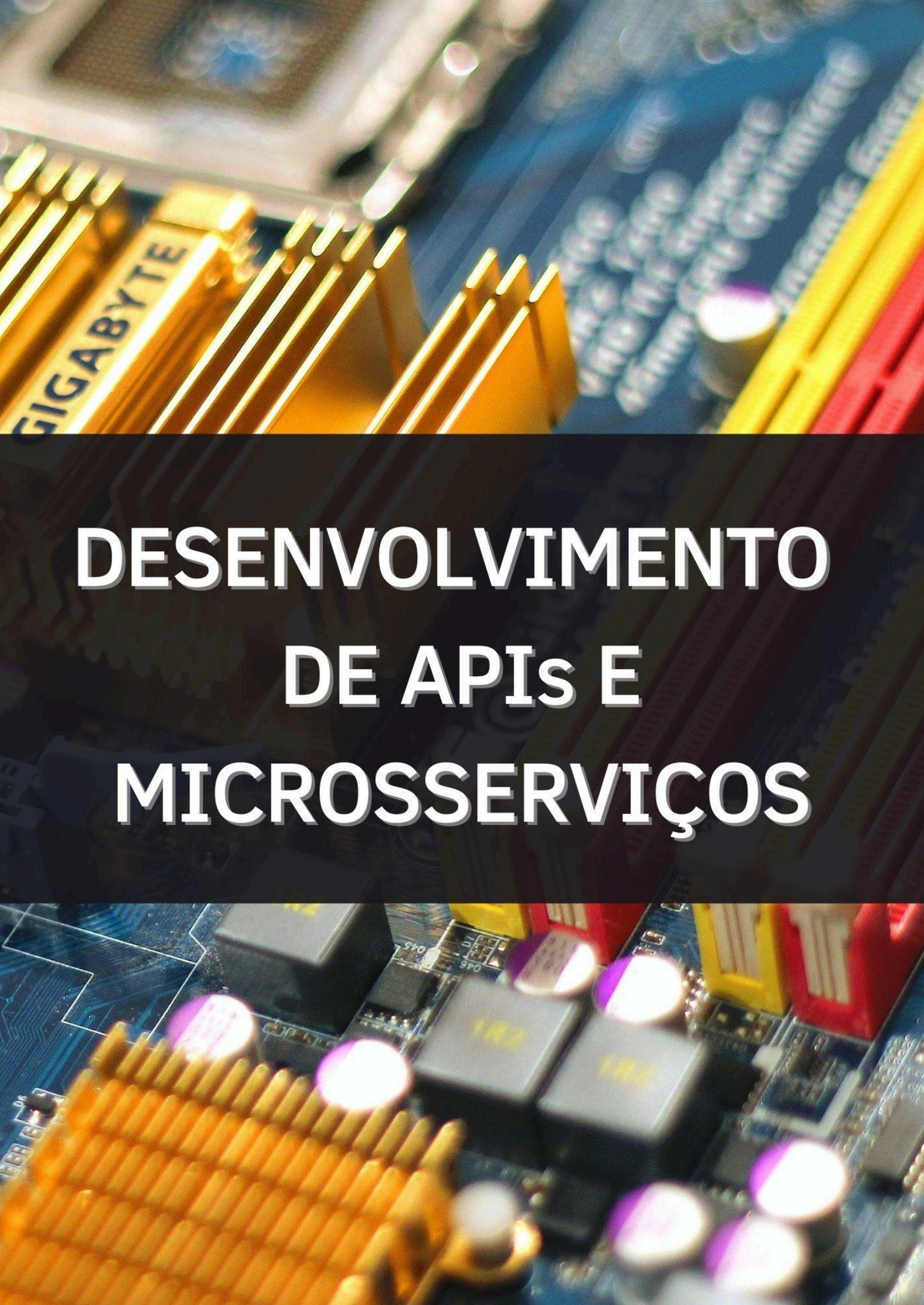
DESENVOLVIMENTO
DE APIs E
MICROSSERVIÇOS

8

JSON, requisições e respostas no Flask

Victor Williams Stafusa da Silva

Resumo

Começamos a desenvolver com o Flask, mas muitos recursos ainda são necessários, tais como lidar com requisições que utilizem POST, PUT ou DELETE, produzir e consumir dados no formato JSON, realizar redirecionamentos, ler a query-string e os cabeçalhos da requisição. Nesta unidade, você verá como utilizar todas essas funcionalidades que o Flask oferece.

