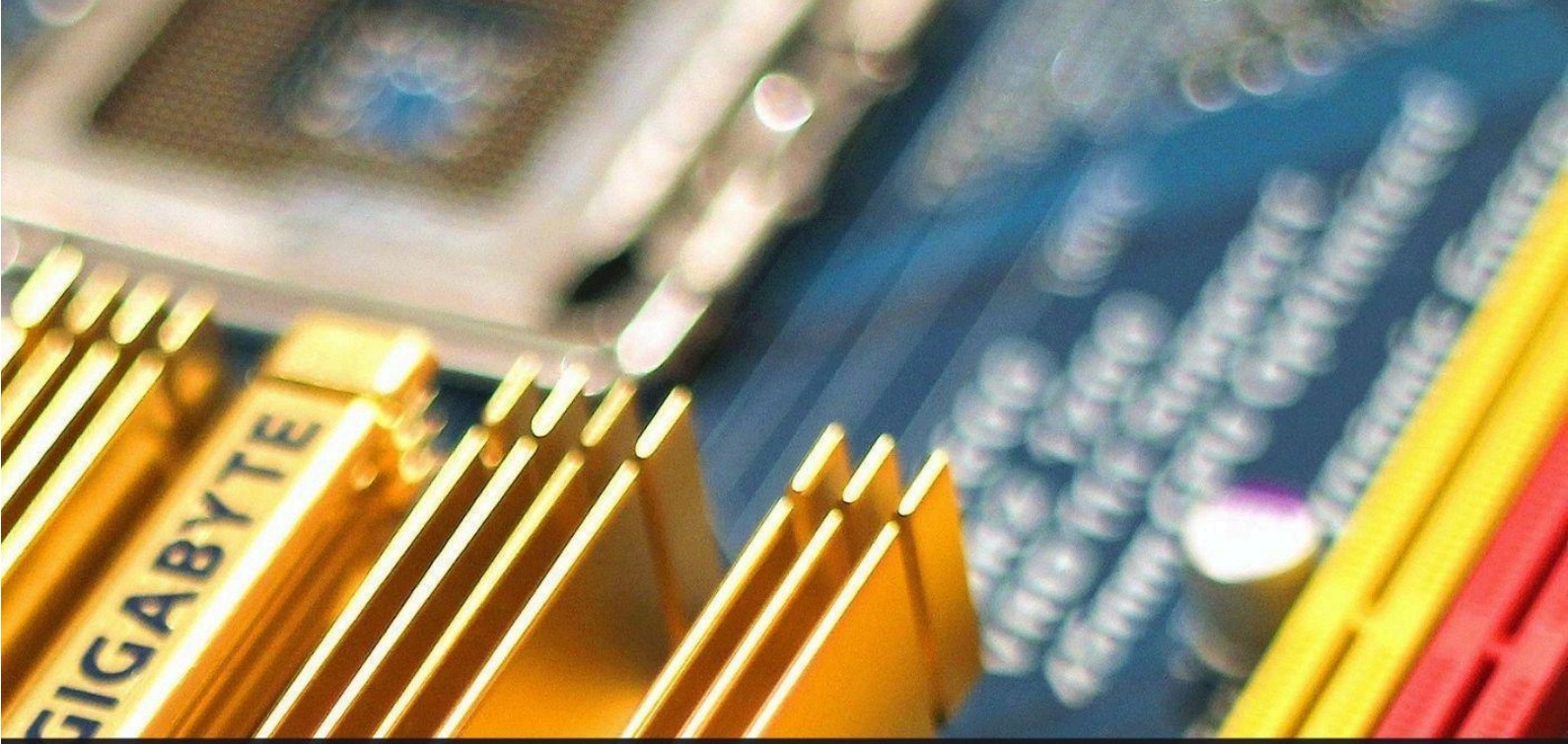
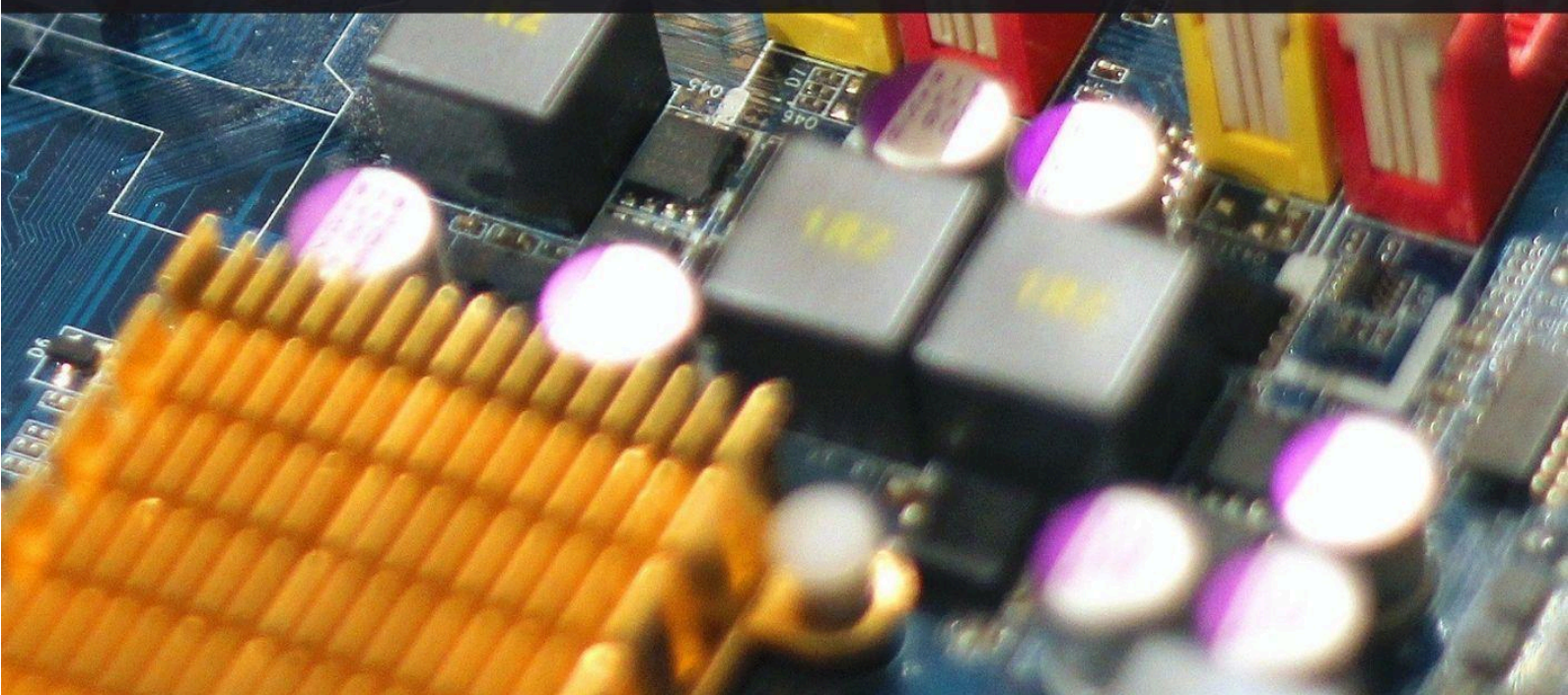


DESENVOLVIMENTO DE APIs E MICROSSERVIÇOS



10

Sistemas Distribuídos: Integração entre Requests e Flask

Andréia Cristina dos Santos Gusmão

Resumo

Aprendemos que é possível acessar informações na internet e baixá-las para integrar essas informações em aplicações, com a biblioteca Requests. Também aprendemos a desenvolver o lado servidor de aplicações HTTP com o framework Flask. Agora, vamos integrar nossa aplicação com Flask e Request com métodos de requisições HTTP para que possamos criar servidores com mais funcionalidades.

