



A CAMADA VIEW DO DJANGO (PYTHON)

Nesse ebook, vamos tratar da camada _View_ do Django. Nela, nós desenvolvemos a lógica de negócio da nossa aplicação. Vamos ver sobre rotas, processamento de requisições e respostas, utilização de formulários e muito mais!

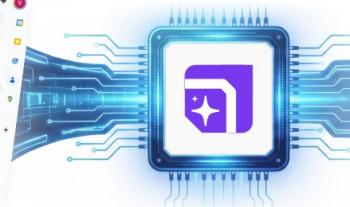
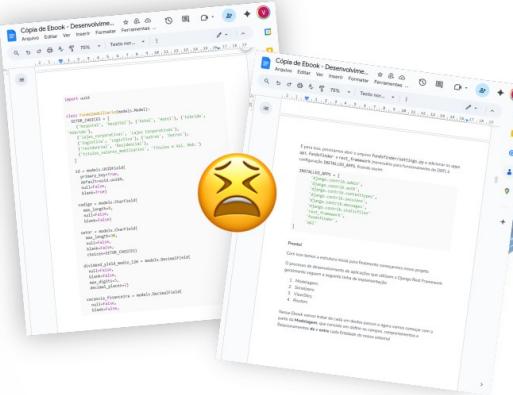
Gere ebooks como este com



Ebookr

em <https://ebookr.ai>

Crie ebooks profissionais incríveis em minutos com IA



Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!



Capas gerados por IA



Infográficos feitos por IA



Edite em Markdown em Tempo Real



Adicione Banners Promocionais

TESTE AGORA



PRIMEIRO CAPÍTULO 100% GRÁTIS

 **Artigo atualizado para Django 5.1** (Dezembro 2025) Código modernizado com `path()` ao invés de `url()`. Views e CBVs permanecem estáveis.

Salve salve Pythonista! 

Continuando a nossa série de *posts* sobre Django, no artigo de hoje vamos tratar sobre a camada *View* da sua arquitetura!

Se você ainda não leu os primeiros *posts* da série, você **COM CERTEZA** está perdendo informações importantes.

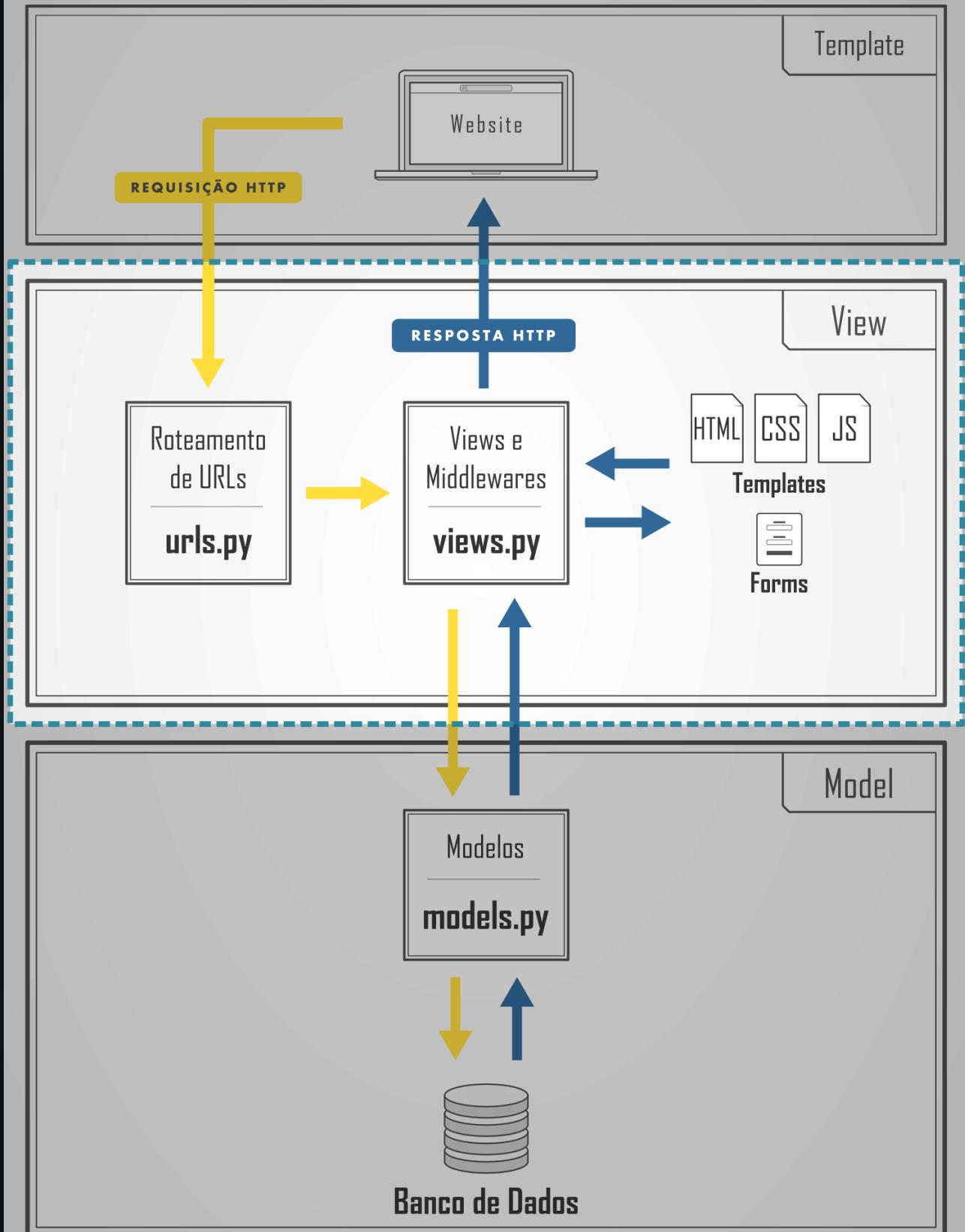
Se esse for o seu caso, então já [clica aqui pra ver o post](#) introdutório e/ou [aqui para ler o post sobre a camada Model e fique por dentro!](#)

Agora, faça uma **xícara de café**, ajuste sua cadeira e vamos nessa que o *post* de hoje tá **COMPLETÃO!!**

Onde estamos...

Primeiramente, vamos nos situar:

ARQUITETURA DO django



No [primeiro post](#), tratamos da parte introdutória do Django, uma visão geral das suas camadas, a instalação do *framework* e (**CLARO**) a criação do famoso **Hello World** *Django-based*.

Já no [segundo post](#), tratamos da camada *Model*, onde definimos as entidades do nosso sistema, a interface com o banco de dados e aprendemos como utilizar a API de acesso a dados provida pelo Django - que facilita muito nossa vida!

No *post* sobre a Camada *Model* desenvolvemos a base do nosso *HelloWorld*: um projeto de gerenciamento de funcionários.

Vamos continuar desenvolvendo-o nesse *post*, portanto, se você não tem o código desenvolvido, [baixe-o aqui][django-project-download] pois iremos utilizá-lo com o base!

Ao final do *post* se encontra o código final com o que foi passado aqui! 😊

Agora, vamos um pouco mais fundo no Django e vamos tratar da camada *View* da arquitetura **MTV** do Django (*Model Template View*).

É nessa camada que descrevemos a **lógica de negócio** da nossa aplicação!

Sem mais delongas, apresento-lhes: **A camada View!**

Detalhes da Camada

Essa camada tem a responsabilidade de processar as **requisições** vindas dos usuários, formar uma **resposta** e enviá-la de volta ao usuário. É aqui que residem nossas **lógicas de negócio**!

Ou seja, essa camada deve: **recepçionar, processar e responder!**

Para isso, começamos pelo **roteamento de URLs**!

A partir da URL que o usuário quer acessar (`/funcionarios`, por exemplo), o Django irá rotear a requisição para quem irá tratá-la.

Mas primeiro, o Django precisa ser informado para **onde** mandar a requisição.

Fazemos isso no chamado **URLconf** e damos o nome à esse arquivo, por convenção, de `urls.py` !

Geralmente, temos um arquivo de rotas por *app* do Django. Portanto, crie um arquivo `urls.py` dentro da pasta `/helloworld` e outro na pasta `/website`.

Como o *app helloworld* é o núcleo da nossa aplicação, ele faz o papel de centralizador de rotas, isto é:

- Primeiro, a requisição cai no arquivo `/helloworld/urls.py` e é roteada para o *app* correspondente.
- Em seguida, o URLConf do *app* (`/website/urls.py`, no nosso caso) vai rotear a requisição para a *view* que irá processar dada requisição.

Dessa forma, o arquivo `helloworld/urls.py` deve conter:

```
from django.urls.conf import include
from django.contrib import admin
from django.urls import path

urlpatterns = [
    # URL padrão
    path('', include('website.urls', namespace='website')),

    # Interface administrativa
    path('admin/', admin.site.urls),
]
```

Assim, toda requisição sem o caminho (`path`) `/admin` vai ser roteado pela primeira regra: `website.urls` (URLConf do *app website*).

Em seguida, a requisição segue para o roteamento do *app* específico (`website`, no caso).

Pode parecer complicado, mas ali embaixo, quando tratarmos mais sobre Views, vai fazer mais sentido (se não fizer, poste sua dúvida aqui embaixo)!

A configuração do URLConf é bem simples! Basta definirmos qual função ou `View` irá processar requisições de **tal** URL.

Por exemplo, queremos que:

Quando um usuário acesse a URL raíz `/`, o Django chame a função `index()` para processar tal requisição

Vejamos como poderíamos configurar esse roteamento no nosso arquivo `urls.py`:

```
# Importamos a função index() definida no arquivo views.py
from . import views

app_name = 'website'

# urlpatterns a lista de roteamentos de URLs para funções/Views
urlpatterns = [
    # GET /
    path('', views.index, name='index'),
]
```

O atributo `app_name = 'website'` define o namespace do app **website** (lembre-se do décimo nono Zen do Python: **namespaces** são uma boa ideia! - clique [aqui para saber mais sobre o Zen do Python](#))

Django **antes da versão 2.0** utilizava **Expressões Regulares** (RegEx) para o “casamento” de URLs.

Desde Django 2.0+, a sintaxe foi simplificada! Ao invés de usar RegEx complexas:

```
url(r'^funcionarios/(?P<ano>[0-9]{4})/$', views.funcionarios_por_ano),
```

podemos fazer apenas:

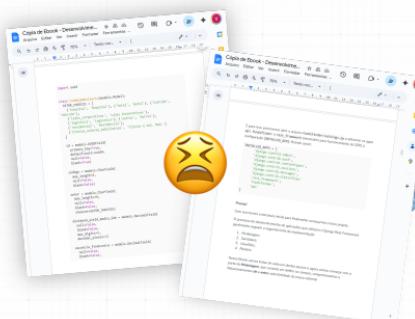
```
path('funcionarios/<int:ano>', views.funcionarios_por_ano),
```

Mais legível, né?!



*Estou desenvolvendo o **Ebookr.ai**, uma plataforma que transforma suas ideias em ebooks profissionais usando IA — com geração de capa, infográficos e exportação em PDF. Te convido a conhecer!*

Crie Ebooks profissionais incríveis em minutos com IA



Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...



... e deixe que nossa IA faça o trabalho pesado!

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

 Capas gerados por IA

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

Funções vs *Class Based Views*

Com as URLs corretamente configuradas, o Django irá rotear a sua requisição para onde você definiu. No caso acima, sua requisição irá cair na função `views.funcionarios_por_ano()`.

Podemos tratar as requisições de duas formas: através de funções ou através de *Class Based Views* (**CBVs** - ou apenas *Views*).

Utilizando **funções**, você basicamente vai definir uma função que recebe como parâmetro uma requisição (`request`), realizar algum processamento e retornar alguma informação.

Já as *Views* são classes que herdam de `django.views.generic.base.View` e que agrupam diversas funcionalidades e facilitam a vida do desenvolvedor.

Nós podemos herdar e estender as funcionalidades das *Views* do Django para atender a lógica da nossa aplicação.

Por exemplo, suponha você quer criar uma tela com a **lista de todos os funcionários**.

Utilizando **funções**, você poderia fazer da seguinte forma:

```
def lista_funcionarios(request):
    # Primeiro, buscamos os funcionários
    funcionarios = Funcionario.objects.all()

    # Incluímos no contexto
    contexto = {
        'funcionarios': funcionarios
    }

    # Retornamos o template no qual os funcionários serão dispostos
    return render(request, "templates/funcionarios.html", contexto)
```

Algumas colocações:

- Toda função que vai processar requisições no Django recebe como primeiro parâmetro, um objeto `request` contendo os dados dessa requisição.
- Contexto é o conjunto de dados que estarão disponíveis no *template*.
- A função `django.shortcuts.render()` é um atalho (*shortcut*) do próprio Django que facilita a renderização de *templates*: ela recebe a própria requisição, o diretório do *template* e o contexto.

Já utilizando *Views*, podemos utilizar a *View* `django.views.generic.ListView` para listar os funcionários, da seguinte forma:

```
from django.views.generic import ListView

class ListaFuncionarios(ListView):
    template_name = "templates/funcionarios.html"
    model = Funcionario
    context_object_name = "funcionarios"
```

Perceba que você não precisou descrever a lógica para buscar a lista de funcionários?

É **exatamente isso** que as `Views` do Django proporcionam: elas descrevem o comportamento padrão para as funcionalidades mais simples (listagem, exclusão, busca simples, atualização).

O caso comum para uma listagem de objetos é buscar todo o conjunto de dados daquela entidade e mostrar no template, certo?! É exatamente **isso** que a `ListView` faz!

Com isso, um objeto `funcionarios` estará disponível, **magicamente**, no seu *template* para iteração.

Dessa forma, podemos então criar uma tabela no nosso *template* com os dados de todos os funcionários:

```
<table>
  <tbody>
    {% raw %}{% for funcionario in funcionarios %}
      <tr>
        <td>{{ funcionario.nome }}</td>
        <td>{{ funcionario.sobrenome }}</td>
        <td>{{ funcionario.remuneracao }}</td>
        <td>{{ funcionario.tempo_de_servico }}</td>
      </tr>
    {% endfor %}{% endraw %}
  </tbody>
</table>
```

*Se já quiser saber mais sobre **templates**, [acesse aqui o post sobre a Camada de Templates!](#)*

O Django tem uma diversidade enorme de *Views*, uma para cada finalidade, por exemplo:

- `CreateView` : Facilita a criação de objetos (*É o **Create** do **CRUD***)
- `DetailView` : Traz os detalhes de um objeto (*É o **Retrieve** do **CRUD***)
- `UpdateView` : Facilita a atualização de um objeto (*É o **Update** do **CRUD***)
- `DeleteView` : Facilita a deleção de objetos (*É o **Delete** do **CRUD***)

E várias outras muito úteis!

Agora vamos tratar detalhes do tratamento de requisições através de funções. Em seguida, trataremos mais sobre as *Class Based Views*.

Funções (*Function Based Views*)

Utilizar funções é a maneira mais explícita para tratar requisições no Django (veremos que as *Class Based Views* podem ser um pouco mais complexas pois muita coisa acontece implicitamente).

Utilizando funções, geralmente tratamos primeiro o método HTTP da requisição: foi um `GET`? Foi um `POST`? Um `OPTION`?

A partir dessa informação, processamos a requisição da maneira desejada.

Vamos seguir o exemplo abaixo:

```
def cria_funcionario(request, pk):
    # Verificamos se o método POST
    if request.method == 'POST':
        form = FormularioDeCriacao(request.POST)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('list_view'))

    # Qualquer outro método: GET, OPTION, DELETE, etc...
    else:
        return render(request, templates/form.html, {'form': form})
```

O fluxo é bem simples de entender, vamos lá:

- Primeiro, conforme mencionei, verificamos o método HTTP da requisição no campo `method` do objeto `request` na **linha 3**.
- Depois instanciamos um `form` com os dados da requisição (no caso `POST`) com `FormularioDeCriacao(request.POST)` na **linha 4** (vamos falar mais sobre `Form` já já).
- Verificamos os campos do `form` com `form.is_valid()` na **linha 6..**

- Se tudo estiver **OK**, retornamos um *redirect* para uma *view* de listagem na **linha 8**.
- Se for qualquer outro método, apenas renderizamos a página novamente com `render(request, templates/form.html, {'form': form})` na **linha 12**.

Apesar de ser pouco código, foram introduzidos diversos conceitos novos. Então vamos por partes!

Vamos começar abrindo o objeto `request` para ver o que tem dentro e ver o que pode ser útil para nós.

Observação: Para saber mais sobre os campos do objeto `request`, dá uma olhada na classe `django.http.request.HttpRequest`!

Separai aqui alguns atributos que provavelmente serão os mais utilizados por você:

- `request.scheme` : String representando o esquema (*HTTP* ou *HTTPS*).
- `request.path` : String com o caminho da página requisitada - exemplo: **/cursos/curso-de-python/detalhes**.
- `request.method` : Conforme citamos, contém o método *HTTP* da requisição (**GET**, **POST**, **UPDATE**, **OPTION**, etc).
- `request.content_type` : Representa o tipo MIME da requisição - `text/plain` para texto plano, `image/png` para arquivos .PNG, por exemplo - saiba mais [clicando aqui](#)
- `request.GET` : Um *dict* contendo os parâmetros GET da requisição.
- `request.POST` : Um *dict* contendo os parâmetros do corpo de uma requisição POST.

- `request.FILES`: Caso seja uma página de *upload*, contém os arquivos que foram enviados. Só contém dados se for uma requisição do tipo *POST* e o `<form>` da página *HTML* tenha o parâmetro `enctype="multipart/form-data"`.
- `request.COOKIES`: *Dict* contendo todos os *COOKIES* no formato de string.

Em 99.9% dos casos, estaremos processando dados desses campos!

Dica: Quer ver os campos da requisição em “tempo real” que chegaram no servidor? Então liga o debug da sua IDE (caso tenha) e coloque um breakpoint em alguma view, dispare alguma requisição e veja o que acontece! Por exemplo, utilizando o PyCharm:

```

 61
 62
 63     return HttpResponseRedirect(self.get_success_url())
 64
 65
 66 # COURSE DETAIL VIEW
 67
 68
 69 class CourseDetailView(DashboardMixin, DetailView):
 70     template_name = 'dashboard/course/detail.html'
 71     model = Course
 72
 73     def get_context_data(self, **kwargs):
 74         context = super().get_context_data()
 75
 76         # Get course questions
 77         context['questions'] = CourseDiscussionQuestion.objects.filter(
 78             course_id=context['course'].id
 79         ).all()
 80
 81         # Put comment form on context
 82         context['form'] = CommentForm()
 83
 84         # All lessons' progress
 85         context['user_progress'] = UserCourseProgress.objects.filter(
 86             user_id=self.request.user.id,
 87             course=context['course'],
 88         ).all()
 89
 90         # Last watched lesson
 91         last_watched_id = UserCourseProgress.objects.filter(
 92             user_id=self.request.user.id,
 93             course=context['course'],
 94             accomplished=True
 95         ).aggregate(Max('lesson'))
 96
 97         if last_watched_id is None:
 98             context['last_watched_lesson'] = None
 99         else:
100             context['last_watched_lesson'] = Lesson.objects.filter(
101                 id=last_watched_id['lesson__max']
102             ).first()
103
104         return context
105
106
107 # LESSON DETAIL VIEW
108
109
110 class LessonDetailView(DashboardMixin, DetailView):
111     template_name = 'dashboard/lesson/player.html'
112     model = Lesson
113     context_object_name = 'lesson'
114
115     def get_context_data(self, **kwargs):
116         context = super().get_context_data()
117         context['form'] = CommentForm()
118
119         return context
  
```

(Caso esteja ruim de ver a imagem, clique com o botão direito em cima da imagem e depois em “Abrir imagem em uma nova guia”).

Do **lado esquerdo**, o código com o breakpoint que coloquei (círculo vermelho). Do **lado direito**, as variáveis do objeto `request` que tratamos aqui em cima!

Outra dica: Recomendo fortemente a utilização do PyCharm! Ela é MUITO boa!

Dito isso, agora vamos tratar detalhes do tratamento de requisições através de *Class Based Views*.

Classes (CBV - *Class Based Views*)

Conforme expliquei ali em cima, as *Class Based Views* servem para automatizar e facilitar nossa vida, encapsulando funcionalidades comuns que todo desenvolvedor sempre acaba implementando. Por exemplo, geralmente:

- Queremos que quando um usuário vá para página inicial, seja mostrado **apenas uma página** simples.
- Queremos que nossa **página de listagem** contenha a **lista** de todos os funcionários (por exemplo) cadastrados no banco de dados.
- Queremos uma **página com um formulário** contendo todos os campos pré-preenchidos para **atualização** de dado funcionário.
- Queremos uma **página de exclusão** de funcionários.
- Queremos um formulário em branco para **inclusão de um novo funcionário**.

Certo?!

Pois é, as **CBVs** - *Class Based Views* - fazem isso!

Temos basicamente **duas formas** para utilizar uma **CBV**.

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

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

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

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

Eu particularmente prefiro a segunda forma, pois encapsula todas as Views no mesmo arquivo: no views.py

TemplateView

Por exemplo, para o **primeiro caso**, podemos utilizar a [TemplateView \(documentação\)](#) para apenas mostrar uma página, da seguinte forma:

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

E a configuração de rotas fica assim:

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

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

ListView

Já para o segundo caso, de **listagem de funcionários**, podemos utilizar a `ListView` ([documentação](#)). 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.

Por exemplo, a seguinte *View*:

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

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

Com essa configuração:

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

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

Resulta em uma página `lista.html` contendo um objeto chamado "**funcionarios**" contendo todos os Funcionários disponível para iteração.

Dica: Eu geralmente coloco o nome da View como sendo o Model com a CBV base. Por exemplo: se eu fosse criar uma view que vai listar todos os Cursos cadastrados, eu daria o nome de `CursoListView` (`Model=<Curso>, CBV=<ListView>`).

UpdateView

Já para **aualização de usuários** podemos utilizar a `UpdateView` ([documentação](#)). Com ela, setamos (no mínimo) qual o *Model*, quais campos e qual o nome do template, e com isso temos um formulário para atualização do modelo em questão.

```
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`), dessa forma:

```
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 e 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`:

```

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**).*

DeleteView

Para deletar funcionários, utilizamos 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 = "website/exclui.html"  
    model = Funcionario  
    context_object_name = 'funcionario'  
    success_url = reverse_lazy("website:lista_funcionarios")
```

Assim como na `UpdateView`, fazemos a configuração do `id` a 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 }}</b>? <br><br>

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

Algumas colocações:

- Se lembra do atributo `context_object_name`? Olha ele ali presente 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 no **próximo post!**

CreateView

Essa *view*, de criação, é bem simples!

Precisamos apenas dizer para o Django o *model*, o nome do *template*, a classe do formulário (vamos tratar mais sobre *Forms* ali embaixo) e a URL de retorno, caso o haja sucesso na inclusão do Funcionário.

Podemos fazer isso dessa forma:

```
from django.views.generic import CreateView

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

`reverse_lazy()` traduz a View em URL. No nosso caso, queremos que quando haja a inclusão do Funcionário, sejamos redirecionados para a página de listagem, para podermos conferir que o Funcionário foi realmente adicionado.

E a configuração da rota no arquivo `urls.py`:

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

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

Com isso, estará disponível no `template` configurado (`website/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 **cru**, isto é, 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 que 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 }}{% endraw %}">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>

```

Dessa forma:

- `{% 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.

*Antes de continuarmos, vamos respirar um pouco... **Encha sua xícara de café** que ainda tem muita coisa!*

Agora vamos detalhar mais a classe `Form`, abordada aqui em cima!

Forms

O tratamento de formulários é uma tarefa que pode ser **bem complexa**.

Considere um formulário com diversos campos e diversas regras de validação: seu tratamento não é mais um processo simples.

Os *Forms* do Django são formas de descrever os elementos `<form> . . . </form>` das páginas HTML, simplificando e automatizando o processo de validação.

O Django trata três partes distintas dos formulários:

- Preparação dos dados tornando-os prontos para renderização
- Criação de formulários HTML para os dados
- Recepção e processamento dos formulários enviados ao servidor

Basicamente, queremos uma forma de renderizar em nosso *template* o código HTML:

```
<form action="/insere-funcionario/" method="post">
    <label for="nome">Your name: </label>
    <input id="nome" type="text" name="nome" value="">
    <input type="submit" value="Enviar">
</form>
```

Que, ao ser submetido ao servidor, tenha seus campos de entrada validados e inseridos no banco de dados.

No centro desse sistema de formulários do Django está a classe `Form`.

Nela, nós descrevemos os campos que estarão disponíveis no formulário HTML e os métodos de validação.

Para o formulário acima, podemos descrevê-lo da seguinte forma.

```
from django import forms

class InsereFuncionarioForm(forms.Form):
    nome = forms.CharField(
        label='Nome do Funcionário',
        max_length=100
    )
```

Nesse formulário:

- Utilizamos a classe `forms.CharField` para descrever um campo de texto.
- O parâmetro `label` descreve um rótulo para esse campo.
- `max_length` decreve o tamanho máximo que esse *input* pode receber (100 caracteres, no caso).

Veja os diversos tipos de campos disponíveis [acessando aqui](#)

A classe `forms.Form` possui um método muito importante, chamado `is_valid()`.

Quando um formulário é submetido ao servidor, esse é um dos métodos que irá realizar a validação dos campos do formulário.

Se tudo estiver **OK**, ele colocará os dados do formulário no atributo `cleaned_data` (que pode ser acessado por você posteriormente para pegar alguma informação - como o nome que foi inserido pelo usuário no campo `<input name='nome'>`).

Como o processo de validação do Django é bem complexo e para não prolongar muito o *post*, [acesse a documentação aqui](#) para saber mais.

Vamos ver agora um exemplo mais complexo com um formulário de inserção de um Funcionário com todos os campos.

Vamos começar criando o arquivo `forms.py` dentro do app `website` do nosso projeto usando como base os campos do `model Funcionario`.

Se você não se lembra dos campos - que descrevemos no [post passado sobre a camada Model](#) - aqui vão eles:

```
from django.db import models

class Funcionario(models.Model):

    nome = models.CharField(
        max_length=255,
        null=False,
        blank=False
    )

    sobrenome = models.CharField(
        max_length=255,
        null=False,
        blank=False
    )

    cpf = models.CharField(
        max_length=14,
        null=False,
        blank=False
    )

    tempo_de_servico = models.IntegerField(
        default=0,
        null=False,
        blank=False
    )

    remuneracao = models.DecimalField(
        max_digits=8,
        decimal_places=2,
        null=False,
        blank=False
    )
```

Dessa forma, e consultando a [documentação](#) dos possíveis campos do nosso formulário, nós podemos descrever um formulário de inserção da seguinte forma:

```
from django import forms

class InsereFuncionarioForm(forms.Form):

    nome = forms.CharField(
        required=True,
        max_length=255
    )

    sobrenome = forms.CharField(
        required=True,
        max_length=255
    )

    cpf = forms.CharField(
        required=True,
        max_length=14
    )

    tempo_de_servico = forms.IntegerField(
        required=True
    )

    remuneracao = forms.DecimalField(
    )
```

Affff, o Model e o Form são quase iguais... Terei que reescrever os campos toda vez?



Claro que não, jovem! Pra isso o Django criou o incrível `ModelForm` !!! 😊

Com o `ModelForm` nós descrevemos os campos que queremos (atributo `fields`) e/ou os campos que não queremos (atributo `exclude`) no formulário em forma de lista.

Para isso, utilizamos a classe interna `Meta` para incluirmos esses metadados na nossa classe.

Metadado (no caso do *Model* e do *Form*) é tudo aquilo que não será transformado em campo, como `model`, `fields`, `ordering` etc ([mais sobre Meta options](#))

Nosso `ModelForm`, pode ser descrito da seguinte forma:

```
from django import forms

class InsereFuncionarioForm(forms.ModelForm):
    class Meta:
        # Modelo base
        model = Funcionario

        # Campos que estarão no form
        fields = [
            'nome',
            'sobrenome',
            'cpf',
            'remuneracao'
        ]

        # Campos que não estarão no form
        exclude = [
            'tempo_de_servico'
        ]
```

Podemos utilizar apenas o campo `fields`, apenas o campo `exclude` ou os dois juntos.

Mesmo utilizando os atributos `fields` e `exclude`, ainda podemos adicionar outros campos, independente dos campos do *Model*.

O resultado será um formulário com todos os campos presentes no `fields`, menos os campos do `exclude` mais os outros campos que adicionarmos.

Ficou confuso? Então vamos ver o exemplo:

```

from django import forms

class InsereFuncionarioForm(forms.ModelForm):

    chefe = forms.BooleanField(
        label='Chefe?',
        required=True,
    )

    biografia = forms.CharField(
        label='Biografia',
        required=False,
        widget=forms.TextArea
    )

    class Meta:
        # Modelo base
        model = Funcionario

        # Campos que estarão no form
        fields = [
            'nome',
            'sobrenome',
            'cpf',
            'remuneracao'
        ]

        # Campos que não estarão no form
        exclude = [
            'tempo_de_servico'
        ]

```

Isso vai gerar um formulário com:

- Todos os campos contidos em `fields` menos os campos contidos em `exclude`
- O campo `forms.BooleanField`, renderizado como um `checkbox`
`(<input type='checkbox' name='chefe' ...>)`

- Uma área de texto (`<textarea name='biografia' ...></textarea>`)

Assim como é possível definir atributos nos modelos, os campos do formulário também são customizáveis.

Veja que o campo `biografia` é do tipo `CharField`, portanto deveria ser renderizado como um campo `<input type='text' ...>'.`

Contudo, eu modifiquei o campo setando o atributo `widget` com `forms.TextArea`.

Assim, ele não mais será um simples `input`, mas será renderizado como um `<textarea></textarea>` no nosso `template!`



*Criei o **Ebookr.ai**, uma plataforma que usa IA para gerar ebooks profissionais sobre qualquer tema — com capa gerada por IA, infográficos automáticos e exportação em PDF. Confere!*

Crie **Ebooks profissionais incríveis** em minutos com IA



Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

... e deixe que nossa IA faça o trabalho pesado!

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

 Capas gerados por IA

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

Middlewares

Middlewares são trechos de códigos que podem ser executados antes ou depois do processamento de requisições/respostas pelo Django.

É uma forma que os desenvolvedores, nós, temos para alterar como o Django processa algum dado de entrada ou de saída.

Se você olhar no arquivo `settings.py`, nós temos a lista `MIDDLEWARE` com diversos *middlewares* pré-configurados:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

Por exemplo, o `middleware AuthenticationMiddleware` é responsável por adicionar a variável `user` a todas as requisições.

Dessa forma, você pode, por exemplo, mostrar o usuário logado no seu *template*:

```
{% raw %}  
<li>  
    <a href="{% url 'profile' id=user.id %}">Olá, {{ user.email }}</a>  
</li>  
{% endraw %}
```

Vamos ver agora como podemos criar o nosso próprio *middleware*.

Um *middleware* é um método *callable* (que tem uma implementação do método `__call__()`) que recebe uma **requisição** e retorna uma **resposta**, assim como uma *View*, e pode ser escrito como função ou como Classe.

Um exemplo de *middleware* escrito como função é:

```

def middleware_simples(get_response):
    # Código de inicialização do Middleware

    def middleware(request):
        # Código a ser executado para cada requisição
        # antes da View, e outros middlewares, serem executada

        response = get_response(request)

        # Código a ser executado para cada requisição/resposta
        # após a execução da View que irá processar

        return response

    return middleware

```

E como Classe:

```

class MiddlewareSimples:
    def __init__(self, get_response):
        self.get_response = get_response
        # Código de inicialização do Middleware

    def __call__(self, request):
        # Código a ser executado para cada requisição
        # antes da View, e outros middlewares, serem executada

        response = self.get_response(request)

        # Código a ser executado para cada requisição/resposta
        # após a execução da View que irá processar

        return response

```

Como cada *Middleware* é executado de maneira encadeada, do topo da lista **MIDDLEWARE** para o fim, a saída de um é a entrada do próximo.

O método `get_response()` pode ser a própria *View*, caso ela seja a última configurada no `MIDDLEWARE` do `settings.py`, ou o próximo *middleware* da cadeia.

Utilizando a construção do *middleware* via Classe, nós temos três métodos importantes:

process_view

Assinatura: `process_view(request, func, args, kwargs)`

Esse método é chamado logo antes do Django executar a *View* que vai processar a requisição e possui os seguintes parâmetros:

- `request` é o objeto `HttpRequest`.
- `func` é a própria *view* que o Django está para chamar ao final da cadeia de *middlewares*.
- `args` é a lista de parâmetros posicionais que serão passados à *view*.
- `kwargs` é o *dict* contendo os argumentos nomeados (*keyword arguments*) que serão passados à *view*.

Esse método deve retornar `None` ou um objeto `HttpResponse`:

- Caso retorne `None`, o Django entenderá que deve continuar a cadeia de *Middlewares*.
- Caso retorne `HttpResponse`, o Django entenderá que a resposta está pronta para ser enviada de volta e não vai se preocupar em chamar o resto da cadeia de *Middlewares*, nem a *view* que iria processar a requisição.

process_exception

Assinatura: `process_exception(request, exception)`

Esse método é chamada quando uma *view* lança uma exceção e deve retornar ou `None` ou `HttpResponse`. Caso retorne um objeto `HttpResponse`, o Django irá aplicar o *middleware* de resposta e o de *template*, retornando a requisição ao *browser*.

