WSGI
Django	WSGI / ASGI
Flask	WSGI / ASGI
Tornado	WSGI / ASGI
FastAPI	ASGI
Starlette	ASGI
Quart	ASGI
BlackSheep	ASGI

- - -

E caso você queira adaptar um framework WSGI para ASGI (ou o inverso), você poderia usar um middleware para isso. Como o a2wsgi.

No fim...

Entender a separação entre servidor web, servidor de aplicação e aplicação é essencial para desenvolver, escalar e manter aplicações Python modernas.

A existência de padrões como WSGI e ASGI permite que o ecossistema Python evolua de forma interoperável, desacoplada e escalável.

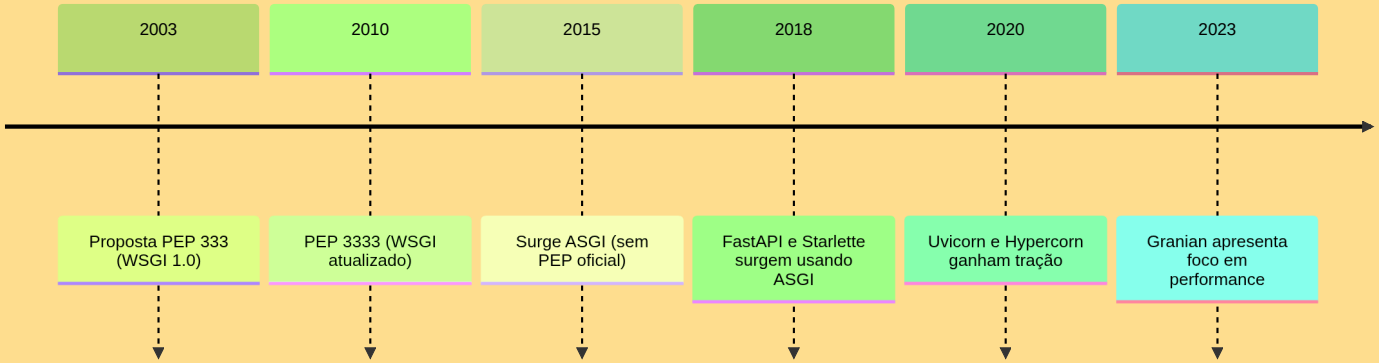
Seja para criar microserviços, aplicações monolíticas ou APIs em tempo real, o SGI que você escolhe vai guiar o tipo de aplicação que você pode construir.

ultimas

considerações

Uma pequena linha do tempo

Pode te ajudar a entender melhor onde cada coisa se posiciona no tempo:



Quando usar WSGI ou ASGI?

- Use **WSGI** se:
  - Sua aplicação é 100% síncrona (ex: Django clássico, Flask)
  - Você quer simplicidade e maturidade
  - Não vai usar WebSockets nem SSE
- Use **ASGI** se:
  - Você precisa de **concorrência real** com `asyncio`
  - Vai usar **WebSockets**, **HTTP/2/3**, **streaming**
  - Quer construir APIs mais performáticas e modernas

Referencias

[1] “ASGI Documentation — ASGI 3.0 documentation”. Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://asgi.readthedocs.io/en/latest/index.html

[2] emmett-framework/granian. (8 de outubro de 2025). Rust. Emmett. Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://github.com/emmett-framework/granian

[3] F. Manca, florimondmanca/awesome-asgi. (20 de outubro de 2025). Python. Acesso em: 20 de outubro de 2025. [Online]. Disponível em: https://github.com/florimondmanca/awesome-asgi

[4] “Gunicorn - Python WSGI HTTP Server for UNIX”. Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://gunicorn.org/

[5] “Hello, ASGI - Encode”. Acesso em: 19 de outubro de 2025. [Online]. Disponível em: https://www.encode.io/articles/hello-asgi

[6] “Hypercorn documentation — Hypercorn 0.17.3 documentation”. Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://hypercorn.readthedocs.io/en/latest/

[7] “Let’s build a WSGI server | csrgxtu”. Acesso em: 20 de outubro de 2025. [Online]. Disponível em: https://csrgxtu.github.io/2020/03/22/Let-s-build-a-WSGI-server/

[8] “PEP 333 – Python Web Server Gateway Interface v1.0 | peps.python.org”, Python Enhancement Proposals (PEPs). Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://peps.python.org/pep-0333/

[9] “PEP 3333 – Python Web Server Gateway Interface v1.0.1 | peps.python.org”, Python Enhancement Proposals (PEPs). Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://peps.python.org/pep-3333/

[10] “Understanding and Creating your Own ASGI Web Framework [Part-1] - JT Blog”. Acesso em: 20 de outubro de 2025. [Online]. Disponível em: https://teod-sh.github.io//diy-asgi-web-framework

[11] “Uvicorn”. Acesso em: 8 de outubro de 2025. [Online]. Disponível em: https://uvicorn.dev/

[12] “Writing an ASGI server from scratch and using it with FastAPI | by Rahul Salgare - Freedium”. Acesso em: 20 de outubro de 2025. [Online]. Disponível em: https://freedium.cfd/https://medium.com/@rsalgare95/writing-an-asgi-server-from-scratch-and-using-it-with-fastapi-21ec1191f3c7

[13] “WSGI — WSGI.org”. Acesso em: 19 de outubro de 2025. [Online]. Disponível em: https://wsgi.readthedocs.io/en/latest/