Programação de Sistemas para Internet

Prof. Diego Cirilo

Aula 17: Class-Based Views (CBV)

Function-Based Views (FBV)

- Até agora usamos apenas FBVs;
- Views como funções que recebem `request` e retornam `response`;

```
def lista_produtos(request):
    produtos = Produto.objects.all()
    return render(request, 'produtos/lista.html', {'produtos': produtos})
```

- Simples e diretas;
- Funciona bem para todos os casos.

Problema com FBVs

- Muitas views seguem o mesmo padrão;
- Listar objetos, exibir detalhes, criar, editar, deletar...
- Repetimos muito código entre views similares;
- Difícil reaproveitar lógica comum.

Exemplo: Views de Listagem

```
def lista_produtos(request):
    produtos = Produto.objects.all()
    return render(request, 'lista.html', {'produtos': produtos})

def lista_categorias(request):
    categorias = Categoria.objects.all()
    return render(request, 'lista.html', {'categorias': categorias})

def lista_pedidos(request):
    pedidos = Pedido.objects.all()
    return render(request, 'lista.html', {'pedidos': pedidos})
```

- Mesmo padrão, código repetido!

Class-Based Views (CBV)

- Views como classes ao invés de funções;
- Permitem herança e reutilização de código;
- Django oferece views genéricas prontas;
- Menos código para operações comuns.

CBV Básica

```python
from django.views import View

class MinhaView(View):
    def get(self, request):
        return render(request, 'pagina.html')

    def post(self, request):
        # processa formulário
        return redirect('sucesso')
```

- Métodos separados para cada verbo HTTP;
- `get()`, `post()`, `put()`, `delete()`, etc.

Registrando no urls.py

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

urlpatterns = [
    # FBV
    path('fbv/', minha_funcao, name='fbv'),

    # CBV - precisa do .as_view()
    path('cbv/', MinhaView.as_view(), name='cbv'),
]
```

- O método `.as_view()` converte a classe em uma view.

FBV vs CBV Simples

```
# FBV
def minha_view(request):
    if request.method == 'POST':
        # processa POST
        return redirect('sucesso')
    return render(request, 'pagina.html')

# CBV
class MinhaView(View):
    def get(self, request):
        return render(request, 'pagina.html')

    def post(self, request):
        return redirect('sucesso')
```

Views Genéricas

- Django oferece classes prontas para operações comuns;
- Basta configurar alguns atributos;
- Muito menos código!
- Principais:
  - `ListView` - listar objetos
  - `DetailView` - exibir um objeto
  - `CreateView` - criar objeto
  - `UpdateView` - editar objeto
  - `DeleteView` - deletar objeto

Customização por Herança

- CBVs usam **herança** para customização;
- Duas formas de alterar comportamento:
    i. **Atributos de classe**: configurações simples
    ii. **Reescrita de métodos** (*override*): lógica personalizada
- Essa é a metodologia padrão das CBVs do Django.

Reescrita de Métodos (*Override*)

- Cada CBV tem métodos que podemos reescrever;
- Chamamos `super()` para manter o comportamento original;

```
class ProdutoListView(ListView):
    model = Produto

    def get_context_data(self, **kwargs):
        # Chama o método original da classe pai
        context = super().get_context_data(**kwargs)
        # Adiciona dados extras
        context['titulo'] = 'Meus Produtos'
        return context
```

Por que usar super()?

- `super()` chama o método da classe pai (superclasse);
- Mantém o comportamento padrão da view genérica;
- Sem ele, perdemos funcionalidades importantes!

```
# ERRADO - perde o contexto padrão (object_list, etc)
def get_context_data(self, **kwargs):
    return {'titulo': 'Produtos'}

# CORRETO - mantém o contexto e adiciona mais
def get_context_data(self, **kwargs):
    context = super().get_context_data(**kwargs)
    context['titulo'] = 'Produtos'
    return context
```

Principais Métodos para Reescrever

- `get_queryset()` - filtrar/ordenar objetos
- `get_context_data()` - adicionar variáveis ao template
- `form_valid()` - lógica após validação do form
- `get_success_url()` - URL de redirecionamento dinâmica
- `get_object()` - customizar busca do objeto

ListView

- Lista objetos de um model;

```
from django.views.generic import ListView
from .models import Produto

class ProdutoListView(ListView):
    model = Produto
```

- Busca todos os objetos;
- Renderiza template `produto_list.html`;
- Passa a lista como `object_list`.

ListView - Customizando

```
class ProdutoListView(ListView):
    model = Produto
    template_name = 'produtos/lista.html'  # template customizado
    context_object_name = 'produtos'       # nome da variável
    ordering = ['-data_criacao']           # ordenação
    paginate_by = 10                       # paginação
```

ListView - Template

