Módulo 3 – Python para aplicações web

Bootcamp: Desenvolvedor Phyton

Marcelo Sampaio

2020

Python para aplicações web

Bootcamp: Desenvolvedor Python

Marcelo Sampaio

Sumário

Capítulo 1. Introdução ao HTML

HTML (Hypertext Markup Language) é uma linguagem de marcação utilizada para estruturar uma página na web. A sua organização é feita através de tags, que são rótulos utilizados para informar ao navegador como ele deve ser apresentado.

O HTML é dividido em duas partes: head e body:

- head: contém informações sobre a página e sobre como ela deve ser processada pelo navegador.
- body: contém as informações que serão exibidas ao usuário. Essa parte, por sua vez, pode ser dividida em header, main e footer, como veremos posteriormente.

A forma básica de apresentação de um documento HTML é:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset= “uf-8”>
      <title>Digite seu título aqui</title>
       <link rel="stylesheet" href="style.css">
   <head>
     <body>
```

Tudo o que fica visível ao usuário é escrito neste intervalo:

```
    </body>
</html>
```

- **<!DOCTYPE html>** é a primeira linha de código. Indica ao navegador qual a especificação de códigos utilizar (HTML).
- **<html>** representa a raiz de qualquer documento HTML. É o elemento principal: iniciamos e finalizamos o nosso documento com esta tag.

- **<head>** é onde ficam todas as instruções sobre o conteúdo do documento. Não é mostrado pelo navegador ao usuário final.
    - **<meta charset= “utf-8”>** informa qual conjunto de caracteres será utilizado na página.
    - **<title>** é o título da página. Aparece na barra do navegador, bem como nos bookmarks.
    - **<link>** links de fontes externas que utilizaremos no documento. No nosso exemplo, estamos informando que o nosso documento deve importar um arquivo CSS.
- **<body>** é o espaço no qual escrevemos todo o conteúdo visível da página.

HTML semântico

Segundo a Wikipedia, “semântica é o estudo do significado. Incide sobre a relação entre significantes, tais como palavras, frases, sinais e símbolos e o que eles representam, a sua denotação”.

No caso do HTML não é muito diferente. O HTML semântico é o uso do HTML de forma a dar significado à página, e cada tag no HTML semântico tem um significado.

Por exemplo, podemos utilizar a tag para iniciar qualquer divisão em nossa página. Porém, ela pode ser utilizada em diferentes contextos. Ao utilizarmos um elemento semântico, como a tag <form>, sabemos que tudo que está dentro dela é referente a um formulário.

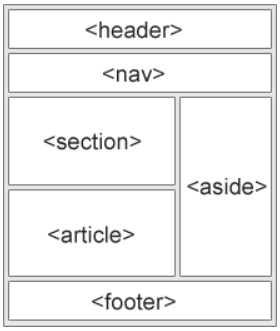
A finalidade da utilização das tags semânticas é facilitar a leitura do documento, tanto por desenvolvedores quanto por navegadores, tornar o site mais acessível para deficientes visuais que utilizam leitor de tela e melhorar a qualidade dos mecanismos de busca, como o Google.

Principais tags semânticas do HTML

- **<header>** representa o cabeçalho do documento. Pode ser aplicada para inserir o conteúdo inicial de uma página.

- **<nav>** é utilizada para apresentar um agrupamento de links de navegação.
- **<section>** define uma seção do documento.
- **<article>** define uma composição independente.
- **<aside>** geralmente representadas como barras laterais, é relacionada a conteúdo do seu entorno
- **<footer>** é o rodapé do documento.

Figura 1 – HTML semântico



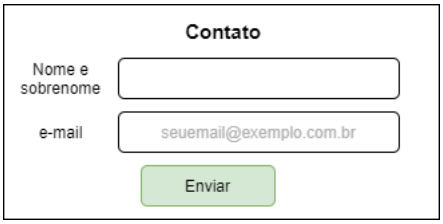
Fonte: https://www.w3schools.com/html/html5_semantic_elements.asp

Construindo formulários e tabelas em HTML

1. Formulário

Para que dados sejam informados pelo usuário, tais como nome, e-mail, telefone, devemos utilizar um formulário. Esses dados normalmente serão processados por um backend (aplicação no servidor web). Veja na figura abaixo um exemplo de um formulário:

Figura 2 – Forma de apresentação de um formulário HTML em um navegador



Como escrevemos este formulário em HTML?

