

Projeto FlaskAPI

1 - Introdução ao Flask

Lançado em 2010 e desenvolvido por Armin Ronacher, o Flask é um micro-framework destinado principalmente a pequenas aplicações com requisitos mais simples, como por exemplo, a criação de um site básico.



Possui um núcleo simples e expansível que permite que um projeto possua apenas os recursos necessários para sua execução (conforme surja a necessidade, um novo pacote pode ser adicionado para incrementar as funcionalidades da aplicação).

1.1 - Características do Flask

1.1.1 - Simplicidade

Por possuir apenas o necessário para o desenvolvimento de uma aplicação, um projeto escrito com Flask é mais simples se comparado aos frameworks maiores, já que a quantidade de arquivos é muito menor e sua arquitetura é muito mais simples.

1.1.2 - Rapidez no Desenvolvimento

Com o Flask, o desenvolvedor se preocupa em apenas desenvolver o necessário para um projeto, sem a necessidade de realizar configurações que muitas vezes não são utilizadas.

1.1.3 - Projetos Menores

Por possuir uma arquitetura muito simples (um único arquivo inicial) os projetos escritos em Flask tendem a ser menores e mais leves se comparados a frameworks maiores.

1.1.4 - Aplicações Robustas

Apesar de ser um micro-framework, o Flask permite a criação de aplicações robustas, já que é totalmente personalizável, permitindo, caso necessário, a criação de uma arquitetura mais definida.

1.2 - Pré-requisitos

Antes de começar, você vai precisar ter o seguinte instalado em sua máquina:

- [Python 3.6+](#)
- [Pip](#)

2 - Preparação do Ambiente

Primeiro, [baixe o repositório](#) e copie a pasta "flaskAPI" para a área de trabalho.

3 - Instalação do PIP

Pip é um sistema de gerenciamento de pacotes padrão, usado para instalar e gerenciar pacotes de software escritos em Python.

Para instalar o pip no Linux, abra o terminal e digite/cole o seguinte comando:

```
sudo apt install python3-pip
```

4 - Configuração do Ambiente Virtual

Um ambiente virtual é uma instância independente do Python, que permite isolar as bibliotecas e dependências de um projeto específico. Isso é útil para evitar conflitos entre diferentes versões de pacotes e garantir que o projeto seja executado consistentemente, independentemente do ambiente de desenvolvimento.

4.1 - Instalando a biblioteca venv

O *venv* é uma biblioteca Python padrão, capaz de criar ambiente virtual, disponível nas versões 3.3 e posteriores.

Para instalar a biblioteca venv no Windows, Linux ou Mac, abra o terminal em qualquer local e use o seguinte comando:

```
sudo apt install python3-venv
```

4.2 - Criando o Ambiente Virtual

Dentro do diretório do seu projeto, onde está localizado o arquivo *main.py*, *sendo este o diretório raiz do projeto*, crie um ambiente virtual:

Observação: O **nome** do ambiente virtual pode ser de sua escolha, neste projeto o nome será **ambiente-virtual**

```
python3 -m venv ambiente-virtual
```

4.3 - Ativando o Ambiente Virtual

4.3.1 - Windows

Para ativar o ambiente virtual no Windows, use um dos seguintes comandos, dependendo do seu terminal:

*Observação: O **Visual Studio Code** no **Windows** usa o **Terminal PowerShell***

Terminal normal

```
.\ambiente-virtual\Scripts\activate.bat
```

Terminal PowerShell

```
.\ambiente-virtual\Scripts\Activate.ps1
```

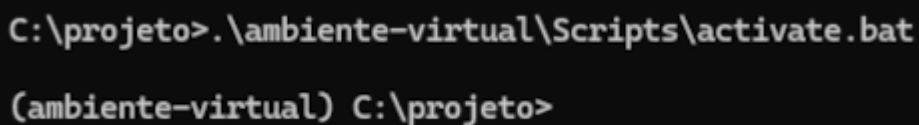
4.3.2 - Linux e Mac

Para ativar o ambiente virtual no Linux ou Mac, use o seguinte comando:

```
source ambiente-virtual/bin/activate
```

4.3.3 - Ambiente virtual ativo

Como deve ficar se o ambiente virtual estiver ativo:



```
C:\projeto>.\ambiente-virtual\Scripts\activate.bat  
(ambiente-virtual) C:\projeto>
```



```
md@MD:~/projeto$ source ambiente-virtual/bin/activate  
(ambiente-virtual) md@MD:~/projeto$ |
```

4.4 - Desativando o Ambiente Virtual

Para desativar o ambiente virtual no Windows, Linux ou Mac, use:

```
deactivate
```

5 - Instalando bibliotecas

Dentro do arquivo *requirements.txt* deve conter todas as bibliotecas que vamos utilizar dentro do projeto, para não ter a necessidade de instalar uma por uma manualmente. Vamos instalar todas juntas, pedido para o pip instalar todas bibliotecas que foram listadas dentro do arquivo, necessário informar o nome e se quiser, a versão da biblioteca.

