
*O objetivo deste mini tutorial é demonstrar ao discente como criar e consumir uma API REST, utilizando o micro framework Flask e a linguagem Python.*

## 1. Entendendo cada tecnologia envolvida

### 1.1 Flask

Framework é um componente que ajuda a simplificar o desenvolvimento de aplicativos, principalmente web. É como um kit de ferramentas que inclui serviços da web, recursos, APIs e fornece uma base estável para seu aplicativo.

Flask é um pequeno framework web escrito em Python. É classificado como um microframework porque não requer ferramentas ou bibliotecas particulares, mantendo um núcleo simples, porém, extensível.

“Micro” não significa que a sua aplicação web inteira tem que se encaixar em um único arquivo Python, embora certamente pode. Também não quer dizer que o Flask está faltando em termos de funcionalidade. O “micro” no microframework Flask visa manter o núcleo simples, mas extensível. Flask não vai tomar muitas decisões para você, como o banco de dados para usar. Essas decisões que ele faz, como o que motor de templates usar, são fáceis de mudar. Todo o resto é com você, então o Flask que pode ser tudo que você precisa e nada que você não faz.

Por padrão, Flask não inclui uma camada de abstração de banco de dados, uma validação de form ou qualquer outra coisa que já exista em bibliotecas diferentes que pode lidar com isso. Em vez disso, Flask suporta extensões para adicionar essa funcionalidade a sua aplicação como se fosse implantado no Flask. Há várias extensões para proporcionar a integração de banco de dados, validação de formulário, manipulação de upload, vários tecnologias de autenticação abertas, e muito mais. Flask pode ser “micro”, mas está pronto para uso na produção para uma variedade de necessidades.

**Fontes:** (1) https://flask-ptbr.readthedocs.io/en/latest/foreword.html#qual-o-significado-de-micro (2) http://pythonclub.com.br/what-the-flask-pt-1-introducao-ao-desenvolvimento-web-com-python.html

## 1.2 Python

Python é uma linguagem fácil de aprender e poderosa. Ela tem estruturas de dados de alto nível eficientes e uma abordagem simples, mas efetiva de programação orientada a objetos. A elegância de sintaxe e a tipagem dinâmica do Python aliadas com sua natureza interpretativa, o fazem a linguagem ideal para programas e desenvolvimento de aplicações rápidas em diversas áreas e na maioria das plataformas.
O interpretador Python e a extensiva biblioteca padrão estão disponíveis gratuitamente em código ou na forma binária para todas as maiores plataformas no endereço eletrônico do Python ( https://www.python.org ) e pode ser livremente distribuído.

**Fonte:** https://docs.python.org/pt-br/3/tutorial/index.html

## 1.3 Webservices, API REST e JSON

A primeira coisa a ser entendida é que reutilizar sistemas já existentes numa organização e acrescentar-lhes novas funcionalidades sem que seja necessário criar um sistema a partir do zero é possível. Ou seja, é possível melhorar os sistemas já existentes, integrando mais informação e novas funcionalidades de forma simples e rápida, seja por APIs remotas ou *webservices*.
Um *webservice* é uma coleção de protocolos e padrões de código aberto usados para trocar dados entre sistemas ou aplicativos, enquanto uma API é uma interface que pode ser usada para programar um software que interaja com um aplicativo existente. Ou seja, uma API é uma interface de software que permite que dois aplicativos interajam entre si sem qualquer envolvimento do usuário. Na prática, uma API é “um conjunto de funções e procedimentos” que permite acessar e construir algo sobre os dados e a funcionalidade de um aplicativo existente.

Ao contrário do que você possa imaginar, APIs e *webservices* não são mutuamente exclusivos. Na verdade, um é um subconjunto do outro: todo *webservice* é uma API - uma vez que expõe os dados e/ou funcionalidades de um aplicativo - mas nem toda API é um serviço da web. Isso ocorre porque a definição de um *webservice* é bastante restritiva quando se trata de implementação:

• Os *webservices* requerem uma rede: embora as APIs possam ser on-line ou off-line, os *webservices* devem usar uma rede.
• APIs são agnósticas de protocolo: embora as APIs possam usar qualquer protocolo ou estilo de *design*, os *webservices* geralmente usam SOAP (mas às vezes REST, UDDI e XML-RPC).

