



Aprenda com quem faz

# Python para Aplicações Web

Prof. Lucas Crispim

2023



## SUMÁRIO

Capítulo 1. Introdução aos sistemas web.....	4
Capítulo 2. HTML .....	6
2.1. Escolha do editor.....	6
2.2. Primeira página web .....	6
2.3. Validadores.....	9
2.4. Estrutura do documento.....	10
2.5. Tags.....	11
Capítulo 3. CSS.....	21
3.1. Seletores.....	23
Capítulo 4. Eventos em HTML .....	31
Capítulo 5. Django.....	36
5.1. Desenvolvimento Web e Fluxos .....	36
5.2. Instalação .....	38
5.3. Criando o primeiro aplicativo .....	42
5.4. Criando as views .....	43
5.5. Models .....	45
5.6. Templates .....	48
Capítulo 6. Flask.....	52
6.1. Instalação .....	52
6.2. Templates .....	54
6.3. Configuração .....	57
6.4. Models .....	58
6.5. Roteamento .....	60
Referências .....	70



**XP**e

# > Capítulo 1



## Capítulo 1. Introdução aos sistemas web

---

O início da internet foi basicamente composto de páginas de texto linkadas umas com as outras, onde era permitido passear entre elas bastando clicar em links de acesso. A evolução da internet dos tempos atuais é inegável. As operações mais complexas podem ser realizadas através dela (de serviços bancários até todo tipo de comunicação). Basicamente, o usuário, quando vai utilizar um serviço baseado em web, interage com uma interface e realiza as operações desejadas. Este fato traz a percepção que existe uma certa organização (estrutura) nos serviços hospedados na web. A primeira a ser citada aqui é justamente aquela que o usuário interage (frontend). A camada onde são processadas as informações é conhecida como backend. Nessa camada a interação do usuário com o frontend é recebida, processada e devolvida ao usuário. Pode-se imaginar, por exemplo, num aplicativo bancário que quando é realizada uma transferência o valor de saldo na conta é atualizado de modo quase instantâneo. O usuário nesse caso interage com o frontend, enviando uma ordem (transferência de valor), o backend recebe a operação desejada com seus parâmetros, a executa e devolve o valor de saldo. Não obstante, as duas camadas já elucidadas aqui, existe uma terceira camada. Este é o lugar onde os dados devem ficar armazenados para serem consultados, atualizados ou até mesmo deletados.

Esta introdução é um pequeno prólogo do que será trabalhado neste módulo. Ademais, sempre será encorajado aos leitores serem curiosos em relação às tecnologias apresentadas aqui.



**XP**e

## > Capítulo 2



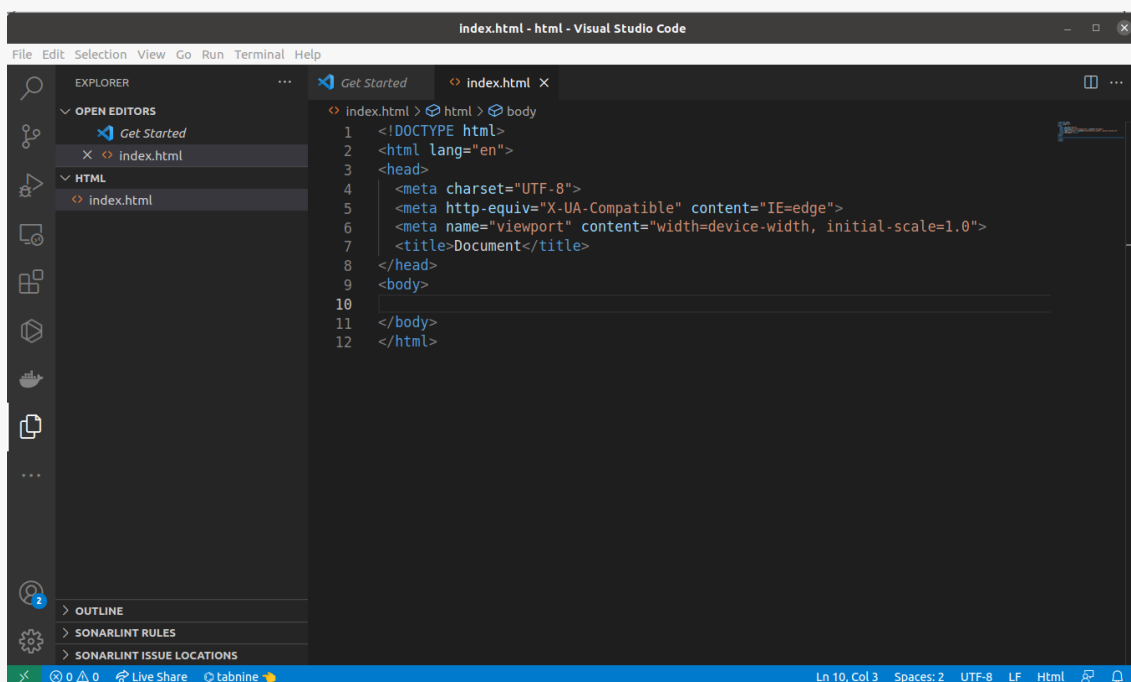
## Capítulo 2. HTML

O HTML (*HyperText Markup Language*) é a linguagem de marcação padrão para documentos que serão exibidos em algum navegador web. Basicamente, o HTML é a estrutura de uma página *web*. Além do HTML, o CSS é utilizado para dar aparência/apresentação e o javascript para adicionar funcionalidade/comportamento nas páginas web.

### 2.1. Escolha do editor

Neste curso iremos utilizar o editor Visual Studio Code (<https://code.visualstudio.com/>). A razão de escolher este editor se deu em função da sua facilidade de uso e popularidade. A Figura 1 mostra o Visual Studio Code com uma quantidade de código HTML escrito.

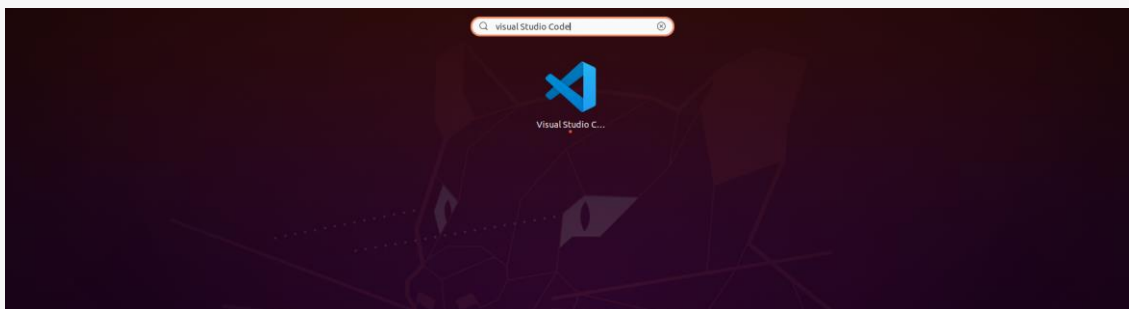
Figura 1 - Aparência do editor Visual Studio Code.



### 2.2. Primeira página web

Abra o editor Visual Studio Code. Veja a Figura 2. Observe que como estamos usando o sistema operacional linux ubuntu a aparência do sistema deve ser similar ao mostrado na Figura 2.

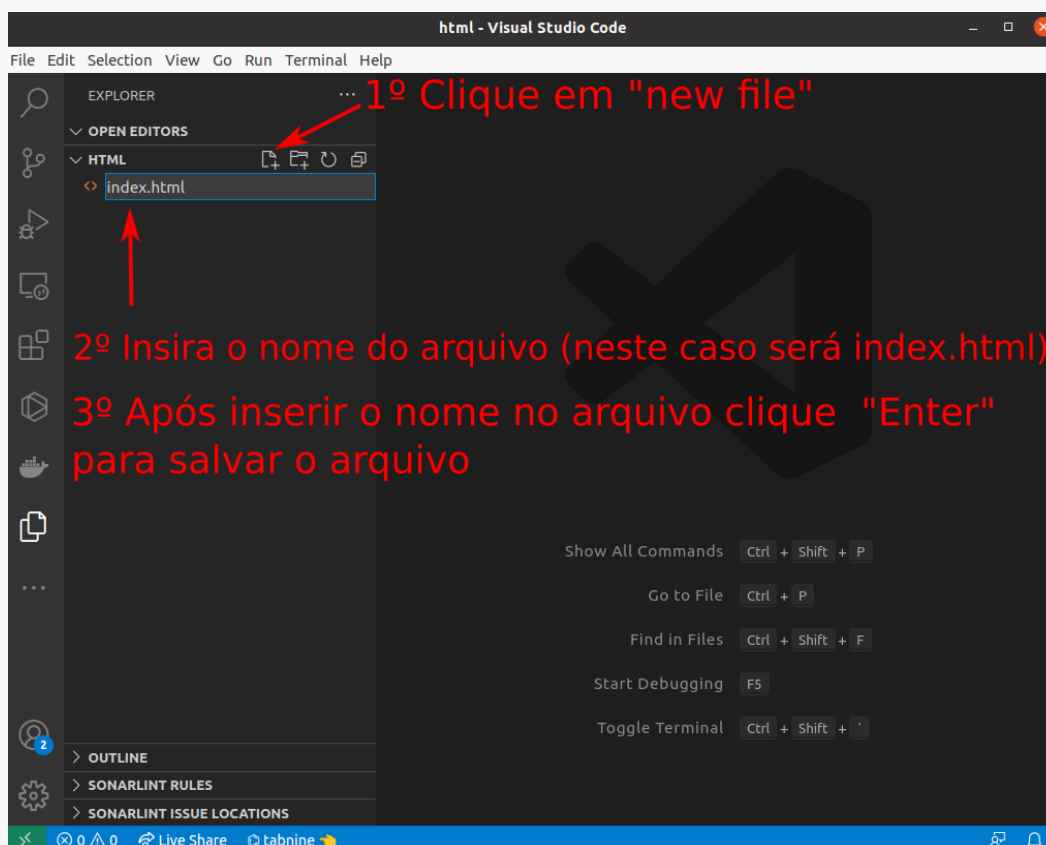
Figura 2 - Abertura do Visual Studio Code no Ubuntu.



Com o Visual Studio Code aberto, clique em Open Folder (atalho Ctrl + k, e depois Ctrl + o). Escolha uma pasta previamente criada. No caso deste exemplo em particular, a pasta HTML é utilizada e foi criada em “home/lucas/desenvolvimento\_web”.

Crie um arquivo com o nome index.html. Clique em “new file” e depois insira o nome do arquivo (neste caso será index.html). As instruções são mostradas na Figura 3.

Figura 3 - Criação do primeiro arquivo HTML.



Com o arquivo aberto, o seguinte trecho de código será inserido no arquivo:

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

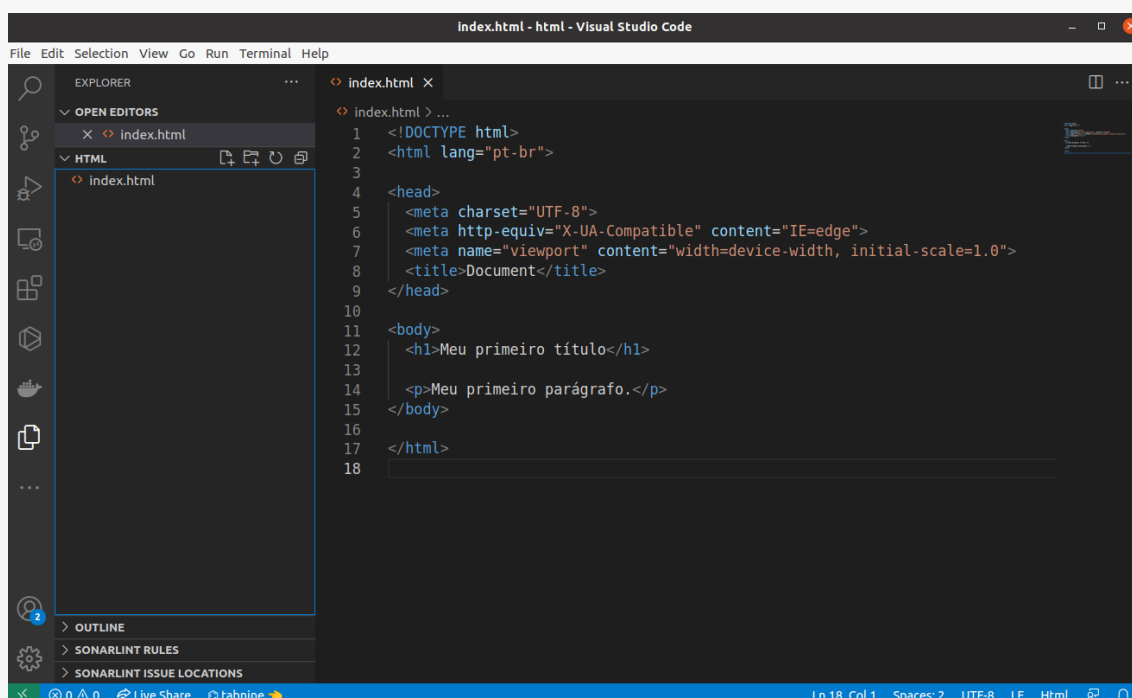
<body>
  <h1>Meu primeiro título</h1>

  <p>Meu primeiro parágrafo.</p>
</body>

</html>
```

No editor temos a aparência mostrada na Figura 4.

Figura 4 - Primeiro arquivo HTML.



Neste ponto é possível visualizar nossa primeira página web. Clique no arquivo index.html na pasta selecionada (neste caso a pasta é HTML que



está dentro de “home/lucas/desenvolvimento\_web”). Agora será aberta uma aba no seu navegador padrão e o conteúdo mostrado será como o da Figura 5.

Figura 5 - Primeira página *web*.



A primeira página produzida expõe uma estrutura do documento que deve ser seguida. Neste exemplo também é possível observar duas inserções por parte do criador do documento. A primeira inserção é o título do documento (“Meu primeiro título”) que é envolto na estrutura `<h1></h1>`. A segunda inserção é o início de um parágrafo (“Meu primeiro parágrafo”) que é envolto na estrutura `<p></p>`. Ambas as inserções mostram que o conteúdo deve ser geralmente envolto em algum tipo de código que é comumente conhecido como “Tags”.

### 2.3. Validadores

Antes que páginas mais complexas sejam criadas é necessário que seja possível validar os arquivos HTML. Existem diversos validadores de código HTML disponíveis na internet. Nesta seção será recomendada a página “<https://validator.w3.org/>”. Existem três possíveis modos de validação no site: validação por URL (endereço da página *web*), validação por

arquivo (inserir o arquivo HTML diretamente no site) ou inserir o código HTML diretamente no site. Como mostrado na Figura 6, foi escolhida a opção de inserção de código diretamente no site.

Figura 6 - Ferramenta de validação de HTML.

**Nu Html Checker**

This tool is an ongoing experiment in better HTML checking, and its behavior remains subject to change

Showing results for contents of text-input area

Checker Input

Show ☒ source ☐ outline ☐ image report [Options...](#)

Check by **text input** ☐ css

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <h1>Meu primeiro titulo</h1>
  <p>Meu primeiro parágrafo.</p>
</body>
```

[Check](#)

Use the Message Filtering button below to hide/show particular messages, and to see total counts of errors and warnings.

[Message Filtering](#)

**Document checking completed. No errors or warnings to show.**

**Source**

```
1. <!DOCTYPE html>
2. <html lang="pt-br">
3.
4. <head>
5.   <meta charset="UTF-8">
6.   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7.   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8.   <title>Document</title>
9. </head>
10.
11. <body>
12.   <h1>Meu primeiro titulo</h1>
13.   <p>Meu primeiro parágrafo.</p>
14. </body>
15.
16. </html>
```

Used the HTML parser.  
Total execution time 3 milliseconds.