```
<form action="salvar_dados_form" method="post">
    <label> "Nome e Sobrenome" </label>
    <input type= "text">
    <label for= "email"> "e-mail" </label>
    <input type= "email" required placeholder =
"seuemail@exemplo.com.br">
    <input type= "submit" value= "Enviar">
</form>
```

- <form> é a tag que define o formulário.
    - action é o atributo que define o local para onde os dados recolhidos no formulário devem ser enviados.
    - method é o atributo que define o método HTTP para enviar os dados (entre as possibilidades *put* e *post*).
- <label> é a tag que funciona como uma etiqueta para o input, na qual informamos para o usuário qual dado deverá ser inserido (texto, email, telefone, etc).
    - for define a quem o label se refere (a qual input ele se refere).

- <input> é o elemento mais importante de um formulário. Ele pode ser utilizado de várias formas, dependendo *type* ao qual ele foi atribuído.

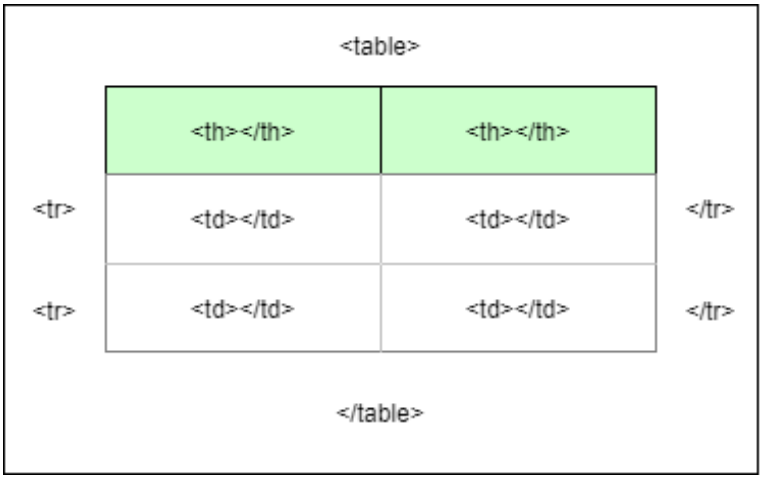
    - type é o atributo para especificar qual tipo de informação será recebida em cada campo do nosso formulário **através do elemento input**. Por exemplo, na Figura 1, a primeira informação que queremos que o usuário digite é seu nome, e para isso, utilizamos *type= "text"*. Outro bom exemplo é o *type= "password,"* que serve para criar um campo para digitação de senha.

    - placeholder- neste atributo, informamos um texto que serve de exemplo para o usuário, que desaparecerá assim que ele iniciar a digitação dos dados.
    - value- é o valor do campo. Para colocar um valor inicial você deve mudar esse atributo.

2. Tabela

Tabela é uma lista de dados composta por linhas e colunas. Na figura 2, podemos ver como a tabela se apresenta em nosso navegador.

Figura 3 – Forma de apresentação de uma tabela HTML em um navegador



Como escrever a tabela da página anterior em HTML?

```
<table border="1">
  <thead>
  <tr>
    <th>titulo 1</th>
    <th>titulo 2</th>
  </tr>
  </thead>
  <tbody>
  <tr>
    <td>Campo 1</td>
    <td>Campo 2</td>
  </tr>
  </tbody>
</table>
```

- <table> é a tag que define a tabela. Pode ser dividida em duas tags:
  - <thead> - representa o cabeçalho da tabela, composta por células título.
  - <tbody> - representa o corpo da tabela.
- <th> tag aplicada para escrever as colunas do <thead>.
- <tr> tag utilizada para inserir as linhas em uma planilha.
- <td> tag utilizada para inserir as células em uma planilha.

Capítulo 2. Introdução ao CSS

CSS (Cascading Style Sheets) é a forma como adicionamos estilo ou mudamos a aparência dos elementos do html, definindo a maneira que eles devem se comportar visualmente para o usuário.

Para incorporar o CSS ao html, devemos criar um arquivo dentro do editor de textos e nomeá-lo com extensão css. Caso o seu site tenha apenas um arquivo CSS, é uma boa prática chamá-lo de style.css.

Precismos linká-lo ao nosso html, conforme demonstrado abaixo:

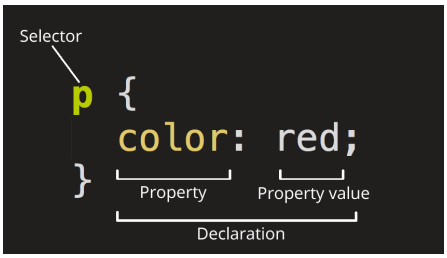
```
<head>
    <meta charset="uf-8">
    <title>Escreva um título</title>
    <link rel="stylesheet" href="style.css">
</head>
```

Agora, podemos formatar o documento html. Vamos usar um exemplo para demonstrar como seria sua formatação no CSS:

```
<body>
    <header>
        <h1>Digite seu texto aqui</h1>
    </header>
</body>
```

A Sintaxe básica dentro do CSS é:

Figura 4 – Conjunto de regras CSS



Fonte: https://developer.mozilla.org/pt-BR/docs/Aprender/Getting_started_with_the_web/CSS_basico

- **Selector:** define a quais elementos (tags HTML) um conjunto de regras CSS se aplica.
    - **Pseudoclasse:** é uma palavra-chave adicionada ao seletor que especifica um estado especial do elemento a ser selecionado.
- **Declaration:** é a especificação do que será estilizado neste elemento;
- **Property:** indica qual propriedade será estilizada. Neste caso, será alterada a cor.
- **Property value:** é o valor que será dado à propriedade.

Tabela 1 – Exemplos de seletores em CSS

Seletor	Sintaxe	Comentário
tag	tag { }	Seleciona uma tag qualquer, exemplo: H1 { } seleciona todos os H1.
classe	.XXX { }	Seleciona todas as tags que tiverem um class=”XXX” (repare que o seletor

		começa com um ponto).
ID	#identificador { }	Digite cerquilha (#) antes do ID que deseja usar como seletor.
Seletor de descendência	pai filho { }	Seleciona todos os filhos de um determinado pai. Exemplo: nav li seleciona os li que estiverem dentro de um nav.

Tabela 2 – Exemplos de pseudoclasses

Pseudoclasse	Sintaxe	Comentário
:hover	selector:hover { }	Altera a aparência do item no momento em que o mouse está sobre ele.
:first-child	selector:first-child { }	Estiliza somente o primeiro item de uma lista.
:first-letter	selector:first-letter { }	Estiliza somente a primeira letra de um texto.
:first-line	Selection:first-line{ }	Estiliza somente a primeira linha de um texto.

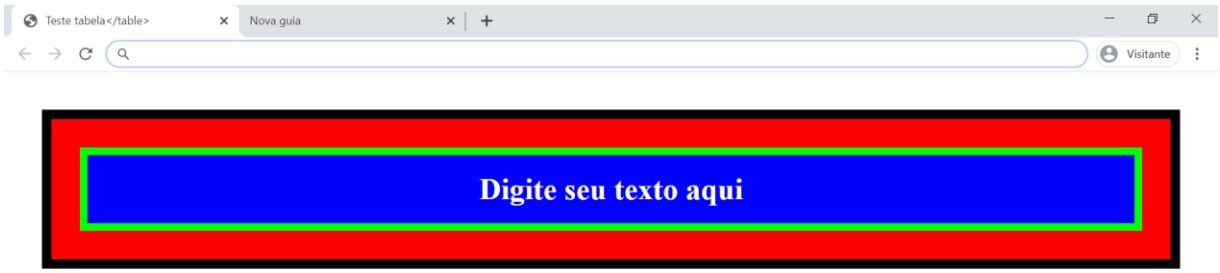
Aplicando esta formatação para executar o CSS do exemplo:

```
        header {
    background: red;
    border: solid black 10px;
    margin:40px;
        }
h1 {
```

```
    margin: 30px 30px;
    padding: 20px;
    border: solid #00cc77 8px;
  background: blue;
    text-align: center;
    font-size: 2em;
    color: #ffffff;
    font-weight: bold;
        }
```

Ao entrar no navegador, veremos que a formatação ficou assim:

Figura 5 – Navegador



Vamos entender tudo que fizemos para conseguir este resultado final, iniciando pelas representações de cores no CSS:

- Cores básicas: é possível escrever o nome da cor onde queremos aplicá-la.
- RGB: é a utilização das cores primárias (**R**ed, **G**reen, **B**lue) para formar outras cores.

___ ___ ___ ___ ___ ___

RED GREEN BLUE

0= ausência de cor

F= máximo de cor

Ex: #000000 é a representação do preto

#00ff00 é a representação do verde

#ffffff é a representação do branco

- Hexadecimal é a representação das cores através de números e letras, como exemplificado abaixo:

PRETO 0 1 2 3 4 5 6 7 8 9 A B C D E F BRANCO

Ex: #000000 é a representação do preto

#aaaaaa é a representação do cinza

#00cc77 é a representação do verde

Tendo o conhecimento de como representar as cores, podemos alterar todo o documento conforme a necessidade.

Ainda formatando os textos, podemos alterar, dentre outras opções:

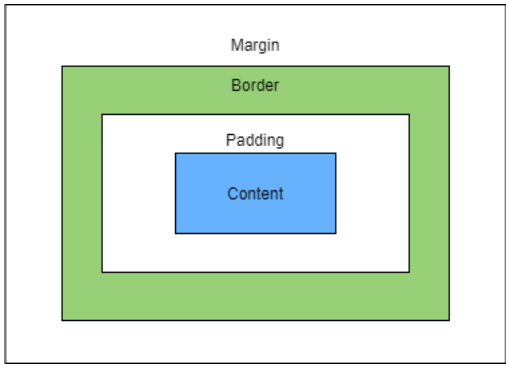
- text- align: permite alterar o alinhamento do texto: justificado, centro, direita, esquerda.
- font-size: permite alterar tamanho da fonte.
- font-weight: permite colocar uma fonte como negrito, normal, mais fino.

Box Model

Um aspecto muito importante na formatação é entender o conceito de box model. Os elementos do HTML podem ser considerados “caixas”. O termo box model é utilizado quando falamos em layot da página, pois este modelo descreve o conteúdo do espaço ocupado por um elemento. Este conceito ficará mais claro nas aulas práticas.

Cada box model tem quatro componentes que compõem a dimensão do elemento, conforme a figura abaixo. Vamos entender cada um deles:

Figura 6 – The Box Model



- **Content** é o conteúdo propriamente dito, seja ele um texto ou uma imagem. Podemos ajustar a altura e largura da área do conteúdo, utilizando as propriedades:
    - width- determina a largura da área do content
    - height- determina a altura da área do contente
- **Padding** é o espaço que circula o conteúdo, ficando entre ele e a borda.
- **Border** é a área que circula o content e o padding.
- **Margin** é o espaço entre este elemento e os seus vizinhos.

Podemos definir valores para cada um destes componentes, como demonstramos no nosso exemplo. A ordem de aplicação será:

superior --- direita --- inferior --- esquerda.

Se um único valor for aplicado, ele será utilizado em todos os lados.

Se aplicarmos dois valores, o primeiro será aplicado nas bordas superior e inferior e o segundo nas bordas direita e esquerda.

Flexbox

O próximo passo para aprender a criar estilos é entender o flexbox, que é um modelo capaz de organizar os elementos espacialmente em uma interface. O layout fica flexível, o que permite que elementos responsivos dentro de um contêiner sejam organizados automaticamente, dependendo do tamanho da tela.

Um elemento ser responsivo significa que ele irá se adequar ao tamanho da sua tela. Lembre-se de que estamos falando de diversos monitores, com tamanhos e resoluções diferentes.

Para entendermos o flexbox, precisamos primeiro entender um pequeno conceito: cada elemento do html tem um padrão de disposição, dependendo do seu tipo.

- **Display Block:** ocupa 100% da largura do elemento pai. Sempre será posicionado abaixo do elemento anterior. Exemplos: section, div, ul, li,
- **Display Inline:** ocupa apenas a largura total do seu conteúdo. Exemplos: span, b, I.

Veja abaixo como é exibido no navegador uma lista de elementos que, por padrão tem o seu **display block.** Repare que, mesmo o elemento não usando todo o espaço, ele ocupa a linha toda:

Ao aplicarmos o flexbox (usamos a propriedade CSS display flex), nosso layout será assim:

Veja o passo a passo para a aplicação do display flex:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="uf-8">
    <title>Flexbox</table></title>
    <link rel="stylesheet" href="reset.css">
    <link rel="stylesheet" href="flex.css">
  </head>

  <body>
    <ul>
      <li>Como faço</li>
      <li>para deixar</li>
      <li>estas caixas</li>
      <li>lado a lado?</li>
    </ul>
  </body>

</html>
```

Ao escrever o html, o elemento <li>, por padrão, é block. No entanto, queremos que nossas caixas sejam dispostas lado a lado. Este elemento é “filho” do elemento <ul>, portanto, é no “pai” que colocaremos a propriedade **display:flex.** Automaticamente, todos os filhos serão dispostos lado a lado. Veja abaixo como é feito no CSS:

```
ul {
    display: flex;
    justify-content: space-around;
    margin: 50px;
    background: chartreuse;
}

li {
    align-items: baseline;
    padding: 20px;
    margin: 10px 50px;
    border: solid black 1px;
    text-align: center;
    font-size:1.2em;
}
```

No exemplo acima, incluímos mais algumas propriedades além do display: flex. Vamos vê-las em detalhes a seguir.

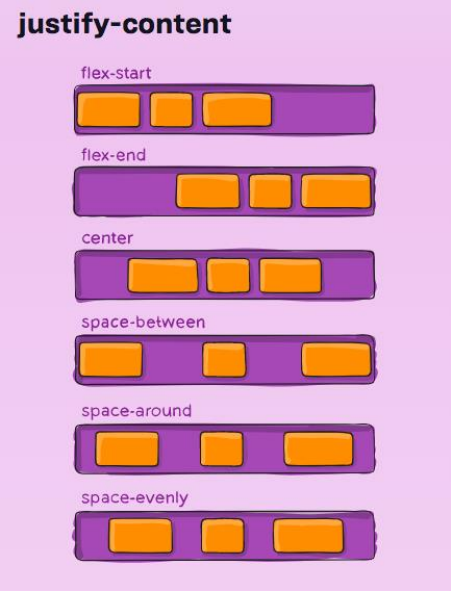
No flexbox, temos parâmetros para os elementos pai, que chamamos de flex conteiner, e para os filhos, que são os flex itens. Veja abaixo as propriedades que utilizamos em cada um deles:

1. **Flex conteiner**
    - **display:** ao informarmos em um elemento que ele tem o display flex, ele passa ser definido como o flex conteiner, e todos os elementos que estão dentro dele passam a ser os flex items.

    - **Justify-content:** alinha os flex itens dentro do conteiner de acordo com a direção.

- **flex-start:** os componentes ficarão sempre no início do eixo.
- **flex-end:** os componentes ficarão sempre no final do eixo.
- **center:** os componentes ficarão centralizados sem espaçamento.
- **space-between:** haverá espaçamento entre os componentes, mas não nas bordas.
- **space-around:** haverá espaçamento entre os componentes e nas bordas.
- **space-evenly:** haverá espaçamento entre os componentes e nas bordas, e esse espaçamento será igualmente distribuído

Figura 7 – Justify content

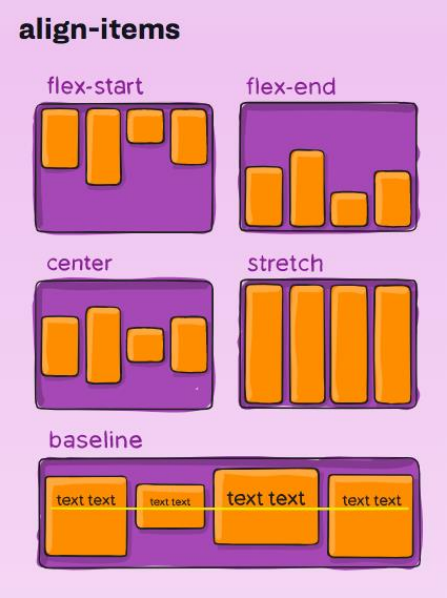


Fonte: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

2. **Flex itens:** São utilizados para alterar um item em especial, e não todos os itens do conteiner. Seus valores possíveis são os seguintes:

- **flex-start, flex-end e center:** iguais ao justifyContent
- **stretch:** tomarão todo o espaço do seu eixo
- **baseline:** ficarão no início da linha do texto.

Figura 8 – Align-items



Fonte: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

Capítulo 3. Rest e JSon

O termo REST foi criado por Roy Fielding na sua tese de doutorado. Ele define uma arquitetura para troca de informações entre cliente e servidor.

Todas aplicações desse modelo contém um backend, responsável por acessar dados, realizar cálculos e outras operações comuns a todas as plataformas, e por enviá-las para os clientes (chamado de front end).

Existe, hoje, a necessidade de que uma mesma aplicação seja visualizada em vários tipos de dispositivos e mídias diferentes. O home banking é um ótimo exemplo disso. Ele pode ser acessado por um programa instalado no seu computador, pela web (através da página do banco), por um celular, tablet, etc.

Provavelmente, todas essas mídias têm necessidades de negócio parecidas, como buscar um saldo, um extrato, fazer transferências, etc. Não faria sentido que cada uma implementasse sua forma de acesso a dados. Além disso, uma estratégia única de acesso ajuda na segurança, pois diminui o que conhecemos como “superfície de contato”, ou seja, as formas de iteração com os dados do banco.

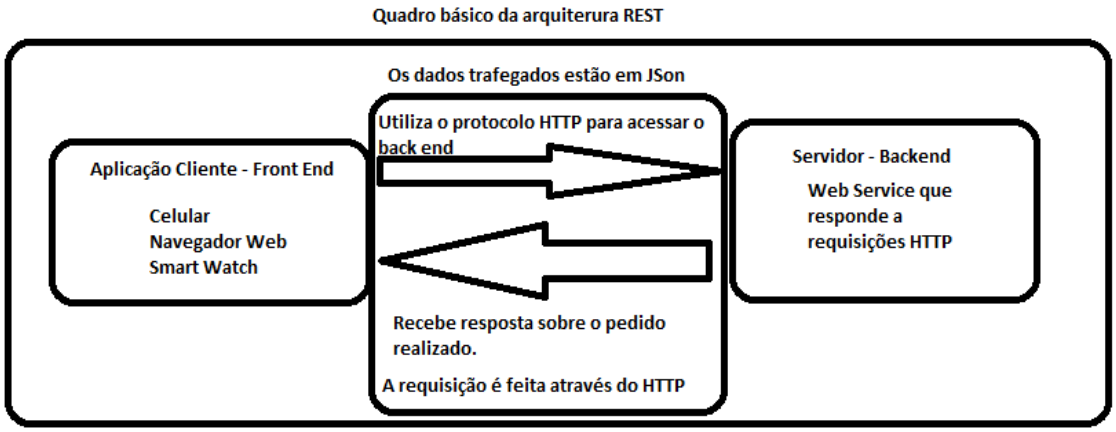
Dessa forma, todas as aplicações modernas estão baseadas no modelo em que um back end pode ser acessado por diversos front ends diferentes em dispositivos e mídias diferentes.

Para que isso seja possível, três coisas são necessárias:

- Um protocolo de comunicação padrão (HTTP)
- Uma arquitetura que permita padronizar a forma de apresentação do dado (REST)
- Uma forma padrão de representar o dado (JSon)

Apresentamos abaixo uma figura ilustrativa dessa arquitetura. Cada um dos itens será explicado melhor abaixo:

Figura 9 - Arquitetura REST



A arquitetura Rest define que os dados são trafegados através do protocolo HTTP. Isso gera várias vantagens, como veremos adiante.

O protocolo HTTP é Stateless, o que significa que cada chamada ao servidor não guarda o estado da chama anterior. Isso tem implicações diretas especialmente na autenticação. Cada vez que o seu celular, por exemplo, precisa pedir o saldo da sua conta bancária, a aplicação precisa enviar dados sobre a autenticação.

Uma requisição ao servidor sempre contém um verbo. Cada verbo tem uma aplicação específica. Os verbos mais importantes são:

GET: Informa ao servidor que estamos requisitando que ele nos devolva uma informação qualquer. Por exemplo, quando você requisita o seu saldo no home banking, uma requisição GET é feita, pedindo um recurso especial (o seu saldo).

POST: Envia informações que devem incluir um recurso. Por exemplo, quando você inclui um cliente no sistema, o front end envia uma requisição com o verbo POST para incluir o cliente no banco de dados.

PUT: Envia informações que devem modificar um recurso. O mesmo cliente do exemplo anterior pode ser modificado por uma requisição com o verbo PUT.

PATCH: Também serve para modificar um recurso, com a diferença de que, com o PUT, a modificação no banco de dados do recurso deverá ser feita por completo (em todos os campos), enquanto o PATCH deverá modificar apenas um campo.

DELETE: Exclui um recurso no servidor. É utilizado, por exemplo a exclusão de um cliente.

Como o protocolo HTTP não nos obriga a realizar as operações conforme o REST, é muito comum ver APIs utilizando verbos errados para realizar as operações. Isso, apesar de não gerar grande diferença no desenvolvimento, pode ser desastroso na produção. Por exemplo, o verbo POST é conhecido por **não** ser idempotente, enquanto os verbos DELETE, PUT e PATCH são idempotentes.

Que problema isso gera? Bom, entre o cliente e o servidor, muitos servidores recebem e repassam a aplicação, e cada um é conhecido pelo nó da rede. Se a comunicação entre dois desses servidores falhar, é realizada a tentativa de repetir a comunicação. Os verbos idempotentes podem ser repetidos quantas vezes for necessário sem gerar problemas. Imagine, se eu enviei a requisição de deletar um cliente da minha base de dados, ela pode ser repetida pois não há risco de "dupla exclusão". Já se eu pedir a inclusão de um cliente e a comunicação entre um dos nós precisar ser repetida por algum motivo, ela poderá gerar duplicidade de dados.

O protocolo HTTP nos garante que uma requisição POST não gerará repetição entre dois nós. No caso de falha de comunicação, será retornado um erro. Já uma requisição DELETE poderá ser tentada novamente.

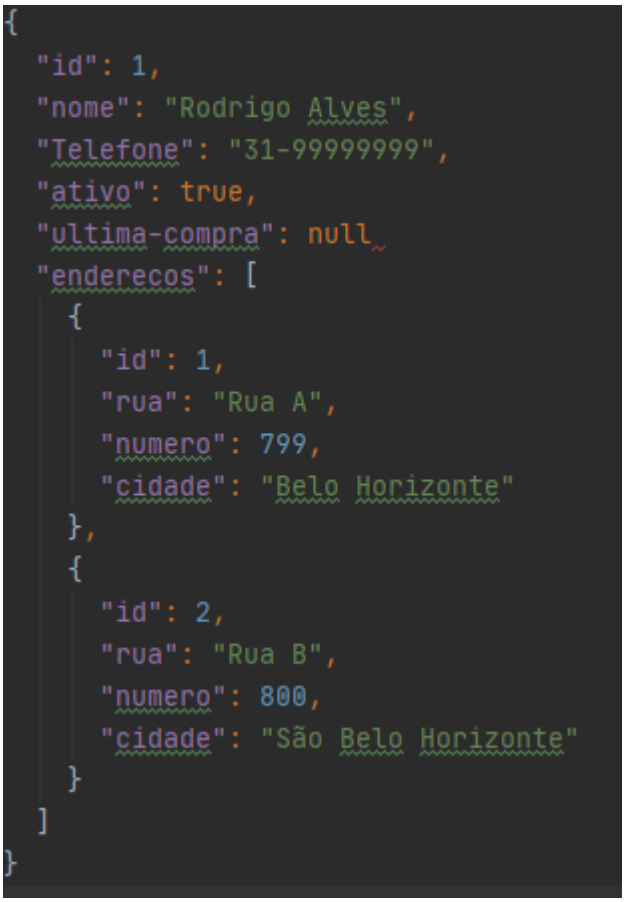
Atenção: tudo isso ocorre internamente. Do ponto de vista da sua aplicação apenas uma requisição está ocorrendo.

Já sabemos como a requisição ocorre, mas como os dados são trafegados?

Aí entra o padrão JSON (JavaScript Object Notation). Apesar do nome, esta notação não serve apenas para programas feitos em JavaScript. Ela é utilizada de uma forma geral para tráfego de dados no REST.

