



A CAMADA TEMPLATE DO DJANGO (PYTHON)

Nesse ebook, vamos tratar da camada _Template_ do Django. Vamos aprender como utilizar, configurar e estender os templates do Django, como utilizar filtros e tags do Django e como criar nossas tags e filtros customizados!

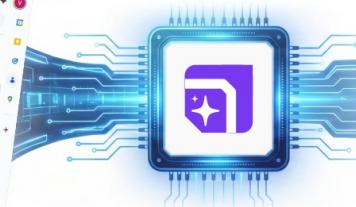
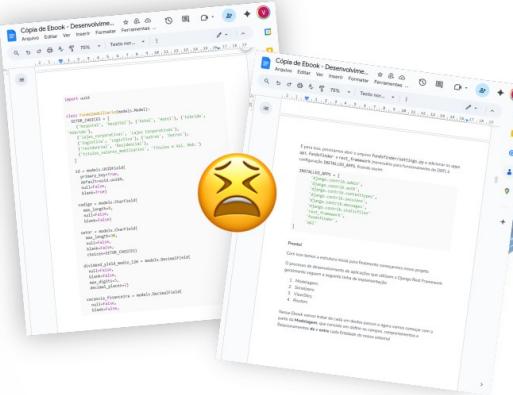
Gere ebooks como este com



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) *Django Template Language (DTL)* permanece estável. Considere alternativas modernas como **Jinja2** para projetos complexos.

Olá sr. Pythonista!

Finalmente chegamos ao nosso último *post* sobre Django (*pelo menos dessa série - com certeza ainda teremos **muito conteúdo sobre Django**, ok?!*)

Primeiramente, quero garantir que você está no ponto certo.

Você já leu os outros *posts* da série, certo!? Se não, então já se liga:

[Django: Introdução ao framework](#)

[Django: A Camada Model](#)

[Django: A Camada View](#)

Agora sim estamos prontos para começar!

Nesse *post*, vamos tratar da camada que dá a “cara” à nossa aplicação: a Camada **Template!**

É nela que se encontra o código Python, responsável por renderizar nossas páginas web, e os arquivos HTML, CSS e Javascript que darão vida à nossa aplicação!

Vamos tratar das configurações necessárias para fazer essa camada funcionar corretamente, das ferramentas *built-ins* do Django para construção de *templates*, da *Django Template Language*: sua sintaxe, *tags*, *filters*, e muito mais!

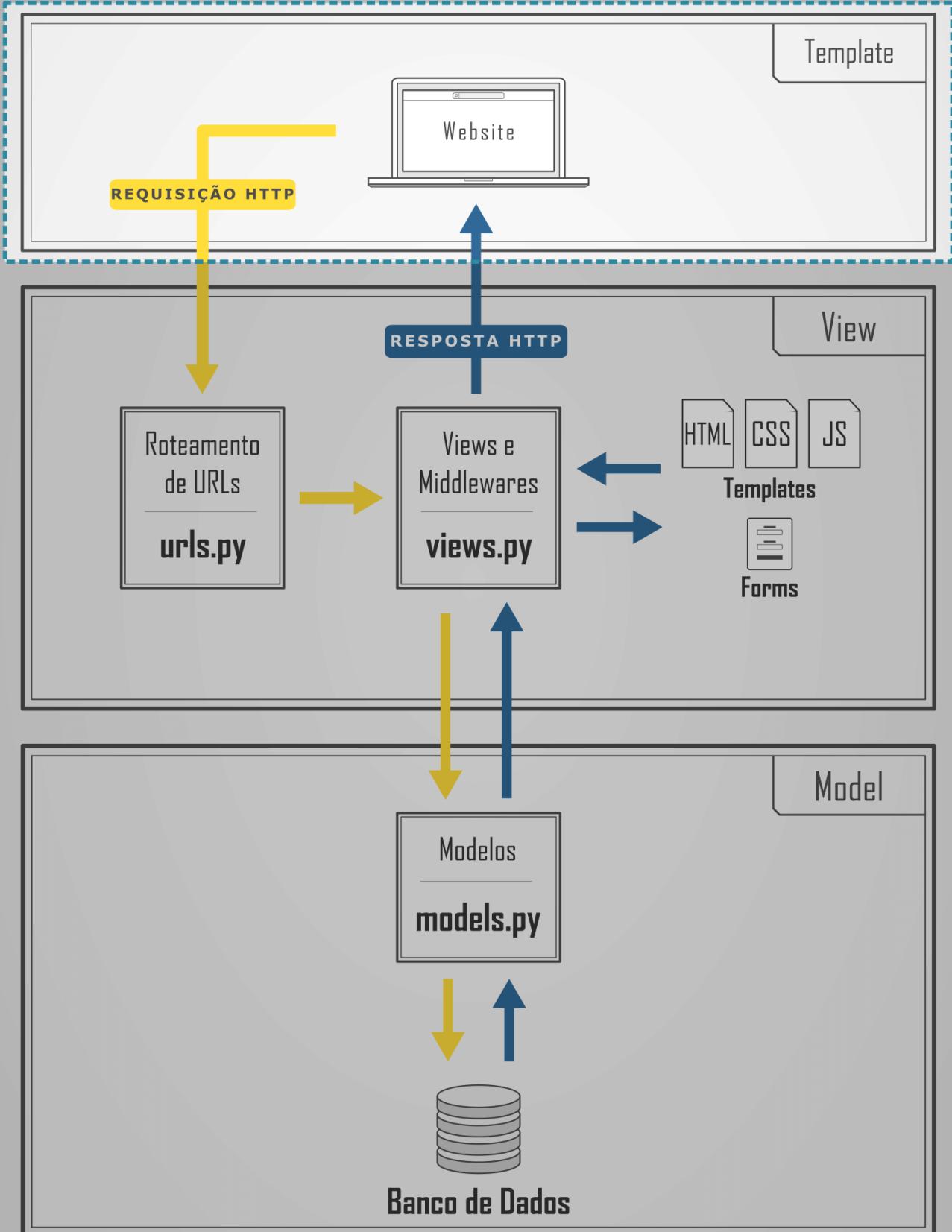
Está curioso?

Então **VAMOS NESSA!!!**

Onde estamos...

Primeiro, vamos relembrar onde estamos no fluxo requisição/resposta do nosso servidor Django:

ARQUITETURA DO django



Ou seja, estamos na camada que faz a interface do nosso código Python/Django com o usuário, interagindo, trocando informações, captando dados de *input* e gerando dados de *output*.

Caso você não tenha o código que estamos desenvolvendo nessa série de *posts* (o conhecido **Hello World**), [baixe-o aqui][helloworld-download]!

SPOILER ALERT: Nesse *post* vamos concentrar nossos esforços em entender a camada de templates para **construção de páginas**. Nesse momento, não vamos focar na implementação da lógica por trás da Engine de templates, pois acredito que é algo que dificilmente você se verá fazendo, ok?! Se você, mesmo assim, sentir necessidade de uma explicação ou tirar uma dúvida sobre a API de Templates do Django, use o **box** de comentários ali embaixo para eu saber como te ajudar!

Para relembrar, estamos desenvolvendo um **Sistema de Gerenciamento dos Funcionários** da sua empresa.

Queremos que, ao final do *post*, tenhamos páginas customizadas para “mostrar ao seu chefe” seu exímio trabalho! 😊

Vamos começar pelo começo: **o que é um Template?**

Definição de Template

Basicamente, um *template* é um arquivo de texto que pode ser transformado em outro arquivo (um arquivo HTML, um CSS, um CSV, etc...).

Um *template* contém:

- **Variáveis** que podem ser substituídas por valores, a partir do processamento por uma *Engine de Templates* (núcleo, *core*, “motor” de *templates*).
- **Tags** que controlam a lógica do *template*.

- **Filtros** que adicionam funcionalidades ao *template*.

Por exemplo, abaixo está representado um *template* mínimo que demonstra alguns conceitos básicos:

Exemplo prático com DTL:

```
{% raw %}{# base.html contém o template que usaremos como esqueleto #}
      {% endraw %}

{% raw %}{% extends "base.html" %}{% endraw %}

{% raw %}{% block conteudo %}{% endraw %}
<h1>{% raw %}{{ section.title }}{% endraw %}</h1>

{% raw %}{% for funcionario in funcionarios %}{% endraw %}
<h2>
  <a href="{% raw %}{% url 'website:funcionario_detalhe'
    pk=funcionario.id %}{% endraw %}">
    {% raw %}{{ funcionario.nome|upper }}{% endraw %}
  </a>
</h2>
{% raw %}{% endfor %}{% endraw %}
{% raw %}{% endblock %}{% endraw %}
```

Alguns pontos importantes:

- **Linha 1:** Utilizamos comentário com a tag


```
{% raw %}{# comentário #}{% endraw %}.
```
- **Linha 2:** Utilizamos


```
{% raw %}{% extends "base.html" %}{% endraw %}
```

 para estender de outro *template*, ou seja, utilizá-lo como base, passando o caminho para ele.
- **Linha 4:** Podemos facilitar a organização do *template*, criando blocos com


```
{% raw %}{% block <nome_do_bloco> %}{% endblock %}{% endraw %}.
```

- **Linha 5:** Podemos interpolar código vindo do servidor com o conteúdo do nosso *template* com `{% raw %}{{ secao.titulo }}{% endraw %}` - dessa forma, estamos acessando o atributo `titulo` do objeto `secao` (que deve estar no **Contexto** da requisição).
- **Linha 7:** É possível iterar sobre objetos de uma lista através da tag `{% raw %}{% for objeto in lista %}{% endfor %}{% endraw %}`.
- **Linha 10:** Podemos utilizar **filtros** para aplicar alguma função à algum conteúdo. Nesse exemplo, estamos aplicando o filtro `upper`, que transforma todos os caracteres da *string* em maiúsculos, no conteúdo de `funcionario.nome`. Também é possível encadear filtros, por exemplo:
`{% raw %}{{ funcionario.nome|upper|cut:' ' }}{% endraw %}`

Django criou uma linguagem que contém todos esses elementos.

