REST API: Por que você complica?

Boas práticas para simplificar e melhorar suas APIs

Quem sou eu

Victor Raton (v_raton)

- Dessenvolvedor fullstack +5 anos
- Cursando pós-graduação em Data science & Analytcs pelo SENAI - CIMATEC
- Contribuidor independente em SDR Virtual e Wokpy

O que NÃO é esta apresentação

- ❌ Criar API REST do zero
- ❌ Curso de HTTP
- ❌ Curso de FastAPI/Python

O que É esta apresentação

- ✅ Conceitos RESTful
- ✅ Facilitar a tomada de decisão e padronização
- ✅ Evitar erros comuns que causam problemas de performance

> Para aprender a desenvolver uma API REST veja o curso do Eduardo Mendes (dunossauro) FastAPI do zero

REST API - Conceito

Representational State Transfer

- Transferência de estado de recursos
- Servidor HTTP entrega JSON
- Verbos: GET, POST, PUT, PATCH, DELETE
- Dados e metadados

RESTful

> As APIs que seguem o estilo de arquitetura REST são chamadas de APIs REST. Os serviços da Web que implementam a arquitetura REST são chamados de serviços da Web RESTful

- AWS: O que é uma API RESTful.

"RESTaurante"

Verbo	Ação	Exemplo
GET	Ler cardápio	`/cardapio`
GET	Detalhes item	`/cardapio/12`
POST	Fazer pedido	`/pedido`
PATCH	Solicitar mudança de pedido (Remover abacaxi)	`/pedido/{id}`
PUT	Trocar pedido inteiro	`/pedido/{id}`
DELETE	Cancelar	`/pedido/{id}`

Status Codes - Comunicação Imediata

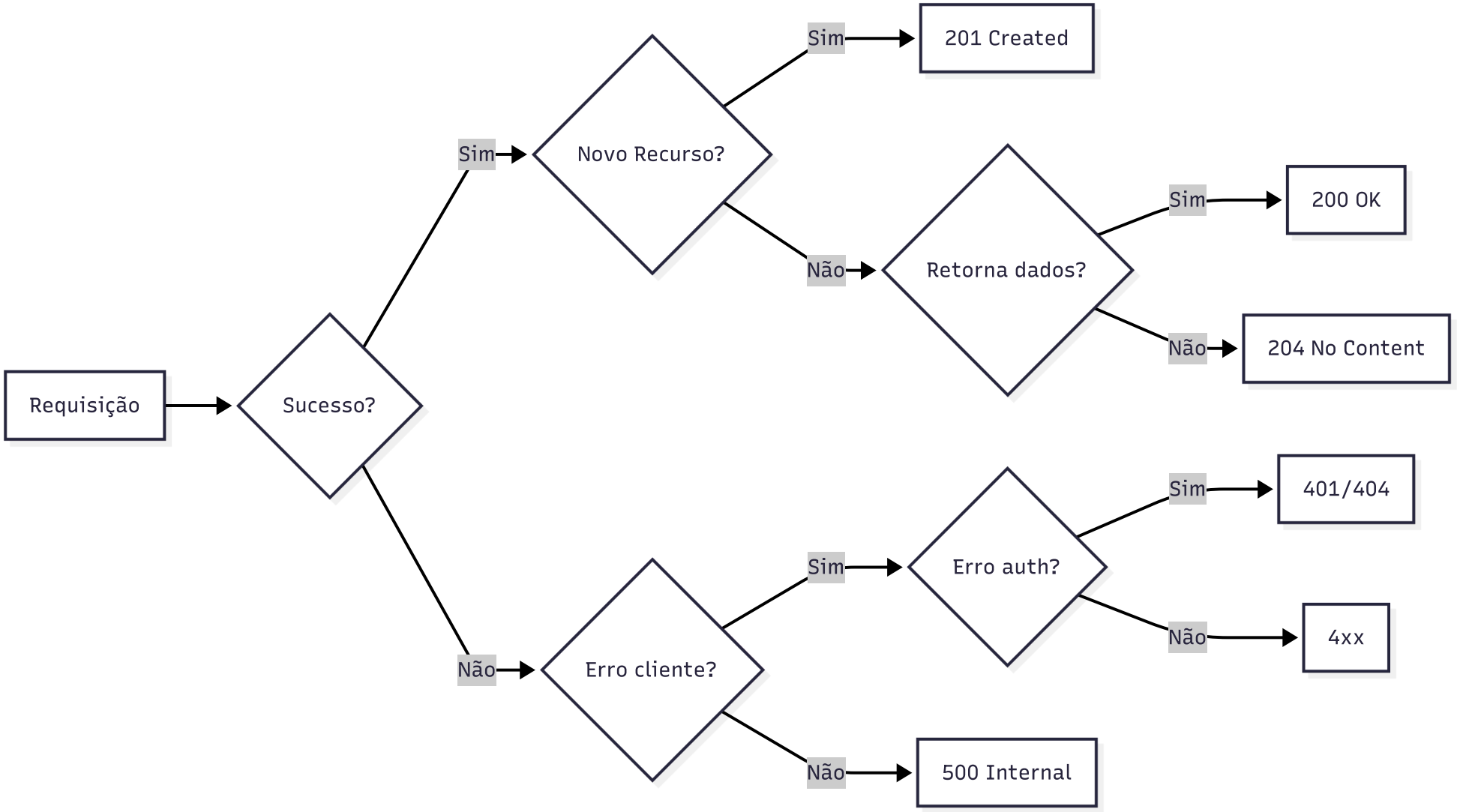
Repreesentação numérica usada pelo protocolo HTTP onde faixas representam os grupos

```
1xx → Informação
2xx → Sucesso     ← "Caminho feliz"
3xx → Redirecionamento
4xx → Erro do cliente
5xx → Erro do servidor
```

