APIs REST

PUCRS

Pontifícia Universidade Católica
do Rio Grande do Sul

– Aula 09 –
Coleta, Preparação e Análise de Dados

Prof. Me. Lucas R. C. Pessutto

Adaptação dos slides dos profs. Luan Garcia e Lucas Kupssinskü

Application Programming Interface

- Uma GUI (Graphical User Interface) é uma forma de comunicação entre humanos e computadores.
  - É uma interface entre os dois.

- Uma API (Application Programming Interface) é uma forma de programas comunicarem entre si.

Service-Oriented Architecture

- SOA é um estilo de **arquitetura** de software.

- Funcionalidades de uma aplicação são disponibilizadas como serviços **individuais**, ao invés de um grande programa monolítico.

- Um serviço:
  - É uma **unidade lógica** representando uma atividade de negócio.
  - É autocontido.
  - Implementação é uma **caixa-preta** para os consumidores.
  - Pode ser composto de outros serviços.

Application Programming Interface

- API é uma descrição de alto nível de um conjunto de **serviços**.

- API ***especifica*** como deve ser o comportamento de um serviço.
  - Como chamar, quais parâmetros passar, qual retorno esperar, etc.
  - Ela é uma **descrição**, uma **interface**, não é uma implementação!

- Uma mesma API pode ter diversas implementações **diferentes**!
  - Uma para cada linguagem de programação, por exemplo.

REST – REpresentation State Transfer

- Também é um estilo de **arquitetura** de software.
- Tese de doutorado de Roy Fielding em 2000.
- Pensada como um arquitetura para **web** com objetivo de promover:
  - Escalabilidade de interação entre componentes.
  - Modularidade.
  - Uniformidade de interfaces.
  - Portabilidade.
  - Deployment independente de componentes.
  - Segurança.
- Sistema **consegue** essas propriedades **não-funcionais** seguindo alguns princípios de arquitetura.

Princípios ou restrições REST

- Arquitetura cliente-servidor:
  - Separa o consumidor de quem provê o serviço em um arquitetura baseada em redes.
- Stateless:
  - Servidor não mantém informação de sessão, cada comunicação é em teoria isolada e totalmente autocontida.
- Cacheability:
  - Clientes e nodos intermediários podem armazenar respostas como cache para evitar overhead de comunicação.
- Sistema em camadas:
  - Cliente não deve saber se está comunicando diretamente com servidor ou algum balanceador de carga.
- **Interface uniforme:**
  - Simplifica e desacopla arquitetura para permitir escalabilidade e evolução independente.
- Código sob demanda:
  - Servidor pode enviar código para ser executado no lado do cliente, como por exemplo scripts JavaScript.

Interface Uniforme REST

- Baseada na ideia de **recursos** da web.
- Identificação de recursos:
  - URI
- Manipulação de recursos:
  - São passadas representações que permitem manipular o recurso no servidor.
- Comunicação autocontida:
  - Mensagens trocadas incluem informação suficiente para descrever como processar a mensagem (Stateless!)

RESTful API

- É uma API web que adere aos princípios REST.
- REST é um estilo de arquitetura, não é um protocolo.
  - Não define como fazer.
- Que protocolo de comunicação usar?
  - HTTP
- Quais métodos implementar?
  - Métodos HTTP
- Como representar os recursos?
  - HTML, XML, **JSON**...

Tipos de métodos em uma API RESTful

- Comunicação é feita através de métodos HTTP:

Método HTTP	Função em uma API REST
GET	Recupera um recurso existente.
POST	Cria um novo recurso no servidor.
PUT	Atualiza um recurso existente no servidor.
DELETE	Deleta um recurso no servidor.
PATCH	Atualiza de forma parcial um recurso no servidor.

- Códigos de repostas HTTP:

Intervalo do Código	Categoria
2xx	Operação com sucesso.
3xx	Redirecionamento de recurso.
4xx	Erro no pedido do cliente.
5xx	Erro no servidor.

API Endpoint

- Uma API REST expõe endereços URL para que aplicações clientes possam acessar um serviço via web.
- Esses endereços são chamados de **Endpoints**.

