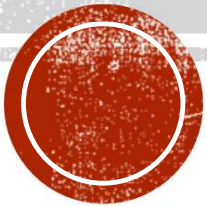
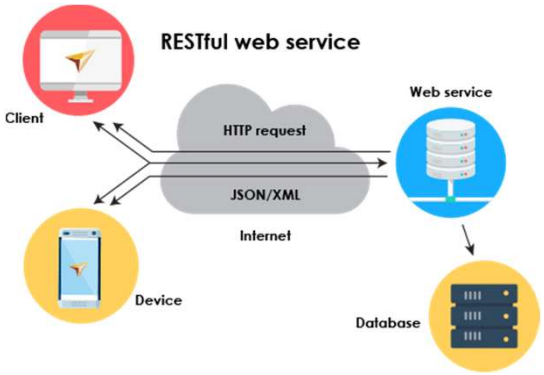
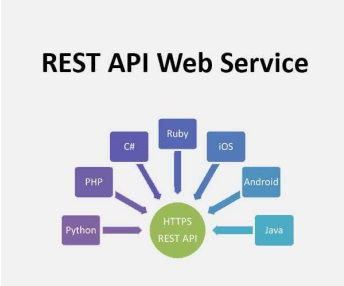
AULA 18 DE 20

21/06/2022





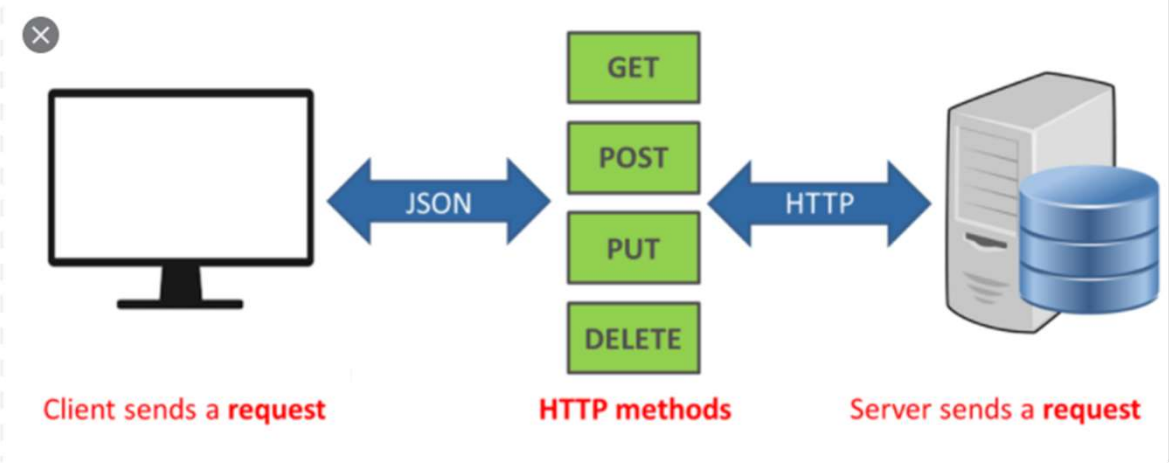
REST - REPRESENTATIONAL STATE TRANSFER



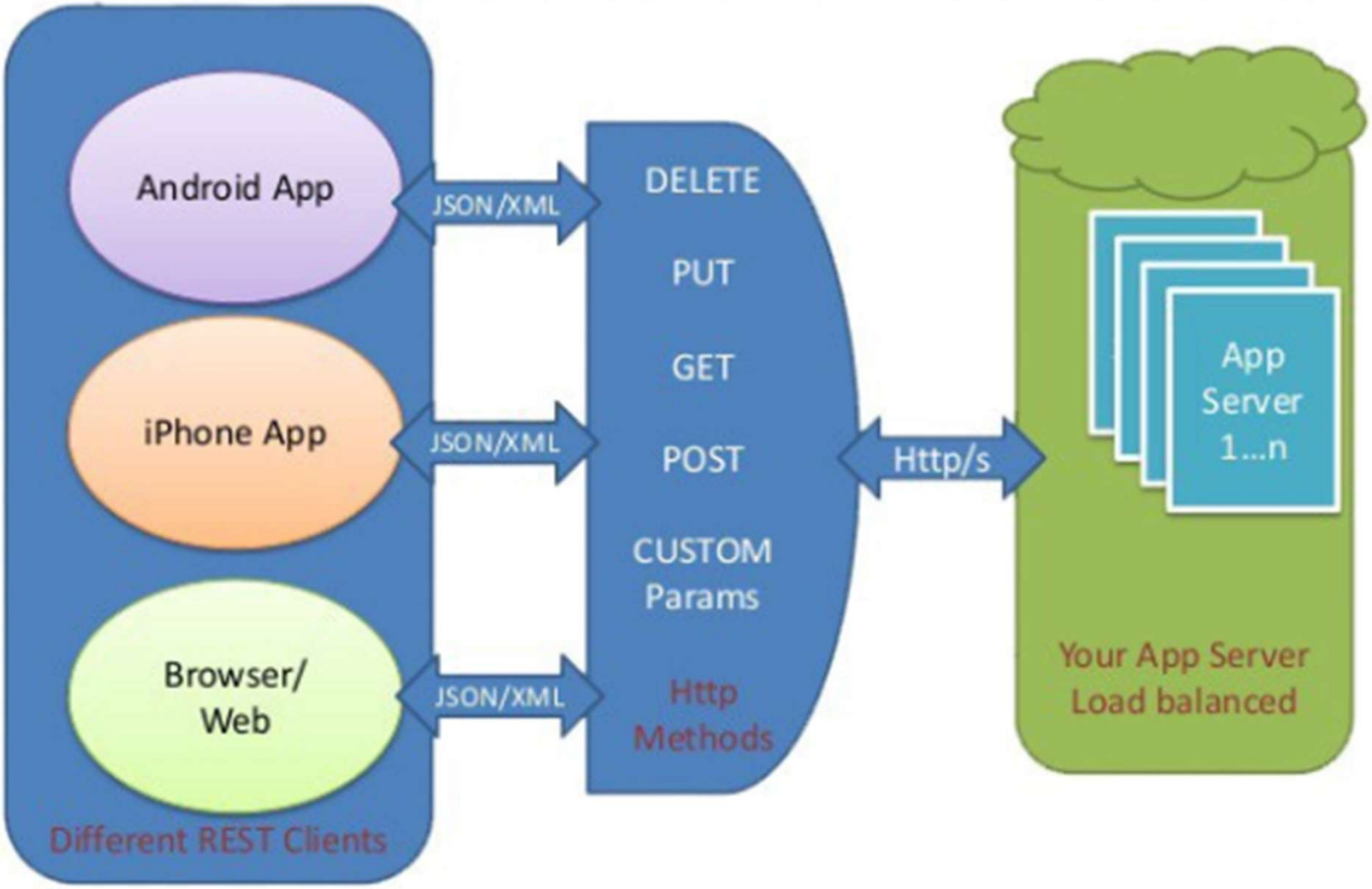
- Tem como objetivo primário a definição de características fundamentais para a construção de aplicações Web seguindo **boas práticas**.
- A Web como a conhecemos hoje, funciona seguindo práticas **REST**.
- Não impõe restrições ao formato da mensagem, apenas no comportamento dos componentes envolvidos.
- A maior vantagem do protocolo REST é sua **flexibilidade**.
- O desenvolvedor pode optar pelo formato mais adequado para as mensagens do sistema de acordo com sua necessidade específica.
- Os formatos mais comuns são **JSON**, **XML** e **texto puro**, mas em teoria qualquer formato pode ser usado.
- **Em algum momento alguém parou e pensou... Porque não usar REST como Web Service???**
- Quase sempre Web Services que usam REST são mais "leves" e, portanto, mais rápidos.
- O problema com o REST pode surgir justamente por causa de suas vantagens.
- **Como a definição do corpo de dados fica totalmente a cargo do desenvolvedor, os problemas de interoperabilidade são mais comuns**.

