
FIAP

Trabalhando com Django

Parte II

Class Based Views

Além das funções, é possível utilizar no Django um recurso chamado **Class Based View**. Como o nome sugere, ao invés de utilizar funções para retornar uma requisição, podemos utilizar uma classe.
Vamos pegar como exemplo a função `index`, no arquivo `views.py`, onde é exibida a lista de livros.
Ao invés de uma função, vamos utilizar uma classe.
Para isso, importe o seguinte módulo:

```
from django.views import generic
```

Então, mude sua função `index` para uma classe da seguinte forma:

```
class ListView(generic.ListView):
    template_name = "gestao/list.html"
    context_object_name = "livros"

    def get_queryset(self):
        return Livro.objects.all()
```

A propriedade `template_name` determina qual **template** será utilizado para renderizar a página.
Já a propriedade `context_object_name` determina qual será a variável poderá ser utilizada no **template**

Class Based Views

Por fim, mude o arquivo `urls.py`, na rota onde utiliza a função `index` para utilizar sua classe. Ou seja, onde tem:

```
path("", views.index, name="index"),
```

```
path("", views.ListView.as_view(), name="index"),
```

Pronto! Dessa forma você esta utilizando uma **Class Based View**

Class Based Views

O Django encoraja o uso das **Class Based View**, por diminuir a quantidade de código que pode ser utilizada.
Vamos pegar agora a função que visualiza os detalhes de um livro. Ela está escrita da seguinte forma:

```
def view(request, livro_id):
    livro = get_object_or_404(Livro, pk=livro_id)
    return render(request, "gestao/view.html", {"livro": livro})
```

Utiliando uma **Class Based View**, fica da seguinte forma:

```
class DetailView(generic.DetailView):
    model = Livro
    template_name = "gestao/view.html"
```

Ao herdar a classe `DetailView`, já é abstraido a parte de utilizar a função `get_object_or_404`.
Na propriedade `model` você informa qual modelo será utilizado para fazer a busca dos dados

Class Based Views

Por fim, mude a rota no arquivo `urls.py` para ficar da seguinte forma:

```
path("livro/<int:pk>/", views.DetailView.as_view(), name="view"),
```

Note que a rota recebia um paramêtro chamado `livro_id`, que agora foi renomeado para `pk`.
Por que? Assim o `DetailView` automaticamente já recebe esse argumento e faz a busca da mesma forma que era feito quando você tinha a função.

Class Based Views

Vamos agora converter a função onde salva os dados do livro para uma Class Based View
Para isso criaremos uma classe que herderá uma `CreateView`:

```
class CreateView(generic.CreateView):
    model = Livro
    template_name = "gestao/form.html"
    success_url = reverse_lazy("gestao:index")
    fields = ["titulo", "paginas"]
```

A propriedade `success_url` determina para qual URL a aplicação deverá ser redirecionada caso tenha salvo os dados com sucesso. Repare que ele utiliza a função `reverse_lazy` ao invés da função `reverse`. Então não esqueça de alterar onde essa função é importada para importar a `reverse_lazy`
A propriedade `fields` determina quais campos serão utilizados na hora que for criar um modelo. O que classe vai fazer é procurar na requisição os campos `titulo` e `paginas`, criar uma instância do `model` informado e atribuir esses mesmos campos. Ou seja, o modelo precisa ter os campos `titulo` e `paginas` também!

Class Based Views

Iremos mudar as rotas para utilizar a classe recém criada:
Onde existiam as duas URLs:

```
path("salvar/", views.save, name="save"),
path("cadastrar/", views.form, name="form"),
```

Vamos deixar apenas uma rota

```
path("salvar/", views.CreateView.as_view(), name="save"),
```

E por último, no arquivo `gestao/form.html` apenas altere para utilizar a rota `gestao:save`

```
<form action="{% url 'gestao:save' %}" method="post">
```

Class Based Views

Pronto! Ao final desse processo toda sua aplicação deverá rodar da mesma forma, só que agora utilizando **Class Based Views**

Veja como o `views.py` fica no final:

```
from django.urls import reverse_lazy
from django.views import generic

from .models import Livro


# Create your views here.
class ListView(generic.ListView):
    template_name = "gestao/list.html"
    context_object_name = "livros"

    def get_queryset(self):
        return Livro.objects.all()


class DetailView(generic.DetailView):
    model = Livro
    template_name = "gestao/view.html"


class CreateView(generic.CreateView):
    model = Livro
    template_name = "gestao/form.html"
    success_url = reverse_lazy("gestao:index")
    fields = ["titulo", "paginas"]
```

