Boas práticas para clientes HTTP

Live de Python # 305

Esta aula **não vai ensinar HTTP em si**. Isso já foi abordado em lives anteriores.

Antes de tudo

Roteiro

1. Clientes?

Existem tantos...

2. A rede

E as coisas dando errado!

3. O protocolo a nosso favor

Headers de chace e rate-limit

4. Logs

É bom sempre saber o que está rolando

Existem tantos...

Clientes

O que é um cliente HTTP?

- Responsável pela **comunicação** com serviços externos
- Lida com:
  - Latência: o tempo entre o pedido e a resposta
  - Falhas: as coisas dão errado
  - Protocolos: muitas coisas rodam sob o HTTP
- **Não** deveria conter regra de negócio

Cliente HTTP não é só "`GET` e `POST`". Diversas coisas devem ser levadas em consideração a depender dos cenários.

A primeira

Em Python, não existe uma única biblioteca HTTP padrão dominante. Temos várias opções maduras, com propostas diferentes:

- **http.client**: cliente HTTP da stdlib, extremamente **baixo nível**, raramente usado diretamente
- **urllib**: biblioteca padrão do Python, **baixo nível**, API verbosa, pouco ergonômica
- **Requests**: biblioteca mais popular, API simples e amigável, apenas **síncrona**
- **AIOHTTP**: totalmente **assíncrona**, alta performance, API mais complexa, cliente e servidor no mesmo pacote
- **Niquests**: fork moderno do Requests, API muito semelhante, melhor suporte a TLS e HTTP moderno
- **HTTPX**: **síncrono e assíncrono**, suporte a HTTP/2, configurações explícitas

Comparação rápida

Algumas coisas simples para considerar:

Biblioteca	Sync	Async	HTTP/2	HTTP/3	Facilidade
http.client	✅	❌	❌	❌	⭐
urllib	✅	❌	❌	❌	⭐⭐
Requests	✅	❌	❌	❌	⭐⭐⭐⭐⭐
AIOHTTP	❌	✅	❌	❌	⭐⭐
Niquests	✅	⚠️	⚠️	⚠️	⭐⭐⭐⭐
HTTPX	✅	✅	✅	❌	⭐⭐⭐⭐

Às vezes seu projeto é só sync, às vezes só async. Tem vezes que não sabemos.

Às vezes precisa de HTTP/2. Outras vezes temos que lidar somente com a `stdlib`.

Por isso nessa live vamos de HTTPX

Porque ele:

- Funciona em código sync e async
- Incentiva boas práticas
- Tem API clara e explícita
- Tem um bom ecossistema de plugins
- Escala bem para aplicações reais *
- `pip install httpx`

*: nada ciêntifico, minha experiência somente

Tudo que vamos falar aqui, pode ser adaptado em qualquer cliente, até mesmo em outras linguagens.

Porém, vale lembrar

Isso decidido

O mínimo sobre HTTPX que precisamos saber:

```
import httpx

url = 'https://apilegal/users'
params={'key1': ['value1', 'value2']}
content = b'A request body on a DELETE request.'
json = {'name': 'Fausto'}
headers = {'Content-Type': 'application/json'}
proxy = httpx.Proxy("http://127.0.0.1:8080")

httpx.head(url)
httpx.get(url, params=params)
httpx.post(url, content=content)
httpx.put(url, json=json)
httpx.patch(url, headers=headers)
httpx.delete(url, proxy=proxy)
```

A rede

e as coisas dando errado!

A rede pode ser traiçoeira

Talvez, os problemas mais comuns, sejam relacionados a:

- A resposta pode ser demorada (timeouts)
- Pode dar errado por nenhum motivo específico (500)
- O que se procura pode não ser encontrado (300/400)
- Estabelecer a conexão pode ser exaustivo (sockets, malditos sockets)
  - Muitas vezes repetitivo também

Como ser resiliente nesses casos? **Esperando sempre o pior a todo custo** e estando preparado para lidar com essas coisas.