Além disso, pode-se dizer que muitas APIs públicas (remotas) são transparentes, com documentação aberta, além de portais de autoatendimento para rápida integração do desenvolvedor. Isso porque o objetivo de muitas APIs modernas é, afinal, facilitar a interação com um aplicativo. Por outro lado, os *webservices* não têm uma história tão aberta, em vez disso, eles tendem a oferecer dados e/ou funcionalidades específicas para parceiros específicos.
Transferência Representacional de Estado ou simplesmente REST (*Representational State Transfer*) não é um protocolo ou padrão, mas sim um conjunto de princípios de arquitetura. Os desenvolvedores de API podem implementar a arquitetura REST de maneiras variadas.
API REST, também chamada de API RESTful, é uma interface de programação de aplicações que segue conformidade com as restrições da arquitetura REST.
Quando uma solicitação é feita por meio de uma API RESTful, essa API transfere uma representação do estado do recurso ao solicitante. Essa informação, ou representação, é fornecida utilizando um dos vários formatos possíveis via HTTP: *Javascript Object Notation* (JSON), HTML, XLT ou texto simples. O formato JSON é o mais usado porque, apesar de seu nome, é independente de qualquer linguagem e pode ser lido por máquinas e humanos.
As APIs RESTful utilizam os verbos HTTP para definir qual a finalidade da requisição que está sendo enviada:

**GET:** a requisição é um pedido de dados para a API (leitura), ou seja, irá buscar os dados solicitados em algum banco e, provavelmente, irá retornar em formato JSON.

**POST:** tipo de requisição utilizada para criar um recurso em uma determinada API (inserção de dados).

**PUT:** requisição utilizada para atualizar o recurso (dados) indicado com alguma informação.

**DELETE:** requisição para excluir um recurso (dados).

Uma requisição REST pode enviar e receber dados em formato JSON, como no exemplo abaixo:

```
{
    "nome": "Elson",
    "sobrenome": "Abreu",
    "idade": 21,

    "endereco":  {
                        "logradouro": "Rua dos Tutoriais, 420",
                        "cidade": "Belo Horizonte",
                        "estado": "MG",
                        "cep": 32600500
                    },

    "telefone": [
                        {
                            "tipo": "fixo",
                            "numero": "31 555-1234"
                        },
                        {
                            "tipo": "celular",
                            "numero": "31 555-4567"
                        }
                ]
}
```

É preciso observar que as marcações, assim como os dados, são estipuladas pelo criador da aplicação. Os dados em JSON também trabalham com pares de atributos e valores. E em vez de marcadores, como no XML, utilizam delimitadores em cadeias: {}, []; e "". O delimitador { marca o início de uma seção e o } marca seu fim. Os pares de valor e atributo são separados por : e seus valores, quando texto, ficam entre aspas (números, por exemplo, não recebem as aspas).
Embora uma API REST precise estar em conformidade com os critérios acima, ela é considerada mais fácil de usar do que um protocolo prescrito, como o Protocolo Simples de Acesso a Objetos (SOAP). Esse tipo de protocolo tem requisitos específicos, como o sistema de mensagens XML, além de precisar cumprir com exigências de segurança incorporada e transações, o que o torna mais lento e pesado. Como dito anteriormente, este padrão é muito usado em *webservices*.
Em comparação, a arquitetura REST é composta de um conjunto de diretrizes que podem ser implementadas conforme necessário. Isso faz com que as APIs REST sejam mais rápidas e leves, o que é ideal para a Internet das Coisas (IoT) e o desenvolvimento de aplicações mobile.

No que se refere às boas práticas recomendadas para API REST, um ponto interessante a ser mencionado está relacionado ao design de *endpoints*.

Use substantivos ao invés de verbos em *endpoints*. Ao projetar uma API REST recomenda-se usar substantivos nos caminhos de *endpoints*, dando significado o que cada um deles. Isso ocorre porque métodos HTTP como GET, POST, PUT, PATCH e DELETE já estão na forma verbal para executar operações básicas de CRUD (Criar, Ler, Atualizar, Excluir).