- Exemplo de serviços disponibilizados em http://api.exemplo/

Método	Endpoint	Descrição
GET	.../clientes	Retorna uma lista de clientes
GET	.../clientes/<id_cliente>	Retorna o cliente com o id passado
POST	.../clientes	Cria novo cliente
PUT	.../clientes/<id_cliente>	Atualiza cliente com o id passado
DELETE	.../clientes/<id_cliente>	Deleta cliente

Consumo de APIs

- API web funciona através de mensagens HTTP
- Já vimos que podemos enviar mensagens HTTP com bibliotecas como urrlib ou requests no Python.

- Vamos ver como testar uma API de duas formas:
  - Postman
  - Python (notebook disponível no moodle)
- API que vamos testar:
  - https://jsonplaceholder.typicode.com

Criando APIs no Python

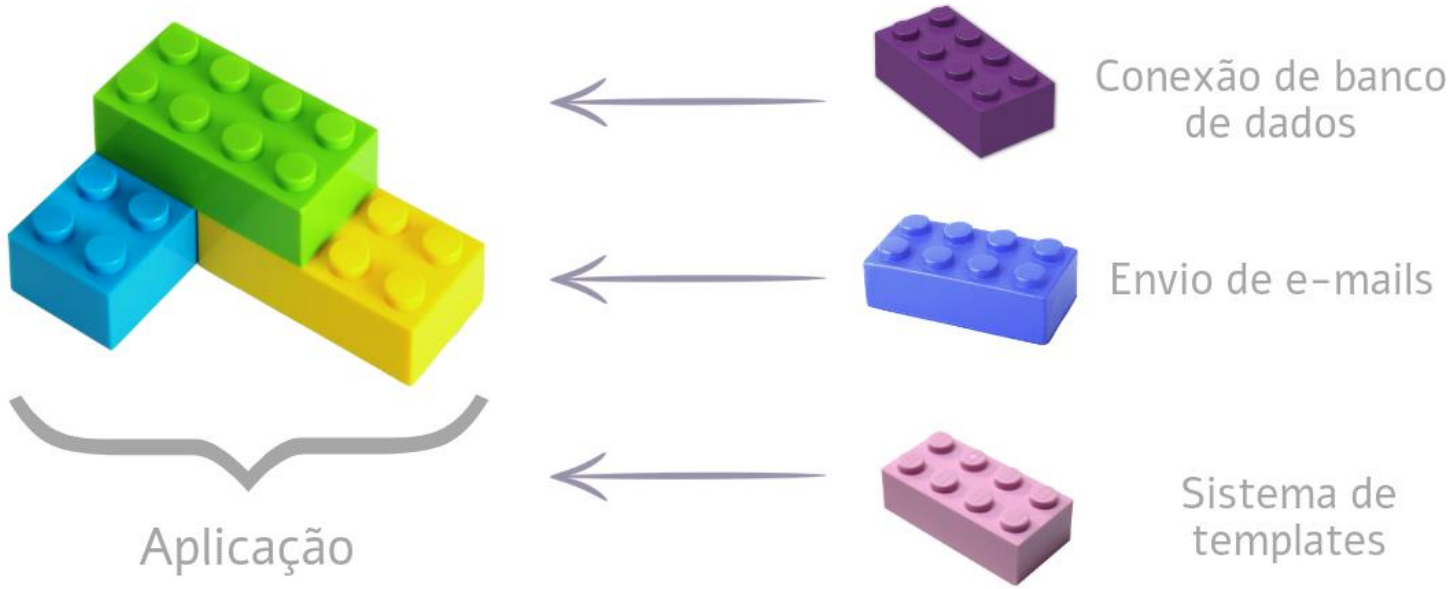
- Existem alguns frameworks para facilitar a disponibilização de serviços de um API em Python.

- Os mais famosos são:
  - Flask
  - Django
  - FastAPI

- Vamos utilizar o framework **Flask** porque é simples e possui excelente documentação.

Flask

* Flask é um micro-framework para desenvolvimento de aplicações web.
  - → Um micro-framework são Frameworks modularizados que possuem uma estrutura inicial muito mais simples quando comparado a um Framework convencional.



Flask

* Essa biblioteca é bastante utilizado para criação de microsserviços, como APIs RESTful.