```
<!-- produtos/lista.html -->
<h1>Produtos</h1>

<ul>
{% for produto in produtos %}
    <li>
        <a href="{% url 'produto_detalhe' produto.pk %}">
            {{ produto.nome }} - R$ {{ produto.preco }}
        </a>
    </li>
{% empty %}
    <li>Nenhum produto cadastrado.</li>
{% endfor %}
</ul>
```

ListView - Filtrando

```
class ProdutoListView(ListView):
    model = Produto
    template_name = 'produtos/lista.html'
    context_object_name = 'produtos'

    def get_queryset(self):
        # Sobrescreve a query padrão
        return Produto.objects.filter(ativo=True)
```

ListView - Contexto Extra

```
class ProdutoListView(ListView):
    model = Produto
    context_object_name = 'produtos'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['categorias'] = Categoria.objects.all()
        context['titulo'] = 'Lista de Produtos'
        return context
```

DetailView

- Exibe detalhes de um objeto;

```
from django.views.generic import DetailView

class ProdutoDetailView(DetailView):
    model = Produto
```

- Busca objeto pelo `pk` ou `slug` da URL;
- Renderiza `produto_detail.html`;
- Passa objeto como `object` ou `produto`.

DetailView - URL

```
urlpatterns = [
    # Usando pk
    path('produto/<int:pk>/', ProdutoDetailView.as_view(), name='produto_detalhe'),

    # Usando slug
    path('produto/<slug:slug>/', ProdutoDetailView.as_view(), name='produto_detalhe'),
]
```

DetailView - Template

```
<!-- produtos/produto_detail.html -->
<h1>{{ produto.nome }}</h1>

<p>{{ produto.descricao }}</p>
<p>Preço: R$ {{ produto.preco }}</p>
<p>Categoria: {{ produto.categoria }}</p>

<a href="{% url 'produto_editar' produto.pk %}">Editar</a>
<a href="{% url 'produto_deletar' produto.pk %}">Deletar</a>
```

DetailView - Customizando

```
class ProdutoDetailView(DetailView):
    model = Produto
    template_name = 'produtos/detalhe.html'
    context_object_name = 'produto'
    slug_field = 'slug'          # campo do model
    slug_url_kwarg = 'slug'      # nome na URL
```

CreateView

- Formulário para criar objetos;

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

class ProdutoCreateView(CreateView):
    model = Produto
    fields = ['nome', 'descricao', 'preco', 'categoria']
    success_url = reverse_lazy('produto_lista')
```

- Gera formulário automaticamente;
- Salva o objeto se válido;
- Redireciona para `success_url`.

Por que reverse_lazy?

- `reverse()` é executado na importação do módulo;
- Nesse momento as URLs podem não estar carregadas;
- `reverse_lazy()` só executa quando necessário;
- **Sempre use `reverse_lazy` em atributos de classe**.

CreateView - Template

```
<!-- produtos/produto_form.html -->
<h1>Novo Produto</h1>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Salvar</button>
</form>

<a href="{% url 'produto_lista' %}">Cancelar</a>
```

- Template padrão: `<model>_form.html`

CreateView - Customizando

```
class ProdutoCreateView(CreateView):
    model = Produto
    fields = ['nome', 'descricao', 'preco', 'categoria']
    template_name = 'produtos/criar.html'
    success_url = reverse_lazy('produto_lista')

    def form_valid(self, form):
        form.instance.criado_por = self.request.user
        return super().form_valid(form)
```

- `form_valid()` é chamado quando o form é válido.

UpdateView

- Formulário para editar objetos;

```
from django.views.generic import UpdateView

class ProdutoUpdateView(UpdateView):
    model = Produto
    fields = ['nome', 'descricao', 'preco', 'categoria']
    success_url = reverse_lazy('produto_lista')
```

- Carrega objeto existente no formulário;
- Usa o mesmo template que CreateView (`produto_form.html`).

UpdateView - Template Compartilhado

```
<!-- produtos/produto_form.html -->
<h1>
    {% if object %}
        Editar {{ object.nome }}
    {% else %}
        Novo Produto
    {% endif %}
</h1>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Salvar</button>
</form>
```

DeleteView

- Confirmação e exclusão de objetos;

```
from django.views.generic import DeleteView

class ProdutoDeleteView(DeleteView):
    model = Produto
    success_url = reverse_lazy('produto_lista')
```

- GET: exibe página de confirmação;
- POST: deleta o objeto;
- Template padrão: `produto_confirm_delete.html`.