REST

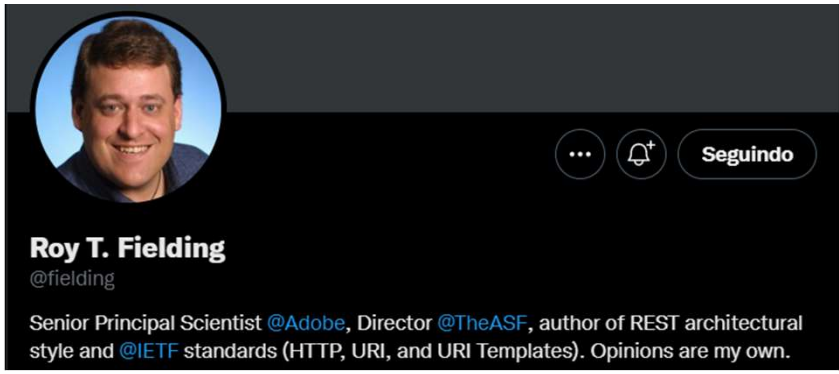
- Cada URL deve representar um recurso;
- Geralmente, **via método GET**, cada recurso deve ser diferençável;
- Uso dos principais métodos de requisições HTTP
  - GET, POST, PUT, DELETE
  - https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods
- **NÃO TEM WSDL** e nem **UDDI**!
- Retorno livre:
  - XML;
  - JSON;
  - Etc.



REST API Architecture



REST - ORIGEM



- **Representational State Transfer**, abreviado como REST, **não é uma tecnologia**, **não é uma biblioteca**, e **nem tampouco uma arquitetura**.
- É um **modelo** a ser utilizado para se projetar arquiteturas de software distribuído, baseadas em comunicação via rede.
- Foi criado por **Roy Fielding**, um dos principais criadores do protocolo HTTP e cofundador do projeto Apache HTTP Server. O termo REST, foi apresentado pela primeira vez em **2000** na sua tese de doutorado, foi uma forma de dizer:
  - "**Vocês estão subestimando o HTTP. Aprendam como utilizá-lo da forma correta!**".
- Muitos desenvolvedores perceberam que também poderiam utilizar o modelo REST para a implementação de Web Services, com o objetivo de se integrar aplicações pela Web, e passaram a utilizá-lo como uma alternativa ao SOAP.
- REST na verdade pode ser considerado como um conjunto de princípios, que quando aplicados de maneira correta em uma aplicação, a beneficia com a arquitetura e padrões da própria Web.

REST - TRANSFERÊNCIA DE ESTADO REPRESENTACIONAL (REPRESENTATIONAL STATE TRANSFER)

- O padrão REST determina como deve ser realizada a **Transferência de Estado Representacional** (Representational State Transfer — REST), ou seja, a representação que corresponde ao conjunto de valores que representa uma determinada entidade em um dado momento.
- Essa transmissão de estados se dá a partir da especificação de parâmetros, em alguns casos chamados de restrições, que podem ser aplicados a web services.
- Quando isso ocorre, o que se obtém são serviços escaláveis, de fácil modificação e manutenção, e que apresentam boa performance, tornando esses serviços adequados a serem utilizados através da internet.
- Outra característica do padrão REST é que a aplicação fica limitada a uma arquitetura **cliente/servidor** na qual um protocolo de comunicação que não mantenha o estado das transações entre uma solicitação e outra deve ser utilizado.

- Ou seja... WebService...

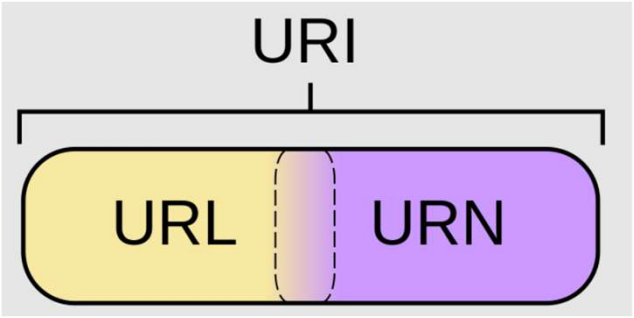
SERVIÇOS RESTFUL

- Nos serviços **RESTful**, tanto os dados quanto as funcionalidades são considerados recursos e ficam acessíveis aos clientes através da utilização de **URIs** (Uniform Resource Identifiers), que normalmente são endereços na web que identificam tanto o servidor no qual a aplicação está hospedada quanto a própria aplicação e qual dos recursos oferecidos pela mesma está sendo solicitado.

- Ou seja... Aplicação web utilizando rest...

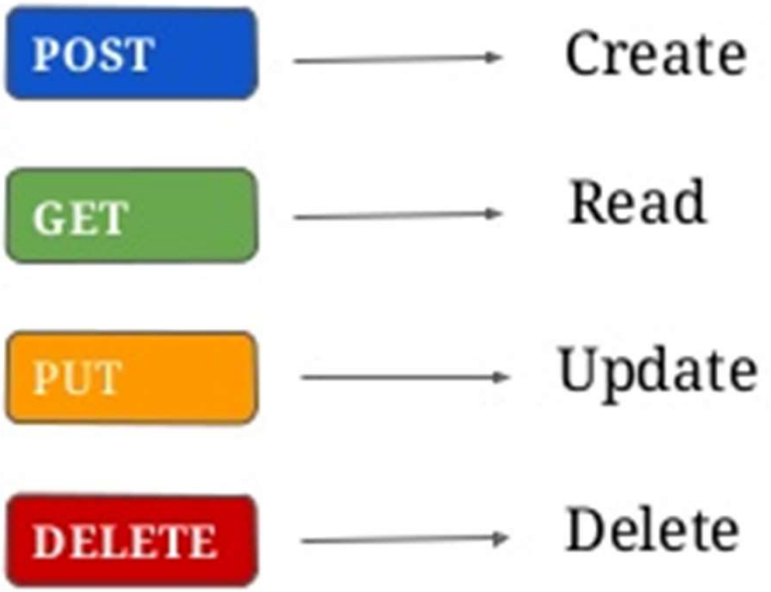
- Existe uma certa confusão quanto aos termos REST e RESTful.
- Entretanto, ambos representam os mesmo princípios.
- A diferença é apenas gramatical.
- Em outras palavras, sistemas que utilizam os princípios REST são chamados de RESTful.
  - REST: conjunto de princípios de arquitetura.
  - RESTful: capacidade de determinado sistema aplicar os princípios de REST.