5.1 - Lista de bibliotecas

Abrir o arquivo *requirements.txt*, que está no diretório *flaskAPI/src/requirements.txt* e digite/cole a lista de bibliotecas utilizadas:

```
Flask==3.0.3
Flask-HTTPAuth==4.8.0
SQLAlchemy==2.0.31
PyMySQL==1.1.1
requests==2.32.3
responses==0.25.3
```

5.2 - Comando de instalação

Para instalar as bibliotecas no ambiente virtual, digite/cole o seguinte comando:

Atenção: Este comando deve ser executado dentro do ambiente virtual, caso contrario, todas as bibliotecas serão instaladas na maquina.

```
pip install -r requirements.txt
```

6 - SQL MariaDB

O *MariaDB* Server é um dos bancos de dados relacionais de código aberto mais populares. Ele é feito pelos desenvolvedores originais do *MySQL* e tem garantia de permanecer de código aberto. Ele faz parte da maioria das ofertas de nuvem e é o padrão na maioria das distribuições Linux.

6.1 - Código SQL MariaDB

Salvar o código SQL MariaDB no arquivo *script.sql* no diretório *flaskAPI/sql/script.sql*

```
-- Adminer 4.8.1 MySQL 11.3.2-MariaDB-1:11.3.2+maria~ubu2204 dump

SET NAMES utf8;
SET time_zone = '-03:00';
SET foreign_key_checks = 0;
SET sql_mode = 'NO_AUTO_VALUE_ON_ZERO';

-- -----
-- Database db
```

```
-- -----  
CREATE DATABASE IF NOT EXISTS `db-flask` DEFAULT CHARACTER SET utf8 ;  
USE `db-flask`;  
  
-- -----  
-- Tabela db.usuario  
-- -----  
  
CREATE TABLE IF NOT EXISTS `usuario` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `nome` varchar(100) DEFAULT 'Não Informado',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 COLLATE=utf8mb3_general_ci;  
  
-- -----  
-- Tabela db.produto  
-- -----  
  
CREATE TABLE IF NOT EXISTS `produto` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `usuarioId` int(11) NOT NULL,  
  `nome` varchar(100) DEFAULT 'Não Informado',  
  PRIMARY KEY (`id`),  
  KEY `fk_produto_usuario` (`usuarioId`),  
  CONSTRAINT `fk_produto_usuario` FOREIGN KEY (`usuarioId`) REFERENCES `usuario`  
    (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 COLLATE=utf8mb3_general_ci;
```

7 - Ambiente Docker

7.1 - Novos containers Docker

7.1.1 - Código YAML

Abrir o arquivo `docker-compose.yaml` no diretório `flaskAPI/docker-compose.yaml` e digitar/colar o seguinte código:

```
version: '3.1'  
services:  
  mariadb:  
    container_name: mariadbflask  
    image: mariadb:11.3.2  
    restart: always  
    volumes:  
      - ./sql/script.sql:/docker-entrypoint-initdb.d/script.sql  
    environment:  
      - MARIADB_DATABASE=db-flask  
      - MARIADB_USER=root
```

```
- MARIADB_ROOT_PASSWORD=123456789
ports:
- "3305:3306"

adminer:
  image: adminer:4.8.1
  container_name: adminerflask
  restart: always
  ports:
  - 8081:8080
```

7.1.2 - Criar containers Docker

Para criar os novos containers, abrir o terminal no diretório `flaskAPI/` que contém o arquivo `docker-compose.yaml` e depois digitar/colar o seguinte comando:

```
docker compose up -d
```

7.1.3 - Visualizar os novos containers Docker

```
docker ps -a
```

Caso não tenha ocorrido nenhum erro, vá para o tópico **8**

7.2 - Correções de erro

7.2.1 - Listar containers Docker

```
docker ps -a
```

7.2.2 - Parar container Docker

Substituir o `ContainerID` pelo que será exibido no terminal, sendo necessário apenas as primeiras letras/numeros do `Container Id`

```
docker stop ContainerID
```