8.1. Objetivos deste capítulo

- Responder requisições com corpo da resposta em formato JSON.
- Rotear funções no Flask utilizando os verbos POST, PUT e DELETE.
- Ler o conteúdo do corpo de uma requisição POST ou PUT.
- Ler informações da *query string* com o Flask.
- Ler cabeçalhos da requisição HTTP com o Flask.
- Definir cabeçalhos da resposta HTTP com o Flask.
- Realizar redirecionamentos.

8.2. Respondendo JSON

Até o momento, o que aprendemos com Flask tem devolvido apenas respostas em formato de texto puro. No entanto, esse formato é difícil de se trabalhar em uma aplicação cliente desenvolvida com requests. Obviamente, não há nada que impeça de programarmos no servidor alguma funcionalidade que o faça produzir uma resposta em formato JSON a partir de um dicionário e a envie em formato texto, e é exatamente esse um dos recursos que o Flask oferece.

Uma forma simples de se devolver um JSON, é simplesmente retornar um dicionário, ao invés de retornar uma *string*. O Flask também já vai alterar o conteúdo do cabeçalho Content-Type da resposta para `application/json`, o que permitirá que aplicativos clientes (inclusive navegadores) identifiquem que a resposta HTTP contém dados em formato JSON. Por exemplo:

Codificação 8.1. JSON

```
from flask import Flask
app = Flask(__name__)

cores_frutas = {
    "morango": "vermelho",
    "uva": "roxo",
    "banana": "amarelo",
    "abacaxi": "amarelo",
    "limão": "verde",
    "framboesa": "vermelho",
    "melancia": "vermelho",
    "laranja": "laranja",
    "abacate": "verde"
}

@app.route("/frutas/todas")
def frutas():
    return cores_frutas

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

Ao navegar na rota `http://localhost:5000/frutas/todas`, o resultado será um JSON contendo todas as frutas, em formato de dicionário.

No entanto, nem todo o JSON tem necessariamente o formato de um dicionário. Em algumas ocasiões, pode ser necessário se devolver uma lista em formato JSON, ou até mesmo um número, uma string, ou um dos literais `true`, `false` ou `null`. Para casos como esses, utilizamos a função `jsonify` do Flask (não se esqueça de importá-la). Veja alguns exemplos no código que se segue:

Codificação 8.2. JSON - exemplos

```
from flask import Flask, jsonify
app = Flask(__name__)

@app.route("/uma-lista")
def uma_lista():
    return jsonify([0, 1, 2, 3])

@app.route("/bom-dia")
def bom_dia():
    return jsonify("Bom dia")

@app.route("/numero-da-sorte")
def numero_da_sorte():
    return jsonify(42)

@app.route("/nada-a-dizer")
def nada():
    return jsonify(None)

@app.route("/sim")
```

```
def sim():
    return jsonify(True)

# Continua na próxima página.
# Continuação da página anterior.

@app.route("/nao")
def nao():
    return jsonify(False)

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

Ao acessar qualquer uma das rotas desse código, o resultado será um JSON, mesmo que nenhum deles seja um dicionário no formato JSON.

8.3. Utilizando o POST no Flask

Até o momento, o que aprendemos com Flask tem utilizado apenas o verbo GET para acessar rotas. No entanto, para que desenvolvamos aplicações, isso claramente não basta, uma vez que o GET não deve alterar o estado da aplicação e isso é algo que se faz necessário para o desenvolvimento de diversas funcionalidades em aplicações típicas. Assim sendo, precisamos utilizar o verbo POST também.

Entretanto, relembremos que requisições POST têm corpo na requisição. Para recuperar o conteúdo desse corpo, fazemos uso do objeto `request` e a partir dele, pegamos o corpo da requisição como JSON com `request.json` (não se esqueça de importá-lo). Observe um exemplo:

Codificação 8.3. Exemplo de requisição POST

```
from flask import Flask, request, jsonify
app = Flask(__name__)

pessoas = [
    {"nome": "Paulo", "sexo": "M", "cabelo": "loiro"},
    {"nome": "Maria", "sexo": "F", "cabelo": "preto"},
    {"nome": "Fernanda", "sexo": "F", "cabelo": "ruivo"},
    {"nome": "José", "sexo": "M", "cabelo": "careca"}
]

@app.route("/pessoa", methods = ["POST"])
def cadastrar():
    pessoa = request.json
    pessoas.append(pessoa)
    return jsonify(pessoas)

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

Perceba que a função cadastrar lê o corpo da requisição com `request.json`, trazendo então os dados da pessoa a ser cadastrada como um dicionário e

acrescentado-o à lista `pessoas`. Por fim, toda essa lista é devolvida na resposta HTTP graças à função `jsonify`.

Ainda neste exemplo, perceba o parâmetro `methods` definido na rota. Ele define quais são os verbos HTTP (pode haver mais de um) que devem ser roteados no padrão de URL definido na rota. Quando ausente, `["GET"]` é definido por padrão.

Para testar este exemplo, considere o seguinte programa cliente:

Codificação 8.4. Programa cliente

```
import requests

nome = input("Digite o nome: ")
sexo = input("Digite o sexo: ")
cabelo = input("Digite a cor do cabelo: ")
dados = {"nome": nome, "sexo": sexo, "cabelo": cabelo}

x = requests.post("http://localhost:5000/pessoa", json = dados)
if x.status_code != 200:
    print(f"[{x.status_code}] {x.text}")
else:
    print(x.text)
```

Fonte: do autor, 2021

Ao executar o código do cliente (duas vezes, para termos certeza), eis uma possível saída:

Codificação 8.5. Execução do código do cliente

```
C:\DAM\Capitulo_08>python cliente_pessoas.py
Digite o nome: Amanda
Digite o sexo: F
Digite a cor do cabelo: loiro
[{"cabelo":"loiro","nome":"Paulo","sexo":"M"},{"cabelo":"preto","nome":
"Maria","sexo":"F"},{"cabelo":"ruivo","nome":"Fernanda","sexo":"F"},{"c
abelo":"careca","nome":"Jos\u00e9","sexo":"M"},{"cabelo":"loiro","nome"
:"Amanda","sexo":"F"}]


C:\DAM\Capitulo_08>python cliente_pessoas.py
Digite o nome: Rodrigo
Digite o sexo: M
Digite a cor do cabelo: castanho
[{"cabelo":"loiro","nome":"Paulo","sexo":"M"},{"cabelo":"preto","nome":
"Maria","sexo":"F"},{"cabelo":"ruivo","nome":"Fernanda","sexo":"F"},{"c
abelo":"careca","nome":"Jos\u00e9","sexo":"M"},{"cabelo":"loiro","nome"
:"Amanda","sexo":"F"},{"cabelo":"castanho","nome":"Rodrigo","sexo":"M"}
]


C:\DAM\Capitulo_08>
```

Fonte: do autor, 2021

Atenção:

1. Não confunda o objeto `request` do Flask com a biblioteca requests (que tem um "s" no final), ainda mais se estiver utilizando os dois no mesmo projeto ou até dentro de um mesmo arquivo de código-fonte.
2. Note que o parâmetro a colocar na rota se chama `methods`, no plural. E é por isso que ele deve receber uma lista com o nome dos verbos HTTP, e nunca apenas uma *string* singela. Logo, usar `methods = ["POST", "GET"]` ou `methods = ["POST"]` está correto, mas usar `method = "POST"` está errado.
3. Tentar acessar uma rota definida com `methods = ["POST"]` com GET, tal como o navegador realiza ao visitar uma URL, resultará em um erro 405. Afinal de contas, a rota estará definida para aceitar apenas POST, o que implica em recusar o GET. Nesse caso, utilize `methods = ["POST", "GET"]` para que a requisição seja aceita.

8.4. Validando os dados

Prestando um pouco de atenção no exemplo anterior, que tem a função de cadastrar pessoas, talvez você possa ter se perguntado: E se o conteúdo da requisição não for um JSON? E se o conteúdo da requisição for um JSON mal formado? E se, mesmo que o conteúdo seja um JSON bem formado, se as informações que nele estiverem não corresponderem aos dados de uma pessoa? Bem, vamos criar um programa para testar essas possibilidades:

Codificação 8.6. Validando os dados

```
import requests

url = "http://localhost:5000/pessoa"

x = requests.post(url, data = "XXX")
if x.status_code != 200:
    print(f"[{x.status_code}] {x.text}")
else:
    print(x.text)

h = {"Content-Type": "application/json"}
y = requests.get(url, data = "XXX", headers = h)
if y.status_code != 200:
    print(f"[{y.status_code}] {y.text}")
else:
    print(y.text)

z = requests.get(url, json = {"foo": "bar"})
if z.status_code != 200:
    print(f"[{z.status_code}] {z.text}")
else:
    print(z.text)
```

Fonte: do autor, 2021

Ao executar isso, eis o resultado:

Codificação 8.7. Validando os dados 2

```
C:\DAM\Capitulo_08>python bomba.py
[{"cabelo":"loiro","nome":"Paulo","sexo":"M"},{"cabelo":"preto","nome":
"Maria","sexo":"F"},{"cabelo":"ruivo","nome":"Fernanda","sexo":"F"},{"c
abelo":"careca","nome":"Jos\u00e9","sexo":"M"},{"cabelo":"loiro","nome"
:"Amanda","sexo":"F"},{"cabelo":"castanho","nome":"Rodrigo","sexo":"M"}
,null]

[400] <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>400 Bad Request</title>
<h1>Bad Request</h1>
<p>The browser (or proxy) sent a request that this server could not
understand.</p>

[{"cabelo":"loiro","nome":"Paulo","sexo":"M"},{"cabelo":"preto","nome":
"Maria","sexo":"F"},{"cabelo":"ruivo","nome":"Fernanda","sexo":"F"},{"c
abelo":"careca","nome":"Jos\u00e9","sexo":"M"},{"cabelo":"loiro","nome"
:"Amanda","sexo":"F"},{"cabelo":"castanho","nome":"Rodrigo","sexo":"M"}
,null,{"foo":"bar"}]


C:\DAM\Capitulo_08>
```

Fonte: do autor, 2021

A requisição com o JSON mal formado (a segunda), resultou num erro 400. Se você tiver interesse em investigar mais a fundo, a chamada à propriedade `request.json` internamente invoca o método `request.get_json()`, e este método lança uma exceção chamada `BadRequest` quando encontra um JSON mal formado. O próprio Flask trata esta exceção enviando uma resposta com uma página de erro padrão com o código 400. Saber disso é importante, pois o método `cadastrar` foi invocado e só executou parcialmente, o que significa que quando você estiver criando suas aplicações com Flask, terá que se atentar a possíveis problemas ou efeitos colaterais ocasionados pela execução parcial da função. Além disso, a depender do tratamento de erros que você estiver desenvolvendo, poderá capturar a exceção `BadRequest`, seja de forma proposital ou de forma acidental, e então ter que tratá-la ou relançá-la.

No entanto, aqui não precisamos nos preocupar muito com o segundo caso, e sim com o primeiro e com o terceiro. No primeiro caso, um `null` acabou aparecendo no JSON, o que significa que `None` foi adicionado à lista de pessoas e isso ocorreu porque a propriedade `request.json` devolve `None` quando a requisição não é do tipo JSON. No terceiro caso, um conjunto de dados que não corresponde à uma pessoa foi adicionado à lista de pessoas, pois não há nada que verifica se o JSON, ainda que bem formado, tenha a estrutura correta. Precisamos realizar o tratamento adequado desses casos, e para isso, vamos alterar a função `cadastrar` e adicionar uma outra função auxiliar:

Codificação 8.8. Validando os dados 3

```
from werkzeug.exceptions import BadRequest

def pessoa_ok(dic):
    return type(dic) == dict \
        and len(dic) == 3 \
        and "nome" in dic \
```

```
        and "sexo" in dic \
        and "cabelo" in dic \
        and type(dic["nome"]) == str \
        and dic["sexo"] in ["M", "F"] \
        and type(dic["cabelo"]) == str

@app.route("/pessoa", methods = ["POST"])
def cadastrar():
    pessoa = request.json
    if not pessoa_ok(pessoa):
        raise BadRequest
    pessoas.append(pessoa)
    return jsonify(pessoas)

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

A ideia aqui é que se a estrutura do JSON não for bem formada (e isso inclui o caso do `None`), então a função `pessoa_ok` retorna `False`. Caso contrário (se a estrutura estiver bem formada), então retorna `True`. Para verificarmos se o JSON é bem formado, temos que verificar se o que dele foi obtido é de fato um dicionário, se todos os pares chave-valor que deveriam estar presentes realmente estão presentes, se não existe nenhum par chave-valor a mais e se todos os valores são dos tipos corretos.