Mas como?

A rede pode ser traiçoeira

Talvez, os problemas mais comuns, sejam relacionados a:

- A resposta pode ser demorada (timeouts)
- Pode dar errado por nenhum motivo específico (500)
- O que se procura pode não ser encontrado (300/400)
- Estabelecer a conexão pode ser exaustivo (sockets, malditos sockets)
  - Muitas vezes repetitivo também

Como ser resiliente nesses casos? **Esperando sempre o pior a todo custo** e estando preparado para lidar com essas coisas.

Mas como? **TESTES**

Timeouts

Vamos pensar em um exemplo extremamente simples:

```
import httpx


def get_page_content():
    return httpx.get('https://httpbin.org/delay/6').text


get_page_content()
```

O que acontece se a resposta não voltar no tempo que esperamos?

```
  File "live_http_client/.venv/.../httpx/_transports/default.py",
  line 118, in map_httpcore_exceptions
    raise mapped_exc(message) from exc
httpx.ReadTimeout: timed out
```

É essencial testar

Se quisermos simular cenários do que pode acontecer é sempre bom poder contar com um bom apoio de testes.

Primeiro por que você não quer descobrir que deu erro em produção e quando acontecer, porque **vai**, você garante que não vai ser pelo cenário mais bobo possível.

Nossa missão vamos pegar carona como o `RESPX` e o `pytest`:

- `pip install respx`
- `pip install pytest`

Para o requests temos o `Responses`, o AIOHTTP tem um plugin para pytest

Simulando esse cenário

O RESPX fornece uma fixture para o pytest chamada `respx_mock`:

```python
from respx import MockRouter


def test_get_page_content_timeout(respx_mock: MockRouter):
    # no get você poderia explicitar a URL, se quiser
    mocked = respx_mock.get().mock(
        side_effect=httpx.ReadTimeout('timed out')
    )
    content = get_page_content()
    assert mocked.called
```

Simulando esse cenário

O RESPX fornece uma fixture para o pytest chamada `respx_mock`:

```
from respx import MockRouter


def test_get_page_content_timeout(respx_mock: MockRouter):
    mocked = respx_mock.get().mock(
        side_effect=httpx.ReadTimeout('timed out')
    )
    content = get_page_content()
    assert mocked.called
```

E com isso temos:

```
FAILED exemplos_slides/timeout_v0.py::test_get_page_content_timeout
httpx.ReadTimeout: timed out
```

O que aprendemos?

Que precisamos efetivamente nos preocupar com esse problema no fluxo:

```python
def get_page_content() -> str:
    try:
        response = httpx.get('http://127.0.0.1:8000/delay/6')
    except httpx.ReadTimeout:
        # Aqui entra a sua regra
        return ''
    else:
        return response.text

def test_get_page_content_timeout(respx_mock: MockRouter):
    # ...
    content = get_page_content()

    assert mocked.called
    assert content == ''
```

Claro, tem mais coisa...

É bom **sempre** ser explícito com `timeouts`, você nunca sabe o quanto pode demorar. Mas também é bom deixar explícito o quanto de tempo dispõe para esperar.

Logo:

```
def get_page_content() -> str:
    try:
        response = httpx.get(
            'http://127.0.0.1:8000/delay/6',
            timeout=3
        )
    except httpx.ReadTimeout:
        # Aqui entra a sua regra
        return ''
    else:
        return response.text
```

Erros sem motivo do cliente

Como nem tudo na rede são flores, vamos imaginar um erro do grupo `500`. Quando você envia tudo certo, mas por algo com nenhuma relação com o cliente, dá errado.

Temos que nos preparar para isso.

```
from http import HTTPStatus

def test_get_page_content_server_error(respx_mock: MockRouter):
    mocked = respx_mock.get().mock(
        return_value=httpx.Response(
            HTTPStatus.INTERNAL_SERVER_ERROR,
            content=b'deu ruim no server :('
        ))
    content = get_page_content()
    assert mocked.called
```

Usar constantes como `HTTPStatus.INTERNAL_SERVER_ERROR` é muito melhor que tentar adivinhar o status code :)

A grande surpresa

```
timeout_v0.py::test_get_page_content_server_error PASSED
```

QQQQ??????????????????