10.1. Relembrando o primeiro código com Flask

Para iniciar, caso ainda não tenha instalado o framework Flask e a biblioteca Request, é obrigatório fazer as instalações. No *prompt de comando*, basta digitar os seguintes comandos:

- Windows

```
pip install --user flask
pip install --user requests
```

- Mac ou Linux

```
pip3 install --user flask
pip3 install --user requests
```

10.1.1. Vamos recordar como escrevemos nosso primeiro código com Flask?

Precisamos fazer o import: `from flask import Flask`. Como podemos ver temos **flask** (com letras minúsculas) que é o nome da biblioteca de onde é feita a importação e também **Flask** (com a inicial maiúscula) que é o nome de uma classe que será importada. Para saber mais, consulte o material da Parte 7 - Unidade 3.

A Figura 10.1 mostra um exemplo de aplicação em Flask para recordarmos a sintaxe, a qual denominamos de **exemplo1.py**.

Figura 10.1. Primeiro código com Flask: exemplo1.py

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route("/")
6  def start():
7      return "Vamos aprender a integrar Requests e Flask!"
8
9  if __name__ == '__main__':
10     app.run(host = 'localhost', port = 5002, debug = True)
11
```

Fonte: do autor, 2021

Na linha 10 da Figura 10.1, estamos configurando para que nossa aplicação seja executada no host 'localhost' e na porta 5002. Quando assumimos 'debug=True', queremos garantir que, toda vez que fizermos qualquer alteração no arquivo de código e salvá-lo, será feito um 'reload' automático no servidor.

Vamos então, executar o código de exemplo1.py no cmd (prompt de comando). A saída deve ser como mostra na Figura 10.2. Caso seja diferente, provavelmente ocorreu algum erro.

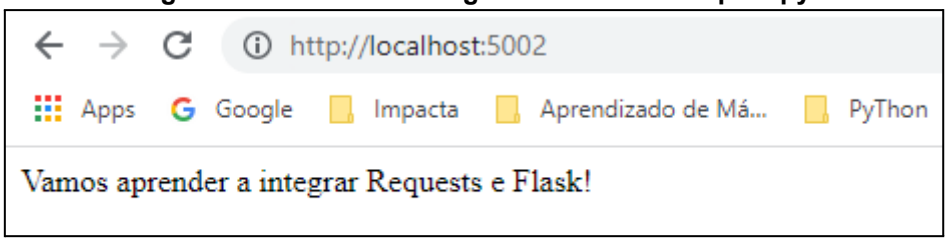
Figura 10.2. Primeiro código com Flask: exemplo1.py

```
C:\dam>python exemplo1.py
 * Serving Flask app "exemplo1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
 * Debugger is active!
 * Debugger PIN: 119-118-421
 * Running on http://localhost:5002/ (Press CTRL+C to quit)
```

Fonte: do autor, 2021

No navegador, digitamos http://localhost:5002/, e será mostrado o texto 'Vamos aprender a integrar Requests e Flask!' (Figura 10.3).

Figura 10.3. Primeiro código com Flask: exemplo1.py



Fonte: do autor, 2021

10.2. Criando nosso projeto para integração de Requests e Flask

Vamos criar um projeto para testar as requisições com Flask. Consideramos para esse exemplo duas entidades: **alunos** e **professores**, em que cada entidade, tenha um id (valor inteiro) e um nome (texto/string). Iremos salvar nosso código Python com o nome **exemplo2.py**.

Para representar os dados que vamos armazenar, iremos criar um dicionário com

duas chaves, em que o valor para cada chave é uma lista.

Os alunos serão armazenados na lista associada à chave **'ALUNO'** e os professores na lista associada à chave **'PROFESSOR'.**

Inicializamos nosso dicionário com três valores para cada chave, ou seja, três alunos e três professores, conforme mostra o quadro a seguir:

Figura 10.4. Criando dicionário com duas chaves e listas

```
database = {
    'ALUNO' : [{"id": 1, "nome": "Andreia"},
               {"id": 2, "nome": "Arthur"},
               {"id": 3, "nome": "Pedro"}],

    'PROFESSOR' : [{"id": 1, "nome": "Professor1"},
                   {"id": 2, "nome": "Professor2"},
                   {"id": 3, "nome": "Professor3"}],
}
```

Fonte: do autor, 2021

Outra opção, poderíamos iniciar nosso dicionário vazio, e depois, adicionar as chaves com as respectivas listas vazias, conforme imagem 10.5.

Figura 10.5. Criando dicionário com listas vazias

```
database = {}
database['ALUNO'] = []
database['PROFESSOR'] = []
```

Fonte: do autor, 2021

Mas nesse exemplo, vamos considerar como dados de entrada os três alunos e três professores mostrados anteriormente. A seguir, vamos aprender a criar rotas especificando os métodos.

10.3. Criando rotas com o verbo GET

Os exemplos que iremos apresentar, estão no arquivo de código Python **exemplo2.py**. Lembre-se: para testar a url no navegador, antes o código em Python deve ser executado no prompt de comando.

10.3.1. Definindo nossa primeira rota: /alunos com GET

Figura 10.5. Definição da primeira rota

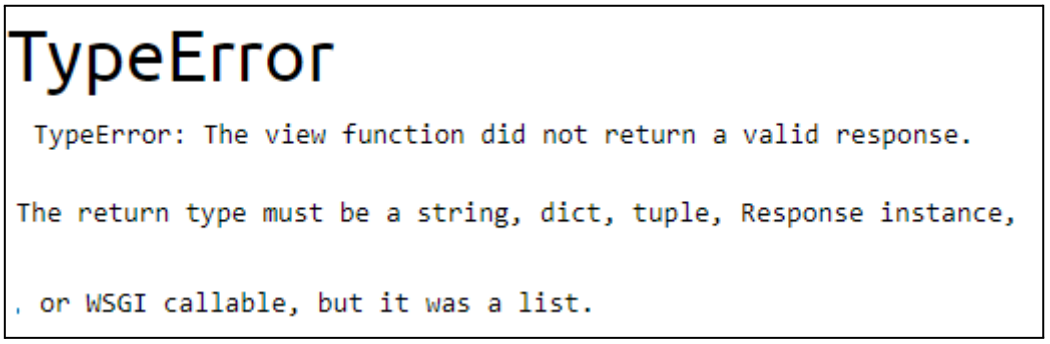
```
@app.route('/alunos')
def getAlunos():
    return database['ALUNO']
```

Fonte: do autor, 2021

Nossa rota chama-se '/alunos', ou seja, quando no navegador, testarmos http://localhost:5002/alunos, iniciará a execução da função getAlunos(), que retorna todos os alunos cadastrados no dicionário database.

E o que aconteceu? A figura 10.6 mostra que retornou a um erro de tipo inválido. Estamos retornando uma lista do nosso dicionário.

Figura 10.6. Apresentação de erro de tipo inválido



Fonte: do autor, 2021

10.3.2. Como resolver esse problema?

O retorno de um método tratado como rota pelo flask (@app.route) deve ser um texto (string). É possível converter uma lista/dicionário em string usando o formato json através do método **jsonify.** É preciso então, fazer o import do jsonify no nosso código Python: from flask import Flask, jsonify

A partir do import, podemos alterar o código da nossa rota /alunos, para que os dados de retorno sejam convertidos para string, conforme figura 10.7, a seguir:

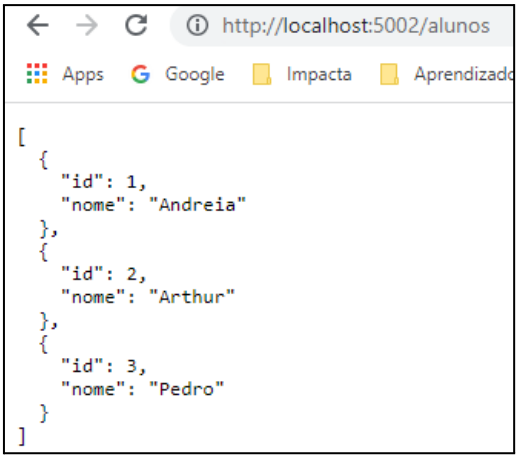
Figura 10.7. Convertendo uma lista/dicionário em string