Atenção:

Quando você estiver desenvolvendo as suas aplicações, valide tudo o que for necessário no lado do servidor para evitar que requisições inválidas (incluindo aquelas possivelmente forjadas de forma maliciosa por um adversário, mas não somente essas) venham a comprometer a estabilidade, a consistência dos dados e/ou a segurança do seu servidor. Note que muitas vezes, especialmente quando trabalha-se com dados estruturados de formas bem complexas ou que só são válidos em determinadas situações e/ou contextos, que realizar a devida validação pode ser algo bem trabalhoso.

Reiniciamos então o servidor (para nos livrarmos do lixo que foi colocado na lista `pessoas`) e executamos o teste novamente. Eis a saída conforme o esperado, evidenciando que todos os casos inválidos são rejeitados:

Codificação 8.9. Validando os dados 4

```
C:\DAM\Capitulo_08>python bomba.py
[400] <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>400 Bad Request</title>
<h1>Bad Request</h1>
<p>The browser (or proxy) sent a request that this server could not
understand.</p>

[400] <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>400 Bad Request</title>
<h1>Bad Request</h1>
<p>The browser (or proxy) sent a request that this server could not
understand.</p>
```

```
[400] <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>400 Bad Request</title>
<h1>Bad Request</h1>
<p>The browser (or proxy) sent a request that this server could not
understand.</p>


C:\DAM\Capitulo_08>
```

Fonte: do autor, 2021

8.5. Lendo dados da *query string*

Requisições do tipo GET (e não apenas essas) podem passar informações ao servidor não apenas por meio do *path*, mas também por meio da *query string*. Por exemplo, na URL `http://localhost:5000/produtos?pg=1&filtro=Notebook`, temos que o trecho `pg=1&filtro=Notebook` é uma *query string* que especifica um conjunto de pares chave-valor que faz parte da URL.

Atenção:

Embora seja incomum, é permitido que existam chaves repetidas na *query string*, tal como por exemplo, `mes=1&mes=2&mes=5&tipo=cliente`. Por tal razão, quando você for procurar pelo valor correspondente a uma determinada chave, esteja preparado para situações onde haja mais do que um valor como resultado.

A forma principal de se ler os dados da *query string* é pelo uso de `request.args`, que é do tipo `MultiDict`. O `MultiDict` pode ser acessado como um dicionário, mas também disponibiliza o método `get` com 1 ou com 2 parâmetros e o método `getlist`. Por exemplo:

Codificação 8.10. Lendo dados da *query string*

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/exemplo1")
def exemplo1():
    pagina = request.args["pg"]
    return f"O valor lido foi {pagina}."

@app.route("/exemplo2")
def exemplo2():
    pagina = request.args.get("pg")
    return f"O valor lido foi {pagina}."

@app.route("/exemplo3")
def exemplo3():
    pagina = request.args.get("pg", "vazio")
    return f"O valor lido foi {pagina}."

@app.route("/exemplo4")
def exemplo4():
    pagina = request.args.getlist("pg")
    return f"O valor lido foi {pagina}."
```

```
if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

E então, podemos acessar várias URLs com diferentes *query strings* e ver quais são os resultados, tal como exibido na Tabela 8.1:

Tabela 8.1. Consulta nas URLs com *query strings* e respectivos resultados.

URL	Resultado
`http://localhost:5000/exemplo1`	Bad Request (erro 400)
`http://localhost:5000/exemplo1?pg=123`	O valor lido foi 123.
`http://localhost:5000/exemplo1?pg=123&pg=456`	O valor lido foi 123.
`http://localhost:5000/exemplo2`	O valor lido foi None.
`http://localhost:5000/exemplo2?pg=123`	O valor lido foi 123.
`http://localhost:5000/exemplo2?pg=123&pg=456`	O valor lido foi 123.
`http://localhost:5000/exemplo3`	O valor lido foi vazio.
`http://localhost:5000/exemplo3?pg=123`	O valor lido foi 123.
`http://localhost:5000/exemplo3?pg=123&pg=456`	O valor lido foi 123.
`http://localhost:5000/exemplo4`	O valor lido foi [].
`http://localhost:5000/exemplo4?pg=123`	O valor lido foi ['123'].
`http://localhost:5000/exemplo4?pg=123&pg=456`	O valor lido foi ['123', '456'].

Fonte: do autor, 2021

Assim sendo, podemos concluir que ler o `request.args` como um dicionário poderá causar um erro 400 se a chave não existir na *query string*. Já ao utilizar `request.args.get(chave)`, `None` é retornado nesse caso. Ao utilizar `request.args.get(chave, padrao)`, o valor `padrao` é retornado se a chave não existir. Todas essas formas retornarão o valor quando esse existir. Se houver mais de um valor para a mesma chave, todas essas formas retornarão apenas o primeiro valor.