Arquivos estáticos

Agora, que tal dar uma estilizada na aplicação? Isso é possível atráves do recurso de arquivos estáticos que o **Django** possui.
Para isso, dentro da pasta da aplicação `gestao` crie uma pasta chamada `static` e dentro dessa pasta, novamente a pasta `gestao`. Dentro dela, crie um arquivo chamado `list.css`

Arquivos estáticos

Agora, deixe o arquivo `list.css` com o seguinte conteúdo:

```css
body {
    display: flex;
    justify-content: center;
    padding: 50px;
    background-color: #66CCFF;
    font-family: sans-serif;
}

ul {
    list-style: none;
    width: 600px;
    padding: 0;
}

ul li {
    color: #333;
    background-color: rgba(255, 255, 255, .5);
    padding: 15px;
    margin-bottom: 15px;
    border-radius: 5px;
}

ul li::after {
    border-style: solid;
    border-width: 0.25em 0.25em 0 0;
    content: '';
    height: 0.45em;
    position: relative;
    top: 0.15em;
    transform: rotate(-45deg);
    vertical-align: top;
    width: 0.45em;
    transform: rotate(45deg);
    float: right;
}

ul a {
    color: #595959;
    text-decoration: none;
}
```

Arquivos estáticos

E edite seu arquivo `gestao/list.html` para ficar da seguinte forma:

```
{% load static %}
<link rel="stylesheet" href="{% static 'list.css' %}" />

{% if livros %}
<ul>
  {% for livro in livros %}
  <a href="{% url 'gestao:view' livro.id %}">
    <li>{{ livro.titulo }}</li>
  </a>
  {% endfor %}
</ul>
{% else %}
<p>Nenhum livro cadastrado.</p>
{% endif %}
```

Na primeira linha do arquivo, importamos o módulo `static` que existe no sistema de **template tags** do **Django**.

Seria o equivalente ao `import`

Na segunda linha, usamos o **template tag** `static`, que vai buscar na estrutura do projeto um arquivo chamado `list.css` e montar sua URL

Arquivos estáticos

Toda vez que precisar criar arquivos para serem exibidos na sua aplicação, é obrigatório que exista a pasta `templates` para ter arquivos HTML e a pasta `static` para colocar arquivos Javascript, CSS e afins.
E isso se aplica globalmente, ou seja, não apenas sua aplicação, mas seu projeto todo pode ter uma pasta `templates` e `static`
Primeiro na raiz do seu projeto (que é onde se encontra o arquivo `manage.py`), crie as duas pastas: `templates` e `static`

Arquivos estáticos

Agora, no arquivo `settings.py` procure a constante `TEMPLATES` e deixe ela da seguinte forma:

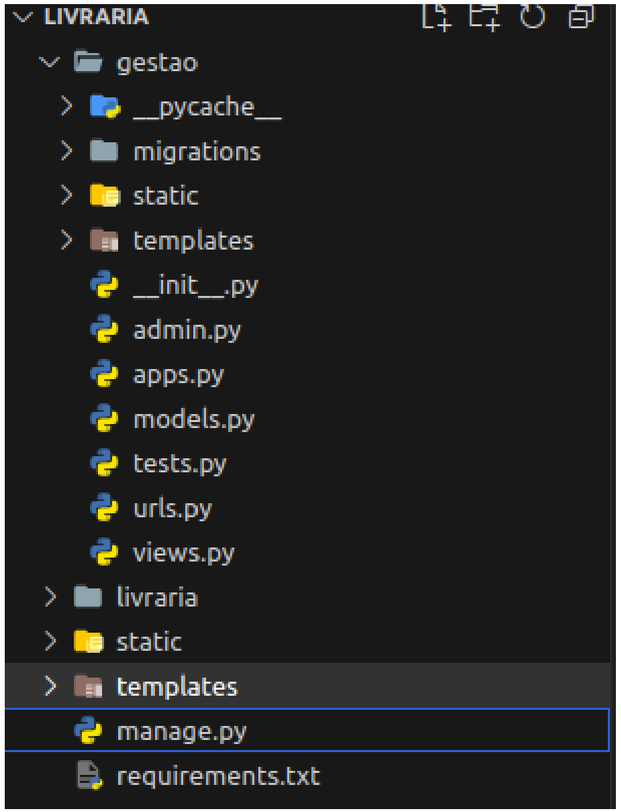
```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]
```