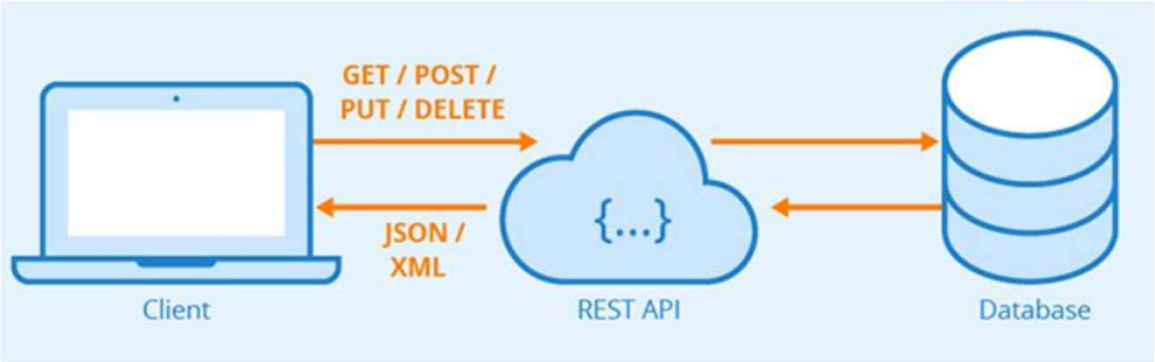
- **URI:** Uniform Resource Identifier
  - **URL**: Uniform Resource Locator
  - **URN**: Uniform Resource Name



- Basicamente uma URI **identifica,** uma URL **localiza** e uma URN **nomeia** um recurso.

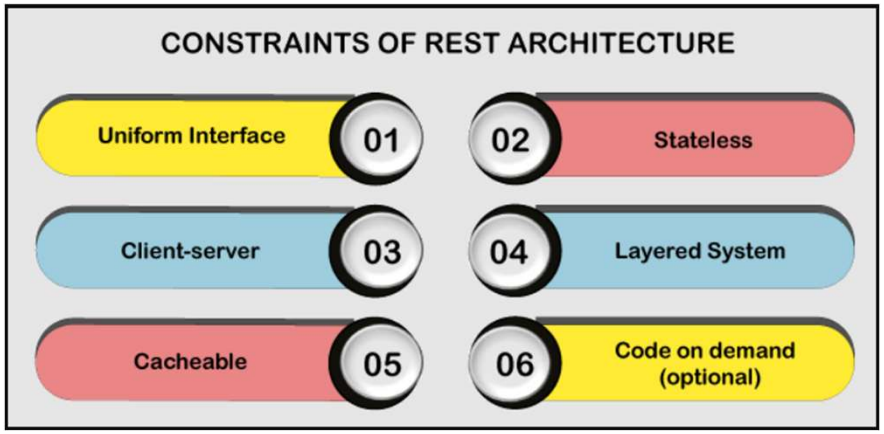
- Os recursos disponíveis em um serviço RESTful podem ser acessados ou manipulados a partir de um conjunto de operações predefinido pelo padrão http.
- As operações possibilitam criar (**POST**), ler (**GET**), alterar (**PUT**) e apagar (**DELETE**) recursos, e estão disponíveis a partir de mensagens utilizando o protocolo **HTTP**.
- Ao receber uma solicitação, ou mensagem **HTTP**, todas as informações necessárias para a realização da transação estão inclusas, não sendo necessário salvar informações para requisições de serviço futuras, o que resulta em aplicações simples e leves do ponto de vista do desenvolvedor.
- Isso quer dizer que além de as solicitações aos serviços apresentarem boa responsividade, de acordo com a infraestrutura disponível, os usuários não fazem muita distinção entre um sistema executando localmente ou em um servidor de aplicação.





- Além dessas qualidades, a adoção do RESTful também tem impacto sobre outras métricas de qualidade de software, por exemplo: a facilidade de uso da aplicação e o tempo necessário pelo usuário para aprender a utilizá-la.

- Outra característica importante dos recursos no padrão RESTful é que esses são dissociados de sua representação, ou seja, a maneira como os dados são manipulados na implementação do serviço não está vinculada ao formato da resposta a ser fornecida a uma solicitação, o que permite que os clientes peçam os dados em uma grande variedade de formatos, como HTML, XML, texto puro, PDF, JPG, entre outros.

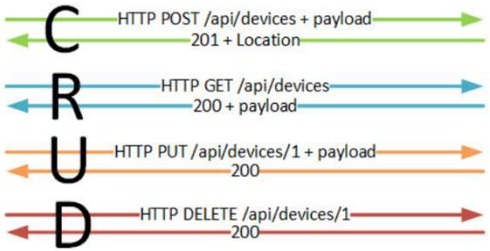


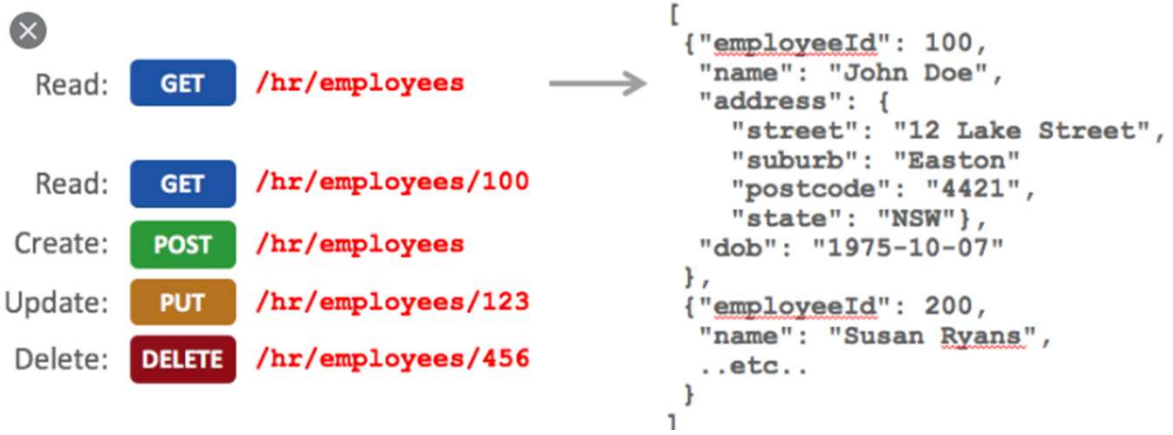
Falhando em um dos cinco itens obrigatórios, a arquitetura não pode ser classificada formalmente como RESTful.

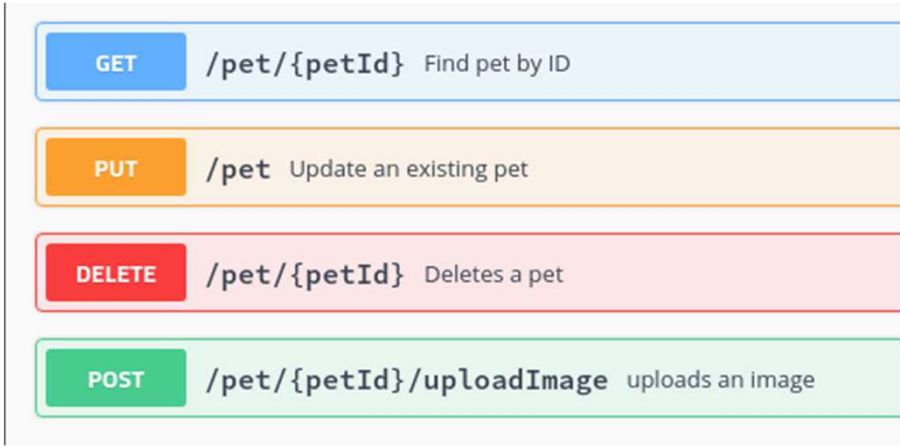
- Arquitetura REST:
  - **Interface uniforme**: O método de comunicação entre um cliente e um servidor deve ser uniforme.
  - **Stateless**: Cada solicitação de um cliente deve conter todas as informações exigidas pelo servidor para realizar a solicitação. Em outras palavras, o servidor não pode armazenar informações fornecidas pelo cliente em uma solicitação e usá-las em outra solicitação.
  - **Cliente-Servidor**: Deve haver uma separação entre o servidor que oferece um serviço e o cliente que o consome.
  - **Sistema em camadas**: A comunicação entre um cliente e um servidor deve ser padronizada de tal forma que permita que os intermediários respondam às solicitações em vez do servidor final, sem que o cliente tenha que fazer nada diferente.
  - **Cacheable**: O servidor deve indicar ao cliente se as solicitações podem ser armazenadas em cache ou não.
  - **Código sob demanda**: Os servidores podem fornecer código executável ou scripts para os clientes executarem em seu contexto. Esta restrição é a única que é opcional.

- Exemplo:
  - http://example.com/**apagar**/produto/1234
  - significa que você está pedindo para apagar o produto de ID 1234.
  - Há quem diga que isto está errado e já que a ênfase é em cima dos recursos e não das operações.

  - O correto seria fazer através do uso do **verbo DELETE do HTTP**, deveria usar assim:
  - **http://example.com/produto/1234**





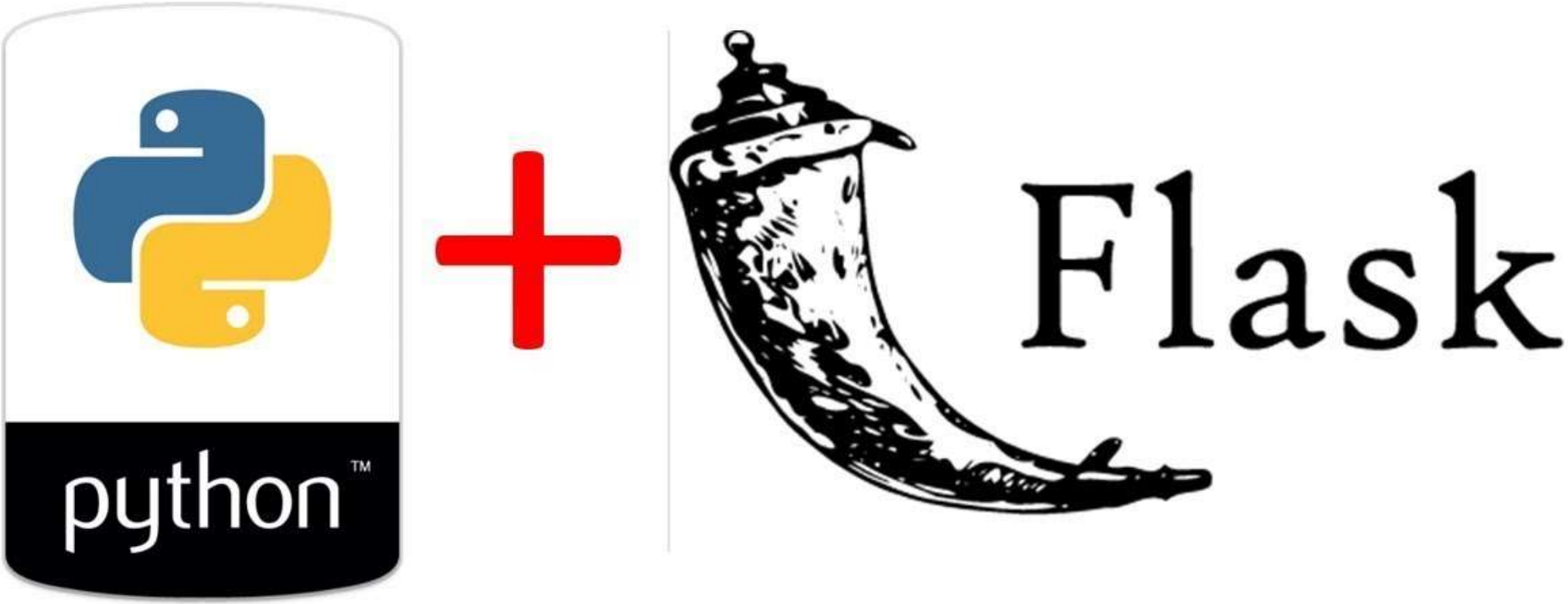


- https://developer.mozilla.org/pt-BR/docs/Glossary/REST
- https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api
- https://www.totvs.com/blog/developers/rest/

- https://becode.com.br/o-que-e-api-rest-e-restful/
- https://pt.stackoverflow.com/questions/45783/o-que-%C3%A9-rest-e-restful

- https://flask-restful.readthedocs.io/en/latest/
- https://realpython.com/api-integration-in-python/
- https://blog.miguelgrinberg.com/post/designing-a-restful-api-using-flask-restful
- https://flask-restful.readthedocs.io/en/latest/quickstart.html
- https://dev.to/aligoren/building-basic-restful-api-with-flask-restful-57oh
- https://medium.com/@apcelent/how-to-create-rest-api-using-flask-439673610cd0
- https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

REST API using Flask



EXEMPLO DE WEB SERVICE RESTFUL

- O **Flask** é um microframework WEB baseado em Werkzeug, Jinja2 etc etc
- É licenciado em BSD!
- Microframework refere-se a uma estrutura de aplicativo da Web leve, em contraste com estruturas completas, como seria o caso do Django.
- Isso significa que o Flask fornece apenas o necessário para criar um servidor de back-end, mas oferece a flexibilidade de instalar quaisquer extensões para suportar recursos como interface de banco de dados, autenticação, criptografia, proteção CSRF e assim por diante.
  - O cross-site request forgery (CSRF ou XSRF), em português falsificação de solicitação entre sites, também conhecido como ataque de um clique (one-click attack) ou montagem de sessão (session riding), é um tipo de exploit malicioso de um website, no qual comandos não autorizados são transmitidos a partir de um usuário em quem a aplicação web confia.

- OBS: O possui uma extensão chamada Flask-RESTful, que permite o rápido desenvolvimento da API REST com configuração mínima. Aqui não faremos uso dela, assim conseguiremos trabalhar bem a essência do REST.

AMBIENTE VIRTUAL DE DESENVOLVIMENTO E INSTALAÇÃO

- Para começar, precisaremos instalar as bibliotecas e as dependências necessárias, mas antes vamos criar nosso **Ambiente Virtual de Desenvolvimento**, algo que já utilizamos muito na disciplina de Dev Web.
  - Abra o VSCode e através da opção **File – Open Folder**, abra o diretório onde deseja salvar seu projeto, por exemplo: c:\abcBolinhas
  - Usando o TERMINAL do VSCode, vamos criar um Ambiente Virtual de Desenvolvimento em um diretório chamado WebServiceREST, onde será armazenado nosso Web Service rest.

    python -m venv WebServiceREST
  - O comando acima criou um ambiente virtual de desenvolvimento **totalmente isolado** e **zerado**! Sem importar nenhuma das bibliotecas/pacotes anteriormente instalados via pip ou easy_install
- OK, agora já temos nosso virtual env zerado e pronto para colocar os pacotes que vamos precisar, como por exemplo o Flask.
- Para poder utilizar o virtual env criado, precisamos ativa-lo. Para ativar e usar este virtual env é bem simples, bastando executar na raiz do projeto via terminal do VSCode:

  Scripts\activate
- Quando estiver ativo, vai aparecer uma flag com o nome do projeto no inicio da linha de comando
- (WebServiceREST) C:\Users\coelho\Desktop\SD\WebServiceREST>
- Uma vez ativo o virtual env, você já esta pronto para instalar os pacotes necessários através do pip. Lembrando que nesse caso eles não serão mais instalados globalmente, e sim em Lib/site-packages/
- Toda vez que você quiser rodar sua aplicação, o virtual env deve estar habilitado!!!
- Usaremos o **Flask-RESTful**, uma extensão do **Flask** que permite o rápido desenvolvimento da **API REST** com configuração mínima.

  pip install flask

  pip install flask-restful

IMPLEMENTAÇÃO

- Criaremos uma API RESTful usada para armazenar detalhes dos usuários, que terão funções CRUD (Criar, Ler, Atualizar, Excluir), permitindo criar um novo usuário, obter detalhes do usuário existente, atualizar detalhes do usuário existente e excluir usuário existente.
- Tudo será realizado através dos verbos HTTP (POST, GET, PUT, DELETE)
  - https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods
- Vamos começar importando o módulo necessário e configurando o aplicativo Flask-RESTful :

Crie e salve o arquivo como **REST_Servidor.py**:

```
from flask import Flask
from flask_restful import Api, Resource, reqparse

app = Flask(__name__)
api = Api(app)
```

- *Estamos importando "**Flask**", "**Api**" e "**Resource**" com as iniciais maiúsculas, isso significa que uma classe está sendo importada.*
- ***reqparse** é a interface de análise de solicitação **Flask-RESTful** que será usada posteriormente.*
- *Em seguida, criamos um aplicativo usando a classe "**Flask**", "**__name__**" é uma variável especial do Python que dá ao arquivo Python um nome exclusivo.*
- *Nesse caso, estamos dizendo ao aplicativo para executar neste local específico.*

```
usuarios = [
    {
        "id": 1,
        "nome": "Abc",
        "idade": 42
    },
    {
        "id": 2,
        "nome": "Bolinhas",
        "idade": 32
    },
    {
        "id": 3,
        "nome": "Uniplac",
        "idade": 22
    }
]
```

- Aqui não vamos nos preocupar com banco de dados etc etc
- Usaremos uma lista de usuários
- Esta lista terá como finalidade simular nosso banco de dados!

- Agora começaremos a criar nossos endpoints da API definindo uma classe de recursos do Usuário.
- Quatro funções que correspondem a quatro métodos de solicitação HTTP serão definidas e implementadas:

```
class User(Resource):
    def get(self, id):

    def post(self, id):

    def put(self, id):

    def delete(self, id):
```

*Uma das boas qualidades de uma API REST é que ele segue o **método HTTP padrão** para indicar a ação a ser executada.*

HTTP request methods
GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

- Usaremos o verbo **get** do http para recuperar detalhes dos usuários.
- A chave da busca é o id.

```
    def get(self, id):
        if(id == 0):
            return jsonify(usuarios)
        else:
            for user in usuarios:
                if(id == user["id"]):
                    return user, 200
            msg = {"msg": "Usuário não encontrado"}
            return msg, 404
```

- *Percorreremos nossa lista de usuários para procurar o usuário*
  - *se o id especificado corresponder a um usuário na lista de usuários, retornaremos o usuário, juntamente com o código de status do HTTP **200 OK***
  - *caso contrário, retornaremos uma mensagem de usuário não encontrado com **404 Não encontrado***
- *Outra característica de uma API REST bem projetada é que ela usa **código de status de resposta HTTP** padrão para indicar se uma solicitação está sendo processada com êxito ou não.*

HTTP response status codes

Informational responses (100–199), Successful responses (200–299), Redirects (300–399), Client errors (400–499), and Server errors (500–599).

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

- Usaremos o verbo **post** para criar um novo usuário:

```
    def post(self, id):
        parser = reqparse.RequestParser()
        parser.add_argument("nome")
        parser.add_argument("idade")
        args = parser.parse_args()

        for user in usuarios:
            if(id == user["id"]):
                msg = {"msg": "Usuário com id {} já existe".format(id)}
                return msg, 400

        user = {
            "id": id,
            "idade": args["idade"],
            "nome": args["nome"]
        }
        usuarios.append(user)
        return user, 201
```

- *Criaremos um analisador usando **reqparse** que importamos anteriormente*
- *Adicionaremos os argumentos de idade e nome ao analisador e armazenaremos os argumentos analisados em uma variável, args (os argumentos virão do corpo da solicitação na forma de dados do formulário, JSON ou XML).*
- *Se um usuário com o mesmo id já existir, a API retornará uma mensagem juntamente com **400 Solicitação inválida** ;*
- *caso contrário, criaremos o usuário anexando-o à lista de usuários e retorná-lo junto com **201 Created** .*

- Utilizaremos o método **put** para atualizar detalhes do usuário especificado.

```
    def put(self, id):
        parser = reqparse.RequestParser()
        parser.add_argument("nome")
        parser.add_argument("idade")
        args = parser.parse_args()

        for user in usuarios:
            if(id == user["id"]):
                user["nome"] = args["nome"]
                user["idade"] = args["idade"]
                return user, 200
        msg = {"msg": "Usuário com id {} não localizado".format(id)}
        return msg, 404
```

- *Se o usuário já existir, atualizaremos seus detalhes com os argumentos analisados e retornaremos o usuário junto com **200 OK***

- O método **delete** é usado para excluir o usuário informado:

```
    def delete(self, id):
        global usuarios
        usuarios = [user for user in usuarios if user["id"] != id]
        msg = {"msg": "{} deletado.".format(id)}
        return msg, 200
```

- *Ao especificar usuários como uma variável no escopo global, atualizamos a lista de usuários usando a compreensão da lista para criar uma lista sem o id especificado (simulando exclusão) e, em seguida, retornamos uma mensagem com **200 OK** .*

- Finalmente, implementamos todos os métodos contidos na classe User, adicionando o recurso à nossa API e especificando sua **rota**.
- Em seguida, executamos nossa aplicação Flask:

```
api.add_resource(User, "/user/<int:id>")


app.run(debug=True)
```

- *<int: id> indica que é uma parte variável da rota que aceita qualquer id.*
- *Especificar o Flask para ser executado no modo de depuração permite que ele seja recarregado automaticamente quando o código é atualizado e nos fornece mensagens de aviso úteis se algo der errado.*
- *É útil na configuração de desenvolvimento, mas **nunca** deve ser usado na configuração de produção.*

- Salve o arquivo como **REST_Servidor.py** e execute-o:

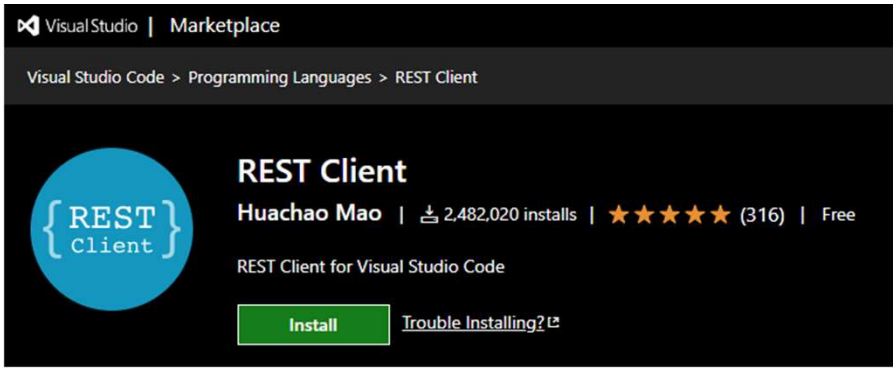
```
python REST_Servidor.py
```

```
(WebServiceREST) C:\Users\coelho\Desktop\SD\WebServiceREST>python REST_Servidor.py
 * Serving Flask app 'REST_Servidor' (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 stat
 * Debugger is active!
 * Debugger PIN: 115-583-162
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

- Agora, temos nossa API REST criada e em execução, portanto, é hora de testá-la!
- Como já vimos anteriormente, podemos realizar isso de varias formas, entre elas:
  - Terminal - **cURL**
  - VSCode – **REST client**
  - Ferramentas gráficas - Postman, Insomnia, SoapUI
  - Aplicação desenvolvida

VSCODE - REST CLIENT

HTTPS://MARKETPLACE.VISUALSTUDIO.COM/ITEMS?ITEMNAME=HUMAO.REST-CLIENT



- **Rest Client** é uma extensão do VSCode onde você consegue criar seus "scripts" com a codificação necessária para executar as suas **requisições HTTP**, **GraphQL** e **cURL**.

- Tudo muito simples!
- Após instalar a extensão, basta criar um arquivo com a extensão "**.http**" ou "**.rest**“, incluir suas requisições e pronto!

- Com isso você já consegue realizar todos seus testes sem precisar sair do editor de código.

TesteRest.http

```
### Verbo GET – listar
GET http://localhost:5000/user/0 HTTP/1.1

### Verbo GET – listar
GET http://localhost:5000/user/3 HTTP/1.1

### Verbo GET – listar
curl -X GET http://127.0.0.1:5000/user/1



### Verbo POST – adicionar
POST http://127.0.0.1:5000/user/4 HTTP/1.1
content-type: application/json

{
    "nome": "luciano",
    "idade": 10
}

### Verbo POST – adicionar
curl -X POST http://127.0.0.1:5000/user/5
    -H 'Content-Type: application/json'
    -d '{"nome":"dggdgd2","idade":90}'
```

```
### Verbo PUT – atualiza
PUT http://127.0.0.1:5000/user/2 HTTP/1.1
content-type: application/json

{
    "nome": "coelho",
    "idade": 22
}

### Verbo PUT – atualiza
curl -X PUT http://127.0.0.1:5000/user/52
    -H 'Content-Type: application/json'
    -d '{"nome":"novo nome","idade":25}’



### Verbo DELETE - excluir
DELETE http://127.0.0.1:5000/user/3 HTTP/1.1

### Verbo DELETE - excluir
curl -X DELETE http://127.0.0.1:5000/user/1
```

TESTES VIA TERMINAL - CURL

- O **cURL** é um projeto de software de computador que fornece uma biblioteca e uma ferramenta de linha de comando para transferir dados usando vários protocolos. O projeto cURL produz dois produtos, libcurl e cURL. Lançado pela primeira vez em 1997, o nome cURL vem do inglês "Client URL".

- Verbo **GET** – Lista os dados do usuário

  curl -X GET http://127.0.0.1:5000/user/0

  curl -X GET http://127.0.0.1:5000/user/3

- Verbo **POST** - adiciona outro usuário

  curl -X POST http://127.0.0.1:5000/user/4 -H 'Content-Type: application/json' -d '{\"nome\":\"luciano\",\"idade\":10}'

  curl -X POST http://127.0.0.1:5000/user/5 -H 'Content-Type: application/json' -d '{\"nome\":\"ze\",\"idade\":90}'

- Verbo **PUT** – atualiza ou adiciona um novo usuário

  curl -X PUT http://127.0.0.1:5000/user/2 -H 'Content-Type: application/json' -d '{\"nome\":\"coelho\",\"idade\":22}'

  curl -X PUT http://127.0.0.1:5000/user/5 -H 'Content-Type: application/json' -d '{\"nome\":\"novo nome\",\"idade\":25}'

- Verbo **DELETE**

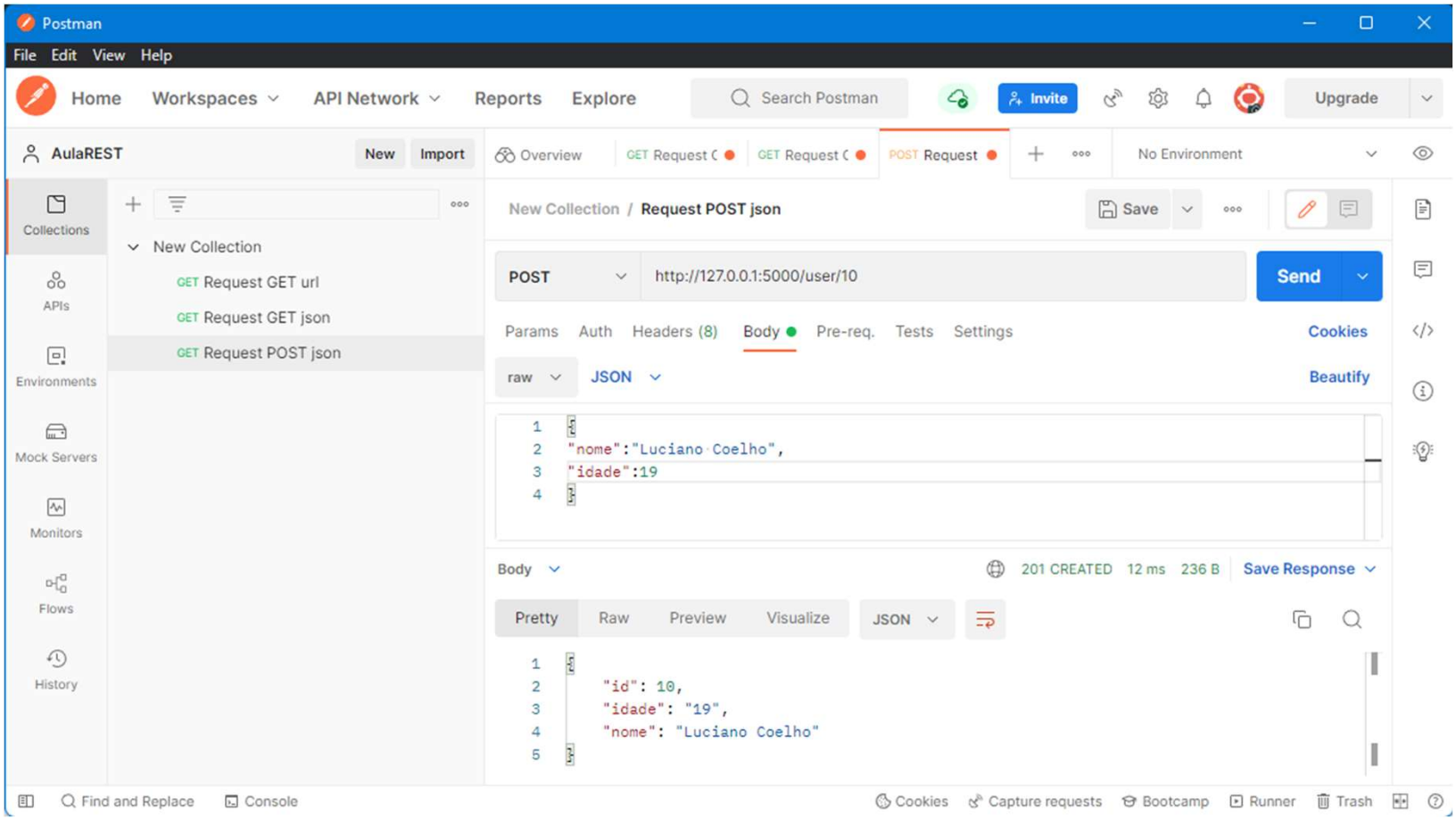
  curl -X DELETE http://127.0.0.1:5000/user/1

- **Postman**
  - https://www.getpostman.com/
  - Essa ferramenta basicamente permite testar os endpoint da API, observar as respostas.
  - Você pode ir ainda mais longe para criar scripts e fazer testes automatizados.

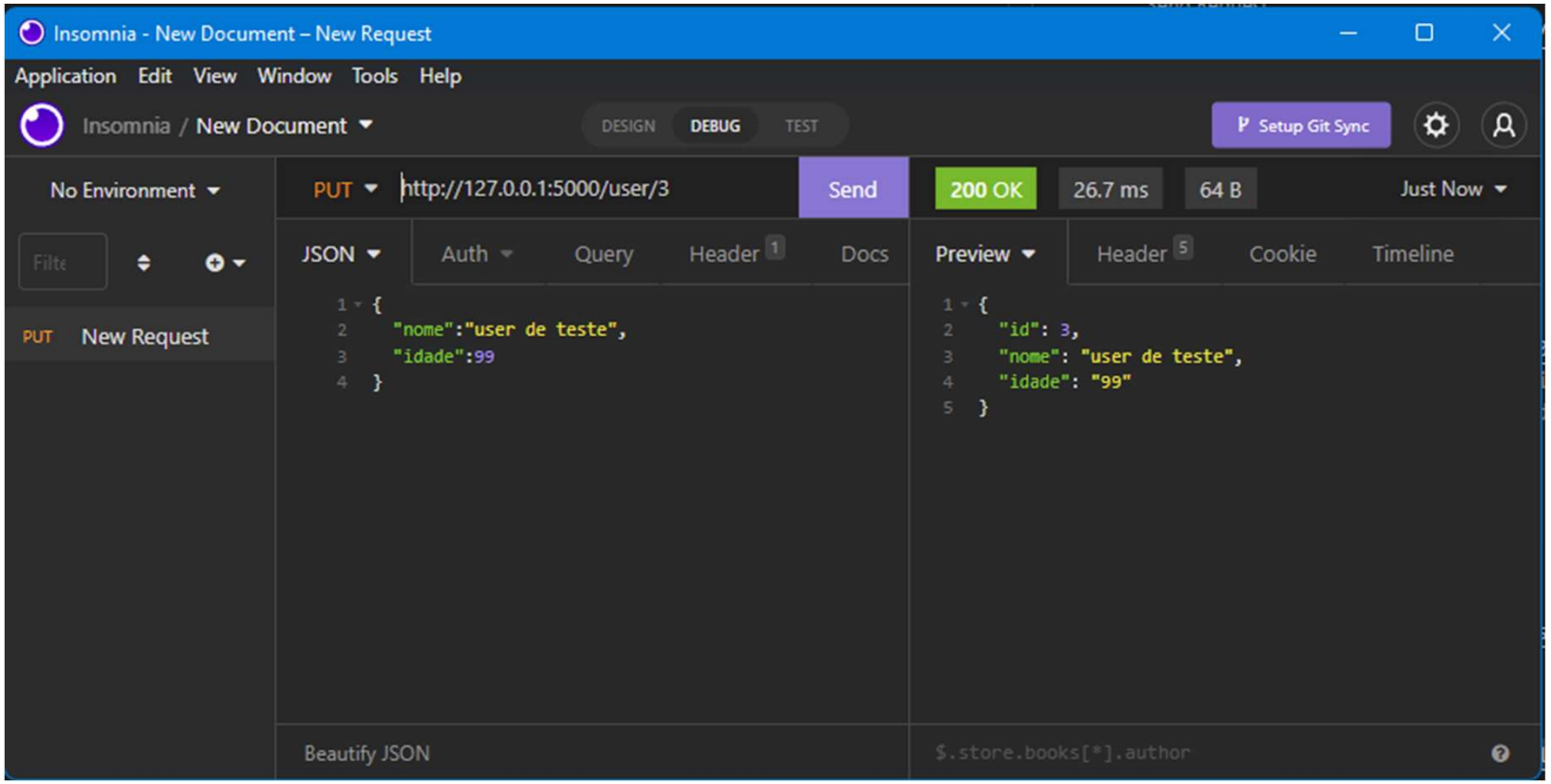
- **Insomnia** – somente rest
  - https://insomnia.rest/
  - Uma alternativa de código aberto ao Postman.
  - Ele vem com todos os recursos básicos necessários para o teste de terminais de API e um IMO de design melhor.

- **SoapUI**
  - https://www.soapui.org/
  - Tornam mais fácil criar, gerenciar e executar testes ponta a ponta em APIs REST, SOAP e GraphQL, JMS, JDBC e outros serviços da web para que você possa entregar software mais rápido do que nunca.
  - Trial

- New Workspaces
- Add a request



Insomnia


Insomnia - New Document – New Request
Application Edit View Window Tools Help
Insomnia / New Document
DESIGN DEBUG TEST
Setup Git Sync
No Environment
PUT http://127.0.0.1:5000/user/3
Send
200 OK
26.7 ms
64 B
Just Now
Filte
PUT New Request
JSON Auth Query Header 1 Docs
Preview Header 5 Cookie Timeline
{
"nome":"user de teste",
"idade":99
}
{
"id": 3,
"nome": "user de teste",
"idade": "99"
}
Beautify JSON
$.store.books[*].author

UTILIZANDO APLICAÇÃO

- Segue um exemplo básico de uma aplicação desenvolvida em Python, consumindo os dados do nosso Web Service REST.

- Para poder realizar chamadas dos métodos HTTP, precisaremos instalar o requests