SIM! não é pq deu erro que perde o content. Não é o que queremos, mas ele está lá, mesmo que vazio. E lá vamos nós...

```python
def get_page_content() -> str:
    try:
        response = httpx.get('http://127.0.0.1:8000/delay/6', timeout=3)
        response.raise_for_status()
    except httpx.ReadTimeout:
        return ''
    except httpx.HTTPStatusError:
        # Aqui entra a sua outra regra
        return ''
    else:
        return response.text
```

Uma coisa digna de nota

Mesmo quando `raise_for_status()` é levantado, o conteúdo do `Response` continua disponível:

```
def get_page_content() -> str:
    try:
        response = httpx.get('http://127.0.0.1:8000/delay/6', timeout=3)
        response.raise_for_status()
    except httpx.HTTPStatusError:
        # O Response permanece com os dados do retorno intáctos
        return response.text
    # ...
```

Outra coisa importante é que todos os erros 4xx e 5xx entram no raise.

Posso sobreviver com esse erro?

Às vezes a resposta da requisição é uma **dependência lógica** do seu código e você não conseguiria seguir a execução por um erro estranho.

- Pode ser que algo errado seja corrigido em breve, pode ser que pegamos a coisa no meio de um processo de deploy
- Pode ser que todas as conexões do pool do banco estivessem sendo usadas na hora.
- Pode ser somente **flaky**

Como tentar de novo?

No ecossistema do httpx, temos uma biblioteca pensada para isso:

```
pip install httpx-retries
```

Também suporta urllib3 e requests

Retries e Backoff

Podemos repetir a chamada e adicionar um fator de "afastamento". A cada chamada, multiplicamos por esse tempo:

```python
from httpx_retries import Retry, RetryTransport

retry = Retry(total=5, backoff_factor=0.5)
transport = RetryTransport(retry=retry)

with httpx.Client(transport=transport) as client:  # já chegamos aqui...
    response = client.get("https://example.com")
```

Seguindo algo como `backoff_factor * (2 ** attempts_made)`: 0.5 * (2 ** 0)$, $0.5 * (2 * *1)$, ..., $0.5 * (2 * *N)$

PS: retry não resolve tudo, se for erro lógico, você só vai errar mais vezes em um espaço de tempo

Como isso ficaria no nosso código?

Algo parecido com isso:

```python
from httpx_retries import Retry, RetryTransport

retry = Retry(total=5, backoff_factor=0.5)
transport = RetryTransport(retry=retry)

def get_page_content() -> str:
    with httpx.Client(transport=transport) as client:  # Próximo tópico!
        try:
            response = client.get('http://127.0.0.1:8000/delay/6')
            response.raise_for_status()
        except httpx.ReadTimeout: # sua regra
            return ''
        except httpx.HTTPStatusError: # sua regra
            return ''
        else:
            return response.text
```

Mas como sei se a regra funciona?

Como sempre... **testes**

```python
def test_get_page_content_timeout_retry(respx_mock: MockRouter):
    mocked = respx_mock.get().mock(
        side_effect=[
            httpx.ReadTimeout('pei'),
            httpx.Response(HTTPStatus.OK, content=b'Deu bom')
        ]
    )

    content = get_page_content()

    assert mocked.call_count == 2
    assert content == 'Deu bom'
```

Sempre importante dar uma olhada na documentação, para entende os casos em que o retry é ativado e quem sabe adaptar as regras para o seu contexto.

O custo escondido de um request

Falando em boas práticas, uma coisa que não costumamos pensar muito é no **custo computacional** envolvido em uma requisição.

Excluindo as coisas mais básicas de código. Todo request deve:

- Abrir socket
- Resolver DNS
- Handshake TCP
- Handshake TLS
- Enviar request
- Receber response
- Fechar tudo 😩

Fazer isso toda vez é caro e desnecessário.

