

Curso Programação Backend com Python

Servidores e clientes HTTP / REST

**© Cleuton Sampaio 2021**

**Sumário**

[HTTP e HTTPS](#_3dy6vkm) **3**

[Request / Response](#_n3u3niuhc1lf) 3

[Um servidor python com Flask](#_p7tslu2bcrnq) 9

[Clientes HTTP python com Requests](#_4mqiowtxo4) 10

[Enviando e recebendo JSON](#_edb8c6muuonm) 12

[**RESTful services**](#_429scqeg7s8w) **14**

[Recursos](#_523391ike6gh) 14

[API REST](#_bz15esxnxt6y) 15

[RESTful-URI x Non RESTful-URI](#_r88078clanqz) 15

[JSON](#_i291pbn0e10m) 15

[Métodos HTTP e request REST](#_atlq7gkgh159) 18

# HTTP e HTTPS

Vamos começar com servidores HTTP (ou HTTPS). A diferença entre ambos é que no HTTPS você precisa instalar um Certificado digital, que pode ser adquirido em vários fornecedores. Neste curso, vamos focar em HTTP, mas a diferença é apenas a instalação e configuração do certificado.

HTTP (e o HTTPS) é um protocolo de nível de aplicação que trafega recursos (texto, HTML, JSON, imagens etc) entre um **Cliente** (Frontend) e um **Servidor** (Backend).

Se vocẽ não conhece bem redes e TCP/IP, eu recomendaria dar uma estudada no assunto antes de prosseguir.

## Request / Response

O HTTP é um protocolo de request/response, ou seja, você envia uma mensagem contendo o pedido de um recurso e, caso ele exista e você tenha permissão, o servidor responde com o recurso.

Vamos brincar com o cURL, que é um utilitário para enviar comandos HTTP. Se você usa Linux, provavelmente já o tem instalado. Caso contrário, é simples:

sudo apt update

sudo apt install curl

Se você tem um Linux diferente de Debian ou Ubuntu, que não use apt, então pode instalar assim:

yum install curl

Agora, se tem MS Windows, não se preocupe, pois tem curl para ele. Baixe deste site:

<https://curl.se/windows/>

Temos um arquivo **exemplos\_http.zip** com vários exemplos interessantes. Vejamos o exemplo **servidorHTTP.py**:

from flask import Flask

from flask import request

app = Flask(\_\_name\_\_)

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/",methods=['GET'])

def raiz():

return "<p>Raiz GET!</p>"

@app.route("/usuarios/<usuario>", methods=['GET'])

def usuario(usuario):

return f"<p>Usuario = {usuario}</p>"

@app.route("/usuarios/incluir",methods=['POST'])

def incluir():

return f"<p>{request.form['usuario']}, {request.form['senha']}</p>"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=8080, debug=True)

Este é um exemplo de servidor HTTP feito em python, utilizando a biblioteca **Flask**. É claro que precisamos instalar a Flask assim como qualquer outra dependência antes de executar o programa. Leia o arquivo **instrucoes.txt** que está no mesmo zip.

Como você deve saber, o HTTP é baseado em request e response. O request pode ser feito por vários **métodos** diferentes:

* GET: Envia tudo na mesma linha da URL;
* POST: Envia um conteúdo adicional;
* DELETE: Solicita a remoção de um recurso;
* PUT: Altera um recurso;

No exemplo, o servidor lida com dois métodos diferentes: GET e POST. O servidor determinou 3 **rotas** HTTP:

* GET /
* GET /usuarios/<nome do usuário>
* POST /usuarios/incluir

Vamos testar todas elas com o curl. Para testar a rota raiz:

curl <http://localhost:8080/>

E a resposta é: <p>Raiz GET!</p>

Agora, vamos testar a segunda rota GET, enviando um parâmetro na URL (extra path):

curl <http://localhost:8080/usuarios/fulano>

Eis a resposta: <p>Usuario = fulano</p>

Finalmente, vamos enviar um método POST contendo dados no formato FORM (formulário):

curl -X POST -F "usuario=beltrano" -F "senha=teste" http://localhost:8080/usuarios/incluir