Os Mais Importantes

Código	Significado	Uso
200	OK	Sucesso geral
201	Created	POST/PUT criam recurso
204	No Content	DELETE bem-sucedido*
401	Unauthorized	Não autenticado
403	Forbidden	Sem permissão
404	Not Found	Recurso não existe
500	Internal Error	Erro no servidor

Escolhendo um status code



Headers HTTP - Metadados

> Informações extras sobre a requisição/transação

Exemplos de uso:

- Autenticação (Bearer token)
- Content-Type
- Cache policy
- Paginação

Headers HTTP - Metadados

```
HTTP/1.1 200 OK
Content-Type: application/json
Total-Count: 1000
Items-Per-Page: 10
Current-Page: 1
Items-Returned: 1
Link: </items?page=2>; rel="next"
```

HATEOAS - Por que complica?

Hypermedia as the Engine of Application State

- Mapeando dados nos reetornos
- Links de referência para entidades

Formatos existentes:

- HAL, JSON:API, Siren, Hydra, Collection+JSON, UBER

Problemas

- Serialização bloqueante para metadados
- Interfere no retorno dos dados
- Overhead mesmo em listas vazias

```
{
  "_links": { "self": { "href": "..." } },
  "id": "1",
  "name": "Livro"
}
// Metadados no body = problema de performance
```

Solução: Separar Dados de Metadados

> RFC 8288 - Web Linking

Headers para metadados

```
HTTP/1.1 200 OK
Content-Type: application/json
Total-Count: 1000
Total-Pages: 100
Items-Per-Page: 10
Current-Page: 1
Items-Returned: 1
Link: </items?page=2>; rel="next",
      </items?page=100>; rel="last"
```

Solução: Separar Dados de Metadados

Body simples

```
[
  { "id": 1, "title": "O Senhor dos Anéis" }
]
```

Vantagem: Metadados não bloqueiam leitura dos dados **Desvantagem** Exige a necessidade de documentação externa exportada como OpenAPI/Swagger

Teste de comparação: HATEOS vs HEAD metadata

Metodologia de testes

1. Teste de carga por meio do utilitário `wrk`
    - Tempo de resposta médio
    - Total de requisições
    - 100 conexoes concorrentes, 4 threads 10s
2. Teste de cliente com request (Python)
    - Consumo de API
    - Tempo médio para consultar lista paginada
    - Tempo médio para consultar lista vazia
    - Uso de RAM em serialização