Para obter uma lista de valores, nos casos onde podem existir vários valores para uma mesma chave, usa-se o método `request.args.getlist(chave)`, que trará uma lista com todos os valores. Se a chave não for encontrada, uma lista vazia é devolvida.

8.6. Implementando o CRUD

Já temos rotas utilizando os verbos GET e POST. Podemos implementar o processo de CRUD completo se também utilizarmos PUT e DELETE:

Codificação 8.11. Implementando o CRUD

```
from flask import Flask, request, jsonify
from werkzeug.exceptions import BadRequest, NotFound
app = Flask(__name__)

n = 1 # Apenas para exemplificar, isto não é thread-safe.
```

```
pessoas = dict()

def pessoa_ok(dic):
    return type(dic) == dict \
        and len(dic) == 3 \
        and "nome" in dic \
        and "sexo" in dic \
        and "cabelo" in dic \
        and type(dic["nome"]) == str \
        and dic["sexo"] in ["M", "F"] \
        and type(dic["cabelo"]) == str

@app.route("/pessoa", methods = ["POST"])
def cadastrar():
    pessoa = request.json
    if not pessoa_ok(pessoa):
        raise BadRequest
    pessoas[n] = pessoa
    n += 1
    return pessoas

@app.route("/pessoa/<int:id_pessoa>", methods = ["PUT"])
def atualizar(id_pessoa):
    pessoa = request.json
    if not pessoa_ok(pessoa):
        raise BadRequest
    if id_pessoa not in pessoas:
        raise NotFound
    pessoas[id_pessoa] = pessoa
    return pessoas

# Continua na próxima página.
# Continuação da página anterior.

@app.route("/pessoa", methods = ["GET"])
def listar():
    return pessoas

@app.route("/pessoa/<int:id_pessoa>", methods = ["GET"])
def selecionar(id_pessoa):
    if id_pessoa not in pessoas:
        raise NotFound
    return jsonify(pessoas[id_pessoa])

@app.route("/pessoa/<int:id_pessoa>", methods = ["DELETE"])
def deletar():
    if id_pessoa in pessoas:
        del pessoas[id_pessoa]
    return pessoas

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

No código acima, podemos notar que temos as operações básicas para cadastrar, atualizar e deletar o cadastro de uma pessoa. Também temos as operações de listar todas as pessoas e obter os dados de uma pessoa em especial. Isso tudo incluindo a validação dos dados e trabalhando com o formato JSON tanto na entrada como na saída. Note também que assim como usamos a classe `BadRequest` provida pelo Werkzeug para causar um erro 400, também usamos a classe `NotFound` para causar um erro 404.

Algumas pessoas podem ser tentadas a juntar funções que usam a mesma rota, mas verbos HTTP diferentes num mesmo caminho ao utilizar o `request.method` para inspecionar qual foi o verbo HTTP utilizado, como por exemplo, neste código que se segue:

Codificação 8.12. Implementando o CRUD 2

