



A CAMADA VIEW DO DJANGO (PYTHON)

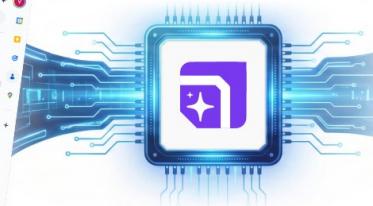
Nesse ebook, vamos tratar da camada _View_ do Django. Nela, nós desenvolvemos a lógica de negócio da nossa aplicação. Vamos ver sobre rotas, processamento de requisições e respostas, utilização de formulários e muito mais!

Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA 

 **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.

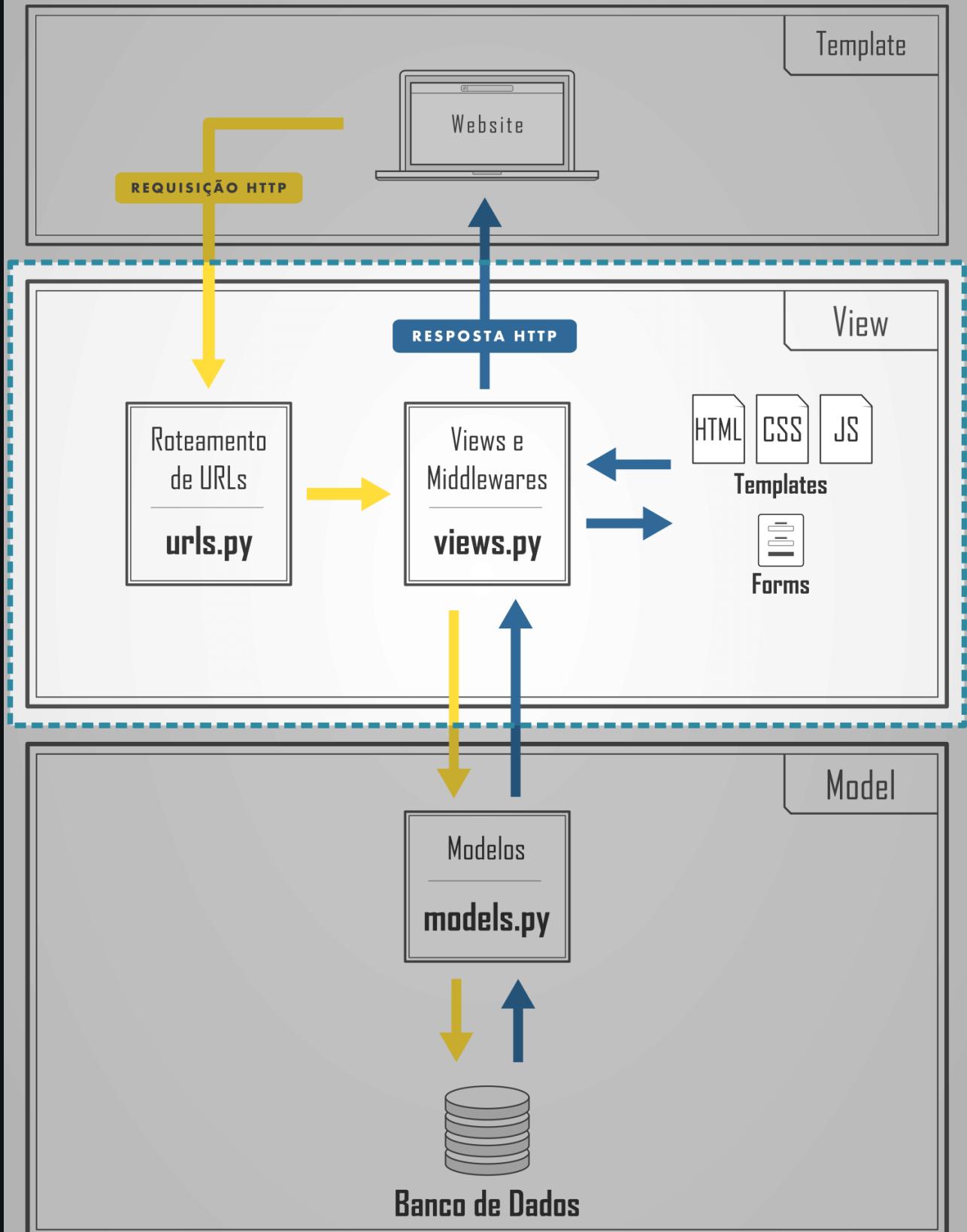
Se esse for o seu caso, então já [clica aqui pra ver o post introdutório](#) e/ou [aqui para ler o post sobre a camada Model](#) e **fique por dentro!**

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 “cascimento” 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),
```

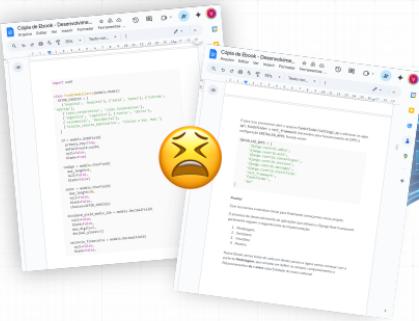
Mais legível, né?!



*Estou construindo o **DevBook**, uma plataforma que usa IA para criar e-books técnicos — com código formatado e exportação em PDF. Te convido a conhecê-lo!*

Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

Syntax Highlight

Adicione Banners Promocionais

Edite em Markdown em Tempo Real

Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS

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 `List-View` 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`!

Separarei 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:

```

1  def get_context_data(self, **kwargs):
2      # Get super() content
3      context = super().get_context_data()
4
5      # Get course questions
6      context['questions'] = CourseDiscussionQuestion.objects.filter(
7          course_id=context['course'].id
8      ).all()
9
10     # Put comment form on context
11     context['form'] = CommentForm()
12
13     # All lessons' progress
14     context['user_progress'] = UserCourseProgress.objects.filter(
15         user_id=self.request.user.id,
16         course=context['course'],
17     ).all()
18
19     # Last watched lesson
20     last_watched_id = UserCourseProgress.objects.filter(
21         user_id=self.request.user.id,
22         course=context['course'],
23     ).first()
24     accomplished=True
25     .aggregate(Max('lesson'))
26
27     if last_watched_id is None:
28         context['last_watched_lesson'] = None
29     else:
30         context['last_watched_lesson'] = Lesson.objects.filter(
31             id=last_watched_id.get('lesson__max')
32         ).first()
33
34     return context
35
36
37 # LESSON DETAIL VIEW
38
39
40 class LessonDetailView(DashboardMixin, DetailView):
41     template_name = 'dashboard/lesson/player.html'
42     model = Lesson
43     context_object_name = 'lesson'
44
45     def get_context_data(self, **kwargs):
46         context = super().get_context_data()
47
48         # Get course questions
49         context['questions'] = CourseDiscussionQuestion.objects.filter(
50             course_id=context['course'].id
51         ).all()
52
53         # Put comment form on context
54         context['form'] = CommentForm()
55
55
56         # All lessons' progress
57         context['user_progress'] = UserCourseProgress.objects.filter(
58             user_id=self.request.user.id,
59             course=context['course'],
60         ).all()
61
62         # Last watched lesson
63         last_watched_id = UserCourseProgress.objects.filter(
64             user_id=self.request.user.id,
65             course=context['course'],
66         ).first()
67         accomplished=True
68         .aggregate(Max('lesson'))
69
70         if last_watched_id is None:
71             context['last_watched_lesson'] = None
72         else:
73             context['last_watched_lesson'] = Lesson.objects.filter(
74                 id=last_watched_id.get('lesson__max')
75             ).first()
76
77         return context
78
79
80 # COURSE DETAIL VIEW
81
82 class CourseDetailView(DashboardMixin, DetailView):
83     template_name = 'dashboard/course/detail.html'
84     model = Course
85
86     def get_context_data(self, **kwargs):
87         # Get super() content
88         context = super().get_context_data()
89
90         # Get course questions
91         context['questions'] = CourseDiscussionQuestion.objects.filter(
92             course_id=context['course'].id
93         ).all()
94
95         # Put comment form on context
96         context['form'] = CommentForm()
97
98         # All lessons' progress
99         context['user_progress'] = UserCourseProgress.objects.filter(
100            user_id=self.request.user.id,
101            course=context['course'],
102        ).all()
103
104         # Last watched lesson
105         last_watched_id = UserCourseProgress.objects.filter(
106             user_id=self.request.user.id,
107             course=context['course'],
108         ).first()
109         accomplished=True
110         .aggregate(Max('lesson'))
111
112         if last_watched_id is None:
113             context['last_watched_lesson'] = None
114         else:
115             context['last_watched_lesson'] = Lesson.objects.filter(
116                 id=last_watched_id.get('lesson__max')
117             ).first()
118
119         return context
120
121
122 # COURSE LIST VIEW
123
124 class CourseListView(DashboardMixin, ListView):
125     template_name = 'dashboard/course/list.html'
126     model = Course
127
128     def get_context_data(self, **kwargs):
129         context = super().get_context_data()
130
131         # Get course questions
132         context['questions'] = CourseDiscussionQuestion.objects.filter(
133             course_id=context['course'].id
134         ).all()
135
136         # Put comment form on context
137         context['form'] = CommentForm()
138
139         # All lessons' progress
140         context['user_progress'] = UserCourseProgress.objects.filter(
141             user_id=self.request.user.id,
142             course=context['course'],
143         ).all()
144
145         # Last watched lesson
146         last_watched_id = UserCourseProgress.objects.filter(
147             user_id=self.request.user.id,
148             course=context['course'],
149         ).first()
150         accomplished=True
151         .aggregate(Max('lesson'))
152
153         if last_watched_id is None:
154             context['last_watched_lesson'] = None
155         else:
156             context['last_watched_lesson'] = Lesson.objects.filter(
157                 id=last_watched_id.get('lesson__max')
158             ).first()
159
160         return context
161
162
163 # LESSON LIST VIEW
164
165 class LessonListView(DashboardMixin, ListView):
166     template_name = 'dashboard/lesson/list.html'
167     model = Lesson
168
169     def get_context_data(self, **kwargs):
170         context = super().get_context_data()
171
172         # Get course questions
173         context['questions'] = CourseDiscussionQuestion.objects.filter(
174             course_id=context['course'].id
175         ).all()
176
177         # Put comment form on context
178         context['form'] = CommentForm()
179
180         # All lessons' progress
181         context['user_progress'] = UserCourseProgress.objects.filter(
182             user_id=self.request.user.id,
183             course=context['course'],
184         ).all()
185
186         # Last watched lesson
187         last_watched_id = UserCourseProgress.objects.filter(
188             user_id=self.request.user.id,
189             course=context['course'],
190         ).first()
191         accomplished=True
192         .aggregate(Max('lesson'))
193
194         if last_watched_id is None:
195             context['last_watched_lesson'] = None
196         else:
197             context['last_watched_lesson'] = Lesson.objects.filter(
198                 id=last_watched_id.get('lesson__max')
199             ).first()
200
201         return context
202
203
204 # COURSE SEARCH VIEW
205
206 class CourseSearchView(DashboardMixin, View):
207     def get(self, request):
208         query = request.GET.get('q')
209
210         courses = Course.objects.filter(name__icontains=query)
211
212         context = {
213             'courses': courses,
214             'query': query
215         }
216
217         return render(request, 'dashboard/course/search.html', context)
218
219
220 # COURSE DOWNLOAD VIEW
221
222 class CourseDownloadView(DashboardMixin, View):
223     def get(self, request, slug):
224         course = get_object_or_404(Course, slug=slug)
225
226         file = course.get_file()
227
228         response = FileResponse(file, content_type='application/pdf')
229
230         return response
231
232
233 # COURSE DOWNLOAD VIEW
234
235 class CourseDownloadView(DashboardMixin, View):
236     def get(self, request, slug):
237         course = get_object_or_404(Course, slug=slug)
238
239         file = course.get_file()
240
241         response = FileResponse(file, content_type='application/pdf')
242
243         return response
244
245
246 # COURSE DOWNLOAD VIEW
247
248 class CourseDownloadView(DashboardMixin, View):
249     def get(self, request, slug):
250         course = get_object_or_404(Course, slug=slug)
251
252         file = course.get_file()
253
254         response = FileResponse(file, content_type='application/pdf')
255
256         return response
257
258
259 # COURSE DOWNLOAD VIEW
260
261 class CourseDownloadView(DashboardMixin, View):
262     def get(self, request, slug):
263         course = get_object_or_404(Course, slug=slug)
264
265         file = course.get_file()
266
267         response = FileResponse(file, content_type='application/pdf')
268
269         return response
270
271
272 # COURSE DOWNLOAD VIEW
273
274 class CourseDownloadView(DashboardMixin, View):
275     def get(self, request, slug):
276         course = get_object_or_404(Course, slug=slug)
277
278         file = course.get_file()
279
280         response = FileResponse(file, content_type='application/pdf')
281
282         return response
283
284
285 # COURSE DOWNLOAD VIEW
286
287 class CourseDownloadView(DashboardMixin, View):
288     def get(self, request, slug):
289         course = get_object_or_404(Course, slug=slug)
290
291         file = course.get_file()
292
293         response = FileResponse(file, content_type='application/pdf')
294
295         return response
296
297
298 # COURSE DOWNLOAD VIEW
299
300 class CourseDownloadView(DashboardMixin, View):
301     def get(self, request, slug):
302         course = get_object_or_404(Course, slug=slug)
303
304         file = course.get_file()
305
306         response = FileResponse(file, content_type='application/pdf')
307
308         return response
309
310
311 # COURSE DOWNLOAD VIEW
312
313 class CourseDownloadView(DashboardMixin, View):
314     def get(self, request, slug):
315         course = get_object_or_404(Course, slug=slug)
316
317         file = course.get_file()
318
319         response = FileResponse(file, content_type='application/pdf')
320
321         return response
322
323
324 # COURSE DOWNLOAD VIEW
325
326 class CourseDownloadView(DashboardMixin, View):
327     def get(self, request, slug):
328         course = get_object_or_404(Course, slug=slug)
329
330         file = course.get_file()
331
332         response = FileResponse(file, content_type='application/pdf')
333
334         return response
335
336
337 # COURSE DOWNLOAD VIEW
338
339 class CourseDownloadView(DashboardMixin, View):
340     def get(self, request, slug):
341         course = get_object_or_404(Course, slug=slug)
342
343         file = course.get_file()
344
345         response = FileResponse(file, content_type='application/pdf')
346
347         return response
348
349
350 # COURSE DOWNLOAD VIEW
351
352 class CourseDownloadView(DashboardMixin, View):
353     def get(self, request, slug):
354         course = get_object_or_404(Course, slug=slug)
355
356         file = course.get_file()
357
358         response = FileResponse(file, content_type='application/pdf')
359
360         return response
361
362
363 # COURSE DOWNLOAD VIEW
364
365 class CourseDownloadView(DashboardMixin, View):
366     def get(self, request, slug):
367         course = get_object_or_404(Course, slug=slug)
368
369         file = course.get_file()
370
371         response = FileResponse(file, content_type='application/pdf')
372
373         return response
374
375
376 # COURSE DOWNLOAD VIEW
377
378 class CourseDownloadView(DashboardMixin, View):
379     def get(self, request, slug):
380         course = get_object_or_404(Course, slug=slug)
381
382         file = course.get_file()
383
384         response = FileResponse(file, content_type='application/pdf')
385
386         return response
387
388
389 # COURSE DOWNLOAD VIEW
390
391 class CourseDownloadView(DashboardMixin, View):
392     def get(self, request, slug):
393         course = get_object_or_404(Course, slug=slug)
394
395         file = course.get_file()
396
397         response = FileResponse(file, content_type='application/pdf')
398
399         return response
400
401
402 # COURSE DOWNLOAD VIEW
403
404 class CourseDownloadView(DashboardMixin, View):
405     def get(self, request, slug):
406         course = get_object_or_404(Course, slug=slug)
407
408         file = course.get_file()
409
410         response = FileResponse(file, content_type='application/pdf')
411
412         return response
413
414
415 # COURSE DOWNLOAD VIEW
416
417 class CourseDownloadView(DashboardMixin, View):
418     def get(self, request, slug):
419         course = get_object_or_404(Course, slug=slug)
420
421         file = course.get_file()
422
423         response = FileResponse(file, content_type='application/pdf')
424
425         return response
426
427
428 # COURSE DOWNLOAD VIEW
429
430 class CourseDownloadView(DashboardMixin, View):
431     def get(self, request, slug):
432         course = get_object_or_404(Course, slug=slug)
433
434         file = course.get_file()
435
436         response = FileResponse(file, content_type='application/pdf')
437
438         return response
439
440
441 # COURSE DOWNLOAD VIEW
442
443 class CourseDownloadView(DashboardMixin, View):
444     def get(self, request, slug):
445         course = get_object_or_404(Course, slug=slug)
446
447         file = course.get_file()
448
449         response = FileResponse(file, content_type='application/pdf')
450
451         return response
452
453
454 # COURSE DOWNLOAD VIEW
455
456 class CourseDownloadView(DashboardMixin, View):
457     def get(self, request, slug):
458         course = get_object_or_404(Course, slug=slug)
459
460         file = course.get_file()
461
462         response = FileResponse(file, content_type='application/pdf')
463
464         return response
465
466
467 # COURSE DOWNLOAD VIEW
468
469 class CourseDownloadView(DashboardMixin, View):
470     def get(self, request, slug):
471         course = get_object_or_404(Course, slug=slug)
472
473         file = course.get_file()
474
475         response = FileResponse(file, content_type='application/pdf')
476
477         return response
478
479
480 # COURSE DOWNLOAD VIEW
481
482 class CourseDownloadView(DashboardMixin, View):
483     def get(self, request, slug):
484         course = get_object_or_404(Course, slug=slug)
485
486         file = course.get_file()
487
488         response = FileResponse(file, content_type='application/pdf')
489
490         return response
491
492
493 # COURSE DOWNLOAD VIEW
494
495 class CourseDownloadView(DashboardMixin, View):
496     def get(self, request, slug):
497         course = get_object_or_404(Course, slug=slug)
498
499         file = course.get_file()
500
501         response = FileResponse(file, content_type='application/pdf')
502
503         return response
504
505
506 # COURSE DOWNLOAD VIEW
507
508 class CourseDownloadView(DashboardMixin, View):
509     def get(self, request, slug):
510         course = get_object_or_404(Course, slug=slug)
511
512         file = course.get_file()
513
514         response = FileResponse(file, content_type='application/pdf')
515
516         return response
517
518
519 # COURSE DOWNLOAD VIEW
520
521 class CourseDownloadView(DashboardMixin, View):
522     def get(self, request, slug):
523         course = get_object_or_404(Course, slug=slug)
524
525         file = course.get_file()
526
527         response = FileResponse(file, content_type='application/pdf')
528
529         return response
530
531
532 # COURSE DOWNLOAD VIEW
533
534 class CourseDownloadView(DashboardMixin, View):
535     def get(self, request, slug):
536         course = get_object_or_404(Course, slug=slug)
537
538         file = course.get_file()
539
540         response = FileResponse(file, content_type='application/pdf')
541
542         return response
543
544
545 # COURSE DOWNLOAD VIEW
546
547 class CourseDownloadView(DashboardMixin, View):
548     def get(self, request, slug):
549         course = get_object_or_404(Course, slug=slug)
550
551         file = course.get_file()
552
553         response = FileResponse(file, content_type='application/pdf')
554
555         return response
556
557
558 # COURSE DOWNLOAD VIEW
559
560 class CourseDownloadView(DashboardMixin, View):
561     def get(self, request, slug):
562         course = get_object_or_404(Course, slug=slug)
563
564         file = course.get_file()
565
566         response = FileResponse(file, content_type='application/pdf')
567
568         return response
569
570
571 # COURSE DOWNLOAD VIEW
572
573 class CourseDownloadView(DashboardMixin, View):
574     def get(self, request, slug):
575         course = get_object_or_404(Course, slug=slug)
576
577         file = course.get_file()
578
579         response = FileResponse(file, content_type='application/pdf')
580
581         return response
582
583
584 # COURSE DOWNLOAD VIEW
585
586 class CourseDownloadView(DashboardMixin, View):
587     def get(self, request, slug):
588         course = get_object_or_404(Course, slug=slug)
589
590         file = course.get_file()
591
592         response = FileResponse(file, content_type='application/pdf')
593
594         return response
595
596
597 # COURSE DOWNLOAD VIEW
598
599 class CourseDownloadView(DashboardMixin, View):
600     def get(self, request, slug):
601         course = get_object_or_404(Course, slug=slug)
602
603         file = course.get_file()
604
605         response = FileResponse(file, content_type='application/pdf')
606
607         return response
608
609
610 # COURSE DOWNLOAD VIEW
611
612 class CourseDownloadView(DashboardMixin, View):
613     def get(self, request, slug):
614         course = get_object_or_404(Course, slug=slug)
615
616         file = course.get_file()
617
618         response = FileResponse(file, content_type='application/pdf')
619
620         return response
621
622
623 # COURSE DOWNLOAD VIEW
624
625 class CourseDownloadView(DashboardMixin, View):
626     def get(self, request, slug):
627         course = get_object_or_404(Course, slug=slug)
628
629         file = course.get_file()
630
631         response = FileResponse(file, content_type='application/pdf')
632
633         return response
634
635
636 # COURSE DOWNLOAD VIEW
637
638 class CourseDownloadView(DashboardMixin, View):
639     def get(self, request, slug):
640         course = get_object_or_404(Course, slug=slug)
641
642         file = course.get_file()
643
644         response = FileResponse(file, content_type='application/pdf')
645
646         return response
647
648
649 # COURSE DOWNLOAD VIEW
650
651 class CourseDownloadView(DashboardMixin, View):
652     def get(self, request, slug):
653         course = get_object_or_404(Course, slug=slug)
654
655         file = course.get_file()
656
657         response = FileResponse(file, content_type='application/pdf')
658
659         return response
660
661
662 # COURSE DOWNLOAD VIEW
663
664 class CourseDownloadView(DashboardMixin, View):
665     def get(self, request, slug):
666         course = get_object_or_404(Course, slug=slug)
667
668         file = course.get_file()
669
670         response = FileResponse(file, content_type='application/pdf')
671
672         return response
673
674
675 # COURSE DOWNLOAD VIEW
676
677 class CourseDownloadView(DashboardMixin, View):
678     def get(self, request, slug):
679         course = get_object_or_404(Course, slug=slug)
680
681         file = course.get_file()
682
683         response = FileResponse(file, content_type='application/pdf')
684
685         return response
686
687
688 # COURSE DOWNLOAD VIEW
689
690 class CourseDownloadView(DashboardMixin, View):
691     def get(self, request, slug):
692         course = get_object_or_404(Course, slug=slug)
693
694         file = course.get_file()
695
696         response = FileResponse(file, content_type='application/pdf')
697
698         return response
699
700
701 # COURSE DOWNLOAD VIEW
702
703 class CourseDownloadView(DashboardMixin, View):
704     def get(self, request, slug):
705         course = get_object_or_404(Course, slug=slug)
706
707         file = course.get_file()
708
709         response = FileResponse(file, content_type='application/pdf')
710
711         return response
712
713
714 # COURSE DOWNLOAD VIEW
715
716 class CourseDownloadView(DashboardMixin, View):
717     def get(self, request, slug):
718         course = get_object_or_404(Course, slug=slug)
719
720         file = course.get_file()
721
722         response = FileResponse(file, content_type='application/pdf')
723
724         return response
725
726
727 # COURSE DOWNLOAD VIEW
728
729 class CourseDownloadView(DashboardMixin, View):
730     def get(self, request, slug):
731         course = get_object_or_404(Course, slug=slug)
732
733         file = course.get_file()
734
735         response = FileResponse(file, content_type='application/pdf')
736
737         return response
738
739
740 # COURSE DOWNLOAD VIEW
741
742 class CourseDownloadView(DashboardMixin, View):
743     def get(self, request, slug):
744         course = get_object_or_404(Course, slug=slug)
745
746         file = course.get_file()
747
748         response = FileResponse(file, content_type='application/pdf')
749
750         return response
751
752
753 # COURSE DOWNLOAD VIEW
754
755 class CourseDownloadView(DashboardMixin, View):
756     def get(self, request, slug):
757         course = get_object_or_404(Course, slug=slug)
758
759         file = course.get_file()
760
761         response = FileResponse(file, content_type='application/pdf')
762
763         return response
764
765
766 # COURSE DOWNLOAD VIEW
767
768 class CourseDownloadView(DashboardMixin, View):
769     def get(self, request, slug):
770         course = get_object_or_404(Course, slug=slug)
771
772         file = course.get_file()
773
774         response = FileResponse(file, content_type='application/pdf')
775
776         return response
777
778
779 # COURSE DOWNLOAD VIEW
780
781 class CourseDownloadView(DashboardMixin, View):
782     def get(self, request, slug):
783         course = get_object_or_404(Course, slug=slug)
784
785         file = course.get_file()
786
787         response = FileResponse(file, content_type='application/pdf')
788
789         return response
790
791
792 # COURSE DOWNLOAD VIEW
793
794 class CourseDownloadView(DashboardMixin, View):
795     def get(self, request, slug):
796         course = get_object_or_404(Course, slug=slug)
797
798         file = course.get_file()
799
800         response = FileResponse(file, content_type='application/pdf')
801
802         return response
803
804
805 # COURSE DOWNLOAD VIEW
806
807 class CourseDownloadView(DashboardMixin, View):
808     def get(self, request, slug):
809         course = get_object_or_404(Course, slug=slug)
810
811         file = course.get_file()
812
813         response = FileResponse(file, content_type='application/pdf')
814
815         return response
816
817
818 # COURSE DOWNLOAD VIEW
819
820 class CourseDownloadView(DashboardMixin, View):
821     def get(self, request, slug):
822         course = get_object_or_404(Course, slug=slug)
823
824         file = course.get_file()
825
826         response = FileResponse(file, content_type='application/pdf')
827
828         return response
829
830
831 # COURSE DOWNLOAD VIEW
832
833 class CourseDownloadView(DashboardMixin, View):
834     def get(self, request, slug):
835         course = get_object_or_404(Course, slug=slug)
836
837         file = course.get_file()
838
839         response = FileResponse(file, content_type='application/pdf')
840
841         return response
842
843
844 # COURSE DOWNLOAD VIEW
845
846 class CourseDownloadView(DashboardMixin, View):
847     def get(self, request, slug):
848         course = get_object_or_404(Course, slug=slug)
849
850         file = course.get_file()
851
852         response = FileResponse(file, content_type='application/pdf')
853
854         return response
855
856
857 # COURSE DOWNLOAD VIEW
858
859 class CourseDownloadView(DashboardMixin, View):
860     def get(self, request, slug):
861         course = get_object_or_404(Course, slug=slug)
862
863         file = course.get_file()
864
865         response = FileResponse(file, content_type='application/pdf')
866
867         return response
868
869
870 # COURSE DOWNLOAD VIEW
871
872 class CourseDownloadView(DashboardMixin, View):
873     def get(self, request, slug):
874         course = get_object_or_404(Course, slug=slug)
875
876         file = course.get_file()
877
878         response = FileResponse(file, content_type='application/pdf')
879
880         return response
881
882
883 # COURSE DOWNLOAD VIEW
884
885 class CourseDownloadView(DashboardMixin, View):
886     def get(self, request, slug):
887         course = get_object_or_404(Course, slug=slug)
888
889         file = course.get_file()
890
891         response = FileResponse(file, content_type='application/pdf')
892
893         return response
894
895
896 # COURSE DOWNLOAD VIEW
897
898 class CourseDownloadView(DashboardMixin, View):
899     def get(self, request, slug):
900         course = get_object_or_404(Course, slug=slug)
901
902         file = course.get_file()
903
904         response = FileResponse(file, content_type='application/pdf')
905
906         return response
907
908
909 # COURSE DOWNLOAD VIEW
910
911 class CourseDownloadView(DashboardMixin, View):
912     def get(self, request, slug):
913         course = get_object_or_404(Course, slug=slug)
914
915         file = course.get_file()
916
917         response = FileResponse(file, content_type='application/pdf')
918
919         return response
920
921
922 # COURSE DOWNLOAD VIEW
923
924 class CourseDownloadView(DashboardMixin, View):
925     def get(self, request, slug):
926         course = get_object_or_404(Course, slug=slug)
927
928         file = course.get_file()
929
930         response = FileResponse(file, content_type='application/pdf')
931
932         return response
933
934
935 # COURSE DOWNLOAD VIEW
936
937 class CourseDownloadView(DashboardMixin, View):
938     def get(self, request, slug):
939         course = get_object_or_404(Course, slug=slug)
940
941         file = course.get_file()
942
943         response = FileResponse(file, content_type='application/pdf')
944
945         return response
946
947
948 # COURSE DOWNLOAD VIEW
949
950 class CourseDownloadView(DashboardMixin, View):
951     def get(self, request, slug):
952         course = get_object_or_404(Course, slug=slug)
953
954         file = course.get_file()
955
956         response = FileResponse(file, content_type='application/pdf')
957
958         return response
959
960
961 # COURSE DOWNLOAD VIEW
962
963 class CourseDownloadView(DashboardMixin, View):
964     def get(self, request, slug):
965         course = get_object_or_404(Course, slug=slug)
966
967         file = course.get_file()
968
969         response = FileResponse(file, content_type='application/pdf')
970
971         return response
972
973
974 # COURSE DOWNLOAD VIEW
975
976 class CourseDownloadView(DashboardMixin, View):
977     def get(self, request, slug):
978         course = get_object_or_404(Course, slug=slug)
979
980         file = course.get_file()
981
982         response = FileResponse(file, content_type='application/pdf')
983
984         return response
985
986
987 # COURSE DOWNLOAD VIEW
988
989 class CourseDownloadView(DashboardMixin, View):
990     def get(self, request, slug):
991         course = get_object_or_404(Course, slug=slug)
992
993         file = course.get_file()
994
995         response = FileResponse(file, content_type='application/pdf')
996
997         return response
998
999
1000 # COURSE DOWNLOAD VIEW
1001
1002 class CourseDownloadView(DashboardMixin, View):
1003     def get(self, request, slug):
1004         course = get_object_or_404(Course, slug=slug)
1005
1006         file = course.get_file()
1007
1008         response = FileResponse(file, content_type='application/pdf')
1009
1010         return response
1011
1012
1013 # COURSE DOWNLOAD VIEW
1014
1015 class CourseDownloadView(DashboardMixin, View):
1016     def get(self, request, slug):
1017         course = get_object_or_404(Course, slug=slug)
1018
1019         file = course.get_file()
1020
1021         response = FileResponse(file, content_type='application/pdf')
1022
1023         return response
1024
1025
1026 # COURSE DOWNLOAD VIEW
1027
1028 class CourseDownloadView(DashboardMixin, View):
1029     def get(self, request, slug):
1030         course = get_object_or_404(Course, slug=slug)
1031
1032         file = course.get_file()
1033
1034         response = FileResponse(file, content_type='application/pdf')
1035
1036         return response
1037
1038
1039 # COURSE DOWNLOAD VIEW
1040
1041 class CourseDownloadView(DashboardMixin, View):
1042     def get(self, request, slug):
1043         course = get_object_or_404(Course, slug=slug)
1044
1045         file = course.get_file()
1046
1047         response = FileResponse(file, content_type='application/pdf')
1048
1049         return response
1050
1051
1052 # COURSE DOWNLOAD VIEW
1053
1054 class CourseDownloadView(DashboardMixin, View):
1055     def get(self, request, slug):
1056         course = get_object_or_404(Course, slug=slug)
1057
1058         file = course.get_file()
1059
1060         response = FileResponse(file, content_type='application/pdf')
1061
1062         return response
1063
1064
1065 # COURSE DOWNLOAD VIEW
1066
1067 class CourseDownloadView(DashboardMixin, View):
1068     def get(self, request, slug):
1069         course = get_object_or_404(Course, slug=slug)
1070
1071         file = course.get_file()
1072
1073         response = FileResponse(file, content_type='application/pdf')
1074
1075         return response
1076
1077
1078 # COURSE DOWNLOAD VIEW
1079
1080 class CourseDownloadView(DashboardMixin, View):
1081     def get(self, request, slug):
1082         course = get_object_or_404(Course, slug=slug)
1083
1084         file = course.get_file()
1085
1086         response = FileResponse(file, content_type='application/pdf')
1087
1088         return response
1089
1090
1091 # COURSE DOWNLOAD VIEW
1092
1093 class CourseDownloadView(DashboardMixin, View):
1094     def get(self, request, slug):
1095         course = get_object_or_404(Course, slug=slug)
1096
1097         file = course.get_file()
1098
1099         response = FileResponse(file, content_type='application/pdf')
1100
1101         return response
1102
1103
1104 # COURSE DOWNLOAD VIEW
1105
1106 class CourseDownloadView(DashboardMixin, View):
1107     def get(self, request, slug):
1108         course = get_object_or_404(Course, slug=slug)
1109
1110         file = course.get_file()
1111
1112         response = FileResponse(file, content_type='application/pdf')
1113
1114         return response
1115
1116
1117 # COURSE DOWNLOAD VIEW
1118
1119 class CourseDownloadView(DashboardMixin, View):
1120     def get(self, request, slug):
1121         course = get_object_or_404(Course, slug=slug)
1122
1123         file = course.get_file()
1124
1125         response = FileResponse(file, content_type='application/pdf')
1126
1127         return response
1128
1129
1130 # COURSE DOWNLOAD VIEW
1131
1132 class CourseDownloadView(DashboardMixin, View):
1133     def get(self, request, slug):
1134         course = get_object_or_404(Course, slug=slug)
1135
1136         file = course.get_file()
1137
1138         response = FileResponse(file, content_type='application/pdf')
1139
1140         return response
1141
1142
1143 # COURSE DOWNLOAD VIEW
1144
1145 class CourseDownloadView(DashboardMixin, View):
1146     def get(self, request, slug):
1147         course = get_object_or_404(Course, slug=slug)
1148
1149         file = course.get_file()
1150
1151         response = FileResponse(file, content_type='application/pdf')
1152
1153         return response
1154
1155
1156 # COURSE DOWNLOAD VIEW
1157
1158 class CourseDownloadView(DashboardMixin, View):
1159     def get(self, request, slug):
1160         course = get_object_or_404(Course, slug=slug)
1161
1162         file = course.get_file()
1163
1164         response = FileResponse(file, content_type='application/pdf')
1165
1166         return response
1167
1168
1169 # COURSE DOWNLOAD VIEW
1170
1171 class CourseDownloadView(DashboardMixin, View):
1172     def get(self, request, slug):
1173         course = get_object_or_404(Course, slug=slug)
1174
1175         file = course.get_file()
1176
1177         response = FileResponse(file, content_type='application/pdf')
1178
1179         return response
1180
1181
1182 # COURSE DOWNLOAD VIEW
1183
1184 class CourseDownloadView(DashboardMixin, View):
1185     def get(self, request, slug):
1186         course = get_object_or_404(Course, slug=slug)
1187
1188         file = course.get_file()
1189
1190         response = FileResponse(file, content_type='application/pdf')
1191
1192         return response
1193
1194
1195 # COURSE DOWNLOAD VIEW
1196
1197 class CourseDownloadView(DashboardMixin, View):
1198     def get(self, request, slug):
1199         course = get_object_or_404(Course, slug=slug)
1200
1201         file = course.get_file()
1202
1203         response = FileResponse(file, content_type='application/pdf')
1204
1205         return response
1206
1207
1208 # COURSE DOWNLOAD VIEW
1209
1210 class CourseDownloadView(DashboardMixin, View):
1211     def get(self, request, slug):
1212         course = get_object_or_404(Course, slug=slug)
1213
1214         file = course.get_file()
1215
1216         response = FileResponse(file, content_type='application/pdf')
1217
1218         return response
1219
1220
1221 # COURSE DOWNLOAD VIEW
1222
1223 class CourseDownloadView(DashboardMixin, View):
1224     def get(self, request, slug):
1225         course = get_object_or_404(Course, slug=slug)
1226
1227         file = course.get_file()
1
```

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 `List-View` ([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 }}{{ fun-
    cionario.nome }}</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`.

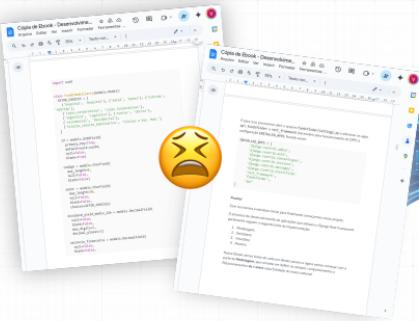
Assim, ele não mais será um simples `input`, mas será renderizado como um `<textarea></textarea>` no nosso `template`!



*Estou desenvolvendo o **DevBook**, uma plataforma que usa IA para gerar ebooks técnicos profissionais. Não deixe de conferir clicando no botão abaixo!*

Crie Ebooks técnicos incríveis em minutos com IA

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



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

Syntax Highlight

Adicione Banners Promocionais

Edite em Markdown em Tempo Real

Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS

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 URL-Conf, 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! 😊

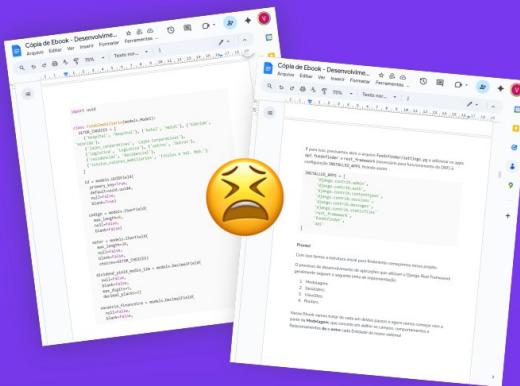
Não se esqueça de conferir!



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



Adicione Banners Promocionais



• Infográficos feitos para...

Deixe que nossa IA faça o trabalho pesado



 Edite em Markdown em Tempo Real

TESTE AGORA



 PRIMEIRO CAPÍTULO 100% GRÁTIS