7.2.3 - Deletar container Docker

Substituir o `ContainerID` pelo que será exibido no terminal, sendo necessário apenas as primeiras letras/numeros do `Container Id`

```
docker rm ContainerID
```

Após deletar containers, volte no tópico **7.1**

8 - Início do projeto

Abra a arquivo `main.py` no diretório `flaskAPI/src/main.py` e digitar/colar o seguinte código:

```
from flask import Response, request
from routes import flaskRoutes

app = flaskRoutes()

##### Inicia execução do Flask API #####
if __name__ == "__main__":
    app.run(debug=True)
```

9 - SQLAlchemy

SQLAlchemy é um ORM (*Object Relational Mapper*), ORM é uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que os mesmos representam.

9.1 - Connection

No arquivo `connection.py`, localizado em `flaskAPI/src/config/connection.py` digitar/colar o código necessário para estabelecer uma conexão com o banco de dados:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

##### Declaração de variaveis de configurações de conexão #####
# "dialect+driver://user:password@host/dbname"
# "mariadb+pymysql://root:123456789@localhost:3309/db-flask"

dialect = "mariadb"
driver = "pymysql"
user = "root"
password = "123456789"
host = "localhost"
port = "3309"
dbname = "db-flask"

##### Conexão ao banco de dados #####
```

```
engine = create_engine(f"{dialect}+{driver}://{user}:{password}@{host}:{port}/{dbname}")
Session = sessionmaker(bind=engine)
```

9.2 - Model

9.2.1 - DTO

DTO (Data Transfer Object) é um padrão de design usado para transferir dados entre subsistemas ou camadas de uma aplicação.

9.2.1.1 - Usuário

Abrir o arquivo `usuario.py` no diretório `flaskAPI/src/model/dto/usuario.py` e digitar/colar o código de criação da classe Usuário:

```
from sqlalchemy import Column, String, Integer
from sqlalchemy.ext.declarative import declarative_base

##### Base para reconhecimento do ORM #####
Base = declarative_base()

##### Criação da classe Usuario #####
class Usuario(Base):
    __tablename__ = "usuario"

    ##### Atributos #####
    id = Column(Integer, primary_key=True, autoincrement=True)
    nome = Column(String, default="Usuario não informado")

    ##### Padrão da mensagem de retorno para o usuário #####
    def to_json(self):
        return {"id":self.id, "nome":self.nome}
```

9.2.1.1 - Produto

Abrir o arquivo `produto.py` no diretório `flaskAPI/src/model/dto/produto.py` e digitar/colar o código de criação da classe Produto:

```
from sqlalchemy import Column, String, Integer, ForeignKey
from sqlalchemy.orm import relationship
from model.dto.usuario import Base

##### Criação da classe produto #####
class Produto(Base):
    __tablename__ = "produto"
```



```
##### Atributos #####
id = Column(Integer, primary_key=True)
usuarioId = Column(Integer, ForeignKey("usuario.id"))
nome = Column(String, default="Não informado")
usuario = relationship("Usuario", lazy="subquery")

##### Padrão da mensagem de retorno para o usuário #####
def to_json(self):
    return {"id":self.id, "usuario":{"id":self.usuario.id,
"nome":self.usuario.nome}, "nome":self.nome}
```

9.2.2 - DAO

DAO (Data Access Object) são responsáveis por trocar informações com o SGBD e fornecer operações CRUD

9.2.2.1 - Usuário

Abrir o arquivo `usuario.py` no diretório `flaskAPI/src/model/dao/usuario.py` e digitar/colar o código interação com o bando de dados:

```
from config.connection import Session
from model.dto.usuario import Usuario
from model.dto.produto import Produto

##### Insert no bando de dados #####
def inserirUsuario(usuario):
    session = Session()
    novaUsuario = Usuario(nome=usuario.nome)
    session.add(novaUsuario)
    session.commit()
    session.close()
    return novaUsuario

##### Select Where no bando de dados #####
def listarUsuario(id):
    session = Session()
    usuario = session.query(Usuario).filter(Usuario.id==id).first()
    session.close()
    return usuario

##### Select All no bando de dados #####
def listarUsuarios():
    session = Session()
    usuario = session.query(Usuario).all()
    session.close()
    return usuario