Por exemplo, os *endpoints* de uma API com esta aparência:

http://www.meusite.com.br/**inserir-cliente**
http://www.meusite.com.br/**alterar-cliente**
http://www.meusite.com.br/**excluir-cliente**
http://www.meusite.com.br/**obter-cliente**

Poderiam se tornar um único *endpoint* da seguinte forma:

http://www.meusite.com.br/**clientes**

Assim, você deixaria os verbos HTTP lidarem com o que os *endpoints* fazem:

**GET** - obter dados
**POST** - inserir dados
**PUT** - atualizar dados
**DELETE** - excluir dados

**Fontes:**
https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api
https://www.hostgator.com.br/blog/api-restful/
https://ceweb.br/guias/dados-abertos/capitulo-38/
https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/

## 1.4 SQLite e DB Browser

SQLite é uma biblioteca desenvolvida em linguagem C que implementa um mecanismo de banco de dados SQL pequeno, rápido, independente, de alta confiabilidade e recursos completos. SQLite é o mecanismo de banco de dados mais usado no mundo, estando integrado em todos os telefones celulares e na maioria dos computadores, e vem integrado em inúmeros outros aplicativos que as pessoas usam todos os dias (ex.: Skype, RedHat, Python, PHP, FireFox, Microsoft, McAffe, Google especialmente nos celulares Android, dentre outros).

O formato de arquivo SQLite é estável, multiplataforma e compatível com versões anteriores e os desenvolvedores se comprometem a mantê-lo assim até o ano de 2050. Arquivos de banco de dados SQLite são comumente usados como contêineres para transferir conteúdo rico entre

sistemas e como um formato de arquivo de longo prazo para dados. Existem mais de 1 trilhão de bancos de dados SQLite em uso ativo. O código-fonte do SQLite é de domínio público e pode ser usado gratuitamente por todos para qualquer propósito.
Já o DB Browser for SQLite (DB4S) é uma ferramenta de código aberto, visual e de alta qualidade para criar, projetar e editar arquivos de banco de dados compatíveis com SQLite.
DB4S é para usuários e desenvolvedores que desejam criar, pesquisar e editar bancos de dados. DB4S usa uma interface familiar, semelhante a uma planilha, e comandos SQL complicados não precisam ser aprendidos.
Os controles e assistentes estão disponíveis para os usuários:

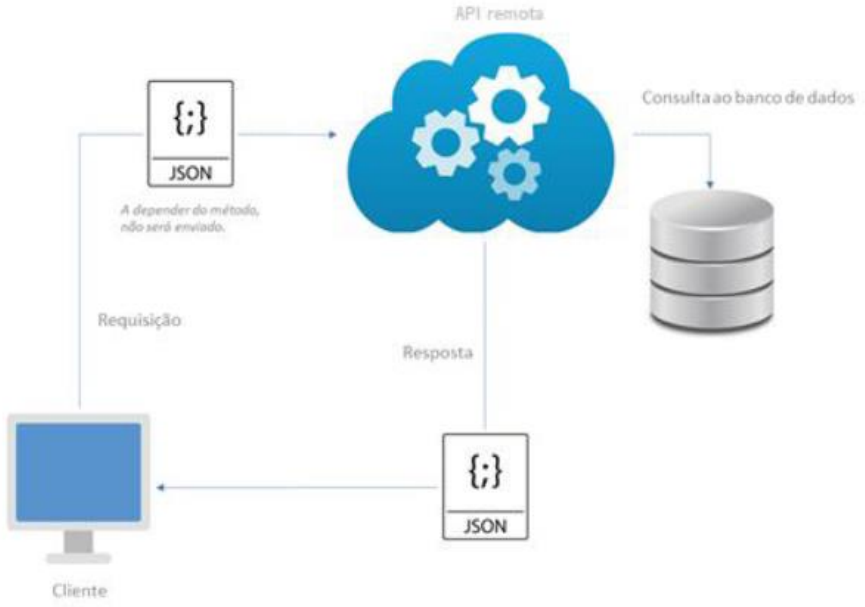
- Criar e compactar arquivos de banco de dados;
- Criar, definir, modificar e excluir tabelas;
- Crie, defina e exclua índices;
- Navegar, editar, adicionar e excluir registros;
- Registros de busca;
- Importar e exportar registros como texto;
- Importar e exportar tabelas de / para arquivos CSV;
- Importar e exportar bancos de dados de / para arquivos de despejo SQL;
- Emita consultas SQL e inspecione os resultados;
- Examine um registro de todos os comandos SQL emitidos pelo aplicativo.