```
@app.route("/pessoa/<int:id_pessoa>", methods=["PUT","GET","DELETE"])
def faz_tudo(id_pessoa):
    if request.method == "PUT":
        pessoa = request.json
        if not pessoa_ok(pessoa):
            raise BadRequest
        if id_pessoa not in pessoas:
            raise NotFound
        pessoas[id_pessoa] = pessoa
        return pessoas
    if request.method == "GET":
        if id_pessoa not in pessoas:
            raise NotFound
        return jsonify(pessoas[id_pessoa])
    if request.method == "DELETE":
        if id_pessoa in pessoas:
            del pessoas[id_pessoa]
        return pessoas
```

Fonte: do autor, 2021

Atenção:

Não faça isso! Isso é uma má prática de programação, e trata-se de um exemplo de acoplamento de controle. Nesse caso, ao invés de definir funções para tratar cada funcionalidade separadamente, junta-se diversas funcionalidades que operam de formas completamente diferentes numa única função que usa uma sequência interna de `if`s com base em algum parâmetro ou em alguma *flag* para separar os casos novamente, resultando em um código mais amarrado, mais difícil de testar, mais difícil de dar manutenção e mais fácil de quebrar.

Além disso, se o Flask / Werzeug já provê um bom mecanismo para separar as rotas satisfatoriamente, não há porque subutilizar e/ou subverter esse mecanismo para depois reimplementar essa mesma separação de uma forma manual que nenhuma vantagem traz sobre o mecanismo já provido. Enfim, isso trata-se de um caso de "reinventar a roda quadrada".

8.7. Outras informações da requisição

No exemplo anterior, embora tenha sido um exemplo do que não fazer, usamos `request.method` para obter o verbo HTTP utilizado na requisição. Talvez você pondere o que mais podemos obter a partir do objeto `request`. Há diversas informações que estão disponíveis no objeto `request`, as principais são essas:

- `request.host` informa qual é o *hostname* utilizado na requisição, incluindo a porta TCP caso não seja a padrão.
- `request.host_url` informa qual é a URL utilizada para realizar a requisição.
- `request.is_secure` tem o valor `True` se o protocolo utilizado é seguro (HTTPS ao invés de HTTP ou WSS ao invés de WS).
- `request.is_json` tem o valor `True` se o conteúdo da requisição parece conter um JSON.
- `request.path` informa qual é o caminho (*path*) utilizado na requisição. É esse caminho que o Flask / Werkzeug utilizam para encontrar a rota e a função do seu programa que deve ser chamada.
- `request.method` informa qual é o verbo HTTP utilizado na requisição.
- `request.root_path` informa qual é o domínio utilizado na requisição.
- `request.origin` informa qual é *host* que originou a requisição. Isso é útil para implementar-se validações de CORS, por exemplo.
- `request.referrer` informa qual é a página a partir da qual a requisição foi realizada. Útil para proteção de *links* e para impedir ataques de CSRF.
- `request.remote_addr` informa qual é o endereço IP do cliente.
- `request.query_string` informa qual é a *query string* completa.
- `request.scheme` informa qual é o protocolo utilizado na requisição.
- `request.url` informa qual é a URL completa utilizada na requisição, incluindo protocolo, domínio, caminho e *query string*.
- `request.base_url` informa qual é a URL sem a *query string*.
- `request.full_path` informa qual é o caminho (*path*) incluindo a *query string*.
- `request.user_agent.browser` informa qual é o navegador de *internet* ou qual é o tipo de cliente que está realizando a requisição, se for possível determinar.
- `request.user_agent.language` informa qual é a língua utilizada pelo navegador de *internet* ou cliente que está realizando a requisição, se for possível determinar.
- `request.user_agent.version` informa qual é a versão do navegador de *internet* ou cliente que está realizando a requisição, se for possível determinar.
- `request.user_agent.platform` informa qual é o sistema operacional do navegador de *internet* ou cliente que está realizando a requisição, se for possível determinar.
- `request.user_agent.string` contém a informação bruta acerca do que foi informado como navegador de *internet* ou cliente que está realizando a requisição.
- `request.data` provê uma sequência de *bytes* que pode ser utilizada para ler o conteúdo bruto da requisição bruto caso ele não tenha uma *string* ou um JSON.
- `request.content_type` informa qual é o tipo de dado da requisição.

- `request.mimetype` informa qual é o tipo de dado da requisição, mas sem considerar o *charset* ou outros eventuais parâmetros do tipo MIME.
- `request.cookies` é um dicionário (tanto as chaves quanto os valores são *strings*) que informa quais são os *cookies* existentes na requisição. Veremos mais sobre *cookies* no capítulo 9.
- `request.headers` é um dicionário (tanto as chaves quanto os valores são *strings*) que informa quais são os cabeçalhos da requisição (todos os cabeçalhos, inclusive os que podem ser mais convenientemente obtidos por outras propriedades do objeto `request`).

8.8. Manipulando a resposta HTTP

Já temos o poder de realizar diversas operações com o Flask e fazer praticamente qualquer coisa com a requisição. No entanto, ainda falta podermos manipular a resposta HTTP de uma forma mais fina do que simplesmente definir o conteúdo dela e um código de *status* opcional, tal como por exemplo, definir cabeçalhos personalizados na resposta HTTP. Um outro caso, é para realizar redirecionamentos. Para isso, podemos utilizar as funções `make_response` e `redirect`:

Codificação 8.13. Manipulando a resposta HTTP

```
from flask import Flask, make_response, jsonify, redirect
app = Flask(__name__)

@app.route("/exemplo5")
def exemplo5():
    x = jsonify(["pêra", "maçã", "laranja"])
    r = make_response(x)
    r.headers["X-tipo-dado"] = "frutas"
    r.status_code = 275
    return r