E a resposta é: <p>beltrano, teste</p>

No GET, enviamos variáveis na própria URL, de duas formas diferentes:

* Querystring: /usuarios?id=1
* Extra path: /usuarios/1

No POST, os dados são enviados no corpo do request. Vejamos a anatomia do POST:

curl -X POST --trace-ascii /dev/stdout \

-F "usuario=beltrano" -F "senha=teste" \ http://localhost:8080/usuarios/incluir

Essa barra invertida no final da linha é para permitir pular linha na console do Linux. No MS Windows seria ^. A opção “--trace-ascii” permite mostrar o comando POST completo:

0000: POST /usuarios/incluir HTTP/1.1

0021: Host: localhost:8080

0037: User-Agent: curl/7.68.0

0050: Accept: \*/\*

005d: Content-Length: 249

0072: Content-Type: multipart/form-data; boundary=--------------------

00b2: ----bfc967140684f75e

00c8:

=> Send data, 249 bytes (0xf9)

0000: --------------------------bfc967140684f75e

002c: Content-Disposition: form-data; name="**usuario**"

005c:

005e: **beltrano**

0068: --------------------------bfc967140684f75e

0094: Content-Disposition: form-data; name="**senha**"

00c2:

00c4: **teste**

00cb: --------------------------bfc967140684f75e--

Como pode ver, ele enviou as variáveis codificadas como “form-data”, e após os cabeçalhos HTTP.

A anatomia de um request, seja ele GET ou POST, contém:

<mètodo> URL

Cabeçalhos

Um exemplo de request GET:

0000: GET / HTTP/1.1

0010: Host: localhost:8080

0026: User-Agent: curl/7.68.0

003f: Accept: \*/\*

004c:

Os números à esquerda são contadores do curl, basta ignorá-los pois eles não fazem parte do request nem do response. Este request GET tem os cabeçalhos: Host, User-Agent e Accept.

Um exemplo de request POST:

0000: POST /usuarios/incluir HTTP/1.1

0021: Host: localhost:8080

0037: User-Agent: curl/7.68.0

0050: Accept: \*/\*

005d: Content-Length: 249

0072: Content-Type: multipart/form-data;

Este tem os cabeçalhos: Host, User-Agent, Accept, Content-Length e Content-Type.

O POST tem também o corpo do request, com as variáveis enviadas. No GET isso não existe, pois se quisermos passar variáveis temos que passar na URL (querystring ou extra path).

**Response**

Todo request deve ter uma response, que o servidor HTTP envia de volta. Esta response também tem cabeçalhos. Vejamos alguns exemplos:

GET:

0000: HTTP/1.0 200 OK

0000: Content-Type: text/html; charset=utf-8

0000: Content-Length: 16

0000: Server: Werkzeug/2.0.1 Python/3.8.5

0000: Date: Wed, 16 Jun 2021 19:58:45 GMT

Mais uma vez, ignore os números à esquerda, pois não fazem parte da response. Podemos ver o primeiro cabeçalho que contém o **http-status**, ou o código de estado da resposta, que, entre vários outros, pode ser:

* Respostas de informação (100-199),
* Respostas de **sucesso** (200-299),
* Redirecionamentos (300-399)
* Erros do cliente (400-499)
* Erros do servidor (500-599).

A seguir, aparecem os cabeçalhos de resposta:

* Content-Type: Tipo de dados da resposta;
* Content-Length: Tamanho da resposta;
* Server: Qual é o servidor que processou e gerou a resposta;
* Date: A data da resposta.

E, finalmente, vem a resposta, no formato especificado no Content-Type.

## Um servidor python com Flask

No capítulo anterior, executamos um servidor HTTP básico em python, utilizando a biblioteca Flask.

Este programa é o **servidorHTTP.py.** Vamos analisar sua estrutura. Para começar, vejamos as importações:

from flask import Flask

from flask import request

Da biblioteca flask, importamos as classes Flask e request, para podermos criar um objeto que representa nossa aplicação e podermos lidar com dados vindos de um request POST.

