Módulo 02
Tecnologias alternativas de desenvolvimento web

Experiência Criativa: Criando Soluções Computacionais

ANTÔNIO DAVID VINISKI
antonio.david@pucpr.br
PUCPR

Agenda

- O que é o HTTP?
- Como funciona o HTTP?
- Flask HTTP *request* methods.
- Flask HTTP response data types.
- *Blueprints* e módulos no Flask.

HTTP - Hypertext Transfer Protocol

O que é o HTTP?

- O HTTP é um protocolo de comunicação:
  - uma convenção de regras e padrões que controla e possibilita uma conexão e troca de dados entre dois sistemas computacionais.

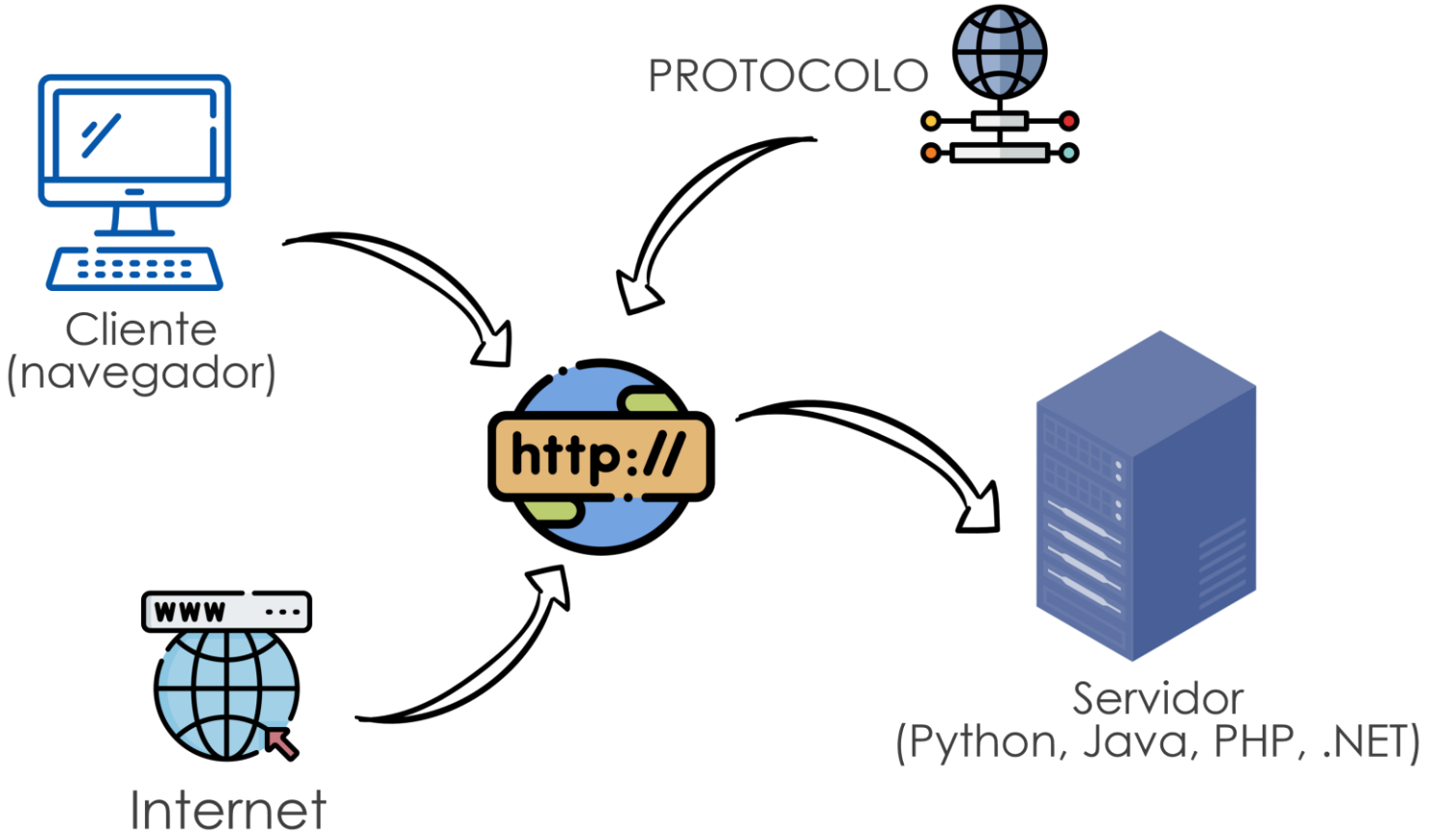
- É baseado no modelo de cliente-servidor:
  - de um lado, um navegador requisita um determinado dado;
  - do outro, um computador (ou servidor) retorna a informação desejada (ou não, caso não ela seja encontrada, ocorra um erro ou não exista).
- Criado na década de 1990, o HTTP surgiu da necessidade de se padronizar a troca de informações pela internet, de uma maneira que fosse leve, rápida e compreendida por todos os computadores conectados à rede.