```
@app.route('/alunos')
def getAlunos():
    #estou convertendo uma lista de dicionários em uma string no formato json
    return jsonify(database['ALUNO'])
```

Fonte: do autor, 2021.

Não definimos o método na rota, não é mesmo? Mas, por padrão, é **GET**, por isso, não escrevemos na rota o nome do método. O método **get** é utilizado para obter algo ou alguma coisa, ou seja, aqui vamos buscar os dados armazenados no dicionário e retornar para o usuário. A Figura 10.8 mostra o resultado apresentado no navegador após a execução do código http://localhost:5002/alunos.

Figura 10.8. Resultado da execução da rota /alunos: exemplo2.py



Fonte: do autor, 2021

Como podemos ver na Figura 10.8, os três alunos inicialmente adicionados no

nosso database foram mostrados no navegador, isso porque, nossa rota retorna todos os dados de todos os alunos.

Analogamente, podemos definir a mesma rota, para mostrar todos os **professores** do nosso database, conforme apresentado na figura 10.09.

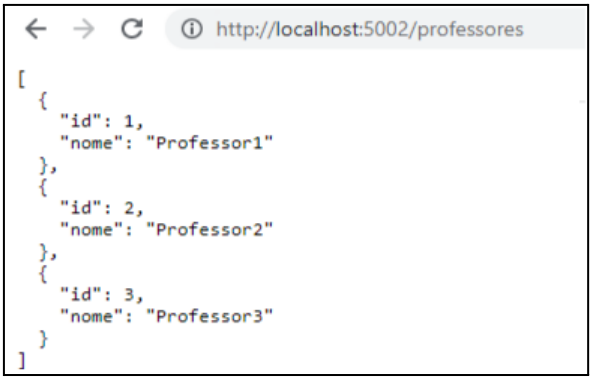
Figura 10.9. Definindo para mostrar todos os professores

```
@app.route('/professores')
def getProfesores():
    return jsonify(database['PROFESSOR'])
```

Fonte: do autor, 2021

No navegador, através de http://localhost:5002/professores, podemos ver o resultado da rota /professores, através da função getProfessores(), como mostra na Figura 10.10.

Figura 10.10. Resultado da execução da rota /professores: exemplo2.py



Fonte: do autor, 2021

Podemos definir uma rota que mostre todos os alunos e professores, ou seja, todos os dados do dicionário? Podemos, para isso, nós definimos uma nova rota denominada /show_all com a função getAll(), que retorna para o usuário o database completo, de acordo com a figura 10.11.

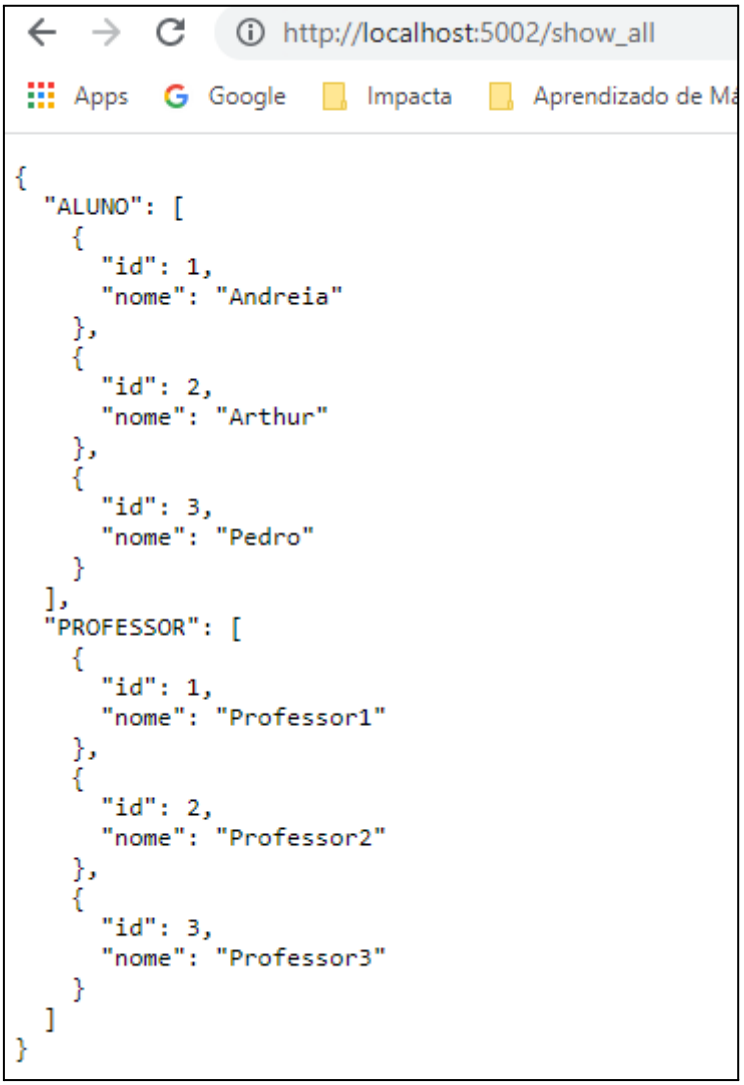
Figura 10.11. Definindo rota para mostrar todos os alunos e professores

```
@app.route('/show_all')
def getAll():
    return jsonify(database)
```

Fonte: do autor, 2021

Na Figura 10.12, mostramos o resultado da rota /show_all no navegador, e na Figura 10.13 apresentamos o código Python completo de exemplo2.py.

Figura 10.12. Resultado da execução da rota /show_all: exemplo2.py



Fonte: do autor, 2021

Segue um resumo, de como testar no navegador todas as requisições que criamos no arquivo exemplo2.py.

Lembre-se: para funcionar no navegador, nosso servidor deve estar executando, ou seja, devemos executar no prompt de comando (dentro do diretório onde está nosso código Python):

python exemplo2.py

- http://localhost:5002/
- http://localhost:5002/alunos
- http://localhost:5002/professores
- http://localhost:5002/show_all

Figura 10.13. Código completo do exemplo: exemplo2.py

```python
from flask import Flask, jsonify

app = Flask(__name__)

database = {
    'ALUNO' : [{"id": 1, "nome": "Andreia"},
               {"id": 2, "nome": "Arthur"},
               {"id": 3, "nome": "Pedro"}],

    'PROFESSOR' : [{"id": 1, "nome": "Professor1"},
               {"id": 2, "nome": "Professor2"},
               {"id": 3, "nome": "Professor3"}],
}

@app.route('/alunos')
def getAlunos():
    return jsonify(database['ALUNO'])


@app.route('/professores')
def getProfesores():
    return jsonify(database['PROFESSOR'])


@app.route('/show_all')
def getAll():
    return jsonify(database)