Depois, criamos o objeto que representa nossa aplicação servidora flask:

app = Flask(\_\_name\_\_)

Neste caso, criamos uma instância usando o nome do módulo. A partir dai, podemos usar o objeto app para lidarmos com o servidor HTTP.

A próxima coisa a fazer é declarar as rotas HTTP que o servidor atenderá. Uma rota HTTP é a junção de uma URI com o método HTTP permitido. Por exemplo:

@app.route("/",methods=['GET'])

def raiz():

return "<p>Raiz GET!</p>"

O símbolo “@” na frente do método app.route indica que ele é um **decorator**. Funciona como as anotações em Java. Um decorator adiciona funcionalidade a um objeto sem mudar sua estrutura. Declaramos as rotas anotando funções com um decorator. No exemplo acima, a função raiz() será invocada sempre que a URI do request for “/” e o método, GET.

No caso do GET, podemos passar variáveis para a função, como neste exemplo:

@app.route("/usuarios/<usuario>", methods=['GET'])

def usuario(usuario):

return f"<p>Usuario = {usuario}</p>"

Neste exemplo, “<usuario>” representa um valor passado como extra path no request, e a função recebe uma variável com o mesmo nome. Podemos usar essa variável na função que trata o request e gera a resposta.

Finalmente, temos uma rota POST:

@app.route("/usuarios/incluir",methods=['POST'])

def incluir():

return f"<p>{request.form['usuario']}, {request.form['senha']}</p>"

A diferença é que vamos pegar os dados do dicionário “form”. Como passamos os dados no formato “multipart/form-data”, as variáveis são criadas como atributos do **dictionary**.

## Clientes HTTP python com Requests

Outra biblioteca que vamos usar muito é a **requests**: <https://docs.python-requests.org/en/master/>

Com a biblioteca requests podemos criar clientes HTTP, acessando recursos disponíveis na Web. No arquivo “**exemplos\_http.zip**” você verá o programa **clienteHTTP.py**, que acessa o servidor que acabamos de criar:

import requests

def main():

# Request da raiz:

r = requests.get('http://localhost:8080')

print(r.text)

# Request passando usuario:

r = requests.get('http://localhost:8080/usuarios/fulano')

print(r.text)

# Request POST:

payload = {'usuario': 'beltrano', 'senha': 'xpto'}

r = requests.post('http://localhost:8080/usuarios/incluir', data=payload)

print(r.text)

if \_\_name\_\_=="\_\_main\_\_":

main()

A resposta deste programa é:

python3 clienteHTTP.py

<p>Raiz GET!</p>

<p>Usuario = fulano</p>

<p>beltrano, xpto</p>

Muito simples mesmo. Usamos o método correspondente ao método HTTP que desejamos enviar. No caso do POST com dados, é só passar um dictionary no argumento “data”.

Para ver a resposta, é só usar a propriedade “**text**” do objeto retornado pelo requests.

## Enviando e recebendo JSON

Até agora vimos apenas o formato “multpart/form-data”, que é bem comum em websites. Mas, no caso de servidores backend, o JSON é mais utilizado. Felizmente, o Flask e o Requests podem trabalhar muito bem com JSON. Vejamos o exemplo **servidorJSON.py** e o **clienteJSON.py**, ambos no arquivo **exemplos\_http.zip**.

Servidor JSON:

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

@app.route('/api/add\_message/<uuid>', methods=['GET', 'POST'])

def add\_message(uuid):

content = request.json

content\_type = request.headers.get('Content-Type')

print(content\_type)

print(content['mytext'])

return **jsonify**({"uuid":uuid})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host= '0.0.0.0', port=8080)

Cliente JSON:

import requests

url = 'http://localhost:8080/api/add\_message/1234'

dados = {"mytext":"lalala"}

res = requests.post(url, **json**=dados)

print(res.json())

Executamos o servidorJSON e depois o clienteJSON:

python3 clienteJSON.py

{'uuid': '1234'}

E na console do servidorJSON vemos que ele mostrou o conteúdo que foi enviado no arquivo JSON: “lalala”. A resposta do servidorJSON foi um objeto JSON contendo o extra path enviado.

Afinal, qual foi a diferença?