Lhe apresento o `Client`

Uma simplória linha de código:

```
with httpx.Client() as client:
    client.get(url)
    client.get(url)
    client.get(url)
```

- Mantém conexões abertas
- Reutiliza sockets
- Respeita keep-alive
- Controla limites

Além de evitar dizer a mesma coisa diversas vezes

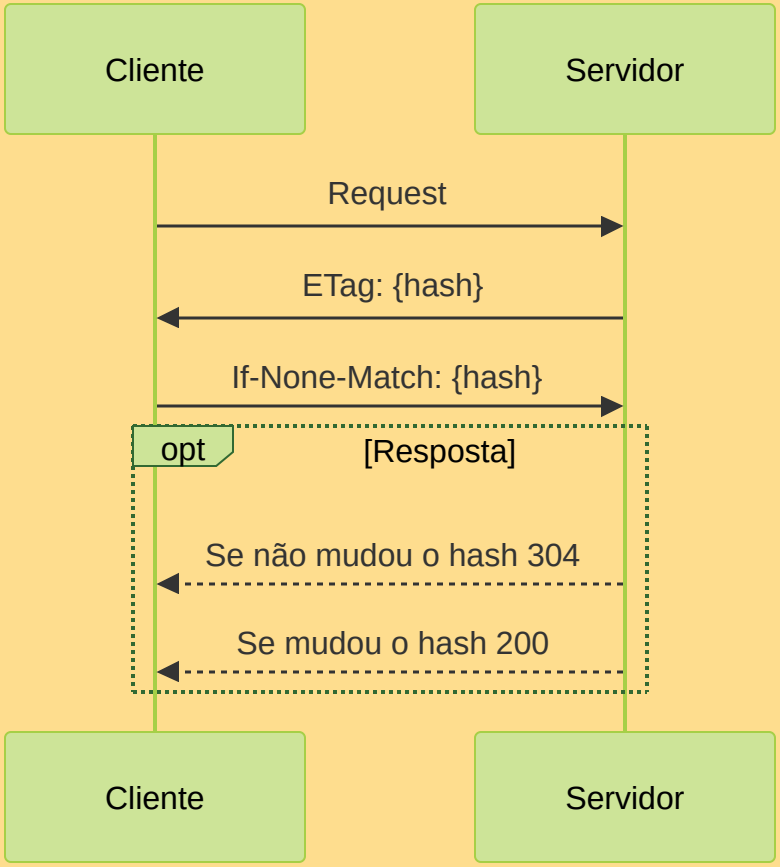
- Centraliza config (timeout, base_url, proxy, retry, headers, ...)

Usando o protocolo

a nosso favor

Cache HTTP

Existem diversos mecanismos de cache no protocolo HTTP (RFC 9111). Como, por exemplo, a `ETag`, que vamos usar para não sermos exaustivos.



Um exemplo em código

Algo +- assim:

```python
import httpx
url = 'https://dunossauro.com'

# 1a request
resp = httpx.get(url)
print(f'status: {resp.status_code}')  # 200: OK

cached_etag = resp.headers.get('ETag')

# 2a request
resp = httpx.get(url, headers={'If-None-Match': cached_etag})
print(f'status: {resp.status_code}')  # 304: Not Modified
```

Aqui vale lembrar, que se você precisar do conteúdo da resposta anterior, é bom que você guarde ele também. Pois, o 304 não terá content.

Como simular isso nos testes?

Algo parecido como isso deve bastar (olhar `etag_maluca.py`):

```
def test_get_page_content_etag(respx_mock):
    mocked = respx_mock.get()
    mocked.side_effect = [
        httpx.Response(
            HTTPStatus.OK, content=b'test', headers={'ETag': 'test-tag'}
        ),
        httpx.Response(HTTPStatus.NOT_MODIFIED)
    ]

    content_a = get_page_content()
    content_b = get_page_content()

    assert mocked.call_count == 2
    assert content_a == content_b
```

Embora eu não queira me aprofundar nisso, por um motivo bastante específico!