DeleteView - Template

```
<!-- produtos/produto_confirm_delete.html -->
<h1>Confirmar Exclusão</h1>

<p>Tem certeza que deseja excluir "{{ object.nome }}"?</p>

<form method="post">
    {% csrf_token %}
    <button type="submit">Sim, excluir</button>
    <a href="{% url 'produto_lista' %}">Cancelar</a>
</form>
```

Redirecionamento Dinâmico

```
class ProdutoUpdateView(UpdateView):
    model = Produto
    fields = ['nome', 'descricao', 'preco']

    def get_success_url(self):
        # Redireciona para o detalhe do objeto editado
        return reverse('produto_detalhe', kwargs={'pk': self.object.pk})
```

- `get_success_url()` para URLs dinâmicas.

Usando Form Class

```
from .forms import ProdutoForm

class ProdutoCreateView(CreateView):
    model = Produto
    form_class = ProdutoForm  # ao invés de fields
    success_url = reverse_lazy('produto_lista')
```

- Use `form_class` para forms personalizados;
- Não pode usar `fields` junto com `form_class`.

Resumo - Templates Padrão

View	Template Padrão
ListView	`<model>_list.html`
DetailView	`<model>_detail.html`
CreateView	`<model>_form.html`
UpdateView	`<model>_form.html`
DeleteView	`<model>_confirm_delete.html`

Resumo - Variáveis de Contexto

View	Variável Padrão
ListView	`object_list`
DetailView	`object`
CreateView	`form`
UpdateView	`object`, `form`
DeleteView	`object`

CRUD Completo - Views

```python
# views.py
from django.views.generic import (
    ListView, DetailView, CreateView, UpdateView, DeleteView
)
from django.urls import reverse_lazy
from .models import Produto

class ProdutoListView(ListView):
    model = Produto
    context_object_name = 'produtos'

class ProdutoDetailView(DetailView):
    model = Produto

class ProdutoCreateView(CreateView):
    model = Produto
    fields = ['nome', 'descricao', 'preco', 'categoria', 'ativo']
    success_url = reverse_lazy('produto_lista')
```

CRUD Completo - Views (cont.)

```
class ProdutoUpdateView(UpdateView):
    model = Produto
    fields = ['nome', 'descricao', 'preco', 'categoria', 'ativo']
    success_url = reverse_lazy('produto_lista')

class ProdutoDeleteView(DeleteView):
    model = Produto
    success_url = reverse_lazy('produto_lista')
```

CRUD Completo - URLs

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

urlpatterns = [
    path('', views.ProdutoListView.as_view(), name='produto_lista'),
    path('<int:pk>/', views.ProdutoDetailView.as_view(), name='produto_detalhe'),
    path('novo/', views.ProdutoCreateView.as_view(), name='produto_criar'),
    path('<int:pk>/editar/', views.ProdutoUpdateView.as_view(), name='produto_editar'),
    path('<int:pk>/deletar/', views.ProdutoDeleteView.as_view(), name='produto_deletar'),
]
```

CRUD - Lista (produto_list.html)

```
<h1>Produtos</h1>
<a href="{% url 'produto_criar' %}">Novo Produto</a>

<table>
    <tr><th>Nome</th><th>Preço</th><th>Ações</th></tr>
    {% for produto in produtos %}
    <tr>
        <td><a href="{% url 'produto_detalhe' produto.pk %}">{{ produto.nome }}</a></td>
        <td>R$ {{ produto.preco }}</td>
        <td>
            <a href="{% url 'produto_editar' produto.pk %}">Editar</a>
            <a href="{% url 'produto_deletar' produto.pk %}">Deletar</a>
        </td>
    </tr>
    {% endfor %}
</table>
```

CRUD - Detalhe (produto_detail.html)

```
<h1>{{ produto.nome }}</h1>

<p><strong>Descrição:</strong> {{ produto.descricao }}</p>
<p><strong>Preço:</strong> R$ {{ produto.preco }}</p>
<p><strong>Categoria:</strong> {{ produto.categoria }}</p>
<p><strong>Status:</strong> {% if produto.ativo %}Ativo{% else %}Inativo{% endif %}</p>

<a href="{% url 'produto_editar' produto.pk %}">Editar</a>
<a href="{% url 'produto_deletar' produto.pk %}">Deletar</a>
<a href="{% url 'produto_lista' %}">Voltar</a>
```

CRUD - Form (produto_form.html)

```
<h1>{% if object %}Editar{% else %}Novo{% endif %} Produto</h1>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Salvar</button>
    <a href="{% url 'produto_lista' %}">Cancelar</a>
</form>
```

CRUD - Delete (produto_confirm_delete.html)

```
<h1>Excluir Produto</h1>

<p>Tem certeza que deseja excluir <strong>{{ object.nome }}</strong>?</p>
<p>Esta ação não pode ser desfeita.</p>