* No servidorJSON praticamente nada, a não ser a forma como pegamos os dados enviados no POST: **request.json** por que? Por que o clienteJSON enviou o cabeçalho Content-Type assim: **application/json**, daí para pegarmos o corpo do request precisamos usar a propriedade json. E usamos o método “**jsonify**” para serializar a resposta em formato JSON.
* No clienteJSON passamos um dictionary utilizando o argumento “json”: requests.post(url, json=dados). Isto fez com que ele enviasse o cabeçalho adequado ao servidor.

# RESTful services

**RE**presentational **S**tate **T**ransfer não é exatamente uma tecnologia ou protocolo, mas uma nova arquitetura de comunicação entre provedor e consumidor de serviços.

Ele utiliza os métodos (http): **GET**, **POST**, **PUT** e **DELETE** para transferir o estado de um objeto interno no servidor.

O REST é, por natureza, **stateless** ou seja o protocolo não mantém informações de estado. Apenas transfere o estado de objetos entre o produtor e o consumidor.

O método GET serve para obter o estado atual de um objeto no Servidor e o método POST para alterar este estado, adicionando informações. O método PUT serve para substituir totalmente o estado do objeto no servidor pelo que está sendo enviado e o método DELETE serve para eliminar um objeto.

Os objetos no REST são chamados de **RECURSOS** (resources) e possuem sua própria URI. Cada recurso pode ser acessado independentemente e expõe apenas os quatro métodos para alteração ou troca de estado.

## Recursos

Um recurso disponibilizado via REST possui uma única URI que o identifica. Por exemplo, "http://minhaloja.com/produtos/405934". Isto denota um produto, cujo identificador é 405934, e que pode ser acessado unicamente através dessa URI. É algo como um "**permalink**"... Uma vez que publicamos uma interface REST, devemos manter a sua estabilidade, pois esta é uma das bases de todo o estilo arquitetural REST.

Podemos expor vários recursos, de diferentes tipos, para o mundo. Nossa interface pode até permitir que os usuários criem novos recursos, alterem os já existentes ou mesmo os apaguem. Tudo depende da exposição que fazemos em nossa API, com relação aos métodos HTTP.

## API REST

Uma API REST é o conjunto formado por:

* Formato de URI;
* Parâmetros;
* Métodos disponíveis.

Uma API REST é composta por rotas de recursos providos por servidores **Stateless**. Stateless significa “sem armazenamento de estado”, ou seja, o Servidor não se “lembra” de dados entre dois acessos diferentes.

### RESTful-URI x Non RESTful-URI

Existem alguns princípios que devem guiar a criação de uma API REST, usando RESTful URIs. Vejamos algumas características e outros conceitos relevantes:

1. A URI deve ser **curta**;
2. Deve seguir uma **estrutura de árvore**, permitindo o "drill-down" e a sumarização. Por exemplo, na URI "/produtos/405934", se eu remover o código do produto, deve ser possível obter uma lista de produtos, e, se eu acrescentar: ".../foto" à URI, deve ser possível ver a foto do produto "405934";
3. Deve usar **substantivos** e não verbos;
4. Deve retornar um "**Content-type**" com base nos **HEADERS** do Request, e não em informação da própria URI;
5. Só usar **querystrings** para filtrar listagens, por exemplo: "/produtos?tipo=1", e não para identificar produtos, por exemplo, ao invés de "/produtos/id=405934", devemos usar: /produtos/405934";

### JSON

O JSON (Javascript Object Notation) é um formato de troca de dados que também serve para serializar objetos. Um objeto na memória pode ser transformado em um formato independente de linguagem e ser transportado por qualquer meio até o seu destino, de maneira semelhante ao que é feito com o XML, pelo framework JAX-WS.

Utilizamos o JSON para troca de dados entre funções de APIs existentes na Web e é muito utilizado em Mashups.

Eis alguns casos que requerem o uso de JSON como interface de troca de dados:

* Google: Usa para várias APIs, como Google Places, por exemplo;
* Flickr: Usa para buscar e compartilhar fotos;
* Twitter: Usa para acessar o microblog;
* Yahoo: Várias APIs (busca web, busca imagens etc);

É uma alternativa muito utilizada como contrapartida ao XML na troca de mensagens entre o cliente e a API.