Código completo e atualizado em: https://codeberg.org/v_raton/restapi-simplifcado-demo

Estrutura da API

- endpoint de health em `/feed/health`
- endpoint de request em HAL `/feed/hal`
- endpoint de request + headers metadata `/feed/headers`
- apispec `/docs`

Teste de carga

```
#!/usr/bin/env bash
set -e

SERVER_URL="${SERVER_URL:-http://localhost:8000}"
DURATION="${DURATION:-10s}"
THREADS="${THREADS:-4}"
CONNECTIONS="${CONNECTIONS:-100}"

echo "=== REST API Server Benchmark (wrk) ==="
echo "Server: $SERVER_URL"
echo "Duration: $DURATION"
echo "Threads: $THREADS"
echo "Connections: $CONNECTIONS"
echo ""

echo "--- Benchmarking /feed/headers endpoint ---"
wrk -t$THREADS -c$CONNECTIONS -d$DURATION "$SERVER_URL/feed/headers?page=1&per_page=10"

echo ""
echo "--- Benchmarking /feed/hal endpoint ---"
wrk -t$THREADS -c$CONNECTIONS -d$DURATION "$SERVER_URL/feed/hal?page=1&per_page=10"

echo ""
echo "--- Benchmarking /feed/headers (empty page) ---"
wrk -t$THREADS -c$CONNECTIONS -d$DURATION "$SERVER_URL/feed/headers?page=10000&per_page=10"

echo ""
echo "--- Benchmarking /feed/hal (empty page) ---"
wrk -t$THREADS -c$CONNECTIONS -d$DURATION "$SERVER_URL/feed/hal?page=10000&per_page=10"

echo ""
echo "=== Benchmark complete ==="
```

Teste de client

```
def benchmark_headers(self, page: int, per_page: int) -> dict:
    latencies = []
    parse_times = []
    mem_usage = []
    body_sizes = []
    items_count = 0

    for _ in range(self.warmup):
        self.http_client.get(
            f"{self.base_url}/feed/headers", params={"page": page, "per_page": per_page}
        )

    for _ in range(self.iterations):
        start = time.perf_counter()
        response = self.http_client.get(
            f"{self.base_url}/feed/headers", params={"page": page, "per_page": per_page}
        )
        http_latency = (time.perf_counter() - start) * 1000
        latencies.append(http_latency)
        body_sizes.append(len(response.content))

        items_count = int(response.headers.get("X-Total-Count", 0))

        if items_count > 0:
            parse_time, mem_kb, _ = measure_json_parse(response.text)
            parse_times.append(parse_time * 1000)
            mem_usage.append(mem_kb)
        else:
            parse_times.append(0)
            mem_usage.append(0)

    return {
        "latency_ms": {"mean": mean(latencies), "std": stdev(latencies) if len(latencies) > 1 else 0},
        "parse_ms": {"mean": mean(parse_times), "std": stdev(parse_times) if len(parse_times) > 1 else 0},
        "memory_kb": mean(mem_usage) if mem_usage else 0,
        "body_bytes": mean(body_sizes),
        "items_count": items_count,
    }
```

Teste de client

```python
def benchmark_hal(self, page: int, per_page: int) -> dict:
    latencies = []
    parse_times = []
    mem_usage = []
    body_sizes = []

    for _ in range(self.warmup):
        self.http_client.get(
            f"{self.base_url}/feed/hal", params={"page": page, "per_page": per_page}
        )

    for _ in range(self.iterations):
        start = time.perf_counter()
        response = self.http_client.get(
            f"{self.base_url}/feed/hal", params={"page": page, "per_page": per_page}
        )
        http_latency = (time.perf_counter() - start) * 1000
        latencies.append(http_latency)
        body_sizes.append(len(response.content))

        parse_time, mem_kb, _ = measure_json_parse(response.text)
        parse_times.append(parse_time * 1000)
        mem_usage.append(mem_kb)

    return {
        "latency_ms": {"mean": mean(latencies), "std": stdev(latencies) if len(latencies) > 1 else 0},
        "parse_ms": {"mean": mean(parse_times), "std": stdev(parse_times) if len(parse_times) > 1 else 0},
        "memory_kb": mean(mem_usage),
        "body_bytes": mean(body_sizes),
    }
```