O código HTML utilizado no primeiro exemplo de página *web* foi validado pela ferramenta (sem erros e nem *warnings*).

## 2.4. Estrutura do documento

A estrutura do código utilizado vai ser explicada nesta seção. O primeiro elemento do código é `<!DOCTYPE html>`. A declaração `<!DOCTYPE>` representa o tipo de documento e ajuda os navegadores a exibir as páginas *web* corretamente. Ela deve aparecer no topo da página, apenas uma vez

(antes de qualquer tag HTML). A declaração `<!DOCTYPE html>` identifica, portanto, que se trata de um documento HTML5.

A declaração `<html lang="pt-br">` representa a raiz de um documento HTML. Essa tag é um container para todos os outros elementos HTML (exceto `<!DOCTYPE>`). Coloque o atributo “lang” para declarar o idioma da página *web*. Este atributo se destina a ajudar os motores de busca e navegadores.

A tag `<head>` é um container para configurações do documento HTML. Normalmente, as configurações contidas são: título do documento (`<title> </title>`), codificação do documento (`<meta charset="UTF-8">`), além de outras configurações (estilização e código javascript).

A tag `<body>` define o corpo do documento. Esta tag contém todo conteúdo de um documento HTML, como títulos, parágrafos, imagens, hiperlinks, tabelas, listas e outros mais. Só pode existir uma tag `<body>` em um documento HTML.

As tags apresentadas configuram a estrutura básica para criação de páginas web.

## 2.5. Tags

Nesta seção apresentaremos as principais tags que serão utilizadas nos projetos futuros. Como visto anteriormente, as tags no HTML geralmente seguem uma sintaxe: `<elemento nome_do_atributo="valor_do_atributo"> Conteúdo </elemento>`. Por exemplo:

```
<p lang="pt-br"> Conteúdo </p>
```

Nesse caso o nome do atributo é “lang” (linguagem) e seu conteúdo é “pt-br” (português brasileiro). Muito embora esse tipo de estrutura seja comum para as tags, existe também o caso em que a tag não possui um par

de fechamento. Um exemplo desse tipo de tag é a de inserção de imagem na página:

```
.
```

No exemplo anterior, o atributo é “src” e seu valor é o caminho da imagem (“../../imagens/IGTI.png”).

### 2.5.1. Títulos

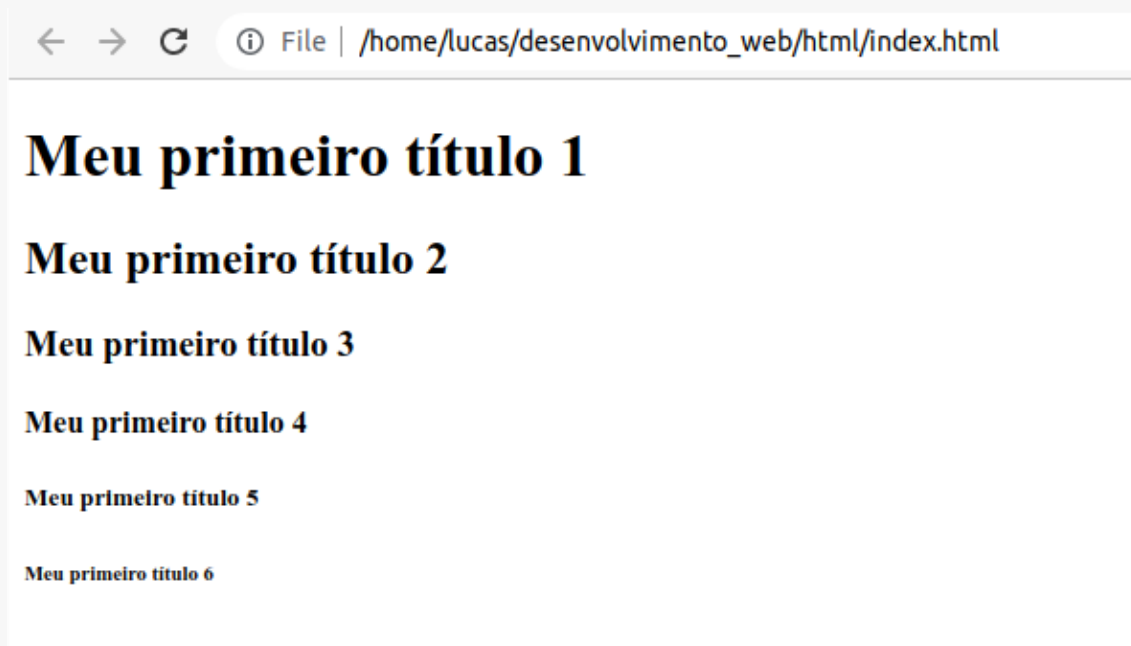
Os títulos no HTML são definidos com as tags de `<h1>` a `<h6>`. A ordem de importância é decrescente na ordem crescente dos números. Utilizando o documento criado anteriormente (index.html), a Figura 7 mostra um código de exemplo que utiliza as tags de títulos.

Figura 7 - Exemplo de utilização de títulos.

```
<> index.html > html > body > h6
1  <!DOCTYPE html>
2  <html lang="pt-br">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <title>Document</title>
9  </head>
10
11 <body>
12   <h1>Meu primeiro título 1</h1>
13   <h2>Meu primeiro título 2</h2>
14   <h3>Meu primeiro título 3</h3>
15   <h4>Meu primeiro título 4</h4>
16   <h5>Meu primeiro título 5</h5>
17   <h6>Meu primeiro título 6</h6>
18 </body>
19
20 </html>
```

O resultado da utilização das tags de título é mostrado na Figura 8.

Figura 8 - Cabeçalhos.



### 2.5.2. Parágrafos

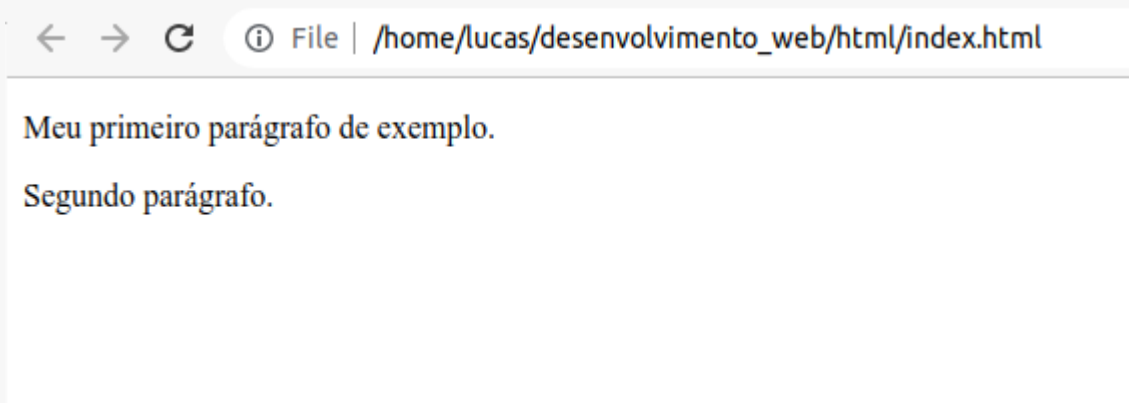
A tag para criar parágrafos foi utilizada no primeiro exemplo de documento HTML que foi criado neste capítulo. Na Figura 9 é mostrado um código de exemplo de utilização da tag de parágrafo.

Figura 9 - Exemplo de utilização de parágrafos.

```
<> index.html > html > body > p
1  <!DOCTYPE html>
2  <html lang="pt-br">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <title>Document</title>
9  </head>
10
11 <body>
12   <p>
13     Meu primeiro parágrafo de exemplo.
14   </p>
15   <p>
16     Segundo parágrafo.
17   </p>
18 </body>
19
20 </html>
```

O resultado da utilização dessa tag é mostrado na Figura 10.

Figura 10 - Parágrafos.



### 2.5.3. HyperLinks

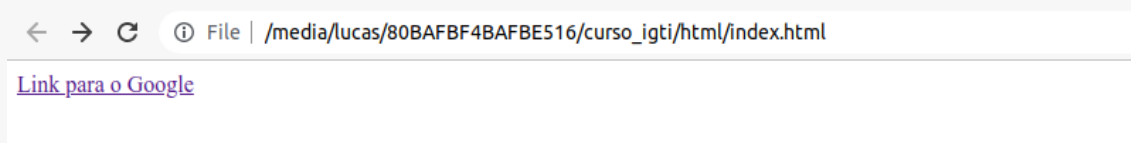
A tag `<a>` é utilizada para vincular uma página a outra. O atributo mais importante desta tag é `href`, que indica o destino do *link*. O exemplo do uso dessa tag é mostrado na Figura 11.

Figura 11 - Exemplo da inserção de um link na página.

```
<> index.html > html > body > a
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta http-equiv="X-UA-Compatible" content="IE=edge">
6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
7    <title>Document</title>
8  </head>
9  <body>
10  <a href="https://www.google.com.br"> Link para o Google </a>
11  </body>
12  </html>
```

O resultado é mostrado na Figura 12.

Figura 12 - Link para o google.



Ao clicar no link, a página é redirecionada para o google. O uso dessa tag também pode conduzir o browser a exibir páginas externas.

#### 2.5.4. Imagens

A tag `<img>` é usada para incorporar uma imagem em uma página HTML. As imagens não são tecnicamente inseridas em uma página web, elas estão vinculadas à página. A tag `<img>` tem dois atributos obrigatórios:

- `src` - Especifica o caminho para a imagem.
- `alt` - Especifica um texto alternativo para a imagem se ela por algum motivo não puder ser exibida.

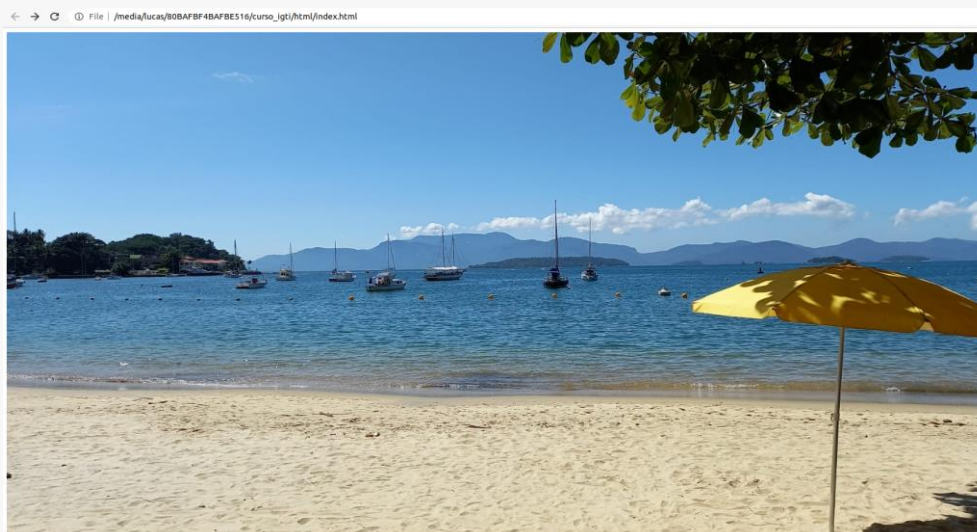
O exemplo do uso dessa tag é mostrado na Figura 13.

Figura 13 - Exemplo da inserção de uma imagem na página.

```
<> index.html > html > body > img
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta http-equiv="X-UA-Compatible" content="IE=edge">
6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
7    <title>Document</title>
8  </head>
9  <body>
10   
11 </body>
12 </html>
```

O resultado é mostrado na Figura 14.

Figura 14 - Imagem inserida na página.



#### 2.5.5. Tables

A tag `<table>` define uma tabela HTML. Além da tag principal (`<table>`), existem mais elementos (`<tr>`, `<th>` e `<td>`). O elemento `<tr>` define uma linha de tabela, o elemento `<th>` define um cabeçalho de tabela e o elemento `<td>` define uma célula de tabela. O exemplo do uso dessa tag é mostrado na Figura 15.

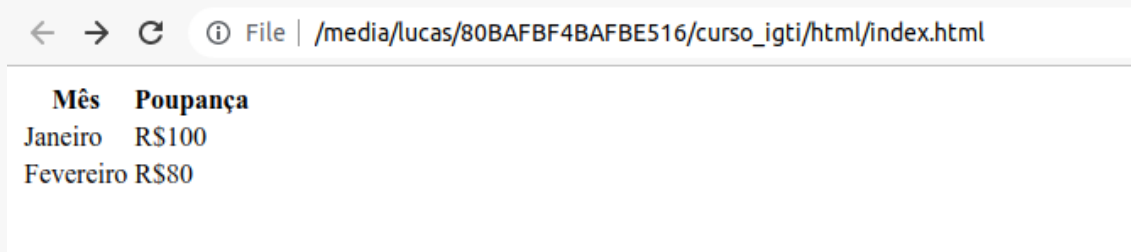
Figura 15 - Exemplo da inserção de uma tabela na página.

```
<> index.html > html > body > table > tr > td
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta http-equiv="X-UA-Compatible" content="IE=edge">
6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
7    <title>Document</title>
8  </head>
9  <body>
10   <table>
11     <tr>
12       <th>Mês</th>
13       <th>Poupança</th>
14     </tr>
15     <tr>
16       <td>Janeiro</td>
17       <td>R$100</td>
18     </tr>
19     <tr>
20       <td>Fevereiro</td>
21       <td>R$80</td>
22     </tr>
23   </table>
24 </body>
25 </html>
```



O resultado é mostrado na Figura 16.

Figura 16 - Tabela inserida na página.

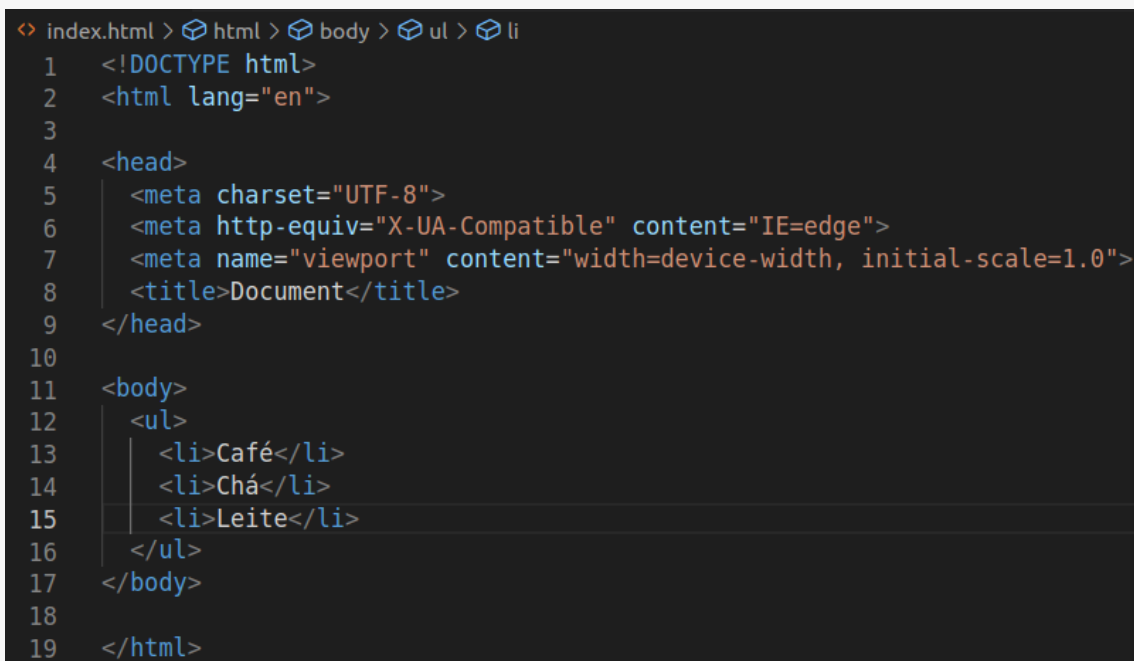


Mês	Poupança
Janeiro	R\$100
Fevereiro	R\$80

### 2.5.6. Listas

A tag `<ul>` define uma lista não ordenada (com marcadores). A tag `<li>` é utilizada para inserir um item na lista. O exemplo do uso dessa tag é mostrado na Figura 17.

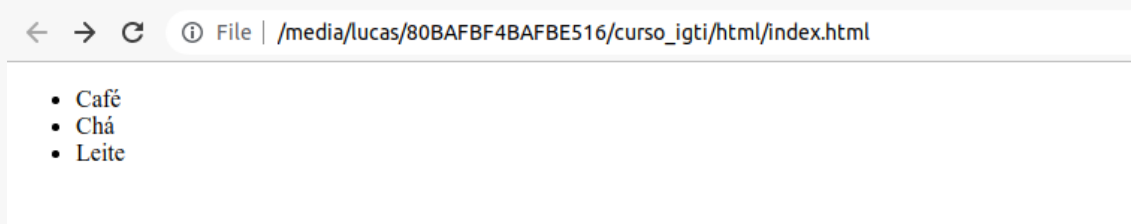
Figura 17 - Exemplo da inserção de uma lista não ordenada na página.



```
<> index.html > html > body > ul > li
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <title>Document</title>
9  </head>
10
11 <body>
12   <ul>
13     <li>Café</li>
14     <li>Chá</li>
15     <li>Leite</li>
16   </ul>
17 </body>
18
19 </html>
```

O resultado é mostrado na Figura 18.

Figura 18 - Lista não ordenada inserida na página.



Existem, como as listas não ordenadas, as ordenadas, e sua tag é `<ol>` e os itens são marcados pela tag `<li>`, do mesmo modo que nas listas não ordenadas.

### 2.5.7. Formulários

A tag `<form>` é usada para criar um formulário HTML. Ela pode conter um ou mais dos seguintes itens:

- `<input>`
- `<textarea>`
- `<button>`
- `<select>`
- `<option>`
- `<optgroup>`
- `<fieldset>`
- `<label>`
- `<output>`

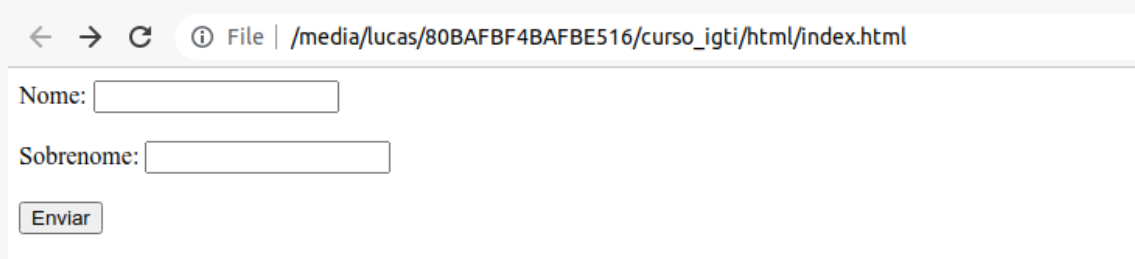
Todos os elementos mostrados acima têm uma função específica no formulário. O exemplo do uso dessa tag é mostrado na Figura 19.

Figura 19 - Exemplo da inserção de um formulário na página.

```
index.html > html > body > form > input
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <title>Document</title>
9  </head>
10
11 <body>
12   <form action="" method="">
13     <label for="nome">Nome:</label>
14     <input type="text" id="nome" name="nome"><br><br>
15     <label for="sobrenome">Sobrenome:</label>
16     <input type="text" id="sobrenome" name="sobrenome"><br><br>
17     <input type="submit" value="Enviar">
18   </form>
19 </body>
20
21 </html>
```

Pode-se perceber que apenas alguns itens dos que foram mostrados foram incorporados no formulário. Para o entendimento mínimo do uso dessa tag os itens que foram incorporados no formulário são suficientes. Os atributos *action* e *method* serão abordados com mais detalhes nos próximos capítulos. O resultado é mostrado na Figura 20.

Figura 20 - Formulário inserido na página.



← → ↻ ⓘ File | /media/lucas/80BAFBF4BAFBE516/curso\_igti/html/index.html

Nome:

Sobrenome:



**XP**e

## > Capítulo 3



## Capítulo 3. CSS

O css é a “linguagem” utilizada para dar estilo ao documento HTML. O css descreve como os elementos HTML devem ser exibidos. Para começar, é necessário mostrar como inserir o css no documento HTML.

- Dentro de um elemento:

Esta forma de inserir o estilo é individual do elemento. A propriedade *style* é utilizada e pode conter um ou mais valores. O exemplo do uso é mostrado na Figura 21.

Figura 21 - Exemplo de uso do css dentro da tag.

```
index.html > html > body > h3
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <title>Document</title>
9  </head>
10
11 <body>
12   <h2 style="background-color: red;">Vermelho</h2>
13   <h3 style="background-color: blue;">Azul</h3>
14 </body>
15
16 </html>
```

Nesse exemplo, dentro do *style* foi utilizada a propriedade *background-color*. O resultado do uso deste estilo é mostrado na Figura 22.

Figura 22 - Resultado do uso do css dentro da tag.



- No cabeçalho (dentro da tag `<head>`):

Nesta situação os estilos das tags são escritas em uma tag `<style>` dentro do `<head>` do documento. O exemplo mostrado na Figura 21 é repetido, porém, agora o estilo ficará dentro do `<head>` da página. Este novo exemplo é mostrado na Figura 23.

Figura 23 - Exemplo de uso do css no `<head>` da página.

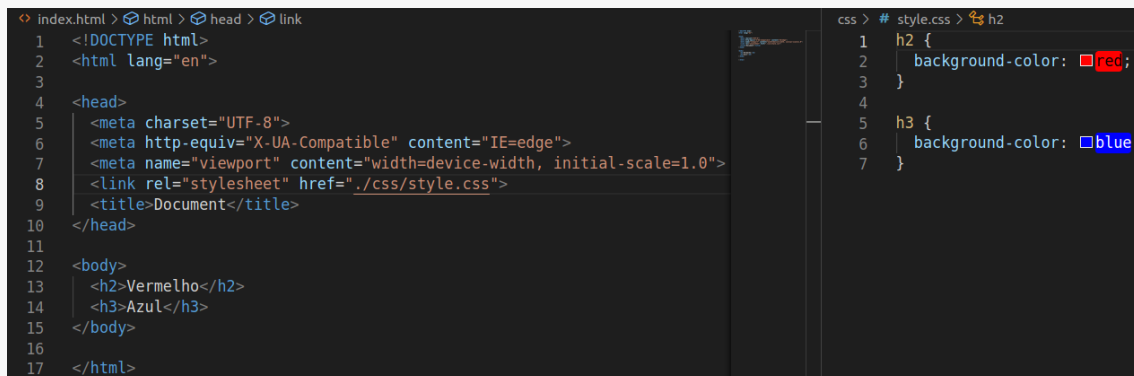
```
<> index.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta http-equiv="X-UA-Compatible" content="IE=edge">
7      <meta name="viewport" content="width=device-width, initial-scale=1.0">
8      <title>Document</title>
9
10     <style>
11         h2 { background-color: red; }
12         h3 { background-color: blue; }
13     </style>
14
15 </head>
16
17 <body>
18     <h2>Vermelho</h2>
19     <h3>Azul</h3>
20 </body>
21
22 </html>
```

O resultado do uso deste style é exatamente o mesmo do mostrado na Figura 22.

- Arquivo externo:

A terceira maneira da aplicação do css em uma página HTML é colocar essa estilização em um arquivo externo. O exemplo será exatamente o mesmo utilizado no primeiro e segundo caso de estilização. A inserção da estilização externa foi realizada com a tag `<link>` na linha 8 do documento index.html. Na Figura 24 são mostrados os dois arquivos (index.html e style.css).

Figura 24 - Exemplo de uso do css em arquivo externo.



```

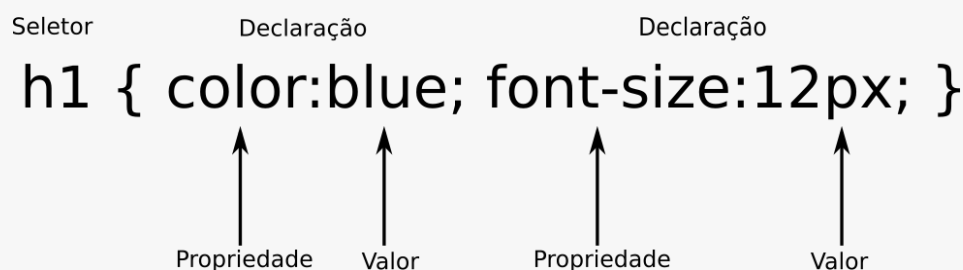
index.html > html > head > link
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <link rel="stylesheet" href="/css/style.css">
9   <title>Document</title>
10 </head>
11
12 <body>
13   <h2>Vermelho</h2>
14   <h3>Azul</h3>
15 </body>
16
17 </html>

css > # style.css > h2
1 h2 {
2   background-color: red;
3 }
4
5 h3 {
6   background-color: blue;
7 }
  
```

Esta última maneira de estruturar o css deve ser encorajada nos projetos que serão desenvolvidos neste curso. Pode-se perceber que desta forma cada documento HTML pode ter seu estilo em um arquivo separado. Este modo de estruturar um projeto o torna muito mais organizado.

A sintaxe css é mostrada na Figura 25.

Figura 25 - Exemplo de sintaxe css.



### 3.1. Seletores

Na seção passada foram mostrados modos de estruturar a estilização do documento HTML. Nesta seção será mostrado como são utilizados os seletores no css para estilizar o documento HTML.

- Seletor de elemento:

Este seletor é aquele que utilizamos nos exemplos das Figuras 23 e 24. Este tipo de seletor afeta todos os elementos com a mesma tag no documento HTML. O exemplo de aplicação é mostrado na Figura 26.

Figura 26 - Exemplo de uso do seletor de elementos.

```

index.html > html > head > meta
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <link rel="stylesheet" href="./css/style.css">
9    <title>Document</title>
10 </head>
11
12 <body>
13   <h2>Vermelho</h2>
14   <h3>Azul</h3>
15   <h2> Vermelho</h2>
16 </body>
17
18 </html>

# style.css
css > # style.css > h2
1  h2 {
2    background-color: red;
3  }
4
5  h3 {
6    background-color: blue;
7  }

```

O resultado do uso desse seletor é mostrado na Figura 27.

Figura 27 - Exemplo de aplicação do seletor de elementos.



É possível perceber que esse seletor alterou o estilo dos elementos que tem a tag `<h2>`. Pode ser requerido que a estilização seja feita em um

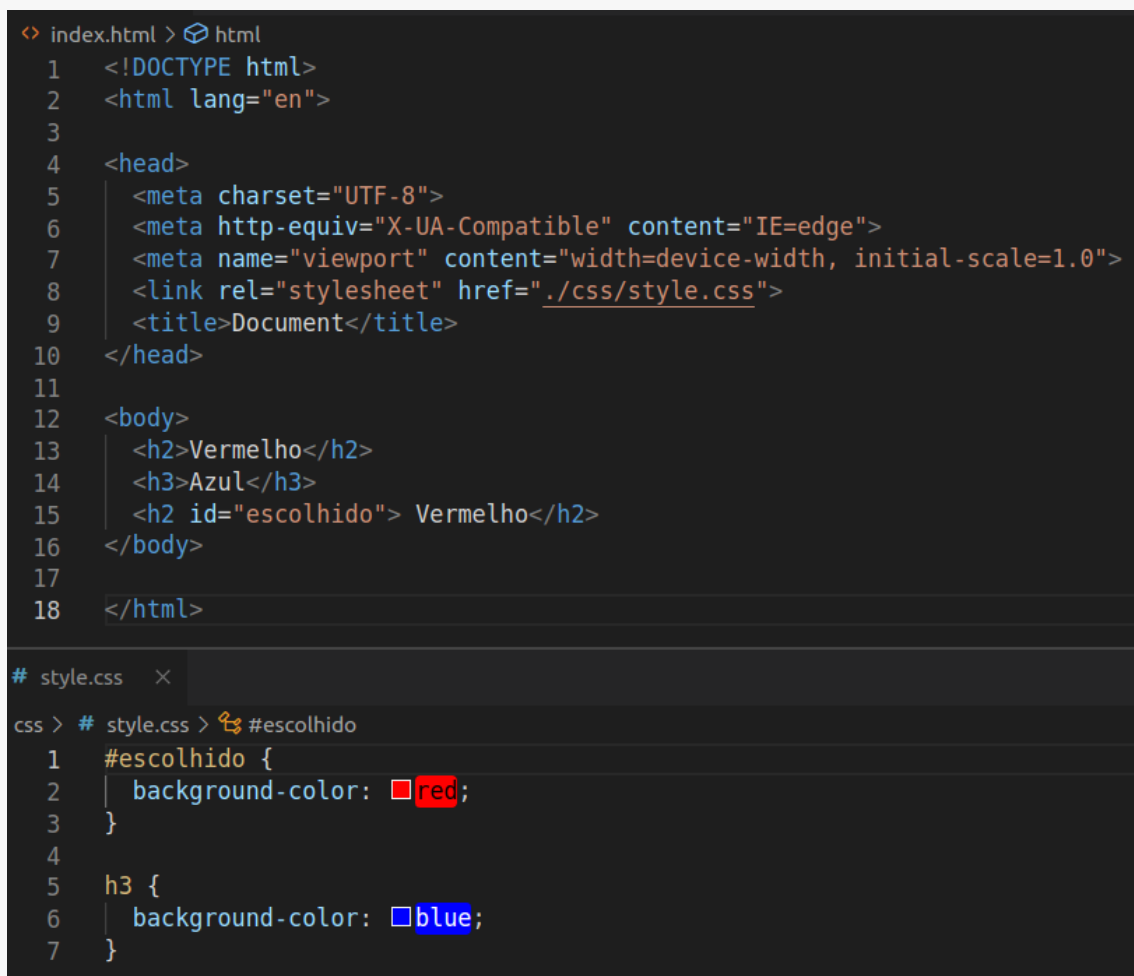


elemento específico. Caso isso aconteça, outros seletores podem ser usados.

- Seletor de id:

Pode-se aplicar estilização em um elemento apenas. Para isso é utilizado o seletor de id. O exemplo de uso desse seletor é mostrado na Figura 27. Ele é iniciado com o símbolo # e seguido pelo id do elemento (sem espaço).

Figura 28 - Exemplo de uso do seletor de elementos.



```
index.html > html
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <link rel="stylesheet" href="./css/style.css">
9   <title>Document</title>
10 </head>
11
12 <body>
13   <h2>Vermelho</h2>
14   <h3>Azul</h3>
15   <h2 id="escolhido"> Vermelho</h2>
16 </body>
17
18 </html>

# style.css
css > # style.css > #escolhido
1 #escolhido {
2   background-color: red;
3 }
4
5 h3 {
6   background-color: blue;
7 }
```

O resultado do uso desse seletor é mostrado na Figura 29.

Figura 29 - Exemplo da aplicação do seletor de id.

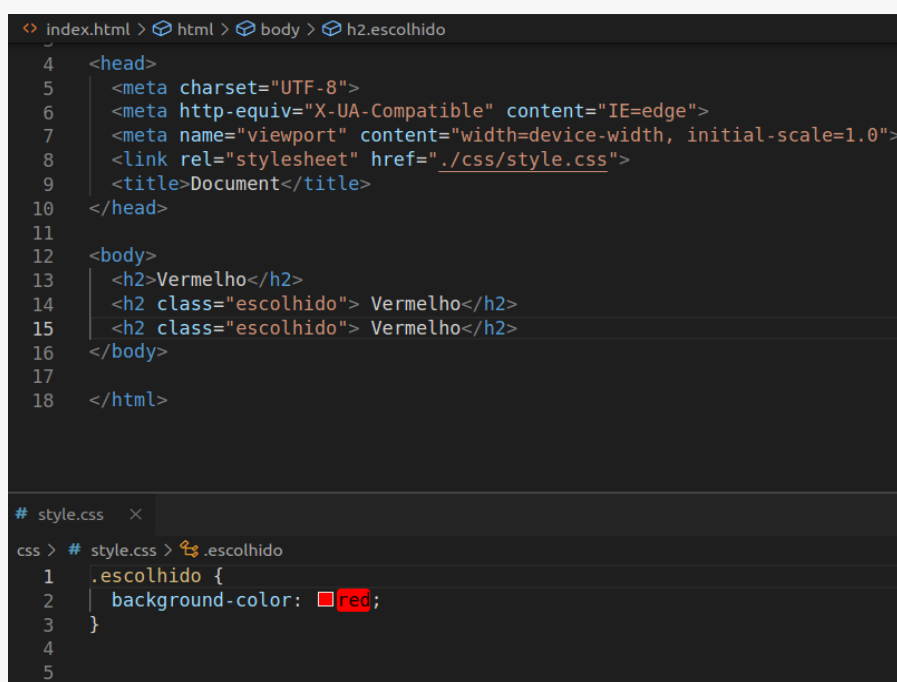


Esse exemplo ilustra o modo estilização de um único elemento pelo identificador dele (id). O leitor atento pode se perguntar o porquê de não colocar o estilo dentro do elemento. Esse fato pode ser explicado pela razão que já foi mostrada anteriormente. Os projetos serão construídos de modo a ficar mais organizado.

- Seletor de classe:

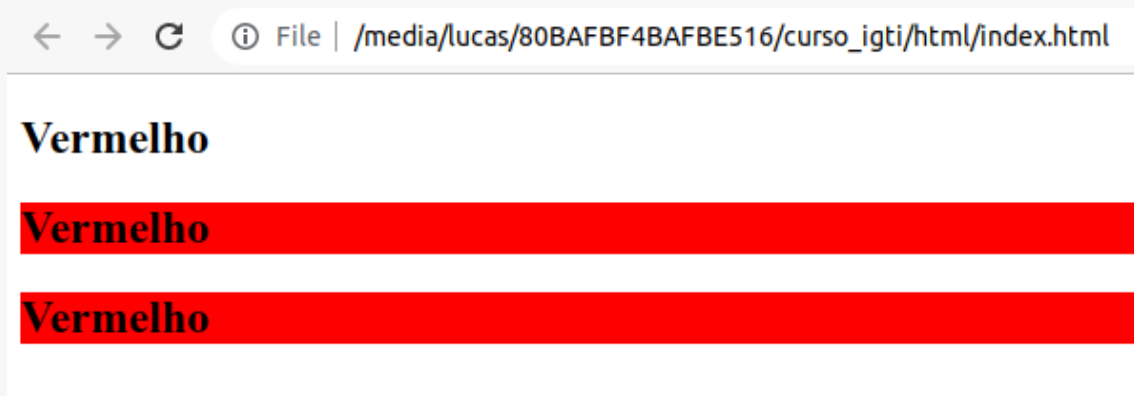
Este tipo de seletor pode estilizar vários elementos (os extremos: um ou todos). Seu modo de uso é iniciado com o símbolo de ponto (.) e seguido pelo nome da classe sem espaço, como podemos observar na Figura 30.

Figura 30 - Exemplo de uso do seletor de classe.



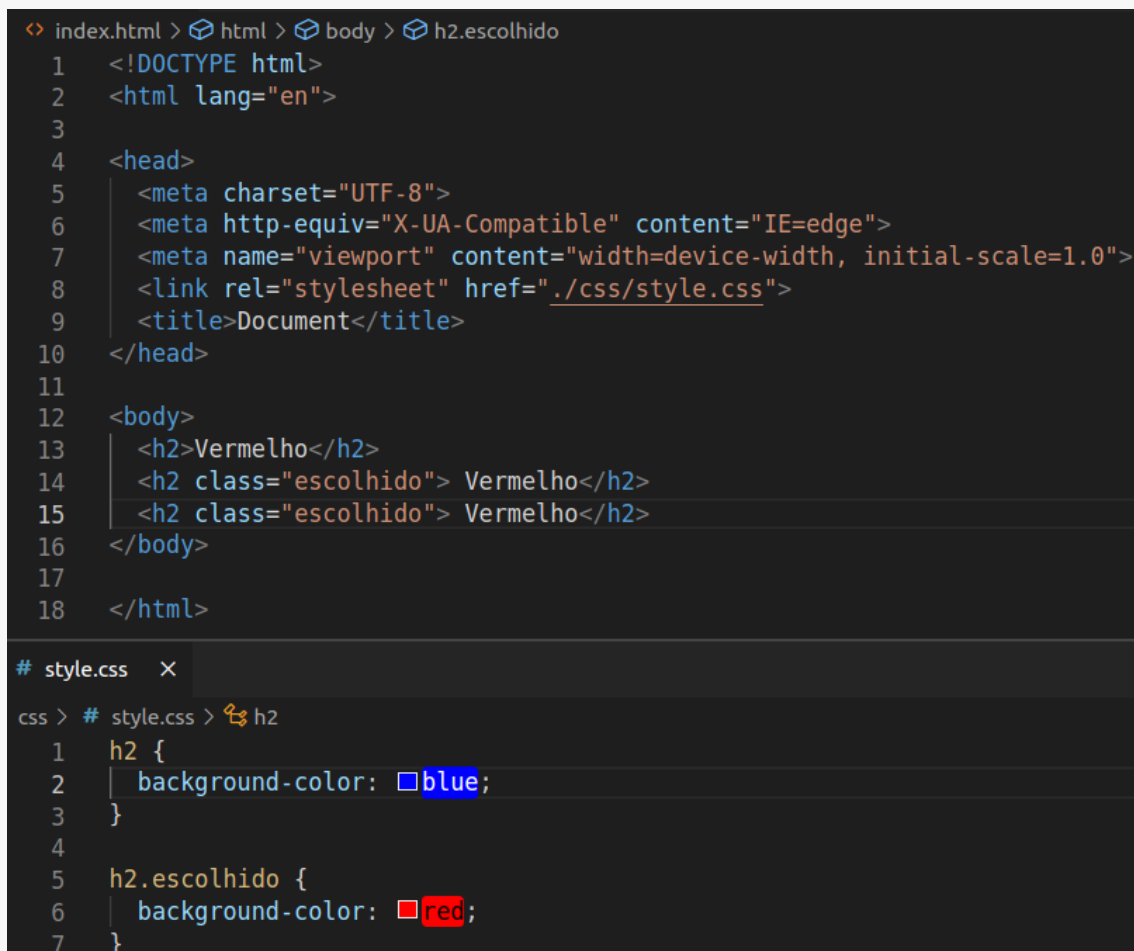
O resultado do uso desse seletor é mostrado na Figura 31.

Figura 31 - Exemplo de aplicação do seletor de classe.



Esse seletor pode ser combinado com elementos (tag.nome\_da\_classe). O exemplo de uso é mostrado na Figura 32.

Figura 32 - Exemplo de uso do seletor de classe com elemento.



O resultado do uso desse seletor é mostrado na Figura 33.

Figura 33 - Exemplo de aplicação do seletor de classe e elemento.

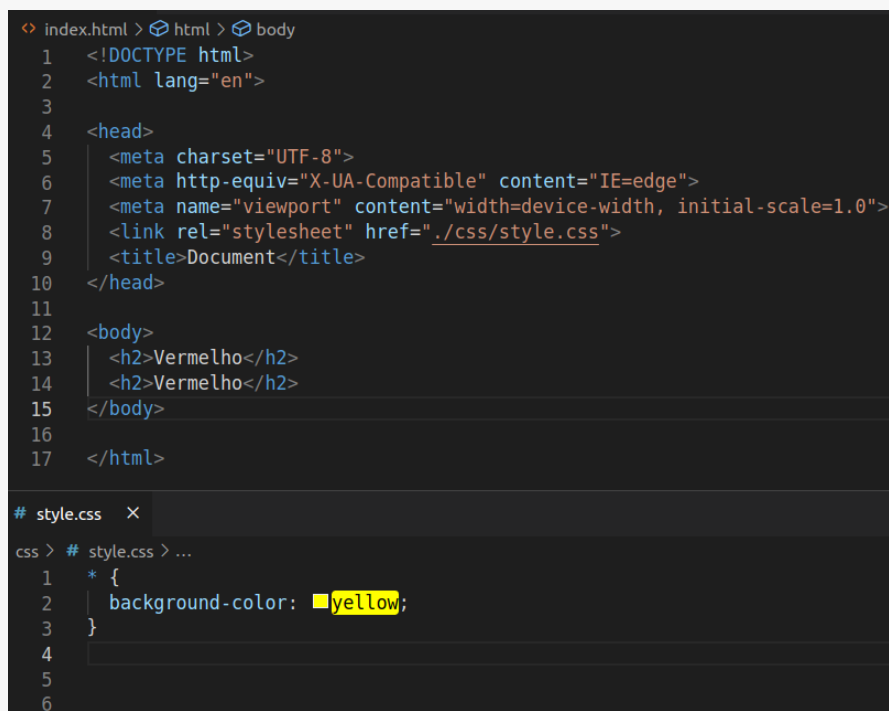


Nesse exemplo foram selecionados alguns elementos da mesma tag para colocar a cor de fundo como vermelha. Se isso não tivesse acontecido, todos os elementos com a tag `<h2>` teriam a cor de fundo vermelha.

- Seletor universal:

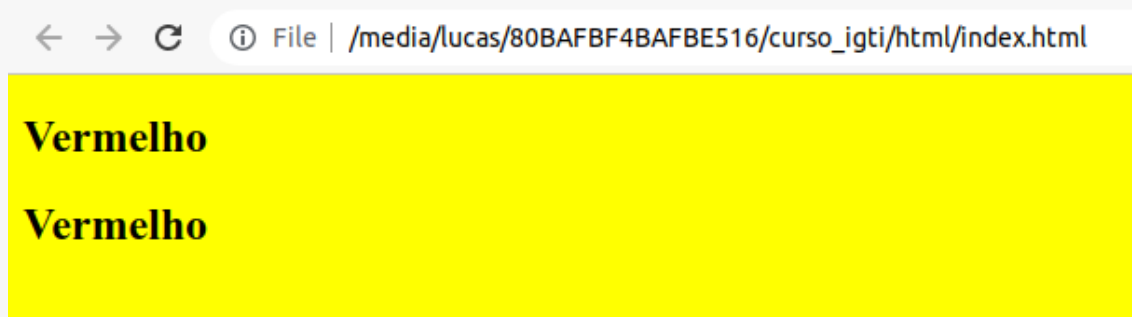
Neste seletor a estilização será universal no documento como o próprio nome sugere. O modo de uso é com o símbolo asterisco (\*), como podemos ver no exemplo da Figura 34.

Figura 34 - Exemplo de uso do seletor universal.



O resultado do uso desse seletor é mostrado na Figura 35.

Figura 35 - Exemplo de aplicação do seletor de universal.



Esse exemplo ilustra a forma de aplicar o valor de uma propriedade no documento todo.



**XP**e

## > Capítulo 4



## Capítulo 4. Eventos em HTML

---

Neste capítulo trataremos de eventos no documento HTML. Para capturar comportamentos no documento HTML, a linguagem javascript é utilizada, do mesmo modo que fizemos no css, criando um arquivo externo para guardar o código javascript que irá capturar os eventos no documento HTML.

O que são eventos?

Eventos no documento HTML são um conjunto de ações que são realizadas em um determinado elemento do documento HTML. Muitas interações podem acontecer quando o usuário visita sua página *web*. Existe uma infinidade de eventos que podem ser utilizados no javascript, na tabela abaixo é possível ver os principais.

onBlur	Remover o foco do elemento
OnChange	Muda o valor do elemento
onClick	O elemento é clicado pelo usuário
onFocus	O elemento é focado
onKeyPress	O usuário pressiona uma tecla sobre o elemento
onLoad	Carrega o elemento por completo
onMouseOver	Define ação quando usuário passa o mouse sobre o elemento
onMouseOut	Define ação quando usuário retira o mouse sobre o elemento
onSubmit	Define ação ao enviar um formulário

A sintaxe de eventos é mostrada na Figura 36.

Figura 36 - Exemplo de sintaxe de evento.

`<p id = "p1" onClick="change()"> Hello World!</p>`

Diagram illustrating the components of the event syntax:

- Tag: `<p`
- id do elemento: `id = "p1"`
- Evento clicar: `onClick`
- Função que será chamada: `= "change()"`
- Conteúdo do elemento: `> Hello World!</p>`

No exemplo mostrado na Figura 36 foi utilizado o evento *onClick*. Quando o usuário clicar no elemento alguma ação vai ser tomada. No código da Figura 37 o conteúdo do elemento será mudado.

Figura 37 - Exemplo de uso do evento *onClick*.

```
<> index.html > html > head
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta http-equiv="X-UA-Compatible" content="IE=edge">
7    <meta name="viewport" content="width=device-width, initial-scale=1.0">
8    <script type="text/javascript" src="./js/code.js"></script>
9    <title>Document</title>
10 </head>
11
12 <body>
13   <p id="p1" onclick="change()">Hello world!</p>
14 </body>
15
16 </html>

JS code.js  X
js > JS code.js > change
1  function change(){
2    document.getElementById("p1").innerHTML = "Conteúdo trocado!";
3  }
```

O resultado do uso desse seletor é mostrado na Figura 38.



Figura 38 - Exemplo de aplicação do evento *onClick*.



Nesse exemplo deve atentar para alguns detalhes. Primeiramente, foi utilizada a tag *script* no `<head>` do documento (linha 8). O motivo de estruturar o exemplo desse modo é o mesmo de quando foram construídos os exemplos de ccs (organização!). O próximo detalhe se refere ao modo que o elemento foi capturado, com `document.getElementById()`. Esse é o seletor de elemento pelo identificador dele dentro do *javascript* (Existem outros seletores!). O último detalhe é a propriedade *innerHTML*. Essa propriedade modifica o conteúdo do elemento (existem outras propriedades!).

Como dito anteriormente, existem diferentes modos de selecionar elementos no *javascript*. Os principais modos de selecionar elementos são mostrados na tabela abaixo.

<code>document.getElementById()</code>	Selecionar por identificador do elemento
<code>document.getElementsByName()</code>	Selecionar pela propriedade name do elemento
<code>document.getElementsByClassName()</code>	Selecionar pela propriedade class do elemento
<code>document.querySelectorAll()</code>	Selecionar pelo seletor css

Uma das características ressaltada anteriormente foi qual atributo do elemento se está procurando mudar. O leitor mais atento deve ficar curioso sobre as propriedades que podem ser mudadas. Pense no seguinte exemplo com o elemento de imagem: ``. A propriedade evidente no elemento é a fonte (*src*). Pode-se mudar a imagem do elemento da seguinte forma:

```
document.getElementById("myImage").src = "landscape.jpg";
```

Finalmente, este parágrafo foi necessário para expor ao leitor que os atributos inerentes ao elemento podem ser modificados dinamicamente usando os eventos.



**XP**e

# > Capítulo 5



## Capítulo 5. Django

---

Nesta seção será apresentado o *framework* Django. O Django é uma ferramenta completa e muito utilizada. Empresas como Spotify, Instagram e YouTube, por exemplo, têm seu *backend* construído com Django.

Antes de mais nada, é necessário dizer que é possível construir o *backend* de uma aplicação sem utilizar *framework*. A pergunta mais natural a se fazer depois dessa afirmação é: Por que usar um *framework* e estudá-lo? A resposta para essa pergunta é feita utilizando uma analogia. Imagine que se queira fazer uma feijoada. O primeiro passo para executar esse prato tão brasileiro é comprar os ingredientes e na sequência realizar o cozimento. Apesar do processo como um todo ter sido muito simplificado, é basicamente assim que acontece. Agora pense que antes mesmo de comprar os ingredientes tivéssemos que construir o fogão e as panelas em que será realizado o cozimento. O processo seria bem mais complexo. Por este motivo que se utiliza de ferramentas pré-constituídas (fogão, panelas e utensílios) para agilizar todo o processo. Nesse momento aplica-se a analogia ao caso de serviços *web* e *apis*. Os frameworks são os utensílios que agilizam a construção de serviços *web* e *apis*. Nesta seção o utensílio para construção de web services será o Django.

### 5.1. Desenvolvimento Web e Fluxos

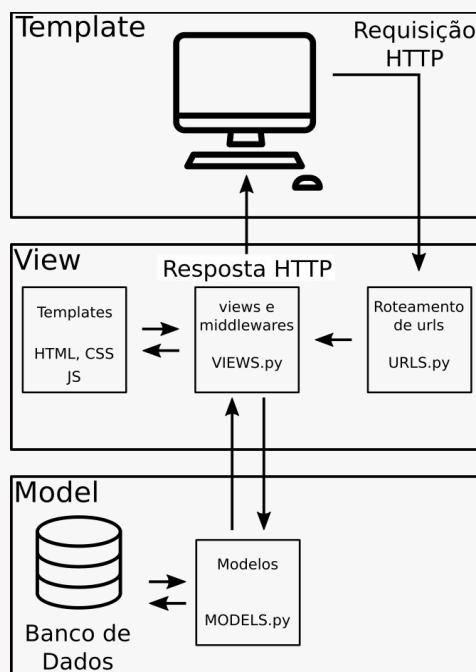
Antes de se falar no funcionamento do *framework* é necessário entender o desenvolvimento *web* em si. O objetivo deste modo de desenvolvimento de software é criar aplicações/sites que serão executados em um *browser*. Dentro deste contexto é necessário salientar que existem duas áreas principais: *backend* e *frontend*. O *frontend* é uma “camada” do *software* que o usuário vê e interage. São principalmente as telas que o usuário pode interagir com o sistemas e suas funcionalidades. O *backend*, que também é conhecido como servidor, tem a responsabilidade de

processar as requisições feitas pelo *frontend*. Além disso, o backend contém toda a regra de negócio e gerencia a comunicação com o banco de dados. É nesta camada que o Python está presente.

O Django, por sua vez, é responsável por tratar as requisições que o sistema irá receber, o mapeamento objeto-relacional e a preparação das respostas (HTTP) que o sistema fará. Desta forma o trabalho do desenvolvedor fica facilitado e assim deve se preocupar apenas com as regras de negócio. O Django trabalha num sistema de camadas comumente conhecido como MVT - isto é: *Model-View-Template*. Estas são as camadas em que o sistema Django será organizado.

O Fluxo de uma requisição acontece como mostrado na Figura 39. Para resumir, uma requisição HTTP é feita no frontend (Tela do computador). Essa requisição vai para a camada de *View*. Primeiramente chega no arquivo de roteamento (`urls.py`) e depois disso é encaminhada para o arquivo `views.py`. Este arquivo que contém as funções de view faz comunicação com as camadas de *Model* (que faz a comunicação com o banco de dados) e *Template*. De modo geral, as funções de view (uma vez que receberam uma requisição) trazem os dados da camada de *Model* e os apresentam usando um arquivo HTML que está na camadas de templates. Para que não reste nenhuma dúvida essa explicação é denotada na Figura 39.

Figura 39 -Fluxo de uma requisição.

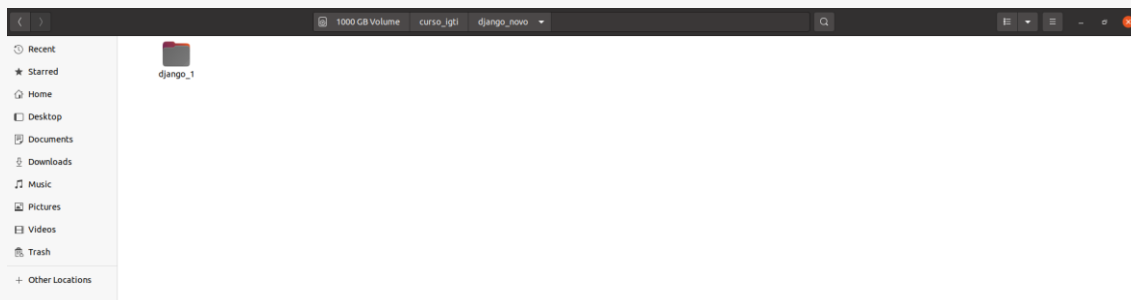


## 5.2. Instalação

Nesta seção será apresentada a forma como pode ser realizada a instalação do *framework* Django. Para prosseguir com este objetivo, primeiramente será criado um ambiente virtual. Esse ambiente virtual (tanto em Linux quanto em Windows) irá guardar as dependências de qualquer projeto de python que serão criados aqui neste curso. Num segundo momento o *framework* Django será instalado no ambiente virtual anteriormente citado. Uma vez que a instalação foi realizada, será feita uma conferência da instalação e da versão do *framework*. No próximo passo será realizado o backup das bibliotecas usadas no ambiente virtual. Finalmente o comando para a inicialização de um projeto Django será executado para criar um primeiro projeto Django.

O editor utilizado será o *visual studio code*. Primeiramente o leitor deve criar uma pasta para hospedar o projeto que será criado (será chamado aqui de “django\_1”).

Figura 40 - Pasta do primeiro projeto teste.



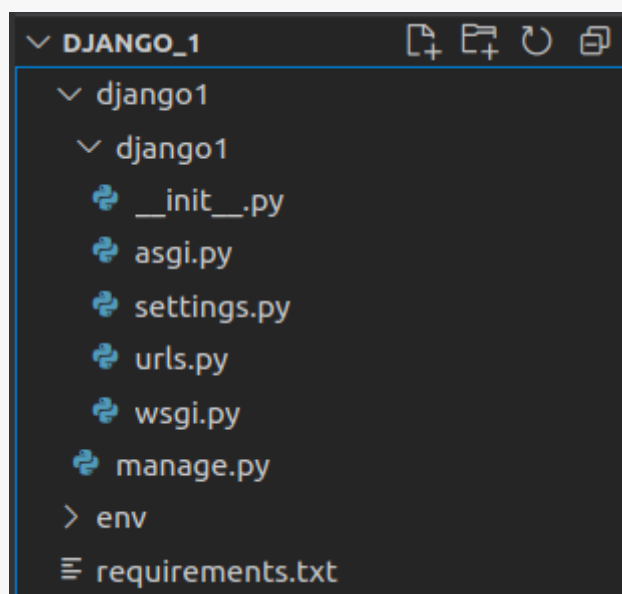
Abra o *visual studio code*. Abra o editor na pasta do projeto (ctrl + k ctrl + o).

Com o editor aberto na pasta, abra um terminal (Terminal -> New Terminal). Agora deve-se seguir os seguintes passos:

1. Verifique se você possui o python 3 instalado. No terminal do editor digite “python3 --version”. Caso não possua a versão do python 3 (3.8 e superior), instale o python 3 em sua máquina.
2. Crie um ambiente virtual. Este modo de trabalho vai tornar o seu projeto mais organizado e portátil. No terminal digite: “python -m venv ./env”. Pode-se perceber que uma nova pasta irá aparecer dentro do diretório do projeto (env). Digite “source ./env/bin/activate” para ativar o ambiente virtual. Se você digitar “python --version” verá que a versão python nativa é a versão 3.x.x.
3. Instale o Django em seu ambiente virtual. Digite o comando “pip install django”. Verifique a versão instalada digitando o comando “python -m django --version”.
4. Crie um arquivo de dependências. No terminal digite: “pip freeze > requirements.txt”. Esse arquivo vai conter as dependências do projeto toda vez que uma dependência for adicionada. Esse comando vai ser utilizado para guardar essa dependência e sua versão no arquivo requirements.txt.

Cumprindo esses pequenos passos é possível criar a primeira aplicação em Django. No terminal digite: “django-admin startproject django1”. A estrutura criada é mostrada na Figura 41.

Figura 41 -Estrutura do primeiro projeto django.



Estes arquivos são:

1. django1. Diretório raiz. Esse diretório é um container para seu projeto. O nome do diretório não importa para o Django e você pode até mudá-lo se quiser.
2. manage.py. Arquivo de linha de comando que permite o desenvolvedor interagir com o projeto de várias maneiras.
3. django1/django1. Diretório interno. Esse é o diretório real do seu projeto, nele será desenvolvido o seu projeto.
4. django1/\_\_init\_\_.py. Arquivo vazio que informa ao python que esse diretório deve ser considerado um pacote python.
5. django1/settings.py. Esse arquivo contém as configurações de seu projeto Django.



6. `django1/urls.py`. Esse arquivo contém as declarações das URLs de seu projeto django. As URLs são endereços que serão acessados no projeto.
7. `django1/asgi.py`. Um ponto de entrada para servidores web compatíveis com ASGI para servir seu projeto.
8. `django1/wsgi.py`. Um ponto de entrada para servidores web compatíveis com WSGI para servir seu projeto.

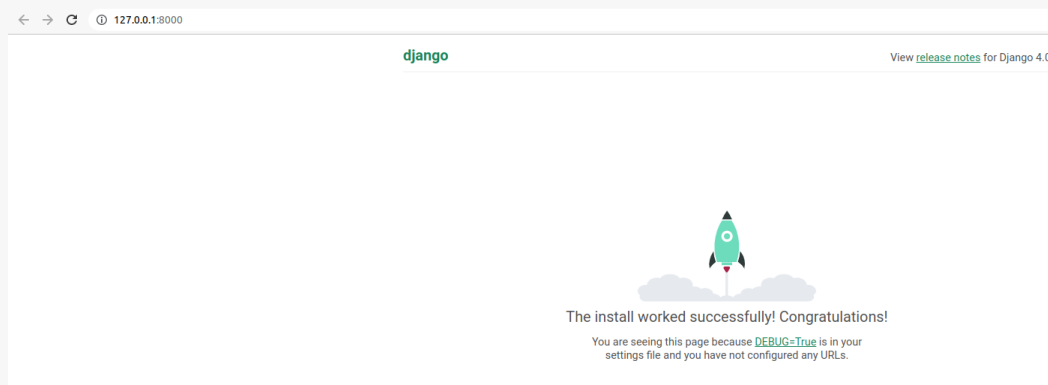
Agora é possível testar o projeto. Estando no diretório raiz `django1` (Note que nesse diretório está contido o arquivo `manage.py`), digite “`python manage.py runserver`”. Na Figura 42 é mostrado o console da aplicação.

Figura 42 - Console da aplicação `django1`.

```
(env) lucas@RI0-D6KR9R2:/media/lucas/80BAFBF4BAFBE516/django_1/django1$  
(env) lucas@RI0-D6KR9R2:/media/lucas/80BAFBF4BAFBE516/django_1/django1$ python manage.py runserver  
Watching for file changes with StatReloader  
Performing system checks...  
  
System check identified no issues (0 silenced).  
  
You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.  
Run 'python manage.py migrate' to apply them.  
February 06, 2022 - 22:01:08  
Django version 4.0.2, using settings 'django1.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

O endereço da aplicação está mostrado no console em <http://127.0.0.1:8000/>. Colando esse endereço em um browser (aqui está sendo utilizado o Google Chrome) é possível ver a seguinte imagem mostrada na Figura 43.

Figura 43 - Imagem da primeira aplicação (`django1`) no browser.

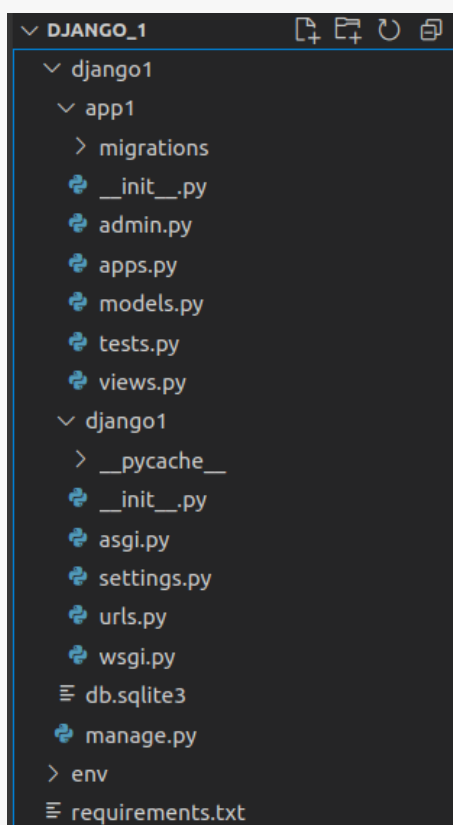


Neste momento deve-se atentar para um detalhe. A porta em que o servidor da aplicação é a 8000. Pode-se trocar a porta utilizada simplesmente executando o comando “python manage.py runserver 3001”.

### 5.3. Criando o primeiro aplicativo

A estrutura comum de um projeto Django consiste no projeto em si e seus aplicativos. Cada projeto pode conter vários aplicativos. Qual é a diferença entre o projeto e o aplicativo? A diferença é que projeto é um conjunto de configurações e aplicativos para um determinado site, o aplicativo, por sua vez, é uma estrutura que possui uma função específica (blog, aplicativo de pesquisa, uma agenda etc.). O primeiro aplicativo será criado na pasta que contém o arquivo manage.py (o aplicativo poderia ser construído em outro lugar). O comando de criação do aplicativo é “python manage.py startapp app1”. O resultado da estrutura do projeto ficará como mostrado na Figura 44.

Figura 44 - Estrutura do projeto após a criação do primeiro aplicativo.



A nova estrutura de pastas se refere a aplicação (app1). Nessa nova estrutura tem-se os seguintes arquivos:

1. app1/migrations. Diretório onde ficam guardadas as modificações que serão aplicadas no banco de dados.
2. app1/\_\_init\_\_.py. Arquivo vazio que informa ao python que esse diretório deve ser considerado um pacote python.
3. app1/admin.py. Arquivo os *models* (espelho da estrutura de dados) do projeto.
4. app1/apps.py. Arquivo de configuração do aplicativo.
5. app1/models.py. Arquivo onde a estrutura de dados deve ser escrita.
6. app1/tests.py. Arquivo de testes do aplicativo.
7. app1/views.py. Arquivo que contém as views do aplicativo.

#### 5.4. Criando as views

Nesta seção será mostrado como criar a primeira *view* do aplicativo app1. Abra o arquivo “app1/views.py” e coloque o seguinte código:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world!")
```

O próximo procedimento é criar uma configuração de url para o aplicativo app1. Crie um arquivo na pasta app1 e chame o de urls.py. Coloque nesse arquivo o seguinte código:

```
from django.urls import path

from . import views
```

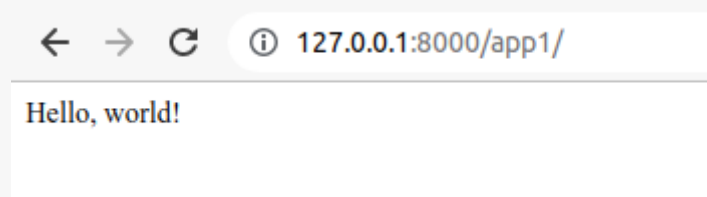
```
urlpatterns = [  
    path('', views.index, name='index'),  
]
```

Basicamente os trechos de códigos mostrados fazem: o primeiro cria a primeira função (def index) de *view* do aplicativo app1. Esta função deve ser usada para dar uma resposta a uma requisição recebida e retorna uma resposta (`return HttpResponse("Hello, world!")`). O segundo cria uma configuração que invoca a função index que está na *view*. O próximo passo é configurar/adicionar o arquivo app1/urls.py no arquivo django1/urls.py como mostrado no código mostrado abaixo:

```
from django.contrib import admin  
from django.urls import include, path  
  
urlpatterns = [  
    path('app1/', include('app1.urls')),  
    path('admin/', admin.site.urls),  
]
```

Para testar se é possível acessar a primeira *view*, acesse o endereço em seu browser “<http://127.0.0.1:8000/app1/>”. A página deve exibir a mensagem mostrada na Figura 45.

Figura 45 - Primeira *view* do aplicativo app1.



Neste ponto é possível criar views para o primeiro aplicativo do projeto django1. Antes de prosseguir a leitura verifique se é possível repetir os passos mostrados anteriormente. O próximo passo será mostrar como criar modelos de estrutura de dados e aplicá-los em um banco de dados.

## 5.5. Models

Nesta seção será criada a primeira estrutura de *models*. Para esse objetivo é necessário primeiramente pensar que modelo de dados será criado. Numa primeira abordagem será utilizado uma estrutura bem simples. O modelo de dados deve conter apenas uma tabela. Essa tabela conterá os campos: nome, e-mail e telefone. O motivo de se utilizar um modelo simples é conduzir o leitor a saber configurar a estrutura de *models* que em outros aplicativos será realizada pelo leitor de modo confortável.

No Arquivo `app1/models` insira o código mostrado abaixo:

```
from django.db import models

class User(models.Model):
    nome = models.CharField('nome', max_length=20)
    telefone = models.CharField('telefone', max_length=15)
    email = models.CharField('email', max_length=30)

    def __str__(self):
        return f'Usuário: {self.nome}, Telefone: {self.telefone}, Email: {self.email}'
```

O código mostrado acima possui o *model* de usuário que é descrito pela classe *User*. Os campos nome, telefone e e-mail estão inseridos nessa classe e possuem os tipos de dados recebidos pelos atributos do argumento. Este é um caso bem particular em que todos os campos são do mesmo tipo, em aplicações mais complexas isso não acontecerá.

Existem os mais variados tipos de dados que podem compor um *model* (inteiros, decimais, booleanos, data, e-mail) a referência desses campos pode ser encontrada no site do Django (<https://docs.djangoproject.com/en/4.0/ref/models/fields/>).

### 5.5.1. Primeira migração

Antes mesmo de configurar as *models* é necessário criar o *superadmin*. Primeiramente deve ser criadas as tabelas. Execute o seguinte comando: `python manage.py migrate`.

Figura 46 - O comando migrate.

```
(env) lucas@RIO-D6KR9R2:/media/lucas/80BAFBF4BAFBE516/django_1/django1$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
(env) lucas@RIO-D6KR9R2:/media/lucas/80BAFBF4BAFBE516/django_1/django1$
```

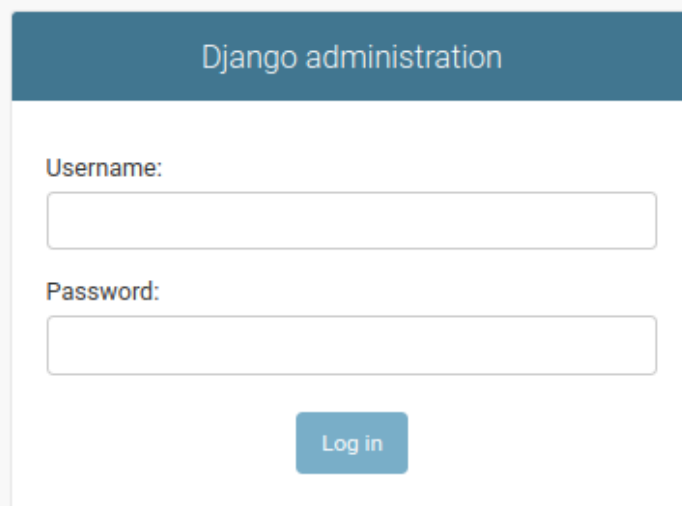
As tabelas padrão foram criadas juntamente com a tabela do *model User*, construído na seção passada. Para criar o usuário *superadmin*, execute o seguinte comando: `python manage.py createsuperuser`. Esse comando pedirá um nome de usuário, e-mail e senha. Um resultado similar ao mostrado na Figura 47 será mostrado.

Figura 47 - O comando createsuperuser.

```
(env) lucas@RIO-D6KR9R2:/media/lucas/80BAFBF4BAFBE516/django_1/django1$ python manage.py createsuperuser
Username (leave blank to use 'lucas'): lucas
Email address: lucaskrispin@hotmail.com
Password:
Password (again):
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
(env) lucas@RIO-D6KR9R2:/media/lucas/80BAFBF4BAFBE516/django_1/django1$
```

Com o *superadmin* criado é possível acessar a sessão administrativa do projeto. Basta levantar o projeto com o comando: `python manage.py runserver`. Após o comando ser executado, acesse a url <http://127.0.0.1:8000/admin>. Pode-se ver uma tela semelhante à mostrada na Figura 48. Basta inserir o login e a senha criada nos passos anteriores para adentrar a área administrativa.

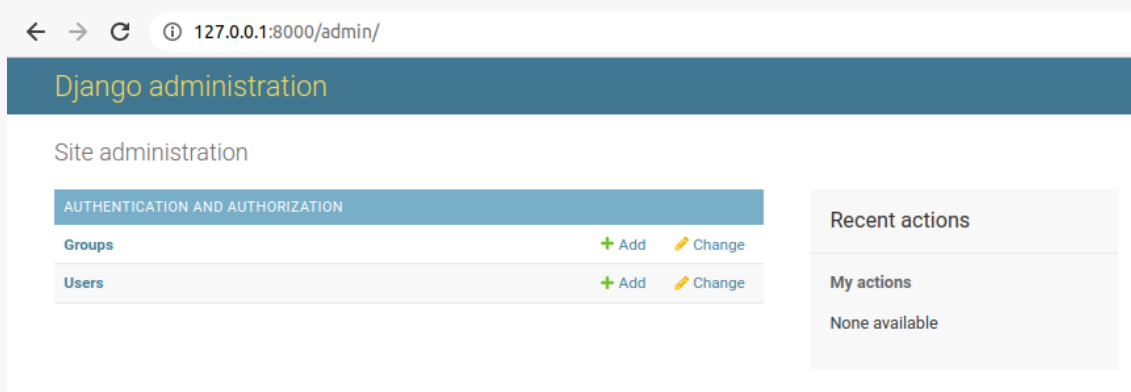
Figura 48 - Área de login da seção administrativa.



The image shows the Django administration login interface. It features a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". A blue "Log in" button is positioned below the password field.

Após realizar o login, a área administrativa é mostrada (Figura 49).

Figura 49 - Área administrativa do Django.



The image shows the Django administration site after a successful login. The browser address bar displays "127.0.0.1:8000/admin/". The site has a dark blue header with "Django administration" in yellow. The main content area is titled "Site administration" and contains a table with the following structure:

AUTHENTICATION AND AUTHORIZATION	
Groups	<a href="#">+ Add</a> <a href="#">Change</a>
Users	<a href="#">+ Add</a> <a href="#">Change</a>

To the right of the table, there is a "Recent actions" section with a "My actions" link and the text "None available".

Na área administrativa é possível cadastrar novos usuários para o sistema, além de manipular informações dos *models* criados anteriormente.

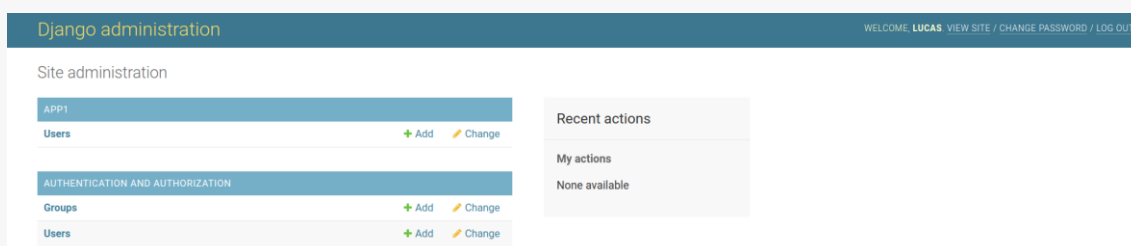
Para usar a área administrativa como *frontend*. Basta colocar o model de usuário no arquivo `admin.py` para criar funcionalidades de usuário

(criar, alterar, exibir, deletar). Para fazer isso insira no arquivo de admin.py que está na pasta do app1 o seguinte código: `from .models import User` e `admin.site.register(User)`. O arquivo admin ficará assim:

```
django.contrib import admin
from .models import User
from
# Register your models here.
admin.site.register(User)
```

Agora basta logar na área administrativa e a seguinte tela aparecerá (Figura 50).

Figura 50 - Área administrativa do django.



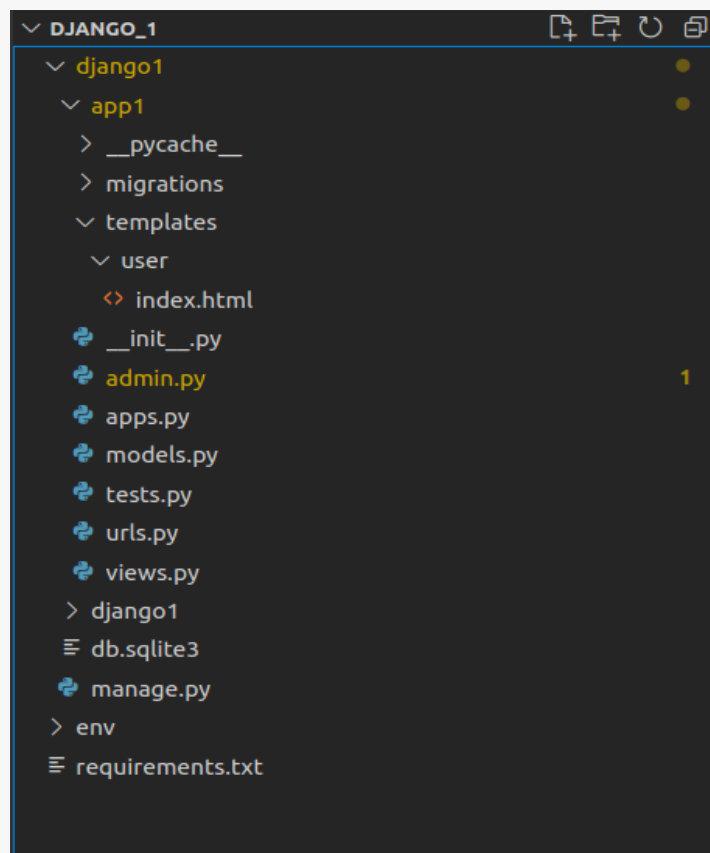
É possível ver que existe a aba APP1. Se o usuário clicar em Users dentro da aba APP1 é direcionado para uma tela em que o usuário vai poder manipular dados de usuário (exibir, criar, atualizar e deletar).

## 5.6. Templates

Nesta seção a parte dos templates será apresentada. Para começar, uma pasta deve ser criada dentro do app (app1) com nome de preferência, porém, será criada aqui com o nome sugestivo de 'templates'. Dentro da pasta templates será criada outra pasta que irá se referir ao *model* de usuário (Figura 51). Dentro desta pasta é criado também um arquivo HTML simples (index.html Figura 51) para ser utilizado neste exemplo.



Figura 51 - Estrutura de pastas após a criação da pasta template.



Além da criação das pastas e do arquivo html, há uma configuração no arquivo `django1/django1/settings.py`. Na chave `DIRS` dentro da variável `TEMPLATES` o nome da pasta de templates deve ser adicionado. Na Figura 52 essa configuração é mostrada.

Figura 52 - Configuração da pasta templates no arquivo settings.py.

```

55 TEMPLATES = [
56     {
57         'BACKEND': 'django.template.backends.django.DjangoTemplates',
58         'DIRS': ['templates'],
59         'APP_DIRS': True,
60         'OPTIONS': {
61             'context_processors': [
62                 'django.template.context_processors.debug',
63                 'django.template.context_processors.request',
64                 'django.contrib.auth.context_processors.auth',
65                 'django.contrib.messages.context_processors.messages',
66             ],
67         },
68     },
69 ]

```

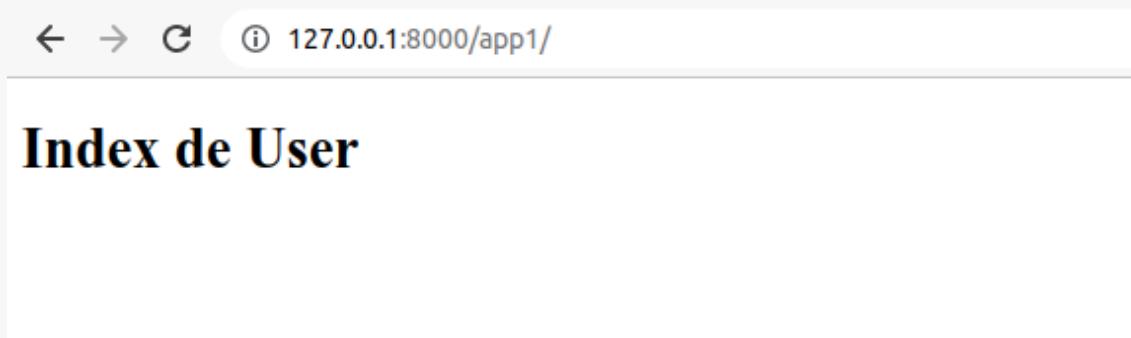
Agora no arquivo de *views* o retorno da função será a renderização do arquivo html (*index.html*). O código da *view* é mostrado abaixo:

```
from django.shortcuts import render

def index(request):
    return render(request, 'user/index.html')
```

A renderização desse view será realizada através dele e o que será mostrado no browser é o conteúdo do arquivo html. O resultado é mostrado na Figura 53.

Figura 53 - Resultado da renderização da view.





**XP**e

# > Capítulo 6



## Capítulo 6. Flask

---

Nesta seção será apresentado o *framework* Flask. O Flask é comumente conhecido por ser um *framework* pequeno (*microframework*). O objetivo do Flask é justamente contrário ao do Django, por exemplo. O Django possui uma gama de ferramentas nativas, e já na criação de um projeto é gerado uma estrutura de pastas que obedece a um padrão (MVT). O Flask, por outro lado, é pequeno, possui poucas funcionalidades nativas e as extensões são adicionadas de acordo com necessidade do desenvolvedor. O leitor deve estar se perguntando nesse momento qual framework é “melhor”. Essa pergunta é respondida da seguinte forma: Não existe uma bala de prata ou solução perfeita para qualquer problema. O que se quer dizer com essa frase é que para alguns tipos de sistemas o Django será mais apropriado, em outros o Flask será mais apropriado.

### 6.1. Instalação

Inicialmente deve ser criada uma pasta onde será desenvolvido um projeto com Flask. Neste caso será criada a pasta Flask. No próximo passo crie um ambiente virtual com python 3 (esse procedimento é mostrado na seção 5). Ative o ambiente virtual (source ./env/bin/activate) e instale o Flask com o comando: `pip install flask`. Guarde as dependências do projeto no arquivo `requirements.txt` com o comando: `pip freeze > requirements.txt`.

Cumprido esses passos, crie na raiz do projeto um arquivo. Neste caso o arquivo será chamado de `app.py`. Agora será criado o primeiro aplicativo web com flask. No arquivo `app.py`, digite o seguinte código python:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
```

```
return '<h1> Hello World! </h1>'

if __name__ == "__main__":
    app.run(debug=True, port=5000)
```

Agora digite no terminal: `python app.py`. Acesse a url <http://127.0.0.1:5000/> em seu browser e será vista a tela mostrada na Figura 50. O leitor deve se atentar ao detalhe na linha do comando `app.run(debug=True, port=5000)`. Nessa linha está configurado o debug da aplicação como ativo e a porta que vai ser usada pela aplicação.

Figura 54 - Primeira aplicação com Flask.

← → ↻ ⓘ 127.0.0.1:5000

# Hello World!

Parabéns, o primeiro aplicativo web feito com Flask está de pé. A primeira linha de código importa o Flask. A segunda linha invoca o Flask e guarda na variável `app`. A linha de código `@app.route('/')` é um decorator que indica o caminho da rota. Esta rota executa a função `index` que retorna um texto com um código HTML.

Pode-se também criar rotas dinâmicas. Veja o código abaixo:

```
app = Flask(__name__)

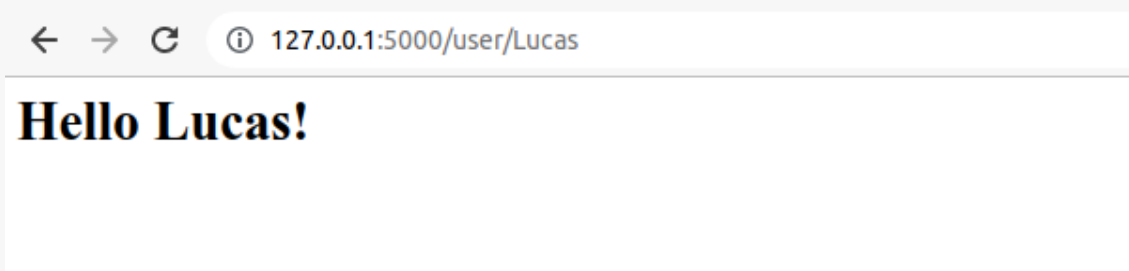
@app.route('/')
def index():
    return '<h1> Hello World! </h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1> Hello {name}! </h1>'
```

```
if __name__ == "__main__":  
    app.run(debug=True, port=5000)
```

Acesse a url <http://127.0.0.1:5000/user/Lucas>. Você verá na tela a mensagem mostrada na Figura 55.

Figura 55 - Rotas dinâmicas com Flask.



Você pode perceber isso trocando o nome que vem na url pelo seu nome, por exemplo. Pode-se observar que o nome exibido é dinâmico, depende do nome enviado na url.

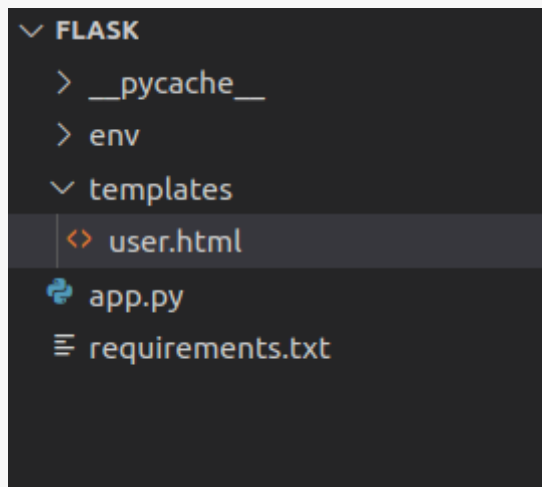
<https://jinja.palletsprojects.com/en/3.0.x/>.

## 6.2. Templates

Similar ao Django, o Flask também possui um sistema de templates. Este sistema de templates renderiza documentos html contidos num repositório específico.

Da mesma forma que foi feito com o Django é necessário com o flask mostrar a forma com que é possível renderizar documentos html. Por padrão, o Flask procura os templates em um subdiretório “templates” no diretório principal da aplicação. Na Figura 56 é mostrada a nova estrutura de pastas do projeto.

Figura 56 - Estrutura do projeto com os templates.



O código que será utilizado para renderizar o template (user.html) é mostrado abaixo:

```
from flask import Flask, render_template
app = Flask(__name__)

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

if __name__ == "__main__":
    app.run(debug=True, port=5000)
```

Pode-se perceber que foi importada a função `render_template` que é utilizada para renderizar o arquivo html. O resultado é mostrado na Figura 57.

Figura 57 - Resultado da renderização do template.



O usuário mais atento deve estar se perguntando “como renderizar um template de modo dinâmico?”. A resposta é sim! É possível. Veja o código abaixo:

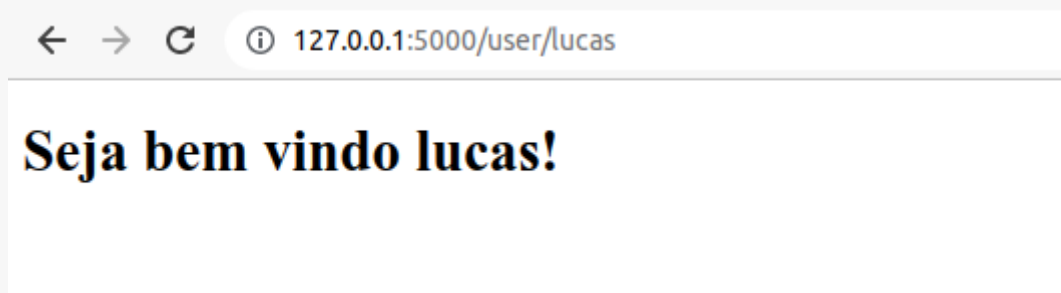
```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/user/<name>')
def index(name):
    return render_template('user.html', name=name)

if __name__ == "__main__":
    app.run(debug=True, port=5000)
```

O resultado é mostrado na Figura 58.

Figura 58 - Resultado da renderização do template.



O html que gerou esse resultado é mostrado abaixo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Index de User</title>
</head>
<body>
  {% if name %}
  <h1>Seja bem vindo {{name}}!</h1>
  {% else %}
  <h1>Não temos um nome!</h1>
  {% endif %}
</body>
```



```
</html>
```

Repare que existe um tipo de algoritmo embutido no html. Esse código que pode ser inserido no documento html é provido pela *engine* de templates Jinja2. Assim como o desvio de fluxo, as estruturas comuns de algoritmos podem ser utilizadas. Veja mais referências em <https://jinja.palletsprojects.com/en/3.0.x/>.

### 6.3. Configuração

Para aumentar a aplicação Flask é necessário começar a separar partes do código para organizar melhor o projeto. A primeira separação/criação é do arquivo de configurações (config.py) que deve estar na mesma pasta do arquivo app.py. O código desse arquivo será mostrado abaixo.

```
import os

class Config(object):
    basedir = os.path.abspath(os.path.dirname(__file__))

class DevelopmentConfig(Config):
    DEVELOPMENT = True
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + os.path.join(Config.basedir,
'data.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    IP_HOST = 'localhost'
    PORT_HOST = 3000
```

É necessário antes de qualquer novo passo entender o que está neste arquivo. Neste arquivo estão contidas duas classes. A primeira Config contém a configuração básica da aplicação. Ela contém o atributo `basedir`. Este atributo indica o diretório base do projeto. A segunda classe (`DevelopmentConfig`) contém os atributos do ambiente de desenvolvimento do projeto. Existe nesta classe uma configuração especial. A configuração

`SQLALCHEMY_DATABASE_URI` mostra onde será criado o database e o nome que ele terá (“data.db”).

Neste momento uma pequena modificação será realizada no arquivo `app.py` para incluir as configurações do arquivo `config.py`. O código será mostrado abaixo.

```
from flask import Flask, render_template
from config import DevelopmentConfig
app = Flask(__name__)
app.config.from_object(DevelopmentConfig())

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

if __name__ == "__main__":
    app.run(debug=DevelopmentConfig.DEBUG, port=DevelopmentConfig.PORT_HOST)
```

## 6.4. Models

Neste momento é possível escrever a primeira *model* da aplicação em *Flask*. Crie uma pasta `model` e dentro desta pasta crie o arquivo `user.py` que vai conter o model de usuário. Esta pasta deve estar no mesmo nível do arquivo `app.py`. Veja abaixo o código que vai ser inserido no arquivo.

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
class User(db.Model):

    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    nome = db.Column(db.String(20))
    telefone = db.Column(db.String(15))
    email = db.Column(db.String(30))

    def serialize(self):
        return {'id': self.id, 'Nome': self.nome, 'Telefone':
```

```
self.telefone, 'Email': self.email}
```

Nesta *model* criada só possuímos campos string assim como o projeto Django. Mas existem vários tipos de campos (ver em [https://docs.sqlalchemy.org/en/14/core/type\\_basics.html](https://docs.sqlalchemy.org/en/14/core/type_basics.html)).

Agora é necessário instalar o flask migrate (pip install Flask-migrate) que será responsável pelas migrações da aplicação. Em seguida, o comando de guardar as dependências do projeto deve ser executado (pip freeze > requirements.txt). Neste momento é necessário modificar o arquivo app.py para poder executar as migrações (versionamento de banco de dados) e conexão com o banco de dados. O código do arquivo app.py:

```
from flask import Flask, render_template
from flask_migrate import Migrate
from config import DevelopmentConfig
from models.user import db

app = Flask(__name__)
app.config.from_object(DevelopmentConfig())
db.init_app(app)
migrate = Migrate(app, db)

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

if __name__ == "__main__":
    app.run(debug=DevelopmentConfig.DEBUG, port=DevelopmentConfig.PORT_HOST)
```

É possível observar que foram adicionadas as linhas: `from flask_migrate import Migrate` que importa o módulo de migrações, `from models.User import db` o *model* de usuário e `db.init_app(app)` que inicializa o banco e `migrate = Migrate(app, db)` que disponibiliza o serviço de migração. Agora três comandos são necessários: flask db init (para inicializar o banco), flask db migrate -m "primeira migração." (comando de

criação de migração) e flask db upgrade (comando que aplica a migração no banco de dados).

O leitor deve estar atento ao detalhe que o comando “flask db init” só será executado apenas na primeira vez da inicialização do banco. Toda vez que os *models* forem modificados, os comandos flask db migrate e flask db upgrade devem ser executados nessa ordem para aplicar as modificações no banco de dados.

## 6.5. Roteamento

Neste momento já é possível conectar o banco de dados. Lembre-se que se trata do banco de dados *sqlite* que está na mesma pasta do arquivo app.py e tem o nome data.db (lembre-se que foi configurado assim no arquivo config.py). Portanto, é possível construir as rotas que vão fazer as operações básicas nos dados de usuário (exibir, criar, modificar e deletar).

### 6.5.1 Rota para buscar todos os dados

A primeira rota que será construída é a rota para buscar todos os dados. O código usado para construir a rota deve ser incorporado ao arquivo app.py e é mostrado logo abaixo.

```
@app.route('/user/getall', methods=['GET'])
def getAll():
    session = db.session()
    users = session.query(User).all()
    users_json = [user.serialize() for user in users]
    session.close()
    return Response(json.dumps(users_json))
```

É necessário entender cada uma das linhas da rota para buscar todos os usuários. A primeira linha (`@app.route('/user/getall', methods=['GET'])`) se refere ao caminho que deve ser acessado para buscar os dados e seguido dessa string vem a definição do método que será usado nessa rota (neste caso método HTTP GET). Após isso, vem a função que será acessada quando uma requisição de rota for feita. A função tem por nome

“getall”. Na próxima linha uma sessão com o banco de dados deve ser aberta (`session = db.session()`). Na linha seguinte a “query” ao banco de dados é feita buscando todos os usuários (`users = session.query(User).all()`). A próxima linha o resultado trazido da “query” é serializado usando a função que está contida dentro do *model* de usuário (`users_json = [User.serialize() for user in users]`). Na próxima linha a sessão com o banco de dados é fechada (). Na última linha o resultado é retornado da forma de uma lista de objetos json (json é um formato de objeto usado para transitar dados nas requisições HTTP).

No início do arquivo `app.py` devem ser acrescentadas as seguintes linhas.

```
from models.user import db, User
from flask import Response, request
import json
```

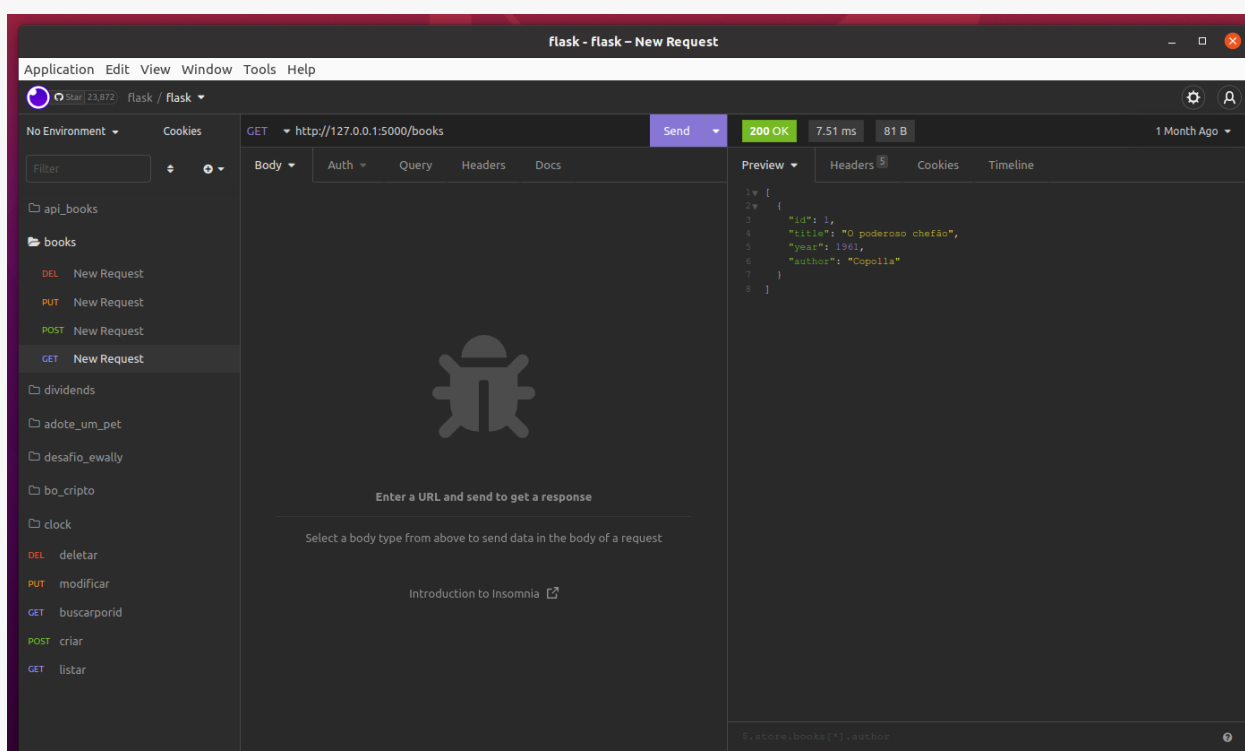
As últimas linhas que devem ser acrescentadas ao arquivo `app.py` são: a importação do *model* de usuário (`from models.user import db, User`), a importação de função de envio e recebimento de requisições e dados (`from flask import Response, request`) e a importação da função json para converter o formato da resposta no padrão json (`import json`).

Estamos prontos para testar nossa primeira rota (`"/user/getall"`), porém, o leitor mais atento deve estar se perguntando por que não foi escrito um arquivo HTML em que nossos dados pudessem ser exibidos. No início da sessão fizemos exatamente deste modo quando criamos rotas dinâmicas. Porém, aqui será feito de modo diferente. Não será necessário ter um *frontend* para mostrar os dados.

Instale em seu computador o programa insomnia (<https://insomnia.rest/>). Este programa fará as requisições HTTP para nossa aplicação web. Veja o programa aberto na Figura 59.

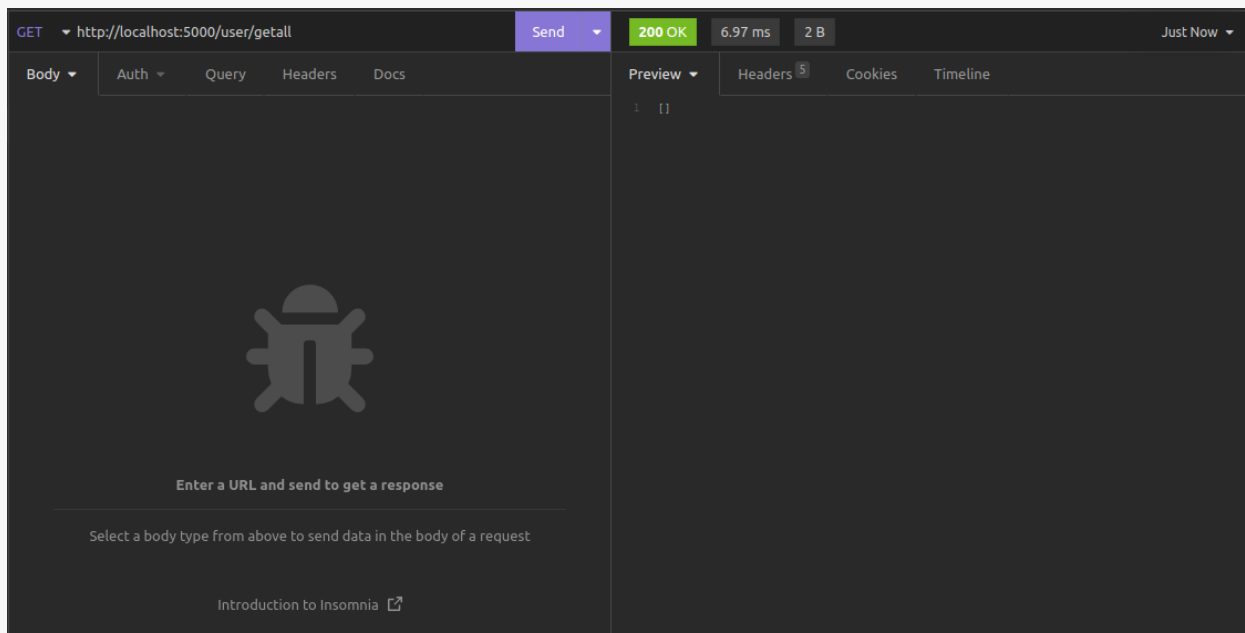
Para inserir dados em nosso banco de dados via sistema é necessário primeiramente criar essa rota. O código da rota de criação será mostrado abaixo.

Figura 59 - Programa insomnia.



Para criar uma requisição clique em “+” no canto superior esquerdo do programa e selecione “HTTP request”. Uma tela abrirá no meio do programa. Nessa tela selecione a opção GET (pois a primeira rota que vamos testar utiliza o método GET). Na caixa de seleção ao lado da opção GET escreva a rota que será acessada “<http://localhost:5000/user/getall>”. Sua requisição deve ficar exatamente como mostrado na Figura 60.

Figura 60 - Requisição para exibir todos os usuários.



Clique em “Send” para enviar a requisição. Repare que a resposta foi um array vazio [], e que existe o código “200 ok” na parte superior da resposta. Agora o leitor deve se perguntar por que isso aconteceu. O array vazio se deve ao fato de que não tem nenhum usuário cadastrado no sistema. O código “200 ok” se deve ao fato que a requisição foi bem sucedida.

Para criar um usuário, a rota “create” será criada. O código dessa rota será mostrado abaixo.

```
@app.route('/user/create', methods=['POST'])
def create():
    body = request.get_json()
    session = db.session()
    try:
        user =
        User(nome=body['nome'], telefone=body['telefone'], email=body['email'])
        session.add(user)
        session.commit()
        return Response(json.dumps([user.serialize()]))
    except Exception as e:
        print(e)
        session.rollback()
```

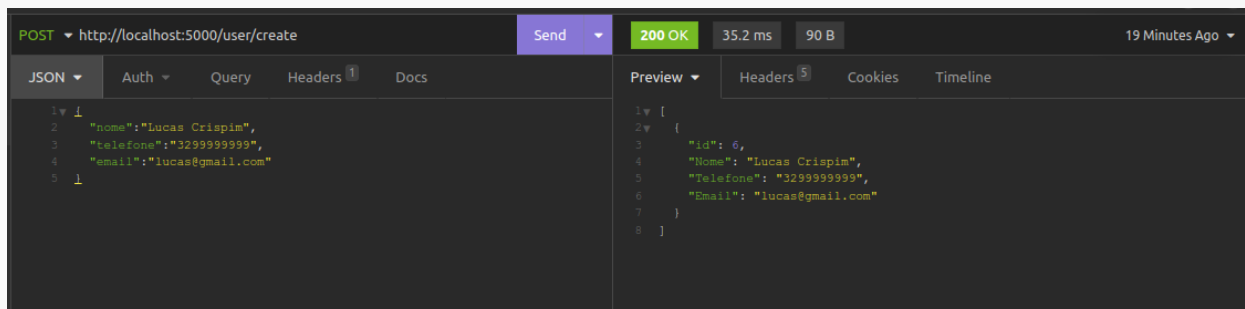
```
return {"erro": "não conseguimos gravar o usuário"}
finally:
    session.close()
```

Novamente, a primeira linha se refere a rota (`'/user/create'`) e o método (`methods=['POST']`) que é usado para enviar os dados. Na linha seguinte a função que será chamada quando a requisição chegar na rota, começa a ser escrita (`def create():`). As linhas seguintes são de recebimento dos dados da requisição pelo corpo dela (`body = request.get_json()`) e abertura de sessão com o banco de dados (`session = db.session()`). Na sequência é aberto um bloco (`try: except Exception as e: finally:`). O objetivo desse bloco é fazer a operação de persistência (guardar o dado no banco) e caso ocorra algum erro crítico no bloco `try:`, a operação vá para o bloco `except Exception as e:` e se desfça o que já foi feito. O bloco `finally:` é executado independentemente se a operação tenha sucesso ou não. Dentro do bloco `try:` os dados são extraídos (`user = User(nome=body['nome'], telefone=body['telefone'], email=body['email'])`) e guardados (`session.add(user) session.commit()`) e depois retornados (`return Response(json.dumps([user.serialize()]))`). Para o bloco `except Exception as e:` toda a operação já feita é revertida (`session.rollback()`) e uma mensagem de erro é retornada para o usuário (`return {"erro": "não conseguimos gravar o usuário"}`). Finalmente no bloco `finally:` a sessão é terminada `session.close()`.

Para enviar o dado de um usuário para ser guardado na aplicação, uma nova requisição terá que ser criada. Veja na Figura 61 como a requisição deve ser montada no programa insomnia.



Figura 61 -Requisição de criação de usuário.



Na Figura 61 é possível ver a requisição de criação de um usuário com sua respectiva resposta. Desta vez o método da requisição deve ser POST, como exigido na rota de criação, e a requisição deve ter um corpo (“body”) com os dados de usuário (nome, e-mail e telefone). Aqui o leitor é encorajado a mudar a requisição e ver os efeitos. Por exemplo, pode-se mudar de POST para GET ou alterar os campos no corpo da requisição.

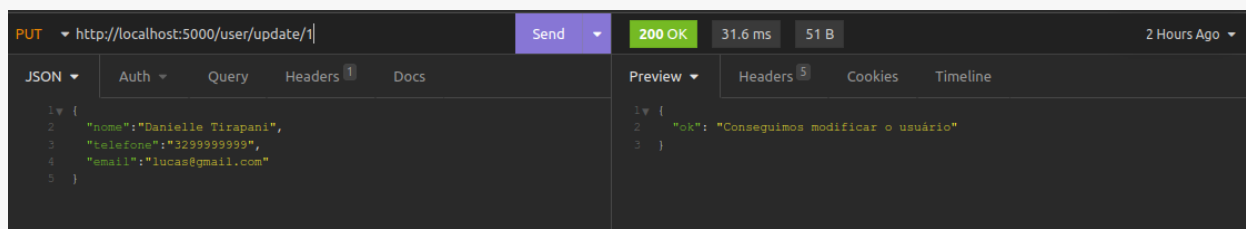
Para alterar o usuário pré existente uma nova rota deve ser criada. O código dessa rota será mostrado abaixo.

```
@app.route('/user/update/<int:user_id>', methods=['PUT'])
def update(user_id):
    session = db.session()
    try:
        body = request.get_json()
        user = session.query(User).get(user_id)
        if body and body['nome']:
            user.nome = body['nome']
        if body and body['telefone']:
            user.telefone = body['telefone']
        if body and body['email']:
            user.email = body['email']

        session.commit()
        return {"ok": "Conseguimos modificar o usuário"}
    except Exception as e:
        session.rollback()
        return {"erro": "não conseguimos atualizar o usuário"}
    finally:
        session.close()
```

Para a etapa de atualização de usuário, o id deve ser enviado como um parâmetro no endereço da requisição (`"/user/update/<int:user_id>"`) e o método utilizado neste caso foi PUT. Todavia, o leitor não deve se prender a demonstração que foi feita aqui e caso seja necessário ele deve experimentar outras formas de construir as rotas mostradas. Por outro lado, existe a crença por parte do autor que o modo que foi feito é uma boa abordagem para desenvolvedores iniciantes. Neste caso a função que é invocada (`def update(user_id):`) recebe o id do usuário para o qual algum campo será modificado. Nas próximas linhas o corpo da requisição é extraído (`body = request.get_json()`) e uma requisição para buscar o usuário com o id recebido é realizada (`user = session.query(User).get(user_id)`). As linhas seguintes se referem a extração dos dados do corpo da requisição, sempre testando se os mesmos existem (`if body and body['nome']:`). As modificações são salvas (`session.commit()`) e uma resposta de confirmação da alteração do usuário é enviada (`return {"ok": "Conseguimos modificar o usuário"}`). Caso algo de errado aconteça no bloco `try:`, qualquer modificação é desfeita (`session.rollback()`) e uma mensagem de que a operação não foi concluída é enviada ao usuário (`return {"erro": "não conseguimos atualizar o usuário"}`). Ao final, mesmo que a operação não tenha sido completada, a sessão é finalizada (`session.close()`). A requisição que faz a alteração no usuário de id 1 é mostrada na Figura 62.

Figura 62 - Requisição para alterar um usuário.

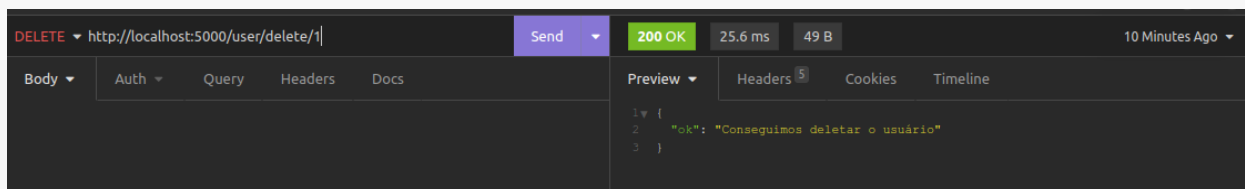


Por último, será mostrado o código para a deleção de um registro de usuário.

```
@app.route('/user/delete/<int:user_id>', methods=['DELETE'])
def delete(user_id):
    session = db.session()
    try:
        user = session.query(User).get(user_id)
        session.delete(user)
        session.commit()
        return {"ok": "Conseguimos deletar o usuário"}
    except Exception as e:
        session.rollback()
        return {"erro": "não conseguimos deletar o usuário"}
    finally:
        session.close()
```

Este código é mais do mesmo do que já foi mostrado nas outras rotas. Existe uma pequena diferença no método. O método utilizado é o DELETE. Para deleção, o código `session.delete(user)` é utilizado. A requisição para deleção é mostrada na Figura 63.

Figura 63 - Requisição para alterar um usuário.



A requisição mostrada na Figura 63 é bem semelhante à mostrada para a atualização de usuário. Neste caso a modificação é apenas no método que é utilizado, DELETE.

O código completo que está no arquivo `app.py` é mostrado abaixo.

```
from flask import Flask, render_template
from flask_migrate import Migrate
from config import DevelopmentConfig
from models.user import db, User
from flask import Response, request
import json

app = Flask(__name__)
app.config.from_object(DevelopmentConfig())
db.init_app(app)
```

```
migrate = Migrate(app, db)

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

@app.route('/user/getall', methods=['GET'])
def getAll():
    session = db.session()
    users = session.query(User).all()
    users_json = [user.serialize() for user in users]
    session.close()
    return Response(json.dumps(users_json))

@app.route('/user/create', methods=['POST'])
def create():
    body = request.get_json()
    session = db.session()
    try:
        user = User(nome=body['nome'], telefone=body['telefone'], email=body['email'])
        session.add(user)
        session.commit()
        return Response(json.dumps([user.serialize()]))
    except Exception as e:
        print(e)
        session.rollback()
        return {"erro": "não conseguimos gravar o usuário"}
    finally:
        session.close()

@app.route('/user/update/<int:user_id>', methods=['PUT'])
def update(user_id):
    session = db.session()
    try:
        body = request.get_json()
        user = session.query(User).get(user_id)
        if body and body['nome']:
            user.nome = body['nome']
        if body and body['telefone']:
            user.telefone = body['telefone']
        if body and body['email']:
            user.email = body['email']

        session.commit()
        return {"ok": "Conseguimos modificar o usuário"}
    except Exception as e:
        session.rollback()
        return {"erro": "não conseguimos atualizar o usuário"}
    finally:
```

```
session.close()

@app.route('/user/delete/<int:user_id>', methods=['DELETE'])
def delete(user_id):
    session = db.session()
    try:
        user = session.query(User).get(user_id)
        session.delete(user)
        session.commit()
        return {"ok": "Conseguimos deletar o usuário"}
    except Exception as e:
        session.rollback()
        return {"erro": "não conseguimos deletar o usuário"}
    finally:
        session.close()

if __name__ == "__main__":
    app.run(debug=True, port=5000)
```

Conforme mostrado no código acima, é possível criar quatro operações fundamentais num sistema (exibir, criar, modificar e deletar dados) de modo simples usando o Flask. A abordagem escolhida para mostrar essas funcionalidades pode ser inédita para o leitor. Por outro lado, o leitor teve contato com uma estratégia de desenvolvimento de sistemas web que está muito popularizada. Esta abordagem é a de ter um *frontend* separado do *backend*. Para agregar mais ainda, o leitor teve contato com uma ferramenta que realiza requisições (insomnia) e que também está muito popularizada. Para além do que foi mostrado aqui, o autor encoraja o leitor a fazer modificações na aplicação construída aqui.

## Referências

---

GRINBERG, Miguel. Desenvolvimento web com Flask: Desenvolvendo aplicações web com Python. Brasil: Novatec Editora, 2019. p. 341.

MAZZA, Lucas. HTML5 e CSS3: Domine a web do futuro. Brasil: Casa do Código, 2014. p. 265.

SILVA, Tiago. Django de A a Z: Crie aplicações web rápidas, seguras e escaláveis com Python. Brasil: Casa do Código, 2021. p. 502.

SILVA, Tiago. Flask de A a Z: Crie aplicações web mais completas e robustas em Python. Brasil: Casa do Código, 2019. p. 353.