<form method="post">
    {% csrf_token %}
    <button type="submit">Sim, excluir</button>
    <a href="{% url 'produto_lista' %}">Cancelar</a>
</form>
```

Mixins

- Classes que adicionam funcionalidades específicas;
- Não funcionam sozinhas, devem ser combinadas com outras classes;
- Permitem compor comportamentos de forma modular;
- Django oferece vários mixins prontos;
- Conceito importante de POO: herança múltipla controlada.

Como Mixins Funcionam

```
# Mixin adiciona uma funcionalidade
class TituloMixin:
    titulo = 'Página'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['titulo'] = self.titulo
        return context

# Combinamos mixin + view genérica
class ProdutoListView(TituloMixin, ListView):
    model = Produto
    titulo = 'Lista de Produtos'
```

- Mixins vêm **antes** da view na herança!

Ordem dos Mixins

```
# CORRETO: Mixins primeiro, View por último
class MinhaView(MixinA, MixinB, ListView):
    ...

# ERRADO: View não deve vir primeiro
class MinhaView(ListView, MixinA):  # Pode não funcionar!
    ...
```

- Python resolve métodos da esquerda para direita (MRO);
- A view genérica deve ser a última classe base.

Autenticação em CBV

- Proteger views para usuários logados;
- Em FBV usamos o decorator `@login_required`;
- Em CBV usamos o mixin `LoginRequiredMixin`.

LoginRequiredMixin

```
from django.contrib.auth.mixins import LoginRequiredMixin

class ProdutoCreateView(LoginRequiredMixin, CreateView):
    model = Produto
    fields = ['nome', 'preco']
    success_url = reverse_lazy('produto_lista')
```

- Redireciona para login se não autenticado;
- Mixin deve vir **antes** da view genérica.

Configurando LoginRequiredMixin

```
class ProdutoCreateView(LoginRequiredMixin, CreateView):
    model = Produto
    fields = ['nome', 'preco']

    # URL de login (ou configure LOGIN_URL no settings.py)
    login_url = '/login/'

    # Redireciona de volta após login
    redirect_field_name = 'next'
```

PermissionRequiredMixin

- Verifica se o usuário tem permissões específicas;

```
from django.contrib.auth.mixins import PermissionRequiredMixin

class ProdutoCreateView(PermissionRequiredMixin, CreateView):
    model = Produto
    fields = ['nome', 'preco']
    permission_required = 'app.add_produto'

    # Múltiplas permissões
    # permission_required = ['app.add_produto', 'app.change_produto']
```

- Permissões são criadas automaticamente pelo Django para cada model.

Permissões Padrão do Django

- Para cada model, Django cria 4 permissões:
  - `app.add_<model>` - criar
  - `app.change_<model>` - editar
  - `app.delete_<model>` - deletar
  - `app.view_<model>` - visualizar
- Ex: `produtos.add_produto`, `produtos.change_produto`

UserPassesTestMixin

- Teste customizado para autorização;

```
from django.contrib.auth.mixins import UserPassesTestMixin

class ProdutoUpdateView(UserPassesTestMixin, UpdateView):
    model = Produto
    fields = ['nome', 'preco']

    def test_func(self):
        # Só o dono pode editar
        produto = self.get_object()
        return self.request.user == produto.criado_por
```

- Retorna True para permitir, False para negar.

Combinando Mixins

```
from django.contrib.auth.mixins import LoginRequiredMixin, PermissionRequiredMixin

class ProdutoDeleteView(LoginRequiredMixin, PermissionRequiredMixin, DeleteView):
    model = Produto
    success_url = reverse_lazy('produto_lista')
    permission_required = 'produtos.delete_produto'
```

- Primeiro verifica login, depois permissão;
- Ordem dos mixins importa!

CRUD Protegido - Exemplo

```python
from django.contrib.auth.mixins import LoginRequiredMixin

class ProdutoListView(ListView):  # Público
    model = Produto

class ProdutoDetailView(DetailView):  # Público
    model = Produto

class ProdutoCreateView(LoginRequiredMixin, CreateView):
    model = Produto
    fields = ['nome', 'preco']
    success_url = reverse_lazy('produto_lista')

class ProdutoUpdateView(LoginRequiredMixin, UpdateView):
    model = Produto
    fields = ['nome', 'preco']
    success_url = reverse_lazy('produto_lista')

class ProdutoDeleteView(LoginRequiredMixin, DeleteView):
    model = Produto
    success_url = reverse_lazy('produto_lista')
```

Referências

- https://docs.djangoproject.com/en/5.1/topics/class-based-views/
- https://docs.djangoproject.com/en/5.1/ref/class-based-views/
- https://ccbv.co.uk/ (referência visual das CBVs)

Dúvidas?