@app.route("/")
def start():
    return "Vamos aprender a integrar Requests e Flask!"

if __name__ == '__main__':
    app.run(host = 'localhost', port = 5002, debug = True)
```

Fonte: do autor, 2021

Criamos rotas com o verbo GET (obter) para nosso exemplo para os dados de alunos. Como podemos observar **GET é o PADRÃO**, mesmo que não especifiquemos que estamos usando o método get, o flask já entende que se não definimos um método, estamos executando **get**.

Aprendemos como funciona o método GET, mas e as outras ações possíveis de requisições? Iremos ver a seguir, exemplos com *POST, PUT, DELETE.*

10.4. Criando rotas com o verbo POST

O método **POST** é utilizado para salvar novos dados, ou seja, busca dados de entrada do usuário, para atualizar seu banco de dados. Anteriormente, vimos o método **GET**, que mostra os dados que já temos.

Agora, vamos inserir novos dados. Continuando com o exemplo, vamos salvar nosso arquivo de código como **exemplo3.py**, que contém o que já aprendemos e os novos métodos que serão mostrados a seguir.

Vamos adicionar um item importante no nossos **import** no código Python, precisamos especificar que estamos utilizando **requests:**

```
from flask import Flask, jsonify, request
```

10.4.1. Definindo nosso rota /alunos com o método POST

Observe, na figura 10.14 que temos o mesmo nome de rota '/alunos', com a diferença que temos um parâmetro novo '**methods=[]**', a qual informamos o valor POST, ou seja, vamos utilizar a rota que se chama alunos, porém com o método POST, para inserir um novo aluno.

Figura 10.14. Rota com método POST

```
@app.route('/alunos', methods=['POST'])
def inserir_aluno():
    novo_aluno = request.json
    database['ALUNO'].append(novo_aluno)
    return jsonify(database['ALUNO'])
```

Fonte: do autor, 2021

Na linha novo_aluno = request.json, quer dizer que vamos buscar através do request os dados de entrada digitados pelo usuário no formato json e atribuímos para a variável novo_aluno. Logo em seguida, acessamos nosso dicionário na chave ALUNO e adicionamos o valor que está em novo_aluno. Ou seja, vamos adicionar um novo aluno, no mesmo formato dos dados da existentes: com um atributo id e um atributo nome.

Será retornado para o usuário todos os alunos cadastrados através do comando return jsonify(database['ALUNO']).

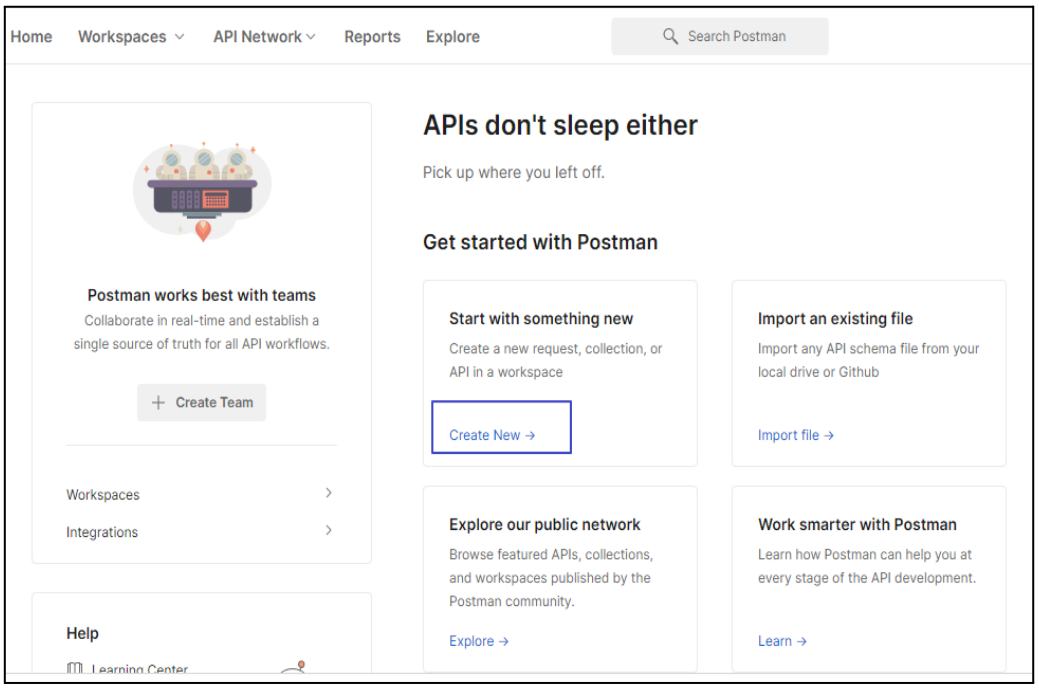
E agora, precisamos testar nossa aplicação, para verificar se está funcionando corretamente a inserção de um novo aluno. Mas afinal, como testar? Os testes serão feitos através do POSTMAN, diferentemente dos exemplos com método GET.

O POSTMAN pode ser instalado na sua máquina, ou acessado de forma online, através da url. Consulte a seção 10.10 no final desta aula para saber mais.

A demonstração aqui será feita pelo acesso a url, porém, as telas são as mesmas. Acesse no seu navegador **https://www.postman.com/. Clique em Sign in e digite seu usuário/email e senha. Caso não tenha cadastrado, é possível fazê-lo de forma rápida e gratuita.**

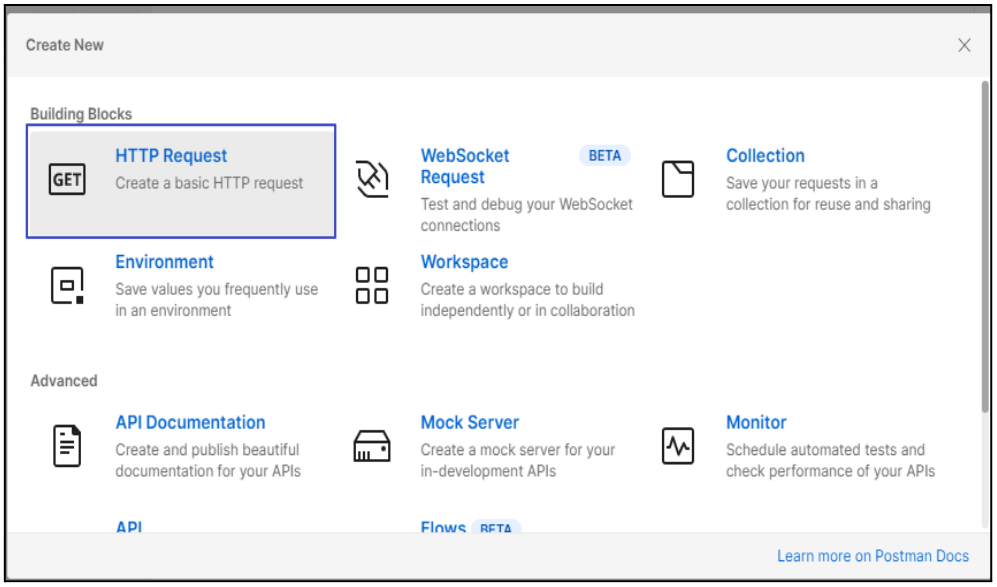
Na tela apresentada conforme Figura 10.15, clique em Create New e depois em HTTP Request (Figura 10.16).

Figura 10.15. Tela inicial POSTMAN



Fonte: do autor, 2021

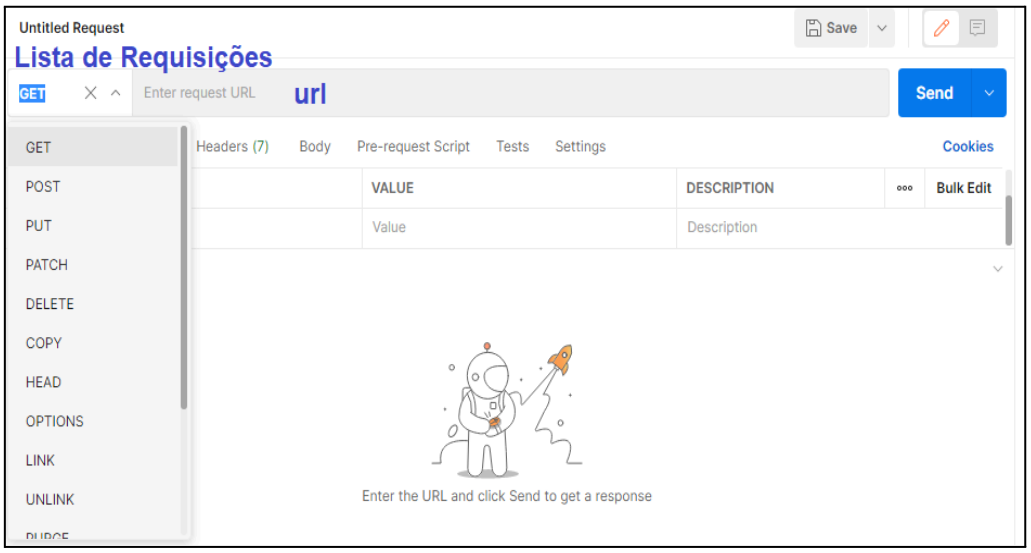
Figura 10.16. Criando nova HTTP Request com POSTMAN



Fonte: do autor, 2021

Será mostrada a tela onde iremos inserir nossas requisições para testar**.** **Lembre-se**, nosso servidor (código Python) deve estar em execução através do prompt de comando. Na Figura 10.17 podemos ver a lista das requisições possíveis (GET, POST, PUT, entre outras). O método GET também pode ser testado aqui, porém, mostramos que essa requisição pode ser testada diretamente no navegador. Além das requisições, temos o espaço destinado à nossa url, a qual iremos testar.

Figura 10.17. Testando uma request com POSTMAN



Fonte: do autor, 2021

Bom, agora que apresentamos as telas do POSTMAN, vamos testar nossa rota /alunos com POST.

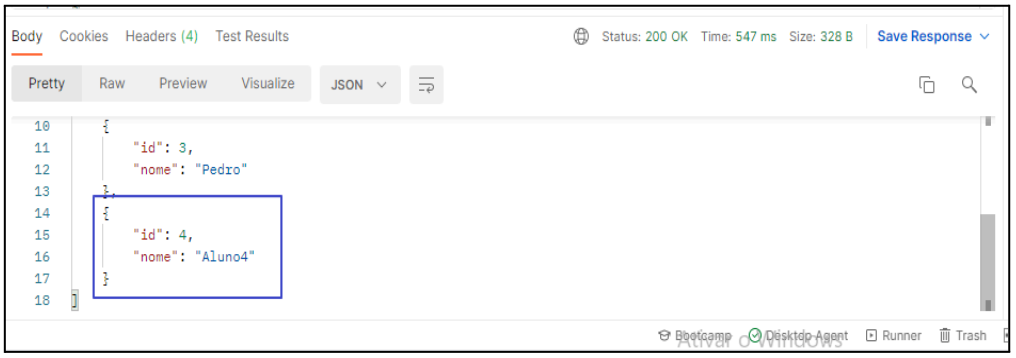
Na lista de requisições (Figura 10.17) escolhemos o método POST, em url digitamos http://localhost:5002/alunos. Precisamos passar um aluno como parâmetro para inserir. Devemos escolher então a opção Body, selecionar raw e escolher JSON, que é o formato da nossa entrada de dados (Figura 10.18). Feito essas configurações, precisamos digitar os dados do aluno que iremos inserir, no mesmo formato dos demais: **id e nome**, com a mesma estrutura, conforme mostrada na Figura 10.19 ({"id": 4, "nome": "Aluno4"}). Em seguida, é só clicar no botão Send para executar.

O resultado é apresentado na Figura 10.19. Como podemos observar, retorna todos os alunos já cadastrados, considerando o 'Aluno4', que acabamos de adicionar.

Figura 10.18. Testando o método POST com a rota /alunos: exemplo3.py

Fonte: do autor, 2021

Figura 10.19. Resultado do método POST com a rota /alunos: exemplo3.py



Fonte: do autor, 2021

O mesmo resultado pode ser obtido no navegador (Figura 10.20), através do método GET para a rota /alunos, que também irá retornar todos os alunos, mas somente após a inserção com método POST no POSTMAN, digitando http://localhost:5002/alunos.

Figura 10.20. Resultado do método GET após o método POST com a rota /alunos: exemplo3.py

Fonte: do autor, 2021

10.5. Criando rotas com o verbo DELETE

Além de poder salvar um novo aluno, podemos também excluí-lo através do método DELETE. Continuaremos nosso exemplo no arquivo **exemplo3.py**.

10.5.1. Definindo nosso rota /alunos com o método DELETE

Observe, na figura, 10.21 que temos o mesmo nome de rota '/alunos', com a diferença que temos um parâmetro novo '**methods=[]**', a qual informamos o valor DELETE, ou seja, vamos utilizar a rota que se chama alunos, porém com o método DELETE, para que seja possível excluir um aluno do nosso database.

Criamos a rota @app.route('/alunos/<int:id_aluno>', methods=['DELETE']), em que <int:id_aluno> é o parâmetro que iremos passar com o id do aluno que será

excluído.

Esse parâmetro de entrada é o valor que será passado para a função def excluir_aluno(id_aluno). Será verificado para cada aluno do database, se o id do aluno cadastrado é igual ao id_aluno passado como parâmetro if aluno['id'] == id_aluno:. Se for igual, esse aluno será excluído e será retornado para o usuário todos os alunos atualizados. Se não encontrar o aluno com o id pesquisado, será informado uma mensagem 'Aluno não encontrado'.

Figura 10.21. Rota com método DELETE

```
@app.route('/alunos/<int:id_aluno>', methods=['DELETE'])
def excluir_aluno(id_aluno):
    for aluno in database['ALUNO']:
        if aluno['id'] == id_aluno:
            database['ALUNO'].remove(aluno)
            return jsonify(database['ALUNO'])
    return 'Aluno não encontrado', 404
```

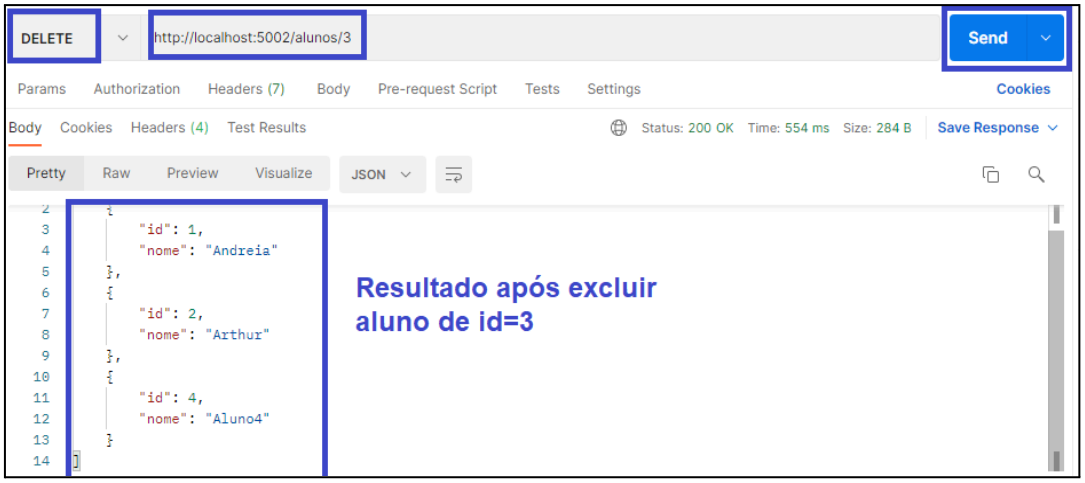
Fonte: do autor, 2021

Na Figura 10.22 apresentamos como o teste deve ser realizado dentro do POSTMAN. Observe que escolhemos o método DELETE, digitamos a url http://localhost:5002/alunos/3, em que o valor '3' é o id do aluno que queremos excluir. Não precisamos digitar mais nada. Em Body, podemos marcar a opção NONE. E clique em SEND. A função para excluir o aluno de id=3 é executada. O aluno é encontrado e excluído. Após, é mostrada a saída com o database atualizado, já sem o aluno de id = 3.

Mas temos outro caso: tentar excluir um aluno que não exista cadastrado. Considere a url http://localhost:5002/alunos/10. Após o clique no botão SEND, é mostrado como resultado a mensagem 'Aluno não encontrado', pois não existe no nosso database nenhum aluno com id=10 (Figura 10.23).

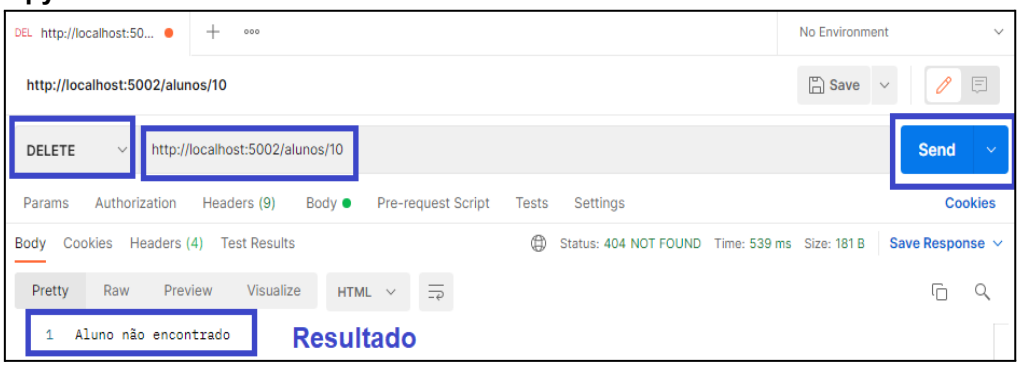
Assim como apresentamos para o método POST, podemos visualizar o resultado com o método GET (http://localhost:5002/alunos) no navegador, em que mostra na tela todos os alunos cadastrados (Figura 10.24). **ATENÇÃO**: só executa no navegador após ter executado o DELETE, para ver o database atualizado.

Figura 10.22. Testando o método DELETE com a rota /alunos e id_aluno=3: exemplo3.py



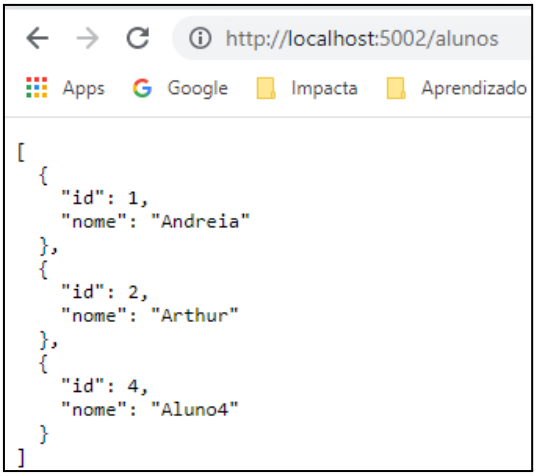
Fonte: do autor, 2021

Figura 10.23. Testando o método DELETE com a rota /alunos e id_aluno=10: exemplo3.py.



Fonte: do autor, 2021

Figura 10.24. Resultado do método GET após o método DELETE com a rota /alunos: exemplo3.py.



Fonte: do autor, 2021

10.6. Criando rotas com o verbo PUT

Verificamos que digitamos o nome de um aluno errado. O aluno com id=4 tem o nome Heitor, e não Aluno4. Como posso alterar o nome? A alteração pode ser feita através do método PUT. Continuaremos nosso exemplo no arquivo **exemplo3.py**.

10.6.1. Definindo nossa rota /alunos com o método PUT

Observe que temos o mesmo nome de rota '/alunos', e para o parâmetro '**methods=[]**' informamos o valor PUT, ou seja, vamos utilizar a rota que se chama alunos, porém com o método PUT para que seja possível atualizar um aluno do nosso database.

Criamos a rota @app.route('/alunos/<int:id_aluno>', methods=['PUT']), em que <int:id_aluno> é o parâmetro que iremos passar com o id do aluno que será alterado.

Esse parâmetro de entrada é o valor que será passado para a função def atualizar(id_aluno). Além de digitar na url qual id do aluno que será alterado, o usuário deve fornecer em formato json os dados do aluno (id e nome) da mesma forma já explicado para o método POST. Buscamos então, os dados digitados e armazenamos em atualiza_aluno. Será verificado para cada aluno do database, se o id do aluno cadastrado é igual ao id_aluno passado como parâmetro if aluno['id'] = id_aluno:. Se for igual, esse

aluno será excluído e adicionado o novo aluno que está armazenado na variável atualiza_aluno. Será retornado para o usuário todos os alunos, já com a alteração solicitada. Se não encontrar o aluno com o id pesquisado, será informado uma mensagem 'Aluno não encontrado', conforme figura 10.25.

Figura 10.25. id pesquisado, não encontrado

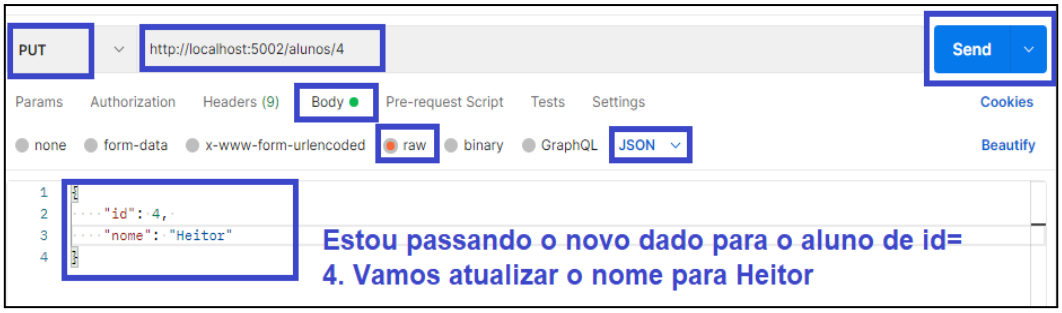
```
@app.route('/alunos/<int:id_aluno>', methods=['PUT'])
def atualizar(id_aluno):
    atualiza_aluno = request.get_json()
    for aluno in database['ALUNO']:
        if aluno['id'] == id_aluno:
            database['ALUNO'].remove(aluno)
            database['ALUNO'].append(atualiza_aluno)
            return jsonify(database['ALUNO'])
    return 'Aluno não encontrado', 404
```

Fonte: do autor, 2021

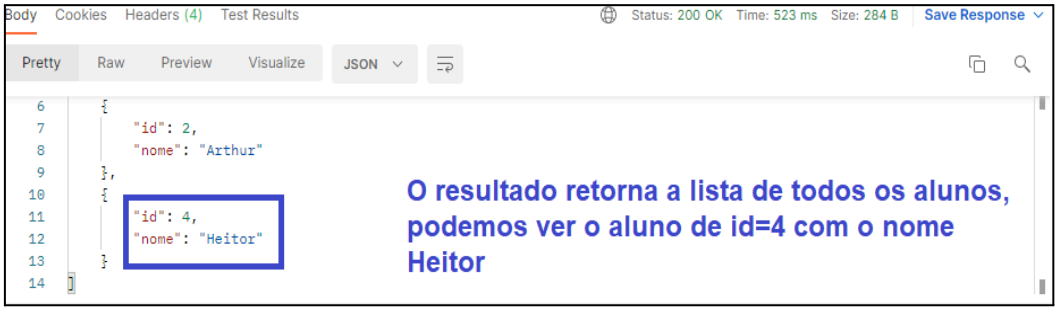
Na Figura 10.26, (a) apresentamos como o teste deve ser realizado dentro do POSTMAN. Observe que escolhemos o método PUT, digitamos a url http://localhost:5002/alunos/4 , em que o valor '4' é o id do aluno que queremos alterar. Em Body marcamos a opção raw e escolhemos o formato JSON, que é o formato dos dados de entrada que vamos digitar. Observe que mantivemos o mesmo id (4) e somente alteramos o nome para Heitor. E clique em SEND. A função para alterar o aluno de id 4 é executada.

O aluno é encontrado e excluído e logo em seguida, adicionado o novo aluno com os dados de entrada. A alteração então, exclui e insere novamente. Após, é mostrada a saída com o database atualizado, já com o aluno de id = 4 com o nome Heitor (Figura 10.26 (b)). Na Figura 10.27 mostramos o resultado do método PUT no navegador, através do método GET: http://localhost:5002/alunos/.

Figura 10.26. Testando o método PUT com a rota /alunos para id_aluno=4: exemplo3.py



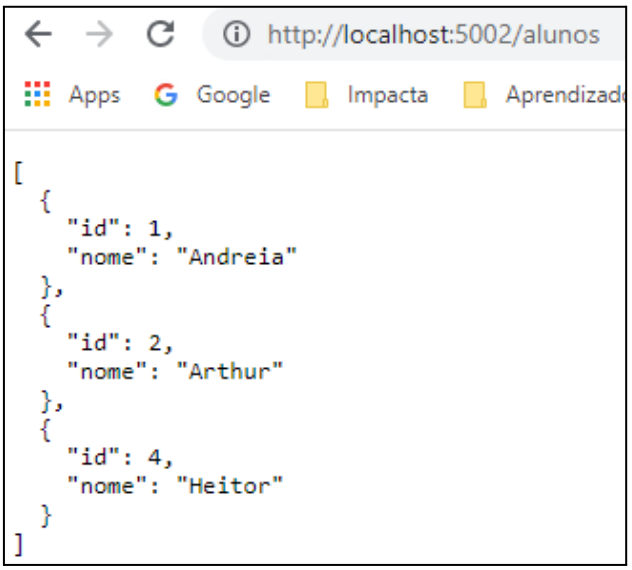
(a)



(b)

Fonte: do autor, 2021

Figura 10.27. Resultado do método GET após o método PUT com a rota /alunos: exemplo3.py



Fonte: do autor, 2021

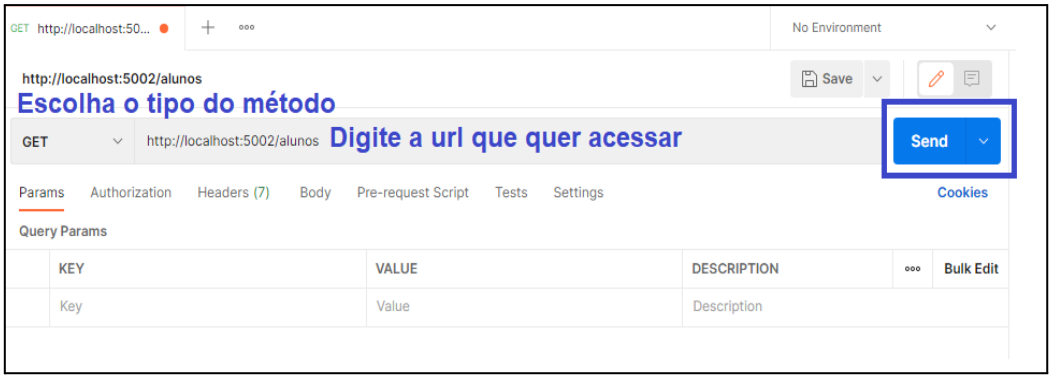
Podemos testar também, passar um id de aluno que não existe para alterar. Não apresentamos esse teste aqui, mas segue a mesma lógica apresentada com a diferença na url onde escrevemos '4', colocaremos outro valor inválido. E deve mostrar na tela 'Aluno não encontrado'.

10.7. Criando novas rotas com o verbo GET

Iniciamos nossas rotas com o método GET para mostrar todos os alunos cadastrados. Vimos que por padrão, quando não especificamos o método na declaração da rota no código em Python, já entende que estamos usando o método GET. Vimos também que o método GET pode ser testado facilmente diretamente no navegador.

A Figura 10.28 (a) mostra no ambiente POSTMAN como testar a rota http://localhost:5002/alunos/ com o método GET, para mostrar todos os alunos cadastrados. O resultado é apresentado na Figura 10.28(b).

Figura 10.28. Testando o método GET com a rota /alunos no POSTMAN: exemplo3.py



(a)

(b)
Fonte: do autor, 2021

Bom, vimos que GET é obter, e mostramos uma função para retornar todos os alunos. Mas se eu quiser buscar um único aluno? É possível utilizando um parâmetro informando qual aluno deseja mostrar os dados.

Observe que temos o mesmo nome de rota '/alunos', e para o parâmetro '**methods=[]**' informamos o valor GET.

Criamos a rota @app.route('/alunos/<int:id_aluno>', methods=['GET']), em que <int:id_aluno> é o parâmetro que iremos passar com o id do aluno que será pesquisado.

Esse parâmetro de entrada é o valor que será passado para a função def localizar_aluno(id_aluno), através da url. Será verificado para cada aluno do database, se o id do aluno cadastrado é igual ao id_aluno passado como parâmetro if aluno['id'] = id_aluno:. Se for igual, esse aluno será retornado para o usuário. Se não encontrar o aluno com o id pesquisado, será informado uma mensagem 'Aluno não encontrado', conforme figura 10.29.

Figura 10.29. id pesquisado, não encontrado