No mesmo arquivo, procure a constante `STATIC_URL` e depois dela, declare a variável `STATICFILES_DIRS` da seguinte forma:

```
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Arquivos estáticos

A estrutura do seu projeto deve ficar assim:



Arquivos estáticos

Dentro da pasta `templates` do projeto, vamos criar o arquivo `base.html` com o seguinte conteúdo:

```
{% load static %}
<link rel="stylesheet" href="{% static 'base.css' %}" />

{% block static %}
{% endblock static %}

<h1>Gestão de livros</h1>
{% block content %}
{% endblock content %}
```

A **template tag** `block` determina uma área que você pode inserir conteúdo atráves de outras páginas, utilizando um recuro quase que similar a herança.
Nesse arquivo, estamos criando duas áreas: a `static` e a `content`
Veremos mais adiante como elas serão utilizadas

Arquivos estáticos

Na pasta `static`, vamos criar o arquivo `base.css` e colocar o seguinte conteúdo:

```
body {
    display: flex;
    align-items: center;
    flex-direction: column;
    padding: 50px;
    background-color: #66CCFF;
    font-family: sans-serif;
}
```

Arquivos estáticos

Vamos editar agora o arquivo `list.html` e deixar com o seguinte conteúdo:

```
{% extends 'base.html' %}
{% load static %}

{% block static %}
    <link rel="stylesheet" href="{% static 'gestao/list.css' %}" />
{% endblock static %}

{% block content %}
    {% if livros %}
        <ul>
        {% for livro in livros %}
        <a href="{% url 'gestao:view' livro.id %}">
            <li>{{ livro.titulo }}</li>
        </a>
        {% endfor %}
        </ul>
        {% else %}
        <p>Nenhum livro cadastrado.</p>
    {% endif %}
{% endblock content %}
```

Arquivos estáticos

Na primeira linha, utilizamos a **template tag** `extends` que vai herdar todo o conteúdo do arquivo informado, que no caso é o `base.html`
Como podemos notar, utilizando o mesmo `block` que foi utilizado no `base.html` para definir qual o conteúdo será exibido dentro dele.
No exemplo, é definido um conteúdo para o bloco `static` e mais um conteúdo para o bloco `content`

Utilizando bancos de dados

Até agora, todo o projeto está rodando utilizando o SQLite, porém ele é recomendado apenas para problemas pontuais.
Um projeto de verdade utiliza um banco de dados mais robusto. Vamos utilizar o MongoDB e o PostgreSQL na nossa aplicação

Djongo

Para utilizar o **MongoDB** no **Django**, existe a biblioteca chamada djongo.
Para utiliza-la, basta rodar o comando:

```
pip install djongo
```

> ⚠️ Vale ressaltar que o **Django NÃO** suporta oficialmente o **MongoDB**

Djongo

No `settings.py` deixe a seguinte configuração:

```
DATABASES = {
    "default": {
        "ENGINE": "djongo",
        "NAME": "livraria",
    }
}
```

Após essa configuração, basta rodar o comando :

`python manage.py migrate`

> ⚠️ Caso você tenha problemas com esse comando, instale as seguintes bibliotecas:
>
> `pip install pymongo==3.12.1`
>
> `pip install pytz`

Djongo

Com isso, sua aplicação rodará normalmente utilizando o **MongoDB** como base de dados

PostgreSQL

Para utilizar o PostgreSQL é basicamente seguir os mesmo procedimentos, apenas diferenciando quais bibliotecas serão utilizadas.
No caso do PostgreSQL, devemos instalar a psycopg.
Para instala-la, rode o comando:

```
pip install psycopg2-binary
```

PostgreSQL

Depois de instalada a biblioteca, deixe seu `settings.py` da seguinte forma:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "livraria",
        "HOST": "localhost",
        "USER": "usuario",
        "PASSWORD": "senha"
    }
}
```

Então rode o comando para criar a estrutura de tabelas:

`python manage.py migrate`

> ⚠️ No caso do PostgreSQL, esteja ciente de que o banco de dados já deve existir

PostgreSQL

Pronto! Com isso seu projeto já está pronto para rodar usando o banco de dados PostgreSQL!


FIAP