**Fontes:**

https://www.sqlite.org
https://sqlitebrowser.org/

## 2. Entendendo como as tecnologias se relacionam

O cenário a seguir ilustra como as tecnologias se relacionam:




A API remota foi desenvolvida utilizando a linguagem Python 3x e o framework Flask para lidar com as solicitações HTTP. Sempre que houver uma solicitação, o código responderá de acordo com o método (ou verbo) invocado. A ideia foi definir uma API para o controle de produtos de uma loja qualquer, apenas com os dados básicos. Um cliente qualquer poderá interagir com a API chamando o verbo correspondente.

Observe os trechos do código, que definem respectivamente a rota (url) que deverá ser acessada, possíveis parâmetros e o verbo HTTP que deverá ser atendido (GET, POST, DELETE e PUT). Além disso, cada rota está associada a uma função que implementa as ações necessárias. No trecho a seguir, apenas os cabeçalhos das respectivas funções são mostrados.
O comando **@app.route(*param1*, *param2*)** pertence à biblioteca Flask e pede dois parâmetros: rota e verbo HTTP. Após o comando, uma função em Python deve ser implementada para atender ao verbo HTTP. Perceba que temos apenas uma única rota, variando apenas o método (boas práticas de API REST):

```
@app.route('/api-loja/produtos/', methods=['GET'])
@app.route('/api-loja/produtos/<int:idproduto>', methods=['GET'])
```

```
@app.route('/api-loja/produtos', methods=['POST'])
```

```
@app.route('/api-loja/produtos/<int:idproduto>', methods=['DELETE'])
```

```
@app.route('/api-loja/produtos/', methods=['PUT'])
```

A seguir, o código completo para ser analisado. Observe a interação com o banco SQLite que deverá ficar na pasta "*database*", dentro do diretório da aplicação:

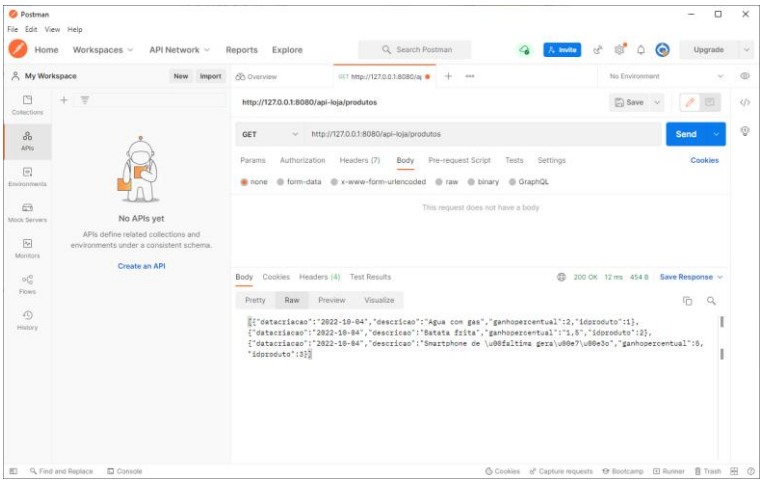
```
'''

Este código é uma API para "produtos", entretanto apenas alguns conceitos
de REST são implementados.

O intuito é realizar ajustes na tentativa de transformá-la em uma API RESTFul!
Utilize como referência o documento "Design da API Restful" disponibilizado.

'''

import sqlite3
from sqlite3 import Error
from flask import Flask, request, jsonify
from datetime import date


app = Flask(__name__)

########################################################
#  1) GET: pesquisar
########################################################

@app.route('/api-loja/produtos/', methods=['GET'])
@app.route('/api-loja/produtos/<int:idproduto>', methods=['GET'])
def pesquisar(idproduto=None):
    if request.method == 'GET':
        if idproduto is not None:
            sql = '''SELECT * FROM produtos WHERE idproduto = ''' + str(idproduto)
        else:
            sql = '''SELECT * FROM produtos'''


        try:
            conn = sqlite3.connect('database/db-loja.db')

            cur = conn.cursor()
            cur.execute(sql)
            registros = cur.fetchall()

            if registros:
                nomes_colunas = [x[0] for x in cur.description]

                json_dados = []
                for reg in registros:
                    json_dados.append(dict(zip(nomes_colunas, reg)))

                return jsonify(json_dados)
            else:
                return jsonify({'mensagem': 'registro nao encontrado'})

        except Error as e:
            return jsonify({'mensagem': e})
        finally:
            conn.close()
```

```
#######################################################
#  2) POST: inserir
#######################################################

@app.route('/api-loja/produtos', methods=['POST'])
def inserir():
    if request.method == 'POST':
        dados = request.get_json()

        descricao = dados['descricao']
        ganhopercentual = dados['ganhopercentual']
        datacriacao = date.today()

        if descricao and ganhopercentual and datacriacao:
            registro = (descricao, ganhopercentual, datacriacao)

            try:
                conn = sqlite3.connect('database/db-loja.db')

                sql = ''' INSERT INTO produtos(descricao, ganhopercentual, datacriacao)
VALUES(?,?,?) '''
                cur = conn.cursor()
                cur.execute(sql, registro)
                conn.commit()

                return jsonify({'mensagem': 'registro inserido com sucesso'})

            except Error as e:
                return jsonify({'mensagem': e})
            finally:
                conn.close()

        else:
            return jsonify({'mensagem': 'campos <descricao> e <ganhopercentual> sao
obrigatorios'})


#######################################################
#  3) DELETE: excluir
#######################################################

@app.route('/api-loja/produtos/<int:idproduto>', methods=['DELETE'])
def excluir(idproduto):
    if request.method == 'DELETE':
        if idproduto:
            try:
                conn = sqlite3.connect('database/db-loja.db')

                sql = '''DELETE FROM produtos WHERE idproduto = ''' + str(idproduto)

                cur = conn.cursor()
                cur.execute(sql)

                conn.commit()

                return jsonify({'mensagem': 'registro excluido'})

            except Error as e:
                return jsonify({'mensagem': e})
            finally:
                conn.close()
        else:
            return jsonify({'mensagem': 'campo <idproduto> obrigatorio'})


#######################################################
#  4) PUT: alterar
#######################################################

@app.route('/api-loja/produtos/', methods=['PUT'])
def alterar():
    if request.method == 'PUT':
        dados = request.get_json()
```

```
        descricao = dados['descricao']
        ganhopercentual = dados['ganhopercentual']
        idproduto = dados['idproduto']

        if descricao and ganhopercentual and idproduto:
            registro = (descricao, ganhopercentual, idproduto)

            try:
                conn = sqlite3.connect('database/db-loja.db')

                sql = ''' UPDATE produtos SET descricao=?, ganhopercentual=? WHERE
idproduto = ?'''
                cur = conn.cursor()
                cur.execute(sql, registro)
                conn.commit()

                return jsonify({'mensagem': 'registro alterado com sucesso'})

            except Error as e:
                return jsonify({'mensagem': e})
            finally:
                conn.close()

    else:
        return jsonify({'mensagem': 'campos <descricao>, <ganhopercentual> e <idproduto>
sao obrigatorios'})


##########################################################
# 5) UrlPoint nao localizado
##########################################################

@app.errorhandler(404)
def endpoint_nao_encontrado(e):
    return jsonify({'mensagem': 'erro - endpoint nao encontrado'}), 404

@app.errorhandler(405)
def endpoint_nao_encontrado(e):
    return jsonify({'mensagem': 'erro - endpoint nao encontrado'}), 405


##########################################################
# XX Execucao da Aplicacao
##########################################################

if __name__ == '__main__':
    app.run(host='0.0.0.0', port='8080')
```