```
@app.route('/alunos/<int:id_aluno>', methods=['GET'])
def localizar_aluno(id_aluno):
    for aluno in database['ALUNO']:
        if aluno['id'] == id_aluno:
            return jsonify(aluno)
    return 'Aluno não encontrado', 404
```

Fonte: do autor, 2021

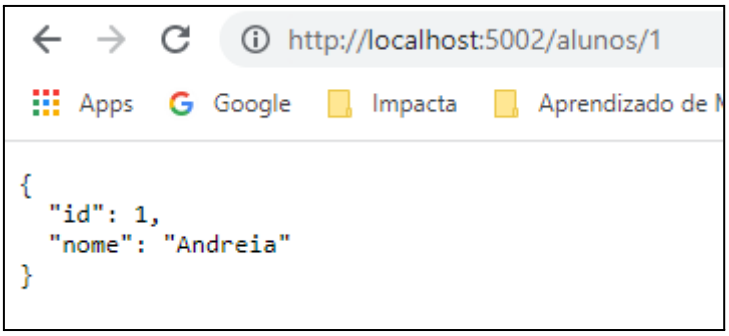
Na Figura 10.30 mostramos a configuração necessária no POSTMAN para executar o método GET com a url http://localhost:5002/alunos/1, em que '1' é o id do aluno que será pesquisado. E na Figura 10.31 mostramos a execução da mesma url http://localhost:5002/alunos/1 no navegador.

Caso o id do aluno pesquisado não exista no database, será retornada a mensagem 'Aluno não encontrado', conforme Figura 10.32, executada no navegador.

Figura 10.30. Testando o método GET no POSTMAN com a rota /alunos para pesquisar o aluno com id=1: exemplo3.py

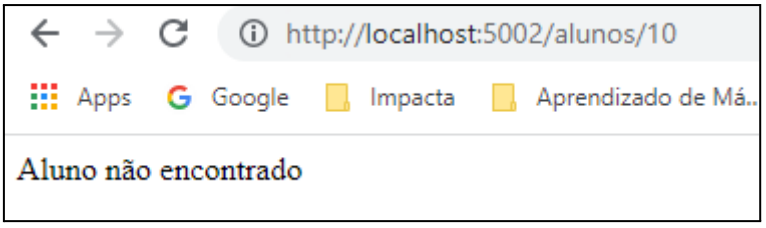
Fonte: do autor, 2021

Figura 10.31. Testando o método GET no navegador com a rota /alunos para pesquisar o aluno com id=1: exemplo3.py



Fonte: do autor, 2021

Figura 10.32. Testando o método GET no navegador com a rota /alunos para pesquisar o aluno com id=10: exemplo3.py



Fonte: do autor, 2021

10.8. Criando novas rotas com o verbo POST

Vimos que é possível salvar um novo aluno com o método POST. Agora vamos apresentar uma nova rota, com a função de resetar nosso database com o método POST.

Observe que temos agora o nome de rota '/reseta', e para o parâmetro '**methods=[]**' informamos o valor POST.

Criamos a rota @app.route('/reseta', methods=['POST']). Note que aqui, não vamos passar parâmetro, como o id do aluno, pois iremos resetar nosso database, ou seja, excluir todos os dados cadastrados (alunos e professores).

Na linha database['ALUNO'] = [], estamos criando novamente a chave 'ALUNO' do nosso dicionário database, com uma lista vazia. O mesmo fazemos para database['PROFESSOR'] = []. Depois, retornamos para o usuário o database, a qual está vazio.

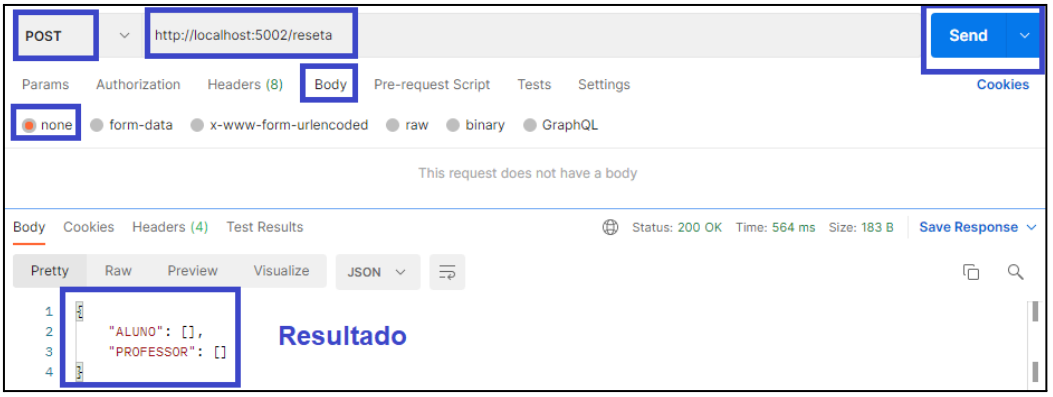
Figura 10.34. Criando rotas com o método POST

```
@app.route('/reseta', methods=['POST'])
def resetar():
    database['ALUNO'] = []
    database['PROFESSOR'] = []
    return jsonify(database)
```

Fonte: do autor, 2021

Na Figura 10.35 mostramos a configuração no POSTMAN para executar a rota reseta com o método POST. Observe que em Body, marcamos NONE, ou seja, não vamos digitar os dados de entrada em formato JSON (como fizemos no método POST para inserir novo aluno), pois aqui não vamos salvar novos dados, e sim salvar uma lista vazia. O resultado pode ser visto na mesma figura.

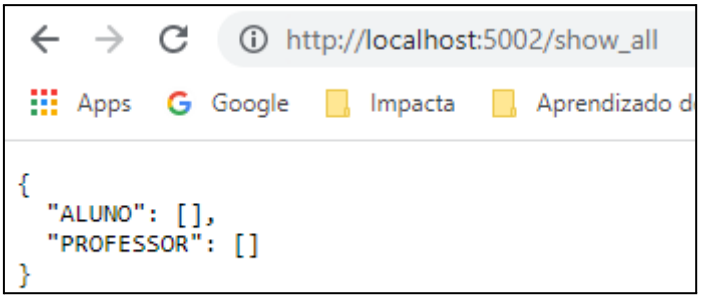
Figura 10.35. Testando o método POST no POSTMAN com a rota /reseta: exemplo3.py



Fonte: do autor, 2021

No navegador, podemos visualizar o mesmo resultado, após a execução da rota /reseta, com a rota do método GET /show_all: http://localhost:5002/show_all (Figura 10.36).

Figura 10.36. Resultado da rota /show_all após a execução da rota /reseta: exemplo3.py



Fonte: do autor, 2021

Aprendemos a integrar o Flask com Requests e vimos exemplos com os métodos GET, POST, DELETE e PUT. Agora, é hora de praticar os conceitos para implementar novas rotas e funcionalidades.

10.9. Código Completo

Os códigos em Python dos exemplos demonstrados nessa aula (exemplo1.py, exemplo2.py e exemplo3.py) estão disponibilizados junto ao conteúdo dessa aula como material complementar. Aproveite para implementar as mesmas funcionalidades para os professores, visto que mostramos as funções e rotas somente para os alunos (exceto as rotas /professores e /show_all).

10.10. Acesso via POSTMAN

Para quem não fez o download do programa POSTMAN e está com dúvidas, consulte o material e vídeo da aula Parte 5 – Unidade 2 (https://www.postman.com/downloads/).

Para quem prefere testar de forma online pode ser feita através do site https://web.postman.co/home após a criação de uma conta.

ATENÇÃO: para realizar os testes através da versão online, é preciso fazer o download e instalação do Postman Agent, a qual é redirecionado no próprio site. Explore a documentação do POSTMAN em https://learning.postman.com/docs/getting-started/introduction/.

Referências

Múltiplos autores. **Flask HTTP methods, handle GET & POST requests.** Disponível em: <https://pythonbasics.org/flask-http-methods/>. Acesso em: 15 out. 2021.

Gonçalves, Lucas Mendes Marques; Silva, Victor Williams Stafusa da; Resende, Emilio Murta. ***Flask – LMS.*** [PowerPoint de apoio à disciplina de Desenvolvimento de APIs e Microsserviços, lecionada na Faculdade Impacta]. 2021.