Parser de json manual a partir de string

```
def measure_json_parse(data: str) -> tuple:
    tracemalloc.start()
    start = time.perf_counter()
    result = json.loads(data)
    parse_time = time.perf_counter() - start
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return parse_time, peak / 1024, result
```

```python
def main():
    import sys

    base_url = sys.argv[1] if len(sys.argv) > 1 else "http://localhost:8000"
    iterations = int(sys.argv[2]) if len(sys.argv) > 2 else 100

    benchmark = ClientBenchmark(base_url, iterations)

    print(f"=== Client-side Benchmark ===")
    print(f"Server: {base_url}")
    print(f"Iterations: {iterations}, Warm-up: 10")

    headers_result = benchmark.benchmark_headers(page=1, per_page=10)
    hal_result = benchmark.benchmark_hal(page=1, per_page=10)
    print_results(headers_result, hal_result, "Non-empty Page (page=1, per_page=10)")

    headers_empty = benchmark.benchmark_headers(page=1001, per_page=10)
    hal_empty = benchmark.benchmark_hal(page=1001, per_page=10)
    print_results(headers_empty, hal_empty, "Empty Page (page=1001, per_page=10)")
```

Resultados de teste de carga

Feed com dados (page=1, per_page=10)

Endpoint	Latência Média	Req/Sec	Total Requests	Transferência
/feed/headers	84.75ms	1174.65	11760	2.38MB/s
/feed/hal	96.27ms	1033.43	10346	2.02MB/s

Feed vazia (page=10000, per_page=10)

Endpoint	Latência Média	Req/Sec	Total Requests	Transferência
/feed/headers	76.90ms	1295.31	12964	430.08KB/s
/feed/hal	83.86ms	1187.43	11892	398.90KB/s

Resumo do impacto:

- Headers: ~12% mais rápido que HAL com dados, ~9% mais rápido em feed vazia
- Transfer: Headers transfere ~18% mais dados (metadados nos headers vs body)
- Vazia: HAL tem overhead de serialização mesmo sem dados

Resultados de benchmark de cliente

Página com dados (page=1, per_page=10)

Métrica	Headers	HAL
HTTP latency (ms)	2.55±0.27	2.56±0.23
JSON parse (ms)	0.1506	0.1626
Memory (KB)	3.93	4.59
Response (bytes)	1794	1925

Página vazia (page=1001, per_page=10)

Métrica	Headers	HAL
HTTP latency (ms)	2.34±0.22	2.44±0.14
JSON parse (ms)	0.0000	0.0580
Memory (KB)	0.00	1.88
Response (bytes)	2	217

Resumo do impacto:

- **Com dados**: Headers ~7% mais rápido em parse JSON, ~14% menos memória
- **Vazia**: Headers evita realizar o parse completo, ~99% menos bytes transferidos
- **Economia total**: ~131 bytes por requisição com dados, ~215 bytes por requisição vazia

Boas Práticas

- ✅ POST/PUT → **201 Created**
- ✅ GET/HEAD → **200**
- ✅ DELETE → **204 No Content**
- ✅ Location header para recursos criados
- ⚠️ Moderar contadores em bases grandes
- ⚠️ Evitar headers customizados que proxies sobrescrevem

Dica de Segurança

> Para prevenir brute-force: responder **404** em vez de **401** em falhas de autenticação

Referências

- RFC 8288 - Web Linking
- RFC 6648 - Deprecating "X-" Prefix
- MDN - HTTP Headers
- FastAPI do Zero - Curso recomendado
- O que é uma API RESTful

FIM

Perguntas?