Aula 06

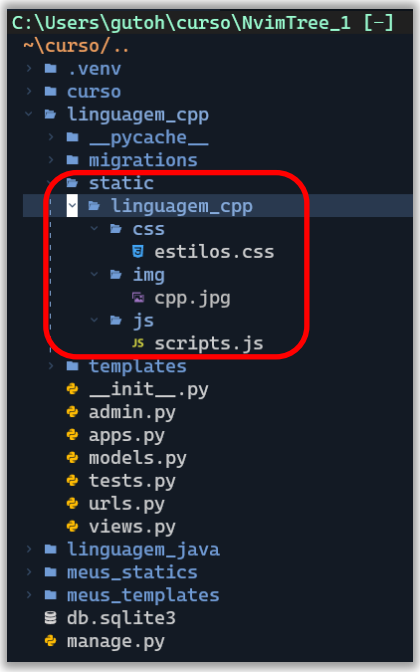
Django

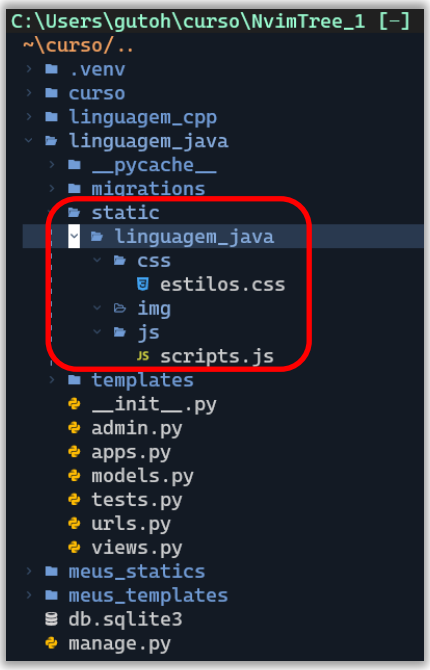
Arquivos Estáticos – part Trois

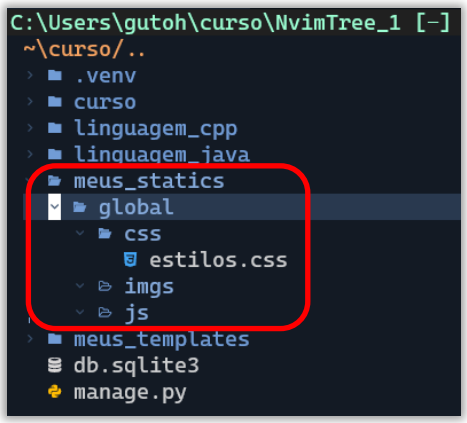
Nesse ponto, já estamos com os arquivos estáticos criados nos nossos aplicativos e para o nosso projeto.

Você deve estar se perguntando, por que criar tantas pastas dessa forma? Pois bem. O Django tem um comando onde ele coleta todos esses arquivos estáticos e os reorganiza em uma nova pasta na raiz do nosso projeto.

Veja como está a organização agora:







Cada uma das imagens acima está com o destaque nas pastas estáticas.

O Django possui um comando chamado collectstatic, que tem a funcionalidade de reorganizar os arquivos estáticos. Ele percorre todos os aplicativos e o projeto e copia todos os arquivos estáticos para uma nova pasta na raiz do projeto.

Veja o comando:

```
(.venv) C:\Users\gutoh\curso>python manage.py collectstatic

You have requested to collect static files at the destination
location as specified in your settings.

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel: yes
```

Ao realizarmos a chamada dele, ele nos avisa que requisitamos a coleta dos arquivos estáticos para a localização especificada nas nossas configurações e que eles serão sobrescritos. Depois de confirmar, será gerada uma mensagem de erro.

Veja ela abaixo:

```
  File "C:\Users\gutoh\curso\.venv\Lib\site-packages\django\contrib\staticfiles\storage.py", line 39,
in path
    raise ImproperlyConfigured(
django.core.exceptions.ImproperlyConfigured: You're using the staticfiles app without having set the
STATIC_ROOT setting to a filesystem path.
```

O erro nos diz que estamos usando o aplicativo **staticfiles** (ele é um aplicativo do Django responsável por fazer a coleta dos arquivos estáticos) sem definir o **STATIC_ROOT** nas configurações do nosso projeto, no arquivo **/curso/settings.py**.

Veja como vai ficar o arquivo **/curso/settings.py**:

```
C:\Users\gutoh\curso\curso\settings.py
119
120 # Static files (CSS, JavaScript, Images)
121 # https://docs.djangoproject.com/en/4.2/howto/static-files/
122
123 STATIC_URL = 'static/'
124 STATICFILES_DIRS = [
125     BASE_DIR / 'meus_statics',
126 ]
127 STATIC_ROOT = BASE_DIR / 'static'
128
```

Acima, especificamos o nome da pasta a ser criada e onde ela será criada. Nesse caso, na raiz do nosso projeto.

Agora, vamos executar novamente o comando collectstatic:

```
(.venv) C:\Users\gutoh\curso>python manage.py collectstatic

131 static files copied to 'C:\Users\gutoh\curso\static'.

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

Dessa vez, o comando foi executado sem erros e nos é notificado quantos arquivos estáticos foram copiados.

Veja como vai ficar a nova configuração das pastas do projeto:

```
C:\Users\gutoh\curso\NvimTree_1 [-]
 ~\curso/..
  .venv
  curso
  linguagem_cpp
  linguagem_java
  meus_statics
  meus_templates
  static
    admin
      css
      img
      js
    global
      css
    linguagem_cpp
      css
      img
      js
    linguagem_java
      css
      js
  db.sqlite3
  manage.py
```

Repare que uma nova pasta chamada **static** (o mesmo nome que demos no **/curso/settings.py**) foi criada. Lá dentro, temos uma cópia de todos os arquivos estáticos do nosso projeto e aplicativos. Caso esse comando seja executado uma segunda vez, os arquivos dessa pasta serão atualizados com as alterações feitas nos arquivos estáticos de dentro dos aplicativos (por isso a mensagem avisando).

Quando preparamos nosso projeto para publicação, temos que alterar alguns campos em **/curso/settings.py**. Algumas dessas alterações são feitas nos campos **DEBUG** e **ALLOWED_HOSTS**. Após as alterações, as nossas pastas estáticas dentro dos aplicativos não ficaram mais visíveis para o Django, mas ficarão a partir da nova pasta criada.

Urls Dinâmicas

Links dinâmicos são aqueles que possuem uma página em comum, mas tem seu conteúdo alterado de acordo com o que é enviado pela url.

Podemos ver isso nas páginas de documentação do Python, por exemplo.

Alguns exemplos:

- https://docs.python.org/pt-br/3.11/tutorial/index.html: essa página contém especificamente a documentação do Python na sua versão 3.11;
- https://docs.python.org/pt-br/3.8/tutorial/index.html: já essa página contém especificamente a documentação do Python na sua versão 3.8;

Para atingir isso, temos que ter uma página padrão que pode aceitar diferentes valores na url e assim alterar seu conteúdo, mas sem mudar a estrutura geral.

Agora, para o nosso projeto, vamos criar uma nova página chamada documentacao.html, onde será a porta de entrada da documentação de todas as versões do C++, que é um dos aplicativos que temos até então.

Veja como ele vai ficar:

C:\Users\gutoh\curso\linguagem_cpp\templates\linguagem_cpp\paginas\documentacao.html

```
16 {% load static %}
15 <!DOCTYPE html>
14 <html lang="en">
13     {% include 'linguagem_cpp/parciais/head.html' %}
12 <body>
11     {% include 'global/titulo.html' %}
10     {% include 'linguagem_cpp/parciais/titulo.html' %}
 9     <h2>sou a página da Documentação</h2>
 8     {% include 'linguagem_cpp/parciais/menu.html' %}
 7     <ul>
 6         <li><a href="#">Versão 1.0</a></li>
 5         <li><a href="#">Versão 2.0</a></li>
 4         <li><a href="#">Versão 3.0</a></li>
 3     </ul>
 2 </body>
 1 </html>
17
```

PS.: lembre de também realizar as alterações nos arquivos menu.html, views.py e urls.py.

Agora, vamos criar a página que será responsável por mostrar os dados de cada versão.

Página temporária para mostrar as versões:

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>
</body>
</html>

```

Repare que a função **view_versao** tem um parâmetro novo na sua definição (o id):

C:\Users\gutoh\curso\linguagem_cpp\views.py

```
from django.shortcuts import render


def view_inicio(request):
    dicionario = {'nome': 'Tom', 'sobrenome': 'Cruise'}
    return render(request, 'linguagem_cpp/paginas/inicio.html', context=dicionario)

def view_sobre(request):
    return render(request, 'linguagem_cpp/paginas/sobre.html')

def view_contato(request):
    return render(request, 'linguagem_cpp/paginas/contato.html')

def view_documentacao(request):
    return render(request, 'linguagem_cpp/paginas/documentacao.html')

def view_versao(request, id):
    return render(request, 'linguagem_cpp/paginas/versao.html')

```

Isso é necessário, pois ela é quem vai receber o valor extra enviado pela url.

Veja o arquivo das urls como ficará:

C:\Users\gutoh\curso\linguagem_cpp\urls.py

```
from django.urls import path
from . import views

app_name = 'cpp'

urlpatterns = [
    path('', views.view_inicio, name='inicio'),
    path('sobre/', views.view_sobre, name='sobre'),
    path('contato/', views.view_contato, name='contato'),
    path('documentacao/', views.view_documentacao, name='documentacao'),
    path('documentacao/versao/<int:id>/', views.view_versao, name='versao'),
]

```

Veja que a página da versão é uma subpágina da documentação e, além disso, ela também tem um parâmetro logo depois, definido por **<int:id>**. O **int** define o tipo da variável que será enviada, já o **id** é o nome variável que será enviado para a função. Ela tem que ter o mesmo nome do parâmetro da função.

Agora, se tentarmos carregar a página http://127.0.0.1:8000/cpp/documentacao/versao/1/, veremos que ela carrega normalmente, independente do valor enviado na url. Nesse caso, enviamos o valor 1.

Nesse caso, estamos enviando explicitamente um inteiro, mas podemos enviar também os tipos str, slug, uuid. Veremos mais sobre eles adiante (mais conteúdo pode ser encontrado no link https://docs.djangoproject.com/en/4.2/topics/http/urls/).

Tag for

Agora, antes de continuarmos, vamos ver como usar o loop for em arquivos Django HTML.

Para isso, vamos criar vários dicionários em uma lista para ser enviado para nossa página documentacao.html, para que os links sejam criados de maneira dinâmica. Veja como vai ficar nosso arquivo **/linguagem_cpp/views.py**:

C:\Users\gutoh\curso\linguagem_cpp\views.py

```python
from django.shortcuts import render

VERSOES = [
    {'titulo':'Versão 1.0', 'versao':1, 'descrição':'Sou a versão 1.0 do C++'},
    {'titulo':'Versão 2.0', 'versao':2, 'descrição':'Sou a versão 2.0 do C++'},
    {'titulo':'Versão 3.0', 'versao':3, 'descrição':'Sou a versão 3.0 do C++'},
]

def view_inicio(request):
    dicionario = {'nome': 'Tom', 'sobrenome': 'Cruise'}
    return render(request, 'linguagem_cpp/paginas/inicio.html', context=dicionario)

def view_sobre(request):
    return render(request, 'linguagem_cpp/paginas/sobre.html')

def view_contato(request):
    return render(request, 'linguagem_cpp/paginas/contato.html')

def view_documentacao(request):
    dicionario = {'vers': VERSOES}

    return render(request, 'linguagem_cpp/paginas/documentacao.html', context=dicionario)

def view_versao(request, id):
    return render(request, 'linguagem_cpp/paginas/versao.html')

```

Acima, temos uma lista criada como constante e depois é inserida em um dicionário na função view_documentacao. Depois, é enviada para a página documentacao.html usando o parâmetro **context**, que irá, então, exibir em uma lista dinâmica, como o HTML abaixo:

C:\Users\gutoh\curso\linguagem_cpp\templates\linguagem_cpp\paginas\documentacao.html

```
<!DOCTYPE html>
<html lang="en">
    {% include 'linguagem_cpp/parciais/head.html' %}
<body>
    {% include 'global/titulo.html' %}
    {% include 'linguagem_cpp/parciais/titulo.html' %}
    <h2>sou a página da Documentação</h2>
    {% include 'linguagem_cpp/parciais/menu.html' %}
    <ul>
        {% for ver in vers %}
        <li><a href="versao/{{ver.versao}}">{{ver.titulo}}</a></li>
        {% endfor %}
    </ul>
</body>
</html>

```

Repare que, para a tag for, temos que inserir endfor de forma a determinar onde que o for irá terminar. Afinal, diferente do código em Python puro que os blocos de código são definidos pela indentação, no Django usamos em delimitador manual para o fim das tags Django como for, if, etc.

Uma vez que temos criadas os links para as versões, temos que adaptar a função **view_versao** para que possa enviar a versão correta para a página HTML versão.html.

Para isso, usaremos como critério de busca o valor recebido pela URL na variável id.

Veja como ela vai ficar (abaixo está sendo mostrado apenas o trecho da função):

C:\Users\gutoh\curso\linguagem_cpp\views.py

```
def view_versao(request, id):
    resultado = {}
    for item in VERSOES:
        if item['versao'] == id:
            resultado = item
            break
    return render(request, 'linguagem_cpp/paginas/versao.html', context=resultado)

```

Veja como ficará o arquivo versao.html para poder exibir as informações corretas de cada versão:

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>
    <h3>{{titulo}}</h3>
    <p>{{descricao}}</p>
    <a href="{% url 'cpp:documentacao' %}">voltar</a>
</body>
</html>

```

Agora, a cada versão que carregarmos na página versao.html irá mostrar um conteúdo diferente.

Exercícios para Praticar

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

2. Desafio: agora que sabe como a tag for do Django funciona, veja se consegue usar a tag if do Django para que seja exibida uma mensagem de versão incorreta, caso seja digitada uma versão inexistente na url.

3. Continue a desenvolver os sites da aula passada, e adicione um aplicativo principal para gerenciar todos os demais. De modo que se possa navegar entre todos eles sem digitar nada na url.