Como funciona Protocolo HTTP?

- Clientes e servidores se comunicam pela internet trocando mensagens individuais.
- As mensagens enviadas pelo cliente, geralmente navegadores web, são chamadas de requisições (*requests*).
- As réplicas dos servidores são chamadas de respostas (responses), podendo conter algum conteúdo (como arquivos HTML) além de informações sobre o status da requisição.
- Elas são executadas e tratadas por navegadores, programas ou servidores proxy e web. Estes serviços proveem mensagens HTTP por meio de arquivos de configuração (no caso de servidores), APIs (para navegadores) e outras interfaces.

Usuários comuns não lidam diretamente com essas mensagens.

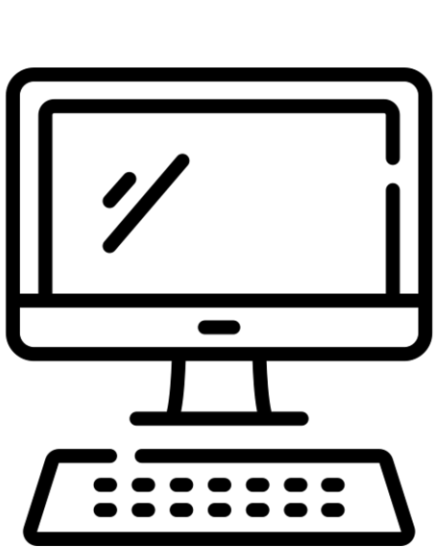
Como funciona Protocolo HTTP?



Requisições e respostas HTTP

- Requisições e respostas HTTP são estruturadas da seguinte forma:
  - Uma linha única inicial (*start-line*) que descreve as requisições a serem implementadas ou seu status de sucesso (ou falha);
  - Um conjunto opcional de cabeçalhos HTTP especificando a requisição ou descrevendo o conteúdo da mensagem;
  - Uma linha em branco apenas para indicar que toda a meta-informação da requisição já foi enviada;
  - O conteúdo da mensagem, chamado de corpo (*body*), conforme solicitado pela requisição. A presença ou não do corpo e seu tamanho são especificados pelos cabeçalhos HTTP (*head*).

Requisições e respostas HTTP



JSON

GET
POST
PUT
DELETE
HEAD
TRACE
OPTIONS
CONNECT

HTTP

Cliente envia uma Requisição (**request**)

Métodos HTTP

Servidor envia uma Resposta (**response**)

Métodos HTTP

- O protocolo HTTP define oito métodos de requisição (GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS e CONNECT) para indicar qual ação deve ser realizada no recurso especificado.
- Os métodos GET e POST, PUT e DELETE são os mais utilizados em aplicações web.
- Um servidor HTTP deve implementar, pelo menos, os métodos GET e HEAD para ser funcional.

Métodos HTTP I

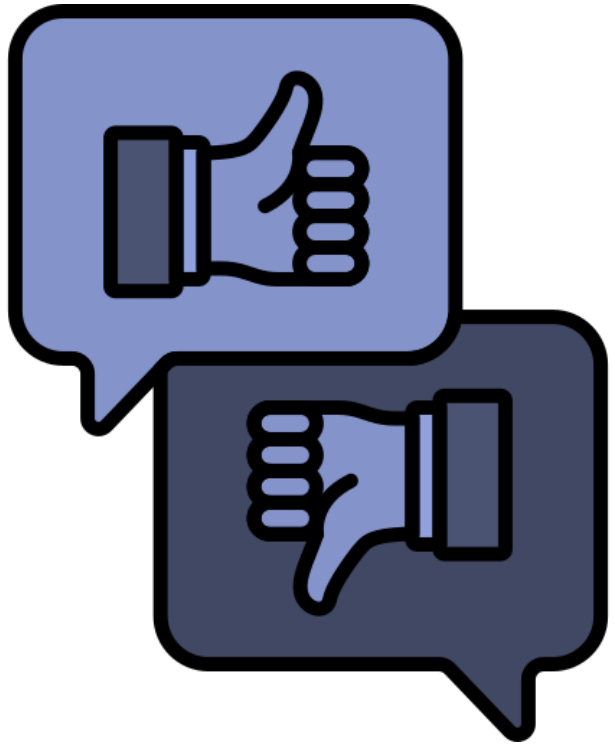
- **GET**: essa requisição é usada para ler ou entregar dados de um servidor web. Quando realizada com sucesso, o servidor retorna o código de status 200.
- **HEAD**: uma requisição HEAD solicita ao servidor somente informações sobre o cabeçalho da página requisitada. O cabeçalho possuí diversas informações que podem ser úteis, como tamanho da página, *cookies*, *tags* e muito mais.
- **POST**: é usado para transferir dados para o servidor (arquivos, formulários, etc.). Quando ocorre com sucesso, é retornado o código de status 201.
- **PUT**: essa requisição é utilizada quando se deseja modificar algum dado no servidor ou caso não exista nenhum dado para se atualizar, será gerado um.

Métodos HTTP II

- **PATCH**: essa requisição aplica modificações parciais em um dado do servidor.
- **DELETE**: essa requisição deleta um dado especificado no servidor.
- **CONNECT**: essa requisição estabelece um túnel TCP/IP com o servidor, geralmente para facilitar a comunicação criptografada com SSL (HTTPS).
- **OPTIONS**: descreve as opções de comunicações de um determinado recurso do servidor.
- **TRACE**: essa requisição envia um teste de *loopback* mostrando o caminho que uma requisição faz para chegar até o recurso especificado no servidor destinado (útil para fazer debug caso a requisição esteja resultando em erro).

Protocolo HTTP (alguns códigos de resposta)

- 200 OK
- 401 Not Authorized
- 403 Forbidden
- 404 Not Found
- 408 Request Timeout
- 429 Too Many Requests
- 500 Internal Server Error
- 503 Service Unavailable



Segurança HTTP

- O HTTP funciona em conjunto com algum outro protocolo de transferência, sendo o TCP/IP (*Transmission Control Protocol*) o mais comum.
- Os recursos enviados por HTTP são identificados e localizados na rede por URLs (*Uniforme Resource Locators*), o tipo mais comum de identificador de recursos uniforme (URI, da sigla em inglês) para a web.
- Uma característica importante do protocolo HTTP que todo usuário deve se atentar é quanto as conexões seguras.
- Na web, ela normalmente é feita pelo HTTPS (*Hyper Text Transfer Protocol Secure*), uma implementação do protocolo HTTP sobre uma camada adicional de encriptação.

Segurança HTTPS

- Essa camada transmite os dados de forma criptografada, além de permitir a verificação de autenticidade do servidor e do cliente por meio de certificados digitais.
- Sem criptografia, os dados na web seriam lidos como texto simples por qualquer pessoa com acesso à rede relevante.
- As páginas da Internet utilizam este protocolo para evitar que terceiros manipulem as informações trocadas entre o site e o usuário.
- Isso nem sempre significa que o site em si é seguro, mas apenas que a conexão está protegida do acesso de terceiros.

Flask HTTP Methods

Flask

web development,
one drop at a time

HTTP Request com Flask

- Por padrão, uma rota no Flask responde a solicitações GET.
- No entanto, você pode alterar essa preferência fornecendo parâmetros para o argumento `method` do decorador `route()`.

```
@app.route('/login’, methods = ['POST', 'GET'])
```

- Para identificar qual método foi usado na requisição, devemos importar o objeto `request` do pacote `Flask`.

```
if request.method == 'POST':
    info = request.form['info']
else:
    info = request.args.get('info', None)
```

Note que cada método permite o acesso aos dados enviados de forma específica.

HTTP Response com Flask

- As repostas representam os retornos das requisições realizadas ao servidor e por ser de várias formas:
  - String:
  - Template:
  - Json:
  - Redirecionamento:

```
return "Hello World!!"
```

```
return render_template("form.html")
```

```
return jsonify({"nome:": "teste", "status":"OK"})
```

```
return redirect(url_for('home'))
```

Exemplo – Requisições e respostas

- Ajustar a aplicação do restaurante que receba informações do usuário e apresente os dados após enviá-los para o servidor.
  - Criar um arquivo **`base.html`** que contenha as informações comuns entre as páginas e tenha um link para cadastro de um pedido.
  - Criar um arquivo **`register_order.html`** que estende o *template* base e contenha um formulário para entrada do **`nome do produto`** e número da **`comanda`** do usuário. O formulário pode ser enviado por meio do método GET ou POST.

```
<form action = "/route" method = "POST">
     <p><input type = "text" name = "name" placeholder="Nome"/></p>
</form>
```

  - Enviar o formulário para o servidor, o qual irá redirecionar os dados para a página **`orders.html`**