Um Objeto JSON é definido como um conjunto de nomes e valores, separados por dois pontos, e delimitados por vírgulas. Este conjunto de nomes/valores representa as propriedades do objeto sendo definido. O objeto todo fica delimitado por chaves. Veja a sintaxe básica:

objetoJSON = {nome1:valor1[,nome2:valor2[,...,nomeX:valorX]]}

Vejamos um exemplo simples:

var objeto = {"nome" : "Fulano", "endereco" : "rua x numero y", "codigo" : 12345};

Neste exemplo estamos definindo um objeto contendo as propriedades: "nome", "endereço" e "codigo". E podemos acessar essas propriedades de forma muito simples: "alert(objeto.nome)".

Os objetos JSON podem ter funções agregadas (métodos), por exemplo:

{

"nome": "fulano",

"funcao": "function(){ alert("ok"); }"

}

Existem duas vantagens imediatas de usar JSON:

1. É fácil de entender;

2. Representa bem a maioria das estruturas de dados;

Os valores das propriedades do JSON não precisam necessariamente ser strings ou números. Podemos ter Arrays também. Para isto, delimitamos os elementos do array entre colchetes "[" e "]":

{nome : [valor1, valor2, ... , valorX]}

Por exemplo:

var aluno = {"nome":"fulano", "notas": [10,5,7]};

Neste caso definimos uma propriedade "notas" que é um array contendo três elementos do tipo numérico.

O valor de uma propriedade pode ser também um objeto JSON ou Javascript:

{nome : [

{<objeto JSON1>}, {<objeto JSON2},...,{<objeto JSON3>}]};

Confuso? Não. Veja um exemplo prático:

var pessoas = {"dados":[

{"nome":"maria","email":"maria@teste.org"},

{"nome":"pedro","email":"petro@xpto.com"},

{"nome":"paulo","email":"paulo@abcd.com"}

]};

A propriedade "dados" é um array contendo três elementos que, por si só, são objetos JSON.

### Métodos HTTP e request REST

Uma API REST é um **CRUD** (Create-Read-Update-Delete) de recursos não-http, utilizando os métodos HTTP para isso.

Se você quer apenas invocar uma função remotamente, há vários protocolos e arquiteturas para isto, como RPC – Remote Procedure Call (<https://en.wikipedia.org/wiki/Remote_procedure_call>).

Uma API REST é composta por “rotas” padronizadas e recursos identificados por URI acessadas de modo **Stateless**. Se você tiver uma chamada com URL: “calcularJuros”, ou então tiver que fazer várias chamadas para realizar uma operação, com manutenção de estado entre elas, então não tem uma API REST.

Um request REST é **ATÔMICO** e **Stateless**, ou seja, é executado de uma só vez e o servidor não guarda estado entre os requests.

**Semântica dos métodos HTTP**

Em REST, a semântica da operação é dada pelo Método HTTP utilizado:

* **GET**: Obter recurso (ou recursos). É um método **idempotente**, que usa a URI para obter o recurso. É para ser idempotente;
* **PUT**: Criar ou alterar um recurso. Você passa a URI e no corpo do request vai a representação do recurso (JSON, XML etc). Há controvérsias, mas a princípio seria idempotente;
* **POST**: Criar um recurso, sem passar identificador, com a representação no corpo do request. No retorno, deve vir a URI do novo recurso, dentro dos HEADERS. Não é idempotente;
* **DELETE**: Remover um recurso do backend, passando a URI. Deveria ser idempotente;

Sabe o que significa **idempotência**?

<https://tools.ietf.org/html/rfc7231#section-4.2.2>

*“A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. ”*

Um método idempotente é aquele que o efeito de múltiplos requests com o mesmo método é o mesmo que um único request daquele método, sem efeitos colaterais no **backend**. Tem gente que só usa **GET** para tudo, tem gente que só usa **POST** para tudo, inclusive para eliminar recursos! Já vi gente utilizando GET para alterar, eliminar e criar registros!

Use os métodos HTTP como parte da semântica CRUD da sua API REST.