A sintaxe de um JSON é a seguinte:

Figura 10 - Exemplo de JSON



O que é importante saber sobre o formato JSON:

- Cada objeto JSON é delimitado por chaves. Um Array é delimitado por colchetes.
- Cada informação deve ser separada por vírgula.
- Os dados são sempre representados em pares nome:valor. Isso permite que as informações sejam acessadas diretamente pelo seu nome. Assim, se eu quiser saber o telefone do cliente, posso usar algo do tipo “cliente.telefone”
- Um item pode ser dos tipos: string (sempre entre aspas), número, objeto (um outro objeto JSON, um array de objetos JSON, boolean ou nulo).

Apesar de bastante simples, o JSON é apenas isso. É importante mencionar que dados considerados binários, como imagens e áudio, também podem ser enviados dentro de objetos JSON através do formato BASE-64.

Se você quiser se aprofundar sobre o JSON, acesse a página https://www.json.org/json-en.html.

O JSON pode ser usado tanto na chamada da requisição como no retorno dela. Na inclusão de um cliente, por exemplo, o objeto JSON pode conter as informações do cliente. Se, por outro lado, precisamos de buscar uma lista de clientes, ela virá também no formato JSON.

Toda requisição HTTP retorna um código de status. Esse código é extremamente importante para a arquitetura REST mas, infelizmente, por vezes esquecido pelos desenvolvedores. Os status HTTP são divididos da seguinte forma:

- Informational Responses (100 – 199)
- Sucessfull Responses (200 – 299)
- Redirects (300 – 399)
- Client Errors (400 – 499)
- Server Errors (500 – 599)

Não vamos nos estender em explicar cada um dos status. No seu dia a dia, para cada tipo de operação você deve pesquisar qual o status HTTP mais se assemelha com o que você quer dizer.

Alguns exemplos:

200 – OK – A requisição foi realizada com sucesso. Por exemplo, quando pedimos um cliente ao servidor e ele retorna corretamente.

201 – Created – Um objeto foi criado no servidor. Usado, por exemplo, nos verbos POST

400 – Bad Request – Algo da sua requisição está errado. Por exemplo, você enviou alguma informação no JSON não reconhecida pelo servidor ou faltou alguma informação.

403 – Forbidden – Você não tem acesso ao recurso.

404- Not Found - Recurso não encontrado.

A última coisa que devemos saber sobre o REST é como acessar um determinado recurso. A fórmula é sempre a mesma:

PROTOCOLO://ENDEREÇO_DO_SERVIDOR/RECURSO?QUERYSTRING

Para explicar melhor, vamos usar alguns exemplos de requisições, todas com o verbo GET:

Exemplo 1: http://meubanco.com.br/cliente/1

http é o protocolo, essa parte não muda muito. Por vezes utilizaremos a versão segura do protocolo, que é o https.

meubanco.com.br é o servidor.

Cliente significa que queremos informações de clientes.

1 especifica exatamente qual cliente queremos

Exemplo 2: http://blog.com.br/posts?descricaoParcial=”pandemia”

Http e servidor são iguais.

A interrogação significa que iremos passar uma query (conhecida como querystring). Nesse caso, estamos dizendo que queremos descrições que contenham a palavra pandemia.

Exemplo 3: http://blog.com.br/posts/1

Busca o post de id 1

Exemplo 4: http://blog.com.br/posts/1/comments

Busca os comentários do post de id 1

Boas práticas no design de APIs Rest

Segue uma lista de boas práticas no design de APIs Rest. Ela serve como um checklist para quando for criar sua API. Boa parte delas já foram descritas no capítulo:

- Use o verbo correto para a ação que você vai tomar.
- Rest é orientado a recursos. O endpoint (rota) que for criado deve levar isso em conta.
- Um endpoint é a combinação de um verbo (GET, POST etc) e uma URI onde fica o recurso. Exemplo: GET /*comments*/1 busca (GET) o comentário de id 1.
- Use o status de resposta adequadamente. Consulte https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.
- Sempre retorne Json. Lembre-se de retornar o header de content correto.
- Não use verbos nas URIs. Lembre-se, URIs no Rest devem ter recursos (substantivos).
- Use plural para os recursos.
- Quando gerar um erro, retorne a mensagem de erro no body em formato Json.

- Essa é mais uma recomendação de design limpo: não aninhe recursos. Se você quer todos os livros de um autor, ao invés de usar */authors/5/books, use /books/?author=5*

- Filtros e paginações devem ser feitas na querystring. Exemplo*: /authors/?page=3;namePartial="Marcelo"*

- Os status *401 (Unauthorized)* e *403 (Forbidden)* são diferentes! O 401 significa que ele não está autorizado (não se autenticou no sistema), enquanto o 403 significa que ele se autenticou, mas não tem acesso ao recurso (acesso negado).