Exemplo – base.html

```
<!-- base.html -->
<html>
    <head>
        <title>Meu Restaurante</title>
    </head>
    <body>
        <h2>Meu Restaurante</h2>
        <h3>Acesse o menu:</h3>
        <ul>
            <li><a href="register_order">Registrar Pedidos</a></li>
            <li><a href="pedidos">Listar Pedidos</a></li>
            <li><a href="#">Listar Clientes</a></li>
            <li><a href="#">Listar Funcionários</a></li>
        </ul>
        <div id="content">
            {% block content %} {% endblock %}
        </div>
    </body>
</html>
```

Arquivo base que será estendido pelas demais páginas das aplicação. O bloco **content** será usado para manter os dados específicos de cada página filha

Exemplo – register_order.html

```
{% extends "base.html" %}
{% block content %}
   <form action = "/view" method = "POST">
      <p>Registrar Produto:</p>
      <p><input type = "text" name = "name" placeholder="Nome"/></p>
      <p><input type = "text" name = "ticket" placeholder="ID Comanda"/></p>
      <p><input type = "submit" value = "Enviar" /></p>
   </form>
{% endblock %}
```

Página do formulário de entrada do usuário, enviando os dados para a url **/view** da aplicação por meio do método **POST**

Exemplo – orders.html

```
<!-- orderss.html -->
{% extends "base.html" %}
{% block content %}
    <h1>Pedidos</h1>
    <ul>
        {% if orders|length > 0 %}
            {% for order in orders %}
            <li>{{ orders[order] }}</li>
            {% endfor %}
        {% else %}
            <li>Ainda não existem pedidos registrados!</li>
        {% endif %}
    </ul>
    <p>Voltar para <a href="/">página inicial</a>!</p>
{% endblock %}
```

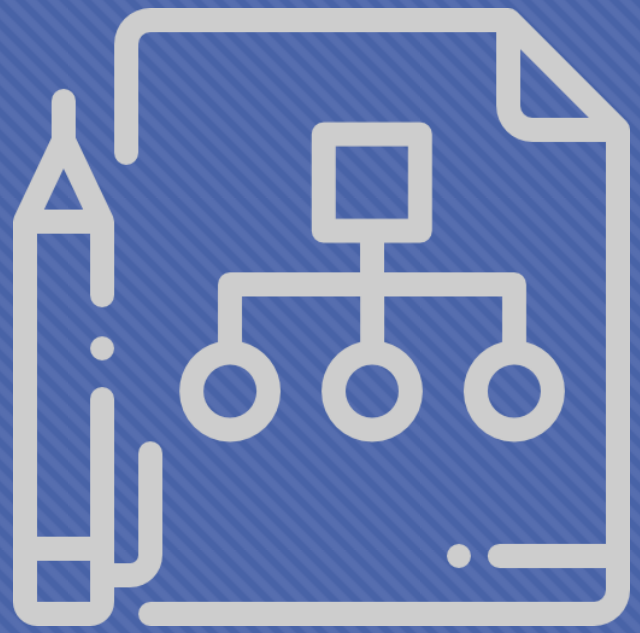
Página de visualização dos dados inseridos pelo usuário.

Exemplo – app.py

- Criação da aplicação `Flask`
- Criação das rotas para as páginas.
- Salvando os dados inseridos pelo usuário em uma variável global

```python
from flask import Flask, request,render_template,redirect,url_for

app = Flask(__name__)

registered = {}

@app.route('/')
def index():
    return render_template("home.html")

@app.route('/orders')
def orders():
    return render_template("orders.html", orders = registered)

@app.route('/order_form')
def order_form():
    return render_template("register_orders.html")

@app.route('/register_order', methods = ['POST', 'GET'])
def register_order():
    if request.method == 'POST':
        name = request.form['name']
        ticket = request.form['ticket']
    else:
        name = request.args.get('name', None)
        ticket = request.args.get('ticket', None)

    global registered
    id = len(registered)+1
    registered[id] = name+", comanda "+ticket

    return redirect(url_for("orders"))

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

Arquitetura de Aplicações Web



Introdução – Estrutura Modular

- As principais linguagens de desenvolvimento de aplicações oferecem abstrações para quebrar a complexidade dos sistemas em módulos.
- Entretanto, são projetadas para a criação de um único executável monolítico, no qual toda a modularização utilizada é executada em uma mesma máquina.
- Assim, os módulos compartilham recursos de processamento, memória, bancos de dados e arquivos.
- Uma alternativa às aplicações monolíticas é a utilização do modelo de microsserviços.

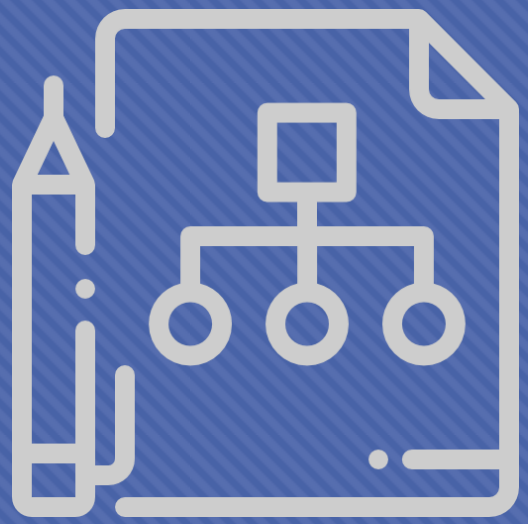
O que são microsserviços?

- Arquitetura em microsserviços trata-se de um modelo arquitetural de desenvolvimento de software em **computação distribuída**.
- O software é desenvolvido em pequenas partes, que são entregues constantemente com o decorrer do projeto.
- Por exemplo, para uma aplicação que possui cadastro de cliente, pagamento, *upload* de fotos, você pode quebrar cada uma dessas *features* em um *microservice*.

Vantagens

- As vantagens de utilizar essa arquitetura são a **modularidade**, **escalabilidade** e **manutenibilidade** de código.
- O baixo acoplamento permite separar as partes da API, possibilitando escalar as aplicações independentemente facilitando sua manutenção sem afetar o negócio como um todo.
- O *microservice*, de maneira geral, possui um número de linhas de código muito reduzido o que facilita seu entendimento e manutenção.

Flask Blueprint para organização da aplicação



Flask Blueprints

- O **`Flask`** usa um conceito de **`blueprints`** para criar componentes da aplicação e dar suporte a padrões comuns dentro de uma aplicação ou entre aplicações.
- Os **`blueprints`** podem simplificar muito o funcionamento de grandes aplicações e fornecer um meio central para que as extensões **`Flask`** registrem operações nas aplicações.
- Um objeto **`blueprint`** funciona de maneira semelhante a um objeto **`Flask`** da aplicação, mas na verdade não é uma aplicação distinta.
- Em vez disso, o **`blueprint`** ajuda a construir ou estender uma aplicação.
  - ele é um projeto de um app que tem praticamente as mesmas características de um app, só que ele não pode ser usado diretamente como um app, para isso ele precisa ser registrado e construído.

Benefícios do Flask Blueprints

- O Flask Blueprint nos permite manter recursos relacionados juntos e ajuda em melhores práticas de desenvolvimento.
- Alguns dos benefícios do Flask Blueprints são os seguintes:
  - Fácil organização das aplicações em grande escala.
  - Aumenta a capacidade de reutilização do código, registrando o mesmo **`Blueprint`** várias vezes.
  - Um conjunto de operações é registrado e pode ser reproduzido posteriormente após registrar um **`blueprint`**.

Criando e registrando uma Blueprint

- admin.py

```
from flask import Blueprint

admin = Blueprint("admin",__name__, static_folder="static", template_folder="templates")

@admin.route("/")
def admin_index():
    return "Admin Blueprint"
```

- app.py

```
from admin.admin import admin
from flask import Flask

app = Flask(__name__)
app.register_blueprint(admin, url_prefix='/admin')
```

Exercício – Aplicação do Restaurante

- Considere a aplicação do restaurante:
  - Login, registro, logout na aplicação (Auth);
  - Admin Pessoas (Cadastros de clientes, funcionários garçons, funcionários serviços gerais);
  - Admin Produtos (Cadastros de produtos e ingredientes, gerenciamento de estoque);
  - Registro de Pedidos (nas comandas dos clientes, pelos garçons);
  - Cobrança das comandas(Entrada);
  - Pagamento mensal dos funcionários, registro de horas (Saída);
  - Gerenciamento de sensores e atuadores para aplicação de IoT;
- Cada estudante irá criar um módulo específico sorteado: os módulos serão disponibilizados no CANVAS para serem registrados na aplicação posteriormente.
- Todos os estudantes precisam ter todos os módulos registrados, para análise do PBL – Portifólio de Aprendizagem.

Referências

- Documentação do *Flask*:
  - https://flask.palletsprojects.com/en/2.2.x/
- Documentação do *Jinja2*:
  - https://jinja.palletsprojects.com/en/3.1.x/
- Documentação *Blueprints*:
  - https://flask.palletsprojects.com/en/2.2.x/blueprints/