##### Update Where no bando de dados #####
```

```

def editarUsuario(usuario):
    session = Session()
    session.query(Usuario).filter(Usuario.id==usuario.id).update({"nome":
usuario.nome})
    session.commit()
    session.close()
    return True

##### Delete Where no bando de dados #####
def deletarUsuario(id):
    usuario = listarUsuario(id)
    ##### Verifica se existe usuário #####
    if usuario:
        ##### Verifica se usuário é vinculado em um produto #####
        session = Session()
        verifica = session.query(Produto).filter(Produto.usuarioId==id).first()
        session.close()
        ##### Se for vinculado, não faz nada #####
        if verifica:
            return False
        else:
            ##### Se não for vinculado, deleta #####
            session = Session()
            session.query(Usuario).filter(Usuario.id==usuario.id).delete()
            session.commit()
            session.close()
            return True
    else:
        ##### Não existe usuário #####
        return None

```

9.2.2.1 - Produto

Abrir o arquivo `produto.py` no diretório `flaskAPI/src/model/dto/produto.py` e digitar/colar o código interação com o bando de dados:

```

from config.connection import Session
from model.dto.produto import Produto

##### Insert no bando de dados #####
def inserirProduto(produto):
    session = Session()
    novoProduto = Produto(usuarioId=produto.usuarioId, nome=produto.nome)
    session.add(novoProduto)
    session.commit()
    session.close()
    return novoProduto

##### Select Where no bando de dados #####

```

```
def listarProduto(id):
    session = Session()
    produto = session.query(Produto).filter(Produto.id==id).first()
    session.close()
    return produto

##### Select All no bando de dados #####
def listarProdutos():
    session = Session()
    produto = session.query(Produto).all()
    session.close()
    return produto

##### Update Where no bando de dados #####
def editarProduto(id, nome):
    ##### verifica de o produto existe #####
    produto = listarProduto(id)
    if produto == None:
        return False
    else:
        ##### produto encontrado #####
        produto.id = id
        produto.nome = nome
        session = Session()

    session.query(Produto).filter(Produto.id==produto.id).update({"nome":produto.nome}
    )
    session.commit()
    session.close()
    return produto

##### Delete Where no bando de dados #####
def deletarProduto(id):
    ##### verifica de o produto existe #####
    produto = listarProduto(id)
    if produto == None:
        return False
    else:
        ##### produto encontrado #####
        session = Session()
        session.query(Produto).filter(Produto.id==produto.id).delete()
        session.commit()
        session.close()
        return True
```

10 - Configuração de acesso

Abra o arquivo `auth.py` no diretório `flaskAPI/src/config/auth.py` para permitir a autorização de acesso a API RESTfull, digitar/colar o seguinte código:

```
from werkzeug.security import generate_password_hash, check_password_hash
from flask_httpauth import HTTPBasicAuth

auth = HTTPBasicAuth()

##### usuário e senha de acesso a API #####
users = {
    "admin": generate_password_hash("123"),
}

##### verificação de usuário e senha de acesso a API #####
@auth.verify_password
def verify_password(username, password):
    if username in users and check_password_hash(users.get(username), password):
        return username
    return None
```

11 - Routes

Rotas que a API RESTfull irá utilizar para o GET, POST, PUT e DELETE

11.1 - Mapeamento das rotas

Abra a arquivo `__init__.py` no diretório `flaskAPI/src/routes/` e digitar/colar o seguinte código:

```
from flask import Flask

##### Mapeia as rotas #####
def flaskRoutes():
    app = Flask(__name__)

    ##### Rotas Usuário #####
    from .usuario.usuario import bp as usuario_bp
    app.register_blueprint(usuario_bp, url_prefix='/usuario')

    ##### Rotas Produto #####
    from .produto.produto import bp as produto_bp
    app.register_blueprint(produto_bp, url_prefix='/produto')

    ##### Retorna as rotas #####
    return app
```

11.2 - Usuário

11.2.1 - Mapeamento usuário

Abra a arquivo `__init__.py` no diretório `flaskAPI/src/routes/usuario/` e digitar/colar o seguinte código:

```
from flask import Blueprint

bp = Blueprint('usuario', __name__)

from . import usuario
```

11.2.2 - Rota Usuário

Abra a arquivo `usuario.py` no diretório `flaskAPI/src/routes/usuario/` e digitar/colar o seguinte código:

```
import json
from flask import Blueprint, Response, jsonify, request
from config.auth import auth
from model.dto.usuario import Usuario
from model.dao.usuario import listarUsuario, listarUsuarios, inserirUsuario,
editarUsuario, deletarUsuario

bp = Blueprint('usuario', __name__)