Chamaram-na de **DTL** - *Django Template Language!* Veremos mais dela ali embaixo!



Django 5.1: Suporte nativo para múltiplos backends de template. **Jinja2** é totalmente suportado como alternativa ao DTL, oferecendo mais flexibilidade e performance.

Para utilizar os *templates* do Django, é necessário primeiro **configurar sua utilização**.

E é isso que veremos agora!

Configuração

Se você já deu uma espiada no nosso arquivo de configurações, o `settings.py`, você já deve ter visto a seguinte configuração:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {}
    },
]
```

Mas você já se perguntou **o que essa configuração quer dizer?**

Nela:

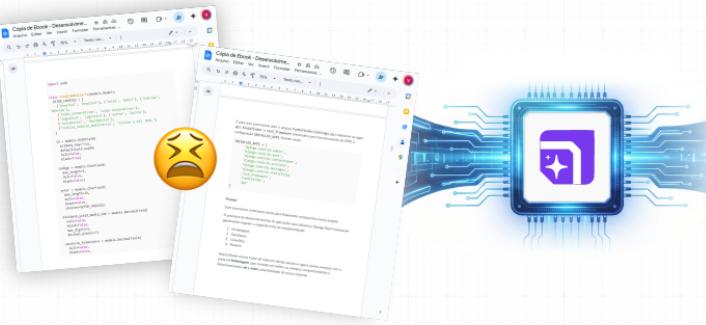
- `BACKEND` é o caminho para uma classe que implementa a API de *templates* do Django.
- `DIRS` define uma lista de diretórios onde o Django deve procurar por *templates*. A ordem da lista define a ordem de busca.
- `APP_DIRS` define se o Django deve procurar por *templates* dentro dos diretórios dos *apps* instalados em `INSTALLED_APPS`.
- `OPTIONS` contém configurações específicas do `BACKEND` escolhido, ou seja, dependendo do *backend* de *templates* que você escolher, você poderá configurá-lo utilizando o `OPTIONS`.

Por ora, vamos utilizar as configurações padrão “de fábrica” pois elas já nos atendem!

 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!

 **Ebookr**

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

Agora, vamos ver sobre a tal *Django Template Language*!

Django Template Language

A *DTL* é a linguagem padrão de *templates* do Django. Ela é simples, porém poderosa.

Dando uma olhada na sua [documentação](#), podemos ver a **filosofia** da **DTL** (traduzido):

Se você tem alguma experiência em programação, ou se você está acostumado com linguagens que misturam código de programação diretamente no HTML, você deve ter em mente que o sistema de templates do Django não é simplesmente código Python embutido no HTML. Isto é: o sistema de templates foi desenhado para ser a apresentação, e não para conter lógica!

Se você vem de outra linguagem de programação deve ter tido contato com o seguinte tipo de construção: código de programação adicionado diretamente no código HTML (como PHP).

Isto é o **terror** dos *designers*!

Ponha-se no lugar de um *designer* que não sabe nada sobre programação. Agora imagina você tendo que dar manutenção nos estilos de uma página **LOTADA** de código de programação?!

Complicado, hein?! 😱

Agora, nada melhor para aprender sobre a *DTL* do que botando a mão na massa e melhorando as páginas da nossa aplicação, né?!

Primeiro, vamos começar fazendo algumas alterações na estrutura do nosso projeto!

Observação: nesse post eu vou utilizar o [Bootstrap 4](#) para dar um “tapa no visual”. Se surgir alguma dúvida, não se esqueça do box de comentário ao final do post, ok?!

Alterações no projeto

Para dar continuidade no nosso projeto, vamos fazer as seguintes alterações:

- Crie a pasta `_layouts` dentro de `website/templates/website` a fim de separar os *templates-base* (*layouts*) da nossa aplicação.
- Crie o arquivo `base.html` dentro de `/layouts`.
- Crie a pasta `static` dentro de `website` para guardar os arquivos estáticos que vamos utilizar (arquivos de estilo CSS, arquivos Javascript, imagens, fontes, etc...). Crie também a pasta `website` dentro dela.
- Dentro dessa pasta (`/static/website/`), crie as pastas `css`, `img` e `js` (arquivos .css, imagens e Javascript).
- Adicione o parâmetro `name=' '` às URLs no arquivo `website/urls.py`, da seguinte forma:

```

urlpatterns = [
    # '/'
    path('',
        IndexTemplateView.as_view(),
        name="index" # <<< Adicionar
    ),

    # '/funcionario/cadastrar'
    path('funcionario/cadastrar',
        FuncionarioCreateView.as_view(),
        name="cadastra_funcionario" # <<< Adicionar
    ),

    # '/funcionarios'
    path('funcionarios/',
        FuncionarioListView.as_view(),
        name="lista_funcionarios" # <<< Adicionar
    ),

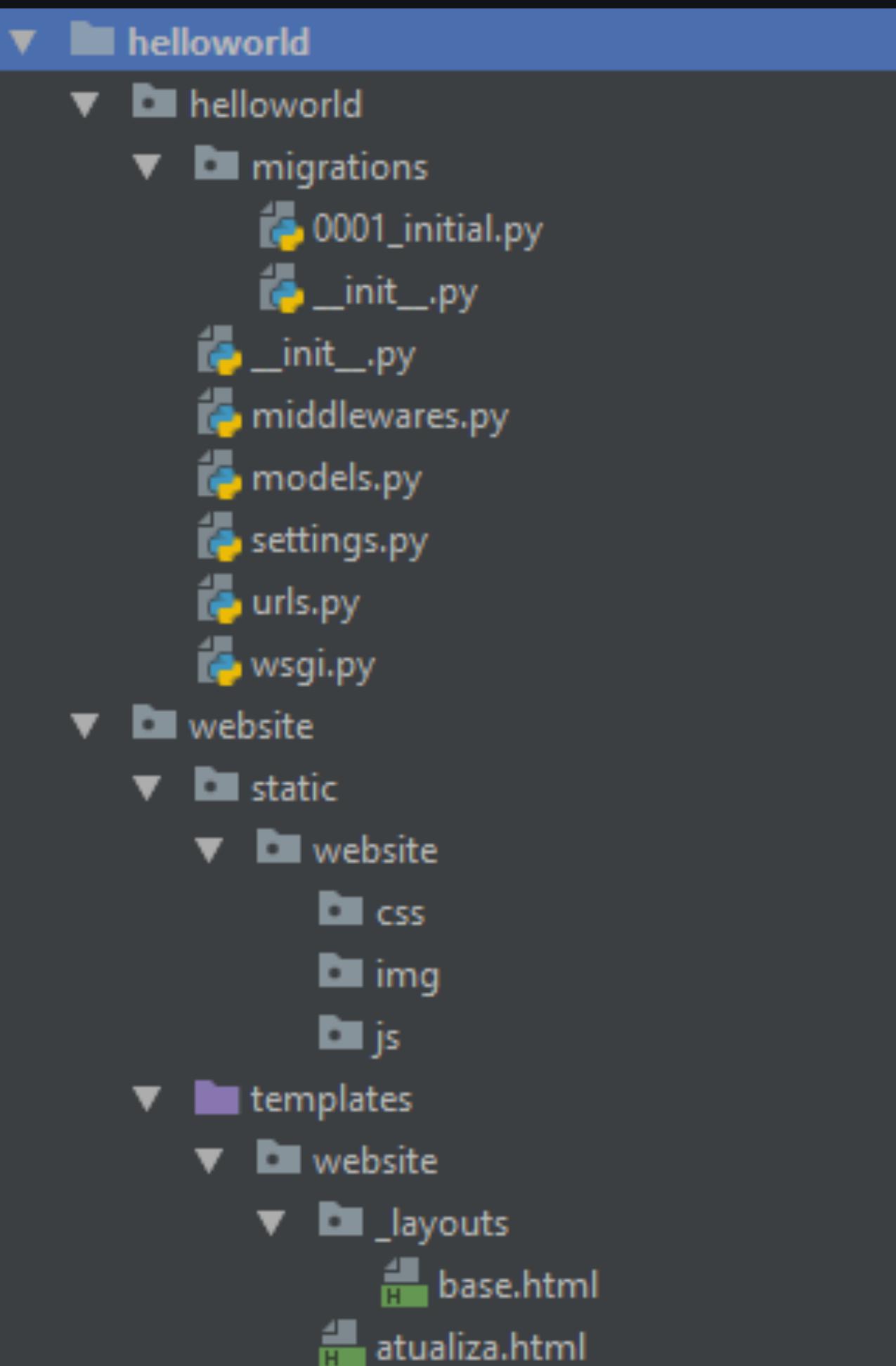
    # '/funcionario/{pk}'
    path('funcionario/<pk>',
        FuncionarioUpdateView.as_view(),
        name="atualiza_funcionario" # <<< Adicionar
    ),

