PYTHON ACADEMY

APRENDA A UTILIZAR AS CLASS BASED VIEWS (CBV) DO DJANGO

Aprenda a desenvolver views eficientes em Django com Class Based Views. Nesse ebook você vai aprender a utilizar as Class Based Views do Django para construir views mais especializadas, legíveis e organizadas, com menos código.

PYTHONACADEMY.COM.BR

Salve salve Pythonista Web!

Se você é um desenvolvedor de Django em busca de uma maneira mais eficiente e organizada de desenvolver suas views, as **Class Based Views** podem ser exatamente o que você precisa.

As Class Based Views são um recurso poderoso, que permitem uma abordagem Orientada a Objetos para a construção de suas Views.

Isso significa que, ao invés de criar funções separadas para cada view em seu projeto, você pode utilizar classes para definir suas views, tornando seu código mais legível e fácil de manter.

Com as Class Based Views, você tem acesso a diversas funcionalidades que não estão disponíveis nas views baseadas em funções, como herança, *mixins* e atributos de classe.

Neste artigo, vamos explorar as vantagens das Class Based Views e como utilizá-las em seu projeto Django.

Então já prepara o café, e vamos nessa!

Introdução às Class Based Views do Django

Neste artigo, nós vamos trabalhar em cima do nosso ***HelloWorld***, um projeto de gerenciamento de funcionários.

Caso você ainda não o conheça, sugiro que visite o nosso primeiro post sobre a Camada Model do Django, onde desenvolvemos a base do nosso projeto.

As Class Based Views - ou CBVs, para os íntimos - que vamos estudar hoje, nos ajudam a agilizar o desenvolvimento da nossa aplicação.

Temos basicamente **duas formas** para utilizá-las.

Primeiro, podemos usá-las diretamente no nosso **URLConf** (no arquivo `urls.py`):

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

urlpatterns = [
    path('', TemplateView.as_view(template_name="index.html")),
]
```

E a segunda maneira, sendo a mais utilizada e poderosa, é herdando da *View* desejada e sobrescrevendo os atributos e métodos na subclasse.

Criação de Templates HTML com a CBV `TemplateView`

Por exemplo, se você precisar apenas renderizar uma página HTML, podemos utilizar a `TemplateView` para isso (veja a documentação aqui da TemplateView).

Para isso é necessário 3 passos: - Arquivo contendo o código HTML que deve ser renderizado pela `TemplateView` - A `TemplateView`, propriamente dita. - A configuração de rotas configurando qual URL deve ser servida por esta `TemplateView`

Portanto, se criarmos um arquivo HTML (por exemplo):

```html
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Primeira Template</title>
</head>
<body>
    <h1>Esse template será renderizado pelo Django</h1>
</body>
</html>
```

Com a seguinte `TemplateView`:

```python
from django.views.generic import TemplateView

class IndexTemplateView(TemplateView):
  template_name = "index.html"
```

E a seguinte configuração de rotas:

```python
from django.urls import path
from helloworld.views import IndexTemplateView

urlpatterns = [
    path('', IndexTemplateView.as_view()),
]
```

Teremos um template sendo renderizado no caminho raiz da sua aplicação (geralmente em `http://localhost:8000`).

Listando objetos com a Class Based View `ListView`

Já para listar registros de uma tabela do Banco de Dados, podemos utilizar a `ListView` (documentação).

Aqui, vamos utilizar a entidade `Funcionario` do nosso projeto HelloWorld.

Nela, nós configuramos o *Model* que deve ser buscado (`Funcionario` no nosso caso), e ela automaticamente faz a busca por todos os registros presentes no banco de dados da entidade informada.

Então, a *View* ficará da seguinte forma:

```
from django.views.generic.list import ListView
from helloworld.models import Funcionario

class FuncionarioListView(ListView):
  template_name = "lista.html"
  model = Funcionario
  context_object_name = "funcionarios"
```

Essa *View* vai expor o objeto declarado em `context_object_name` no template para iteração.

Já a configuração de rotas:

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

urlpatterns = [
    path('funcionarios/', FuncionarioListView.as_view()),
]
```

E o template HTML pode iterar sobre a variável `funcionarios` da seguinte forma, criando uma Tabela HTML com uma linha por Funcionário:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Primeira Template</title>
</head>
<body>
    <h1>Esse template será renderizado pelo Django</h1>