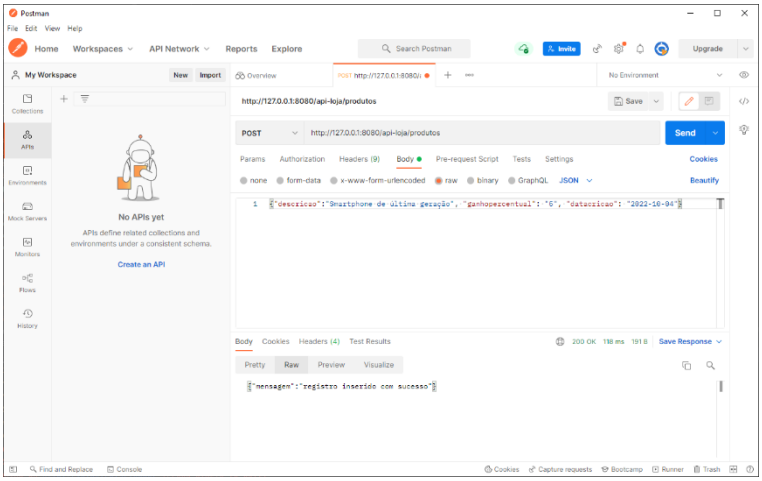
Caso queira testar o consumo da API via Postman ou código Python, aqui vai a dica:

**GET**



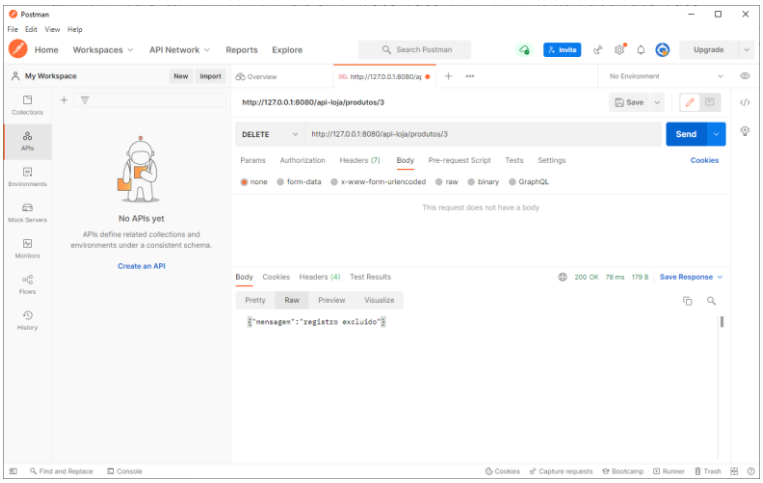
```
import json
import requests

url = 'http://127.0.0.1:8080/api-loja/produtos/'

r = requests.get(url)

print(r.text)
```

**POST**



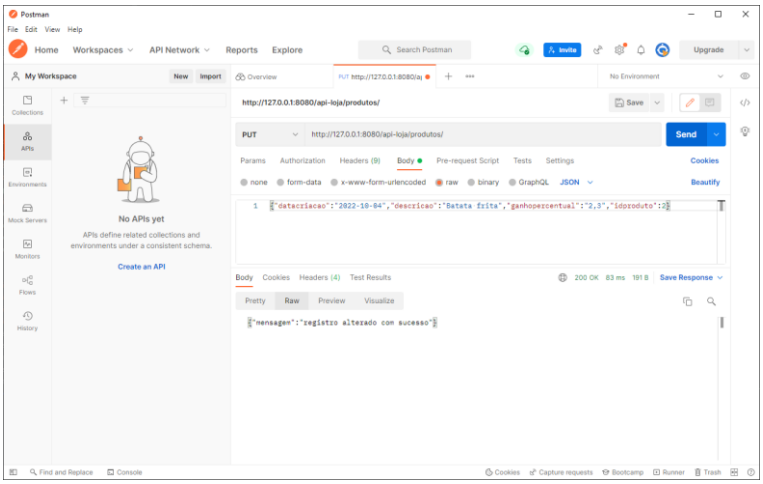
```
import json
import requests

dado = {'descricao':'picanha maturada argentina','ganhopercentual': '3.5'}
dado_json = json.dumps(dado)

url = 'http://127.0.0.1:8080/api-loja/produtos'

r = requests.post(url, data=dado_json, headers={'content-type': 'application/json'})

print(r.text)
```

**DELETE**




```
import json
import requests

url = 'http://127.0.0.1:8080/api-loja/produtos/1'

r = requests.delete(url)

print(r.text)
```

**PUT**




```
import json
import requests

dado = {'descricao':'baralho para truco','ganhopercentual': '2','idproduto': '2'}
dado_json = json.dumps(dado)

url = 'http://127.0.0.1:8080/api-loja/produtos'

r = requests.put(url, data=dado_json, headers={'content-type': 'application/json'})

print(r.text)
```