Aula 07

Django

Tag if

Anteriormente, vimos como repetir trechos de código usando a tag for do Django.

O Django também possui outra tag de controle, a tag if. Assim como no Python, ela é usada para executar determinados trechos de códigos se uma determinada condição for satisfeita.

Veja como podemos reorganizar a página **/linguagem_cpp/templates/linguagem_cpp/paginas/versao.html**:

```
C:\Users\gutoh\curso\linguagem_cpp\templates\linguagem_cpp\paginas\versao.html
<!DOCTYPE html>
<html lang="en">
    {% include 'linguagem_cpp/parciais/head.html' %}
<body>
    {% include 'linguagem_cpp/parciais/titulo.html' %}
    <p>sou a página das versões</p>
    {% if titulo %}
        <h3>{{titulo}}</h3>
        <p>{{descricao}}</p>
    {% else %}
        <h3>Versão não encontrada</h3>
    {% endif %}
    <a href="{% url 'cpp:documentacao' %}">voltar</a>
</body>
</html>
```

Agora, adicionamos uma condição para o texto das versões aparecer somente quando algo for enviado pela função view_versao da **/linguagem_cpp/views.py**. Se alguma versão inexistente for digitada na url, vai exibir a mensagem de "Versão não encontrada".

Faker

Nosso projeto está ficando grande. Agora, se faz necessário que sejamos capazes de gerar dados aleatórios. Imagina ter que criar dados falsos para TODOS os nossos aplicativos? Isso não fica prático e usar dados reais não é algo que se costuma em um ambiente de desenvolvimento.

Para isso, o Python tem um pacote chamado Faker que possui como funcionalidade justamente gerar dados fictícios na forma especificada pelo programador. Como ainda não temos instalado esse pacote, instalá-lo-emos.

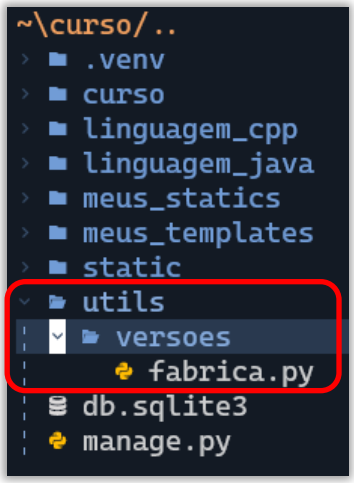
Veja o comando abaixo (lembre-se de executar com o ambiente virtual ativado):

```
(.venv) C:\Users\gutoh\curso>pip install Faker
Collecting Faker
  Downloading Faker-18.11.1-py3-none-any.whl (1.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 6.8 MB/s eta 0:00:00
Collecting python-dateutil≥2.4 (from Faker)
  Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
Collecting six≥1.5 (from python-dateutil≥2.4→Faker)
  Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, python-dateutil, Faker
Successfully installed Faker-18.11.1 python-dateutil-2.8.2 six-1.16.0

(.venv) C:\Users\gutoh\curso>
```

Uma vez instalado, vamos criar uma nova pasta na raiz do nosso projeto. Diferente das outras pastas, onde tivemos que especificar para o Django da existência dela, essa não vai ser necessário, pois quem vai realizar a busca dela será o próprio Python.

Vamos chamá-la de de utils e dentro dela criar uma pasta especificando a funcionalidade dela. Veja a nova estrutura de pastas abaixo:



Agora temos um arquivo chamado **fabrica.py** dentro da pasta **versoes**. Isso é importe porque se tivemos mais de um aplicativo com funcionalidade diferentes (como um aplicativo para gerenciar uma revenda de automóveis e outro para gerenciar um time de futebol), vai nos ajudar a identificar qual a finalidade dos arquivos dentro da pasta. Afinal, pode ser que precisemos de mais arquivos assim no futuro.

O nome do arquivo **/utils/versoes/fabrica.py** será esse nome, pois essa é a finalidade dele: fabricar versões falsas para nosso programa.

O arquivo mostrado a seguir é específico para ser usado para gerar versões falsas para nosso programa. Cada aplicativo terá uma fábrica completamente diferente. Fica a cargo do desenvolvedor realizar essa configuração.

Veja a fábrica das versões:

C:\Users\gutoh\curso\utils\versoes\fabrica.py

```
26 from faker import Faker
25
24 # especificando o idioma do texto gerado
23 FAKE = Faker('pt_BR')
22
21 # variável global usada gerar um valor
20 # sequencial para as versões
19 ver_num = 0
18
17 def cria_versao():
16     global ver_num
15     ver_num += 1
14     return {
13         'titulo': f'Versão {ver_num}',
12         'versao': ver_num,
11         'descricao': FAKE.sentence(nb_words=10),
10         'data_lanc': FAKE.date_time(),
 9         'tags': [FAKE.word() for _ in range(5)],
 8         'responsavel': {
 7             'nome': FAKE.first_name(),
 6             'sobrenome': FAKE.last_name(),
 5         },
 4         'foto': {
 3             'url': 'https://loremflickr.com/320/240/python',
 2         },
 1     }
27
```