Simplificando a vida

Lidar com uma RFC toda no fonte, você deve imaginar que não é tão interessante. Para resolver esses casos, temos coisas como o Hishel.

- Biblioteca para lidar com a RFC 9111
- Adaptável para o cliente
- Custo praticamente zero de implementação
- Facilita as respostas
- `pip install hishel`

Curiosidade: Se pronuncia como a gíria "ish", com o "el" do espanhol. Significa 'lembrar' em Armeno.

Hishel suporta tanto o cliente (requests, httpx), quando o servidor (fastapi, blacksheep).
Vale uma live somente dele + HTTP cache?

Hishel

Adicionando ele na camada de transporte do httpx, já temos toda a implementação de cache disponível:

```python
import httpx
import hishel
import hishel.httpx

transport = hishel.httpx.SyncCacheTransport(  # Cria um cache
    next_transport=httpx.HTTPTransport(),
    storage=hishel.SyncSqliteStorage(),  # Salva no DB
)

with httpx.Client(transport=transport) as client:
    r1 = client.get('https://dunossauro.com')
    r2 = client.get('https://dunossauro.com')
    print(r1.status_code, r2.status_code)  # 200, 200
```

Analisando o resultado

Embora tenha voltado `200 OK` para as duas, o que ele fez foi retornar o resultado anterior, em cache, quando o segundo deu `304 NOT_MODIFIED`.

Vamos inspecionar usando um `proxy` [abrir o `mitmproxy` e o `proxy_hisel.py`]:

```
proxy = httpx.Proxy("http://127.0.0.1:8080")

proxy_transport = httpx.HTTPTransport(proxy=proxy, verify=False)

transport = hishel.httpx.SyncCacheTransport(
    # delega pro proxy os headers do proxy
    next_transport=proxy_transport,
    storage=hishel.SyncSqliteStorage(),
)

with httpx.Client(transport=transport) as client: ...
```

O que aprendemos aqui?

- A lógica do cache não precisa existir
- A implementação é bastante simples
- Diminuí o custo de rede
- Por consequência, algumas respostas mais rápidas

Lidando com limites!

Uma das coisas que lidamos no dia-a-dia, mas que tem uma RFC ainda em draft são os rate-limits

Um conjunto de Headers para o server comunicar quantas requisições ele está disposto a receber em um delta de tempo.

- Quantos requests por dia, minuto, semana, ...
- Sem um padrão oficial **ainda**
- Mas evita levar BAN de alguns servidores
- A descoberta vem do seu relacionamento com a API :(
- `pip install pyrate-limiter`
  - Escolhida por funcionar sync e async

Existem várias libs para isso, aiolimiter, aiometer (minha preferida), httpx-limiter.

Não vou me estender aqui por falamos bastante sobre limits na Live 234: Requests assíncronos com HTTPX

Aplicando!

Um exemplo bobo:

```python
from pyrate_limiter import limiter_factory
from pyrate_limiter.extras.httpx_limiter import RateLimiterTransport
from pyrate_limiter.abstracts.rate import Duration
import httpx


limiter = limiter_factory.create_inmemory_limiter(
    rate_per_duration=1, duration=Duration.SECOND,
)  # Um request por segundo

limiter_transport = RateLimiterTransport(limiter=limiter)

with httpx.Client(transport=limiter_transport) as client:
    print(client.get('https://dunossauro.com'))
```

Logs

sempre bom saber o que está rolando!

Logs?

Quando falamos em logs aqui, quero me referir a algo um pouco mais do que "enviei X / recebi Y". Em produção, às vezes precisamos mais do que um "Enviei o request". Precisamos saber:

- **Qual request**
- Quais os dados
- Se falhou, por que falhou?
- Isso foi proliferado na infra da empresa?
- Onde foi parar?

O problema clássico

Em produção, isso não ajuda muito:

"Fiz um request para /users"

Quando existem:

- Vários requests ao mesmo tempo
- Mais de uma instância
- Falhas intermitentes

O básico do básico

`Request-ID`:

- Cada request recebe um ID único
- Esse ID aparece em todos os logs
- Se algo der errado, você sabe **qual foi**
- UUID é simples: não precisa ser perfeito, só precisa existir

```
import time

def before_request(request: httpx.Request):
    request_id = str(uuid.uuid4())
    request.headers['X-Request-ID'] = request_id
    request.extensions['request_id'] = request_id
    request.extensions['start_time'] = time.monotonic() # isso é bom!

with httpx.Client(event_hooks={'request': [before_request]}) as client:
          ...
```

Outro clássico

- "As requisições XPTO estão demorando."
- "O sistema parece lento"
- Aquela sensação sem dados reais!
- Que tal logar o tempo passado após o request?

```
def after_request(response: httpx.Response):
    request = response.request
    start = response.request.extensions.get("start_time", None)
    if start:
        elapsed = time.monotonic() - start
    else:
        elapsed = None

    logger.info(
        f'<sua DSL> {request.method} {request.url} {response.status_code} '
        f'{elapsed} {request.extensions.get("request_id")}'
    )
```

O ganho real

Mesmo nesse nível simples, você já consegue:

- Identificar requests problemáticos
- Ver lentidão real
- Entender comportamento em produção
- Debugar sem chutar

`Importante lembrar`:

- Não é tracing
- Não é observabilidade completa
- Não resolve tudo

Mas é o **mínimo saudável** para clientes HTTP.

apoia.se/livedepython

pix@dunossauro.com

patreon.com/dunossauro

Ajude o projeto

Albano Maywitz, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Neto, Alynnefs, Alysson Oliveira, Andre Makoski, Andre Mesquita, Andre Paula, Andre Souza, Antonio Filho, Apolo Ferreira, Aurelio Costa, Belisa Arnhold, Bennie Lima, Bernarducs, 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, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Linux, Danilo Silva, Darcioalberico, David Couto, Dh44s, Diego Guimarães, Diego Suhett, Dilan Nery, Dunossauro, Edgar, Elias Moura, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fabiano Gomes, Fábio Belotto, Fabiokleis, Fabricio Segundo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Ferrabras, Fightorfall, Francisco Aclima, Franklin Sousa, Franko, Frederico Damian, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Gnomo Nimp, Grinaode, Guilherme Felitti, Guintter, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Herian Cavalcante, Hiago Couto, Iuri Kintschev, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, Jonas Araujo, Jose Andrade, Jose Barroso, Joseíto Júnior, José Predo), Jota_lugs, Jplay, Jrborba, Ju14x, Juan Felipe, Juliana Machado, Julio Cesar, Julio Franco, Julio Gazeta, Julio Silva, Kaio Engineer, Kaio Peixoto, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lisandro Pires, Looshigooshi, Lucas Carderelli, Lucas Castro, Lucas Polo, Lucas Schneider, Luciano Ratamero, Lúcia Silva, Luidduarte, Luis Eduardo, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Marcel Liguili, Marcelo Fonseca, Marcio Freitas, Marcos, Marcos Almeida, Marina Passos, Mateusamorim96, Matheus Vian, Medalutadorespacialx, Mlevi Lsantos, Murilo Carvalho, Nhambu, Oopaze, Otávio Carneiro, Patrick Felipe, Prettyhuman, Programming, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, 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 Vieira, Rogério Nogueira, Rui Jr, Samael Picoli, Santhiago Cristiano, Scrimf00x, Scrimfx, Shirakawa, Studies, Téo Calvo, Teomewhy, Thamires Betin, Tharles Andrade, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tomás Tamantini, Valdir, Varlei Menconi, Vinícius Areias, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vladimir Lemos, Williamslews, Willian Lopes, Xxxxxxxxx, Zero! Studio

Obrigado você <3