    # '/funcionarios/excluir/{pk}'
    path('funcionario/excluir/<pk>',
        FuncionarioDeleteView.as_view(),
        name="deleta_funcionario" # <<< Adicionar
    ),
]

```

Observação: Nós criamos uma pasta com o nome do app (`website`, no caso) dentro das pastas `static` e `templates` para que o Django crie o namespace do app. Dessa forma, o Django entende onde buscar os recursos quando você precisar!

Dessa forma, devemos estar com a estrutura da seguinte forma:



Com a estrutura acima, vamos começar pelo *template-base* do app **website**!

Template-base

Nosso *template* que servirá de esqueleto deve conter o código HTML que irá se repetir em todas as páginas.

Devemos colocar nele os trechos de código mais comuns das páginas HTML.

Por exemplo, toda página HTML:

- Deve ter as *tags*: `<html></html>`, `<head></head>` e `<body></body>`.
- Deve ter os *links* para os arquivos estáticos: `<link></link>` e `<script></script>`.
- Quaisquer outros trechos de código que se repitam em nossas páginas.

Você pode fazer o *download* dos arquivos necessários para o nosso projeto [aqui \(Bootstrap\)](#) e [aqui \(jQuery\)](#), que é uma dependência do *Bootstrap*, **ou** utilizar os arquivos que eu já baixei e estão na pasta `website/static/`.

Faça isso para todos as bibliotecas externas que queira utilizar (ou utilize um **CDN - Content Delivery Network**).

Ok! Agora, com os arquivos devidamente colocados na pasta `/static/`, podemos começar com:

```

<!DOCTYPE html>
<html>
{%
    raw %
}{%
    load static %
}{%
    endraw %
}
<head>
<!-- Title -->
<title>
{%
    raw %
}{%
    block title %
}Sistema de Gerenciamento de
Funcionários{%
    endblock %
}{%
    endraw %
}
</title>

<!-- Favicon -->
<link rel="shortcut icon" href="{%
    raw %
}{%
    static 'website/img/
favicon.png' %
}{%
    endraw %
}" type="image/png">

<!-- Arquivos CSS -->
<link rel="stylesheet" href="{%
    raw %
}{%
    static 'website/css/
bootstrap.min.css' %
}{%
    endraw %
}">
<link rel="stylesheet" href="{%
    raw %
}{%
    static 'website/css/
master.css' %
}{%
    endraw %
}">

<!-- Bloco de Estilos -->
{%
    raw %
}{%
    block styles %
}{%
    endblock %
}{%
    endraw %
}
</head>

<body>
<!-- Navbar -->
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="{%
        raw %
}{%
        url 'website:index'
    %
}{%
        endraw %
}">
        
    </a>
    <button class="navbar-toggler" type="button"
        data-toggle="collapse" data-target="#conteudo-navbar"
        aria-controls="conteudo-navbar" aria-expanded="false"
        aria-label="Ativar navegação">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="conteudo-navbar">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">

```

```

        <a class="nav-link" href="{% raw %}{% url
'website:index' %}{% endraw %}">
            Página Inicial
        </a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="{% raw %}{% url
'website:lista_funcionario' %}{% endraw %}">
            Funcionários
        </a>
    </li>
</ul>
</div>
</nav>

<!-- Bloco de conteúdo -->
{% raw %}{% block conteudo %}{% endblock %}{% endraw %}

<!-- Arquivos necessários para o Bootstrap -->
<script src="{% raw %}{% static 'website/js/jquery.min.js' %}{%
endraw %}"></script>
<script src="{% raw %}{% static 'website/js/bootstrap.min.js' %}{%
endraw %}"></script>

<!-- Bloco de scripts -->
{% raw %}{% block scripts %}{% endblock %}{% endraw %}

<!-- Scripts.js -->
<script
    src="{% raw %}{% static 'website/js/scripts.js' %}{% endraw %
}"></script>
</body>
</html>

```

E vamos as explicações:

- `<!DOCTYPE html>` serve para informar ao *browser* do usuário que se trata de uma página HTML5.
- Para que o Django possa carregar dinamicamente os arquivos estáticos do site, utilizamos a tag `static`. Ela vai fazer a busca do arquivo que você

quer e fazer a conversão dos *links* corretamente. Para utilizá-la, é necessário primeiro carregá-la. Fazemos isso com `{% raw %}{% load <modulo> %}`
`{% endraw %}`. Após seu carregamento, utilizamos a tag `{% raw %}{%`
`static 'caminho/para/arquivo' %}{% endraw %}`, passando como parâmetro a localização relativa à pasta `/static/`.

- Podemos definir quaisquer blocos no nosso *template* com a tag `{% raw %}`
`{% block nome_do_bloco %}{% endblock %}{% endraw %}`. Fazemos isso para organizar melhor as páginas que irão estender desse *template*. Podemos passar um valor padrão dentro do bloco (igual está sendo utilizado na **linha 6**) - dessa forma caso não seja definido nenhum valor no *template* filho - é aplicado o valor padrão.
- Colocamos nesse *template* os arquivos necessários para o funcionamento do Bootstrap, isto é: o jQuery, o CSS e Javascript do Bootstrap.
- O *link* para outras páginas da nossa aplicação é feito utilizando-se a tag `{%`
`raw %}{% url 'nome_da_view' parametros... %}{% endraw %}`. Dessa forma, deixamos que o Django cuide da conversão para URLs válidas!
- O conjunto de tags `<nav></nav>` definem a barra superior de navegação com os *links* para as páginas da aplicação. Esse também é um trecho de código presente em todas as páginas, por isso, adicionamos ao *template*. ([Documentação da Navbar - Bootstrap](#))

E pronto! Temos um *template* base!

Agora, vamos customizar a tela principal da nossa aplicação: a **index.html**!

Página Inicial

Template: `website/index.html`

Nossa tela inicial tem o objetivo de apenas mostrar as opções disponíveis ao usuário, que são:

- *Link* para a página de cadastro de novos Funcionários.
- *Link* para a página de listagem de Funcionários.

Primeiramente, precisamos dizer ao Django que queremos utilizar o *template* que definimos acima como base.

Para isso, utilizamos a *tag* `{% extends "caminho/para/template" %}`, que serve para que um *template* estenda de outro.

Com isso, podemos fazer:

```

<!-- Estendemos do template base -->
{% raw %}{% extends "website/_layouts/base.html" %}{% endraw %}

<!-- Bloco que define o <title></title> da nossa página -->
{% raw %}{% block title %}Página Inicial{% endblock %}{% endraw %}

<!-- Bloco de conteúdo da nossa página -->
{% raw %}{% block conteudo %}{% endraw %}
<div class="container">
    <div class="row">
        <div class="col-lg-6 col-md-6 col-sm-6 col-xs-12">
            <div class="card">
                <div class="card-body">
                    <h5 class="card-title">Cadastrar Funcionário</h5>
                    <p class="card-text">
                        Cadastre aqui um novo <code>Funcionário</code>.
                    </p>
                    <a href="{% raw %}{% url 'website:cadastra_funcionario' %}{% endraw %}">
                        class="btn btn-primary">
                            Novo Funcionário
                    </a>
                </div>
            </div>
        </div>
        <div class="col-lg-6 col-md-6 col-sm-6 col-xs-12">
            <div class="card">
                <div class="card-body">
                    <h5 class="card-title">Lista de Funcionários</h5>
                    <p class="card-text">
                        Veja aqui a lista de <code>Funcionários</code> cadastrados.
                    </p>
                    <a href="{% raw %}{% url 'website:lista_funcionarios' %}{% endraw %}">
                        class="btn btn-primary">
                            Vá para Lista
                    </a>
                </div>
            </div>
        </div>
    </div>
</div>

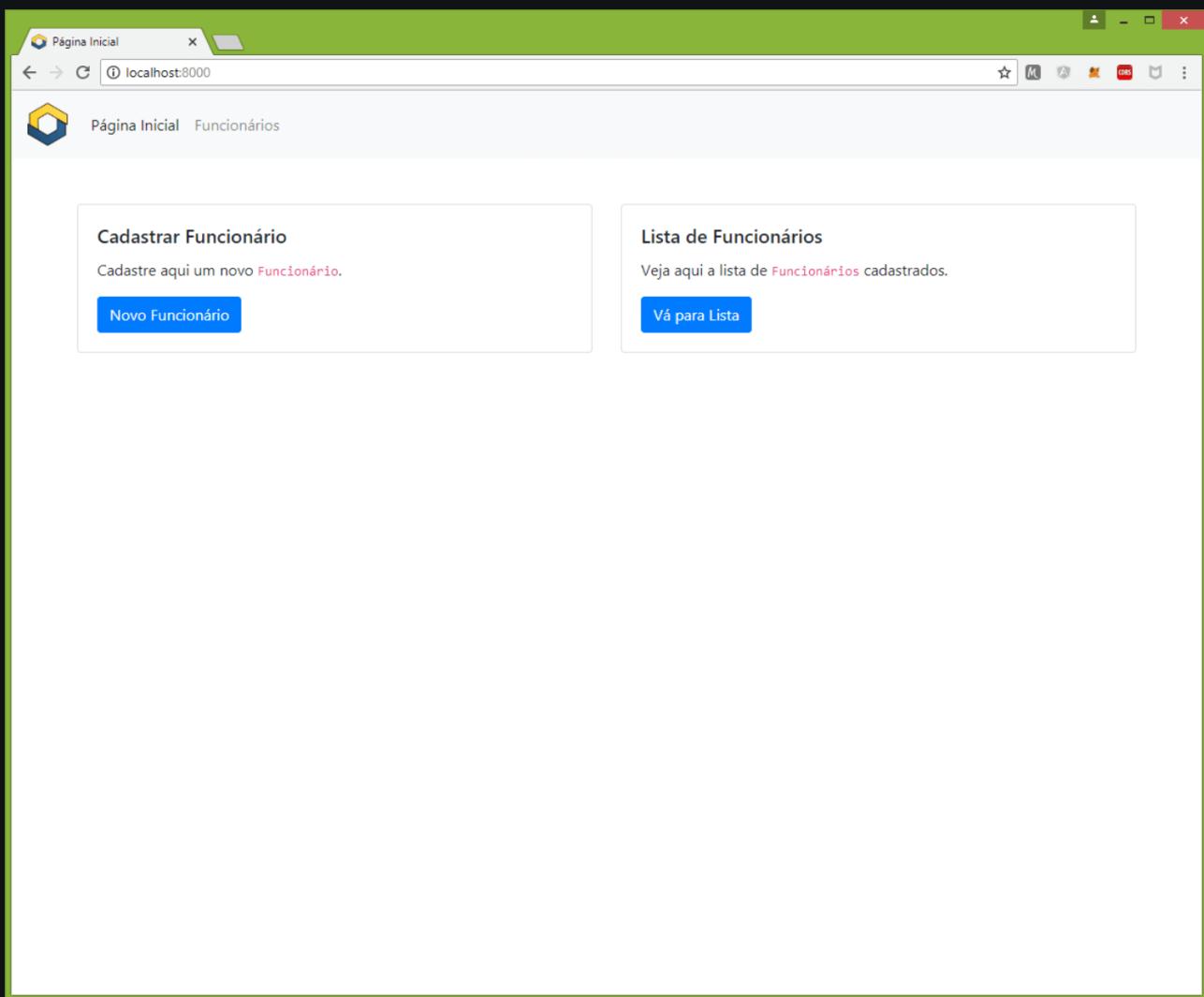
```

```
</div>  
{% raw %}{% endblock %}{% endraw %}
```

Nesse *template*:

- A classe `container` do Bootstrap (linha 9) serve para definir a área útil da nossa página (para que nossa página fique centralizada e não fique ocupando todo o comprimento da página).
- As classes `row` e `col-*` fazem parte do sistema *Grid* do Bootstrap e servem para tornar nossa página **responsiva** (que se adapta aos diversos tipos e tamanhos de tela: celular, *tablet*, desktop etc...).
- As classes `card*` fazem parte do componente *Card* do Bootstrap.
- As classes `btn` e `btn-primary` ([documentação](#)) são usados para dar o visual de botão à algum elemento.

Com isso, nossa Página Inicial - ou nossa *Homepage* - deve ficar assim:



Top, hein?! 🎉

Agora vamos para a página de cadastro de Funcionários: a `cria.html`

Template de Cadastro de Funcionários

Template: `website/cria.html`

Nesse *template*, mostramos o formulário para cadastro de novos funcionários.

Se lembra que definimos o formulário `InsereFuncionarioForm` no *post* passado?

Pois então... Dessa forma, nossa View `FuncionarioCreateView` expõe um objeto `form` no nosso *template* para que possamos utilizá-lo.

Mas antes de seguir, vamos instalar uma biblioteca que nos ajuda **e muito** a renderizar os campos de *input* do nosso formulário: [a *Widget Tweaks*!](#)

Com ela, nós temos maior liberdade para customizar os campos de *input* do nosso formulário (adicionando classes CSS e/ou atributos, por exemplo).

Para isso, primeiro nós a instalamos com:

```
pip install django-widget-tweaks
```

Depois a adicionamos a lista de *apps* instalados, no arquivo `helloworld/settings.py`:

```
INSTALLED_APPS = [  
    ...  
    'widget_tweaks',  
    ...  
]
```

E, no *template* onde formos utilizá-lo, carregamos ela com `{% raw %}{% load widget_tweaks %}{% endraw %}` !

E pronto, podemos utilizar a *tag* `{% raw %}{% render_field nome_do_campo parâmetros %}{% endraw %}` passando parâmetros, configurando a forma como o *input* será renderizado!

Assim, nós podemos construir nosso *template* da seguinte forma:

```

{% raw %}{% extends "website/_layouts/base.html" %}{% endraw %}

{% raw %}{% load widget_tweaks %}{% endraw %}

{% raw %}{% block title %}Cadastro de Funcionários{% endblock %}{%
endraw %}

{% raw %}{% block conteudo %}{% endraw %}


##### Cadastro de Funcionário



Complete o formulário abaixo para cadastrar  

        um novo Fucionário.


```

```

<!-- Nome -->


Nome



{{ form.nome }}


```

```

<!-- Sobrenome -->


Sobrenome



{{ form.sobrenome }}


```

```

<!-- CPF -->

```

```

<div class="input-group-prepend">
    <span class="input-group-text">CPF</span>
</div>
{%
    raw
    render_field form.cpf
    class+="form-control"
%}
{%
    endraw
%}
</div>

<!-- Tempo de Serviço -->
<div class="input-group mb-3">
    <div class="input-group-prepend">
        <span class="input-group-text">Tempo de Serviço</span>
    </div>
    {%
        raw
        render_field form.tempo_de_servico
        class+="form-control"
    %}
    {%
        endraw
    %}
</div>

<!-- Remuneração -->
<div class="input-group mb-3">
    <div class="input-group-prepend">
        <span class="input-group-text">Remuneração</span>
    </div>
    {%
        raw
        render_field form.remuneracao
        class+="form-control"
    %}
    {%
        endraw
    %}
</div>

    <button class="btn btn-primary">Enviar</button>
</form>
</div>
</div>
</div>
</div>
{%
    raw
%}
{%
    endblock
%}
{%
    endraw
%}

```

Aqui:

- Utilizamos, novamente as classes `container`, `row`, `col-*` e `card*` do Bootstrap.

- Conforme mencionei no *post* passado, devemos adicionar a tag `{% raw %}`
`{% csrf_token %}{% endraw %}` para evitar ataques de *Cross Site Request Forgery*.
- As classes *Input Group* do Bootstrap `input-group`, `input-group-prepend` e `input-group-text` servem para customizar o estilo do `<input ... />`.
- Utilizamos o `{% raw %}{% render_field form.campo class='form-control' %}{% endraw %}` para aplicar a classe `form-control` do Bootstrap ao *input* do formulário.

Observação: É possível adicionar a classe CSS `form-control` diretamente no nosso `InsereFuncionarioForm`, da seguinte forma:

```
class InsereFuncionarioForm(forms.ModelForm):
    ...
    nome = forms.CharField(
        max_length=255,
        widget=forms.TextInput(
            attrs={
                'class': "form-control"
            }
        )
    )
    ...
    ...
```

Mas eu **não aconselho fazer isso**, pois deixa nosso código extremamente acoplado. Veja que para mudar a classe CSS (atributo da interface) teremos que mudar código Python (backend). Por isso, aconselho a utilização de ferramentas para tal, como o *Widget Tweaks*, pois alteramos código apenas no template!

Com isso, nosso formulário deve ficar assim:

Cadastro de Funcionário

Complete o formulário abaixo para cadastrar um novo [Funcionário](#).

Nome

Sobrenome

CPF

Tempo de Serviço 0

Remuneração

Voltar Enviar

Agora, vamos desenvolver o *template* de listagem de Funcionários.

Template de Listagem de Funcionários

Template: website/lista.html

Nessa página, nós queremos mostrar o conjunto de Funcionários cadastrado no banco de dados e as ações que o usuário pode tomar: atualizar o Funcionário ou excluí-lo.

Se lembra da view `FuncionarioListView` ?

Ela é responsável por buscar a lista de Funcionários e expor um objeto chamado `funcionarios` para iteração no *template*.

Podemos construir nosso *template* da seguinte forma:

```

{% raw %}{% extends "website/_layouts/base.html" %}{% endraw %}

{% raw %}{% block title %}Lista de Funcionários{% endblock %}{% endraw %}

{% raw %}{% block conteudo %}{% endraw %}


##### Lista de Funcionário



{% raw %}{% if funcionarios|length > 0 %}{% endraw %}


Aqui está a lista de <code>Funcionários</code>  

        cadastrados.



| ID                                        | Nome                                        | Sobrenome                                        | Tempo de Serviço                                        | Remuneração                                        | Ações |
|-------------------------------------------|---------------------------------------------|--------------------------------------------------|---------------------------------------------------------|----------------------------------------------------|-------|
| {% raw %}{{ funcionario.id }}{% endraw %} | {% raw %}{{ funcionario.nome }}{% endraw %} | {% raw %}{{ funcionario.sobrenome }}{% endraw %} | {% raw %}{{ funcionario.tempo_de_servico }}{% endraw %} | {% raw %}{{ funcionario.remuneracao }}{% endraw %} |       |


```

```

<td>
    <a href="{% raw %}{% url
'website:atualiza_funcionario' pk=funcionario.id %}{% endraw
%}">
        class="btn btn-info">
            Atualizar
    </a>
    <a href="{% raw %}{% url
'website:deleta_funcionario' pk=funcionario.id %}{% endraw %}">
        class="btn btn-outline-danger">
            Excluir
    </a>
</td>
</tr>
{% raw %}{% endfor %}{% endraw %}
</tbody>
</table>
{% raw %}{% else %}{% endraw %}
<div class="text-center mt-5 mb-5 jumbotron">
    <h5>Nenhum <code>Funcionário</code> cadastrado ainda.</h5>
</div>
{% raw %}{% endif %}{% endraw %}
<hr />
<div class="text-right">
    <a href="{% raw %}{% url 'website:cadastra_funcionario' %}{%
endraw %}" class="btn btn-primary">
        Cadastrar Funcionário
    </a>
</div>
</div>
</div>
</div>
</div>
<% raw %}{% endblock %}{% endraw %}

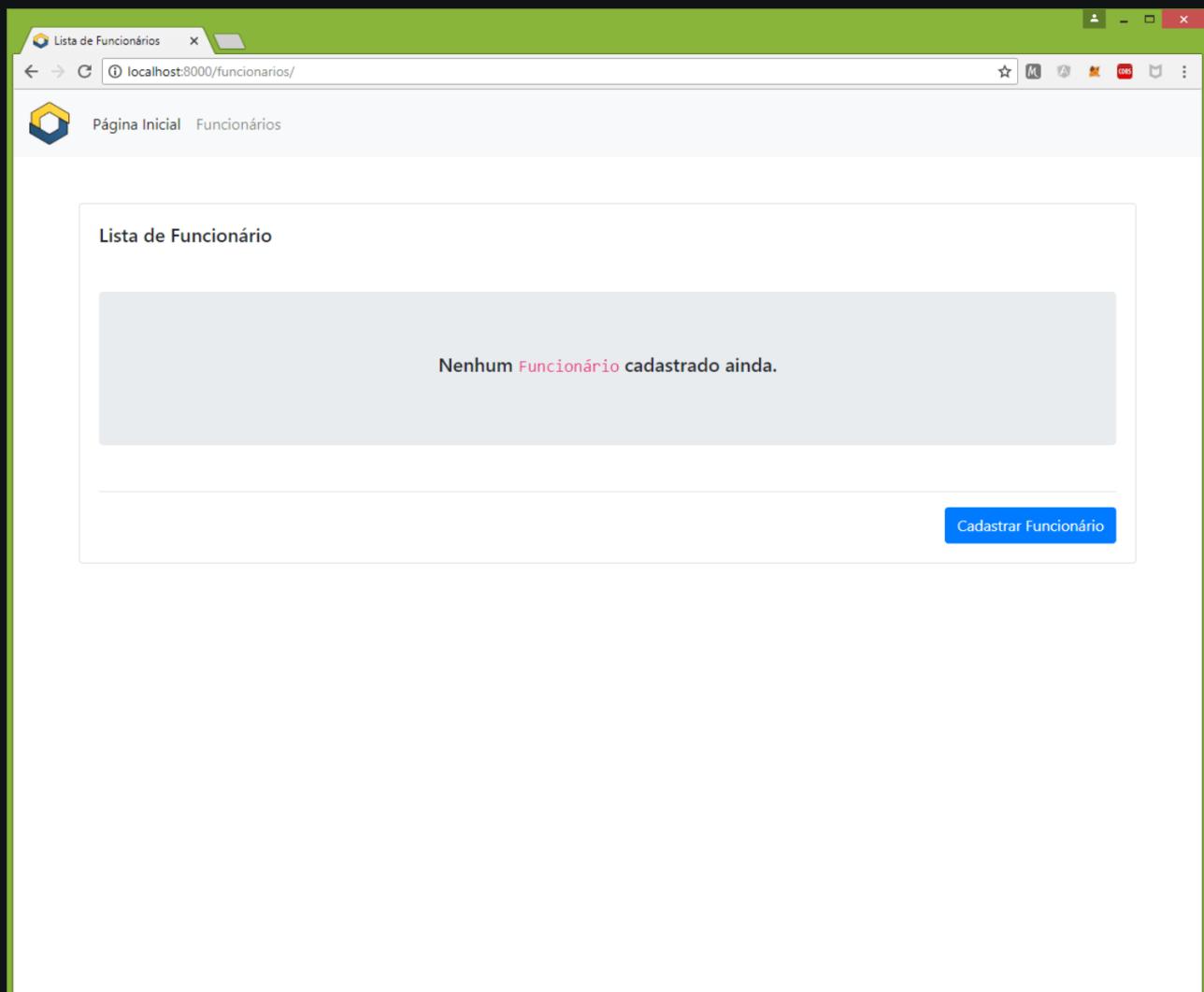
```

Nesse *template*:

- Utilizamos classes **do Bootstrap para estilizar as tabelas**. São elas: `table` para estilizar o cabeçalho e as linhas e `thead-dark` para escurecer o cabeçalho.

- Na **linha 13**, utilizamos o **filtro length** para verificar se a lista de funcionários está vazia. Se ela tiver dados, é mostrada a tabela. Se estiver vazia, é mostrado o componente *Jumbotron* do Bootstrap com o texto “*Nenhum Funcionário cadastrado ainda*”.
- Na **linha 30** utilizamos a tag `{% raw %}{% for funcionario in funcionarios %}{% endraw %}` para iterar sobre a lista `funcionarios`.
- Nas **linhas 39 e 46** fazemos o *link* para as páginas de atualização e exclusão do usuário.

O resultado, sem Funcionários cadastrados, deve ser esse:



E com um Funcionário cadastrado:

ID	Nome	Sobrenome	Tempo de Serviço	Remuneração	Ações
3	Vinicius	Ramos	1	10000.00	Atualizar Excluir

Quando o usuário clicar em “Excluir”, ele será levado para a página `exclui.html` e quando clicar em “Atualizar”, ele será levado para a página `atualiza.html`.

Vamos começar construindo a página de Atualização de Funcionários!

Template de Atualização de Funcionários

Template: `website/atualiza.html`

Nessa página, queremos que o usuário possa ver os dados atuais do Funcionário e possa atualizá-los, conforme sua vontade.

Para isso utilizamos a view `FuncionarioUpdateView` implementada no [post](#) passado.

Ela expõe um formulário com os dados do modelo previamente preenchidos para que o usuário possa alterar.

Vamos utilizar novamente a biblioteca *Widget Tweaks* para facilitar a renderização dos campos de *input*.

Abaixo, como podemos fazer nosso *template*:

```

{%
    raw
}{%
    extends "website/_layouts/base.html"
}{%
    endraw
}

{%
    raw
}{%
    load widget_tweaks
}{%
    endraw
}

{%
    raw
}{%
    block title
}{%
        Atualização de Dados do Funcionários
}{%
    endblock
}{%
    endraw
}

{%
    raw
}{%
    block conteudo
}{%
    endraw
}
<div class="container">
    <div class="row">
        <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
            <div class="card">
                <div class="card-body">
                    <h5 class="card-title">Atualização de Dados do Funcionário</h5>
                    <form method="post">
                        <!-- Não se esqueça dessa tag -->
                        {% raw %}{% csrf_token %}{% endraw %}

                        <!-- Nome -->
                        <div class="input-group mb-3">
                            <div class="input-group-prepend">
                                <span class="input-group-text">Nome</span>
                            </div>
                            {% raw %}{%
                                render_field form.nome class+="form-control"
                            }{% endraw %}
                        </div>

                        <!-- Sobrenome -->
                        <div class="input-group mb-3">
                            <div class="input-group-prepend">
                                <span class="input-group-text">Sobrenome</span>
                            </div>
                            {% raw %}{%
                                render_field form.sobrenome class+="form-control"
                            }{% endraw %}
                        </div>

                        <!-- CPF -->
                        <div class="input-group mb-3">
                            <div class="input-group-prepend">
                                <span class="input-group-text">CPF</span>
                            </div>

```

```

    {% raw %}{% render_field form.cpf class+="form-control" %}
    {% endraw %}
    </div>

    <!-- Tempo de Serviço -->
    <div class="input-group mb-3">
        <div class="input-group-prepend">
            <span class="input-group-text">Tempo de Serviço</span>
        </div>
        {% raw %}{% render_field form.tempo_de_servico
        class+="form-control" %}{% endraw %}
    </div>

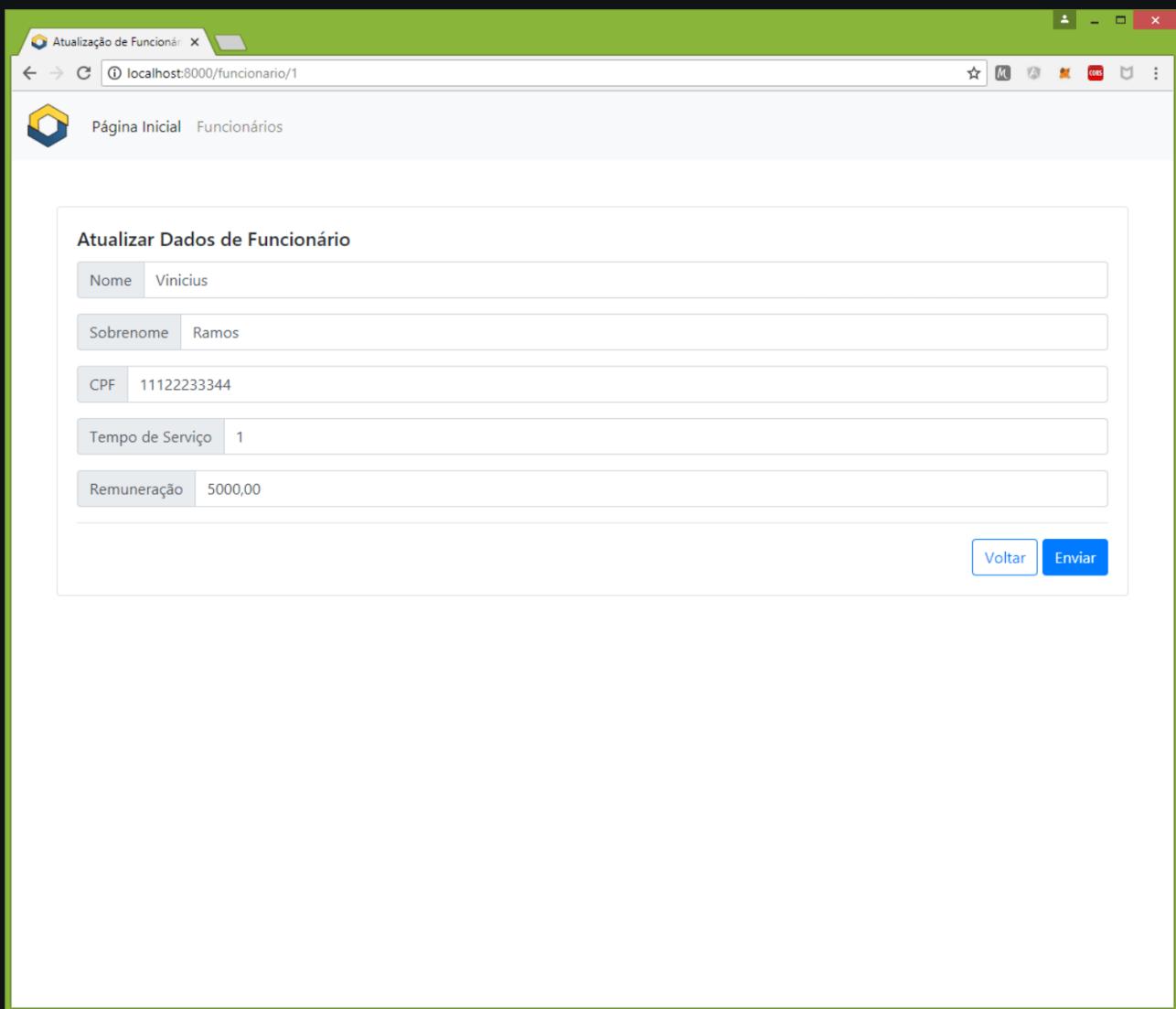
    <!-- Remuneração -->
    <div class="input-group mb-3">
        <div class="input-group-prepend">
            <span class="input-group-text">Remuneração</span>
        </div>
        {% raw %}{% render_field form.remuneracao class+="form-
control" %}{% endraw %}
    </div>

        <button class="btn btn-primary">Enviar</button>
    </form>
</div>
</div>
</div>
</div>
<% raw %}{% endblock %}{% endraw %}

```

Nesse *template*, não temos nada de novo. Perceba que seu código é similar ao *template* de adição de Funcionários.

Sua interface fica bem similar também:



E por último, temos o *template* de exclusão de Funcionários.

Template de Exclusão de Funcionários

Template: website/exclui.html

A função dessa página é mostrar uma página de confirmação para o usuário antes da exclusão de um Funcionário.

A view que fizemos, a `FuncionarioDeleteView`, facilita bastante nossa vida. Com ela, basta dispararmos uma requisição `POST` para a URL de exclusão!

Dessa forma, nosso objetivo se resume à:

```
<!-- Estendemos do template base -->
{% raw %}{% extends "website/_layouts/base.html" %}{% endraw %}

<!-- Bloco que define o <title></title> da nossa página -->
{% raw %}{% block title %}Página Inicial{% endblock %}{% endraw %}

<!-- Bloco de conteúdo da nossa página -->
{% raw %}{% block conteudo %}{% endraw %}
<div class="container mt-5">
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">Exclusão de Funcionário</h5>

      <p class="card-text">
        Você tem certeza que quer excluir o funcionário
        <b>{{ funcionario.nome }}</b>?
      </p>

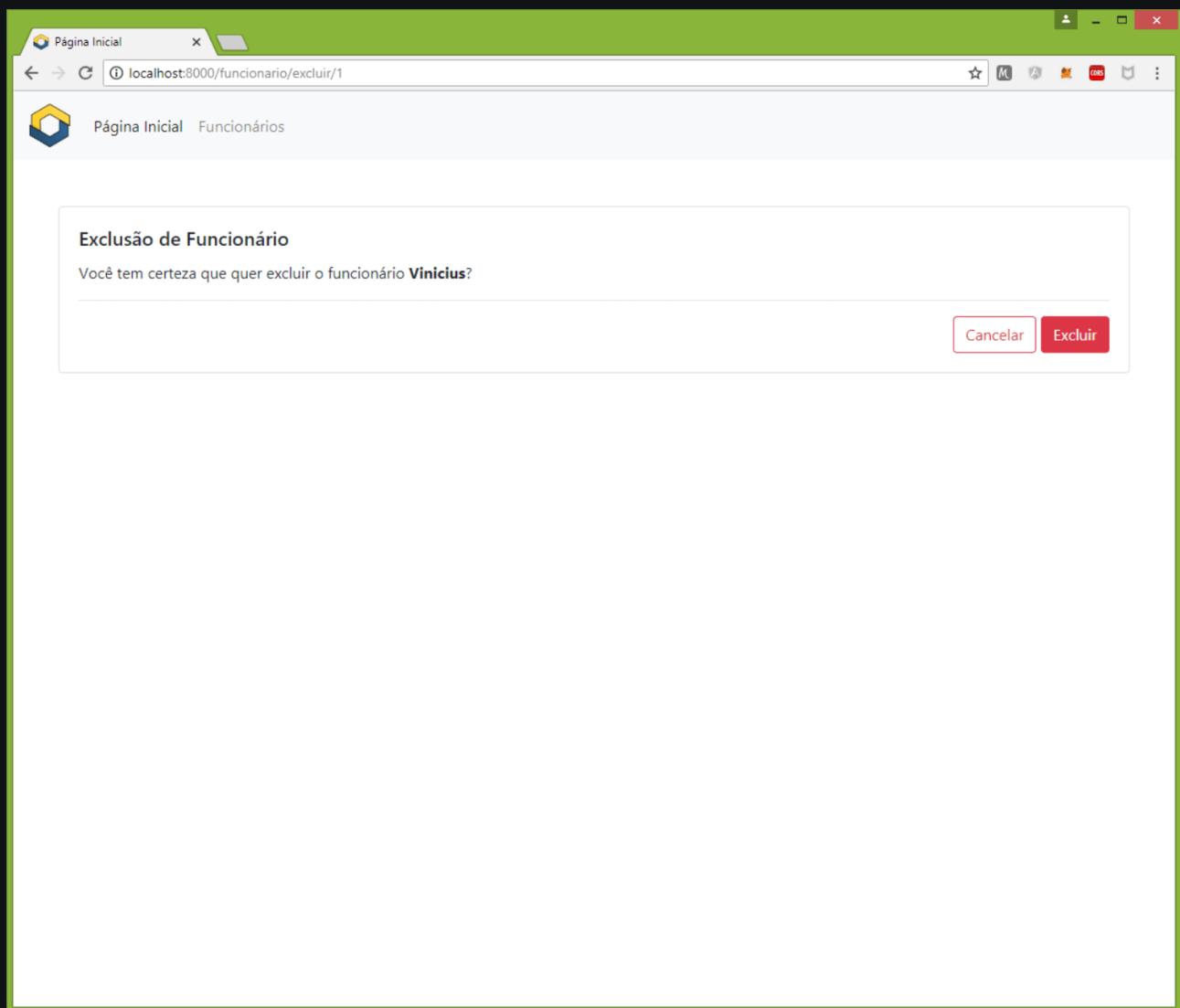
      <form method="post">
        {% raw %}{% csrf_token %}{% endraw %}

        <hr />
        <div class="text-right">
          <a href="{% raw %}{% url 'website:lista_funcionarios' %}{% endraw %}">
            class="btn btn-outline-danger">
              Cancelar
          </a>
          <button class="btn btn-danger">Excluir</button>
        </div>
      </form>
    </div>
  </div>
</div>

{% raw %}{% endblock %}{% endraw %}
```

Aqui, nada de novo também.

Apenas mostramos o formulário onde o usuário pode decidir excluir ou não o Funcionário, que deve ficar assim:



Pronto!

Com isso, temos todas as telas do nosso projeto! 😊

Agora vamos ver como construir **tags** e **filtros customizados**!

Tags e Filtros customizados

Sabemos, agora, que o Django possui uma grande variedade de filtros e *tags* pré-configurados.

Contudo, é possível que, em alguma situação específica, o Django não te ofereça o filtro ou *tag* necessário.

Para isso, ele previu a possibilidade de você **construir seus próprios filtros e tags!**

E é exatamente isso que veremos **agora!**

Vamos construir uma **tag** que irá nos dizer o tempo atual formatado e um **filtro** que irá retornar a primeira letra da string passada.

Para isso, vamos começar com a **configuração** necessária!

Configuração

Para utilizar *tags* e filtros customizados, precisamos criar uma pasta chamada `/templatetags` na raiz do app.

Crie a pasta `website/templatetags/` e dentro dela, adicione:

- Um script `__init__.py` em branco
- O script `tempo_atual.py` referente à nossa *tag*
- O script `primeira_letra.py` referente ao nosso filtro.

Nossa estrutura deve ficar:

```
website/
...
templatetags/
__init__.py
tempo_atual.py
primeira_letra.py
...
```

Para que o Django enxergue nossas *tags* e filtros é necessário que o *app* onde eles estão instalada esteja configurada na lista `INSTALLED_APPS` do `settings.py` (no nosso caso, `website` já está lá, portanto, nada a fazer aqui).

Também é necessário carregá-los com o comando:

```
{% raw %}{% load filtro/tag %}{% endraw %}.
```

Vamos começar com o **filtro**.

Vamos chamá-lo de `primeira_letra` e iremos utilizá-lo da seguinte maneira:

```
<p>{% raw %}{{ valor|primeira_letra }}{% endraw %}</p>
```

Filtro `primeira_letra`

Filtros customizados são apenas funções que recebem um ou dois argumentos. São eles:

- O valor do *input*.
- O valor do argumento - que pode ter um valor padrão ou não receber nenhum valor.

No nosso filtro `{% raw %}{{ valor|primeira_letra }}{% endraw %}`:

- `valor` é o *input value*.

- Nosso filtro não receberá argumentos, portanto não foi passado nada como tal.

Para ser um filtro válido, é necessário que o código dele contenha uma variável chamada `register` que seja uma instância de `template.Library` (onde todos os *tags* e filtros são registrado).

Isso **define** um filtro.

Outra questão são as **Exceções**.

Como a *engine* de *templates* do Django não provê tratamento de exceção à execução do filtro, qualquer exceção no filtro será exposta como uma exceção do próprio servidor.

Por isso, nosso filtro deve evitar lançar exceções e, ao invés disso, deve retornar um valor padrão.

Vamos ver um exemplo de filtro do Django.

Abra o arquivo `django/template/defaultfilter.py`. Lá temos o exemplo do filtro `lower`:

```
@register.filter(is_safe=True)
@stringfilter
def lower(value):
    """Convert a string into all lowercase."""
    return value.lower()
```

Nele:

- `@register.filter(is_safe=True)` é utilizado para registrar sua função **como um filtro para o Django*. Só assim o *framework* vai enxergar seu código.

- `@stringfilter` é um *decorator* utilizado para dizer ao Django que seu filtro espera uma string como argumento (saiba mais sobre *decorators* no nosso post [Domine Decorators em Python](#)).

Com essas informações, vamos agora codificar e registrar nosso próprio filtro!

Uma forma de pegarmos a primeira letra de uma string é transformá-la em lista e pegar o elemento de índice 0, da seguinte forma:

```
from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
@stringfilter
def primeira_letra(value):
    return list(value)[0]
```

Nesse código:

- O código `register = template.Library()` é necessário para registrarmos nosso filtro no catálogo de filtros do Django.
- `@register.filter` e `@stringfilter` são os *decorators* que citei aqui em cima.

E agora vamos fazer o teste em algum *template* nosso.

Vamos alterar nossa tabela no *template* `website/lista.html` para incluir nosso filtro da seguinte forma:

```


| <!-- Retiramos o "ID" aqui -->                             | Nome                                        | Sobrenome                                        | Tempo de Serviço                                        | Remuneração                                        | Ações                                                                                                                                                                                                                                                          |
|------------------------------------------------------------|---------------------------------------------|--------------------------------------------------|---------------------------------------------------------|----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| {% raw %}{{ funcionario.nome primeira_letra }}{% endraw %} | {% raw %}{{ funcionario.nome }}{% endraw %} | {% raw %}{{ funcionario.sobrenome }}{% endraw %} | {% raw %}{{ funcionario.tempo_de_servico }}{% endraw %} | {% raw %}{{ funcionario.remuneracao }}{% endraw %} | <a class="btn btn-primary" href="{% raw %}{% url 'website:atualiza_funcionario' pk=funcionario.id %}{% endraw %}"> Atualizar </a> <a class="btn btn-danger" href="{% raw %}{% url 'website:deleta_funcionario' pk=funcionario.id %}{% endraw %}"> Excluir </a> |


```

O que resulta em:

	Nome	Sobrenome	Tempo de Serviço	Remuneração	Ações
V	Vinicius	RAMOS	1	1000.00	<button>Atualizar</button> <button>Excluir</button>

E com isso, fizemos nosso **primeiro filtro!**

Agora vamos fazer nossa *tag* customizada: a `tempo_atual` !

Tag `tempo_atual`

De acordo com a documentação do Django, “tags são mais complexas que filtros pois podem fazer **qualquer coisa**”.

Desenvolver uma *tag* pode ser algo bem trabalhoso, dependendo do que você deseja fazer. Mas também pode ser simples.

Como nossa *tag* vai apenas mostrar o tempo atual, sua implementação não deve ser complexa.

Para isso, utilizaremos um “atalho” do Django: a `simple_tag`!

A `simple_tag` é uma ferramenta para construção de *tags* simples (assim como o próprio nome já diz).

Com ela, a criação de *tags* fica similar à criação de filtros, que vimos na seção passada.

Precisamos incluir uma instância de `template.Library`, utilizar o *decorator* `@register` e definir nossa função.

Dessa forma, podemos definí-la como:

```
import datetime
from django import template

register = template.Library()

@register.simple_tag
def tempo_atual():
    return datetime.datetime.now().strftime('%H:%M:%S')
```

E para utilizá-la, fazemos o carregamento com `{% raw %}{% load tempo_atual %}{% endraw %}` e utilizamos em nosso *template* com `{% raw %}{% tempo_atual %}{% endraw %}`.

No nosso caso, vamos alterar o arquivo `website/_layouts/base.html` para utilizar nossa *tag*.

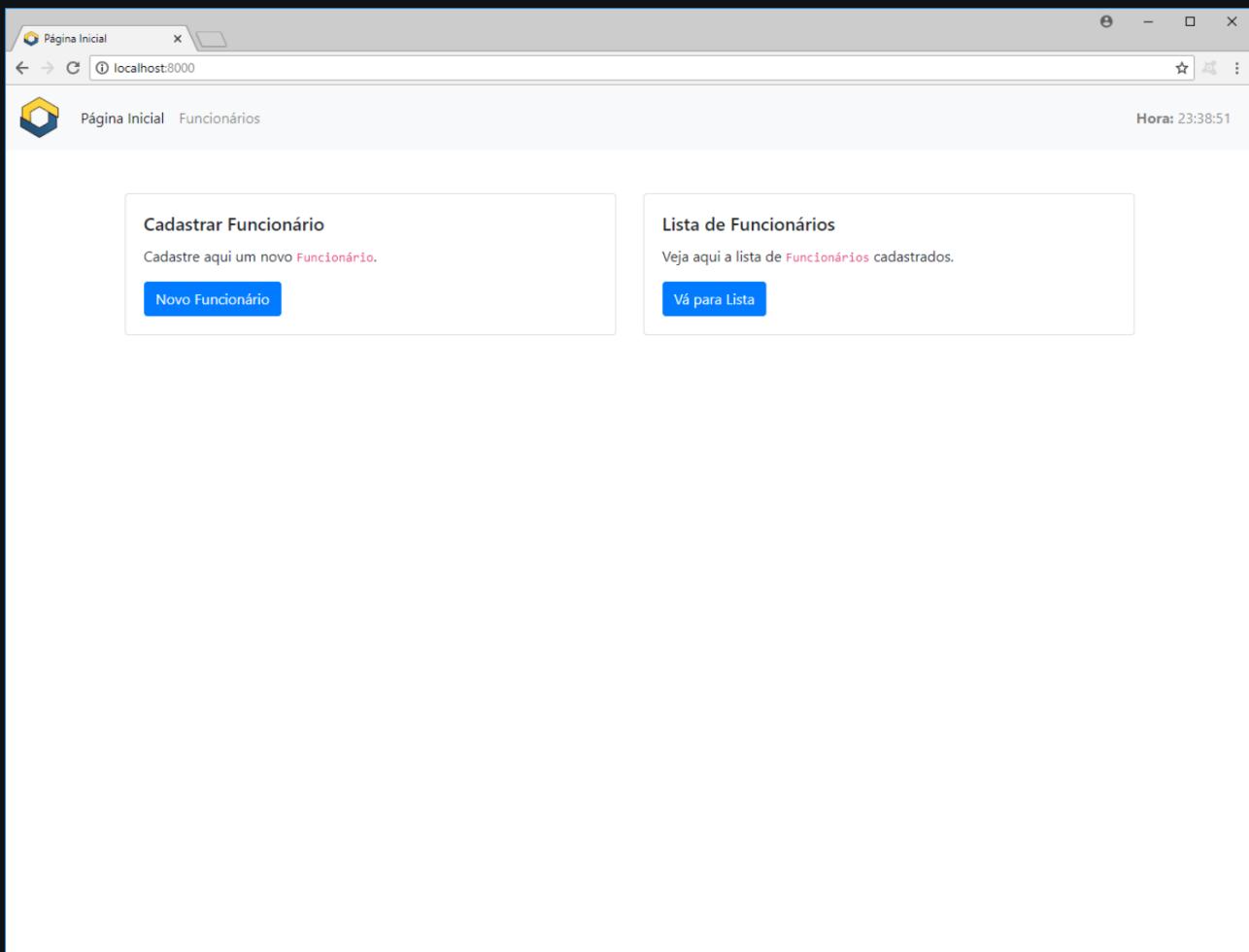
Vamos adicionar um novo item à barra de navegação (do lado direito), da seguinte forma:

```

<body>
    <!-- Navbar -->
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        ...
        <div class="collapse navbar-collapse"
            id="navbarSupportedContent">
            <ul class="navbar-nav mr-auto">
                <li class="nav-item active">
                    <a class="nav-link"
                        href="{% raw %}{% url 'website:index' %}{% endraw %}">
                        Página Inicial
                    </a>
                </li>
                <li class="nav-item">
                    <a class="nav-link"
                        href="{% raw %}{% url 'website:lista_funcionarios' %}{% endraw %}">
                        Funcionários
                    </a>
                </li>
            </ul>
            <!-- Adicione a lista abaixo -->
            <ul class="navbar-nav float-right">
                <li class="nav-item">
                    <!-- Aqui está nosso filtro -->
                    <a class="nav-link" href="#">
                        <b>Hora: </b>{%
                            raw %}{%
                            tempo_atual %}{%
                            endraw %}
                    </a>
                </li>
            </ul>
        </div>
    </nav>
    ...

```

O resultado deve ser:



Com isso, temos nosso filtro e *tag* customizados!

Agora vamos dar uma olhada no filtros que estão presentes no próprio Django: os *Built-in Filters*!

Built-in Filters

É possível fazer muita coisa com os filtros que já veem instalados no próprio Django!

Muitas vezes, é melhor você fazer algumas operações no *template* do que fazê-las no *backend*. Sempre verifique a viabilidade de um ou de outro para facilitar sua vida!

Como a lista de *built-in filters* do Django é bem extensa ([veja a lista completa aqui](#)), vou listar aqui os que eu considero mais úteis!

Use essa página como **referência**!

Favorite-a se quiser para ter acesso rápido à esse conteúdo!

Sem mais delongas, aí vai o primeiro: o `capfirst` !!!

Filtro `capfirst`

O que faz: Torna o primeiro caracter do valor para maiúsculo.

Exemplo:

Entrada: `valor = 'esse é um texto' .`

Utilização:

```
{% raw %}{{ valor|capfirst }}{% endraw %}
```

Saída:

```
Esse é um texto
```

Filtro `cut`

O que faz: Remove todas as ocorrências do parâmetro no valor passado.

Exemplo:

Entrada: `valor = 'Esse É Um Texto De Testes'`

Utilização:

```
{% raw %}{{ valor|cut:" " }}{% endraw %}
```

Saída:

```
EsseÉUmTextoDeTestes
```

Filtro date

O que faz: Utilizado para formatar datas. Possui uma grande variedade de configurações ([veja aqui](#)).

Exemplo:

Entrada: Objeto `datetime`.

Utilização:

```
{% raw %}{{ data|date:'d/m/Y' }}{% endraw %}
```

Saída:

```
01/07/2018
```

Filtro default

O que faz: Caso o valor seja `False`, utiliza o valor `default`.

Exemplo:

Entrada: valor = False

Utilização:

```
{% raw %}{{ valor|default:'Nenhum valor' }}{% endraw %}
```

Saída:

```
Nenhum valor
```

Filtro default_if_none

O que faz: Similar ao filtro `default`, caso o valor seja `None`, utiliza o valor configurado em `default_if_none`.

Exemplo:

Entrada: valor = None

Utilização:

```
{% raw %}{{ valor|default:'Nenhum valor' }}{% endraw %}
```

Saída:

```
Nenhum valor
```

Filtro `divisibleby`

O que faz: Retorna `True` se o valor for divisível pelo argumento.

Exemplo:

Entrada: `valor = 14` e `divisibleby:'2'`

Utilização:

```
{% raw %}{{ valor|divisibleby:'2' }}{% endraw %}
```

Saída:

```
True
```

Filtro `filesizeformat`

O que faz: Transforma tamanhos de arquivos em valores legíveis por humanos.

Exemplo:

Entrada: `valor = 123456789`

Utilização:

```
{% raw %}{{ valor|filesizeformat }}{% endraw %}
```

Saída:

```
117.7 MB
```

Filtro first

O que faz: Retorna o primeiro item em uma lista

Exemplo:

Entrada: valor = ["Marcos", "João", "Luiz"]

Utilização:

```
{% raw %}{{ valor|first }}{% endraw %}
```

Saída:

```
Marcos
```

Filtro last

O que faz: Retorna o último item em uma lista

Exemplo:

Entrada: valor = ["Marcos", "João", "Luiz"]

Utilização:

```
{% raw %}{{ valor|last }}{% endraw %}
```

Saída:

Filtro floatformat

O que faz: Arredonda números com ponto flutuante com o número de casas decimais passado por argumento.

Exemplo:

Entrada: valor = 14.25145

Utilização:

```
{% raw %}{{ valor|floatformat:"2" }}{% endraw %}
```

Saída:

```
14.25
```

Filtro join

O que faz: Junta uma lista utilizando a string passada como argumento como separador.

Exemplo:

Entrada: valor = ["Marcos", "João", "Luiz"]

Utilização:

```
{% raw %}{{ valor|join:" - " }}{% endraw %}
```

Saída:

```
Marcos - João - Luiz
```

Filtro length

O que faz: Retorna o comprimento de uma lista ou string. É muito utilizado para saber se existem valores na lista.

Exemplo:

```
Entrada: valor = ['Marcos', 'João']
```

Utilização:

```
{% raw %}{% if valor|length > 0 %}{% endraw %}
<p>Lista contém valores</p>
{% raw %}{% else %}{% endraw %}
<p>Lista vazia</p>
{% raw %}{% endif %}{% endraw %}
```

Saída:

```
<p>Lista contém valores</p>
```

Filtro lower

O que faz: Transforma todos os caracteres de uma string em minúsculas.

Exemplo:

Entrada: valor = PaRaLeLePíPeDo

Utilização:

```
{% raw %}{{ valor|lower }}{% endraw %}
```

Saída:

```
paralelepípedo
```

Filtro pluralize

O que faz: Retorna um sufixo plural caso o número seja maior que 1.

Exemplo:

Entrada: valor = 12

Utilização:

```
{% raw %}Sua empresa tem Funcionário{{ valor|pluralize:"s" }}{% endraw %}
```

Saída:

Sua empresa tem 12 Funcionários

Filtro random

O que faz: Retorna um item aleatório de uma lista

Exemplo:

Entrada: valor = [1, 2, 3, 4, 5, 6, 7, 9]

Utilização:

```
{% raw %}{{ valor|random }}{% endraw %}
```

Sua saída será um valor da lista escolhido randomicamente.

Filtro title

O que faz: Transforma em maiúsculo o primeiro caracter de todas as palavras do texto.

Exemplo:

Entrada: valor = 'Esse é o primeiro post do blog'

Utilização:

```
{% raw %}{{ valor|title }}{% endraw %}
```

Saída:

```
Esse é o Primeiro Post Do Blog
```

Filtro upper

O que faz: Transforma em maiúsculo todos caracteres da string.

Exemplo:

Entrada: valor = texto de testes

Utilização:

```
{% raw %}{{ valor|upper }}{% endraw %}
```

Saída:

```
TEXTO DE TESTES
```

Filtro wordcount

O que faz: Retorna o número de palavras da string.

Exemplo:

Entrada: valor = Django é o melhor framework web

Utilização:

```
{% raw %}{{ valor|wordcount }}{% endraw %}
```

Saída:

```
6
```

Bom...

Eu acho que está bom de filtros por aqui! 😊

Código

O código completo desenvolvido nesse projeto está **disponível no Github da Python Academy**. Clique aqui para acessá-lo e baixá-lo!

Para rodar o projeto, execute em seu terminal:

- `pip install -r requirements.txt` para instalar as dependências.
- `python manage.py makemigrations` para criar as **Migrações**.
- `python manage.py migrate` para efetivar as **Migrações** no banco de dados.
- `python manage.py runserver` para executar o servidor de testes do Django.
- Acessar o seu navegador na página **http://localhost:8000** (por padrão).

E pronto... Servidor rodando! 😊

Conclusão

Ufa... Por hoje é só!

Nesse *post* vimos como configurar, customizar e estender *templates*, como utilizar os filtros e *tags* do Django, como criar *tags* e filtros customizados e um pouquinho de *Bootstrap*, pra deixar as páginas **bonitonas**!

E com isso, senhores Pythonistas, **terminamos nossa sequência de posts baseados em Django!**

Espero que tenham gostado do conteúdo... Tentei abordar a maior quantidade de conhecimento aqui para que você possa iniciar sua jornada no Django **com o pé direito!**

Creio que o conhecimento abordado aqui é suficiente para vocês iniciarem o desenvolvimento de projetos pessoais e até profissionais utilizando Django!

Claro que ainda há muito para aprender, mas vimos bastante coisa por aqui...

E você... Como se saiu?!

Que tal customizar, modificar e deixar esse projeto **um brinco**?!

Faça e **poste o resultado aqui embaixo** no box de comentários!

Quero ver como você se saiu no seu **primeiro projeto Django!** 😊

Ah, quer levar essa série completa de *posts* para onde for com nosso **ebook GRÁTIS?**

Então aproveita essa chance 

É isso pessoal!

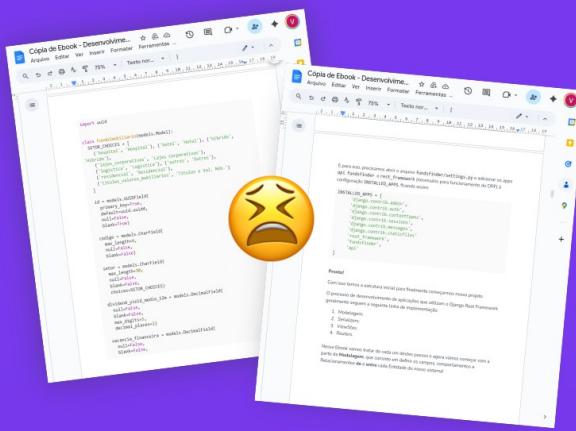
Até a próxima!

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