    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Nome</th>
                <th>Sobrenome</th>
                <th>Remuneração</th>
            </tr>
        </thead>
        <tbody>
            {% raw %}{% for funcionario in funcionarios %}
                <tr>
                    <td>{{ funcionario.id }}</td>
                    <td>{{ funcionario.nome }}</td>
                    <td>{{ funcionario.sobrenome }}</td>
                    <td>R$ {{ funcionario.remuneracao }}</td>
                </tr>
            {% endfor %}{% endraw %}
        </tbody>
    </table>
</body>
</html>
```

E o resultado é uma página **lista.html** contendo a lista de todos os funcionários cadastrados.

> *Dica: Eu geralmente coloco o nome da View como sendo o Model com a CBV base. Por exemplo: se eu fosse criar uma view para listar todos os Cursos cadastrados, eu daria o nome de: Model="Curso" + CBV="ListView" =* `Curso-ListView` *.*

Atualizando objetos com a Class Based View `UpdateView`

Para atualizar registros de uma tabela do Banco de Dados, utilizamos a `Update-View` do Django (documentação)

Com ela, configuramos qual o *Model*, os campos que estarão disponíveis para atualização e qual o nome do template, dessa forma.

```
from django.views.generic.edit import UpdateView
from helloworld.models import Funcionario

class FuncionarioUpdateView(UpdateView):
    template_name = "atualiza.html"
    model = Funcionario
    fields = [
      'nome',
      'sobrenome' ,
      'cpf',
      'tempo_de_servico',
      'remuneracao'
    ]
```

> *Dica! Ao invés de todos os campos em `fields` em formato de lista de strings, podemos utilizar `fields = '__all__'` que o Django irá buscar todos os campos para você!*

*** Mas… E de onde o Django vai pegar o `id` do objeto a ser buscado?***

O Django precisa ser informado do `id` ou `slug` para poder buscar o objeto correto a ser atualizado.

Podemos fazer isso de duas formas:

Primeiro, na configuração de rotas (urls.py).

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

urlpatterns = [
    # Utilizando o {id} para buscar o objeto
    path('funcionario/<id>', FuncionarioUpdateView.as_view()),

    # Utilizando o {slug} para buscar o objeto
    path('funcionario/<slug>', FuncionarioUpdateView.as_view()),
]
```

*** Mas o que é slug?***

Slug é uma forma de gerar URLs mais legíveis a partir de dados já existentes.

Exemplo: podemos criar um campo *slug* utilizando o campo `nome` do funcionário. Dessa forma, as URLs ficariam assim:

👉 `/funcionario/vinicius-ramos`

e não assim (utilizando o *id* na URL):

👉 `/funcionario/175`

No campo *slug*, **todos** os caracteres são transformados em minúsculos e os espaços são transformados em hífens.

A segunda forma de buscar o objeto que estará disponível na tela de atualização é utilizando (ou **sobrescrevendo**) o método `get_object()` da classe pai `UpdateView`.

A documentação desse método traz (traduzido):

> *"Retorna o objeto que a View irá mostrar. Requer `self.queryset` e um argumento `pk` ou `slug` no URLConf. Subclasses podem sobrescrever esse método e retornar qualquer objeto."*

Ou seja, o Django nos dá total liberdade de utilizarmos a **convenção** (parâmetros passados pela URLConf) ou a **configuração** (sobrescrevendo o método `get_object()`).

Basicamente, o método `get_object()` deve pegar o `id` ou `slug` da url e realizar a busca no banco de dados até encontrar aquele `id`:

```python
from django.views.generic.edit import UpdateView
from helloworld.models import Funcionario

class FuncionarioUpdateView(UpdateView):
    template_name = "atualiza.html"
    model = Funcionario
    fields = '__all__'
    context_object_name = 'funcionario'

    def get_object(self, queryset=None):
      funcionario = None

      # Se você utilizar o debug, verá que os
      # campos {pk} e {slug} estão presente em self.kwargs
      id = self.kwargs.get(self.pk_url_kwarg)
      slug = self.kwargs.get(self.slug_url_kwarg)

      if id is not None:
        # Busca o funcionario apartir do id
        funcionario = Funcionario.objects.filter(id=id).first()

      elif slug is not None:
        # Pega o campo slug do Model
        campo_slug = self.get_slug_field()

        # Busca o funcionario apartir do slug
        funcionario = Funcionario.objects.filter(**{campo_slug:
        slug}).first()

      # Retorna o objeto encontrado
      return funcionario
```

Dessa forma, os dados do funcionário de `id` ou `slug` igual ao que foi passado na URL estarão disponíveis para visualização no *template* `atualiza.html` utilizando o objeto `funcionario`!