##### inserir novo produto #####
@bp.route('/usuario', methods=['POST'])
@auth.login_required
def inserirNovoUsuario():
    try:
        body = request.get_json()
        usuario = Usuario(nome=body["nome"])
        inserirUsuario(usuario)
        return geraResponse(201, "Usuario", usuario.to_json(), "Usuario inserido
com sucesso")
    except Exception as error:
        return geraResponse(400, "Usuario", {}, f"Erro ao inserir: {error}")

##### listar um produto #####
@bp.route('/usuario/<id>', methods=['GET'])
@auth.login_required
def listarUmUsuario(id):
    try:
        lista = listarUsuario(id)
        if lista == None:
            return geraResponse(400, "Usuario", {}, "Usuario não encontrado")
        else:
            listarJson = lista.to_json()
```

```
        return geraResponse(200, "Usuario", listarJson)
    except Exception as error:
        return geraResponse(400, "Usuario", {}, f"Erro ao listar usuarios:
{error}")

##### listar produtos #####
@bp.route('/usuarios', methods=['GET'])
@auth.login_required
def listarTodosUsuarios():
    try:
        lista = listarUsuarios()
        listaJson = [item.to_json() for item in lista]
        return geraResponse(200, "Usuarios", listaJson)
    except Exception as error:
        return geraResponse(400, "Usuario", {}, f"Erro ao listar usuarios:
{error}")

##### editar um produto #####
@bp.route('/usuario/<id>', methods=['PUT'])
@auth.login_required
def editarUmUsuario(id):
    try:
        usuario = listarUsuario(id)
        body = request.get_json()
        if('nome' in body):
            usuario.nome = body['nome']
            aux = editarUsuario(usuario)
            if aux:
                return geraResponse(200, "Usuario", usuario.to_json(), "Usuario
editado com sucesso")
            else:
                return geraResponse(400, "Usuario", {}, "Erro ao editar usuario")
    except Exception as error:
        return geraResponse(400, "Usuario", {}, f"Erro ao editar usuario:
{error}")

##### deletar um produto #####
@bp.route('/usuario/<id>', methods=['DELETE'])
@auth.login_required
def deletarUmUsuario(id):
    try:
        deleta = deletarUsuario(id)
        if deleta: ##### True #####
            return geraResponse(200, "Usuario", {}, "Usuario deletado com
sucesso")
        elif deleta == False: ##### False #####
            return geraResponse(400, "Usuario", {}, "Usuario está vinculado à um
produto")
        else: ##### None #####
            return geraResponse(400, "Usuario", {}, "Usuario não existe")
    except Exception as error:
```

```

        return geraResponse(400, "Usuario", {}, f"Erro ao deletar usuario:
{error}")

##### Gera resposta em json para o usuário #####
def geraResponse(status, nomeConteudo, conteudo, mensagem=False):
    body = {}
    body[nomeConteudo] = conteudo
    if(mensagem):
        body["mensagem"] = mensagem
    return Response(json.dumps(body), status=status, mimetype="application/json")

```

11.3 - Produto

11.3.1 - Mapeamento produto

Abra a arquivo `__init__.py` no diretório `flaskAPI/src/routes/produto/` e digitar/colar o seguinte código:

```

from flask import Blueprint

bp = Blueprint('produto', __name__)

from . import produto

```

11.3.2 - Rota Produto

Abra a arquivo `produto.py` no diretório `flaskAPI/src/routes/produto/` e digitar/colar o seguinte código:

```

import json
from flask import Blueprint, Response, request
from config.auth import auth
from model.dto.produto import Produto
from model.dao.produto import listarProduto, listarProdutos, inserirProduto,
editarProduto, deletarProduto

bp = Blueprint('produto', __name__)

##### inserir novo produto #####
@bp.route('/produto', methods=['POST'])
@auth.login_required
def inserirNovoProduto():
    try:
        body = request.get_json()
        produto = Produto(usuarioId=body["usuarioId"], nome=body["nome"])
        inserirProduto(produto)
        return geraResponse(201, "Produto", {}, "Produto inserido com sucesso")

```

```
except Exception as error:
    return geraResponse(400, "Produto", {}, f"Erro ao inserir novo produto:
{error}")

##### listar um produto #####
@bp.route('/produto/<id>', methods=['GET'])
@auth.login_required
def listarUmProduto(id):
    try:
        lista = listarProduto(id)
        if lista == None:
            return geraResponse(400, "Produto", {}, "Produto não encontrado")
        else:
            listarJson = lista.to_json()
            return geraResponse(200, "Produto", listarJson)
    except Exception as error:
        return geraResponse(400, "Produto", {}, f"Erro ao listar o produto:
{error}")

##### listar produtos #####
@bp.route('/produtos', methods=['GET'])
@auth.login_required
def listarTodosProdutos():
    try:
        lista = listarProdutos()
        listaJson = [item.to_json() for item in lista]
        return geraResponse(200, "Produtos", listaJson)
    except Exception as error:
        return geraResponse(400, "Produtos", {}, f"Erro ao listar produtos:
{error}")

##### editar um produto #####
@bp.route('/produto/<id>', methods=['PUT'])
@auth.login_required
def editarUmProduto(id):
    try:
        body = request.get_json()
        if('nome' in body):
            nome = body['nome']
            produto = editarProduto(id, nome)
            if produto:
                return geraResponse(200, "Produto", produto.to_json(), "Produto
editado com sucesso")
            else:
                return geraResponse(400, "Produto", {}, f"Produto de {id} não
existe")
        else:
            return geraResponse(400, "Produto", {}, "Informe nome do produto
corretamente")
    except Exception as error:
        return geraResponse(400, "Produto", {}, f"Erro ao editar produto:
```



```
{error}")

##### deletar um produto #####
@bp.route('/produto/<id>', methods=['DELETE'])
@auth.login_required
def deletarUmProduto(id):
    try:
        deleta = deletarProduto(id)
        if deleta:
            return geraResponse(200, "Produto", {}, "Produto deletado com
sucesso")
        elif deleta == False:
            return geraResponse(400, "Produto", {}, f"Produto de {id} não existe")
        else:
            return geraResponse(400, "Produto", {}, "Erro ao deletar produto")
    except Exception as error:
        return geraResponse(400, "Produto", {}, f"Erro ao deletar produto:
{error}")

##### Gera resposta em json para o usuário #####
def geraResponse(status, nomeConteudo, conteudo, mensagem=False):
    body = {}
    body[nomeConteudo] = conteudo
    if(mensagem):
        body["mensagem"] = mensagem
    return Response(json.dumps(body), status=status, mimetype="application/json")
```

12 - Inicializar API

No terminal, dentro o ambiente virtual, digitar/colar o seguinte comando:

Observação: certifique-se de que o diretório no terminal esteja sendo executado em **flaskAPI/src**

```
python main.py
```

13 - Consumir Flask API

Para consumir Flask API, lembre-se que é necessário informar os dados de **Authorization** (*autorização*) do tipo **Basic** no **Postman** ou **Insomnia**

Observação: Deve informar estes dados de autorização, para cada requisição GET, POST, PUT E DELETE

Username:

```
aluno
```

Password:

```
123
```

13.1 - Usuário

13.1.1 - POST

URL de acesso:

```
http://127.0.0.1:5000/usuario/usuario
```

Padrão *JSON* para envio de dados:

```
{  
  "nome": "Rafael"  
}
```

13.1.2 - GET

URL de acesso:

```
http://127.0.0.1:5000/usuario/usuario/1
```

13.1.3 - GET ALL

URL de acesso:

```
http://127.0.0.1:5000/usuario/usuarios
```

13.1.4 - PUT

URL de acesso:

```
http://127.0.0.1:5000/usuario/usuario/1
```

Padrão *JSON* para envio de dados:

```
{  
  "nome": "Rafael Feitosa"  
}
```

13.1.5 - DELETE

URL de acesso:

```
http://127.0.0.1:5000/usuario/usuario/1
```

13.2 - Produto

13.2.1 - POST

URL de acesso:

```
http://127.0.0.1:5000/produto/produto
```

Padrão *JSON* para envio de dados:

```
{  
  "usuarioId": 1,  
  "nome": "Arroz"  
}
```

13.2.2 - GET

URL de acesso:

```
http://127.0.0.1:5000/produto/produto/1
```

13.2.3 - GET ALL

URL de acesso:

```
http://127.0.0.1:5000/produto/produtos
```

13.2.4 - PUT

URL de acesso:

```
http://127.0.0.1:5000/produto/produto/1
```

Padrão *JSON* para envio de dados:

```
{  
  "nome": "Melancia"  
}
```

13.2.5 - DELETE

URL de acesso:

```
http://127.0.0.1:5000/produto/produto/1
```