Vamos passar pelo código acima:

- Linha 1: importamos o Faker;
- Linha 4: criamos uma constante a partir da classe Faker passando como parâmetro o idioma que queremos que o texto seja gerado;
- Linha 8: como as versões são progressivas, criamos uma variável para ser usada de forma global e incrementada sempre que uma versão é criada;
- Linha 10: é iniciada a função que vai ser chamada sempre que quisermos criar uma nova versão falsa;
- A partir da linha 13: retornarmos um dicionário que será montado com os dados falsos;
    - Usamos o método sentence() para gerar frases, que espera receber a quantidade de palavras geradas através do parâmetro **nb_words**;
    - Usamos o método date_time() para gerar uma data falsa;
    - Usamos o método word() para gerar apenas uma palavra;
    - Usamos os métodos first_name() e last_name() para gerar nomes próprios para construir um nome;
    - Linha 24: usamos o site https://loremflickr.com para gerar uma foto aleatória (nesse caso estamos gerando uma imagem de tamanho 320x240 com o tema do Python;

Esse código vai gerar todos esses dados falsos. Podemos testar esses dados executando o módulo diretamente para ver se está gerando como o esperado:

```python
if __name__ == '__main__':
    for _ in range(5):
        print(cria_versao())
```

```
(.venv) C:\Users\gutoh\curso>python utils\versoes\fabrica.py
{'titulo': 'Versão 1', 'versao': 1, 'descricao': 'Architecto eum magni libero possimus quia nostrum neque asperiores corporis re
iciendis.', 'data_lanc': datetime.datetime(1988, 2, 12, 6, 46, 4), 'tags': ['suscipit', 'earum', 'necessitatibus', 'officiis', '
consequatur'], 'responsavel': {'nome': 'Arthur', 'sobrenome': 'Oliveira'}, 'foto': {'url': 'https://loremflickr.com/320/240/pyth
on'}}
{'titulo': 'Versão 2', 'versao': 2, 'descricao': 'Quisquam id beatae et consectetur eligendi vel.', 'data_lanc': datetime.dateti
me(1975, 2, 17, 18, 16, 50), 'tags': ['cupiditate', 'eligendi', 'id', 'optio', 'eos'], 'responsavel': {'nome': 'Luiz Gustavo', '
sobrenome': 'Rezende'}, 'foto': {'url': 'https://loremflickr.com/320/240/python'}}
{'titulo': 'Versão 3', 'versao': 3, 'descricao': 'Vero natus dicta quasi aliquam molestiae illum illum deserunt dolores.', 'data
_lanc': datetime.datetime(1982, 6, 19, 14, 38, 22), 'tags': ['laboriosam', 'distinctio', 'possimus', 'est', 'fugit'], 'responsav
el': {'nome': 'Gabriel', 'sobrenome': 'Teixeira'}, 'foto': {'url': 'https://loremflickr.com/320/240/python'}}
{'titulo': 'Versão 4', 'versao': 4, 'descricao': 'Incidunt explicabo eos a dolor maxime consectetur expedita facilis amet nulla.
', 'data_lanc': datetime.datetime(1985, 3, 12, 13, 58, 54), 'tags': ['voluptatum', 'exercitationem', 'aspernatur', 'natus', 'cor
poris'], 'responsavel': {'nome': 'Vitor Gabriel', 'sobrenome': 'da Mata'}, 'foto': {'url': 'https://loremflickr.com/320/240/pyth
on'}}
{'titulo': 'Versão 5', 'versao': 5, 'descricao': 'Architecto itaque accusantium voluptates autem perspiciatis ullam at.', 'data_
lanc': datetime.datetime(1976, 11, 5, 23, 23), 'tags': ['mollitia', 'ex', 'natus', 'expedita', 'ipsum'], 'responsavel': {'nome':
 'Catarina', 'sobrenome': 'Martins'}, 'foto': {'url': 'https://loremflickr.com/320/240/python'}}

(.venv) C:\Users\gutoh\curso>
```

Agora que estão sendo gerados os dados falsos, vamos chamar a função nos nossos arquivos das **views.py**.

Veja como ficará o arquivo **/linguagem_cpp/views.py**:

```
C:\Users\gutoh\curso\linguagem_cpp\views.py
 27 from django.shortcuts import render
 26 from utils.versoes.fabrica import cria_versao
 25
 24 VERSOES = [cria_versao() for _ in range(5)]
 23
 22 def view_inicio(request):
 21     dicionario = {'nome': 'Tom', 'sobrenome': 'Cruise'}
 20     return render(request, 'linguagem_cpp/paginas/inicio.html', context=dicionario)
 19
 18 def view_sobre(request):
```

Agora, estamos criando as versões de forma dinâmica, sem ficar se preocupando em criar os dados manualmente.

Importamos a função e chamamos ela 5 vezes, criando 5 versões para nosso projeto.

E vamos atualizar o nosso HTML **/linguagem_cpp/templates/linguagem_cpp/paginas/versão.html** para mostrar todos os dados que estamos criando:

```
C:\Users\gutoh\curso\linguagem_cpp\templates\linguagem_cpp\paginas\versao.html
<!DOCTYPE html>
<html lang="en">
    {% include 'linguagem_cpp/parciais/head.html' %}
<body>
    {% include 'linguagem_cpp/parciais/titulo.html' %}
    <p>sou a página das versões</p>
    {% if titulo %}
        <h3>{{titulo}}</h3>
        <p>Descrição: {{descricao}}</p>
        <p>Data de lançamento: {{data_lanc}}</p>
        <p>Tags: {{tags}}</p>
        <p>Responsável: {{responsavel.nome}} {{responsavel.sobrenome}}</p>
        <img src="{{foto.url}}"><br>

    {% else %}
        <h3>Versão não encontrada</h3>
    {% endif %}
    <a href="{% url 'cpp:documentacao' %}">voltar</a>
</body>
</html>

```

Herança de Templates, a tag block e a tag extends

Dependendo do nosso projeto, pode ser que haja várias páginas de layout muito semelhantes.

Nos exemplos apresentados, temos os aplicativos **linguagem_cpp** e **linguagem_java** que têm muitas coisas semelhantes, por exemplo, as páginas de exibição da lista de versões, de contato, de início. Imagina que resolvemos alterar o layout geral da página de **documentacao.html**. Temos agora dois aplicativos, então não seria uma tarefa muito dispendiosa, mas e se tivéssemos mais de 20 aplicativos? Já se mostra um grande trabalho de atualização.

Pensando nisso, o Django possui uma forma de herdar um layout padrão por outras páginas com o uso da tag block.

Vamos criar uma página modelo para as páginas iniciais dos nossos aplicativos na pasta **/meus_templates/global/paginas/base_inicio.html**:

```
C:\Users\gutoh\curso\meus_templates\global\paginas\base_inicio.html
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="{% static 'global/css/estilo.css' %}">
    {% block css_app %}{% endblock css_app %}
    <title>{% block titulo %}TÍTULO INÍcio PADRÃO{% endblock %}</title>
</head>
<body>
    <h1>{% block tag_h1 %}Sou a <b>INÍCIO</b> geral{% endblock tag_h1 %}</h1>
    {% block menu %} <div>Menu do Site</div> {% endblock menu %}
    {% block um_texto %}
    <p>sou um texto genérico para todos meus aplicativos</p>
    {% endblock um_texto %}
</body>
</html>

```

Veja como funciona a tag block: temos que criar um nome para ela, por exemplo **block css_app**, e na chamada do endblock também temos que especificar o nome, **endblock app_css**.

Para uma página herdar esse modelo, temos que usar a tag extends nas páginas que usarão esse layout. Veja agora como ficará as páginas de início do C++ e do Java:

```
C:\Users\gutoh\curso\linguagem_cpp\templates\linguagem_cpp\paginas\inicio.html
{% extends 'global/paginas/base_inicio.html' %}
{% load static %}

{% block css_app %}
    <link rel="stylesheet" href="{% static 'linguagem_cpp/css/estilos.css' %}">
{% endblock css_app %}

{% block titulo %}C++{% endblock titulo %}

{% block tag_h1 %}sou o início do C++{% endblock tag_h1 %}

{% block menu %}
    {% include 'linguagem_cpp/parciais/menu.html' %}
{% endblock menu %}

<!-- repare que não estou carregando o bloco um_texto ⟶
```

```
C:\Users\gutoh\curso\linguagem_java\templates\linguagem_java\paginas\inicio.html
{% extends 'global/paginas/base_inicio.html' %}
{% load static %}

{% block css_app %}
    <link rel="stylesheet" href="{% static 'linguagem_java/css/estilos.css' %}">
{% endblock css_app %}

{% block titulo %}C++{% endblock titulo %}

{% block tag_h1 %}sou o início do Java{% endblock tag_h1 %}

<!-- repare que não estou carregando o bloco menu ⟶

{% block um_texto %}
    <p>sou um texto legal</p>
    <p>sou outro texto do bloco um_texto</p>
{% endblock um_texto %}
```

Como vai funcionar: quando se trabalha com herança de templates, a tag extends precisa ser a primeira tag no arquivo. Se você reparou, alguns blocos não foram chamados. Esses blocos serão carregados do **/meus_templates/global/paginas/base_inicio.html**, pois não sobre escritos. Já os blocos que são chamados, têm seus conteúdos originais sobrescritos com o novo conteúdo passado.

A herança de páginas do Django é uma ferramenta muito poderosa, mas também uma das mais complexas.

Exercícios para Praticar

1. Realize o que foi visto em aula nos seus projetos.

2. Continue o desenvolvimento dos sites passados na aula 05.