```
pip install requests
```

```python
import requests

base_url = "http://localhost:5000/user/%s"

while True:

    operacao = input("Escolha uma operação (GET, POST, PUT, DELETE, SAIR): ")

    if operacao.upper() == "GET":
        print(">>> GET <<<")

        id = input("Id: ")
        response = requests.get(base_url % id)

        print(response.json())
        print("HTTP Code: %s" % response.status_code)

    elif operacao.upper() == "POST":
        print(">>> POST <<<")

        id = input("Id: ")
        nome = input("Nome: ")
        idade = input("Idade: ")
        payload = {'nome':nome, 'idade':idade}
        response = requests.post(base_url % id, json=payload)

        print(response.json())
        print("HTTP Code: %s" % response.status_code)

    elif operacao.upper() == "PUT":
        print(">>> PUT <<<")

        id = input("Id: ")
        nome = input("Nome: ")
        idade = input("Idade: ")
        payload = {'nome': nome, 'idade': idade}
        response = requests.put(base_url % id, json=payload)

        print(response.json())
        print("HTTP Code: %s" % response.status_code)

    elif operacao.upper() == "DELETE":
        print(">>> DELETE <<<")

        id = input("Id: ")
        response = requests.delete(base_url % id)

        print(response.json())
        print("HTTP Code: %s" % response.status_code)

    else:
        print(">>> SAIR <<<")
        break
```

AVALIAÇÃO 02 – PARCIAL DE NOTA
LABORATÓRIOS PRÁTICOS (PROCESSUAL E CONTINUA)
4 PARCIAIS DE NOTA

- Utilizando REST, implemente e publique em seu servidor os seguintes serviços:
  - **CRUD de usuários**
  - **Validação de CPF**
    - Cliente envia um CPF
    - Servidor devolve boolean indicando se é valido ou não

- Realizar o consumo dos serviços publicados através de **4 clientes** desenvolvidos em diferentes linguagens.
  - Exemplos de linguagens: GO, Python, Java, JS, PHP, C etc

- **Atenção:**
  - No mínimo 02 clientes devem, obrigatoriamente, possuir interface gráfica, podendo ser Desktop ou WEB.
  - Um dos clientes obrigatoriamente tem que ser desenvolvido em GO.

AVALIAÇÃO 02 – PARCIAL DE NOTA
LABORATÓRIOS PRÁTICOS (PROCESSUAL E CONTINUA)
4 PARCIAIS DE NOTA

- Data de entrega:
  - **28/06/2022**

- Data de entrega como recuperação:
  - **03/07/2022**

  - Deve ser postado na atividade correspondente no classroom os seguintes arquivos:
    - 5 Prints demonstrando o funcionamento dos serviços publicados. Cada funcionalidade deve ser demostrada através de uma técnica/ferramenta diferente, seguindo o seguinte padrão:
      - **C**: **REST Client** via VSCode
      - **R**: **Postman**
      - **U**: **cURL** via Terminal
      - **D**: **Insomnia**
      - **validaCPF**: **SoapUI**
    - Todos os fontes desenvolvidos, organizados por clientes e servidor!
    - Arquivo descrevendo todas as tecnologias e versões utilizadas para o desenvolvimento.
    - Vídeo demonstrando a execução de cada cliente, devendo ser demonstrado cada um dos serviços.

- Os prints e vídeos NÃO podem ser compactados!
- O trabalho será desenvolvido em **DUPLA**, conforme listado na página seguinte!

- Duplas:
  - Lucas Vinicius Peixer Vieira
  - Nicolas Adams Diniz

  - João Gabriel Boeira Vidolin
  - Matheus Felipe Burigo

  - Felipe Parizzi Galli
  - Matheus Barbosa Bandeira

  - Eliza Muniz de Souza
  - Marcos Adriano Ribeiro Lima

  - Eduardo Alves Ortiz
  - Renata Lia Zanatta

  - Eduardo Rodolfi Beppler
  - Luan Adrian da Silva

- Duplas:
  - Tiaraju Roger de Oliveira
  - Vinícius Gabriel de Jesus

  - Angela Izabel Frescki
  - Edson dos Passos Amarante

  - Emannuel Amaral Vieira
  - Gabriel Vieira dos Santos

  - Clébson Custódio Castilho
  - Thiago Cezar Posanske Sartorel

  - Gabriel de Moraes