Capítulo 4. Introdução ao Django e Flesk

Depois de todos os conceitos da Web apresentados, fica muito fácil falarmos de Django e Flesk. Na verdade, esses frameworks não trazem quase nenhuma novidade conceitual. São apenas respostas da comunidade Python para problemas que outras comunidades já enfrentam.

O Django é um framework que permite o desenvolvimento full-stack (tanto da parte do servidor quanto da geração do cliente html e CSS)

Enquanto isso, o Flask é um framework leve para geração de API REST.

Na verdade, os dois fazem um pouco de tudo. A nossa definição baseia-se única e exclusivamente na forma como o mercado e a comunidade veem os dois produtos.

O Django é um pouco mais complexo, com mais detalhes arquiteturais a ser conhecidos, enquanto que o Flesk basicamente é um framework para desenvolver APIs Web. Vamos falar um pouco mais em detalhes sobre o Django

Django em detalhes

A história da criação do Django explica muito sobre sua arquitetura. Ele foi criado por web designers de um jornal. Sua maior utilização, na época, era permitir que o jornal conseguisse criar rapidamente “hot-sites” com determinadas coberturas especiais.

A ideia deu muito certo, e o Django se tornou um framework centrado em aplicações de bancos de dados.

Sua maior utilização atual é a de criar páginas estáticas com CRUDs de acesso a dados.

O Django trabalha com um modelo chamado de MTV.

MTV significa Model Template View. Vamos explicar passo a passo cada um desses conceitos a seguir.

O “M” significa “Modelo”. É uma representação, em Python, do nosso modelo de banco de dados. Nada mais é do que uma classe Python com informações importantes sobre como a sua tabela será. Na verdade, podemos dizer que cada tabela terá um modelo no Django. É possível, inclusive, criar o seu modelo de banco de dados a partir do modelo criado no Django. Já temos embutido no framework um banco de dados SQLLite, mas o Django aceita vários bancos de dados diferentes.

O “T” significa “Template”. Os templates são arquivos HTML que podem conter uma meta-linguagem que permite torná-los dinâmicos, ou seja, podemos incluir informações no HTML de forma programática. Os templates recebem informação (vindas da view) que podem ser mostradas para o usuário final. Um exemplo, podemos mandar uma lista de dados para o usuário final que será processada e renderizada no template.

O “V” significa View. O View é a ligação entre o Model e o Template. É no View que boa parte da programação ocorre. É aqui que recebemos as requisições HTTP e fazemos os tratamentos necessários.

Enquanto o Template é em HTML com uma meta-linguagem própria do Django, o View e o Model são Python puro, o que já não é novidade para vocês.

Apps e Projects

Dois conceitos importantes no Django são os de Apps e Projects.

Quando você inicia seu website, está criando um projeto. Basicamente, ele contém as informações básicas para o seu website, como arquivos de configuração e de definição de que endereços serão chamados por cada app.

Então, chegamos no conceito de App. Na verdade, o termo não é muito bom, e vez ou outra a comunidade se pergunta se esse é o melhor termo.

Application (ou app, para os íntimos) é um módulo autocontido. Ser autocontido significa que ele não depende do projeto em si. Na verdade, você pode usar apps de terceiros (o Django já disponibiliza várias apps e várias outras podem ser encontradas). Acredito que a melhor definição de uma app é um módulo.

Na prática, é assim que usamos nossas apps, como módulos de nosso projeto web (nosso web-site)

Flask

Depois de toda a teoria sobre APIs data, não há muito a falar do Flask.

Basicamente, o Flask é o framework que nos permite criar um servidor REST (que nada mais é do que um servidor Web que recebe requisições no padrão REST, as processa, e as retorna com um status e com um payload [body]).

No Flask, usamos o conceito (ou abstração, como preferir) de application. Uma application nada mais é do que uma instância do Flask que colocamos no ar e que nos permite executar as instruções necessárias.

Vamos ver isso em detalhes nas aulas práticas.

Apenas para lembrar: apesar de não conhecermos essa parte do Flask, é possível com ele também retornar páginas HTML como o Django faz. Pessoalmente, acredito que o forte do Flask é ser um bom framework para REST. Deixemos o Django fazer o trabalho sujo de renderizar páginas, combinado?

Referências

JSon Oraganization website. Disponível em: https://www.json.org/json-en.html. Acesso em: 30 jul. 2020.

Wikipedia. *Conceitos de REST*. Disponível em: https://en.wikipedia.org/wiki/Representational_state_transfer#References. Acesso em: 30 jul. 2020.

Wikipedia. *EcmaScript*. Disponível em: https://en.wikipedia.org/wiki/ECMAScript. Acesso em: 30 jul. 2020.

W3 Schools. Disponível em: https://www.w3schools.com/. Acesso em: 30 jul. 2020.

CSS Tricks. Disponível em: https://css-tricks.com/. Acesso em: 30 jul. 2020.

Mozilla Foundation – Developer Site. Disponível em https://developer.mozilla.org/. Acesso em: 30 jul. 2020.