> *No caso geral, eu prefiro utilizar a convenção (configuração no URLConf).*

> 💡 *Estou construindo o **DevBook**, uma plataforma que usa IA para criar ebooks técnicos — com código formatado e exportação em PDF. Depois de ler, dá uma passada lá!*

Deletando objetos com a Class Based View `DeleteView`

Para deletar funcionários, utilizaremos a `DeleteView` (documentação).

Sua configuração é similar à `UpdateView`: nós devemos informar via URLConf ou `get_object()` qual o objeto que queremos excluir.

Precisamos configurar:

- O template que será renderizado.
- O model associado à essa view.
- O nome do objeto que estará disponível no template (para confirmar ao usuário, por exemplo, o nome do funcionário que será excluído).
- A URL de retorno, caso haja sucesso na deleção do Funcionário.

Com isso, a view pode ser codificada da seguinte forma:

```
class FuncionarioDeleteView(DeleteView):
  template_name = "exclui.html"
  model = Funcionario
  context_object_name = 'funcionario'
  success_url = reverse_lazy("website:lista_funcionarios")
```

Assim como na `UpdateView`, fazemos a configuração do `id` para ser buscado no URLConf, da seguinte forma:

```
urlpatterns = [
    path('funcionario/excluir/<pk>', FuncionarioDeleteView.as_view()),
]
```

Assim, precisamos apenas fazer um *template* de confirmação da exclusão do funcionário.

Podemos fazer da seguinte forma:

```
<form method="post">
    {% raw %}{% csrf_token %}{% endraw %}

    Você tem certeza que quer excluir o funcionário <b>{% raw %}{{ fun-
        cionario.nome }}{% endraw %}</b>? <br><br>

    <button type="button">
        <a href="{% raw %}{% url 'lista_funcionarios' %}{% endraw
        %}">Cancelar</a>
    </button>
    <button>Excluir</button>
</form>
```

Algumas observações:

- Lembra do atributo `context_object_name`? Olha ele presente lá na quarta linha!
- A tag do Django `{% raw %}{% csrf_token %}{% endraw %}` é obrigatório em todos os *forms* pois está relacionado à proteção que o Django provê ao **CSRF - Cross Site Request Forgery** (tipo de ataque malicioso - saiba mais aqui).
- Não se preocupe com a sintaxe do *template*! Veremos mais sobre ele em outro post!!!

Criando objetos com a Class Based View `CreateView`

A *View* para criação de novos registros em nosso banco de dados é bem simples!

Aqui precisamos apenas mostrar para o Django o *model*, dizer o nome do *template*, a classe do Formulário e a URL de retorno - caso haja sucesso na inclusão do Funcionário.

Podemos fazer isso da seguinte forma:

```
from helloworld.models import Funcionario
from helloworld.forms import InsereFuncionarioForm
from django.views.generic import CreateView
from django.urls import reverse_lazy

class FuncionarioCreateView(CreateView):
  template_name = "cria.html"
  model = Funcionario
  form_class = InsereFuncionarioForm
  success_url = reverse_lazy("website:lista_funcionarios")
```

A função `reverse_lazy()` traduz a *View* em URL.

No nosso caso, queremos que quando haja a inclusão do Funcionário, sejamos re-direcionados para a página de listagem, para podermos conferir que o Funcionário foi realmente adicionado.

E a configuração da rota no arquivo urls.py fica assim:

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

urlpatterns = [
    path('funcionario/cadastrar/', FuncionarioCreateView.as_view()),
]
```

Com isso, estará disponível no *template* configurado (`cria.html`, no nosso caso), um objeto `form` contendo o formulário para criação do novo funcionário.

Podemos mostrar o formulário de duas formas.

A **primeira** mostra o formulário inteiro sem formatação e da forma como o Django nos entrega.

Podemos mostrá-lo no nosso template da seguinte forma:

```
<form method="post">
  {% raw %}{% csrf_token %}{% endraw %}

  {% raw %}{% endraw %}

  <button type="submit">Cadastrar</button>
</form>
```

Uma observação: apesar de ser um `Form`, sua renderização não contém as tags `<form></form>` - cabendo a nós incluí-los no *template*.

Já a **segunda**, é mais trabalhosa, pois temos de renderizar campo a campo no *template*. Porém, nos dá um nível maior de customização.

Podemos renderizar cada campo do form dessa forma:

```
<form method="post">
  {% raw %}{% csrf_token %}{% endraw %}

  <label for="{% raw %}{{ form.nome.id_for_label }}{% endraw %}">Nome</
        label>
  {% raw %}{{ form.nome }}{% endraw %}

  <label for="{% raw %}{{ form.sobrenome.id_for_label }}{% endraw
        %}">Sobrenome</label>
  {% raw %}{{ form.sobrenome }}{% endraw %}

  <label for="{% raw %}{{ form.cpf.id_for_label }}{% endraw %}">CPF</
        label>
  {% raw %}{{ form.cpf }}{% endraw %}

  <label for="{% raw %}{{ form.tempo_de_servico.id_for_label }}{% en-
        draw %}">Tempo de Serviço</label>
  {% raw %}{{ form.tempo_de_servico }}{% endraw %}

  <label for="{% raw %}{{ form.remuneracao.id_for_label }}{% endraw
        %}">Remuneração</label>
  {% raw %}{{ form.remuneracao }}{% endraw %}

  <button type="submit">Cadastrar</button>
</form>
```

E assim, temos:

- `{% raw %}{{ form.campo.id_for_label }}{% endraw %}` traz o `id` da tag `<input ...>` para adicionar à tag `<label></label>`.
- Utilizamos o `{% raw %}{{ form.campo }}{% endraw %}` para renderizar um campo do formulário, e não ele inteiro.

Conclusão

É isso galera! 👋

Chegamos ao final de mais um post em nosso Blog!

Você aprendeu sobre as Class Based Views (ou CBV) para tratar as requisições no Django e como utilizar as principais Views que o Django nos fornece.

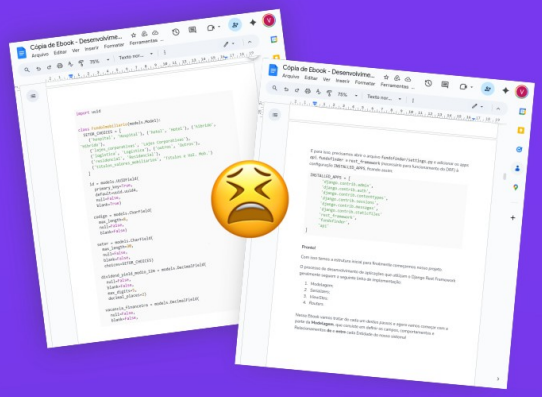
Fique por dentro que ainda tem muito mais conteúdos para vocês!!!

Não se esqueça de conferir!

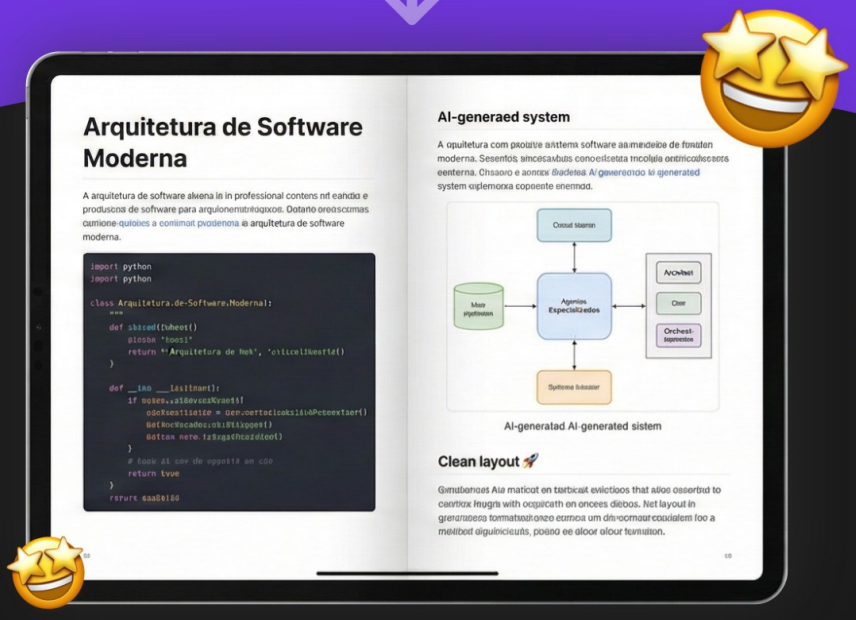
DevBook

Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código**!



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS