Renderização de páginas no servidor (*Server Side Rendering*)

Disciplina: Desenvolvimento Web

Prof. Dr. Rafael Will M. de Araujo

Conteúdo

1 Introdução

2 Server Side Rendering

3 Herança de Templates

Renderização de páginas no Flask

- Em nossa disciplina, vamos renderizar as páginas HTML no lado do servidor.
- Para isso, o Flask utiliza a linguagem de templates **Jinja2**, que já vem junto com o framework;
- Um *template* nada mais é do que um **modelo** de uma página HTML;
- O **Jinja** é uma linguagem de *template* muito utilizada no mundo Python, baseado na sintaxe do Django (outro framework Python).

Renderização de páginas no Flask

- O Flask permite definir arquivos *templates*, que são arquivos HTML com anotações específicas, chamadas de *template tags*, e um motor de renderização que utiliza códigos Python e os *templates* para gerar o HTML necessário.
- Jinja2 já vem por padrão na instalação do Flask e sua integração é automática. Basta configurar na aplicação Flask qual a pasta que ela deve procurar os arquivos *templates*, que por padrão é a **pasta templates** junto com o arquivo da aplicação. Assim, eles já poderão ser usados na função `render_template()`.

Para facilitar a notação de diretórios e subdiretórios nos próximos slides, vamos utilizar a seguinte notação:

- `/` - indica o diretório raiz do projeto;
- `/abc/xyz.html` - indica o diretório *abc* na raiz do projeto e o arquivo *xyz.html* dentro deste diretório;
- `/abc/def/xyz.html` - indica o diretório *abc* na raiz do projeto que contém o diretório *def* que, por sua vez, contém o arquivo *xyz.html*.

Alterando o diretório de *templates*

- O Flask permite que o diretório de templates seja alterado:

Aplicação Flask

```
1  from flask import Flask, render_template

2  app = Flask(__name__, template_folder='meus_modelos')

3  @app.route('/')
4  def index():
5    return render_template('index.html')

6  app.run()
```

- Note que foi adicionado o parâmetro nomeado `template_folder` ao construtor da classe `Flask`.

Arquivo: */templates/index.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Exemplo</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Bem-vindo ao index.html!</h1>
9    <p>Esse arquivo está
10   na pasta <strong>"templates"</strong></p>
11 </body>
12 </html>
```

Arquivo: */meus_modelos/index.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Exemplo</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Bem-vindo ao index.html!</h1>
9    <p>Esse arquivo está
10   na pasta <strong>"meus_modelos"</strong></p>
11 </body>
12 </html>
```

O diretório de arquivos estáticos (*static*)

- O Flask utiliza o diretório *static* como o diretório padrão para arquivos estáticos. Em geral utilizamos esse diretório para guardar arquivos CSS, JavaScript, imagens e outros arquivos que não terão o seu conteúdo alterado dinamicamente.
- É possível definir um novo diretório estático no construtor da classe Flask através do parâmetro nomeado `static_folder`:

Aplicação Flask

```
1  from flask import Flask, render_template
2
2  app = Flask(__name__, static_folder='arquivos_estaticos')
3
3  @app.route('/')
4  def index():
5    return render_template('index.html')
6
6  app.run()
```

- Em geral não vamos alterar esse diretório, isto é, vamos continuar utilizando o diretório padrão *static*. Crie esse diretório no projeto, e adicione os arquivos necessários lá.
  - ▷ Você pode criar subdiretórios (estilos, scripts, imagens, etc) dentro do diretório *static* para organizar melhor o conteúdo.

O diretório de arquivos estáticos (*static*)

- A referência aos arquivos CSS e JavaScript dentro dos nossos *templates* pode ser feita conforme o seguinte exemplo:

Arquivo: */templates/index.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Exemplo</title>
5    <meta charset="UTF-8">
6    <link rel="stylesheet" type="text/css" href="/static/estilos.css">
7    <script src="/static/scripts.js" defer></script>
8  </head>
9  <body>
10   <h1>Bem-vindo ao index.html!</h1>
11   <p>Esse arquivo está
12   na pasta <strong>"templates"</strong></p>
13 </body>
14 </html>
```

Páginas de erro personalizadas

- Também é possível personalizar as páginas de erro:

Personalizando páginas de erro

```
1  from flask import Flask, render_template
2
2  app = Flask(__name__)
3
3  @app.errorhandler(404)
4  def erro404(e):
5    return render_template('erro.html'), 404
6
6  app.run()
```

Arquivo: *templates/erro.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Exemplo</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Oops!</h1>
9    <p>O arquivo que você procura não existe!</p>
10 </body>
11 </html>
```

- Note que o retorno da função também deve ter o código de erro 404, caso contrário será considerado o código 200.

Conteúdo

1 Introdução

2 Server Side Rendering

3 Herança de Templates

Template tags no Jinja

- Arquivos HTML, nos moldes que vimos anteriormente, já estão prontos para serem usados.
- O Jinja2 introduz *template tags* específicas para tornar os arquivos mais inteligentes para o uso com os nossos dados. Essas *template tags* são divididos nos seguintes tipos:
  - `{% TEMPLATE TAG %}` - expressões de código ou lógica;
  - `{{ VALOR }}` - expressões de impressão no HTML;
  - `{# COMENTARIO #}` - usado para comentários em bloco.

Utilizando *template tags* no Jinja: exemplo 1

- Podemos enviar valores para os nossos templates da seguinte forma:

Aplicação em Flask: exemplo 1

```
1  from flask import Flask, render_template
2
2  app = Flask(__name__)
3
3  @app.route('/aluno')
4  def aluno():
5    return render_template('aluno.html', nome_aluno='Rafael', matricula=123456)
6
6  app.run(debug=True)
```

Observe os parâmetros nomeados adicionais na função `render_template()`: `nome_aluno` e `matricula`.

Arquivo: */templates/aluno.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ nome_aluno }}!</h1>
9    <p>O seu número de matrícula é: {{ matricula }}</p>
10 </body>
11 </html>
```

Utilizando *template tags* no Jinja: exemplo 1

Arquivo: */templates/aluno.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ nome_aluno }}!</h1>
9    <p>O seu número de matrícula é: {{ matricula }}</p>
10 </body>
11 </html>
```

- Ou seja, os valores dentro das *templates tags* `{{ nome_aluno }}` e `{{ matricula }}` serão substituidos pelos parâmetros nomeados respectivos.
- Você pode adicionar quantos parâmetros nomeados desejar na função `render_template()`.

Utilizando *template tags* no Jinja: exemplo 2

- Também é possível enviar objetos mais complexos, como dicionários:

Aplicação em Flask: exemplo 2

```
1  from flask import Flask, render_template

2  app = Flask(__name__)

3  @app.route('/aluno')
4  def aluno():
5    pessoa = {'nome': 'Rafael', 'matricula': 123456}
6    return render_template('aluno.html', dados=pessoa)

7  app.run(debug=True)
```

Arquivo: */templates/aluno.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ dados.nome }}!</h1>
9    <p>O seu número de matrícula é: {{ dados.matricula }}</p>
10 </body>
11 </html>
```

Utilizando *template tags* no Jinja: exemplo 2

- As chaves do dicionário podem ser acessados usando o operador colchete: `dicionario['chave']`

Arquivo: *templates/aluno.html* (outra maneira de acessar os atributos)

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ dados['nome'] }}!</h1>
9    <p>O seu número de matrícula é: {{ dados['matricula'] }}</p>
10 </body>
11 </html>
```

Utilizando *template tags* no Jinja: exemplo 3

- É possível enviar objetos instanciados a partir de uma classe e acessar os seus atributos e métodos:

Arquivo *pessoa.py*: exemplo 3

```
1  class Pessoa:

2    def __init__(self, nome, matricula):
3      self.nome = nome
4      self.matricula = matricula
5
6    def get_nome(self):
7      return self.nome
8
9    def get_matricula(self):
10     return self.matricula
```

Aplicação em Flask: exemplo 3

```
1  from flask import Flask, render_template
2  from pessoa import Pessoa

3  app = Flask(__name__)

4  @app.route('/aluno')
5  def aluno():
6    pessoa = Pessoa('Rafael', 123456)
7    return render_template('aluno.html', objeto=pessoa)

8  app.run(debug=True)
```

Utilizando *template tags* no Jinja: exemplo 3

- Dessa forma, podemos acessar os atributos da classe diretamente no template (caso os atributos sejam públicos):

Arquivo: */templates/aluno.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ objeto.nome }}!</h1>
9    <p>O seu número de matrícula é: {{ objeto.matricula }}</p>
10 </body>
11 </html>
```

- Se os atributos forem privados ou se o desenvolvedor assim desejar, é possível acessar métodos do objeto. Também é possível enviar valores para os parâmetros dos métodos (se o método permitir):

Arquivo: */templates/aluno.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ objeto.get_nome() }}!</h1>
9    <p>O seu número de matrícula é: {{ objeto.get_matricula() }}</p>
10 </body>
11 </html>
```

Utilizando *template tags* no Jinja: exemplo 4

- Além de exibir valores, podemos utilizar algumas estruturas de fluxo para melhorar os nossos *templates*:

Aplicação em Flask: exemplo 4

```
1  from flask import Flask, render_template

2  app = Flask(__name__)

3  @app.route('/aluno')
4  def aluno():
5    return render_template('aluno.html', nome_aluno='Rafael', matricula=123456, nota=5.9)

6  app.run(debug=True)
```

Arquivo: */templates/aluno.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Página do aluno</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Olá {{ nome_aluno }}!</h1>
9    <p>O seu número de matrícula é: {{ matricula }}</p>
10   {% if nota >= 6 %}
11     <p>Situação: aprovado</p>
12   {% elif nota >= 3 and nota < 6 %}
13     <p>Situação: em recuperação</p>
14   {% else %}
15     <p>Situação: reprovado</p>
16   {% endif %}
17 </body>
18 </html>
```

Utilizando *template tags* no Jinja: exemplo 5

- Também é permitido usar o comando **for** para iterar sobre uma sequência:

Aplicação em Flask: exemplo 5

```
1  from flask import Flask, render_template

2  app = Flask(__name__)

3  @app.route('/alunos')
4  def listagem_alunos():
5    alunos = ['Ana Maria', 'Beatriz Silva', 'Carlos Eduardo', 'Daniel Santos', 'Emilia Ferreira']
6    return render_template('alunos.html', disc='Desenvolvimento Web', lista_alunos=alunos)

7  app.run(debug=True)
```

Arquivo: */templates/alunos.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Listagem de alunos</title>
5    <meta charset="UTF-8">
6  </head>
7  <body>
8    <h1>Lista de alunos da disciplina {{ disc }}</h1>
9    <ul>
10     {% for aluno in lista_alunos %}
11       <li>{{ aluno }}</li>
12     {% endfor %}
13   </ul>
14 </body>
15 </html>
```

Conteúdo

1 Introdução

2 Server Side Rendering

3 Herança de Templates

Herança de *templates*

- Uma das grandes vantagens do Jinja2 é a herança de templates. É possível criar um *template* com um modelo básico com todas as estruturas básicas de uma página (como o cabeçalho, rodapé, menus de navegação, etc), bem como as *tags* de configuração dentro da **head** do HTML.
- Em seguida, podemos fazer com que todos os outros *templates* que criarmos herdem (incorporem) essa estrutura básica. Assim, podemos nos concentrar apenas no seu conteúdo específico de cada nova página criada.

Exemplo: herança de *templates* no Jinja

- Suponha as seguintes páginas/rotas na nossa aplicação:

Aplicação no Flask: herança de templates

```
1  from flask import Flask, render_template

2  app = Flask(__name__)

3  @app.route('/pagina1')
4  def pagina1():
5    return render_template('pagina1.html')

6  @app.route('/pagina2')
7  def pagina2():
8    return render_template('pagina2.html')

9  @app.route('/pagina3')
10 def pagina3():
11   return render_template('pagina3.html')

12 app.run(debug=True)
```

Exemplo: herança de *templates* no Jinja

- Vamos criar um arquivo chamado *base.html* que contém a estrutura básica da nossa aplicação:

Arquivo: */templates/base.html*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <meta charset="UTF-8">
5    <link rel="stylesheet" type="text/css" href="/static/estilos.css">
6    {% block head %} {% endblock %}
7  </head>
8  <body>
9    <header>
10     <nav>
11       <a href="pagina1">Página 1</a> |
12       <a href="pagina2">Página 2</a> |
13       <a href="pagina3">Página 3</a>
14     </nav>
15   </header>
16   <main>
17     {% block conteudo %}
18     {% endblock %}
19   </main>
20   <footer>
21     Rodapé da nossa página - Todos os direitos reservados
22   </footer>
23 </body>
24 </html>
```

- Note que definimos duas regiões (blocos):
  - `head` - onde poderemos adicionar conteúdo extra dentro do *head* (título da página, outros arquivos de estilos ou scripts);
  - `conteudo` - onde iremos adicionar o conteúdo principal (miolo) da página.

Exemplo: herança de *templates* no Jinja

- A partir do arquivo *base.html*, podemos herdar (incorporar) a sua estrutura nas páginas 1, 2 e 3, através do comando: `{% extends "base.html" %}`

Arquivo: *templates/pagina1.html*

```
1  {% extends "base.html" %}

2  {% block head %}
3  <title>Título da página 1</title>
4  {% endblock head %}

5  {% block conteudo %}
6  <h1>Bem-vindo à página 1</h1>
7  <p>Conteúdo da página 1</p>
8  {% endblock conteudo %}
```

Arquivo: *templates/pagina2.html*

```
1  {% extends "base.html" %}

2  {% block head %}
3  <title>Título da página 2</title>
4  {% endblock head %}

5  {% block conteudo %}
6  <h1>Bem-vindo à página 2</h1>
7  <p>Conteúdo da página 2</p>
8  {% endblock conteudo %}
```

Arquivo: *templates/pagina3.html*

```
1  {% extends "base.html" %}

2  {% block head %}
3  <title>Título da página 3</title>
4  {% endblock head %}

5  {% block conteudo %}
6  <h1>Bem-vindo à página 3</h1>
7  <p>Conteúdo da página 3</p>
8  {% endblock conteudo %}
```

- Note a sintaxe utilizada para redefinir o conteúdo dos blocos nomeados **head** e **conteudo**.
- Ao abrir as páginas no navegador, observe o código fonte gerado.