- `request` é o objeto `HttpRequest`
- `exception` é a exceção propriamente dita lançada pela *view* (`Exception`).

process_template_response

Assinatura: `process_template_response(request, response)`

Esse método é chamado logo após a *view* ter terminado sua execução, caso a resposta tenha uma chamada ao método `render()` indicando que a resposta possui um *template*.

Possui os seguintes parâmetros:

- `request` é um objeto `HttpRequest`.
- `response` é o objeto `TemplateResponse` retornado pela *view* ou por outro *middleware*.

Agora vamos criar um *middleware* um pouco mais complexo para exemplificar o que foi dito aqui!

Vamos supor que queremos um *middleware* que filtre requisições e só processe aquelas que venham de uma determinada lista de IP's.

O que precisamos fazer é abrir o cabeçalho de todas as requisições que chegam no nosso servidor e verificar se o IP de origem bate com a nossa lista de IP's.

Para isso, colocamos a lógica no método `process_view`, da seguinte forma:

```
class FiltraIPMiddleware:

    def __init__(self, get_response=None):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)

        return response

    def process_view(self, request, func, args, kwargs):
        # Lista de IPs autorizados
        ips_autorizados = ['127.0.0.1']

        # IP do usuário
        ip = request.META.get('REMOTE_ADDR')

        # Verifica se o IP do cliente está na lista de IPs autorizados
        if ip not in ips_autorizados:
            # Se usuário não autorizado > HTTP 403: Não Autorizado
            return HttpResponseForbidden("IP não autorizado")

        # Se for autorizado, não fazemos nada
        return None
```

Depois disso, precisamos registrar nosso *middleware* no arquivo de configurações `settings.py` (na configuração `MIDDLEWARE`):

```

MIDDLEWARE = [
    # Middlewares do próprio Django
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',

    # Nosso Middleware
    'helloworld.middlewares.FiltrarIPMiddleware',
]

```

Agora, podemos testar seu funcionamento alterando a lista `ips_autorizados` :

- Coloque `ips_autorizados = ['127.0.0.1']` e tente acessar alguma URL da nossa aplicação: devemos conseguir acessar normalmente nossa aplicação, pois como estamos executando o servidor localmente, nosso IP será 127.0.0.1 e, portanto, passaremos no teste.
- Coloque `ips_autorizados = []` e tente acessar alguma URL da nossa aplicação: deve aparecer a mensagem de “**IP não autorizado**”, pois nosso IP (127.0.0.1) não está autorizado a acessar o servidor.

Código

Se quiser fazer o *download* do código desenvolvido até aqui, [clique aqui para baixá-lo][django-view-code]!

Conclusão

Ufa! Acho melhor parar por aqui... 😞

Vimos vários conceitos sobre os tipos de *Views* (funções e classes), alguns tipos de CBV (*Class Based Views*), como mapear suas URL para suas *views* através do *URLConf*, como entender o fluxo da sua requisição utilizando o debug da sua IDE, como utilizar os poderosos `Forms` do Django, como utilizar *middlewares* para adicionar camadas extras de processamento às requisições e respostas que chegam e saem da nossa aplicação.

Com certeza ainda tem muita coisa para você descobrir e desvendar! Mas não se esqueça que qualquer dúvida que você tiver no seu processo de aprendizagem, não exite em entrar em contato comigo pelas minhas redes sociais ou pelo **box de comentário** aqui embaixo!

Espero ter facilitado seu entendimento sobre a camada *View* do Django!

No *post* sobre a Camada *Template* ([que já está disponível aqui](#)) construímos os *templates* e páginas HTML da nossa aplicação!

Quer levar esse conteúdo para onde for com nosso **ebook GRÁTIS?**

Então aproveita essa chance 

Nos vemos!

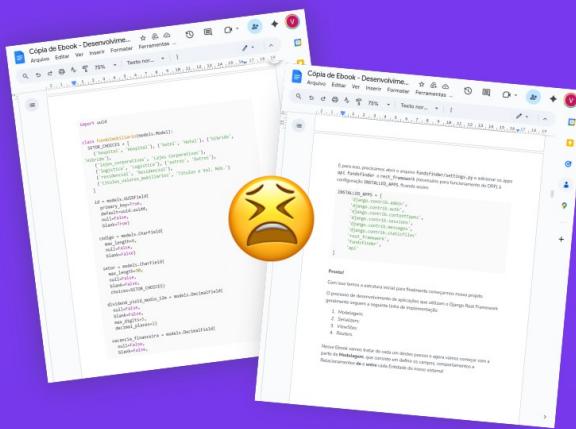
Bom desenvolvimento! 😊

Não se esqueça de conferir!



Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA



• Infográficos feitos



Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado



 Edite em Markdown em Tempo Real

TESTE AGORA



 PRIMEIRO CAPÍTULO 100% GRÁTIS