* Características do Flask:
  - Simplicidade: Possui apenas o necessário para o desenvolvimento de uma aplicação
  - Rapidez no desenvolvimento: desenvolvedor se preocupa em apenas desenvolver o necessário para um projeto, sem a necessidade de realizar configurações que muitas vezes não são utilizadas.
  - Projetos menores: Projetos escritos em Flask tendem a ser menores e mais leves se comparados a frameworks maiores.
  - Aplicações robustas: Apesar de ser um micro-framework, o Flask permite a criação de aplicações robustas, já que é totalmente personalizável

Flask

- Flask possui dois componentes principais:
  - Werkzeug:
    - Sistema de Rotas
    - Debugging
    - Web Server Gateway Interface (WSGI)
  - Jinja2:
    - Mecanismo de Templates
  - Não possui suporte nativo para acesso a banco de dados, autenticação de usuários ou outros utilitários de alto nível (existem extensões para isso!)
  - Página web: https://flask.palletsprojects.com/

Quantas linhas você acha que são necessárias para ter um servidor rodando uma app flask?

Primeiro Projeto Flask

* Instale a biblioteca flask

```
pip install flask
```

* Cuidado! Esse comando deve rodar na instalação do Python que está no ambiente virtual do projeto

Primeiro Programa Flask

* Passo 1: Importar a classe Flask e criar uma instância dessa classe

```
from flask import Flask

app = Flask(__name__)
```

O argumento recebido pelo construtor da classe `Flask` é o nome do módulo principal da aplicação. Na maioria das vezes esse nome encontra-se na variável do Python `__name__`

Primeiro Programa Flask

* Passo 2: Criar as rotas da aplicação
  - Rotas servem para atender à requisições de clientes
  - A aplicação precisa saber que código rodar quando cada URL for chamada
  - O flask faz o mapeamento das URLs com as funções contendo seu código. Chamamos isso de rotas

Primeiro Programa Flask

* Passo 2: Criar as rotas da aplicação
  - Fazemos esse mapeamento adicionando um decorador na frente da função, indicando qual rota ela atende

```
@app.route('/')
def index():
    return "Olá Mundo"
```

A URL '/' é considerada a URL raiz da página Web. Ela é chamada quando você navega diretamente para a página inicial do site

Funções como esta também são chamadas de view functions. Elas podem retornar uma string simples, uma string contendo código HTML ou também dados no formato json

Primeiro Programa Flask

* Passo 3: Inicializar o servidor da aplicação

```
if __name__ == '__main__':
    app.run(debug=True)
```

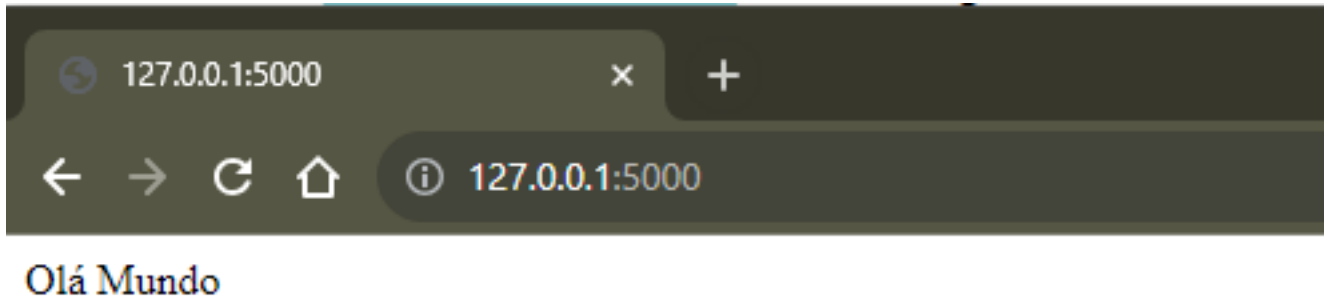
Exibe mensagens especiais no navegador caso haja algum erro

Rodando nosso programa

* Ao rodar o programa você deve ver algo como:

Rodando nosso programa

* Podemos ver a aplicação rodando acessando o site 127.0.0.1:5000 no navegador



Tipos de Requests HTTP

Verb	CRUD	Operation
GET	Read	Fetch a single or multiple resource
POST	Created	Insert a new resource
PUT	Update/ Create	Insert a new resource or update existing
DELETE	Delete	Delete a single or multiple resource
OPTIONS	READ	List allowed operations on a resource
HEAD	READ	Return only response headers and no body
PATCH	Update/ Modify	Only update the provided changes to the resource

Troca de Mensagens

Cliente envia um request:

```
GET /comments/postID/1/ HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
   OS X 10.15; rv:80.0) Gecko/20100101
   Firefox/80.0 \r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Connection: keep-alive\r\n
\r\n
```

Servidor gera um response:


```
[
    {
        "postId": 1,
        "id": 1,
        "name": "id labore ex et quam laborum",
        "email": "Eliseo@gardner.biz",
        "body": "laudantium enim quasi est quidem magnam voluptate"
    },
    {
        "postId": 1,
        "id": 2,
        "name": "quo vero reiciendis velit similique earum",
        "email": "Jayne_Kuhic@sydney.com",
        "body": "est natus enim nihil est dolore omnis voluptatem"
    },
    {
        "postId": 1,
        "id": 3,
        "name": "odio adipisci rerum aut animi",
        "email": "Nikita@garfield.biz",
        "body": "quia molestiae reprehenderit quasi aspernatur\naut"
    },
    {
        "postId": 1,
        "id": 4,
        "name": "alias odio sit",
        "email": "Lew@alysha.tv",
        "body": "non et atque\noccaecati deserunt quas accusantium"
```

Problema 01: TODO list

* Elaborar uma API rest para um programa que gerencie uma TODO list, que consiste em uma lista de strings.

* Devem ser previstas as quatro operações CRUD na API

Create

Retrieve

Update

Delete

TODO list: Planejamento

Método	Rota	CRUD	Ação
GET	`/tarefas`	Retrieve	Listar todas as tarefas existentes
GET	`/tarefa/<int:id>`	Retrieve	Listar uma tarefa, dado o id
POST	`/tarefas`	Create	Cria uma nova tarefa
PUT	`/tarefa/<int:id>`	Update	Edita uma tarefa
DELETE	`/tarefa/<int:id>`	Delete	Deleta uma tarefa

Rota /tarefas

* Passo 1: Definir quais métodos serão aceitos pela rota

```
lista_tarefas = []

@app.route('/tarefas', methods=['GET', 'POST'])
def tarefas():
    pass
```

Métodos HTTP aceitos por essa rota

Rota /tarefas

* Passo 2: Identificar o tipo de método da solicitação

```
from flask import Flask, request, abort

lista_tarefas = []

@app.route('/tarefas', methods=['GET', 'POST'])
def tarefas():
    if request.method == "GET":
        pass
    elif request.method == "POST":
        pass
    else:
        abort(404)
```

Rota /tarefas

* Passo 3: Implementar a rota GET

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

lista_tarefas = []

@app.route('/tarefas', methods=['GET', 'POST'])
def tarefas():
    if request.method == "GET":
        return make_response(jsonify(lista_tarefas))
    elif request.method == "POST":
        pass
    else:
        abort(404)
```

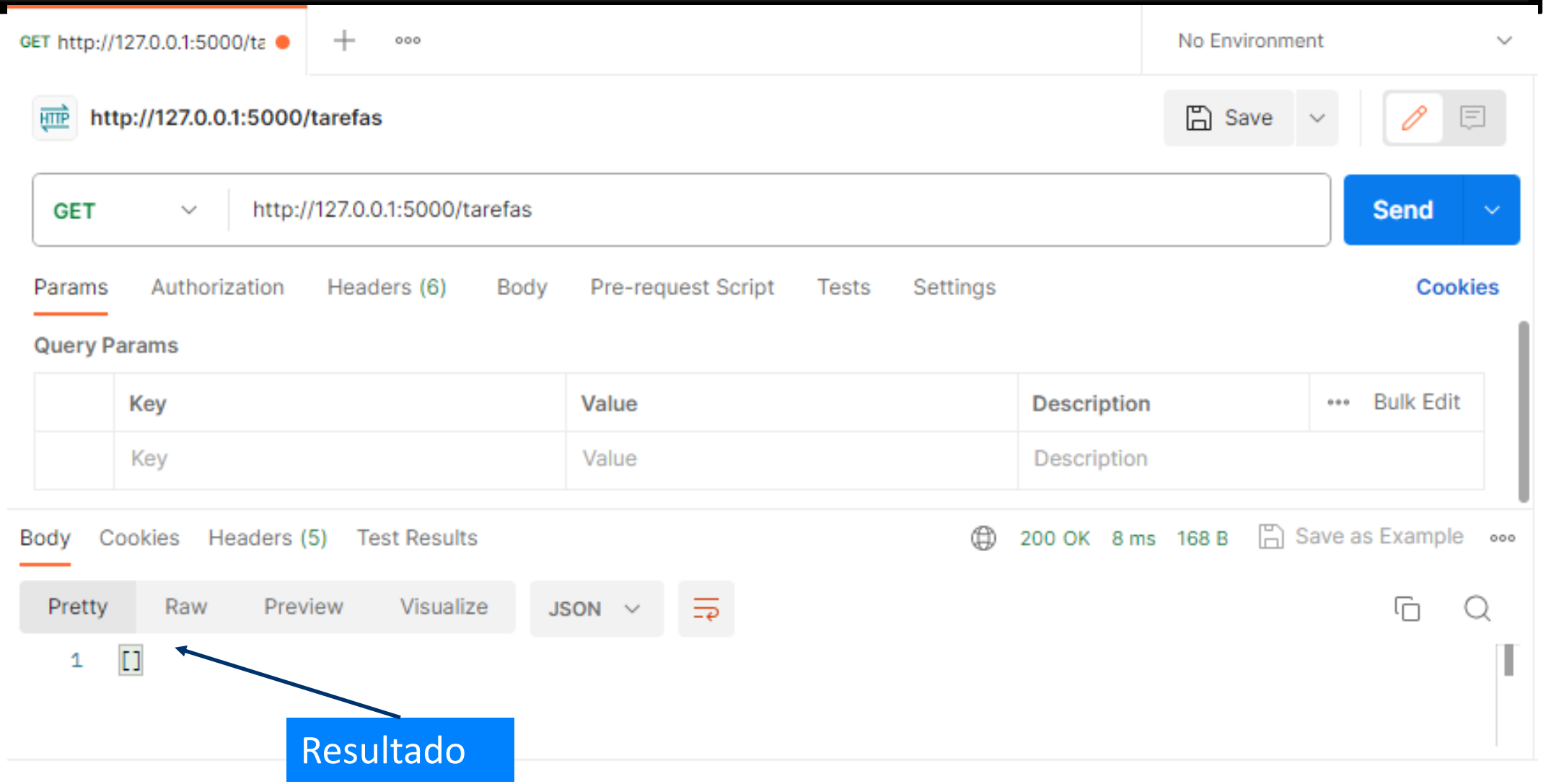
Rota /tarefas

* Passo 4: Testar a rota no Postman

  - Rodar o programa

  - Criar nova coleção no Postman
  - Fazer uma requisição do tipo GET para a rota tarefas

Rota /tarefas

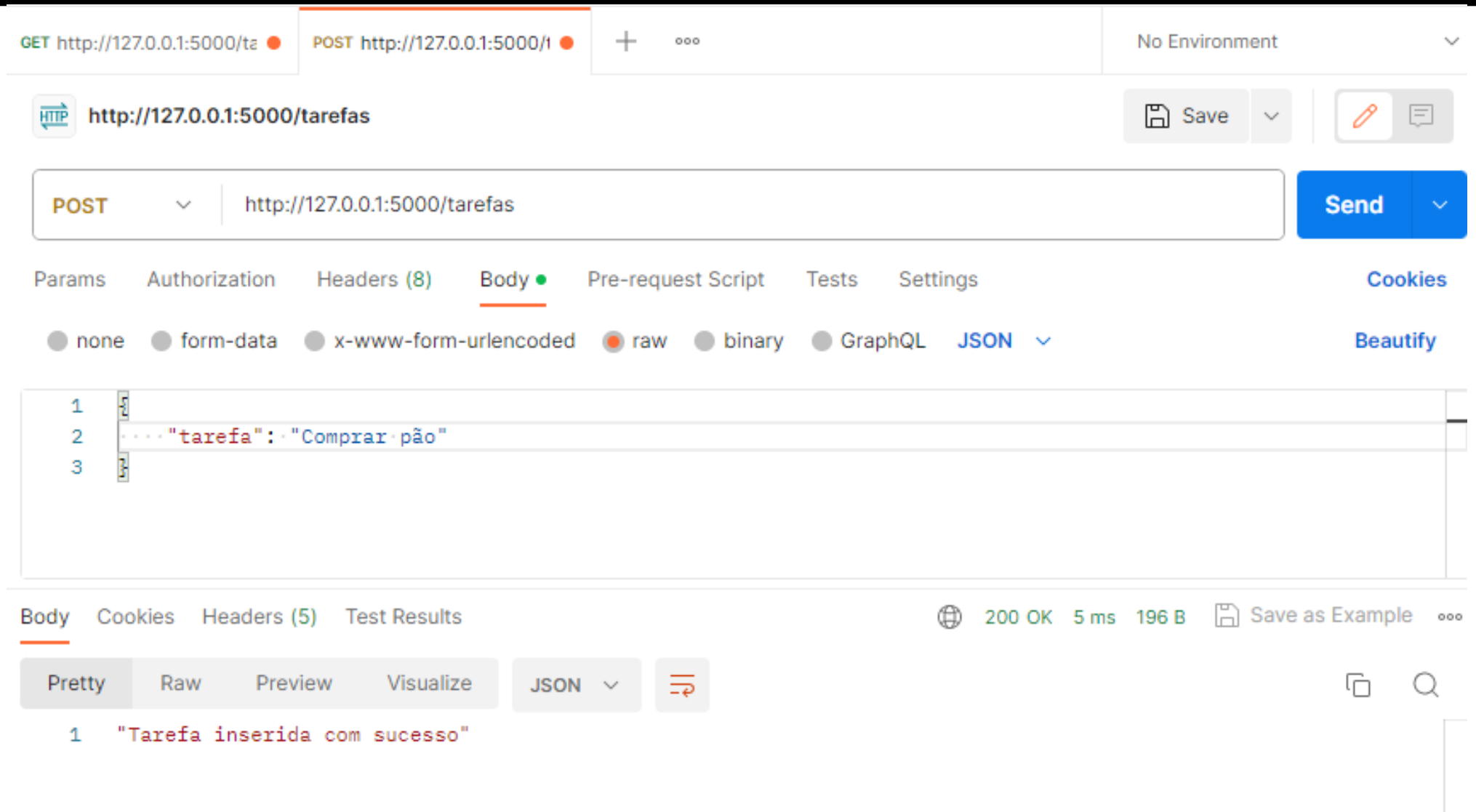


Rota /tarefas

* Passo 5: Implementar a rota POST

```
@app.route('/tarefas', methods=['GET', 'POST'])
def tarefas():
    if request.method == "GET":
        return make_response(jsonify(lista_tarefas))
    elif request.method == "POST":
        dados = request.get_json()
        nova_tarefa = dados['tarefa']
        if nova_tarefa not in lista_tarefas:
            lista_tarefas.append(nova_tarefa)
            return make_response(jsonify("Tarefa inserida com sucesso"), 200)
        else:
            return make_response(jsonify("Tarefa já está na lista!"), 400)
    else:
        abort(404)
```

Rota /tarefas



Rota /tarefa/<int:id>

```python
@app.route('/tarefa/<int:id>', methods=['GET', 'PUT', 'DELETE'])
def tarefa(id):
    if id >= len(lista_tarefas):
        return make_response(jsonify("Tarefa não existente"), 400)
    else:
        if request.method == "GET":
            return make_response(jsonify(lista_tarefas[id]))
        elif request.method == "PUT":
            dados = request.get_json()
            tarefa_editada = dados['tarefa']
            lista_tarefas[id] = tarefa_editada
            return make_response(jsonify("Tarefa Editada com sucesso"), 200)
        elif request.method == "DELETE":
            lista_tarefas.pop(id)
            return make_response(jsonify("Tarefa Removida com sucesso"), 200)
        else:
            abort(400)
```