@app.route("/exemplo6")
def exemplo6():
    # Se code fosse omitido, o 302 seria utilizado por padrão.
    return redirect("/exemplo5", code = 305)

# Continua na próxima página.
# Continuação da página anterior.

if __name__ == "__main__":
    app.run(host = "0.0.0.0", port = 5000)
```

Fonte: do autor, 2021

Assim como o objeto `request` tem diversas propriedades, o objeto `response` (que é retornado pelo `make_response`) também tem algumas, e a maioria delas podem ser alteradas conforme a necessidade do programador. Dentre elas podemos destacar:

- `response.status_code` é o código HTTP da resposta.
- `response.content_type` é o tipo de dado da resposta.
- `response.mimetype` é o tipo de dado da resposta, mas sem considerar o *charset* ou outros eventuais parâmetros do tipo MIME.

- `response.is_json` tem o valor `True` se o conteúdo da resposta parece conter um JSON. Essa propriedade é somente-leitura.
- `response.data` tem o conteúdo da resposta, que pode ser uma string ou, no caso de uma resposta em formato binário, pode ser do tipo *bytes*. Definir essa propriedade pode ser muito útil para definir o conteúdo da resposta caso não seja possível ou desejável fazê-lo por outros meios.
- `response.headers` é um dicionário (tanto as chaves quanto os valores são *strings*) que informa quais são os cabeçalhos da resposta (todos os cabeçalhos, inclusive os que podem ser mais convenientemente obtidos por outras propriedades do objeto `response`).

Vamos praticar?

Observe que com o que foi apresentado até aqui, já temos o suficiente para criarmos um *back-end* completo com Flask. Combinando-se a possibilidade de atender requisições GET e POST em diversas rotas distintas no Flask com entrada e saída em formato JSON com o que você já aprendeu nas unidades anteriores acerca de *requests* e de banco de dados, um sistema de CRUD completo com um banco de dados real já é possível de ser feito.

Entretanto, construir um *back-end* completo em Flask com essa abordagem ainda poderá causar dificuldades acerca da modularização, e para isso o Flask tem a funcionalidade de Blueprints que serão apresentados no capítulo 10. Há ainda a questão da produção do *front-end*, para o qual o Jinja2, que será apresentado no capítulo 9, pode ser utilizado.

Recomendamos que pratique com os seguintes exercícios:

- Tente implementar o CRUD de alguma entidade de negócio de um sistema, tal como funcionário, cliente, pedido ou fornecedor.
- Explore o objeto `request` e suas diversas propriedades. Tente descobrir quais são as informações que você pode obter desse objeto e como pode utilizá-la para desenvolver programas interessantes.
- Tente criar uma função de validação do JSON mais robusta e geral do que a provida no código.
- Ao invés de utilizar dicionários para representar entidades de negócio, tente modelá-los como classes.
- Utilize o banco de dados ao invés de apenas um dicionário na memória para realizar o CRUD.

Referências

Orenstein, Ben. **Types of Coupling.** 2019. Disponível em: <https://thoughtbot.com/blog/types-of-coupling>. Acesso em 10 ago. 2021.

The Pallets Projects. **Data Structures.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x/datastructures/>. Acesso em 10 jul. 2021.

The Pallets Projects. **API.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x/api>. Acesso em 10 jul. 2021.

The Pallets Projects. **Handling Application Errors.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x/errorhandling/>. Acesso em 10 ago. 2021.

The Pallets Projects. **Quickstart.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x/quickstart>. Acesso em 10 jul. 2021.

The Pallets Projects. **Request / Response Objects.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x/wrappers>. Acesso em 10 jul. 2021.

The Pallets Projects. **URL Routing.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x/routing>. Acesso em 10 jul. 2021.

The Pallets Projects. **Utilities.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x/utils>. Acesso em 10 jul. 2021.

The Pallets Projects. **Welcome to Flask - Flask Documentation.** (s.d.) Disponível em: <https://flask.palletsprojects.com/en/2.0.x>. Acesso em 10 jul. 2021.

The Pallets Projects. **Werkzeug.** (s.d.) Disponível em: <https://werkzeug.palletsprojects.com/en/2.0.x>. Acesso em 10 jul. 2021.

Welch, Kevin; Breckenridge, Sean; *et al.* **List of query params with Flask request.args.** 2019. Disponível em: <https://stackoverflow.com/a/57914827/540552>. Acesso em 10 ago. 2021.